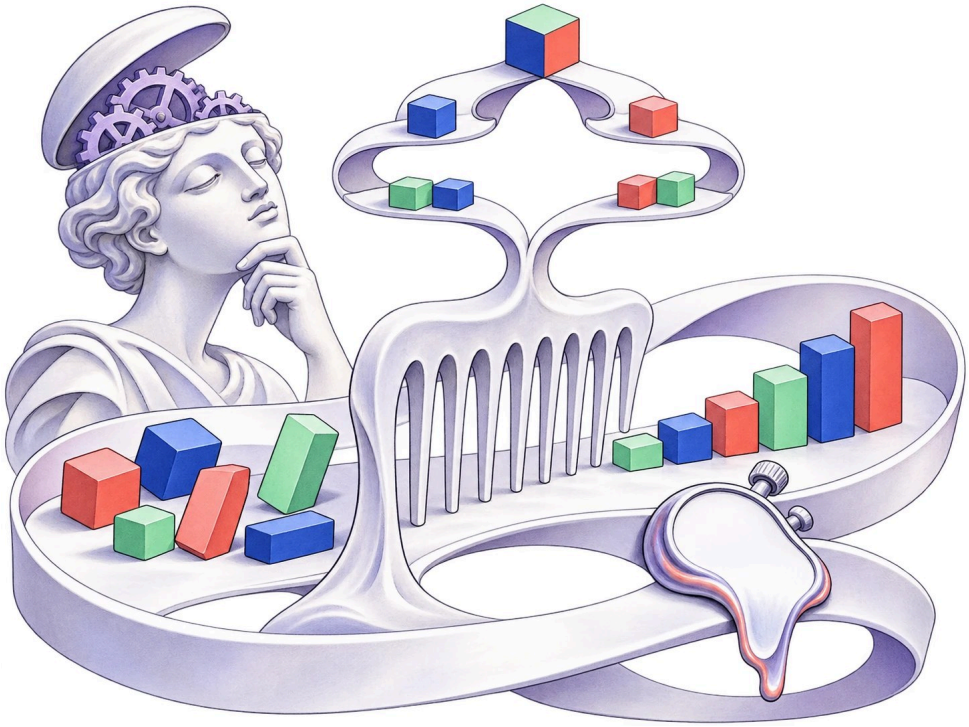


COURSE TEXTBOOK

Algorithms

Sercan Külcü

Giresun University · Department of Computer Engineering

Algorithms

Sercan Külcü
Giresun University, Department of Computer Engineering
Giresun, Türkiye

This book is the printed form of the course notes published at
`sercankulcu.github.io/teaching/algorithms/`

Slides, study questions and simulators are on that page. The code for each chapter is in the `github.com/sercankulcu/algorithms-java-en` repository.

© Sercan Külcü. All rights reserved.

Generated from the notes of 22 November 2026.

Preface

An algorithm is a finite sequence of well-defined steps that solves a problem or carries out a computation. This book studies how to argue that an algorithm is correct, how to measure its time and memory cost, and how these tools apply to sorting, searching, graph and string problems, together with the trade-offs between different solutions. It is made up of the notes for the Algorithms course taught in the Department of Computer Engineering at Giresun University, and its chapters follow the order of the course. The examples are written in Java.

The book has three parts. In order, they are **Foundations** (Chapters 1–4), **Graph Algorithms** (Chapters 5–8), and **String Algorithms** (Chapters 9–11). Design paradigms, hard problems and other models of computation are covered in the follow-up book, *Advanced Algorithms*.

Every chapter follows the same layout. It opens with learning outcomes. Formal definitions, theorems and worked examples are set in named boxes. *Common mistake* boxes point out the misconceptions students fall into most often. Code traces show, step by step, how an algorithm changes its data. Each chapter closes with a summary, a list of key terms, *Test Yourself* questions with their answers, suggestions for further reading and a bibliography. The glossary at the end of the book defines each key term in a sentence and names the chapter that explains it.

The code for each chapter is in its own folder in the companion repository, `github.com/sercankulcu/algorithms-java-en`, and the programs use the same names as the listings in this book. Slides, study questions and simulators that run the algorithms step by step are on the course page.

Sercan Külcü
Giresun

Contents

Course Map 1

PART I · FOUNDATIONS

1 Introduction 4

1.1 The concept of an algorithm 5
1.2 Algorithm analysis 11
1.3 Complexity examples 17
1.4 Euclid's algorithm 31
1.5 Common mistakes 38
1.6 Summary 39
1.7 Key terms 40
1.8 Test Yourself 40
1.9 Further reading 43
1.10 References 43

2 Algorithm Complexity 44

2.1 The concept of complexity 45
2.2 Asymptotic notation 53
2.3 Loop analysis 61
2.4 Analysis of recursion 70
2.5 The Master Theorem 74
2.6 Application: f4 and f5 80
2.7 Comparison and amortized analysis 85
2.8 Common mistakes 89
2.9 Summary 89
2.10 Key terms 91
2.11 Test Yourself 91
2.12 Further reading 93
2.13 References 93

3 Sorting Algorithms 94

3.1 The sorting problem and evaluation criteria 94
3.2 Bubble sort 99
3.3 Insertion sort 105
3.4 Selection sort 110
3.5 Shell sort 115
3.6 Merge sort 118
3.7 Quicksort 126
3.8 Heap sort 134
3.9 Linear-time sorting 142
3.10 Timsort 151

3.11 Comparing the algorithms 153
3.12 Common mistakes 155
3.13 Summary 156
3.14 Key terms 156
3.15 Test Yourself 157
3.16 Further reading 159
3.17 References 159

4 Searching Algorithms 160

4.1 The search problem 161
4.2 Linear search 163
4.3 Binary search 170
4.4 Searching with hashing 180
4.5 Interpolation search 190
4.6 Jump search 195
4.7 Exponential search 199
4.8 Comparison and choosing a method 202
4.9 Common mistakes 204
4.10 Summary 206
4.11 Key terms 206
4.12 Test Yourself 207
4.13 Further reading 209
4.14 References 209

PART II · GRAPH ALGORITHMS

5 Graph Traversal 211

5.1 Graphs and their representations 212
5.2 Graph traversal 218
5.3 Depth-first search (DFS) 219
5.4 Breadth-first search (BFS) 232
5.5 Strongly connected components 245
5.6 Applications of traversal 258
5.7 Common mistakes 266
5.8 Summary 267
5.9 Key terms 268
5.10 Test Yourself 268
5.11 Further reading 271
5.12 References 271

6 Shortest Paths 272

6.1 The shortest path problem 273
6.2 Dijkstra's algorithm 278
6.3 The Bellman-Ford algorithm 291
6.4 The Floyd-Warshall algorithm 302
6.5 A* search 311

6.6 Shortest paths in unweighted graphs: BFS 320
6.7 Comparison and choice of algorithm 321
6.8 Common mistakes 322
6.9 Summary 323
6.10 Key terms 323
6.11 Test Yourself 324
6.12 Further reading 326
6.13 References 327

7 Minimum Spanning Trees 328

7.1 Spanning trees and minimum spanning trees 329
7.2 Kruskal's algorithm 337
7.3 Disjoint sets (union-find) 340
7.4 Kruskal's code and complexity 346
7.5 Prim's algorithm 352
7.6 Prim's code and complexity 357
7.7 Borůvka's algorithm 365
7.8 The reverse-delete algorithm 373
7.9 Comparing the four algorithms 377
7.10 Common mistakes 378
7.11 Summary 380
7.12 Key terms 380
7.13 Test Yourself 381
7.14 Further reading 383
7.15 References 383

8 Network Flow 385

8.1 Flow networks 386
8.2 The residual network and augmenting paths 389
8.3 Cuts 393
8.4 The Ford-Fulkerson method 398
8.5 The Edmonds-Karp algorithm 412
8.6 Dinic's algorithm 425
8.7 Push–relabel 434
8.8 Comparison and applications 444
8.9 Common mistakes 447
8.10 Summary 448
8.11 Key terms 448
8.12 Test Yourself 449
8.13 Further reading 451
8.14 References 451

PART III · STRING ALGORITHMS

9 String Matching and Compression 453

9.1 The string matching problem 454

9.2 The brute-force algorithm 456
9.3 The Knuth-Morris-Pratt algorithm 461
9.4 The Boyer-Moore algorithm 471
9.5 The Rabin-Karp algorithm 482
9.6 Comparing the matching algorithms 489
9.7 String compression 490
9.8 Run-length encoding (RLE) 491
9.9 Lempel-Ziv-Welch (LZW) 497
9.10 Common mistakes 507
9.11 Summary 508
9.12 Key terms 508
9.13 Test Yourself 509
9.14 Further reading 511
9.15 References 511

10 Advanced String Algorithms 513

10.1 Four families of problems 514
10.2 Lexicographic order 515
10.3 Radix sort 520
10.4 Regular expressions 528
10.5 Finite state machines 537
10.6 Edit distance 549
10.7 The suffix array 559
10.8 The Burrows–Wheeler transform 565
10.9 Common mistakes 573
10.10 Summary 574
10.11 Key terms 575
10.12 Test Yourself 575
10.13 Further reading 578
10.14 References 578

11 Tries 579

11.1 The trie concept 580
11.2 The node class 586
11.3 Insertion 588
11.4 Search and prefix queries 593
11.5 Autocomplete and the longest common prefix 603
11.6 Deletion 607
11.7 Analysis 612
11.8 Applications and variants 618
11.9 Common mistakes 630
11.10 Summary 632
11.11 Key terms 633
11.12 Test Yourself 633
11.13 Further reading 635
11.14 References 635

APPENDICES

A **Glossary of Terms** 637

B **Prerequisites for Algorithms** 644

C **Important People in Algorithms** 664

ALGORITHMS

Course Map

This page puts every chapter of the course on one map. Each chapter is a node, and an arrow shows which idea from an earlier topic the next one relies on. Read it before the course starts to see where the topics are heading, or during the course to see where you are.

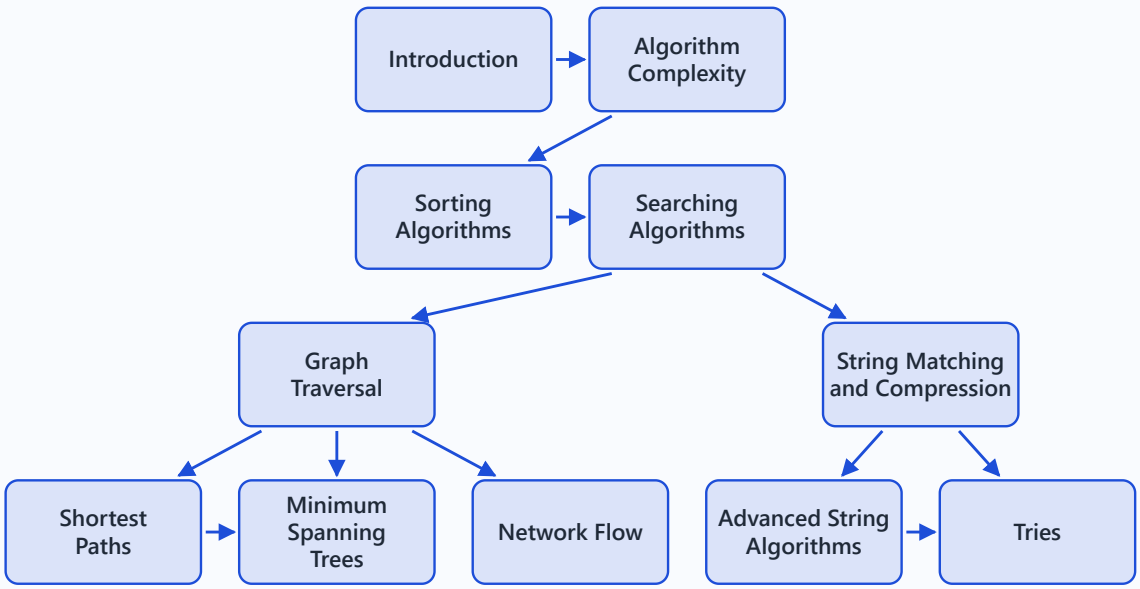


Figure 1: The course map. Each arrow points from a topic to one that builds on it. From analysis the path leads to sorting and searching, and from there to graph algorithms on the left and string algorithms on the right.

Figure 1 shows the connections between the topics. Each node is a chapter, and an arrow indicates which idea of the earlier topic the later one relies on. Only the closest connections are drawn on the map; every chapter that a given chapter depends on is listed in Table 1.

Topic (chapter)	Builds on	Connection
Introduction (1)	—	The notion of an algorithm, input size, and how running time grows with the input; counting operations line by line.
Algorithm Complexity (2)	Introduction	Asymptotic notation, loop analysis, recurrence relations and the Master Theorem: the yardstick for every later chapter.
Sorting Algorithms (3)	Algorithm Complexity	The first large class of problems; the move from $O(n^2)$ methods to $O(n \log n)$ methods is measured by analysis.
Searching Algorithms (4)	Algorithm Complexity, Sorting Algorithms	Binary search in a sorted array; average $O(1)$ search with hashing.

Topic (chapter)	Builds on	Connection
Graph Traversal (5)	Algorithm Complexity, Searching Algorithms	Searching is carried over to graphs: DFS and BFS, strongly connected components, topological sorting.
Shortest Paths (6)	Graph Traversal	Weights are added to traversal: relaxation, Dijkstra, Bellman-Ford, Floyd-Warshall and A*.
Minimum Spanning Trees (7)	Graph Traversal, Shortest Paths	The tree that connects all vertices at the lowest cost; Prim uses the same priority-queue pattern as Dijkstra.
Network Flow (8)	Graph Traversal	Augmenting paths are found by traversal (BFS in Edmonds-Karp); the max-flow min-cut theorem.
String Matching and Compression (9)	Algorithm Complexity, Searching Algorithms	Pattern search in text: KMP, Boyer-Moore, Rabin-Karp; compression with RLE and LZW.
Advanced String Algorithms (10)	Sorting Algorithms, String Matching and Compression	String sorting (radix sort), regular expressions and automata, edit distance, the suffix array.
Tries (11)	String Matching and Compression, Advanced String Algorithms	A tree that stores strings by their common prefixes; autocomplete, the suffix trie and Aho–Corasick.

Table 1: The connections on the map.

The backbone of the map can be read as **analysis → sorting and searching → graphs and strings**. Counting operations and asymptotic analysis in the first two chapters are the yardstick for every later chapter. Sorting and searching are both the tools and the examples for what follows: traversing a graph is searching carried over to graphs, and the shortest path, spanning tree and flow algorithms are built on traversal. String matching is searching applied to text; advanced string algorithms and the trie grow out of this foundation. Design paradigms, hard problems and other models of computation are built on this foundation in the Advanced Algorithms course.

PART I

Foundations

Chapters 1–4

PART I · CHAPTER 1

Introduction

Everything a computer does rests on a list of steps that someone wrote down beforehand. That list is called an algorithm. Many algorithms can solve the same problem, and they can differ enormously in speed: one finishes in milliseconds on a million items, another needs years for the same job. These notes begin with what an algorithm is and which properties a good one has. They then show how running time is measured, not in seconds but in the number of operations as a function of input size. Short Java programs with constant, linear, square-root, cubic, factorial, and exponential running times are traced line by line. The last topic is Euclid's algorithm, one of the oldest algorithms we know.

LEARNING OUTCOMES

Algorithm
State the definition and the properties of an algorithm (input, output, definiteness, finiteness, effectiveness, correctness); write an everyday task as an algorithm and show which property a given procedure fails.

Analysis
Express running time as a function $T(n)$ of the input size n; count the operations of a short program line by line; read from a growth table how running time changes as n grows.

Complexity examples
Find, with justification, the complexity class of short programs (1, n, $\sqrt{n}$, n^3, $n!$, 2^n); explain how memoization reduces an exponential running time to a linear one.

Euclid's algorithm
Trace the computation of the GCD step by step; justify the algorithm's correctness and its $O(\log \min(a, b))$ running time; explain why the worst-case inputs are consecutive Fibonacci numbers.

1.1 The concept of an algorithm

The word “algorithm” comes from the Latinized form of the name of al-Khwarizmi, a mathematician who lived in Baghdad in the 9th century. His book on calculating with Hindu numerals became known in Europe through a Latin translation that begins with the words “Dixit Algorizmi” (“Al-Khwarizmi says”). Over time the name came to denote any method for carrying out a particular calculation step by step. Today behind every problem a computer program solves there is an algorithm: a search engine ranking its results, a map finding the shortest route, a photograph being compressed.

1.1.1 What is an algorithm?

A recipe, an assembly manual for a piece of furniture, and the method of long division by hand all share one structure. Each starts with something in hand: ingredients, parts, two numbers. Then come steps in a fixed order. When the steps are done, the desired result appears: a dish, a cupboard, a quotient and a remainder. An algorithm is this structure as computer science defines it, with precise rules.

Definition 1.1

An **algorithm** is a finite sequence of instructions, each precisely defined, for solving a problem or performing a task. It takes input data, performs operations on it, and produces an **output**.

Three parts of the definition stand out. The first is *input and output*: an algorithm takes something in and gives something back. The second is *definiteness of the steps*: each step can be understood in only one way. The third is *finiteness*: the steps do not go on forever. Each of the three is treated below as a separate property.

An algorithm need not be written in a programming language. It can be described in English sentences, numbered steps, a flowchart, or pseudocode. A program is the algorithm written in a particular language to run on a particular machine. The same algorithm can be written in Java, Python, or C; each version is a separate program, but all of them follow the same algorithm. This distinction is the basis of algorithm analysis: whether an algorithm is good or bad is decided independently of the language it is written in and the machine that runs it.

1.1.2 Properties of an algorithm

Not every list of steps counts as an algorithm. Donald Knuth names five properties: input, output, definiteness, finiteness, and effectiveness. In practice a sixth is added, correctness. Table 1.1 lists the six with short explanations; each is then discussed in turn.

Property	Meaning
Input	Zero or more values are supplied from outside.
Output	At least one result is produced, and it is related to the input.
Definiteness	Each step has exactly one meaning and leaves no room for interpretation.
Finiteness	For every input, it stops after a finite number of steps.
Effectiveness	Each step is a basic operation that can actually be carried out.
Correctness	For every valid input, it produces the correct output.

Table 1.1: The properties of an algorithm.

Input. An algorithm receives values from outside, either before it starts or while it runs. The number of inputs may be zero. For example, the task “print the first 100 primes” takes no value from outside, yet it is a perfectly valid algorithm. In most algorithms, though, the input is the problem itself: the array to be sorted, the number to be found, the road network between two cities.

Output. An algorithm produces at least one result, and the result is related to the input. A list of steps that produces nothing is useless, however long it runs. The output need not be a value printed on the screen. A value returned by a method, a modified array, or a yes/no decision is also an output.

Definiteness. Each step is defined precisely and can be understood in only one way. “Add a little salt” may be enough for a person, but it is meaningless to a machine. An algorithm requires that everyone who reads a step does the same thing. This is why phrases such as “a little,” “enough,” or “a suitable value” do not belong in an algorithm.

Finiteness. For every input, an algorithm stops after a finite number of steps. A program whose loop never ends on some inputs is not an algorithm for those inputs. The most common way to lose finiteness is a loop condition that never becomes false. For example, a loop that subtracts 2 from a number on every round and stops when it reaches zero skips over zero for odd numbers and runs forever.

Effectiveness. Each step is basic enough to be carried out in finite time, with pencil and paper or on a machine. “Find the prime factors of the number” cannot be written as a single step, because it is a problem in its own right; it becomes part of an algorithm only after it is broken into more basic steps. “Try every real number one by one” can never be carried out, because there are uncountably many real numbers.

Correctness. For every valid input, an algorithm produces the correct output. Giving the right answer on a few examples is not enough; correctness must be shown for all valid inputs, which usually takes a proof. The correctness of Euclid's algorithm is shown by such a proof in the last section of these notes.

Example 1.1

The following method is meant to find the number of digits of an integer: "1. Counter = 0. 2. As long as the number is greater than 0, divide it by 10 (integer division) and increase the counter by 1. 3. Print the counter." Which properties does this method satisfy?

1. **Input and output:** it takes an integer and prints a counter value. Both are present.
2. **Definiteness:** each step has one meaning: assignment, comparison, integer division, increment.
3. **Effectiveness:** all of these operations can be done in a single machine step.
4. **Finiteness:** a positive number shrinks every time it is divided by 10 and eventually becomes 0. For example $4075 \to 407 \to 40 \to 4 \to 0$; it stops after four divisions and the counter is 4. If the number is 0 or negative, the loop is never entered and the method stops at once.
5. **Correctness:** for positive numbers the result is correct. But for 0 the output is 0, although the number 0 has one digit. For negative numbers the output is also 0. The method gives wrong results on these inputs. Either the set of inputs must be restricted to positive numbers or these cases must be handled separately.

The example also shows that correctness depends on the set of inputs. An algorithm must be correct "for every valid input," and which inputs are valid is stated at the outset in the definition of the problem. If the problem definition stays vague, whether the algorithm is correct stays vague too.

1.1.3 An everyday example: making tea

An easy way to make the idea of an algorithm concrete is to write down a familiar everyday task step by step. The input to making tea is water, tea leaves, and optionally sugar and milk; the output is a cup of tea. The algorithm in the slides has six steps:

1. Boil the water.
2. Pour the boiled water into the teapot.
3. Add the tea leaves to the teapot.
4. Wait for the tea to brew.
5. Strain the brewed tea into a cup.
6. Optionally add sugar and milk.

This short list shows three important qualities of an algorithm.

Order matters. An algorithm is an ordered list of steps. The position of some steps can change: if steps 2 and 3 swap places, that is, if the leaves go into the teapot first and the boiling water is added afterward, the result is the same. But steps 1 and 5 cannot swap. Tea cannot be strained before the

water has boiled; there would be nothing to strain. When writing an algorithm, one thinks carefully about which step must come before which, that is, about the *dependencies* between steps.

Definiteness is tested separately for each reader. The step "wait for the tea to brew" is enough for a person; an experienced one looks at the color of the tea and decides. For a computer the step is ambiguous: how long should it wait? For a machine the step would have to read "wait 10 minutes." This is exactly what definiteness demands: everyone who reads a step must be able to do the same thing without interpreting it.

Algorithms contain decisions. Step 6 carries a condition: "if sugar is wanted, add sugar; if milk is wanted, add milk." The path the algorithm follows can depend on the input. In programming this corresponds to the `if` statement. Likewise a step such as "wait until the water boils" is a loop. Sequences of steps, decisions, and loops are the building blocks of every algorithm.

> **Common mistake**
>
> Counting a recipe in everyday language as an algorithm as it stands. Phrases such as "by eye," "until it reaches the right consistency," or "pick a suitable number" destroy definiteness. Whether a step is definite is tested by asking whether someone who knows nothing about it (or a machine) could do the same thing.

1.1.4 From algorithm to code: computing an average

Finding the average of three numbers is a small, convenient example of the relationship between an algorithm and a program. The input is three numbers (Number1, Number2, Number3), and the output is their average. The algorithm has three steps:

1. Sum = Number1 + Number2 + Number3
2. Average = Sum / 3
3. Print the value of Average.

Written in Python, the algorithm looks like this:

```
def average(a, b, c):
    avg = (a + b + c) / 3
    return avg

a = float(input("Enter the first number: "))
b = float(input("Enter the second number: "))
c = float(input("Enter the third number: "))
result = average(a, b, c)
print("Average of the numbers:", result)
```

The same algorithm in Java is shown below. The code is written as the `main` method of a class; the `Scanner` class needs the line `import java.util.Scanner;` at the top of the file.

```
public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  System.out.println("Enter the first number: ");
  double a = in.nextDouble();
  System.out.println("Enter the second number: ");
  double b = in.nextDouble();
  System.out.println("Enter the third number: ");
  double c = in.nextDouble();
  double avg = (a + b + c) / 3;
  System.out.println("Average of the numbers: "
                     + avg);
}
```

The two programs look different. In Python the computation is separated into a function, and the inputs are read with `input` and converted with `float`. In Java everything is inside `main`, the inputs are read through a `Scanner` object with `nextDouble()`, and the type of each variable (`double`) is written out. But both programs follow the same three steps: add the three numbers, divide the sum by 3, print the result. For example, for inputs 4, 7, and 10 both programs print 7.0.

This observation shows that an algorithm is **independent of the language**. A program is the algorithm's counterpart in a particular language. To understand and analyze an algorithm it does not matter which language it is written in. In these notes the code is given in Java, but the idea behind each piece of code carries over unchanged to any other language.

Common mistake

Declaring the variables in Java as `int` and writing `(a + b + c) / 3`. When two integers are divided, Java discards the fractional part: for 4, 7, and 9 the result is 6 instead of 6.67. The algorithm is correct, but the choice of type in the program implements it wrongly. This is why the code in the slides uses `double`; when working with integers, writing `/ 3.0` is enough.

1.1.5 Kinds of algorithms

Algorithms are grouped by the problem they solve or by the design idea they follow. This course covers basic algorithms from each of the groups below.

- **Sorting algorithms** put items into order. Insertion sort, merge sort, and quicksort are the best-known members of this group. Because sorted data speeds up searching and many other operations, sorting is one of the most studied problems in computer science.

- **Searching algorithms** find a particular item in a collection of data. Linear search examines the items one by one. Binary search halves the range at each step in a sorted array. A hash table computes the location of the item directly.
- **Graph algorithms** work on structures made of vertices and the edges that connect them. Dijkstra's algorithm finds the shortest path between two points. Minimum spanning tree algorithms connect all vertices at the least total cost. The Ford–Fulkerson method computes the maximum flow that can pass through a network.
- **String and compression algorithms** work with text. Searching for a word in a text, storing words in a tree called a trie, and shrinking a file (Huffman coding, gzip) belong to this group.
- **Design techniques** describe not a single problem but a general solution idea that applies to many problems. Divide and conquer (merge sort, binary search, fast exponentiation), dynamic programming, greedy algorithms, backtracking, and branch and bound are the main ones.
- **Randomized and approximation algorithms** offer two different routes for hard problems. A randomized algorithm makes random choices in some of its steps. An approximation algorithm, for problems where finding the best solution would take far too long, finds quickly a solution that is provably within a given ratio of the best.

These groups are not mutually exclusive. Merge sort is both a sorting algorithm and an example of the divide-and-conquer technique. Quicksort with a randomly chosen pivot is at the same time a randomized algorithm.

1.1.6 Deterministic and randomized algorithms

Another way to classify algorithms is to ask whether they always follow the same path when given the same input.

> **Definition 1.2**
>
> A **deterministic** algorithm follows the same steps and produces the same output on every run with the same input. A **randomized algorithm**, in contrast, makes random choices in some of its steps; so on the same input, different runs may follow different paths.

An algorithm that finds the largest element of an array is deterministic: if the array is the same, so are the elements compared and the order of comparison. Quicksort with a random pivot, on the other hand, may split the array at different points on each run. In a sense this algorithm rolls dice. But the result does not change; the array is sorted correctly every time. Only the path followed, and therefore the running time, changes.

What randomness buys an algorithm is not obvious at first glance. If quicksort always picks the first element as the pivot, an already sorted array drives the algorithm into its slowest behavior. A random pivot removes the notion of a “bad input”: no input gives a bad result on every run, and the expected

time stays equally short on every input. Some randomized algorithms gain a great deal of speed by accepting a very small probability of giving a wrong answer. All the algorithms in the rest of these notes are deterministic.

1.2 Algorithm analysis

Finding an algorithm that solves a problem is only half the job. If several algorithms solve the same problem, one also has to decide which to choose. The tool for that decision is algorithm analysis. This section explains what analysis measures, why the measurement is made in operation counts rather than seconds, and how far apart different algorithms drift as n grows.

1.2.1 Why analyze?

Definition 1.3

Algorithm analysis studies how much of its resources, such as processor time and memory, an algorithm uses as a function of input size. The time requirement is called **time complexity**, and the extra memory requirement is called **space complexity**.

A small example shows the value of analysis clearly. Suppose we search for a number in a sorted array of one million (10^6) elements. Linear search starts at the beginning of the array and examines the elements one by one. If the number is not in the array, or is at the very end, it makes one million comparisons. Binary search looks at the middle element and continues in the left half if the target is smaller and in the right half if it is larger. Because each comparison halves the range, it makes at most 20 comparisons, since one million is about 2^{20}. Figure 1.1 compares the two numbers on a logarithmic scale.



Figure 1.1: The number of comparisons in the worst case on a sorted array of one million elements. The bar lengths are on a logarithmic scale; on a true scale the bar for binary search would be too short to see.

Both algorithms are correct; both find the number. The difference between them lies only in resource use, and it is a factor of fifty thousand. Analysis makes it possible to see such a difference in advance, without writing the program, just by looking at the algorithm's steps. Which algorithm to choose is often decided by this kind of comparison.

1.2.2 Input size and the basic operation

The first way that comes to mind to measure how long an algorithm takes is to run the program and time it. But the measurement depends on many things: the speed of the processor, the programming language, the compiler, the other programs running at the same time. The same code may take 2 seconds on a laptop and 0.3 seconds on a server. These numbers tell us about the machine, not about the algorithm.

Algorithm analysis therefore does not measure time in seconds. Instead it counts the **basic operations** the algorithm performs. A basic operation is one that takes constant time on a machine: adding or comparing two numbers, accessing an array cell, assigning a value to a variable. The operation count does not change from machine to machine. A fast machine performs each operation in less time, but the number of operations stays the same.

The operation count does depend on the input. Summing an array of five elements takes less work than summing an array of a million. The operation count is therefore written as a function of a number that measures the size of the input.

Definition 1.4

The **input size** n is the number that measures the size of the problem: the number of elements in an array, the number of rows of a matrix, the number of vertices of a graph. The number of basic operations an algorithm performs on an input of size n is denoted $T(n)$ and is called the algorithm's **running time**.

What counts as the input size is chosen according to the problem. For an algorithm that processes an array, the natural choice is the number of elements. For an algorithm that multiplies two $n \times n$ matrices, it is n. For an algorithm that takes a single number as input there are two options: the value of the number, or the number of digits needed to write it. The two options give very different results; we return to this distinction when we study primality testing.

Two inputs of the same size can also require different numbers of operations. Linear search finishes with one comparison if the target is in the first cell of the array; if it is not in the array at all, it makes n comparisons. In analysis one therefore often studies the **worst case**: the input that requires the most operations among all inputs of the same size. The worst case gives a guarantee. The algorithm never performs more operations than this on any input.

1.2.3 Counting line by line: the sum of an array

To see how operations are counted, consider a short program that computes the sum of an array of n elements. Next to each line of the code in the slides is written how many times that line runs.

```
int sum = 0;                    // 1 time
for (int i = 0; i < n; i++)      // n+1 comparisons
  sum += a[i];                   // n times
return sum;                      // 1 time
```

The count is done line by line.

- `int sum = 0;` runs once, before the loop: **1** operation.
- The comparison `i < n` in the `for` line is made once each time the loop is entered. While i is 0, 1, ..., $n - 1$ the condition is true and the loop body runs; that makes n comparisons. When $i = n$ the condition is tested once more, turns out false, and the loop ends. In all, $\boldsymbol{n + 1}$ comparisons.
- `sum += a[i];` is the loop body and runs $\boldsymbol{n}$ times.
- `return sum;` runs once: **1** operation.

The sum of these counts gives the running time:

$$T(n) = 1 + (n + 1) + n + 1 = 2n + 3$$

Example 1.2

For `a = {4, 1, 3}`, counting step by step ($n = 3$):

1. `sum = 0` : 1 operation.
2. $i = 0$: `0 < 3` is true (1 comparison), `sum = 4` (1 operation).
3. $i = 1$: `1 < 3` is true, `sum = 5` .
4. $i = 2$: `2 < 3` is true, `sum = 8` .
5. $i = 3$: `3 < 3` is false (1 comparison); the loop ends.
6. `return 8` : 1 operation.
7. **Total:** 1 + 4 comparisons + 3 additions + 1 = 9 operations. The formula agrees: $T(3) = 2 \cdot 3 + 3 = 9$.

In this count the `i++` increments and the `a[i]` accesses were not counted as separate operations. Had they been, the result would have been a different expression such as $4n + 3$. Which operations to count is a choice, and the choice changes the coefficients. But one thing does not change: the operation count is always a multiple of n plus a constant. If the array doubles in size, the work roughly doubles.

1.2.4 Why do constants become irrelevant?

The numbers 2 and 3 in the expression $2n + 3$ depend on the machine and on the way of counting. On one machine a comparison may be twice as slow as an addition; a compiler may merge two operations into one instruction. Knowing these constants exactly is both hard and not very useful. What matters is how fast $T(n)$ grows as n grows.

For $n = 10$, $2n + 3 = 23$, and the constant term 3 makes up about an eighth of the total. For $n = 10^6$, $2n + 3 = 2{,}000{,}003$, and the contribution of the 3 drops to about one part in a million. For large n only the fastest-growing term decides. Its coefficient also becomes irrelevant, because the coefficient affects every n by the same factor: it doubles the time but does not change the shape of the growth.

This idea leads to classifying algorithms by their growth rates. The functions $2n + 3$, $5n$, and $n/2 + 100$ all grow in proportion to n. They are called **linear** and are written briefly as $\Theta(n)$. The Θ (theta) notation discards constant coefficients and lower-order terms and keeps only the growth rate. Similarly, the $O(n)$ (big O) notation means "grows at most in proportion to n"; that is, it gives an upper bound. In these notes both notations are used in these intuitive senses. For $T(n) = 2n + 3$ one can write both $T(n) = \Theta(n)$ and $T(n) = O(n)$.

Example 1.3

What is the growth rate of $T(n) = 5n^2 + 100n \log_2 n + 7$? Which term dominates, and when?

1. The terms are $5n^2$, $100n \log_2 n$, and 7. The constant 7 becomes irrelevant at once for large n.
2. $n = 100$: $5n^2 = 50{,}000$, $100n \log_2 n \approx 100 \cdot 100 \cdot 6.64 \approx 66{,}439$. At this point the second term is larger, because its coefficient is 20 times greater.
3. $n = 1000$: $5n^2 = 5{,}000{,}000$, $100n \log_2 n \approx 996{,}578$. Now the first term is five times larger.
4. The ratio of the two terms is $5n^2 / (100n \log_2 n) = n / (20 \log_2 n)$, which grows without bound as n grows. A calculation shows that the first term is ahead for good from $n = 144$ on.
5. Result: $T(n) = \Theta(n^2)$. The growth rate is determined not by the term with the larger coefficient but by the term that increases fastest as n grows.

1.2.5 The growth table

The running times of algorithms fall into a few familiar families of functions: constant (1), logarithmic ($\log n$), square-root ($\sqrt{n}$), linear (n), linearithmic ($n \log n$), quadratic (n^2), cubic (n^3), exponential (2^n), and factorial ($n!$). The best way to see how these functions behave as n grows is to put their values side by side. Table 1.2 gives the growth table from the slides.

f(n)	n = 10	n = 100	n = 1000	$n = 10^6$	time ($n = 10^6$)
1	1	1	1	1	1 ns

f(n)	n = 10	n = 100	n = 1000	$n = 10^6$	time ($n = 10^6$)
$\log_2 n$	3.3	6.6	10.0	19.9	19.9 ns
$\sqrt{n}$	3.2	10	31.6	1,000	1 µs
n	10	100	1,000	1,000,000	1 ms
$n \log_2 n$	33.2	664.4	9,966	$2.0 \cdot 10^7$	19.9 ms
n^2	100	10,000	1,000,000	$1.0 \cdot 10^{12}$	16.7 min
n^3	1,000	1,000,000	$1.0 \cdot 10^9$	$1.0 \cdot 10^{18}$	≈ 32 years
2^n	1,024	$1.3 \cdot 10^{30}$	$\approx 1.1 \cdot 10^{301}$	$\approx 10^{301030}$	far longer than the age of the universe
$n!$	3,628,800	$9.3 \cdot 10^{157}$	$\approx 10^{2567}$	$\approx 10^{5565708}$	far longer than the age of the universe

Table 1.2: The growth table. The last column is the time needed for $n = 10^6$ on a computer that performs 10^9 operations per second.

Reading the table row by row yields several important observations.

Constant and logarithmic growth hardly grow at all. When n increases from 10 to 10^6, a factor of a hundred thousand, $\log_2 n$ rises only from 3.3 to 19.9. The logarithm answers the question "how many times must I halve n to get down to 1?" Halving a million twenty times brings it down to 1. The 20 comparisons of binary search come from this number.

The square root grows much more slowly than the linear function. For $n = 10^6$, $\sqrt{n}$ is only 1000. Reducing an algorithm from n to $\sqrt{n}$ cuts the work by a factor of a thousand on an input of a million.

***n log n* is very close to linear.** For a million elements, $n \log_2 n$ is about $2 \cdot 10^7$, only twenty times the linear time. On a machine that performs 10^9 operations per second this is 20 milliseconds. Efficient sorting algorithms belong to this class.

Even among polynomials there are huge gaps. On the same input of a million, n^2 operations take 16.7 minutes and n^3 operations take about 32 years. A quadratic algorithm causes no trouble on small inputs, but becomes unusable as the input grows.

Exponential and factorial growth exceed every limit. 2^n reaches $1.3 \cdot 10^{30}$ even at $n = 100$. At 10^9 operations per second that is about $4 \cdot 10^{13}$ years, while the age of the universe is about $1.4 \cdot 10^{10}$ years. $n!$ is faster still: it reaches 3,628,800 at $n = 10$ and a number with 158 digits at $n = 100$. Algorithms in these two classes are usable only on very small inputs.

The ordering in the table is used constantly in the rest of the course:

> $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$

Here the sign "<" means that for sufficiently large n the function on the left stays below the one on the right, and moreover that the ratio between them grows without bound. For small values of n the order can break down: at $n = 10$, $\sqrt{n} \approx 3.2$ is smaller than $\log_2 n \approx 3.3$. At $n = 16$ both equal 4, and from then on $\sqrt{n}$ is always ahead.

1.2.6 A faster computer or a better algorithm?

There are two ways to speed up a slow program: buy a faster computer or find a better algorithm. The growth table shows that the two have very different effects.

A faster computer performs each operation in less time. This amounts to shrinking the constant coefficient in $T(n)$. A machine a thousand times faster speeds up every algorithm a thousand times, but it does not change the shape of the growth. A better algorithm, on the other hand, changes the growth rate. Using a sort that performs $n \log n$ operations instead of one that performs n^2 on a million elements brings the time down from 16.7 minutes to 20 milliseconds. That is a gain of about fifty thousand, and it grows further as n grows.

The same comparison can be made the other way round: what is the largest input that can be solved in a given time? Table 1.3 gives the largest values of n that a computer performing 10^9 operations per second can handle in one second.

f(n)	Condition	Largest n
n	$n \le 10^9$	10^9
$n \log_2 n$	$n \log_2 n \le 10^9$	$\approx 3.96 \cdot 10^7$
n^2	$n^2 \le 10^9$	31,622
n^3	$n^3 \le 10^9$	1000
2^n	$2^{29} \approx 5.4 \cdot 10^8 \le 10^9 < 2^{30}$	29
$n!$	$12! \approx 4.8 \cdot 10^8 \le 10^9 < 13!$	12

Table 1.3: The largest n that can be solved in one second at 10^9 operations per second.

Now suppose the computer becomes a thousand times faster. For the linear algorithm the solvable input also grows a thousandfold. For the quadratic algorithm it grows only by a factor of $\sqrt{1000} \approx 31.6$, and for the cubic algorithm by a factor of 10. For the exponential algorithm, since $2^{10} \approx 1000$, n grows by only 10: from 29 to 39. A machine a thousand times faster lets an exponential algorithm handle ten more elements. This is why in algorithm design the real gain comes not from hardware but from ideas that lower the growth rate.

Common mistake

Judging an algorithm's speed from a single measurement in seconds. The verdict “my program finished in 0.2 seconds, so it is fast” holds only for that machine and that input size. When the input grows tenfold, a linear algorithm takes 2 seconds and a cubic algorithm takes 200. Analysis looks not at seconds but at the operation count as a function of n and at how fast it grows.

1.3 Complexity examples

Behind every row of the growth table there are real algorithms. This section examines a short Java program from each class. For each program the same question is asked: what is the input size n, which operation is counted, and how many times, as a function of n, is that operation performed? Most of the programs are traced line by line, with the intermediate values given in a table.

1.3.1 Constant time: printing the first element

The fastest algorithms are those that perform the same number of operations however large the input is. A method that prints the first element of an array is of this kind.

```
void printFirst(int[] a) {
  if (a.length > 0)
    System.out.println("First element: "
                       + a[0]);
  else
    System.out.println("Array is empty");
}
```

The method first tests whether the array is empty with the condition `a.length > 0`. If the array is not empty, it accesses the cell `a[0]` and prints its value; if it is empty, it prints the message “Array is empty”. In Java the length of an array is stored in the array object; reading `length` does not require counting the elements. Accessing any cell of an array also takes a single step, because the address of the cell is computed directly from the starting address.

So whether the array has 10 elements or 10 million, the method performs one comparison, one array access, and one print. For example, called with `{5, 8, 12, 4}` it prints “First element: 5”, and called with an empty array it prints “Array is empty”. The operation count is independent of n: $T(n) = O(1)$.

Definition 1.5

An algorithm whose running time is independent of the input size and bounded by a constant is called **constant time** and its running time is written $O(1)$. An algorithm whose running time grows in direct proportion to n is called **linear time**: $O(n)$.

$O(1)$ does not mean “a single operation.” A method that performs three operations and a method that performs a thousand are both constant time if that number does not depend on n. What matters is that the operation count does not change when the input grows.

1.3.2 Linear time: finding the largest element

To find the largest element of an unsorted array one must look at every element at least once. An element that is not looked at could be the largest. The `findMax` method in the slides does this in a single pass.

```
Integer findMax(int[] a) {
  if (a.length == 0)
    return null;            // empty array
  int max = a[0];
  for (int x : a)
    if (x > max)
      max = x;
  return max;
}
```

The return type of the method is `Integer`, not `int`. The reason is the empty-array case: an empty array has no largest element, and the method reports this by returning `null`. The type `int` cannot hold `null`; `Integer` is an object reference and can.

If the array is not empty, the variable `max` is initialized with the first element. Then the loop `for (int x : a)` walks through all the elements of the array in order. This “for each” loop delivers the elements from beginning to end without using an index; the first element is part of the traversal too. Each element is compared with `max`. If the element is larger, `max` is updated. When the loop ends, `max` is the largest of all the elements seen so far; and since the loop traversed the whole array, it is the largest element of the array.

Table 1.4 shows the method running on the array `{5, 8, 12, 4}`; Figure 1.2 shows the same trace with the figure from the slides.

Round	x	x > max?	max (end of round)
start	—	—	5
1	5	5 > 5 false	5
2	8	8 > 5 true	8
3	12	12 > 8 true	12
4	4	4 > 12 false	12

Table 1.4: Trace of `findMax({5, 8, 12, 4})`.

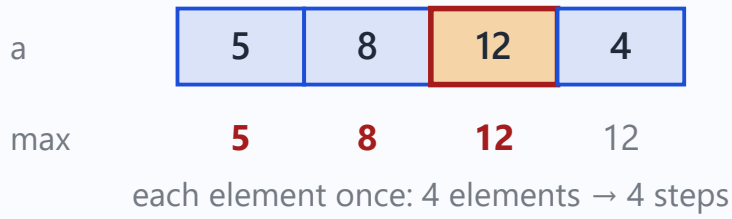


Figure 1.2: `findMax` looks at each element once. The top row shows the elements of the array; the bottom row shows the value of `max` after that element has been examined. After 12 is found the value does not change.

On the four-element array the method makes four comparisons and returns 12. In general, on an array of n elements exactly n comparisons are made; this number does not depend on the contents of the array. The number of updates does depend on the contents. If the array is in decreasing order (`{9, 5, 3, 2}`) there are no updates at all. If it is in increasing order (`{2, 3, 5, 9}`), every element is larger than the one before, so $n - 1$ updates are made. In both cases the total work is proportional to n: $T(n) = \Theta(n)$.

The extra memory the method uses consists of the variables `max` and `x`. These two variables suffice however large the array is. The space complexity is therefore $O(1)$. The input array itself is not counted in the space; space complexity measures the memory the algorithm allocates in addition to the input.

Common mistake

Initializing `max` with `0`. If all the numbers in the array are negative, for example `{-7, -3, -9}`, no element is greater than 0, so the method wrongly returns 0. The starting value must be an element of the array itself (`a[0]`). This is why the code in the slides first handles the empty array and then starts with `a[0]`.

1.3.3 Square-root time: primality testing

Definition 1.6

An integer greater than 1 that has no positive divisors other than 1 and itself is called a **prime number**. The numbers 2, 3, 5, 7, 11, 13 are prime. Numbers greater than 1 that are not prime are called **composite numbers**: $9 = 3 \cdot 3$, $91 = 7 \cdot 13$.

The most direct way to tell whether a number n is prime is to check whether any number from 2 to $n - 1$ divides n. This method performs $n - 2$ divisions; for 97 it needs 95. But most of these divisions are unnecessary. The following theorem says that the search can be ended much earlier.

Theorem 1.1 (Smallest divisor)

If $n > 1$ is composite, then n has a divisor greater than 1 that is less than or equal to $\sqrt{n}$.

Proof. *Proof.* Since n is composite we can write $n = a \cdot b$ with $1 < a, b < n$. If both factors were greater than $\sqrt{n}$, their product would exceed $\sqrt{n} \cdot \sqrt{n} = n$. This contradicts $a \cdot b = n$. So at least one of a and b is less than or equal to $\sqrt{n}$, and that factor is the required divisor. ■

The contrapositive of the theorem says: if there is no divisor up to $\sqrt{n}$, then n is prime. For example, in $91 = 7 \cdot 13$ the smaller factor is 7, and $7 \le \sqrt{91} \approx 9.5$. There is no need to look at the larger factor 13 at all; once 7 is found, we know the number is composite.

A second saving is possible. An even number greater than 2 cannot be prime, because it is divisible by 2. An odd number, on the other hand, is not divisible by any even number, since an even divisor would make the number even too. So we first check whether the number is even; for odd numbers we try only odd divisors. The `isPrime` method in the slides combines these two ideas.

```
boolean isPrime(int n) {
  if (n <= 1)
    return false;
  else if (n == 2)
    return true;
  else if (n % 2 == 0)
    return false;
  int limit = (int) Math.sqrt(n) + 1;
  for (int i = 3; i < limit; i += 2)
    if (n % i == 0)
      return false;
  return true;
}
```

The first three branches of the method separate out the special cases. Numbers 1 and below are not prime by definition. 2 is the only even prime. Even numbers greater than 2 are not prime. These three branches return a result without entering the loop.

For odd numbers, `limit = (int) Math.sqrt(n) + 1` is computed. `Math.sqrt` returns the square root as a `double`; the `(int)` conversion discards the fractional part and leaves $\lfloor\sqrt{n}\rfloor$. The loop condition `i < limit` means $i \le \lfloor\sqrt{n}\rfloor$, so the integer part of $\sqrt{n}$ is also tried. The loop starts at 3 and with `i += 2` tries only odd numbers. If a divisor is found, the method returns `false` at once. If the loop ends without finding a divisor, the number is prime and the method returns `true`.

How many divisions are performed? The worst case is when the number is prime, because then the loop runs to the end. The loop advances from 3 to $\sqrt{n}$ in steps of two. There are about $\sqrt{n} / 2$ odd numbers in this range. Dropping the constant coefficient $1/2$ gives $T(n) = O(\sqrt{n})$. A few concrete values confirm this count:

- *n = 97*: $\lfloor\sqrt{97}\rfloor = 9$; the divisors tried are 3, 5, 7, 9: **4 divisions**. Trying every number from 2 to 96 would have taken 95 divisions.
- *n = 1,000,000,007* (prime): $\sqrt{n} \approx 31{,}622.8$; the odd numbers tried are 3, 5, ..., 31,621: **15,810 divisions**. The simple method would have performed about a billion divisions.

If the number is composite, the loop usually ends much earlier. For 221, $\lfloor\sqrt{221}\rfloor = 14$ and `limit = 15`; 3, 5, 7, 9, 11 do not divide it, 13 does ($221 = 13 \cdot 17$), and the method returns `false` after 6 divisions.

1.3.4 Code trace: isPrime(97)

The trace slide in the presentation executes the call `isPrime(97)` line by line. Table 1.5 gives the variable values of this trace, and Figure 1.3 shows the initial and final states.

Step	Line	limit	i	97 % i	Explanation
1	2, 4, 6	—	—	—	97 > 1, 97 ≠ 2, 97 is odd: all three checks pass
2	8	10	—	—	(int) √97 + 1 = 9 + 1 = 10
3	9, 10	10	3	1	3 does not divide
4	9, 10	10	5	2	5 does not divide
5	9, 10	10	7	6	7 does not divide
6	9, 10	10	9	7	9 does not divide
7	9	10	11	—	11 < 10 is false: the loop ends
8	12	10	11	—	`return true`: 97 is prime

Table 1.5: Trace of `isPrime(97)`. The line numbers are lines of the code.

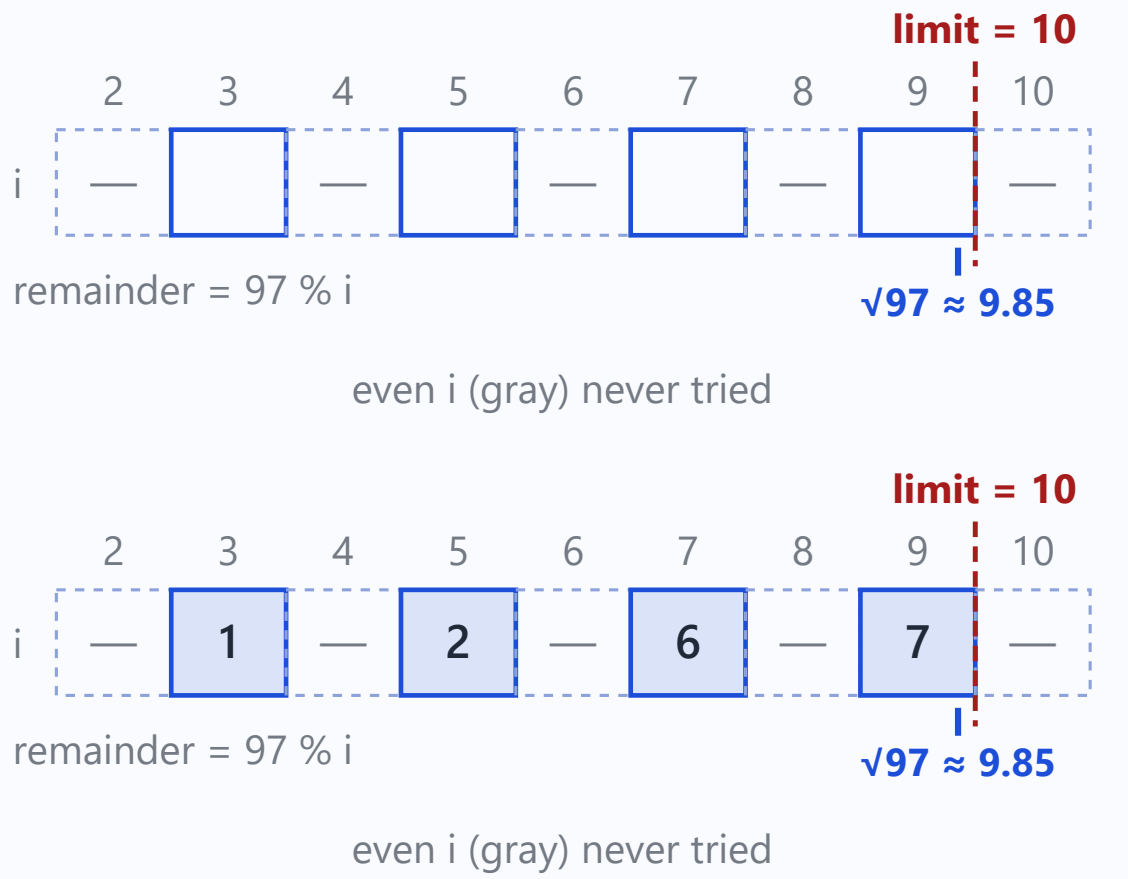


Figure 1.3: Two moments of the `isPrime(97)` trace. (a) `limit = 10` has been computed; the dashed vertical line marks the bound, and the label $\sqrt{97} \approx 9.85$ shows the true square root. The gray boxes are even numbers and are never tried. (b) The loop has ended: the remainders for 3, 5, 7, and 9 (1, 2, 6, 7) are written in; none of them is 0.

The notable point of the trace is the step $i = 11$. The increment `i += 2` makes `i` equal to 11 after 9. Because the loop condition `11 < 10` is false, the body does not run; 11 is never tried. There is no need to try any odd number above 10 either. If 97 had a divisor greater than $\sqrt{97}$, then by Theorem 1.1 it would also have a small divisor. That small divisor would already have been tried.

Common mistake

Setting the loop bound as `i < (int) Math.sqrt(n)`. Then $\lfloor\sqrt{n}\rfloor$ itself is not tried. For perfect squares the only small divisor can be exactly $\sqrt{n}$: 9, 25, 49, 121 come out prime. The error is not limited to perfect squares either. For numbers such as 15 ($\lfloor\sqrt{15}\rfloor = 3$) and 35 ($\lfloor\sqrt{35}\rfloor = 5$) the small divisor equals exactly $\lfloor\sqrt{n}\rfloor$ and is skipped too. Below 100,000 this faulty version gives a wrong result for 84 numbers. The `+ 1` in the slides puts the bound itself into the loop.

1.3.5 Input size: the value of the number or its length?

In primality testing we took the *value* of the number n as the input size and found a running time of $O(\sqrt{n})$. At first sight this looks even better than linear time. But the size of a number in a computer is measured not by its value but by the number of bits it takes to write it. The number 1,000,000,007 has a value of a billion, but is written with only 30 bits.

Let the bit length of the number be b. Then n is about 2^b, and

> $\sqrt{n} \approx \sqrt{(2^b)} = 2^{b/2}$

follows. So the number of divisions grows *exponentially* in the bit length. Adding one bit to the number doubles its value and multiplies the number of divisions by about $\sqrt{2} \approx 1.41$. For a 30-bit number, 15,810 divisions are easy. For a 1024-bit number of the kind used in cryptography, on the order of 2^{512} divisions are needed; this falls in the “far longer than the age of the universe” row of the growth table.

This observation shows that the input size must be chosen carefully in a complexity calculation. For an algorithm that processes an array, the number of elements is a natural measure. For algorithms that work with a single number, the two measures give very different results. Complexity theory always takes the input size to be the number of bits needed to write the input. Running times that are polynomial in the value but exponential in the bit length are called **pseudo-polynomial** time. This is why methods that test the primality of large numbers quickly rest on ideas entirely different from trial division.

1.3.6 Cubic time: matrix multiplication

Matrix multiplication is a basic operation in many fields, from scientific computing to computer graphics and neural networks. For $n \times n$ matrices $A = [a_{ij}]$ and $B = [b_{ij}]$, each entry of the product $C = A \cdot B$ is defined by the formula

> $c_{ij} = \Sigma_{k=1..n}\, a_{ik} \cdot b_{kj} \quad (i, j = 1, 2, \ldots, n)$

That is, c_{ij} is the sum of the products of corresponding entries of row i of A and column j of B. In the example from the slides,

1	2	·	5	6	=	19	22
3	4		7	8		43	50

For example, $c_{11} = 1 \cdot 5 + 2 \cdot 7 = 19$ and $c_{21} = 3 \cdot 5 + 4 \cdot 7 = 43$. This definition turns directly into three nested loops:

```
int[][] multiply(int[][] A, int[][] B) {
  if (B.length == 0 || (A.length > 0 && A[0].length != B.length))
    throw new IllegalArgumentException("dimensions");
  int n = A.length, m = B[0].length, p = B.length;
  int[][] result = new int[n][m];
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) {
      int sum = 0;
      for (int k = 0; k < p; k++)
        sum += A[i][k] * B[k][j];
      result[i][j] = sum;
    }
  return result;
}
```

The code is not limited to square matrices. `n` is the number of rows of `A`; `m` is the number of columns of `B`; and `p` is the number of rows of `B`, which must equal the number of columns of `A`. The method begins with this check: if `B` is empty or the dimensions do not match, an `IllegalArgumentException` is thrown; only after that is `B[0]` accessed. The result has size $n \times m$. In Java the code indexes the matrix from 0; there is a shift relative to the indices starting from 1 in the formula, but the computation is the same.

The two outer loops visit each cell *(i, j)* of the result matrix once. For each cell, `sum` is reset, the innermost loop adds up `p` products, and the result is written to `result[i][j]`. The innermost line `sum += A[i][k] * B[k][j]` runs $n \cdot m \cdot p$ times in all. For square matrices $n = m = p$, so this number is n^3:

> $T(n) = n \cdot n \cdot n = n^3$ multiplications $\Rightarrow$ $T(n) = \Theta(n^3)$

The reasoning in brief: the result has n^2 entries, and each entry requires n multiplications. For $n = 100$ a million multiplications are performed, and for $n = 200$ eight million. Doubling n multiplies the time by eight; this is the signature of cubic growth. Because the result matrix holds n^2 cells, the space complexity is $\Theta(n^2)$.

This straightforward method is not the fastest way to multiply matrices. Strassen's algorithm, which works with the divide-and-conquer idea, does the same job with about $n^{2.81}$ multiplications. But the straightforward method is both the easiest to understand and still very widely used in practice on small and medium-sized matrices.

1.3.7 Code trace: multiplying 2 × 2 matrices

When `multiply` is called with the 2×2 matrices from the slides, $n = m = p = 2$ and the result is created as a 2×2 zero matrix. Table 1.6 gives the variable values after each multiplication.

i	j	k	A[i][k] · B[k][j]	sum	multiplications	Cell written
0	0	0	1 · 5 = 5	5	1	—
0	0	1	2 · 7 = 14	19	2	result[0][0] = 19
0	1	0	1 · 6 = 6	6	3	—
0	1	1	2 · 8 = 16	22	4	result[0][1] = 22
1	0	0	3 · 5 = 15	15	5	—
1	0	1	4 · 7 = 28	43	6	result[1][0] = 43
1	1	0	3 · 6 = 18	18	7	—
1	1	1	4 · 8 = 32	50	8	result[1][1] = 50

Table 1.6: Trace of `multiply`, *A = [[1, 2], [3, 4]], B = [[5, 6], [7, 8]]*.

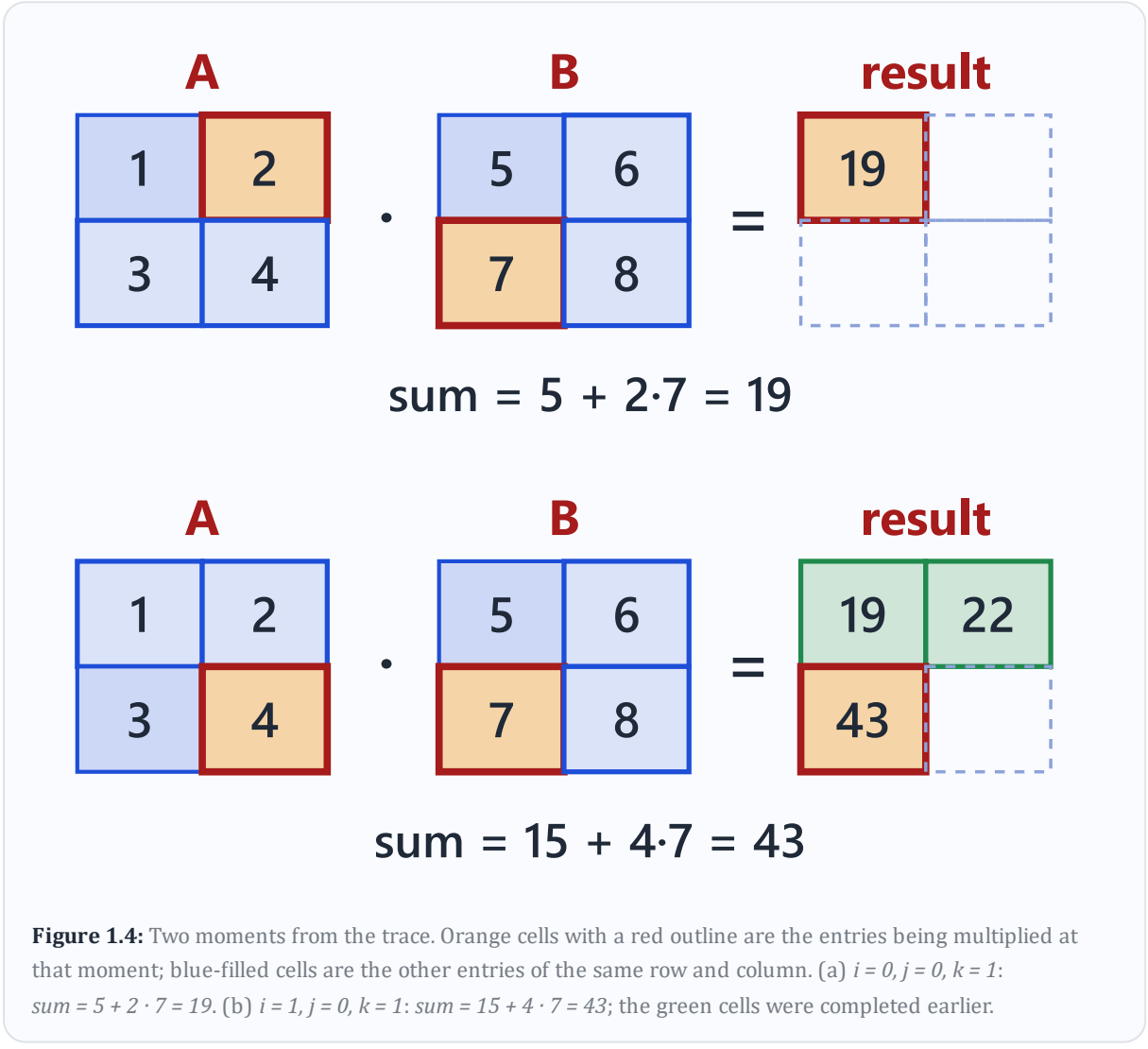


Figure 1.4: Two moments from the trace. Orange cells with a red outline are the entries being multiplied at that moment; blue-filled cells are the other entries of the same row and column. (a) *i = 0, j = 0, k = 1*: *sum = 5 + 2 · 7 = 19*. (b) *i = 1, j = 0, k = 1*: *sum = 15 + 4 · 7 = 43*; the green cells were completed earlier.

At the end of the trace `result = [[19, 22], [43, 50]]` is returned, and exactly $8 = 2^3$ multiplications have been performed. The pattern in the table also shows the general case: `i` and `j` choose each result cell once, and `k` takes *n* values for each choice. For *3 × 3* matrices the same count

gives $3^3 = 27$ multiplications.

Common mistake

Thinking that matrix multiplication takes n^2 time. The result matrix has n^2 entries, and each entry is written once; this gives the impression that the work is n^2. But the computation of each entry is a separate loop of length n. The number of nested loops determines how many times the innermost operation runs: here three loops, n^3 operations.

1.3.8 Factorial time: all permutations

Definition 1.7

Each ordering of n items is called a **permutation**. The number of permutations of n distinct items is $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$ (n factorial).

Why the number of permutations is $n!$ becomes clear when the ordering is built position by position. Any of the n items can go in the first position. For the second position $n - 1$ choices remain, for the third $n - 2$, and so on. The last position is filled by the one remaining item. Multiplying the choices gives $n!$. For three items there are $3 \cdot 2 \cdot 1 = 6$ permutations: 123, 132, 213, 231, 321, 312.

For some problems the most direct solution is to try every permutation one by one. For example, the order in which a traveler visits n cities with the shortest total distance can be found by trying all orders. The `permute` method in the slides prints all permutations of an array.

```
void permute(int[] a, int k) {
  if (k == a.length) {              // a permutation is ready
    System.out.println(Arrays.toString(a));
    return;
  }
  for (int i = k; i < a.length; i++) {
    swap(a, k, i);                  // put a[i] at position k
    permute(a, k + 1);              // arrange the rest
    swap(a, k, i);                  // restore
  }
}
```

The method is recursive, and the parameter `k` tells which position of the array is being filled. Positions 0 through `k - 1` are settled; the elements from `k` to the end are not yet arranged.

- **Base case:** when `k == a.length`, all positions are filled. The array is then a permutation and is printed.

- **Recursive step:** the elements at positions `k`, `k + 1`, ..., up to the last are brought to position `k` in turn. For this, `swap(a, k, i)` moves element `i` to position `k`. Then `permute(a, k + 1)` fills the remaining positions in the same way.
- **Undo:** after the recursive call returns, the second `swap` restores the array. So the next round of the loop starts with the original order of the elements after position `k`.

The helper method `swap` used in the code exchanges the values at two positions of the array: `int t = a[x]; a[x] = a[y]; a[y] = t;`. `Arrays.toString` converts the array to text in the form `[1, 2, 3]` and comes from the `java.util.Arrays` class.

Example 1.4

Tracing the call `permute({1, 2, 3}, 0)`:

1. *k = 0, i = 0*: `swap(0, 0)` leaves the array unchanged: [1, 2, 3]. The call descends to *k = 1*.
   - *k = 1, i = 1*: the array stays [1, 2, 3]; at *k = 2* there is only one choice, and at *k = 3* **[1, 2, 3]** is printed.
   - *k = 1, i = 2*: `swap(1, 2)` → [1, 3, 2]; **[1, 3, 2]** is printed. The undo swap returns the array to [1, 2, 3].
2. *k = 0, i = 1*: `swap(0, 1)` → [2, 1, 3]. The same process prints **[2, 1, 3]** and **[2, 3, 1]**. The undo swap restores the array to [1, 2, 3].
3. *k = 0, i = 2*: `swap(0, 2)` → [3, 2, 1]. This time **[3, 2, 1]** and **[3, 1, 2]** are printed. The last swap again returns the array to [1, 2, 3].
4. **Result:** 6 lines are printed, and when the method returns the array is in its original state. The method is called 16 times in all: 1 (*k = 0*) + 3 (*k = 1*) + 6 (*k = 2*) + 6 (*k = 3*).

Note that the order of output is 123, 132, 213, 231, 321, 312. This is not lexicographic order: 312 and 321 are in reverse positions. The swap method generates each permutation once, but it does not preserve lexicographic order.

How many operations? The number of lines printed is exactly *n!*. On each line `Arrays.toString` converts *n* elements to text. The total work is therefore $\Theta(n \cdot n!)$. The calls at the intermediate nodes do not change this: the number of calls is a constant multiple of *n!* (about $e \cdot n! \approx 2.72 \cdot n!$). This class is briefly called the "*O(n!)* class." Table 1.7 shows how fast the factorial grows.

n	n!	Time at 10^9 permutations per second
3	6	instant
5	120	instant
10	3,628,800	about 3.6 ms
15	$1.3 \cdot 10^{12}$	about 22 minutes

n	n!	Time at 10^9 permutations per second
20	$2.4 \cdot 10^{18}$	about 77 years

Table 1.7: The number of permutations *n!*.

The permutations of ten items are generated in the blink of an eye; those of twenty items take a human lifetime. Trying all permutations is therefore a method usable only on very small inputs. For larger inputs, techniques such as backtracking and branch and bound prune most of the search; or one looks for an entirely different algorithm that exploits the structure of the problem.

Common mistake

Forgetting the second `swap` call. Without the undo, the array continues in each round from the scrambled state the previous round left behind. For `{1, 2, 3}` the output becomes [1, 2, 3], [1, 3, 2], [3, 1, 2], [3, 2, 1], [1, 2, 3], [1, 3, 2]: 123 and 132 are printed twice, and 213 and 231 are never produced. Undoing the change a call made once the call returns is also the basic rule of backtracking algorithms.

1.3.9 Exponential time: naive recursive Fibonacci

The Fibonacci sequence begins with 0 and 1; each following term is the sum of the previous two: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … The definition is written as follows:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \quad (n \geq 2)$$

This definition is recursive and can be turned directly into a Java method. The first method in the slides, `fib`, does this; the second method, `fibMemo`, is the subject of the next subsection.

```
long fib(int n) {           // naive recursion
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

static final int N = 92;            // largest n for long
long[] memo = new long[N + 1];    // 0: not computed
long fibMemo(int n) {         // memoization
  if (n <= 1) return n;
  if (memo[n] == 0)
    memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
  return memo[n];
}
```

The `fib` method is an exact copy of the definition. `n ≤ 1` is the base case and returns the value of `n` : *F(0) = 0*, *F(1) = 1*. Otherwise the method calls itself twice and adds the results. The return type is `long` because Fibonacci numbers grow quickly: *F(47)* no longer fits in an `int` .

The code is short and correct, but very slow. The reason is clearly visible in the call tree in Figure 1.5. `fib(5)` calls `fib(4)` and `fib(3)` . `fib(4)` in turn calls `fib(3)` again. So `fib(3)` is computed twice, separately, from scratch. The repetition grows further down: `fib(2)` is called three times and `fib(1)` five times.

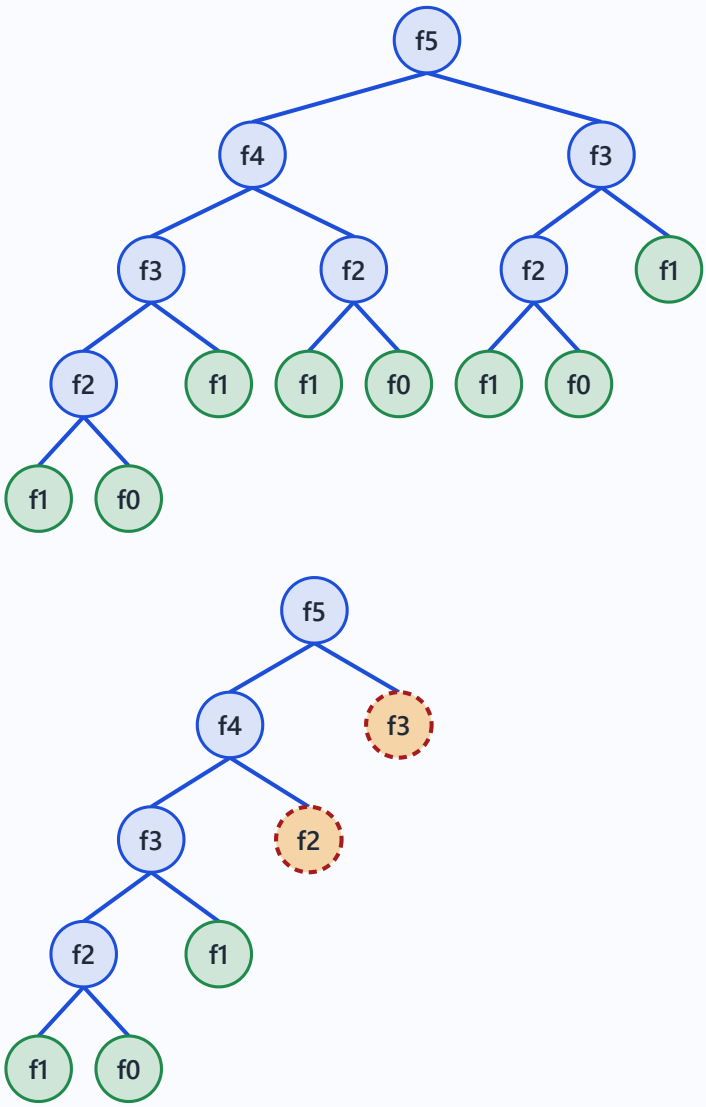


Figure 1.5: (a) The call tree of `fib(5)` : 15 calls. Green nodes are base cases (`f1` , `f0`). The left and right subtrees solve the same subproblems from scratch. (b) The call tree of `fibMemo(5)` : 9 calls. Orange dashed nodes (`f2` , `f3`) take the result ready-made from the memo array; their subtrees are never expanded.

How many calls are made? Let *C(n)* be the total number of calls made by `fib(n)` . In the base cases there is only the call itself: *C(0) = C(1) = 1*. In the other cases the counts of the two subcalls are added to the call itself:

$$C(n) = 1 + C(n-1) + C(n-2)$$

This recurrence is very similar to the Fibonacci recurrence. Computing the values in order gives $C(0..6) = 1, 1, 3, 5, 9, 15, 25$. By induction one can show that $C(n) = 2F(n+1) - 1$. For $n = 5$ that is $2 \cdot 8 - 1 = 15$, and for $n = 30$ it is $2 \cdot 1{,}346{,}269 - 1 = 2{,}692{,}537$ calls. `fib(40)` makes 331,160,281 calls; that means more than three hundred million method calls to find the 40th Fibonacci number.

Growth rate. Each call spawns at most two calls, and the depth of the tree is at most n. A binary tree of depth n can have at most 2^n nodes. Hence $T(n) = O(2^n)$. The true growth is a little slower, because the right branches are shorter than the left ones. $C(n)$ grows at the same rate as $F(n)$, and $F(n)$ is multiplied at each step by about the golden ratio $\varphi = (1 + \sqrt{5}) / 2 \approx 1.618$. The exact class is $\Theta(\varphi^n) = \Theta(1.618^n)$. Although the base is 1.618 rather than 2, the growth is still exponential: when n increases by one, the time grows by about 62 percent, and when n increases by ten, it grows by more than a hundred and twenty times.

1.3.10 Memoization: solving each subproblem once

The slowness of the naive Fibonacci code comes from solving the same subproblem again and again. When `fib(30)` is computed, $F(2)$ is recomputed hundreds of thousands of times. Yet $F(2)$ is the same value every time. If it were computed once and written down somewhere, every later request could read it from there.

Definition 1.8

Memoization is the technique of storing the results of subproblems computed in a recursive algorithm in a table and, when the same subproblem is requested again, returning the stored value without recomputing it.

The `fibMemo` method in the slides uses a `long` array called `memo`. Cell k of the array holds $F(k)$. In Java all cells of a new `long` array start at 0. Here the value 0 is used to mean “not yet computed.” This choice is safe, because in the range without overflow ($n \le 92$) $F(k)$ is never 0 for $k \ge 1$; $F(0) = 0$ is handled separately in the base case. The constant `N` is the largest n to be computed, and the array is allocated with $N + 1$ cells. $N = 92$ is chosen because $F(92)$ is the largest Fibonacci number that fits in a `long` ; $F(93)$ overflows. For larger n, `BigInteger` should be used.

The method first tests the base case. Then, if `memo[n] == 0`, that is, if $F(n)$ has not been computed before, it computes it with two recursive calls and writes it into the array. In every case it returns the result from the array. In this way each $F(k)$ is computed only once.

Example 1.5

Tracing the call `fibMemo(5)` when the `memo` array is all zeros:

1. `fibMemo(5)` → `memo[5] == 0`; first `fibMemo(4)` is called. In the same way the chain 4 → 3 → 2 descends.
2. `fibMemo(2)` makes the calls `fibMemo(1) = 1` and `fibMemo(0) = 0`; `memo[2] = 1` is written.
3. Back in `fibMemo(3)`; the second call is `fibMemo(1) = 1`. `memo[3] = 1 + 1 = 2`.
4. Back in `fibMemo(4)`; the second call is `fibMemo(2)`. `memo[2]` is now 1; the call returns 1 without computing anything. `memo[4] = 2 + 1 = 3`.
5. Back in `fibMemo(5)`; the second call `fibMemo(3)` returns 2 from memory. `memo[5] = 3 + 2 = 5`.
6. **Result:** *F(5) = 5*. A total of 9 calls were made: `f5, f4, f3, f2, f1, f0, f1, f2, f3`. The last two are the calls that returned from memory.

How many calls? For $n \geq 1$, `fibMemo(n)` makes exactly two subcalls for each k from 2 to n. The first computes $F(k - 1)$. The second asks for $F(k - 2)$; this call either returns from memory or is a base case. Adding the first call gives $2(n - 1) + 1 = 2n - 1$ calls in all. For $n = 5$ that is 9, for $n = 30$ it is 59. Compared with the 2,692,537 calls of the naive method, the difference is a factor of about forty-five thousand. The running time drops to $O(n)$.

This gain has a price: the `memo` array holds $n + 1$ cells. Moreover, since the recursion descends to depth n, the call stack also occupies $O(n)$ space. Memoization spends memory to save time. This trade-off between time and memory is common in algorithm design. The same idea written bottom-up with a loop is the basis of dynamic programming.

Common mistake

Translating a recursive definition directly into code without checking whether the subproblems repeat. The naive `fib` gives the correct result, but in $O(2^n)$ time. If the call tree of a recursion shows several calls with the same argument, storing the results often brings the time down from exponential to polynomial.

1.4 Euclid's algorithm

Euclid's algorithm is one of the oldest algorithms known. The Greek mathematician Euclid describes the method in his *Elements*, written around 300 BC. The algorithm has been in use for about two thousand three hundred years and still does useful work today in places ranging from cryptography to reducing fractions. Because it is short, has an easily proved correctness, and is surprisingly fast, it is also a very good vehicle for learning analysis.

1.4.1 The GCD and Euclid's idea

Definition 1.9

The largest positive integer that divides two integers without remainder is called their **greatest common divisor** (GCD) and is written *gcd(a, b)*. For example *gcd(12, 18) = 6* and *gcd(17, 5) = 1*.

The most direct way to find the GCD is to start at *min(a, b)* and try every number downward. The first number that divides both is the GCD. In the worst case this method makes *min(a, b)* trials: *O(min(a, b))*. Another way is to factor both numbers into primes and multiply the common factors; but factoring large numbers is itself a hard problem.

Euclid's idea is to divide the larger number by the smaller and continue with the remainder. The basic identity is

> gcd(a, b) = gcd(b, a mod b), gcd(a, 0) = a

Here *a mod b* is the remainder of dividing *a* by *b*. The algorithm is described in three steps:

1. The larger number is divided by the smaller and the remainder is found.
2. The operation is repeated with the pair (divisor, remainder) until the remainder is zero.
3. When the remainder is zero, the last divisor is the GCD.

The identity turns directly into a recursive method:

```
int gcd(int a, int b) {
  if (b == 0)
    return a;
  return gcd(b, a % b);
}
```

`b == 0` is the base case; *gcd(a, 0) = a*, because every number divides 0 and the largest number dividing *a* is *a* itself. Otherwise the method calls itself with `(b, a % b)`. In Java the `%` operator gives the remainder.

The method also works when *a < b*. For example, in the call `gcd(650, 1599)`, *650 % 1599 = 650*, and the first call becomes `gcd(1599, 650)`. The numbers swap themselves; the cost is just one extra division.

1.4.2 Why is the algorithm correct?

Theorem 1.2 (Euclid)

Let $a \geq 0$ and $b > 0$ be integers. The common divisors of a and b are the same as the common divisors of b and $a \bmod b$. Therefore $\gcd(a, b) = \gcd(b, a \bmod b)$.

Proof. *Proof.* Divide a by b: $a = q \cdot b + r$, where q is the quotient, $r = a \bmod b$ is the remainder, and $0 \leq r < b$.

(1) Let d divide both a and b. Then d also divides $q \cdot b$ and hence the difference $r = a - q \cdot b$. So every common divisor of a and b is also a common divisor of b and r.

(2) Conversely, let d divide both b and r. Then it also divides the sum $a = q \cdot b + r$. So every common divisor of b and r is also a common divisor of a and b.

Since the two sets are equal, their largest elements are equal too. ■

The theorem says that every recursive call preserves the GCD. Finiteness is also easy to show. In each call the second argument goes from b to $a \bmod b$, and $a \bmod b < b$. Since nonnegative integers cannot decrease forever, the second argument eventually becomes 0 and the base case is reached. The a returned in the base case is the correct GCD. So the algorithm both stops and gives the correct result.

1.4.3 Code trace: gcd(1599, 650)

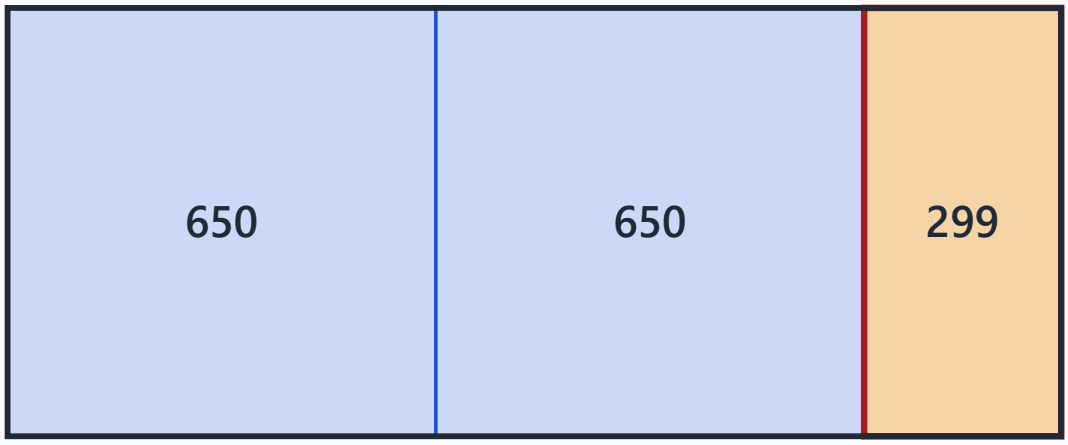
The trace example in the slides is `gcd(1599, 650)`. Table 1.8 gives the division in each call.

Call	a	b	Division	a % b
1	1599	650	1599 = 2 · 650 + 299	299
2	650	299	650 = 2 · 299 + 52	52
3	299	52	299 = 5 · 52 + 39	39
4	52	39	52 = 1 · 39 + 13	13
5	39	13	39 = 3 · 13 + 0	0
6	13	0	`b == 0 : return 13`	—

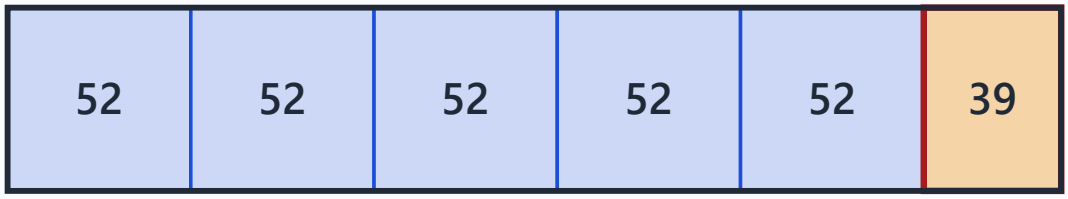
Table 1.8: Trace of `gcd(1599, 650)`.

After five divisions the remainder is 0, and the sixth call is the base case. The value 13 returns unchanged through all the calls; each call merely passes on the result of its subcall. The result is $\gcd(1599, 650) = 13$. Factoring confirms this: $1599 = 3 \cdot 13 \cdot 41$ and $650 = 2 \cdot 5^2 \cdot 13$; the only common factor is 13.

The figures in the slides give a geometric interpretation of the algorithm. From a rectangle of size $a \times b$, squares with side b are cut. As many squares as fit (q of them) are cut, leaving a strip of size $b \times r$. The same operation is applied to this strip. When the strip is finally divided exactly into squares, the side of the last square is the GCD. Figure 1.6 shows three steps of this process.



1599 × 650 rectangle



299 × 52 rectangle

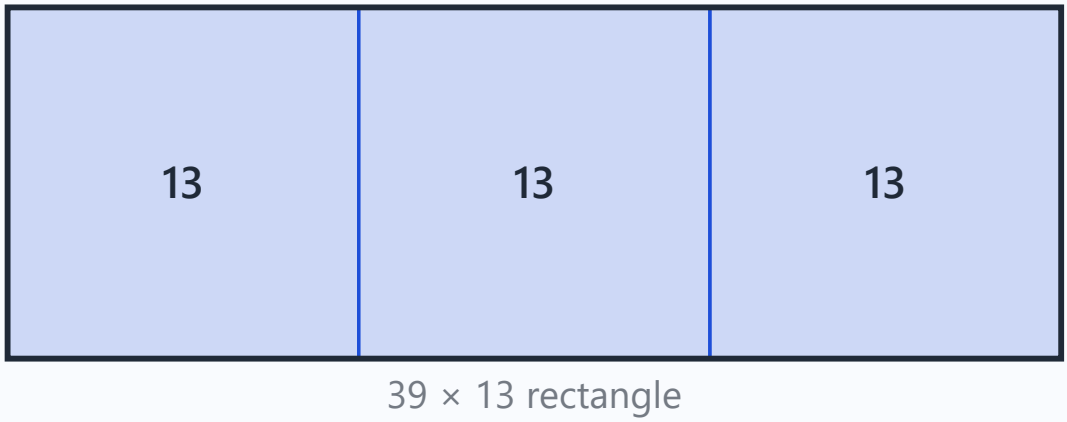


Figure 1.6: The square-cutting interpretation of Euclid's algorithm. (a) From the *1599 × 650* rectangle two *650 × 650* squares are cut; the orange strip *650 × 299* remains. (b) From the *299 × 52* rectangle five squares are cut; the remaining strip is *52 × 39*. (c) The *39 × 13* rectangle divides exactly into three squares; nothing is left over. The squares with side 13 also tile the original rectangle without gaps.

The last observation also explains the meaning of the GCD. The squares with side 13 tile *39 × 13*. These squares also tile *52 × 39*, because both 52 and 39 are multiples of 13. The same reasoning works step by step backward, and in the end the *1599 × 650* rectangle too is tiled by *123 × 50* squares of size *13 × 13*. This is the largest square that can tile this rectangle.

The recursive calls accumulate on the call stack. When the base case is reached, the stack holds six frames: from `gcd(1599, 650)` to `gcd(13, 0)`. Since this depth is proportional to the number of divisions, the space complexity of the recursive version is of the same order as the number of divisions.

1.4.4 Why is the algorithm fast?

`gcd(1599, 650)` finished in just five divisions. The trial method would have tried 638 numbers, from 650 down to 13. This difference is no accident. In Euclid's algorithm the numbers shrink very fast, and the following observation guarantees it.

Theorem 1.3 (Halving)

If $a \geq b > 0$, then $a \bmod b < a / 2$.

Proof. *Proof.* There are two cases. If $b \leq a / 2$, then since the remainder is smaller than the divisor, $a \bmod b < b \leq a / 2$. If $b > a / 2$, then b fits into a only once; the quotient is 1 and the remainder is $a - b$. Since $b > a / 2$, we get $a - b < a / 2$. In both cases the remainder is less than half of a. ■

Consider two consecutive steps of the algorithm: $(a, b) \to (b, r) \to (r, b \bmod r)$. After two steps, the place of the first number is taken by $r = a \bmod b$, and by the theorem $r < a / 2$. So every two steps the larger number at least halves. A number a can be halved at most $\log_2 a$ times before it drops below 1. The number of divisions is therefore at most about $2 \log_2 \min(a, b)$. After the first step the larger number of the pair is already at most $\min(a, b)$; this is why the smaller number appears in the bound. The running time is

> $T(a, b) = O(\log \min(a, b))$

Compared with the $O(\min(a, b))$ time of the trial method, the difference is large. For two numbers of size one billion the trial method may make a billion trials, while Euclid's algorithm makes at most about sixty divisions.

It is useful to return once more to the matter of input size. If the numbers have bit length ℓ, then $\log_2 \min(a, b) \le \ell$. So Euclid's algorithm makes a *linear* number of divisions in the bit length. The trial division in primality testing was exponential in the bit length. This is why Euclid's algorithm runs instantly even on numbers with hundreds of digits and is used freely in cryptography.

1.4.5 The worst case: consecutive Fibonacci numbers

The halving bound is an upper bound; on most inputs the algorithm finishes earlier. But which inputs make the algorithm run longest? The numbers shrink most slowly when the quotient is 1 in every division. Then the remainder is the difference of the two numbers: $r = a - b$. Looking backward in this case, each number is the sum of the two numbers after it. This is exactly the rule of the Fibonacci sequence. The worst input is two consecutive Fibonacci numbers.

Example 1.6

Tracing `gcd(55, 34)` (55 and 34 are consecutive Fibonacci numbers):

1. $55 = 1 \cdot 34 + 21$
2. $34 = 1 \cdot 21 + 13$
3. $21 = 1 \cdot 13 + 8$
4. $13 = 1 \cdot 8 + 5$
5. $8 = 1 \cdot 5 + 3$
6. $5 = 1 \cdot 3 + 2$
7. $3 = 1 \cdot 2 + 1$
8. $2 = 2 \cdot 1 + 0$

9. **Result:** 8 divisions, *gcd(55, 34) = 1*. The remainders follow the Fibonacci sequence backward: 21, 13, 8, 5, 3, 2, 1, 0. Except for the last division, every quotient is 1. Although the numbers are as small as 55 and 34, the number of divisions exceeds the 5 divisions of the much larger `gcd(1599, 650)`.

The precise form of this observation is the theorem proved by Gabriel Lamé in 1844.

Theorem 1.4 (Lamé)

For $a > b \geq 1$, if the call `gcd(a, b)` makes k divisions, then $b \geq F(k + 1)$. Accordingly, the number of divisions does not exceed five times the number of decimal digits of b.

Proof. *Proof sketch.* Let the remainders be $r_0 = a$, $r_1 = b$, r_2, ..., r_k, $r_{k+1} = 0$; the i-th division is $r_{i-1} = q_i \cdot r_i + r_{i+1}$. The last nonzero remainder satisfies $r_k \geq 1 = F(2)$. In the last division the remainder is 0 and $r_{k-1} > r_k$, so $q_k \geq 2$ and $r_{k-1} \geq 2 = F(3)$. In every other division $q_i \geq 1$, so $r_{i-1} \geq r_i + r_{i+1}$. Applied backward, this inequality shows that every remainder is at least a Fibonacci number, and we obtain $r_1 = b \geq F(k + 1)$. Because Fibonacci numbers grow at a rate of about $\varphi \approx 1.618$, k stays on the order of $\log_\varphi b$. ■

The theorem is tight in the example `gcd(55, 34)`: 8 divisions are made and $b = 34 = F(9)$. That is, 34 is the smallest b that gives 8 divisions. The next pair `gcd(89, 55)` makes nine divisions. Among all pairs of numbers below 1000, the most divisions (14) are made by `gcd(987, 610)`; 987 and 610 are also consecutive Fibonacci numbers. So even the worst case of Euclid's algorithm is logarithmic: $\Theta(\log \min(a, b))$.

1.4.6 Euclid with a loop

The recursive `gcd` calls itself as its last operation and passes the return value on unchanged. Such a recursion can be converted directly into a loop: in each round the pair *(a, b)* is replaced by *(b, a mod b)*.

```
int gcd(int a, int b) {
  while (b != 0) {
    int r = a % b;
    a = b;
    b = r;
  }
  return a;
}
```

The loop runs until `b` is zero. In each round the remainder `r` is computed first, then `a` is assigned the old `b` and `b` the remainder. The order of the assignments matters: if `a = b` were not done first, the old value of `b` would be lost. When the loop ends, `a` is the last nonzero remainder, that is, the

GCD. Table 1.9 gives the loop rounds of the call `gcd(1071, 462)`.

Round	a (at start)	b (at start)	r = a % b	(a, b) at end of round
1	1071	462	147	(462, 147)
2	462	147	21	(147, 21)
3	147	21	0	(21, 0)
—	21	0	—	`b == 0`: `return 21`

Table 1.9: Trace of the loop version of `gcd(1071, 462)`.

The divisions are $1071 = 2 \cdot 462 + 147$, $462 = 3 \cdot 147 + 21$, and $147 = 7 \cdot 21 + 0$. The loop version makes the same number of divisions as the recursive version and has the same running time: $O(\log \min(a, b))$. The difference is in memory. The recursive version opens a stack frame for every call; the loop version uses only three variables and works with $O(1)$ extra memory. Because the Java compiler does not turn tail recursion into a loop by itself, this transformation is done by hand.

1.5 Common mistakes

The mistakes discussed in various places in these notes are collected here. All of them come from the same source: a detail of the code or of the analysis is set up without thinking about all inputs.

- **Measuring time in seconds.** Seconds depend on the machine, the language, and the load at that moment. Analysis uses the operation count as a function of n and how fast it grows. A measurement tells us something about the algorithm only when it is repeated at different values of n and the shape of the growth is seen.
- **Forgetting memoization.** Naive recursive Fibonacci is $O(2^n)$ because it solves the same subproblems over and over. Storing the results in an array brings the number of calls down to $2n - 1$ and the time to $O(n)$.
- **Setting the $\sqrt{n}$ bound too low.** If `i < (int) Math.sqrt(n)` is written, $\lfloor\sqrt{n}\rfloor$ itself is not tried; perfect squares such as 9, 25, 49, and numbers such as 15 and 35 whose small factor equals exactly $\lfloor\sqrt{n}\rfloor$ come out prime. The correct forms are `i <= (int) Math.sqrt(n)` or, as in the slides, `i < (int) Math.sqrt(n) + 1`.
- **Confusing the input size.** In primality testing, n is the value of the number. For a number of b bits, $\sqrt{n} = 2^{b/2}$; that is, trial division is exponential in the number of bits. For algorithms that take a single number as input, what the word "n" stands for must always be stated explicitly.
- **Thinking matrix multiplication is n^2.** The result has n^2 entries, but each entry needs n multiplications: n^3 multiplications in all. With nested loops, one counts how many times in total the innermost operation runs.
- **Not undoing the swap.** If the second `swap` is forgotten in permutation generation, the array gets corrupted; for `{1, 2, 3}` some permutations are printed twice and some are never produced.

- **Choosing the initial value independently of the input.** Starting with `max = 0` when looking for the largest element gives a wrong result on arrays whose elements are all negative. The initial value must be an element of the array itself.
- **Overlooking integer division.** In Java the quotient of two `int` values discards the fractional part. In computations such as an average, where a fractional result is expected, at least one operand must be a `double`.

1.6 Summary

- An **algorithm** is a finite sequence of precisely defined instructions for solving a problem. It takes input and produces output; its steps are **definite**, **effective** (doable), and **finite**; and it gives the **correct** result for every valid input. An algorithm is independent of the language; a program is its counterpart in a particular language.
- Algorithms fall into problem groups such as sorting, searching, graphs, and strings and compression. Divide and conquer, dynamic programming, greedy methods, backtracking, and branch and bound are design techniques. A **deterministic** algorithm always follows the same path on the same input; a **randomized** algorithm makes random choices in its steps.
- **Algorithm analysis** studies time and memory use as a function of the input size n. The running time $T(n)$ is measured not in seconds but in the number of basic operations. For the sum of an array, line-by-line counting gives $T(n) = 2n + 3$.
- For large n, constant coefficients and lower-order terms become irrelevant and only the growth rate remains: $2n + 3 = \Theta(n)$, $5n^2 + 100n \log_2 n + 7 = \Theta(n^2)$.
- The order of growth: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$. A faster computer only shrinks the constant; a better algorithm changes the growth rate.
- Examples: printing the first element is $O(1)$; finding the largest element takes $\Theta(n)$ time and $O(1)$ extra memory; primality testing with odd divisors up to $\sqrt{n}$ is $O(\sqrt{n})$; straightforward matrix multiplication is $\Theta(n^3)$; generating all permutations is $\Theta(n \cdot n!)$; naive recursive Fibonacci is $\Theta(1.618^n)$, that is, exponential.
- **Memoization** stores the results of subproblems already computed; for Fibonacci it reduces the number of calls from $2F(n + 1) - 1$ to $2n - 1$ and the time to $O(n)$. The price is $O(n)$ memory.
- **Euclid's algorithm** rests on the identities $\gcd(a, b) = \gcd(b, a \bmod b)$ and $\gcd(a, 0) = a$. Its correctness comes from the common divisors of the two pairs being the same. Because the larger number at least halves every two steps, the time is $O(\log \min(a, b))$. The worst-case input is consecutive Fibonacci numbers (Lamé). The loop version runs in the same time with $O(1)$ extra memory.

Code	Input size	Operation counted	Time	Extra memory
sum of an array	number of elements n	comparisons, additions	2n + 3 = Θ(n)	O(1)

Code	Input size	Operation counted	Time	Extra memory
`printFirst`	number of elements n	comparison, access	$O(1)$	$O(1)$
`findMax`	number of elements n	comparisons	$n = \Theta(n)$	$O(1)$
`isPrime`	value of the number n	divisions (`%`)	$\approx \sqrt{n} / 2 = O(\sqrt{n})$	$O(1)$
`multiply`	matrix size n	multiplications	$n^3 = \Theta(n^3)$	$\Theta(n^2)$ (result)
`permute`	number of elements n	lines printed	$n!$ lines, $\Theta(n \cdot n!)$	$O(n)$ (stack)
`fib`	n	calls	$2F(n + 1) - 1 = \Theta(1.618^n)$	$O(n)$ (stack)
`fibMemo`	n	calls	$2n - 1 = O(n)$	$O(n)$
`gcd`	a, b	divisions (`%`)	$O(\log \min(a, b))$	recursive $O(\log \min(a, b))$, loop $O(1)$

Table 1.10: The programs examined in these notes.

1.7 Key terms

These are the terms introduced in this chapter.

algorithm input output definiteness finiteness effectiveness correctness pseudocode deterministic algorithm randomized algorithm algorithm analysis time complexity space complexity input size running time worst case growth rate Big O notation Big Theta (Θ) constant time logarithmic linear linearithmic quadratic cubic exponential factorial primality test pseudo-polynomial swap recursion base case memoization Euclid's algorithm

1.8 Test Yourself

1. “Let *n* be a positive integer. Repeat until *n* is 1: if *n* is even, *n = n / 2*; if it is odd, *n = 3n + 1*.” In how many steps does this procedure stop for *n = 6*? Can it be said for certain that the procedure is an

algorithm?

Answer

For $n = 6$ the sequence is 6, 3, 10, 5, 16, 8, 4, 2, 1: it stops in 8 steps. The steps are definite and doable. But finiteness has not been proved for every n. This question is an unsolved problem in mathematics known as the Collatz conjecture. The procedure has been tried by computer up to very large numbers, and the sequence has always reached 1. But there is no proof that it stops for every positive integer. So it cannot be said for certain that the procedure is finite for every input, and hence that it is an algorithm.

2. In the code `for (int i = 0; i < n; i++) for (int j = 0; j < i; j++) count++;`, how many times does `count++` run? What is the value for $n = 10$, and what is the complexity class?

Answer

In round i the inner loop runs i times. The total is $0 + 1 + \ldots + (n - 1) = n(n - 1) / 2$. For $n = 10$ that is $10 \cdot 9 / 2 = 45$. Because the dominant term is $n^2 / 2$, the class is $\Theta(n^2)$.

3. In the call `isPrime(199)`, what is `limit`, which values of `i` are tried in the loop, how many divisions are made, and what is the result?

Answer

$\sqrt{199} \approx 14.1$, *(int)* $\sqrt{199} = 14$, `limit = 15`. The values tried are 3, 5, 7, 9, 11, 13; the remainders are 1, 4, 3, 1, 1, 4 respectively, and none is 0. After 6 divisions the loop ends and the method returns `true`: 199 is prime.

4. With `multiply`, an `A` of size 3×2 is multiplied by a `B` of size 2×3. What are `n`, `m`, `p`, what is the size of the result, and how many multiplications are made? What if the product were taken in the opposite order (`B · A`)?

Answer

`A · B`: `n = 3`, `m = 3`, `p = 2`; the result is 3×3; $3 \cdot 3 \cdot 2 = 18$ multiplications. `B · A`: `n = 2`, `m = 2`, `p = 3`; the result is 2×2; $2 \cdot 2 \cdot 3 = 12$ multiplications. In matrix multiplication the order changes both the result and the operation count.

5. In the call `permute({1, 2, 3, 4}, 0)`, on which line is the first permutation starting with 2

printed, and what is it? How many times is the method called in all?

Answer

As long as 1 stays in the first position, the $3! = 6$ permutations of the remaining three elements are printed. In the round $k = 0$, $i = 1$, `swap(0, 1)` makes the array [2, 1, 3, 4], and this is the first line printed: **line 7, [2, 1, 3, 4]**. The number of calls is the sum of the calls at each level: $1 + 4 + 12 + 24 + 24 = 65$.

6. `fib(7)` and `fibMemo(7)` (memo array initially zero) are called separately. How many calls does each make, and what does it return?

Answer

Both return $F(7) = 13$. `fib(7)` : $C(7) = 2F(8) - 1 = 2 \cdot 21 - 1 = 41$ calls. `fibMemo(7)` : $2 \cdot 7 - 1 = 13$ calls.

7. How many divisions does `gcd(377, 233)` make, and what is the result? How does this number compare with the bound given by Lamé's theorem?

Answer

377 and 233 are consecutive Fibonacci numbers ($F(14)$ and $F(13)$). The remainders are 144, 89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 0: **12 divisions**, result 1. By the theorem, k divisions require $b \geq F(k + 1)$; since $b = 233 = F(13)$, $k \leq 12$. The example meets the bound exactly. The digit bound gives $5 \cdot 3 = 15$.

8. What does the recursive call `gcd(210, 84)` return? How many calls are made, and how many frames deep does the call stack go at most?

Answer

$210 = 2 \cdot 84 + 42$, $84 = 2 \cdot 42 + 0$. The calls are `gcd(210, 84)` , `gcd(84, 42)` , `gcd(42, 0)` : 3 calls, 2 divisions, result 42. When the base case is reached the stack holds 3 frames.

9. A computer is made 100 times faster. By what factor does the largest input solvable in the same time grow for a $\Theta(n^2)$ algorithm? By how much does n grow for a $\Theta(2^n)$ algorithm?

Answer

For the quadratic algorithm, the condition $(kn)^2 = 100\ n^2$ gives $k = 10$: the input grows tenfold. For the exponential algorithm, the condition $2^{n+d} = 100 \cdot 2^n$ gives $d = \log_2 100 \approx 6.6$: n grows by only 6 (as an integer).

10. Two primes, one of 40 bits and one of 80 bits, are tested with `isPrime`. About how many divisions are made for each? At 10^9 divisions per second, how long does the second take?

    Answer

    The number of divisions is about $\sqrt{n}\,/\,2$. For the 40-bit number $n \approx 2^{40}$ and $\sqrt{n} \approx 2^{20}$; the number of divisions is about $2^{19} \approx 524{,}000$, and it finishes instantly. For the 80-bit number $\sqrt{n} \approx 2^{40}$; the number of divisions is about $2^{39} \approx 5.5 \cdot 10^{11}$, and it takes about 550 seconds, that is, more than nine minutes. When the number of bits doubles, the time grows about a millionfold: exponential growth in the bit length.

1.9 Further reading

- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3rd ed., 1997), Section 1.1: introduces the concept of an algorithm through Euclid's algorithm and discusses the properties of input, output, definiteness, finiteness, and effectiveness under the names used in these notes.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 1–3: give the place of algorithms in computing, the line-by-line analysis of insertion sort, and asymptotic notation with its formal definitions; Chapter 31 analyzes the running time of Euclid's algorithm with Fibonacci numbers.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 1.4: shows, with Java examples, how to measure running time by experiments and explain the measurements with a mathematical model.
- J. Kleinberg and É. Tardos, *Algorithm Design* (2005), Chapter 2: treats polynomial time as the criterion of efficiency and compares common running times through example algorithms.

1.10 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005.
3. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, 1997.
4. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

PART I · CHAPTER 2

Algorithm Complexity

To say which of two algorithms that solve the same problem is better, we need a yardstick. Time in seconds will not do, because it varies from machine to machine. Complexity analysis instead counts the basic operations an algorithm performs and looks at how that count grows as the input grows. These notes begin with the concepts of time and space complexity and distinguish the best, worst, and average cases. They then define the O, Ω, and Θ notations for growth rate and show how the constants are found. The following sections compute the step count of looping code with summation formulas and the running time of recursive code with recurrence relations. The recurrences are solved by unrolling, by recursion trees, and by the Master Theorem. All these tools are applied to six short Java functions (f1–f6), each representing a different growth class. The notes close with a brief look at amortized analysis.

LEARNING OUTCOMES

Concepts

Define time and space complexity; tell best-, worst-, and average-case analysis apart; count the steps of a piece of code in the RAM model.

Notation

Write the definitions of Big O, Ω, and Θ; prove a bound for a function by finding the constants c and n_0; order the common growth classes.

Loop analysis

Compute the iteration counts of single, nested, and multiplicative loops with summation formulas; show how the bound of the inner loop changes the result (f1, f2, f6).

Recursion

Set up a recurrence for recursive code; solve it by unrolling, with a recursion tree, and with the Master Theorem; recognize the recurrences to which the theorem does not apply (f3, f4, f5).

Amortized analysis

Find the cost per operation from the total cost of a sequence of operations and distinguish it from the average case.

2.1 The concept of complexity

That an algorithm works correctly is the first requirement, but not the only one. A correct algorithm may finish in a second on an input of a thousand elements and take days on an input of a million. When choosing an algorithm one therefore also asks, “how many resources does it use?” This section explains how that question is measured. What is measured is not seconds but the number of basic operations; and what matters is how this number grows with the input.

2.1.1 What is algorithm complexity?

Complexity theory studies the resources an algorithm uses while it runs. Two resources stand out: time and memory. Both depend on the size of the input. Sorting five numbers takes far less time than sorting five million. Resource use is therefore written not as a single number but as a function of the input size.

The input size is usually denoted n. For an algorithm that processes an array, n is the number of elements in the array. For a string algorithm it is the number of characters, and for a graph algorithm the number of vertices and edges. The time requirement is denoted $T(n)$ and the memory requirement $S(n)$. For example, $T(n) = 3n + 3$ says that the algorithm performs $3n + 3$ basic operations on an input of size n.

Definition 2.1

Time complexity is the number of basic operations an algorithm performs on an input of size n, denoted $T(n)$. Basic operations are those that finish in constant time, such as assignment, comparison, arithmetic, and array access.

Space complexity is the amount of memory the algorithm uses in addition to the input, denoted $S(n)$. Local variables, auxiliary arrays, and the space that recursive calls occupy on the call stack all count toward this memory.

The purpose of analysis is not to measure seconds. The purpose is to see *how* the resource requirement grows as the input grows. The basic question is this: if the input size doubles, what happens to the running time? Does it stay the same, double, quadruple, or get squared? Figure 2.1 shows the answer to this question for several growth classes.

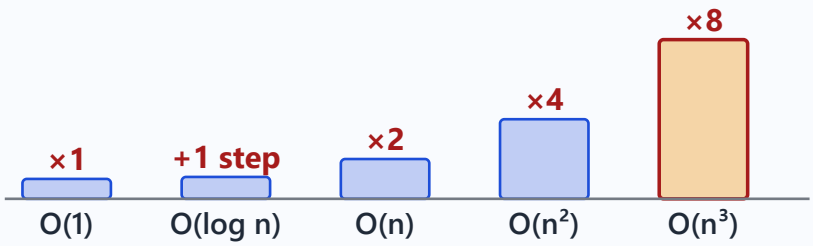


Figure 2.1: How the step count changes when the input size doubles. For a constant-time algorithm nothing changes (×1). A logarithmic algorithm takes just one more step. The running time of a linear algorithm doubles, that of a quadratic algorithm quadruples, and that of a cubic algorithm grows eightfold.

The factors in the figure come from simple arithmetic. If $T(n) = n^2$, then $T(2n) = 4n^2 = 4 \cdot T(n)$. If $T(n) = n^3$, then $T(2n) = 8n^3$. For the logarithm, $log_2(2n) = log_2\ n + 1$, so the time grows by just one step. This one question alone makes the difference between algorithms plain. When the input grows a thousandfold, a cubic algorithm slows down a billionfold; a logarithmic algorithm works only about ten steps more.

Common mistake

Stating complexity as a single number, for example “this algorithm takes 12 steps.” Twelve steps is true only for a particular input. Complexity is always a function of n: “this algorithm takes $3n + 3$ steps, which is 12 steps for $n = 3$.”

2.1.2 Why timing is not enough

The first way that comes to mind to learn the speed of an algorithm is to run it and measure the time. In Java this is done by reading the system clock before and after the run and printing the difference:

```
long start = System.currentTimeMillis();
linearSearch(a, 10001);  // 10^8 elements
long end = System.currentTimeMillis();
System.out.println("Elapsed: "
    + (end - start) + " ms");
```

The code measures how many milliseconds a linear search takes on an array of 10^8 elements. `System.currentTimeMillis()` returns the current time in milliseconds. The difference between the two readings is the time the search used. This method is called **experimental measurement**. It gives a real number, but that number has three serious weaknesses.

- **The result depends on the machine.** The same code may take 40 ms on a fast processor and 200 ms on an old laptop. In Java the situation is even more tangled. The **JIT compiler** (just-in-time compiler) translates frequently run code into machine code while the program runs; so the same

measurement comes out slow on the first try and fast on later ones. The **cache**, the operating system's load at that moment, and the time when the garbage collector kicks in also change the result.

- **A single measurement does not predict the future.** A search that took 40 ms on 10^8 elements, how long does it take on 10^9? The measurement does not say. If the algorithm grows linearly, the time grows tenfold to 400 ms; if it grows quadratically, it grows a hundredfold and reaches 4 seconds. The answer depends on the growth pattern of the algorithm, and a single measurement does not reveal that pattern.
- **The code must already be written.** To compare two designs, both have to be coded, tested, and measured under the same conditions. At the design stage, when there is no code yet, no measurement can be made.

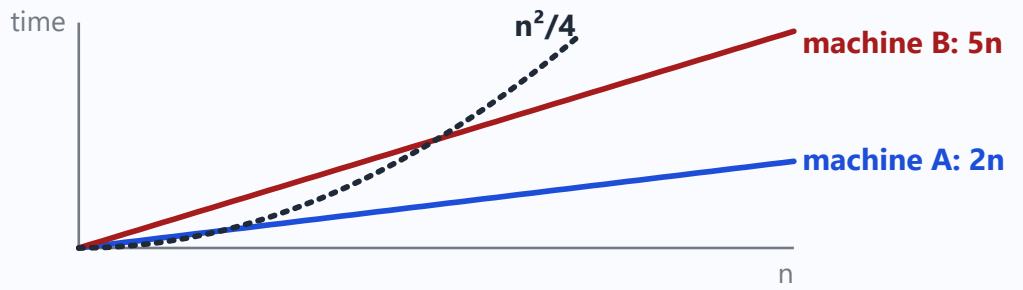


Figure 2.2: The same linear algorithm grows with different constants on two machines: *2n* on machine A and *5n* on machine B. A quadratic algorithm ($n^2/4$) is faster than both for small *n*, but beyond some point overtakes both.

Figure 2.2 makes the second weakness concrete. Machine B is 2.5 times slower than A; but both lines are correct and grow in the same way. The dashed curve belongs to a different algorithm and grows quadratically. On small inputs this algorithm looks faster than the linear algorithm on both machines. The equality $n^2/4 = 5n$ holds at $n = 20$; beyond this point the quadratic algorithm falls behind even the linear algorithm on the slow machine, and the gap keeps widening. A measurement made on small inputs would have led us to pick the wrong algorithm here.

The remedy for these weaknesses is **asymptotic analysis**. Asymptotic analysis gives the step count as a function of *n* and looks only at what happens as *n* grows. It is independent of the machine; it can be done without writing code, from the description of the algorithm alone. Experimental measurement is not worthless: if two algorithms are in the same growth class, only measurement can tell which one is actually faster. The two methods complement each other. First analysis picks the right growth class, then measurement compares the constant factors.

	Experimental measurement	Asymptotic analysis
What it gives	actual time (ms)	growth class of the step count
Depends on the machine	yes	no
Needs code	yes	no, a description suffices

	Experimental measurement	Asymptotic analysis
Predicts for large inputs	no	yes
Sees constant factors	yes	no

Table 2.1: Experimental measurement compared with asymptotic analysis.

2.1.3 Counting steps and the RAM model

To count in a machine-independent way we imagine a simple, idealized machine. In this machine every basic operation takes exactly one unit of time. Every cell in memory is reached in the same time, whatever its address. Operations run one at a time, in sequence. This abstract machine is called the **RAM model** (random-access machine). Real computers do not work this way: data in the cache is read many times faster than data in main memory. But the model is accurate enough for finding the growth class, and it makes the calculation much easier.

Let us apply the model to linear search. Linear search looks at the elements of the array one by one from the beginning to the end. When the sought value (`target`) is found, it returns its index; if the array ends, it returns −1:

```
for (int i = 0; i < a.length; i++) {
  if (a[i] == target)
    return i;
}
return -1;
```

For the count we find how many times each operation runs. Consider the worst case: the target is not in the array, so the loop runs to the end. Let the length of the array be n.

- The assignment `i = 0` is performed once when the loop starts.
- The condition `i < a.length` is checked at the start of each round. For each value of i from 0 to $n - 1$ the condition is true; that makes n checks. When $i = n$ the condition is checked once more and turns out false. In all, $n + 1$ checks are made.
- The comparison `a[i] == target` is made once per round: n times.
- `i++` is performed once at the end of each round: n times.
- `return -1` runs once at the very end.

Operation	How many times in the worst case?
`i = 0`	1
`i < a.length`	n + 1
`a[i] == target`	n
`i++`	n
`return -1`	1

Operation	How many times in the worst case?
Total *T(n)*	3n + 3

Table 2.2: Step count of `linearSearch` in the worst case.

The total is $T(n) = 1 + (n + 1) + n + n + 1 = 3n + 3$. This number carries two kinds of information. The coefficient 3 and the constant +3 depend on what is counted as a step and on the machine. Had the array access and the comparison been counted as two separate operations, the coefficient would have been 4; had the compiler merged the condition check with the increment, it would have been 2. What does not change is that the step count grows in direct proportion to n. When the array doubles in size, the step count roughly doubles too. Asymptotic analysis keeps this unchanging part and discards the rest: $T(n) = \Theta(n)$. Here $\Theta(n)$ is read as "grows at the same rate as n"; its precise definition is given together with the asymptotic notation.

Common mistake

Thinking the loop condition is checked n times. The condition is checked $n + 1$ times, counting the final check that ends the loop. This difference does not change the growth class, but it leads to a wrong result in questions that ask for the exact step count.

2.1.4 Code trace: linearSearch

Let us verify the count step by step on a small example. The array is `a = [4, 7, 2]` and the target is 5. Since 5 is not in the array, the worst case occurs; the expected step count is $3 \cdot 3 + 3 = 12$.

Step	Operation	`i`	Result	Total steps
1	`i = 0`, then `0 < 3`	0	condition true	2
2	`a[0] == 5`	0	4 ≠ 5	3
3	`i++`, then `1 < 3`	1	condition true	5
4	`a[1] == 5`	1	7 ≠ 5	6
5	`i++`, then `2 < 3`	2	condition true	8
6	`a[2] == 5`	2	2 ≠ 5	9
7	`i++`, then `3 < 3`	3	condition false, loop ends	11
8	`return -1`	3	returns −1	12

Table 2.3: Trace of `linearSearch([4, 7, 2], 5)`. The last column is the total step count so far.

The condition was checked four times ($n + 1 = 4$), and the comparison and the increment three times each. The total of 12 steps is the value the formula $3n + 3$ gives for $n = 3$. A constant number of operations, three, is spent on each element: one condition check, one comparison, and one increment. As the number of elements grows, this triple is repeated; this is why the time grows linearly. Figure 2.3 shows the last moment of the trace.

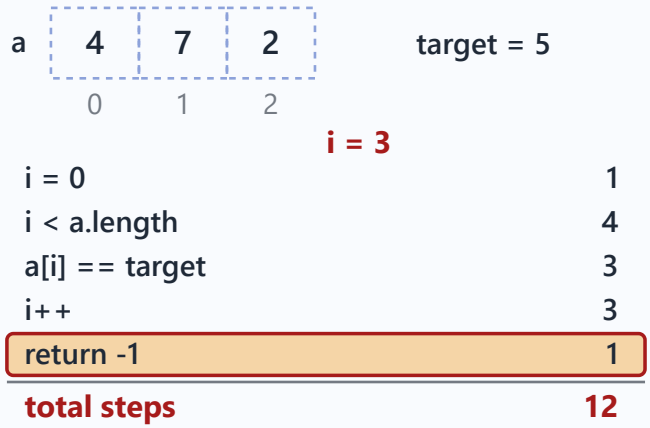


Figure 2.3: The end of the trace of `linearSearch([4, 7, 2], 5)`. The loop has ended at $i = 3$ and `return -1` has run. The table shows how many times each operation was performed: the condition 4 times, the comparison 3, the increment 3. The total is 12 steps.

2.1.5 Best, worst, and average case

The step count of linear search depends not only on n. In two arrays of the same length, if the target sits in different places, the step counts differ. If the target is in the first cell, the search ends at once; if it is absent, the whole array is scanned. So the question "how many steps does linear search take?" has no single answer. The question must be asked while saying which input is meant. Three measures are used for this.

Definition 2.2

Consider all inputs of size n.

- **Best case:** the number of operations of the input that requires the fewest operations among these inputs.
- **Worst case:** the number of operations of the input that requires the most.
- **Average case:** the expected number of operations under a probability distribution over the inputs.

The three measures do not have the same value. The **worst case** is the most used, because it gives a guarantee: the algorithm never takes longer than this on any input. In a real-time system, in the control software of an aircraft, or on a web server, this guarantee matters. The **best case** gives a lower bound, but by itself carries little information. Almost every algorithm is fast on some lucky inputs. The **average case** can give the most realistic estimate in practice; but to compute it one must know with what probability the inputs arrive. This information is often missing, and if the assumed distribution is wrong, so is the result.

Example 2.1

The array is `[7, 3, 9, 4, 8, 1]` ($n = 6$). The number of comparisons of linear search is computed for three cases.

1. **Best case.** The target is 7, the first element. It is found at the first comparison: 1 comparison. However long the array is, this number does not change: $\Theta(1)$.
2. **Worst case.** The target 5 is not in the array. All six elements are examined: 6 comparisons. If the target were the last element (1), 6 comparisons would be needed as well. In general, n comparisons: $\Theta(n)$.
3. **Average case.** Assume that the target is in the array and is equally likely (1/6) to be at each of the six positions. A target at position k is found with k comparisons. The expected value is $(1 + 2 + 3 + 4 + 5 + 6) / 6 = 21 / 6 = 3.5$. In general, $(1 + 2 + \ldots + n) / n = (n + 1) / 2$ comparisons are made. This number is half the worst case, but it still grows linearly with n: $\Theta(n)$.



Figure 2.4: Three searches on the array of Example 2.1. Top, the best case: the target 7 is in the first cell, 1 comparison. Middle, the worst case: the target 5 is absent, 6 comparisons. Bottom, a search close to the average: the target 4 is found in the fourth cell. Orange cells are elements looked at and passed over; the green cell is the element found.

The result $(n + 1)/2$ in the average-case computation comes from Gauss's summation formula: $1 + 2 + \ldots + n = n(n + 1)/2$. The same formula appears often in loop analysis. The factor 1/2 between the average and the worst case is a constant of the kind that asymptotic analysis discards. This is why the

average and worst cases of linear search fall in the same class. It is not so for every algorithm: quicksort takes $\Theta(n \log n)$ on average but $\Theta(n^2)$ in the worst case.

> **Common mistake**
>
> Rounding the average number of comparisons to $n/2$. If the target is equally likely at each position, the average is exactly $(n + 1)/2$; for $n = 6$ it is 3.5, not 3. The class is the same, but in questions that want an exact number this difference means a wrong answer.

2.1.6 Space complexity

The second resource, as important as time, is memory. Space complexity is usually given for the **auxiliary space**. The input itself is not counted, because every algorithm has to keep its input somewhere; what we really want to know is how much memory the algorithm asks for on top of that. An algorithm that sums an array with a single total variable uses only a few variables however long the array is. Its auxiliary space is $\Theta(1)$.

In recursive code there is a memory item that is easy to overlook. When a method is called, a **stack frame** is opened on the **call stack** for its parameters, local variables, and return address. When the method finishes, the frame is removed. As a recursive function calls itself, frames pile up. The largest number of frames on the stack at one time is the **depth** of the recursion. The auxiliary space is at least this depth.

Code	Where does the extra memory come from?	Space	Time
f1: a single loop	only `x` and `i`	$\Theta(1)$	$\Theta(n)$
creating `new int[n]`	an auxiliary array of n elements	$\Theta(n)$	$\Theta(n)$
f3: `f3(n-1) + f3(n-1)`	stack depth n	$\Theta(n)$	$\Theta(2^n)$
f4: `f4(n/2) + f4(n/2)`	stack depth about $\log_2 n$	$\Theta(\log n)$	$\Theta(n)$
Merge sort	auxiliary array for merging	$\Theta(n)$	$\Theta(n \log n)$

Table 2.4: Auxiliary space and time of the examples in this chapter.

Table 2.4 shows an important fact: time and space can grow at different rates. The function f3, which calls itself twice with $n - 1$, makes an exponential number ($2^n - 1$) of calls. Yet its memory is only linear. The reason is that the calls run one after another, not at the same time. `f3(n)` first calls the left `f3(n - 1)` and does not move to the right one until all subcalls of this call have finished and their frames have been removed. At any moment the stack holds only one chain: `f3(n)`, `f3(n - 1)`, …, `f3(1)`. The length of this chain is n.

The same reasoning applies to f4. In f4 the input halves with each call, so the chain has the form $n, n/2, n/4, \ldots, 1$ and its length is about $\log_2 n + 1$. For example, while `f4(16)` runs, the stack holds at most 5 frames (16, 8, 4, 2, 1).

Common mistake

Confusing the memory of a recursive function with the total number of calls. For f3 it is wrong to say "there are 2^n calls, each opens a frame, so the memory is $\Theta(2^n)$." The frame of a finished call is removed and its space is reused for the next call. The memory is determined by the largest number of frames open at the same time, that is, by the recursion depth.

2.2 Asymptotic notation

When the step count came out as $3n + 3$, we said that the coefficients would be discarded and only the fact that the growth is "linear" kept. The rules for this "discarding" must be precise; otherwise two people could assign different classes to the same code. **Asymptotic notation** sets these rules. The word "asymptotic" says that we are concerned with what happens as n goes to infinity. Behavior on small inputs and constant factors are deliberately ignored. Three notations are used: O for an upper bound, Ω for a lower bound, and Θ for a tight bound.

The O notation appeared long before algorithms, at the end of the 19th century, in number theory. Paul Bachmann used it in 1894, and Edmund Landau popularized it. In a short article in 1976, Donald Knuth defined Ω and Θ in their present senses and proposed that the three be used together in the analysis of algorithms. Today all algorithms textbooks use these definitions.

2.2.1 Big O

Big O states an upper bound. The statement "$f(n) = O(g(n))$" says that for every sufficiently large n, f does not exceed a constant multiple of g. Two details matter. First, the constant multiple is free: g may be multiplied by 4 or by 100. Second, small n are irrelevant: it is enough for the inequality to hold beyond a threshold value.

Definition 2.3

$f(n) = O(g(n))$ if and only if there exist a positive constant c and a threshold n_0 such that for every $n \geq n_0$

$$0 \leq f(n) \leq c \cdot g(n)$$

holds. In this case g is an **upper bound** for f. The notation is called **Big O notation**.

The definition has two constants, and each has its own job. c swallows constant factors. The factor-of-3 difference between $3n^2$ and n^2 disappears by choosing $c = 3$. n_0 leaves out small inputs. The function $n^2 + 100$ is larger than $2n^2$ at $n = 1$; but once $n \geq 10$ it does not exceed it. Because the definition is concerned only with what comes after n_0, such differences at the start do not affect the result.

An example: the function $f(n) = 3n^2 + 5n + 2$ is $O(n^2)$. Indeed, with $c = 4$, $3n^2 + 5n + 2 \le 4n^2$ holds for every $n \ge 6$. How these constants are found is shown step by step a little further down. Figure 2.5 shows the situation on a graph.

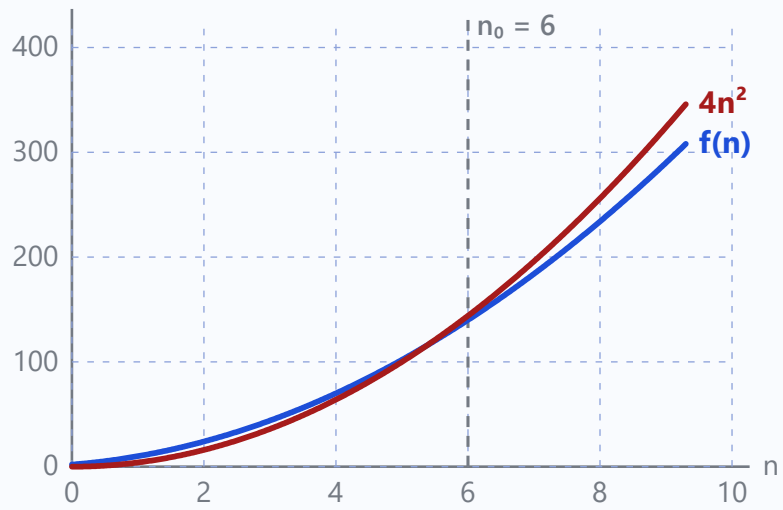


Figure 2.5: $f(n) = 3n^2 + 5n + 2$ (blue) and $4n^2$ (red). For small n, f is on top. The curves cross at $n \approx 5.37$. After $n_0 = 6$, f always stays below $4n^2$; hence $f(n) = O(n^2)$.

Because Big O is an upper bound, it can also be loose. $3n^2 + 5n + 2$ is $O(n^2)$, and also $O(n^3)$ and $O(2^n)$; all these statements are true. But their information value is not the same. The sentence "this algorithm runs in $O(2^n)$ time" is true but misleading for a quadratic algorithm. In analysis one usually gives the smallest upper bound that can be found.

The equals sign in the notation is also open to misunderstanding. $f(n) = O(g(n))$ actually means "f belongs to the set of functions bounded above by a constant multiple of g." The statement is therefore one-directional. $n = O(n^2)$ is true; but it does not follow that "$O(n^2) = n$." For this reason some books write $f(n) \in O(g(n))$.

Common mistake

Saying "Big O represents the worst case." Big O is a *kind of bound*; the worst case is a *kind of input*. The two are independent. An upper bound can be written for the worst-case running time, for the best-case running time, and for the average running time. "The best case of linear search is $O(1)$" is a valid sentence. The source of the confusion is that in practice Big O is mostly used for the worst-case running time.

2.2.2 Big Omega and Big Theta

The opposite of an upper bound is a lower bound. To say that an algorithm takes "at least this much," Ω (Big Omega) is used. If the two bounds meet at the same function, the growth rate is known exactly; Θ (Big Theta) expresses this.

Definition 2.4

$f(n) = \Omega(g(n))$: there exist $c > 0$ and n_0 such that for every $n \geq n_0$, $f(n) \geq c \cdot g(n) \geq 0$. In this case g is a **lower bound** for f.

$f(n) = \Theta(g(n))$: there exist $c_1, c_2 > 0$ and n_0 such that for every $n \geq n_0$, $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. In this case g is a **tight bound** for f.

Notation	Meaning	Definition	Analogue for numbers
f = O(g)	upper bound	$f(n) \leq c \cdot g(n)$	$f \leq g$
f = Ω(g)	lower bound	$f(n) \geq c \cdot g(n)$	$f \geq g$
f = Θ(g)	tight bound	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$f = g$

Table 2.5: The three asymptotic notations. In the definitions we look for positive constants for which the inequality holds for every $n \geq n_0$.

The last column of the table is a handy analogy: O behaves like "less than or equal," Ω like "greater than or equal," and Θ like "equal." The analogy also recalls the basic relation: a number that is both $\leq g$ and $\geq g$ equals g. In the same way,

$$f = \Theta(g) \Leftrightarrow f = O(g) \text{ and } f = \Omega(g).$$

Θ means squeezing between two constant multiples. Figure 2.6 shows this on $3n^2 + 5n + 2$: for $n \geq 6$ the function stays between $3n^2$ and $4n^2$. It is therefore $\Theta(n^2)$. Although the same function is $O(n^3)$, it is not $\Theta(n^3)$. Every lower curve of the form $c \cdot n^3$, however small c is chosen, overtakes the function at some point. This is because the ratio $f(n)/n^3$ tends to zero as n grows.

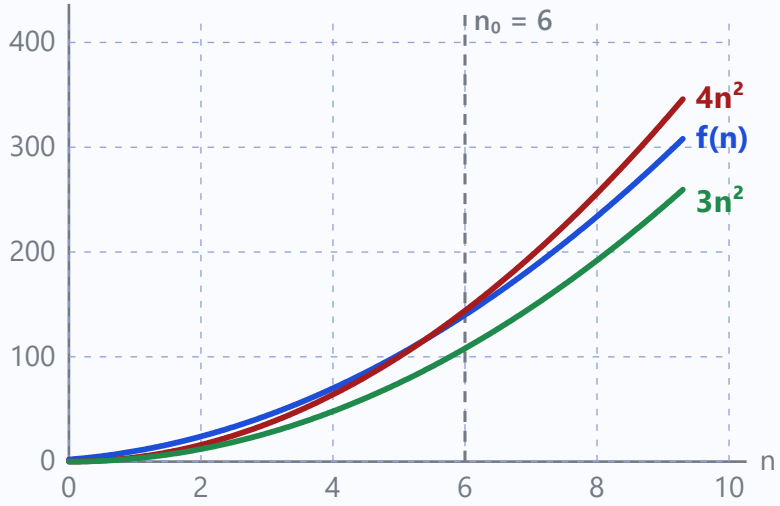


Figure 2.6: The squeezing meaning of Θ. For $n \geq 6$, $f(n) = 3n^2 + 5n + 2$ (blue) is bounded below by $3n^2$ (green) and above by $4n^2$ (red).

The distinction between growth class and case must be made once more here. For linear search all of the following sentences are true: the worst-case running time is $\Theta(n)$; the best-case running time is $\Theta(1)$; over all inputs the running time is $O(n)$ and $\Omega(1)$. But the sentence "linear search is $\Theta(n)$" is incomplete unless it says which case is meant. Insertion sort is similar: it takes $\Theta(n^2)$ in the worst case and $\Theta(n)$ on sorted input; a single Θ cannot be written for all inputs. Merge sort, on the other hand, takes $\Theta(n \log n)$ on every input; for it, writing Θ without naming a case is correct.

Common mistake

Using Θ and O interchangeably. "$3n^2 + 5n + 2 = \Theta(n^3)$" is wrong; what is correct is $O(n^3)$. Θ requires both a lower and an upper bound and gives the growth rate of the function exactly. In everyday speech, "this algorithm is $O(n^2)$" often means $\Theta(n^2)$; in a written analysis this distinction should be kept.

2.2.3 Finding the constants

Proving a bound means finding the constants in the definition concretely. The method is the same every time: choose a c, solve the inequality, and find from which n on it holds. The constants are not unique; any valid pair suffices for a proof.

Example 2.2

We show that $f(n) = 3n^2 + 5n + 2 = \Theta(n^2)$. For this two inequalities are needed: an upper bound and a lower bound.

1. **Choosing c_2 for the upper bound.** Since the leading coefficient is 3, a number slightly larger than 3 is tried: $c_2 = 4$. The desired inequality is $3n^2 + 5n + 2 \le 4n^2$.
2. **Solving the inequality.** Everything is collected on one side: $n^2 - 5n - 2 \ge 0$. The positive root of the quadratic equation is $(5 + \sqrt{33})/2 \approx 5.37$. The inequality holds for n larger than this root. Since the smallest integer is 6, $n_0 = 6$.
3. **Checking the bound.** At $n = 5$ the left side is $75 + 25 + 2 = 102$ and the right side is 100; the inequality fails. At $n = 6$ the left side is $108 + 30 + 2 = 140$ and the right side is 144; the inequality holds.
4. **Seeing that it keeps holding.** When n increases by one, the left side grows by $6n + 8$ and the right side by $8n + 4$. For $n \ge 6$, $8n + 4 > 6n + 8$, so the right side grows more at every step. The gap (4) that opened at $n = 6$ therefore never closes; it only grows.
5. **The lower bound.** Choose $c_1 = 3$. The inequality $3n^2 + 5n + 2 \ge 3n^2$ is true for every $n \ge 1$ because $5n + 2 > 0$.
6. **Combining.** Together the two inequalities hold for $n \ge 6$: $3n^2 \le f(n) \le 4n^2$. With $c_1 = 3$, $c_2 = 4$, $n_0 = 6$ the definition is satisfied: $f(n) = \Theta(n^2)$.

n	f(n)	$4n^2$	$f(n) \le 4n^2$?
1	10	4	no
3	44	36	no
5	102	100	no
6	140	144	yes
7	184	196	yes
8	234	256	yes

Table 2.6: Comparison of $f(n) = 3n^2 + 5n + 2$ with $4n^2$.

To see that the constants are not unique, another choice can be made. If $c_2 = 10$ is chosen, then already at $n = 1$, $3 + 5 + 2 = 10 \le 10$. Also, for every $n \ge 1$, $5n \le 5n^2$ and $2 \le 2n^2$, so one can write $3n^2 + 5n + 2 \le 3n^2 + 5n^2 + 2n^2 = 10n^2$. This proof does not require computing the root: each term is bounded by the largest power and the coefficients are added. A large c means a small n_0; a small c requires a larger threshold.

The same method applies to a linear function. To show that $2n + 10 = O(n)$, choose $c = 3$; then $2n + 10 \le 3n$ holds for $n \ge 10$: $c = 3$, $n_0 = 10$. With the term-by-term bounding method one writes $2n + 10 \le 2n + 10n = 12n$, which gives $c = 12$, $n_0 = 1$.

It is also possible to show that a bound is *wrong*. For this one proves that no pair of constants works, usually by contradiction. Suppose $n^2 = O(n)$. Then for some c and n_0, for every $n \ge n_0$ we must have $n^2 \le c \cdot n$, that is, $n \le c$. But n can be chosen larger than both n_0 and c, for example $n = max(n_0, \lceil c \rceil + 1)$. For this n, $n \le c$ fails. Since the assumption leads to a contradiction, it is false: $n^2 \ne O(n)$.

Common mistake

Trying only a single value of n in a proof. "At $n = 10$, $f(10) \leq 4 \cdot 100$, so it is $O(n^2)$" is insufficient. The definition requires the inequality to hold for *every* n beyond n_0. One has to solve the inequality algebraically or bound each term by the largest power.

2.2.4 Rules of calculation

Searching for constants every time is laborious. A few rules that follow from the definitions often let us read off the class directly.

- **Constant factors are dropped.** $c \cdot f(n) = \Theta(f(n))$; for example $5n = \Theta(n)$ and $n^2/3 = \Theta(n^2)$. The constant factor is absorbed into the c of the definition.
- **Lower-order terms are dropped.** In a polynomial the term with the largest power remains: $4n^3 + 100n^2 + 7 = \Theta(n^3)$. As n grows, $100n^2$ becomes negligible next to $4n^3$; at $n = 1000$ it is about a fortieth of it.
- **Sum rule: sequential code.** The total running time of two pieces of code that run one after the other is the running time of the larger: $O(f) + O(g) = O(\max(f, g))$. If a loop that takes $\Theta(n)$ is followed by a loop that takes $\Theta(n^2)$, the total is $\Theta(n^2)$.
- **Product rule: nested code.** If a loop that runs n times contains work that takes $O(g)$, the total is $O(n \cdot g)$. The rule assumes that the inner work is the same in every round. If the running time of the inner work depends on the counter of the outer loop, a sum is computed instead of a product.
- **The base of a logarithm is irrelevant.** By the change-of-base formula, $\log_a n = \log_b n / \log_b a$. The difference between two bases is only a constant factor. This is why one writes $O(\log_2 n) = O(\log_{10} n) = O(\log n)$ and often does not state the base.
- **In exponentials the base matters.** $3^n = O(2^n)$ is false. The ratio $3^n/2^n = 1.5^n$ goes to infinity; no constant factor can close this gap. Unlike logarithms, exponential bases are not related by a constant factor.

As a consequence of these rules, the common functions fall into a definite order of growth:

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$$

Here the sign "<" says that the one on the left becomes negligible relative to the one on the right: for large n the ratio goes to zero. For example the ratio $\log n / \sqrt{n}$ approaches zero, though slowly at first.

Common mistake

Leaving constants and small terms in the result: writing $O(3n + 3)$ or $O(n^2/3 + n)$. These expressions are not wrong, but they go against the purpose of the notation. The correct forms are $O(n)$ and $O(n^2)$. Similarly one writes $O(2^n)$ instead of $O(2^{n+1})$, since $2^{n+1} = 2 \cdot 2^n$; but $2^{2n} = 4^n$

is not $O(2^n)$.

2.2.5 Common growth classes

A very large share of algorithms fall into one of a few growth classes. Which class an algorithm belongs to tells at once how it will behave as the input grows. Table 2.7 lists these classes from fastest to slowest. The last column shows which class each of the six example functions of this chapter falls into.

Class	Name	Typical example	When n doubles	In this chapter
O(1)	constant	array access, printing the first element	unchanged	—
O(log n)	logarithmic	binary search, fast exponentiation, Euclid's algorithm	+1 step	f6 (upper bound)
O(√n)	square-root	primality testing by trial division	≈ 1.41 times	—
O(n)	linear	linear search, largest element, sum of an array	2 times	f1, f4
O(n log n)	linearithmic	merge sort, heap sort, quicksort (average)	slightly more than 2 times	f5
O(n^2)	quadratic	bubble, selection, and insertion sort	4 times	—
O(n^3)	cubic	classical matrix multiplication	8 times	f2
O(2^n)	exponential	all subsets, naive recursive Fibonacci	squared	f3
O(n!)	factorial	all permutations	explodes much faster	—

Table 2.7: Common growth classes.

The "when n doubles" column of the table is the shortest way to convey the difference between the classes. In a logarithmic algorithm only one step is added when the input doubles; this is why binary search finishes with about thirty comparisons even on a billion elements. For the square root the factor is $\sqrt{2} \approx 1.41$. For linearithmic the factor is slightly more than 2: $2n \log(2n) = 2n \log n + 2n$. In the exponential class the situation is entirely different: since $2^{2n} = (2^n)^2$, the time is squared. A computation of 2^n steps that takes 1 second on a computer performing 10^9 operations per second ($n \approx 30$) takes about 36 years when the input doubles.

There is one more important line between the classes. Classes of the form n^k are called **polynomial time**; they are generally regarded as "efficient." The exponential and factorial classes are regarded as "inefficient"; beyond small inputs they cannot be run in practice. This distinction is a rough rule (n^{100} is a polynomial but not at all efficient), yet it still guides algorithm design.

A note is needed on f6 in the "In this chapter" column of the table. It is true that f6 is $O(\log n)$, but this bound is not tight. The true class of f6 is $\log^* n$, which grows even much more slowly than the logarithm. This interesting function is examined in detail in the section on loop analysis.

2.2.6 Growth curves and numbers

The order of the classes can also be seen on a graph. Figure 2.7 plots five functions for n from 1 to 20. For small n the curves are close together; at $n = 4$, n^2 and 2^n are even equal (both are 16). After $n = 5$, 2^n pulls ahead and rapidly moves away. The curve 2^n exceeds 100 at $n \approx 6.6$; n^2 exceeds it only after $n = 10$. $n \log_2 n$ stays close to linear, and $\log_2 n$ is nearly flat.

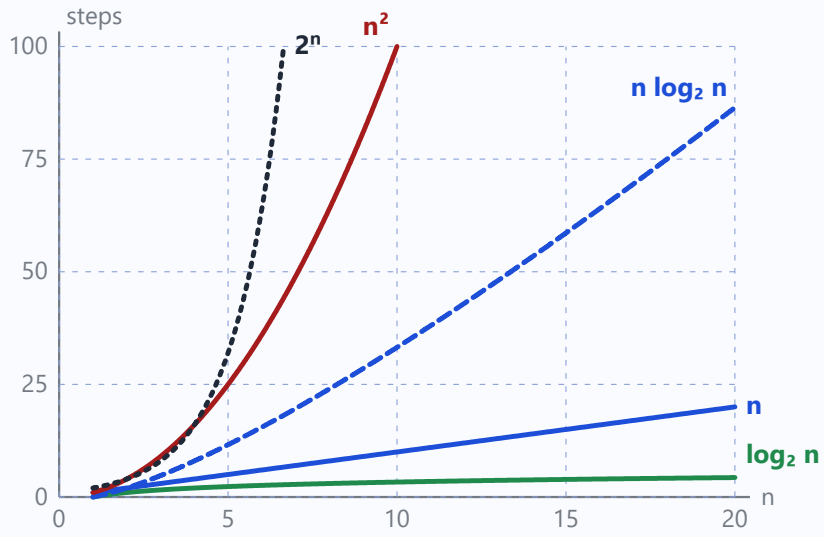


Figure 2.7: $\log_2 n$, n, $n \log_2 n$, n^2, and 2^n for n from 1 to 20. The vertical axis is cut off at 100 steps; 2^n and n^2 soon exceed this limit.

The difference in the graph is even more striking in numbers. Table 2.8 shows how long each class takes on a computer that performs 10^9 basic operations per second. The values were computed by dividing the operation count by 10^9.

f(n)	n = 10	n = 20	n = 50	n = 100	n = 10^6
$\log_2 n$	3.3 ns	4.3 ns	5.6 ns	6.6 ns	19.9 ns
n	10 ns	20 ns	50 ns	100 ns	1 ms
$n \log_2 n$	33 ns	86 ns	282 ns	664 ns	19.9 ms
n^2	100 ns	400 ns	2.5 µs	10 µs	16.7 minutes
n^3	1 µs	8 µs	125 µs	1 ms	31.7 years
2^n	1 µs	1 ms	13 days	$4 \cdot 10^{13}$ years	—
$n!$	3.6 ms	77 years	10^{48} years	$3 \cdot 10^{141}$ years	—

Table 2.8: Approximate running times on a computer performing 10^9 operations per second.

The table says two things. First, the polynomial classes remain reasonable even on a million elements; the quadratic algorithm takes 17 minutes and the cubic algorithm more than thirty years. Second, the exponential and factorial classes become unrunnable even on very small inputs. For $n = 100$, 2^n steps take three thousand times the age of the universe (about $1.4 \cdot 10^{10}$ years).

The interactive calculator in the slides produces the same table for any desired n; the "n × 2" button doubles the input. When the button is pressed, the n column doubles, the n^2 column quadruples, and the n^3 column grows eightfold; $log_2\ n$ rises by only 1, and 2^n squares itself. The lesson is this: a computer ten times faster buys a constant factor, while a better algorithm changes the growth class. An algorithm that takes $n\ log\ n$ instead of n^2 reduces a 17-minute job to 20 milliseconds at $n = 10^6$; no hardware improvement gains that much.

2.3 Loop analysis

A piece of code without loops and without recursion performs a constant number of operations; its running time is $\Theta(1)$. What ties the running time to the input is loops and recursion. This section deals with loops. The method can be stated in one sentence: find how many rounds each loop runs, and for nested loops add up these round counts. The method is applied to three example functions: f1 with a single loop, f2 whose inner loop bound depends on the outer counter, and f6 whose counter grows at an unusual rate.

The six example functions of this chapter share a trick: each returns a counter `x` that increases with the work done. The return value thus becomes the step count itself. The formula found by analysis can be checked directly by running the function and looking at the return value. All the return values and call counts given in these notes were obtained by running the code in Java.

2.3.1 f1: a single loop

```
public int f1(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    x++;
  }
  return x;
}
```

The code has three parts.

- **Initialization:** the line `int x = 0;` assigns 0 to `x`. This is a single operation, independent of the input: $O(1)$.
- **The loop:** in the header `for (int i = 0; i < n; i++)`, `i` starts at 0, increases by one each round, and goes up to $n - 1$. The values $0, 1, \ldots, n - 1$ number n; the loop runs n rounds.
- **The body:** `x++;` runs once per round and is a constant-time operation: $O(1)$.

By the product rule, a loop that runs n times doing $O(1)$ work per round does $n \cdot O(1) = O(n)$ work in total. The initialization and the `return` outside the loop are constant work; by the sum rule they do not change the result:

$$T(n) = O(1) + n \cdot O(1) = \Theta(n)$$

Since the body runs exactly n times, the lower bound is also n; the result is therefore written with Θ. The function returns `x = n`: `f1(16) = 16`, `f1(32) = 32`. The return value is the step count itself, and it doubles when the input doubles. This is the signature of linear growth.

2.3.2 Code trace: f1(4)

Let us trace f1 for $n = 4$. On each line the value of the condition and the new value of `x` are written.

Step	`i`	Condition `i < 4`	`x` (after the body)
start	—	—	0
round 1	0	true	1
round 2	1	true	2
round 3	2	true	3
round 4	3	true	4
exit	4	false	returns 4

Table 2.9: Trace of `f1(4)`.

The condition was checked five times ($n + 1$) and the body ran four times (n). These counts have the same form as the count in linear search. Since the body runs exactly n times, the result $\Theta(n)$ is visible in the trace too. Figure 2.8 shows the final state of the trace.

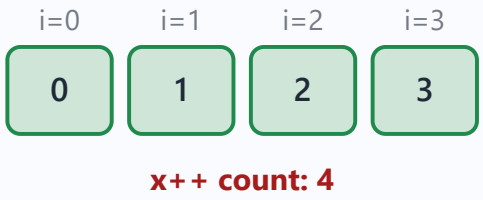


Figure 2.8: The end of `f1(4)`. Each box is a loop round; all four rounds, $i = 0, 1, 2, 3$, have been completed and `x++` has been executed four times.

2.3.3 f2: nested loops

```
public int f2(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i * i; j++) {
      x++;
    }
  }
  return x;
}
```

In f2 two loops are nested. The outer loop is like the one in f1: `i` goes from 0 to $n - 1$ and runs n rounds. The bound of the inner loop, however, is not constant. `j` goes from 0 to $i \cdot i - 1$; that is, in the current round of the outer loop the inner loop runs i^2 times. At $i = 0$ the inner loop does not run at all, at $i = 1$ it runs once, at $i = 2$ four times, at $i = 3$ nine times.

Because the number of rounds of the inner loop depends on the current value of i, the product rule cannot be applied directly. To say "the outer loop runs n rounds, the inner loop n rounds, total $n \cdot n$" is wrong, because the inner loop runs a different number of times in each round. The right way is to write down the inner round count for each outer round and add them up:

$$T(n) = 0^2 + 1^2 + 2^2 + \ldots + (n-1)^2 = \Sigma_{i=0}^{n-1} i^2$$

Common mistake

Automatically counting two nested loops as $O(n^2)$. The result is determined by the bound of the inner loop. In f2 the bound is i^2, so the result is of order n^3. With the bound `j < i` it would be quadratic, with `j < 10` linear. If the outer loop advanced with `i *= 2`, the result would again come out different.

2.3.4 Code trace: f2(4) and f2(5)

In the call `f2(4)` the outer loop runs four rounds ($i = 0, 1, 2, 3$). Table 2.10 shows the value of `x` after each outer round. The values were verified by running the code in Java.

`i`	Inner loop condition	Inner rounds	`x` (after the round)
0	`j < 0`	0	0
1	`j < 1`	1	1
2	`j < 4`	4	5
3	`j < 9`	9	14

`i`	Inner loop condition	Inner rounds	`x` (after the round)
4	outer condition `4 < 4` false	—	returns 14

Table 2.10: Trace of `f2(4)`.

The round *i = 0* is interesting: the condition of the inner loop is `j < 0 · 0`, that is, `0 < 0`, which is false at the first check. The inner loop does not run at all, but the outer round still counts. The result is *0 + 1 + 4 + 9 = 14*. Figure 2.9 shows the rounds as boxes, row by row: row *i* has i^2 boxes.

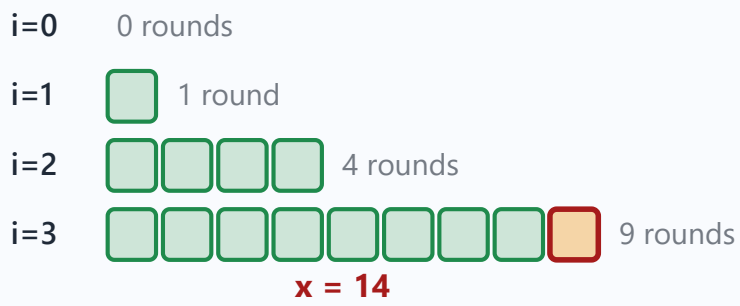


Figure 2.9: The rounds of the inner loop in `f2(4)`. Each row is an outer round and each box is an `x++`. The rows have lengths 0, 1, 4, 9; total 14.

On a slightly larger input the pattern becomes clearer. In `f2(5)` the outer loop runs one more round, and at *i = 4* the inner loop runs 16 times. The total is *0 + 1 + 4 + 9 + 16 = 30*. Figure 2.10 shows each outer round as a bar. The bar heights are i^2 and the total area is the number of `x++` the function performs. The last bar alone makes up more than half of the total: the last rounds dominate.

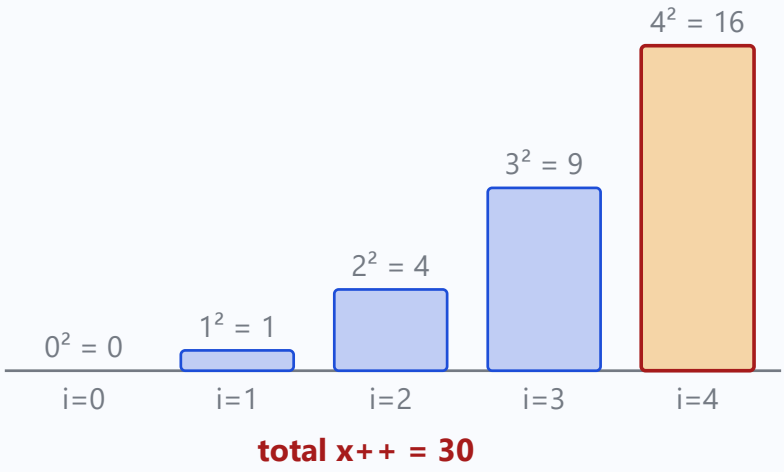


Figure 2.10: The number of inner rounds in each outer round of `f2(5)`. The bars have heights 0, 1, 4, 9, and 16; their sum is 30 and `f2(5)` returns 30.

2.3.5 Computing the sum

There is a closed formula for the sum of consecutive squares:

$$\Sigma_{i=0}^{n-1} i^2 = (n - 1) \cdot n \cdot (2n - 1) / 6 = (2n^3 - 3n^2 + n) / 6$$

The formula can be checked with small values. For $n = 5$ it gives $4 \cdot 5 \cdot 9 / 6 = 30$, the total in the trace. For $n = 4$ it gives $3 \cdot 4 \cdot 7 / 6 = 14$. In the expanded form the dominant term is $2n^3/6 = n^3/3$. Dropping the constant $1/3$ and the lower-order terms gives the result:

$$T(n) = \Theta(n^3)$$

n	f2(n)	$n^3/3$
4	14	21
5	30	42
16	1,240	1,365
32	10,416	10,923

Table 2.11: The return value of `f2(n)` compared with $n^3/3$ (measured in Java).

The two columns in the table approach each other as n grows. At $n = 4$ the ratio is $14/21 \approx 0.67$, while at $n = 32$ it is $10{,}416/10{,}923 \approx 0.95$. The lower-order term $-3n^2/6$ matters at small n and loses its importance at large n. A further sign of cubic growth is the doubling test. When n goes from 16 to 32, the return value grows by a factor of $10{,}416 / 1{,}240 = 8.4$. This number is close to $2^3 = 8$; as n grows, the ratio gets closer still to 8.

The result in this section is an example of a general rule: $\Sigma_{i<n} i^k = \Theta(n^{k+1})$. If the round count of the inner loop is the k-th power of the outer counter, the total work goes up by one degree. The rule comes from the sum resembling an integral: $\int_0^n x^k \, dx = n^{k+1}/(k + 1)$. For f2, $k = 2$ and the integral gives $n^3/3$, the same as the dominant term in the exact formula.

2.3.6 Loop patterns and summation formulas

In loop analysis a few patterns come up again and again. Each reduces to a known series: an arithmetic series, a sum of squares, or a geometric series. Table 2.12 collects these patterns. The round counts were checked by counting in Java for $n = 16$ and $n = 20$.

Pattern	Round count	n = 16	Result
`for (i = 0; i < n; i++)`	$\Sigma\, 1 = n$	16	$\Theta(n)$
`for (j = 0; j < n; j++)` inside `i < n`	$n \cdot n$	256	$\Theta(n^2)$

Pattern	Round count	n = 16	Result
`for (j = 0; j < i; j++)` inside `i < n`	$\Sigma\, i = n(n-1)/2$	120	$\Theta(n^2)$
`for (j = 0; j < i * i; j++)` inside `i < n`	$\Sigma\, i^2 = (n-1)n(2n-1)/6$	1,240	$\Theta(n^3)$
`for (i = 1; i < n; i *= 2)`	$\lceil \log_2 n \rceil$	4	$\Theta(\log n)$
`j < n` inside `for (i = n; i > 0; i /= 2)`	$n \cdot (\lfloor \log_2 n \rfloor + 1)$	80	$\Theta(n \log n)$
`j < i` inside `for (i = 1; i < n; i *= 2)`	$1 + 2 + 4 + \ldots < 2n$	15	$\Theta(n)$
`for (i = 0; i < n; i = 1 << i)` (f6)	$\log^* n + 1$	4	$\Theta(\log^* n)$ (mathematical model; in Java, $n \le 65536$)

Table 2.12: Frequently encountered loop patterns.

The third row of the table is a lower-degree sibling of f2. The inner loop runs i times; the total is $0 + 1 + \ldots + (n-1) = n(n-1)/2$. This number is about half of $n \cdot n$; since the difference is a constant factor, the class is still $\Theta(n^2)$. This pattern is the reason selection sort and insertion sort are quadratic.

The fifth row is a multiplicative loop. `i` doubles in each round: 1, 2, 4, 8, ... After k rounds $i = 2^k$. The loop continues while $2^k < n$; it stops when $k \ge \log_2 n$. The round count is therefore $\lceil \log_2 n \rceil$. At $n = 16$, `i` takes the values 1, 2, 4, 8 and the loop runs 4 rounds; at $n = 1000$ it runs 10 rounds, from 1 to 512. Every loop whose counter grows or shrinks by a constant ratio is logarithmic.

In the sixth row the outer loop is logarithmic and the inner loop is linear. The bound of the inner loop does not depend on the outer counter; it runs exactly n times in every round. The product rule therefore applies: $(\lfloor \log_2 n \rfloor + 1) \cdot n$. For $n = 16$ the outer loop runs 5 rounds, at `i` = 16, 8, 4, 2, 1; the total is $5 \cdot 16 = 80$.

The seventh row is surprising. On the surface its structure resembles the previous one: a logarithmic outer loop with an inner loop inside it. But the bound of the inner loop is i, and i grows quickly. The total is a geometric series:

$$1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1 < 2n$$

Here 2^k is the largest power of two smaller than n; hence $2^{k+1} < 2n$. Although the outer loop runs $\log n$ times, the total work is linear. At $n = 16$, $1 + 2 + 4 + 8 = 15$ rounds are made. This is the property of a geometric series: the sum is determined by its last term, and the last term is larger than the sum of all the earlier ones.

Example 2.3

Find the number of `x++` executions and the growth class of the following two pieces of code.

```
// first part
for (int i = 1; i < n; i *= 2)
  for (int j = 0; j < i; j++)
    x++;

// second part
for (int i = n; i > 0; i /= 2)
  for (int j = 0; j < n; j++)
    x++;
```

1. **First part.** `i` takes the values 1, 2, 4, ..., $2^k < n$. In each round the inner loop runs `i` times. The total is $1 + 2 + \ldots + 2^k = 2^{k+1} - 1 < 2n$: $\Theta(n)$. For $n = 16$, `x++` is executed 15 times.
2. **Second part.** The outer loop runs $\lfloor \log_2 n \rfloor + 1$ rounds, at `i` $= n, n/2, n/4, \ldots, 1$. The inner loop runs n times in each round, because its bound is n. The total is $n \cdot (\lfloor \log_2 n \rfloor + 1)$: $\Theta(n \log n)$. For $n = 10$, `i` = 10, 5, 2, 1 and `x++` is executed $4 \cdot 10 = 40$ times.

The outer loops of the two parts run a similar number of rounds; the difference comes from the bound of the inner loop. If the bound depends on the outer counter, the round counts are added; if it does not, they are multiplied.

Common mistake

Always counting an inner loop inside a logarithmic outer loop as $\Theta(n \log n)$. The first part of the example above is only $\Theta(n)$, although the outer loop runs $\log n$ times. One does not decide without writing down the inner round count for each outer round and adding them up.

2.3.7 f6: a loop with a shifting bound

```
public static int f6(int n) {
  int x = 0;
  // 1 << i equals 2^i (overflow is ignored)
  for (int i = 0; i < n; i = 1 << i) {
    x++;
  }
  return x;
}
```

At first sight the loop of f6 resembles a multiplicative loop, but its update is different.

- **Initialization:** `int i = 0;` is a constant-time assignment.
- **Condition:** `i < n` checks at the start of each round whether `i` is smaller than n.

- **Update:** `i = 1 << i`. The left-shift operator `<<` shifts the bits of the number 1 to the left by `i` places. In binary, shifting 1 to the left by *i* places multiplies it by 2^i. The result is 2^i, computed in a single operation, in constant time.

The critical point is this: the update is not `i = 2 * i` but `i = 2^i`. In a multiplicative loop `i` doubles in each round. In f6, `i` itself becomes the exponent. This makes `i` grow much faster: the value 4 becomes not 8 in the next round but $2^4 = 16$. And 16 becomes not 32 but $2^{16} = 65{,}536$.

2.3.8 Code trace: f6(20)

Round	`i`	Condition `i < 20`	`x`	Next `i = 1 << i`
1	0	true	1	1 << 0 = 1
2	1	true	2	1 << 1 = 2
3	2	true	3	1 << 2 = 4
4	4	true	4	1 << 4 = 16
5	16	true	5	1 << 16 = 65,536
—	65,536	false	returns 5	—

Table 2.13: Trace of `f6(20)`.

The loop runs only five rounds. The values taken by `i` are 0, 1, 2, 4, 16; the next value is 65,536, so the loop ends. Figure 2.11 shows this chain. The same loop also runs five rounds for $n = 65{,}536$; although n has grown more than three thousandfold, the round count does not change.

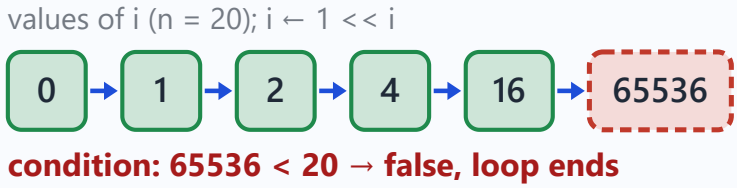


Figure 2.11: The values taken by `i` in `f6(20)`: 0 → 1 → 2 → 4 → 16 → 65,536. The last value fails the condition (*65,536 < 20* is false) and the loop ends after five rounds.

2.3.9 How many rounds does f6 really run?

The values of `i` in f6 grow like a tower of powers of two:

$$0 \to 2^0 = 1 \to 2^1 = 2 \to 2^2 = 4 \to 2^4 = 16 \to 2^{16} = 65{,}536 \to 2^{65{,}536}$$

The last value is a number with 19,729 digits. The number of atoms in the universe is estimated at about 10^{80}; next to this number, 10^{80} is tiny. Since no real input can be that large, f6 runs at most six rounds for any n one can imagine.

A common line of reasoning for this loop goes: "after k rounds $i \geq 2^k$; the loop stops when $2^k \geq n$; so $k \approx \log_2 n$ and the time is $O(\log n)$." The reasoning is correct for the loop `i = 2 * i`. For f6 the result is not wrong either, because `i` at least doubles, so the round count cannot exceed $\log_2 n$. But this bound is not tight. In f6, after k rounds `i` is a tower of height $k - 1$, and the round count is given by a function that grows much more slowly.

Definition 2.5

The **iterated logarithm** $\log^* n$ gives the number of times $\log_2$ must be applied to n in succession until the result is less than or equal to 1. For example, since $\log_2 16 = 4$, $\log_2 4 = 2$, $\log_2 2 = 1$, we have $\log^* 16 = 3$.

The round count of f6 is exactly $\log^* n + 1$. This equality was checked for every value from $n = 1$ to 70,000 with a Python script. $\log^* n$ does not exceed 5 for any input one can imagine: $\log^* 65{,}536 = 4$, and to reach 5, n must exceed 65,536. The smallest number with value 6 is larger than $2^{65{,}536}$. This function, which behaves like a constant in practice, also appears in the analysis of the union-find data structure.

n	f6(n)	$\lceil \log_2 n \rceil$	$\log^* n$
4	3	2	2
16	4	4	3
32	5	5	4
1,000	5	10	4
65,536	5	16	4

Table 2.14: The return value of `f6(n)` compared with $\lceil \log_2 n \rceil$ (measured in Java).

For small n, f6 and $\lceil \log_2 n \rceil$ are close together; at $n = 32$ both are 5. As n grows, the logarithm keeps increasing while f6 stays at 5. Conclusion: $O(\log n)$ is a correct upper bound, but the tight bound is $\Theta(\log^* n)$.

The comment line in the code of f6 says "overflow is ignored." This assumption matters in Java. In Java an `int` has 32 bits, and the shift operator uses only the low five bits of the shift amount; that is, it takes the amount modulo 32. Since $65{,}536 = 32 \cdot 2048$, `1 << 65536` is the same as `1 << 0` and gives 1. So for $n > 65{,}536$, `i` enters the cycle 0, 1, 2, 4, 16, 65,536, 1, 2, 4, 16, 65,536, …; the condition never becomes false and f6 never terminates. This behavior was observed with a copy run at $n = 100{,}000$ with the round count capped. The result $\Theta(\log^* n)$ in the analysis holds for an abstract machine with unbounded integers.

Common mistake

Confusing a shift with a multiplication. The update `i = 1 << i` is $i = 2^i$, not `i = 2 * i`. The latter is written `i <<= 1` or `i = i << 1`. If the shifted number and the shift amount trade places, the round count of the loop changes between *log n* and *log* n*.

2.4 Analysis of recursion

In looping code the round counts are added. In recursive code, the running time depends on the running time of the same function on smaller inputs. To find *T(n)*, values such as *T(n − 1)* or *T(n/2)* are needed first. The equation that expresses this relation is called a recurrence relation. This section explains how the relation is set up and the most basic solution method, unrolling.

2.4.1 The recurrence relation

Definition 2.6

A **recurrence relation** is an equation that defines *T(n)* in terms of the values of *T* on smaller inputs. For the running time of a recursive function, the general form is:

> T(n) = (running times of the subcalls) + (work done outside the calls), T(base) = Θ(1)

The second equation gives the running time of the **base case**, which the recursion solves directly without calling itself.

To set up the relation, three questions are asked of the code:

1. **How many recursive calls are there?** This number becomes the coefficient of *T* in the relation.
2. **By how much does the input of each call shrink?** If the input decreases by one, write *T(n − 1)*; if it halves, write *T(n/2)*.
3. **How much work is done outside the calls?** The running time of work such as splitting, merging, and adding enters the relation as an additional term.

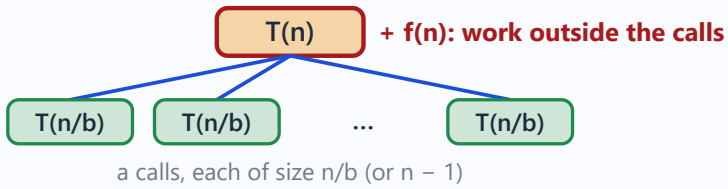


Figure 2.12: The structure of a recursive call. A call that takes $T(n)$ starts a subcalls; each works with a smaller input (n/b or $n - 1$). The work the call itself does is $f(n)$.

Once the relation is set up, it has to be solved; that is, a closed form of $T(n)$ not containing T is found. There are three methods:

- **Unrolling** (substitution): the relation is substituted into itself repeatedly, a pattern is found, and when the base case is reached the terms are added up. Used for f3.
- **Recursion tree:** each call is drawn as a node, the work at each level is added up, and then the level sums are added. Used for f3, f4, and f5.
- **Master Theorem:** gives a ready-made result for relations of the form $T(n) = aT(n/b) + f(n)$. Used for f4 and f5.

2.4.2 f3: two calls, n − 1

```java
public long f3(int n) {
  if (n <= 1) {
    return 1;
  }
  return f3(n - 1) + f3(n - 1);
}
```

f3 has two parts.

- **Base case:** when $n \le 1$ the function returns 1. This is constant-time work: $T(1) = c$.
- **Recursive case:** when $n > 1$, `f3(n - 1)` is called twice and the results are added. The addition and the comparison are constant work; this too is called c.

Once the three questions are answered, the relation emerges: there are two calls, each of size $n - 1$, and constant work is done outside the calls.

$$T(n) = 2T(n - 1) + c, \quad n > 1; \qquad T(1) = c$$

The input shrinks by only one in each call, but the number of calls doubles at every level. `f3(4)` calls two `f3(3)`; each `f3(3)` calls two `f3(2)`; each `f3(2)` calls two `f3(1)`. The levels have 1, 2, 4, 8 calls, and the total is $1 + 2 + 4 + 8 = 15 = 2^4 - 1$ (Figure 2.13).

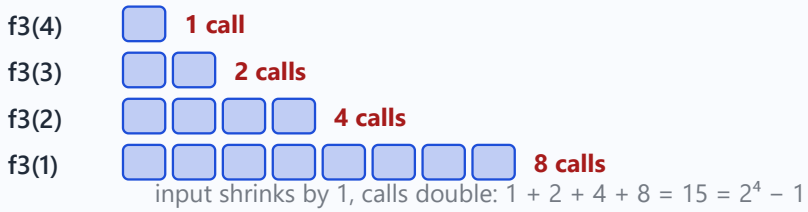


Figure 2.13: The levels of `f3(4)`. The input shrinks by one at each level while the number of calls doubles: 1, 2, 4, 8. In all, $15 = 2^4 - 1$ calls.

This relation is not of the form $T(n/b)$; the subproblem shrinks not by division but by subtracting one. The Master Theorem therefore does not apply, and the relation is solved by unrolling.

2.4.3 Code trace: f3(3)

In tracing recursive code, the order in which calls start and end matters. The steps of the call `f3(3)` are as follows; the indentation shows the depth of the call. The order was verified with a copy run in Java that prints the start and end of every call.

Step	Event	Stack (bottom to top)	Calls
1	`f3(3)` is called; 3 > 1, not the base case	f3(3)	1
2	the left `f3(2)` is called	f3(3) f3(2)	2
3	`f3(1)`: base case, returns 1	f3(3) f3(2) f3(1)	3
4	second `f3(1)`: returns 1	f3(3) f3(2) f3(1)	4
5	`f3(2) = 1 + 1 = 2` returns	f3(3)	4
6	the right `f3(2)` is called; the same work is done again from scratch	f3(3) f3(2)	5
7	two `f3(1)`, each returns 1	f3(3) f3(2) f3(1)	7
8	the second `f3(2) = 2` returns	f3(3)	7
9	`f3(3) = 2 + 2 = 4` returns	empty	7

Table 2.15: Trace of `f3(3)`. The last column is the number of calls started so far.

The trace shows three things. In all $7 = 2^3 - 1$ calls were made. The stack held at most three frames at the same time; this is the depth, which equals n. Third and most important, the right `f3(2)` repeats exactly the work the left one did. This repetition is the source of f3's exponential running time. Had the code been written as `return 2 * f3(n - 1);`, the return value would have stayed the same (2^{n-1}), but only one call would be made at each level. The relation would reduce to $T(n) = T(n-1) + c$ and the time would be $\Theta(n)$.

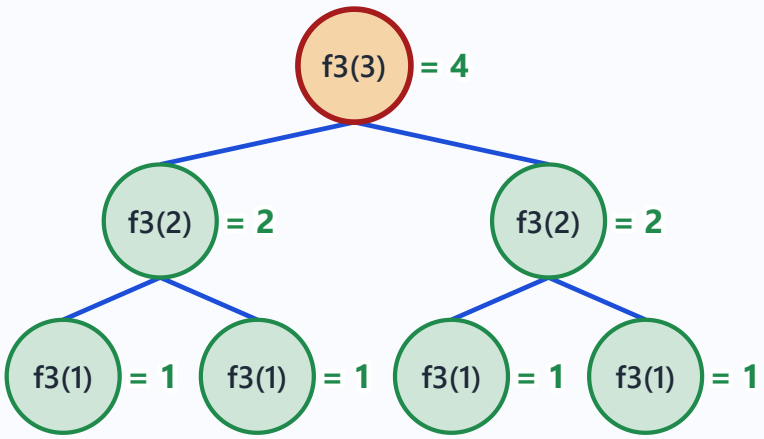


Figure 2.14: The completed call tree of `f3(3)`. Next to each node is the value it returns. The four leaves return 1, the two `f3(2)` return 2, the root returns 4. The tree has 7 nodes.

2.4.4 Unrolling the recurrence

In the unrolling method, the $T(n-1)$ on the right-hand side of the relation is replaced by its own definition. After a few repetitions a pattern appears in the coefficients. The pattern is written in general form and the step at which the base case is reached is found.

Example 2.4

Solve the relation $T(n) = 2T(n-1) + c$, $T(1) = c$ by unrolling.

1. **The relation itself:** $T(n) = 2T(n-1) + c$. Two subcalls, constant extra work.
2. **Unrolling once:** substitute $T(n-1) = 2T(n-2) + c$: $T(n) = 2[2T(n-2) + c] + c = 4T(n-2) + 3c$.
3. **Once more:** substitute $T(n-2) = 2T(n-3) + c$: $T(n) = 4[2T(n-3) + c] + 3c = 8T(n-3) + 7c$.
4. **The pattern:** the coefficients of T are 2, 4, 8; those of c are 1, 3, 7. The first are powers of two, the second are powers of two minus one. After k steps: $T(n) = 2^k\,T(n-k) + (2^k - 1)c$.
5. **Reaching the base case:** when $n - k = 1$, that is, $k = n - 1$, use $T(1) = c$: $T(n) = 2^{n-1}c + (2^{n-1} - 1)c = (2^n - 1)c$.
6. **Result:** $T(n) = \Theta(2^n)$.

The result agrees exactly with the number of calls of f3: each call does c work and there are $2^n - 1$ calls. The formula can be checked with small values: $T(1) = c$, $T(2) = 3c$, $T(3) = 7c$. Counted in Java, `f3(16)` makes 65,535 calls and `f3(32)` makes 4,294,967,295; both equal $2^n - 1$. When n goes from 16 to 32, the number of calls grows by a factor of 65,537. This is the signature of exponential growth: when the input doubles, the time reaches a value close to its square.

In unrolling it is also possible to neglect the extra work (c) and write $T(n) = 2^k\ T(n - k)$. This gives $T(n) = 2^{n-1}\ T(1)$ and the growth class does not change. The extra work only doubles the constant factor.

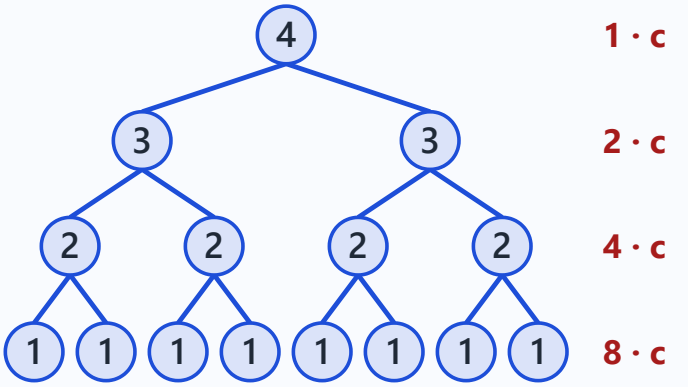


Figure 2.15: The recursion tree of `f3(4)`. The number in each node is its input. The work of the levels is $c, 2c, 4c, 8c$; the total is $15c = (2^4 - 1)c$.

Unrolling also applies to other subtractive relations. Unrolling $T(n) = T(n - 1) + n$, $T(1) = 1$ gives $T(n) = n + (n - 1) + \ldots + 2 + 1 = n(n + 1)/2$: $\Theta(n^2)$. This relation arises in algorithms that do n work at each step and shrink the problem by only one. The worst case of quicksort is an example.

Common mistake

Memorizing the rule “two recursive calls mean exponential time.” In f3 two calls give $\Theta(2^n)$ because the input shrinks by only one and the depth of the tree is n. If the input halved, the depth would be $\log_2 n$ and the time would drop to linear. What decides is not the number of calls but the joint effect of the number of calls and the way the input shrinks.

2.5 The Master Theorem

Unrolling works for every recurrence, but it is laborious. A large share of algorithms use the **divide-and-conquer** approach, which splits the problem into pieces of equal size. The recurrences of these algorithms always have the same form. The Master Theorem gives a ready-made result for recurrences of this form. The theorem rests on a general method published by Jon Bentley, Dorothea Haken, and James Saxe in 1980; the name “Master Theorem” became popular through the textbook of Cormen and colleagues.

2.5.1 The divide-and-conquer recurrence

A divide-and-conquer algorithm splits a problem of size n into a subproblems of size n/b each, solves the subproblems recursively, and combines the results. Its recurrence is:

$$T(n) = a \cdot T(n/b) + f(n), \quad a \geq 1, \quad b > 1$$

- a: the number of subproblems (recursive calls).
- b: the shrinking ratio; each subproblem has size n/b.
- $f(n)$: the work the call itself does, that is, the work of splitting the problem and combining the results.

The recursion tree of this recurrence has a regular structure (Figure 2.16). There is a single node at the root, and it does $f(n)$ work. Each node has a children; level j has a^j nodes, each dealing with a problem of size n/b^j. Since the input is divided by b at every level, the depth of the tree is $\log_b n$. The number of leaves at the last level is

$$a^{\log_b n} = n^{\log_b a}$$

The equality of the two expressions is seen by taking $\log_b$ of both sides: both give $\log_b n \cdot \log_b a$. Since each leaf does constant work, the total work of the leaves is $\Theta(n^{\log_b a})$. This quantity is at the center of the theorem: $f(n)$ is always compared with it.

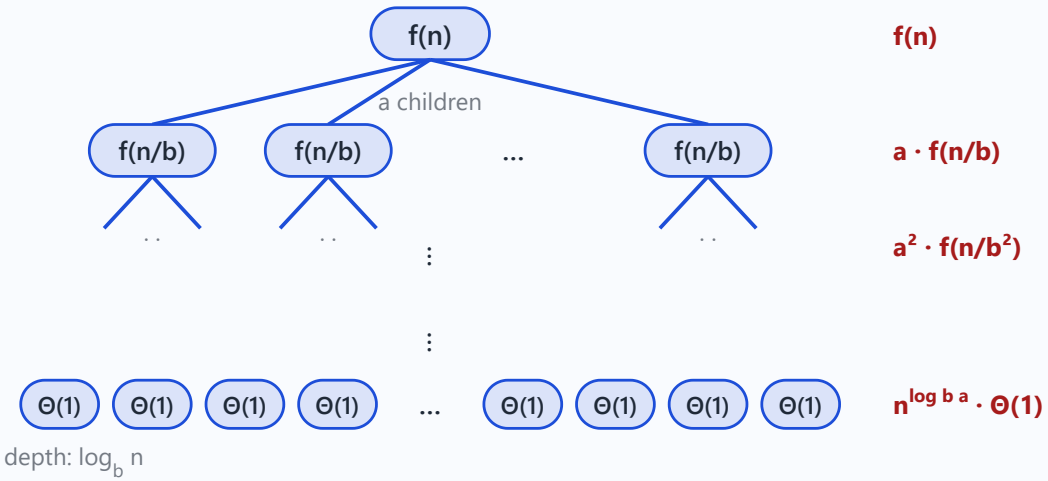


Figure 2.16: The recursion tree of the recurrence $T(n) = aT(n/b) + f(n)$. There is $f(n)$ work at the root, a total of $a \cdot f(n/b)$ at the first level, and $a^2 \cdot f(n/b^2)$ at the second. The depth is $\log_b n$; at the leaves a total of $n^{\log_b a} \cdot \Theta(1)$ work is done.

2.5.2 The three cases

The theorem compares $f(n)$ with the leaf work $n^{\log_b a}$. The comparison has three outcomes: the work near the root dominates, the work at the leaves dominates, or the levels are balanced. Below is the commonly used form of the theorem, written with the polynomial degree c of $f(n)$.

Theorem 2.1 (Master Theorem)

Let $a \geq 1$ and $b > 1$ be constants and $T(n) = aT(n/b) + f(n)$.

1. If $f(n) = O(n^c)$ for some $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$. (The leaves dominate.)
2. If $f(n) = \Theta(n^c)$ for $c = \log_b a$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$. (The levels are balanced.)
3. If $f(n) = \Omega(n^c)$ for some $c > \log_b a$, and a constant $k < 1$ exists such that $a \cdot f(n/b) \leq k \cdot f(n)$ for every sufficiently large n, then $T(n) = \Theta(f(n))$. (The root dominates.)

Case	Condition	Result	Interpretation
1	$f(n) = O(n^c)$, $c < \log_b a$	$\Theta(n^{\log_b a})$	leaves dominate
2	$f(n) = \Theta(n^c)$, $c = \log_b a$	$\Theta(n^{\log_b a} \cdot \log n)$	levels balanced
3	$f(n) = \Omega(n^c)$, $c > \log_b a$ and $a \cdot f(n/b) \leq k \cdot f(n)$, $k < 1$	$\Theta(f(n))$	root dominates

Table 2.16: The three cases of the Master Theorem (c: the polynomial degree of $f(n)$).

The use of different notations in the three cases is no accident. In Case 1 an upper bound on $f(n)$ suffices, because the leaf work is already larger than f and swamps it. In Cases 2 and 3, writing only $f(n) = O(n^c)$ is not enough. For example $f(n) = 1$ is also $O(n^2)$; but the solution of $T(n) = T(n/2) + 1$ cannot be $\Theta(n^2)$. This is why Case 2 requires $f = \Theta(n^c)$, and Case 3 requires $f = \Omega(n^c)$ and in addition the regularity condition. When these conditions hold, the results are given with Θ, that is, as tight bounds.

The **regularity condition** is specific to Case 3: $a \cdot f(n/b) \leq k \cdot f(n)$. It means that the total work of the root's children is no more than a constant fraction of the root's own work. Thus the work grows steadily toward the root. For polynomials such as $f(n) = n^c$ the condition holds automatically: $a(n/b)^c = (a/b^c)\, n^c$, and since $c > \log_b a$, $a/b^c < 1$; one can take $k = a/b^c$.

2.5.3 Why three cases? Level sums

Why the theorem has three cases becomes clear when we look at the level sums of the recursion tree. Let $f(n) = n^c$. Level j has a^j nodes, each doing $(n/b^j)^c$ work. The total work of the level is

$$a^j \cdot (n/b^j)^c = n^c \cdot (a/b^c)^j$$

The level sums form a **geometric series** with ratio $r = a/b^c$. $T(n)$ is the sum of this series from $j = 0$ to $\log_b n$. The behavior of a geometric series depends only on the ratio:

- ***r > 1*: the series grows.** The last term, that is, the leaves, determines the sum. This is Case 1. Example: for $4T(n/2) + n$, $r = 4/2 = 2$; the level sums grow as $n, 2n, 4n, \ldots$ and the result is $\Theta(n^2)$.
- ***r = 1*: the terms are equal.** Every level does the same work; the total is the number of levels times the work per level. This is Case 2. Example: for $2T(n/2) + n$, $r = 2/2 = 1$; each of the $\log n$ levels does n work and the result is $\Theta(n \log n)$.
- ***r < 1*: the series shrinks.** The first term, that is, the root, determines the sum. This is Case 3. Example: for $2T(n/2) + n^2$, $r = 2/4 = 1/2$; the level sums shrink as $n^2, n^2/2, n^2/4, \ldots$, their total does not exceed $2n^2$ and the result is $\Theta(n^2)$.

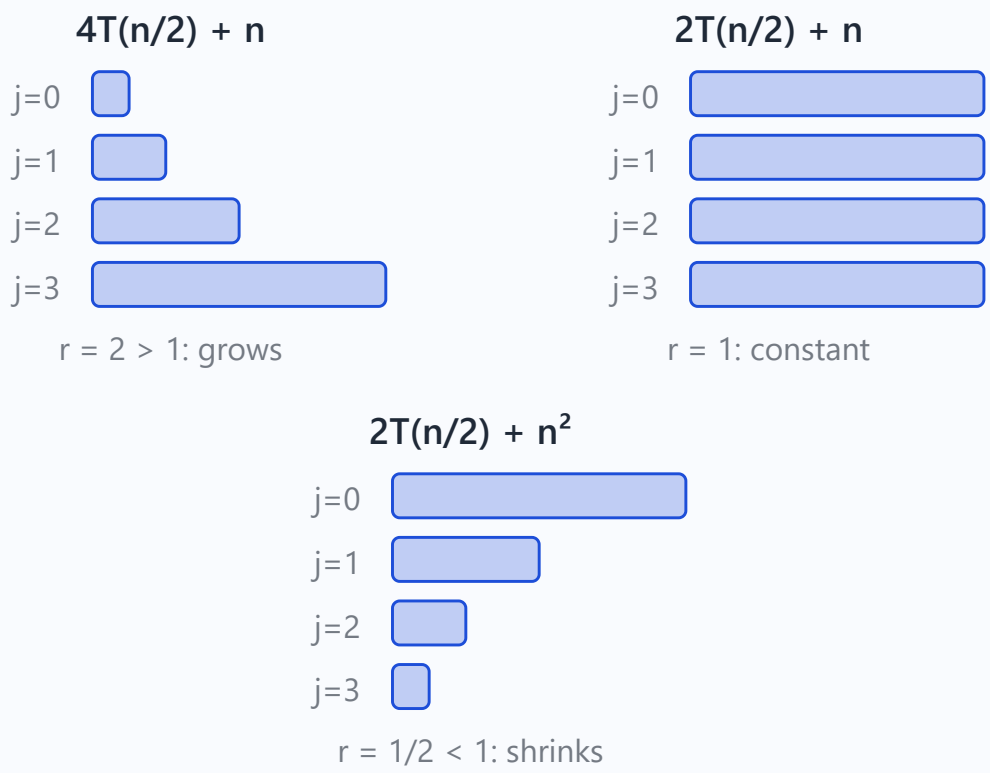


Figure 2.17: The level sums of the three cases; each bar is the total work of one level. Left, $4T(n/2) + n$: the bars grow ($r = 2$), the leaves dominate. Middle, $2T(n/2) + n$: the bars are equal ($r = 1$). Right, $2T(n/2) + n^2$: the bars shrink ($r = 1/2$), the root dominates.

The ratio being greater than 1 is the same as $c < \log_b a$: $r > 1 \Leftrightarrow b^c < a \Leftrightarrow c < \log_b a$. The three cases of the Master Theorem thus correspond to the geometric series growing, staying constant, or shrinking. Instead of memorizing the theorem, it is enough to keep this picture in mind: a is compared with b^c.

These results can also be checked with a small computation. When the recurrences are computed directly with $T(1) = 1$, for $n = 16$ the recurrence $2T(n/2) + n$ gives 80, which is the value of $n \log_2 n + n$. The recurrence $2T(n/2) + n^2$ gives 496, that is, $2n^2 - n$; the $n^2 = 256$ at the root is more than half of the total.

2.5.4 Example: 2T(n/2) + n

Example 2.5

Solve the recurrence $T(n) = 2T(n/2) + \Theta(n)$ with the Master Theorem.

1. **Reading the parameters.** Two subproblems ($a = 2$), each of half the size ($b = 2$), linear merging work ($f(n) = \Theta(n)$).
2. **The exponent of the leaf work.** $\log_b a = \log_2 2 = 1$; the leaf work is $n^1 = n$.
3. **The degree of $f(n)$.** $f(n) = \Theta(n^1)$, that is, $c = 1$.
4. **Comparison.** $c = \log_b a = 1$: the exponents are equal, and each level does the same amount (n) of work. Case 2.
5. **Result.** $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$.

This recurrence turns up in several of the best-known algorithms.

- **Merge sort:** the array is split in two, the two halves are sorted separately, and the sorted halves are merged in $\Theta(n)$ time.
- **f5**: one of the example functions of this chapter; it calls $f1(n)$ (linear work) and calls itself twice with $n/2$.
- **Quicksort:** if the pivot splits the array into two equal parts every time, the recurrence is the same.

2.5.5 More examples

Table 2.17 applies the theorem to frequently encountered recurrences. In each row $\log_b a$ is computed first, then the degree c of $f(n)$ is compared with it. The logarithm values were computed with Python.

Recurrence	a, b	$\log_b a$	c	Case	Result	Algorithm
T(n/2) + 1	1, 2	0	0	2	Θ(log n)	binary search
2T(n/2) + 1	2, 2	1	0	1	Θ(n)	f4, tree traversal
2T(n/2) + n	2, 2	1	1	2	Θ(n log n)	f5, merge sort
3T(n/2) + n	3, 2	≈ 1.585	1	1	$\Theta(n^{1.585})$	Karatsuba multiplication
4T(n/2) + n	4, 2	2	1	1	$\Theta(n^2)$	—
7T(n/2) + n^2	7, 2	≈ 2.807	2	1	$\Theta(n^{2.807})$	Strassen's algorithm
2T(n/2) + n^2	2, 2	1	2	3	$\Theta(n^2)$	—
T(n/2) + n	1, 2	0	1	3	Θ(n)	quickselect (average)

Table 2.17: The Master Theorem applied to example recurrences.

The first row of the table is binary search: at each step a single subproblem ($a = 1$) of half the size is solved and constant work is done. Since $\log_2 1 = 0$ and $c = 0$, Case 2 applies: $\Theta(n^0 \cdot \log n) = \Theta(\log n)$. It may seem surprising that a constant $f(n)$ falls under Case 2; but the tree has a single node at each level, and each level does the same constant work.

The Karatsuba and Strassen rows show the power of divide and conquer. Multiplying two n-digit numbers by the method taught in school takes $\Theta(n^2)$. Splitting the numbers in two and making four half-size multiplications does not change this: $4T(n/2) + n = \Theta(n^2)$. In 1962 Anatoly Karatsuba showed that one of the four products can be computed from the others. With three multiplications the recurrence becomes $3T(n/2) + n$ and the result drops to $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$. In 1969 Volker Strassen applied the same idea to matrix multiplication: seven subproducts instead of eight. The recurrence $8T(n/2) + n^2$ gives $\Theta(n^3)$; $7T(n/2) + n^2$ gives $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$. In both examples removing a single subcall lowered the exponent. Because in Case 1 only a and b determine the result, every reduction in a turns directly into a gain.

In the Case 3 rows the regularity condition must also be checked. For $2T(n/2) + n^2$, $2 \cdot (n/2)^2 = n^2/2 \le \frac{1}{2} \cdot n^2$, so the condition holds with $k = \frac{1}{2}$. For $T(n/2) + n$, $1 \cdot (n/2) \le \frac{1}{2} \cdot n$, again with $k = \frac{1}{2}$.

Example 2.6

Solve three recurrences: (a) $T(n) = 3T(n/3) + n$, (b) $T(n) = 9T(n/3) + n$, (c) $T(n) = T(n/2) + n$.

1. **(a)** $a = 3$, $b = 3$, $\log_3 3 = 1$; $f(n) = n$, $c = 1$. The exponents are equal: Case 2, $T(n) = \Theta(n \log n)$. In the tree each level splits into three parts but the total work is still n; the number of levels is $\log_3 n$. For $n = 729$ direct computation gives $5103 = 729 \cdot 6 + 729$.
2. **(b)** $a = 9$, $b = 3$, $\log_3 9 = 2$; $c = 1 < 2$. Case 1, $T(n) = \Theta(n^2)$. The work of each level is three times that of the previous one ($r = 9/3 = 3$); the leaves dominate.
3. **(c)** $a = 1$, $b = 2$, $\log_2 1 = 0$; $c = 1 > 0$. Regularity: $1 \cdot (n/2) \le \frac{1}{2} \cdot n$, $k = \frac{1}{2} < 1$. Case 3, $T(n) = \Theta(n)$. The total is $n + n/2 + n/4 + \ldots < 2n$.

2.5.6 Recurrences the theorem does not cover

The Master Theorem is powerful, but it applies only to recurrences of a particular form. For recurrences outside this form another method is needed.

- **Subtractive recurrences.** $T(n) = 2T(n - 1) + c$ (f3) or $T(n) = T(n - 1) + n$: the subproblem is $n - 1$, not n/b. Unrolling is used.
- **A non-polynomial gap.** $T(n) = 2T(n/2) + n / \log n$: $n^{\log_2 2} = n$ is compared with $f(n) = n / \log n$. $f(n)$ is smaller than n, but only by a log factor, not by a polynomial factor like n^{ε}. So neither the condition of Case 1 nor that of Case 2 holds. (The solution of this recurrence is $\Theta(n \log \log n)$ and is found with a recursion tree.)

- **Unequal splits.** $T(n) = T(n/3) + T(2n/3) + n$: the subproblems have different sizes. In the recursion tree the work at each level is at most n and the longest branch has depth $\log_{3/2} n$; the result is $\Theta(n \log n)$. Recurrences of this kind are generally solved by the Akra–Bazzi method.
- **Non-constant a or $a < 1$.** $T(n) = 2^n\, T(n/2) + n$ or $T(n) = \tfrac{1}{2}\, T(n/2) + n$: the theorem's assumption that $a \geq 1$ is constant is violated.

A commonly used extension of the basic form carries Case 2 over to logarithmic factors: if $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ and $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$. For example, for $T(n) = 2T(n/2) + n \log n$, $k = 1$ and the result is $\Theta(n \log^2 n)$. This extension also solves some recurrences that the basic form calls "not applicable."

> **Common mistake**
>
> In Case 3, checking only the condition $c > \log_b a$ and forgetting regularity. For polynomials the condition holds automatically, so the mistake usually does not change the result; but in examples where $f(n)$ oscillates the condition fails. In the recurrence $T(n) = T(n/2) + n(2 - \cos n)$, $c = 1 > 0$; but for the values of n where $\cos n$ is near 1 and $\cos(n/2)$ near -1, the ratio $f(n/2)/f(n)$ approaches 1.5 and no $k < 1$ satisfies the condition. When using Case 3, the inequality $a \cdot f(n/b) \leq k \cdot f(n)$ should be written out explicitly.

2.6 Application: f4 and f5

This section solves two recursive functions with all three methods: tracing, the recursion tree, and the Master Theorem. f4 differs from f3 only in the argument of the recursive calls (`n / 2` instead of `n - 1`); but this change brings the time down from exponential to linear. f5 adds linear work to f4 at each call and gives the recurrence of merge sort.

2.6.1 f4: two calls, n/2

```
public int f4(int n) {
  if (n <= 1) {
    return 1;
  }
  return f4(n / 2) + f4(n / 2);
}
```

- **Base case:** returns 1 when $n \leq 1$; $O(1)$.
- **Recursive case:** when $n > 1$, `f4(n / 2)` is called twice and the results are added. The addition is constant work.

$$T(n) = 2T(n/2) + O(1), \quad T(1) = O(1)$$

In Java `n / 2` is integer division; the fractional part is discarded. In the analysis n is assumed to be a power of two, so that every division is exact. This assumption does not change the growth class.

The only difference from f3 is that the input is $n/2$ instead of $n - 1$. The number of calls still doubles at every level; but since the input halves at every level, the tree is only $\log_2 n$ levels deep. For $n = 8$ the levels have 1, 2, 4, 8 calls and the inputs are 8, 4, 2, 1. In f3 a tree starting from 8 would descend 8 levels; here it ends at 4 levels.

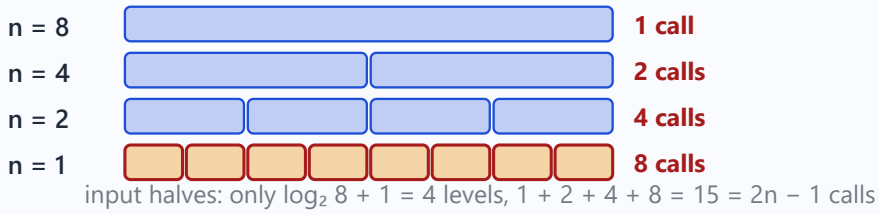


Figure 2.18: The levels of `f4(8)`. The input halves at each level (8, 4, 2, 1) and the number of calls doubles (1, 2, 4, 8). There are only $\log_2 8 + 1 = 4$ levels; in all $1 + 2 + 4 + 8 = 15 = 2n - 1$ calls are made.

Two calls do not always mean exponential time. What decides is how much the input shrinks. If the input decreases by one, the tree descends to depth n and reaches 2^n leaves. If the input halves, the tree stays at depth $\log_2 n$ and reaches $2^{\log_2 n} = n$ leaves.

2.6.2 The recursion tree of f4

In the recursion-tree method each call is drawn as a node and the work it does is written next to it. Then the work of each level is added up, and finally the level sums are added. For `f4(8)`:

1. **Root:** `f4(8)` does only constant work (c) and starts two calls. The work of level 0 is c.
2. **Level 1:** two calls `f4(4)`; total $2c$.
3. **Level 2:** four calls `f4(2)`; total $4c$. The work per level doubles at each step.
4. **Leaves:** eight `f4(1)`; total $8c = n \cdot c$.

The total is $c(1 + 2 + 4 + 8) = 15c = (2n - 1)c$; that is, $\Theta(n)$. The level sums grow geometrically and the last term, the work of the leaves, determines the sum. Counted in Java, `f4(16)` makes 31 calls and `f4(32)` makes 63; both equal $2n - 1$. The return value of the function also equals the number of leaves: `f4(n) = n`.

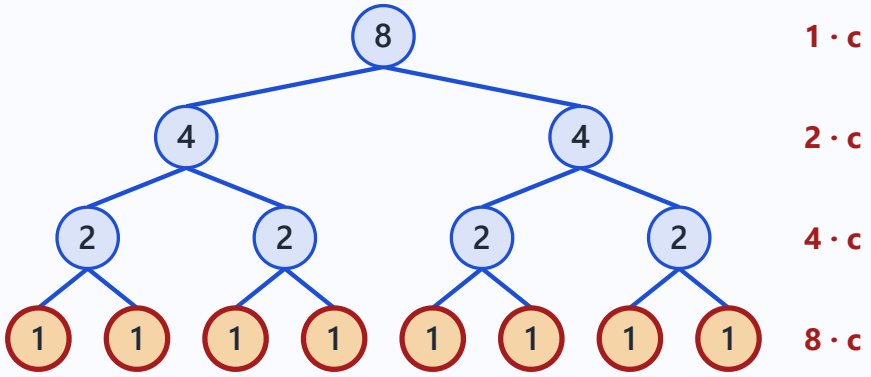


Figure 2.19: The recursion tree of `f4(8)`. The number in each node is its input. The column on the right gives the level sums: *c, 2c, 4c, 8c*. The eight leaves, drawn in orange, do more than half of the total work.

If n is not a power of two, integer division comes into play. `f4(10)` first calls two `f4(5)`, each `f4(5)` calls two `f4(2)`, and each `f4(2)` calls two `f4(1)`. The number of calls is 15 and the return value is 8. The result is as if it had worked for the largest power of two (8) not exceeding n; the growth class is still linear.

2.6.3 Solving f4 with the Master Theorem

Example 2.7

1. **Matching.** The recurrence $T(n) = 2T(n/2) + O(1)$ is compared with the general form: $a = 2$, $b = 2$, $f(n) = O(1)$.
2. **Leaf work.** $\log_b a = \log_2 2 = 1$; the leaf work is $n^1 = n$.
3. **Comparison.** $f(n) = O(1) = O(n^0)$, that is, $c = 0$. $0 < 1$: Case 1.
4. **Result.** $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

The Master Theorem gives the result of the recursion tree without computing the sum. In the tree we counted $2n - 1$ nodes; the theorem merely says “the leaves dominate,” and since the number of leaves is n, it arrives at $\Theta(n)$.

The memory of f4 is a different question. Because the calls run in sequence, the stack holds only one branch at a time: `f4(n)`, `f4(n/2)`, …, `f4(1)`. The length of this branch is $\log_2 n + 1$; while `f4(16)` runs, the stack holds at most 5 frames. The time is $\Theta(n)$ and the auxiliary space is $\Theta(\log n)$.

Common mistake

Saying of f4, "the input halves at every step, so it is *O(log n)*." This reasoning is correct for binary search, which makes a single call. f4 makes two calls, and both branches descend all the way. *log n* is only the stack depth, that is, it gives the memory; the time is *Θ(n)*.

2.6.4 f5: linear work and two calls

```
public int f5(int n) {
  if (n <= 1) {
    return 1;
  }
  return f1(n) + f5(n / 2) + f5(n / 2);
}
```

- **Base case:** returns 1 when $n \le 1$; *O(1)*.
- **Recursive case:** when $n > 1$, `f1(n)` is called first. f1 is a loop that runs n rounds and does *Θ(n)* work. Then `f5(n / 2)` is called twice.

$$T(n) = 2T(n/2) + f1(n) = 2T(n/2) + \Theta(n)$$

Its only difference from f4 is the linear work done at each call. The recurrence is the same as that of merge sort. A common mistake here is to consider the two parts separately and add them: "f1 is linear, the recursion is logarithmic, the total is *O(n + log n) = O(n)*." This is wrong. f1 is called not only at the root but at every node of the tree; each node does as much work as its input size. The sum of these works must be found.

2.6.5 Code trace: f5(4)

Step	Event	Value
1	`f5(4)` is called; 4 > 1	—
2	`f1(4)` : a loop of 4 rounds, 4 units of work	4
3	the left `f5(2)` is called; `f1(2)` : 2 units of work	2
4	two `f5(1)` : base case	1, 1
5	`f5(2) = f1(2) + f5(1) + f5(1) = 2 + 1 + 1`	4
6	the right `f5(2)` : the same work is done again from scratch	4
7	`f5(4) = f1(4) + f5(2) + f5(2) = 4 + 4 + 4`	12

Table 2.18: Trace of `f5(4)` (from the output of a copy run in Java).

The result is 12. This number also shows how the work is distributed over the levels. At the root `f1(4)` does 4 units, and at the first level the two `f1(2)` do 4 units in all. Each of the four leaves returns 1; total 4. The work of each level is $n = 4$. There are $\log_2 4 = 2$ inner levels and one leaf level: $2 \cdot 4 + 4 = 12 = n \log_2 n + n$.

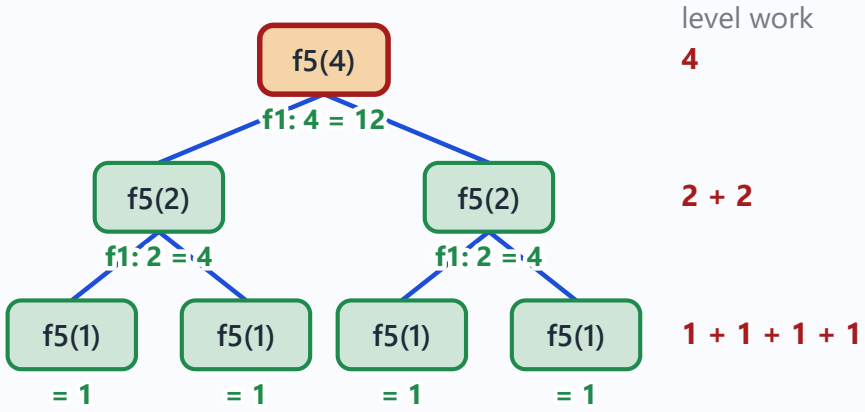


Figure 2.20: The completed tree of `f5(4)`. Below each node is written the work done by `f1` and the total the node returns: at the root `f1` does 4 work and `f5(4)` returns 12; at the `f5(2)` nodes `f1` does 2 work and each returns 4. The column on the right gives the level work: 4, 2 + 2, 1 + 1 + 1 + 1.

2.6.6 The recursion tree of f5 and the Master Theorem

In the tree of `f5(8)` the pattern is easier to see (Figure 2.21).

1. **Root:** inside `f5(8)`, `f1(8)` takes 8 steps.
2. **Level 1:** two `f5(4)`, each with 4 steps by `f1(4)`; total 8.
3. **Level 2:** four `f5(2)`, each with 2 steps; total again 8.
4. **Leaves:** eight `f5(1)`, each 1; total 8.

The total is $3 \cdot 8 + 8 = 32 = n \log_2 n + n$, and in Java `f5(8)` really returns 32. Each level does n work and there are $\log_2 n$ inner levels; the result is $\Theta(n \log n)$. The formula holds on larger inputs too: `f5(16) = 80 = 16 · 4 + 16`, `f5(32) = 192 = 32 · 5 + 32`.

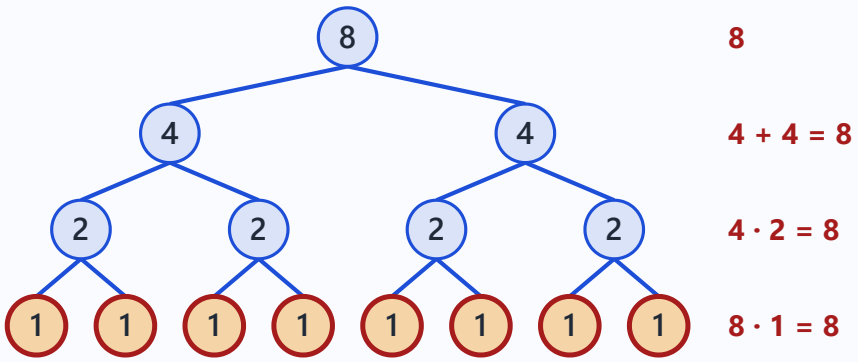


Figure 2.21: The recursion tree of `f5(8)`. It has the same shape as the tree of f4, but each node does as much work as its input size. The level sums are equal: $8 = 4 + 4 = 4 \cdot 2 = 8 \cdot 1$.

The solution with the Master Theorem follows the same steps as for f4; only $f(n)$ has changed:

1. $T(n) = 2T(n/2) + \Theta(n)$: $a = 2$, $b = 2$, $f(n) = \Theta(n)$.
2. $\log_b a = 1$, $c = 1$. Exponents equal: Case 2.
3. $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$.

	f4	f5
Calls	2 × n/2	2 × n/2
Extra work f(n)	O(1)	Θ(n)
c versus $\log_b a$	0 < 1	1 = 1
Case	1	2
Level sums	c, 2c, 4c, … (growing)	n, n, n, … (equal)
Result	Θ(n)	Θ(n log n)

Table 2.19: Comparison of f4 and f5.

The table sums up what the f3–f4–f5 triple teaches. All three functions call themselves twice. The difference between f3 and f4 is the way the input shrinks ($n - 1$ versus $n/2$), and it brings the result down from $\Theta(2^n)$ to $\Theta(n)$. The difference between f4 and f5 is the extra work (constant versus linear), and it raises the result from $\Theta(n)$ to $\Theta(n \log n)$.

2.7 Comparison and amortized analysis

2.7.1 f1–f6: measured step counts

The classes of the six functions found by analysis can be checked by running the functions. Table 2.20 gives the measured step count of each function for $n = 16$ and $n = 32$. For f1, f2, f5, and f6 the return value is used, and for f3 and f4 the call count. The values were measured in Java.

Function	Recurrence / sum	n = 16	n = 32	Ratio	Class
f1 (return)	$\Sigma\ 1 = n$	16	32	× 2	$\Theta(n)$
f2 (return)	$\Sigma\ i^2$	1,240	10,416	× 8.4	$\Theta(n^3)$
f3 (calls)	$2T(n-1) + c$	65,535	4,294,967,295	× 65,537	$\Theta(2^n)$
f4 (calls)	$2T(n/2) + c$	31	63	× 2	$\Theta(n)$
f5 (return)	$2T(n/2) + n$	80	192	× 2.4	$\Theta(n \log n)$
f6 (return)	$i \leftarrow 2^i$	4	5	+ 1	$\Theta(\log^* n)$ (mathematical model; in Java, $n \leq 65536$)

Table 2.20: Measured step counts of f1–f6. The ratio column gives how the step count changes when *n* doubles (the counter is `long`; f3(32) ≈ $4.3 \cdot 10^9$ calls).

The ratio column is a practical test that gives away the growth class. When *n* doubles, linear functions (f1, f4) double. The linearithmic f5 grows by a factor slightly above 2, 2.4; the excess comes from the factor *log n* rising from 4 to 5. The cubic f2 grows by about a factor of 8. The ratio of the exponential f3 is $2^{16} + 1 = 65{,}537$; when the input doubles, the step count roughly squares. f6 increases by just one step.

This test is an experimental way of verifying an analysis, and it is also used when the complexity of a piece of code is unknown. A program is run with different values of *n*, the time or step count is measured, and the ratio of consecutive measurements is examined. Sedgewick and Wayne call this method the “doubling experiment.” The interactive step counter in the slides performs the same experiment for every *n* from 1 to 40 and shows the results as bars on a logarithmic scale. There, the number of calls is also counted for f5, so the values are slightly larger than those in the table; the ratios, however, show the same class.

2.7.2 Amortized analysis

In some data structures most operations are cheap and an occasional few are expensive. In such a structure, looking at the worst case of a single operation overstates the total cost. Amortized analysis looks at the whole sequence of operations.

Definition 2.7

Amortized analysis finds the cost per operation by dividing the worst-case total cost of a sequence of operations by the number of operations. The value found is called the **amortized cost**.

The classic example is the **dynamic array**, such as Java's `ArrayList`. When the array is full, its capacity is doubled: a new array of twice the size is allocated and all the elements are copied into it. This insertion is expensive. Insertions made while there is room are a single write and cost 1 unit. The code below spells out this logic; the cost is the sum of the number of writes and the number of elements copied.

```
int[] data = new int[1];
int size = 0;
long cost = 0;

void add(int x) {
  if (size == data.length) {                // array is full
    int[] bigger = new int[2 * data.length]; // double the capacity
    for (int i = 0; i < size; i++) {
      bigger[i] = data[i];                  // copy: 1 unit
      cost++;
    }
    data = bigger;
  }
  data[size++] = x;                         // write: 1 unit
  cost++;
}
```

Example 2.8

Nine elements are added to an empty dynamic array of capacity 1. The cost of each insertion (1 write + the number of copies) is:

1. Insertion 1: there is room, 1 write. Cost 1.
2. Insertion 2: the array is full; capacity 1 → 2, 1 copy + 1 write. Cost 2.
3. Insertion 3: full; capacity 2 → 4, 2 copies + 1 write. Cost 3.
4. Insertion 4: there is room. Cost 1.
5. Insertion 5: full; capacity 4 → 8, 4 copies + 1 write. Cost 5.
6. Insertions 6, 7, and 8: there is room. Each costs 1.
7. Insertion 9: full; capacity 8 → 16, 8 copies + 1 write. Cost 9.

The total is $1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 = 24$ units; the average per insertion is $24/9 \approx 2.67$. The worst cost of a single insertion is 9, but for the whole array $24 < 3 \cdot 9 = 27$ units were spent.

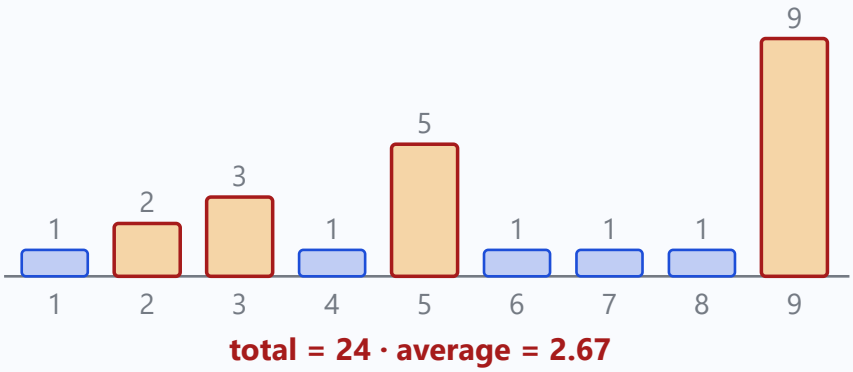


Figure 2.22: Nine insertions into a dynamic array whose capacity starts at 1 and doubles. The orange bars are insertions that include a capacity increase (copying): insertions 2, 3, 5, and 9. Total 24, average 2.67.

Theorem 2.2 (Amortized cost of doubling)

The total cost of n insertions into a dynamic array whose capacity starts at 1 and is doubled when full is less than $3n$. The amortized cost per insertion is $\Theta(1)$.

Proof. Capacity increases occur when the array is full with 1, 2, 4, ..., 2^k elements, where 2^k is the largest power of two smaller than n. The total number of copies is a geometric series: $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1 < 2n$. Each insertion also makes one write; that is n units. The total is $< 2n + n = 3n$. Divided by n, the cost per insertion comes out below 3, that is, constant. □

The theorem rests on the moments when copying is expensive being rare. A capacity increase copies m elements, but after it m more insertions are done cheaply. The cost of the expensive operation is "spread" over the cheap operations that follow it. In a count done with Python, the total cost of 1000 insertions comes out to 2023; this is below $3n = 3000$.

Why doubling matters becomes clear when it is compared with growing by a fixed amount. If the capacity were increased by just 1 when the array is full, every insertion would copy the whole array. The total cost of n insertions would be $1 + 2 + \ldots + n = n(n + 1)/2$, that is, $\Theta(n^2)$; for $n = 1000$, 500,500 units. Even if the capacity were increased by a fixed k, the result would not change; only the constant factor would shrink.

Amortized analysis differs from average-case analysis. The average case rests on a probability distribution over the inputs and gives the "expected" time. In amortized analysis there is no probability: a guarantee is given for every sequence of operations, even the worst one. In a dynamic array a single insertion can take $\Theta(n)$; but whatever the order of insertions, the total of n insertions does not exceed $3n$. There are three standard methods of finding the amortized cost: the aggregate method (used here), the accounting method, and the potential method.

Common mistake

Reading amortized $\Theta(1)$ as "every insertion takes constant time." The amortized bound says nothing about the time of a single operation; the 9th insertion took 9 units. In real-time systems where every operation must be short, an amortized bound is not enough; a worst-case bound is needed.

2.8 Common mistakes

The most important of the mistakes given in the boxes throughout the chapter are collected here:

- **Equating Big O with the worst case.** O is an upper bound, while the worst case is a kind of input. "The best case is $O(1)$" is a valid sentence.
- **Writing constants and small terms.** One does not write $O(3n + 3)$ or $O(n^2/3 + n)$; $O(n)$ and $O(n^2)$ suffice.
- **The rule "nested loops = n^2."** The bound of the inner loop determines the result. In f2 the bound is i^2, so the result is $\Theta(n^3)$; a loop advancing with `i *= 2` runs $\log n$ rounds.
- **The rule "two calls = exponential."** f3 ($n - 1$) is $\Theta(2^n)$, f4 ($n/2$) is $\Theta(n)$. The way the input shrinks determines the result.
- **Applying the Master Theorem to every recurrence.** The theorem does not hold for recurrences that contain $T(n - 1)$, split unevenly, or have a non-polynomial gap such as $f(n) = n / \log n$.
- **Forgetting regularity in Case 3.** $c > \log_b a$ is not enough; $a \cdot f(n/b) \le k \cdot f(n)$, $k < 1$ must hold as well.
- **Confusing the memory of recursion with the call count.** Memory is determined by the number of frames open at the same time, that is, the depth: $\Theta(n)$ for f3, $\Theta(\log n)$ for f4.
- **Confusing a shift with a multiplication.** `i = 1 << i` is 2^i, not `2 * i`. The tight bound for f6 is $\Theta(\log^* n)$.

2.9 Summary

- **Time complexity** $T(n)$ gives how the number of basic operations grows with the input size n. **Space complexity** $S(n)$ gives the memory used in addition to the input; in recursion the depth of the call stack counts toward this memory.
- Timing depends on the machine and gives no prediction for large inputs. In the **RAM model** every basic operation counts as one unit; linear search takes $3n + 3$ steps in the worst case and is $\Theta(n)$.
- The **best**, **worst**, and **average case** concern different inputs of the same size. In linear search they make 1, n, and $(n + 1)/2$ comparisons respectively.

- $f = O(g)$ is an upper bound, $f = \Omega(g)$ a lower bound, and $f = \Theta(g)$ a tight bound. Bounds are proved by finding the constants c and n_0; for $3n^2 + 5n + 2$, $c_1 = 3$, $c_2 = 4$, $n_0 = 6$. The case (which input) and the bound (O, Ω, Θ) are separate notions.
- Rules of calculation: constant factors and lower-order terms are dropped; for sequential code the larger remains, for nested code the round counts are multiplied or added; the base of a logarithm is irrelevant, while the base of an exponential matters. The order of growth: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$.
- In **loop analysis** the inner round count is written for each outer round and added up. f1: $\Sigma\ 1 = n$, $\Theta(n)$. f2: $\Sigma\ i^2 = (n - 1)n(2n - 1)/6$, $\Theta(n^3)$. A multiplicative loop is $\Theta(\log n)$; a multiplicative loop with `j < i` inside it is $\Theta(n)$ because of the geometric series.
- In f6, `i = 1 << i`, that is, $i \leftarrow 2^i$; the round count is $\log^* n + 1$. $O(\log n)$ is correct but not tight; the tight bound is $\Theta(\log^* n)$. In Java, for $n > 65{,}536$ the shift overflows and the loop does not terminate.
- A **recurrence relation** is built from the number of calls, the way the input shrinks, and the extra work. f3: $T(n) = 2T(n - 1) + c$; by unrolling, $(2^n - 1)c = \Theta(2^n)$. Its memory is only $\Theta(n)$.
- The **Master Theorem** compares $f(n)$ in $T(n) = aT(n/b) + f(n)$ with the leaf work $n^{\log_b a}$: leaves dominate (Case 1), levels balanced (Case 2, an extra factor $\log n$), root dominates (Case 3, with the regularity condition). The three cases correspond to the geometric series with ratio a/b^c growing, staying constant, or shrinking.
- The theorem does not apply to subtractive recurrences, recurrences with unequal splits, a non-polynomial gap in $f(n)$, or a non-constant a.
- f4: $2T(n/2) + 1$, Case 1, $\Theta(n)$; $2n - 1$ calls, $\Theta(\log n)$ memory. f5: $2T(n/2) + n$, Case 2, $\Theta(n \log n)$; $f5(n) = n \log_2 n + n$.
- **Amortized analysis** divides the total cost of a sequence of operations by the number of operations. In a dynamic array that doubles, the total of n insertions is less than $3n$; insertion is amortized $\Theta(1)$. Unlike the average case, it uses no probability.

Function	Structure	Analysis	Time	Auxiliary space
f1	single loop	$\Sigma\ 1 = n$	$\Theta(n)$	$\Theta(1)$
f2	inner loop `j < i * i`	$\Sigma\ i^2$	$\Theta(n^3)$	$\Theta(1)$
f3	two calls, n − 1	$2T(n - 1) + c$, unrolling	$\Theta(2^n)$	$\Theta(n)$
f4	two calls, n/2	$2T(n/2) + 1$, Case 1	$\Theta(n)$	$\Theta(\log n)$
f5	f1(n) + two calls, n/2	$2T(n/2) + n$, Case 2	$\Theta(n \log n)$	$\Theta(\log n)$
f6	`i = 1 << i`	$i \leftarrow 2^i$	$\Theta(\log^* n)$ (mathematical model; in Java, $n \le 65536$)	$\Theta(1)$

Table 2.21: Summary of the six example functions.

2.10 Key terms

These are the terms introduced in this chapter.

time complexity space complexity complexity class growth rate
random-access machine (RAM model) best case worst case average case asymptotic notation
Big O notation Big Omega (Ω) Big Theta (Θ) upper bound lower bound tight bound constant time
logarithmic linear linearithmic quadratic cubic exponential factorial polynomial time
iterated logarithm recursion base case call stack recurrence relation recursion tree
divide and conquer Master Theorem regularity condition amortized analysis dynamic array

2.11 Test Yourself

1. For the bound $5n^3 + 2n + 8 = O(n^3)$, $c = 6$ is chosen. What is the smallest n_0 for which the inequality $5n^3 + 2n + 8 \le 6n^3$ holds for every $n \ge n_0$?

   Answer

   $5n^3 + 2n + 8 \le 6n^3 \Leftrightarrow n^3 \ge 2n + 8$. At $n = 2$, $8 < 12$ and the inequality fails; at $n = 3$, $27 \ge 14$ and it holds. As n increases, n^3 grows faster than $2n + 8$, so the inequality always holds from there on: $n_0 = 3$. Check: at $n = 3$, $149 \le 162$.

2. How many times does the fragment `for (i = 0; i < n; i++) for (j = i; j < n; j++) x++;` execute `x++` for $n = 6$? What is the growth class?

   Answer

   In round i the inner loop runs $n - i$ times: $6 + 5 + 4 + 3 + 2 + 1 = 21$. In general $n(n + 1)/2$, that is, $\Theta(n^2)$. That the inner loop depends on the outer counter halves the total but does not change the class.

3. For $n = 64$, how many rounds does the loop `for (int i = 1; i * i <= n; i++) x++;` run? What is the growth class?

   Answer

   The loop continues while $i^2 \le n$, that is, $i \le \sqrt{n}$. For $n = 64$, $i = 1, \ldots, 8$: 8 rounds. The class is $\Theta(\sqrt{n})$; primality testing by trial division uses this pattern.

4. Which of the statements $2^{n+1} = O(2^n)$ and $2^{2n} = O(2^n)$ is true?

Answer

The first is true: $2^{n+1} = 2 \cdot 2^n$; $c = 2$, $n_0 = 1$ suffice. The second is false: $2^{2n} = 4^n$ and the ratio $4^n/2^n = 2^n$ goes to infinity; no constant c can close this gap. A constant added to the exponent only brings a constant factor; a constant multiplying the exponent changes the base.

5. What is the solution of the recurrence $T(n) = 2T(n/4) + \sqrt{n}$ by the Master Theorem?

Answer

$a = 2$, $b = 4$, $\log_4 2 = 1/2$. $f(n) = \sqrt{n} = n^{1/2}$, $c = 1/2$. Exponents equal: Case 2, $T(n) = \Theta(\sqrt{n} \log n)$. Check: with $T(1) = 1$, $T(256) = 80 = 16 \cdot 4 + 16$.

6. What is the solution of the recurrence $T(n) = 5T(n/2) + n^2$? Do the level sums grow or shrink?

Answer

$\log_2 5 \approx 2.32 > c = 2$: Case 1, $T(n) = \Theta(n^{\log_2 5}) \approx \Theta(n^{2.32})$. The ratio is $r = 5/2^2 = 1.25 > 1$; the level sums grow by a factor of 1.25 at each level, and the leaves dominate.

7. What is the closed form of the recurrence $T(n) = T(n - 1) + 1$, $T(1) = 1$, found by unrolling? Why does the Master Theorem not apply?

Answer

$T(n) = T(n - 1) + 1 = T(n - 2) + 2 = \ldots = T(n - k) + k$. At $k = n - 1$, $T(n) = T(1) + n - 1 = n$: $\Theta(n)$. Because the subproblem is $n - 1$, not n/b, the recurrence does not fit the form of the Master Theorem. If the input of a recursion with a single call decreases by one at each step, the time is n times the work of each call.

8. How many calls does `f4(10)` make and what does it return? (In Java `n / 2` is integer division.)

Answer

`f4(10)` calls two `f4(5)`, each `f4(5)` calls two `f4(2)`, each `f4(2)` calls two `f4(1)`. The calls number $1 + 2 + 4 + 8 = 15$. Eight leaves return 1; the result is 8. The tree has the same shape as the tree for input 8.

9. Seventeen elements are added to an empty dynamic array of capacity 1 that doubles when full.

How many copies are made in all, what is the total cost, and what is the final capacity?

> **Answer**
>
> Capacity increases occur at insertions 2, 3, 5, 9, and 17; the copies are $1 + 2 + 4 + 8 + 16 = 31$. With 17 writes the total is 48 units, which stays below $3 \cdot 17 = 51$. The final capacity is 32.

10. What are the number of `x++` executions and the growth class of the fragment `for (i = 0; i < n; i++) for (j = 1; j < n; j *= 2) x++;` for $n = 10$?

> **Answer**
>
> The inner loop is independent of the outer counter; it runs 4 rounds, at `j` = 1, 2, 4, 8. The outer loop runs 10 rounds; total $10 \cdot 4 = 40$. In general $n \cdot \lceil \log_2 n \rceil$, that is, $\Theta(n \log n)$.

2.12 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 3 and 4: give the formal definitions of asymptotic notation, the substitution and recursion-tree methods, and the proof of the Master Theorem; they also touch on the Akra–Bazzi method.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 1.4: combines measuring running time by experiments, the doubling experiment, and mathematical models, with Java examples.
- J. Kleinberg and É. Tardos, *Algorithm Design* (2005), Chapters 2 and 5: discuss the notion of efficiency and the polynomial-time criterion; solve divide-and-conquer recurrences clearly with recursion trees.
- D. E. Knuth, “Big Omicron and Big Omega and Big Theta” (1976): a short article on the history of the O, Ω, and Θ notations and why their present definitions were chosen.

2.13 References

1. J. L. Bentley, D. Haken, J. B. Saxe, “A General Method for Solving Divide-and-Conquer Recurrences”, *SIGACT News* 12(3), 1980.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
3. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005.
4. D. E. Knuth, “Big Omicron and Big Omega and Big Theta”, *SIGACT News* 8(2), 1976.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

PART I · CHAPTER 3

Sorting Algorithms

Sorting is one of the oldest and most thoroughly studied problems in computer science. The problem is simple: arrange the elements of an array from smallest to largest. Yet its solutions display nearly all the basic ideas of algorithm design: brute force, the incremental approach, divide and conquer, random choice, exploiting a data structure, and counting without comparing. These notes begin with the criteria used to evaluate a sorting algorithm. They then cover bubble, insertion, selection, and Shell sort, followed by merge sort, quicksort, and heap sort. Next come the lower bound for comparison sorting and the counting, radix, and bucket sorts that get around it. The last sections treat Timsort and an empirical comparison of the algorithms.

LEARNING OUTCOMES

Concepts
Evaluate a sorting algorithm by time, extra memory, stability, in-place operation, adaptivity, and whether it is comparison-based.

Simple algorithms
Trace bubble, insertion, selection, and Shell sort step by step on an array, write them in Java, and compute their comparison and move counts.

Divide and conquer, heaps
Explain how merge sort, quicksort, and heap sort work; justify their $O(n \log n)$ running times and worst-case behavior.

Linear time and selection
Prove the comparison lower bound; explain why counting, radix, and bucket sort can beat it; and choose a suitable algorithm for a given problem.

3.1 The sorting problem and evaluation criteria

To say whether an algorithm is good, we must first define the problem precisely and then decide what "good" is measuring. This section defines the sorting problem, briefly introduces the algorithms of the chapter, and explains the criteria we will use to compare them. One criterion, stability, comes up

several times in the chapter and gets a subsection of its own.

3.1.1 What is sorting?

Data is everywhere in daily life: shopping lists, bank transactions, social media posts, exam results. To be useful, it usually has to be kept in some order. An inbox is arranged by date, a dictionary by alphabet, a grade list by score. **Sorting** is the task of putting the items of a list in order according to some criterion. The criterion can be numeric value, alphabetical order, or date, and the order can be ascending (smallest to largest) or descending. Unless stated otherwise, these notes sort integers in ascending order. Descending order only requires reversing the direction of the comparison.

Definition 3.1

Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
Output: a **permutation** (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the same numbers that satisfies $a'_1 \le a'_2 \le \ldots \le a'_n$.

Both conditions in the definition matter. The output must be sorted, but that is not enough: it must also be a permutation of the input. No element may be lost, none duplicated, and no new value introduced. An "algorithm" that sets every element to 0 produces a sorted sequence, but it is not correct, because it violates the permutation condition.

In real applications, what gets sorted is often not a single number but a **record**: a row holding, say, a student number, a name, and a department. The field of the record used for sorting is called the **key**. The other fields of the record travel along with the key. In this chapter's examples a record consists of the key alone; for stability we will attach a label to each key.

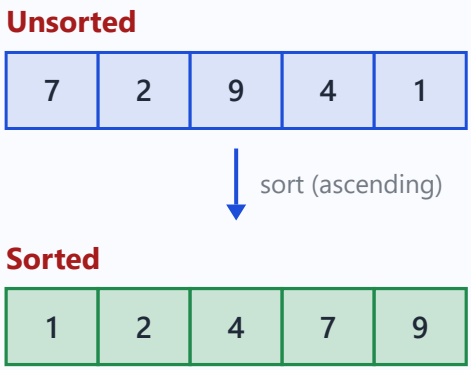


Figure 3.1: The sorting problem. The unsorted sequence 7, 2, 9, 4, 1 becomes 1, 2, 4, 7, 9 in ascending order. The output is a rearrangement of the five input numbers.

Why is sorting so important? Because many operations become much easier on sorted data. In a sorted array, **binary search** finds a value in $O(\log n)$ steps, whereas in an unsorted array the worst case inspects every element. Duplicates sit next to each other in a sorted array, so finding them takes a single pass. Two sorted lists can be merged in one pass. The median, the minimum, or the largest k elements can be read directly by index. For this reason sorting is often not a goal in itself but the first step of another algorithm.

3.1.2 Kinds of algorithms

Many algorithms solve the sorting problem, and each rests on a different working principle. Table 3.1 lists the algorithms studied in this chapter with their central ideas. The first four are simple and quadratic. The next three run in $n \log n$ time. Counting sort and radix sort do not compare elements. Timsort is a hybrid of two algorithms and is the one used today for sorting objects in the Java and Python libraries.

Algorithm	Central idea
Bubble sort	Compares neighbors and swaps them if they are out of order.
Insertion sort	Inserts the next element into its proper place in the sorted part.
Selection sort	Finds the smallest element, puts it first, then sorts the rest the same way.
Shell sort	Insertion-sorts distant elements first, with gaps that shrink step by step.
Merge sort	Splits the array in half, sorts the halves, then merges them.
Quicksort	Partitions around a pivot and sorts the two parts recursively.
Heap sort	Builds a max-heap and repeatedly moves the largest element to the end.
Counting / radix sort	No comparisons; counts the keys or distributes them digit by digit.
Timsort	A hybrid of insertion sort and merge sort; the object sort in Java and Python.

Table 3.1: The sorting algorithms of this chapter and their central ideas.

3.1.3 Evaluation criteria

Which of two algorithms for the same problem is better depends on what one cares about. Six criteria stand out for sorting algorithms.

Time. The running time of a sorting algorithm is determined largely by the number of two basic operations: **comparing** two elements and **moving** elements. A move can be a **swap**, which exchanges two elements, or a **shift**, which copies an element from one cell to another. These counts depend not only on the input size n but also on the input itself. So three cases are examined separately: the **best case**, the **average case**, and the **worst case**. Results are stated in asymptotic notation: $\Theta(n^2)$, $\Theta(n \log n)$, $\Theta(n + k)$, and so on. These complexities count one comparison as $O(1)$; for strings of length L, each comparison can cost $O(L)$ (see Chapters 10–11).

Extra memory. The memory an algorithm uses besides the input array matters too. An algorithm that works with only a few variables, that is, $O(1)$ extra memory, sorts **in place**. An in-place algorithm rearranges the elements within the input array. Some algorithms, such as merge sort, need auxiliary arrays.

Stability. An algorithm is **stable** if elements with equal keys keep, in the output, the relative order they had in the input. The next subsection explains this criterion with an example.

Comparison-based or not. Most algorithms in the chapter ask about the elements only one question: "is $a \le b$?" Such algorithms are called **comparison sorts**. In the worst case a comparison sort must make $\Omega(n \log n)$ comparisons. Counting sort and radix sort use the key values directly as array indices, so they fall outside this bound.

Adaptivity. An algorithm that sorts a nearly sorted input faster than a random one is **adaptive**. Insertion sort, bubble sort with a flag, and Timsort are adaptive. Selection sort, by contrast, makes the same number of comparisons whatever the input.

Online operation. An algorithm that can sort elements as they arrive, without seeing all the data in advance, is **online**. Insertion sort is online: the elements received so far are always sorted, and each new element is inserted in its place. Selection sort is not online, because it must see all elements to find the smallest.

These criteria are independent of one another, and one is often traded for another. Merge sort is both fast and stable but needs extra memory. Heap sort is fast and in place but unstable. No algorithm is best on every criterion; the right one is chosen by what the application needs.

3.1.4 Stability

Definition 3.2

A sorting algorithm is **stable** if, for every two records with equal keys, the record that comes first in the input also comes first in the output. In short, the relative order of records with equal keys is preserved. An algorithm that breaks this condition on at least one input is **unstable**.

Stability is meaningful only when records have an identity beyond the key. In an array of plain integers, two 3s cannot be told apart, so it does not matter which came first. Figure 3.2 therefore shows the records as (key, label) pairs. Sorting looks only at the number; the letter is the record's identity.

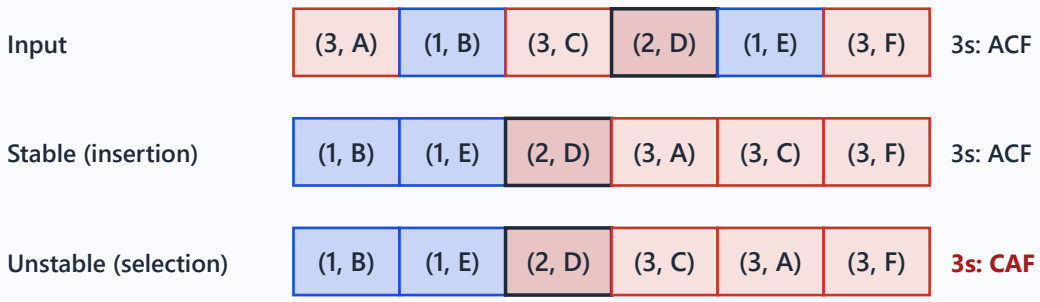


Figure 3.2: A stability example. The input consists of the records (3, A), (1, B), (3, C), (2, D), (1, E), (3, F). Insertion sort leaves the records with key 3 in their input order, A, C, F. The long-distance swaps of selection sort turn that order into C, A, F.

Example 3.1

Here is, step by step, why selection sort (3.4) breaks the order on this input. In each round it finds the smallest element of the unsorted part and swaps it with the first element of the unsorted part.

1. **Round 1.** The smallest key is 1; the first 1, that is (1, B), is chosen and swapped with (3, A) at the front. The array is (1, B), (3, A), (3, C), (2, D), (1, E), (3, F). This swap moved (3, A) to index 1.
2. **Round 2.** The unsorted part starts at (3, A); the smallest is (1, E), which is swapped with (3, A). The array is (1, B), (1, E), (3, C), (2, D), (3, A), (3, F). Now (3, A) has landed *behind* (3, C).
3. **Round 3.** The smallest is (2, D), swapped with (3, C): (1, B), (1, E), (2, D), (3, C), (3, A), (3, F).
4. **Rounds 4 and 5.** The keys of the remaining records are equal, and the code replaces the minimum only on a strictly smaller key, so no swap happens.

In the end the records with key 3 appear in the order C, A, F. The array is correctly sorted by key, but the relative order of A and C has been destroyed. The culprit is the long-distance swap in round 2: (3, A) was sent backward, jumping over (3, C) in between.

Stability is most useful in **multi-key sorting**. If a student list is sorted first by name and then, with a stable algorithm, by department, each department stays alphabetical. Because the second sort does not touch the order of students in the same department, the alphabetical order built by the first sort survives. An unstable algorithm may scramble the alphabetical order within a department. The same idea underlies radix sort: when numbers are sorted digit by digit, every intermediate step must be stable (3.9.6).

Common mistake

Testing stability on an array of plain integers. There, equal elements are indistinguishable, so even an unstable algorithm looks "correct". Stability is tested with records that have equal keys but different identities, such as the pairs (3, A) and (3, C).

3.2 Bubble sort

Bubble sort is one of the first ways one thinks of to sort an array. It looks only at two neighboring elements and swaps them if they are out of order. It is slow in practice, but it shows two important notions clearly: a sorted region that grows from pass to pass, and the **inversion**. Both reappear in the analysis of insertion sort.

3.2.1 The idea

Bubble sort scans the array from left to right and compares two neighboring elements at each step. If the left element is larger than the right one, the two are swapped. A large element is thus carried to the right through successive comparisons, like a bubble rising in water, which is where the algorithm gets its name.

One scan of the array from beginning to end is called a **pass**. At the end of a pass, the largest element of the part not yet sorted has reached its proper place at the end of the array. The reason is simple: once the scan reaches the largest element, no neighbor is larger, so it is swapped at every comparison, moves one position right each time, and is carried to the end of the pass. After the first pass the largest element is in place, after the second pass the two largest, and in general after pass i the largest i elements are in their final positions. Each pass can therefore be one element shorter than the one before.

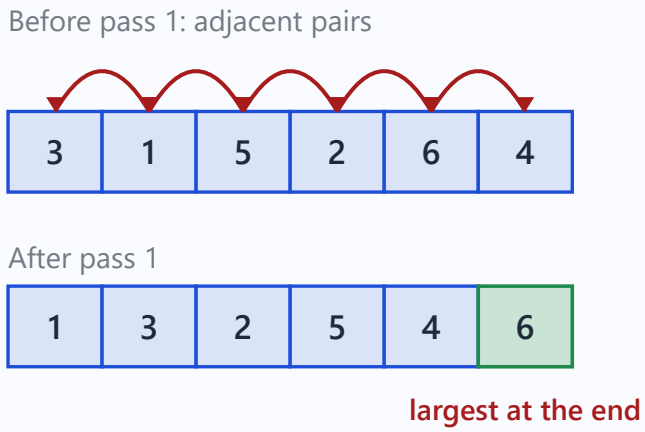


Figure 3.3: The first pass of bubble sort. At the top, the neighboring pairs being compared are marked with arrows. At the end of the pass, the array 3, 1, 5, 2, 6, 4 becomes 1, 3, 2, 5, 4, 6; the largest element, 6, has settled

in the last cell.

A small improvement is possible. If a pass makes no swap, every neighboring pair is in the right order, which means the array is sorted. The remaining passes are then unnecessary and the algorithm can stop early. This information is kept in a **flag** variable. The version with the flag recognizes a sorted input in a single pass.

3.2.2 Step by step: [3, 1, 5, 2, 6, 4]

Let us follow the algorithm on the six-element array [3, 1, 5, 2, 6, 4]. Table 3.2 shows each comparison, its outcome, and the array after the swap. The pair being compared is `a[j]` and `a[j+1]`.

Pass	`j`	Comparison	Result	Array
1	0	3 > 1?	yes, swap	[1, 3, 5, 2, 6, 4]
1	1	3 > 5?	no	[1, 3, 5, 2, 6, 4]
1	2	5 > 2?	yes, swap	[1, 3, 2, 5, 6, 4]
1	3	5 > 6?	no	[1, 3, 2, 5, 6, 4]
1	4	6 > 4?	yes, swap	[1, 3, 2, 5, 4, **6**]
2	0	1 > 3?	no	[1, 3, 2, 5, 4, 6]
2	1	3 > 2?	yes, swap	[1, 2, 3, 5, 4, 6]
2	2	3 > 5?	no	[1, 2, 3, 5, 4, 6]
2	3	5 > 4?	yes, swap	[1, 2, 3, 4, **5**, **6**]
3	0	1 > 2?	no	[1, 2, 3, 4, 5, 6]
3	1	2 > 3?	no	[1, 2, 3, 4, 5, 6]
3	2	3 > 4?	no	[1, 2, 3, 4, 5, 6]

Table 3.2: All comparisons of bubble sort with a flag on [3, 1, 5, 2, 6, 4].

Pass 1 makes five comparisons and three swaps; at the end, 6 sits in its final cell (index 5). Pass 2 is one element shorter: four comparisons, two swaps, and 5 settles at index 4. Pass 3 makes three comparisons and needs no swap in any of them. Because the flag stays `false`, the algorithm stops here; passes 4 and 5 never start. In total there are 12 comparisons and 5 swaps.

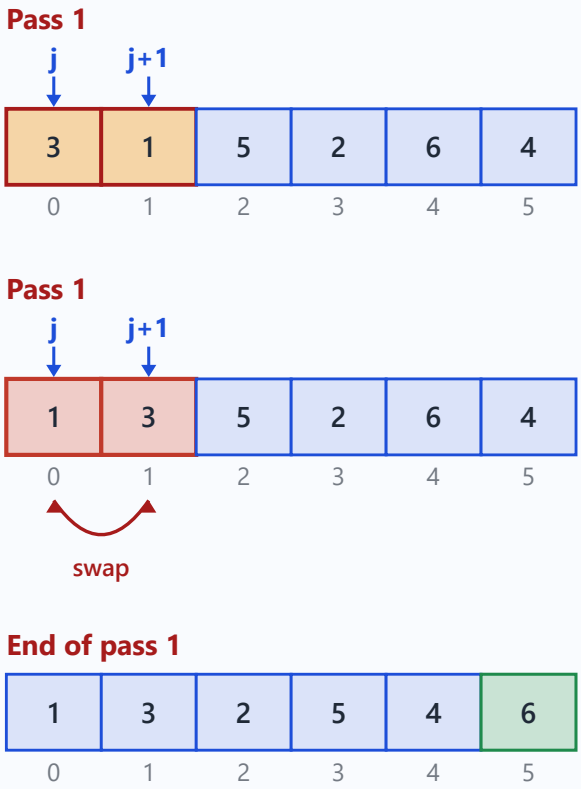


Figure 3.4: Three moments of the first pass. (a) `j = 0` : 3 is compared with 1. (b) Since 3 > 1, they were swapped; the array is [1, 3, 5, 2, 6, 4]. (c) End of the pass: 6 has settled in the last cell (green).

3.2.3 Java code

The following code sorts the array `a` of length `n` with bubble sort and a flag.

```
boolean swapped;
for(int i = 0; i < n - 1; i++) {
  swapped = false;
  for(int j = 0; j < n - 1 - i; j++) {
    if(a[j] > a[j+1]) {
      int temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
      swapped = true;
    }
  }
  if(swapped == false) break;
}
```

The code works line by line as follows:

- The **outer loop** (`i`) counts passes. At most *n − 1* passes are needed: once *n − 1* elements are in place, the one remaining element is automatically in place too.
- At the start of each pass the `swapped` flag is set to `false`. If any swap happens during the pass, it becomes `true`.
- The **inner loop** (`j`) walks over the neighboring pairs in the unsorted part. The last `i` elements were placed in earlier passes, so the condition is `j < n - 1 - i`. The final value of `j` is `n - 2 - i`; then `a[j+1]` is still the last cell of the unsorted part.
- If `a[j] > a[j+1]`, the two are swapped. A swap takes three assignments: the left value is saved in `temp`, the right value is written to the left, and the saved value is written to the right. Without `temp`, the first assignment would destroy the left value.
- If the flag is still `false` when a pass ends, no swap occurred in that pass, and `break` leaves the outer loop.

Note that the condition uses `>`, not `>=`. Two equal elements are never exchanged, so the order of elements with equal keys is preserved and bubble sort is **stable**. The algorithm uses space only for `temp`, `i`, `j`, and the flag: the extra memory is *O(1)* and the sort is in place.

> **Common mistake**
>
> Writing the inner loop condition as `j < n`. When `j = n - 1`, the code tries to read `a[j+1]`, that is, `a[n]`, and Java throws `ArrayIndexOutOfBoundsException`. In any loop that compares neighboring pairs, the last index is *n − 2*. Writing `j < n - 1` is correct but does unnecessary work; `j < n - 1 - i` does not look at elements already in place.

3.2.4 Code trace: [5, 1, 9, 2, 10]

This time let us run the code on the five-element array [5, 1, 9, 2, 10]. Since `n = 5`, the outer loop runs at most four passes. Table 3.3 gives the values of the variables at every step.

`i`	`j`	`a[j] > a[j+1]`	`swapped`	`a`
0	—	start of pass	false	[5, 1, 9, 2, 10]
0	0	5 > 1: yes, swap	true	[1, 5, 9, 2, 10]
0	1	5 > 9: no	true	[1, 5, 9, 2, 10]
0	2	9 > 2: yes, swap	true	[1, 5, 2, 9, 10]
0	3	9 > 10: no	true	[1, 5, 2, 9, **10**]
1	—	start of pass	false	[1, 5, 2, 9, 10]
1	0	1 > 5: no	false	[1, 5, 2, 9, 10]
1	1	5 > 2: yes, swap	true	[1, 2, 5, 9, 10]
1	2	5 > 9: no	true	[1, 2, 5, **9**, **10**]
2	—	start of pass	false	[1, 2, 5, 9, 10]

`i`	`j`	`a[j] > a[j+1]`	`swapped`	`a`
2	0	1 > 2: no	false	[1, 2, 5, 9, 10]
2	1	2 > 5: no	false	[1, 2, 5, 9, 10]
2	—	flag false → `break`	false	[1, 2, 5, 9, 10]

Table 3.3: Trace of the bubble sort code for `a = [5, 1, 9, 2, 10]`.

In the first pass (`i = 0`) the inner loop runs for `j = 0` to 3; two of the four comparisons cause a swap, and at the end 10 is in place. In the second pass (`i = 1`) the inner loop runs three times under the condition `j < 3`; 5 and 2 are exchanged and 9 is in place. In the third pass (`i = 2`) both comparisons answer "no". The flag stays `false`, so `break` executes and the fourth pass never starts. The result is [1, 2, 5, 9, 10], with 9 comparisons and 3 swaps in total.

The trace contains an interesting detail: at the end of the second pass the array is in fact sorted, but the algorithm cannot know it. It learns that only when a pass finishes without a swap. So bubble sort with a flag always makes one more pass after the array has become sorted.

3.2.5 Analysis and inversions

To understand the cost of bubble sort, let us first define a notion that measures how far an array is from being sorted.

Definition 3.3

An index pair *(i, j)* with *i < j* and *a[i] > a[j]* is called an **inversion**. An inversion is a pair of elements that are in the wrong order relative to each other; they need not be neighbors.

A sorted array has no inversions. In a reverse-sorted array of *n* elements, every pair is an inversion; there are *n(n − 1)/2* pairs, so that is the number of inversions. Every other array lies between these two extremes.

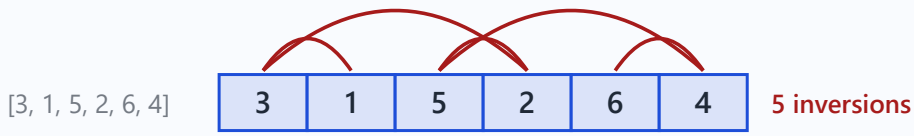


Figure 3.5: The five inversions of the array [3, 1, 5, 2, 6, 4]: (3, 1), (3, 2), (5, 2), (5, 4), and (6, 4). Each arc joins two elements that are in the wrong order relative to each other.

Theorem 3.1 (Adjacent swaps and inversions)

Swapping two adjacent elements that are out of order reduces the number of inversions in the array by exactly one. Hence the number of swaps made by bubble sort equals the number of inversions in the input.

Proof. Suppose $a[j] > a[j+1]$ and these two elements are swapped. The two formed an inversion between themselves; after the swap they are in the right order, so that inversion disappears. Now consider any other element x. It lies either to the left or to the right of both elements, and the swap does not change the position of the two elements relative to x. For instance, if x is on the left, it is to the left of both elements before and after. So the inversions involving x stay exactly as they were. The total number of inversions drops by exactly one. Bubble sort makes only such swaps, and a sorted array has no inversions left; therefore the number of swaps equals the initial number of inversions. □

The worst case follows directly from the theorem. A reverse-sorted array has $n(n-1)/2$ inversions, so that many swaps are needed. The comparison count is the same: no pass ends early, and the passes make $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$ comparisons. In the best case the input is sorted: the flagged version makes $n-1$ comparisons in the first pass, no swaps, and stops. In a random permutation, each pair is inverted with probability 1/2; the expected number of inversions, and hence of swaps, is $n(n-1)/4 \approx n^2/4$. The number of comparisons is close to $n^2/2$ on average, because the array is usually not sorted until the last passes.

Case	Input	Comparisons	Swaps	Time
Best	sorted	$n-1$ (one pass)	0	$\Theta(n)$
Average	random	$\approx n^2/2$	$\approx n^2/4$	$\Theta(n^2)$
Worst	reverse sorted	$n(n-1)/2$	$n(n-1)/2$	$\Theta(n^2)$

Table 3.4: The cost of bubble sort with a flag.

Bubble sort has few advantages, but they are real: it is very easy to understand and write, it is in place and stable, and with a flag it recognizes a sorted input in $\Theta(n)$. Its drawbacks weigh more. Quadratic time is unacceptable for large arrays. It performs a separate swap for every inversion, and each swap costs three assignments. Insertion sort, which makes one move per inversion, uses shifts instead of swaps. For this reason bubble sort is always slower than insertion sort in practice and is used mostly for teaching.

Example 3.2

Apply bubble sort with a flag to [5, 2, 4, 6, 1, 3]. The array and the counts at the end of each pass are:

1. **Pass 1** (5 comparisons): 5 ↔ 2, 5 ↔ 4, 6 ↔ 1, and 6 ↔ 3 are swapped; 4 swaps. The array is [2, 4, 5, 1, 3, 6].

2. **Pass 2** (4 comparisons): 5 ↔ 1 and 5 ↔ 3; 2 swaps. The array is [2, 4, 1, 3, 5, 6].
3. **Pass 3** (3 comparisons): 4 ↔ 1 and 4 ↔ 3; 2 swaps. The array is [2, 1, 3, 4, 5, 6].
4. **Pass 4** (2 comparisons): 2 ↔ 1; 1 swap. The array is [1, 2, 3, 4, 5, 6].
5. **Pass 5** (1 comparison): no swap, the flag stays `false`, `break`.

In total there are *5 + 4 + 3 + 2 + 1 = 15* comparisons and *4 + 2 + 2 + 1 = 9* swaps. The inversions of the array are (5,2), (5,4), (5,1), (5,3), (2,1), (4,1), (4,3), (6,1), (6,3); there are 9, equal to the number of swaps. Although the array is sorted after the fourth pass, the fifth pass is still needed to find that out.

3.3 Insertion sort

Insertion sort is among the fastest known algorithms for small arrays and for nearly sorted ones. It is therefore more than a teaching example: it is also a component of real library algorithms such as Timsort.

3.3.1 The idea and the invariant

Insertion sort resembles sorting playing cards in your hand. The cards are taken one at a time, and each new card is placed in its proper position among the sorted cards already held. The cards in hand are sorted at every moment; the sorted part grows by one card at each step.

In an array the idea is applied as follows. The array is divided into two parts: a **sorted part** on the left and an **unsorted part** on the right. At the start the sorted part consists of the first element alone; a one-element array is already sorted. At each step the first element of the unsorted part is taken and placed in its proper position within the sorted part. To make room, the elements larger than it are shifted one cell to the right.

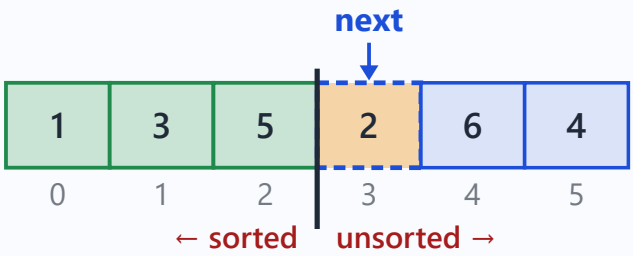


Figure 3.6: A moment in insertion sort. The sorted part is 1, 3, 5; the first element of the unsorted part, 2, is about to be inserted into it. For this, 5 and 3 shift one cell to the right and 2 is written to index 1.

The correctness of the algorithm is shown with a **loop invariant**. An invariant is a statement that is true at the start of every iteration of a loop. The invariant of insertion sort is this: at the start of each step of the outer loop, `a[0..i-1]` holds the elements originally in those positions, in sorted order. Initially *i = 1*, and the one-element `a[0..0]` is sorted. Each step inserts `a[i]` in its proper place in the sorted part, so `a[0..i]` becomes sorted and the invariant holds for the next step as well. The loop ends when *i = n*; at that moment the invariant says that `a[0..n-1]`, the whole array, is sorted.

3.3.2 Step by step: [3, 1, 5, 2, 6, 4]

Let us sort the array [3, 1, 5, 2, 6, 4], used earlier for bubble sort, with insertion sort. The element being inserted is held in a variable called `key`.

`i`	`key`	Elements shifted right	Insertion index	Shifts	Array
1	1	3	0	1	[**1, 3**, 5, 2, 6, 4]
2	5	—	2	0	[**1, 3, 5**, 2, 6, 4]
3	2	5, 3	1	2	[**1, 2, 3, 5**, 6, 4]
4	6	—	4	0	[**1, 2, 3, 5, 6**, 4]
5	4	6, 5	3	2	[**1, 2, 3, 4, 5, 6**]

Table 3.5: Insertion sort on [3, 1, 5, 2, 6, 4]. Each row is the insertion of one element into the sorted part.

In the first step `key = 1`. The 3 to its left is larger than 1, so 3 shifts one cell right and we reach the start of the array; 1 is written to index 0. In the second step `key = 5`; the 3 to its left is smaller, so no element shifts and 5 stays where it is. The third step is longer: for `key = 2`, first 5 and then 3 shift right; the scan stops at 1 ≤ 2, and 2 is written to index 1. In the fourth step 6 stays in place. In the last step, for 4, the 6 and the 5 shift and 4 lands at index 3. There are 5 shifts in total, a number equal to the number of inversions of the array.

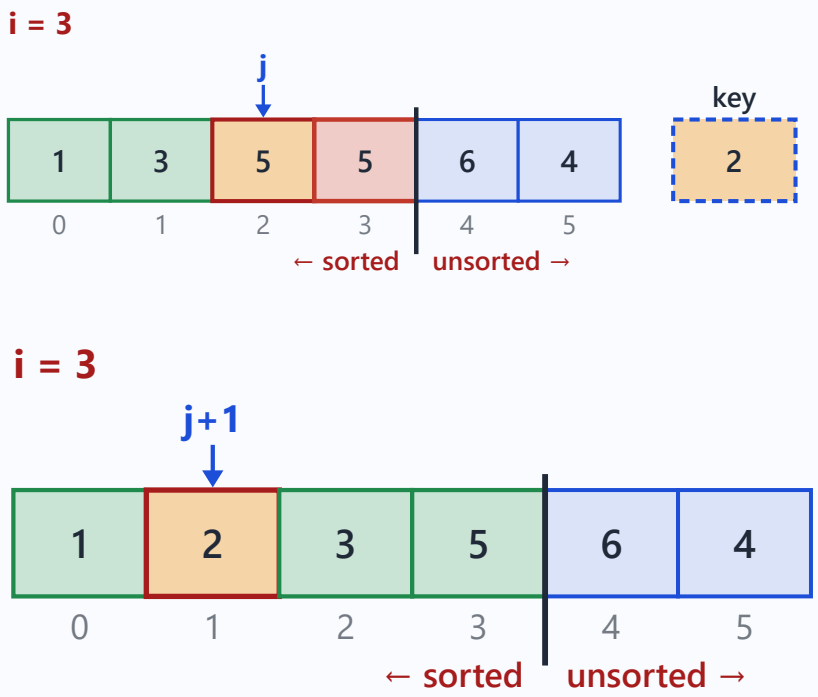


Figure 3.7: Two moments of the third step (`i = 3`). (a) Since 5 > 2, 5 was shifted one cell right; 5 now appears at both index 2 and index 3, because index 2 has not been overwritten yet. The `key` box holds 2. (b) 3 has shifted too; since 1 ≤ 2 the scan stops and 2 is placed at index 1.

Moment (a) in Figure 3.7 shows an important point. A shift is a copy: after the assignment `a[j+1] = a[j]` the same value sits in two cells. This is not an error; the left copy is overwritten in the next step, either by another shift or by `key`. The original cell of the element being inserted, however, is overwritten by the first shift. That is why the value must be saved in `key` before the loop starts.

3.3.3 Java code

```
for(int i = 1; i < n; i++) {
  int key = a[i];
  int j = i - 1;
  while(j >= 0 && a[j] > key) {
    a[j+1] = a[j];
    j = j - 1;
  }
  a[j+1] = key;
}
```

- The outer loop starts at `i = 1` because the one-element `a[0..0]` is already sorted.
- `key` is the element being inserted. The shifts will overwrite its cell, so it is saved first.
- `j` starts at the last index of the sorted part (`i - 1`) and moves left.

- As long as the element on the left is larger than `key`, the `while` loop shifts it one cell right and decrements `j`. The check `j >= 0` in the condition keeps the scan from running past the start of the array. In Java, `&&` short-circuits: when `j = -1`, the second condition is never evaluated and `a[-1]` is never read.
- When the loop ends, either `j = -1` or `a[j] <= key`. In both cases the vacated slot is `j + 1`, and `key` is written there.

Insertion sort makes shifts, not swaps. A swap is three assignments; a shift is one. Moving an element *k* cells to the left costs *k* swaps, that is, *3k* assignments, in bubble sort. In insertion sort, *k* shifts and two assignments (saving into `key` and writing it into place), *k + 2* assignments in all, suffice. This is why insertion sort is markedly faster than bubble sort.

The condition uses `a[j] > key`, not `>=`. On reaching an equal element, the scan stops and the new element lands to the right of the equal one. The input order of equal elements is thus preserved, and insertion sort is **stable**. The extra memory is only `key`, `i`, and `j`; the algorithm is **in place**.

> **Common mistake**
>
> Starting to shift without saving the key. If the loop compares directly against `a[i]` instead of `key`, the first shift (`a[i] = a[i-1]`) overwrites the key. From then on the comparisons use the wrong value, and one value in the array is lost while another is duplicated. This is a bug that violates the permutation condition.

3.3.4 Code trace: [5, 1, 9, 2, 10]

Let us run the code on the array [5, 1, 9, 2, 10] (`n = 5`). Table 3.6 shows each `while` check on its own row.

`i`	`key`	`j`	`while` condition	Action	`a`
1	1	0	0 ≥ 0 and 5 > 1: true	`a[1] = 5`, `j = -1`	[5, 5, 9, 2, 10]
1	1	−1	−1 ≥ 0: false	`a[0] = 1`	[1, 5, 9, 2, 10]
2	9	1	5 > 9: false	`a[2] = 9`	[1, 5, 9, 2, 10]
3	2	2	9 > 2: true	`a[3] = 9`, `j = 1`	[1, 5, 9, 9, 10]
3	2	1	5 > 2: true	`a[2] = 5`, `j = 0`	[1, 5, 5, 9, 10]
3	2	0	1 > 2: false	`a[1] = 2`	[1, 2, 5, 9, 10]
4	10	3	9 > 10: false	`a[4] = 10`	[1, 2, 5, 9, 10]

Table 3.6: Trace of the insertion sort code for `a = [5, 1, 9, 2, 10]`.

At `i = 1`, 5 shifts right and `j` drops to −1. At that point `j >= 0` is false and the loop ends; 1 is written to `a[0]`. At `i = 2`, the 5 on the left is ≤ 9, so there is no shift; 9 is simply written back into its own cell. At `i = 3`, 9 and 5 shift, the scan stops at 1 ≤ 2, and 2 goes into `a[1]`. At `i = 4`, 10 stays

in place. When `i = 5`, the outer loop condition (`5 < 5`) is false and the algorithm ends. There are 6 comparisons and 3 shifts in total. The inversions of the array are (5,1), (5,2), (9,2); their number equals the number of shifts.

3.3.5 Analysis

In insertion sort, each shift repairs exactly one inversion, formed by the element being inserted and a larger element. So the total number of shifts equals the number of inversions I, just as the number of swaps does in bubble sort. The number of comparisons is at most $I + (n - 1)$. Each successful comparison causes a shift. In addition, each element has at most one failed comparison, the one that ends the loop. If the element slides all the way to the start of the array, that last comparison is not made, because the check `j >= 0` becomes false first.

Theorem 3.2 (Running time of insertion sort)

On an array of n elements with I inversions, insertion sort makes exactly I shifts and at most $I + n - 1$ comparisons. The running time is $\Theta(n + I)$.

This theorem shows plainly that insertion sort is **adaptive**. In a sorted array $I = 0$: one comparison is made per element, $n - 1$ in all, and the time is $\Theta(n)$. In a reverse-sorted array $I = n(n - 1)/2$: every element slides to the start of the array and the time is $\Theta(n^2)$. In a random array the expected I is $n(n - 1)/4$; the number of comparisons and shifts is about $n^2/4$. That is about half the average number of comparisons of bubble sort.

Case	Input	Comparisons	Shifts	Time
Best	sorted	$n - 1$	0	$\Theta(n)$
Average	random	$\approx n^2/4$	$\approx n^2/4$	$\Theta(n^2)$
Worst	reverse sorted	$n(n - 1)/2$	$n(n - 1)/2$	$\Theta(n^2)$

Table 3.7: The cost of insertion sort.

Because the best and worst cases grow at different rates, the running time of insertion sort cannot be written as a single Θ for all inputs. The correct statement is: the time is $O(n^2)$ and $\Omega(n)$ over all inputs; the worst case is $\Theta(n^2)$ and the best case is $\Theta(n)$. An intermediate inference such as "upper bound n^2, lower bound n, hence the time is $\Theta(n \log n)$" is wrong; the bounds are tight for different families of inputs.

The advantages of insertion sort form a long list: it is stable, in place, online, and adaptive, and its loop is very simple. On small arrays (roughly $n \lesssim 20$–50) its low overhead makes it faster even than $n \log n$ algorithms. That is why Timsort and many libraries leave small subarrays to insertion sort inside quicksort and merge sort. Its drawback is quadratic time on large random arrays. The root of this is that an element can advance only one cell per step: a small element at the end of the array needs about n shifts to reach the front. Shell sort was devised precisely to solve this problem (3.5).

Example 3.3

While insertion sort sorts [5, 2, 4, 6, 1, 3], how many times is the comparison `a[j] > key` made, and how many shifts are performed?

1. `i = 1, key = 2`: 5 > 2, shift; `j = -1`, the condition ends here. 1 comparison, 1 shift. The array is [2, 5, 4, 6, 1, 3].
2. `i = 2, key = 4`: 5 > 4, shift; 2 > 4 is false. 2 comparisons, 1 shift. The array is [2, 4, 5, 6, 1, 3].
3. `i = 3, key = 6`: 5 > 6 is false. 1 comparison, 0 shifts.
4. `i = 4, key = 1`: 6, 5, 4, 2 shift; `j = -1`. 4 comparisons, 4 shifts. The array is [1, 2, 4, 5, 6, 3].
5. `i = 5, key = 3`: 6, 5, 4 shift; 2 > 3 is false. 4 comparisons, 3 shifts. The array is [1, 2, 3, 4, 5, 6].

In total there are 12 comparisons and 9 shifts. The number of shifts equals the number of inversions of the array (9). The number of comparisons is 2 below the bound $I + n - 1 = 9 + 5 = 14$. The reason: at `i = 1` and `i = 4` the element slid all the way to the start of the array. In these two steps the loop ended through the `j >= 0` check, with no extra comparison.

3.4 Selection sort

Selection sort rests on a different principle from bubble sort and insertion sort. It does not move elements a little at a time; in each round it selects the right element and puts it in its final place with a single swap. This approach minimizes the number of swaps, but it keeps the number of comparisons high regardless of the input.

3.4.1 The idea

Selection sort also divides the array into a sorted and an unsorted part. At the start the sorted part is empty. Each round does three things:

1. It finds the smallest element of the unsorted part.
2. It swaps that element with the first element of the unsorted part.
3. After the swap, that cell joins the sorted part, which grows by one element.

An element that enters the sorted part is now in its final position. All smaller elements were placed in earlier rounds in the cells to its left, and everything to its right is larger than or equal to it. In this respect selection sort differs from insertion sort. In insertion sort the sorted part is sorted internally,

but the elements are not yet in their final places; a small element arriving later may shift all of them. On an array of n elements, $n - 1$ rounds suffice, because the last remaining element is automatically the largest. Each round makes at most one swap.

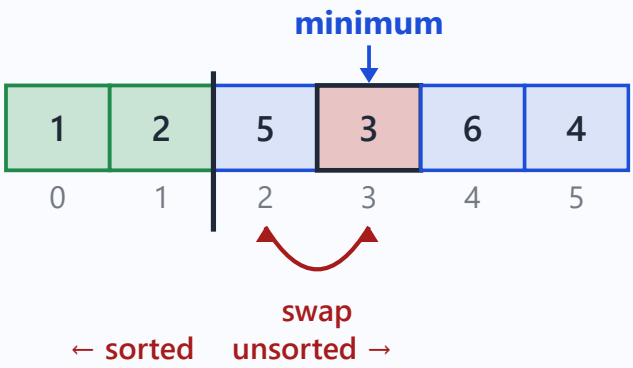


Figure 3.8: One round of selection sort. The sorted part is 1, 2. The smallest element of the unsorted part, 3, is found and swapped with the 5 at the front of the unsorted part.

3.4.2 Step by step: [3, 1, 5, 2, 6, 4]

In each round, the first element of the unsorted part is taken as the provisional minimum; the remaining elements are then compared with it one by one. If a smaller element is found, the minimum is updated.

Round	Initial minimum	Scanned	Minimum found	Swap	Array
1	3	1, 5, 2, 6, 4	1	3 ↔ 1	[**1**, 3, 5, 2, 6, 4]
2	3	5, 2, 6, 4	2	3 ↔ 2	[**1, 2**, 5, 3, 6, 4]
3	5	3, 6, 4	3	5 ↔ 3	[**1, 2, 3**, 5, 6, 4]
4	5	6, 4	4	5 ↔ 4	[**1, 2, 3, 4**, 6, 5]
5	6	5	5	6 ↔ 5	[**1, 2, 3, 4, 5**, 6]

Table 3.8: Selection sort on [3, 1, 5, 2, 6, 4]. The "Scanned" column lists, in order, the elements compared with the provisional minimum.

In the first round the provisional minimum is 3. Since 1 < 3, the minimum becomes 1; none of the following 5, 2, 6, 4 is smaller than 1. After five comparisons, 1 is swapped with the 3 at the front. In the second round the scan starts from 3 and finds 2; 2 is swapped with the 3 at the front of the unsorted part. Notice that this swap sent 3 to index 3, that is, to the right of 5; in the third round it is selected again and brought to index 2. In the fourth round 4 is swapped with 5, and in the fifth round 5 is swapped with 6. After five rounds the last element, 6, is automatically in place. In total there are $5 + 4 + 3 + 2 + 1 = 15$ comparisons and 5 swaps.

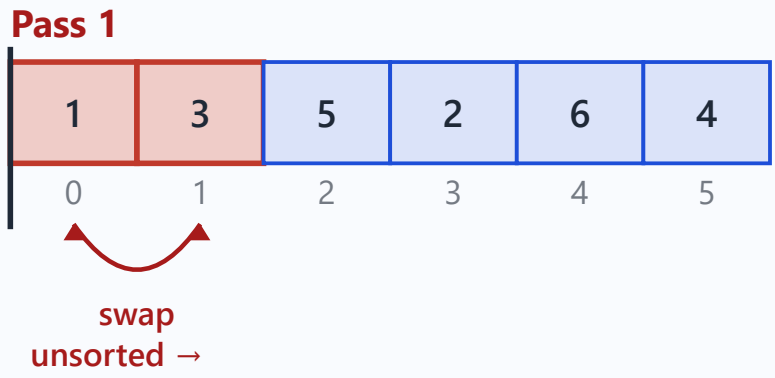


Figure 3.9: The end of the first round. The minimum found, 1, has been swapped with the 3 at the front of the unsorted part; the thick line is the boundary of the sorted part, and it moves one cell right in the next round.

3.4.3 Java code

```
public void sort(int[] a) {
  int n = a.length;
  for(int i = 0; i < n - 1; i++) {
    int min = i;
    for(int j = i + 1; j < n; j++) {
      if(a[j] < a[min]) {
        min = j;
      }
    }
    int temp = a[min];
    a[min] = a[i];
    a[i] = temp;
  }
}
```

- The outer loop runs from `i = 0` to `n - 2`; `i` is the start of the unsorted part and the cell to be filled in this round.
- `min` holds the *index* of the smallest element found so far, not its value. The swap needs to know which cell to exchange with; the value can always be read as `a[min]`.
- The inner loop scans the whole unsorted part, from `j = i + 1` to the end of the array. Even if the input is sorted, this scan does not get shorter; the only way to confirm that the minimum really is the minimum is to look at every element.
- The minimum is updated only when `a[j] < a[min]`, that is, on a strictly smaller element. On a tie, the first one found is kept.
- After the inner loop, `a[min]` and `a[i]` are exchanged with three assignments. If `min == i`, the element is already in place; the code still "swaps it with itself", which is harmless but costs three needless assignments.

Selection sort makes one swap per round, at most $n - 1$ in total. That is far fewer than the number of swaps in the other simple algorithms. In settings where writing is expensive, for example flash memory where each write wears the cell, the small number of swaps is the one real advantage of selection sort.

Common mistake

Keeping the minimum as a value instead of an index. If you write `int minValue = a[i]` and update only the value in the inner loop, at the end of the round you do not know *which cell* holds the minimum, and the swap cannot be made. Since the value can always be read from the index, keeping the index is enough.

3.4.4 Code trace: [5, 1, 10, 2, 9]

`i`	`j`	`a[j] < a[min]`	`min`	`a` at end of round
0	1	1 < 5: yes	1	
0	2	10 < 1: no	1	
0	3	2 < 1: no	1	
0	4	9 < 1: no	1	`a[1] ↔ a[0]` : [1, 5, 10, 2, 9]
1	2	10 < 5: no	1	
1	3	2 < 5: yes	3	
1	4	9 < 2: no	3	`a[3] ↔ a[1]` : [1, 2, 10, 5, 9]
2	3	5 < 10: yes	3	
2	4	9 < 5: no	3	`a[3] ↔ a[2]` : [1, 2, 5, 10, 9]
3	4	9 < 10: yes	4	`a[4] ↔ a[3]` : [1, 2, 5, 9, 10]

Table 3.9: Trace of the selection sort code for `a = [5, 1, 10, 2, 9]`. Each row is one comparison.

At the start of each round `min = i` is assigned. At `i = 0` the minimum is found right away at `j = 1`, and the next three comparisons do not change it. When `j = 5` the inner loop ends and 1 and 5 are exchanged. At `i = 1` the minimum is 2 (index 3); at `i = 2` it is 5 (again index 3); at `i = 3` it is 9 (index 4). For `i = 4` the outer loop condition `4 < n - 1` is false and the algorithm ends. In total there are $4 + 3 + 2 + 1 = 10$ comparisons and 4 swaps.

3.4.5 Analysis and instability

The number of comparisons in selection sort does not depend on the input at all. In round i the inner loop runs $n - 1 - i$ times; the total is

$$(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$$

comparisons. Whether the input is sorted, reverse sorted, or random, this many comparisons are made. So the best, average, and worst cases are all the same: $\Theta(n^2)$. Selection sort is **not adaptive**; it spends $n(n-1)/2$ comparisons even on a sorted array.

Measure	Value
Comparisons	$n(n-1)/2$, on every input
Swaps	at most $n-1$
Time	best = average = worst = $\Theta(n^2)$
Extra memory	$O(1)$, in place
Stable?	No (long-distance swap)

Table 3.10: Properties of selection sort.

Selection sort is not stable. The reason is that a swap does not merely exchange two elements; it makes one of them jump over all the elements in between. The element at the front of the unsorted part is sent to a distant cell where the minimum was found, and on the way it may jump over an element equal to itself. The smallest example has three elements.

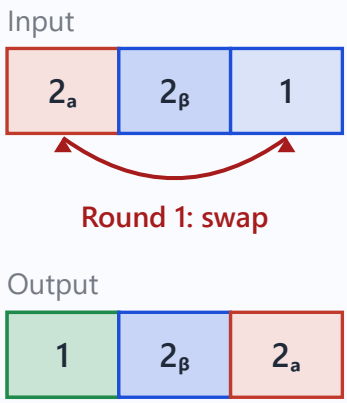


Figure 3.10: The instability of selection sort. The input is $[2_a, 2_b, 1]$. In the first round the minimum, 1, is swapped with the 2_a at the front, and 2_a moves behind 2_b. The output is $[1, 2_b, 2_a]$.

In the second round the unsorted part is $[2_b, 2_a]$; since the code updates only on a strictly smaller element, 2_b stays in place and no swap occurs. The result is numerically correct, but the relative order of 2_a and 2_b has been reversed. Selection sort can be made stable: instead of swapping, the minimum is brought to the front by shifting the elements in between one cell right. But that removes the advantage of few swaps and brings the cost close to that of insertion sort.

3.4.6 Three simple algorithms on the same input

Bubble, insertion, and selection sort belong to the same quadratic class, but their constant factors and their reactions to the input differ. Table 3.11 gives the comparison and move (swap or shift) counts of the three algorithms on three inputs. The values were obtained by running the code in this chapter; $n = 6$.

Algorithm	Kind of move	[3, 1, 5, 2, 6, 4]	sorted 1…6	reversed 6…1
Bubble (with flag)	swap	12 / 5	5 / 0	15 / 15
Insertion	shift	9 / 5	5 / 0	15 / 15
Selection	swap	15 / 5	15 / 0	15 / 3

Table 3.11: Cells have the form "comparisons / moves". For bubble and selection sort a move is a swap; for insertion sort it is a shift.

Three observations follow from the table. First, in bubble sort and insertion sort the number of moves equals the number of inversions: 5 for [3, 1, 5, 2, 6, 4] and $6 \cdot 5/2 = 15$ for the reversed array. But a swap is three assignments and a shift is one, so the same number of moves is much cheaper in insertion sort. Second, selection sort always makes $n(n - 1)/2 = 15$ comparisons regardless of the input; in return, it makes the fewest swaps. On the reversed array only 3 swaps suffice: 6 ↔ 1, 5 ↔ 2, and 4 ↔ 3. Elements trade places in pairs, and the swaps in the middle become unnecessary. Third, on a sorted input, bubble sort with a flag and insertion sort make only $n - 1 = 5$ comparisons; both are adaptive.

The 0 in the swap column of selection sort for the sorted input appears because the count includes only real swaps, those with `min ≠ i`. As the code is written, it swaps an element with itself in every round; if those are counted as assignments, a sorted input also involves $n - 1$ "swaps".

3.5 Shell sort

The weak point of insertion sort is that an element can advance only one cell per step. Shell sort removes this restriction: it moves elements first in big steps and then in steadily smaller ones. The last pass is ordinary insertion sort again, but by then the array is nearly sorted.

3.5.1 The idea and the code

Shell sort, proposed by D. L. Shell in 1959, is a generalization of insertion sort. The algorithm works with a **gap** value. Elements that are a gap apart form a group: with gap 4, indices 0, 4, 8, … form one group and 1, 5, 9, … another. Each group is sorted internally by insertion sort. In the process a small element jumps a whole gap in one move. Then the gap is reduced and the procedure is repeated. In the last pass the gap is 1, and that pass is ordinary insertion sort.

Since the last pass is insertion sort anyway, why do the earlier passes help? Because they remove most of the distant inversions cheaply. In a pass with gap 4, one shift moves an element four cells; in insertion sort the same move would take four shifts. By the time the gap reaches 1, few inversions remain in the array, and by the theorem in 3.3.5, insertion sort finishes such an array quickly.

```
void shellSort(int[] a) {
  int n = a.length;
  for (int gap = n / 2; gap > 0; gap /= 2)
    for (int i = gap; i < n; i++) {
      int key = a[i], j;
      for (j = i; j >= gap && a[j - gap] > key; j -= gap)
        a[j] = a[j - gap];
      a[j] = key;
    }
}
```

- The outer loop starts the gap at `n / 2` and halves it each pass (`gap /= 2`); it stops when the gap reaches 0. With integer division the gap sequence is 4, 2, 1 for $n = 8$ and 3, 1 for $n = 6$.
- The middle loop runs from `i = gap` to the end of the array. Each `i` is the element to be inserted in its own group. The groups are processed interleaved, not one at a time: when `i` increases by one, the next element belongs to a different group.
- The inner loop is the same as the `while` loop of insertion sort, except that the step is `gap` instead of 1. It stops when `j` falls below `gap` or when the group element on the left is not larger than `key`. `key` is written to the vacated cell `a[j]`.

Shell sort is in place: it uses only a few variables, so the extra memory is $O(1)$. But it is not stable. Two equal elements can lie in different groups; in a pass with a large gap, one can jump over the other and end up in front of it.

3.5.2 Step by step: [6, 5, 3, 1, 8, 7, 2, 4]

For an eight-element array the gap sequence is 4, 2, 1. Table 3.12 gives the state after each gap.

gap	Groups (before)	Groups (after)	Shifts	Array
4	[6, 8], [5, 7], [3, 2], [1, 4]	[6, 8], [5, 7], [2, 3], [1, 4]	1	[6, 5, 2, 1, 8, 7, 3, 4]
2	[6, 2, 8, 3], [5, 1, 7, 4]	[2, 3, 6, 8], [1, 4, 5, 7]	6	[2, 1, 3, 4, 6, 5, 8, 7]
1	[2, 1, 3, 4, 6, 5, 8, 7]	[1, 2, 3, 4, 5, 6, 7, 8]	3	[1, 2, 3, 4, 5, 6, 7, 8]

Table 3.12: Shell sort on [6, 5, 3, 1, 8, 7, 2, 4].

With gap 4 there are four groups of two elements each. Only the group [3, 2] is out of order; one shift makes it [2, 3]. With gap 2 there are two groups: the even indices [6, 2, 8, 3] and the odd indices [5, 1, 7, 4]. The first group becomes [2, 3, 6, 8] under insertion sort (for 2, the 6 shifts; for 3, the 8 and the 6 shift: 3 shifts). The second group also becomes [1, 4, 5, 7] with 3 shifts. When the groups are written

back to their places in the array, we get [2, 1, 3, 4, 6, 5, 8, 7]. Only three inversions remain in this array: (2, 1), (6, 5), (8, 7). The gap-1 pass, which is ordinary insertion sort, therefore makes only 3 shifts. Ten shifts were needed in all; sorting the same array directly with insertion sort would have taken as many shifts as there are inversions, namely 16.

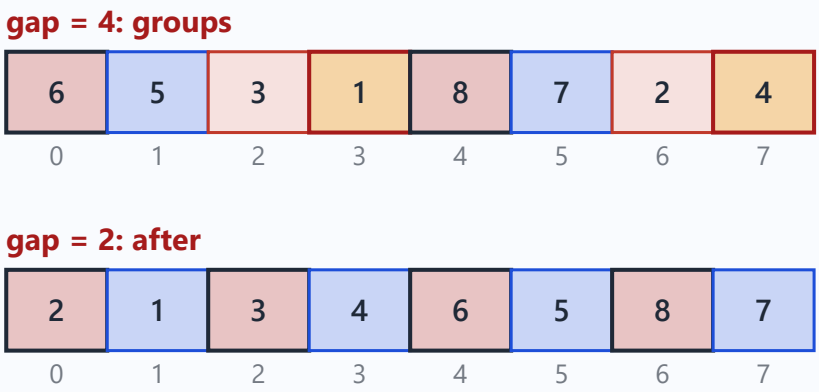


Figure 3.11: (a) The groups with gap 4: cells of the same color belong to the same group. (b) After the gap-2 pass the array is [2, 1, 3, 4, 6, 5, 8, 7]; the two groups, even and odd indices, are each sorted internally.

3.5.3 Code trace: [5, 2, 6, 1, 4, 3]

The behavior of the code on a six-element array is shown in Table 3.13. Since $n = 6$, the first gap is 3; the groups are the index pairs (0, 3), (1, 4), and (2, 5). The next gap is $3/2 = 1$.

gap	i	key	Inner loop	a
3	3	1	`a[0] = 5 > 1` : 5 shifts to index 3; stops at `j = 0`	[1, 2, 6, 5, 4, 3]
3	4	4	`a[1] = 2 ≤ 4` : no shift	[1, 2, 6, 5, 4, 3]
3	5	3	`a[2] = 6 > 3` : 6 shifts to index 5; stops at `j = 2`	[1, 2, 3, 5, 4, 6]
1	1	2	`a[0] = 1 ≤ 2` : no shift	[1, 2, 3, 5, 4, 6]
1	2	3	`a[1] = 2 ≤ 3` : no shift	[1, 2, 3, 5, 4, 6]
1	3	5	`a[2] = 3 ≤ 5` : no shift	[1, 2, 3, 5, 4, 6]
1	4	4	`a[3] = 5 > 4` : 5 shifts to index 4; stops at `a[2] = 3 ≤ 4`	[1, 2, 3, 4, 5, 6]
1	5	6	`a[4] = 5 ≤ 6` : no shift	[1, 2, 3, 4, 5, 6]

Table 3.13: Trace of the Shell sort code for `a = [5, 2, 6, 1, 4, 3]`.

In the gap-3 pass, 1 jumps from index 3 to index 0 in one move; insertion sort would have needed three shifts for that. In the same way 3 moves from index 5 to index 2. After this pass the array is [1, 2, 3, 5, 4, 6] and only one inversion, (5, 4), remains. The gap-1 pass fixes it with a single shift. When the gap becomes $1/2 = 0$, the outer loop ends. Only 3 shifts were made in total. The same array has 9 inversions; insertion sort alone would have made 9 shifts.

3.5.4 Gap sequences and complexity

The running time of Shell sort depends on the gap sequence used, and a full analysis is hard. Shell's original sequence is *n/2, n/4, …, 1*; the code in these notes uses it. The worst case of this sequence is $\Theta(n^2)$. The cause is that when n is a power of two, all the gaps are even until the last pass. Even indices and odd indices are never compared until the last pass. For example, if all the large elements are at even indices and all the small ones at odd indices, the early passes cannot mix them, and the last pass does quadratic work.

Better gap sequences exist. T. N. Hibbard's sequence $2^k - 1$ (1, 3, 7, 15, …) keeps consecutive gaps free of common factors and lowers the worst case to $O(n^{1.5})$. Other sequences improve on this further; in practice Shell sort is quite fast on arrays of moderate size. In the best case, a sorted input, each pass makes about n comparisons and there are $\log n$ passes: the time is $\Theta(n \log n)$. No general closed formula is known for the average running time of Shell sort, so comparison tables list it as "depends on the gaps".

> **Common mistake**
>
> Believing that Shell sort is stable because each group is sorted by insertion sort. Insertion sort is stable within a single group, but two equal elements can be in different groups. For example, in the array [5, 2_a, 2_b, 1], with gap 2 the even-index group [5, 2_b] is sorted and 2_b moves to the front; 2_a stays in the odd-index group. In the end 2_b lands in front of 2_a.

3.6 Merge sort

All the algorithms examined so far were quadratic in the worst case. Merge sort is the first to break this barrier: it runs in $\Theta(n \log n)$ time on every input. It does so with a design method that splits a problem into small pieces and combines the solutions of the pieces.

3.6.1 Divide and conquer

Divide and conquer is an algorithm design method that splits a problem into smaller subproblems of the same kind, solves the subproblems recursively, and combines the results. Every divide-and-conquer algorithm has three steps.

1. **Divide**: The problem is split into smaller subproblems.
2. **Conquer**: The subproblems are solved by the same method, recursively. If a subproblem is small enough, it is solved directly; this is the **base case** of the recursion.
3. **Combine**: The solutions of the subproblems are turned into a solution of the original problem.

Merge sort applies this method to sorting. The array is split into two halves at the middle. Each half is sorted by the same method. Then the two sorted halves are merged into a single sorted array. The base case is the one-element array; a one-element array is sorted by definition. The algorithm is attributed to John von Neumann, 1945.

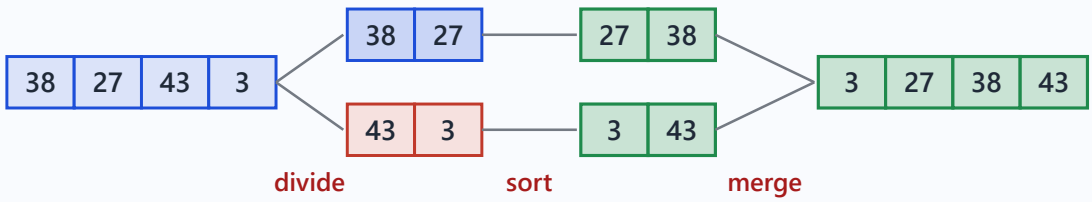


Figure 3.12: The three steps of divide and conquer on [38, 27, 43, 3]. The array is split into the halves [38, 27] and [43, 3]; the halves are sorted separately into [27, 38] and [3, 43]; merging gives [3, 27, 38, 43].

In merge sort the divide step is almost free: only the middle index is computed. The real work is done in the combine step. The idea of merging two sorted arrays is simple. The smallest element of each array is at its front. The two fronts are compared, the smaller is written to the output, and we advance to the next element of that array. When one array runs out, whatever is left in the other is copied to the output in order. Since each comparison writes one element to the output, merging m elements in total takes at most $m - 1$ comparisons and m writes: the time is $\Theta(m)$.

3.6.2 Example: [38, 27, 43, 3, 9, 82, 10]

Consider an array of seven elements. Because the middle index is computed as $\lfloor (lo + hi)/2 \rfloor$, in pieces with an odd number of elements the left half takes one extra element: [38, 27, 43, 3] and [9, 82, 10]. Figure 3.13 shows the splits and the merges together.

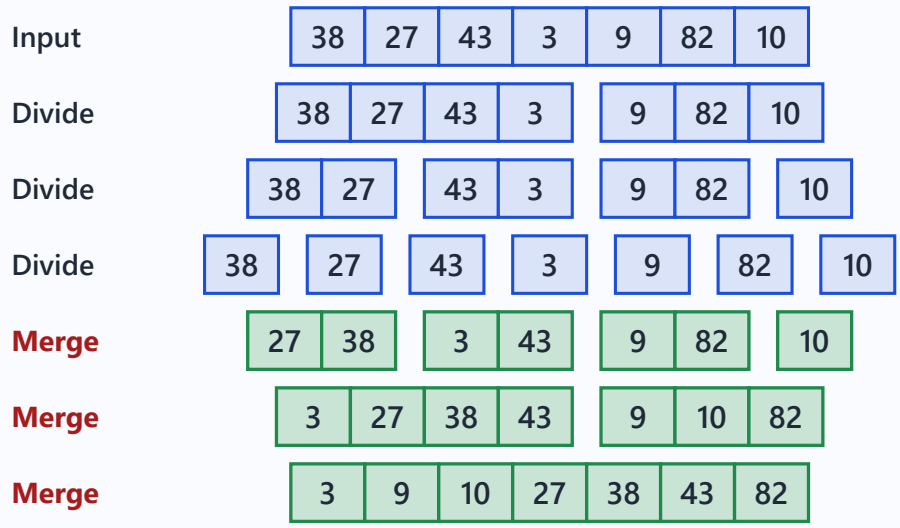


Figure 3.13: A merge sort example. In the upper half the array is split down to single elements. In the lower half the pieces are merged in pairs, giving first [27, 38], [3, 43], [9, 82], [10]; then [3, 27, 38, 43] and [9, 10,

82]; and finally [3, 9, 10, 27, 38, 43, 82].

Because the code works depth first, the merges happen not in the level order of the figure but in the order below. The right half is not started before the left half is completely sorted.

Order	Range	Left + right	Result	Comparisons
1	[0..1]	[38] + [27]	[27, 38]	1
2	[2..3]	[43] + [3]	[3, 43]	1
3	[0..3]	[27, 38] + [3, 43]	[3, 27, 38, 43]	3
4	[4..5]	[9] + [82]	[9, 82]	1
5	[4..6]	[9, 82] + [10]	[9, 10, 82]	2
6	[0..6]	[3, 27, 38, 43] + [9, 10, 82]	[3, 9, 10, 27, 38, 43, 82]	6

Table 3.14: The `merge` calls of `mergeSort(a, 0, 6)`, in the order they occur.

`merge` is called 6 times in all. This number is always $n - 1$. Each merge reduces two pieces to one, so going from n one-element pieces to a single array takes $n - 1$ merges. The comparison total is $1 + 1 + 3 + 1 + 2 + 6 = 14$.

3.6.3 The order of the recursion

To see in what order the recursive calls proceed, let us follow the eight-element array [6, 5, 3, 1, 8, 7, 2, 4]. The array is first split into [6, 5, 3, 1] and [8, 7, 2, 4]. The left half is split into [6, 5] and [3, 1], and [6, 5] into [6] and [5]. The first merge happens here: [6] + [5] → [5, 6]. Then [3, 1] is split and merged as [1, 3]. When [5, 6] and [1, 3] merge, the left half is sorted as [1, 3, 5, 6]. Only after that do we move to the right half: [8, 7] → [7, 8], [2, 4] → [2, 4], and merging the two gives [2, 4, 7, 8]. The last merge is [1, 3, 5, 6] + [2, 4, 7, 8] → [1, 2, 3, 4, 5, 6, 7, 8].

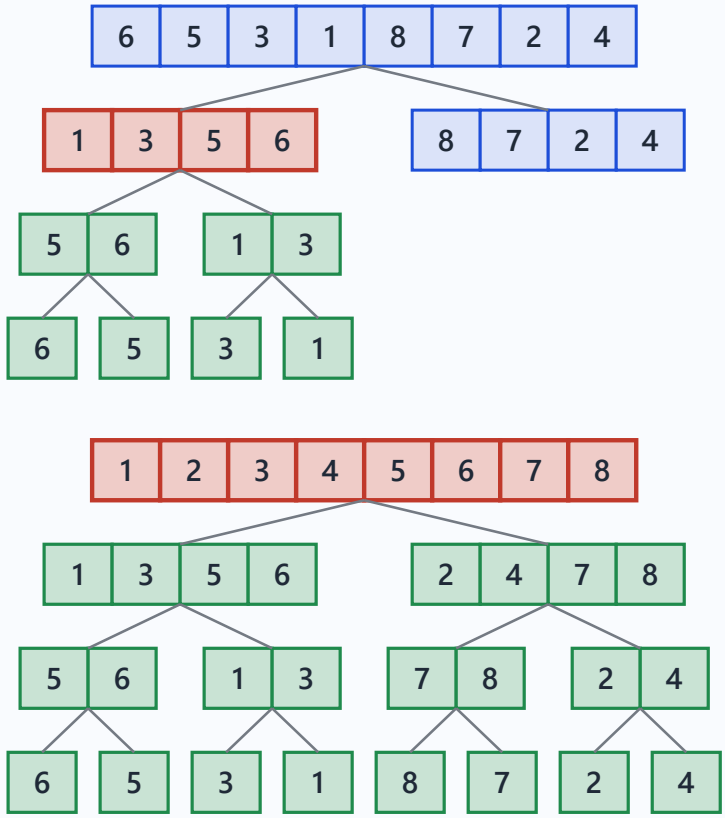


Figure 3.14: The recursion tree. (a) The left half is complete: [5, 6] and [1, 3] merged into [1, 3, 5, 6]; the right half ([8, 7, 2, 4]) has not been touched yet. (b) All merges are done; the root is [1, 2, 3, 4, 5, 6, 7, 8]. Green boxes are sorted pieces.

This order is the natural consequence of keeping the recursive calls on a stack, the call stack. At any moment the stack holds at most $\lfloor \log_2 n \rfloor + 1$ calls, because each call halves the piece size. For eight elements the stack depth is at most 4: four pieces of sizes 8, 4, 2, and 1.

3.6.4 Java code

The code consists of two methods. `mergeSort` sorts the range `a[lo..hi]`; the initial call for the whole array is `mergeSort(a, 0, n - 1)`.

```
void mergeSort(int[] a, int lo, int hi) {
  if (lo < hi) {
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, lo, mid);
    mergeSort(a, mid + 1, hi);
    merge(a, lo, mid, hi);
  }
}
```

- If `lo < hi` does not hold, the range has at most one element and the method returns without doing anything; this is the base case.
- The middle index is computed as `lo + (hi - lo) / 2`. In mathematics this equals `(lo + hi) / 2`. But in Java the sum `lo + hi` can exceed the `int` limit ($2^{31} - 1$) on very large arrays and wrap around to a negative number. The first form avoids this overflow.
- The left half `[lo..mid]` and the right half `[mid+1..hi]` are sorted recursively; then `merge` is called.

`merge` merges the sorted pieces `A[p..q]` and `A[q+1..r]` into `A[p..r]`.

```
void merge(int[] A, int p, int q, int r) {
  int[] L = Arrays.copyOfRange(A, p, q + 1);
  int[] R = Arrays.copyOfRange(A, q + 1, r + 1);
  int i = 0, j = 0, k = p;
  while (i < L.length && j < R.length) {
    if (L[i] <= R[j]) A[k++] = L[i++];
    else              A[k++] = R[j++];
  }
  while (i < L.length) A[k++] = L[i++];
  while (j < R.length) A[k++] = R[j++];
}
```

- The two halves are first copied into the auxiliary arrays `L` and `R` with `Arrays.copyOfRange`. The copy is necessary because the result is written directly into the same cells of `A`, and without a copy it would overwrite elements not yet read. The upper bound of `copyOfRange` is exclusive, which is why `q + 1` and `r + 1` are written.
- `i` and `j` point to the next element of `L` and `R` respectively, and `k` to the cell of `A` to be written.
- The first `while` compares the fronts as long as both arrays have elements; it writes the smaller or equal one to `A[k]` and advances the corresponding index. The expression `A[k++] = L[i++]` first writes, then increments both indices.
- When one array runs out, only one of the remaining two `while` loops runs and copies what is left of the other array in order. The remaining elements are already sorted and are greater than or equal to everything written so far; no comparison is needed.

The condition is `L[i] <= R[j]`. On a tie the left element is taken first. Since elements in the left half came before those in the right half in the input, the relative order of equal keys is preserved: merge sort is **stable**.

Common mistake

Writing `<` instead of `<=` in the merge. The array is still sorted correctly, but on a tie the right element is taken first. The order of two records with equal keys is reversed and the algorithm loses its stability. This bug does not show up in tests with arrays of numbers; it appears only when records are sorted.

3.6.5 Code trace: merge()

In the last step of the [6, 5, 3, 1, 8, 7, 2, 4] example, `merge(A, 0, 3, 7)` is called. At that moment `A[0..3] = [1, 3, 5, 6]` and `A[4..7] = [2, 4, 7, 8]`.

Step	`i, j, k`	Comparison	Written	`A[0..k]`
1	0, 0, 0	`L[0] = 1 ≤ R[0] = 2`	`A[0] = 1`	[1]
2	1, 0, 1	`L[1] = 3 > R[0] = 2`	`A[1] = 2`	[1, 2]
3	1, 1, 2	`L[1] = 3 ≤ R[1] = 4`	`A[2] = 3`	[1, 2, 3]
4	2, 1, 3	`L[2] = 5 > R[1] = 4`	`A[3] = 4`	[1, 2, 3, 4]
5	2, 2, 4	`L[2] = 5 ≤ R[2] = 7`	`A[4] = 5`	[1, …, 5]
6	3, 2, 5	`L[3] = 6 ≤ R[2] = 7`	`A[5] = 6`	[1, …, 6]
7	4, 2, 6	`L` is exhausted; the rest of `R` is copied	`A[6] = 7`	[1, …, 7]
8	4, 3, 7	—	`A[7] = 8`	[1, …, 8]

Table 3.15: Trace of `merge` for `L = [1, 3, 5, 6]`, `R = [2, 4, 7, 8]`.

In the first six steps one comparison is made each, and the result comes alternately from the two arrays. After step six `i = 4`; `L` is exhausted and the first `while` ends. The remaining 7 and 8 are copied without comparison. There are 6 comparisons and 8 writes in total. Merging eight elements takes at most 7 comparisons; the bound is reached when the two arrays run out together at their last elements. For example, `L = [1, 3, 5, 7]`, `R = [2, 4, 6, 8]` needs 7 comparisons.

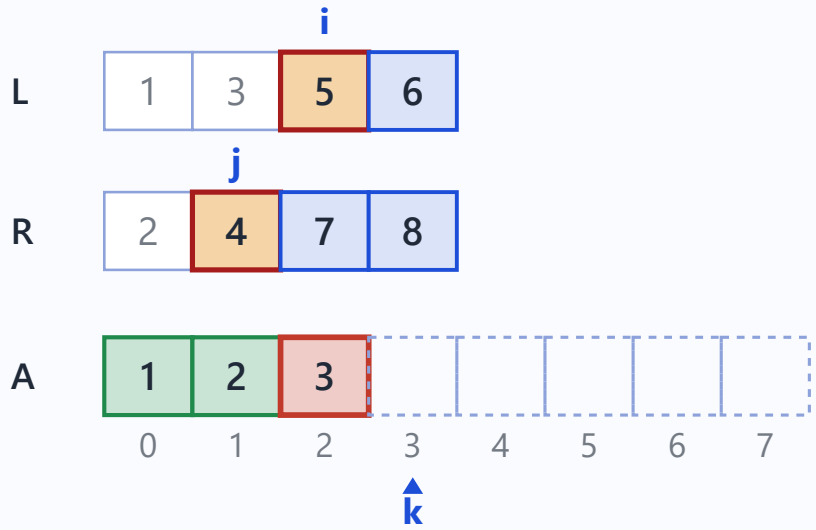


Figure 3.15: The state after the third step. 1 and 3 have been taken from `L` and 2 from `R` (faded cells). The next comparison is between `L[2] = 5` and `R[1] = 4`; `k = 3`.

3.6.6 Analysis

The running time of merge sort is expressed with a **recurrence**. Let $T(n)$ be the time to sort an array of n elements. The algorithm solves two subproblems of size $n/2$ and merges the results in $\Theta(n)$ time. The divide step takes constant time. This gives the recurrence

> $T(n) = 2\,T(n/2) + \Theta(n),\ T(1) = \Theta(1)$

The most intuitive way to solve this recurrence is the **recursion tree**. At the root there is one problem of size n, and the merge work is n. One level down there are two problems of size $n/2$; each does $n/2$ work, and the level's total is again n. One more level down there are four of size $n/4$; the total is again n. At each level the number of pieces doubles, the piece size halves, and the total work stays the same. There are $\log_2 n$ levels until the piece size drops to 1. The total work is n per level times the number of levels, that is, $n \log_2 n$.

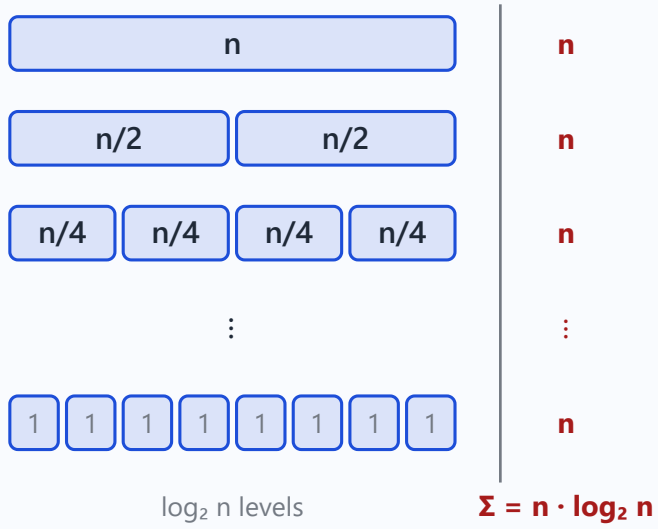


Figure 3.16: The recursion tree of merge sort. The total merge work at each level is n; there are $\log_2 n$ levels, and the total is $n \cdot \log_2 n$.

The same result follows from the **master theorem**. The recurrence has the form $T(n) = aT(n/b) + f(n)$, here with $a = 2$, $b = 2$, and $f(n) = \Theta(n)$. We get $n^{\log_b a} = n^{\log_2 2} = n$, and $f(n) = \Theta(n^{\log_b a})$ is the second case of the theorem. In this case $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.

Merge sort does not look at the arrangement of the input; in every case it splits the array the same way and performs the same merges. So the best, average, and worst cases coincide: $\Theta(n \log n)$. The number of comparisons varies a little with the input, because when one array ends early the rest is copied without comparison, but this variation does not affect the growth rate.

As for extra memory, merge sort is not in place. During a merge, `L` and `R` together hold n elements; the extra memory is $\Theta(n)$. To this are added the $\Theta(\log n)$ frames of the recursion stack. For large data sets this extra memory is a significant cost, and it is one of the reasons quicksort is preferred in practice.

Example 3.4

What is the maximum number of comparisons merge sort needs to sort an 8-element array?

1. **Levels.** For 8 elements the recursion tree has three levels of merging: four merges of 2 elements, two merges of 4 elements, and one merge of 8 elements.
2. **One merge.** Merging m elements takes at most $m - 1$ comparisons: each comparison writes one element, and the last element is written without a comparison.
3. **Total.** $4 \cdot 1 + 2 \cdot 3 + 1 \cdot 7 = 4 + 6 + 7 = 17$.

The answer is 17. When the code is run on all $8! = 40{,}320$ permutations, the maximum number of comparisons is indeed 17. The rough upper estimate is $n \log_2 n = 24$; compared with the $n(n - 1)/2 = 28$ comparisons of the simple algorithms, the difference shows even at

small n.

A by-product of merge sort is **counting inversions**. When an element is taken from `R` during a merge, all the elements remaining in `L` are larger than it, and each forms an inversion with it. We add `L.length - i` to a counter. The inversions within the left and right halves are counted recursively. The total work is the same as for merge sort: $\Theta(n \log n)$. The brute-force method, which checks every pair, takes $\Theta(n^2)$. For example, in the array [3, 1, 2, 5, 4] the inversions are (3, 1), (3, 2), and (5, 4); there are 3.

3.7 Quicksort

Quicksort also uses divide and conquer, but it moves the work to a different place. In merge sort, dividing is easy and merging is hard. In quicksort, dividing is hard and no merging is needed at all. It is $\Theta(n \log n)$ on average, works in place, and is cache-friendly; for these reasons it is usually the fastest comparison sort in practice.

3.7.1 The idea

Quicksort is an algorithm that C. A. R. Hoare developed in 1959 and published in 1961–1962. It consists of three steps:

1. A **pivot** element is chosen from the array. Call the pivot x.
2. The array is split around the pivot into two parts: elements less than or equal to x on the left, elements greater than or equal to x on the right. The pivot is placed between the two. This operation is called **partitioning**.
3. The two parts are sorted recursively.

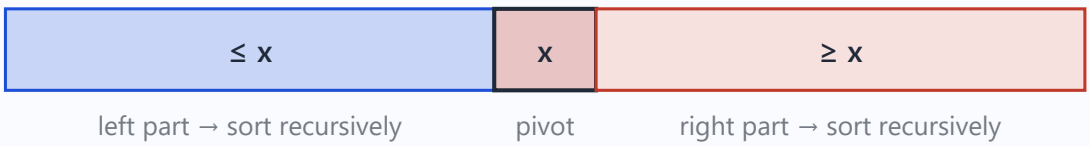


Figure 3.17: The array after partitioning: elements less than or equal to the pivot on the left, the pivot x in the middle, elements greater than or equal to it on the right. The left and right parts are sorted separately, recursively.

Partitioning has an important consequence: after partitioning, the pivot is in its final position. All elements to its left are less than or equal to it and those to its right are greater than or equal to it; in the sorted array it will stand at exactly that index. When the two parts are sorted, elements never cross the boundary between the parts. So no extra work is needed to combine the two sorted parts: the array is sorted in place.

3.7.2 The partition procedure

There are several ways to partition. These notes use the method named after N. Lomuto. The last element of the range is chosen as the pivot, and the array is scanned in a single pass.

```
partition(A, p, q):
  pivot = A[q]        // choose the pivot
  i = p - 1           // end of the "<= pivot" region
  for j = p to q - 1:
    if A[j] <= pivot:
      i = i + 1
      swap(A[i], A[j])
  swap(A[i + 1], A[q])  // put the pivot in place
  return i + 1          // index of the pivot
```

The procedure works with two indices. `i` is the last cell of the "less than or equal to the pivot" region; the region is empty at first, so `i = p - 1`. `j` is the next element to examine. At each step `A[j]` is compared with the pivot. If it is larger, only `j` advances; the element has joined the "larger" region. If it is smaller or equal, `i` is incremented and `A[i]` and `A[j]` are swapped. The element is thereby taken to the end of the smaller region, and the first element of the larger region moves into `j`'s place. When the scan ends, the pivot is swapped with `A[i + 1]`, the first cell of the larger region, and its index is returned.

The correctness of the procedure rests on the following loop invariant: at the start of each step,

- the elements in `A[p..i]` are less than or equal to the pivot,
- the elements in `A[i+1..j-1]` are greater than the pivot,
- `A[j..q-1]` has not yet been examined and `A[q]` is the pivot.

Initially the first two regions are empty and the invariant holds trivially. Each step preserves the invariant by adding the examined element to the right region. When the loop ends, the unexamined region is empty, and moving the pivot to `A[i+1]` puts the array in the desired form. The scan is a single pass: for a range of n elements it makes $n - 1$ comparisons, the time is $\Theta(n)$, and the extra memory is $O(1)$.

The same idea can be implemented with the first element as the pivot. In this version the pivot stays in `A[p]`, the scan starts at `p + 1`, and at the end the pivot is swapped with the last element of the smaller region.

```
PARTITION(A, p, q):
  x = A[p]            // pivot
  i = p
  for j = p + 1 to q:
    if A[j] <= x:
      i = i + 1
      swap(A[i], A[j])
  swap(A[p], A[i])
  return i
```

3.7.3 Partition example: the first element as pivot

Let us trace the version that takes the first element as the pivot on the array [6, 10, 13, 5, 8, 3, 2, 11]. The pivot is $x = 6$; `i = p = 0`, and `j` starts at 1.

`j`	`A[j]`	`A[j] ≤ 6 ?`	`i`	**Array**
1	10	no	0	[6, 10, 13, 5, 8, 3, 2, 11]
2	13	no	0	[6, 10, 13, 5, 8, 3, 2, 11]
3	5	yes: `A[1] ↔ A[3]`	1	[6, 5, 13, 10, 8, 3, 2, 11]
4	8	no	1	[6, 5, 13, 10, 8, 3, 2, 11]
5	3	yes: `A[2] ↔ A[5]`	2	[6, 5, 3, 10, 8, 13, 2, 11]
6	2	yes: `A[3] ↔ A[6]`	3	[6, 5, 3, 2, 8, 13, 10, 11]
7	11	no	3	[6, 5, 3, 2, 8, 13, 10, 11]
last	—	`A[0] ↔ A[3]`	3	[2, 5, 3, **6**, 8, 13, 10, 11]

Table 3.16: Partitioning [6, 10, 13, 5, 8, 3, 2, 11] with `PARTITION` (first element as pivot).

During the scan, the front of the array is divided into three regions: the pivot, the smaller region, and the larger region. The first two elements (10 and 13) are larger and join the larger region. When 5 arrives, the smaller region grows by one and 5 is swapped with 10, the first element of the larger region. The same happens for 3 and 2. When the scan ends, `i = 3`. The pivot is swapped with 2, the last element of the smaller region. In the end the pivot 6 settles at index 3; the 2, 5, 3 to its left are smaller and the 8, 13, 10, 11 to its right are larger. Note that the parts are not sorted internally; partitioning only puts the pivot in place.

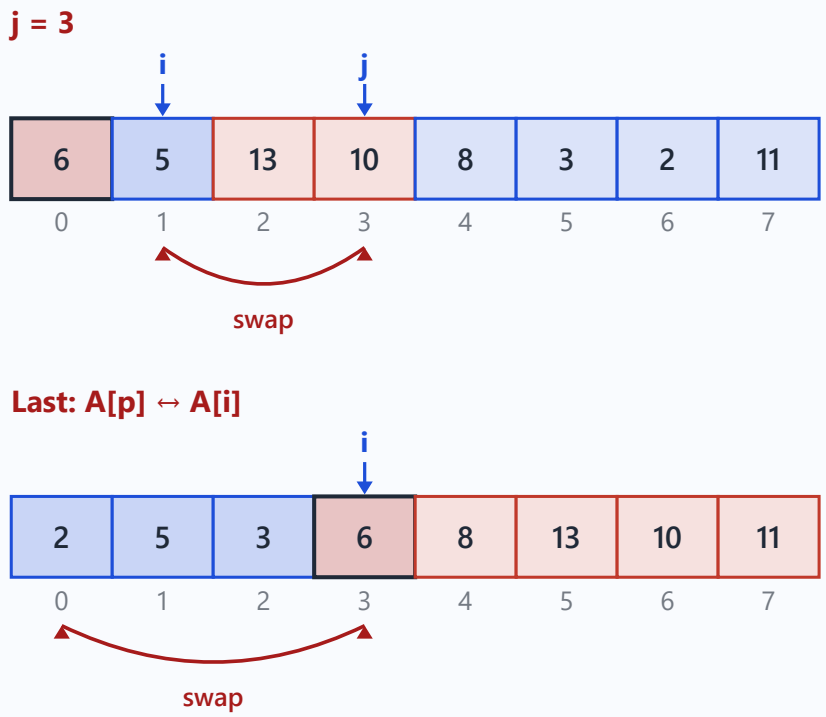


Figure 3.18: (a) `j = 3`: since 5 ≤ 6, `i` becomes 1 and `A[1]` and `A[3]` are swapped. Blue: ≤ pivot, red: > pivot, white: not yet examined. (b) The final swap: the pivot 6 trades places with the last element of the smaller region and lands at index 3.

3.7.4 Code trace: partition()

The Java version of Lomuto partitioning follows. `swap(a, x, y)` is a helper method that exchanges the values of two cells with three assignments.

```
int partition(int[] a, int lo, int hi) {
  int pivot = a[hi];
  int i = lo - 1;
  for (int j = lo; j < hi; j++) {
    if (a[j] <= pivot) {
      i++;
      swap(a, i, j);
    }
  }
  swap(a, i + 1, hi);
  return i + 1;
}
```

Let us trace the method on [2, 8, 7, 1, 3, 5, 6, 4] with the call `partition(a, 0, 7)`. The pivot is `a[7] = 4` and initially `i = -1`.

`j`	`a[j] ≤ 4`?	`i`	**Action**	`a`
0	2: yes	0	`swap(0, 0)` (with itself)	[2, 8, 7, 1, 3, 5, 6, 4]
1	8: no	0	—	[2, 8, 7, 1, 3, 5, 6, 4]
2	7: no	0	—	[2, 8, 7, 1, 3, 5, 6, 4]
3	1: yes	1	`swap(1, 3)`	[2, 1, 7, 8, 3, 5, 6, 4]
4	3: yes	2	`swap(2, 4)`	[2, 1, 3, 8, 7, 5, 6, 4]
5	5: no	2	—	[2, 1, 3, 8, 7, 5, 6, 4]
6	6: no	2	—	[2, 1, 3, 8, 7, 5, 6, 4]
last	—	2	`swap(3, 7)`, `return 3`	[2, 1, 3, **4**, 7, 5, 6, 8]

Table 3.17: Trace of `partition(a, 0, 7)`, `a = [2, 8, 7, 1, 3, 5, 6, 4]`.

The first element, 2, is smaller than the pivot; `i` becomes 0 and the element is swapped with itself. This is harmless but an unnecessary swap. 8 and 7 are larger and only `j` advances. When 1 arrives, `i = 1` and 1 is swapped with 8, the first element of the larger region. The same happens for 3: 3 and 7 trade places. 5 and 6 are larger. The loop ends at `j = hi`; the pivot is moved to index `i + 1 = 3` and 3 is returned. The elements in the left part `[0..2]` are smaller than 4, and those in the right part `[4..7]` are larger than 4. There are 7 comparisons in all, which is $n - 1 = 7$ for a range of 8 elements.

3.7.5 The recursive structure

Once partitioning is available, quicksort itself is a few lines. The pseudocode is written with 1-based indices:

```
QUICKSORT(A, p, r)
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)

Initial call: QUICKSORT(A, 1, n)
```

The Java version is 0-based, and the initial call for the whole array is `quickSort(a, 0, n - 1)`.

```
void quickSort(int[] a, int lo, int hi) {
  if (lo < hi) {
    int p = partition(a, lo, hi);
    quickSort(a, lo, p - 1);
    quickSort(a, p + 1, hi);
  }
}
```



Figure 3.19: After the `partition` call, the pivot is at index `p`, in its final place. The left part `lo … p - 1` and the right part `p + 1 … hi` are sorted by two separate recursive calls.

Two points in the code deserve attention. First, the pivot is included in neither recursive call; the calls end at `p - 1` and start at `p + 1`. The pivot has settled in its place and is never touched again. Second, the base case is that the condition `lo < hi` is false: the range has zero or one element, and such a range is already sorted. When the pivot is the smallest or the largest of its part, an empty part results; for example `quickSort(a, 0, -1)`. That call returns immediately.

Common mistake

Including the pivot in the recursion, that is, writing `quickSort(a, lo, p)`. If the pivot is the largest of its part, the left part is again the entire range and never shrinks. For example, in a sorted two-element range the pivot is the largest, the left part is again two elements, and the same call repeats forever. The result is a `StackOverflowError`.

3.7.6 A complete run: [6, 10, 13, 5, 8, 3, 2, 11]

Table 3.16 showed only the first partition. Let us trace the whole sort, again with the first element as pivot, together with the depth of the recursion.

Depth	Range	Piece	Pivot → index	Next calls
0	[0..7]	[6, 10, 13, 5, 8, 3, 2, 11]	6 → 3	[0..2], [4..7]
1	[0..2]	[2, 5, 3]	2 → 0	[0..−1], [1..2]
2	[1..2]	[5, 3]	5 → 2	[1..1], [3..2]
3	[1..1]	[3]	base case	—
1	[4..7]	[8, 13, 10, 11]	8 → 4	[4..3], [5..7]
2	[5..7]	[13, 10, 11]	13 → 7	[5..6], [8..7]
3	[5..6]	[11, 10]	11 → 6	[5..5], [7..6]
4	[5..5]	[10]	base case	—

Table 3.18: Quicksort of [6, 10, 13, 5, 8, 3, 2, 11] (first element as pivot). The rows are in the order the calls occur.

In the table, empty ranges (such as `[0..-1]` and `[3..2]`) are not listed as separate rows; those calls return immediately. When all calls are finished, the array is [2, 3, 5, 6, 8, 10, 11, 13]. There is no merge step: each pivot settled in its own partitioning, and the parts were sorted in place. On the right side the recursion goes four levels deep, because the pivots 8 and 13 are the smallest and the largest of their parts; each splits off only itself from its part.

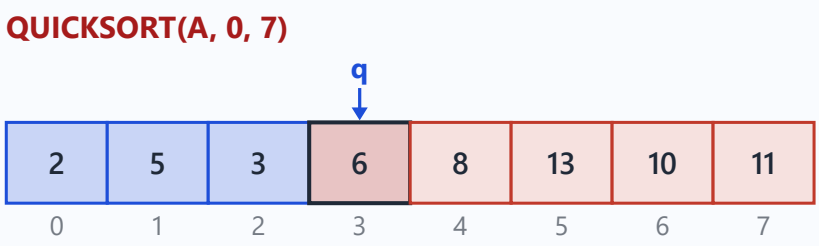


Figure 3.20: The array after the partition at depth 0. The pivot 6 has settled at index 3 (`q`); the blue left part `[0..2]` and the red right part `[4..7]` will be sorted separately.

3.7.7 Call trace: [5, 2, 8, 1, 9, 3]

The calls of the Java code (last element as pivot) on a six-element array are given in order in the table below. Each row is one `quickSort` call.

Call	Status	Pivot → `p`	`a` (after the call)
`(0, 5)`	`lo < hi`	3 → 2	[2, 1, 3, 5, 9, 8]
`(0, 1)`	`lo < hi`	1 → 0	[1, 2, 3, 5, 9, 8]
`(0, -1)`	empty part, base case	—	unchanged
`(1, 1)`	one element, base case	—	unchanged
`(3, 5)`	`lo < hi`	8 → 4	[1, 2, 3, 5, 8, 9]
`(3, 3)`	one element, base case	—	unchanged
`(5, 5)`	one element, base case	—	unchanged

Table 3.19: Trace of the calls of `quickSort(a, 0, 5)`, `a = [5, 2, 8, 1, 9, 3]`.

In the first call the pivot is 3; the elements smaller than 3, namely 2 and 1, are gathered on the left, 5, 9, 8 stay on the right, and the pivot settles at index 2. Then the left part `(0, 1)` is sorted: pivot 1, empty left, right [2]. Empty and one-element parts return immediately. When the left side is done, the second recursive line of the first call runs and `(3, 5)` is called: pivot 8, with [5] on its left and [9] on its right. There are 3 `partition` calls and 4 base-case calls in all; the result is [1, 2, 3, 5, 8, 9].

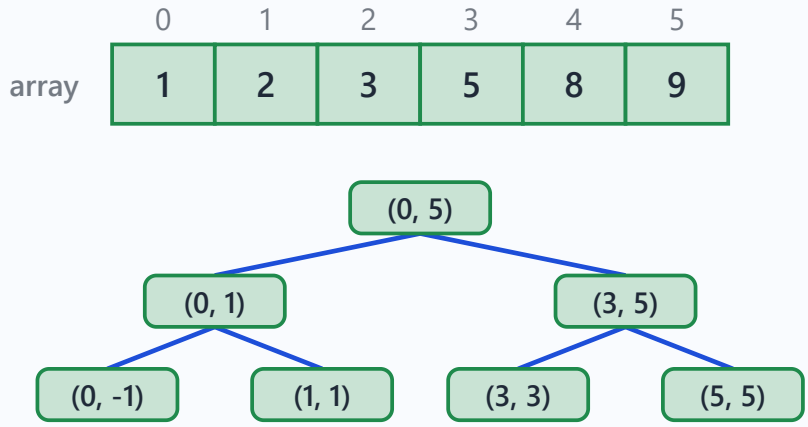


Figure 3.21: The `quickSort` call tree. The root is `(0, 5)`, with children `(0, 1)` and `(3, 5)`; the leaves are base-case calls on empty or one-element parts. The sorted array is shown at the top.

3.7.8 Analysis and pivot choice

The running time of quicksort depends on how evenly the pivot splits the array. Each partition does work linear in the size of the range. The balance of the splits determines the depth of the recursion tree.

Best case. If the pivot is the median of its part every time, the array splits into two equal halves. The recurrence is the same as for merge sort: $T(n) = 2T(n/2) + \Theta(n)$, with solution $\Theta(n \log n)$.

Worst case. If the pivot is the smallest or the largest of its part every time, one part is empty and the other has $n - 1$ elements. The recurrence becomes $T(n) = T(n - 1) + \Theta(n)$. Unrolled, it gives $n + (n - 1) + \ldots + 1$, that is, $\Theta(n^2)$. Surprisingly, this happens on the "easiest" input: on an already sorted array, the version that takes the last element as pivot picks the largest every time. On a sorted array of 1000 elements the number of comparisons is $999 + 998 + \ldots + 1 = 499{,}500$, and the recursion goes 1000 levels deep. On very large arrays this depth overflows the call stack and gives a `StackOverflowError`.

Average case. On a random input the pivot usually lands "central enough". Even in the bad-looking case where the pivot always splits 1 to 9, the depth of the recursion tree is $\log_{10/9} n$, which is still logarithmic; the total work stays $O(n \log n)$. A detailed analysis shows that the expected number of comparisons is about $2n \ln n \approx 1.39\, n \log_2 n$ for large n. For small n the lower-order terms matter as well. The exact expected value for Lomuto partitioning is $2(n + 1)H_n - 4n$ (H_n is the harmonic number), which comes to about 10,986 for $n = 1000$. That is about 26% more than the number of comparisons of merge sort. Even so, quicksort is usually faster in practice: its inner loop is very simple, it copies no auxiliary array, and because it scans the array sequentially it uses the processor cache well.

Case	When?	Recurrence	Time
Best	pivot is the median every time	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \log n)$

Case	When?	Recurrence	Time
Average	random input or random pivot	expected $\approx 1.39\, n \log_2 n$ comparisons	$\Theta(n \log n)$
Worst	pivot is always the smallest or the largest (e.g., sorted array with the last element as pivot)	$T(n) = T(n-1) + \Theta(n)$	$\Theta(n^2)$

Table 3.20: The cases of quicksort.

To avoid the worst case, the choice of pivot is improved.

- **Random pivot.** The pivot is chosen at random from the range and swapped with the last cell before partitioning. No fixed input can then produce the worst case every time; $\Theta(n^2)$ happens only with very small probability, through unlucky random choices. The expected time is $\Theta(n \log n)$ on every input.
- **Median of three.** The median of the first, middle, and last elements becomes the pivot. On sorted and reverse-sorted inputs this choice gives exactly the middle and prevents the bad case, but on specially constructed inputs it can still be $\Theta(n^2)$.
- **Small parts.** When the part size becomes small (roughly $n \lesssim 16$), switching to insertion sort speeds things up in practice. Also, if one recurses on the smaller part first and handles the larger part with a loop, the recursion depth stays $O(\log n)$ even in the worst case.

Quicksort is in place: it uses extra memory only for the recursion stack, $O(\log n)$ on average. It is not stable; the long-distance swaps in partitioning can disturb the order of equal elements. Java's `Arrays.sort(int[])` uses a version of quicksort with two pivots (dual-pivot quicksort) on primitive arrays. Stability does not matter for primitive types, because two equal `int` values cannot be told apart.

> **Common mistake**
>
> Saying "quicksort is always $n \log n$". With a fixed pivot rule (first or last element), sorted and reverse-sorted inputs give $\Theta(n^2)$ time and a recursion of depth n. In real applications data often arrives partly sorted, so this case is not merely theoretical; use a random pivot or median of three.

3.8 Heap sort

Merge sort is $\Theta(n \log n)$ even in the worst case but needs extra memory. Quicksort is in place but quadratic in the worst case. Heap sort combines the good sides of both: it takes $O(n \log n)$ time in the worst case and works in place with $O(1)$ extra memory. It does so by using the array itself as a data structure.

3.8.1 The heap and its array representation

A **heap** is a complete binary tree, with every level full except possibly the last, which is filled from left to right, and it satisfies an ordering condition. In a **max-heap** every node is greater than or equal to its children. Since this holds everywhere in the tree, the largest element is at the root. The heap is familiar from data structures as the priority queue; here we look only at how it is used in sorting.

The most useful property of a heap is that the tree need not be stored as a separate structure. A complete binary tree is stored in an array level by level, from left to right. With 0-based indices, the left child of `a[i]` is at `2i + 1`, the right child at `2i + 2`, and the parent at `(i - 1) / 2` (integer division). It is therefore possible to move around the tree with index arithmetic alone, without pointers. This is exactly why heap sort is in place: the array to be sorted is itself the heap.

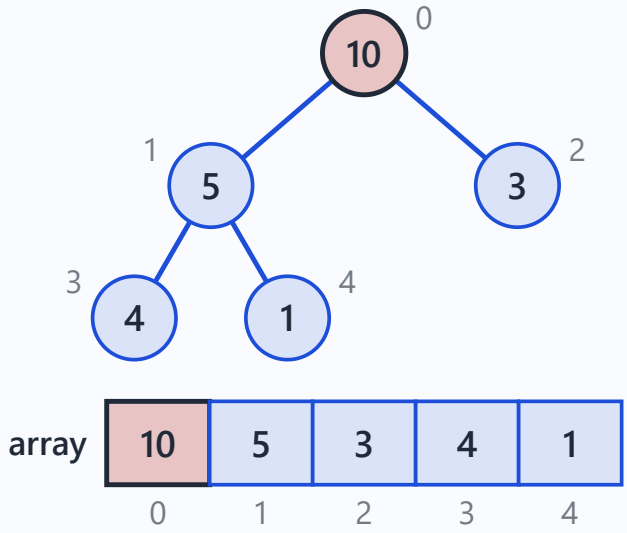


Figure 3.22: The max-heap [10, 5, 3, 4, 1] as a tree and as an array. The root is 10 (index 0); its children are 5 (index 1) and 3 (index 2), and the children of 5 are 4 (index 3) and 1 (index 4). The small numbers next to the nodes are array indices.

Heap sort works in two phases:

1. The array is turned into a max-heap.
2. The root, that is, the largest element, is swapped with the last element of the heap. That element now belongs to the sorted region at the end of the array, and the heap shrinks by one. Since the element that arrived at the root may violate the heap condition, it sinks down and the heap is restored. This step is repeated until one element is left in the heap.

Both phases rest on the same operation: **sifting a node down**. If the node is smaller than one of its children, it is swapped with its larger child, and the same check is repeated at its new position. Sifting ends when the node is greater than or equal to both children or when it reaches a leaf. The precondition for this operation is that both subtrees of the node are already heaps.

3.8.2 Example: [4, 10, 3, 5, 1]

Let us see the whole algorithm on a small array. Table 3.21 shows, at each step, the heap part and the sorted part of the array separately.

Step	Array (heap | sorted)	Explanation
Input	[4, 10, 3, 5, 1]	not a heap yet
Build max-heap	[10, 5, 3, 4, 1]	the root is the largest element
Root ↔ `a[4]`	[1, 5, 3, 4 | 10]	10 moved to the sorted region
Restore max-heap	[5, 4, 3, 1 | 10]	1 sifted down (1 ↔ 5, 1 ↔ 4)
Root ↔ `a[3]`	[1, 4, 3 | 5, 10]	5 moved to the sorted region
Restore max-heap	[4, 1, 3 | 5, 10]	1 sifted down (1 ↔ 4)
Root ↔ `a[2]`	[3, 1 | 4, 5, 10]	4 moved to the sorted region
Restore max-heap	[3, 1 | 4, 5, 10]	3 is already larger; no change
Root ↔ `a[1]`	[1 | 3, 4, 5, 10]	3 moved to the sorted region
Result	[1, 3, 4, 5, 10]	the one-element heap is in place too

Table 3.21: Sorting [4, 10, 3, 5, 1] with heap sort. In the array column a vertical bar separates the heap from the sorted region.

In the heap-building step, the non-leaf nodes are processed from bottom to top. The 10 at index 1 is larger than its children (5 and 1); nothing changes. The 4 at the root is swapped with its larger child, 10. At its new position (index 1) its children are 5 and 1; 4 is swapped with 5 and sinks to a leaf. The result is [10, 5, 3, 4, 1]. In each later round the largest element of the heap is moved to the end; the sorted region fills from right to left, from largest to smallest.

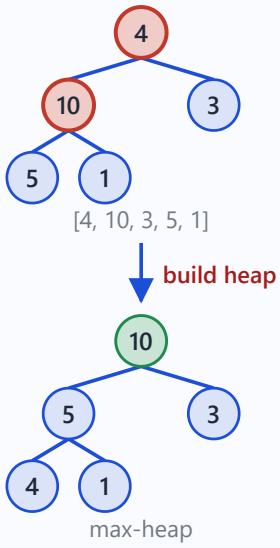


Figure 3.23: The tree of [4, 10, 3, 5, 1] turning into the max-heap [10, 5, 3, 4, 1]. In the upper tree the root 4 is smaller than its child 10; in the lower tree every parent is larger than its children.

3.8.3 Phase one: building a max-heap

Consider the eight-element array [6, 5, 3, 1, 8, 7, 2, 4]. Viewed as a complete binary tree, it is not yet a heap: for example, the 5 at index 1 is smaller than its child 8.

The efficient way to build a heap is to work from the bottom up. The leaves (index $\lfloor n/2 \rfloor$ and beyond, here 4–7) are one-element heaps already; nothing needs to be done with them. Processing starts at the last internal node, index $n/2 - 1 = 3$, and proceeds toward the root. Each node is sifted down. When a node is processed, its two subtrees have been processed earlier, so they are already heaps; the precondition of sifting is therefore always met.

Step	Node	Swaps	Array
`heapify(3)`	1 (child 4)	1 ↔ 4	[6, 5, 3, 4, 8, 7, 2, 1]
`heapify(2)`	3 (children 7, 2)	3 ↔ 7	[6, 5, 7, 4, 8, 3, 2, 1]
`heapify(1)`	5 (children 4, 8)	5 ↔ 8	[6, 8, 7, 4, 5, 3, 2, 1]
`heapify(0)`	6 (children 8, 7)	6 ↔ 8	[8, 6, 7, 4, 5, 3, 2, 1]

Table 3.22: Bottom-up max-heap construction on [6, 5, 3, 1, 8, 7, 2, 4].

After the 6 at the root is swapped with its larger child 8, it sinks to index 1. There its children are 4 and 5; since 6 is larger than both, the sifting stops. When all internal nodes have been processed, the array is [8, 6, 7, 4, 5, 3, 2, 1]. This is a max-heap, and the largest element, 8, is at the root. Note that a heap is not a sorted array; it only guarantees that every parent is larger than its children.

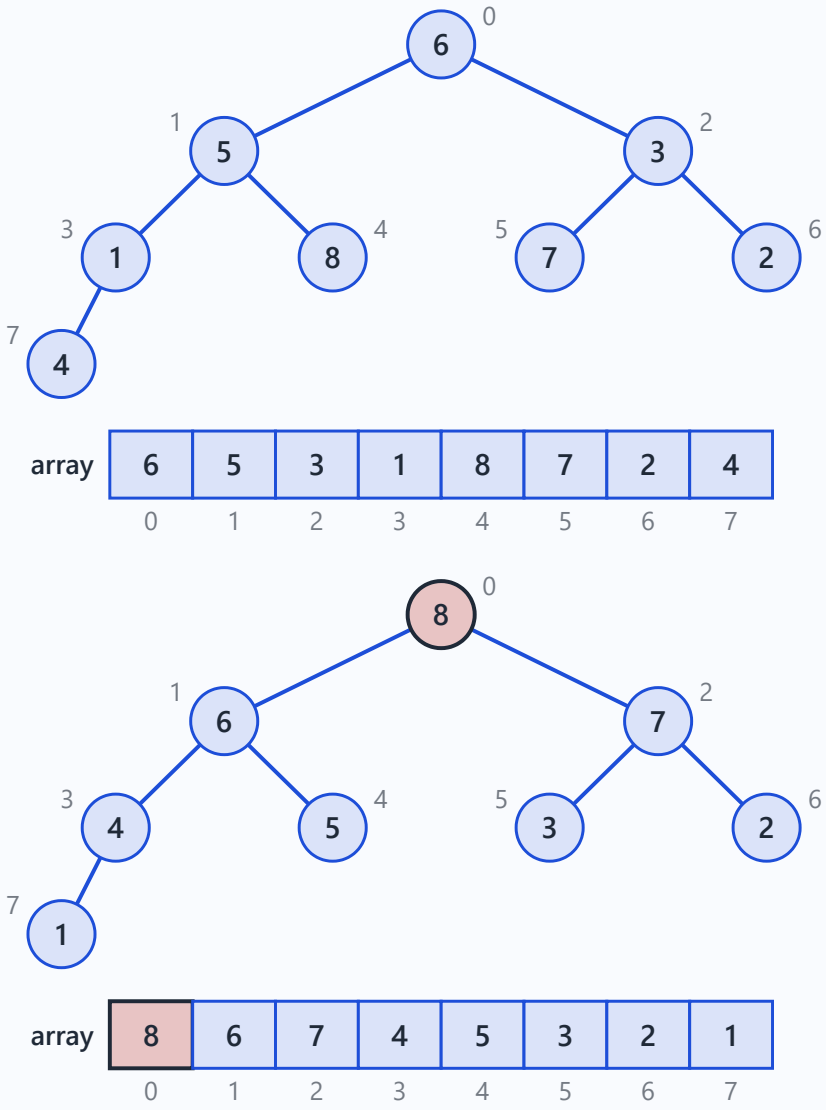


Figure 3.24: (a) The start: [6, 5, 3, 1, 8, 7, 2, 4] as a complete binary tree; not yet a heap. (b) After the non-leaf nodes (indices 3, 2, 1, 0) are processed, the max-heap [8, 6, 7, 4, 5, 3, 2, 1]; the largest element is at the root.

3.8.4 Phase two: moving the root to the end

The max-heap [8, 6, 7, 4, 5, 3, 2, 1] is ready. In each round the root is swapped with the last element of the heap; the heap shrinks by one and the element that arrived at the root sifts down.

Round	Root ↔ last	Sift	Heap	Sorted region
1	8 ↔ 1	1 ↔ 7, 1 ↔ 3	[7, 6, 3, 4, 5, 1, 2]	[8]
2	7 ↔ 2	2 ↔ 6, 2 ↔ 5	[6, 5, 3, 4, 2, 1]	[7, 8]
3	6 ↔ 1	1 ↔ 5, 1 ↔ 4	[5, 4, 3, 1, 2]	[6, 7, 8]
4	5 ↔ 2	2 ↔ 4	[4, 2, 3, 1]	[5, 6, 7, 8]
5	4 ↔ 1	1 ↔ 3	[3, 2, 1]	[4, 5, 6, 7, 8]
6	3 ↔ 1	1 ↔ 2	[2, 1]	[3, 4, 5, 6, 7, 8]

Round	Root ↔ last	Sift	Heap	Sorted region
7	2 ↔ 1	—	[1]	[2, 3, 4, 5, 6, 7, 8]

Table 3.23: The second phase of heap sort. The "Sift" column gives the swaps made by the element that arrived at the root.

In round 1, 8 moves to the last cell (index 7) and 1 arrives at the root. The children of 1 are 6 and 7; it is swapped with the larger, 7, and sinks to index 2. There its children are 3 and 2; it is swapped with 3 and reaches a leaf. The heap is [7, 6, 3, 4, 5, 1, 2]. The later rounds proceed in the same way. After seven rounds one element is left in the heap; it is the smallest and is in place at index 0. The sorted array is [1, 2, 3, 4, 5, 6, 7, 8].

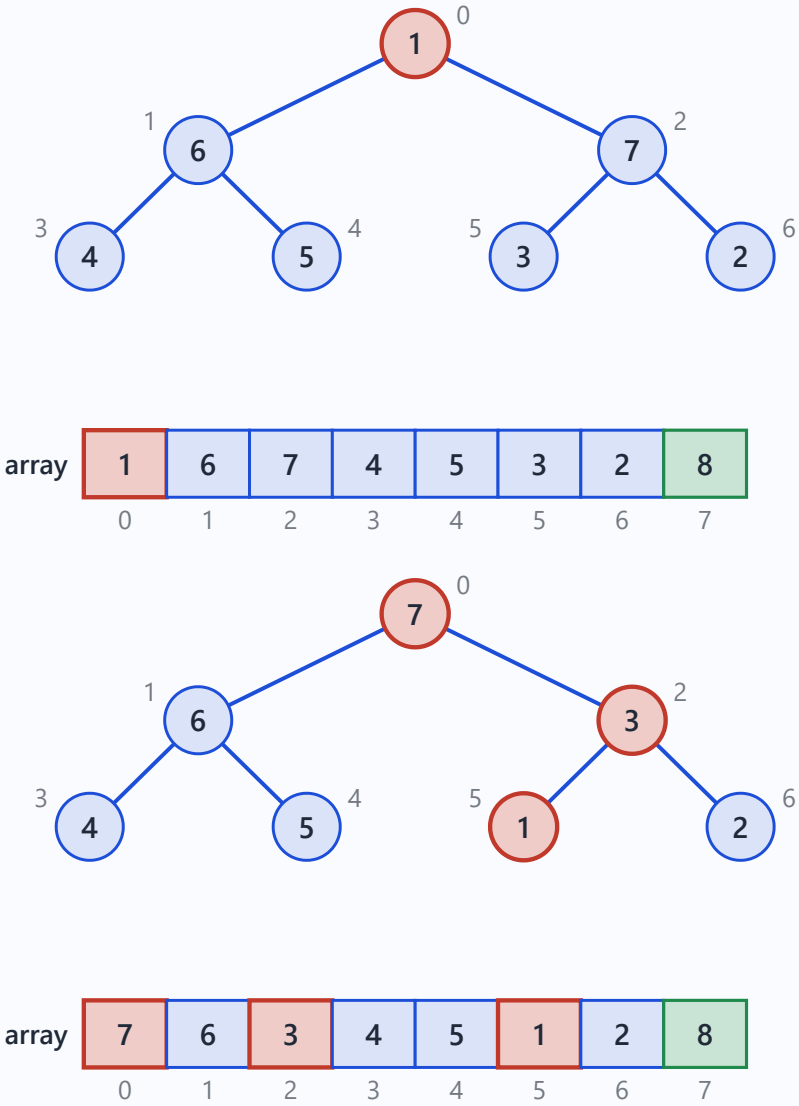


Figure 3.25: Round 1 of the second phase. (a) The root 8 has been swapped with `a[7] = 1`; 8 has moved to the sorted region (green), and the heap size is 7. (b) The 1 at the root has sifted down (1 ↔ 7, 1 ↔ 3); the heap is [7, 6, 3, 4, 5, 1, 2].

3.8.5 Java code and analysis

```
void heapSort(int[] a) {
  int n = a.length;
  for (int i = n / 2 - 1; i >= 0; i--)   // 1) build the heap
    heapify(a, n, i);
  for (int i = n - 1; i > 0; i--) {      // 2) sort
    swap(a, 0, i);                       // max to the end
    heapify(a, i, 0);                    // sift the root
  }
}
void heapify(int[] a, int n, int i) {
  while (true) {
    int largest = i, left = 2 * i + 1, right = 2 * i + 2;
    if (left < n && a[left] > a[largest]) largest = left;
    if (right < n && a[right] > a[largest]) largest = right;
    if (largest == i) return;         // in place
    swap(a, i, largest);
    i = largest;                       // keep sinking
  }
}
```

- `heapSort` consists of two loops. The first builds the heap: `i` decreases from the last internal node `n / 2 - 1` to the root, and `heapify(a, n, i)` is called for each node.
- The second loop sorts: `i` decreases from the last index to 1. In each round the root is swapped with `a[i]`. Then `heapify(a, i, 0)` is called; the second argument `i` says that the heap now consists only of the first `i` elements. The elements already moved to the end are thus not reused in sifting.
- `heapify` sifts node `i` down in a heap of size `n`. First the index of the largest among `i` and its left and right children is found in `largest`. The checks `left < n` and `right < n` ask whether the child is inside the heap.
- If `largest == i`, the node is in place and the method returns. If not, the node is smaller than one of its children; the two are swapped, `i = largest` is set, and the `while` loop continues sifting at the child's position.

The cost of building the heap. The build loop sifts about $n/2$ nodes, and a single sift takes $O(\log n)$. This gives a crude $O(n \log n)$ bound. But the bound is loose, because most nodes are low in the tree. A node of height h sifts at most h steps. The number of nodes of height h is at most $\lceil n/2^{h+1} \rceil$: there are about $n/2$ leaves, $n/4$ nodes one level up, and so on. The total work is on the order of

> $\Sigma_{h \geq 0} (n/2^{h+1}) \cdot h = (n/2) \cdot \Sigma_{h \geq 0} h/2^h = (n/2) \cdot 2 = n$

That is, bottom-up heap construction is $O(n)$. This method is also called Floyd's method, after R. W. Floyd. A heap can also be built by inserting the elements one at a time and sifting each one up; that also yields a valid max-heap, but it takes $O(n \log n)$ in the worst case.

The cost of the sorting phase. The second loop runs $n - 1$ times, and in each round the root sifts down at most the height of the heap, that is, $O(\log n)$ steps. This phase is $O(n \log n)$ and determines the total time. Even in the worst case the time is $O(n \log n)$.

Memory and stability. Heap sort uses only a few variables; it works in place with $O(1)$ extra memory. `heapify` is written with a loop, not recursion, so the call stack does not grow either. Heap sort is not stable: swapping the root with the last element can make an element jump over others with an equal key. It is not adaptive either; even a sorted input is first turned into a heap and still takes $\Theta(n \log n)$.

> **Common mistake**
>
> Writing `heapify(a, n, 0)` in the second loop. If the heap size is not reduced, sifting counts the large elements already moved to the sorted region as children again and brings them back to the root. The sort breaks. The correct call is `heapify(a, i, 0)`; the heap is only `a[0..i-1]`.

3.8.6 Code trace: building the heap

Here is the heap-building loop run on [3, 9, 2, 1, 4, 5], line by line. Here $n = 6$; the leaves are indices 3, 4, and 5. The loop starts at $i = n/2 - 1 = 2$.

```
for (int i = n / 2 - 1; i >= 0; i--)
  heapify(a, n, i);

void heapify(int[] a, int n, int i) {
  while (true) {
    int largest = i, left = 2 * i + 1, right = 2 * i + 2;
    if (left < n && a[left] > a[largest]) largest = left;
    if (right < n && a[right] > a[largest]) largest = right;
    if (largest == i) return;         // in place
    swap(a, i, largest);
    i = largest;                      // keep sinking
  }
}
```

Call / round	Checks	`largest`	Action	`a`
`heapify(6, 2)`	left = 5 (5) > 2: yes; right = 6 ≥ n	5	`swap(2, 5)`	[3, 9, 5, 1, 4, 2]
↳ `i = 5` (loop continues)	left = 11 ≥ n	5	sifting ends	[3, 9, 5, 1, 4, 2]
`heapify(6, 1)`	left = 3 (1) > 9: no; right = 4 (4) > 9: no	1	no change	[3, 9, 5, 1, 4, 2]
`heapify(6, 0)`	left = 1 (9) > 3: yes; right = 2 (5) > 9: no	1	`swap(0, 1)`	[9, 3, 5, 1, 4, 2]
↳ `i = 1` (loop continues)	left = 3 (1) > 3: no; right = 4 (4) > 3: yes	4	`swap(1, 4)`	[9, 4, 5, 1, 3, 2]

Call / round	Checks	`largest`	Action	`a`
↳ `i = 4` (loop continues)	left = 9 ≥ n	4	sifting ends	[9, 4, 5, 1, 3, 2]

Table 3.24: Trace of heap construction for `a = [3, 9, 2, 1, 4, 5]`. The numbers in parentheses are the values at those indices.

At `i = 2`, the only child of 2 is 5 (index 5); since 5 is larger, they are swapped and sifting continues at index 5. Index 5 is a leaf; sifting ends. At `i = 1`, 9 is larger than its children (1 and 4); nothing changes. At `i = 0`, 3 is swapped with its larger child 9 and sinks to index 1. There its children are 1 and 4; it is swapped with 4 and sinks to a leaf. The loop ends at `i = -1`. The result is [9, 4, 5, 1, 3, 2]: every parent is larger than its children, and the largest element, 9, is at the root.

Note that the check for `right` is made against `a[largest]`, not against `a[i]`. In the `i = 0` step, the right child 5 is larger than the node's own value 3 but smaller than the left child 9. The correct choice is the larger child, 9. Had the right child been compared with `a[i]`, 5 would have been chosen; 5 would rise to the root with 9 left below it, and the heap condition would be broken.

3.9 Linear-time sorting

Merge sort and heap sort are $\Theta(n \log n)$ in the worst case. Is it possible to find a faster comparison sort? This section proves that the answer is no. It then presents three algorithms that sort in linear time by using the key values directly instead of comparing elements.

3.9.1 The lower bound for comparison sorting

A comparison sort learns about the input only from the answers to questions of the form "is $a_i \le a_j$?" The behavior of such an algorithm can be shown with a **decision tree**. Each internal node of the tree is a comparison; the label *i:j* says that a_i and a_j are compared. The left branch follows the answer "≤" and the right branch the answer ">". Each leaf is the ordering the algorithm produces after that sequence of answers. Given an input, the algorithm follows a single path from the root to a leaf. The length of the path is the number of comparisons made on that input.

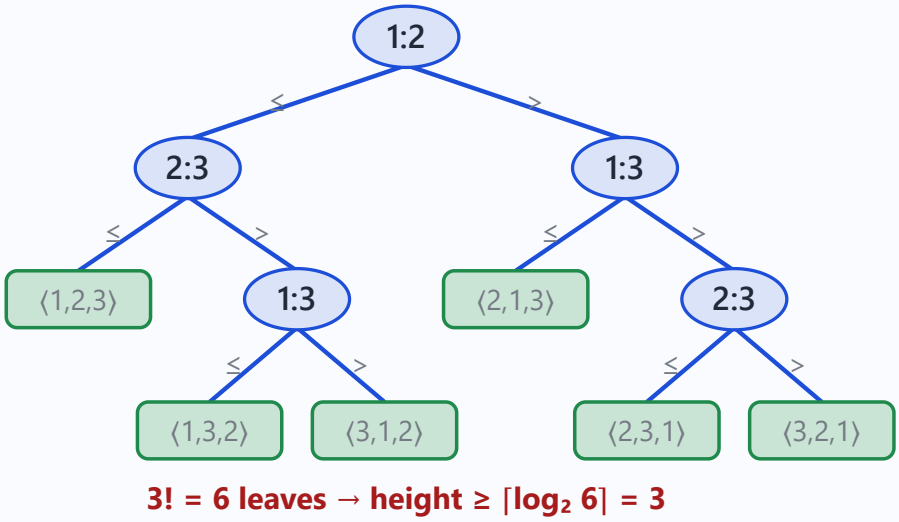


Figure 3.26: The decision tree of insertion sort on three elements. Internal nodes are the comparisons $i{:}j$, and the leaves are the six permutations. For $3! = 6$ leaves the height must be at least $\lceil \log_2 6 \rceil = 3$.

Theorem 3.3 (Comparison lower bound)

Every comparison sort that sorts n distinct elements makes $\Omega(n \log n)$ comparisons in the worst case.

Proof. Consider the decision tree of the algorithm. An input of distinct elements has $n!$ possible orderings, and each is sorted by a different permutation. If the algorithm is correct, every permutation must appear in at least one leaf; the tree has at least $n!$ leaves. A binary tree of height h can have at most 2^h leaves. Hence $2^h \geq n!$, that is, $h \geq \log_2(n!)$. Since each of the last $n/2$ factors of $n!$ is at least $n/2$, we have $n! \geq (n/2)^{n/2}$. Taking logarithms gives $\log_2(n!) \geq (n/2)\log_2(n/2) = \Omega(n \log n)$. The height of the tree is the number of comparisons in the worst case. □

The theorem has two important consequences. First, merge sort and heap sort are asymptotically optimal comparison sorts; their worst cases equal the lower bound. Second, sorting faster requires information other than comparisons. If the keys are small integers, the values themselves can be used as array indices; counting sort and radix sort do this.

Example 3.5

What is the least number of comparisons that every algorithm sorting three distinct elements by comparison alone must make in the worst case?

The decision tree must have $3! = 6$ leaves. Two comparisons can distinguish at most $2^2 = 4$ outcomes, which is not enough for six permutations. Three comparisons can distinguish $2^3 = 8 \geq 6$ outcomes. The answer is $\lceil \log_2 6 \rceil = 3$. Note that in the tree of Figure 3.26 some leaves are

reached with two comparisons; the lower bound applies only to the longest path. For eight elements the bound is $\lceil \log_2 40{,}320 \rceil = 16$; the worst case of 17 comparisons for merge sort is very close to it.

3.9.2 Counting sort

Counting sort sorts elements whose keys are integers in a small range without comparing them at all. The input is the array `A[1..n]`, and every key is in the set `{1, 2, …, k}`. The output is written to the array `B[1..n]`. A counter array `C[1..k]` is used as temporary storage.

The idea of the algorithm is this: if the number of elements with key less than or equal to i is known, then the output position of the last element with key i is known as well. For example, if five elements have a key less than or equal to 3, the last 3 is written to the fifth cell of the output. The algorithm consists of four loops:

```
for i = 1 to k
  C[i] = 0
for j = 1 to n
  C[A[j]] = C[A[j]] + 1   // C[i] = |{key = i}|
for i = 2 to k
  C[i] = C[i] + C[i-1]    // C[i] = |{key ≤ i}|
for j = n downto 1
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j]] - 1
```

1. The first loop zeroes all the counters.
2. The second loop scans the input and counts how many times each key occurs. At the end, `C[i]` is the number of elements with key i.
3. The third loop computes the **prefix sums**: each counter is added to the one on its left. At the end, `C[i]` is the number of elements with key less than or equal to i.
4. The fourth loop scans the input *from back to front*. Each element is written to the cell `B[C[A[j]]]` and the counter is decremented; the next element with the same key falls one cell to the left.

3.9.3 Step by step: A = [4, 1, 3, 4, 3]

The input is $A = [4, 1, 3, 4, 3]$, $n = 5$, with keys in $\{1, \ldots, 4\}$ ($k = 4$). The arrays are 1-based.

Loop	Step	Action	`C`	`B`
1	—	counters are zeroed	[0, 0, 0, 0]	[–, –, –, –, –]
2	`j = 1`	`A[1] = 4`, `C[4]++`	[0, 0, 0, 1]	
2	`j = 2`	`A[2] = 1`, `C[1]++`	[1, 0, 0, 1]	
2	`j = 3`	`A[3] = 3`, `C[3]++`	[1, 0, 1, 1]	

Loop	Step	Action	`C`	`B`
2	`j = 4`	`A[4] = 4`, `C[4]++`	[1, 0, 1, 2]	
2	`j = 5`	`A[5] = 3`, `C[3]++`	[1, 0, 2, 2]	
3	`i = 2`	`C[2] = 0 + 1`	[1, 1, 2, 2]	
3	`i = 3`	`C[3] = 2 + 1`	[1, 1, 3, 2]	
3	`i = 4`	`C[4] = 2 + 3`	[1, 1, 3, 5]	
4	`j = 5`	`A[5] = 3` → `B[3]`	[1, 1, 2, 5]	[–, –, 3, –, –]
4	`j = 4`	`A[4] = 4` → `B[5]`	[1, 1, 2, 4]	[–, –, 3, –, 4]
4	`j = 3`	`A[3] = 3` → `B[2]`	[1, 1, 1, 4]	[–, 3, 3, –, 4]
4	`j = 2`	`A[2] = 1` → `B[1]`	[0, 1, 1, 4]	[1, 3, 3, –, 4]
4	`j = 1`	`A[1] = 4` → `B[4]`	[0, 1, 1, 3]	[1, 3, 3, 4, 4]

Table 3.25: Trace of counting sort. The array `C` is written as `[C[1], C[2], C[3], C[4]]`.

After the counting loop, `C = [1, 0, 2, 2]`: there is one 1, two 3s, and two 4s; there is no 2. After the prefix sums, `C = [1, 1, 3, 5]`. These values are the last output position of each key: the last position of 1 is 1, of 3 is 3, and of 4 is 5. The final loop walks the input from right to left. `A[5] = 3` is written to `B[3]` and `C[3]` drops to 2; the next 3 (`A[3]`) therefore falls into `B[2]`. The result is $B = [1, 3, 3, 4, 4]$. No comparison has been made.

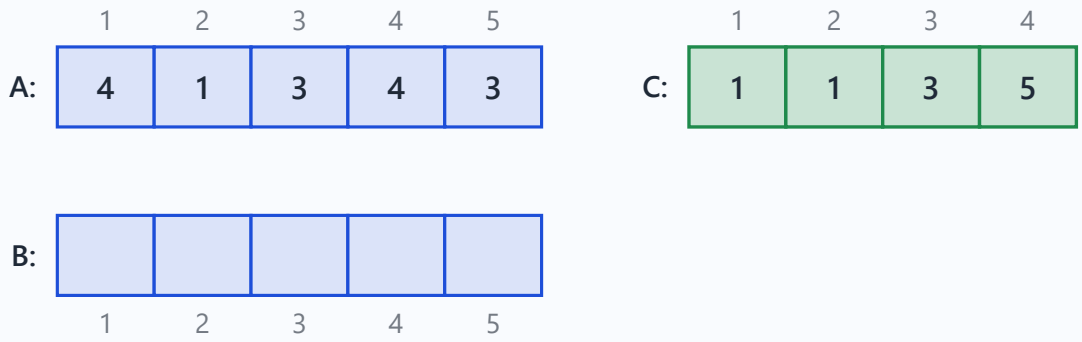


Figure 3.27: The prefix sums are ready: `C = [1, 1, 3, 5]`. `C[i]` is the position in `B` of the last element with key i. `B` is still empty.

Walking the last loop from back to front gives stability. Of two elements with equal keys, the one that comes later in the input is processed first and written to the cell farther right; the earlier one lands to its left. The input order of equal elements is thus preserved in the output. Had the loop run from front to back, the order of equal elements would be reversed; the result would still be sorted, but the algorithm would be unstable.

3.9.4 Complexity and Java code

Loop	Work	Time
1	`C[1..k] = 0`	$\Theta(k)$
2	counting	$\Theta(n)$
3	prefix sums	$\Theta(k)$
4	placing into `B`	$\Theta(n)$
Total		$\Theta(n + k)$

Table 3.26: The cost of each loop of counting sort.

The total time is $\Theta(n + k)$. If the key range is $k = O(n)$, the time is $\Theta(n)$. This does not contradict the comparison lower bound; counting sort makes no comparisons, so the assumption of the theorem does not apply to it. The extra memory is $\Theta(n + k)$: the output array and the counter array. If k is very large the algorithm is impractical. For 32-bit integers, for example, about 4.3 billion counters would be needed.

The Java version is 0-based and uses the range `0 … max`, where `max` is the largest element of the array.

```
int[] countingSort(int[] a) { // values ≥ 0
  if (a.length == 0) return a;
  int max = Arrays.stream(a).max().getAsInt();
  int[] count = new int[max + 1];
  for (int x : a) count[x]++;
  for (int i = 1; i <= max; i++) count[i] += count[i - 1];
  int[] out = new int[a.length];
  for (int i = a.length - 1; i >= 0; i--) {
    out[count[a[i]] - 1] = a[i];
    count[a[i]]--;
  }
  return out;
}
```

The code is written for non-negative integers only; an empty array returns immediately, because a largest element exists only in a non-empty array. Unlike the pseudocode, the counters are zeroed automatically by `new int[max + 1]`. After the prefix sums, `count[x]` is the number of elements with key less than or equal to x; since the output is 0-based, the last position of x is `count[x] - 1`. For example, for $A = [2, 5, 3, 0, 2, 3, 0, 3]$ the tallies are `[2, 0, 2, 3, 0, 1]` and the prefix sums are `[2, 2, 4, 7, 7, 8]`; the output is [0, 0, 2, 2, 3, 3, 3, 5].

Common mistake

Running the last loop from front to back. The output is still sorted, but elements with equal keys are written in reverse order. Used alone, counting sort may not show this difference; used as an intermediate step of radix sort, it destroys the order built by the earlier digits and makes the result wrong.

3.9.5 Radix sort

Counting sort is very fast when the key range is small, but its memory need explodes as the range grows. **Radix sort** solves this problem by sorting large numbers digit by digit. Each digit takes only a value from 0 to 9, so one counting sort with $k = 10$ per pass is enough.

The version described here is **LSD** (least significant digit) radix sort: sorting starts from the least significant digit, the ones digit, and continues with the tens, the hundreds, and so on. In each pass a *stable* sort is made on that digit. Let d be the number of digits and b the base. Each pass takes $\Theta(n + b)$ and there are d passes; the total time is $\Theta(d \cdot (n + b))$. If d and b are constants, the time is $\Theta(n)$. The extra memory is $\Theta(n + b)$; the algorithm is not in place.

```
int[] radixSort(int[] a) { // values ≥ 0
  if (a.length == 0) return a;
  int max = Arrays.stream(a).max().getAsInt();
  for (long exp = 1; max / exp > 0; exp *= 10)
    a = countByDigit(a, (int) exp);
  return a;
}
int[] countByDigit(int[] a, int exp) {
  int n = a.length;
  int[] out = new int[n], count = new int[10];
  for (int x : a) count[(x / exp) % 10]++;
  for (int i = 1; i < 10; i++) count[i] += count[i - 1];
  for (int i = n - 1; i >= 0; i--) {
    int d = (a[i] / exp) % 10;
    out[--count[d]] = a[i];
  }
  return out;
}
```

- `radixSort` grows the variable `exp` as 1, 10, 100, and so on. The condition `max / exp > 0` holds as long as the largest number still has an unprocessed digit. For 802, `exp` takes the values 1, 10, and 100; at 1000, $802/1000 = 0$ and the loop ends. `exp` is of type `long`: if max is 10^9 or larger, `exp *= 10` on an `int` would overflow and the loop could continue wrongly. Like counting sort, the code is meant only for non-negative integers.
- `countByDigit` is counting sort adapted to a single digit. The digit of a number is found with `(x / exp) % 10`: for 802 at `exp = 10`, $802/10 = 80$ and $80 \bmod 10 = 0$.

- The last loop again walks from back to front and ensures stability. The expression `out[--count[d]]` first decrements the counter and then uses the decremented value as the index; it does in one expression what `count[x] - 1` does in the Java version of counting sort.

Example 3.6

Let the input be [170, 45, 75, 90, 802, 24, 2, 66]. The largest number, 802, has three digits, so three passes are made. Missing digits count as 0: 45 is 045 as a three-digit number.

1. **Ones digit** (0, 5, 5, 0, 2, 4, 2, 6): [170, 90, 802, 2, 24, 45, 75, 66]. 170 and 90, whose ones digit is 0, kept their input order; so did 802 and 2, and 45 and 75.
2. **Tens digit**: [802, 2, 24, 45, 66, 170, 75, 90]. 802 and 2, whose tens digit is 0, took their places in the order of the previous pass (802 first).
3. **Hundreds digit**: [2, 24, 45, 66, 75, 90, 170, 802]. All digits have been processed and the array is sorted.



Figure 3.28: After the first pass the array is sorted by the ones digit. The underlined digits are the ones digits, used as the key in this pass.

3.9.6 Why is stability needed?

The correctness of LSD radix sort rests on the following invariant: after pass *k*, the array is sorted by the last *k* digits of the numbers. The next pass sorts by digit *(k + 1)*. Two numbers that differ in this digit come into the right order. For two numbers with the same digit, the earlier passes have determined the order, and that order is correct with respect to the last *k* digits. It is essential that this order be preserved, that is, that the sub-sort be stable.

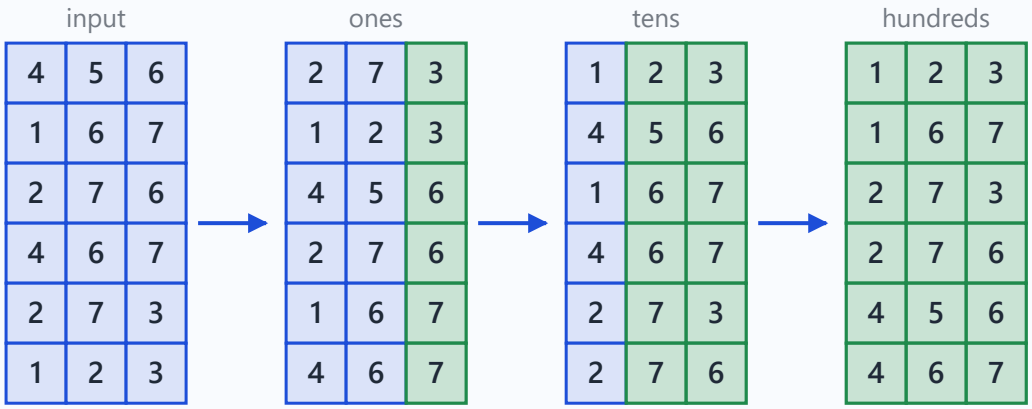


Figure 3.29: Sorting 456, 167, 276, 467, 273, 123 in three passes. The green columns are the trailing digits that are sorted so far: after k passes the array is sorted by the last k digits.

In the example in the figure, after the first pass the array is [273, 123, 456, 276, 167, 467]. In the second pass 167 and 467 have equal tens digits (6). A stable sort keeps the order coming from the first pass: 167 comes first. This order is correct, since the ones digits of the two numbers are also equal (7) and the input order was preserved. In the last pass, 273 and 276 have equal hundreds digits (2). Had the sub-sort been unstable, these two could have traded places, and the result would be a wrong array such as [123, 167, 276, 273, ...]. The result is [123, 167, 273, 276, 456, 467].

Radix sort can also start from the most significant digit; this version is called **MSD** (most significant digit). The MSD version first splits the numbers into buckets by the first digit and then sorts each bucket separately, recursively. It is widely used for sorting strings. The LSD version is simpler but requires all keys to be of the same length.

3.9.7 Code trace: countByDigit()

```
int[] countByDigit(int[] a, int exp) {
  int n = a.length;
  int[] out = new int[n], count = new int[10];
  for (int x : a) count[(x / exp) % 10]++;
  for (int i = 1; i < 10; i++) count[i] += count[i - 1];
  for (int i = n - 1; i >= 0; i--) {
    int d = (a[i] / exp) % 10;
    out[--count[d]] = a[i];
  }
  return out;
}
```

Let us trace the call `countByDigit(a, 1)` on [32, 15, 41, 23, 11]. The ones digits are 2, 5, 1, 3, 1 in order.

Stage	Action	`count[0..5]`	`out`
counting	digits 2, 5, 1, 3, 1	[0, 2, 1, 1, 0, 1]	[–, –, –, –, –]
prefix	prefix sums	[0, 2, 3, 4, 4, 5]	[–, –, –, –, –]
`i = 4`	11, `d = 1`, `--count[1] = 1`	[0, 1, 3, 4, 4, 5]	[–, 11, –, –, –]
`i = 3`	23, `d = 3`, `--count[3] = 3`	[0, 1, 3, 3, 4, 5]	[–, 11, –, 23, –]
`i = 2`	41, `d = 1`, `--count[1] = 0`	[0, 0, 3, 3, 4, 5]	[41, 11, –, 23, –]
`i = 1`	15, `d = 5`, `--count[5] = 4`	[0, 0, 3, 3, 4, 4]	[41, 11, –, 23, 15]
`i = 0`	32, `d = 2`, `--count[2] = 2`	[0, 0, 2, 3, 4, 4]	[41, 11, 32, 23, 15]

Table 3.27: Trace of `countByDigit([32, 15, 41, 23, 11], 1)`. `count` is shown only with cells 0–5; cells 6–9 are always 5.

After the prefix sums, `count[d]` is the number of elements whose digit is less than or equal to *d*; for example, `count[3] = 4`. This number is one more than the position of the last element with that digit in `out`. Walking from right to left, 11 is processed first and written to `out[1]`. 41 and 11 have the same ones digit; 41 is processed later and falls into `out[0]`, to the left of 11. In the input too, 41 came before 11: the order is preserved. The result is [41, 11, 32, 23, 15], sorted by the ones digit. The next pass (`exp = 10`) looks at the tens digit and produces [11, 15, 23, 32, 41]. 11 and 15 have equal tens digits; the order built by the first pass (11 first) is preserved.

3.9.8 Bucket sort

Bucket sort reaches linear expected time by making an assumption about the distribution of the input. Suppose the input consists of real numbers uniformly distributed over the interval *[0, 1)*. The algorithm opens *n* buckets and puts the value *x* into bucket number *⌊n · x⌋*. Each bucket is sorted internally, for example with insertion sort. Then the buckets are concatenated in order.

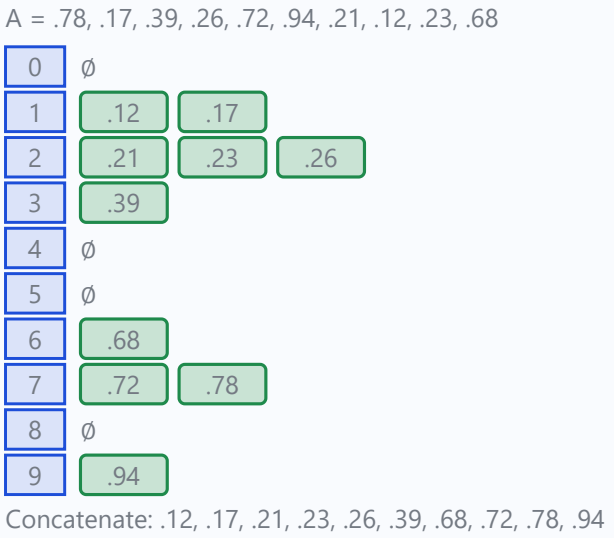


Figure 3.30: Bucket sort. Ten numbers (.78, .17, .39, .26, .72, .94, .21, .12, .23, .68) are distributed into ten

buckets: for example, .78 falls into bucket $\lfloor 10 \cdot 0.78 \rfloor = 7$. The buckets are sorted and concatenated in order.

In the example in the figure, three numbers (.21, .23, .26) fall into bucket 2, and two numbers each into buckets 1 and 7; buckets 0, 4, 5, and 8 are empty. If the distribution is uniform, on average one element falls into each bucket, and sorting the buckets takes $\Theta(n)$ expected time in total. Distributing and concatenating are also linear; the expected total time is $\Theta(n)$. But if the assumption fails, for instance if all elements fall into a single bucket, that bucket is sorted by insertion sort in $\Theta(n^2)$. Bucket sort is stable if the buckets are sorted with a stable algorithm, and it needs $\Theta(n)$ extra memory.

3.10 Timsort

The sorting algorithms used in libraries differ somewhat from the pure algorithms of textbooks. They exploit patterns common in real data and combine several algorithms. Timsort is the best-known example.

3.10.1 A hybrid algorithm

Timsort is a hybrid sorting algorithm that Tim Peters designed for Python in 2002. Today Java uses Timsort for object arrays (`Arrays.sort(Object[])`, `List.sort`), and so does Python's `sorted`. The starting point of the algorithm is the observation that real data is often not completely random. It contains long stretches that are already sorted. A few records appended to a table, a partly sorted log file, or two sorted lists concatenated are examples.

Timsort uses these sorted stretches, called **runs**. First the array is scanned from left to right and natural runs are found; a descending run is reversed to make it ascending. Very short runs are extended with insertion sort up to a minimum length (*minrun*, in practice between 32 and 64). Since insertion sort is very fast on small and nearly sorted pieces, this choice is sound. Then the runs are merged in pairs with the merge step of merge sort. If many elements in a row are taken from one run during a merge, the algorithm switches to "galloping" mode. In this mode the position of the next element is found by exponential search, and a whole block of elements is copied at once.

Timsort is stable, because both insertion sort and merging are stable. In the worst case it is $O(n \log n)$. If the input is already sorted, a single run is found and the algorithm finishes with $n - 1$ comparisons, that is, in $\Theta(n)$. It uses at most $O(n)$ extra memory for merging. Because stability matters when sorting objects, Java uses quicksort for primitive types and Timsort for objects.

The following code is a simplified version of Timsort: it does not look for natural runs but splits the array into fixed-length pieces. It sorts the pieces with `insertionSort(a, lo, hi)` and then merges them with the `merge` method from 3.6.4.

```
static final int MIN_RUN = 4;
void timSort(int[] a) {
  int n = a.length;
  for (int i = 0; i < n; i += MIN_RUN)            // 1) runs
    insertionSort(a, i, Math.min(i + MIN_RUN - 1, n - 1));
  for (int size = MIN_RUN; size < n; size *= 2)  // 2) merge
    for (int lo = 0; lo < n; lo += 2 * size) {
      int mid = lo + size - 1;
      int hi = Math.min(lo + 2 * size - 1, n - 1);
      if (mid < hi) merge(a, lo, mid, hi);
    }
}
```

- The first loop divides the array into pieces of length `MIN_RUN` and sorts each with insertion sort. The last piece may be shorter; `Math.min(i + MIN_RUN - 1, n - 1)` keeps the index from leaving the array.
- The second loop does the merging from the bottom up. `size` is the length of the sorted pieces to be merged and doubles in each round. In each round, two adjacent pieces `[lo..mid]` and `[mid+1..hi]` are merged.
- If `mid < hi` does not hold, there is no piece on the right to merge with; the single remaining piece is left as it is in this round.

This loop structure is a non-recursive, "bottom-up" version of merge sort. It uses no recursion stack.

3.10.2 Step by step

Let us trace the simplified Timsort with `MIN_RUN = 4` on the array [5, 21, 7, 23, 19, 10, 3, 12, 8, 1] (*n = 10*).

Step	Action	Array
Run [0..3]	insertion: [5, 21, 7, 23] → [5, 7, 21, 23]	[5, 7, 21, 23, 19, 10, 3, 12, 8, 1]
Run [4..7]	insertion: [19, 10, 3, 12] → [3, 10, 12, 19]	[5, 7, 21, 23, 3, 10, 12, 19, 8, 1]
Run [8..9]	insertion: [8, 1] → [1, 8]	[5, 7, 21, 23, 3, 10, 12, 19, 1, 8]
`size = 4`	[0..3] + [4..7] are merged	[3, 5, 7, 10, 12, 19, 21, 23, 1, 8]
`size = 4`	`lo = 8` : `mid = 11 ≥ hi = 9`, no partner	unchanged
`size = 8`	[0..7] + [8..9] are merged	[1, 3, 5, 7, 8, 10, 12, 19, 21, 23]

Table 3.28: The steps of the simplified Timsort.

The array is first divided into three runs; the last run has two elements. Each run is sorted with insertion sort. In the `size = 4` round the first two runs merge. The third run has no partner to merge with, because `mid = 8 + 4 − 1 = 11` goes past the end of the array (`hi = 9`). In the `size = 8` round the eight-element piece and the two-element piece merge and the array is sorted. When `size = 16 ≥ n`, the loop ends.



Figure 3.31: After `merge(0, 3, 7)` in the `size = 4` round. The first eight cells are sorted (green); the run [1, 8] in the last two cells awaits the next round.

3.11 Comparing the algorithms

We can now set the chapter's algorithms side by side. This section first collects the asymptotic properties in a table, then gives actual comparison counts, and discusses which algorithm to choose in which situation.

3.11.1 Complexity table

Algorithm	Best	Average	Worst	Extra memory	Stable	In place
Bubble (with flag)	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Yes	Yes
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	Yes	Yes
Selection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$	No	Yes
Shell (n/2 gaps)	$\Theta(n \log n)$	depends on gaps	$\Theta(n^2)$	$O(1)$	No	Yes
Merge	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	Yes	No
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$O(\log n)$*	No	Yes
Heap sort	$O(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(1)$	No	Yes
Counting	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$	Yes	No
Radix (LSD)	$\Theta(d(n + b))$	$\Theta(d(n + b))$	$\Theta(d(n + b))$	$\Theta(n + b)$	Yes	No
Bucket	$\Theta(n)$	$\Theta(n)$†	$\Theta(n^2)$	$\Theta(n)$	Yes	No
Timsort	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$	Yes	No

Table 3.29: Comparison of the sorting algorithms. (*) recursion stack, if the smaller part is processed first. (†) expected under a uniform distribution. k: key range, d: number of digits, b: base. The *$\Theta(n)$* best case of Timsort applies to the real version, which looks for natural runs.

Two details in the table call for explanation. The best case of heap sort is written with O, not Θ: if all keys are equal, no sift goes anywhere and the algorithm finishes in linear time. With distinct keys the best case is also $\Theta(n \log n)$. The extra memory of quicksort is only the recursion stack; if the larger part is handled with a loop and the smaller with recursion, the stack depth does not exceed $O(\log n)$.

3.11.2 Measurements: n = 1000

Asymptotic notation hides constant factors. To see actual behavior, the code in this chapter was run on three kinds of input with $n = 1000$ elements, and the element comparisons were counted. The random array was generated with a fixed seed.

Algorithm	Random	Sorted	Reverse sorted
Bubble (with flag)	497,960	999	499,500
Insertion	252,205	999	499,500
Selection	499,500	499,500	499,500
Shell (n/2)	14,723	8,006	11,716
Merge	8,693	5,044	4,932
Quicksort (last element as pivot)	11,384	499,500	499,500
Heap sort	16,791	17,583	15,965

Table 3.30: Element comparison counts for $n = 1000$. For reference, $n^2/2 \approx 500{,}000$ and $n \log_2 n \approx 9{,}966$.

The table confirms the theory with numbers. Selection sort makes exactly $n(n - 1)/2 = 499{,}500$ comparisons on all three inputs. Bubble sort and insertion sort finish with only $n - 1 = 999$ comparisons on the sorted input. On the random input the count for insertion sort is about $n^2/4 = 250{,}000$, and for bubble sort about $n^2/2$. Merge sort stays below $n \log_2 n$ on every input. On sorted and reverse-sorted inputs it makes about half as many comparisons, because one of the arrays being merged always ends early. Quicksort is a little above merge sort on the random input and close to the exact expected value in 3.7.8 (about 10,986). But on sorted and reverse-sorted inputs the version that takes the last element as pivot falls into its worst case and makes as many comparisons as selection sort. Heap sort is almost independent of the input; since it is not adaptive, it spends 17,583 comparisons even on the sorted input.

The comparison count is not the only criterion. Quicksort, although it makes more comparisons than merge sort, is usually faster in practice. The reason is that it does not copy to auxiliary arrays and it walks through memory sequentially. Heap sort, besides its comparison count, jumps around in memory (from index i to $2i + 1$), uses the cache badly, and is often slower than quicksort.

3.11.3 Which algorithm when?

Small or nearly sorted array

: Insertion sort: low overhead, adaptive, and stable. Since its time is $\Theta(n + I)$, it is also fast on large arrays with few inversions.

General purpose, speed first

: Quicksort, with a random pivot or median of three. It is cache-friendly and in place.

Stability required

: Merge sort or Timsort. This is why the object sort in libraries is Timsort.

Worst-case guarantee and little memory

Heap sort: $O(n \log n)$ and $O(1)$ extra memory. A hybrid algorithm called introsort starts with quicksort and switches to heap sort if the recursion gets too deep, thereby holding the worst case to $O(n \log n)$.

Small integer keys

Counting sort or radix sort: $\Theta(n + k)$ and $\Theta(d(n + b))$.

Linked list or external memory

Merge sort. It needs no random access; it reads elements sequentially and writes them sequentially. A large file that does not fit in memory is sorted piece by piece and the pieces are merged. A linked list can also be merged without an extra array, simply by changing links.

Bubble sort and selection sort are mostly for teaching. Selection sort makes sense only where writing is very expensive, since it makes at most $n - 1$ swaps.

3.12 Common mistakes

Most errors in sorting code cluster around boundary indices, temporary variables, and equality conditions. The list below brings together the mistakes described throughout the chapter.

Mistake	Consequence	Correct form
Running past the array bound	Writing `j < n` in bubble sort makes `a[j+1]` fall outside the array.	`j < n - 1 - i`
Not saving the key	In insertion sort the first shift overwrites the element being inserted.	`key = a[i]` before the loop
Keeping the minimum as a value	In selection sort the cell to swap is unknown.	Keep the index `min`
Breaking stability	`<` in the merge, or walking counting sort front to back, reverses the order of equal elements.	`L[i] <= R[j]`; walk back to front
Including the pivot in the recursion	`quickSort(a, lo, p)` may fail to shrink the part; infinite recursion.	`lo..p - 1` and `p + 1..hi`
"Quicksort is always *n log n*"	On a sorted input a fixed pivot gives $\Theta(n^2)$ and recursion of depth n.	Random pivot or median of three
Not shrinking the heap size	Elements moved to the end mix back into the heap.	`heapify(a, i, 0)`

Table 3.31: Common mistakes in sorting code.

What these mistakes have in common is that most are spotted at once when traced by hand on a small example. Good test inputs are: the shortest arrays (one and two elements) for boundary errors, records with equal keys for stability errors, and a sorted array for the bad case of quicksort.

3.13 Summary

- **Sorting** means finding a permutation that puts the elements of an array in ascending (or descending) order by key. Algorithms are compared by **time** (best, average, worst), **extra memory**, **stability**, **in-place** operation, **adaptivity**, and whether they are online.
- A **stable** algorithm preserves the relative order of records with equal keys; multi-key sorting and radix sort rely on this. Bubble, insertion, merge, counting, radix, and Timsort are stable; selection, Shell, quicksort, and heap sort are unstable.
- An **inversion** is a pair of elements in the wrong order relative to each other. An adjacent swap removes exactly one inversion; the number of swaps in bubble sort and the number of shifts in insertion sort equal the number of inversions. A reverse-sorted array has $n(n - 1)/2$ inversions.
- **Bubble sort** swaps neighbors; in each pass the largest of the remaining elements goes to the end. The version with a flag is $\Theta(n)$ in the best case and $\Theta(n^2)$ otherwise.
- **Insertion sort** saves the key, shifts the larger elements, and places the key. Its time is $\Theta(n + I)$; it is the best choice for small and nearly sorted arrays.
- **Selection sort** moves the minimum to the front in each round: $n(n - 1)/2$ comparisons on every input, at most $n - 1$ swaps.
- **Shell sort** performs insertion sort with decreasing gaps; its time depends on the gap sequence (worst case $\Theta(n^2)$ with $n/2$ gaps, $O(n^{1.5})$ with Hibbard's sequence).
- **Merge sort** follows the steps divide, sort, merge: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ on every input. It needs $\Theta(n)$ extra memory and is stable.
- **Quicksort** partitions around a pivot; the pivot settles in its final place and no merging is needed. It is $\Theta(n \log n)$ on average and $\Theta(n^2)$ in the worst case. With a random pivot, $\Theta(n^2)$ occurs only with very small probability; median of three prevents the bad case on sorted and reverse-sorted inputs but not on specially constructed ones.
- **Heap sort** turns the array into a max-heap in $O(n)$, then moves the root to the end and sifts $n - 1$ times: $O(n \log n)$ in the worst case, $O(1)$ extra memory.
- Every **comparison** sort makes $\Omega(n \log n)$ comparisons in the worst case (the decision tree has $n!$ leaves). **Counting sort** takes $\Theta(n + k)$, **radix sort** $\Theta(d(n + b))$, and **bucket sort** an expected $\Theta(n)$ under a uniform distribution; since they make no comparisons, they are not bound by this limit.
- **Timsort** is a stable hybrid algorithm that extends sorted runs with insertion sort and combines them by merging; it is the object sort of Java and Python.

3.14 Key terms

sorting swap in-place stable (sort) adaptive (sort) inversion bubble sort insertion sort selection sort Shell sort divide and conquer merge merge sort recurrence (relation) recursion tree master theorem quicksort pivot partition median of three heap heap sort

comparison sort lower bound lower bound counting sort radix sort LSD / MSD radix sort bucket sort Timsort quadratic linearithmic best / average / worst case

3.15 Test Yourself

1. How many comparisons and how many shifts does insertion sort make while sorting [2, 4, 1, 3]?

   Answer

   `i = 1`: 2 > 4 is false; 1 comparison, no shift. `i = 2` (1): 4 and 2 shift, the scan stops at `j = -1`; 2 comparisons, 2 shifts. `i = 3` (3): 4 shifts, 2 > 3 is false; 2 comparisons, 1 shift. In total 5 comparisons and 3 shifts. The inversions are (2, 1), (4, 1), (4, 3); their number equals the number of shifts.

2. How many passes, comparisons, and swaps does bubble sort with a flag make on [2, 3, 4, 5, 1]? Why is the result bad even though the array is nearly sorted?

   Answer

   In each pass the 1 moves only one cell left: [2, 3, 4, 1, 5], [2, 3, 1, 4, 5], [2, 1, 3, 4, 5], [1, 2, 3, 4, 5]. A swap occurs in all four passes, so the flag never allows an early exit. There are *4 + 3 + 2 + 1 = 10* comparisons and 4 swaps. Because a small element at the end can advance only one cell per pass through adjacent swaps, bubble sort reaches its worst-case comparison count on this input. Insertion sort sorts the same array with 3 + 4 = 7 comparisons.

3. Let the records be (4, a), (2, b), (4, c), (1, d), sorted by the number only. Does selection sort break stability on this input?

   Answer

   Yes. In the first round the minimum, (1, d), is swapped with the (4, a) at the front: (1, d), (2, b), (4, c), (4, a). In the second round (2, b) is in place. In the third round no swap happens, because (4, a) is not strictly smaller. In the end (4, c) is in front of (4, a); in the input a came first.

4. `partition(a, 0, 4)` (Lomuto, last element as pivot) runs on [3, 7, 1, 6, 4]. What does the method return and what does the array become?

   Answer

   The pivot is 4, `i = -1`. `j = 0`: 3 ≤ 4, `i = 0`, swap with itself. `j = 1`: 7 is larger. `j = 2`: 1 ≤ 4, `i = 1`, 7 and 1 trade places: [3, 1, 7, 6, 4]. `j = 3`: 6 is larger. The final swap `a[2] ↔ a[4]` gives [3, 1, 4, 6, 7]. The method returns 2; the pivot 4 is in its final place at index 2.

5. What is the max-heap built bottom-up from [1, 2, 3, 4, 5, 6, 7]?

   Answer

   `i = 2`: 3 is swapped with its larger child 7: [1, 2, 7, 4, 5, 6, 3]. `i = 1`: 2 with its larger child 5: [1, 5, 7, 4, 2, 6, 3]. `i = 0`: 1 is swapped with its larger child 7 and sinks to index 2; there its children are 6 and 3, and it is swapped with 6. The result is [7, 5, 6, 4, 2, 1, 3]. Even a sorted array takes quite a few swaps to build a heap; heap sort is not adaptive.

6. What is the maximum number of comparisons merge sort needs to sort a 16-element array in the worst case? What is the comparison lower bound?

   Answer

   There are four levels of merging: eight merges of 2 elements, four of 4 elements, two of 8 elements, and one of 16 elements. $8 \cdot 1 + 4 \cdot 3 + 2 \cdot 7 + 1 \cdot 15 = 49$. The lower bound is $\lceil \log_2 16! \rceil = \lceil 44.25 \rceil = 45$; merge sort is close to this bound.

7. The Java method `countingSort` is called with [1, 0, 2, 1, 0]. What is the `count` array after the counting loop and the prefix sums?

   Answer

   `max = 2`, so the counter has three cells. The tallies are [2, 2, 1]; the prefix sums become [2, 4, 5]. The last loop walks from right to left and the output is [0, 0, 1, 1, 2]. No comparisons are made; the time is $\Theta(n + k)$.

8. The array [53, 21, 33, 12] is sorted with LSD radix sort. What is the array after the first pass? Why is the order of 53 and 33 preserved?

   Answer

   The ones digits are 3, 1, 3, 2; a stable sort gives [21, 12, 53, 33]. The ones digits of 53 and 33 are equal, and because 53 came first in the input it stays first. The second pass produces [12, 21, 33, 53] by the tens digit.

9. In an array of a million elements only ten elements are out of place, each at most a few cells from its correct position. Which algorithm should be chosen for a fast, stable sort?

   Answer

   Insertion sort or Timsort. The number of inversions I is very small; the time of insertion sort is $\Theta(n + I)$, that is, linear. Timsort also finds the long sorted runs and finishes in nearly linear time. Merge sort would still spend $\Theta(n \log n)$; selection sort is $\Theta(n^2)$.

3.16 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 2, 6, 7, and 8: insertion sort and merge sort with loop invariants, heap sort, the probabilistic analysis of quicksort, and linear-time sorting, all with proofs.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Chapter 2: the same algorithms in Java code; its sections that measure comparison counts experimentally are an extended version of the measurement table in these notes.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2nd ed., 1998), Section 5.2: the most detailed treatment of the gap sequences of Shell sort and of the exact comparison counts of sorting algorithms.
- T. Peters, "listsort.txt" (CPython source code): explains Timsort's run detection, minrun selection, and galloping mode in its designer's own words.

3.17 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. E. D. Demaine, C. E. Leiserson, MIT 6.046J *Introduction to Algorithms* lecture notes, 2005.
3. C. A. R. Hoare, "Quicksort", *The Computer Journal* 5(1), 1962.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.
5. T. Peters, "listsort.txt" (Timsort design note), CPython, 2002.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.
7. D. L. Shell, "A High-Speed Sorting Procedure", *Communications of the ACM* 2(7), 1959.

PART I · CHAPTER 4

Searching Algorithms

Finding a requested value in a collection is one of the things computers do most often. A name is looked up in a phone book, a student number in a database, a variable name in a compiler's symbol table. The question is always the same: is this value in the collection, and if so, where? These notes present seven methods that answer it. We begin with linear search and its sentinel variant, which need no preconditions. Next comes binary search, which halves the range at every step of a sorted array. A hash table, instead of comparing its way to a location, computes it. Interpolation, jump and exponential search exploit different properties of a sorted array. Each method comes with code, a step-by-step trace and a cost analysis, and the notes close by comparing the methods on the same array and choosing the right one.

LEARNING OUTCOMES

Basic searches

Write linear search and binary search (the latter both iteratively and recursively) in Java, trace them step by step on an array, and work out their best, average and worst cases.

Searching with hashing

Explain hash functions, buckets, collisions, linear probing and chaining; interpret how the load factor affects cost.

Advanced searches on a sorted array

Justify the idea, the code and the complexities of interpolation, jump and exponential search: $O(\log \log n)$, $O(\sqrt{n})$ and $O(\log i)$.

Choosing a method

Pick a suitable search algorithm according to whether the data is sorted, how it is distributed and how large it is; avoid common implementation mistakes.

4.1 The search problem

Searching is one of the oldest and most widespread problems in computer science. A login screen that checks a user name, a point-of-sale program that looks up a product price and a text editor that runs a spell checker all do the same basic job. How fast that job is done can determine the performance of the whole program. If a search runs millions of times inside a loop, the few microseconds saved on each search add up to hours.

4.1.1 Problem definition

Search algorithms are used to find a **requested value** in a collection of data. The algorithm scans the data according to a **search criterion** and tries to find an item that satisfies it. The criterion is usually a **key**. In student records the key is the student number, in a dictionary it is the word itself, in a phone book it is the person's name. In these notes we treat the problem in its simplest form: the items are in an array of integers and the key is the item's own value. In real applications each cell of the array holds a record and the comparison is made on the record's key field; the logic of the algorithms does not change.

> **Definition 4.1**
>
> **Search problem:** The input is an array `arr` of n items and a value `key`. The output is an **index** i such that `arr[i] == key`. If there is no such index, the output is -1; this outcome is called an **unsuccessful search**.

Two details of the definition matter. First, the output is not the value itself but its *position*. The sought value is already known; the real question is where it sits in the array. Once the position is known, the other fields of the record in that cell can be reached too. Second, an unsuccessful search is also a valid outcome. Since Java array indices start at 0, -1 cannot be confused with any valid index and safely carries the meaning "not found".

If the sought value occurs several times in the array, the definition accepts the index of *any one* of them. Which match is returned depends on the algorithm. Linear search scans from the front, so it finds the first match; binary search starts in the middle, so it may return a match that looks arbitrary. If the first or last match is specifically required, the algorithm has to be adapted accordingly.

4.1.2 The cost measure: probes

Measuring how fast a search algorithm is in seconds is misleading, because the time depends on the processor, the language and the compiler. Instead, we count an operation that reflects the nature of the algorithm. In search algorithms the operation that determines the running time is comparing the key with an item of the array.

Definition 4.2

Looking at an array cell and comparing it with the key is called a **probe**. The cost of a search is measured by the number of probes it makes. All complexities in these notes describe how this number grows with n.

The reason for choosing the probe count is simple. The other work, such as incrementing loop counters and updating bounds, accompanies each probe in constant amount. The total time is therefore a constant times the probe count, and the order of growth is the same. In a hash table a probe means looking at a bucket and comparing the key stored there with the sought key; the definition fits that case as well.

Even for the same n, the cost varies from input to input. If the key is at the beginning of the array, linear search ends immediately; if it is at the end, the search walks the whole array. For this reason three cases are examined separately for each algorithm. The **best case** is the input that needs the fewest probes. The **worst case** is the input that needs the most probes, and it shows a guarantee the algorithm can give. The **average case** is the expected number of probes, computed under the assumption that the inputs follow a particular probability distribution. The worst case is often the most important measure, because it lets us say "no input will be slower than this".

4.1.3 Kinds of search algorithms

Different search algorithms work on different principles and require different properties of the data as preconditions. The more preconditions an algorithm demands, the faster it tends to run once they are met. Table 4.1 summarizes the six methods studied in these notes with their preconditions and principles.

Algorithm	Precondition	Operating principle
Linear search	none	Compares the items one by one.
Binary search	sorted array	Halves the collection to narrow the search range.
Hash search	hash table	The items are placed in a hash table; the search takes place in the key's bucket.
Interpolation search	sorted, uniformly distributed	Estimates the position from the ratio of the values.
Jump search	sorted array	Jumps over blocks of size $\sqrt{n}$, then searches linearly inside a block.
Exponential search	sorted array	Grows the bound 1, 2, 4, 8, ..., then runs binary search.

Table 4.1: Preconditions and operating principles of the search algorithms.

The preconditions in the table fall into three groups. Linear search asks for nothing; the array may be unsorted, or even a linked list. Hash search requires the data to be placed beforehand in a special structure, a hash table. The remaining four methods require a sorted array. What a sorted array tells

us is this: if the value in a cell is smaller than the key, the key cannot be to the left of that cell. This single comparison eliminates a large part of the array at once. Binary, interpolation, jump and exponential search use this information in different ways.

The sortedness precondition has a price. Sorting an array takes *O(n log n)* time with the best comparison-based algorithms. For a single search, sorting the array first is more expensive than simply running linear search. Sorting pays for itself (it is amortized) only if many searches will be made on the same array.

4.2 Linear search

Linear search is the first and simplest method that comes to mind. Because it requires nothing, it can be used in every situation. This generality is paid for with speed.

4.2.1 The idea

Linear search checks every item in the list in order. Going through the products on a shopping list one by one, or combing a room corner by corner for a lost object, rests on the same idea. Starting at the beginning of the list, the key is compared with each item. If a match is found, the index of that item is returned. If the list ends without a match, the result is *−1*.

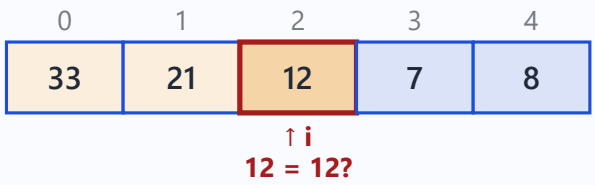


Figure 4.1: Searching for 12 with linear search. 33 and 21 have been checked with no match; now the 12 at position `i = 2` is being compared.

Sometimes the best way to find the information you want is really to look item by item. If the array is unsorted, there is no other choice. In an unsorted array, looking at one cell tells nothing about the others. Learning that 33 is not the key says nothing about whether 21 is. For an unsuccessful search to be able to answer "not present", therefore, it is necessary to look at every cell at least once. This simple observation shows that on unsorted data no algorithm can make fewer than *n* probes in the worst case.

4.2.2 Code: linearSearch()

The Java code for linear search is as follows:

```
int linearSearch(int[] arr, int n,
                 int key) {
  int i = 0;
  while (i < n && arr[i] != key)
    i++;
  if (i < n)
    return i;      // found: index
  else
    return -1;     // not found
}
```

The method takes three parameters: `arr` holds the items to be searched, `n` the used size of the array, and `key` the value to look for. The return value is the index of the sought item, or *−1* if it is not found. Passing the size as a separate parameter also covers the case where only the first `n` cells of the array are filled.

The line `int i = 0;` starts at the beginning of the array. The condition of the `while` loop has two parts: `i < n` checks that the end of the array has not been reached, and `arr[i] != key` checks that the key is not in that cell. As long as both are true, the loop body runs and `i` increases by one. The loop stops for one of two reasons: either `i` has reached `n`, or `arr[i]` equals the key. The `if` after the loop tells these two cases apart. If `i < n` is true, the loop stopped because of a match and `i` is the index of the match. If it is false, the array has been scanned from start to end without a match, and the method returns *−1*.

The order of the conditions matters. Java's `&&` operator **short-circuits**: if the left side is false, the right side is never evaluated. When `i == n`, `i < n` is false and `arr[n]` is never accessed. Had the conditions been reversed, that is, `arr[i] != key && i < n`, the last step of an unsuccessful search would try to read `arr[n]` and throw an `ArrayIndexOutOfBoundsException`.

Common mistake

Writing the check after the loop as `if (arr[i] == key)`. In an unsuccessful search the loop ends with `i = n`, and this check reads outside the array. The safe way to find out why the loop stopped is to test the bound (`i < n`) first.

4.2.3 Best and worst case

The behavior of the algorithm can be seen on an unsorted example array of ten items:

> `arr = {33, 21, 12, 7, 8, 1, 3, 44, 56, 18}`, *n = 10*

Example 4.1

Best case: searching for 33. With `i = 0`, `arr[0] = 33` equals the key. The second part of the loop condition is false on the first test, so the loop body never runs. Since `i = 0 < n`, the method returns 0. Only **1 comparison** was made.

Worst case: searching for 17. 17 is not in the array. The algorithm can find that out only by looking at every item:

1. `i = 0`: `arr[0] = 33 ≠ 17` → `i = 1`.
2. `i = 1`: `arr[1] = 21 ≠ 17` → `i = 2`.
3. `i = 2`: `arr[2] = 12 ≠ 17` → `i = 3`.
4. `i = 3`: `arr[3] = 7 ≠ 17` → `i = 4`.
5. `i = 4`: `arr[4] = 8 ≠ 17` → `i = 5`.
6. `i = 5`: `arr[5] = 1 ≠ 17` → `i = 6`.
7. `i = 6`: `arr[6] = 3 ≠ 17` → `i = 7`.
8. `i = 7`: `arr[7] = 44 ≠ 17` → `i = 8`.
9. `i = 8`: `arr[8] = 56 ≠ 17` → `i = 9`.
10. `i = 9`: `arr[9] = 18 ≠ 17` → `i = 10`.
11. `i = 10 = n`: `i < n` is false and the loop ends. The `if` check is false too, and the method returns *–1*.

The value comparison was made *n = 10* times. The worst case is *O(n)*.

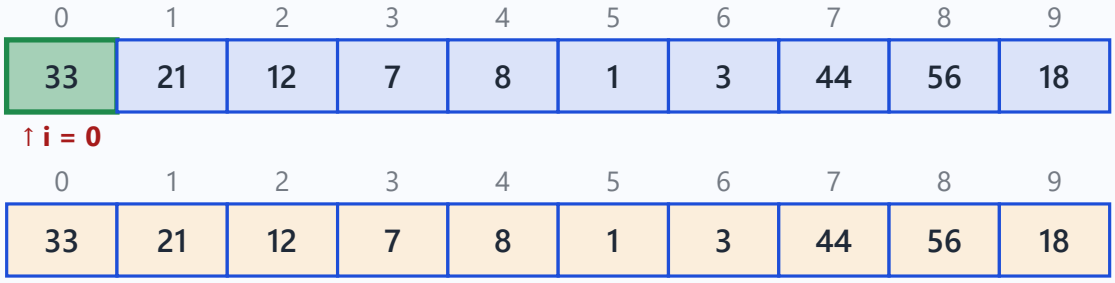


Figure 4.2: The two extremes of linear search. Top, the best case: 33 is found in the first cell. Bottom, the worst case: while searching for 17, every cell has been checked and no match found.

The worst case arises with two different inputs: the key is absent, or it is found only in the last cell. In both, *n* value comparisons are made. The only difference is the final test of the loop. If the key is in the last cell, the loop stops at `i = n – 1` with a match. If it is absent, the loop goes around once more and stops at `i = n` through the bound condition.

4.2.4 Code trace: an average search

The best and worst cases are the two extremes. For a typical search, we look for 1 in the same array. The 1 sits near the middle of the array, at index 5. Table 4.2 shows every evaluation of the loop condition in the call `linearSearch(arr, 10, 1)`.

Test	`i`	`i < n`	`arr[i]`	`arr[i] != 1`	Result
1	0	true	33	true	`i++`
2	1	true	21	true	`i++`
3	2	true	12	true	`i++`
4	3	true	7	true	`i++`
5	4	true	8	true	`i++`
6	5	true	1	false	exit the loop

Table 4.2: Trace of `linearSearch(arr, 10, 1)`.

On the sixth test `arr[5] = 1` equals the key; the `while` condition becomes false and the loop is left. Then `i = 5 < n` is true and the method returns 5. In total 6 value comparisons were made. The general rule is clear: an item at index k is found with $k + 1$ comparisons.

4.2.5 Complexity and evaluation

The cost of the best and worst cases is easy to read off: one is 1 comparison, the other n. The average case needs an assumption. The most common one is that the search is successful and that the key is equally likely to be at each position.

> **Theorem 4.1 (Average cost of a successful linear search)**
>
> If the key is in the array and is equally likely, with probability $1/n$, to be at each of the n positions, then the average number of comparisons in linear search is $(n + 1)/2$.

Proof. The item at position i (counting from 1) is found with i comparisons. The expected value is the sum of each position's cost times its probability: $\Sigma_{i=1}^{n} \, i \cdot (1/n) = (1/n) \cdot n(n + 1)/2 = (n + 1)/2$. Here we used the identity $1 + 2 + \ldots + n = n(n + 1)/2$. □

For $n = 10$ the average is $11/2 = 5.5$ comparisons. The result agrees with intuition: on average the key is in the middle of the array, so about half of it is examined. In an unsuccessful search, n comparisons are always made. Table 4.3 puts the three cases together.

Case	Comparisons	Order	Example ($n = 10$)
Best (first item)	1	$O(1)$	33 → 1 comparison
Average (successful, equally likely)	$(n + 1)/2$	$O(n)$	5.5 on average

Case	Comparisons	Order	Example ($n = 10$)
Worst / unsuccessful	n	$O(n)$	17 → 10 comparisons

Table 4.3: The cost of linear search.

It is an important result that the average case is also $O(n)$. The average cost is only half of the worst case, and halving does not change the order. If the array grows tenfold, the average search takes ten times as long. As for extra memory, the situation is good: the algorithm uses only the variable `i`, so the extra memory is $O(1)$.

The advantages of linear search can be listed as follows:

- It is easy to code and to understand; it needs no complicated data structure.
- It needs no preparation: the array does not have to be sorted.
- It is ideal for small collections. To look up a number in a personal contact list, no more complicated method is needed.
- It also works on structures that can only be accessed sequentially, such as a linked list. Binary search cannot work efficiently on a linked list, because even reaching the middle item takes $O(n)$ steps.

Its disadvantages are equally clear:

- It is slow on large data. In an array of 10^6 items a successful search makes 500,000 comparisons on average.
- Even if the array is sorted, it does not use that information. In a sorted array the search could stop on meeting a value larger than the key; the basic algorithm does not do this.
- If many queries will be made on the same array, sorting it once and using binary search is much cheaper.

4.2.6 Sentinel linear search

Looking carefully at the loop of linear search, we see that *two* comparisons are made on every iteration: `i < n` and `arr[i] != key`. Only the second is the real work of the search. The first is a safety check put there so that we do not run off the end of the array. There is a clever way to remove this check from the loop.

Definition 4.3

A **sentinel** is a special value placed at the end of the array to guarantee that the loop stops. In sentinel linear search the key itself is written into the last cell. The loop then finds a match at the last cell at the latest, and the `i < n` test is no longer needed.

The code looks like this:

```
int sentinelSearch(int[] arr, int n,
                   int key) {
  if (n == 0) return -1;
  int last = arr[n - 1];
  arr[n - 1] = key;         // sentinel
  int i = 0;
  while (arr[i] != key)
    i++;
  arr[n - 1] = last;        // restore
  if (i < n - 1 || last == key)
    return i;
  return -1;
}
```

The method first saves the last item in the variable `last` and writes the key into the last cell. From this point on the array holds at least one copy of the key. The loop now tests only `arr[i] != key`; `i` cannot exceed the bound, because a match is found at position `n - 1` at the latest. When the loop ends, the old value is written back into the last cell, and the array returns to its original state.

The last step needs care. The place where the loop stops can mean two things. If `i < n - 1`, the match was found before the sentinel, in a real item. If `i == n - 1`, the loop stopped at the sentinel. Even then the key may have been found: if the array's real last item equals the key. That is why the second part of the `if` condition is `last == key`. If either condition holds, `i` is returned; if both are false, *–1*.

The gain in numbers is as follows. In an unsuccessful search the classic loop makes *n* value comparisons and *n + 1* bound tests, about *2n* comparisons in all. The sentinel loop makes *n* value comparisons. Two more comparisons come after the loop: `i < n - 1` and `last == key`. The total is about *n + 2*. The number of comparisons almost halves, but the order is still *O(n)*. This is an improvement that shrinks the constant factor.

Common mistake

Using sentinel search on a read-only array, or on an array that other threads are reading at the same time. During the search the last cell is temporarily changed. If another thread reads the array in the meantime, it sees a wrong value. Forgetting to write the last item back also corrupts the array permanently.

4.2.7 Code trace: sentinelSearch()

A trace of sentinel search on an unsuccessful search: `arr = [7, 3, 9, 4, 6]`, `n = 5`, key 5. The 5 is not in the array.

Example 4.2

1. **Setup.** The last item, 6, is saved in `last`. The key 5 is written into the last cell as the sentinel; the array temporarily becomes `[7, 3, 9, 4, 5]`. `i = 0`.
2. `arr[0] = 7 ≠ 5` → `i = 1`. There is only one comparison in this iteration; the `i < n` test is not made.
3. `arr[1] = 3 ≠ 5` → `i = 2`.
4. `arr[2] = 9 ≠ 5` → `i = 3`.
5. `arr[3] = 4 ≠ 5` → `i = 4`.
6. `arr[4] = 5` equals the key, and the loop stops. But what was found is not a real item; it is the sentinel.
7. The last item is put back: the array is again `[7, 3, 9, 4, 6]`.
8. The test `i < n - 1` is *4 < 4*, which is false. The test `last == key` is *6 ≠ 5*, which is also false. The method returns *–1*.

Total number of comparisons: 5 in the loop and 2 at the end, that is *7 = n + 2*. The classic loop would have made 5 value comparisons and 6 bound tests on the same search; counted as about *2n*, that is 10.

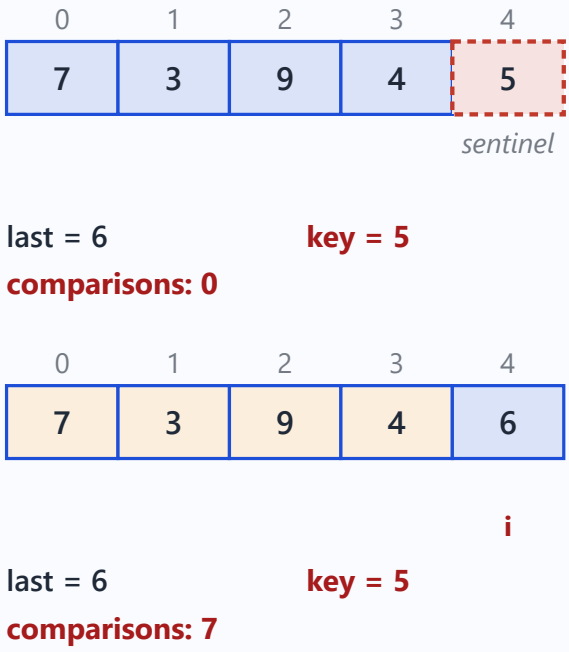


Figure 4.3: Sentinel search. (a) The key 5 has been written into the last cell as the sentinel; the real last item, 6, waits in `last`. (b) The loop stopped at the sentinel and 6 has been put back; since both final tests are false, the result is *–1* and 7 comparisons were made in total.

A different example shows why the second `if` condition is needed: `arr = [12, 5, 9, 20, 7]` and key 7. Here the key really is in the last cell. 7 is written as the sentinel anyway, and the loop stops at `i = 4`. `i < n - 1` is false, but `last == key` is true, and the method correctly returns 4. Without this condition, the real last item of the array could never be found.

4.3 Binary search

Linear search eliminates only one cell of the array per probe. If the array is sorted, a single probe can say much more. If the middle item is smaller than the key, then by sortedness every item to its left is also smaller than the key, and none of them can be the key. A single probe eliminates half the array. Binary search is built on this observation.

4.3.1 The idea

Looking for a book in a library is a good analogy. The books are arranged alphabetically by the author's surname. Instead of walking along all the shelves from the start, you first go to the middle section. If the book you want falls to the left of the middle shelf, you continue on the shelves to the left; if to the right, on the shelves to the right. In the remaining section you again go to the middle. Every comparison halves the part that has to be searched.

Definition 4.4

Binary search compares the item in the middle of the search range of a sorted array with the key. If the values are equal, the search ends. If the key is smaller, the search continues in the left half of the range; if larger, in the right half. If the range becomes empty, the key is not in the array.

Binary search has two preconditions. First, the array must be sorted; throughout these notes, sorted means sorted in increasing order. Second, items must be accessible by index in *O(1)* time. This second condition holds for an array but not for a linked list. In a linked list, reaching the middle item means walking half the list from the front. In that case even a single "middle" computation is as expensive as linear search.

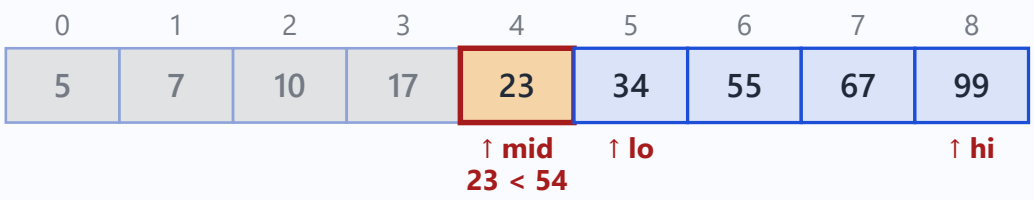


Figure 4.4: Searching for 54 in a sorted array. The middle item, 23, is smaller than 54; so 23 and all the items to its left (gray) are eliminated with a single comparison. The search continues between `lo = 5` and `hi = 8`.

4.3.2 Why is binary search fast?

The power of binary search comes from the range halving at every step. In a list of 1000 items, linear search makes 500 checks on average. Binary search finishes in at most 10 steps. The range sizes shrink after each comparison as follows:

> 1000 → 500 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1

There are ten range sizes in this sequence and each corresponds to one comparison: 10 comparisons in all. In halving, the fractional part is dropped, because once the middle item has been compared it is no longer in the range. For example, when the middle item is removed from a range of 125 items, 62 items remain on each side.

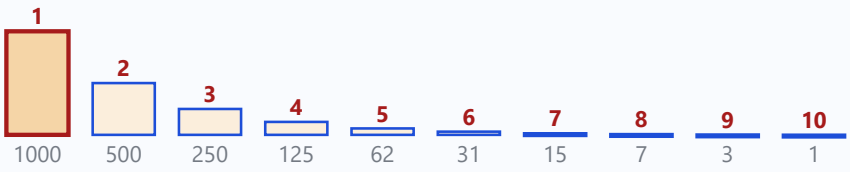


Figure 4.5: The range size halves with every comparison. The number above each bar is the step number; in 10 steps a range of 1000 items shrinks to a single item.

How effective this shrinking is shows up for large values of n. Table 4.4 puts the worst-case comparison counts of the two methods side by side.

n	Linear (worst)	Binary (worst)
10	10	4
1,000	1,000	10
10^6	10^6	20
10^9	10^9	30

Table 4.4: Worst-case comparison counts.

Binary search makes at most $\lfloor \log_2 n \rfloor + 1$ comparisons. In an array of a billion items, 30 comparisons suffice. When the data grows 1000-fold, binary search gains only about 10 steps, since $2^{10} = 1024 \approx 1000$. Linear search becomes 1000 times slower. This kind of growth is called **logarithmic** growth.

4.3.3 Code: iterative binarySearch()

The Java code for binary search written with a loop (iteratively) is as follows:

```
int binarySearch(int[] arr, int n, int key) {
  int lo = 0, hi = n - 1;
  while (lo <= hi) {
    int mid = (lo + hi) / 2;
    if (key < arr[mid])
      hi = mid - 1;        // left half
    else if (key > arr[mid])
      lo = mid + 1;        // right half
    else
      return mid;         // found
  }
  return -1;
}
```

`lo` and `hi` are the lower and upper bounds of the range to be searched; both are inclusive. At the start the range is the whole array: `lo = 0`, `hi = n - 1`. The condition `while (lo <= hi)` keeps the loop going as long as the range is not empty. When `lo == hi` the range holds a single item, and that item must be tested as well; this is why the condition uses `<=`.

In each iteration, `mid` is computed as the index of the item in the middle of the range. Because integer division drops the fractional part, if the range has an even number of items the left one of the two middle items is chosen. Then there are three paths. If the key is smaller than `arr[mid]`, the key, if present, is to the left of the middle; `hi = mid - 1` moves to the left half. If the key is larger, `lo = mid + 1` moves to the right half. If they are equal, `mid` is returned. If the loop ends without finding a match, the range has become empty (`lo > hi`) and the method returns *–1*.

Updating the bounds with `mid - 1` and `mid + 1` is critical. The middle item has just been tested and found not to be the key, so it is not taken into the new range. This rule also guarantees that the range shrinks by at least one item in each iteration, and hence that the loop always terminates.

4.3.4 The recursive version

Binary search is naturally recursive: searching in one half of the range is a smaller instance of the same problem. The recursive Java code is as follows:

```
int search(int[] arr, int lo,
           int hi, int key) {
  if (lo > hi) return -1;  // empty range
  int mid = (lo + hi) / 2;
  if (arr[mid] == key) return mid;
  if (arr[mid] < key)           // right half
    return search(arr, mid + 1, hi, key);
  return search(arr, lo, mid - 1, key);
}
```

The first call is `search(arr, 0, n - 1, key)`. The method first checks the base case: if `lo > hi`, the range is empty and it returns *−1*. Otherwise the middle item is computed, and if it equals the key its index is returned. If not, the same method is called again on the right half if the key is larger, or on the left half if it is smaller, and its result is returned as is. The parameters of the new call take the place of the `lo` and `hi` updates of the iterative version.

From the point of view of algorithm design, binary search is the simplest example of the **divide-and-conquer** approach. The divide step is choosing the middle item. The conquer step is to continue in one half only; the other half is never touched. There is no combine step, because the answer to the subproblem is directly the answer to the problem. Since there is only one branch, this approach is also called **decrease and conquer**. The cost is described by the following recurrence:

> $T(n) = T(n/2) + 1$

The recurrence says that a search on n items makes one comparison and then a search on $n/2$ items. Its solution is $T(n) = O(\log n)$.

The two versions make the same comparisons, but their memory use differs. The iterative version uses only a few variables: the extra memory is $O(1)$. In the recursive version each call opens a frame on the call stack, and the calls nest at most $\lfloor \log_2 n \rfloor + 1$ deep. The extra memory is $O(\log n)$. This is small; even for a billion items 30 frames suffice. Still, the iterative version is lighter both in memory and in call cost, and is more often preferred in practice.

4.3.5 Binary search and the balanced binary search tree

The comparisons made by binary search can be drawn as a tree. This drawing lets us see why the algorithm takes at most $\lfloor \log_2 n \rfloor + 1$ steps. As an example, take a sorted array of 15 items:

> `5, 11, 12, 14, 15, 18, 19, 21, 23, 25, 27, 28, 30, 32, 37`

Example 4.3

1. **Root.** The middle of the whole array is 21 at index 7. The first comparison of binary search is always with 21; 21 is the root of the tree.
2. **Second level.** If the key is smaller than 21, the search continues among indices 0–6; the middle of this range is 14 at index 3. If larger, it continues in the range 8–14; the middle is 28 at index 11. 14 becomes the left child of the root and 28 the right child.
3. **Third level.** The middle of each subrange forms the next level: 11 from the range 0–2, 18 from 4–6, 25 from 8–10, 32 from 12–14.
4. **Leaves.** The remaining eight items (5, 12, 15, 19, 23, 27, 30, 37) are one-item ranges and become leaves.

The tree has 4 levels and height 3. Since $\lfloor \log_2 15 \rfloor + 1 = 3 + 1 = 4$, every search makes at most 4 comparisons. For successful searches the average number of comparisons is $(1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 8 \cdot 4)/15 = 49/15 \approx 3.27$.

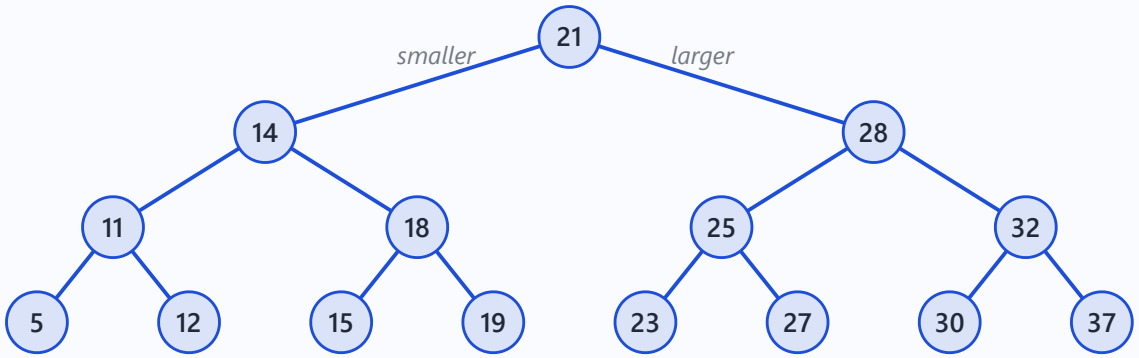


Figure 4.6: The balanced binary search tree obtained from the sorted array of 15 items. To the left of the root are the values smaller than 21, to the right the larger ones. Four levels mean at most four comparisons.

This tree is a **binary search tree**: the values in the left subtree of every node are smaller than the node, and those in the right subtree are larger. Moreover, it is the most balanced tree possible; every level is completely full. Binary search on a sorted array is the same thing as walking down from the root of this best tree. The cost of a search in a tree is found by adding one to the depth of the node sought. The worst case depends on the height of the tree. The height of a balanced tree with n nodes is $\lfloor \log_2 n \rfloor$. Had the tree been built unbalanced, for example with every node having only a right child, the height would be $n - 1$ and the search would degrade to linear search.

4.3.6 Searching the implicit tree

The tree does not have to be built in memory. The sorted array is the tree itself; the `mid` computation walks the tree **implicitly**. The left child of a node is the middle of the left half range, and its right child is the middle of the right half. A second sorted array shows this. The array again has 15 items:

```
1, 3, 5, 6, 7, 8, 11, 14, 16, 18, 20, 22, 23, 25, 27
```

Example 4.4

Searching for 20.

1. `lo = 0`, `hi = 14` → `mid = 7`, `arr[7] = 14`. Since $20 > 14$, we go right: `lo = 8`.
2. `mid = (8 + 14)/2 = 11`, `arr[11] = 22`. Since $20 < 22$, we go left: `hi = 10`.
3. `mid = (8 + 10)/2 = 9`, `arr[9] = 18`. Since $20 > 18$, we go right: `lo = 10`.
4. `mid = (10 + 10)/2 = 10`, `arr[10] = 20`: found, 10 is returned.

From root to leaf, 4 comparisons were made. The path in the tree is 14 → 22 → 18 → 20.

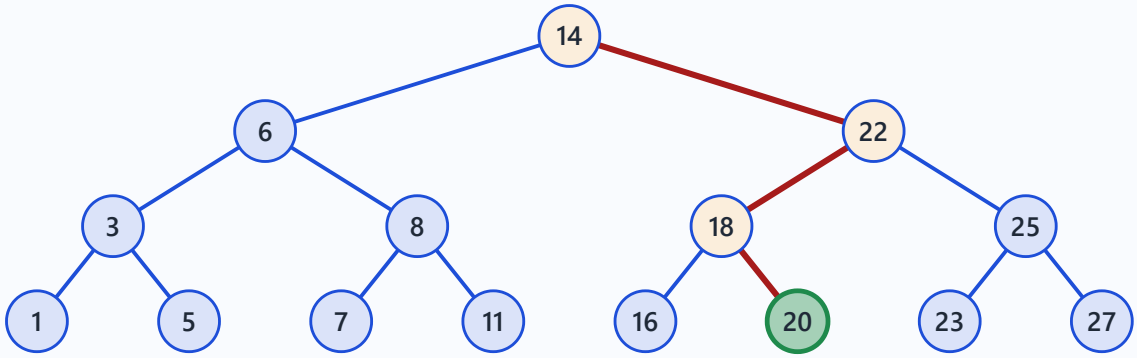


Figure 4.7: Searching for 20 in the implicit tree of the second array. The red edges show the path taken and the green node the item found. 20 is a leaf; that is why the worst-case 4 comparisons were made.

The tree view also explains an unsuccessful search. If the key is not in the array, the search descends to a leaf and tries to go to an empty child of that leaf. This empty child corresponds to the gap into which the key would be inserted without breaking the order. For example, if 19 is searched for in this array, the path is 14 → 22 → 18 → 20. Since 19 is smaller than 20, the search wants to go to the left child of 20, but that child is empty. 19 would have fit between 18 and 20.

4.3.7 Best and worst case

The two extremes of the code can be counted on an array of nine items:

`arr = {5, 7, 10, 17, 23, 34, 55, 67, 99}`, $n = 9$

Example 4.5

Best case: searching for 23. `lo = 0`, `hi = 8`; `mid = (0 + 8)/2 = 4` and `arr[4] = 23` equals the key. It is found in one step and 4 is returned. The best case is $O(1)$; the key is exactly in the middle.

Worst case: searching for 54. 54 is not in the array.

1. `mid = 4`: $23 < 54 \rightarrow$ `lo = 5`. The left half (0–4) is eliminated.
2. `mid = (5 + 8)/2 = 6`: $55 > 54 \rightarrow$ `hi = 5`.
3. `mid = (5 + 5)/2 = 5`: $34 < 54 \rightarrow$ `lo = 6`.
4. `lo = 6 > hi = 5`: the range is empty, the loop ends and -1 is returned.

The search was found to be unsuccessful in 3 comparisons. If 54 were in the array, it would sit between 34 and 55, at index 6. When the loop ends, `lo` shows exactly this insertion position.



Figure 4.8: The end of the search for 54. After three comparisons `lo = 6` and `hi = 5`; the range is empty and all cells have been eliminated.

In this array the worst case is actually 4 comparisons: $\lfloor \log_2 9 \rfloor + 1 = 4$. The search for 54 ending in 3 steps is because the key fails on a shallow branch of the tree. For example, if 99 is searched for, the probed indices are 4, 6, 7, 8 and 4 comparisons are made. The bound gives the worst case; not every unsuccessful search reaches it.

4.3.8 Code trace: binarySearch(arr, 9, 47)

A trace of a successful search with variable values is below. The array is `{0, 4, 7, 10, 14, 23, 45, 47, 53}` and the key is 47.

Iteration	`lo`	`hi`	`mid`	`arr[mid]`	Comparison	Action
1	0	8	4	14	*47 > 14*	`lo = 5`
2	5	8	6	45	*47 > 45*	`lo = 7`
3	7	8	7	47	*47 = 47*	`return 7`

Table 4.5: Trace of `binarySearch(arr, 9, 47)`.

In the first iteration `lo ≤ hi` is true; the middle is computed as *(0 + 8)/2 = 4* and `arr[4] = 14`. Since 47 is larger, the left part is eliminated: `lo = 5`. In the second iteration the middle is *(5 + 8)/2 = 6*, and `arr[6] = 45` is again smaller: `lo = 7`. In the third iteration the middle is *(7 + 8)/2 = 7* and `arr[7] = 47` equals the key; the method returns 7. In nine items, 3 comparisons were made. For this array the bound is $\lfloor \log_2 9 \rfloor + 1 = 4$.



Figure 4.9: The end of the third iteration: `lo = mid = 7`, `hi = 8`. 47 has been found (green); the cells eliminated in the first two iterations are gray.

There is a detail in counting comparisons. Each iteration of the code contains two written comparisons: `key < arr[mid]` and, if needed, `key > arr[mid]`. But both look at the same cell; the probe count goes up by one. In these notes, looking at one middle item is counted as one comparison. In Java, for integers, a single `Integer.compare` call can also be used instead of two comparisons.

4.3.9 Complexity

The worst case of binary search is stated exactly as follows.

Theorem 4.2 (Comparison bound of binary search)

On a sorted array of $n \geq 1$ items, binary search runs for at most $\lfloor \log_2 n \rfloor + 1$ iterations. Hence the worst-case cost is $O(\log n)$.

Proof. Let the range hold s items at the start of an iteration. After the middle item is removed, $\lfloor (s - 1)/2 \rfloor$ items remain on the left and $\lceil (s - 1)/2 \rceil = \lfloor s/2 \rfloor$ on the right. Whichever half is taken, the new range has at most $\lfloor s/2 \rfloor$ items. Since $s = n$ at the start, after k iterations the range holds at most $\lfloor n/2^k \rfloor$ items. For an iteration to be made, the range must not be empty. For iteration $k + 1$ to be made, we need $n/2^k \geq 1$, that is, $k \leq \log_2 n$. Hence the iteration number can be at most $\lfloor \log_2 n \rfloor + 1$. □

The theorem also agrees with the solution of the recurrence $T(n) = T(n/2) + 1$. Unrolling the recurrence gives $T(n) = T(n/2^k) + k$. When $n/2^k = 1$, that is for $k = \log_2 n$, the subproblem has constant cost. The total is of order $\log_2 n$.

Case	Order
Best (first middle matches)	$O(1)$
Average	$O(\log n)$
Worst / unsuccessful	$O(\log n)$
Extra memory	iterative $O(1)$, recursive $O(\log n)$
Preparation (sorting)	$O(n \log n)$, once

Table 4.6: The cost of binary search.

That the average case is also $O(\log n)$ is clear from the tree view. In a balanced tree about half of the nodes are leaves, and the leaves are at the greatest depth. A randomly chosen item is most likely on the lower levels. In the example with 15 items the average was 3.27 and the worst case 4 comparisons; the difference between the two is constant.

The last row of the table is the hidden price of binary search. If the array is unsorted, it must first be sorted, and that takes $O(n \log n)$. Sorting is done once, searches many times. For k queries the costs of the two routes are as follows:

- Linear search: $O(k \cdot n)$ in total.

- Sort once, then binary search: $O(n \log n + k \log n)$ in total.

If k is small, for example a single query, linear search is cheaper. Once k exceeds the order of $\log n$, the cost of sorting pays for itself, and the gap widens quickly as k grows.

Example 4.6

On an unsorted array of $n = 10^6$ items, $k = 1000$ searches will be made. Linear search needs $1000 \cdot 10^6 = 10^9$ comparisons in the worst case. Sorting first takes about $n \log_2 n \approx 10^6 \cdot 20 = 2 \cdot 10^7$ comparisons. Then 1000 binary searches make at most $1000 \cdot 20 = 2 \cdot 10^4$ comparisons. The total is about $2 \cdot 10^7$, roughly 50 times fewer than the linear route.

4.3.10 Implementation details

Binary search is simple as an idea, but surprisingly hard to write correctly. Knuth reports that the first binary search was published in 1946, but a bug-free version appeared only in 1962. Most of the bugs cluster around four details.

Overflow in the midpoint. The expression `(lo + hi) / 2` is correct mathematically, but in Java an `int` holds at most $2^{31} - 1 = 2{,}147{,}483{,}647$. If `lo + hi` exceeds this limit, the sum wraps around to a negative number and `mid` becomes negative. For the sum of two indices to exceed this limit the array must have more than about a billion (2^{30}) items. This limit once looked far away; today it is a real risk for large arrays and file offsets. In the Java library's `Arrays.binarySearch` this bug went unnoticed for years and was fixed in 2006. Safe ways to write it are:

> `lo + (hi - lo) / 2` or `(lo + hi) >>> 1`

The first divides the difference and never overflows. The second turns the overflowed sum back into the correct value with an unsigned right shift. For example, for `lo = 2,000,000,000` and `hi = 2,100,000,000`, the expression `(lo + hi) / 2` gives $-97{,}483{,}648$; both safe forms give the correct result, 2,050,000,000.

Updating the bounds. Writing `hi = mid` or `lo = mid` may fail to shrink the range. When `lo == hi` and the key is smaller than `arr[mid]`, the assignment `hi = mid` leaves the range unchanged and the loop runs forever. Since the middle item has already been checked, the bounds must be updated by ± 1.

Duplicate items. Returning immediately on equality gives any one of the matches. If 4 is searched for in the array `{2, 4, 4, 4, 4, 7, 9, 12}`, iterative binary search looks at index 3 in the first iteration and returns 3, although the first 4 is at index 1. If the first position is wanted, the result is recorded on equality and the search continues to the left with `hi = mid - 1`.

The Java library. `Arrays.binarySearch(arr, x)` runs binary search on a sorted array. If the value is found it returns its index; with duplicate items it is unspecified which one is returned. If it is not found, it returns *−(insertion point) − 1*. The insertion point is the index at which *x* would be inserted without breaking the order. The return value is always negative to signal "not found", and its absolute value carries the position. For example, if 5 is searched for in `{2, 4, 4, 4, 4, 7, 9, 12}`, the insertion point is 5 and the method returns *−5 − 1 = −6*. Had it returned only "−(insertion point)", a value whose insertion point is 0 would return 0 and be confused with a found item.

> **Common mistake**
>
> Writing the loop condition as `while (lo < hi)`. This condition skips a single-item range: if the key is in the last remaining cell, the loop is left without testing it and *−1* is returned. If 10 is searched for in `{5, 7, 10}`, then in the first iteration `lo = 2`, `hi = 2` and the loop ends without ever looking at 10.

4.3.11 First occurrence among duplicates

Binary search that finds the *first* position of the key in a sorted array containing duplicates is written as follows:

```
int firstIndex(int[] arr, int key) {
  int lo = 0, hi = arr.length - 1, result = -1;
  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    if (arr[mid] == key) {
      result = mid;      // record,
      hi = mid - 1;      // look left
    } else if (arr[mid] < key)
      lo = mid + 1;
    else
      hi = mid - 1;
  }
  return result;
}
```

The code departs from iterative binary search in two places. First, the variable `result` holds the leftmost match found so far; its initial value is *−1*. Second, on equality the method does not return. It records the index of the match in `result` and, with `hi = mid − 1`, checks whether there was an earlier match to the left. The loop runs until the range is empty, and finally `result` is returned. The midpoint is also written in the overflow-safe way.

Iteration	`lo`	`hi`	`mid`	`arr[mid]`	Action	`result`
1	0	7	*0 + (7 − 0)/2 = 3*	4	equal: record, `hi = 2`	3

Iteration	`lo`	`hi`	`mid`	`arr[mid]`	Action	`result`
2	0	2	*0 + (2 − 0)/2 = 1*	4	equal: record, `hi = 0`	1
3	0	0	0	2	*2 < 4*: `lo = 1`	1
—	1	0	`lo > hi` : the loop ends			`return 1`

Table 4.7: Trace of `firstIndex({2, 4, 4, 4, 4, 7, 9, 12}, 4)`.

The array holds four 4s (indices 1–4). In the first iteration the 4 at index 3 is found, but it may not be the first 4. It is recorded and the search looks left. In the second iteration the 4 at index 1 is found; it is recorded too. In the third iteration the 2 at index 0 is smaller than the key and the range shifts right. When the range becomes empty, the last recorded value, 1, is returned. There were 3 probes, and the time is still *O(log n)*.

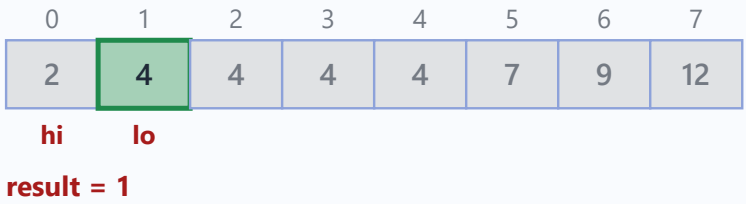


Figure 4.10: The end of `firstIndex`. `lo = 1 > hi = 0` has been reached; `result = 1` is the index of the first 4.

The same idea is used to find the last position: on equality the result is recorded and the search continues to the right with `lo = mid + 1`. The difference between the two methods also gives the number of times the key occurs in the array. In the example the last position is 4 and the first is 1; the value 4 occurs *4 − 1 + 1 = 4* times. This count takes *O(log n)* however many repetitions there are.

Common mistake

Finding the first position by scanning linearly to the left after finding any match. The result is correct, but if all items are equal the scan takes *n* steps and the cost rises to *O(n)*. Narrowing the search range on equality keeps the cost logarithmic in every case.

4.4 Searching with hashing

The methods so far find the location of the key by comparing. Each comparison yields some information and reduces the candidate positions. Every comparison-based method has a lower bound: even in a sorted array, about $log_2\ n$ comparisons cannot be avoided in the worst case. Hash search gets past this bound by another route. It does not search for the location; it *computes* it.

4.4.1 The idea: compute the location

Hash search works on a principle like a key fitting exactly into its own lock. The data consist of **keys** and the **values** that go with them. The key is converted into a table index by a **hash function**. This index is called the key's **hash value**, and it shows where the data sit in the table.

> **Definition 4.5**
>
> A **hash table** is an array of m cells; each cell is called a **bucket**. A **hash function** h converts each key into a bucket number between 0 and $m - 1$. A key x is placed in bucket $h(x)$ and searched for there.

The search takes place in the bucket that corresponds to the key's hash value. If the bucket is not empty, the key is compared with the keys in the bucket. If a match is found, the search ends. The cost of a search is made up of computing the hash value and a few comparisons in that bucket. This cost is independent of the number of keys in the table. That is why in a well-designed hash table the average search time is $O(1)$.

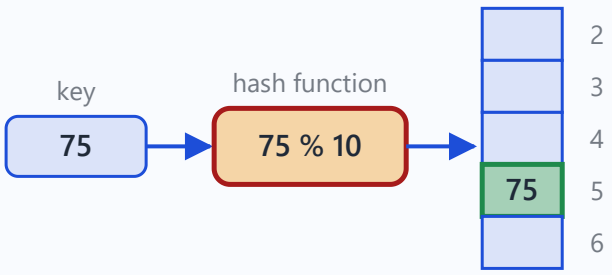


Figure 4.11: The key 75 is directed to bucket 5 by the hash function $h(x) = x \bmod 10$. The search looks directly at this bucket; the other buckets are not touched.

In the examples of these notes the keys are integers and the hash function is $h(x) = x \% 10$. This function takes the last digit of the key and makes it the bucket number. The function is simple, but enough to show the principle. In real applications the number of buckets is often chosen as a prime or a power of two, and the function uses all the bits of the key.

4.4.2 Advantages and disadvantages

The advantages of hash search are as follows:

- It is very fast on average: $O(1)$.
- The data do not have to be sorted beforehand.

- It is ideal for large collections and for applications that mix many insertions and searches. Inserting a new item into a sorted array requires shifting items; in a hash table insertion is also *O(1)* on average.
- If the hash function is well chosen, the sought item is reached directly.

It also has disadvantages:

- The hash function can lead to **collisions**.
- When a collision occurs, extra processing steps are needed. In the worst case all keys fall into the same bucket and the search degrades to *O(n)*.
- It needs extra memory; part of the table is always kept empty.
- The table scatters keys without regard to order. For that reason, visiting the keys in sorted order, finding the smallest key, or answering range queries such as "keys between 50 and 100" is not efficient.

> **Definition 4.6**
>
> A **collision** is when two different keys fall on the same hash value: $h(x) = h(y)$ while $x \neq y$.

Collisions are unavoidable. The number of possible keys is far greater than the number of buckets; by the pigeonhole principle some keys must share a bucket. For example, if all integers are distributed over 10 buckets, every number with the same last digit falls in the same bucket. The design of a hash table therefore involves two questions: how do we choose a hash function that reduces collisions, and how do we resolve the unavoidable ones?

4.4.3 The hash function

When the key is a string, it must first be converted into a number. The simplest way is to add up the numeric codes of the characters:

```
int hash(String key, int m) {
  int sum = 0;
  for (char c : key.toCharArray())
    sum += c;           // Unicode code
  return sum % m;       // m: table size
}
```

The method walks over the characters of the key one by one. In Java the `char` type behaves like an integer; the line `sum += c` adds the character's Unicode code to the sum. When the loop ends, the sum is reduced to the range *0* to $m - 1$ by taking it modulo the table size `m`. The method runs in *O(L)* time, where L is the length of the key.

Key	Character codes	Sum *% 10*
`"ali"`	*97 + 108 + 105 = 310*	0
`"ila"`	*105 + 108 + 97 = 310*	0
`"veli"`	*118 + 101 + 108 + 105 = 432*	2

Table 4.8: The character-sum hash function, *m = 10*.

The function is simple but weak. Addition does not see order: words made of the same letters always give the same sum. That is why "ali" and "ila" collide in bucket 0. "kar" and "ark" collide for the same reason. Moreover, the sums of short words are bunched in a narrow range; in a large table most buckets would never be used.

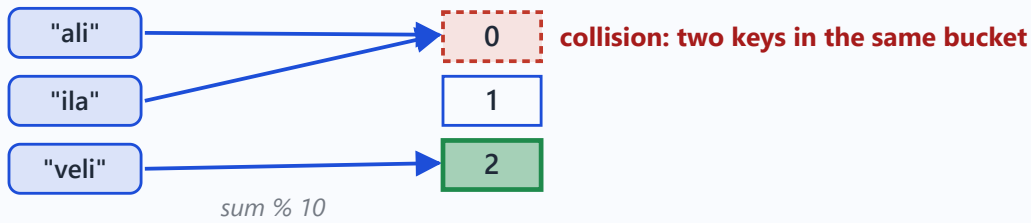


Figure 4.12: A collision in the character-sum hash function. Because "ali" and "ila" give the same sum (310), they fall into bucket 0; "veli" goes to bucket 2.

A good hash function has two properties at once: it is computed quickly, and it spreads keys evenly over the buckets. Java's `String.hashCode()` gives each character a weight that depends on its position:

$$h(s) = \Sigma_{i=0}^{L-1} s[i] \cdot 31^{L-1-i}$$

In this sum the position of the character matters too. For "ali" the value is $97 \cdot 31^2 + 108 \cdot 31 + 105 = 96{,}670$, and for "ila" it is 104,350; the two words are now told apart. Because the sum can be large, an `int` overflows, but this is not a problem: overflow keeps on mixing the values. The bucket number is found by taking this value modulo *m* after fixing its sign.

4.4.4 Code trace: hash("veli", 10)

Example 4.7

`key = "veli"`, `m = 10`. At the start `sum = 0`.

1. `c = 'v'`: its Unicode code 118 is added, `sum = 118`.
2. `c = 'e'`: 101 is added, `sum = 219`.
3. `c = 'l'`: 108 is added, `sum = 327`.

4. `c = 'i'` : 105 is added, `sum = 432` .
5. *432 % 10 = 2*: "veli" falls into bucket 2.

The same computation gives 310 for "ali" and "ila"; both collide in bucket 0.

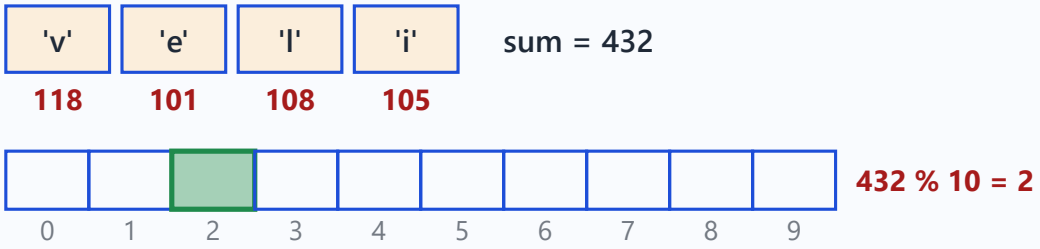


Figure 4.13: The character codes of the key "veli" and their sum. Since *432 mod 10 = 2*, the key is placed in bucket 2.

4.4.5 Searching with linear probing

There are two main ways to resolve a collision. In the first, each key is placed in one of the table's own cells; if its bucket is full, another empty bucket is looked for. This approach is called **open addressing**. The simplest open-addressing method is linear probing.

Definition 4.7

Linear probing: If the key's bucket *h(x)* is full, the buckets *h(x) + 1*, *h(x) + 2*, ... are examined in turn; on reaching the end of the table, the search wraps around to the start. Insertion goes into the first empty bucket. Search follows the same sequence and stops when it finds the key or reaches an empty bucket.

Search in a table that uses linear probing is written as follows:

```
int hashSearch(Integer[] table, int key) {
  int m = table.length;     // m > 0
  int index = Math.floorMod(key, m);   // hash
  while (table[index] != null) {
    if (table[index] == key)
      return index;          // found
    index = (index + 1) % m; // next bucket
  }
  return -1;                 // empty: absent
}
```

The table is of type `Integer[]`, so an empty bucket can be represented by `null`. In this example the table stores only keys; in a general implementation each bucket holds a key-value pair. `m` is the table size. The first bucket to look at is `Math.floorMod(key, m)`; `key % m` would give a negative index for a negative key. The table must not be empty ($m > 0$).

The `while` loop runs until `table[index]` is empty. In each iteration the key in the bucket is compared with the sought key. If they match, the bucket's index is returned; in a key-value table the value would be returned here. If there is no match, we move on to the next bucket. The expression `(index + 1) % m` wraps from the last bucket back to bucket 0; the table is scanned circularly. If an empty bucket is reached, the key is not in the table and the method returns -1.

The logic of stopping at an empty bucket comes from how insertion works. When a key is inserted, it is placed in the first empty bucket starting from its own bucket. So if the key is in the table, all the buckets from its own bucket up to its location are full. If an empty bucket is met during a search, the key cannot be further on.

Common mistake

Assuming that the comparison `table[index] == key` is safe with `Integer` objects. Here `key` is an `int`, so Java unboxes `table[index]` and compares the numbers, and the result is correct. If both sides were `Integer`, `==` would compare object identities; for values outside −128 to 127, even equal numbers could come out "not equal". When comparing two objects, `equals` must be used.

4.4.6 Inserting with linear probing

Let the following keys be inserted in order into an empty table ($m = 10$): 16, 12, 45, 29, 6, 122, 5, 58, 75. The hash function is $h(x) = x \% 10$.

Key	$h(x)$	Buckets probed	Bucket used	Probes
16	6	6	6	1
12	2	2	2	1
45	5	5	5	1
29	9	9	9	1
6	6	6 (full), 7	7	2
122	2	2 (full), 3	3	2
5	5	5, 6, 7 (full), 8	8	4
58	8	8, 9 (full), 0	0	3
75	5	5, 6, 7, 8, 9, 0 (full), 1	1	7

Table 4.9: Insertion with linear probing.

The first four keys go directly into their own buckets. The bucket of 6, bucket 6, holds 16, so 6 moves on to the next bucket, 7. 122 likewise slides from 2 to 3. The case of 5 is more interesting: buckets 5, 6 and 7 are full, and 5 can be placed only in 8. The bucket 8 of 58 has just been taken by 5; 58 finds 9 full as well and wraps from the end of the table to the start, settling in 0. The last key, 75, has bucket 5, and all buckets from 5 through 0 are full. After seven probes, 75 is placed in bucket 1.

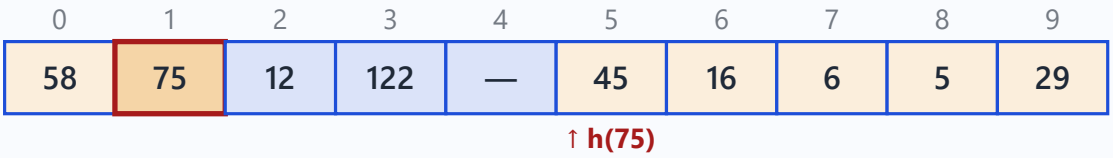


Figure 4.14: The table after nine insertions. The hash value of 75 is 5, but it has been placed in bucket 1. Only bucket 4 is empty; the full buckets from 5 through 3 form a single cluster.

In the final state the full buckets 5, 6, 7, 8, 9, 0, 1, 2, 3, in that order, have formed one contiguous cluster; only bucket 4 is empty. This phenomenon is called **primary clustering**. As a cluster grows, a new key that falls into any bucket of the cluster has to walk to the end of the cluster and makes the cluster one bucket larger. Large clusters tend to get larger. This is why linear probing slows down quickly as the table fills.

4.4.7 Code trace: hashSearch(table, 75)

We search for 75 in the resulting table. The table is `[58, 75, 12, 122, null, 45, 16, 6, 5, 29]`.

Probe	`index`	`table[index]`	Comparison	Next `index`
1	5	45	*45 ≠ 75*	*(5 + 1) % 10 = 6*
2	6	16	*16 ≠ 75*	7
3	7	6	*6 ≠ 75*	8
4	8	5	*5 ≠ 75*	9
5	9	29	*29 ≠ 75*	*(9 + 1) % 10 = 0*
6	0	58	*58 ≠ 75*	1
7	1	75	*75 = 75*	`return 1`

Table 4.10: Trace of `hashSearch(table, 75)`.

The search starts at bucket *75 % 10 = 5*. 45, 16, 6, 5 and 29 are passed in turn; after bucket 9 the search wraps from the end to the start. After 58 in bucket 0 is passed as well, 75 is found in bucket 1 and the method returns 1. As with insertion, the search took 7 probes: 75's location was reached at the end of the cluster.



Figure 4.15: At the end of `hashSearch(table, 75)`, `index = 1`; 75 has been found.

An unsuccessful search is more expensive. If 35 is searched for in the same table, the search again starts at 5 and continues until it reaches 4, the only empty bucket: buckets 5, 6, 7, 8, 9, 0, 1, 2, 3, 4 are examined. Nine key comparisons and one empty bucket make 10 probes. In this table the average over successful searches is $22/9 \approx 2.44$ probes; most keys sit close to their own bucket.

4.4.8 Chaining

The second way to resolve collisions is to put more than one key in a bucket.

Definition 4.8

In **separate chaining**, each bucket holds the head of a linked list. Keys that fall into the same bucket are added to this list. Search walks the list in the key's bucket from start to end.

Let the same keys be inserted, with the same hash function, into a table that uses chaining. 16, 12, 45 and 29 fall into empty buckets and form one-item lists. When 6 is inserted, bucket 6 is full; this is a collision, but 6 does not slide to another bucket, it is appended to the end of 16's list. 122 is appended to the list of bucket 2 after 12. 5 goes behind 45 in the list of bucket 5. 58 settles in the empty bucket 8. Finally 75 is appended to the list of bucket 5 as the third item.

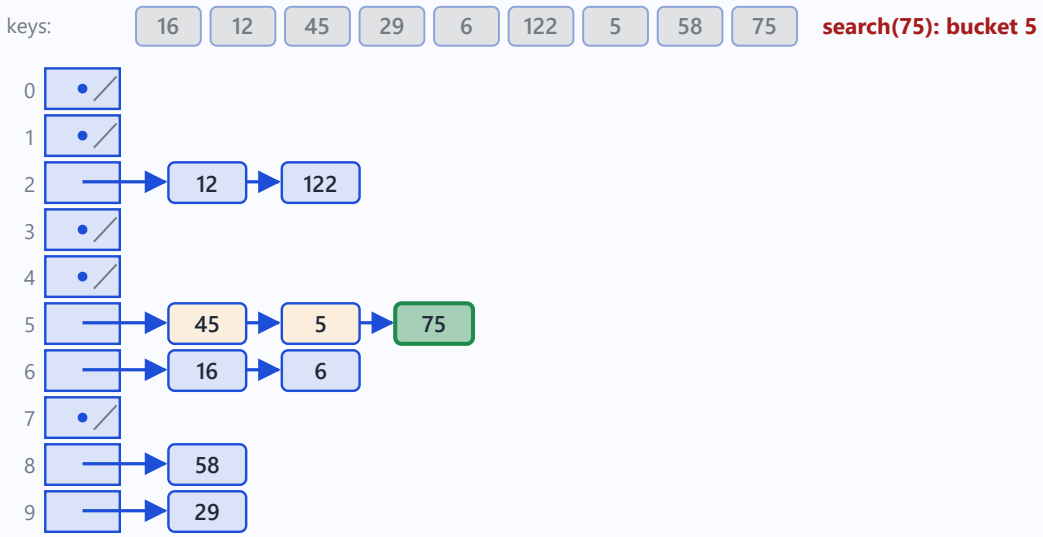


Figure 4.16: The table built with chaining. The list of bucket 2 is 12 → 122, that of bucket 5 is 45 → 5 → 75, and that of bucket 6 is 16 → 6. While 75 is searched for, the list of bucket 5 is walked: 45 ≠ 75, 5 ≠ 75, 75 = 75.

In chaining, to search for 75 only the list of bucket 5 is walked: 45 ≠ 75, 5 ≠ 75, 75 = 75. It is found in three comparisons. This is far fewer than the 7 probes of linear probing, because chaining has no clustering. Each key is compared only with keys that share its own hash value. 16, which falls into bucket 6, and 75, which falls into bucket 5, never meet.

Chaining has other advantages as well. The table never "fills up"; the lists can grow. Deletion is easy too: the item is removed from the list. In linear probing, by contrast, deleting a key and emptying its bucket cuts the path to the keys behind it; the deleted bucket has to be marked "deleted". The price of chaining is storing a separate object and a pointer for each node. That means both extra memory and scattered access in memory. Linear probing keeps keys in consecutive cells and so uses the cache better.

Example 4.8

Searching for 35 in the chaining table. *h(35) = 5*; the list of bucket 5 is 45 → 5 → 75. 45 ≠ 35, 5 ≠ 35, 75 ≠ 35 and the list ends. After 3 key comparisons the search turns out to be unsuccessful. The same search in the linear-probing table had needed 9 key comparisons and one empty bucket, 10 probes in all.

4.4.9 Load factor and complexity

The quantity that really determines the performance of a hash table is how full the table is.

Definition 4.9

The **load factor** is $\alpha = n / m$, where n is the number of keys in the table and m is the number of buckets.

In chaining, α is the average length of a list and can exceed 1. In linear probing each bucket holds at most one key, so $\alpha \le 1$. In our example tables 9 keys were put into 10 buckets: $\alpha = 0.9$.

Under the uniform hashing assumption, in which keys are distributed over the buckets randomly and independently of one another, Knuth gives the following approximate formulas for the average number of probes:

Method	Successful search	Unsuccessful search	$\alpha = 0.5$	$\alpha = 0.9$
Chaining	$\approx 1 + \alpha/2$	$\approx 1 + \alpha$	1.25 / 1.5	1.45 / 1.9
Linear probing	$\approx \tfrac{1}{2}(1 + 1/(1-\alpha))$	$\approx \tfrac{1}{2}(1 + 1/(1-\alpha)^2)$	1.5 / 2.5	5.5 / 50.5

Table 4.11: Average probe counts (uniform hashing assumption). In the last two columns the successful search is given first, then the unsuccessful one.

The numbers in the table are computed directly from the formulas. For example, in linear probing with $\alpha = 0.9$, an unsuccessful search needs $\tfrac{1}{2}(1 + 1/0.1^2) = \tfrac{1}{2}(1 + 100) = 50.5$ probes. In chaining the cost grows linearly with α. In linear probing the term $1/(1-\alpha)^2$ grows without bound as α approaches 1. When the table is half full, the difference between the two methods is small; at ninety percent full, an unsuccessful search in linear probing is about 27 times as expensive as in chaining.

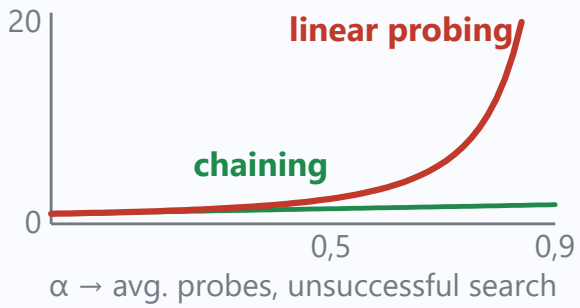


Figure 4.17: How the average number of probes in an unsuccessful search varies with the load factor. In chaining the curve is almost flat; in linear probing it rises sharply as α approaches 0.9 (the curve is cut off at 20 probes).

The formulas are averages, and an actual table may deviate from them. In our example linear-probing table with $\alpha = 0.9$, the average over successful searches was 2.44 probes; the formula gives 5.5. The difference comes from the fact that in the small example of nine keys most keys happened to be placed close to their own buckets. In the chaining table the average over successful searches is $14/9 \approx 1.56$, close to the 1.45 given by the formula.

Theorem 4.3 (Average cost of hash search)

Under the uniform hashing assumption, in a hash table whose load factor is kept below a constant $\alpha_0 < 1$, the average search time is $O(1)$.

Proof. If $\alpha \le \alpha_0$, the average number of probes is at most $1 + \alpha_0$ in chaining and at most $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha_0)^2}\right)$ in linear probing. Neither bound depends on n; both are constants. Computing the hash value also takes constant time when the key length is taken as constant. The total average cost is constant. □

The way to keep the load factor constant is to grow the table when it reaches a certain fullness. The common practice is this: when α exceeds a threshold such as 0.75, a new table twice the size is allocated and all keys are placed again in the new table. Since hash values depend on the number of buckets, the keys are not copied but rehashed. This takes $O(n)$, but it is done rarely; as with doubling a dynamic array, the amortized cost per insertion is constant. Java's `HashMap` class uses a threshold of 0.75 by default.

The worst case, however, is $O(n)$. A bad hash function can drop all keys into the same bucket. The search then scans a single long list or cluster; that is, it degrades to linear search. With the function $h(x) = x \% 10$, keys such as 10, 20, 30, … all fall into bucket 0. In linear probing, clustering inflates the cost quickly as α approaches 1. If the table becomes completely full and the key is not in it, an empty bucket is never found, and the loop of `hashSearch` never ends.

Common mistake

Letting a linear-probing table fill up completely. Because `hashSearch` stops only at an empty bucket, searching for an absent key in a full table leads to an infinite loop. The load factor must be limited, or the loop must be stopped after at most m probes.

4.5 Interpolation search

Binary search looks at the exact middle of the range whatever the key is. People do not search that way. When looking for "Yılmaz" in a phone book, you open the book near the end, not in the middle; for "Akın", near the front. Interpolation search puts this behavior into a formula.

4.5.1 The idea: estimating the position

Interpolation search is used on sorted collections. It assumes that the values are spread at equal intervals across the range: the difference between consecutive items is about the same. Under this assumption the relation between index and value is close to a straight line. The position of the key is

estimated on the line through the smallest and largest items of the range. The estimated position is probed. If the value does not match, the search continues, just as in binary search, in the narrower range to the left or right of the estimate.

The estimation formula comes from the equation of a line. The ends of the range are the points *(lo, arr[lo])* and *(hi, arr[hi])*. On the line through these two points, the index of the point whose value is *key* is as follows:

Definition 4.10

The index to probe in interpolation search

> pos = lo + (key − arr[lo]) · (hi − lo) / (arr[hi] − arr[lo])

is computed by the formula. The division is integer division; the result is rounded down.

The structure of the formula can be read intuitively. The ratio *(key − arr[lo]) / (arr[hi] − arr[lo])* shows how far along the value range the key lies; it is a number between 0 and 1. Multiplying this ratio by the length of the index range, *hi − lo*, and adding it to *lo* gives the estimated position. If the key equals the smallest value the ratio is 0 and *lo* is probed; if it equals the largest value the ratio is 1 and *hi* is probed. Binary search can be thought of as a special case that always takes this ratio to be ½.

Example 4.9

Searching for 70 in `arr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}`. *lo = 0*, *hi = 9*, *arr[lo] = 10*, *arr[hi] = 100*.

> pos = 0 + (70 − 10) · (9 − 0) / (100 − 10) = 60 · 9 / 90 = 540 / 90 = 6

`arr[6] = 70`: found in a single probe. Because the distribution is exactly uniform, the estimate is perfect. Binary search probes indices 4, 7, 5, 6 in the same search and finds it on the fourth probe.

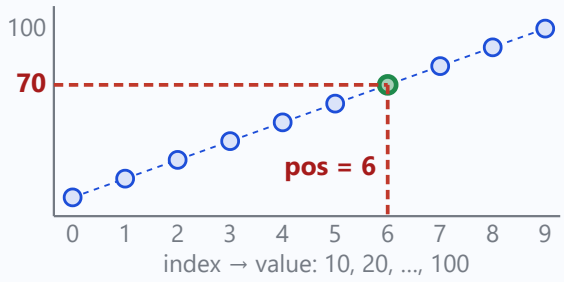


Figure 4.18: In a uniform distribution the index–value relation is a straight line. Going from 70 on the value axis to the line and from there down to the index axis gives the estimate 6.

4.5.2 Code: interpolationSearch()

```
int interpolationSearch(int[] arr, int n,
                        int key) {
  int lo = 0, hi = n - 1;
  while (lo <= hi && key >= arr[lo]
                  && key <= arr[hi]) {
    if (arr[hi] == arr[lo]) return lo; // avoid /0
    long num = (long) (key - arr[lo]) * (hi - lo);
    int pos = lo + (int) (num / (arr[hi] - arr[lo]));
    if (arr[pos] == key) return pos;
    if (arr[pos] < key) lo = pos + 1;
    else hi = pos - 1;
  }
  return -1;
}
```

`lo` and `hi` are again the bounds of the search range. The loop condition is richer than in binary search: the range must not be empty (`lo <= hi`), and the key must lie between the end values of the range (`key >= arr[lo]` and `key <= arr[hi]`). If the key is outside the ends, it cannot be in the range; the loop is not entered and *−1* is returned. This check also guarantees that the estimated index stays inside the range. If the key were outside the ends, the ratio would come out smaller than 0 or larger than 1, and the index could fall outside the array.

The first line of the loop prevents division by zero. If `arr[hi] == arr[lo]` , all values in the range are equal. Since the loop condition holds, the key equals this value too, and `lo` can be returned. Next the numerator and the index are computed. Writing the numerator and denominator separately fixes the order of operations explicitly: multiplication first, then division. If the division were done first, integer division would round the ratio down to 0 most of the time. Finally the value at the estimated index is probed. If equal, the index is returned. If smaller, the key is to the right: `lo = pos + 1` . If larger, the key is to the left: `hi = pos - 1` . If the range becomes empty or the key falls outside the ends, the loop ends and *−1* is returned.

4.5.3 Code trace: interpolationSearch(arr, 15, 24)

An array whose distribution is roughly, but not exactly, uniform is this:

> `arr = {10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47}`, $n = 15$

The key is 24. The first eleven items of the array are dense, and the last four are sparse.

Probe	`lo`	`hi`	Computation	`pos`	`arr[pos]`	Action
1	0	14	$0 + (24 - 10) \cdot 14 / (47 - 10) = 196/37$	5	19	`lo = 6`
2	6	14	$6 + (24 - 20) \cdot 8 / (47 - 20) = 6 + 32/27$	7	21	`lo = 8`
3	8	14	$8 + (24 - 22) \cdot 6 / (47 - 22) = 8 + 12/25$	8	22	`lo = 9`
4	9	14	$9 + (24 - 23) \cdot 5 / (47 - 23) = 9 + 5/24$	9	23	`lo = 10`
5	10	14	$10 + (24 - 24) \cdot 4 / (47 - 24) = 10 + 0/23$	10	24	`return 10`

Table 4.12: Trace of `interpolationSearch(arr, 15, 24)`.

The first estimate is $196/37 \approx 5.3$, which rounds down to 5. `arr[5] = 19` is smaller than the key; the search continues from index 6. With each probe the left end of the range rises, but the right end stays at 47. Because 47 is markedly larger than the rest of the array, it pulls the estimates to the left: the ratio is $4/27$ in the second probe and $2/25$ in the third. The search approaches the target step by step and finds 24 at index 10 on the fifth probe.

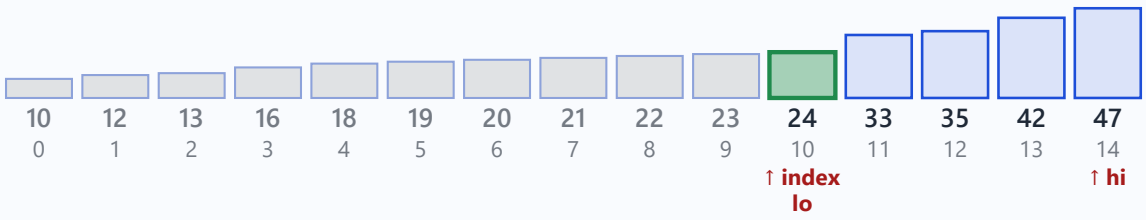


Figure 4.19: The end of interpolation search. The height of the bars shows the value. The probed indices 5, 7, 8, 9 (gray) turned out smaller than the key, and 24 was found at index 10.

In this array binary search would find 24 in four steps, probing indices 7, 11, 9, 10. Here interpolation search is slower than binary search. Although the distribution of the array is "roughly uniform", the sparse values at the end have spoiled the estimates. The speed of interpolation search depends entirely on how uniform the distribution is.

4.5.4 Complexity and the bad case

Case	Order
Best	$O(1)$

Case	Order
Average (uniform distribution)	*O(log log n)*
Worst (skewed distribution)	*O(n)*
Extra memory	*O(1)*

Table 4.13: The cost of interpolation search.

For keys that are uniformly and randomly distributed, the average number of probes is *O(log log n)*. This result was proved by Perl, Itai and Avni in 1978. The proof is beyond the scope of these notes, but the intuition can be put as follows. In a uniform distribution the error of each estimate is of the order of the square root of the remaining range size. That is, the range shrinks with each probe from about n to $\sqrt{n}$: $n \to n^{1/2} \to n^{1/4} \to \ldots$. The number of steps needed for the range to shrink to constant size is of order *log log n*. Binary search takes the logarithm once; interpolation search takes it twice.

In numbers the difference is striking. For $n = 10^6$, $\log_2 \log_2 n \approx \log_2 19.93 \approx 4.3$, while binary search makes about 20 probes. But this gain has two costs:

- Each probe is more expensive than in binary search: instead of an addition and a division it contains a multiplication, a division and several subtractions. If reading from memory is slow, for example when the data is on disk, reducing the number of probes is worth this cost. If the data is in memory, the gain is often small.
- The intermediate product $(key - arr[lo]) \cdot (hi - lo)$ can overflow an `int` for large values. If the values are of order 10^9 and $hi - lo$ is of order 10^6, the product is 10^{15}, far beyond the limit $2^{31} - 1$. That is why `num` in the code is a `long` and the multiplication is done in `long`.

Example 4.10

Searching for 9 in `arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 1000}`. The array is sorted, but the last item is much larger than the others.

1. $lo = 0$: $pos = 0 + (9 - 1) \cdot 9 / (1000 - 1) = 72/999 \to 0$. `arr[0] = 1 < 9`, `lo = 1`.
2. $lo = 1$: $pos = 1 + (9 - 2) \cdot 8 / 998 = 1 + 56/998 \to 1$. `arr[1] = 2 < 9`, `lo = 2`.
3. In the same way the numerator always stays smaller than the denominator, and in each probe the index advances by only one: 2, 3, 4, 5, 6, 7.
4. $lo = 8$: $pos = 8 + 0 \cdot 1 / 991 = 8$. `arr[8] = 9`: found.

The probed indices are 0, 1, 2, 3, 4, 5, 6, 7, 8: **9 probes**. Interpolation search has degenerated into linear search. Binary search finishes the same search in 3 probes, with indices 4, 7, 8.

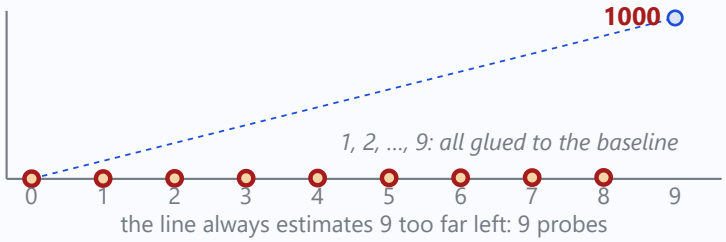


Figure 4.20: A skewed distribution. The extreme value 1000 makes the estimation line almost horizontal; all the values from 1 to 9 stay glued to the baseline, and each estimate lands on the next index.

The source of the bad case is that the formula trusts the end values. A single outlier spoils all the estimates in the rest of the range. The same happens in an exponentially growing array, for example when searching for 32 in the array 2, 4, 8, ..., 256: the estimates advance one at a time, 0, 1, 2, 3, 4. The worst case is *O(n)*. That is why interpolation search is used on numeric keys whose distribution is known and uniform. Sequential ID numbers or timestamps taken at regular intervals are good examples. In practice, hybrid methods that mix interpolation steps with binary search steps are also used; this keeps the worst case at *O(log n)*.

Common mistake

Skipping the end-value check in interpolation search. If `key >= arr[lo] && key <= arr[hi]` is removed from the loop condition, then for a value outside the range the index comes out negative or greater than `hi`, and an out-of-bounds access error occurs. If the check `arr[hi] == arr[lo]` is forgotten, a range whose items are all equal causes a division by zero.

4.6 Jump search

Binary search makes big leaps inside the array: from the middle to a quarter, from there to an eighth. If the data is in memory, these leaps cost nothing. But in some environments going backwards, or jumping to a distant position, is expensive. Jump search is designed for such environments.

4.6.1 The idea: advancing in blocks

Jump search works like a hurdler clearing hurdles as they run; the array must be sorted. The list is divided into blocks of size m, and the block size is usually chosen as $m = \sqrt{n}$. A big step is taken from the beginning of the list, and the item at that position, the last item of the block, is checked. If this item is smaller than the key, the key cannot be in this block; we jump to the next block. If the last item of the block is larger than or equal to the key, the key, if present, is in this block. The search returns to the start of that block and searches linearly inside it.

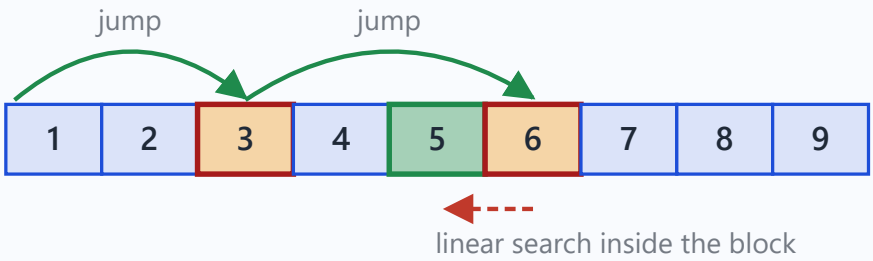


Figure 4.21: Jump search on an array of nine items ($m = 3$). First we jump to the block ends with values 3 and 6; while searching for 5 we stop at 6 and search linearly inside the block.

Going back does not require big leaps as in binary search. The data is always read forward; only once, in the last block, do we step back at most m steps. That is why jump search is preferred in environments where going backwards is expensive. A magnetic tape is read quickly forward, and rewinding is slow. A large file on disk is read in blocks; each block read is expensive, and the blocks that are skipped are never read.

4.6.2 Code: jumpSearch()

```
int jumpSearch(int[] arr, int key,
               int n) { // n ≤ arr.length
  if (n == 0) return -1;
  int block = (int) Math.sqrt(n);
  int jump = block, pos = 0;
  while (arr[Math.min(jump, n) - 1] < key) {
    pos = jump; jump += block;     // jump
    if (pos >= n) return -1;
  }
  while (arr[pos] < key) {     // linear
    pos++;
    if (pos == Math.min(jump, n)) return -1;
  }
  if (arr[pos] == key) return pos;
  return -1;
}
```

The jump distance is computed with `Math.sqrt(n)` and converted to an integer; `block` is $\lfloor\sqrt{n}\rfloor$. `pos` marks the start of the current block and `jump` the start of the next block. The last item of the block is `arr[jump - 1]`.

The first loop skips blocks. In each iteration the last item of the block is examined. The expression `Math.min(jump, n)` keeps us from running off the array when the last block is not completely full; in that case the last item of the array is examined. If the last item of the block is smaller than the key, the

key is not in this block: `pos` moves to the start of the next block and `jump` one block ahead. If `pos` has passed the end of the array, the key is larger than all items of the array and *−1* is returned.

When the first loop ends, the key, if present, lies in the range `[pos, jump)`. The second loop scans this range linearly. As long as the scanned item is smaller than the key, `pos` increases by one. If the end of the block is reached, the key is not in this block and *−1* is returned. The loop stops at an item that is not smaller than the key. The check on the last line looks at whether this item equals the key: if so the index is returned, otherwise *−1*.

4.6.3 Code trace: jumpSearch(arr, 69, 10)

The array has ten items: `{2, 9, 30, 32, 38, 47, 61, 69, 79, 81}`. The key is 69.

Example 4.11

1. $n = 10 \rightarrow$ `block = ⌊√10⌋ = 3`; `jump = 3`, `pos = 0`.
2. `arr[min(3, 10) − 1] = arr[2] = 30 < 69` → jump one block: `pos = 3`, `jump = 6`.
3. `arr[min(6, 10) − 1] = arr[5] = 47 < 69` → jump one block: `pos = 6`, `jump = 9`.
4. `arr[8] = 79 ≥ 69` → the jumping ends. The key may be in the block `[6, 9)`.
5. Linear scan: `arr[6] = 61 < 69` → `pos = 7`; 7 is not equal to 9, the end of the block.
6. `arr[7] = 69` is not smaller than the key, and the loop stops. In the final check `arr[7] == 69` is true; the method returns 7.

The probed indices are 2, 5, 8, 6, 7 in that order: 5 probes. After looking at index 8 the search returned to 6; this is the only step back in the search.

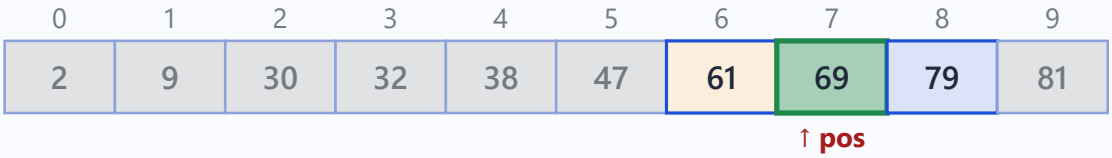


Figure 4.22: The end of jump search. The first two blocks that were jumped over are gray; inside the last block 61 was passed and 69 was found.

An unsuccessful search goes similarly. If 50 is searched for in the same array, indices 2, 5 and 8 are examined; since 79 ≥ 50, we move to the block `[6, 9)`. `arr[6] = 61` is already larger than 50, so the linear loop never runs. In the final check 61 ≠ 50, so *−1* is returned. The probed indices are 2, 5, 8, 6.

4.6.4 Why √n?

With block size m, the worst-case number of probes is found as follows. In the jumping phase at most n/m block ends are examined. In the linear phase at most $m - 1$ items inside the block are scanned; the last item of the block had already been examined. The total:

$$f(m) = n/m + (m - 1)$$

The two terms behave in opposite directions. If the blocks are small, the number of jumps grows; for $m = 1$ the search turns into linear search. If the blocks are large, the scan inside the block gets long; for $m = n$ we again return to linear search. The best value is somewhere in between.

Theorem 4.4 (Best block size in jump search)

The function $f(m) = n/m + m - 1$ attains its minimum at $m = \sqrt{n}$. This value is $2\sqrt{n} - 1$; hence the worst case of jump search is $O(\sqrt{n})$.

Proof. The derivative of f is $f'(m) = -n/m^2 + 1$. Setting the derivative to zero gives $m^2 = n$, that is, $m = \sqrt{n}$. Since the second derivative $f''(m) = 2n/m^3 > 0$, this point is a minimum. Substituting gives $f(\sqrt{n}) = n/\sqrt{n} + \sqrt{n} - 1 = 2\sqrt{n} - 1$. □

The same result can be seen without derivatives. If the product of two positive numbers is fixed, their sum is smallest when the numbers are equal. The product of n/m and m is always n; so the sum is smallest when $n/m = m$, that is, when $m = \sqrt{n}$.

n	$m = \sqrt{n}$	$\approx 2\sqrt{n} - 1$	Binary
100	10	19	7
10,000	100	199	14
10^6	1,000	1,999	20

Table 4.14: Worst-case probe counts of jump search and binary search.

The complexity is $O(\sqrt{n})$: it lies between the $O(n)$ of linear search and the $O(\log n)$ of binary search. The extra memory is $O(1)$. Choosing the block size wrongly raises the cost noticeably. For $n = 10{,}000$, if $m = 50$ is chosen the worst case is $200 + 49 = 249$ probes; with the best choice, $m = 100$, it is 199.

Once the block is found, it can be scanned in two directions. The code scans forward from the start of the block (6 → 7 in the search for 69). It could also have scanned backward from the end of the block (8 → 7). Either way makes at most $m - 1$ extra probes. One more improvement is possible: once the block is found, instead of a linear search inside it we can again jump (with steps of $\sqrt{m}$) or run binary search. But if binary search can be used, jump search is unnecessary from the start. Jump search makes sense only where going backwards is expensive.

4.7 Exponential search

Binary search must know the length of the array; in its first step it assigns `hi = n - 1`. In some cases the length is unknown or very large. For example, a sorted data stream keeps getting longer, or reaching the end of a very large file is expensive. Exponential search first finds a range that contains the key, then runs binary search in that range.

4.7.1 The idea: doubling the bound

Exponential search divides the sorted list into exponentially growing pieces. The bound, that is `bound`, is doubled at each step: 1, 2, 4, 8, 16, ... We stop when `arr[bound]` is larger than the key. At that moment the key, if present, is in the range `[bound/2, bound]`, because the value at the previous bound, `bound/2`, was not larger than the key. Binary search is run in this range. The method is based on the search that Bentley and Yao proposed in 1976 for unbounded arrays.

Example 4.12

In a list of 1024 items, let the key be around index 600. The bound passes through the values 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and at all of them the value is smaller than the key. When the bound reaches 1024 we are outside the array and the doubling stops. The search continues with binary search in the range *[512, min(1024, 1023)] = [512, 1023]*. The doubling phase makes 11 tests (counting index 0), and binary search makes at most 10 probes in the range of 512 items.

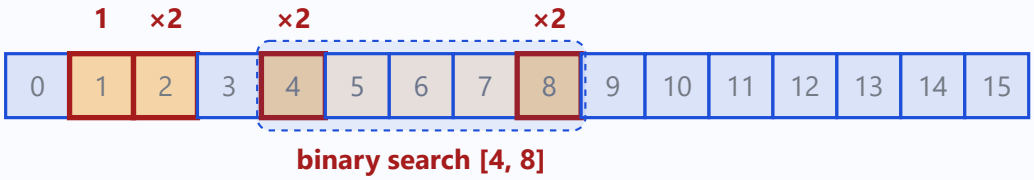


Figure 4.23: Exponential search: indices 1, 2, 4 and 8 are probed; the bound doubles at every step. When the value at index 8 turns out larger than the key, binary search is run in the range *[4, 8]*.

The strength of the method is that the cost depends not on n but on the position i of the sought item. If the length of the array is unknown (unbounded or streaming data), or if the sought item is near the front, exponential search is ideal. If the key is at index 10, it does not matter whether the array has a million or a billion items; the search ends in a few steps.

4.7.2 Code: exponentialSearch()

```
int exponentialSearch(int[] arr, int key, int n) {
  if (n == 0) return -1;
  if (arr[0] == key) return 0;
  int bound = 1;
  while (bound < n && arr[bound] <= key)
    bound *= 2;               // 1, 2, 4, 8, ...
  int lo = bound / 2, hi = Math.min(bound, n - 1);
  while (lo <= hi) {        // binary search
    int mid = (lo + hi) / 2;
    if (arr[mid] == key) return mid;
    if (arr[mid] < key) lo = mid + 1;
    else hi = mid - 1;
  }

  return -1;
}
```

The method first checks index 0 separately, because the doubling starts at 1 and index 0 is never probed during doubling. Then `bound = 1` is set. As long as `arr[bound] <= key`, `bound` is multiplied by 2. The condition `bound < n` keeps us from running off the array. At the end of the loop, `bound` marks the end of the range in which the key may lie, and `bound / 2` its start. Because after the last doubling `bound` may have gone beyond the array, the upper limit is taken as `Math.min(bound, n - 1)`. The remaining part is searched exactly as in iterative binary search.

If, in the last phase, a linear scan from `bound / 2` onward had been used instead of binary search, the cost would have been *O(i)*; if the key were absent, the scan would have gone to the end of the array. The length of the range is proportional to *i*, and searching it with binary search lowers the cost to *O(log i)*.

4.7.3 Code trace: exponentialSearch(arr, 47, 10)

The array is the one from the jump search example: `{2, 9, 30, 32, 38, 47, 61, 69, 79, 81}`. The key is 47.

Phase	Index probed	Value	Decision
initial check	0	2	*2 ≠ 47*
doubling	1	9	*9 ≤ 47* → `bound = 2`
doubling	2	30	*30 ≤ 47* → `bound = 4`
doubling	4	38	*38 ≤ 47* → `bound = 8`
doubling	8	79	*79 > 47* → stop; range *[8/2, min(8, 9)] = [4, 8]*
binary	6	61	`mid = (4 + 8)/2 = 6`; *61 > 47* → `hi = 5`
binary	4	38	`mid = (4 + 5)/2 = 4`; *38 < 47* → `lo = 5`

Phase	Index probed	Value	Decision
binary	5	47	`mid = (5 + 5)/2 = 5` ; equal → `return 5`

Table 4.15: Trace of `exponentialSearch(arr, 47, 10)` .

In total 8 probes were made: 0, 1, 2, 4, 8, 6, 4, 5. Index 4 was looked at twice. The test 38 ≤ 47 was made during doubling, and binary search probed 4, the lower end of the range, again. The doubling phase had already shown that the value at index 4 is not larger than the key; this repetition would have been avoided had binary search been started with `lo = bound / 2 + 1` . In this array binary search finds 47 in 3 probes: if the key is not near the start of the array, exponential search is slower than binary search.

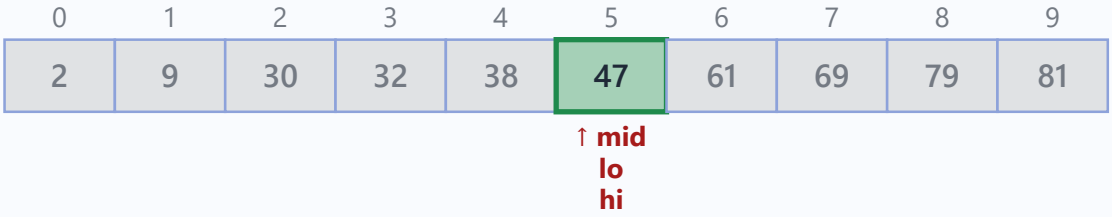


Figure 4.24: The end of exponential search. In the third iteration of binary search, `lo = hi = mid = 5` and 47 has been found.

4.7.4 Complexity

Theorem 4.5 (Cost of exponential search)

If the key is at index $i \geq 1$, exponential search makes $O(\log i)$ probes. In the worst case ($i \approx n$) the cost is $O(\log n)$, in the best case $O(1)$; the extra memory is $O(1)$.

Proof. Doubling stops when `bound` first exceeds i or reaches the end of the array. Since `bound` takes the values 1, 2, 4, ..., this happens at the smallest k with $2^k > i$; that is $k = \lfloor \log_2 i \rfloor + 1$. The doubling phase looks at the positions 1, 2, ..., 2^k, that is, $k + 1$ positions. The range found is $[2^{k-1}, 2^k]$ and contains at most $2^{k-1} + 1 \leq i + 1$ items. Binary search in this range makes at most about $\log_2 i + 1$ probes. The total of the two phases is about $2 \log_2 i$, that is, $O(\log i)$. □

Exponential search is faster than binary search if the key is near the front, and about twice as slow if it is near the end. Table 4.16 compares the two methods for different positions in an array of a million items.

i	Exponential ($\approx 2 \log_2 i$)	Binary ($\approx \log_2 n$)
10	≈ 7	20
1,000	≈ 20	20

i	Exponential ($\approx 2 \log_2 i$)	Binary ($\approx \log_2 n$)
500,000	≈ 38	20

Table 4.16: Probe counts of exponential and binary search, $n = 10^6$.

The break-even point is around $i \approx \sqrt{n}$: the equation $2 \log_2 i = \log_2 n$ gives $i = \sqrt{n}$. For a million items this point is 1000, and the table shows it.

The main uses of exponential search are these: sorted lists of unknown length, records near the start of a large file, and sorted data streams. One application of the same idea is *galloping* search. When the Timsort algorithm merges two sorted arrays, a long stretch of one array sometimes comes before the other. Timsort finds the end of that stretch not item by item but by galloping search.

Common mistake

Taking the upper limit directly as `bound` instead of `Math.min(bound, n - 1)`. After the last doubling, `bound` may exceed the length of the array; binary search then reads outside the array with `arr[mid]` and an `ArrayIndexOutOfBoundsException` is thrown. When 81 is searched for in the array of 10 items, doubling stops at `bound = 16`; if the limit is not clamped, the first midpoint is $(8 + 16)/2 = 12$.

4.8 Comparison and choosing a method

Each of the seven methods is the best in some situation. To choose correctly, we first put the methods' costs side by side, and then see how they behave on the same data.

4.8.1 Comparison table

Algorithm	Precondition	Best	Average	Worst	Extra memory
Linear	none	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary	sorted	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Hash (table)	hash table	$O(1)$	$O(1)$	$O(n)$	$O(m)$
Interpolation	sorted, uniform	$O(1)$	$O(\log \log n)$	$O(n)$	$O(1)$
Jump	sorted	$O(1)$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(1)$
Exponential	sorted	$O(1)$	$O(\log i)$	$O(\log n)$	$O(1)$

Table 4.17: Comparison of the search algorithms. i: index of the sought item, m: number of buckets.

Three points should be kept in mind when reading the table. First, the average $O(1)$ of a hash table depends on the load factor being kept constant; if the table is left to fill up, this guarantee is lost. Second, for the methods that need a sorted array, if the array is not sorted to begin with, a one-time

sorting cost of *O(n log n)* is added. Third, the worst-case column is the guarantee the algorithm gives. A hash table and interpolation search are very fast on average, but give no guarantee against a bad input. Binary search is *O(log n)* on every input. This predictability makes it the default choice in most situations.

4.8.2 Probe counts on the same array

The orders of growth speak of large *n*. It is also instructive to see how many probes the methods actually make on a small array. We use the ten-item array from the jump search and exponential search examples:

```
arr = {2, 9, 30, 32, 38, 47, 61, 69, 79, 81}
```

Key	Linear	Binary	Jump	Interpolation	Exponential	Result
69	8 (0–7)	2 (4, 7)	5 (2, 5, 8, 6, 7)	1 (7)	7 (0, 1, 2, 4, 8, 6, 7)	7
47	6 (0–5)	3 (4, 7, 5)	5 (2, 5, 3, 4, 5)	1 (5)	8 (0, 1, 2, 4, 8, 6, 4, 5)	5
2	1 (0)	3 (4, 1, 0)	2 (2, 0)	1 (0)	1 (0)	0
50	10 (0–9)	4 (4, 7, 5, 6)	4 (2, 5, 8, 6)	1 (5)	8 (0, 1, 2, 4, 8, 6, 4, 5)	*−1*

Table 4.18: Probe counts on the same array. Each cell gives the number of probes and, in parentheses, the indices probed.

A few lessons come out of the table. Because the array is almost uniformly distributed, interpolation search settles every query in a single probe. For 69 the estimate is $0 + (69 - 2) \cdot 9 / 79 = 603/79 \approx 7.6$ → 7, which lands directly on the target. When 50 is searched for, interpolation search probes the 47 at index 5. Then `lo = 6`, and since the key is smaller than 61, the lower end of the new range, the loop ends immediately; the result is *−1*.

Binary search makes at most 4 probes per query; this is the bound $\lfloor \log_2 10 \rfloor + 1 = 4$. Jump search varies with the block. When 47 is searched for, the last item of the second block is exactly 47, but the code does not stop at the block end; it goes back to the start of the block and scans indices 3, 4, 5. Exponential search shines when the key is at the start of the array: for 2, a single probe suffices. As the key approaches the end, the doubling and binary search phases together become more expensive than binary search. In the searches for 47 and 50, index 4 is probed twice: once in doubling and once in binary search.

In an array this small the differences are a few probes. As the array grows, the differences in order become decisive. With a million items, linear search makes half a million probes on average, jump search at most about two thousand, and binary search at most 20.

4.8.3 Which algorithm when?

The choice depends on the properties of the data and the shape of the queries:

- **Linear search:** unsorted or small data, a linked list, a one-off search.

- **Binary search:** a sorted array and many queries; the general-purpose default.
- **Hash table:** exact-match queries only and many insertions and searches; order is not needed.
- **Interpolation search:** sorted and uniformly distributed numeric keys, for example sequential ID numbers.
- **Jump search:** sorted environments where going backwards is expensive, for example tape or a disk read block by block.
- **Exponential search:** a sorted list of unknown length, or cases where the key is near the start.

These rules can be turned into a sequence of questions. The first question is whether the data is sorted. If it is unsorted, there are two routes: linear search if one or a few searches will be made, or building a hash table if many searches will be made. If the data is sorted, the default choice is binary search. Interpolation search is considered if the distribution is uniform, exponential search if the length is unknown, and jump search if going backwards is expensive.

Example 4.13

The choice is made as follows in three scenarios.

1. **A compiler's symbol table.** Variable names are inserted and looked up all the time; order is not needed, only the question "is this name defined?" is asked. A hash table fits best: both insertion and search are *O(1)* on average.
2. **Prefix search in a dictionary application.** All words that begin with "sea" are wanted. A hash table scatters the words, so it cannot help with this query. In a sorted array of words, first-position binary search finds the place where "sea" would be inserted, and reading proceeds forward from there.
3. **A sorted log file to which one record is added per second.** The file keeps growing, and queries usually concern the most recent records. Running exponential search backward from the end of the file finishes in *O(log i)* cost since the sought record is near the end; the full length of the file does not even need to be known.

4.9 Common mistakes

Search algorithms are short pieces of code, but they are very prone to mistakes. Most mistakes show up at boundary conditions and slip past tests made with small examples.

Common mistake

Binary search on an unsorted array. The program raises no error, but may return a wrong result. If 30 is searched for in `{30, 10, 20}`, the middle item is 10; since 30 is larger, the search goes right, the 20 is smaller, and *−1* is returned. Yet 30 is at index 0. Binary,

interpolation, jump and exponential search need a sorted array.

Common mistake

Off-by-one errors in the bounds. `while (lo < hi)` skips a single-item range. The assignment `hi = mid` may fail to shrink the range and can send the loop into an infinite run.

Common mistake

Overflow in the midpoint. In very large arrays `(lo + hi) / 2` overflows and gives a negative index. `lo + (hi - lo) / 2` does not overflow.

Common mistake

Division in interpolation search. If `arr[hi] == arr[lo]`, there is a division by zero. If the key is outside the end values, the estimated index falls outside the array. The range must therefore be checked first.

Common mistake

A full hash table. If no empty bucket is left in linear probing, the search for an absent key never ends. The load factor must be limited, or the search must stop after *m* probes.

Common mistake

The bound in exponential search. If `Math.min(bound, n - 1)` is forgotten, an `ArrayIndexOutOfBoundsException` is thrown. If a linear scan is used instead of binary search in the last phase, the *O(log i)* cost is also spoiled and rises to *O(i)*.

These mistakes share one lesson: once search code has been written, it must be tried on boundary cases. At the very least, the following cases should be tried: an empty array, a one-item array, the key being the first or last item, the key being smaller or larger than all values, and the key falling between two items. Since binary search is hard to write correctly, it is also a good habit in Java to use `Arrays.binarySearch` or `Collections.binarySearch` whenever possible.

4.10 Summary

- **The search problem:** find the index of the key in an array; return *−1* if it is absent. The cost is measured by the number of probes (comparisons).
- **Linear search:** compares one by one from start to end; no precondition. In a successful search it makes *(n + 1)/2* comparisons on average and *n* in the worst case: *O(n)*. On unsorted data there is nothing better.
- **Sentinel linear search:** by writing the key into the last cell it removes the `i < n` test from the loop; the number of comparisons drops from about *2n* to *n + 2*, and the order does not change.
- **Binary search:** compares with the middle item of a sorted array and eliminates half. It makes at most $\lfloor \log_2 n \rfloor + 1$ comparisons: *O(log n)*. The iterative version uses *O(1)* and the recursive version *O(log n)* extra memory. A sorted array is an implicit balanced binary search tree.
- **Details of binary search:** the midpoint is computed as `lo + (hi - lo) / 2` to avoid overflow; the bounds are updated by *±1*; for the first position, the result is recorded on equality and the search continues to the left.
- **Hash search:** computes the location with *h(key)*. Collisions are resolved by linear probing or chaining. If the load factor $\alpha = n/m$ is kept constant, the average is *O(1)*; the worst case is *O(n)*. In linear probing, primary clustering inflates the cost quickly as α approaches 1.
- **Interpolation search:** estimates the position by proportion from the end values. It is *O(log log n)* on a uniform distribution and *O(n)* on a skewed one.
- **Jump search:** jumps over blocks of size $\sqrt{n}$, then searches linearly inside the block. The worst case is about $2\sqrt{n} - 1$ probes, that is $O(\sqrt{n})$; it makes sense in environments where going backwards is expensive.
- **Exponential search:** doubles the bound 1, 2, 4, ..., then runs binary search in the range *[bound/2, bound]*. The cost depends on the index of the sought item: *O(log i)*. It is used on lists of unknown length and for searches near the start.
- **Choosing:** if the data is unsorted, linear search or a hash table; if sorted, binary search by default; interpolation on a uniform distribution, exponential for unknown length, jump search where going backwards is expensive.

4.11 Key terms

search problem probe best case average case worst case linear search sentinel binary search logarithmic divide and conquer decrease and conquer recurrence relation hash table hash function bucket collision open addressing linear probing primary clustering (separate) chaining load factor interpolation search jump search exponential search

4.12 Test Yourself

1. The call `sentinelSearch(arr, 5, 9)` runs on `arr = {12, 5, 9, 20, 7}`. What does the method return, and how many comparisons are made in total?

   Answer

   The key 9 is written into the last cell as the sentinel, and `last = 7`. After 12 ≠ 9 and 5 ≠ 9, the loop stops at the real 9 at index 2: 3 comparisons. 7 is put back. `i < n - 1`, that is *2 < 4*, is true; because of the short-circuit `||` the second condition is never tested. The method returns 2; 4 comparisons are made in total.

2. Iterative binary search looks for 50 in the array `{4, 9, 13, 20, 26, 31, 40, 48, 55, 61, 77}`. Which indices are probed? When the loop ends, what are `lo` and `hi`, and what does `lo` indicate?

   Answer

   The probed indices are 5 (31 < 50, `lo = 6`), 8 (55 > 50, `hi = 7`), 6 (40 < 50, `lo = 7`) and 7 (48 < 50, `lo = 8`). The loop ends with `lo = 8`, `hi = 7` and returns *–1*. `lo` is the position at which 50 would be inserted without breaking the order: index 8, between 48 and 55.

3. In the call `firstIndex({1, 3, 3, 3, 3, 3, 8}, 3)`, what are the values of `mid` and `result` in each iteration? Which indices does the version that finds the position of the last 3 probe on the same array?

   Answer

   Iteration 1: `mid = 3`, equal, `result = 3`, `hi = 2`. Iteration 2: `mid = 1`, equal, `result = 1`, `hi = 0`. Iteration 3: `mid = 0`, 1 < 3, `lo = 1`. The loop ends and 1 is returned. The last-position version goes right on equality with `lo = mid + 1`: it probes indices 3, 5, 6 and returns 5. The value 3 occurs *5 – 1 + 1 = 5* times.

4. The keys 15, 22, 8, 29, 3 are inserted in order into an empty table with *m = 7*, *h(x) = x % 7* and linear probing. What is the final table? Then, how many buckets are probed when 36 is searched for?

   Answer

   15, 22, 8 and 29 all fall into bucket 1: 15 → 1, 22 → 2, 8 → 3, 29 → 4 (probes 1, 2, 3, 4). The bucket of 3 is 3, but 3 and 4 are full; 3 goes into 5. The table becomes `[–, 15, 22, 8, 29, 3, –]`. For 36, *h = 1*: the keys in buckets 1, 2, 3, 4, 5 are passed and the search stops at the empty bucket 6. There are 6 probes, and the result is *–1*.

5. If the same five keys are inserted into a chaining table with $m = 7$ buckets, what is the load factor and which lists are formed? How many keys are compared when 36 is searched for?

   Answer

   $\alpha = 5/7 \approx 0.71$. The list of bucket 1 is 15 → 22 → 8 → 29, that of bucket 3 is only 3; the other buckets are empty. When 36 is searched for, the 4 keys in the list of bucket 1 are compared and the list ends. In this example chaining is slow too: the source of the problem is that the hash function gathers the keys in a single bucket, not the collision resolution method.

6. On the array 5, 10, 15, ..., 100 (20 items), in how many probes does interpolation search settle the searches for 65 and for 63? Which indices does binary search probe for 65?

   Answer

   65: $pos = 0 + (65 - 5) \cdot 19 / 95 = 1140/95 = 12$, `arr[12] = 65`; 1 probe. 63: $58 \cdot 19 / 95 = 1102/95 \approx 11.6 \to 11$, `arr[11] = 60 < 63`, `lo = 12`. Now $63 < arr[12] = 65$, so the loop ends: 1 probe, result -1. Binary search probes indices 9, 14, 11, 12 for 65: 4 probes.

7. In jump search, what are the best block size and the worst-case probe count for $n = 25$ and $n = 400$? What happens if the block size is mistakenly chosen as 10 for $n = 400$?

   Answer

   $n = 25$: $m = 5$, $25/5 + 4 = 9$ probes. $n = 400$: $m = 20$, $20 + 19 = 39$ probes. If $m = 10$ is chosen, it is $400/10 + 9 = 49$ probes; 10 more than the best choice.

8. In an array of a million items, `arr[i] = 2i`. Which indices does exponential search probe while looking for 74 (index 37)? How many probes would binary search make at most?

   Answer

   First index 0: 0 ≠ 74. Doubling finds values smaller than 74 at indices 1, 2, 4, 8, 16, 32, and at index 64 the value is 128 > 74 and it stops. The range is $[32, 64]$. Binary search probes indices 48, 39, 35, 37 and finds 37. In total 12 probes: 0, 1, 2, 4, 8, 16, 32, 64, 48, 39, 35, 37. Binary search would have made at most $\lfloor \log_2 10^6 \rfloor + 1 = 20$ probes.

9. With `lo = 1,500,000,000` and `hi = 1,600,000,000`, what do `(lo + hi) / 2` and `lo + (hi -`

`lo) / 2` give in Java?

Answer

The true sum is 3,100,000,000 and exceeds $2^{31} - 1$. The `int` sum becomes $3{,}100{,}000{,}000 - 2^{32} = -1{,}194{,}967{,}296$; halved, it gives $-597{,}483{,}648$, and the access `arr[mid]` throws an `ArrayIndexOutOfBoundsException`. The safe form gives $1{,}500{,}000{,}000 + 50{,}000{,}000 = 1{,}550{,}000{,}000$.

4.13 Further reading

- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Sections 3.1 and 3.4: present a symbol table built on binary search in a sorted array, and hash tables (separate chaining, linear probing), with Java code and empirical measurements.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2nd ed., 1998), Sections 6.1, 6.2.1 and 6.4: give sentinel search, the history of binary search and the derivation of the linear-probing formulas in detail.
- J. Bentley, *Programming Pearls* (2nd ed., 2000), Columns 4 and 5: work through writing binary search correctly and justifying its correctness as an example.
- J. Bloch, “Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken”, Google Research Blog, June 2, 2006: a short article on how the midpoint overflow bug was discovered in the Java library.

4.14 References

1. J. L. Bentley, A. C. Yao, “An Almost Optimal Algorithm for Unbounded Searching”, *Information Processing Letters* 5(3), 1976.
2. J. Bentley, *Programming Pearls*, 2nd ed., Addison-Wesley, 2000.
3. J. Bloch, “Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken”, Google Research Blog, 2006.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Sections 2.3, 11).
5. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.
6. Y. Perl, A. Itai, H. Avni, “Interpolation Search — A Log Log N Search”, *Communications of the ACM* 21(7), 1978.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Sections 3.1 Binary search, 3.4 Hash tables).
8. B. Shneiderman, “Jump Searching: A Fast Sequential Search Technique”, *Communications of the ACM* 21(10), 1978.

PART II

Graph Algorithms

Chapters 5–8

Graph Traversal

Roads between cities, friendships in a social network, routers on the internet, and dependencies among the modules of a compiler are all described by the same structure: objects and the connections between them. That structure is a graph. The first step in asking questions about a graph is to walk through it in an orderly way. These notes begin with a short review of graph terminology and of how a graph is stored in memory. They then treat the two basic traversals in detail: depth-first search (DFS) and breadth-first search (BFS). The final sections build on these two: shortest paths in unweighted graphs, maze solving, strongly connected components by the algorithms of Tarjan and Kosaraju, topological sorting, and the bipartiteness test. Every algorithm comes with code, a step-by-step trace, and a cost analysis.

LEARNING OUTCOMES

Graph concepts

Tell directed, undirected, weighted, bipartite, and complete graphs, trees, and non-simple graphs apart; represent a graph by an adjacency list and an adjacency matrix and compare the memory and time costs of the two.

DFS

Write depth-first search both recursively and with an explicit stack; trace the visit order step by step; explain backtracking and why the two versions give different orders.

BFS

Write breadth-first search with a queue; use the notions of layer, parent, and unweighted shortest path; find a shortest path in a maze.

Tarjan and applications

Find strongly connected components with index and low-link values; compare the result with Kosaraju's algorithm; apply topological sorting and the bipartiteness test; justify the $O(V + E)$ running time of all the traversal algorithms.

5.1 Graphs and their representations

Much of the world is organized like a network. Cities are joined by roads, people by acquaintance, computers by cables. What these networks share is a handful of objects and connections between some pairs of them. It does not matter what the objects are, or whether a connection is physical or logical. Mathematics calls this common structure a graph. Graph algorithms answer the questions we ask about such networks. Is there a road between two cities? Which road is shortest? Into which groups does the network fall apart? If one cable is cut, is the network split in two?

5.1.1 What is a graph?

Definition 5.1

A **graph** $G = (V, E)$ consists of two sets. V is the set of **vertices** (also called nodes), and E is the set of **edges**; each edge joins two vertices. The two vertices an edge joins are its **endpoints**, and two vertices joined by an edge are **adjacent**.

The cost of a graph algorithm depends on two sizes of the graph. In these notes the letter V stands both for the vertex set and for the number of vertices, and E both for the edge set and for the number of edges; the context makes clear which is meant. Running times are written in terms of both numbers, as in $O(V + E)$ or $O(V^2)$. We use two numbers instead of a single n because graphs with the same number of vertices can have very different numbers of edges. A graph on ten vertices may have no edges at all, or as many as forty-five.

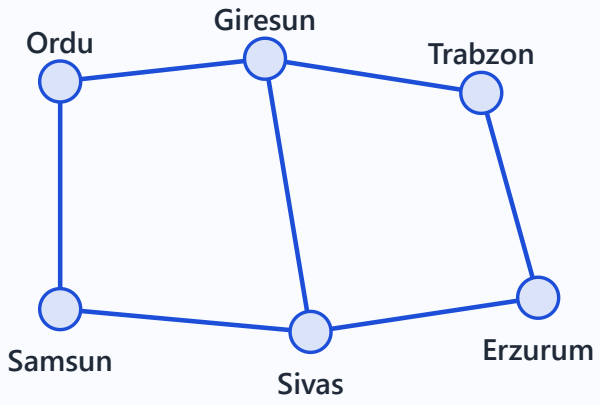


Figure 5.1: Six cities and the roads between them. Each city is a vertex and each road is an edge: $V = 6$, $E = 7$.

Figure 5.1 shows a small road network. To get from Ordu to Erzurum one can take Ordu–Giresun–Trabzon–Erzurum, Ordu–Samsun–Sivas–Erzurum, and so on. A sequence of vertices obtained by starting at a vertex and moving at each step along an edge to a neighbor is called a **path**. The **length**

of a path is the number of edges it contains; the path Ordu–Giresun–Trabzon–Erzurum has length 3. A path that returns to its starting vertex without repeating an edge is a **cycle**. In the figure, Ordu–Giresun–Sivas–Samsun–Ordu is a cycle.

The number of edges at a vertex is its **degree**. In the figure Giresun has degree 3 (Ordu, Trabzon, Sivas) and Samsun has degree 2. The degrees always sum to twice the number of edges, because each edge contributes one to the degree of each of its two endpoints. Here the degrees are 2, 3, 2, 2, 3, 2, which add up to $14 = 2 \cdot 7$. This simple observation is used directly in the cost analysis of the traversals: in an undirected graph the adjacency lists have total length $2E$.

A graph in which there is a path between any two vertices is **connected**. The network in the figure is connected: one can travel from every city to every city. A graph that is not connected falls apart into pieces that are connected inside and have no edge between them. Each such piece is a **connected component**.

5.1.2 Kinds of graphs

Edges may have a direction, a weight, or some special arrangement, and these properties decide which algorithm to use. Figure 5.2 shows the eight most common kinds with small examples.

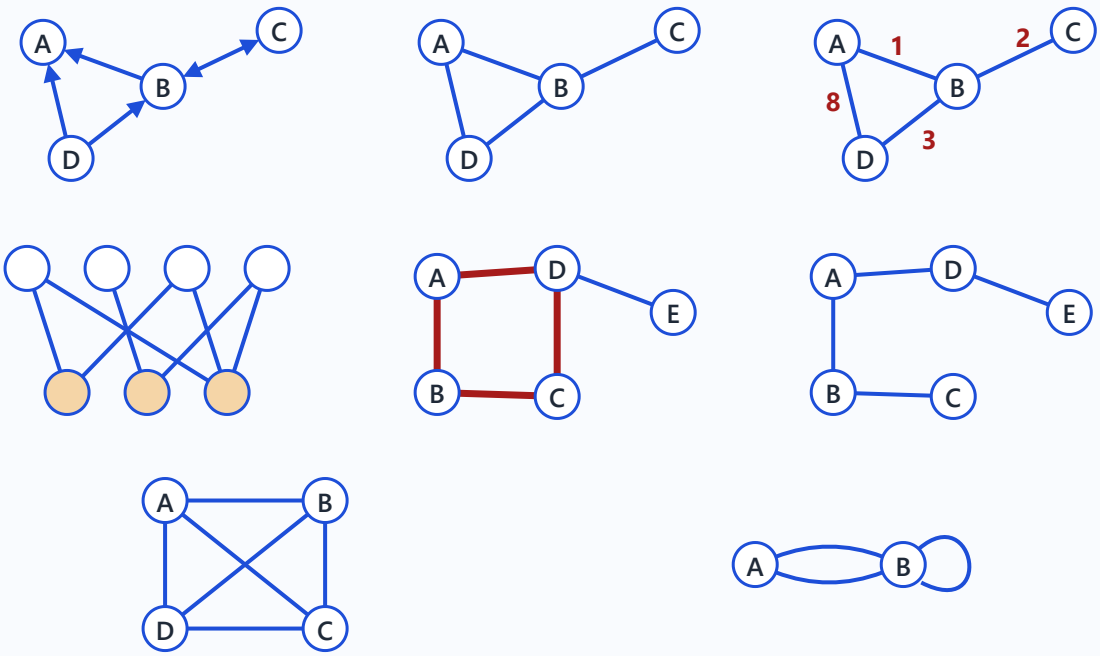


Figure 5.2: Kinds of graphs, left to right and top to bottom: directed, undirected, weighted, bipartite, cyclic, tree, complete, and non-simple. In the bipartite graph the four upper vertices form one part and the three lower vertices the other.

- **Directed graph.** Every edge has a start and an end, and can be traversed only in the direction of the arrow. The edge $u \to v$ lets us go from u to v but not back. One-way streets, links from one web page to another, and dependencies of the form "task A must finish before task B" are directed graphs. In a directed graph the number of edges entering a vertex is its **in-degree**, and the number leaving it is its **out-degree**.
- **Undirected graph.** Edges can be traversed in both directions. Friendship is symmetric, so a social network is modeled by an undirected graph. An undirected edge can be thought of as two directed edges, and it is usually stored that way in memory.
- **Weighted graph.** Every edge carries a number. The number may be a distance, a time, a cost, or a capacity. In a road network the weights are kilometers; in a computer network, latencies. In a weighted graph the length of a path is measured not by the number of edges but by the sum of the weights.
- **Bipartite graph.** The vertices can be split into two sets so that every edge joins vertices in different sets; no edge joins two vertices of the same set. Matchings between students and courses, or between workers and jobs, have this structure.
- **Cyclic graph.** It contains at least one cycle. In the example in the figure, A–D–C–B–A is a path that returns to where it started. A graph with no cycle is **acyclic**. Directed acyclic graphs (DAGs) have a special place in task scheduling.
- **Tree.** A connected, acyclic, undirected graph. In a tree there is exactly one path between any two vertices, and a tree with V vertices always has $V - 1$ edges. Binary search trees, heaps, and file-system directories are familiar examples.
- **Complete graph.** Every pair of vertices is joined by an edge. A complete graph on n vertices has $n(n-1)/2$ edges. Each vertex is joined to the other $n - 1$ vertices; every edge is counted twice this way, so the total is halved. On four vertices this gives $4 \cdot 3/2 = 6$ edges, on ten vertices $10 \cdot 9/2 = 45$. A simple undirected graph cannot have more edges than this.
- **Non-simple graph.** It contains several edges between the same two vertices (**multiple edges**) or an edge joining a vertex to itself (a **self-loop**). A graph with neither is a **simple graph**. Unless stated otherwise, the graphs in these notes are simple; the only exception is the self-loop in the strongly connected components example.

These kinds are not mutually exclusive. A road map can be directed (one-way streets), weighted (lengths), and cyclic all at once. A tree is always bipartite: put the vertices at even distance from the root in one set and those at odd distance in the other, and every edge joins different sets. This observation is also the basis of the bipartiteness test.

5.1.3 Families of graph algorithms

Algorithms on graphs fall into four large groups according to the question they answer:

1. **Traversal.** Starting from one vertex, visit all reachable vertices in an orderly sequence. Depth-first search follows one branch to its end; breadth-first search sweeps the network layer by layer. This group is the subject of these notes.

2. **Shortest paths.** Find the shortest paths from one start vertex to the others. In a weighted graph Dijkstra's algorithm does this.
3. **Minimum spanning tree** (MST). Choose the edges that connect all vertices at the least total cost. Kruskal's algorithm is a well-known example.
4. **Network flow.** In a network whose edges have capacities, find the largest amount of flow that can be sent from a source to a sink.

Traversal is the most basic of these groups, because most of the other algorithms contain a traversal inside. Dijkstra's algorithm resembles a breadth-first search whose queue is replaced by a priority queue. Flow algorithms look for an augmenting path in every round by a traversal. Table 5.1 gives an everyday counterpart for three of the algorithms.

Algorithm	Everyday example
DFS	Searching a maze: follow one corridor to its end, then turn back
BFS	A piece of news spreading through a city from person to person: first the immediate circle, then their circles
Dijkstra	A courier finding the route that completes the deliveries in the shortest time

Table 5.1: Everyday counterparts of graph algorithms.

5.1.4 Storing a graph in memory

The first decision for a program that works with a graph is how to store it, and this decision directly affects the speed of the algorithm. There are two common representations: the adjacency list and the adjacency matrix. In both, the vertices are numbered $0, 1, \ldots, V - 1$, so a vertex can serve as an array index.

Definition 5.2

Adjacency list. For each vertex, a list of its neighbors. The V lists are kept in an array; `adj[u]` holds the neighbors of u.

Adjacency matrix. A table of size $V \times V$. If there is an edge between u and v, then `M[u][v] = 1`; otherwise `M[u][v] = 0`. In a weighted graph the weight of the edge is stored instead of 1.

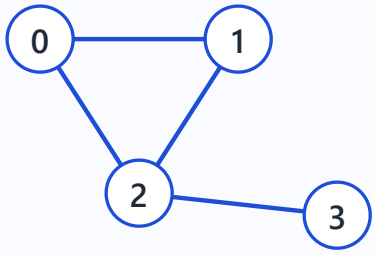


Figure 5.3: An undirected example graph on four vertices: $V = 4$, $E = 4$. The edges are 0–1, 0–2, 1–2, and 2–3.

The adjacency list of the graph in Figure 5.3 is:

```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```

In an undirected graph every edge appears in two lists. The edge 0–1 is written as 1 in the list of 0 and as 0 in the list of 1. The total length of the lists is therefore $2E = 8$. In a directed graph the edge $u \to v$ is added only to the list of u, and the total length is E. The adjacency matrix of the same graph is given in Table 5.2.

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Table 5.2: The adjacency matrix of the graph in the figure.

The matrix of an undirected graph is symmetric about the diagonal: `M[u][v] = M[v][u]`. In a simple graph there are no self-loops, so the diagonal is zero. The number of 1s in a row equals the degree of that vertex; row 2 has three 1s, and vertex 2 has degree 3.

In Java the adjacency list is built as an array of lists. The fragment below builds the lists of an undirected graph from an array `edges` that gives the edges as pairs `{u, v}`:

```
List<Integer>[] adj = new List[n];
for (int v = 0; v < n; v++)
  adj[v] = new ArrayList<>();
for (int[] e : edges) {
  adj[e[0]].add(e[1]);
  adj[e[1]].add(e[0]);  /* undirected: add reverse too */
}
```

The first loop creates an empty list for every vertex: $O(V)$. The second loop makes two insertions per edge: $O(E)$. Construction takes $O(V + E)$ time. If the edges are given in the order `(0, 1), (0, 2), (1, 2), (2, 3)`, the result is exactly the lists above. For a directed graph the second `add` line is left out. All Java code in these notes uses an array `List<Integer>[] adj` of this form.

The costs of the two representations are compared in Table 5.3.

Operation	Adjacency list	Adjacency matrix
Memory	$O(V + E)$	$O(V^2)$
All neighbors of u	$O(\deg(u))$	$O(V)$
Is (u, v) an edge?	$O(\deg(u))$	$O(1)$
Traversal (DFS or BFS)	$O(V + E)$	$O(V^2)$

Table 5.3: Costs of the adjacency list and the adjacency matrix.

Each row of the table has a reason. The list stores one list head per vertex and one or two items per edge, so its memory is proportional to $V + E$. The matrix allocates V^2 cells regardless of the number of edges. Visiting the neighbors of a vertex means reading only that vertex's list in the list representation. In the matrix one has to look at all V cells of the row, because it is not known in advance which cells hold a 1. Asking whether a particular edge exists is a single cell read in the matrix; in the list, the list of u must be scanned from the beginning.

Example 5.1

Compare the approximate work of BFS under the two representations on an undirected graph with $V = 1000$ vertices and $E = 3000$ edges.

1. **Adjacency list.** BFS takes each vertex off the queue once and reads each vertex's list once. The lists have total length $2E = 6000$. The work is about $V + 2E = 1000 + 6000 = 7000$ basic steps.
2. **Adjacency matrix.** For each vertex one row of the matrix is scanned from start to end. The work is $V \cdot V = 1{,}000{,}000$ cell reads.
3. **Ratio.** $1{,}000{,}000 / 7000 \approx 143$. On this graph the matrix does about 140 times as much work. Most cells are 0, and reading these zeros is wasted effort.

A graph whose number of edges is far below the maximum possible value $V(V-1)/2$ is called **sparse**; one close to it is **dense**. Most real networks are sparse. In a road network the number of roads meeting at an intersection is limited to a few; a network with a million intersections may have about three million roads. The adjacency list stores this network in a few million items. The matrix would need 10^{12} cells, which is 125 gigabytes even at one bit per cell. This is why the adjacency list is the default representation for traversal algorithms. The matrix is preferred when the graph is small and dense, or when the question "are these two vertices adjacent?" is asked very often.

Common mistake

Adding an undirected edge to only one list. If the edge 0–1 is written only to `adj[0]`, the graph silently becomes a directed graph with the edge $0 \to 1$. A traversal starting at 1 can no longer reach 0, and the number of connected components comes out wrong. In an undirected graph the lists must have total length $2E$; this is an easy check on the construction.

5.2 Graph traversal

5.2.1 The traversal problem

Definition 5.3

Graph traversal means visiting every vertex reachable from a start vertex in an orderly sequence, **exactly once**. A visit includes the work done at the vertex: printing it, incrementing a counter, writing its distance into an array.

The purpose of a traversal is to explore the structure of the graph systematically. Questions such as which vertices are reachable, how many components there are, and whether the graph has a cycle can be answered by a traversal. The "exactly once" condition says two things. First, no reachable vertex may be skipped. Second, no vertex may be processed twice.

The second condition holds automatically in trees. When a tree is traversed from the root, each vertex is reached by a single path, because a tree has no cycles. In a general graph a vertex can be reached by many paths, and cycles may exist. In the graph of Figure 5.3 we go from 0 to 1, from 1 to 2, and from 2 back to 0. Unless the algorithm remembers which vertices it has already seen, it will circle forever around the cycle 0–1–2. For this reason every traversal keeps a **visited array**: `visited[v]` is a `boolean` array that tells whether v has already been marked. We move to a vertex only if it is not marked.

A traversal sees only the vertices *reachable* from the start. In a directed graph with no path from 0 to 5, a traversal from 0 never visits 5. To traverse all vertices of a graph that is not connected, the traversal is restarted from every vertex that has not been visited yet. In an undirected graph each restart discovers a new connected component.

5.2.2 Two approaches: stack and queue

At every moment during a traversal the algorithm holds a "waiting list" of vertices that have been discovered but not yet processed. The next vertex is chosen from this list, and how it is chosen determines the character of the traversal. The only difference between the two basic approaches is the data structure that holds the waiting list.

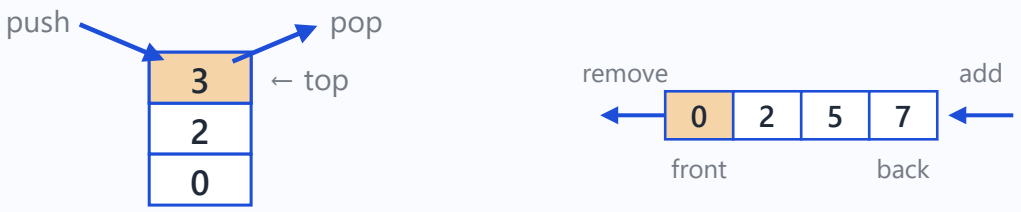


Figure 5.4: Left, a stack: insertion and removal both happen at the same end, the top (last in, first out). Right, a queue: items are added at the back and removed from the front (first in, first out).

- **Depth-first search** (DFS) keeps the waiting list in a **stack**. A stack works by the last in, first out (LIFO) rule. The most recently discovered vertex is processed first, so the algorithm goes from a vertex to its neighbor, then to that neighbor's neighbor, as deep as it can. When there is nowhere left to go, it returns to the previous vertex. This return is called **backtracking**.
- **Breadth-first search** (BFS) keeps the waiting list in a **queue**. A queue works by the first in, first out (FIFO) rule. The vertex discovered first is processed first, so the algorithm visits first the vertices one edge away from the start, then those two edges away. The traversal proceeds level by level.

The stack and the queue are two basic data structures; in both, insertion and removal take *O(1)* time. In Java, `push` and `pop` of `ArrayDeque` serve as a stack, and `add` and `poll` of the `Queue` interface serve as a queue. In a recursive DFS the stack is not visible. The call stack takes its place: each recursive call pushes a frame and each return pops one.

5.3 Depth-first search (DFS)

Depth-first search is perhaps the most widely used building block of graph algorithms. Cycle detection, topological sorting, strongly connected components, and bridge finding are all obtained by adding a few lines to a DFS. This section presents DFS first through an analogy, then on a tree, and then with two Java versions, recursive and stack-based.

5.3.1 The idea: go deep first

DFS behaves like a traveler in a maze. The traveler enters a corridor and goes on until a dead end. At the dead end, they return to the last junction they passed and turn into a corridor they have not tried. When that one ends, they go back to the junction before. This continues until every corridor of the maze has been tried. Two things make the traveler's job easier: they mark the places they have passed with chalk, so as never to walk the same corridor twice, and they remember the junctions to return to, in order.

Definition 5.4

Depth-first search (DFS), after visiting a vertex, moves to its first unvisited neighbor and continues the same way from there. When a vertex has no unvisited neighbor left, DFS returns to the vertex from which it came and tries the next neighbor there.

The chalk marks in the analogy are the array `visited[]`. The junctions to return to are remembered by a stack. In the recursive version this stack is the program's call stack: when the work on a vertex is finished, its call returns and control passes to the previous vertex automatically. The iterative version uses an explicit `Deque` object.

During DFS a vertex is in one of three states. The states are shown by colors, and the figures in these notes use the same colors:

- **White:** the vertex has not been discovered yet.
- **Yellow:** the vertex has been discovered but its work is still going on. Its call sits on the stack; some of its neighbors have not been examined yet.
- **Dark:** the work on the vertex is finished; all its neighbors have been examined and its call has returned.

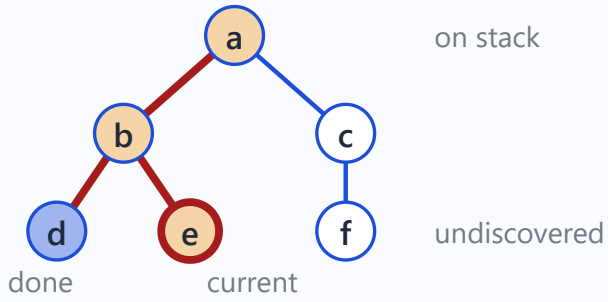


Figure 5.5: Vertex states during DFS. a and b are on the stack (yellow), d is finished (dark), e is being processed (thick outline), and c and f have not been discovered (white). The thick edges are the edges by which visited vertices were first reached.

The edges used to reach a vertex for the first time form a tree, called the **DFS tree**. Its root is the start vertex, and the parent of each vertex is the vertex that discovered it. In Figure 5.5 the edges a–b, b–d, and b–e belong to the DFS tree. The other edges of the graph do not enter the tree, but they are still examined and carry important information. For example, an edge to a yellow vertex signals a cycle.

The visit order of DFS depends on the order in which neighbors are tried. If the neighbor lists of the same graph are ordered differently, a different but equally valid DFS order results. Unless stated otherwise, these notes try neighbors in the order of the adjacency list. In undirected examples the lists are in increasing or alphabetical order.

5.3.2 Visit order on a tree

The clearest example of DFS's "go deep, then come back" pattern is a tree, because a tree has no cycles and no vertex can be reached in two ways. Figure 5.6 shows three moments of a DFS that starts at the root of a tree with ten vertices. The numbers above the vertices give the order in which DFS visits them.

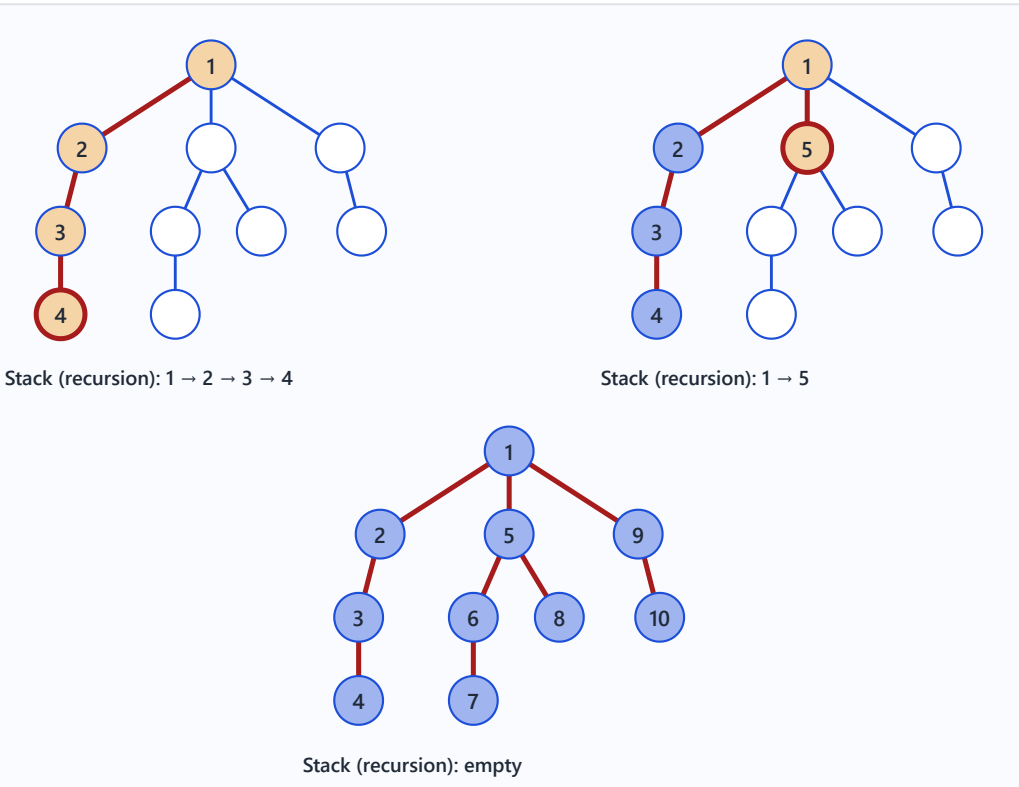


Figure 5.6: DFS on a tree with ten vertices. Left: the leftmost branch has been followed from the root down to 4; the stack holds 1, 2, 3, 4. Middle: 4, 3, and 2 have no undiscovered neighbors left, so DFS has backed up and visited 5, the next child of 1. Right: the traversal is finished and the stack is empty.

Example 5.2

The edges of the tree are 1–2, 1–5, 1–9, 2–3, 3–4, 5–6, 5–8, 6–7, and 9–10. Children are tried from left to right.

1. The root 1 is visited. Stack: 1.
2. The first undiscovered neighbor of 1 is 2; DFS descends to 2. Stack: 1 → 2.
3. From 2 to 3, from 3 to 4. Stack: 1 → 2 → 3 → 4. The end of the leftmost branch has been reached.
4. Vertex 4 has no undiscovered neighbor; 4 finishes and DFS returns to 3. Neither 3 nor 2 has another undiscovered neighbor, so both finish. This chain of returns continues up to 1. Stack: 1.
5. The next child of 1, namely 5, is visited. From 5 to 6, from 6 to 7. Stack: 1 → 5 → 6 → 7.
6. 7 and 6 finish; DFS returns to 5. The next child of 5, namely 8, is visited. Stack: 1 → 5 → 8.
7. 8 and 5 finish; DFS returns to 1. The last child of 1, namely 9, is visited, then the child 10 of 9. Stack: 1 → 9 → 10.
8. 10, 9, and 1 finish; the stack empties.

The visit order is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Because the vertices in this example are already numbered in DFS order, the order is simply the increasing order of the numbers. On the same tree BFS would give 1, 2, 5, 9, 3, 6, 8, 10, 4, 7, with all children of the root coming first.

The example shows that the stack holds at most four vertices. This number is the count of vertices on the path from the root to the deepest leaf. The DFS stack never goes deeper than the longest path, because at every moment it holds the tree path from the root to the current vertex.

5.3.3 Recursive pseudocode

The definition of DFS uses DFS itself: to run DFS from a vertex means to mark it and run DFS from each of its undiscovered neighbors. The algorithm is therefore most naturally written recursively. Two forms are common in the literature. The first is the one often seen in English-language sources:

```
procedure DFS(G, v) is
  label v as discovered
  for all directed edges from v to w
      that are in G.adjacentEdges(v) do
    if vertex w is not labeled
        as discovered then
      call DFS(G, w)
```

The second moves the check to the beginning of the call:

```
DFS(v, visited):
  if v is not in visited:
    add v to visited
    process v (for example, print it)
    for each neighbor w of v:
      DFS(w, visited)
```

The two versions produce the same visit order. The difference is where the "already visited?" check is made. The first version checks a neighbor *before calling*; for a marked neighbor no call is made at all. The second version calls for every neighbor and checks at the start of the call; a call opened for a marked vertex returns immediately. The second version looks shorter but makes needless calls. A call is opened for every edge, so the number of calls is $O(E)$. The order of growth does not change, but the constant factor grows. The Java code follows the first version.

The pseudocode visits only the vertices reachable from a single start vertex. To traverse all vertices of a graph that is not connected, DFS is restarted from every unvisited vertex. In an undirected graph each start traverses one connected component completely.

5.3.4 Code: dfs()

```
void dfs(int v, boolean[] visited) {
  visited[v] = true;
  System.out.print(v + " ");
  for (int w : adj[v])
    if (!visited[w])
      dfs(w, visited);
}
```

The code is the Java counterpart of the first pseudocode and works as follows, line by line:

- `visited[v] = true;` marks the vertex. The mark is set as soon as the vertex is visited. This way v is not called a second time even if one of its neighbors has an edge leading back to v.
- `System.out.print(v + " ");` does the work at the vertex; here the work is printing it. Because this line is at the start of the call, vertices are printed in **discovery order**. Had the line been placed after the loop, vertices would be printed in **finishing order**; topological sorting uses this second order.
- `for (int w : adj[v])` walks through the adjacency list of v from beginning to end.
- `if (!visited[w]) dfs(w, visited);` descends into an undiscovered neighbor. When the recursive call returns, every vertex reachable from w has been visited and the loop continues with the next neighbor. Backtracking is exactly this return of the call.

The method takes the `visited` array as a parameter. The array is created once and all calls share it. This makes it easy to restart DFS on a graph that is not connected: the same array carries the marks of the first start over to the second.

5.3.5 Code trace: dfs(0)

The graph used in the trace is directed. Its edges are *0→1*, *0→2*, *0→7*, *1→2*, *2→0*, *2→3*, *2→4*, *3→5*, *4→6*, and *5→6*. The adjacency lists are filled in this order: `adj[0] = [1, 2, 7]`, `adj[2] = [0, 3, 4]`. Vertices 6 and 7 have no outgoing edges. The graph has the cycles *0 → 1 → 2 → 0* and *0 → 2 → 0*; without a visited array DFS would circle in them forever.

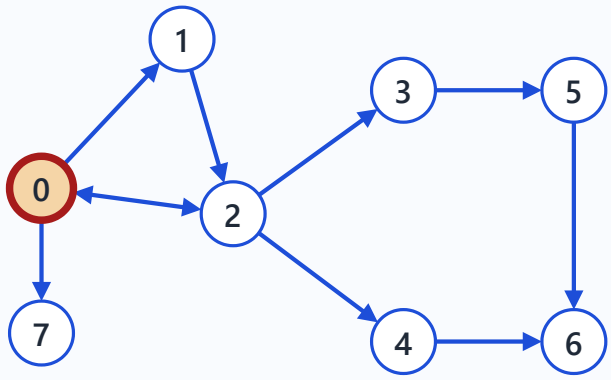


Figure 5.7: The start of the trace: `dfs(0)` has been called, and 0 has been marked and printed. The double-headed line between 0 and 2 stands for two separate edges: *0→2* and *2→0*.

Step	Event	Call stack	Output
1	`dfs(0)` : 0 is marked and printed	0	0
2	neighbor 1 of 0 is unmarked → `dfs(1)`	0, 1	0 1
3	neighbor 2 of 1 is unmarked → `dfs(2)`	0, 1, 2	0 1 2
4	neighbor 0 of 2 is marked → skipped	0, 1, 2	0 1 2
5	neighbor 3 of 2 is unmarked → `dfs(3)`	0, 1, 2, 3	0 1 2 3
6	neighbor 5 of 3 is unmarked → `dfs(5)`	0, 1, 2, 3, 5	0 1 2 3 5
7	neighbor 6 of 5 is unmarked → `dfs(6)`	0, 1, 2, 3, 5, 6	0 1 2 3 5 6
8	6, 5, and 3 finish; control returns to 2	0, 1, 2	0 1 2 3 5 6
9	neighbor 4 of 2 is unmarked → `dfs(4)`	0, 1, 2, 4	0 1 2 3 5 6 4
10	neighbor 6 of 4 is marked → skipped	0, 1, 2, 4	0 1 2 3 5 6 4
11	4, 2, and 1 finish; control returns to 0	0	0 1 2 3 5 6 4
12	neighbor 2 of 0 is marked → skipped	0	0 1 2 3 5 6 4
13	neighbor 7 of 0 is unmarked → `dfs(7)`	0, 7	0 1 2 3 5 6 4 7

Step	Event	Call stack	Output
14	7 and 0 finish; `dfs(0)` returns	empty	0 1 2 3 5 6 4 7

Table 5.4: Trace of `dfs(0)`. The call stack grows from left to right; the rightmost call is the one running now.

Three moments of the trace deserve attention. **In step 4** the edge from 2 to 0 is examined. Vertex 0 is marked and is still on the call stack, that is, yellow. This edge is proof of a cycle: the stack holds a tree path from 0 to 2 (0 → 1 → 2), and the edge *2 → 0* closes it. The code merely skips it here; a version that detects cycles can report "cycle found" at this point.

In step 8 three calls return one after another. Vertex 6 has no outgoing edges at all; the only neighbor of 5 is 6, and the only neighbor of 3 is 5. Control returns to the `for` loop in 2 at the place where it left off, and the next neighbor, 4, is tried. This is what backtracking looks like in code: there is no separate "go back" command; it is enough for the calls to return.

In step 10 the edge from 4 to 6 is examined. Vertex 6 is marked but is not on the stack; its work finished long ago. This edge does not form a cycle: there is no path back from 6 to 4. In a directed graph only an edge to a vertex that is on the stack signals a cycle; an edge to a finished vertex does not.

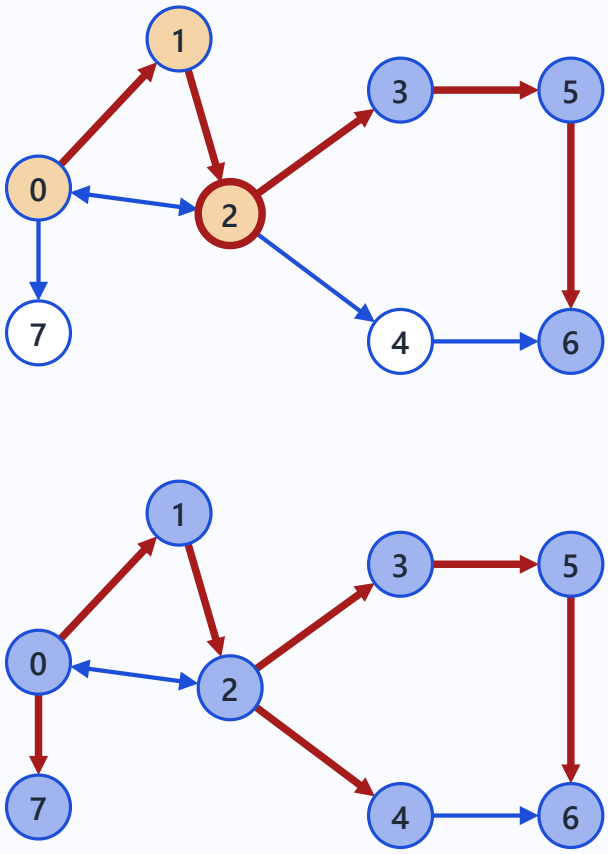


Figure 5.8: Left, step 8: 6, 5, and 3 have finished (dark); control is back at 2 and the stack holds 0, 1, 2. Right, the end of the trace: the thick edges form the DFS tree (*0→1*, *1→2*, *2→3*, *3→5*, *5→6*, *2→4*, *0→7*). The thin edges *0→2*, *2→0*, and *4→6* are not in the tree.

The output is `0 1 2 3 5 6 4 7`. The call stack reached at most six frames (step 7: 0, 1, 2, 3, 5, 6). In the DFS tree the path from 0 to 6 is *0 → 1 → 2 → 3 → 5 → 6*, five edges long. Yet the graph has a path from 0 to 6 of only three edges: *0 → 2 → 4 → 6*. This difference is the first sign that DFS does not find shortest paths.

5.3.6 DFS with a stack

Recursive DFS is short and clear, but it has a weakness: the depth of the call stack equals the length of the DFS path. In a graph containing a very long path this depth can reach tens of thousands. The call stack is a limited region of memory, and when it is exceeded Java throws `StackOverflowError`. In an experiment with the default settings of JDK 21, recursive DFS on a path graph with 100,000 vertices stopped with this error at a call depth of about 10,000. The remedy is to hand the call stack's job to an explicit stack object created by the program. This object lives on the heap, and its size is limited only by total memory. First, the pseudocode common in English-language sources:

```
procedure DFS_iterative(G, v) is
  let S be a stack
  S.push(v)
  while S is not empty do
    v = S.pop()
    if v is not labeled as discovered then
      label v as discovered
      for all edges from v to w
          in G.adjacentEdges(v) do
        S.push(w)
```

The same algorithm in a more compact form:

```
DFS(start):
  create an empty stack
  push start onto the stack
  loop while the stack is not empty:
    current = pop from the stack
    if current is not in visited:
      add current to visited
      for each neighbor w of current:
        push w onto the stack
```

The Java version:

```
void dfsStack(int s) {
  boolean[] visited = new boolean[n];
  Deque<Integer> stack = new ArrayDeque<>();
  stack.push(s);
  while (!stack.isEmpty()) {
    int v = stack.pop();
    if (!visited[v]) {
      visited[v] = true;
      System.out.print(v + " ");
      for (int w : adj[v])
        stack.push(w);
    }
  }
}
```

The code differs from the recursive version in three respects:

- **The waiting list is explicit.** `ArrayDeque` is used as a stack: `push` adds at the top, `pop` removes from the top. The loop runs until the stack is empty.

- **Marking happens at pop time.** A vertex is marked when it is popped, not when it is pushed. A vertex that has not been popped yet can therefore be pushed several times, from different neighbors. If it is already marked when popped, nothing is done. The total number of pushes is $1 + \Sigma\, deg$, which is $1 + E$ in a directed graph and $1 + 2E$ in an undirected one. The stack can grow to size $O(E)$.
- **The order reverses.** Neighbors are pushed in list order, but the last one pushed comes off first. After 0, whose list is `[1, 2, 7]`, the first vertex popped is 7, not 1. The visit order therefore differs from that of the recursive version.

A small example shows why marking is done at pop time. Let the neighbors of 0 be 1 and 2, and let 2 have the neighbor 1. If vertices were marked when pushed, 1 and 2 would be marked together while 0 was processed. When 2 was popped, its neighbor 1 would already be marked and would not enter the stack, and 1 would be processed as the second child of 0, not from below 2. The traversal would still visit each vertex once, but the branch starting at 2 would not have been followed to its end, and the order would no longer be depth-first. Marking at pop time ensures that a vertex is processed from the place where it was discovered last. That is the "deep first" rule of DFS.

5.3.7 Code trace: dfsStack(0)

Let us trace the call `dfsStack(0)` on the same directed graph. In the table the stack is written from left to right, bottom to top; the rightmost item is the top.

Popped	Action	Stack after	Output
—	start: 0 is pushed	0	
0	mark, print; push 1, 2, 7	1, 2, 7	0
7	mark, print; no neighbors	1, 2	0 7
2	mark, print; push 0, 3, 4	1, 0, 3, 4	0 7 2
4	mark, print; push 6	1, 0, 3, 6	0 7 2 4
6	mark, print; no neighbors	1, 0, 3	0 7 2 4 6
3	mark, print; push 5	1, 0, 5	0 7 2 4 6 3
5	mark, print; push 6	1, 0, 6	0 7 2 4 6 3 5
6	already marked, skipped	1, 0	0 7 2 4 6 3 5
0	already marked, skipped	1	0 7 2 4 6 3 5
1	mark, print; push 2	2	0 7 2 4 6 3 5 1
2	already marked, skipped	empty	0 7 2 4 6 3 5 1

Table 5.5: Trace of `dfsStack(0)`. Each row is one `pop` operation.

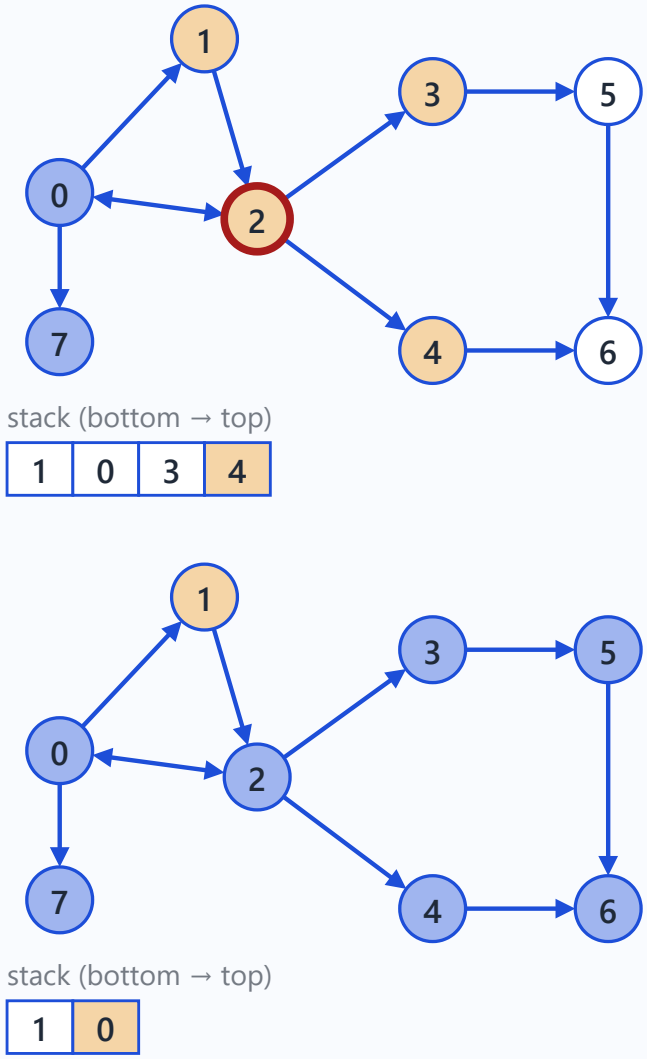


Figure 5.9: Left: 2 has been popped and its neighbors 0, 3, 4 pushed; 0 entered the stack although it is already marked. Right: 6 is popped a second time (it had been pushed first from 4, then from 5); since it is marked, it is skipped. The stack holds 1 and 0.

The output is `0 7 2 4 6 3 5 1`. The recursive version printed `0 1 2 3 5 6 4 7` on the same graph. The two orders differ, but both are valid DFS orders: the stack version gives the order that recursive DFS would give on a graph whose neighbor lists were reversed. In all, 11 pushes were made: the initial 0 and one for each of the ten edges (*1 + E = 11*). Vertices 6, 0, and 2 entered the stack twice each. The stack held at most four items.

If the same order as the recursive version is wanted, the neighbors are pushed in *reverse order*. If the neighbors of 0 are pushed in the order 7, 2, 1, then 1 ends up on top and is popped first. The two versions then produce the same visit order.

Example 5.3

An undirected graph has the edges A–B, A–C, B–D, C–D, C–E, D–F, E–F, and F–G; the adjacency lists are in alphabetical order. Find the visit orders of recursive DFS and of stack-based DFS from A.

Recursive DFS. From A, descend to its first neighbor B. The list of B is `[A, D]`; A is marked, so descend to D. The list of D is `[B, C, F]`; B is marked, so descend to C. The list of C is `[A, D, E]`; A and D are marked, so descend to E. From E go to F (C is marked), from F to G (D and E are marked). The only neighbor of G, F, is marked. After that the calls return one by one and no new vertex appears. The order is **A B D C E F G**; the call stack reaches at most seven frames.

Stack-based DFS. Neighbors are pushed in alphabetical order:

1. A is popped and printed; B and C are pushed. Stack: B, C.
2. C is popped and printed; A, D, E are pushed. Stack: B, A, D, E.
3. E is popped and printed; C, F are pushed. Stack: B, A, D, C, F.
4. F is popped and printed; D, E, G are pushed. Stack: B, A, D, C, D, E, G.
5. G is popped and printed; F is pushed. Stack: B, A, D, C, D, E, F (seven items).
6. F and E are marked and skipped. D is popped and printed; B, C, F are pushed.
7. F and C are skipped. B is popped and printed; A and D are pushed.
8. The remaining items D, A, C, D, A, B are all marked; they are skipped one by one and the stack empties.

The order is **A C E F G D B**. Each vertex is processed once and pushes all its neighbors as it is processed. The adjacency lists have total length $2E = 16$; together with the initial push, **17 pushes** are made.

5.3.8 Complexity

Theorem 5.1 (Running time of DFS)

On a graph stored as adjacency lists, DFS runs in $O(V + E)$ time. On a graph stored as an adjacency matrix the time is $O(V^2)$.

Proof. The cost splits into two parts. The first is the vertices. `dfs` is called at most once per vertex, because a call is made only for an unmarked vertex and the first thing a call does is mark its vertex. The work of each call outside the loop is constant, so the vertices contribute $O(V)$. The second is the edges. Each call walks the adjacency list of its own vertex once and does constant work per neighbor. Since the list of each vertex is walked exactly once, the work equals the total length of all the lists.

This length is E in a directed graph and $2E$ in an undirected one, so the edges contribute $O(E)$. The total is $O(V + E)$. In a matrix, finding the neighbors of a vertex means scanning a row of V cells regardless of its degree. For V vertices, V rows are scanned, giving $O(V^2)$. ■

$O(V + E)$ is the time needed just to read the graph. Nothing can be said about the whole graph without looking at every vertex and every edge at least once. This bound is therefore the best possible for traversal, and algorithms of this kind are called **linear-time** graph algorithms.

Extra memory also has two parts. The array `visited[]` takes $O(V)$ space. The size of the stack depends on the version. In the recursive version the call stack goes as deep as the longest path in the DFS tree, which is V in the worst case, so $O(V)$. On a path graph (0–1–2–…), for example, the stack holds all vertices. In the stack-based version that marks at pop time, a vertex can be pushed several times, so the stack can grow to $O(E)$. In a dense graph E can be of order V^2, so this difference is not negligible. A more elaborate version that avoids the duplicates on the stack can also be written. In it, each stack item holds, together with the vertex, the index of the next neighbor to try. This imitates recursive DFS exactly, and $O(V)$ extra memory suffices.

5.3.9 What DFS is good for, and what it cannot do

The power of DFS comes from the structural information it collects during the traversal. When a vertex was discovered, when it finished, and whether an edge leads to a yellow or a dark vertex are enough to answer many questions:

- **Reachability and connected components.** After a DFS from s, t is reachable if `visited[t]` is true. The number of times DFS is started gives the number of connected components of an undirected graph.
- **Cycle detection.** In a directed graph, an edge to a vertex on the call stack (a yellow vertex) is called a **back edge**. A back edge, together with the tree path from its target down to its source, forms a cycle. Conversely, if the graph has a cycle, all the other vertices of the cycle are discovered before the first-discovered vertex of the cycle finishes, and the cycle's edge back to that vertex is a back edge. A directed graph therefore has a cycle exactly when DFS finds a back edge.
- **Topological sorting.** In a directed acyclic graph, reversing the finishing order of the vertices gives a topological order.
- **Strongly connected components.** The algorithms of Tarjan and Kosaraju are built on DFS.
- **Bridges and articulation points, puzzle solving.** Bridges are found with Tarjan's low-link idea. Backtracking algorithms that solve puzzles such as Sudoku and the eight queens problem are also DFS on an implicit graph.

What DFS cannot do is just as important: **DFS does not find shortest paths.** The path DFS follows when it first reaches a target can be a long detour. In the directed example DFS reached 6 by the path $0 \to 1 \to 2 \to 3 \to 5 \to 6$, in five edges; the shortest path had three. In the undirected graph of Example 5.3 the DFS tree follows the path A–B–D–C–E–F–G from A to G, six edges; the shortest path is A–B–D–F–G, four edges. Finding the path with the fewest edges requires breadth-first search.

5.4 Breadth-first search (BFS)

While DFS follows one branch to its end, BFS advances all branches together, step by step. The difference looks small, but it has an important consequence: in an unweighted graph, BFS finds the path with the fewest edges from the start to every vertex. The route with the fewest transfers in navigation, the length of the chain of acquaintance between two people in a social network, and the solution in the fewest moves in a game are all computed this way.

5.4.1 The idea: layer by layer

The ripples made by a stone dropped into still water describe BFS well. The wave first reaches the immediate surroundings of the stone, then one ring further out, then another. News spreads the same way: the person who receives it tells all their friends, and they tell their own friends. The first to hear are those one step away from the start, then those two steps away.

Definition 5.5

Breadth-first search (BFS) visits the vertices reachable from the start vertex s in order of their distance from s: first s, then the vertices one edge away, then those two edges away. Discovered vertices wait in a queue; the first-in, first-out rule of the queue preserves this order.

Definition 5.6

The **levels** of BFS are defined as follows: $L_0 = \{s\}$; L_1 is the set of neighbors of s; in general L_{k+1} is the set of neighbors of vertices in L_k that are in no level yet. The distance of every vertex in L_k from s, that is, the number of edges of a shortest path from s to it, is exactly k.

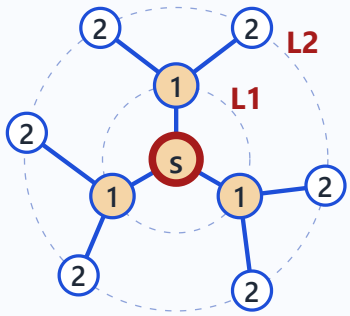


Figure 5.10: BFS levels. s is at the center; vertices at distance 1 are on the inner ring (L_1) and those at distance 2 on the outer ring (L_2).

BFS has two rules. First, the queue preserves order: the vertex discovered first is processed first. All vertices in L_1 enter the queue before those in L_2, so they leave first. Second, a vertex is marked **when it enters** the queue. This way a vertex enters the queue at most once, even if several neighbors see it. This rule is the opposite of the "mark at pop time" rule of stack-based DFS; its purpose is to keep the queue from growing needlessly.

5.4.2 An example with levels

Take an undirected graph on eight vertices: vertex 1 is joined to 2, 3, 4, and 5; vertex 2 to 6 and 7; vertex 3 to 8. BFS starts at 1. Table 5.6 shows, at each step, the vertex taken from the queue, the vertices added, and the state of the queue.

Step	Taken	Newly discovered	Queue	Level
0	—	1	1	$1 \to L_0$
1	1	2, 3, 4, 5	2, 3, 4, 5	$2, 3, 4, 5 \to L_1$
2	2	6, 7	3, 4, 5, 6, 7	$6, 7 \to L_2$
3	3	8	4, 5, 6, 7, 8	$8 \to L_2$
4	4	—	5, 6, 7, 8	
5	5	—	6, 7, 8	
6	6	—	7, 8	
7	7	—	8	
8	8	—	empty	

Table 5.6: BFS from 1. The queue is emptied from the left and filled from the right.

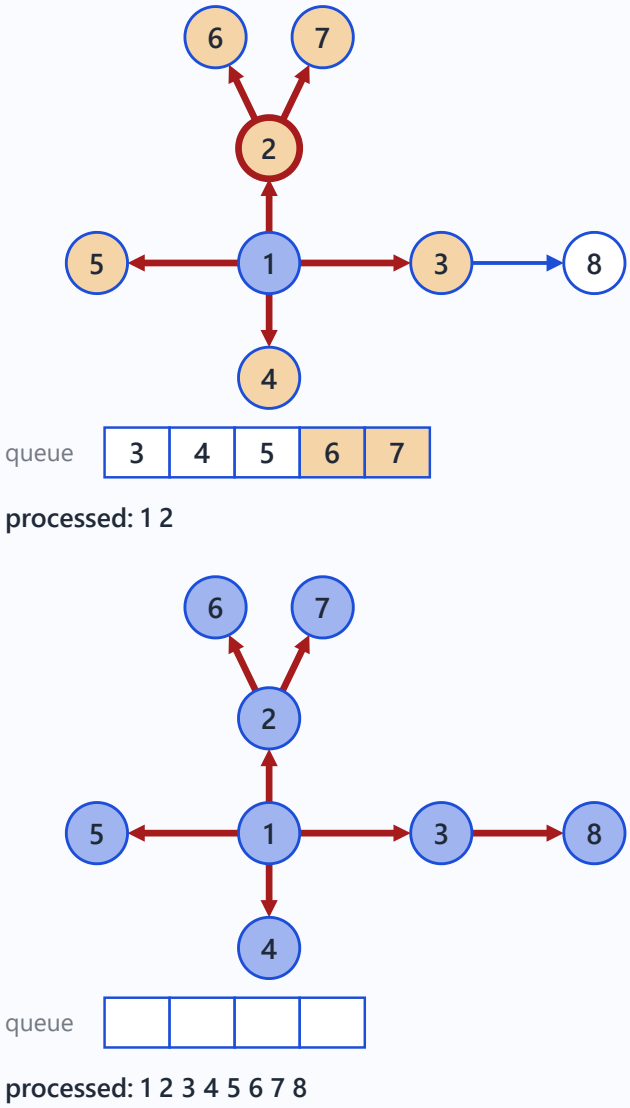


Figure 5.11: Left, the end of step 2: 1 and 2 have been processed (dark); 3, 4, 5, 6, 7 are waiting in the queue (yellow). Right, the end of the traversal: the processing order is 1, 2, 3, 4, 5, 6, 7, 8; the thick arrows show the edge that discovered each vertex.

The visit order is 1, 2, 3, 4, 5, 6, 7, 8. The vertices at distance 1 (2, 3, 4, 5) come first, then those at distance 2 (6, 7, 8). Vertices 6 and 7 were discovered before 3 was processed, but because they were added at the end of the queue they are processed after 3, 4, and 5. The next level is not started before the whole of one level is done; this is what "layer by layer" means for BFS. In this example the graph is a tree, and the edges that discover each vertex are all the edges of the graph.

5.4.3 Pseudocode

```
procedure BFS(G, root) is
  let Q be a queue
  label root as explored
  Q.enqueue(root)
  while Q is not empty do
    v := Q.dequeue()
    for all edges from v to w
        in G.adjacentEdges(v) do
      if w is not labeled as explored then
        label w as explored
        w.parent := v
        Q.enqueue(w)
```

A more compact form:

```
BFS(start):
  create an empty queue
  mark start as explored
  put start in the queue
  loop while the queue is not empty:
    current = take from the queue
    for each neighbor w of current:
      if w is not explored:
        mark w as explored
        parent of w = current
        put w in the queue
```

Two details of the pseudocode matter. The first is the line `w.parent := v`: every vertex records the vertex that discovered it as its parent. These parent links form a tree, called the **BFS tree**. Following parents from the target back to the root yields, in reverse, the shortest path from the start to the target. The second is that marking is done when a vertex is *added* to the queue, not when it is removed. Otherwise a vertex would be added again by every neighbor that sees it during the time before it leaves the queue.

5.4.4 Code: bfs()

```
void bfs(int s) {
  boolean[] visited = new boolean[n];
  Queue<Integer> queue = new LinkedList<>();
  visited[s] = true; queue.add(s);
  while (!queue.isEmpty()) {
    int v = queue.poll();
    System.out.print(v + " ");
    for (int w : adj[v])
      if (!visited[w]) {
        visited[w] = true;
        queue.add(w);
      }
  }
}
```

- `boolean[] visited = new boolean[n];` starts with all vertices unmarked. Java arrays are created filled with `false`.
- `Queue<Integer> queue = new LinkedList<>();` creates the queue. `LinkedList` implements the `Queue` interface; `add` appends at the end and `poll` removes from the front, both in *O(1)*. `ArrayDeque` does the same job.
- `visited[s] = true; queue.add(s);` marks the start and puts it in the queue. The mark is set at the same moment as the entry into the queue.
- `int v = queue.poll();` removes the vertex at the front of the queue, which is then printed. Because vertices are printed in the order they leave the queue, the output is the BFS visit order.
- The inner loop walks over the neighbors of *v*. An unmarked neighbor is marked and appended to the queue; a marked neighbor is skipped.

The time analysis is the same as for DFS. Each vertex enters the queue at most once and leaves once: *O(V)*. The adjacency list of each vertex is walked once, when the vertex leaves the queue: *O(E)* in total. BFS runs in *O(V + E)* time with adjacency lists and in $O(V^2)$ with a matrix. The extra memory is *O(V)* for `visited[]` and as much as the widest level for the queue, which is *O(V)* in the worst case.

5.4.5 Code trace: bfs(0)

Let us run the code on the directed graph used in the DFS trace: `adj[0] = [1, 2, 7]`, `adj[1] = [2]`, `adj[2] = [0, 3, 4]`, `adj[3] = [5]`, `adj[4] = [6]`, `adj[5] = [6]`; 6 and 7 have no neighbors.

Taken `v`	Neighbor scan	Queue (after)	Output
—	0 is marked and enters the queue	0	
0	1, 2, 7 unmarked → mark, add	1, 2, 7	0

Taken ∨	Neighbor scan	Queue (after)	Output
1	2 marked → skip	2, 7	0 1
2	0 marked → skip; 3, 4 → mark, add	7, 3, 4	0 1 2
7	no neighbors	3, 4	0 1 2 7
3	5 → mark, add	4, 5	0 1 2 7 3
4	6 → mark, add	5, 6	0 1 2 7 3 4
5	6 marked → skip	6	0 1 2 7 3 4 5
6	no neighbors	empty	0 1 2 7 3 4 5 6

Table 5.7: Trace of `bfs(0)`. Each row is one `poll` operation followed by the scan of the neighbors.

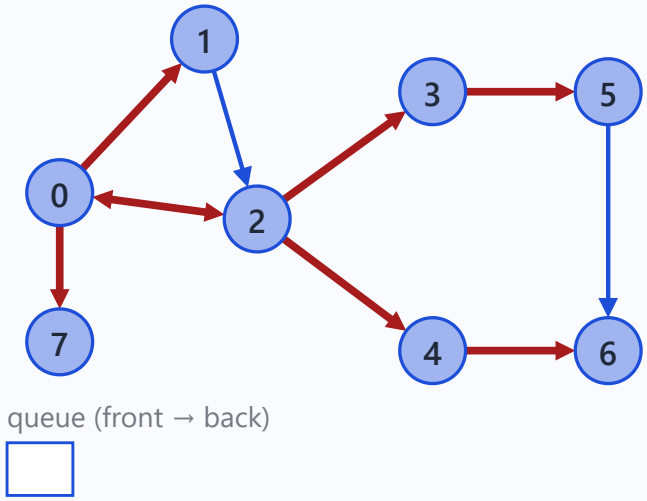


Figure 5.12: When `bfs(0)` finishes, the thick edges form the BFS tree: *0→1, 0→2, 0→7, 2→3, 2→4, 3→5, 4→6*. The tree is shallower and wider than the DFS tree.

The output is `0 1 2 7 3 4 5 6` and the levels are *{0}*, *{1, 2, 7}*, *{3, 4}*, *{5, 6}*. Vertex 2 is also seen as a neighbor of 1, but it was already marked while 0 was processed, so it does not enter the queue a second time. The same holds for 6: 4 marked and added it, and 5 merely sees it. The queue held at most three vertices. The distance of 6 is 3 (*0 → 2 → 4 → 6*); the DFS tree had reached the same vertex in five edges.

Common mistake

Marking a vertex when it is removed from the queue in BFS. In that case an unmarked vertex is added again by every neighbor that sees it while it waits in the queue. The result is often still correct, because a removed vertex is skipped if marked, but the queue grows needlessly. On the complete graph with five vertices, correct BFS from 0 makes 5 insertions and the

queue holds at most 4 vertices. The version that marks on removal makes 11 insertions on the same graph and the queue reaches 7 vertices. In general the number of insertions rises from *O(V)* to *O(E)*. The right way is to mark on insertion.

5.4.6 Distance and parent arrays

To use the shortest-path property of BFS, two arrays are kept in place of the visited array. `dist[v]` holds the least number of edges from *s* to *v*; the parent array (`parent[]` in the code) holds the parent of *v* in the BFS tree, that is, the vertex that discovered *v*. `dist[v] == -1` means "not discovered yet", so a separate `visited[]` array is not needed.

```
void bfsDist(int s) {
  Arrays.fill(dist, -1);
  Arrays.fill(parent, -1); dist[s] = 0;
  Queue<Integer> queue = new LinkedList<>();
  queue.add(s);
  while (!queue.isEmpty()) {
    int v = queue.poll();
    for (int w : adj[v])
      if (dist[w] == -1) {  /* new */
        dist[w] = dist[v] + 1;
        parent[w] = v;
        queue.add(w);
      }
  }
}
```

- `Arrays.fill(dist, -1);` treats all vertices as undiscovered.
- `Arrays.fill(parent, -1); dist[s] = 0;` sets all parents to "none" (*−1*) and sets the distance of the start to 0. The parent of the start also stays *−1*; when a path is built, this value shows that the root has been reached. Filling all of `parent[]` makes unreachable vertices carry *−1* too; we will see below why this is needed when building a path.
- `if (dist[w] == -1)` asks whether the neighbor is new. For a new neighbor, `dist[w] = dist[v] + 1` is written: *w* is in the level after the level of *v*. `parent[w] = v` records that *w* was reached from *v*.

Let us trace the code on the undirected example graph numbered 1 to 8 (1–2, 1–3, 1–4, 1–5, 2–6, 2–7, 3–8). In the table the arrays are written with indices 1–8; "—" is an undiscovered vertex (`dist = -1` , `parent = -1`).

Taken	Queue (after)	1	2	3	4	5	6	7	8
—	1	0/−1	—	—	—	—	—	—	—
1	2, 3, 4, 5	0/−1	1/1	1/1	1/1	1/1	—	—	—

Taken	Queue (after)	1	2	3	4	5	6	7	8
2	3, 4, 5, 6, 7	0/−1	1/1	1/1	1/1	1/1	2/2	2/2	—
3	4, 5, 6, 7, 8	0/−1	1/1	1/1	1/1	1/1	2/2	2/2	2/3
4, 5, 6, 7, 8	empty	no change: the only neighbor of each has already been discovered							

Table 5.8: Trace of `bfsDist(1)`. Cells show `dist/parent`.

When 2 is taken from the queue, its neighbor 1 has already been discovered (`dist[1] = 0`), so it is skipped. 6 and 7 are new and get the value `dist[2] + 1 = 2`. When the queue is empty the result is as in Table 5.9.

vertex	1	2	3	4	5	6	7	8
distance	0	1	1	1	1	2	2	2
parent	—	1	1	1	1	2	2	3

Table 5.9: Result of BFS from source 1.

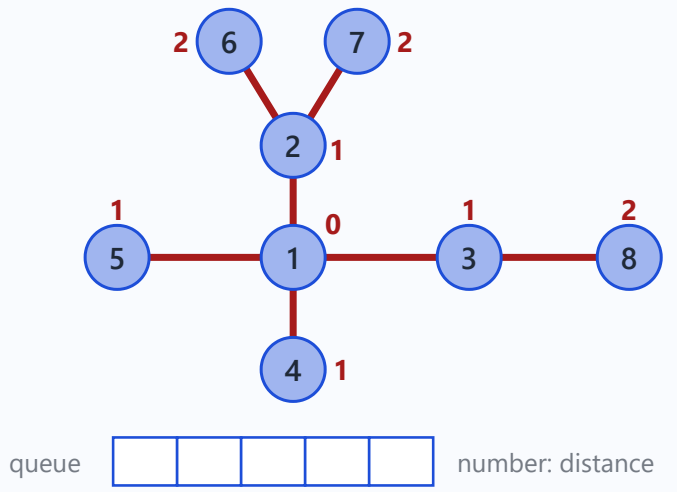


Figure 5.13: When `bfsDist(1)` finishes, the thick edges are the parent links and form the BFS tree. For example, the parent of 8 is 3 and the parent of 3 is 1; the shortest path from 1 to 8 is 1 → 3 → 8, with length `dist[8] = 2`.

The distances of the vertices leaving the queue are, in order, 0, 1, 1, 1, 1, 2, 2, 2; the sequence never decreases. This is the key property of BFS and is used in the proof of the shortest-path theorem.

5.4.7 Unweighted shortest paths

> **Theorem 5.2 (BFS and shortest paths)**
>
> In an unweighted graph, after BFS from s, for every v reachable from s, `dist[v]` equals the number of edges of a shortest path from s to v. Following the parent links from v to s yields a path of this length.

Proof. Two observations come first. (1) The distance of every vertex entering the queue is one more than that of the vertex that adds it. (2) At every moment the distances in the queue are in nondecreasing order, and the difference between the first and the last item is at most 1. The second observation follows by induction: at the start the queue holds only s. When the vertex at the front, with distance d, is removed and new vertices are appended with distance $d + 1$, the smallest value in the queue is at least d and the largest is at most $d + 1$. Now the theorem itself. It is clear that for any v, `dist[v]` cannot be smaller than the true shortest distance $\delta(v)$, because the parent chain is a real path from s to v with `dist[v]` edges. To show the converse, take a v with `dist[v] > δ(v)` whose δ value is smallest. Let u be the vertex before v on a shortest path from s to v; then $\delta(u) = \delta(v) - 1$, and by the choice of v, `dist[u] = δ(u)`. If v has not been discovered yet when u is removed from the queue, u discovers it and `dist[v] = δ(u) + 1 = δ(v)`, a contradiction. If v was discovered earlier, the vertex that added it was removed from the queue before u. By observation (2) the distance of that vertex is at most $\delta(u)$; then `dist[v] ≤ δ(u) + 1 = δ(v)`, again a contradiction. ∎

In plain words, the theorem says: at the moment BFS *first* reaches a vertex, it has reached it by a shortest path. If a shorter path existed, the vertex before the last on that path would have left the queue earlier and discovered the target earlier. BFS therefore writes the distance of a vertex once and never corrects it.

The path itself is rebuilt from the parent array:

```
List<Integer> path(int[] parent, int s, int t) {
  LinkedList<Integer> path = new LinkedList<>();
  for (int v = t; v != -1; v = parent[v])
    path.addFirst(v);   /* from target back */
  if (path.getFirst() != s)
    return null;       /* t unreachable */
  return path;
}
```

The loop starts at the target t and moves to the parent at each step, stopping when it reaches -1. Because the vertices are found from the target toward the source, each is added at the *front* of the list (`addFirst`); the list then comes out ordered from source to target. For source 1 and target 8 the loop

adds 8, moves to `parent[8] = 3`, adds 3, moves to `parent[3] = 1`, adds 1, and stops at `parent[1] = -1`. The list is `[1, 3, 8]`; its first item is the source, so the path is returned. The path has `dist[t] + 1` vertices, so building it takes *O(dist[t])* time.

The final `if` catches the case where the target was never reached. If *t* was not discovered, `parent[t] = -1`; the loop adds only *t* and stops. The first item of the list is *t*, not *s*, and the method returns `null`. This check works correctly only because `parent[]` is filled with *−1* at the start of BFS.

Common mistake

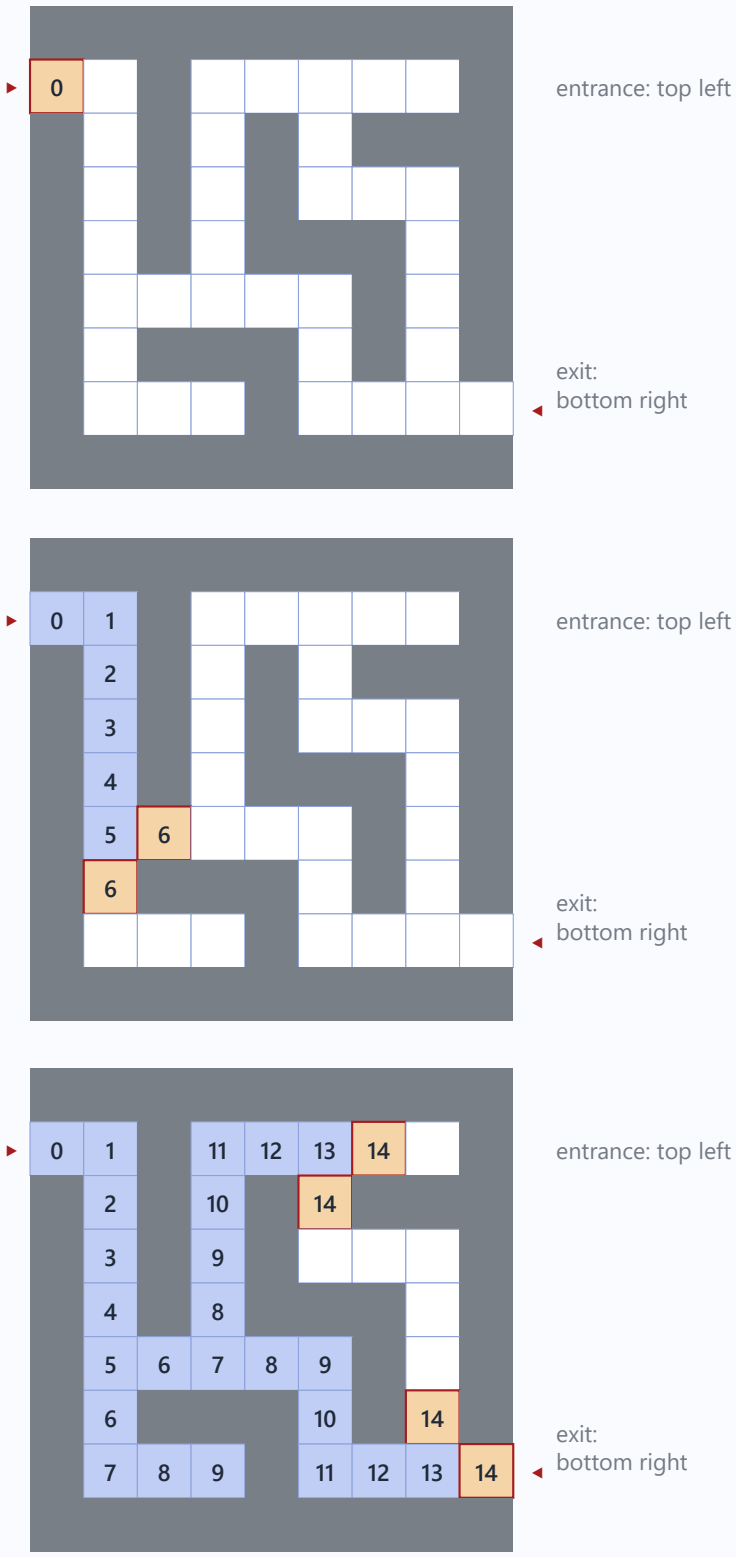
Filling only `dist[]` with *−1* in BFS and leaving `parent[]` at Java's default values. An `int` array is created with 0s, so for an unreachable *t* the entry `parent[t] = 0` remains and the path-building loop moves from *t* to 0. If the source is 0, the loop builds the list `[0, t]` and returns a path that does not exist. For example, in a graph with four vertices and the edges 0–1 and 1–3, there is no edge to 2, yet after a BFS from 0 the call `path(parent, 0, 2)` returns `[0, 2]`. If the source is not 0 and 0 is also unreachable, the loop spins on 0 forever. When `parent[]` is also filled with *−1*, the same call correctly returns `null`.

The theorem holds only in an **unweighted** graph, or one in which all weights are equal. BFS minimizes the number of edges, not the sum of weights. Take a graph with the edges $s \to t$ (weight 10), $s \to a$ (weight 1), and $a \to t$ (weight 1). BFS finds *t* in the first level, as a direct neighbor of *s*. The path found has a single edge but costs 10. The cheapest path is $s \to a \to t$, with cost 2. For shortest paths in a weighted graph, Dijkstra's algorithm is needed.

5.4.8 Solving a maze with BFS

A grid maze can be turned into a graph directly. Every open cell is a vertex. There is an edge between two open cells that are adjacent above, below, to the right, or to the left. Wall cells do not enter the graph at all. The question "fewest steps from entrance to exit" thus becomes "shortest path between two vertices in this graph". The graph is unweighted, since every step costs the same; the right tool is BFS.

The maze in Figure 5.14 has 9 × 9 cells. Of the cells, 47 are walls and 34 are open; the number of adjacent pairs of open cells is also 34. The graph has 34 vertices and 34 edges. The entrance is at the top left and the exit at the bottom right. The neighbors of a vertex are the open cells in the four directions. The neighbors are not stored in a separate list in the code; the row and column numbers of a cell's four neighbors are computed directly. A graph defined this way is called an **implicit graph**.


0
entrance: top left
exit:
bottom right
0
1
2
3
4
5
6
6
entrance: top left
exit:
bottom right
0
1
11
12
13
14
2
10
14
3
9
4
8
5
6
7
8
9
6
10
14
7
8
9
11
12
13
14
entrance: top left
exit:
bottom right

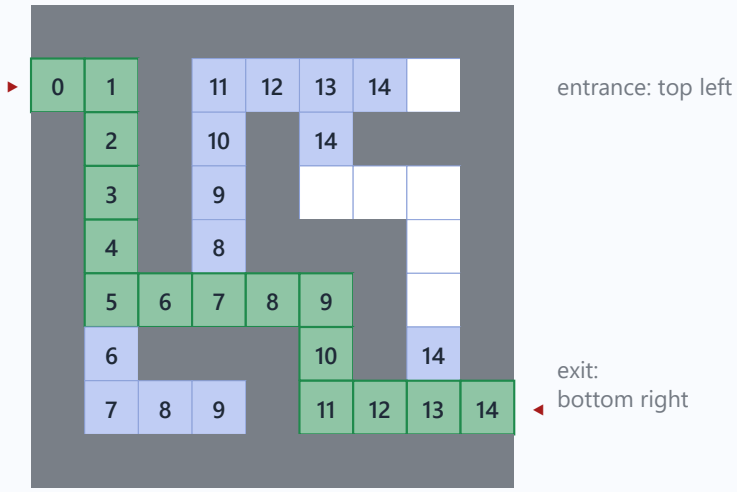


Figure 5.14: BFS in a maze. The number in a cell is the least number of steps from the entrance to that cell; yellow cells are the current frontier. Top left, the start: the entrance is in the queue with distance 0. Top right, distance 6: the path has forked at cell 5 and the wave has spread both to the right and downward. Bottom left, distance 14: the exit has been reached. Bottom right, the shortest path rebuilt by the parent links (green, 15 cells).

The traversal goes like this. The entrance cell is put in the queue with distance 0. In the first steps the maze is a single corridor and the wave descends; the number written in each cell is one more than the previous one. At the cell with distance 5 the path forks. The wave now splits into two arms: one goes to the right and from there to the room above, the other goes down. BFS advances these two arms not one after the other but together; in the queue the cells of distance 6 from both arms wait side by side, followed by the cells of distance 7. For this reason BFS is also described as a search that "tries all existing paths simultaneously".

At distance 10 the wave reaches both the corridor above and the lower corridor below. While the upper arm wanders in dead ends for a while, the lower corridor approaches the exit. The exit cell is reached in 14 steps. By the theorem this value is exactly the shortest; had a shorter path existed, the exit would have been discovered in an earlier level. The search can stop at this point. If it does not, six more cells with distances 15, 16, and 17 are discovered; these are unnecessary for the shortest path. The shortest path is found by starting at the exit and following the parent links back to the entrance. The path consists of 15 cells, that is, 14 steps.

DFS also finds the exit in the same maze, but the path it finds is not guaranteed to be the shortest. If at the fork it enters the upper arm first, it may reach the exit by a long detour after going through all the dead ends of that arm. The price of BFS is memory: it holds the whole frontier of the wave in the queue at once.

5.4.9 Comparing BFS and DFS

	DFS	BFS
Auxiliary structure	Stack (LIFO) or recursion	Queue (FIFO)
Traversal pattern	Follows one branch to its end, then backs up	Layer by layer, in order of distance
Time	$O(V + E)$ with lists; $O(V^2)$ with a matrix	$O(V + E)$ with lists; $O(V^2)$ with a matrix
Extra memory	$O(V)$: the longest path is on the stack	$O(V)$: the widest level is in the queue
Shortest path	No	Yes (in an unweighted graph)
Tree produced	Deep and thin	Shallow and wide
Typical uses	Cycles, components, topological order, SCCs, backtracking	Fewest steps, levels, bipartiteness test, spreading

Table 5.10: Comparison of DFS and BFS.

On the same directed graph, from the same source 0, DFS prints `0 1 2 3 5 6 4 7` and BFS prints `0 1 2 7 3 4 5 6`. Both visit every vertex once and have the same running time; the difference lies in the visit order and in the trees they produce.

The memory comparison depends on the shape of the graph. Traverse a complete binary tree with 15 vertices (four levels) from the root. In BFS the eight leaves of the last level are in the queue at the same time; the queue holds at most 8 vertices. In recursive DFS the call stack goes as deep as the path from the root to a leaf, 4 frames. If the tree had 20 levels, that is 1,048,575 vertices, the BFS queue would reach 524,288 vertices while the DFS stack would stay at 20 frames. Conversely, on a long path graph the DFS stack holds all the vertices, while the BFS queue holds a single vertex if it starts at an end. The worst case of both is $O(V)$; which uses less memory is determined by the depth and width of the graph.

Example 5.4

In the undirected graph of Example 5.3 (A–B, A–C, B–D, C–D, C–E, D–F, E–F, F–G; neighbors in alphabetical order), find the BFS visit order from A, the distances, and the shortest path from A to G.

1. A is marked and enters the queue. Queue: A.
2. A is taken; B and C are discovered (distance 1, parent A). Queue: B, C.
3. B is taken; A is marked, D is discovered (distance 2, parent B). Queue: C, D.
4. C is taken; A and D are marked, E is discovered (distance 2, parent C). Queue: D, E.
5. D is taken; B and C are marked, F is discovered (distance 3, parent D). Queue: E, F.
6. E is taken; C and F are marked. Queue: F.
7. F is taken; D and E are marked, G is discovered (distance 4, parent F). Queue: G.
8. G is taken; F is marked. The queue empties.

The visit order is **A B C D E F G**. The distances are A 0, B 1, C 1, D 2, E 2, F 3, G 4. F was discovered by D, not E, because D leaves the queue before E. Following parents from G: the parent of G is F, of F is D, of D is B, of B is A. Reversed, this gives the shortest path **A–B–D–F–G**, four edges long. The path A–B–D–C–E–F–G that DFS followed on the same graph has six edges.

5.5 Strongly connected components

In an undirected graph connectivity is a simple notion: if there is a path from u to v, there is one from v to u. In a directed graph this symmetry breaks. In a city full of one-way streets one can get from one neighborhood to another and yet be unable to return the same way. Finding which vertices can reach each other in both directions is the first step toward understanding the structure of directed graphs.

5.5.1 Definition

Definition 5.7

In a directed graph, u and v are **mutually reachable** if there is a path from u to v and a path from v to u. A maximal set of vertices in which every vertex can reach every other, and which cannot be enlarged without losing this property, is a **strongly connected component** (SCC).

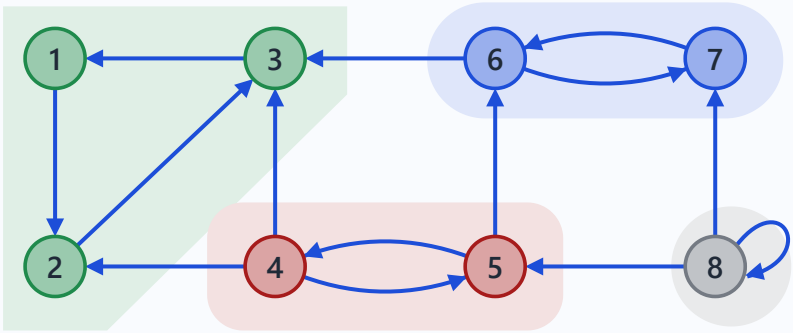


Figure 5.15: A directed graph on eight vertices and its four strongly connected components: *{1, 2, 3}*, *{4, 5}*, *{6, 7}*, and *{8}*. Vertices of the same color are in the same component.

The edges of the graph in Figure 5.15 are $1\to2$, $2\to3$, $3\to1$, $4\to2$, $4\to3$, $4\to5$, $5\to6$, $5\to4$, $6\to3$, $6\to7$, $7\to6$, $8\to5$, $8\to7$, and $8\to8$. Vertices 1, 2, and 3 lie on a cycle ($1 \to 2 \to 3 \to 1$); each reaches the others. There are edges in both directions between 4 and 5, and between 6 and 7. Vertex 8 is a component by itself: every other vertex is reachable from 8, but 8 is reachable from no vertex.

A few consequences follow from the definition. Mutual reachability is an equivalence relation: every vertex reaches itself (a path of length zero), the relation is symmetric, and it is transitive (if $u \rightleftarrows v$ and $v \rightleftarrows w$ then $u \rightleftarrows w$). Strongly connected components therefore partition the vertex set into disjoint parts; every vertex belongs to exactly one component. A single vertex can also be a component; every vertex that takes part in no cycle is one.

A common misconception is to take "a vertex together with all vertices reachable from it" as a component. All vertices are reachable from 8, but *{1, ..., 8}* is not an SCC, because, for example, there is no way back from 1 to 8. An SCC requires reachability in both directions.

If each component is contracted to a single vertex and the edges between components are kept, the result is the **component graph**. In the example in the figure its edges are *{8}* → *{4, 5}*, *{8}* → *{6, 7}*, *{4, 5}* → *{1, 2, 3}*, *{4, 5}* → *{6, 7}*, and *{6, 7}* → *{1, 2, 3}*. The component graph is always acyclic. If there were paths in both directions between two components, the vertices of these components would be mutually reachable, and the two components would in fact be one. This structure is used in many applications: finding communities of people who follow each other in social networks, compiling mutually dependent modules of a compiler together, solving the 2-SAT problem in logic, and others.

The crudest way to find the components is to run a traversal from every vertex and test mutual reachability one pair at a time; this takes $O(V \cdot (V + E))$ time. In 1972 Robert Tarjan published an algorithm that finds all components with **a single DFS**, in $O(V + E)$ time.

5.5.2 The idea behind Tarjan's algorithm

Tarjan's algorithm assigns two numbers to each vertex during DFS and keeps the discovered vertices in a separate stack.

> **Definition 5.8**
>
> **index(v)** is the discovery order of *v*: the first discovered vertex has index 1, the second 2. **low(v)**, the **low-link** value of *v*, is the smallest index reachable from the DFS subtree of *v* using at most one non-tree edge to a vertex that is still on the stack. Initially *low(v) = index(v)*.

This stack differs from the DFS call stack. A vertex is pushed onto it when it is discovered, but it is not popped as soon as its work is finished. It stays on the stack until its component has been determined. The stack therefore holds the vertices that have been discovered so far and not yet assigned to any component.

While examining the edges of a vertex *v*, DFS meets three cases:

1. **Tree edge** $v \to w$, with *w* not yet discovered: continue recursively from *w*. When the call returns, set *low(v) = min(low(v), low(w))*. If the subtree of *w* can reach an older vertex, so can *v*.

2. **An edge to a vertex *w* on the stack:** *w* has been discovered and not yet assigned to a component. Set *low(v) = min(low(v), index(w))*. Vertex *v* can reach a vertex that was discovered before it and whose component is still open.
3. **An edge to a vertex *w* whose component has been found:** *w* has been discovered and popped from the stack. This edge is **ignored**. The component of *w* is closed; there is no path from *w* back to *v*, and *v* cannot join that component.

The decision is made after all edges of *v* have been examined. If *low(v) = index(v)*, then no vertex on the stack older than *v* can be reached from the subtree of *v*. Vertex *v* is the **root** of a component, that is, the first vertex discovered in that component. All the vertices popped from the stack down to *v* (including *v*) form an SCC. If *low(v) < index(v)*, an older stack vertex can be reached back from the subtree of *v*; *v* is in the same component as that vertex and is not a root. Vertex *v* stays on the stack and its component is popped together when the root finishes.

The root condition can be justified briefly as follows. Let *r* be the first vertex discovered in a component. All other vertices of the component are reachable from *r*, so they fall in the subtree of *r* and are discovered before *r* finishes. No vertex of the component can reach a stack vertex older than *r*; if one could, that vertex would reach *r*, *r* would reach it, and it would be in the same component. So *low(r) = index(r)* remains. The other vertices of the component, since they can return to *r*, get a smaller low-link value.

5.5.3 Tarjan step by step

On the graph of Figure 5.15, Tarjan's algorithm tries the vertices in order from 1 to 8. Indices start at 1. Next to each vertex we write the pair "index, low". Table 5.11 gives all the steps of the algorithm.

Step	Event	Stack (after)	SCC found
1	SCC(1): index = low = 1	1	
2	1 → 2 tree edge; SCC(2): 2, 2	1, 2	
3	2 → 3 tree edge; SCC(3): 3, 3	1, 2, 3	
4	3 → 1: 1 is on the stack → low(3) = min(3, 1) = 1	1, 2, 3	
5	3 finishes; 1 < 3, not a root. Back at 2: low(2) = min(2, 1) = 1	1, 2, 3	
6	2 finishes, not a root. Back at 1: low(1) = 1. At 1, low = index → root	empty	*{1, 2, 3}*
7	SCC(4): 4, 4	4	
8	4 → 2 and 4 → 3 ignored (components found); 4 → 5: SCC(5): 5, 5	4, 5	
9	5 → 6: SCC(6): 6, 6	4, 5, 6	
10	6 → 3 ignored; 6 → 7: SCC(7): 7, 7	4, 5, 6, 7	
11	7 → 6: 6 is on the stack → low(7) = min(7, 6) = 6	4, 5, 6, 7	
12	7 finishes, not a root. Back at 6: low(6) = 6 unchanged; 6 is a root	4, 5	*{6, 7}*
13	Back at 5: low(5) = min(5, 6) = 5. 5 → 4: 4 is on the stack → low(5) = min(5, 4) = 4	4, 5	
14	5 finishes, not a root. Back at 4: low(4) = min(4, 4) = 4; 4 is a root	empty	*{4, 5}*

Step	Event	Stack (after)	SCC found
15	SCC(8): 8, 8	8	
16	8 → 5 and 8 → 7 ignored; self-loop 8 → 8: 8 is on the stack, low = min(8, 8) = 8	8	
17	At 8, low = index → root	empty	*{8}*

Table 5.11: Tarjan's algorithm on the graph with eight vertices. The stack grows from left to right.

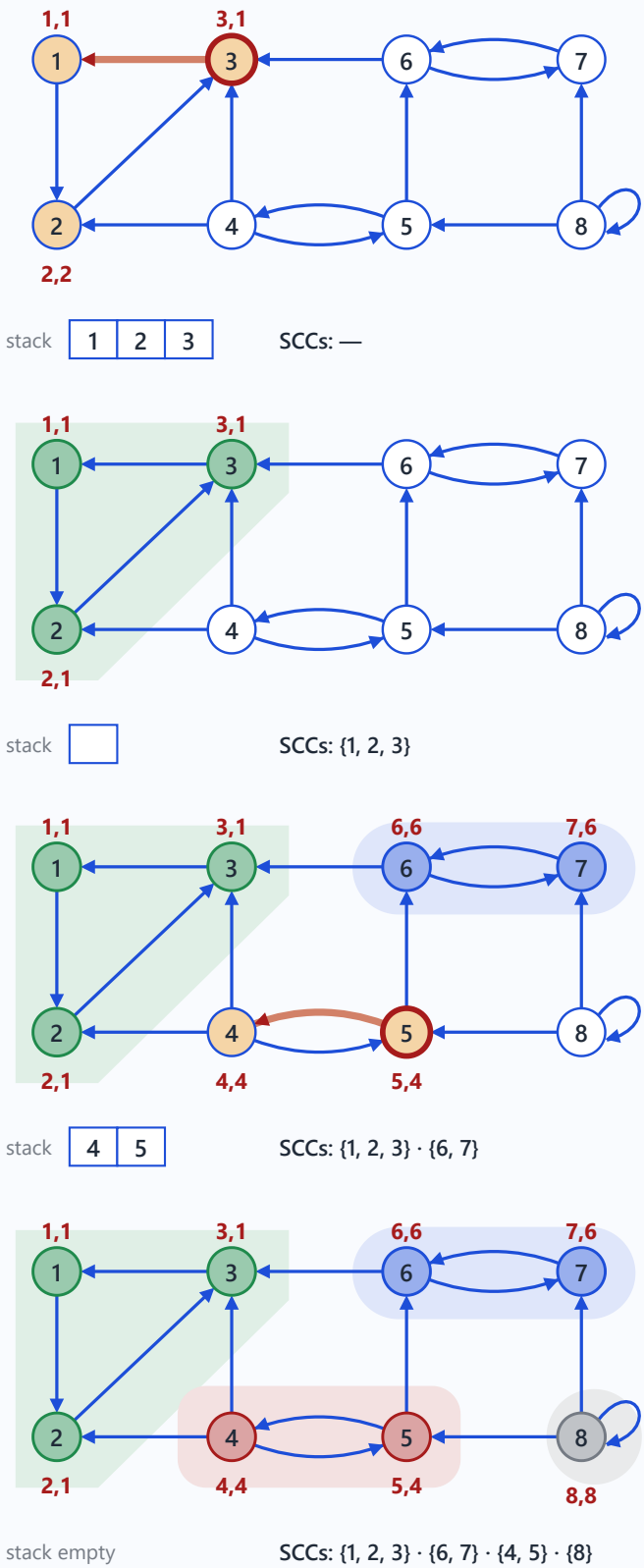


Figure 5.16: Four moments of Tarjan's algorithm; "index, low" is written next to the vertices. Top left: the edge 3 → 1 lowers the low-link of 3 to 1. Top right: 1 has been found to be a root and *{1, 2, 3}* has been popped from the stack. Bottom left: the edge 5 → 4 lowers the value of 5 to 4. Bottom right: all four components have been found.

Three moments in the table show the whole logic of the algorithm. In step 4 the edge 3 → 1 catches a return to an older vertex on the stack and lowers the low-link of 3 to 1. This value is carried to 2 and to 1 by the call returns in steps 5 and 6. In the end only 1 has index equal to low; 1 is a root, and every vertex above 1 on the stack is in its component.

In step 8 the edges 4 → 2 and 4 → 3 are ignored. Vertices 2 and 3 have been discovered but are not on the stack; their components are closed. Had these edges been taken into account, the low-link of 4 would have dropped to 2 and 4 would not have been counted as a root. Yet although 4 reaches 2, there is no path from 2 back to 4; 4 is the first vertex of a new component.

In step 12, 6 is found to be a root and *{6, 7}* is popped, but 5 and 4 stay on the stack. The component of 5 and 4 is closed later, in step 14. The components are found in the order *{1, 2, 3}*, *{6, 7}*, *{4, 5}*, *{8}*. Comparing with the edges of the component graph shows that this order is a **reverse topological order**: each component is found before all components that have edges into it. No edge leaves *{1, 2, 3}*, and it is found first; no edge enters *{8}*, and it is found last. This is no coincidence: a component can close only after all the components reachable from it have been found.

5.5.4 Pseudocode and Java code

The pseudocode of the algorithm has two parts. The outer loop calls `SCC` from every unvisited vertex:

```
TARJAN_SCC(G):
  index = 1
  stack = [ ]
  sccs = [ ]
  for each v in V:
    if v.index is not defined:
      SCC(v)
  return sccs
```

The real work is in the recursive procedure `SCC`:

```
SCC(v):
  v.index = v.low = index++
  stack.push(v)
  for each (v, w) in E:
    if w.index is not defined:
      SCC(w)
      v.low = min(v.low, w.low)
    else if w is on the stack:
      v.low = min(v.low, w.index)
  if v.low == v.index:
    scc = [ ]
    repeat until w == v:
      w = stack.pop()
      scc.add(w)
    sccs.add(scc)
```

The Java code is a direct counterpart of the pseudocode:

```
void scc(int v) {
  index[v] = low[v] = ++counter;
  stack.push(v); onStack[v] = true;
  for (int w : adj[v]) {
    if (index[w] == 0) {
      scc(w);
      low[v] = Math.min(low[v], low[w]);
    } else if (onStack[w])
      low[v] = Math.min(low[v], index[w]);
  }
  if (low[v] == index[v]) {
    int w;
    do {
      w = stack.pop(); onStack[w] = false;
      System.out.print(w + " ");
    } while (w != v);
    System.out.println();
  }
}
```

- `index[v] = low[v] = ++counter;` assigns the discovery order. `counter` starts at 0 and is incremented first, so the first vertex gets index 1. This choice lets the test `index[w] == 0` mean "not discovered yet", so no separate visited array is needed. Starting the indices at 0 gives the same components; then another marker, for example *–1*, is needed for undiscovered vertices.
- `stack.push(v); onStack[v] = true;` pushes the vertex. The array `onStack[]` answers the question "is *w* on the stack?" in *O(1)* time. Searching the stack from end to end would take *O(V)* at every edge.

- The branch `if (index[w] == 0)` is the tree edge: after the recursive call, `low[v]` is updated with `low[w]`.
- The branch `else if (onStack[w])` is the edge to a vertex on the stack: `low[v]` is updated with `index[w]`. For a *w* whose component has been found, both conditions are false and nothing is done.
- The last `if` is the root test. The `do … while` loop pops all vertices from the stack down to *v* and prints them. The loop is of the `do … while` form because at least one vertex (*v* itself) is always popped. Each component is written on a separate line.

5.5.5 Code trace: scc(0)

The graph in the trace has five vertices: *0→1*, *1→2*, *2→0*, *2→3*, *3→4*, *4→3*. `adj[2] = [0, 3]`.

Step	Event	Changed value	Stack (after)	Output
1	`scc(0)`	0: 1/1	0	
2	0 → 1 undiscovered → `scc(1)`	1: 2/2	0, 1	
3	1 → 2 undiscovered → `scc(2)`	2: 3/3	0, 1, 2	
4	2 → 0: 0 is on the stack	`low[2] = min(3, 1) = 1`	0, 1, 2	
5	2 → 3 undiscovered → `scc(3)`	3: 4/4	0, 1, 2, 3	
6	3 → 4 undiscovered → `scc(4)`	4: 5/5	0, 1, 2, 3, 4	
7	4 → 3: 3 is on the stack	`low[4] = min(5, 4) = 4`	0, 1, 2, 3, 4	
8	`low[4] = 4 ≠ index[4] = 5`: not a root, returns		0, 1, 2, 3, 4	
9	`scc(4)` returned	`low[3] = min(4, 4) = 4`	0, 1, 2, 3, 4	
10	`low[3] == index[3] = 4`: 3 is a root; 4 and 3 are popped		0, 1, 2	4 3
11	`scc(3)` returned	`low[2] = min(1, 4) = 1`	0, 1, 2	4 3
12	`low[2] = 1 ≠ 3`: not a root, returns	`low[1] = min(2, 1) = 1`	0, 1, 2	4 3
13	`low[1] = 1 ≠ 2`: not a root, returns	`low[0] = min(1, 1) = 1`	0, 1, 2	4 3
14	`low[0] == index[0] = 1`: 0 is a root; 2, 1, 0 are popped		empty	4 3 / 2 1 0

Table 5.12: Trace of `scc(0)`. The `index/low` values are written in the row where they change.

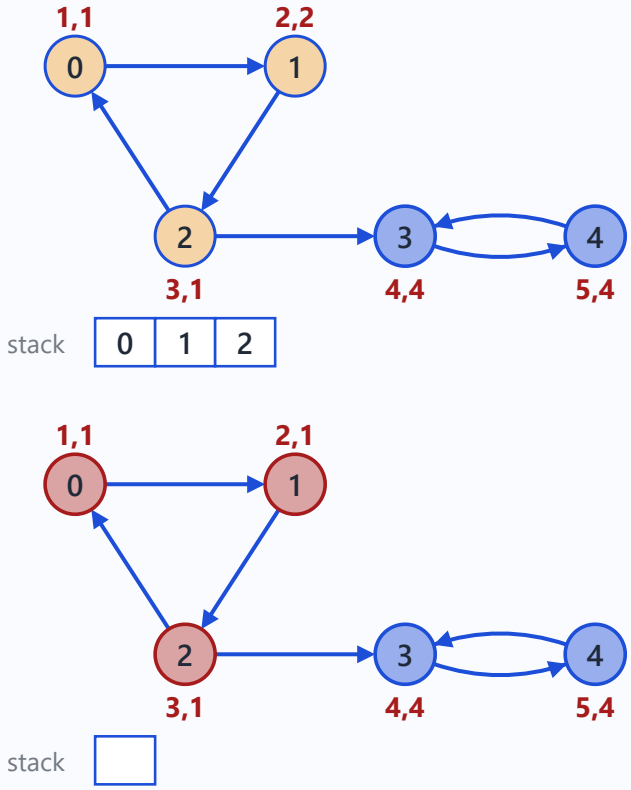


Figure 5.17: Left, step 10: 3 has been found to be a root and *{3, 4}* has been popped from the stack; 0, 1, 2 remain on it. Right, the end of the trace: two components have been found. The numbers next to the vertices are `index/low` values.

The program prints two lines: first `4 3`, then `2 1 0`. Vertices within a component are printed in the order they are popped from the stack, that is, the reverse of discovery order. The edge 2 → 0 in step 4 is a back edge and lowers `low[2]` to 1. In step 11, when `scc(3)` returns, the value `low[3] = 4` does not change that of 2, since 2 has already reached 1. The component *{3, 4}* is reachable from 2 but cannot return to 2, and it closes first.

Example 5.5

Let the directed graph have the edges *0→1*, *1→2*, *2→0*, *1→3*, *3→4*, *4→5*, *5→3*, *5→6* (`adj[1] = [2, 3]`, `adj[5] = [3, 6]`). Tarjan's algorithm is started at 0 and indices start at 1.

1. Vertices 0, 1, 2 are discovered in order: indices 1, 2, 3. The edge 2 → 0 goes to 0 on the stack; `low[2] = 1`. Vertex 2 is not a root; on return, `low[1] = 1`.
2. 1 → 3 is a tree edge. Vertices 3, 4, 5 are discovered in order: indices 4, 5, 6. The edge 5 → 3 goes to 3 on the stack; `low[5] = 4`.
3. 5 → 6 is a tree edge; the index of 6 is 7. Vertex 6 has no outgoing edges, `low[6] = index[6] = 7` : 6 is a root. The first component is *{6}*.

4. Back at 5: `low[5] = min(4, 7) = 4`; 5 is not a root. Return to 4 gives `low[4] = 4`, return to 3 gives `low[3] = 4`. At 3, index and low-link are equal: the second component is *{3, 4, 5}*.
5. Back at 1: `low[1] = min(1, 4) = 1`. Back at 0, and 0 is a root: the third component is *{0, 1, 2}*.

The components are found in the order *{6}*, *{3, 4, 5}*, *{0, 1, 2}*. The component graph is *{0, 1, 2}* → *{3, 4, 5}* → *{6}*; the order of discovery is its reverse.

5.5.6 Complexity

Theorem 5.3 (Cost of Tarjan's algorithm)

On a graph stored as adjacency lists, Tarjan's algorithm finds all strongly connected components in $O(V + E)$ time and $O(V)$ extra memory.

Proof. The algorithm is a DFS, and the DFS count applies unchanged. `scc` is called once for each vertex; each vertex is pushed once and popped once. The work per vertex is constant: $O(V)$. Each edge is examined once, when the adjacency list of its source is walked, and the work done in each examination (one `min` and an `onStack[]` test) is constant: $O(E)$. The total number of iterations of the popping loop is the number of popped vertices, that is, V. The extra memory is $O(V)$ for the arrays `index`, `low`, `onStack` and for the stack, and again $O(V)$ for the recursion depth. ■

Common mistake

Skipping the test `else if (onStack[w])` in Tarjan's code and doing `low[v] = min(low[v], index[w])` for every discovered w. An edge to a vertex whose component has been found is then wrongly counted as a "return". In the eight-vertex example the edge 4 → 2 would lower low(4) to 2, the edge 6 → 3 would lower low(6) to 3, and the edge 8 → 5 would lower low(8) to 5. Vertices 4, 6, and 8 would no longer be counted as roots. The first component *{1, 2, 3}* would still be found correctly, but *{4, 5}*, *{6, 7}*, and *{8}* would never be popped; all the vertices from 4 to 8 would remain on the stack when the algorithm ends.

5.5.7 Kosaraju's algorithm

The second classical method for strongly connected components is Kosaraju's algorithm. It is easier to understand than Tarjan's, but it needs two DFS passes and a copy of the graph. The algorithm works on the **transpose** of a graph: the transpose has the same vertices, and each edge $u \to v$ becomes $v \to u$. A graph and its transpose have the same strongly connected components, because mutual reachability is not affected by the direction of the arrows.

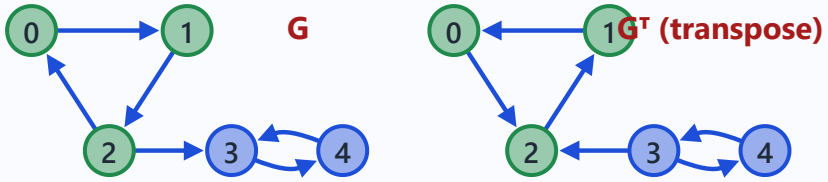


Figure 5.18: A graph on the left and its transpose on the right. The component colors stay the same; only the arrows turn.

The algorithm has three steps:

1. Run DFS on the graph and push each vertex onto a stack **when it finishes**. The vertex that finished last sits on top of the stack.
2. Build the transpose of the graph.
3. Pop vertices from the stack in order. If the popped vertex has not been visited yet, run a DFS from it in the transpose. The vertices this DFS reaches form an SCC.

The part that builds the transpose:

```
List<Integer>[] rev = new List[n];
for (int v = 0; v < n; v++)
  rev[v] = new ArrayList<>();
for (int v = 0; v < n; v++)
  for (int w : adj[v])
    rev[w].add(v);       /* edge v to w becomes w to v */
```

The Java code of the algorithm:

```
void kosaraju() {
  for (int v = 0; v < n; v++)  /* pass 1 */
    if (!visited[v]) dfs1(v);
  Arrays.fill(visited, false); /* pass 2 */
  while (!stack.isEmpty()) {
    int v = stack.pop();
    if (!visited[v]) {   /* new SCC */
      dfs2(v); System.out.println(); }
  }
}
void dfs1(int v) {
  visited[v] = true;
  for (int w : adj[v])
    if (!visited[w]) dfs1(w);
  stack.push(v);       /* v finished */
}
```

`dfs2` in the second pass is an ordinary DFS running on the transpose:

```
void dfs2(int v) {     /* DFS on transpose */
  visited[v] = true;
  System.out.print(v + " ");
  for (int w : rev[v])
    if (!visited[w]) dfs2(w);
}
```

`dfs1` differs from `dfs` by a single line: a vertex is pushed onto the stack after all its neighbors have finished (`stack.push(v); /* v finished */`). This line records the finishing order. `kosaraju()` first starts `dfs1` from every unvisited vertex. Then it clears the visited array and empties the stack. Every unvisited vertex popped from the stack is the start of a new component.

Why the algorithm is correct is explained with the component graph. The vertex that finishes last in the first pass lies in a *source* component, one with no incoming edges in the component graph. In the transpose all the outgoing edges of this component turn inward. A DFS started from it in the transpose therefore cannot leave the component and traverses exactly that component. The next unvisited vertex on the stack lies, among the remaining components, again in a source component. In the transpose the edges leaving it go either to its own component or to components already found; visited vertices are not entered either. Thus each DFS finds exactly one component.

Let us trace Kosaraju's algorithm on the five-vertex graph of the Tarjan trace (*0→1*, *1→2*, *2→0*, *2→3*, *3→4*, *4→3*).

Pass	Event	Stack (after)	SCC found
1	`dfs1(0)` → `dfs1(1)` → `dfs1(2)` ; 2 → 0 visited, skipped; `dfs1(3)` → `dfs1(4)`	empty	
1	The only neighbor of 4, 3, is visited: 4 finishes, is pushed	4	
1	3 finishes, is pushed	4, 3	
1	2, 1, 0 finish in order and are pushed	4, 3, 2, 1, 0	
2	0 is popped; in the transpose `dfs2(0)` : 0 → 2 → 1	4, 3, 2, 1	*{0, 2, 1}*
2	1 and 2 are popped; visited, skipped	4, 3	
2	3 is popped; `dfs2(3)` : the reversed edge 3 → 2 goes to visited 2; 3 → 4	4	*{3, 4}*
2	4 is popped; visited, skipped	empty	

Table 5.13: Trace of Kosaraju's algorithm. The stack grows from left to right; the top is on the right.

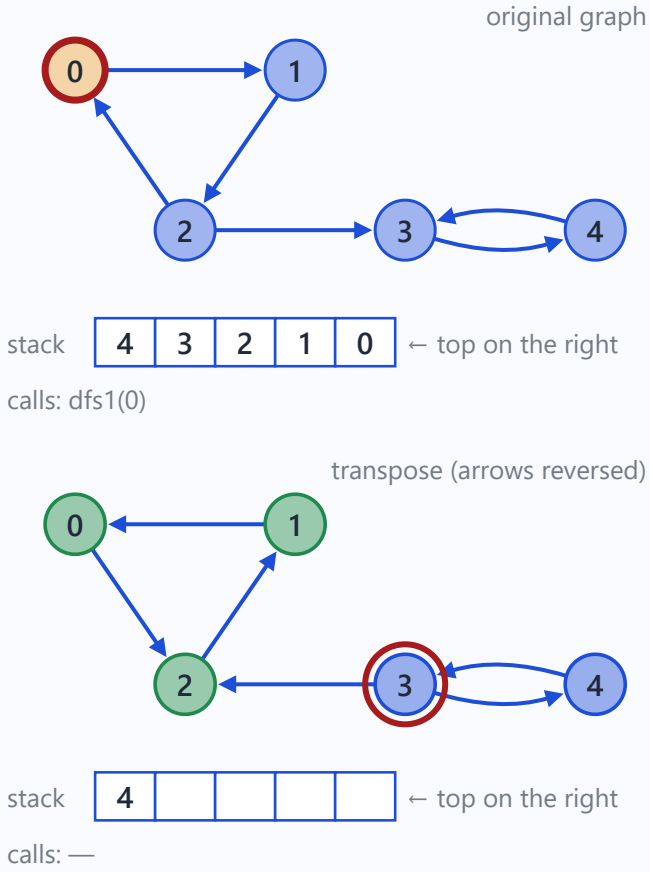


Figure 5.19: Left, the end of the first pass: all vertices have finished and been pushed in finishing order; 0, which finished last, is on top. Right, the second pass: *{0, 1, 2}* has been found in the transpose, and the DFS from 3 skips the reversed edge 3 → 2 and reaches 4.

The program prints two lines: `0 2 1` and `3 4`. In the transpose the adjacency list of 3 is `[2, 4]`; the edge 2 → 3 has become 3 → 2. This edge would join 3 to the previous component, but since 2 is visited, the DFS does not go there. Because 0 finished last in the first pass, the second pass started from *{0, 1, 2}*, a source component. This component has an edge to *{3, 4}* in the original graph; since that edge turns inward in the transpose, `dfs2(0)` cannot reach 3.

On the eight-vertex example Kosaraju's algorithm finds the same four components. The finishing order of the first pass is 3, 2, 1, 7, 6, 5, 4, 8. In the second pass the components are found in the order *{8}*, *{4, 5}*, *{6, 7}*, *{1, 2, 3}*. This order is exactly the reverse of Tarjan's: Kosaraju gives the components in topological order, Tarjan in reverse topological order.

	Tarjan	Kosaraju
Number of DFS passes	1	2 (one on the transpose)
Extra structures	index and low arrays, stack	transpose graph (*O(V + E)* memory), stack
Time	*O(V + E)*	*O(V + E)*
Order of components	Reverse topological	Topological

Table 5.14: Comparison of Tarjan's and Kosaraju's algorithms.

The two algorithms have the same order of growth. Tarjan traverses the graph once and needs no copy; in practice it is usually faster. Kosaraju's logic is easier to grasp and its correctness easier to justify. Building the transpose takes $O(V + E)$ time and memory but does not change the order of growth.

5.6 Applications of traversal

By themselves, DFS and BFS only walk through the vertices in some order. When a few lines are added to a traversal, many questions about the structure of a graph can be answered in the same $O(V + E)$ time. This section treats the most common applications.

5.6.1 Topological sorting

Definition 5.9

A **topological sort** of a directed acyclic graph (DAG) is an ordering of its vertices such that for every edge $u \to v$, u comes before v.

Edges express prerequisites of the kind "this first, then that". Courses with prerequisites, the build order of a project, the order in which the cells of a spreadsheet are computed, and the scheduling of jobs on a construction site are all topological sorting problems. For example, if "Data Structures" must be taken after "Programming", the study plan must contain these two courses in that order.

A topological order exists only in a directed acyclic graph. If the graph had a cycle $u \to v \to \dots \to u$, then u would have to come both before and after every vertex of the cycle. Conversely, every DAG has at least one topological order. Moreover, a topological order is often not unique: two vertices with no path between them can be written in either order.

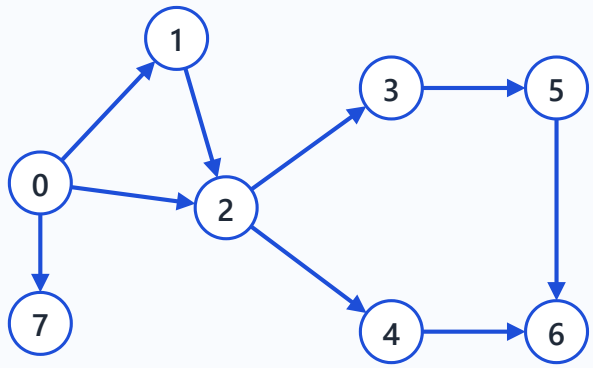


Figure 5.20: A directed acyclic graph on eight vertices: $0\to1$, $0\to2$, $0\to7$, $1\to2$, $2\to3$, $2\to4$, $3\to5$, $4\to6$, $5\to6$. The in-degrees are 0:0, 1:1, 2:2, 3:1, 4:1, 5:1, 6:2, 7:1.

The graph in Figure 5.20 was obtained from the directed graph of the DFS trace by removing the edge $2 \to 0$. That edge added a cycle to the graph; with the edge gone, the graph is acyclic. There are two classical methods for topological sorting: Kahn's algorithm, which is BFS-based, and the reverse finishing order of DFS. On this graph the two give different results, but both are valid:

Method	Order
Kahn	`0 1 7 2 3 4 5 6`
DFS (reverse finishing)	`0 7 1 2 4 3 5 6`

Table 5.15: Two topological orders of the same DAG.

In both orders each of the nine edges goes from left to right. For the edges $4 \to 6$ and $5 \to 6$, for example, 6 comes last in both. Vertex 7 only has to come after 0; Kahn places it after 1, and DFS places it right behind 0.

5.6.2 Kahn's algorithm and a code trace

The idea of Kahn's algorithm is to do, in turn, the jobs that have no prerequisites left. A vertex with in-degree 0 has no prerequisite; it can be written first. A written vertex and its edges are regarded as deleted from the graph. This deletion reduces the in-degree of some vertices to 0, and those can now be written too. Ready vertices wait in a queue.

```
void kahn() {
  int[] indeg = new int[n];
  for (int v = 0; v < n; v++)
    for (int w : adj[v]) indeg[w]++;
  Queue<Integer> queue = new LinkedList<>();
  for (int v = 0; v < n; v++)
    if (indeg[v] == 0) queue.add(v);
  while (!queue.isEmpty()) {
    int v = queue.poll();
    System.out.print(v + " ");
    for (int w : adj[v])
      if (--indeg[w] == 0) queue.add(w);
  }
}
```

- The first nested loop increases the in-degree of each edge's target by one. Since all adjacency lists are walked once, this part is $O(V + E)$.
- The second loop puts all vertices with in-degree 0 into the queue.
- The main loop takes a vertex from the queue and writes it. For each outgoing edge the target's degree is decremented: `--indeg[w]`. The edge itself is not deleted; only the counter is decremented. A target whose degree drops to 0 is added to the queue.

Taken	Degrees decremented	`indeg` (after)	Queue (after)	Output
—	degrees are counted; 0 enters the queue	0 1 2 1 1 1 2 1	0	
0	1 → 0, 2 → 1, 7 → 0	0 0 1 1 1 1 2 0	1, 7	0
1	2 → 0	0 0 0 1 1 1 2 0	7, 2	0 1
7	no outgoing edges	same	2	0 1 7
2	3 → 0, 4 → 0	0 0 0 0 0 1 2 0	3, 4	0 1 7 2
3	5 → 0	0 0 0 0 0 0 2 0	4, 5	0 1 7 2 3
4	6 → 1	0 0 0 0 0 0 1 0	5	0 1 7 2 3 4
5	6 → 0	0 0 0 0 0 0 0 0	6	0 1 7 2 3 4 5
6	no outgoing edges	same	empty	0 1 7 2 3 4 5 6

Table 5.16: Trace of `kahn()`. The array `indeg` is indexed from 0 to 7.

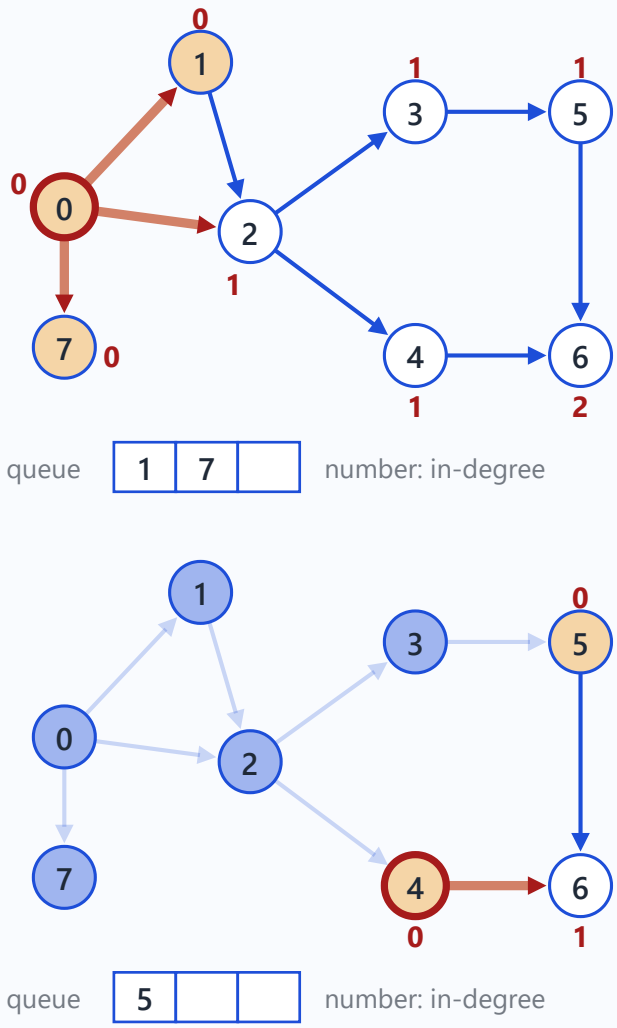


Figure 5.21: Left: 0 has been removed and its edges regarded as deleted; 1 and 7, whose degrees dropped to 0, have entered the queue. Right: 4 has been removed; the degree of 6 has dropped to 1, but 6 cannot enter the queue because the edge 5 → 6 is still waiting.

The output is `0 1 7 2 3 4 5 6`. When 4 is removed, the degree of 6 only drops to 1: one of the two prerequisites of 6 (4) is complete and the other (5) is still pending. Vertex 6 enters the queue only after 5 has been removed. This is the essence of Kahn's algorithm: a vertex is not written before all its prerequisites have been written.

Kahn's algorithm is also a **cycle test**. If the graph has a cycle, none of the vertices on the cycle ever has its degree reduced to 0, because each waits for the previous vertex on the cycle. The queue empties before all vertices have been written. Hence, if fewer than V vertices are written, the graph has a cycle. If the edge $2 \to 0$ is put back into the example graph, 0 gets in-degree 1 and no vertex with in-degree 0 is left at the start. The queue starts empty, no vertex is written, and the algorithm reports the cycle.

The running time is $O(V + E)$: each vertex enters and leaves the queue once, and each edge is counted once and decremented once. Another structure can be used in place of the queue. If, for example, ready vertices are kept in a priority queue, the ready vertex with the smallest number is chosen at each step. This yields the lexicographically smallest topological order; the time rises to $O((V + E) \log V)$.

5.6.3 Topological sorting with DFS

In the DFS-based method, vertices are pushed onto a stack at the moment they finish. When the traversal ends, the order popped from the stack is a topological order. This is the same job as the first pass of Kosaraju's algorithm:

```
void dfsFinish(int v) {
  visited[v] = true;
  for (int w : adj[v])
    if (!visited[w]) dfsFinish(w);
  stack.push(v);           /* push when done */
}
```

The main program calls `dfsFinish` from every unvisited vertex, then empties the stack and writes the vertices. On the graph of Figure 5.20, `dfsFinish(0)` descends first to 1, then from there to 2, 3, 5, and 6. Vertex 6 has no outgoing edges; it finishes first. Then 5 and 3 finish. The next neighbor of 2 is 4; the neighbor of 4, 6, has already finished, and 4 finishes. Then 2, 1, 7, and 0 finish. The finishing order is 6, 5, 3, 4, 2, 1, 7, 0; the order popped from the stack is its reverse, `0 7 1 2 4 3 5 6`.

Theorem 5.4 (Reverse finishing order)

If DFS is run on a directed acyclic graph, then for every edge $u \to v$, v finishes before u. The reverse of the finishing order is therefore a topological order.

Proof. The edge $u \to v$ is examined while the adjacency list of u is being walked. At that moment v can be in one of three states. If v is white (undiscovered), DFS descends from u to it, and the call of v returns before that of u. If v is dark (finished), it has already finished before u. Vertex v cannot be yellow: if it were, v would be on the call stack, that is, an ancestor of u; the tree path from v to u and the edge $u \to v$ would form a cycle. Since the graph is acyclic, this is impossible. In every case v finishes before u, and in the reverse order u comes first. ■

The step "cannot be yellow" in the proof also gives the cycle test: if during DFS an edge to a yellow vertex is found, the graph has a cycle and no topological order exists. The DFS method also runs in $O(V + E)$ time. Kahn's method produces the order from the beginning to the end. Because it keeps ready jobs in a queue, it also suits parallel scheduling. The DFS method is obtained by adding a single line to existing DFS code.

5.6.4 Connected components

The number of connected components of an undirected graph is found by counting how many times the traversal is started. A new DFS (or BFS) is started from every unvisited vertex; each start traverses one new component completely.

```
int countComponents() {
  boolean[] visited = new boolean[n];
  int count = 0;
  for (int v = 0; v < n; v++)
    if (!visited[v]) {   /* new component */
      dfs(v, visited);
      count++;
    }
  return count;
}
```

Because all starts share the same `visited` array, each vertex is visited only once and the total time is again $O(V + E)$. On a graph with nine vertices and the edges 0–1, 1–2, 3–4, 5–6, 6–7, 7–5, the method returns 4: the components are $\{0, 1, 2\}$, $\{3, 4\}$, $\{5, 6, 7\}$, and the lone $\{8\}$. If a component number is written to the vertices visited in each start, the question "are two vertices in the same component?" can later be answered in $O(1)$ time.

The same count does not give the number of components in a directed graph. In a directed graph the set visited by a DFS depends on the choice of the start vertex. Take a graph with six vertices and the edges $0 \to 1$, $2 \to 0$, $3 \to 2$, $5 \to 4$. If vertices are tried from 0 to 5, DFS is started five times: from 0 only 1 is reached, and 2, 3, 4, and 5 are started separately. If vertices are tried from 5 down to 0, two starts suffice: from 5 to 4, and from 3 to 2, 0, and 1. Yet this graph has 2 components if edge directions are ignored and 6 components in the sense of strongly connected components. In a directed graph the right tool is Tarjan's or Kosaraju's algorithm.

5.6.5 The bipartiteness test

To test whether a graph is bipartite is to try to color its vertices with two colors so that adjacent vertices get different colors. BFS does this coloring naturally: the start gets one color, its neighbors the other, and their new neighbors the first color again. In other words, even layers get one color and odd layers the other.

> **Theorem 5.5 (Bipartiteness)**
>
> A graph is bipartite if and only if it contains no cycle of odd length.

Proof. In a bipartite graph every edge changes color. A cycle returns to the vertex where it started, that is, to the color it started with; so the cycle has an even number of color changes, that is, an even number of edges. For the converse, take a connected graph with no odd cycle and use the BFS layers. If there were an edge between two vertices u and w in the same layer, the BFS tree paths from s to u and to w would have equal length from the common ancestor where they part. These two paths and the edge u–w together would form an odd cycle. Hence every edge joins two consecutive layers, and the even/odd layer coloring is valid. In a graph that is not connected the same reasoning is applied to each component separately. ■

```
boolean isBipartite(int s) {
  Arrays.fill(color, -1);
  color[s] = 0; queue.add(s);
  while (!queue.isEmpty()) {
    int v = queue.poll();
    for (int w : adj[v])
      if (color[w] == -1) {
        color[w] = 1 - color[v];
        queue.add(w);
      } else if (color[w] == color[v])
        return false;  /* same color */
  }
  return true;
}
```

`color[v] == -1` marks an uncolored vertex. An uncolored neighbor is given the opposite color with `1 - color[v]` and added to the queue; the color array here also does the job of the visited array. If a colored neighbor has the same color as v, the two endpoints of an edge have fallen into the same set, and the method returns `false` at once. If the queue empties without trouble, the coloring is valid and `true` is returned. The method tests only the component of s; for a whole graph it is restarted from every uncolored vertex.

The graph in the trace is undirected with six vertices: the edges are 0–1, 0–2, 1–3, 2–3, 3–4, 3–5, 4–5. The graph has the cycle 0–1–3–2 (four edges) and the triangle 3–4–5 (three edges).

Taken (color)	Neighbors	`color` (after)	Queue (after)
—	0 enters the queue with color 0	0 −1 −1 −1 −1 −1	0
0 (0)	1, 2 uncolored → color 1	0 1 1 −1 −1 −1	1, 2
1 (1)	0 has the opposite color; 3 → color 0	0 1 1 0 −1 −1	2, 3
2 (1)	0 and 3 have the opposite color	same	3
3 (0)	1, 2 have the opposite color; 4, 5 → color 1	0 1 1 0 1 1	4, 5
4 (1)	3 has the opposite color; **5 has the same color** → `false`	0 1 1 0 1 1	5

Table 5.17: Trace of `isBipartite(0)`. The array `color` is indexed from 0 to 5.

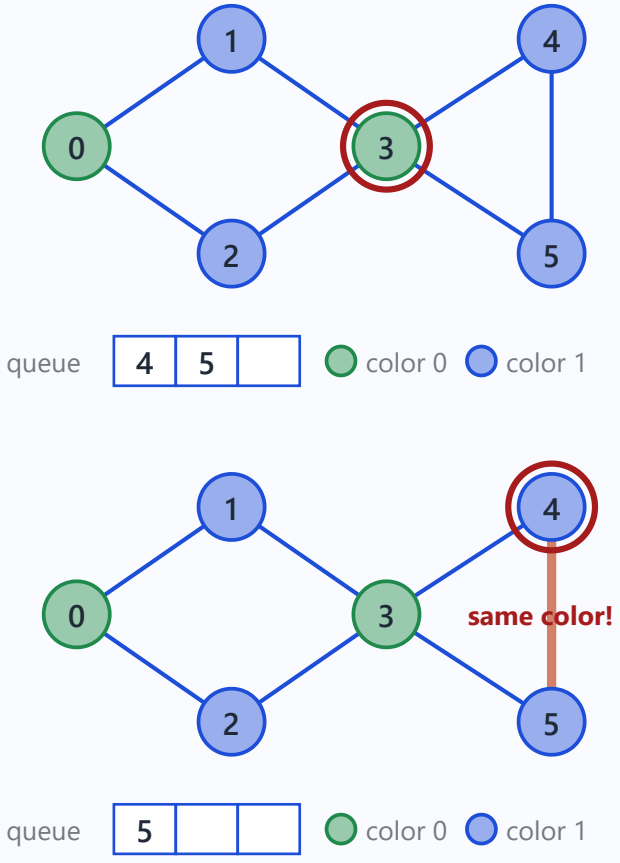


Figure 5.22: Left: 3 has been removed and its neighbors 4 and 5 colored with the opposite color. Right: 4 has been removed; its neighbor 5 is of the same color. The two endpoints of the edge 4–5 have fallen into the same set; the triangle 3–4–5 is a cycle of odd length.

The cycle 0–1–3–2 has four edges and does not spoil the coloring: 0 and 3 get one color, 1 and 2 the other. The problem is in the triangle 3–4–5; a cycle of three edges cannot be colored with two colors. Had the edge 4–5 been deleted, the method would have returned `true`. Because the test is a BFS, it runs in $O(V + E)$ time.

Example 5.6

Is the graph with the edges A–B, A–C, B–D, C–D, C–E, D–F, E–F, F–G bipartite?

Solution: the BFS distances from A are A 0, B 1, C 1, D 2, E 2, F 3, G 4. The vertices at even distance are *{A, D, E, G}* and those at odd distance *{B, C, F}*. Each of the eight edges joins these two sets; for example, the edge C–D joins layers 1 and 2, and the edge E–F joins layers 2 and 3. No two neighbors lie in the same layer, so the graph is bipartite. The cycles of the graph are the four-edge A–B–D–C–A and C–D–F–E–C and the six-edge A–B–D–F–E–C–A; all have even length.

5.6.6 Cycles, bridges, and articulation points

Cycle detection is done differently in the two kinds of graph. In an undirected graph, if DFS meets a visited neighbor that is not the parent of the current vertex, there is a cycle. The parent exception is necessary because an undirected edge is listed in both directions, and the edge from a child back to its parent is not a cycle. In a directed graph only an edge to a vertex on the call stack, that is, a yellow one, indicates a cycle; an edge to a finished vertex does not. In the DFS trace the edge $2 \to 0$ went to the yellow 0 and signaled a cycle; the edge $4 \to 6$ went to the finished 6 and did not.

In a connected undirected graph, an edge whose deletion splits the graph in two is a **bridge**, and a vertex whose deletion splits it is an **articulation point**. Bridges and articulation points in a network are the weak spots where a single failure can split it. Both are found in a single DFS with Tarjan's low-link idea. In an undirected graph, *disc(v)* is the discovery order of *v*, and *low(w)* is the smallest discovery value reachable from the subtree of *w* using a single back edge. For the edge *v*–*w* of the DFS tree (*w* the child):

- if $low(w) > disc(v)$, then *v*–*w* is a bridge: there is no back edge from the subtree of *w* to *v* or to the ancestors of *v*, and when the edge is deleted the subtree breaks off.
- if $low(w) \geq disc(v)$, then *v* is an articulation point (the root excepted; for the root the condition is that it has two or more children). In this case the subtree can return at most to *v*; when *v* is deleted it again breaks off.

In the example graph with letters A–G the only bridge is F–G. Since the DFS order is A, B, D, C, E, F, G, the discovery values are A 1, B 2, D 3, C 4, E 5, F 6, G 7. G's only neighbor is F, so $low(G) = 7 > disc(F) = 6$ and F–G is a bridge. Every other edge lies on a cycle; for example, the edge E–F is part of the cycle C–D–F–E, and even if it is deleted one can still go from E to F through C and D. F is also an articulation point: if it is deleted, G is left alone.

5.6.7 Iterative deepening and random walks

Iterative deepening DFS (IDDFS) performs a series of DFS runs with the depth limit raised through 0, 1, 2, ... In each round DFS goes no deeper than the limit. If the goal is first found in the round with limit *d*, the shallowest solution is at depth *d*. The method combines the low memory use of DFS with the guarantee of BFS that it finds the shallowest solution. At first sight, re-traversing the upper layers in every round looks wasteful. But the great majority of vertices are in the deepest layer. In a tree in which every vertex has 10 children, BFS visits 111,111 vertices down to depth 5. IDDFS makes 123,456 visits in total; the difference is only about eleven percent. The method is common in the huge implicit graphs searched by chess and puzzle solvers.

A **random walk** moves at each step to a randomly chosen neighbor of the current vertex. It is not a traversal, because there is no guarantee that it visits all vertices, but it is a cheap way of sampling very large networks. The PageRank of a web page rests on the probability that someone walking randomly for a long time on the web graph is at that page.

5.7 Common mistakes

Common mistake

Forgetting the visited array. In a graph with cycles, without a visited array DFS recurses forever and BFS keeps an endless queue. This is the source of the misconception that "BFS does not work on graphs with cycles". With marking in place, both algorithms work without trouble on graphs with cycles.

Common mistake

Marking at removal in BFS. The same vertex is added to the queue from many neighbors. The result may stay correct, but the queue grows to $O(E)$. The right way is to mark the vertex when adding it to the queue.

Common mistake

Expecting shortest paths from DFS. The path by which DFS first reaches a target is not a shortest path. In an unweighted graph BFS is needed, in a weighted graph Dijkstra's algorithm. BFS does not find shortest paths in a weighted graph either; it finds only the path with the fewest edges.

Common mistake

Assuming stack-based DFS has the same order as the recursive version. When neighbors are pushed in list order, the visit order reverses: on the directed example, stack-based DFS prints `0 7 2 …` and recursive DFS prints `0 1 2 …`. If the recursive order is wanted, the neighbors are pushed in reverse order.

Common mistake

Skipping the "on the stack?" test in Tarjan's algorithm. The update `min(…, index(w))` is made only for a *w* on the stack; a *w* whose component has been found is ignored. Otherwise the roots are lost and some components are never popped.

Common mistake

Traversing a disconnected graph from a single source. A traversal from one source sees only the vertices reachable from it. To cover the whole graph the traversal is restarted from every unvisited vertex. For the same reason, when a path is built after BFS, `parent[]` must hold *−1* for unreachable targets.

Common mistake

Overlooking the representation. The $O(V + E)$ time of DFS and BFS is for the adjacency list. If the same code is written with an adjacency matrix, each vertex's row is scanned end to end and the time becomes $O(V^2)$. On a sparse graph this difference can be more than a hundredfold.

5.8 Summary

- **Graph:** $G = (V, E)$, vertices and the edges joining them. The main kinds are directed, undirected, weighted, bipartite, cyclic, and complete graphs and trees. A complete graph on n vertices has $n(n - 1)/2$ edges; in an undirected graph the degrees sum to $2E$.
- **Representation:** the adjacency list needs $O(V + E)$ memory, the adjacency matrix $O(V^2)$. Traversal takes $O(V + E)$ time with lists and $O(V^2)$ with a matrix. For sparse graphs the list is preferred.
- **Traversal:** visit every reachable vertex exactly once. Because of cycles a visited array is essential. If the waiting list is a stack the result is DFS; if it is a queue, BFS.

- **DFS:** follows one branch to its end, then backs up. In the recursive version the call stack does the backing up. The visit order depends on the order of neighbors; the edges by which vertices are first reached form the DFS tree.
- **Stack-based DFS:** marks at pop time; a vertex can enter the stack more than once, and the stack can grow to $O(E)$. If neighbors are pushed in list order, the order is the reverse of the recursive version. It removes the risk of `StackOverflowError` on deep graphs.
- **BFS:** advances layer by layer with a queue; it marks a vertex when adding it to the queue. With the arrays `dist[]` and `parent[]` it finds shortest paths in an unweighted graph; the path is built by following parents back from the target. It does not find shortest paths in a weighted graph.
- **Maze:** open cells are vertices and adjacent open cells are edges. BFS tries all paths simultaneously, and the path it finds when it first reaches the exit is the shortest.
- **Strongly connected component:** a maximal set of mutually reachable vertices. The components partition the vertices into disjoint parts; the component graph is acyclic.
- **Tarjan:** the index is the discovery order; the low-link is the smallest index reachable from the subtree by reaching vertices on the stack. If $low(v) = index(v)$, then v is a root, and the vertices popped from the stack down to v form a component. An edge to a vertex whose component has been found is ignored. The time is $O(V + E)$; the components come out in reverse topological order.
- **Kosaraju:** first a DFS that records the finishing order on a stack, then a second DFS on the transpose in stack order. Each DFS of the second pass finds one component. The time is again $O(V + E)$.
- **Applications:** topological order (Kahn or reverse finishing order; if fewer than V vertices are written, there is a cycle), number of connected components, the bipartiteness test by two-coloring with BFS, cycle detection, bridges and articulation points, iterative deepening.

5.9 Key terms

graph vertex edge directed graph weighted graph bipartite graph adjacency list

adjacency matrix graph traversal depth-first search (DFS) breadth-first search (BFS) pseudocode

backtracking back edge ancestor parent shortest path connected component

strongly connected component (SCC) topological sort

5.10 Test Yourself

1. An undirected graph has the edges 0–1, 0–4, 1–2, 1–4, 2–3, 3–4, 3–5; the adjacency lists are in increasing order. What do `dfs(0)` and `bfs(0)` print? According to BFS, which is the shortest

path from 0 to 5?

Answer

The adjacency lists are 0: [1, 4], 1: [0, 2, 4], 2: [1, 3], 3: [2, 4, 5], 4: [0, 1, 3], 5: [3]. DFS goes from 0 down to 1, 2, 3; since the neighbor 2 of 3 is marked, it moves on to 4. All neighbors of 4 are marked, so DFS returns to 3 and visits 5: `0 1 2 3 4 5`. BFS discovers 1 and 4 first, then 2 from 1, 3 from 4, and last 5 from 3: `0 1 4 2 3 5`. The distances are 0, 1, 2, 2, 1, 3. Vertex 3 is discovered by 4, not 2, because 4 leaves the queue before 2. Following the parents gives the shortest path 0–4–3–5 (3 edges).

2. Take the directed graph with the edges *0→1*, *0→2*, *1→3*, *2→3*, *3→4*, *2→4*; the adjacency lists are filled in this edge order. What does `dfsStack(0)` print, and how many pushes does it make? What does recursive `dfs(0)` print?

Answer

`adj[0] = [1, 2]`, `adj[2] = [3, 4]`. Stack-based DFS pops, after 0, the last pushed vertex 2; after 2 the last pushed 4, then 3 (the neighbor 4 of 3 is pushed but is marked), and last 1: `0 2 4 3 1`. The total number of pushes is *1 + E = 7*. Recursive DFS prints `0 1 3 4 2`. If the stack version pushes neighbors in reverse order, it also prints `0 1 3 4 2`.

3. An undirected graph on seven vertices has the edges 0–1, 0–2, 1–3, 2–3, 3–4, 5–6. After `bfsDist(0)`, what is `dist[]`? What do `path(parent, 0, 4)` and `path(parent, 0, 6)` return?

Answer

`dist = [0, 1, 1, 2, 3, -1, -1]`; 5 and 6 are unreachable. Since `parent[4] = 3`, `parent[3] = 1`, `parent[1] = 0`, `path(parent, 0, 4)` returns `[0, 1, 3, 4]`. Since `parent[6] = -1`, the loop adds only 6; the first item of the list is not 0 and the method returns `null`. Had `parent[]` not been filled with *−1*, `parent[6] = 0` would remain and the method would return the nonexistent path `[0, 6]`.

4. Take the directed graph with the edges *0→1*, *1→2*, *2→0*, *2→3*, *3→4*, *4→5*, *5→3*, *4→6* (`adj[2] = [0, 3]`, `adj[4] = [5, 6]`). Which lines does `scc(0)` print? What is the final `index/low` value of each vertex?

Answer

Vertices 0, 1, 2, 3, 4, 5, 6 are discovered in order, with indices 1 to 7. The edge 2 → 0 lowers `low[2]` to 1, and the edge 5 → 3 lowers `low[5]` to 4. Vertex 6 has no outgoing edges and is the first root. Then 3 becomes a root, and last 0. The output is three lines: `6`, `5 4 3`, `2 1 0`. The final values are 0: 1/1, 1: 2/1, 2: 3/1, 3: 4/4, 4: 5/4, 5: 6/4, 6: 7/7.

5. For the DAG *0→2*, *1→2*, *1→3*, *2→4*, *3→4*, *4→5*, *3→5*, what order do Kahn's algorithm and reverse finishing order give? What does Kahn print if the edge *5→1* is added to the graph?

   Answer

   At the start 0 and 1 have in-degree 0. Kahn prints `0 1 2 3 4 5`: after 0 the degree of 2 drops to 1, after 1 vertices 2 and 3 become ready, after 3 vertex 4, and after 4 vertex 5 become ready. In DFS the finishing order is 5, 4, 2, 0, 3, 1; its reverse is `1 3 0 2 4 5`. When *5→1* is added, 1 gets in-degree 1. The queue holds only 0; when 0 is written, the degree of 2 drops to 1 and the queue empties. Only `0` is written; since fewer than 6 vertices are written, the graph has a cycle (*1 → 3 → 5 → 1*).

6. Are a ring on five vertices (0–1–2–3–4–0) and a ring on six vertices (0–1–2–3–4–5–0) bipartite? On which edge does `isBipartite(0)` return `false` for the first one?

   Answer

   The six-vertex ring is bipartite: the colors alternate 0, 1, 0, 1, 0, 1. The five-vertex ring is a cycle of odd length and is not bipartite. BFS colors 0 with color 0, 1 and 4 with color 1, and 2 and 3 with color 0. When 2 is taken from the queue, its neighbor 3 is also of color 0: the method returns `false` at the edge 2–3.

7. BFS is run from 0 on the complete graph with five vertices. How many insertions into the queue does the correct version, which marks on insertion, make, and how many does the version that marks on removal make?

   Answer

   The correct version adds each vertex once: 5 insertions. In the version that marks on removal, when 0 is removed 1, 2, 3, 4 are added; when 1 is removed, the still unmarked 2, 3, 4 are added again; when 2 is removed, 3 and 4; when 3 is removed, 4. In all *1 + 4 + 3 + 2 + 1 = 11* insertions are made and the queue reaches 7 items. The visit order is still 0 1 2 3 4, but work and memory have been wasted.

8. In an undirected graph with $V = 500$ vertices and $E = 2000$ edges, what is the total number of items in the adjacency lists and the number of cells in the adjacency matrix? By what factor does the approximate work of BFS differ between the two representations?

   Answer

   The lists hold $2E = 4000$ items, the matrix has $V^2 = 250{,}000$ cells. BFS takes about $V + 2E = 4500$ steps with lists and $V^2 = 250{,}000$ steps with the matrix; the ratio is about 56.

5.11 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 20: presents BFS and DFS with white–gray–black coloring, discovery and finishing times, edge classification, and correctness proofs; topological sorting and Kosaraju's algorithm are there as well.
- R. Sedgewick, K. Wayne, *Algorithms* (4th ed., 2011), Sections 4.1 and 4.2: explains traversal of undirected and directed graphs with Java code; contains worked examples for mazes, components, and the Kosaraju–Sharir algorithm.
- J. Kleinberg, É. Tardos, *Algorithm Design* (2006), Chapter 3: treats BFS layers, the bipartiteness test, and topological sorting in DAGs with short, readable proofs.
- R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms" (1972): the original paper that showed DFS to be the basis of linear-time graph algorithms and introduced the low-link idea.

5.12 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 20: Elementary Graph Algorithms).
2. A. B. Kahn, "Topological Sorting of Large Networks", *Communications of the ACM* 5(11), 1962.
3. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2006 (Chapter 3: Graphs).
4. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Sections 4.1–4.2).
5. R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms", *SIAM Journal on Computing* 1(2), 1972.

Shortest Paths

Finding the shortest route between two points on a map, getting a packet across a network by the cheapest path, and moving a game character around obstacles to a goal are different faces of the same problem. All of them look for a shortest path in a weighted graph. These notes begin by defining the problem and relaxation, the operation common to all the algorithms. They then treat four algorithms in the same order. Dijkstra's algorithm proceeds by finalizing the nearest vertex, and works when no weight is negative. The Bellman-Ford algorithm relaxes all edges again and again; it handles negative edges and detects negative cycles. The Floyd-Warshall algorithm computes the distances between all pairs of vertices with three nested loops. A* search estimates the distance to the goal and so steers the search toward it. Each algorithm is run step by step on example graphs, and its Java code is followed line by line. The notes close with breadth-first search for unweighted graphs and a comparison of the algorithms.

LEARNING OUTCOMES

Concepts

Tell single-source, all-pairs, and single-pair shortest path problems apart; explain relaxation, the distance estimate (d), and the predecessor array (π).

Dijkstra and Bellman-Ford

Run both algorithms step by step on example graphs, fill in the distance and predecessor tables, follow their Java code, and state their complexity with reasons.

Negative edges and cycles

Show with an example why Dijkstra's algorithm fails on a negative edge; detect negative cycles with Bellman-Ford and Floyd-Warshall.

Floyd-Warshall and A*

Update the matrices D and Π for all pairs in intermediate-vertex order. Apply A* search with $f = g + h$, explain what it means for a heuristic to be admissible and consistent, and choose an algorithm according to the type of problem.

6.1 The shortest path problem

Graph traversal algorithms tell us which vertices can be reached from a vertex. In real applications that is often not enough: the cheapest way to get there is wanted as well. Once numbers such as length, time, or cost are attached to the edges, the question is no longer "can it be reached?" but "what is the least it costs to reach?" This section defines the problem, separates its kinds, and describes relaxation, the common operation on which all the algorithms are built.

6.1.1 A road network is a graph

The road map of a city is a natural graph. Intersections are **vertices** and the roads joining them are **edges**. Each road has a length or a travel time; this number is the **weight** of the edge. A graph whose edges carry weights is a **weighted graph**. If there are one-way streets, the direction of the edges matters too; the graph is then directed.

A **path** in a graph is a sequence of vertices joined by consecutive edges. The length of a path is the sum of the edge weights on it. In this definition what counts is not the number of edges but the sum of the weights. A path of two edges can be longer than a path of five edges.

Definition 6.1

Let $G = (V, E)$ be a weighted graph and $w(u, v)$ the weight of the edge (u, v). The **length** of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum $w(p) = w(v_0, v_1) + w(v_1, v_2) + \cdots + w(v_{k-1}, v_k)$.

A **shortest path** from s to v is a path of least length among the paths between these two vertices. This least length is the **distance** from s to v, written $\delta(s, v)$. If there is no path at all from s to v, we take $\delta(s, v) = \infty$.

Figure 6.1 shows a small road network. There are several paths from Home to School. The path Home–1–2–4–School has four edges and length $4 + 2 + 3 + 3 = 12$. The path Home–1–4–School has only three edges, but because the edge 1–4 costs 6 units its length is $4 + 6 + 3 = 13$. The path Home–3–4–School also has three edges and also comes to 13. The shortest path is not the one with the fewest edges but the four-edge path Home–1–2–4–School: $\delta(\text{Home}, \text{School}) = 12$.

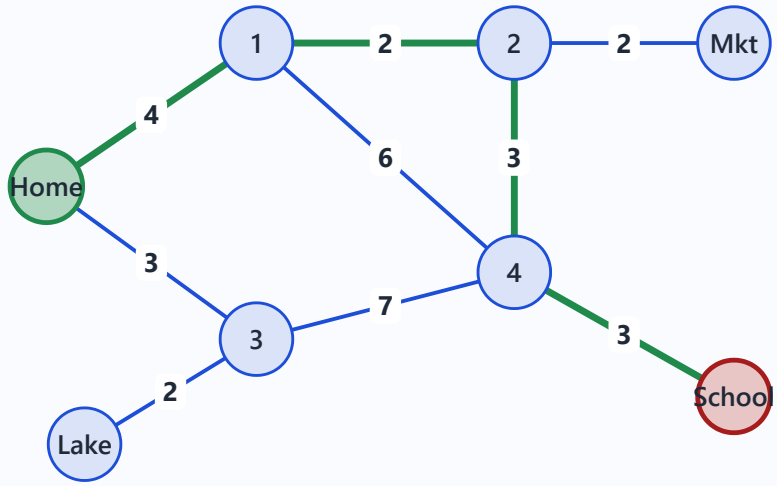


Figure 6.1: A small road network. Vertices are intersections, edges are roads, and the numbers are road lengths. The green edges show the shortest path (12) from Home to School. The path Home–1–4–School, with fewer edges, comes to 13.

The example shows the heart of the problem. In an unweighted graph the shortest path is the path with the fewest edges, and breadth-first search finds it directly. In a weighted graph, counting edges is not enough; the sums of the weights must be compared. This is why separate algorithms are needed for weighted graphs.

The shortest path problem turns up in many places in everyday technology. Navigation apps compute the shortest or fastest route in a road network. Internet routers decide to which neighbor to forward a packet by a shortest-path computation. The OSPF (Open Shortest Path First) protocol uses Dijkstra's algorithm, and the RIP (Routing Information Protocol) protocol uses a distributed form of Bellman-Ford. In games, characters walking around obstacles are handled by A* search. In logistics, vehicle delivery routes also rest on shortest-path computations.

Common mistake

Taking the shortest path in a weighted graph to be the path with the fewest edges. The path Home–1–4–School in Figure 6.1 has three edges but length 13; the four-edge path Home–1–2–4–School comes to 12. The number of edges decides only when all weights are equal.

6.1.2 Kinds of problems

The request "find the shortest path" can come in three different forms. Which form is wanted also determines the algorithm to use.

- **Single-source shortest paths:** the distances from a source *s* to all vertices of the graph are wanted. The result is an array holding one number per vertex. Dijkstra, Bellman-Ford, and breadth-first search on an unweighted graph solve this problem.
- **All-pairs shortest paths:** the distance is wanted for every pair of vertices *(u, v)*. The result is a matrix of size $n \times n$. The Floyd-Warshall algorithm is designed for this problem.
- **Single-pair shortest path:** only the shortest path from one start *s* to one goal *t* is wanted. A single-source algorithm can do this job too, but it expands more vertices than necessary. A* search knows the goal and steers the search toward it.

Besides the kind of problem, properties of the graph also affect the choice. The most important property is whether there are edges with negative weight. When weights are thought of as costs, a negative value may seem strange. But with edges that carry a gain or a refund, negative weights are natural: a transfer that gives a discount in a transport network, a transaction that earns a profit in a financial graph. Table 6.1 lists the algorithms of this chapter with what they find and their restrictions.

Algorithm	What it finds	Restriction / note
Dijkstra	from the source to all vertices	edge weights must not be negative
Bellman-Ford	from the source to all vertices	works with negative edges, detects negative cycles; slower than Dijkstra
Floyd-Warshall	all pairs	works with negative edges; detects a negative cycle from the diagonal
A*	from start to goal	speeds up the search with a heuristic estimate
BFS	from the source to all vertices	only unweighted (or equal-weight) graphs

Table 6.1: Problems solved by the shortest-path algorithms, and their restrictions.

6.1.3 The basic operation: relaxation

All the algorithms in this chapter work on the same idea. For each vertex, the length of the best path found so far is kept as an estimate. As the algorithm proceeds, shorter paths are found and the estimates shrink. In the end every estimate equals the true distance. Two pieces of information are stored for each vertex:

- *d[v]*: the length of the shortest path from the source to *v* found so far. At the start it is 0 at the source and ∞ at all other vertices. ∞ means "no path found yet".
- *π[v]*: the vertex that comes immediately before *v* on this path, called the **predecessor**. At the start it is undefined and is written "/" or "–".

The only operation that shrinks the estimates is **relaxation**. When an edge *(u, v)* is relaxed, the question asked is: if we take the known path from the source to *u* and append the edge *(u, v)*, do we get a path to *v* shorter than the one we know? If so, the estimate and the predecessor of *v* are

updated.

Definition 6.2

Relaxing the edge (u, v): if $d[u] + w(u, v) < d[v]$, set $d[v] \leftarrow d[u] + w(u, v)$ and $\pi[v] \leftarrow u$; otherwise nothing changes.

```
RELAX(u, v, w):
  if d[u] + w(u, v) < d[v]:
    d[v] = d[u] + w(u, v)
    π[v] = u
```

The pseudocode is the definition itself. The second line asks whether the path through u is shorter. If it is, the third line writes the new length and the fourth records that the path now comes from u. The operation consists of one comparison and at most two assignments; it takes $O(1)$ time.

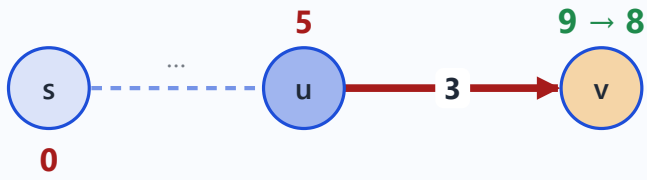


Figure 6.2: One relaxation step. The known path from the source to u is 5 and the edge (u, v) is 3 units. The estimate of v was 9; since $5 + 3 = 8 < 9$, the estimate drops to 8 and $\pi[v] = u$.

Example 6.1

In the situation of Figure 6.2 the edge (u, v) is relaxed twice in a row. Find the result of each relaxation.

1. **First relaxation.** $d[u] = 5$, $w(u, v) = 3$, $d[v] = 9$. The candidate length is $5 + 3 = 8$. Since $8 < 9$, $d[v] = 8$ and $\pi[v] = u$.
2. **Second relaxation.** The candidate length is again 8, but now $d[v] = 8$. Since $8 < 8$ is false, nothing changes.
3. **Result.** Relaxing the same edge again does no harm; only a comparison is wasted. Unless $d[u]$ shrinks, the second relaxation gains nothing.

Relaxation has two important properties. First, $d[v]$ can only decrease; no relaxation ever increases an estimate. Second, $d[v]$ never falls below the true distance $\delta(s, v)$. The reason is simple: every value written to $d[v]$ is the length of a path from the source to v that really exists. Since no path can be shorter than a shortest path, the estimate cannot drop below the distance either. The estimate comes down from above and stops when it reaches the right value.

The difference between the algorithms lies not in relaxation but in *the order* in which and *the number of times* the edges are relaxed. Dijkstra relaxes each edge once, in a cleverly chosen order. Bellman-Ford relaxes all edges again and again without regard to order. Floyd-Warshall does the same operation on pairs of vertices, adding intermediate vertices. A* replaces Dijkstra's order with an estimate of the distance to the goal.

> **Common mistake**
>
> Updating only $d[v]$ in relaxation and forgetting $\pi[v]$. The distances come out right, but the path itself cannot be rebuilt. If not just the length of the shortest path but also which vertices it passes through is wanted, the two assignments must be made together.

6.1.4 Properties of shortest paths

To understand why the algorithms work, we need four properties of shortest paths. These properties are used both in designing the algorithms and in proving them correct.

Optimal substructure. Every part of a shortest path is itself a shortest path. If the path $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path from s to v, then its part $s \rightsquigarrow u$ is a shortest path from s to u. The proof is a short contradiction. If there were a shorter path from s to u, the old part could be removed and this path put in its place. That would give a shorter path to v as well, contradicting the choice of a shortest path. **Optimal substructure** is the common basis of dynamic programming and greedy algorithms. Bellman-Ford and Floyd-Warshall use this property through dynamic programming, and Dijkstra through a greedy choice.

Triangle inequality. For every edge (u, v), $\delta(s, v) \le \delta(s, u) + w(u, v)$ holds. The right side of the inequality is the length of one particular path: go to u by a shortest path and use the edge (u, v). A shortest path cannot be longer than this path. Relaxation finds the edges where this inequality is violated by the estimates and repairs them: every edge with $d[v] > d[u] + w(u, v)$ is an opportunity for repair.

Negative cycle. A cycle whose total weight is negative is a **negative cycle**. If such a cycle is reachable from the source, the path gets a little shorter each time the cycle is traversed. Since the path can be shortened as much as we like, the shortest path is undefined; the distance is written $-\infty$. This affects not only the vertices on the cycle but all vertices reached through the cycle.

Simple path. If there is no negative cycle, a shortest path contains no cycle. Removing a cycle of positive weight from a path shortens it, and removing a cycle of weight zero leaves the length unchanged. In either case a cycle-free shortest path exists. A path with no cycle passes through each vertex at most once. In a graph with V vertices such a path has at most V vertices, hence at most $V - 1$ edges. This is the reason Bellman-Ford runs $V - 1$ rounds.

Definition 6.3

If the edges $(\pi[v], v)$ are taken for every vertex $v \neq s$ reachable from the source, the result is a tree rooted at s. In this tree the unique path from the root to any vertex is a shortest path to that vertex. This tree is called the **shortest-path tree**.

The shortest-path tree gathers all shortest paths in a single structure. Since each vertex stores only its own predecessor, all the paths fit in a single array π of size V. To read a path, one starts at the target and follows the π pointers until the source is reached. The vertices obtained this way are in reverse order; to write the path from source to target the order is reversed.

Common mistake

Confusing a negative edge with a negative cycle. A negative edge by itself does not make the shortest path undefined; Bellman-Ford and Floyd-Warshall give correct results on such graphs. The problem is a cycle of negative total weight that is reachable from the source. In an undirected graph, however, even a single negative edge forms a cycle: the path $u \to v \to u$ carries twice the negative weight.

6.2 Dijkstra's algorithm

Dijkstra's algorithm is the most widely used solution of the single-source shortest path problem. Edsger W. Dijkstra devised it in 1956 and published it in 1959 in a short paper of three pages. The algorithm finds the shortest paths from the source to all other vertices. It has a single condition: none of the edge weights may be negative. When this condition holds it is fast, and it is usable in practice even on large road networks.

6.2.1 The greedy choice and the water wave

An analogy conveys the idea of the algorithm well. When a stone is thrown into a still lake, the wave spreads in all directions at the same speed. The wave first reaches the point nearest the stone, then the next nearest point. Let us think of the edges as water channels and of the weights as channel lengths. The wave leaving the source reaches the vertices in exactly the order of distance. The wave that first reaches a vertex has come by the shortest path; no wave arriving later could have arrived earlier.

Dijkstra's algorithm imitates the movement of this wave step by step. The vertices are divided into two groups. Vertices whose distance has been finalized are in the set S; the wave has already reached them. The other vertices have only an estimate. At each step, the vertex with the smallest estimate among the unfinalized ones is chosen and finalized. Then the edges leaving this vertex are relaxed, which updates the estimates of the neighbors.

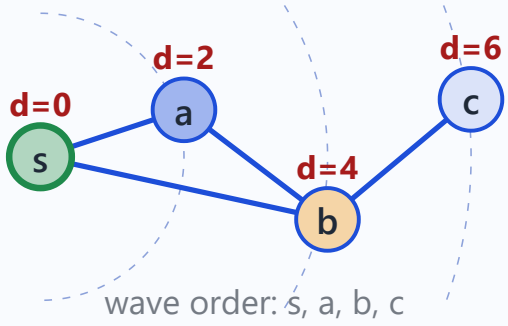


Figure 6.3: The water wave analogy. The wave leaving the source s reaches the vertices in order of distance: s (0), a (2), b (4), c (6). Dijkstra finalizes the vertices in this order.

An algorithm that at every step takes the option that "looks best right now" and never revises that decision is a **greedy algorithm**. Dijkstra is greedy: it accepts the distance of the chosen vertex as final and never touches it again. Greedy decisions give wrong results in most problems. In Dijkstra they are correct because the weights are nonnegative. Why this condition is sufficient, and what happens when it is violated, is shown at the end of the section.

6.2.2 The steps of the algorithm

The algorithm is summarized in five steps:

1. **Initialization.** The distance of the source is set to 0 and that of every other vertex to ∞. The predecessors are undefined and the set of finalized vertices is empty: $d[s] = 0$, $d[v] = \infty$, $\pi[v]$ = undefined, $S = \emptyset$.
2. **Selection.** Among the unfinalized vertices, the one u with the smallest d value is chosen and added to S. The selection is done by scanning an array or with a **priority queue**.
3. **Candidate distances.** For each not yet finalized neighbor v of u, the candidate distance $d[u] + w(u, v)$ is computed.
4. **Relaxation.** If the candidate distance is smaller than $d[v]$, then $d[v]$ and $\pi[v]$ are updated.
5. **Repeat.** Steps 2–4 are repeated until all vertices are finalized or all the remaining vertices have distance ∞. Vertices unreachable from the source keep distance ∞.

Each vertex is chosen once, and at the moment it is chosen its outgoing edges are relaxed once. In a directed graph each edge is considered once overall, in an undirected graph once from each end. All the work of the algorithm consists of these selections and relaxations.

6.2.3 Example 1: an undirected graph

The first example is an undirected graph on six vertices; the source is vertex 1. The edges and their weights are: 1–2 (7), 1–3 (9), 1–6 (14), 2–3 (10), 2–4 (15), 3–4 (11), 3–6 (2), 4–5 (6), 5–6 (9). Figure 6.4 shows the initial and final states of the graph side by side.

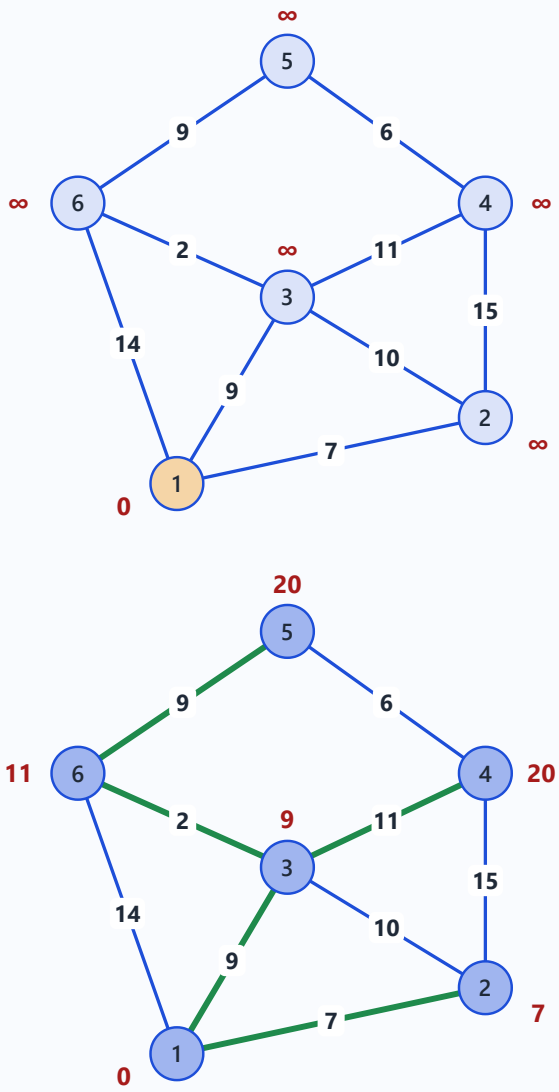


Figure 6.4: Dijkstra Example 1. Left, the start: the source 1 has distance 0, the others ∞. Right, the end: all vertices are finalized, the red numbers are distances, and the green edges form the shortest-path tree.

The algorithm runs six rounds; in each round one vertex is finalized. Table 6.2 gives the distance estimates after each round. Bold values are the estimates that changed in that round.

Round	Chosen (d)	Relaxations	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]
start	—	—	0	∞	∞	∞	∞	∞
1	1 (0)	2: ∞ → 7; 3: ∞ → 9; 6: ∞ → 14	0	**7**	**9**	∞	∞	**14**

Round	Chosen (d)	Relaxations	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]
2	2 (7)	3: 17 ≥ 9, unchanged; 4: ∞ → 22	0	7	9	**22**	∞	14
3	3 (9)	4: 22 → 20; 6: 14 → 11	0	7	9	**20**	∞	**11**
4	6 (11)	5: ∞ → 20	0	7	9	20	**20**	11
5	4 (20)	5: 26 ≥ 20, unchanged	0	7	9	20	20	11
6	5 (20)	no unfinalized neighbor	0	7	9	20	20	11

Table 6.2: Round-by-round trace of Dijkstra Example 1 (source 1). Each row is the state after the edges of the chosen vertex have been relaxed.

The rounds in detail. In the first round only the distance of the source is finite; vertex 1 is chosen. Vertex 1 has three neighbors, all with estimate ∞. The edge weights become the new estimates directly: $d[2] = 7$, $d[3] = 9$, $d[6] = 14$. The predecessor of all three is 1.

In the second round the smallest estimate among the unfinalized vertices is 7; vertex 2 is chosen. Vertex 1 is skipped because it is already finalized. The edge 2–3 proposes $7 + 10 = 17$; this is larger than 9, so the estimate of 3 is unchanged. The edge 2–4 proposes $7 + 15 = 22$, and a path to 4 is found for the first time: $d[4] = 22$, $\pi[4] = 2$.

In the third round 3 is chosen ($d = 9$). This round is the most instructive step of the example. The edge 3–4 proposes $9 + 11 = 20$, which is smaller than the 22 found in the previous round. The estimate of 4 therefore drops to 20 and its predecessor changes from 2 to 3. The edge 3–6 proposes $9 + 2 = 11$ and lowers the estimate of 6 from 14. The direct edge from 1 to 6 costs 14; going around through 3 costs 11. A direct edge is not always the shortest path.

In the fourth round the smallest estimate is 11; vertex 6 is chosen. The edge 6–5 gives the first path to 5: $11 + 9 = 20$. In the fifth round the estimates of 4 and 5 are equal (20). In a tie the lower-numbered vertex was chosen; 4 is finalized. The edge 4–5 proposes $20 + 6 = 26$, which cannot beat 20. In the last round 5 is chosen, and since it has no unfinalized neighbor the algorithm ends.

The final distances are 2: 7, 3: 9, 6: 11, 4: 20, 5: 20. The predecessor array is $\pi[2] = 1$, $\pi[3] = 1$, $\pi[4] = 3$, $\pi[5] = 6$, $\pi[6] = 3$. These five edges form the shortest-path tree. The shortest path to 5, for example, is found by following the pointers backward: $5 \leftarrow 6 \leftarrow 3 \leftarrow 1$. The path is $1 \rightarrow 3 \rightarrow 6 \rightarrow 5$, of length $9 + 2 + 9 = 20$. The order in which the vertices are finalized is 1, 2, 3, 6, 4, 5; this is the same as the order of the distances from smallest to largest.

Example 6.2

What are the distances if vertex 5 is taken as the source in the same graph?

1. **5 is chosen (0).** Neighbors: 4: ∞ → *6*, 6: ∞ → *9*.
2. **4 is chosen (6).** 2: $6 + 15 = 21$, 3: $6 + 11 = 17$.
3. **6 is chosen (9).** 1: $9 + 14 = 23$; 3: $9 + 2 = 11 < 17$, the estimate drops to 11.

4. **3 is chosen (11).** 1: *11 + 9 = 20 < 23*, the estimate drops to 20; 2: *11 + 10 = 21*, equal, so unchanged.
5. **1 is chosen (20), then 2 (21).** There are no new relaxations.

Result: d = 1: 20, 2: 21, 3: 11, 4: 6, 5: 0, 6: 9. For example, the path to 3 is $5 \to 6 \to 3 = 9 + 2$ and the path to 2 is $5 \to 4 \to 2 = 6 + 15$.

6.2.4 Example 2: a directed graph, the sets Q and S

The second example is a directed graph on five vertices with source 5. The edges are: $5 \to 2$ (4), $5 \to 4$ (2), $4 \to 2$ (1), $4 \to 3$ (1), $3 \to 4$ (2), $2 \to 1$ (3), $1 \to 3$ (6), $1 \to 4$ (3). In a directed graph an edge is used only in the direction of the arrow. For example, the edge $2 \to 1$ allows going from 2 to 1 but not the reverse. The same graph will be used in the Bellman-Ford and Floyd-Warshall examples, so the results of the three algorithms can be compared.

This time the state of the algorithm is followed with two sets. Q is the priority queue of vertices not yet finalized, and keeps the vertices ordered by their d value. S is the list of finalized vertices. At each step the vertex at the head of Q is taken, added to S, and its edges are relaxed.

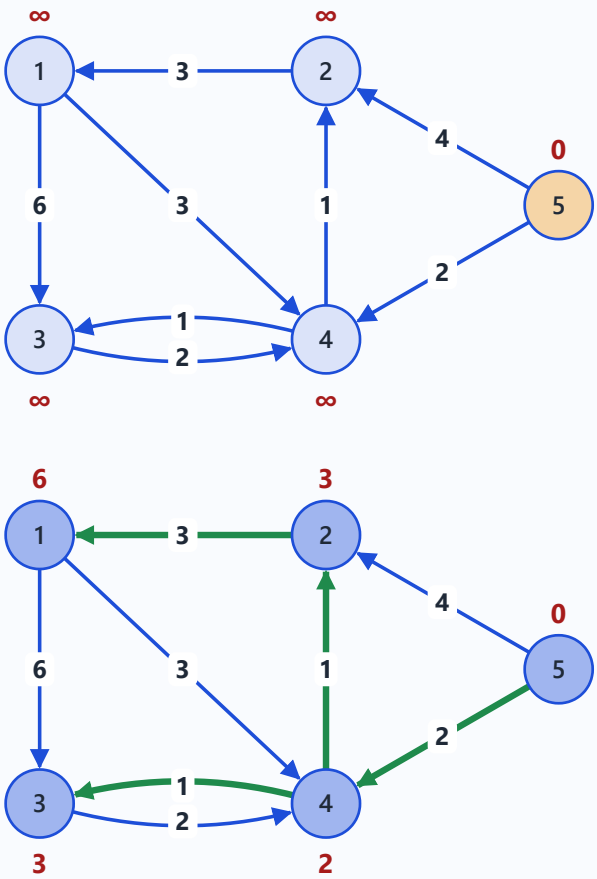


Figure 6.5: Dijkstra Example 2. Left, the directed graph and the initial distances (source 5). Right, the final

state: only the predecessor edges (green) remain and form the shortest-path tree.

Step	Taken from Q	Relaxations	Q (after the step)	S
start	—	—	5 (0), 1 (∞), 2 (∞), 3 (∞), 4 (∞)	∅
1	5, d = 0	(5, 2): ∞ → 4; (5, 4): ∞ → 2	4 (2, π 5), 2 (4, π 5), 1 (∞), 3 (∞)	5
2	4, d = 2	(4, 2): 4 → 3; (4, 3): ∞ → 3	2 (3, π 4), 3 (3, π 4), 1 (∞)	5, 4
3	2, d = 3	(2, 1): ∞ → 6	3 (3, π 4), 1 (6, π 2)	5, 4, 2
4	3, d = 3	(3, 4): 4 is already in S	1 (6, π 2)	5, 4, 2, 3
5	1, d = 6	(1, 3), (1, 4): both are in S	empty	5, 4, 2, 3, 1

Table 6.3: How the sets Q and S change in Dijkstra Example 2. The vertices in Q are written with their (d, π) values, smallest d first.

The second step needs attention. When 4 is finalized, the edge $4 \to 2$ proposes $2 + 1 = 3$. The estimate of 2 had become 4 in the first step by the direct edge; now it drops to 3. The direct edge from the source to 2 costs 4, while going around through 4 costs 3. In the third step the estimates of 2 and 3 are equal (3); the lower-numbered 2 is taken first. In the fourth and fifth steps there are no new relaxations, because the vertices at the ends of the edges are already finalized.

In the final state the distances and predecessors are as follows:

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Table 6.4: Result of Dijkstra Example 2 (source 5).

The green edges give two paths: $5 \to 4 \to 2 \to 1$ (length $2 + 1 + 3 = 6$) and $5 \to 4 \to 3$ (length $2 + 1 = 3$). The edge $5 \to 2$ does not enter the shortest-path tree, because there is a shorter path to 2 through 4.

6.2.5 Pseudocode and lazy deletion

The pseudocode of the algorithm with a priority queue is below. The queue holds *(vertex, distance)* pairs; the operation `pop()` returns the pair with the smallest distance.

```
DIJKSTRA(G, source):
  dist = [ ]; visited = [ ]
  pq = { }
  for each v in V:
    dist[v] = infinity
    visited[v] = 0
  dist[source] = 0
  pq.push(source, 0)
  loop while pq is not empty:
    u = pq.pop()   /* smallest dist */
    if visited[u] == 0:
      visited[u] = 1
      for each (u, v) in G[u]:
        if dist[u] + weight(u, v) < dist[v]:
          dist[v] = dist[u] + weight(u, v)
          pq.push(v, dist[v])
  return dist
```

The first six lines prepare the arrays: the distance of every vertex is infinity and the visited flag is 0. The distance of the source is set to 0 and the source is added to the queue. The main loop runs until the queue is empty. In each round the record with the smallest distance is removed. If the vertex has not been processed before, it is marked and its neighbors are relaxed. If the distance of a neighbor shrinks, the neighbor is added to the queue with its new distance.

One detail of the code stands out: when the distance of a vertex shrinks, its old record in the queue is not deleted; a new record is added. The same vertex can therefore appear in the queue several times. The old record carries a larger distance, so it leaves the queue later. By the time it comes out, the vertex has already been marked, and the `visited` check skips it. This technique is called **lazy deletion**.

The reason for lazy deletion is that Java's `PriorityQueue` class has no operation that decreases the priority of an item. Removing an arbitrary item from the queue also takes linear time. Adding a new record takes $O(\log n)$. The cost is that the queue grows a little: since every relaxation can add a record, at most E records accumulate in the queue. If the path is wanted as well, the assignment `prev[v] = u` is added to the relaxation.

Common mistake

Skipping the `visited` check in code with lazy deletion. When an old record comes out, the vertex is processed a second time and its neighbors are relaxed again needlessly. If the weights are nonnegative the result is still correct, but the work increases. The check ensures that each vertex is processed exactly once.

6.2.6 Java code and a trace

The Java counterpart of the pseudocode is below. The graph is kept in an adjacency list named `adj`: `adj.get(u)` is the list of edges leaving *u*. Each `Edge` object has the fields `to` and `weight`. The vertices are numbered from 1 to `V`; the arrays therefore have size `V + 1` and index 0 is not used. `INF` is a constant holding `Integer.MAX_VALUE`.

```
void dijkstra(int source) {
  int[] dist = new int[V + 1];
  int[] prev = new int[V + 1];
  boolean[] visited = new boolean[V + 1];
  Arrays.fill(dist, INF);
  dist[source] = 0;
  PriorityQueue<int[]> pq =
    new PriorityQueue<>((a, b) -> a[1] - b[1]);
  pq.add(new int[] { source, 0 });
  while (!pq.isEmpty()) {
    int u = pq.poll()[0];
    if (visited[u]) continue; /* stale entry */
    visited[u] = true;
    for (Edge e : adj.get(u)) {
      int v = e.to, w = e.weight;
      if (!visited[v]
          && dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        prev[v] = u;
        pq.add(new int[] { v, dist[v] });
      }
    }
  }
}
```

The code reads line by line as follows:

- **Lines 2–6:** the arrays `dist`, `prev`, and `visited` are created. `dist` is filled with infinity and that of the source is set to 0. A `boolean` array starts out `false` in Java by itself.
- **Lines 7–9:** the queue holds `int[]` pairs: `a[0]` is the vertex and `a[1]` its distance at that moment. The comparator `(a, b) -> a[1] - b[1]` orders the pairs by distance, so `poll()` gives the pair with the smallest distance. At the start the queue holds only `{source, 0}`.
- **Lines 11–13:** the smallest pair is removed and its vertex taken into `u`. If the vertex has been processed before, this is a stale entry and is skipped with `continue`. If not, it is marked; this is the moment the vertex is finalized.
- **Lines 14–21:** for every edge leaving `u`, the neighbor `v` and the weight `w` are taken. If the neighbor is not finalized and the path through `u` is shorter, relaxation is done: the distance and predecessor are updated and a new pair is added to the queue.

The subtraction in the comparator carries an overflow trap. If one of the distances were `INF`, `a[1] - b[1]` could overflow and give the wrong sign. In this code only finite distances are added to the queue, so no problem arises. A safer way to write it would be `Integer.compare(a[1], b[1])` or `Comparator.comparingInt(a -> a[1])`.

When the code is run as `dijkstra(5)` on the graph of Example 2, the queue changes as in Table 6.5. The queue column shows the records in the queue after each operation, in order from smallest to largest.

poll()	Result	Edges examined	Queue (after)
—	start	—	(5, 0)
(5, 0)	5 is finalized	5→2: 0 + 4 < ∞, dist[2] = 4; 5→4: 0 + 2 < ∞, dist[4] = 2	(4, 2) (2, 4)
(4, 2)	4 is finalized	4→2: 2 + 1 < 4, dist[2] = 3; 4→3: 2 + 1 < ∞, dist[3] = 3	(2, 3) (3, 3) (2, 4)
(2, 3)	2 is finalized	2→1: 3 + 3 < ∞, dist[1] = 6	(3, 3) (2, 4) (1, 6)
(3, 3)	3 is finalized	3→4: visited[4] = true, condition false	(2, 4) (1, 6)
(2, 4)	stale entry, skipped	—	(1, 6)
(1, 6)	1 is finalized	1→3 and 1→4: both already visited	empty

Table 6.5: Trace of `dijkstra(5)` (Example 2). The pairs have the form (vertex, distance).

Lazy deletion is clearly visible in the trace. Vertex 2 entered the queue twice: first as *(2, 4)*, then as *(2, 3)* when the shorter path was found. The new record comes out first and finalizes 2. The old record *(2, 4)* comes out at the fifth `poll()` call; since `visited[2]` is already `true`, it is skipped. In all, six `poll()` calls are made: one for each of the five vertices and one for the stale record. When the loop ends, `dist = [6, 3, 3, 2, 0]` and `prev = [2, 4, 4, 5, -]` (indices 1–5); this is the same as the result found by hand.

6.2.7 Rebuilding the path

Dijkstra produces the distances and the predecessor array together. The distance array answers "how much?" and the predecessor array "by which path?". The result of Example 2 was $\pi = [2, 4, 4, 5, /]$. For target 1 the pointers are followed backward: $\pi[1] = 2$, $\pi[2] = 4$, $\pi[4] = 5$. Vertex 5 is the source, and the trace stops there. The collected vertices are in the order *1, 2, 4, 5*, that is, from target to source. When the order is reversed the path is $5 \to 4 \to 2 \to 1$, of length $2 + 1 + 3 = 6 = d[1]$.

The natural way to reverse the order is to use a **stack**. A stack works by the last-in, first-out (LIFO) principle. Vertices pushed from target to source are popped from source to target.

```
void printPath(int[] prev, int[] dist,
               int source, int target) {
  if (dist[target] == INF) {   /* unreachable */
    System.out.print("no path"); return;
  }
  Deque<Integer> stack = new ArrayDeque<>();
  for (int v = target; v != source; v = prev[v])
    stack.push(v);          /* from target back */
  stack.push(source);
  while (!stack.isEmpty())
    System.out.print(stack.pop() + " ");
}
```

The method first checks whether the target is reachable: if `dist[target]` is still `INF`, it prints "no path" and returns. Then an empty stack is created; in Java the `ArrayDeque` class serves as a stack. The `for` loop starts with `v = target`. As long as `v` is not the source, it pushes `v` onto the stack and moves to the previous vertex. The loop does not push the source, because when `v` equals the source the condition becomes false. The line after the loop pushes the source separately. The last two lines empty the stack and print the vertices. Table 6.6 is the trace of the call `printPath(prev, dist, 5, 1)`.

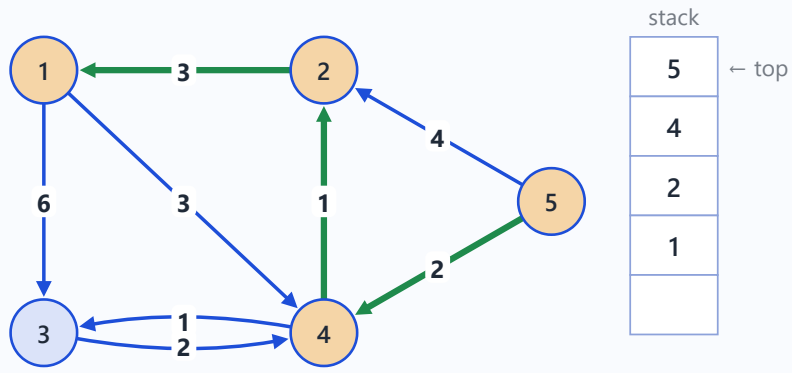


Figure 6.6: The state in the call `printPath(prev, dist, 5, 1)` after the source has been pushed. The source 5 is at the top of the stack and the target 1 at the bottom. The green edges are the shortest path from 5 to 1.

Step	v	Action	Stack	Output
1	1	1 ≠ 5, push(1); v = prev[1] = 2	[1]	
2	2	2 ≠ 5, push(2); v = prev[2] = 4	[2, 1]	
3	4	4 ≠ 5, push(4); v = prev[4] = 5	[4, 2, 1]	
4	5	5 = source, loop ends; push(5)	[5, 4, 2, 1]	
5–8	—	four times pop() and print	[ ]	5 4 2 1

Table 6.6: Trace of `printPath(prev, dist, 5, 1)`; `prev = [2, 4, 4, 5, /]` (indices 1–5). The top of the stack is written on the left.

The cost of the method is proportional to the number of edges on the path. Since a path has at most $V - 1$ edges, the time and the space taken by the stack are $O(V)$. The predecessor array itself takes $O(V)$ space. That all shortest paths can be stored in a single array is a practical consequence of optimal substructure: each path is the path of the previous vertex extended by one edge.

Common mistake

Rebuilding a path without checking reachability. For a target unreachable from the source, the predecessor was never assigned; in Java it stays 0 or holds a sentinel such as −1. The loop can never reach the source: it either spins forever or runs off the end of the array and fails. This is why `printPath` starts with the check `dist[target] == INF`.

6.2.8 Complexity

The cost of Dijkstra comes from two operations: choosing the vertex with the smallest distance and relaxing the edges. Selection is done V times and relaxation at most E times. The total time depends on the data structure used for these two operations.

- **Array.** Distances are kept in an unsorted array. Finding the minimum means scanning the whole array: each selection takes $O(V)$ and V selections take $O(V^2)$. Relaxation is a single array write, $O(1)$. The total is $O(V^2 + E) = O(V^2)$, since in a simple graph $E < V^2$.
- **Binary heap.** The priority queue is built with a binary heap. Removing the minimum and inserting a new record take $O(\log V)$. There are V removals and at most E insertions: the total is $O((V + E) \log V)$. With lazy deletion the queue can hold E records; since $\log E \le \log V^2 = 2 \log V$, the bound does not change.
- **Fibonacci heap.** In this structure removing the minimum takes amortized $O(\log V)$ and decreasing a priority takes amortized $O(1)$. The total is $O(E + V \log V)$. This is the best bound in theory, but because the structure has large constants, a binary heap is often faster in practice.

Selection structure	Selection	Relaxation	Total	Suitable graph
Array (linear scan)	$V \times O(V)$	$E \times O(1)$	$O(V^2)$	dense ($E \approx V^2$)
Binary heap	$O(\log V)$	$O(\log V)$	$O((V + E) \log V)$	sparse ($E \ll V^2$)
Fibonacci heap	$O(\log V)$ amortized	$O(1)$ amortized	$O(E + V \log V)$	best in theory

Table 6.7: Cost of Dijkstra by selection structure.

Which structure is better depends on the density of the graph. In a **dense** graph the number of edges is of order V^2. The heap version then takes $O(V^2 \log V)$ and is slower than the array version. In a **sparse** graph the number of edges is of order V and the heap version clearly wins. Road networks are sparse: usually three or four roads leave an intersection.

Example 6.3

A road network has $V = 10^4$ intersections and $E = 5 \cdot 10^4$ roads. Compare the approximate operation counts of the two versions.

1. **Array version.** $V^2 = 10^8$ operations.
2. **Heap version.** $V + E = 6 \cdot 10^4$; $\log_2 10^4 \approx 13.3$, rounded up to 14. The total is about $6 \cdot 10^4 \cdot 14 \approx 8.4 \cdot 10^5$ operations.
3. **Comparison.** The heap version does about 120 times fewer operations. Because the graph is sparse, the $\log V$ cost of the heap at each operation is much cheaper than the array's scanning of all vertices at each selection.

As for memory, the arrays `dist`, `prev`, and `visited` take $O(V)$ space. In the queue with lazy deletion at most E records can accumulate. The total extra memory is $O(V + E)$. In a queue with a decrease-priority operation each vertex is present once and the queue takes $O(V)$ space.

6.2.9 Correctness and the negative edge problem

At the moment Dijkstra chooses a vertex, it accepts its distance as final. Why this decision is correct is shown by the following theorem.

Theorem 6.1 (Correctness of Dijkstra)

If all edge weights are nonnegative, then at the moment Dijkstra's algorithm chooses a vertex u and adds it to S, $d[u] = \delta(s, u)$.

Proof. Suppose the theorem is false and let u be the first vertex for which $d[u] > \delta(s, u)$ at the moment it is chosen. Take a shortest path P from the source to u. The path P starts at the source, that is, inside S, and ends at u, outside S. So there is a first vertex on P that lies outside S; call it y, and the vertex before it x. Vertex x was finalized earlier, and by the assumption $d[x] = \delta(s, x)$. Since the edge (x, y) was relaxed when x was finalized, $d[y] \le \delta(s, x) + w(x, y) = \delta(s, y)$; that is, $d[y] = \delta(s, y)$. Vertex y comes before u on P, and since the weights are nonnegative, $\delta(s, y) \le \delta(s, u)$. From this we get $d[y] = \delta(s, y) \le \delta(s, u) < d[u]$. But since the algorithm chose u, we must have $d[u] \le d[y]$. Contradiction. □

The key step of the proof is the inequality $\delta(s, y) \le \delta(s, u)$. As the path continues from y to u, its length does not decrease, because every edge added is nonnegative. In short, every path that goes through the outside of S has length at least $d[u]$; a path can only get longer. A negative edge breaks this assumption. A vertex processed later can open a shorter path to an already finalized vertex through a negative edge. Because Dijkstra never returns to finalized vertices, it never sees that path.

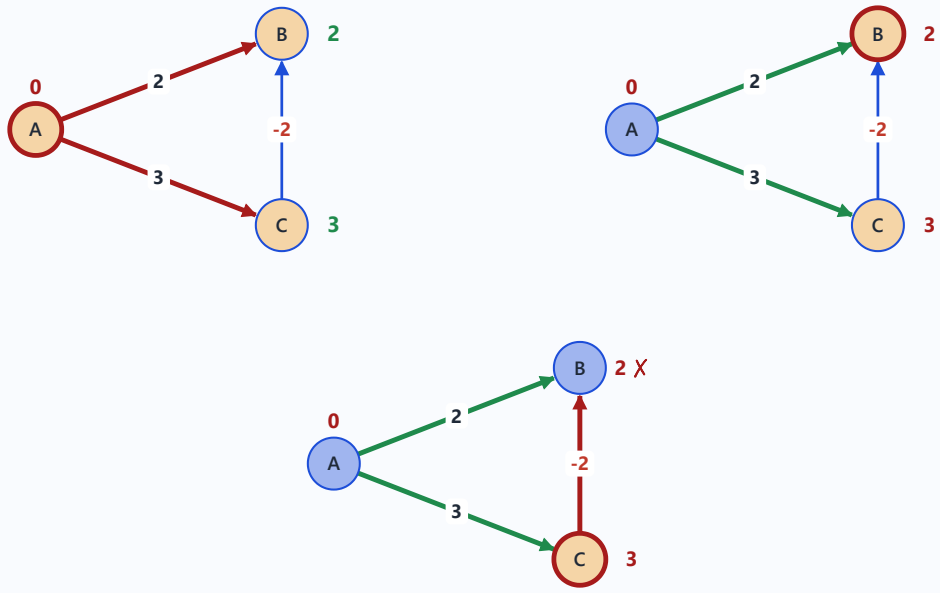


Figure 6.7: Dijkstra going wrong on a negative edge. Left: A is finalized; *d[B] = 2*, *d[C] = 3*. Middle: the smallest estimate is B's; B is finalized. Right: when C is processed, *3 + (−2) = 1 < 2* is found, but B can no longer be changed.

Example 6.4

In the graph with the edges $A \to B$ (2), $A \to C$ (3), and $C \to B$ (−2), the source is A. Compare the *d[B]* found by Dijkstra with the true distance.

1. **A is finalized.** Relaxations: *d[B] = 2*, *d[C] = 3*.
2. **B is chosen.** The smallest estimate is B's (2). Dijkstra finalizes B; that is, it decides "there can be no shorter path to B".
3. **C is chosen.** The edge $C \to B$ proposes *3 + (−2) = 1 < 2*. But B is already in S and is not relaxed.
4. **Result.** Dijkstra gives *d[B] = 2*; the true distance is $\delta(A, B) = 1$ ($A \to C \to B$). The error is silent: the algorithm produces a wrong number without any warning.

One idea that comes to mind is to remove the negative edges by adding the same constant to all weights. This does not work. Since the constant is added to every edge, paths with many edges are penalized more. If 2 is added to all weights in the example, the path $A \to B$ becomes 4 and the path $A \to C \to B$ becomes *5 + 0 = 5*. The direct edge now looks shorter, and the shortest path has changed. Adding a constant preserves the ranking in structures whose number of edges is fixed: for example, every spanning tree has $V - 1$ edges. In shortest paths the number of edges varies from path to path. If there is a negative edge, the right tool is the Bellman-Ford algorithm.

Common mistake

Treating edges of weight 0 as a problem too. The condition of the theorem is "nonnegative" weights; zero satisfies it. The inequality $\delta(s, y) \leq \delta(s, u)$ in the proof also holds with zero-weight edges. Dijkstra fails only on strictly negative edges.

6.3 The Bellman-Ford algorithm

Dijkstra is fast but cannot stand a negative edge. The Bellman-Ford algorithm closes this gap. It finds single-source shortest paths, works correctly with negatively weighted edges, and detects negative cycles reachable from the source. The price is speed: Bellman-Ford takes $O(V \cdot E)$ time and is markedly slower than Dijkstra.

The algorithm is named after Richard Bellman (1958) and Lester Ford Jr. (1956). Edward F. Moore published the same method independently in 1959; for this reason some sources call it the Bellman-Ford-Moore algorithm.

6.3.1 The idea and the steps

Bellman-Ford finalizes no vertex and selects no vertex. Instead it relaxes all the edges in turn and repeats this scan $V - 1$ times. One scan is called a **round**. In each round, the relaxation of an edge lowers the estimate of a vertex. This drop is reflected in the neighbors in later relaxations of the edges leaving that vertex. The improvement thus travels along the path.

This approach is not greedy; it is **dynamic programming**. Round i builds the shortest paths of at most i edges from the paths of at most $i - 1$ edges found in the previous round. The four steps of the algorithm are:

1. **Initialization.** The source is given distance 0 and the other vertices ∞; the predecessors are undefined.
2. **Rounds.** The edges are examined one by one and each is relaxed; this round is repeated $V - 1$ times. The order of the edges within a round is the same in all rounds. The order does not change the result; it only affects in how many rounds the result is reached.
3. **Propagation.** If the distance of a vertex is updated, the distances of its neighbors can be updated in later examinations. The improvement advances at least one edge along the path in every round.
4. **Check.** To check for a negative cycle, all edges are examined once more. If some edge still relaxes, there is a negative cycle reachable from the source. If there is no such cycle, no edge relaxes after $V - 1$ rounds.

Because Bellman-Ford selects no vertex, it does not need an adjacency list either. It is enough to keep the graph as an edge list: each record is a triple (u, v, w). The algorithm reads this list from beginning to end and relaxes each triple.

6.3.2 Why V − 1 rounds? Complexity

The origin of the number of rounds rests on a property of shortest paths. If there is no negative cycle, there is a cycle-free shortest path to every vertex, and it has at most $V - 1$ edges. Each round of the algorithm extends these paths by one more edge.

Theorem 6.2 (Bellman-Ford invariant)

If there is no negative cycle reachable from the source, then at the end of round i, $d[v] = \delta(s, v)$ for every vertex v having a shortest path of at most i edges. Accordingly, after $V - 1$ rounds all distances are correct.

Proof. Let the shortest path to v be $s = v_0, v_1, \ldots, v_k = v$ with $k \le i$. We use induction on j. At the start $d[v_0] = 0 = \delta(s, s)$. Assume that at the end of round $j - 1$, $d[v_{j-1}] = \delta(s, v_{j-1})$. In round j the edge (v_{j-1}, v_j) is relaxed as well. After the relaxation $d[v_j] \le \delta(s, v_{j-1}) + w(v_{j-1}, v_j) = \delta(s, v_j)$; the equality comes from optimal substructure. Since the estimate cannot fall below the distance, $d[v_j] = \delta(s, v_j)$. For $j = k \le i$ the claim follows. If there is no negative cycle, the shortest path of every vertex has at most $V - 1$ edges; taking $i = V - 1$ gives all distances correct. □

The main idea of the proof is this: in round j the j-th edge of the path is certainly relaxed. The order in which the edges are examined does not matter; an order can only make the work finish earlier. With a favorable order, several edges can improve one after another within one round, and the path can be completed in a single round. In the worst order each round advances only one edge, and all $V - 1$ rounds are needed.

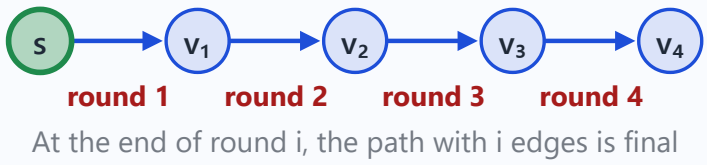


Figure 6.8: In a chain of five vertices the shortest path has four edges. At the end of round i the first i edges of the path are final; four rounds complete the whole path.

The time is easy to compute. Each round examines E edges and each examination takes $O(1)$. $V - 1$ rounds and one check round make $V \cdot E$ examinations in all: $O(V \cdot E)$. The extra memory is only the distance and predecessor arrays: $O(V)$.

Graph	Time
Sparse, $E \approx V$	$O(V^2)$
Dense, $E \approx V^2$	$O(V^3)$

Graph	Time
Best case (early stop)	O(E)

Table 6.8: Running time of Bellman-Ford by density of the graph.

The last row of the table comes from a simple improvement. If no d value changes in a round, later rounds cannot change anything either. This is because a round can find a new relaxation only thanks to estimates that changed in the previous round. After a round with no change, the loop can be left. If the edges are in a favorable order, the first round finds everything and the second confirms it: the time drops to $O(E)$.

```
int[] bellmanFordEarly(int V, List<Edge> edges, int source) {
  int[] dist = new int[V];
  Arrays.fill(dist, INF);
  dist[source] = 0;
  for (int i = 1; i < V; i++) {
    boolean changed = false;
    for (Edge e : edges) {
      int u = e.from, v = e.to, w = e.weight;
      if (dist[u] != INF && dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        changed = true;
      }
    }
    if (!changed) break;   /* nothing changed this round */
  }
  return dist;
}
```

In the code the flag `changed` is set to `false` at the start of each round. Any relaxation in the round sets the flag to `true`. If the flag is still `false` when the round ends, `break` ends the loop. Early stopping does not change the result; it only skips needless rounds. The worst case is still $O(V \cdot E)$, because something may change in every round.

Common mistake

Assuming that Bellman-Ford finishes the job in a single round. This is true if the edge order is lucky, but not in general. In Example 2 with negative edges all five rounds are needed. Also, the check round is a scan separate from the $V - 1$ rounds; it is not counted among them.

6.3.3 Example 1: relaxing edge by edge

The first example is a directed graph on five vertices. The source is s, and the edges are examined in each round in this order: (s, v_1) 9, (s, v_2) 3, (v_1, v_2) 6, (v_1, v_4) 2, (v_2, v_1) 2, (v_2, v_3) 1, (v_3, v_2) 2, (v_3, v_4) 2.

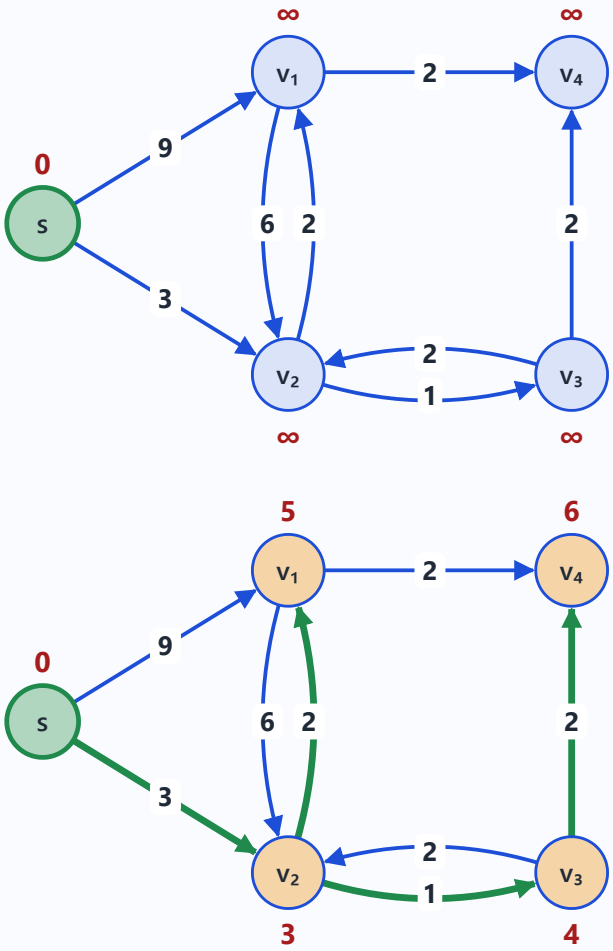


Figure 6.9: Bellman-Ford Example 1. Left, the initial state; right, the result. The green edges are the shortest-path tree: $s \to v_2 \to v_1$ (5), $s \to v_2 \to v_3 \to v_4$ (6).

Edge	Candidate	Result	d[v_1]	d[v_2]	d[v_3]	d[v_4]
start	—	—	∞	∞	∞	∞
(s, v_1) 9	0 + 9 = 9 < ∞	d[v_1] = 9	**9**	∞	∞	∞
(s, v_2) 3	0 + 3 = 3 < ∞	d[v_2] = 3	9	**3**	∞	∞
(v_1, v_2) 6	9 + 6 = 15 ≥ 3	unchanged	9	3	∞	∞
(v_1, v_4) 2	9 + 2 = 11 < ∞	d[v_4] = 11	9	3	∞	**11**
(v_2, v_1) 2	3 + 2 = 5 < 9	d[v_1] = 5	**5**	3	∞	11
(v_2, v_3) 1	3 + 1 = 4 < ∞	d[v_3] = 4	5	3	**4**	11
(v_3, v_2) 2	4 + 2 = 6 ≥ 3	unchanged	5	3	4	11

Edge	Candidate	Result	d[v_1]	d[v_2]	d[v_3]	d[v_4]
(v_3, v_4) 2	4 + 2 = 6 < 11	d[v_4] = 6	5	3	4	**6**

Table 6.9: The first round of Bellman-Ford Example 1, edge by edge.

In the first round six of the eight edges relax. Some estimates change twice within the same round. v_1 first becomes 9 by the direct edge, then drops to 5 through v_2. v_4 first becomes 11, then drops to 6 through v_3. This is a case where the edge order helps. The value of v_2 was found before the edges leaving it were examined; the improvement therefore spreads along the path within the same round.

In the second round none of the eight edges relaxes. The version with early stopping halts here. The result is $d[v_1] = 5$, $d[v_2] = 3$, $d[v_3] = 4$, $d[v_4] = 6$. Without early stopping, three more rounds and a check round would run in vain. In this order the whole job is finished in a single round.

6.3.4 Example 2: negative edges and the worst order

The second example has six vertices and four negative edges. The source is S. The edges are examined in each round in this order: *(E, D)* −4, *(E, A)* 4, *(C, D)* −2, *(C, E)* 1, *(B, C)* −2, *(A, B)* −1, *(S, A)* 3. This order was chosen on purpose: it arranges the edges of the shortest path from last to first. This is the worst order for Bellman-Ford.

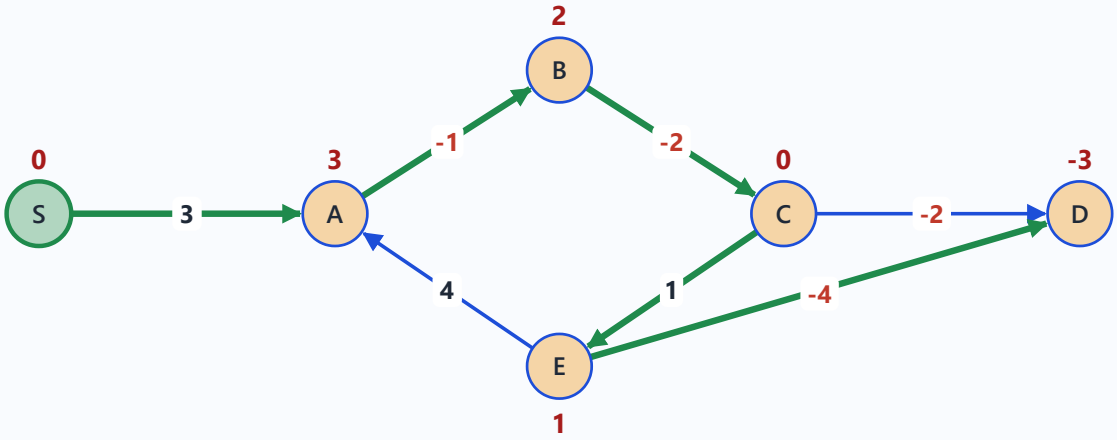


Figure 6.10: Result of Bellman-Ford Example 2 (source S). The red edge weights are negative. The green edges are the shortest-path tree; the shortest path to D comes through E (−3).

Round	Edges that relax	S	A	B	C	D	E
start	—	0	∞	∞	∞	∞	∞
1	(S, A): ∞ → 3	0	**3**	∞	∞	∞	∞
2	(A, B): ∞ → 2	0	3	**2**	∞	∞	∞
3	(B, C): ∞ → 0	0	3	2	**0**	∞	∞
4	(C, D): ∞ → −2; (C, E): ∞ → 1	0	3	2	0	**−2**	**1**
5	(E, D): −2 → −3	0	3	2	0	**−3**	1

Round	Edges that relax	S	A	B	C	D	E
check	none	0	3	2	0	−3	1

Table 6.10: Round-by-round trace of Bellman-Ford Example 2.

In the first round only the edge *(S, A)* is useful. The start vertices of the other edges in the list are still ∞ at the moment they are examined. For example, the edge *(A, B)* comes before *(S, A)* in the list. When it is examined, the estimate of A is still ∞ and the edge is wasted. Each round extends the path by exactly one edge: A, B, C are found in turn. In the fourth round two edges leaving C relax at once. In the fifth round D finds a shorter path through E: $1 + (-4) = -3 < -2$. The direct edge $C \to D$ gives −2, while the path $C \to E \to D$ gives $1 - 4 = -3$.

All $V - 1 = 5$ rounds were used. In the check round no edge relaxes; so there is no negative cycle. Indeed, the only cycle in the graph is $A \to B \to C \to E \to A$ and its weight is $-1 - 2 + 1 + 4 = +2$. Although there are negative edges, the cycle is positive.

Example 6.5

If the same edges are examined in reverse order, that is, in the order *(S, A), (A, B), (B, C), (C, E), (C, D), (E, A), (E, D)*, how many rounds does Bellman-Ford with early stopping run?

1. **First round.** *(S, A)*: A = 3. Right after, *(A, B)*: B = 2. *(B, C)*: C = 0. *(C, E)*: E = 1. *(C, D)*: D = −2. *(E, A)* proposes 5 and cannot beat 3. The last edge *(E, D)*: D = −3. Six edges relax.
2. **Second round.** No edge relaxes; `changed` stays `false` and the loop ends.
3. **Result.** Two rounds suffice; the result is the same as in the worst order. The edge order changes not the result but the number of rounds. These two numbers were also confirmed by running the early-stopping code of these notes in Java: 5 rounds in the worst order, 2 rounds in the good order.

6.3.5 Example 3: the graph of Dijkstra

The third example is the directed graph of Dijkstra Example 2; the source is again 5. There is no negative edge, so the two algorithms must give the same result. The edges are examined in each round in this order: *(1, 3), (1, 4), (2, 1), (3, 4), (4, 2), (4, 3), (5, 2), (5, 4)*.

Round	Edges that relax	d[1..5]	π[1..5]
start	—	∞, ∞, ∞, ∞, 0	/, /, /, /, /
1	(5, 2): ∞ → 4; (5, 4): ∞ → 2	∞, 4, ∞, 2, 0	/, 5, /, 5, /
2	(2, 1): ∞ → 7; (4, 2): 4 → 3; (4, 3): ∞ → 3	7, 3, 3, 2, 0	2, 4, 4, 5, /
3	(2, 1): 7 → 6	6, 3, 3, 2, 0	2, 4, 4, 5, /
4	none	6, 3, 3, 2, 0	2, 4, 4, 5, /

Table 6.11: Bellman-Ford Example 3 (source 5). d and π at the end of each round.

In the first round the two edges leaving the source are at the end of the list; only they relax. In the second round there are three relaxations. The edge *(2, 1)* gives the first path to 1: *4 + 3 = 7*. Right after, the edge *(4, 2)* lowers the estimate of 2 to 3. But because *(2, 1)* comes before *(4, 2)* in the list, this improvement reaches 1 only in the third round: *3 + 3 = 6*. In the fourth round nothing changes.

The result is *d = [6, 3, 3, 2, 0]* and *π = [2, 4, 4, 5, /]*. It is exactly the same as the result Dijkstra found. The difference is in the work done. Dijkstra looked at each edge once; Bellman-Ford makes 32 edge examinations in four rounds, and with the check round the number rises to 40.

The check round asks the question *d[v] > d[u] + w(u, v)* once more for every edge. Table 6.12 shows that on the result of Example 3 the answer to this question is "no" for all eight edges.

Edge	d[v]	d[u] + w(u, v)	Result
(1, 3)	d[3] = 3	6 + 6 = 12	3 ≯ 12 ✓
(1, 4)	d[4] = 2	6 + 3 = 9	2 ≯ 9 ✓
(2, 1)	d[1] = 6	3 + 3 = 6	6 ≯ 6 ✓
(3, 4)	d[4] = 2	3 + 2 = 5	2 ≯ 5 ✓
(4, 2)	d[2] = 3	2 + 1 = 3	3 ≯ 3 ✓
(4, 3)	d[3] = 3	2 + 1 = 3	3 ≯ 3 ✓
(5, 2)	d[2] = 3	0 + 4 = 4	3 ≯ 4 ✓
(5, 4)	d[4] = 2	0 + 2 = 2	2 ≯ 2 ✓

Table 6.12: Negative cycle check on the result of Example 3.

The edges for which equality holds are notable: *(2, 1)*, *(4, 2)*, *(4, 3)*, and *(5, 4)*. These are exactly the edges of the shortest-path tree. For every edge in the tree *d[v] = d[u] + w(u, v)*, because the value of *v* came from this edge. For edges outside the tree the inequality is strict.

6.3.6 Negative cycle detection

The strength of the check round shows in a graph with a negative cycle. Figure 6.11 is an example with four vertices. There is an edge of weight 2 from the source S to A. The cycle $A \to B \to C \to A$ has weight *1 + (−3) + 1 = −1*. The edge order is *(S, A)*, *(A, B)*, *(B, C)*, *(C, A)*.

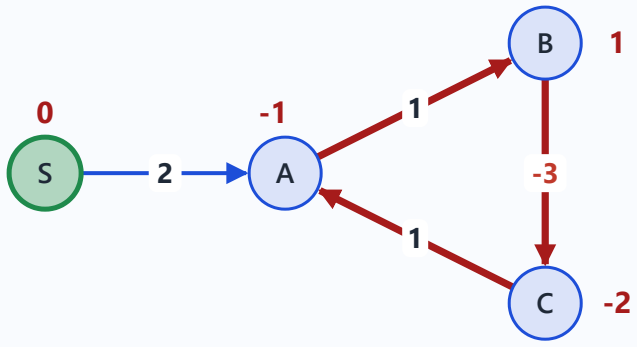


Figure 6.11: A graph with a negative cycle and the estimates after $V - 1 = 3$ rounds. The red cycle $A \to B \to C \to A$ has weight -1.

Round	S	A	B	C
1	0	1	3	0
2	0	0	2	−1
3	0	−1	1	−2

Table 6.13: Estimates at the end of each round in the graph with a negative cycle.

In the first round A first becomes 2 by the edge $S \to A$. Before the round ends, the cycle is traversed: B = 3, C = 0 are found and the edge $C \to A$ lowers A to 1. In every later round the three values on the cycle decrease by one. This decrease never stops, because each time the cycle is traversed the path becomes 1 shorter. In a normal graph the values settle in at most $V - 1$ rounds; here they do not settle.

After $V - 1 = 3$ rounds, $d = [0, -1, 1, -2]$. The check round starts with (S, A): $0 + 2 = 2$ is not smaller than the value -1 of A. Then comes (A, B): $-1 + 1 = 0 < 1$. The edge still relaxes; so there is a negative cycle. The shortest path of every vertex on the cycle and every vertex reached from it is undefined ($-\infty$). The d values found are meaningless for these vertices; they merely reflect the round at which the algorithm was stopped.

Theorem 6.3 (Negative cycle check)

If after the $V - 1$ rounds of Bellman-Ford an edge still relaxes, there is a negative cycle reachable from the source. Conversely, if there is such a cycle, at least one edge relaxes.

Proof. The first direction follows from the previous theorem: with no negative cycle, all estimates would be correct after $V - 1$ rounds and the triangle inequality would allow no edge to relax. For the converse take a reachable negative cycle $v_0, v_1, \ldots, v_k = v_0$ and suppose that no edge relaxes. Then for

every edge of the cycle $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ holds. When these k inequalities are added, the sums of d on the left and on the right contain the same vertices and cancel. What remains is $0 \le$ the weight of the cycle. This contradicts the cycle being negative. Because the cycle is reachable from the source, the estimates are finite and the addition is valid. □

Example 6.6

A cycle consists of the edges $A \to B$ (3), $B \to C$ (−5), and $C \to A$ (w), and A is reachable from the source. For which values of the integer w does the check round report a negative cycle?

1. **Cycle weight.** $3 + (-5) + w = w - 2$.
2. **Negativity condition.** $w - 2 < 0$, that is, $w < 2$. For integers, $w \le 1$.
3. **Boundary case.** At $w = 2$ the weight of the cycle is 0. Going around the cycle does not shorten the path; there is no negative cycle and the check reports nothing.

6.3.7 Pseudocode, Java code, and a trace

The pseudocode of Bellman-Ford has two parts: the outer loop of $V - 1$ rounds and the single check round that follows.

```
BELLMAN_FORD(G, source):
  dist = [ ]
  for each v in V:
    dist[v] = infinity
  dist[source] = 0
  loop i = 1 to |V| - 1:
    for each (u, v, weight) in E:
      if dist[u] + weight < dist[v]:
        dist[v] = dist[u] + weight
  for each (u, v, weight) in E:
    if dist[u] + weight < dist[v]:
      error "Graph contains a negative cycle"
  return dist
```

The outer loop runs $|V| - 1$ times; the inner loop walks the whole edge list in every round. The last four lines are the check round. The check asks the relaxation condition but makes no assignment; if the condition holds, it reports an error. In the Java counterpart the edges are kept in a list named `edges`. Each `Edge` object has the fields `from`, `to`, and `weight`. The vertices are numbered from 0 to `V - 1`.

```
void bellmanFord(int source) {
  int[] dist = new int[V];
  int[] prev = new int[V];
  Arrays.fill(dist, INF);
  dist[source] = 0;
  for (int i = 1; i < V; i++) {
    for (Edge e : edges) {
      int u = e.from, v = e.to, w = e.weight;
      if (dist[u] != INF
          && dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        prev[v] = u;
      }
    }
  }
  for (Edge e : edges) { /* check */
    int u = e.from, v = e.to, w = e.weight;
    if (dist[u] != INF
        && dist[u] + w < dist[v])
      throw new IllegalStateException(
          "Negative cycle");
  }
}
```

- **Lines 2–5:** the arrays are created, the distances are filled with infinity, and that of the source is set to 0.
- **Line 6:** `i` runs from 1 to `V - 1`; in all $V - 1$ rounds.
- **Lines 7–13:** for each edge, `u`, `v`, and `w` are taken and relaxation is done. The first part of the condition is `dist[u] != INF`. If `dist[u]` is infinite, the addition is never performed.
- **Lines 16–21:** the check round asks the same condition. If the condition holds, `IllegalStateException` is thrown.

The test `dist[u] != INF` in the condition is not decoration. `INF` here is `Integer.MAX_VALUE`, that is, 2,147,483,647. In Java, adding a positive number to it overflows and the result wraps to a negative number: `Integer.MAX_VALUE + 3` gives −2,147,483,646. Without the test, an edge leaving an unreached vertex would look "shorter" because of this negative number. In Dijkstra's code this test was not needed, because only vertices with finite distances leave the queue. Bellman-Ford, however, also tries to relax the edges of vertices not yet reached.

Table 6.14 is a trace of the code on a graph with four vertices. The source is 0 and the edge list is in the order *(2, 1, −2), (0, 1, 4), (0, 2, 5), (1, 3, 3)*.

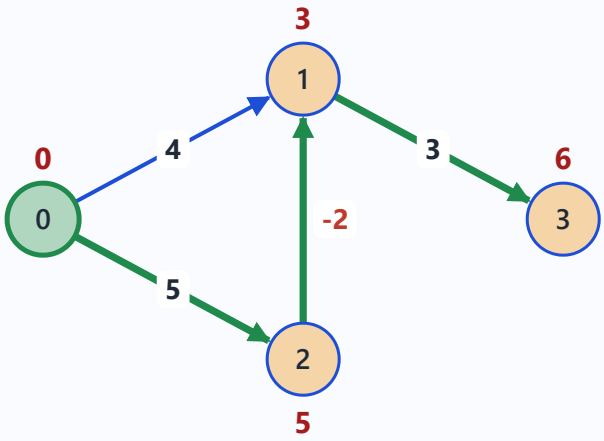


Figure 6.12: The graph of the `bellmanFord(0)` trace and its result. The shortest path to 1 uses the negative edge *2 → 1*: *0 → 2 → 1*, of length *5 − 2 = 3*.

Round	Edge	Condition	Result	dist
—	—	—	start	[0, ∞, ∞, ∞]
1	(2, 1, −2)	dist[2] = ∞	skipped	[0, ∞, ∞, ∞]
1	(0, 1, 4)	0 + 4 < ∞	dist[1] = 4, prev[1] = 0	[0, 4, ∞, ∞]
1	(0, 2, 5)	0 + 5 < ∞	dist[2] = 5, prev[2] = 0	[0, 4, 5, ∞]
1	(1, 3, 3)	4 + 3 < ∞	dist[3] = 7, prev[3] = 1	[0, 4, 5, 7]
2	(2, 1, −2)	5 − 2 = 3 < 4	dist[1] = 3, prev[1] = 2	[0, 3, 5, 7]
2	(0, 1, 4)	4 ≥ 3	unchanged	[0, 3, 5, 7]
2	(0, 2, 5)	5 ≥ 5	unchanged	[0, 3, 5, 7]
2	(1, 3, 3)	3 + 3 = 6 < 7	dist[3] = 6	[0, 3, 5, 6]
3	four edges	none true	unchanged	[0, 3, 5, 6]
check	four edges	none true	no exception	[0, 3, 5, 6]

Table 6.14: Trace of `bellmanFord(0)`.

In the first round the edge *(2, 1)* is wasted, because it is at the head of the list and the value of 2 is still ∞ when it is examined. In the second round this edge is useful: the negative weight lowers the value of 1 from 4 to 3. At the end of the same round the edge *(1, 3)* carries this improvement on to 3: from 7 to 6. The third round changes nothing, and the check round throws no exception. The result is `[0, 3, 5, 6]`.

The same code ends differently on a graph with a negative cycle. Let the vertices S, A, B, C be numbered 0, 1, 2, 3. The edges are in the order *(0, 1, 2), (1, 2, 1), (2, 3, −3), (3, 1, 1)*. In the first round all four edges relax: A = 2, B = 3, C = 0, then A = 1. In the second and third rounds three edges relax each time; the values on the cycle fall by one every round. After three rounds `dist = [0, -1, 1, -2]`. In the check, for the first edge *(0, 1)*, *0 + 2 = 2* is not smaller than −1. For the second edge *(1, 2)*, *−1 + 1 = 0 < 1*; the condition holds and `IllegalStateException("Negative cycle")` is thrown.

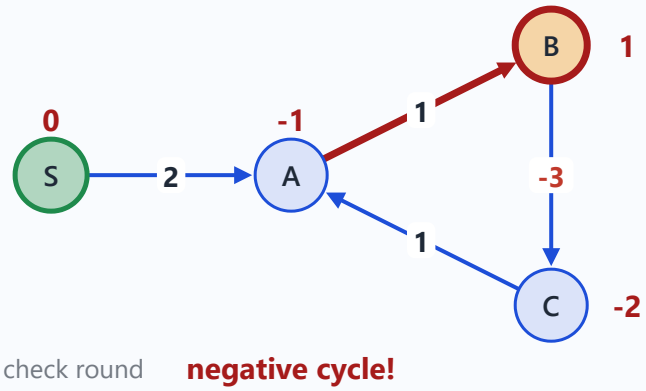


Figure 6.13: In the check round the edge $A \to B$ still relaxes: $-1 + 1 = 0 < 1$. The code throws the negative-cycle exception.

Common mistake

Skipping the check round and using the result of the $V - 1$ rounds directly. On a graph with a negative cycle the algorithm still returns an array; the array is full of ordinary numbers and nothing shows that it is wrong. The array `[0, -1, 1, -2]` in the negative-cycle example is revealed as meaningless only by the check.

6.4 The Floyd-Warshall algorithm

Dijkstra and Bellman-Ford work from a single source. Some applications, however, want the distances between all pairs of vertices: the table of distances between all the cities on a map, or the distances of all the routers in a network from one another. The Floyd-Warshall algorithm solves this all-pairs problem in one go. Its code consists of just three nested loops. It works with negative edges and detects negative cycles too.

6.4.1 The all-pairs problem

The history of the algorithm is interesting. Bernard Roy in 1959, Stephen Warshall in 1962, and Robert Floyd, also in 1962, published the same idea independently of one another. Warshall's paper computed the transitive closure of a relation, that is, which vertex can reach which. Floyd applied the same structure to shortest paths. The algorithm today bears both their names.

Floyd-Warshall is a dynamic programming algorithm. The results are kept in a $V \times V$ matrix D; $D[i][j]$ is the shortest distance from i to j found so far. To rebuild the paths, a second matrix Π is also kept. It is the form of the array π of the single-source algorithms that has a separate row for each source.

The algorithm works in four steps:

1. **Initialization.** D is filled with the edge weights: the diagonal $D[i][i] = 0$, and ∞ where there is no direct edge.
2. **Intermediate vertex.** $k = 1, 2, \ldots, n$ are taken in turn. For each k all pairs (i, j) are visited.
3. **Comparison.** The length of the path through k, $D[i][k] + D[k][j]$, is computed and compared with the current $D[i][j]$.
4. **Update.** If the new path is shorter, the value is written to the matrix and the matrix Π is updated too.

The comparison in the third step is the form of relaxation on pairs of vertices. The paths $i \rightsquigarrow k$ and $k \rightsquigarrow j$ are joined; if the resulting path is shorter than the known path $i \rightsquigarrow j$, it takes its place.

6.4.2 The subproblem and the recurrence

The first job in dynamic programming is to define the subproblem. In Floyd-Warshall the subproblem restricts which intermediate vertices a path may use. The **intermediate vertices** of a path are its vertices other than the start and the end.

Definition 6.4

$D^{(k)}[i][j]$ is the length of the shortest among the paths $i \rightsquigarrow j$ whose intermediate vertices are all chosen from the set $\{1, 2, \ldots, k\}$. In $D^{(0)}$ no intermediate vertex is allowed; this matrix contains only the direct edges. In $D^{(n)}$ all vertices may be intermediate; this matrix is the sought result.

How is $D^{(k)}$ found once $D^{(k-1)}$ is known? Consider a shortest path p from i to j whose intermediate vertices come from $\{1, \ldots, k\}$. There are two cases.

- ***k* is not on *p*.** Then all intermediate vertices of p are from $\{1, \ldots, k-1\}$ and its length is $D^{(k-1)}[i][j]$.
- ***k* is on *p*.** If there is no negative cycle, p is a simple path and passes through k only once. The path p splits into the parts $i \rightsquigarrow k$ and $k \rightsquigarrow j$. The intermediate vertices of both parts are from $\{1, \ldots, k-1\}$. By optimal substructure each part is shortest on its own. The length is $D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$.

The shortest path is the smaller of these two options:

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

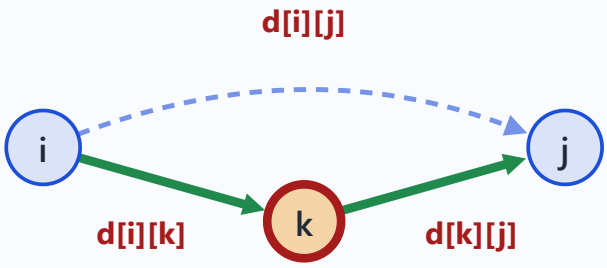


Figure 6.14: The Floyd-Warshall recurrence. The dashed arrow is the known path $i \dashrightarrow j$. The green arrows are the candidate path through k: $D[i][k] + D[k][j]$. The shorter of the two is kept.

The recurrence says that either k is not on the path or the path passes through k exactly once. The set of intermediate vertices grows by one vertex at each step: first only 1 can be used, then 1 and 2, and finally all vertices. After n steps all possible paths have been taken into account for every pair.

6.4.3 Complexity and loop order

The running time can be read off the structure of the code. There are three nested `for` loops, each running V times. The outermost loop walks over the intermediate vertices (k), the second over the source vertices (i), and the innermost over the target vertices (j). The comparison in the innermost loop takes $O(1)$ and is made V^3 times. The time is independent of the number of edges of the graph: $\Theta(V^3)$. This is a tight bound, with the best and worst cases the same.

The memory is $\Theta(V^2)$. From the recurrence it might seem that a new matrix is needed for every k. It is not: D can be updated in place. In round k, row k and column k do not change. For $D[i][k]$, for example, the candidate is $D[i][k] + D[k][k] = D[i][k] + 0$, which cannot beat the current value. All the values $D[i][k]$ and $D[k][j]$ read in this round are therefore the same as their values in $D^{(k-1)}$.

Method	Time
Floyd-Warshall	$O(V^3)$
$V \times$ Dijkstra (array)	$O(V^3)$
$V \times$ Dijkstra (heap)	$O(V(V + E) \log V)$
$V \times$ Bellman-Ford	$O(V^2 E)$

Table 6.15: Options for all-pairs shortest paths.

The table shows that Floyd-Warshall is not always the fastest option. On a sparse graph without negative edges, running heap-based Dijkstra once from every vertex is faster. For $V = 200$ and $E = 800$, for example, $V^3 = 8 \cdot 10^6$, while $V(V + E) \log_2 V \approx 1.5 \cdot 10^6$. The advantage of Floyd-Warshall lies elsewhere. It works with negative edges, its code is very short, and its constant factor is small. It is a practical choice on dense graphs and for problems up to a few hundred vertices. Running Bellman-Ford V times on a graph with negative edges takes $O(V^2 E)$; on a dense graph that is $O(V^4)$.

The order of the loops is not arbitrary: k must be outermost. The recurrence builds $D^{(k)}$ from $D^{(k-1)}$. When round k begins, all pairs must have been computed with the intermediate vertices $\{1, \ldots, k-1\}$. If the loops are put in the order `i`, `j`, `k`, each cell (i, j) is computed once, before the other cells are ready.

Example 6.7

What happens if the loops are run in the order `i`, `j`, `k` (k innermost) on the graph of the Floyd-Warshall example below?

1. **The cell 3 ↛ 1.** The correct result is $D[3][1] = 6$; the path is $3 \to 4 \to 2 \to 1$ and uses two intermediate vertices.
2. **The moment of computation.** In the wrong order $D[3][1]$ is computed a single time, as the first cell of row 3. At that moment only the direct edges are known: $D[3][4] = 2$, but $D[4][1]$ is still ∞, because row 4 will be processed later. $D[3][2]$ has not been computed yet either and is ∞.
3. **Result.** For no k is $D[3][k] + D[k][1]$ finite, and the cell stays ∞. In an experiment in Python this order found all other cells of the matrix correctly and missed only this one. The number of loops does not change and the time is still $\Theta(V^3)$; only the correctness of the result changes.

6.4.4 Example: the matrices D and Π

The example graph is the directed graph on five vertices used in Dijkstra Example 2 and Bellman-Ford Example 3. This time no source is chosen; all pairs are computed. In the matrix Π, $\Pi[i][j]$ is the vertex before j on the shortest path from i to j. At the start, if there is a direct edge, $\Pi[i][j] = i$; otherwise "/" is written.

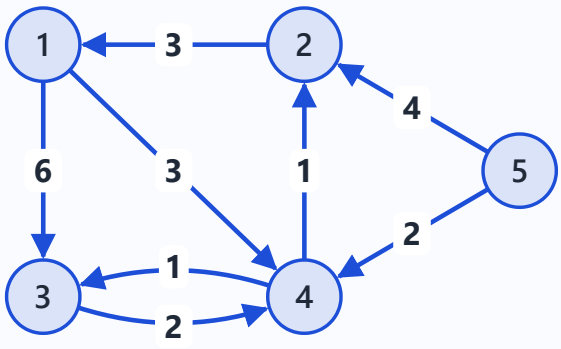


Figure 6.15: The graph of the Floyd-Warshall example. The edges are $1 \to 3$ (6), $1 \to 4$ (3), $2 \to 1$ (3), $3 \to 4$ (2), $4 \to 2$ (1), $4 \to 3$ (1), $5 \to 2$ (4), $5 \to 4$ (2).

	1	2	3	4	5
1	0 / –	∞ / –	6 / 1	3 / 1	∞ / –
2	3 / 2	0 / –	∞ / –	∞ / –	∞ / –
3	∞ / –	∞ / –	0 / –	2 / 3	∞ / –
4	∞ / –	1 / 4	1 / 4	0 / –	∞ / –
5	∞ / –	4 / 5	∞ / –	2 / 5	0 / –

Table 6.16: $D^{(0)}$ and $\Pi^{(0)}$ (initialization). Row i, column j; each cell shows "D / Π"; "–" is written where Π is undefined.

As intermediate vertices are added in turn, the matrix changes. The cells that change in each round are listed below.

- ***k = 1.*** The only edge into 1 is $2 \to 1$, so only row 2 can change. $2 \rightsquigarrow 3$: $D[2][1] + D[1][3] = 3 + 6 = 9$. $2 \rightsquigarrow 4$: $3 + 3 = 6$. The Π value of both cells becomes $\Pi[1][j] = 1$.
- ***k = 2.*** The edges into 2 come from 4 and from 5. $4 \rightsquigarrow 1$: $1 + 3 = 4$. $5 \rightsquigarrow 1$: $4 + 3 = 7$. $5 \rightsquigarrow 3$: $4 + 9 = 13$. The last cell uses the path $2 \rightsquigarrow 3$ found in the previous round; that path already goes through 1. So the path $5 \rightsquigarrow 3$ uses two intermediate vertices: $5 \to 2 \to 1 \to 3$.
- ***k = 3.*** The only edge leaving 3 is $3 \to 4$, and no path gets shorter through 3.
- ***k = 4.*** Vertex 4 is the center of the graph; eight cells get shorter: $1 \rightsquigarrow 2$ (4), $1 \rightsquigarrow 3$ (6 → 4), $2 \rightsquigarrow 3$ (9 → 7), $3 \rightsquigarrow 1$ (6), $3 \rightsquigarrow 2$ (3), $5 \rightsquigarrow 1$ (7 → 6), $5 \rightsquigarrow 2$ (4 → 3), $5 \rightsquigarrow 3$ (13 → 3). For $5 \rightsquigarrow 3$, for example, $D[5][4] + D[4][3] = 2 + 1 = 3 < 13$.
- ***k = 5.*** No edge enters 5; for every $i \neq 5$, $D[i][5] = \infty$. No path gets shorter through 5.

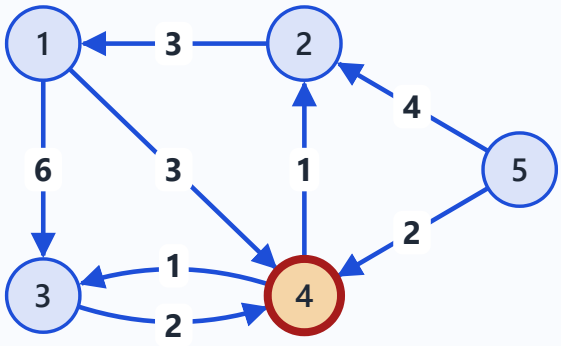


Figure 6.16: The round $k = 4$. The intermediate vertex 4, with the thick outline, shortens eight paths at once through the edges entering and leaving it.

	1	2	3	4	5
1	0 / –	4 / 4	4 / 4	3 / 1	∞ / –
2	3 / 2	0 / –	7 / 4	6 / 1	∞ / –
3	6 / 2	3 / 4	0 / –	2 / 3	∞ / –

	1	2	3	4	5
4	4 / 2	1 / 4	1 / 4	0 / –	∞ / –
5	6 / 2	3 / 4	3 / 4	2 / 5	0 / –

Table 6.17: Final state $D^{(5)}$ and $\Pi^{(5)}$; each cell shows "D / Π"; "–" is written where Π is undefined.

A few results can be read from the final matrix. Column 5 is ∞ from top to bottom except for 5 itself: 5 cannot be reached from any vertex. Row 5 is *[6, 3, 3, 2, 0]*. This is exactly the distance array that Dijkstra and Bellman-Ford found for source 5. Although Floyd-Warshall runs without choosing a source, each row of the matrix gives the single-source result of the vertex of that row. The diagonal is all 0; there is no negative cycle.

6.4.5 Reading a path

Π[i][j] is the vertex before *j* on the shortest path $i \rightsquigarrow j$. The path is read backward from the target, as in the single-source case. The difference is that the whole reading is done in row *i*: the vertex before *j* is *Π[i][j]*, the one before that is *Π[i][Π[i][j]]*, and so on.

Example 6.8

Read the paths $5 \rightsquigarrow 1$ and $1 \rightsquigarrow 2$ from the final matrix *Π*.

1. ***5*** $\rightsquigarrow$ ***1.*** Read in row 5: *Π[5][1] = 2*, *Π[5][2] = 4*, *Π[5][4] = 5*. Vertex 5 is the source and the reading stops. The path is $5 \to 4 \to 2 \to 1$, of length *D[5][1] = 6*.
2. ***1*** $\rightsquigarrow$ ***2.*** Read in row 1: *Π[1][2] = 4*, *Π[1][4] = 1*. The path is $1 \to 4 \to 2$, of length *3 + 1 = 4 = D[1][2]*.
3. **Check.** $3 \rightsquigarrow 1$: *Π[3][1] = 2*, *Π[3][2] = 4*, *Π[3][4] = 3*. The path is $3 \to 4 \to 2 \to 1$, of length *2 + 1 + 3 = 6 = D[3][1]*.

The rule for updating *Π* can be understood from this too. When a shorter path through *k* is found, *Π[i][j] ← Π[k][j]* is done. The last part of the new path is the path $k \rightsquigarrow j$. So the vertex before *j* is the same as the vertex before *j* on the path $k \rightsquigarrow j$. Writing *Π[i][j] ← k* would be wrong; *k* need not be directly joined to *j*.

6.4.6 Pseudocode, Java code, and a trace

The pseudocode has two parts. The first part initializes the matrix.

```
FLOYD_WARSHALL(G):
  dist = [ ][ ]
  for each u in V:
    for each v in V:
      if u == v:
        dist[u][v] = 0
      else if (u, v) in E:
        dist[u][v] = weight(u, v)
      else:
        dist[u][v] = infinity
```

The second part is the three nested loops; the diagonal is checked afterward.

```
  for each k in V:
    for each i in V:
      for each j in V:
        if dist[i][k] + dist[k][j]
             < dist[i][j]:
          dist[i][j] =
            dist[i][k] + dist[k][j]
  for each i in V:
    if dist[i][i] < 0:
      error "Graph contains a negative cycle"
  return dist
```

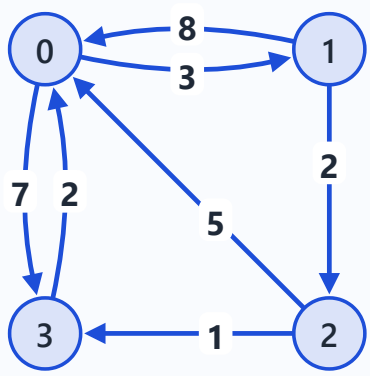
The negative-cycle check is simpler than that of Bellman-Ford. A vertex *i* on a negative cycle finds a path of negative length to itself by going around the cycle. The algorithm computes this path too, and *D[i][i]* falls below 0. A negative value on the diagonal is therefore the sign of a negative cycle.

In the Java code the matrix is updated in place; no separate copy of $D^{(k)}$ is kept. The matrix `prev` is a field of the class and initially holds the values `prev[i][j] = i`.

```
void floydWarshall(int[][] d) {  /* V x V */
  for (int k = 0; k < V; k++)       /* via */
    for (int i = 0; i < V; i++)     /* from */
      for (int j = 0; j < V; j++)   /* to */
        if (d[i][k] != INF
            && d[k][j] != INF
            && d[i][k] + d[k][j] < d[i][j]) {
          d[i][j] = d[i][k] + d[k][j];
          prev[i][j] = prev[k][j];
        }
  for (int i = 0; i < V; i++)
    if (d[i][i] < 0)
      throw new IllegalStateException(
          "Negative cycle");
}
```

- **Lines 2–4:** three loops; the comments give the role of each loop: via, from, to.
- **Lines 5–7:** first it is checked that both parts are finite. Since `INF` is `Integer.MAX_VALUE`, the sum of two infinities, or of an infinity and a number, overflows and can wrap to a negative number. The check prevents this overflow.
- **Lines 8–9:** the new length is written. `prev[i][j] = prev[k][j]`: the vertex before j on the new path is the same as on the path $k \rightsquigarrow j$.
- **Lines 11–14:** if there is a negative value on the diagonal, an exception is thrown.

The code is traced on a graph with four vertices. The vertices are numbered 0–3. The edges are $0 \to 1$ (3), $0 \to 3$ (7), $1 \to 0$ (8), $1 \to 2$ (2), $2 \to 0$ (5), $2 \to 3$ (1), $3 \to 0$ (2).



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

prev

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

Figure 6.17: The graph of the `floydWarshall` trace and the matrices `d` and `prev` after four rounds.

k	Cell	Computation	New d	New prev
0	d[1][3]	d[1][0] + d[0][3] = 8 + 7 = 15 < ∞	15	prev[0][3] = 0
0	d[2][1]	d[2][0] + d[0][1] = 5 + 3 = 8 < ∞	8	prev[0][1] = 0

k	Cell	Computation	New d	New prev
0	d[3][1]	d[3][0] + d[0][1] = 2 + 3 = 5 < ∞	5	prev[0][1] = 0
1	d[0][2]	d[0][1] + d[1][2] = 3 + 2 = 5 < ∞	5	prev[1][2] = 1
1	d[3][2]	d[3][1] + d[1][2] = 5 + 2 = 7 < ∞	7	prev[1][2] = 1
2	d[0][3]	d[0][2] + d[2][3] = 5 + 1 = 6 < 7	6	prev[2][3] = 2
2	d[1][0]	d[1][2] + d[2][0] = 2 + 5 = 7 < 8	7	prev[2][0] = 2
2	d[1][3]	d[1][2] + d[2][3] = 2 + 1 = 3 < 15	3	prev[2][3] = 2
3	d[1][0]	d[1][3] + d[3][0] = 3 + 2 = 5 < 7	5	prev[3][0] = 3
3	d[2][0]	d[2][3] + d[3][0] = 1 + 2 = 3 < 5	3	prev[3][0] = 3
3	d[2][1]	d[2][3] + d[3][1] = 1 + 5 = 6 < 8	6	prev[3][1] = 0

Table 6.18: Cells that change in the `floydWarshall` trace (in order).

Eleven updates are made and eight cells change; some cells get shorter more than once: `d[1][0]` is 8 at first, falls to 7 at $k = 2$ and to 5 at $k = 3$. The final path is $1 \to 2 \to 3 \to 0$ and costs $2 + 1 + 2 = 5$. `d[1][3]` becomes 15 at first, then drops to 3. The long path found through 0 gives way to a much shorter one once 2 is added.

After four rounds there is no negative value on the diagonal and no exception is thrown. To read a path, `prev` is again used. $1 \dashrightarrow 3$: `prev[1][3] = 2`, `prev[1][2] = 1`. The path is $1 \to 2 \to 3$, of length `d[1][3] = 3`. The result matrix is `d = [[0, 3, 5, 6], [5, 0, 2, 3], [3, 6, 0, 1], [2, 5, 7, 0]]`.

Example 6.9

Find the final D matrix of the graph with the edges $1 \to 2$ (4), $2 \to 3$ (1), $1 \to 3$ (7), and $3 \to 1$ (−2). Is there a negative cycle?

1. ***k = 1.*** The edge into 1 is $3 \to 1$. $3 \dashrightarrow 2$: $-2 + 4 = 2$. $3 \dashrightarrow 3$: $-2 + 7 = 5$, greater than 0, unchanged.
2. ***k = 2.*** $1 \dashrightarrow 3$: $4 + 1 = 5 < 7$. $3 \dashrightarrow 3$: $2 + 1 = 3$, unchanged.
3. ***k = 3.*** $1 \dashrightarrow 1$: $5 + (-2) = 3$, unchanged. $2 \dashrightarrow 1$: $1 + (-2) = -1$. $2 \dashrightarrow 2$: $1 + 2 = 3$, unchanged.
4. **Result.** $D = [[0, 4, 5], [-1, 0, 1], [-2, 2, 0]]$. The diagonal is entirely 0; the weight of the cycle $1 \to 2 \to 3 \to 1$ is $4 + 1 - 2 = 3$, and that of $1 \to 3 \to 1$ is $7 - 2 = 5$. There is a negative edge but no negative cycle.

The structure of Floyd-Warshall is used beyond shortest paths. If only the information "is there an edge?" is kept instead of weights, "and" is written in place of addition and "or" in place of minimum. The resulting algorithm computes the **transitive closure**, Warshall's original problem: *t[i][j]* tells whether there is a path from i to j.

```
boolean[][] closure(boolean[][] t) {
  int V = t.length;
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        t[i][j] = t[i][j] || (t[i][k] && t[k][j]);
  return t;
}
```

The code again takes $\Theta(V^3)$ and k again stays outermost. On a graph with four vertices and the edges $0 \to 1$, $1 \to 2$, $2 \to 0$, and $3 \to 2$, all rows come out as `1110`. Vertices 0, 1, and 2 are on the same cycle and reach one another. Vertex 3 also enters this cycle through 2. Vertex 3 itself cannot be reached from any vertex; so the last column is 0 from top to bottom.

6.5 A* search

Dijkstra spreads from the source in all directions at the same speed. If we are going to only one goal, most of this spreading is wasted. A Dijkstra looking for the road from Istanbul to Ankara examines the roads to Edirne and Bursa too until it finds Ankara. A* (A star) search reduces this waste. For each vertex it estimates how far is left to the goal and steers the search toward the goal.

Peter Hart, Nils Nilsson, and Bertram Raphael developed A* in 1968 at the Stanford Research Institute. Their aim was to plan the path of a mobile robot. Today it is one of the most widely used search methods for pathfinding of characters in games, in map applications, and in robotics.

6.5.1 f = g + h

A* combines the strengths of two searches. Dijkstra (called uniform-cost search in AI textbooks) looks only at the actual cost of arriving from the source. It is therefore correct, but does not know where the goal is. **Greedy best-first search**, on the other hand, looks only at the estimated cost remaining to the goal. It runs quickly toward the goal but does not care whether the path taken is expensive. A* uses three values for each vertex:

- ***g(n)*, the actual cost:** the cost of the path found from the start vertex to n. It is the same as *d[n]* in Dijkstra.
- ***h(n)*, the heuristic estimate:** an estimate of the cost remaining from n to the goal. The straight-line distance on a map and the Manhattan distance on a grid are such estimates.
- ***f(n) = g(n) + h(n)*:** the estimated total cost of a solution passing through n. At each step A* expands the vertex with the smallest f value.

Definition 6.5

A **heuristic function** assigns to each vertex n a number $h(n) \geq 0$, an estimate of the cost remaining to the goal. Let $h^*(n)$ be the true shortest distance from n to the goal. The heuristic estimate need not be exact; but the more accurate it is, the fewer vertices the search expands.

Selection criterion	Algorithm	Behavior
g(n) only	Dijkstra	spreads equally in all directions; correct but expands many vertices
h(n) only	greedy best-first	runs toward the goal; fast but does not guarantee the shortest path
g(n) + h(n)	A*	heads for the goal; guarantees the shortest path if h is admissible

Table 6.19: Behavior of the search by selection criterion.

The reason the sum works is this. Vertices that look close to the goal have a small h value and move ahead. But if a vertex was reached by an expensive path, g is large and the vertex stays in the background. The two terms balance each other: one answers "how far is left?", the other "what did it cost to get here?"

Example 6.10

There are four vertices in the open list; their (g, h) values are p (4, 7), q (6, 3), r (2, 10), u (8, 2). Which vertex is expanded?

1. **f values.** $f(p) = 11, f(q) = 9, f(r) = 12, f(u) = 10$.
2. **Choice.** The smallest f is q's; A* expands q.
3. **Comparison.** Dijkstra, looking only at g, would choose r (2); greedy search, looking only at h, would choose u (2). The choice of A* differs from both.

6.5.2 The steps of the algorithm

A* uses two lists. The **open list** is a priority queue holding the vertices that have been discovered but not yet expanded. The **closed list** holds the expanded vertices. To **expand** a vertex means to remove it from the open list and examine its neighbors.

1. The g value of the start vertex is set to 0 and the vertex is added to the open list. The g value of the other vertices is ∞.
2. The vertex with the smallest $f = g + h$ value is removed from the open list and put in the closed list. The selection criterion is neither g nor h alone, but their sum.
3. If the removed vertex is the goal, a solution has been found; the path is rebuilt through the predecessor pointers. The goal is finalized not at the moment it is added to the open list but at the moment it is removed from it.

4. Otherwise, for each neighbor v compute $g' = g(u) + w(u, v)$. If g' is smaller, the values of g, f, and the predecessor are updated and the neighbor is added to the open list. This is exactly the relaxation of Dijkstra. If v is in the closed list and a better g has been found, it is reopened (with a consistent h this never happens).
5. Return to the second step. If the open list becomes empty, there is no path to the goal.

In the second step the selection is made from the whole open list, not just from the neighbors of the last expanded vertex. This lets A* switch to another branch if the branch it is following becomes expensive. In the graph example this is exactly what happens.

Common mistake

Stopping the search at the moment the goal is added to the open list. At the moment of addition the g value of the goal is only an upper bound; a cheaper path may be waiting in the open list. In Dijkstra too, the distance of a vertex is finalized when it leaves the queue, not when it enters. A* applies the same rule to the goal.

6.5.3 Properties of the heuristic function

Whether A* finds the shortest path depends on the heuristic. Two properties matter.

Definition 6.6

If $h(n) \le h^*(n)$ for every n, then h is **admissible**: it never exceeds the true remaining cost.

If $h(u) \le w(u, v) + h(v)$ for every edge (u, v) and $h(goal) = 0$, then h is **consistent**.

Theorem 6.4 (Optimality of A*)

If h is admissible, the path A* has found when it removes the goal from the open list is a shortest path.

Proof. Suppose the goal t is removed via a path with $g(t) > h^*(s)$, that is, a path that is not shortest. Let P be a shortest path from s to t. Before t is removed, one of the vertices on P, say n, is in the open list with a correct $g(n)$; this is shown by a simple induction starting from the first expansion of s. Because h is admissible, $f(n) = g(n) + h(n) \le g(n) + h^*(n) = h^*(s)$. At the goal $h(t) = 0$, so $f(t) = g(t) > h^*(s) \ge f(n)$. At the moment t is chosen, the open list contains n, which has a smaller f value. Since A* always picks the vertex with the smallest f, this is impossible. Contradiction. □

Consistency is a stronger property. A consistent heuristic is always admissible. With a consistent h, when a vertex enters the closed list its g value is final and the vertex is never reopened. This is the exact counterpart of finalization in Dijkstra. Indeed, with a consistent h, A* is Dijkstra running on a

graph with edge weights $w(u, v) - h(u) + h(v) \geq 0$. With an admissible but inconsistent heuristic, a vertex can later be reopened through a cheaper path.

Example 6.11

Chain graph $s \to a \to t$; $w(s, a) = 2$, $w(a, t) = 3$. Two heuristics are to be examined: (1) $h(s) = 4$, $h(a) = 4$, $h(t) = 0$; (2) $h(s) = 4$, $h(a) = 1$, $h(t) = 0$.

1. **True remaining costs.** $h^*(s) = 5$, $h^*(a) = 3$, $h^*(t) = 0$.
2. **First heuristic.** $h(a) = 4 > 3 = h^*(a)$: it exceeds the true remaining cost. It is not admissible.
3. **Second heuristic, admissibility.** $4 \leq 5$, $1 \leq 3$, $0 \leq 0$: it is admissible.
4. **Second heuristic, consistency.** On the edge (s, a) we ask whether $h(s) \leq w(s, a) + h(a)$: $4 \leq 2 + 1 = 3$ is false. The heuristic is admissible but not consistent. In going from s to a the estimate drops by 3 at once, more than the edge of 2 units.

In searches on a grid, the common heuristics are chosen by the movement rule:

- **4-direction movement** (up, down, right, left; each step 1): the **Manhattan distance** $|\Delta x| + |\Delta y|$. Each step reduces this sum by at most 1; hence $h \leq h^*$.
- **8-direction movement** (straight step 10, diagonal step 14): the **octile distance** $14 \cdot \min(|\Delta x|, |\Delta y|) + 10 \cdot (\max - \min)$. First go diagonally as far as possible, then straight. The number 14 is $10 \cdot \sqrt{2} \approx 14.14$ rounded to an integer.
- **Free plane:** the **Euclidean distance** $\sqrt{(\Delta x^2 + \Delta y^2)}$, that is, the straight line.

The Manhattan distance is not admissible with 8-direction movement. A diagonal step reduces the sum $|\Delta x| + |\Delta y|$ by 2 but costs only 1 (or 1.4). For a goal with $\Delta x = \Delta y = 3$, for example, the Manhattan distance is 6. On an 8-direction grid with every step costing 1, the true remaining cost is only 3, with three diagonal steps. The heuristic doubles the true value.

The quality of a heuristic is understood through two extreme cases. If $h = 0$ is taken, $f = g$ and A* turns into exactly Dijkstra. This heuristic is admissible but gives the search no direction. If $h = h^*$ (a perfect estimate) is taken, every vertex on the shortest path has f equal to $h^*(s)$, while vertices off the path have larger f. If ties are broken suitably, only the vertices on the path are expanded. A heuristic with $h > h^*$, that is, one that overestimates, can speed up the search but removes the guarantee of a shortest path.

6.5.4 Complexity

The cost of A* is described through the search tree. Let b be the **branching factor**, that is, the average number of neighbors of a vertex. Let d be the depth of the goal, that is, the number of steps in the solution. In the worst case the search expands all vertices down to depth d. Time and memory are therefore $O(b^d)$. This bound is that of a useless heuristic, for example $h = 0$. With a perfect estimate the search is linear in the path length: only the d vertices on the path are expanded.

On a finite graph with a consistent h, A* does no more work than Dijkstra. Each vertex is expanded at most once and the total time with a binary heap is $O((V + E) \log V)$. In practice a good heuristic stays far below this bound, because the vertices in the direction opposite the goal are never opened.

The real weak point of A* is memory. The open and closed lists grow with the search and can fill memory on large maps. Memory-bounded variants have been developed to solve this problem. IDA* (iterative deepening A*) repeats depth-first search with a rising f limit and keeps only the current path in memory. SMA* (simplified memory-bounded A*) drops the worst-looking vertices from the list when memory fills up.

6.5.5 Example: A* on a graph

The example is an undirected graph; the start is s and the goal is t. The edges are s–a 1.5; s–d 2; a–b 2; b–c 3; c–t 4; d–e 3; e–t 2. The heuristic values are $h(s) = 5$, $h(a) = 4$, $h(b) = 2$, $h(c) = 4$, $h(d) = 4.5$, $h(e) = 2$, $h(t) = 0$.

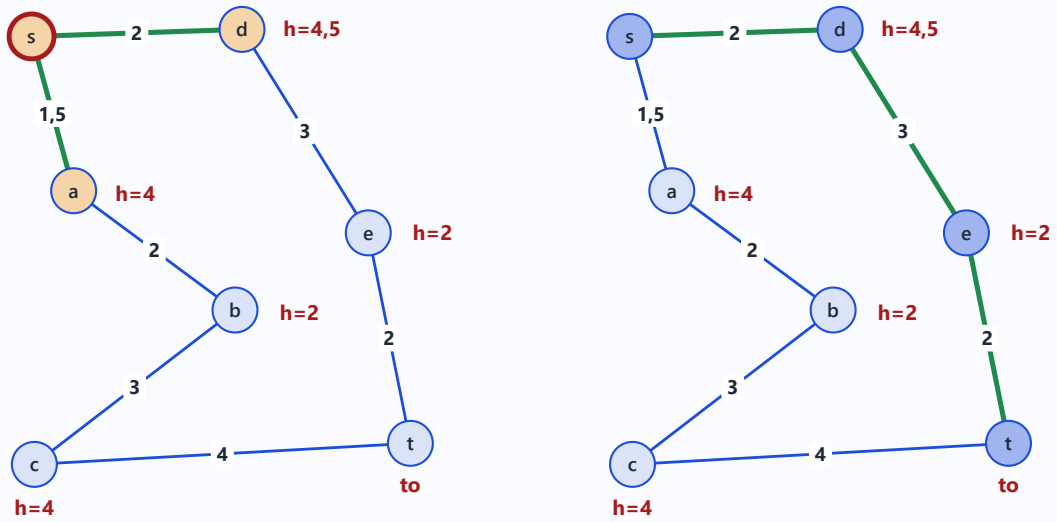


Figure 6.18: The A* example. Left, after the first expansion: s is closed, a and d are open. Right, the final state: the shortest path is $s \to d \to e \to t$ (7). Vertex c is never expanded.

Step	Expanded (f)	Opened / updated	Open list: vertex (g, h, f)	Closed
1	s (5)	a: g = 1,5; d: g = 2	a (1,5; 4; 5,5), d (2; 4,5; 6,5)	s
2	a (5,5)	b: g = 3,5	b (3,5; 2; 5,5), d (2; 4,5; 6,5)	s, a
3	b (5,5)	c: g = 6,5	d (2; 4,5; 6,5), c (6,5; 4; 10,5)	s, a, b
4	d (6,5)	e: g = 5	e (5; 2; 7), c (6,5; 4; 10,5)	s, a, b, d
5	e (7)	t: g = 7	t (7; 0; 7), c (6,5; 4; 10,5)	s, a, b, d, e
6	t (7)	goal removed, stop	c (6,5; 4; 10,5)	s, a, b, d, e, t

Table 6.20: Open and closed lists in the A* example. In the open list the f values are sorted from smallest to largest.

The search first heads for the upper branch. The f value of a is smallest at 5.5, and then b also moves ahead with 5.5. When b is expanded, c is opened, but the g value of c has risen to 6.5 and $f(c) = 10.5$. This branch has become expensive. In the open list the smallest f is now that of d (6.5), and the search switches to the lower branch. This switch is a consequence of the selection being made from the whole open list. From d, e is opened ($f = 7$), and from e, t is opened ($g = 7, f = 7$).

When t is removed from the open list, the search stops. The shortest path is $s \to d \to e \to t$ with cost $2 + 3 + 2 = 7$. The upper branch $s \to a \to b \to c \to t$ costs $1.5 + 2 + 3 + 4 = 10.5$. Vertex c has been opened but never expanded. Six of the seven vertices have been expanded. In the same graph Dijkstra would expand the vertices in order of distance: s, a, d, b, e, c, t; since the distance of c (6.5) is smaller than that of t (7), it would expand c too.

6.5.6 Java code and a trace

The Java code of A* is very similar to that of Dijkstra. The differences are in two places: the queue is ordered by $g + h$ and the search stops when the goal is removed. `g`, `h`, and `parent` are class fields; `g` and `h` are `double` arrays. `adj[u]` is the edge list of u, and each edge has the fields `v` (neighbor) and `w` (weight).

```
double aStar(int s, int t) {
  Arrays.fill(g, INF); g[s] = 0;
  Queue<Integer> open = new PriorityQueue<>(
    Comparator.comparingDouble(v -> g[v] + h[v]));
  open.add(s);
  while (!open.isEmpty()) {
    int u = open.poll();      /* smallest f */
    if (u == t) return g[t];  /* goal */
    for (Edge e : adj[u])
      if (g[u] + e.w < g[e.v]) {
        g[e.v] = g[u] + e.w; parent[e.v] = u;
        open.remove(e.v); open.add(e.v);
      }
  }
  return INF;
}
```

- **Line 2:** all g values are set to infinity and that of the start to 0.
- **Lines 3–5:** the open list is a priority queue. The comparator orders each vertex `v` by the value `g[v] + h[v]`. At the start only `s` is in it.
- **Lines 7–8:** the vertex with the smallest f is removed. If it is the goal, `g[t]` is returned. The stopping condition is checked at removal time.
- **Lines 9–13:** each edge is relaxed. If g decreases, the predecessor is written to `parent`. The vertex is first removed from the queue and then added again.
- **Line 15:** if the queue empties, there is no path to the goal.

The pair `remove` and `add` on line 12 is important. The comparator reads the array `g[v]`. If `g[v]` changes while a vertex is in the queue, the heap's ordering is broken, because the heap placed the vertex according to its old value. Removing the vertex and adding it again puts it in the right place for its new value. `PriorityQueue.remove(Object)` takes linear time; for large graphs lazy deletion, as in Dijkstra, is preferred.

When the code is run on the graph of the A* example, the `poll()` order is s ($f = 5$), a (5.5), b (5.5), d (6.5), e (7), t (7). The g updates at each step are the same as the values in Table 6.20. At the sixth call the goal is removed and `return g[t]` returns 7. Reading the `parent` array backward gives $t \leftarrow e \leftarrow d \leftarrow s$, that is, $s \to d \to e \to t$. Vertex c stays in the open list with $f = 10.5$ and is never removed.

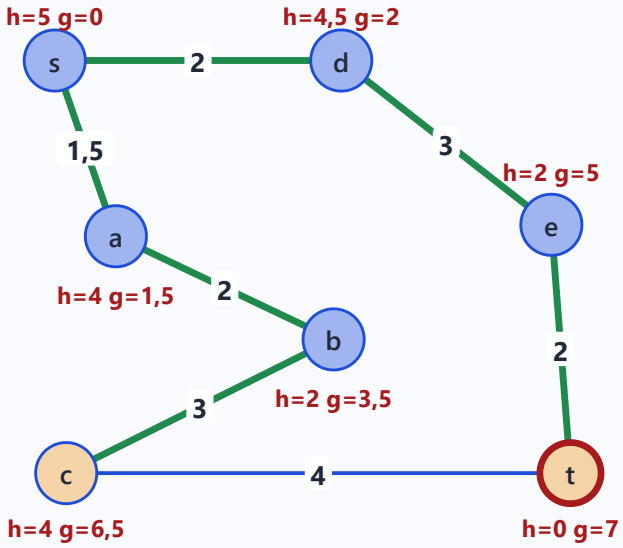


Figure 6.19: The final state of the `aStar(s, t)` trace. The vertex labels show h and the g found; the boxed t is the goal that was removed.

Example 6.12

A directed graph: $s \to a$ (1), $a \to t$ (9), $s \to b$ (2), $b \to c$ (2), $c \to t$ (2). The heuristic: $h(s) = 5$, $h(a) = 1$, $h(b) = 4$, $h(c) = 2$, $h(t) = 0$. Compare greedy best-first search with A*.

1. **Greedy search.** After s, a ($h = 1$) looks more attractive than b ($h = 4$). From a, t is opened ($h = 0$) and chosen at once. The path is $s \to a \to t$, cost 10.
2. **A*.** $f(a) = 1 + 1 = 2$, $f(b) = 2 + 4 = 6$. a is expanded; t is opened with $g = 10$, $f = 10$. Then b (6) and c ($f = 4 + 2 = 6$) are expanded; from c the g value of t drops to 6. t ($f = 6$) is removed: the path is $s \to b \to c \to t$, cost 6.
3. **Result.** This h is admissible, so A* finds the shortest path. Greedy search is fast, but because it ignores g it picks the expensive path.

6.5.7 Example: around a wall on a grid

The most familiar use of A* is on grid maps. In the example there is a grid of 8 columns and 7 rows. The start is A and the goal is B. Between them stands a wall in the shape of an inverted L. Movement is 8-directional: a straight step costs 10 and a diagonal step 14. The heuristic is the octile distance and ignores the walls. A and B differ by 3 columns and 3 rows; $h(A) = 3 \cdot 14 = 42$.

In the figures each cell carries three numbers: g at top left, h at top right, and f in the middle.

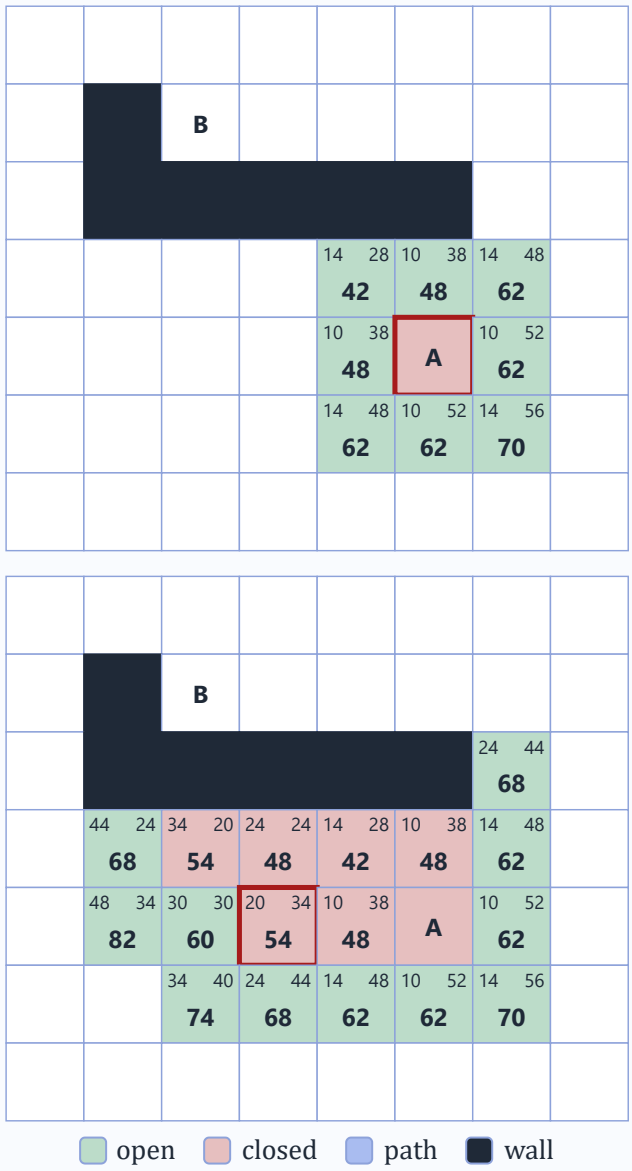


Figure 6.20: A* on a grid. Left, the first expansion: the eight neighbors of A are opened; the smallest f is 42 (straight toward B, the cell below the wall). Right, after the fourth expansion: the cells below the wall are tried first, with $f = 42 \ldots 54$.

The search first heads for the cells right below the wall, in the direction of B. Because the heuristic cannot see the wall, these cells look the most promising. The *f* values of the expanded cells, in order, are:

> 42, 42, 48, 48, 48, 54, 54, 60, 62, 62, 62, 62, 68, 68, 68, 68, 68, 68

The values never decrease. This is a property of a consistent heuristic: the *f* values of expanded vertices form a nondecreasing sequence. The path below the wall is blocked; to get over the wall one must go around, either from the right or from the left. The cost of going around is not in the estimate, and *f* rises step by step. At the thirteenth expansion *f = 68* is reached and the search spreads to both sides of the wall. At the seventeenth expansion the path going around the right side of the wall reaches B; B enters the open list with *f = 68*.

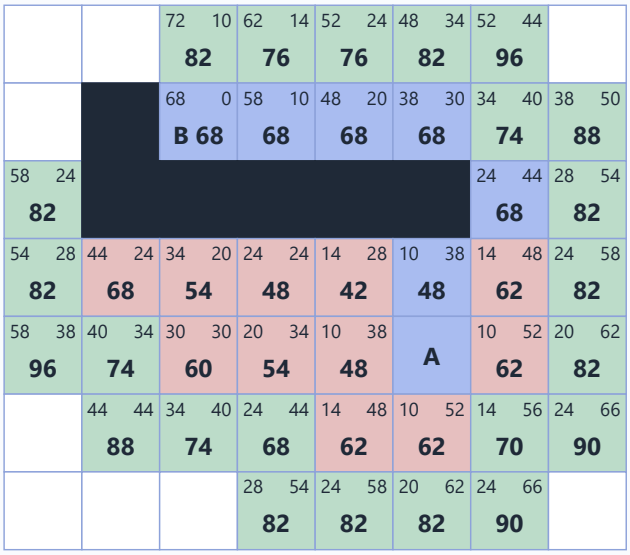


Figure 6.21: The final state when B is removed. The blue cells are the path found: one step up from A, diagonally up and right, diagonally up and left, and three steps left over the top of the wall. The cost is *2 · 14 + 4 · 10 = 68*.

B is removed at the eighteenth expansion and the search stops. The cost of the path found is 68: two diagonal steps (28) and four straight steps (40). Of the eighteen expansions, eleven are closed (pink) cells and seven are cells on the path. The cheapest path going around the left of the wall costs 96. A* has opened the cells of the leftmost column in that direction but has expanded none of them. A few cells in the corners of the grid have never been opened at all.

In the example, diagonal steps can graze the corner of the wall. Both diagonal steps of the path pass the corner of a wall cell. In many games this "corner cutting" is forbidden: a diagonal step is allowed only if the two straight neighbors beside it are also free. Such a rule can change the result of the search, but it does not change how A* works.

Common mistake

Using the Manhattan distance on an 8-direction grid. The heuristic exceeds the true remaining cost, and A* takes the first path it finds to be the shortest. The search seems to speed up, but a shorter diagonal path can be missed. Octile distance should be used with 8 directions, Euclidean distance in the free plane.

6.6 Shortest paths in unweighted graphs: BFS

If all edge weights are equal, for example all 1, the length of a path equals its number of edges. The shortest path is the path with the fewest edges. In this special case heavy algorithms are not needed. **Breadth-first search** (BFS) solves the problem in linear time.

BFS visits the vertices layer by layer, in order of distance from the source. First come the neighbors of the source (distance 1), then their unvisited neighbors (distance 2), and so on. When a vertex v is first found from u, $d[v] = d[u] + 1$ and $\pi[v] = u$ are written. A vertex has reached its shortest distance the moment it is first found; paths found later cannot be shorter.

The relation of BFS to Dijkstra is clear. Dijkstra chooses the vertex with the smallest distance with a priority queue. When the weights are equal, vertices enter the queue in order of distance. An ordinary **queue** working by the first-in, first-out (FIFO) principle already preserves this order, and the priority queue is no longer needed. Since queue operations are $O(1)$, the total time is $O(V + E)$; the $\log V$ factor of the heap disappears.

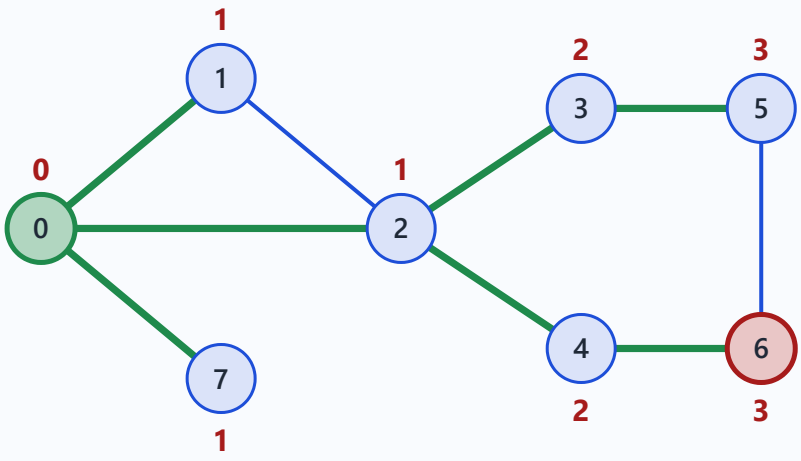


Figure 6.22: BFS on an unweighted graph. The numbers are the BFS level, that is, the number of edges from the source 0. The green edges are the BFS tree. The shortest path from 0 to 6 is $0 \to 2 \to 4 \to 6$ (3 edges).

In the graph in the figure the edges are 0–1, 0–2, 0–7, 1–2, 2–3, 2–4, 3–5, 4–6, and 5–6. The neighbors 1, 2, and 7 of 0 are at level one. The neighbors 3 and 4 of 2 are at level two. The neighbor 5 of 3 and the neighbor 6 of 4 are at level three. Another path to 6 is $0 \to 2 \to 3 \to 5 \to 6$, but it has four edges.

BFS finds 6 first from 4 and writes $\pi[6] = 4$; the path through 5 is seen later and changes nothing.

BFS also suffices if all edge weights are the same constant c. The distances are c times the levels. In the same graph with all edges 5, for example, the distance of 6 is $3 \cdot 5 = 15$. If the weights differ, BFS errs; the path with the fewest edges may not be the shortest path.

When there is a single goal, a variant of BFS is faster. **Bidirectional search** starts BFS at the same time from the source and from the goal. When the two waves meet in the middle, the search stops. In a search with branching factor b and goal depth d, one-directional BFS expands about b^d vertices. Bidirectional search expands about $2 \cdot b^{d/2}$. For $b = 10$ and $d = 6$ this means two thousand vertices instead of a million.

6.7 Comparison and choice of algorithm

The five algorithms of this chapter use the same basic operation, relaxation. Their differences lie in which problem they solve, which restrictions they accept, and how long they take. Table 6.21 gathers these differences in one table.

Algorithm	Problem	Negative edges	Negative cycles	Time	Method
BFS	single source, unweighted	—	—	$O(V + E)$	layer by layer
Dijkstra	single source	no	no	$O(V^2)$ / $O((V + E) \log V)$	greedy
Bellman-Ford	single source	yes	detects	$O(V \cdot E)$	repeated relaxation (DP)
Floyd-Warshall	all pairs	yes	detects	$\Theta(V^3)$	dynamic programming
A*	single pair	no	no	worst case $O(b^d)$	heuristic + greedy

Table 6.21: Comparison of the shortest-path algorithms.

The choice is made with a few questions:

1. **Is the graph unweighted?** BFS is enough.
2. **Are all weights zero or positive?** Use Dijkstra. If there is a single goal and a good heuristic, A* expands fewer vertices.
3. **Are there negative edges?** Use Bellman-Ford; this also checks for negative cycles.
4. **Are all pairs wanted and is V small?** Use Floyd-Warshall.

Example 6.13

Choose an algorithm for four situations.

1. **The fewest intermediaries between two people in a social network.** The edges are unweighted: BFS, $O(V + E)$.
2. **The shortest route between two addresses on a map.** Lengths are positive, there is a single goal, and straight-line distance is an admissible heuristic: A*.
3. ***V = 1000*, *E = 5000*, some edges negative.** Dijkstra is unreliable; use Bellman-Ford. About $V \cdot E = 1000 \cdot 5000 = 5 \cdot 10^6$ relaxation attempts are made.
4. **All distances between 200 cities.** All pairs are wanted and V is small: Floyd-Warshall, $200^3 = 8 \cdot 10^6$ operations.

Example 6.14

A directed graph: $A \to B$ 4, $A \to C$ 2, $C \to B$ 1, $B \to D$ 5, $C \to D$ 8, $C \to E$ 10, $D \to E$ 3; source A. Run Dijkstra and Bellman-Ford on this graph.

1. **Dijkstra.** A is finalized: B = 4, C = 2. C is finalized: B improves to $2 + 1 = 3$; D = 10, E = 12. B is finalized: D drops to $3 + 5 = 8$. D is finalized: E drops to $8 + 3 = 11$. Finally E is finalized. The order is A, C, B, D, E; the distances are A: 0, B: 3, C: 2, D: 8, E: 11.
2. **Path.** $\pi[E] = D$, $\pi[D] = B$, $\pi[B] = C$, $\pi[C] = A$. The path is $A \to C \to B \to D \to E$, of length $2 + 1 + 5 + 3 = 11$.
3. **Bellman-Ford, alphabetical edge order** ($A \to B$, $A \to C$, $B \to D$, $C \to B$, $C \to D$, $C \to E$, $D \to E$). In round 1 five relaxations occur: B = 4, C = 2, D = 9, B = 3, E = 12. In round 2 two relaxations occur: D = 8, E = 11. In rounds 3 and 4 nothing changes; early stopping is possible. The result is the same as Dijkstra's.

6.8 Common mistakes

The most important of the mistakes given in boxes throughout the chapter are collected here.

- **Dijkstra with a negative edge.** A shorter path can later open to a finalized vertex; the result comes out wrong without any warning. With a negative edge Bellman-Ford is needed. Adding a constant to the weights is not a solution either; paths with many edges are penalized more.
- **The overflow of ∞ + w.** `Integer.MAX_VALUE + w` wraps to a negative number and looks "shorter". For this reason `d[u] != INF` is checked before adding. In Floyd-Warshall both parts must be checked.
- **The loop order of Floyd-Warshall.** The loop over intermediate vertices k must be outermost. The order `i`, `j`, `k` leaves some cells at ∞ or larger than necessary.
- **The number of rounds.** In Bellman-Ford all $V - 1$ rounds may be needed; the job may not finish in a single round. The check round is separate from these rounds.

- **Stopping at the goal in A*.** One stops when the goal is removed from the list, not when it is added to the open list. An h that overestimates can miss the shortest path.
- **Negative weight on an undirected edge.** An undirected negative edge is by itself a negative cycle through the path $u \to v \to u$. If an undirected graph has a negative weight, the shortest path is undefined.
- **Printing a path to an unreachable target.** The predecessor array never reaches the source. First check that the distance is finite.

6.9 Summary

- In a weighted graph the length of a path is the sum of its edge weights. The shortest path need not be the path with the fewest edges. $\delta(s, v)$ is the length of the shortest path from s to v.
- There are three kinds of problem: single-source (Dijkstra, Bellman-Ford, BFS), all-pairs (Floyd-Warshall), and single-pair (A*).
- The operation common to all the algorithms is relaxation: if $d[u] + w(u, v) < d[v]$, then $d[v]$ and $\pi[v]$ are updated. An estimate only decreases and never falls below the true distance.
- Every part of a shortest path is itself shortest (optimal substructure). If there is no negative cycle, a shortest path has at most $V - 1$ edges. If a negative cycle is reachable, the shortest path is undefined.
- Dijkstra finalizes the vertex with the smallest estimate and relaxes its neighbors. Weights must be nonnegative. It takes $O(V^2)$ with an array and $O((V + E) \log V)$ with a binary heap. A queue with lazy deletion skips stale records by the visited check.
- Bellman-Ford relaxes all edges for $V - 1$ rounds. It works with negative edges; an edge that still relaxes in round V shows a negative cycle reachable from the source. The time is $O(V \cdot E)$; with early stopping the best case is $O(E)$.
- Floyd-Warshall applies the recurrence $D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$ with k outermost. The time is $\Theta(V^3)$ and the memory $\Theta(V^2)$. $D[i][i] < 0$ shows a negative cycle.
- A* expands the vertex with the smallest $f = g + h$. With an admissible h it finds the shortest path; with a consistent h each vertex is expanded once. If $h = 0$ is taken, A* turns into Dijkstra.
- In an unweighted graph BFS finds the path with the fewest edges in $O(V + E)$ time.
- Paths are read backward from the target through the π (or Π) pointers; together the pointers form the shortest-path tree.

6.10 Key terms

shortest path distance weighted graph single-source shortest paths all-pairs shortest paths relaxation optimal substructure negative cycle Dijkstra's algorithm priority queue greedy algorithm Bellman-Ford algorithm dynamic programming Floyd-Warshall algorithm

A* search heuristic admissible heuristic breadth-first search (BFS)

6.11 Test Yourself

1. A directed graph: $s \to a$ 2, $s \to b$ 6, $a \to b$ 3, $a \to c$ 7, $b \to c$ 1, $c \to t$ 2, $b \to t$ 8. Source s. In what order does Dijkstra finalize the vertices? What are the final distances and the path from s to t?

   Answer

   The order is s, a, b, c, t. a = 2; b is first 6, then drops to $2 + 3 = 5$ through a; c = $5 + 1 = 6$ (it would be 9 through a); t = $6 + 2 = 8$ (it would be 13 directly from b). The path is $s \to a \to b \to c \to t$, of length 8.

2. Roughly how many operations do array-based Dijkstra and binary-heap Dijkstra make on a graph with $V = 10^5$ and $E = 4 \cdot 10^5$? How many relaxation attempts does Bellman-Ford make?

   Answer

   Array: $V^2 = 10^{10}$. Heap: $(V + E) \log_2 V \approx 5 \cdot 10^5 \cdot 16.6 \approx 8.3 \cdot 10^6$. Bellman-Ford: $V \cdot E = 4 \cdot 10^{10}$. Because the graph is sparse, heap-based Dijkstra is by far the fastest.

3. A chain graph $s \to a \to b \to c$ with all weights 1. How many rounds does Bellman-Ford with early stopping run if it examines the edges (a) in the order *(b, c), (a, b), (s, a)*, (b) in the order *(s, a), (a, b), (b, c)*?

   Answer

   (a) Each round advances only one edge: a is found in round 1, b in round 2, c in round 3. All $V - 1 = 3$ rounds are used. (b) All three are found in round 1; in round 2 nothing changes and the algorithm stops: 2 rounds. The result in both is *[0, 1, 2, 3]*.

4. Floyd-Warshall is run on the graph with the edges $0 \to 1$ (4), $1 \to 2$ (2), $2 \to 0$ (1). Which cell changes after round $k = 0$? What is the final D matrix?

   Answer

   $k = 0$: the only edge into 0 is $2 \to 0$; $D[2][1] = 1 + 4 = 5$. $k = 1$: $D[0][2] = 4 + 2 = 6$. $k = 2$: $D[1][0] = 2 + 1 = 3$. The final matrix is *[[0, 4, 6], [3, 0, 2], [1, 5, 0]]*. The diagonal is 0; the weight of the cycle is 7.

5. Floyd-Warshall is run on the graph with the edges $0 \to 1$ (1), $1 \to 2$ (−2), $2 \to 0$ (−1), $0 \to 2$ (4).

What is seen on the diagonal? What does the code do?

Answer

The weight of the cycle $0 \to 1 \to 2 \to 0$ is $1 - 2 - 1 = -2$. At the end of the algorithm negative values appear on the diagonal (−2, −2, and −4 with in-place updating). The check `d[i][i] < 0` is true on the first row and `IllegalStateException("Negative cycle")` is thrown. The other values of the matrix are meaningless.

6. From the final matrix Π of the Floyd-Warshall example, which is the path $3 \rightsquigarrow 1$? Does the length of the path agree with $D[3][1]$?

Answer

In row 3: $\Pi[3][1] = 2$, $\Pi[3][2] = 4$, $\Pi[3][4] = 3$. The path is $3 \to 4 \to 2 \to 1$, of length $2 + 1 + 3 = 6 = D[3][1]$.

7. A chain $s \to a \to t$ with $w(s, a) = 2$, $w(a, t) = 5$. Is the heuristic $h(s) = 6$, $h(a) = 5$, $h(t) = 0$ admissible? Is it consistent?

Answer

The true remaining costs are $h^*(s) = 7$, $h^*(a) = 5$, $h^*(t) = 0$. $6 \le 7$, $5 \le 5$: admissible. Consistency: $h(s) = 6 \le 2 + 5 = 7$ and $h(a) = 5 \le 5 + 0$; both hold, so the heuristic is consistent.

8. Can BFS be used on a graph in which all edge weights are 5? What is the distance of 6 in the graph of the BFS example?

Answer

It can. When all weights are equal, the shortest path is again the path with the fewest edges; the distance is 5 times the BFS level. Vertex 6 is at level three: $3 \cdot 5 = 15$.

9. In the Java code of Dijkstra the comparator is `(a, b) -> a[1] - b[1]`. What could happen if a record with distance `Integer.MAX_VALUE` were added to the queue?

Answer

The subtraction could overflow. For example, `Integer.MAX_VALUE - (-1)` wraps to a negative number and the comparator reports the wrong order. In the code of these notes only finite distances are added to the queue, so no problem arises. The safe way to write it is `Integer.compare(a[1], b[1])`.

10. Dijkstra is modified to also keep the number of shortest paths to each vertex. In the graph with the

edges $s \to a$ 1, $s \to b$ 1, $a \to c$ 1, $b \to c$ 1, $c \to t$ 1, $s \to t$ 3, $a \to t$ 2, what is `count[t]`?

> **Answer**
>
> *d[t] = 3* and four paths have this length: $s \to t$, $s \to a \to t$, $s \to a \to c \to t$, $s \to b \to c \to t$. The algorithm finds `count[c] = 2`; to t come 1 path from s, 1 from a, and 2 from c: `count[t] = 4`. The rule: when a shorter path is found the count starts over (`count[v] = count[u]`); when a path of equal length is found it is added (`count[v] += count[u]`).

The rule in the last question is implemented by the following code. The code does the selection by scanning an array and has been tried out in Java.

```
long[] countPaths(int V, List<List<int[]>> adj, int source, int[] dist) {
  long[] count = new long[V];
  boolean[] done = new boolean[V];
  Arrays.fill(dist, INF);
  dist[source] = 0; count[source] = 1;
  for (int round = 0; round < V; round++) {
    int u = -1;
    for (int x = 0; x < V; x++)
      if (!done[x] && dist[x] != INF && (u == -1 || dist[x] < dist[u])) u = x;
    if (u == -1) break;
    done[u] = true;
    for (int[] e : adj.get(u)) {
      int v = e[0], w = e[1];
      if (dist[u] + w < dist[v]) {        /* shorter path: restart count */
        dist[v] = dist[u] + w;
        count[v] = count[u];
      } else if (dist[u] + w == dist[v]) { /* new paths of equal length */
        count[v] += count[u];
      }
    }
  }
  return count;
}
```

The rule relies on the weights being positive. When a vertex is finalized, all shortest paths into it have come from vertices of smaller distance, that is, from vertices finalized earlier. Therefore `count[u]` is complete the moment `u` is finalized, and the correct value is passed on to the neighbors.

6.12 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 22 and 23: gives the properties of relaxation, the correctness proofs of Dijkstra and Bellman-Ford, Floyd-Warshall, and Johnson's algorithm for sparse graphs in detail.

- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 4.4: presents Java code for the shortest-path algorithms, Dijkstra with an indexed priority queue, and queue-based Bellman-Ford.
- S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach* (4th ed., 2020), Chapter 3: treats A*, admissible and consistent heuristics, IDA*, and memory-bounded search variants from the viewpoint of artificial intelligence.
- E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs" (1959): the original three-page paper of the algorithm; the same paper also solves the minimum spanning tree problem.

6.13 References

1. R. Bellman, "On a Routing Problem", *Quarterly of Applied Mathematics* 16, 1958.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapters 22–23).
3. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik* 1, 1959.
4. R. W. Floyd, "Algorithm 97: Shortest Path", *Communications of the ACM* 5(6), 1962.
5. P. E. Hart, N. J. Nilsson, B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Transactions on Systems Science and Cybernetics* 4(2), 1968.
6. S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., Pearson, 2020.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 4.4: Shortest Paths).
8. S. Warshall, "A Theorem on Boolean Matrices", *Journal of the ACM* 9(1), 1962.

Minimum Spanning Trees

A company wants to connect several cities with power lines. Laying a line between any two cities has a cost. It is enough for every city to be reachable from every other; two cities need not be joined directly. Which lines should the company build to make the total cost as small as possible? In mathematics this question is the minimum spanning tree problem. The chapter first introduces spanning trees, minimum spanning trees, and the two properties on which every algorithm rests: the cut property and the cycle property. It then turns to four greedy algorithms that solve the same problem. Kruskal adds edges from lightest to heaviest, and Prim grows a tree from a single vertex. Borůvka grows all components at once, while reverse-delete removes heavy edges. The disjoint-set structure that Kruskal needs for its cycle test gets a detailed treatment of its own. For each algorithm the notes give Java code, a step-by-step trace, and a cost analysis, and they close by comparing the four.

LEARNING OUTCOMES

Concepts
Define spanning trees and minimum spanning trees (MSTs); explain and justify why a spanning tree has $V - 1$ edges, and state the cut and cycle properties.

Kruskal and disjoint sets
Sort the edges and keep those that do not form a cycle; perform the cycle test with disjoint sets (find / union); trace the effect of path compression and union by rank.

Prim
Grow the tree from one vertex; trace the key and parent arrays step by step, both with an array scan and with a priority queue; explain how Prim differs from Dijkstra.

Borůvka, reverse-delete, and analysis
Apply component-based and deletion-based approaches; compare the complexity of the four algorithms and choose the one that suits the structure of the graph.

7.1 Spanning trees and minimum spanning trees

Graphs are the common language of problems built on objects and the connections between them. The graphs in these notes are **undirected**: an edge between u and v can be used in both directions. They are also **weighted**: each edge carries a number, which may stand for the length of a cable, the cost of a road, or the difference between two points. The number of vertices is written V and the number of edges E. The same letters can also denote the vertex and edge sets, depending on context, as in the notation $G = (V, E)$.

The notion of a tree is already familiar from binary search trees and heaps. There a tree is a rooted data structure. In graph theory a tree is a more general object: every **connected, acyclic** undirected graph is a tree. There is no root and no limit on the number of children. Between any two vertices of a tree there is exactly one path. If there were no path the graph would not be connected; if there were two different paths, together they would form a cycle.

7.1.1 The problem

By deleting some edges from a connected graph we can obtain a subgraph that still connects all the vertices but contains no cycle. Such a subgraph "spans" all the vertices of the graph.

Definition 7.1

Let $G = (V, E)$ be a connected, undirected graph. A **spanning tree** of G is a connected, acyclic subgraph of G that contains *all* of its vertices. A spanning tree uses only edges that exist in G.

The number of edges in a spanning tree is determined exactly by the number of vertices. Every algorithm in this chapter uses this fact in its stopping condition.

Theorem 7.1 (Number of edges in a tree)

Every tree on V vertices has exactly $V - 1$ edges. Adding a new edge to a tree creates exactly one cycle; deleting an edge from a tree splits it into two pieces.

Proof. Start with V vertices and no edges, so there are V separate pieces. Add the edges of the tree one at a time. If an edge joined two vertices of the same piece, a cycle would form, because a path between them already exists; a tree has no cycle, so this cannot happen. Hence every edge joins two different pieces and reduces the number of pieces by exactly one. At the end the tree is connected, so one piece remains. Going from V pieces down to one takes $V - 1$ edges. For the second claim, consider the unique path between u and v in the tree: the new edge (u, v) forms a cycle together with this path. When an edge is deleted, no other path remains between its endpoints, so the tree falls into two pieces. □

A graph usually has many spanning trees. When the edges are weighted, the total weights of these trees differ as well. What we are really after is the tree whose total is smallest.

Definition 7.2

In a connected, undirected graph with edge weights $w(u, v)$, the weight of a spanning tree is the sum of its edge weights: $w(T) = \Sigma_{(u,v) \in T} w(u, v)$. A spanning tree of smallest weight is called a **minimum spanning tree** (MST).

The word "minimum" describes the total weight, not the tree itself. Since every spanning tree has $V - 1$ edges, there is no search for a tree with fewer edges. Trees differ only in which edges they choose. Figure 7.1 shows a small graph on five vertices and its minimum spanning tree.

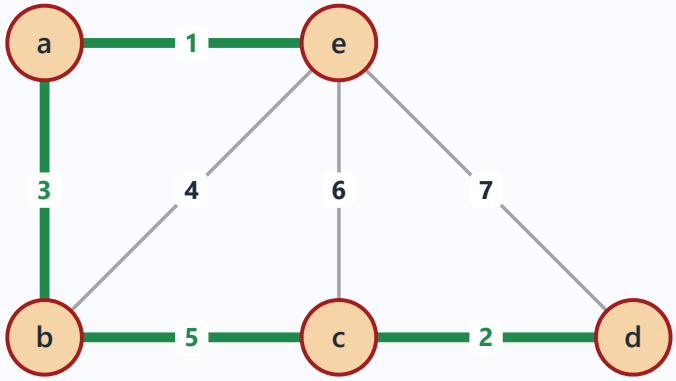


Figure 7.1: A weighted graph on vertices a, b, c, d, e. The bold edges ($ae = 1, cd = 2, ab = 3, bc = 5$) form the minimum spanning tree: $w(T) = 1 + 2 + 3 + 5 = 11$.

The graph in the figure has seven edges. A spanning tree uses four edges to connect five vertices; three edges are left out. The four bold edges form a tree: from a one can reach e and b, from b one can reach c, and from c one can reach d, and there is no cycle. Each of the omitted edges $be = 4$, $ec = 6$, and $ed = 7$ would close a cycle if added to the tree. For example, adding be creates the cycle $b \to a \to e \to b$.

Common mistake

Thinking of the MST as the "$V - 1$ lightest edges." The four lightest edges in the figure are $ae = 1$, $cd = 2$, $ab = 3$, and $be = 4$, with total weight 10. But these four edges do not form a tree: they close a cycle among a, b, e, and $\{a, b, e\}$ and $\{c, d\}$ are never joined. The chosen edges must be both acyclic and connected, and that is where the real difficulty lies.

7.1.2 How many spanning trees are there?

The crudest way to find an MST is to build every spanning tree, one by one, and compare their weights. The number of spanning trees shows why this does not work. A graph in which every pair of vertices is joined by an edge is called a **complete graph**. In 1889 Arthur Cayley showed that the complete graph on n vertices has exactly n^{n-2} spanning trees, a result known as **Cayley's formula**.

n	Number of spanning trees
3	3
4	16
5	125
10	100,000,000
15	about $1.95 \cdot 10^{15}$

Table 7.1: Number of spanning trees in the complete graph on n vertices, n^{n-2}.

Example 7.1

The complete graph on four vertices has six edges, and a spanning tree uses three of them. Three edges can be chosen from six in $C(6, 3) = 20$ ways. Four of these selections are triangles: any three of the four vertices form a cycle and leave the fourth vertex out. Every other selection of three edges is acyclic and connects all four vertices. The number of spanning trees is therefore $20 - 4 = 16$, and Cayley's formula gives $4^{4-2} = 16$ as well.

In a complete graph on fifteen vertices the number of spanning trees is in the quadrillions. Trying every tree is impractical even for small graphs. A graph that is not complete has fewer spanning trees, but the count still grows quickly: the seven-edge graph in Figure 7.1 has 21 spanning trees. Their weights range from 11 to 21, and only one tree has weight 11. Fortunately, finding the MST does not require counting trees. The problem is solved directly by **greedy** algorithms, which at each step make the choice that looks best locally. The cut and cycle properties explain why the greedy approach gives a correct answer here.

7.1.3 An example: the “MST” graph

A larger example helps show which edges a spanning tree selects and which it leaves out. The graph in Figure 7.2 draws the letters M, S, and T. It has 15 vertices and 20 edges.

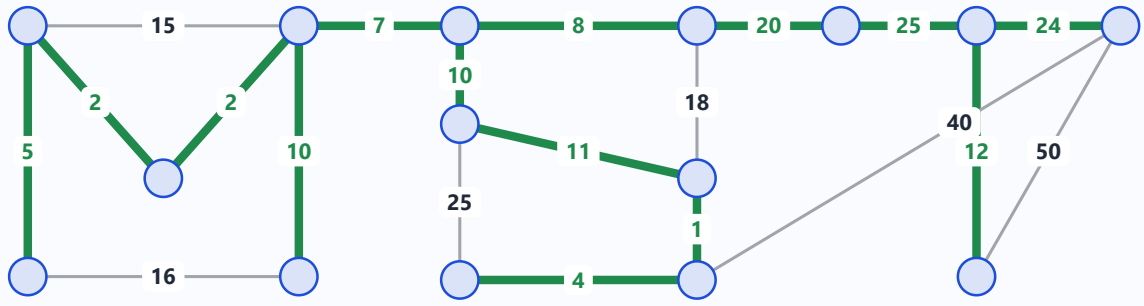


Figure 7.2: A graph with 15 vertices and 20 edges that draws the letters M, S, and T. The 14 bold edges form the minimum spanning tree; its total weight is 141. The thin edges (15, 16, 18, 25, 40, 50) are not in the tree.

There are 14 bold edges, which equals $V - 1 = 15 - 1$. The total weight is computed as follows:

> 2 + 2 + 5 + 10 + 7 + 8 + 10 + 11 + 4 + 1 + 20 + 25 + 24 + 12 = 141

The six thin edges that are left out share a property. Each of them closes a cycle together with bold edges, and it is the *heaviest* edge of that cycle. The edge of weight 15 above the letter M forms a triangle with the two diagonal edges of weight 2; the heaviest edge of the triangle is 15. The edge of weight 16 below the M closes a cycle with the two legs (5 and 10) and the two diagonals (2 and 2). The thin edge of weight 25 at the lower left of the S closes a cycle with bold edges of weight 11, 1, and 4. In every case the other edges of the cycle keep the connection intact in place of the thin edge, and they are cheaper.

Some heavy edges, on the other hand, are in the tree: 20, 24, and the 25 between S and T. None of these is the heaviest edge of any cycle. For example, the edge of weight 40 that runs from the lower right of the S to the upper right end of the T closes a long cycle. The edges of weight 20, 25, and 24 are also on this cycle, but the heaviest edge of the cycle is 40. Likewise, the edge of weight 50 forms a triangle with 24 and 12. The graph also has a **bridge**: deleting the edge of weight 7 that joins M and S would split the graph into two pieces. A bridge lies on no cycle, so it must be in every spanning tree and therefore in every MST.

Common mistake

Explaining the presence of a heavy edge in the tree by saying “this edge is the only way to reach that vertex.” The bold edges of weight 20, 24, and 25 in the figure are not bridges; the graph stays connected if any one of them is deleted. They are in the tree because every cycle that contains them also contains a heavier edge (40 or 50). An edge's weight alone says nothing; what matters is its position in cycles and cuts.

7.1.4 The cut property

All MST algorithms rest on two simple observations. The first concerns splitting the vertices into two groups.

Definition 7.3

A **cut** of a graph is a partition of its vertex set into two nonempty parts, written $(S, V - S)$. An edge with one endpoint in S and the other in $V - S$ is said to **cross the cut**. Among the edges crossing a cut, the one of smallest weight is the cut's **lightest edge**.

Theorem 7.2 (Cut property)

For every cut, the lightest edge crossing the cut belongs to some MST. If the lightest crossing edge is unique, it belongs to *every* MST.

Proof. Let $e = (u, v)$ be the lightest edge crossing the cut, with $u \in S$ and $v \in V - S$. Let T be an MST that does not contain e. Since T is a spanning tree, it contains a path from u to v. This path starts in S and ends in $V - S$, so it crosses the cut at least once. Call the edge where it crosses f. Deleting f from T and adding e gives a new tree T'. Deleting f splits the tree into two pieces, one containing u and the other containing v, and e reconnects them. So T' is also a connected spanning tree with $V - 1$ edges. Its weight is $w(T') = w(T) - w(f) + w(e)$. Since e is the lightest edge of the cut, $w(e) \le w(f)$, and therefore $w(T') \le w(T)$. Because T is minimal, so is T', and it contains e. If e is the unique lightest edge, then $w(e) < w(f)$ and $w(T') < w(T)$, a contradiction; so no MST avoids e. □

The idea of this proof is called an **exchange argument**. Take an optimal solution that does not contain the greedy choice. Modify one part of it, and it becomes a solution that contains the choice and is at least as good. The same idea underlies the correctness proofs of most greedy algorithms.

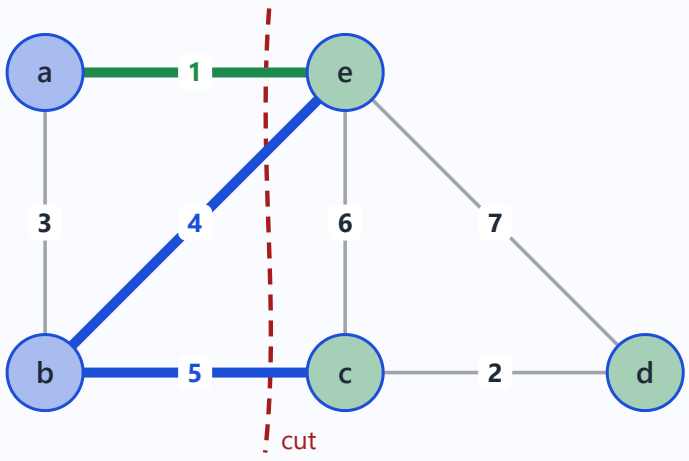


Figure 7.3: The cut $\{a, b\} \mid \{c, d, e\}$. The edges crossing the cut are $ae = 1$, $be = 4$, and $bc = 5$; the lightest, $ae = 1$, is

safe and belongs to the MST.

Example 7.2

Several cuts of the graph in Figure 7.1 can be examined.

1. $\{a, b\} \mid \{c, d, e\}$: the crossing edges are $ae = 1$, $be = 4$, $bc = 5$. The lightest is ae, which is in the MST.
2. $\{d\} \mid \{a, b, c, e\}$: the crossing edges are $cd = 2$ and $ed = 7$. The lightest is cd, which is in the MST.
3. $\{a, b, e\} \mid \{c, d\}$: the crossing edges are $bc = 5$, $ec = 6$, $ed = 7$. The lightest is bc. This edge is not among the lightest edges of the graph, yet it enters the MST.

The cut property speaks only about edges that cross the cut. The edge $ab = 3$ does not cross the first cut, so that cut says nothing about ab. That ab is in the MST is shown by another cut: the edges crossing $\{b\} \mid \{a, c, d, e\}$ are $ab = 3$, $be = 4$, and $bc = 5$, and the lightest is ab.

7.1.5 The cycle property

The second observation is in a sense the reverse of the cut property. The cut property tells us which edge can safely be chosen; the cycle property tells us which edge can safely be discarded.

Theorem 7.3 (Cycle property)

If the heaviest edge of some cycle in the graph is unique, that edge is in no MST.

Proof. Let $f = (u, v)$ be the unique heaviest edge of a cycle C, and suppose an MST T contains f. Deleting f from T splits the tree into two pieces, with u in one and v in the other. The edges of C other than f form a path from u to v. Since this path passes from one piece to the other, at least one of its edges, say e, joins the two pieces. Then $T - f + e$ is again a spanning tree, and because $w(e) < w(f)$, its weight is smaller than that of T. This contradicts the minimality of T. □

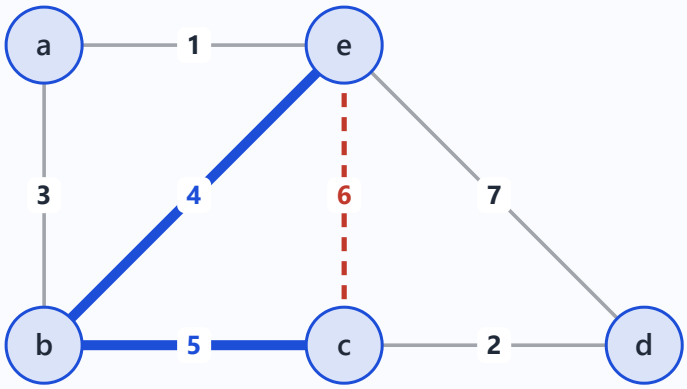


Figure 7.4: The edges of the cycle b–c–e have weights 4, 5, and 6. The heaviest edge, $ec = 6$, is in no MST.

The cycle property carries a small caveat: the heaviest edge must be *unique*. If the largest weight in a cycle appears on two edges, one of them may be in an MST. For example, in a triangle with edge weights 1, 2, and 2, one of the edges of weight 2 is always chosen. In such a case the property says only this: there is an MST that omits one of the heaviest edges.

Together the two properties give the skeleton of the algorithms. Table 7.2 summarizes which algorithm uses which property.

Property	What it says	Algorithms that use it
Cut property	The lightest edge crossing a cut can safely be added.	Kruskal, Prim, Borůvka
Cycle property	The heaviest edge of a cycle can safely be discarded.	Reverse-delete

Table 7.2: The two properties and the algorithms that use them.

7.1.6 When is the MST unique?

A graph can have more than one minimum spanning tree. This requires edges of equal weight. In a triangle with edge weights 1, 2, and 2, the edge of weight 1 together with either edge of weight 2 forms an MST. Both trees have weight 3. If all weights are distinct, no such freedom of choice remains.

Theorem 7.4 (Uniqueness of the MST)

If all edge weights of a connected graph are distinct, the MST is unique.

Proof. Let T be an MST and e any edge of T. Deleting e from T splits the vertices into two pieces, which define a cut. The only edge of T crossing this cut is e. If some crossing edge e' were lighter than e, then $T - e + e'$ would be a lighter spanning tree, which is impossible. Since the weights are distinct, e is the *unique* lightest edge of this cut. By the cut property, e is in every MST. This holds for all edges of T. So every MST contains the $V - 1$ edges of T and has exactly $V - 1$ edges itself; hence every MST equals T. □

The converse of the theorem is false: a graph with equal-weight edges can still have a unique MST. The weights in Figure 7.1 are already distinct, so that graph has a single MST. When equal-weight edges are present, different algorithms, or even the same algorithm with a different edge order, may find different trees. The total weight of the trees found is always the same.

7.1.7 Greedy MST algorithms

The cut property suggests a general method. At the start no edge is chosen. The chosen edges are grown so that at every moment they are part of some MST. At each step, find a cut that none of the chosen edges crosses, and add the lightest edge of that cut. Such an edge is called a **safe edge**: once it is added, the chosen set is guaranteed to be extendable to an MST. After $V - 1$ safe edges have been added, the tree is complete.

The four classical algorithms are different implementations of this general method. All of them are greedy: at each step they choose the locally best edge or discard the worst, and this decision is never revisited.

Algorithm	Idea	Helper structure	Time
Kruskal (1956)	Sort the edges from lightest to heaviest and add those that do not form a cycle	disjoint sets	$O(E \log E)$
Prim (1957)	Start at one vertex and grow the tree by the cheapest adjacent edge	priority queue	$O((V + E) \log V)$
Borůvka (1926)	Every component adds its own cheapest edge simultaneously	disjoint sets	$O(E \log V)$
Reverse-delete	Try the edges from heaviest to lightest and delete any whose removal keeps the graph connected	BFS/DFS or disjoint sets	$O(E \cdot (V + E))$

Table 7.3: Overview of the greedy MST algorithms.

All four algorithms find an MST of the same weight. If the weights are distinct, they find the same tree, because the MST is unique. They differ in the order in which they consider the edges and in the data structure they need. Kruskal and Borůvka work with a **forest**, that is, with several trees. Prim keeps a single tree at all times. Reverse-delete starts from the whole graph and thins it out. The years in the table are interesting too: the oldest algorithm is Borůvka's, written before computers existed, for a power-grid problem.

7.1.8 Applications

The MST is not a theoretical curiosity; it is either the whole of many engineering problems or one step in them.

- **Network design.** Connecting cities, buildings, or devices with as little cable as possible is directly an MST problem. Electricity, water, fiber-optic, and road networks fall into this class. Borůvka's 1926 work also started from the electrical network of Moravia.

- **Clustering.** Treat the points as vertices and the distances between them as weights. If the $k - 1$ heaviest edges are deleted from the MST, k pieces remain and each piece is a cluster. This method is called **single-linkage clustering**. In the nine-vertex Kruskal example, deleting the two heaviest edges of the MST yields the clusters *{A, B}*, *{C, D, E, F, H}*, and *{G, I}*.
- **Approximation algorithms.** In the metric traveling salesman problem (TSP), a tour obtained by a depth-first traversal of the MST that skips repeated vertices is at most twice as long as the best tour. An exact solution of the TSP is very expensive, whereas the MST is found quickly.
- **Image processing.** Pixels are vertices, and the color difference between neighboring pixels is the weight. MST-based methods segment an image by grouping regions of similar color together.
- **Circuit design.** Connecting the pins of a chip that must be wired together with the least total conductor length is a preliminary step of VLSI routing.
- **Other graph problems.** Among the paths between two vertices, the one whose heaviest edge is lightest is called the **bottleneck path**. The path between two vertices in an MST is always such a path. The MST is also a basic approximate solution for the Steiner tree problem.

7.2 Kruskal's algorithm

Kruskal's algorithm gives the most natural answer to the MST problem: take the cheapest edges first, as long as they do not form a cycle. Joseph Kruskal published it in 1956 in a short paper related to the traveling salesman problem. The idea of the reverse-delete algorithm appears in the same paper.

7.2.1 The idea: merging a forest

At the start every vertex is a tree by itself; the graph is a **forest** of V trees. The edges are sorted from lightest to heaviest and examined in that order. If the edge under examination connects vertices of two *different* trees, it is added and the two trees merge. If both endpoints are already in the same tree, adding the edge would create a cycle, so it is skipped. Each addition reduces the number of trees by one. After $V - 1$ additions a single tree remains and the algorithm stops.

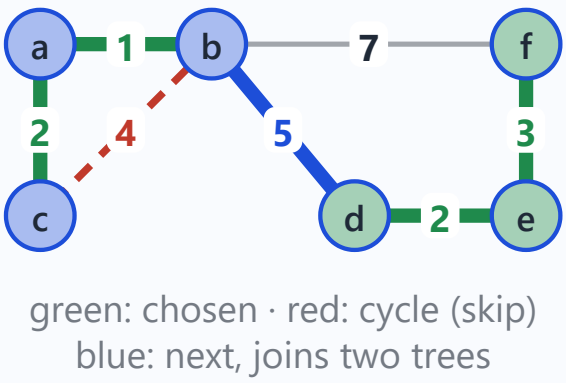


Figure 7.5: An intermediate state of Kruskal: a forest of two trees. The green edges are chosen. The red dashed edge of weight 4 connects two vertices of the same tree and would close a cycle, so it is skipped. The

blue edge of weight 5 joins the two trees and is added.

In the intermediate state shown, there are two trees, *{a, b, c}* and *{d, e, f}*. The next edge is *bc = 4*. A path $b \to a \to c$ already exists between *b* and *c*; adding *bc* would close a cycle with this path. The following edge, *bd = 5*, connects two separate trees and is added. At this point the forest becomes a single tree.

A brief assessment of the algorithm follows. Kruskal is well suited to **sparse** graphs, where the number of edges is close to the number of vertices. Its code is simple: an edge list and a disjoint-set structure are enough. If the graph is not connected, the algorithm produces a separate tree for each component, that is, a **minimum spanning forest**. It also has weak points. Sorting the edges costs *O(E log E)*. On **dense** graphs, where the number of edges approaches V^2, the array-scan version of Prim can be faster. An extra structure is needed for the cycle test.

7.2.2 Steps and pseudocode

The steps of the algorithm are as follows:

1. Sort all the edges of the graph in increasing order of weight.
2. From the sorted list, take the next edge of smallest weight.
3. If the edge joins two vertices that belong to different trees, add it to the tree and merge the two trees.
4. If the edge connects two vertices of the same tree, it would form a cycle and is skipped.
5. Repeat steps 2–4 until *V − 1* edges have been chosen (or the edges run out).

```
KRUSKAL(G):
  MST = [ ]
  disjointSet.create(V)
  edges.sort(by ascending weight)
  for each (u, v, weight) in edges:
    if disjointSet.find(u) ≠ disjointSet.find(v):
      MST.add((u, v, weight))
      disjointSet.union(u, v)
  return MST
```

The only mysterious part of the pseudocode is the comparison `disjointSet.find(u) ≠ disjointSet.find(v)`. `find` returns a representative that says which tree a vertex is in. If the representatives of the two endpoints differ, the edge connects two separate trees. `union` gathers two trees into one. The question “are these two vertices in the same tree?” is answered by the disjoint-set structure in nearly constant time. The structure is treated in detail in the next section.

7.2.3 A worked example

Kruskal is applied to the five-vertex graph of Figure 7.1. The edges of the graph are *ab = 3*, *ae = 1*, *bc = 5*, *be = 4*, *cd = 2*, *ed = 7*, and *ec = 6*.

Example 7.3

1. **Start.** Every vertex is a separate tree: *{a} {b} {c} {d} {e}*. No edge is chosen, *w(T) = 0*.
2. **Sorting.** The edges are arranged by increasing weight: *ae 1, cd 2, ab 3, be 4, bc 5, ec 6, ed 7*.
3. ***ae = 1.*** *a* and *e* are in different trees, so the edge is added. The trees are *{a, e} {b} {c} {d}*, *w(T) = 1*.
4. ***cd = 2.*** *c* and *d* are in different trees, so the edge is added. The trees are *{a, e} {c, d} {b}*, *w(T) = 3*.
5. ***ab = 3.*** *a* and *b* are in different trees, so the edge is added. The trees are *{a, b, e} {c, d}*, *w(T) = 6*.
6. ***be = 4.*** *b* and *e* are already in the same tree. Adding the edge would create the cycle *b → a → e → b*, so it is skipped.
7. ***bc = 5.*** *b* and *c* are in different trees, so the edge is added. One tree remains: *{a, b, c, d, e}*, *w(T) = 11*.

V − 1 = 4 edges have been chosen, so the algorithm stops. There is no need to look at *ec = 6* and *ed = 7*. The result is the tree of Figure 7.1.

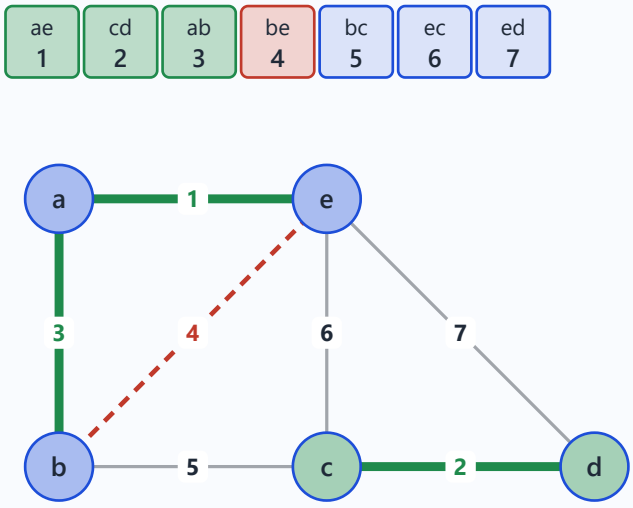


Figure 7.6: The step *be = 4*. The table above shows the sorted edges: green ones are chosen, the red one is rejected, and the blue ones have not been examined yet. Because *b* and *e* are in the same tree (blue vertices), *be* would close a cycle.

The only edge rejected in the trace is *be*. It is the fourth-lightest edge of the graph in Figure 7.1. Kruskal skips it not because of its weight but because of its position: *b* and *e* are already connected by two cheaper edges.

Common mistake

Performing the cycle test by asking "is there a chosen edge between *u* and *v*?" In the example, at the step *be* no edge between *b* and *e* has been chosen; yet the two are connected through *a*, and *be* closes a cycle. The right question is "are *u* and *v* in the same component?" The length of the path between them is irrelevant.

7.2.4 Why is Kruskal correct?

Theorem 7.5 (Correctness of Kruskal)

On a connected graph, the edges chosen by Kruskal's algorithm form a minimum spanning tree.

Proof. We show that every edge the algorithm adds is safe. When Kruskal decides to add an edge $e = (u, v)$, the vertices u and v are in different trees. Let S be the tree of u and consider the cut $(S, V - S)$. None of the chosen edges crosses this cut, because S is an entire tree. Every edge that crosses the cut joins two different trees, so none of these edges was skipped earlier on the grounds of forming a cycle. If a cut edge lighter than e existed, it would come before e in the sorted list and, since it joined two different trees at that moment as well, it would have been added. Hence e is the lightest edge of the cut and is safe by the cut property. The skipped edges, on the other hand, form cycles and so cannot belong to any tree. Since the graph is connected, $V - 1$ edges are chosen in the end, and they form an MST. □

If the weights are distinct, the sorted edge list is unique and Kruskal always finds the same tree. When there are equal-weight edges, the sort may arrange them in different orders. In that case the resulting tree may change, but its weight does not: the proof holds for whichever order is chosen.

7.3 Disjoint sets (union-find)

The speed of Kruskal depends on how quickly the question "are two vertices in the same tree?" can be answered. Searching for a path with BFS or DFS for every edge takes $O(V)$ and raises the total cost to $O(E \cdot V)$. The disjoint-set structure answers the same question in practically constant time.

7.3.1 Structure and operations

Definition 7.4

A **disjoint-set** (union-find) structure maintains a collection of non-overlapping sets and offers three operations:

- `create(V)` : every element forms its own singleton set.
- `find(x)` : returns the **representative** of the set containing *x*. The same representative is returned for all elements of one set.
- `union(x, y)` : merges the sets containing *x* and *y* into one.

The most common representation is a forest. Each set is a tree, and the root of the tree is the representative of the set. The forest is stored in a single parent array (`parent[]` in the code): `parent[i]` is the parent of *i*, and at a root `parent[r] = r` . These trees have nothing to do with the MST itself. They carry only the information “who is in which set?”, and their shapes may differ from the edges of the graph.

`create(V)` writes `parent[i] = i` for every vertex. `find(x)` follows parent links up to the root and returns the root. `union(x, y)` finds the two roots and attaches one under the other. In Kruskal the usage is as follows: if `find(u) == find(v)` , then *u* and *v* are in the same tree and the edge would close a cycle. The Java code from the slides is:

```
int find(int[] parent, int i) {
  if (parent[i] != i)
    parent[i] = find(parent, parent[i]); // path compression
  return parent[i];
}
void union(int[] parent, int x, int y) {
  parent[find(parent, x)] = find(parent, y);
}
```

`find` is recursive. If `parent[i] == i` , then *i* is a root and is returned itself. Otherwise the root of the parent is found recursively, and on the way back the root found is written into `parent[i]` . Besides the method's main job of finding the root, this assignment has a side effect: every vertex on the path is attached directly to the root. This improvement is called **path compression**. `union` is a single line: the parent of the root of *x* is set to the root of *y*. If the two elements are already in the same set, both calls to `find` return the same root and the assignment points the root at itself; the structure does not change.

7.3.2 The cycle test in Kruskal

How disjoint sets work inside Kruskal is traced on a five-vertex graph. The edges are sorted: *ac 1*, *de 2*, *ab 3*, *bc 4*, *bd 5*, *cd 6*, *ce 7*. For every accepted edge *(u, v)*, `parent[find(u)] = find(v)` is applied.

Edge	`find(u)`	`find(v)`	Decision	`parent`
start	—	—	—	a b c d e
ac = 1	a	c	add, `parent[a] = c`	c b c d e
de = 2	d	e	add, `parent[d] = e`	c b c e e
ab = 3	c	b	add, `parent[c] = b`	c b b e e
bc = 4	b	b	same root: cycle, skip	c b b e e
bd = 5	b	e	add, `parent[b] = e`	c e b e e

Table 7.4: The cycle test with disjoint sets in Kruskal. The last column gives the `parent` array in the order *a, b, c, d, e*.

At the step *ab*, `find(a)` first climbs from *a* to its parent *c*. Since *c* is a root, the root *c* is returned. `find(b)` returns *b* itself. The roots differ, so the edge is added and the root *c* is attached under *b*. At the step *bc*, `find(b) = b`, and `find(c)` also reaches *b* along the path $c \to b$. The roots are the same, so the edge would close a cycle. When four edges have been chosen, the MST is complete: *ac + de + ab + bd = 1 + 2 + 3 + 5 = 11*.

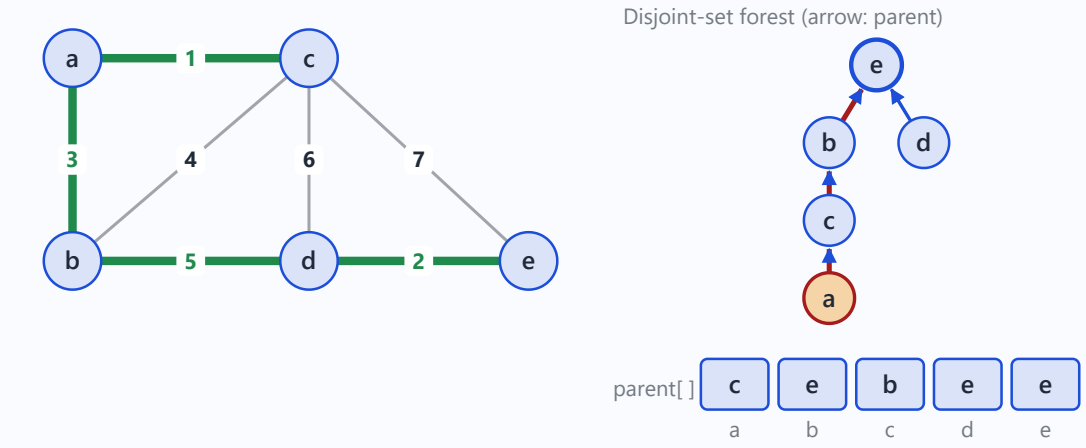


Figure 7.7: The disjoint-set forest after the MST is complete. The parent links have built the chain $a \to c \to b \to e$; `find(a)` reaches the root in 3 steps.

The MST is found correctly, but the disjoint-set forest has turned into a long chain. A call to `find(a)` follows the path $a \to c \to b \to e$. This is where path compression comes in. As `find(a)` returns, it attaches every vertex on the path directly to the root: `parent[a] = parent[c] = e`. Vertex *b* is already attached to the root. Later lookups finish in one step.

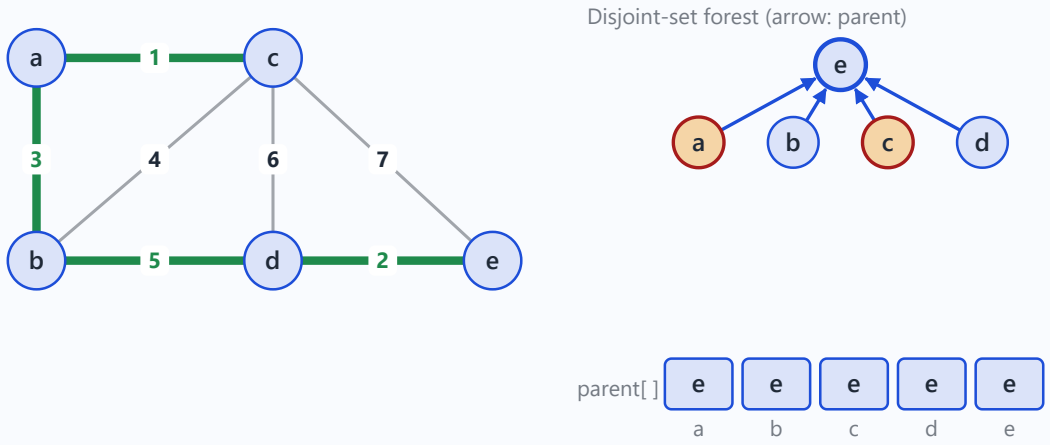


Figure 7.8: The forest after the call `find(a)`: the vertices *a* and *c* on the path are attached directly to the root *e*, and the `parent` array becomes *e e e e e*.

7.3.3 Code trace: find and union

How path compression plays out inside the recursion is traced step by step on a small example with elements *0, …, 4*. The sequence of calls is `union(0, 1)`, `union(1, 2)`, `union(2, 3)`, `find(0)`, and `union(4, 0)`.

Step	Event	`parent`
0	start, every element is a root	[0, 1, 2, 3, 4]
1	`union(0, 1)`: `find(0) = 0`, `find(1) = 1` → `parent[0] = 1`	[1, 1, 2, 3, 4]
2	`union(1, 2)`: roots 1 and 2 → `parent[1] = 2`	[1, 2, 2, 3, 4]
3	`union(2, 3)`: roots 2 and 3 → `parent[2] = 3`	[1, 2, 3, 3, 4]
4	`find(0)` descends: 0 → 1 → 2 → 3; 3 is a root	[1, 2, 3, 3, 4]
5	on return: `parent[2] = 3` (unchanged)	[1, 2, 3, 3, 4]
6	on return: `parent[1] = 3`	[1, 3, 3, 3, 4]
7	on return: `parent[0] = 3`; `find(0) = 3`	[3, 3, 3, 3, 4]
8	`union(4, 0)`: `find(4) = 4`, `find(0) = 3` (one step) → `parent[4] = 3`	[3, 3, 3, 3, 3]

Table 7.5: Trace of `find` and `union`. The `parent` array is given for indices 0, 1, 2, 3, 4.

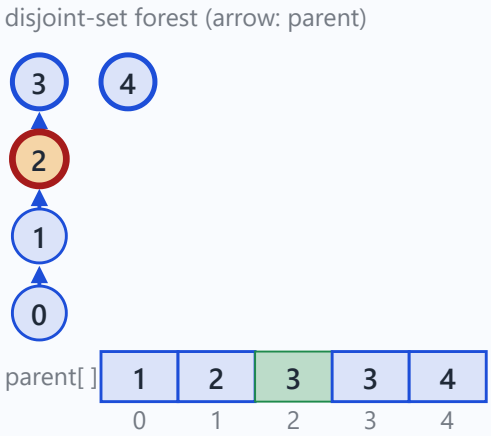


Figure 7.9: The forest after three unions: because one root is attached under the other each time, the chain *0 → 1 → 2 → 3* has formed.

The first three unions produce a bad pattern. Each time one root is attached under the other, and the chain grows by one. `find(0)` now takes three steps. In general, if *n* elements are chained this way, a single `find` takes *n − 1* steps. The call `find(0)` descends to the root recursively: first `find(1)` is called, then `find(2)`, then `find(3)`. `find(3)` finds the root and returns 3. As the recursion unwinds, each frame writes 3 into its own `parent[i]`. As a result 0, 1, and 2 are attached directly to 3. In the last union, `find(0)` finishes in one step and 4 goes under 3; all elements are in one set.

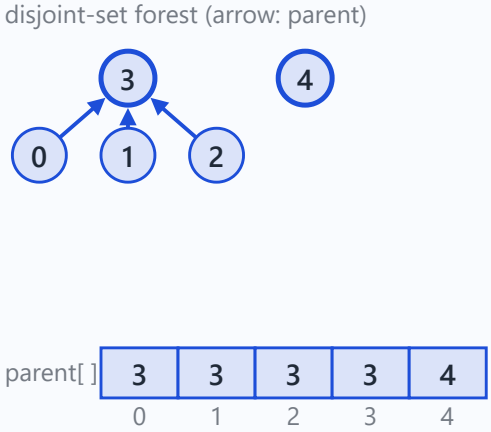


Figure 7.10: The forest after the call `find(0)`: the vertices 0, 1, and 2 on the path are attached directly to the root 3.

Common mistake

Writing `union` as `parent[x] = y`. This assignment attaches *x* itself to *y*; if *x* is not a root, *x* is cut off from its old tree, and the rest of that tree does not join the set of *y*. For example, with `parent = [1, 1, 2]`, writing `parent[0] = 2` merges 0 and 2 but leaves 1 outside. It is always the *roots* that must be linked: `parent[find(x)] = find(y)`.

7.3.4 Path compression and union by rank

The speed of disjoint sets depends on the height of the trees. Two improvements keep the height small. The first is the path compression seen above. The second chooses wisely, at union time, which root goes under which. Each root keeps a **rank**, an upper bound on the height of its tree. In a union, the root of smaller rank is attached under the root of larger rank. If the ranks are equal, one goes under the other and the rank of the new root increases by one. This method is called **union by rank**.

```
void unionByRank(int[] parent, int[] rank, int x, int y) {
  int a = find(parent, x), b = find(parent, y);
  if (a == b) return;                    // already one set
  if (rank[a] < rank[b]) parent[a] = b;   // shorter under taller
  else if (rank[a] > rank[b]) parent[b] = a;
  else { parent[b] = a; rank[a]++; }       // tie: rank grows
}
```

The code first finds the two roots; if they are the same there is nothing to do. If the ranks differ, the shorter tree goes under the taller one and no rank changes: the height of the taller tree does not grow. Only when two trees of equal rank are merged can the height grow by one, so the rank is increased only in that case. If the sequence of calls from the previous trace is repeated with this method, no chain forms. `unionByRank(0, 1)` finds equal ranks, attaches 1 under 0, and raises the rank of 0 to 1. In the later unions, 2, 3, and 4 are single-element trees of rank 0, and each goes under 0, whose rank is 1. The result is `parent = [0, 0, 0, 0, 0]`, and the height of the tree stays at 1.

Theorem 7.6 (Rank and set size)

Under union by rank, a root of rank r has at least 2^r elements in its tree. Hence in a structure of n elements every tree has height at most $\lfloor \log_2 n \rfloor$, and a `find` takes $O(\log n)$ time.

Proof. We use induction. Initially every tree has one element and rank 0: $1 \geq 2^0$. When two trees of different ranks are merged, the rank of the new root does not change but its tree grows, so the inequality is preserved. When two trees of rank r are merged, the rank of the new root becomes $r + 1$. Each of the two trees has at least 2^r elements, so the new tree has at least $2 \cdot 2^r = 2^{r+1}$ elements. Since the total number of elements is n, we get $2^r \leq n$, that is, $r \leq \log_2 n$. Because the height does not exceed the rank, the same bound holds for the height. □

The bound is tight. If 8 elements are merged in pairs, then in fours, then as one group of eight, always joining trees of equal rank, the rank reaches 3; with 1024 elements the same method brings the rank to 10. Adding path compression makes the structure much faster still. In 1975 Robert Tarjan showed that when both improvements are used together, the total cost of m operations is $O(m \cdot \alpha(n))$. Here α is the inverse of the very fast-growing Ackermann function. The inverse Ackermann function grows so slowly that $\alpha(n) \leq 4$ for every practical n not exceeding the number of atoms in the universe. For

this reason the cost per operation is regarded as constant in the **amortized** sense. Amortized cost is the average cost per operation over a long sequence of operations. A single `find` can still take long, but that long walk speeds up the later calls.

> **Common mistake**
>
> Believing that path compression speeds up the single `find` call that performs it. The call that does the compression walks the whole long path itself. The gain comes in *later* calls: the vertices on the path are now one step from the root. The worst-case cost of a single operation and the amortized cost must therefore be assessed separately.

7.4 Kruskal's code and complexity

7.4.1 Java code

The Java implementation of Kruskal rests on an edge class. An edge holds its two endpoints and its weight. So that it can be sorted with `Collections.sort`, it implements the `Comparable` interface and compares edges by weight:

```
class Edge implements Comparable<Edge> {
  int u, v, w;                  // endpoints and weight
  Edge(int u, int v, int w) { this.u = u; this.v = v; this.w = w; }
  public int compareTo(Edge o) {
    return Integer.compare(w, o.w);   // ascending by weight
  }
}
```

`Integer.compare(w, o.w)` is the safe counterpart of the subtraction `w - o.w`. With very large or negative weights the subtraction can overflow; `Integer.compare` cannot. Vertices are numbered from 1 to *V*. The Kruskal code from the slides is:

```
void kruskal(int V, List<Edge> edges) {
  Collections.sort(edges);      // ascending
  int[] parent = new int[V + 1];
  for (int i = 1; i <= V; i++) parent[i] = i;
  List<Edge> tree = new ArrayList<>();
  int k = 0;
  while (tree.size() < V - 1) {
    Edge e = edges.get(k++);
    int x = find(parent, e.u);
    int y = find(parent, e.v);
    if (x != y) {               // different trees
      tree.add(e);
      parent[x] = y;            // union
    }
  }
}
```

The code works as follows, line by line. `Collections.sort(edges)` puts the edge list in ascending order of weight. Java's list sort is stable: edges of equal weight keep their original order in the list. The `parent` array has length *V + 1* because vertices start at 1 and cell 0 is unused. The loop makes every vertex the root of its own set. `tree` holds the chosen edges, and `k` holds the index of the next edge in the sorted list.

The loop `while (tree.size() < V - 1)` runs until *V − 1* edges have been chosen. In each round the next edge is taken and `k` increases by one. The roots of the two endpoints are found as `x` and `y`. If the roots differ, the edge is added to the tree and the two sets merge through `parent[x] = y`. `union` is not called here, because the roots have already been found; the assignment is made directly. If the roots are the same, nothing happens and the loop moves on to the next edge.

7.4.2 Code trace: a five-vertex graph

The code is traced on a graph with vertices 1–5. The edges are *(1,3) = 2*, *(2,5) = 2*, *(1,2) = 3*, *(2,3) = 4*, *(3,4) = 4*, and *(1,4) = 6*.

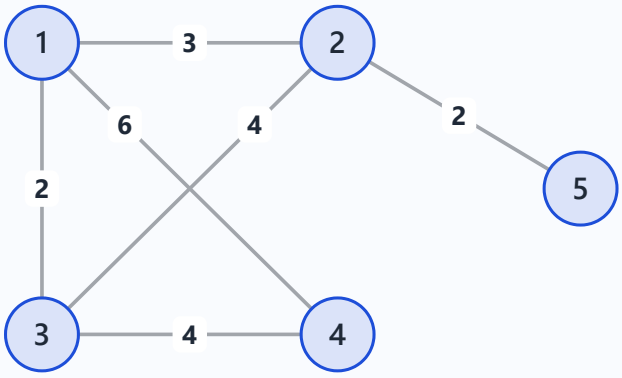


Figure 7.11: A five-vertex graph: *(1,3) = 2*, *(2,5) = 2*, *(1,2) = 3*, *(2,3) = 4*, *(3,4) = 4*, *(1,4) = 6*. The same graph is also used in the Prim traces.

Edge `e`	`x = find(u)`	`y = find(v)`	**Action**	`parent`	***w(T)***
—	—	—	init	[1, 2, 3, 4, 5]	0
(1,3) = 2	1	3	add, `parent[1] = 3`	[3, 2, 3, 4, 5]	2
(2,5) = 2	2	5	add, `parent[2] = 5`	[3, 5, 3, 4, 5]	4
(1,2) = 3	3	5	add, `parent[3] = 5`	[3, 5, 5, 4, 5]	7
(2,3) = 4	5	5	cycle, skip	[3, 5, 5, 4, 5]	7
(3,4) = 4	5	4	add, `parent[5] = 4`	[3, 5, 5, 4, 4]	11

Table 7.6: Trace of `kruskal(5, edges)`. `parent` is given for indices 1–5.

In the third row, `find(1)` climbs to the parent of 1, which is 3, and returns 3; `find(2)` returns 5. The roots differ, and `parent[3] = 5` merges the two trees. Note that the cell that changes is not that of 1 or 2 but that of the root 3. In the fourth row, `find(2)` and `find(3)` both find 5; since *(2,3)* does not join two separate trees, it forms a cycle and is skipped. In the fifth row, 4 is alone; *(3,4)* is added and `parent[5] = 4`. Now `tree.size()` is $4 = V - 1$, and the loop ends. The edge *(1,4) = 6* is never examined. The MST consists of the edges *(1,3)*, *(2,5)*, *(1,2)*, and *(3,4)*, and $w(T) = 11$.

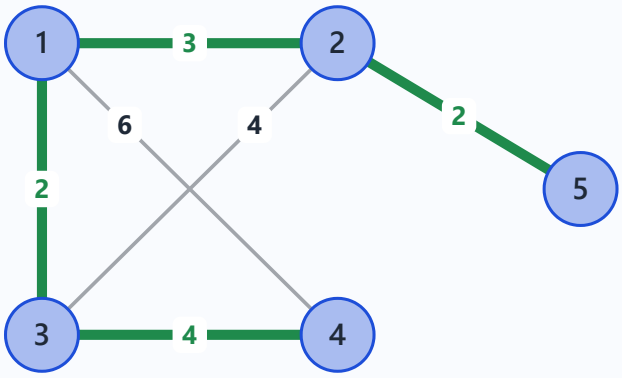


Figure 7.12: The result of Kruskal: *(1,3)*, *(2,5)*, *(1,2)*, *(3,4)*; total weight 11.

The trace contains two pairs of equal-weight edges: *(1,3)* and *(2,5)*, and *(2,3)* and *(3,4)*. A stable sort leaves them in their list order. Had *(3,4)* come before *(2,3)* in the list, *(3,4)* would have been added first, and since the tree is complete with four edges, *(2,3)* would never have been examined. In this graph the result would not change. The reason *(2,3)* is skipped is that the path $2 \to 1 \to 3$ has already been built from cheaper edges.

7.4.3 A nine-vertex example

As a larger example, consider a graph with vertices *A, ..., I* and 15 edges. The sorted edges are:

> EF 1, CF 1, DE 2, FH 3, AB 4, EH 5, GI 5, DG 6, AC 7, BD 9, EG 10, BC 11, HI 12, EI 15, BF 20

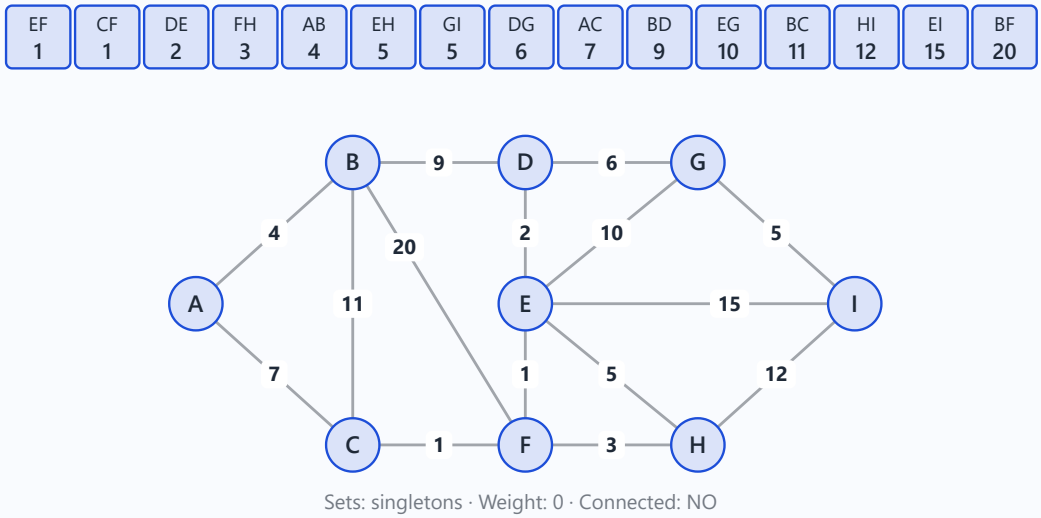


Figure 7.13: The nine-vertex graph and the table of sorted edges. No edge has been chosen yet.

Edge	Decision	Sets (singletons omitted)	Total
EF 1	add	{E, F}	1
CF 1	add	{C, E, F}	2
DE 2	add	{C, D, E, F}	4
FH 3	add	{C, D, E, F, H}	7
AB 4	add	{C, D, E, F, H} {A, B}	11
EH 5	cycle (E–F–H)	unchanged	11
GI 5	add	{C, D, E, F, H} {A, B} {G, I}	16
DG 6	add	{C, D, E, F, G, H, I} {A, B}	22
AC 7	add	{A, B, C, D, E, F, G, H, I}	29

Table 7.7: Kruskal on the nine-vertex graph.

The ninth edge examined is *AC = 7*, and it is the eighth edge added. Since *8 = V − 1*, the algorithm stops here; the last six edges, beginning with BD 9, are never examined. The weight of the MST is 29. This tree is also the source of the clustering example mentioned under applications. The two heaviest MST edges are *AC = 7* and *DG = 6*. Deleting these two edges leaves three clusters: *{A, B}*, *{C, D, E, F, H}*, and *{G, I}*.

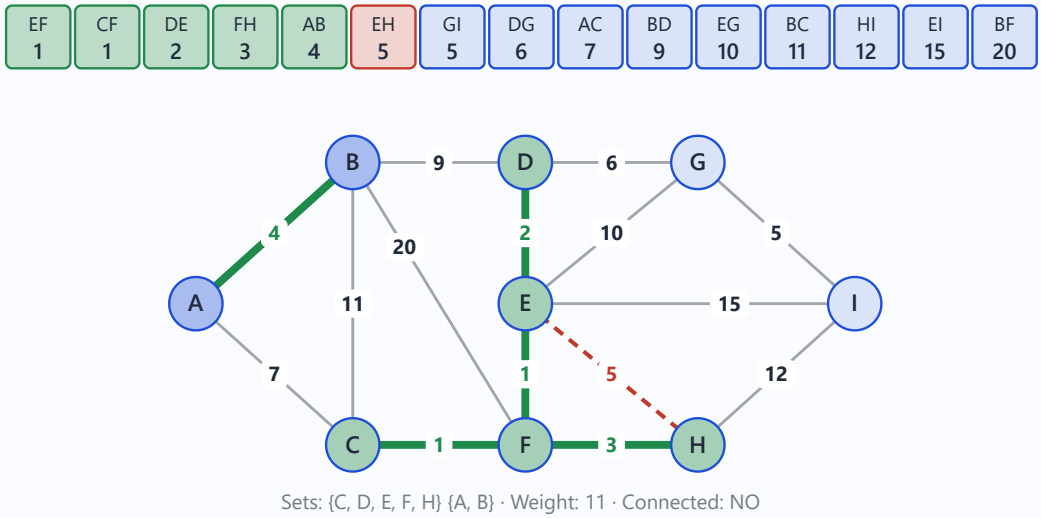


Figure 7.14: The step *EH = 5*. E and H are already in the same set through the path *E–F–H*; the edge would close a cycle and is not added.

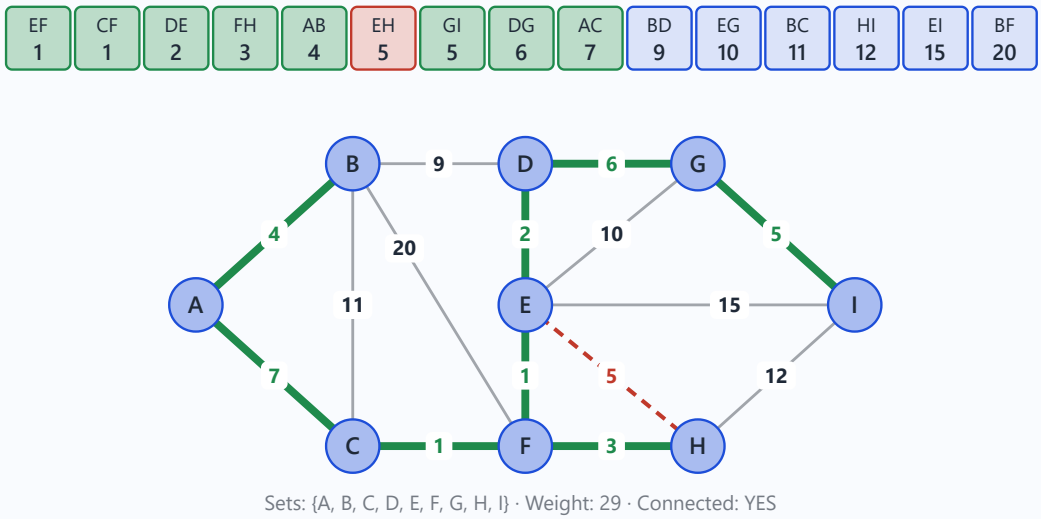


Figure 7.15: After adding *AC = 7*: eight edges have been chosen, the graph is connected, and the MST has weight 29.

7.4.4 Complexity

The cost of Kruskal is computed by splitting it into its operations. Table 7.8 gives how many times each operation is performed and what it costs.

Action	Times	Cost
Sorting the edges by weight	1	$O(E \log E)$
`create(V)`	1	$O(V)$
`find` (2 per edge)	$\leq 2E$	$O(E \cdot \alpha(V))$ with path compression + rank; $O(E \log V)$ with rank only
`union`	$V - 1$	

Table 7.8: Complexity analysis of Kruskal.

The total is $O(E \log E)$, and the sorting determines this cost. In a connected graph $E \geq V - 1$, so the $O(V)$ initialization is also small next to the sort. Most sources write the same bound as $O(E \log V)$. The two forms denote the same order. In a simple undirected graph $E \leq V(V-1)/2 < V^2$. Taking logarithms gives $\log E < 2 \log V$. Since constant factors do not change the order, $O(E \log E) = O(E \log V)$. For example, in a complete graph with $V = 1000$ vertices, $E = 499{,}500$; $\log_2 E \approx 18.9$ and $2 \log_2 V \approx 19.9$.

The situation changes if the edges already arrive sorted or if the weights are small integers. In the second case, counting sort sorts the edges in linear time. In these cases the total time is nearly linear: $O(E \cdot \alpha(V))$. If disjoint sets use only union by rank, each operation takes $O(\log V)$ in the worst case; with path compression added, the amortized cost per operation drops to $O(\alpha(V))$. In both cases the sort dominates the total. The code from the slides does only path compression and uses no rank. This code is also known to be very fast in practice.

The memory requirement is that of the edge list and the parent array: $O(V + E)$. The edge list is the input itself; the extra memory is the `parent` array and the $V - 1$ chosen edges.

7.4.5 A disconnected graph

The code from the slides assumes the graph is connected. In a disconnected graph, $V - 1$ edges can never be chosen. When the list runs out, the loop condition is still true and `edges.get(k++)` runs past the end of the list.

Example 7.4

Let $V = 6$ and the edges be $(1,2) = 2$, $(2,3) = 3$, $(4,5) = 1$, $(5,6) = 4$, $(4,6) = 5$. The graph has two components: $\{1, 2, 3\}$ and $\{4, 5, 6\}$. The sorted list is $(4,5), (1,2), (2,3), (5,6), (4,6)$. The first four edges are added; $(4,6)$ forms a cycle and is skipped. Since `tree.size() = 4 < 5`, the loop continues, and the call `edges.get(5)` throws an `IndexOutOfBoundsException`.

The fix is to add the list bound to the loop condition. If `tree.size() < V - 1` at the end, the graph is not connected. In that case the chosen edges form a **minimum spanning forest**: one minimum spanning tree for each component.

```
List<Edge> kruskalForest(int V, List<Edge> edges) {
  Collections.sort(edges);
  int[] parent = new int[V + 1];
  for (int i = 1; i <= V; i++) parent[i] = i;
  List<Edge> tree = new ArrayList<>();
  int k = 0;
  while (k < edges.size() && tree.size() < V - 1) {
    Edge e = edges.get(k++);
    int x = find(parent, e.u), y = find(parent, e.v);
    if (x != y) { tree.add(e); parent[x] = y; }
  }
  return tree;                  // a forest if tree.size() < V - 1
}
```

On the graph of the example, this version returns the edges *(4,5), (1,2), (2,3), (5,6)*. The forest has two trees and total weight *1 + 2 + 3 + 4 = 10*. The number of edges is $V - 2$. In general, a spanning forest of a graph with c components has $V - c$ edges.

7.5 Prim's algorithm

Kruskal works with small trees spread across the whole graph. Prim starts with a single tree and grows it step by step. The algorithm became known through Robert C. Prim's 1957 paper. Vojtěch Jarník had found the same method in 1930, and Edsger W. Dijkstra rediscovered it in 1959. For this reason some sources call it Jarník's algorithm or the Prim–Jarník algorithm.

7.5.1 The idea: growing a single tree

A starting vertex is chosen; at first the tree consists of this vertex alone. At each step, among the vertices not in the tree, the one that connects to the tree by the *cheapest edge* is selected. This vertex and the edge connecting it are added to the tree. The algorithm ends when all vertices are in the tree. There is a single tree at every moment; there is no forest as in Kruskal.

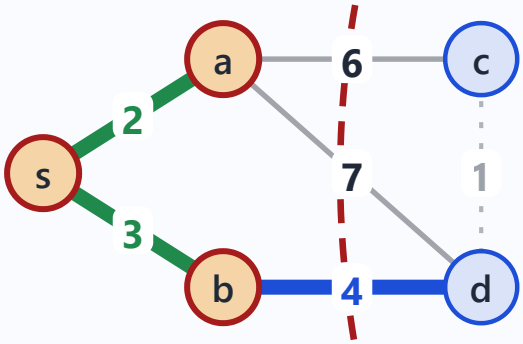


Figure 7.16: An intermediate state of Prim. The tree is *{s, a, b}*. The dashed line shows the cut between the tree and the remaining vertices. The edges crossing the cut have weights 6, 7, and 4; the cheapest, 4, is chosen and *d* enters the tree.

At every step Prim chooses the lightest edge of a cut: the cut lies between the tree built so far and the remaining vertices. The edge *c–d = 1* in the figure is the lightest edge of the graph, but it does not cross the cut; neither of its endpoints is in the tree. Prim can use this edge only after *d* has entered the tree. Kruskal, on the same graph, would have started with *c–d = 1*.

An assessment of the algorithm follows. Prim is well suited to dense graphs: with an adjacency matrix it runs in $O(V^2)$ time. There is no need to sort the edges. It works naturally with an adjacency list or matrix. Its weakness is that its performance depends on the data structure used: an array, a binary heap, and a Fibonacci heap give different running times. On a disconnected graph it covers only the component of the starting vertex.

7.5.2 Key and parent

If “the vertex that connects to the tree by the cheapest edge” were searched for anew at every step, the algorithm would be slow. Prim stores this information for each vertex in two arrays and updates it as the tree grows.

Definition 7.5

In Prim's algorithm, for a vertex *v* not in the tree:

- `key[v]` : the weight of the cheapest edge connecting *v* to the tree built so far. It starts at ∞ and is 0 at the source.
- `parent[v]` (π): the endpoint, in the tree, of this cheapest edge. When the algorithm finishes, the edges *(parent[v], v)* give the MST.

When a new vertex *u* enters the tree, only the keys of the neighbors of *u* can change, because new ways of connecting to the tree open up only through *u*. For each neighbor *v* not in the tree, if *w(u, v) < key[v]*, the key and parent are updated: `key[v] = w(u, v)`, `parent[v] = u`. Then the vertex outside the tree with the smallest key is selected. The key of this vertex is the weight of the lightest edge crossing the cut.

7.5.3 Pseudocode

The steps of Prim can be listed as follows:

1. Choose a starting vertex and add all edges belonging to it to the priority queue.
2. From the queue, select the edge of smallest weight that leads to a vertex not in the tree.
3. Add the new vertex connected by the selected edge to the tree.
4. Add the edges of the new vertex to the queue; repeat steps 2–4 until all vertices are in the tree.

The pseudocode on the slides has two parts. The first part initializes the arrays, and the second is the main loop:

```
PRIM(G, source):
  MST = [ ]
  dist = [ ]
  parent = [ ]
  visited = [ ]
  for each v in V:
    dist[v] = ∞
    visited[v] = 0
  dist[source] = 0
  priorityQueue.add(source, 0)
```

```
while priorityQueue is not empty:
  u = priorityQueue.extractMin()
  if visited[u] == 0:
    visited[u] = 1
    for each (u, v, weight) in G[u]:
      if visited[v] == 0 and
           weight < dist[v]:
        dist[v] = weight
        parent[v] = u
        priorityQueue.add(v, weight)
for each v in V
    (except source and dist[v] ≠ ∞):
  MST.add((parent[v], v, dist[v]))
return MST
```

The last loop adds an edge only for vertices other than the source with *dist[v]* ≠ ∞: a vertex whose key is still ∞ was not reachable from the source, and `parent[v]` is undefined. The `dist` array in the pseudocode is the `key` array of the definition. The name can be misleading: the value is not a distance but the weight of a single edge. The code may add a vertex to the queue more than once. Every time a vertex's key decreases, a new entry goes into the queue; the old entry is not deleted and stays in the queue. This is why this version is called **lazy** Prim. When the old entry finally leaves the queue, the check `visited[u]` discards it. The queue holds at most E entries, since each edge can cause one key decrease. The total time is $O(E \log E)$.

Common mistake

Skipping the `visited` check in the lazy version. When an old entry leaves the queue, its vertex is already in the tree. Without the check the vertex would "enter the tree" a second time, its neighbors would be examined again, and the parent array could be corrupted. The first thing to do after every extraction from the queue is to check whether the vertex has been processed before.

7.5.4 Step by step: the priority queue

Prim is started at vertex 1 on the five-vertex graph of Figure 7.11. The classical textbook notation is used here: the queue Q starts with all vertices, and each vertex stays in the queue together with its key. In the tables π is the parent; "/" means that no parent has been assigned yet.

Step	Vertex extracted	Edge added	Queue Q (after)	$w(T)$
0	—	—	1 (0, /), 2 (∞, /), 3 (∞, /), 4 (∞, /), 5 (∞, /)	0
1	1	—	3 (2, 1), 2 (3, 1), 4 (6, 1), 5 (∞, /)	0
2	3	*(1,3) = 2*	2 (3, 1), 4 (4, 3), 5 (∞, /)	2
3	2	*(1,2) = 3*	5 (2, 2), 4 (4, 3)	5
4	5	*(2,5) = 2*	4 (4, 3)	7
5	4	*(3,4) = 4*	empty	11

Table 7.9: Prim with a priority queue, source 1. The queue contents are sorted by key; each item has the form *vertex (key, π)*.

The steps are explained as follows. At initialization only the source has key 0, so the first vertex extracted is 1. The neighbors of 1 are updated: *key(3) = 2*, *key(2) = 3*, *key(4) = 6*. In the second step the smallest key belongs to 3; vertex 3 is extracted and *(1,3)* enters the tree. For the neighbor 4 of 3, since *(3,4) = 4 < 6*, we get *key(4) = 4* and *π(4) = 3*. For the other neighbor 2, since *(2,3) = 4 > key(2) = 3*, there is no change. In the third step 2 is extracted and *(1,2)* is added; the key of 5 becomes 2. In the fourth step 5 is extracted; it has no other neighbor outside the tree. In the last step 4 is extracted and *(3,4)* is added.

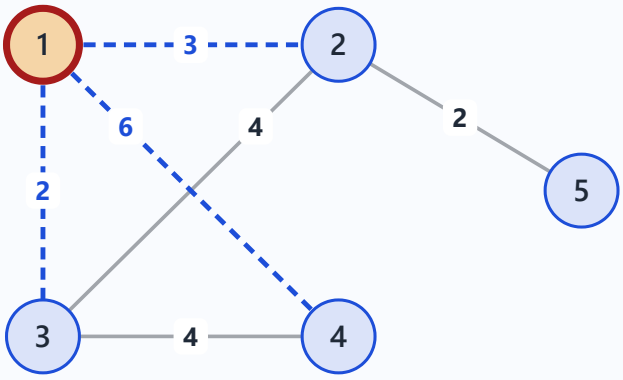


Figure 7.17: The state after 1 leaves the queue. The dashed blue lines are the cheapest connections, shown with π, that are not yet final: 2 to 3, 3 to 2, 6 to 4.

The resulting tree consists of the edges *(1,3), (1,2), (2,5), (3,4)* and *w(T) = 11*. Kruskal had found the same tree on the same graph. The order in which the edges are added differs, however: Kruskal adds 2, 2, 3, 4, whereas Prim adds 2, 3, 2, 4. Prim adds the edge *(2,5)* of weight 2 after the edge *(1,2)* of weight 3 because 5 becomes adjacent to the tree only after 2 has entered it.

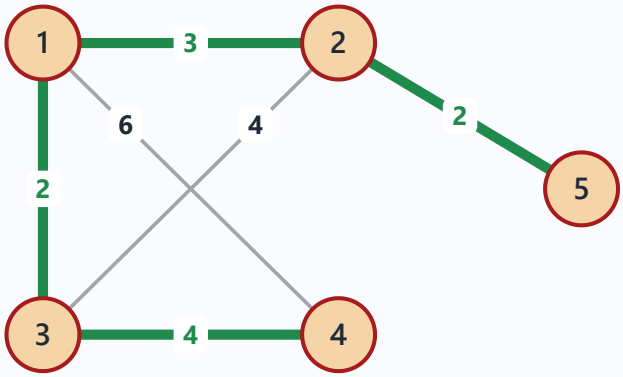


Figure 7.18: The final state of Prim: the queue is empty. The MST consists of the edges *(1,3), (1,2), (2,5), (3,4)*; *w(T) = 11*.

7.5.5 Why is Prim correct?

Theorem 7.7 (Correctness of Prim)

On a connected graph, the edges *(parent[v], v)* chosen by Prim's algorithm form a minimum spanning tree.

Proof. At the beginning of a step, let the vertices in the tree be S and consider the cut $(S, V - S)$. Both endpoints of every chosen edge are in S; none crosses the cut. `key[v]` is the weight of the cheapest edge from v to S. This invariant is maintained at every step, because when a new vertex enters the tree only the values of its neighbors can change, and the update does exactly that. When u with the smallest key is selected, $(parent[u], u)$ is the lightest edge crossing the cut. By the cut property this edge is safe. At each step one vertex and one safe edge are added to the tree; after $V - 1$ steps an MST is obtained. □

A consequence of the proof is that the starting vertex does not matter. Prim started from a different vertex adds the edges in a different order. If the weights are distinct, it still finds the same tree in the end; if there are equal weights, it may find a different tree of the same weight.

7.6 Prim's code and complexity

7.6.1 The adjacency-matrix version

The shortest implementation of Prim uses no priority queue. The graph is given as an adjacency matrix: `G[u][v]` is the weight of the edge between u and v, or 0 if there is no edge. The vertex with the smallest key is found at each step by scanning the array from start to end.

```
void prim(int[][] G, int source) {
  int V = G.length;
  int[] key = new int[V];
  int[] parent = new int[V];
  boolean[] inTree = new boolean[V];
  Arrays.fill(key, Integer.MAX_VALUE);
  key[source] = 0;
  parent[source] = -1;
  for (int s = 0; s < V - 1; s++) {
    int u = minKey(key, inTree);
    inTree[u] = true;
    for (int v = 0; v < V; v++)
      if (G[u][v] != 0 && !inTree[v]
          && G[u][v] < key[v]) {
        key[v] = G[u][v];
        parent[v] = u;
      }
  }
}
```

The code works as follows, line by line. Every cell of the `key` array is set to `Integer.MAX_VALUE`, which stands in for ∞. The source's key is 0 and its parent is *−1*; *−1* means “no parent.” The `inTree` array records which vertices have entered the tree.

The outer loop runs $V - 1$ times. In each round `minKey` finds the vertex outside the tree with the smallest key, and this vertex is taken into the tree. The inner loop visits all neighbors of this vertex. Three conditions are tested together: `G[u][v] != 0` says the edge exists, `!inTree[v]` says the neighbor is not yet in the tree, and `G[u][v] < key[v]` says the new edge is cheaper than the old best. If all three hold, the key and parent are updated.

That the outer loop runs $V - 1$ times rather than V is a deliberate choice. Once $V - 1$ vertices are in the tree, the key and parent of the last vertex can no longer change: the other endpoints of all edges connecting it to the tree are in the tree, and all of them have been examined. The last vertex is not marked in the `inTree` array, but its `parent` value is correct. The code on the slides uses a `minKey` helper. The helper can be written as follows:

```
int minKey(int[] key, boolean[] inTree) {
  int min = Integer.MAX_VALUE, index = -1;
  for (int v = 0; v < key.length; v++)
    if (!inTree[v] && key[v] < min) {
      min = key[v];
      index = v;
    }
  return index;
}
```

In a tie the helper picks the vertex with the smaller index, because the comparison uses `<`: a later equal key does not replace the minimum already found. In a disconnected graph all keys outside the tree may remain ∞; then the method returns -1 and the access `inTree[-1]` inside `prim` raises an error. The code assumes a connected graph.

7.6.2 Code trace: the A–F graph

The code is traced on a graph with six vertices. The vertices are A–F with indices 0–5. The source is B (index 1). The edges are $AB = 1$, $BC = 3$, $AD = 3$, $BD = 2$, $BE = 6$, $BF = 5$, $CE = 4$, $CF = 4$, $DE = 3$, and $EF = 2$.

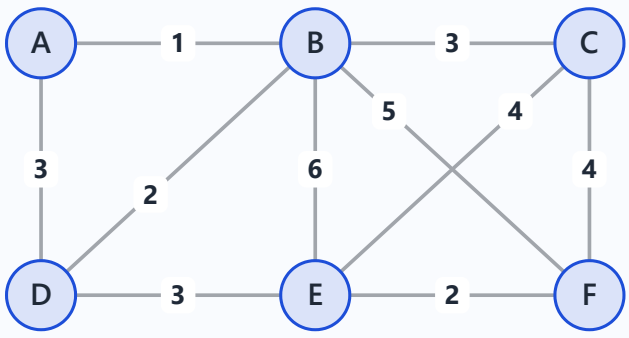


Figure 7.19: The A–F graph. The source is B; initially all keys are ∞ and `key[B] = 0`.

Round `s`	`u`	Neighbors updated	`key`	*w(T)*
—	—	init	∞, 0, ∞, ∞, ∞, ∞	0
0	B (0)	A → 1, C → 3, D → 2, E → 6, F → 5 (parent B in all cases)	1, 0, 3, 2, 6, 5	0
1	A (1)	none: *AD = 3 > key[D] = 2*	1, 0, 3, 2, 6, 5	1
2	D (2)	E → 3 (parent D)	1, 0, 3, 2, 3, 5	3
3	C (3)	F → 4 (parent C); *CE = 4 > 3*, E unchanged	1, 0, 3, 2, 3, 4	6
4	E (3)	F → 2 (parent E)	1, 0, 3, 2, 3, 2	9
—	(F)	loop ends; the parent of F is E	1, 0, 3, 2, 3, 2	11

Table 7.10: Trace of `prim(G, 1)`. The key array is given in the order A, B, C, D, E, F; ∞ is the initial value.

In the first round `minKey` returns B, because only B's key is finite. All five neighbors of B are updated. In the second round the smallest key is A's (1); *(B, A)* enters the tree. The only neighbor of A outside the tree is D, but *AD = 3* is larger than D's current key, 2. In the third round D (2) enters and E's key drops from 6 to 3: connecting E through D is cheaper than connecting it through B.

In the fourth round the keys of C and E are equal (3). `minKey` chooses C, which has the smaller index. The edge of weight 4 from C to F is cheaper than F's current key, 5, so F's parent becomes C. In the fifth round E (3) enters, and through *EF = 2* the key of F drops to 2. The outer loop makes five rounds and ends. F is not formally taken into the tree, but its parent is E. The MST is read from the edges *(parent[v], v)*: *BA, BC, BD, DE, EF*. The total is *1 + 3 + 2 + 3 + 2 = 11*.

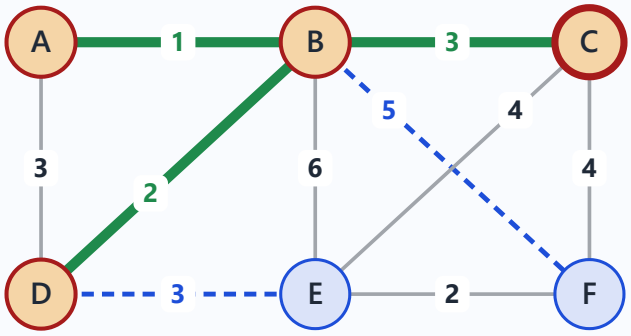


Figure 7.20: The fourth round: the keys of C and E are equal (3); C, with the smaller index, enters the tree. The blue dashed edges are the current cheapest connections of the vertices not yet in the tree.

What would happen if E were chosen in the tie? E enters, and F's key becomes 2; then C (3) and F (2) compete and F is chosen. Finally C enters too. The chosen edges would again be *BA, BD, DE, EF, BC*; only the order would change. Although this graph has equal-weight edges, its MST is unique.

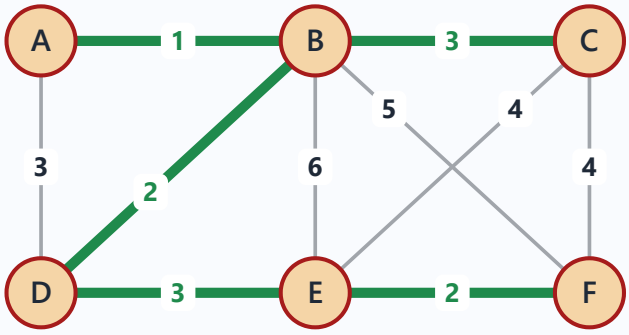


Figure 7.21: The result of Prim: AB, BD, BC, DE, EF; total 11.

7.6.3 The lazy priority-queue version

On sparse graphs, scanning the array from start to end in every round is an unnecessary cost. Java's `PriorityQueue` class removes the smallest element in *O(log n)* time. The lazy version on the slides writes the method of the pseudocode in Java. The graph is given as an adjacency list: `adj.get(u)` is the list of edges leaving *u*. `key`, `parent`, `inTree`, and `adj` are class fields.

```
void primPQ(int source) {
  Arrays.fill(key, INF);
  key[source] = 0;
  PriorityQueue<int[]> pq =
      new PriorityQueue<>((a, b) -> a[1] - b[1]);
  pq.add(new int[] { source, 0 });
  while (!pq.isEmpty()) {
    int u = pq.poll()[0];
    if (inTree[u]) continue;  // stale
    inTree[u] = true;
    for (Edge e : adj.get(u))
      if (!inTree[e.v] && e.w < key[e.v]) {
        key[e.v] = e.w; parent[e.v] = u;
        pq.add(new int[] { e.v, e.w });
      }
  }
}
```

Each entry in the queue is a two-element array: *{vertex, key}*. The comparator `a[1] - b[1]` orders the entries by key. As long as the weights are small, the subtraction is safe; with very large weights it is more correct to write `Integer.compare(a[1], b[1])`. In each round the loop removes the smallest entry. If the vertex is already in the tree, the entry is stale and is skipped with `continue`. Otherwise the vertex enters the tree and its neighbors are examined. For every neighbor whose key decreases, a new entry is added to the queue. The old entry is not removed; the removal is postponed until the moment the entry leaves the queue. This postponement is called **lazy deletion**.

The code is traced on the five-vertex graph with source 1. The queue column in Table 7.11 gives the contents before the entry is removed, sorted by key.

Queue (before)	Extracted	Action	`key`
(1,0)	(1,0)	1 enters the tree; 2 → 3, 3 → 2, 4 → 6	0, 3, 2, 6, ∞
(3,2) (2,3) (4,6)	(3,2)	3 enters (edge 1–3); 4 → 4; for 2, 4 > 3, unchanged	0, 3, 2, 4, ∞
(2,3) (4,4) (4,6)	(2,3)	2 enters (edge 1–2); 5 → 2	0, 3, 2, 4, 2
(5,2) (4,4) (4,6)	(5,2)	5 enters (edge 2–5); no update	0, 3, 2, 4, 2
(4,4) (4,6)	(4,4)	4 enters (edge 3–4); no update	0, 3, 2, 4, 2
(4,6)	(4,6)	4 is already in the tree: stale entry, `continue`	0, 3, 2, 4, 2

Table 7.11: Trace of `primPQ(1)`. Entries have the form *(vertex, key)*; `key` is given for indices 1–5.

In the second row the key of 4 drops from 6 to 4, and the entry *(4,4)* is added to the queue. The old entry *(4,6)* stays in the queue. Vertex 4 enters the tree through the entry *(4,4)*. The entry *(4,6)* is extracted last and skipped. When the queue is empty, the MST edges are read as *(parent[v], v)*: 1–3 (2), 1–2 (3), 2–5 (2), and 3–4 (4); total 11.

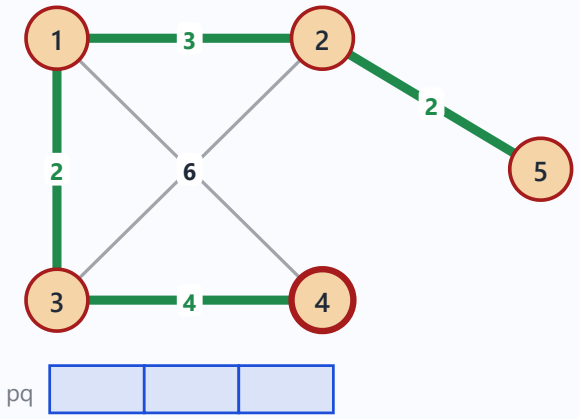


Figure 7.22: The last step: the stale entry *(4,6)* has been extracted and skipped; the queue is empty. The tree edges are 1–3, 1–2, 2–5, 3–4.

The counterpart of the lazy version is the **eager** version, which decreases the key in place. The eager version keeps a single entry per vertex in the queue and updates that entry when the key decreases. This operation is called **decrease-key**. Java's `PriorityQueue` class does not offer this operation directly, which is why the lazy version is common in Java implementations. The lazy version keeps at most E entries in the queue instead of V, but since $\log E \le 2 \log V$, the order does not change.

7.6.4 A nine-vertex example

The nine-vertex example of Kruskal is solved with Prim starting from A. Each row gives the key and parent values after the vertex that enters the tree in that step.

Entering	A	B	C	D	E	F	G	H	I	Total
A	0/–	4/A	7/A	∞	∞	∞	∞	∞	∞	0
B	0/–	4/A	7/A	9/B	∞	20/B	∞	∞	∞	4
C	0/–	4/A	7/A	9/B	∞	1/C	∞	∞	∞	11
F	0/–	4/A	7/A	9/B	1/F	1/C	∞	3/F	∞	12
E	0/–	4/A	7/A	2/E	1/F	1/C	10/E	3/F	15/E	13
D	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	15/E	15
H	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	12/H	18
G	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	5/G	24
I	0/–	4/A	7/A	2/E	1/F	1/C	6/D	3/F	5/G	29

Table 7.12: Prim on the nine-vertex graph (from A). Each cell holds *key / parent*; "–" means that no parent has been assigned.

The table shows several things clearly. The key of F first becomes 20 through B and drops to 1 when C enters. The key of D is 9 through B and drops to 2 when E enters. The key of I changes three times: 15 through E, 12 through H, and 5 through G. A vertex's key only decreases; it cannot grow. When H enters the tree, the edge *EH = 5* can no longer be used, because E and H are both in the tree. Kruskal had skipped the same edge on the grounds of forming a cycle.

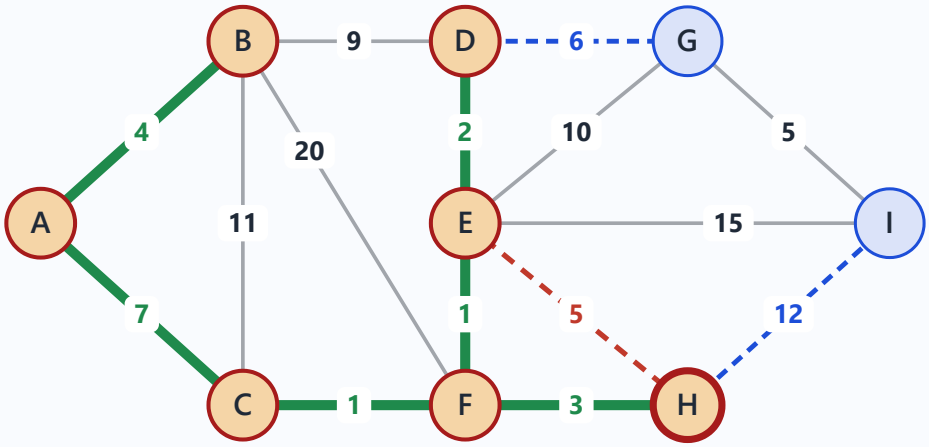


Figure 7.23: After H enters the tree: total 18. The red dashed edge *EH = 5* would now form a cycle; the blue dashed edges are the current cheapest connections for G and I.

The resulting weight is 29, the same as that of the tree Kruskal found. The orders in which the two algorithms add edges, however, are clearly different. Kruskal adds the weights in the order *1, 1, 2, 3, 4, 5, 6, 7*. Prim adds them in the order *4, 7, 1, 1, 2, 3, 6, 5*: it first takes the expensive edges around A, and reaches the cheap edges only when the tree gets to them.

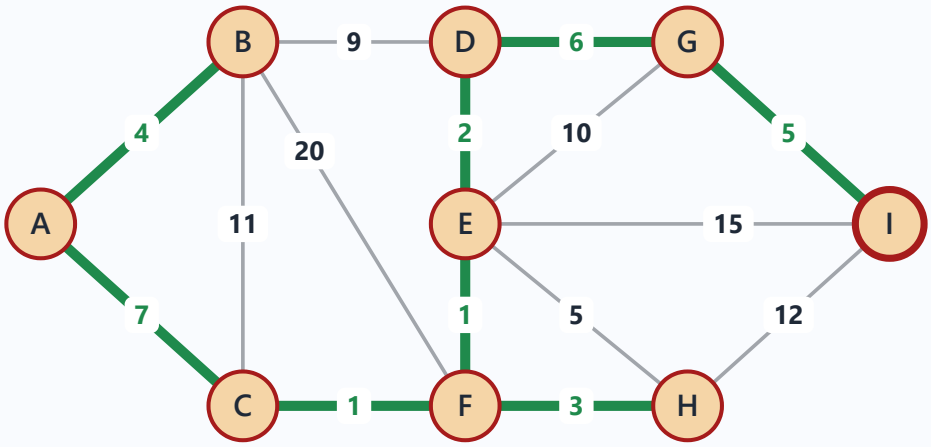


Figure 7.24: I enters: all vertices are in the tree and the MST has weight 29.

7.6.5 Complexity

The running time of Prim depends on the cost of two operations: extracting the vertex with the smallest key and decreasing the key of a neighbor. The first operation is performed V times. The second is performed at most E times, because each edge is examined once, when one of its endpoints enters the tree. In an undirected graph every edge appears in two adjacency lists; the number of examinations does not exceed $2E$.

Implementation	Extract min	Decrease key	Total
Adjacency matrix + array scan	$V \times O(V)$	$O(1)$	$O(V^2)$
Adjacency list + binary heap	$V \times O(\log V)$	$E \times O(\log V)$	$O((V + E) \log V)$
Lazy version: edge copies in the queue	$\le E$ insertions and extractions, each $O(\log E)$		$O(E \log E)$
Fibonacci heap	$O(\log V)$ amortized	$O(1)$ amortized	$O(E + V \log V)$

Table 7.13: The cost of different implementations of Prim.

In the matrix version the outer loop runs $V - 1$ times. In each round `minKey` scans V cells and the inner loop visits V neighbors. The total is $O(V^2)$, independent of the number of edges. In the heap version each extraction and each decrease-key takes $O(\log V)$; the total is $O((V + E) \log V)$. In a connected graph $E \ge V - 1$, so this expression is often shortened to $O(E \log V)$.

Which version is better depends on the density of the graph. In a **dense** graph ($E \approx V^2$) the matrix version is best with $O(V^2)$; the heap version rises to $O(V^2 \log V)$. In a **sparse** graph ($E \approx V$), on the other hand, the heap version takes $O(V \log V)$, while the matrix version needlessly spends $O(V^2)$.

Example 7.5

Compare the two versions on a graph with $V = 1000$ vertices. The matrix version takes about $V^2 = 10^6$ steps. For the heap version take $\log_2 1000 \approx 10$.

1. $E = 5000$ (sparse): $E \log V \approx 5000 \cdot 10 = 5 \cdot 10^4$. The heap version does about 20 times less work.
2. $E = 500{,}000$ (nearly complete): $E \log V \approx 5 \cdot 10^6$. The matrix version does about 5 times less work.

The crossover is roughly at $E \log V \approx V^2$; here $E \approx 10^5$. Constant factors shift the boundary somewhat, but the direction does not change.

7.6.6 How Prim differs from Dijkstra

The code of Prim is very similar to that of Dijkstra's shortest-path algorithm. Both start at a source, extract the vertex with the smallest key from a priority queue, and update its neighbors. The only difference is the meaning of the key.

	Prim (MST)	**Dijkstra (shortest paths)**
`key[v]`	the single cheapest edge connecting v to the tree	total path length from the source to v
Update	$w(u, v) < key[v]$	$dist[u] + w(u, v) < dist[v]$
Result	tree of smallest total weight	shortest-path tree
Source	does not affect the result (same weight)	determines the result
Negative weights	no problem	does not work correctly

Table 7.14: Comparison of Prim and Dijkstra.

Example 7.6

Run Dijkstra from B on the A–F graph. The distances are $A = 1$, $C = 3$, $D = 2$, $E = 5$ (B → D → E), and $F = 5$. F is reached directly by the edge $BF = 5$; the path B → D → E → F costs $2 + 3 + 2 = 7$. The shortest-path tree consists of the edges BA, BC, BD, DE, BF and has total weight $1 + 3 + 2 + 3 + 5 = 14$.

Prim, on the other hand, attaches F to E: the cheapest way to connect F to the tree is $EF = 2$. Prim's tree has weight 11. Both trees span all the vertices. One minimizes the total weight, the other the distances from the source.

A similar difference shows up in the five-vertex graph. The shortest path from 1 to 4 has length 6, and this distance is obtained both by the direct edge $(1,4)$ and by the path $1 \to 3 \to 4$. So the shortest-path tree may use the edge $(1,4) = 6$. The MST chooses the edge $(3,4) = 4$ instead.

Common mistake

Writing Prim's update condition as `key[u] + G[u][v] < key[v]`. This line turns Prim into Dijkstra, and the result is a shortest-path tree. On the A–F graph this mistake attaches F to B and yields a tree of weight 14 instead of 11. In Prim the key is always the weight of a single edge; a sum is never written.

7.7 Borůvka's algorithm

The oldest algorithm for finding a minimum spanning tree is Otakar Borůvka's work of 1926. Borůvka was working on the problem of building the electrical network of the Moravia region as cheaply as possible. The method was rediscovered several times afterwards; because of Georges Sollin's work in the 1960s it is also called **Sollin's algorithm**.

7.7.1 The idea: all components at once

At the start every vertex is an independent component. The algorithm proceeds in rounds. In each round every component chooses the cheapest edge connecting it to the outside. All the chosen edges are added together, and the components they connect merge. Rounds continue until a single component remains. On a connected graph the algorithm produces $V - 1$ edges.

The cheapest outgoing edge of a component is the lightest edge of the cut between that component and the rest of the vertices. By the cut property this edge is safe. Kruskal and Prim choose a single cut at each step; Borůvka uses as many cuts as there are components, all at once, in every round.

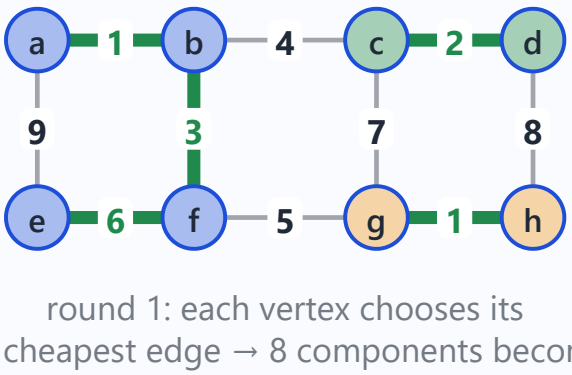


Figure 7.25: The first round of Borůvka. Each vertex chooses its own cheapest edge: *ab = 1*, *cd = 2*, *bf = 3*, *ef = 6*, *gh = 1*. Five edges are added at once; eight components drop to three.

In the graph of the figure, the cheapest edge of *e* is *ef = 6*, because its other edge is *ae = 9*. Vertex *f* chooses *bf = 3*. An edge can be chosen from both ends; *ab* is the choice of both *a* and *b*, but it is added only once. At the end of the first round the components are *{a, b, e, f}*, *{c, d}*, and *{g, h}*. In the second

round *bc = 4* and *fg = 5* are added and the tree is complete; the total weight is 22.

The most important advantage of the algorithm is that the components' choices are independent of one another. The cheapest edge of each component can be found on a separate processor. This makes Borůvka very well suited to parallel and distributed computation. The number of rounds is also small: at most $\lceil \log_2 V \rceil$. It has weaknesses too. It is more complicated to code than Kruskal and Prim. When there are equal-weight edges a consistent tie-breaking rule is needed; otherwise a cycle can form.

7.7.2 Steps and pseudocode

1. At the start every vertex counts as a component (set) by itself.
2. For each component, find the edge of smallest weight that connects it to another component.
3. Add all the edges found; merge the components they connect. The same edge is not added twice.
4. Repeat steps 2 and 3 until a single component remains. If no edge is added in a round, the graph is not connected; stop, and the result is a minimum spanning forest.

```
BORUVKA(G):
  MST = [ ]
  componentCount = |V|
  while componentCount > 1:
    cheapest = [ ]  // per component
    for each e = (u, v, weight) in E:
      cu = disjointSet.find(u)
      cv = disjointSet.find(v)
      if cu ≠ cv:
        cheapest[cu] = min(cheapest[cu], e)
        cheapest[cv] = min(cheapest[cv], e)
    added = false
    for each (u, v, weight) in cheapest:
      if disjointSet.find(u) ≠ disjointSet.find(v):
        MST.add((u, v, weight))
        disjointSet.union(u, v)
        componentCount -= 1
        added = true
    if not added: break   // forest
  return MST
```

Borůvka does not sort the edges. In each round it scans all edges once and, for each component, keeps the smallest outgoing edge seen so far. The components are tracked with a disjoint-set structure: if the `find` results of an edge's two endpoints are the same, the edge lies inside a component and is ignored. In the second loop, `find` is asked again as the chosen edges are added. The reason is that two components may have chosen the same edge. When it is added for the first time, the two components merge; when its turn comes the second time, its endpoints are already in the same component and the edge is skipped.

7.7.3 A six-vertex example

The algorithm is run on a graph with vertices A–F and nine edges. The edges are *AB = 1*, *BC = 5*, *AD = 4*, *BE = 3*, *BD = 7*, *BF = 6*, *CF = 9*, *DE = 8*, and *EF = 2*.

Component	Outgoing edges	Cheapest
{A}	AB 1, AD 4	AB 1
{B}	AB 1, BC 5, BE 3, BD 7, BF 6	AB 1
{C}	BC 5, CF 9	BC 5
{D}	AD 4, BD 7, DE 8	AD 4
{E}	BE 3, DE 8, EF 2	EF 2
{F}	BF 6, CF 9, EF 2	EF 2

Table 7.15: Borůvka on the six-vertex graph, round 1. Each component and its own cheapest outgoing edge.

Two components each choose AB and EF, but each is added once. In round 1 four distinct edges are added: AB, BC, AD, and EF. Two components remain: *{A, B, C, D}* and *{E, F}*. The total weight is *1 + 5 + 4 + 2 = 12*.

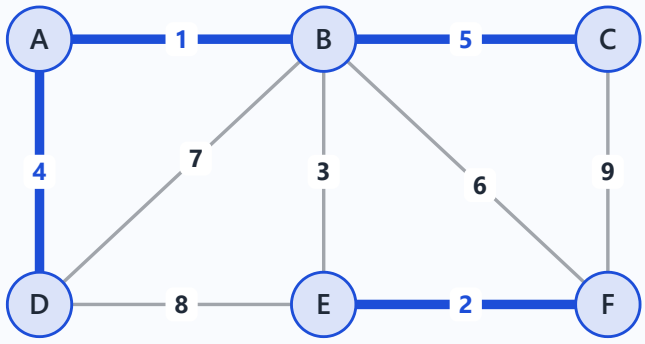


Figure 7.26: Round 1: the cheapest outgoing edge chosen by each component is shown in blue. Four distinct edges are chosen.

In round 2 the edges between the two components are *BE = 3*, *BF = 6*, *CF = 9*, and *DE = 8*. Both components choose the cheapest, *BE = 3*. BE is added and a single component remains. The MST consists of the edges *AB, BC, AD, EF, BE* and has weight *12 + 3 = 15*.

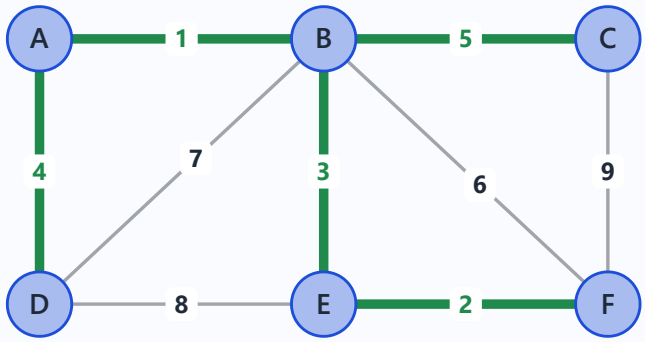


Figure 7.27: End of round 2: BE has been added and a single component remains. The MST has weight 15.

On the same graph Kruskal adds the edges one at a time, in the order of weights *1, 2, 3, 4, 5*: AB, EF, BE, AD, BC. In Borůvka, four edges are added at once in the first round. The tree found is the same.

7.7.4 A twelve-vertex example

The strength of Borůvka shows more clearly on larger graphs. Consider a graph with vertices A–L and 20 edges. The edges are:

> AB 13, AC 6, BC 7, BD 1, CD 14, CE 8, CH 20, DE 9, DF 3, EF 2, EJ 18, GH 15, GJ 19, GK 10, GI 5, HJ 17, IK 11, JK 16, JL 4, KL 12

Round	Cheapest edges of the components	Distinct edges added	Number of components	Total
1	A: AC 6, B: BD 1, C: AC 6, D: BD 1, E: EF 2, F: EF 2, G: GI 5, H: GH 15, I: GI 5, J: JL 4, K: GK 10, L: JL 4	AC, BD, EF, GI, GH, JL, GK	12 → 5	43
2	{A, C}: BC 7, {B, D}: DF 3, {E, F}: DF 3, {G, H, I, K}: KL 12, {J, L}: KL 12	BC, DF, KL	5 → 2	65
3	{A, …, F}: EJ 18, {G, …, L}: EJ 18	EJ	2 → 1	83

Table 7.16: Borůvka rounds on the twelve-vertex graph.

In the first round seven distinct edges are added and the number of components drops from 12 to 5. H's choice is noteworthy: all of H's edges are expensive (15, 17, 20), and the cheapest is GH 15. This edge is safe despite being heavy, because it is the lightest edge of the cut *{H}*. K also chooses GK 10; IK 11 and KL 12 are more expensive. G, H, I, and K are thus gathered into one component.

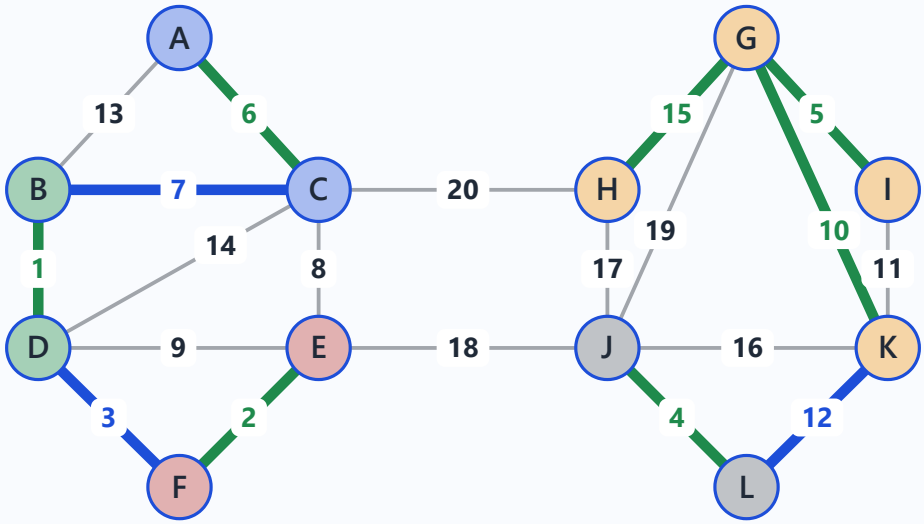


Figure 7.28: Round 2: the five components (in colors) and the cheapest outgoing edge of each (blue). The thick green edges were added in round 1.

In the second round the component *{A, C}* chooses BC 7. The components *{B, D}* and *{E, F}* choose the same edge, DF 3. The two components on the right side both choose KL 12. Three distinct edges are added and two components remain. In the third round only two edges connect the two halves: CH 20 and EJ 18. The cheaper one is EJ 18, and both components choose it. The MST is complete after three rounds and has weight 83.

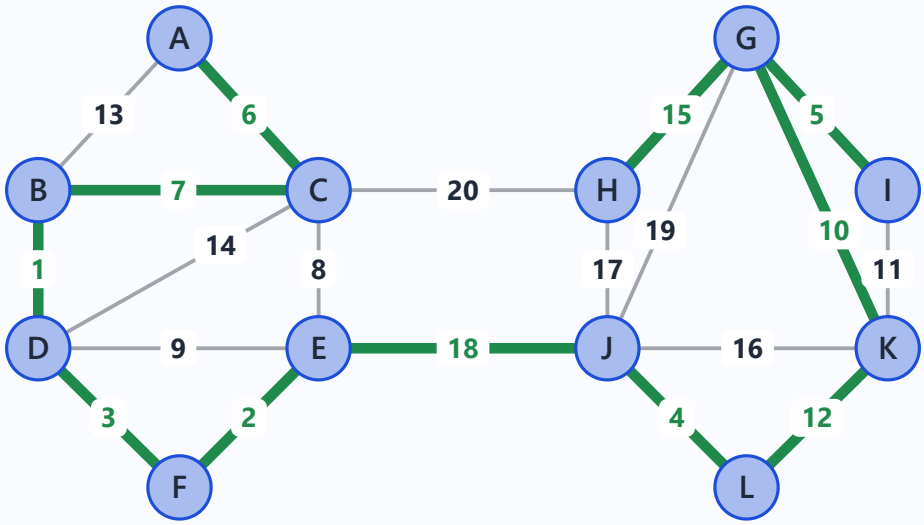


Figure 7.29: End of round 3: EJ has been added; all twelve vertices are in one component. The MST has weight 83.

On this graph Kruskal adds 11 edges in 11 separate steps. Borůvka builds the same tree in three rounds: 7 + 3 + 1 edges. Each round scans all edges once; the total work is $3 \cdot 20 = 60$ edge examinations.

7.7.5 Java code and trace

The Java code on the slides gives the body of the main loop. `edges` is the edge list, `parent` the parent array of the disjoint sets, `tree` the chosen edges, and `componentCount` the number of remaining components, which starts at *V*.

```
while (componentCount > 1) {
  Edge[] cheapest = new Edge[V];
  for (Edge e : edges) {            // 1) find cheapest
    int r1 = find(parent, e.u), r2 = find(parent, e.v);
    if (r1 == r2) continue;         // same component
    if (cheapest[r1] == null || e.w < cheapest[r1].w)
      cheapest[r1] = e;
    if (cheapest[r2] == null || e.w < cheapest[r2].w)
      cheapest[r2] = e;
  }
  boolean added = false;
  for (Edge e : cheapest) {         // 2) add them all
    if (e == null) continue;
    int r1 = find(parent, e.u), r2 = find(parent, e.v);
    if (r1 == r2) continue;         // already added
    tree.add(e);
    parent[r1] = r2;                // union
    componentCount--;
    added = true;
  }
  if (!added) break;  // forest
}
```

In each round the `cheapest` array is created anew. The index of the array is the root of a component; the cell holds the cheapest outgoing edge of that component found so far. The first loop scans all edges. If the roots of an edge's two endpoints are the same, the edge lies inside a component and is passed over with `continue`. Otherwise the edge is a candidate for both roots: for each root, it is written into the cell if the cell is empty or the new edge is cheaper.

The second loop walks through the `cheapest` array. Empty cells belong to vertices that are not roots and are skipped. For each candidate edge the roots are found again. If the roots are the same, the edge has already been added in this round, or the two components have been joined by another edge; the edge is skipped. Otherwise the edge enters the tree, `parent[r1] = r2` merges the components, the component count drops by one, and `added` becomes true. If no edge is added in a round, no component has an outgoing edge, that is, the graph is not connected; `break` leaves the loop and `tree` is a minimum spanning forest. Without this check `componentCount` would never fall to 1 and the loop would run forever.

The code is traced on the six-vertex example. The vertices are A–F with indices 0–5. The edges stand in the list in this order: AB, BC, AD, BE, BD, BF, CF, DE, EF.

Stage	Event	`componentCount`
Round 1, scan	AB(1): `cheapest[A] = cheapest[B] = AB`. BC(5): `cheapest[C] = BC`. AD(4): `cheapest[D] = AD`. BE(3): `cheapest[E] = BE`. BD(7): no change. BF(6): `cheapest[F] = BF`. CF(9), DE(8): no change. EF(2): `cheapest[E] = cheapest[F] = EF`.	6
Round 1, add	`cheapest[A] = AB`: roots A, B → add, `parent[A] = B`	5
	`cheapest[B] = AB`: both endpoints in the same component → skip	5
	`cheapest[C] = BC`: roots B, C → add, `parent[B] = C`	4
	`cheapest[D] = AD`: roots C, D → add, `parent[C] = D`	3
	`cheapest[E] = EF`: roots E, F → add, `parent[E] = F`	2
	`cheapest[F] = EF`: same component → skip	2
Round 2, scan	AB, BC, AD, BD, EF stay inside components. Of the edges between components, BE(3), BF(6), CF(9), DE(8), the cheapest is BE: `cheapest[D] = cheapest[F] = BE`.	2
Round 2, add	`cheapest[D] = BE`: roots D, F → add, `parent[D] = F`	1
	`cheapest[F] = BE`: same component → skip; loop ends	1

Table 7.17: Trace of the Borůvka code (six-vertex example).

In the scan of round 1, E's candidate is first BE(3) and then changes when EF(2) comes. F's candidate is likewise first BF(6) and then EF(2). In the add loop, when AD is added, `find(A)` follows the path $A \to B \to C$ and finds C. Since the roots are C and D, `parent[C] = D` is written. In round 2 the component roots are D and F, so the candidate edge is written into the cells `cheapest[D]` and `cheapest[F]`. The resulting weight is 15.

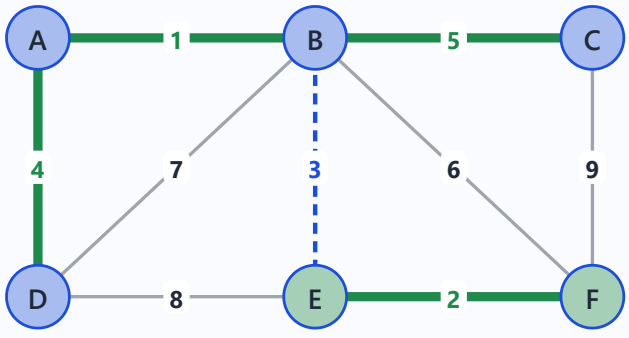


Figure 7.30: The scan of round 2: the cheapest edge between the two components (colors), BE(3), is shown dashed in blue.

7.7.6 Complexity and tie-breaking

Theorem 7.8 (Number of Borůvka rounds)

On a connected graph with V vertices, Borůvka runs at most $\lceil \log_2 V \rceil$ rounds, and its total running time is $O(E \log V)$.

Proof. Suppose $c > 1$ components at the start of a round. Since the graph is connected, every component has at least one outgoing edge, and every component chooses an edge. The chosen edge joins the component with at least one other component. Each new component formed at the end of the round contains at least two old components. Hence the new number of components is at most $\lfloor c/2 \rfloor$. After k rounds at most $V/2^k$ components remain; once $k \geq \log_2 V$, this number drops to 1. Each round scans all edges once and makes two `find` calls per edge. If the nearly constant cost of disjoint sets is ignored, a round takes $O(E)$. The total is $O(E \log V)$. □

The observation $\lfloor c/2 \rfloor$ in the proof actually gives a slightly tighter bound. For $V = 1000$ the number of components falls in the worst case as 1000, 500, 250, 125, 62, 31, 15, 7, 3, 1; 9 rounds suffice. $\lceil \log_2 1000 \rceil = 10$ is a safe upper bound. In practice the number of rounds is often even smaller, because more than two components can merge in one round. The twelve-vertex example finished in 3 rounds; $\lceil \log_2 12 \rceil = 4$. Borůvka does not sort the edges; it only compares weights.

Equal weights create a special problem in Borůvka. As an example take a triangle a, b, c whose edges all have weight 1. If ties are resolved arbitrarily, a may choose ab, b may choose bc, and c may choose ca. If all three edges are added at once, a cycle forms. To prevent this, a consistent **tie-breaking rule** is needed: when weights are equal, edges are compared by, say, their indices in the list. Then all the edges fall into a strict order and no cycle can form. When edges are added simultaneously (for example in a parallel implementation), such a consistent tie-breaker is required. In the sequential code on the slides, however, the check `find(r1) == find(r2)` in the second loop already prevents a cycle: an edge whose endpoints are now in the same component is not added.

Common mistake

Recomputing the edges chosen in a round as the components are updated within the same round. In Borůvka all of a round's choices are made with respect to the components at the start of the round; the additions come afterward. If selection and addition are interleaved, the algorithm turns into something between Kruskal and Prim, and the bound on the number of rounds loses its validity.

7.8 The reverse-delete algorithm

7.8.1 The idea: Kruskal's mirror image

The reverse-delete algorithm is also greedy and was proposed in Kruskal's 1956 paper. Whereas Kruskal starts from an empty graph and adds small edges, reverse-delete starts from the whole graph and tries to delete large edges. The edges are sorted by weight in descending order. Starting from the heaviest edge, every edge whose deletion does not break the connectivity of the graph is deleted. An edge whose deletion would disconnect the graph is left in place.

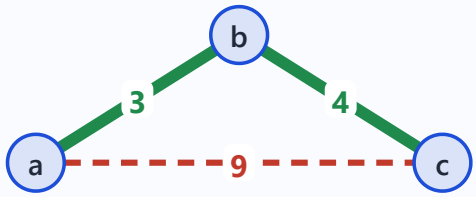


Figure 7.31: The basic step of reverse-delete: the heaviest edge of the cycle a–b–c, of weight 9, is deleted; a and c are still connected through b.

The algorithm rests on the cycle property. If the graph stays connected when an edge is deleted, there is another path between its endpoints; the edge lies on a cycle. Because edges are processed in descending order, the remaining edges of this cycle are at most as heavy as it is. The heavier edges have either already been deleted or remain because they are bridges. So the deleted edge is the heaviest edge of a cycle in the remaining graph, and by the cycle property an MST does not need it. An edge that cannot be deleted is a bridge at that moment and must be in every spanning tree.

Reverse-delete has mostly theoretical and instructional value. Every deletion requires a connectivity test, and this test is expensive; in practice the algorithm is usually slow.

7.8.2 Steps, pseudocode, and complexity

1. Sort all the edges in descending order of weight.
2. Delete the next heaviest edge.
3. If the graph is no longer connected, put the deleted edge back.
4. Repeat steps 2 and 3 until all edges have been processed.

```
REVERSE_DELETE(G):
  MST = all edges
  edges.sort(descending)
  for each (u, v, weight) in edges:
    MST.remove((u, v, weight))
    disjointSet.reset()
    for each (x, y) in MST:
      disjointSet.union(x, y)
    root = disjointSet.find(V[0])
    if some v has disjointSet.find(v) ≠ root:
      MST.add((u, v, weight))  // put back
  return MST
```

The pseudocode tests connectivity with disjoint sets. After every deletion the structure is reset and `union` is called for all remaining edges. Then the root of the first vertex is found. If some vertex has a different root, the graph has fallen apart and the deleted edge is put back once. The edge put back is the edge *(u, v, weight)* deleted at the start of the loop; the *v* in the condition line is used only in the sense of "some vertex," and the two names should not be confused.

The complexity calculation is simple. A connectivity test is made for every edge. The test takes $O(V + E)$ with BFS or DFS, and rebuilding with disjoint sets is of the same order. E tests come to $O(E \cdot (V + E))$ in total. The $O(E \log E)$ cost of sorting is small next to this. For a graph with $V = 1000$ and $E = 5000$, this means about $3 \cdot 10^7$ steps; Kruskal makes do with about $6 \cdot 10^4$ steps on the same graph.

7.8.3 A seven-vertex example

Reverse-delete is run on a graph with vertices 0–6 and 11 edges. The total weight of the edges is 90. The descending order is as follows:

> 3–4 15, 5–6 11, 1–3 9, 4–6 9, 1–2 8, 4–5 8, 1–4 7, 0–1 7, 3–5 6, 2–4 5, 0–3 5

Edge	*w*	Still connected if deleted?	Result
3–4	15	yes	deleted
5–6	11	yes	deleted
1–3	9	yes	deleted
4–6	9	no: 6 would be isolated	kept
1–2	8	yes	deleted
4–5	8	yes	deleted
1–4, 0–1, 3–5, 2–4, 0–3	7, 7, 6, 5, 5	no: each is a bridge	kept

Table 7.18: Reverse-delete on the seven-vertex graph.

The first three deletions remove cycle edges. When 3–4 = 15 is deleted, 3 and 4 stay connected through the path $3 \to 0 \to 1 \to 4$. When 4–6 = 9 comes up, 5–6 has already been deleted, and the only edge of 6 is 4–6. If this edge were deleted, 6 would be isolated, so it is put back. 1–2 = 8 and 4–5 = 8 can be deleted, because 2 is reached through 2–4 and 5 through 3–5. At this point the remaining six edges form a tree and each is a bridge; none can be deleted. The number of remaining edges is $6 = V - 1$, and the MST weight is $9 + 7 + 7 + 6 + 5 + 5 = 39$.

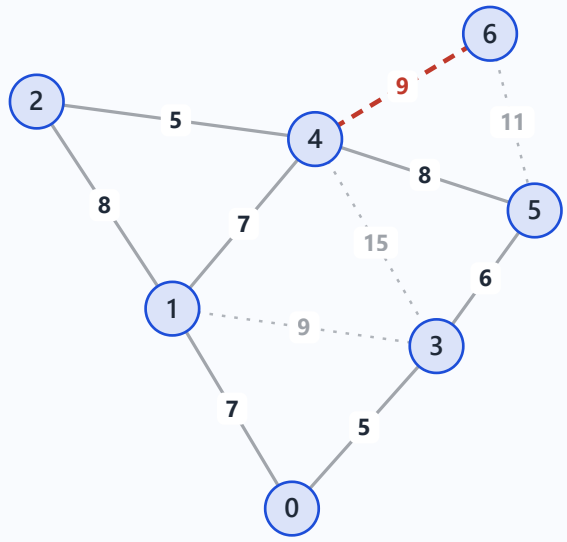


Figure 7.32: The step 4–6 = 9. The gray dotted edges have been deleted. Since 5–6 has been deleted, the only connection of 6 is 4–6 (red dashed); if the edge were deleted, 6 would be cut off, so it is put back.

On the same graph Kruskal first adds the edges of weight 5, 5, 6, 7, 7. It then skips the two edges of weight 8 and the edge 1–3 of weight 9 because they form cycles. Finally it adds 4–6 = 9. The two algorithms reach the same tree from opposite directions: Kruskal decides what to add, reverse-delete decides what to keep.

7.8.4 Java code and trace

```
void reverseDelete(int V, List<Edge> edges) {
  edges.sort((a, b) -> b.w - a.w); // descending
  int i = 0;
  while (i < edges.size()) {
    Edge e = edges.remove(i);   // delete tentatively
    if (!isConnected(V, edges)) {
      edges.add(i, e);          // put back
      i++;
    }
  }
}
```

`edges.sort((a, b) -> b.w - a.w)` puts the edges in descending order; writing the comparator as `b - a` reverses the order. `i` is the index of the next edge to examine in the list. In each round of the loop the edge is tentatively deleted with `edges.remove(i)`. After the deletion the list shifts by one and the next edge comes to index `i` again; so if the deletion succeeds, `i` is not incremented. If deleting the edge disconnects the graph, `edges.add(i, e)` puts the edge back in its old place and `i` increases by one. The loop ends when `i` reaches the length of the list. At that point the edges remaining in the list are the MST. The `isConnected` method used by the code can be written with BFS:

```
boolean isConnected(int V, List<Edge> edges) {
  List<List<Integer>> adj = new ArrayList<>();
  for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
  for (Edge e : edges) { adj.get(e.u).add(e.v); adj.get(e.v).add(e.u); }
  boolean[] seen = new boolean[V];
  ArrayDeque<Integer> queue = new ArrayDeque<>();
  queue.add(0); seen[0] = true;
  int count = 1;
  while (!queue.isEmpty()) {
    int x = queue.poll();
    for (int y : adj.get(x))
      if (!seen[y]) { seen[y] = true; count++; queue.add(y); }
  }
  return count == V;            // reached every vertex?
}
```

The method first builds the adjacency list from the edge list, then starts a BFS from vertex 0 and counts the vertices reached. If the count equals V, the graph is connected. Each call takes $O(V + E)$; since there are E calls, the total cost is the $O(E \cdot (V + E))$ computed above.

The code is traced on the lettered version of the same seven-vertex graph: vertices 0–6 are named A–G. The edge list is given in alphabetical order: AB 7, AD 5, BC 8, BD 9, BE 7, CE 5, DE 15, DF 6, EF 8, EG 9, FG 11. The sort is stable; edges of equal weight keep this order.

Edge	`isConnected`	Action	`i` (after)
DE 15	true	stays deleted	0
FG 11	true	stays deleted	0
BD 9	true	stays deleted	0
EG 9	false: G is isolated	put back	1
BC 8	true	stays deleted	1
EF 8	true	stays deleted	1
AB 7	false: {A, D, F} gets cut off	put back	2
BE 7	false: {C, E, G} gets cut off	put back	3
DF 6	false: {F} gets cut off	put back	4

Edge	`isConnected`	Action	`i` (after)
AD 5	false: {D, F} gets cut off	put back	5
CE 5	false: {C} gets cut off	put back	6

Table 7.19: Trace of `reverseDelete(7, edges)`.

At the EG step FG has already been deleted, and the only connection of G is EG. When the AB step is reached, the edges of A are AB and AD, the remaining edges of D are AD and DF, and the remaining edge of F is only DF. If AB were deleted, A, D, and F would be cut off from the other vertices. After the last row `i = 6` equals the length of the list and the loop ends. The remaining tree consists of the edges AB, AD, BE, CE, DF, and EG; its weight is 39.

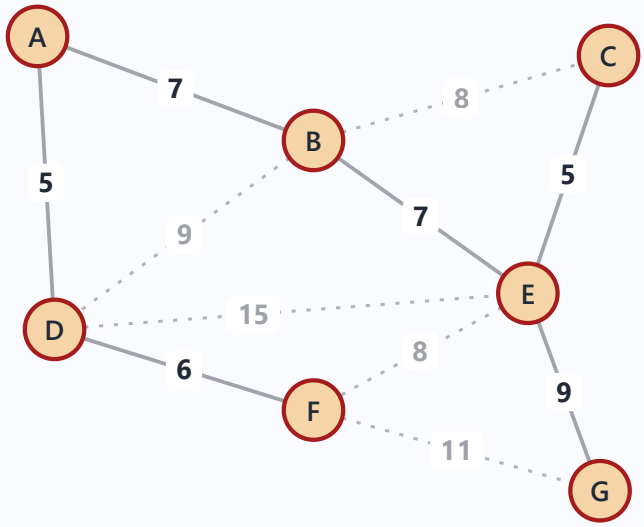


Figure 7.33: The result of reverse-delete: AB, AD, BE, CE, DF, EG; total 39. The gray dotted edges have been deleted.

Common mistake

Incrementing `i` even when the deletion succeeds. `edges.remove(i)` shifts the list one place to the left; the next edge is now at index `i`. If `i` is incremented, an edge is skipped without being examined after every successful deletion, and the result may be not a tree but a subgraph containing superfluous edges.

7.9 Comparing the four algorithms

The four algorithms solve the same problem and rest on the same theoretical foundation, but they work differently and have different costs. Table 7.20 shows the differences side by side.

	Kruskal	Prim	Borůvka	Reverse-delete
Perspective	edge-oriented	vertex-oriented	component-oriented	edge-oriented (deletion)
Intermediate state	forest	single tree	forest	connected graph
Property used	cut	cut	cut	cycle
Data structure	sorted edges + disjoint sets	priority queue / array	disjoint sets	BFS/DFS
Time	$O(E \log E)$	$O(V^2)$ or $O((V + E) \log V)$	$O(E \log V)$	$O(E \cdot (V + E))$
Best suited for	sparse graphs	dense graphs	parallel / distributed	teaching, theory
Disconnected graph	gives a forest	one component	gives a forest	preserves components

Table 7.20: Comparison of the four MST algorithms.

The rows of the table correspond to a few practical questions. If the graph arrives as an edge list and is sparse, Kruskal is the easiest and fastest choice. If the graph is given as an adjacency matrix and is dense, Prim with an array scan gives the best result with $O(V^2)$; no sorting or heap is needed. When a sparse graph is given as an adjacency list, Prim with a heap is of the same order as Kruskal. If the work is to be distributed across several processors, the independent choices of Borůvka are a great convenience. Reverse-delete is hardly used in production code; it is instructive as an algorithmic counterpart of the cycle property.

Behavior on disconnected graphs also matters. Kruskal and Borůvka complete each component separately and return a minimum spanning forest. Prim covers only the component of the starting vertex; to find the forest it must be restarted from each component it has not covered. Reverse-delete never disconnects a component, so it preserves the components as they are; it is enough to write the connectivity test so that it preserves the number of components.

The nine-vertex graph shows the difference concretely. Kruskal and Prim find the same tree (29). Kruskal adds the edges in the order *1, 1, 2, 3, 4, 5, 6, 7*, Prim in the order *4, 7, 1, 1, 2, 3, 6, 5*. Kruskal's insertion order is weight order; Prim's follows the direction in which the tree grows.

These four algorithms are not the end of the story. In 1995 Karger, Klein, and Tarjan gave a randomized algorithm with linear expected time. In 2000 Bernard Chazelle published a deterministic algorithm whose running time involves the inverse Ackermann function and is nearly linear. Whether a deterministic MST algorithm with strictly linear time exists is still an open question.

7.10 Common mistakes

This section gathers the warnings scattered through the chapter, together with the list of mistakes from the slides.

Common mistake

Mistaking the MST for a shortest-path tree. The MST minimizes the total weight; the MST path between two vertices need not be a shortest path. In the A–F graph the B–F path in the MST is $B \to D \to E \to F$ and has length 7; the direct edge BF has weight 5.

Common mistake

Testing for a cycle by adjacency. In Kruskal the question "are u and v directly connected?" is not enough. The question to ask is whether the two endpoints are in the same component: `find(u) == find(v)`.

Common mistake

Adding distances in Prim. Writing `key[u] + w` gives Dijkstra. In Prim the key is the weight of a single edge only.

Common mistake

Fearing negative weights. MST algorithms work correctly with negative weights too, because they only compare weights. The proofs of the cut and cycle properties do not look at the sign of the weights either. The problem with negative weights is specific to Dijkstra. In fact, if the same constant is added to all weights, the MST does not change. The order of the edges is preserved. Since every spanning tree has $V - 1$ edges, the totals of all trees increase by the same amount.

Common mistake

Forgetting disconnected graphs. A disconnected graph has no MST; it has a minimum spanning forest. The condition `while (tree.size() < V - 1)` runs past the end of the list when the edges run out. The condition `k < edges.size()` must be added.

Common mistake

Applying the algorithms to a directed graph. These algorithms are for undirected graphs. In a directed graph, the cheapest structure that reaches every vertex from a root is called a **minimum spanning arborescence**, and the problem is solved by the Chu–Liu/Edmonds algorithm. Applying Kruskal or Prim unchanged to directed edges gives a wrong result.

7.11 Summary

- **Spanning tree and MST:** a connected, acyclic subgraph that contains all the vertices of a connected, undirected graph is a spanning tree and has exactly $V - 1$ edges. The one of smallest total weight is called a minimum spanning tree. A complete graph has n^{n-2} spanning trees, so greedy algorithms are used instead of trying them all.
- **Cut property:** the lightest edge of every cut belongs to some MST; if the lightest edge of the cut is unique, it is in every MST. Kruskal, Prim, and Borůvka rest on this property.
- **Cycle property:** the unique heaviest edge of a cycle is in no MST. Reverse-delete rests on this property. If all weights are distinct, the MST is unique.
- **Kruskal:** edges are examined in ascending order; if `find(u) ≠ find(v)`, the edge is added and two trees merge. The time is $O(E \log E) = O(E \log V)$, and the sort dominates.
- **Disjoint sets:** in the `parent[]` forest each set is a tree and the root is the representative. Path compression and union by rank together give an amortized $O(\alpha(V))$ per operation; α does not exceed 4 in practice. With rank alone each operation is $O(\log V)$.
- **Prim:** a single tree is grown; `key[v]` is the weight of the cheapest single edge connecting v to the tree. It takes $O(V^2)$ with an array scan, $O((V + E) \log V)$ with a binary heap, $O(E \log E)$ with the lazy version, and $O(E + V \log V)$ with a Fibonacci heap.
- **Prim and Dijkstra:** the code is almost identical; Prim makes a single edge weight the key, Dijkstra the total distance from the source. Prim's result has the same weight regardless of the source and is also correct with negative weights.
- **Borůvka:** in each round every component chooses its cheapest outgoing edge and these edges are added at once. The number of components at least halves every round; at most $\lceil \log_2 V \rceil$ rounds, total $O(E \log V)$. With equal weights a consistent tie-breaker is needed.
- **Reverse-delete:** edges are deleted in descending order; if the graph falls apart, the edge is put back. Because a connectivity test is made at every deletion, the time is $O(E \cdot (V + E))$.
- **Choosing:** sparse graph and edge list → Kruskal; dense graph and adjacency matrix → Prim with an array scan; parallel computation → Borůvka; forest in a disconnected graph → Kruskal or Borůvka.

7.12 Key terms

spanning tree minimum spanning tree (MST) minimum spanning forest connected component cut cut property cycle property bridge greedy algorithm exchange argument Kruskal's algorithm union-find, disjoint sets path compression union by rank amortized analysis Prim's algorithm priority queue parent adjacency matrix adjacency list Dijkstra's algorithm Borůvka's algorithm reverse-delete algorithm bottleneck

7.13 Test Yourself

Questions 1–6 use the same graph K. The vertices of K are A–F, and its edges are $AB = 7$, $AC = 3$, $AD = 6$, $BC = 4$, $BE = 2$, $CD = 5$, $CE = 8$, $CF = 9$, $DF = 1$, and $EF = 10$.

1. Which edges cross the cut $\{A, C\} \mid \{B, D, E, F\}$ in K? What does the cut property say, and about which edge?

   Answer

   The edges crossing the cut are AB 7, AD 6, BC 4, CD 5, CE 8, and CF 9. AC 3 does not cross the cut, because both its endpoints are on the same side. The lightest crossing edge is BC 4, and since it is the unique lightest one, it is in every MST.

2. Which edges does Kruskal add in K, and in what order? How many edges are rejected, and which edges are never examined? What is $w(T)$?

   Answer

   The sorted list is DF 1, BE 2, AC 3, BC 4, CD 5, AD 6, AB 7, CE 8, CF 9, EF 10. Each of the first five edges joins two separate trees and is added; no edge is rejected. Since the fifth addition reaches $V - 1 = 5$, AD, AB, CE, CF, and EF are never examined. $w(T) = 1 + 2 + 3 + 4 + 5 = 15$.

3. If Prim is run on K from A, in what order do the vertices enter the tree? What is the key array after C enters the tree?

   Answer

   The order is A (0), C (3), B (4), E (2), D (5), F (1). After A, the keys are B = 7, C = 3, D = 6, E = F = ∞. When C enters, B drops to 4 through BC and D to 5 through CD; E = 8 and F = 9. The array for A, …, F is $0, 4, 3, 5, 8, 9$. The parents end up as B → C, D → C, E → B, F → D; the total is again 15.

4. In how many rounds does Borůvka finish on K? Which edges are added in each round?

   Answer

   In round 1, A and C both choose AC 3, B and E both choose BE 2, and D and F both choose DF 1. Three distinct edges are added, and the components $\{A, C\}$, $\{B, E\}$, and $\{D, F\}$ form. In round 2, $\{A, C\}$ and $\{B, E\}$ choose BC 4, and $\{D, F\}$ chooses CD 5. Two edges are added and a single component remains: 2 rounds, weight 15.

5. Which edges does reverse-delete delete on K? At which edge is the first “kept” decision made, and

why?

Answer

The deleted edges are EF 10, CF 9, CE 8, AB 7, and AD 6. The first "kept" decision is made at CD 5: after AD and CF are deleted, CD is the only edge connecting D and F to the remaining vertices. After that BC, AC, BE, and DF are also kept because they are bridges.

6. If Dijkstra is run on *K* from A, which edges form the shortest-path tree and what is its weight? How does this tree differ from the tree Prim finds from A?

Answer

The distances are B = 7, C = 3, D = 6, E = 9 (A → B → E), F = 7 (A → D → F). The tree consists of the edges AB, AC, AD, BE, DF and has weight *7 + 3 + 6 + 2 + 1 = 19*. Prim's tree (AC, BC, CD, BE, DF) has weight 15. Dijkstra reaches B directly by AB 7; the path A → C → B is also 7 and is not shorter. Prim, on the other hand, connects B to C by the edge of weight 4.

7. A disjoint-set structure with elements 0–6 uses `find` with path compression and `union` from the slides. The calls `union(0, 1)`, `(2, 3)`, `(0, 2)`, `(4, 5)`, `(6, 4)`, `(5, 0)` are made in this order. What is the final `parent` array? What does `find(6)` change afterwards?

Answer

Steps: `parent[0] = 1`; `parent[2] = 3`; `find(0) = 1`, `find(2) = 3` → `parent[1] = 3`; `parent[4] = 5`; `find(6) = 6`, `find(4) = 5` → `parent[6] = 5`. In the last call `find(5) = 5`; `find(0)` follows the path 0 → 1 → 3 and attaches 0 directly to 3. Then `parent[5] = 3` is written. The array becomes `[3, 3, 3, 3, 5, 3, 5]`. `find(6)` follows the path 6 → 5 → 3 and sets `parent[6] = 3`: `[3, 3, 3, 3, 5, 3, 3]`. Since 4 is not on this path, it stays attached to 5.

8. How is the *maximum* spanning tree of *K* found, and what is its weight?

Answer

Kruskal processes the edges in descending order (or all weights are replaced by $-w$ and ordinary Kruskal is run). The order is EF 10, CF 9, CE 8, AB 7, AD 6, CD 5. EF and CF are added; CE forms a cycle (C–F–E); AB and AD are added; finally CD is added. The tree is EF, CF, AB, AD, CD and its weight is *10 + 9 + 7 + 6 + 5 = 37*.

9. How many edges does the minimum spanning forest have in a graph with 50 vertices and 3

components? What does the Kruskal code from the slides do on this graph?

Answer

Each component needs *(component size − 1)* edges; the total is $50 - 3 = 47$ edges. The code on the slides expects 49 edges. After adding 47 edges the list runs out and `edges.get(k++)` throws an `IndexOutOfBoundsException`.

10. Two versions of Prim are compared on a graph with $V = 10^4$ vertices: the array-scan version $O(V^2)$ and the binary-heap version $O((V + E) \log V)$. Which does less work for $E = 5 \cdot 10^4$ and for $E = 5 \cdot 10^7$?

Answer

$V^2 = 10^8$, and $\log_2 10^4 \approx 13.3$. For $E = 5 \cdot 10^4$ the heap version takes about $6 \cdot 10^4 \cdot 13.3 \approx 8 \cdot 10^5$ steps; it is better by more than a factor of a hundred. For $E = 5 \cdot 10^7$ (nearly all of a complete graph) the heap version takes about $6.6 \cdot 10^8$ steps; the array version is better by a factor of 6–7.

7.14 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 19 and 21: gives the inverse-Ackermann analysis of disjoint sets and the general MST method, with the notion of a safe edge, together with proofs.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Sections 1.5 and 4.3: presents disjoint sets, lazy and eager Prim, and Kruskal with Java code and experimental measurements.
- R. L. Graham and P. Hell, “On the History of the Minimum Spanning Tree Problem”, *Annals of the History of Computing* 7(1), 1985: compares the work of Borůvka, Jarník, Kruskal, and Prim and examines the early history of the problem.
- R. E. Tarjan, “Efficiency of a Good But Not Linear Set Union Algorithm”, *Journal of the ACM* 22(2), 1975: the paper that first proved the inverse-Ackermann bound for path compression and union by rank.

7.15 References

1. O. Borůvka, “O jistém problému minimálním”, *Práce Moravské přírodovědecké společnosti* 3, 1926.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 19: disjoint sets, Chapter 21: MST).
3. R. L. Graham, P. Hell, “On the History of the Minimum Spanning Tree Problem”, *Annals of the History of Computing* 7(1), 43–57, 1985.

4. J. B. Kruskal, “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem”, *Proceedings of the American Mathematical Society* 7(1), 1956.
5. R. C. Prim, “Shortest Connection Networks and Some Generalizations”, *Bell System Technical Journal* 36(6), 1957.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 4.3: Minimum Spanning Trees).
7. R. E. Tarjan, “Efficiency of a Good But Not Linear Set Union Algorithm”, *Journal of the ACM* 22(2), 1975.

PART II · CHAPTER 8

Network Flow

A city's water network, a country's highway system, and the links of a data center all ask the same question: how much can be sent from one point to another without exceeding the carrying limits of the links? On graphs this question is written as the **maximum flow problem**. The chapter first defines flow networks, the constraints a flow must satisfy, and the value of a flow. It then builds the concepts that all the algorithms share: the residual network, the augmenting path, and the cut. The max-flow min-cut theorem proves why a flow that has been found is optimal. The Ford-Fulkerson method, Edmonds-Karp, Dinic's algorithm, and the push–relabel algorithm are then run step by step on the same example networks. The code of each is traced line by line, and the source of its running-time bound is shown. The chapter closes with a comparison of the algorithms and applications such as bipartite matching.

LEARNING OUTCOMES

Modeling

Model a problem as a flow network of a source, a sink, and edges with capacities; write the capacity and conservation constraints of a flow.

Residual network and cut

Compute the residual network, an augmenting path, and its bottleneck; find the capacity of an s–t cut and apply the max-flow min-cut theorem.

Algorithms

Run the Ford-Fulkerson, Edmonds-Karp, Dinic, and push–relabel algorithms step by step on an example network.

Analysis

Explain where the bounds $O(E \cdot |f^*|)$, $O(V \cdot E^2)$, and $O(V^2 \cdot E)$ come from; compare the algorithms and choose a suitable one.

8.1 Flow networks

The questions asked about graphs so far have mostly concerned paths: is there a path between two vertices, which path is shortest, which tree connects all vertices most cheaply? Network flow asks something different. Finding one path is not enough; many paths are used at once, and they share edges. Every edge has a carrying limit. The question is how much all the paths can carry together.

This question comes up often in everyday life. Water in a pipeline, traffic on a road network, and data packets in a computer network are all flows. Each pipe carries at most its own capacity. Water does not accumulate at junctions: whatever enters a junction also leaves it. These two observations become the two basic constraints of the flow network model.

8.1.1 The maximum flow problem

The model is a directed graph with two special vertices. The vertex the flow leaves is called the **source**, and the vertex it arrives at is the **sink**. The source is written s and the sink t. Every edge has a **capacity**: the largest amount that can pass through that edge per unit time.

Definition 8.1

A **flow network** consists of a directed graph $G = (V, E)$, two distinct vertices $s, t \in V$, and a function that assigns each edge (u, v) a nonnegative capacity $c(u, v) \geq 0$. For a pair (u, v) that is not in the graph, $c(u, v) = 0$ is assumed. The **maximum flow problem** is to find the largest amount of flow that can be sent from the source to the sink without exceeding the capacities.

The same small network is used for much of the chapter. Figure 8.1 shows it. There are six vertices and seven edges; the edge labels are capacities. The source is S and the sink is T.

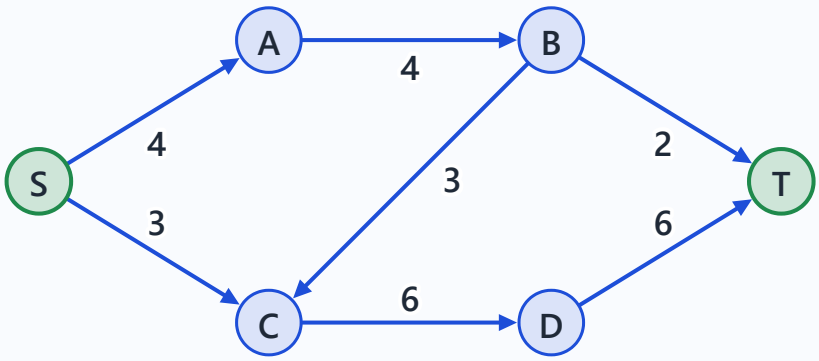


Figure 8.1: The example flow network. The edge labels are capacities: $S \to A$ 4, $S \to C$ 3, $A \to B$ 4, $B \to C$ 3, $B \to T$ 2,

$C \to D$ 6, $D \to T$ 6. The maximum flow in this network is 7.

That the maximum flow is 7 must be seen from two sides. First, 7 units can be sent; a flow that shows this will be given shortly. Second, more than 7 cannot be sent. This second claim is clear at once from the source. The capacities of the two edges leaving S sum to $4 + 3 = 7$. Every unit leaving the source has to use one of these two edges. Hence no more than 7 can reach the sink. This simple observation is the first example of the *cut* concept that comes later.

The same reasoning can be made at the sink. The capacities of the edges entering T sum to $2 + 6 = 8$. This is also an upper bound, but it is looser than 7. The smaller of the two bounds is the valid one. An upper bound that is always correct is this: the maximum flow cannot exceed the sum of the capacities leaving the source or the sum of the capacities entering the sink. This bound is not always tight; the bottleneck can also lie in the middle of the network.

8.1.2 Flows and their constraints

A flow assigns each edge the amount passing through it. For this assignment to make sense, two rules must hold.

Definition 8.2

A **flow** is a function that assigns each edge (u, v) a number $f(u, v)$ and satisfies the following two constraints:

- **Capacity constraint:** for every edge, $0 \le f(u, v) \le c(u, v)$.
- **Flow conservation:** for every vertex v other than s and t, $\Sigma_u f(u, v) = \Sigma_w f(v, w)$; that is, the flow in equals the flow out.

The **value** of a flow is $|f| = \Sigma_v f(s, v) - \Sigma_v f(v, s)$: the net flow leaving the source.

The capacity constraint expresses the carrying limit of a pipe. A flow cannot be negative; a pipe does not run backward. Nor can it exceed the capacity. The conservation constraint says that intermediate vertices are not storage tanks. Every unit that enters an intermediate vertex must leave it somewhere. The source and the sink are exempt from this rule: the source produces flow and the sink consumes it.

The subtraction term in the definition of the value may seem unnecessary at first. In our example network no edge enters the source, so this term is zero. In a general network, however, there may be edges returning to the source. If 5 units leave the source and 1 unit returns, 4 units have actually been sent into the network. This is why the value is the *net* flow leaving the source. Thanks to conservation, this net flow equals the net flow entering the sink: since no intermediate vertex accumulates flow, everything leaving the source eventually arrives at the sink.

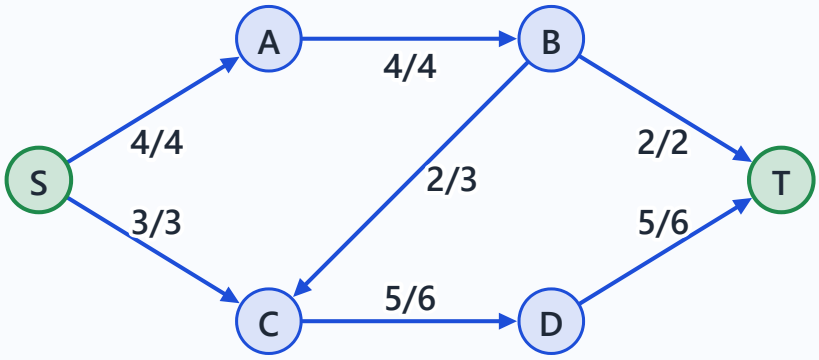


Figure 8.2: A flow of value 7 in the same network. The labels have the form *flow/capacity*. Into C come $3 + 2 = 5$ units and 5 units leave; $|f| = 4 + 3 = 7$.

Example 8.1

Let us check that the flow in Figure 8.2 is valid.

Capacity constraint. Edge by edge: $4 \le 4$, $3 \le 3$, $4 \le 4$, $2 \le 3$, $2 \le 2$, $5 \le 6$, $5 \le 6$. No edge exceeds its limit.

Conservation. There are four intermediate vertices.

- A: in $f(S, A) = 4$, out $f(A, B) = 4$.
- B: in $f(A, B) = 4$, out $f(B, C) + f(B, T) = 2 + 2 = 4$.
- C: in $f(S, C) + f(B, C) = 3 + 2 = 5$, out $f(C, D) = 5$.
- D: in $f(C, D) = 5$, out $f(D, T) = 5$.

Value. $|f| = f(S, A) + f(S, C) = 4 + 3 = 7$. Check from the sink side: $f(B, T) + f(D, T) = 2 + 5 = 7$. The two numbers are equal.

In this flow some edges are filled to their limit. An edge with $f(u, v) = c(u, v)$ is called a **saturated** edge; no more flow can be sent through it. In the flow shown, $S \to A$, $S \to C$, $A \to B$, and $B \to T$ are saturated. That both edges leaving the source are saturated explains why the flow cannot be increased. Saturated edges are part of the cuts that explain why the maximum flow cannot be larger.

Common mistake

Thinking that the value of a flow is the sum of the flows on all edges. In the flow shown, the edge flows sum to $4 + 3 + 4 + 2 + 2 + 5 + 5 = 25$, but the value of the flow is 7. The same unit of flow passes through many edges along its path and is counted again on each. The value is read

only from the edges at the source (or the sink).

8.1.3 Multiple sources and sinks

Real problems rarely have a single source and a single sink. Several factories ship goods to several warehouses; several servers stream data to several groups of users. The way to solve this without a separate algorithm is a small graph transformation.

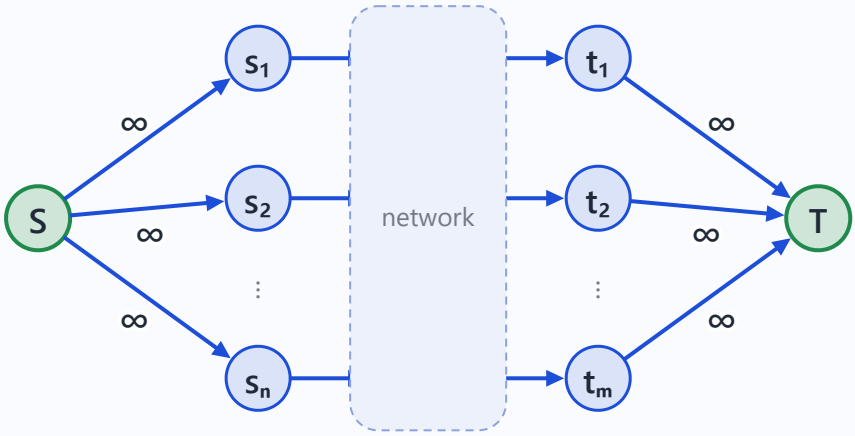


Figure 8.3: Supersource and supersink. A new vertex S is joined by edges of infinite capacity to all the sources s_i, and all the sinks t_j are joined by edges of infinite capacity to a new vertex T.

Two new vertices are added to the network: a **supersource** S and a **supersink** T. For each old source s_i an edge $S \to s_i$ is added, and for each old sink t_j an edge $t_j \to T$. These edges are given unlimited (∞) capacity. The new edges therefore never become a bottleneck; the flow is limited only by the edges of the old network. In the new network the maximum flow from S to T equals the total flow that can be sent from all the sources to all the sinks in the old network. In practice, any number larger than the sum of all capacities of the old network can stand in for ∞.

The transformation can be adapted to other restrictions. For example, if factory s_1 can produce at most 50 units, the edge $S \to s_1$ is given capacity 50 instead of ∞. The same idea is used in problems where vertices also have capacities: a vertex is split into two parts, and an edge with the vertex's capacity is placed between the two parts. This method will be taken up again under applications.

8.2 The residual network and augmenting paths

The first way to find a maximum flow that comes to mind is a greedy method. Find a path from the source to the sink through edges that are not full, send as much flow along it as it can carry, and keep doing this until no path is left. This method sometimes gives the correct result and sometimes does

not. The problem is that early decisions cannot be undone. A badly chosen path can fill edges in a way that other paths could have used better. The solution is to build the possibility of undoing flow already sent into the model. The structure that does this is the residual network.

8.2.1 The residual network

Definition 8.3

Under a flow f, the **residual capacity** of an edge is defined in two ways. For every edge (u, v) of the original network:

- **Forward edge:** $c_f(u, v) = c(u, v) - f(u, v)$. This is the amount that can still be sent through the edge.
- **Backward edge:** $c_f(v, u) = f(u, v)$. This is the amount of the flow already sent that can be undone.

The **residual network** G_f consists only of the edges with $c_f > 0$.

The residual network shows the changes that can still be made under the current flow. A forward edge says "I can send more in this direction." A backward edge says "I can take back part of what I sent in this direction." Figure 8.4 shows this on a single edge. The edge $u \to v$ of capacity 5 carries 3 units of flow. In the residual network this edge turns into two edges: the forward edge $u \to v$ for the remaining 2 units and the backward edge $v \to u$ for the 3 units that can be undone.

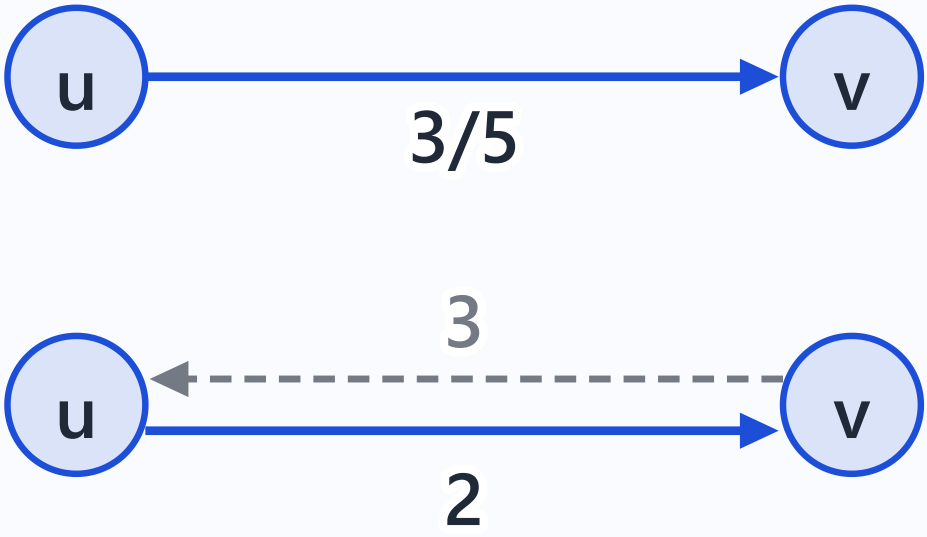


Figure 8.4: (a) An edge in the flow network: flow 3, capacity 5. (b) Its counterpart in the residual network: a forward edge of $5 - 3 = 2$ units (solid arrow) and a backward edge of 3 units (dashed arrow).

The two extreme cases are instructive too. If the flow is zero, the capacity of the backward edge is zero and the backward edge does not enter the residual network; the residual network is identical to the original network. If the edge is saturated, the capacity of the forward edge is zero and only the backward edge remains in the residual network. Each original edge produces at most two edges in the residual network. The number of edges of the residual network is therefore at most $2E$.

Example 8.2

Let us build the residual network of the flow in Figure 8.2. Two numbers are computed for each edge: the forward residual capacity $c - f$ and the backward residual capacity f.

- $S \to A$ (4/4): forward 0, backward $A \to S$ 4.
- $S \to C$ (3/3): forward 0, backward $C \to S$ 3.
- $A \to B$ (4/4): forward 0, backward $B \to A$ 4.
- $B \to C$ (2/3): forward $B \to C$ 1, backward $C \to B$ 2.
- $B \to T$ (2/2): forward 0, backward $T \to B$ 2.
- $C \to D$ (5/6): forward $C \to D$ 1, backward $D \to C$ 5.
- $D \to T$ (5/6): forward $D \to T$ 1, backward $T \to D$ 5.

Once the zeros are dropped, ten edges remain in the residual network. No residual edge leaves S; S has only incoming backward edges. Hence there is no path from S to T in the residual network. The meaning of this observation will be explained shortly: the flow is maximum.

8.2.2 What a backward edge means

A backward edge is not a physical pipe. There need not be a real connection in the direction $v \to u$. A backward edge is a bookkeeping entry: it shows that flow previously sent from u to v can be cancelled. "Sending" δ units along a backward edge means reducing $f(u, v)$ by δ.

Why does this cancellation help? If an augmenting path uses the backward edge $v \to u$, the path replaces the old flow arriving at v from u with its own. The old flow from u to v is now freed, and along the rest of the path it goes from u in another direction. In effect the flow is rearranged: some units abandon their old routes and switch to new ones. Conservation continues to hold at every vertex. Thanks to backward edges the algorithm can correct its first wrong choices. A concrete example of this correction appears in the second Ford-Fulkerson example.

Common mistake

Interpreting a backward edge as a real edge added to the network, or as "sending flow backward." Flow does not increase in the direction of the backward edge; the flow on the forward edge decreases: $f(u, v) \mathrel{-}= \delta$. For this reason no edge has negative flow in the final flow,

and backward edges add no capacity to the original network.

8.2.3 Augmenting paths and bottlenecks

Definition 8.4

An **augmenting path** is a directed path from the source to the sink in the residual network G_f. The **bottleneck** of the path is the smallest residual capacity among its edges: $c_f(p) = \min \{ c_f(u, v) : (u, v) \in p \}$.

Every edge on an augmenting path has positive residual capacity. That is, the path passes either through forward edges that carry less than their capacity or through the reverse directions of edges that carry flow. The most flow that can be sent along the path is the capacity of its narrowest edge. This is the flow counterpart of the saying that a chain is only as strong as its weakest link.

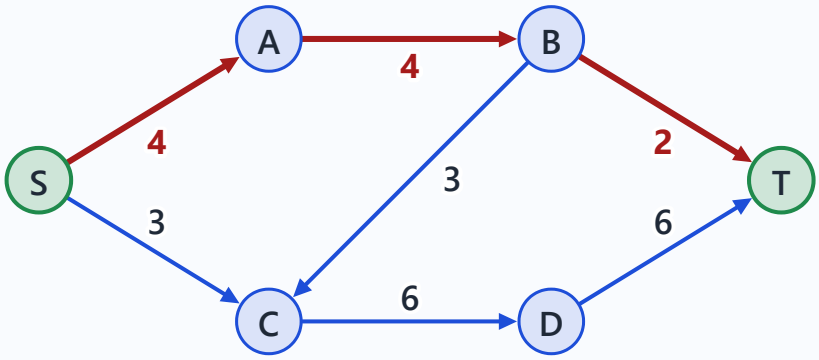


Figure 8.5: The augmenting path $S \to A \to B \to T$ (highlighted) in the initial residual network. The residual capacities are 4, 4, and 2; the bottleneck is $\min(4, 4, 2) = 2$, and it comes from the edge $B \to T$.

If the flow is initially zero everywhere, the residual network is identical to the original network and an augmenting path is an ordinary directed path. Augmenting paths are usually found with depth-first search (DFS) or breadth-first search (BFS). Which search is used does not change the value of the resulting flow, but it changes greatly how many steps the algorithm takes. This difference is the starting point of the Edmonds-Karp algorithm.

8.2.4 Augmenting the flow along a path

Once an augmenting path p and its bottleneck $\delta = c_f(p)$ have been found, the flow is updated as follows. For every edge (u, v) on the path:

- If (u, v) is an edge of the original network (a forward edge), $f(u, v)$ is increased by δ.
- If (u, v) is a backward edge, that is, the original edge is (v, u), then $f(v, u)$ is decreased by δ.

Theorem 8.1 (Augmentation)

Let f be a flow, p an augmenting path in the residual network, and $\delta = c_f(p)$. The f' obtained by the update above is again a flow, and $|f'| = |f| + \delta$.

Proof. **Capacity.** On a forward edge the new flow is $f(u, v) + \delta \le f(u, v) + c_f(u, v) = c(u, v)$. On a backward edge the new flow is $f(v, u) - \delta \ge f(v, u) - c_f(u, v) = 0$. In both cases the flow stays in the interval $[0, c]$.

Conservation. Consider an intermediate vertex v on the path. The path enters v along one edge and leaves along one edge. If the incoming edge is a forward edge, the flow into v increases by δ; if it is a backward edge, the flow out of v decreases by δ. In either case the "in minus out" difference of v changes by $+\delta$. By a similar argument for the outgoing edge, the difference changes by $-\delta$. The total change is zero; conservation is not broken.

Value. Since the path is simple, it uses the source only once, on its first edge. This edge leaves the source. If it is a forward edge, the flow leaving the source increases by δ; if it is a backward edge, the flow entering the source decreases by δ. In both cases the net outflow increases by δ. □

This theorem is the foundation of all augmenting-path algorithms. With every augmentation the value of the flow strictly increases and the flow remains valid. Two questions remain. When no augmenting path is left, is the flow really maximum? How many augmentations are needed? The first question is answered with cuts. The answer to the second depends on how the path is chosen.

The update has two more important consequences. The residual capacity of the bottleneck edge is exactly δ; after the update this residual capacity drops to zero. So with every augmentation at least one edge on the path leaves the residual network. Also, for every edge on the path, δ units of residual capacity appear in the opposite direction. The residual network thus changes a little with each augmentation: some edges disappear and some backward edges appear.

8.3 Cuts

To prove that a flow is maximum, we must show that no larger flow exists. Trying all flows one by one is impossible. The cut concept instead gives an upper bound as a single number. The edges between the side containing the source and the side containing the sink are like a "border gate" that the flow must pass through. The width of the gate limits the flow that can pass.

8.3.1 The s–t cut and its capacity

Definition 8.5

An **s–t cut** *(S, T)* is a partition of the vertex set V into two disjoint sets with $s \in S$ and $t \in T$. The **capacity** of the cut is the sum of the capacities of the edges going from S to T:

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

A cut of smallest capacity is called a **minimum cut**.

In the definition a cut is a partition of the vertices, not a set of edges. The cut edges come out of this partition: the edges with one endpoint in S and the other in T. If these edges are deleted from the network, no directed path from the source to the sink remains. In computing the capacity, only the edges going from S to T are counted. Edges going from T to S are not added to the capacity, because they are not what prevents the flow from reaching the sink.

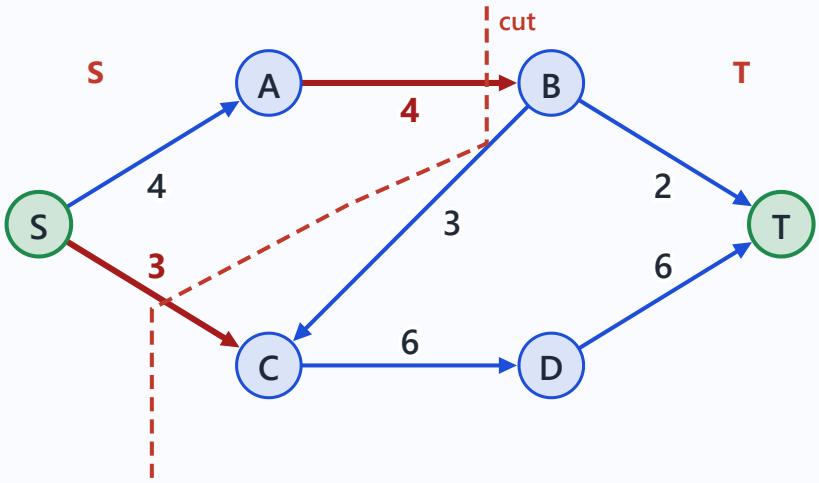


Figure 8.6: The cut $S = \{S, A\}$ and $T = \{B, C, D, T\}$. The edges crossing the dashed line from left to right are $A \to B$ (4) and $S \to C$ (3): $c(S, T) = 4 + 3 = 7$.

In the cut of Figure 8.6 the edge $B \to C$ also crosses the dashed line, but it goes from the T side to the T side; both of its endpoints are in T. For this reason it is not counted. Similarly, both endpoints of the edge $S \to A$ are on the S side.

Example 8.3

The example network has four intermediate vertices: A, B, C, D. Each can be placed on the S side or the T side, so there are $2^4 = 16$ s–t cuts in all. Let us compute a few.

- $S = \{S\}$: the cut edges are $S \to A$ and $S \to C$; $4 + 3 = 7$.

- $S = \{S, A, B\}$: the cut edges are $S \to C$ (3), $B \to C$ (3), $B \to T$ (2); total 8.
- $S = \{S, C\}$: $S \to A$ (4) and $C \to D$ (6); total 10. The edge $B \to C$ goes from T to S and is not counted.
- $S = \{S, B, D\}$: $S \to A$ (4), $S \to C$ (3), $B \to C$ (3), $B \to T$ (2), $D \to T$ (6); total 18. $A \to B$ and $C \to D$ go from T to S.

When all the cuts have been computed, the smallest capacity is 7, and two cuts attain this value: $\{S\}$ and $\{S, A\}$. The minimum cut need not be unique.

S side	Edges from S to T	$c(S, T)$
$\{S\}$	S→A 4, S→C 3	7
$\{S, A\}$	A→B 4, S→C 3	7
$\{S, A, B\}$	S→C 3, B→C 3, B→T 2	8
$\{S, A, B, C, D\}$	B→T 2, D→T 6	8
$\{S, C\}$	S→A 4, C→D 6	10
$\{S, B, D\}$	S→A 4, S→C 3, B→C 3, B→T 2, D→T 6	18

Table 8.1: Some s–t cuts in the example network. The smallest of the sixteen cuts is 7.

Common mistake

Taking the minimum cut to be the partition that cuts the fewest edges. A minimum cut minimizes the capacity *sum*, not the *number* of edges. Cutting a single edge of capacity 100 is more expensive than cutting three edges of capacity 1. The number of edges measures the same thing as the capacity only when all capacities are 1.

8.3.2 Net flow across a cut

The **net flow** across a cut is found by subtracting the total flow on the edges going from T to S from the total flow on the edges going from S to T:

$$f(S, T) = \Sigma_{u \in S, v \in T} f(u, v) - \Sigma_{u \in S, v \in T} f(v, u)$$

Unlike capacity, net flow also counts the edges in the backward direction, because flow that comes back must be subtracted from the flow that crosses over.

Theorem 8.2 (Cuts and flows)

For every flow f and every s–t cut (S, T), $f(S, T) = |f|$. Hence $|f| \le c(S, T)$.

Proof. For every vertex in S, write the "out minus in" difference and add up these differences. For the source the difference is $|f|$ by definition. The other vertices of S are intermediate vertices (the sink is in T); by conservation their differences are zero. So the total is $|f|$.

The same sum can also be computed edge by edge. An edge with both endpoints in S appears once in the outflow of one vertex and once in the inflow of another; their contributions cancel. Only the edges crossing the cut remain. Those going from S to T appear with a plus sign in the total, and those going from T to S with a minus sign. This is exactly $f(S, T)$. Therefore $f(S, T) = |f|$.

For the inequality, use $f(u, v) \le c(u, v)$ on forward edges and $f(v, u) \ge 0$ on backward edges: $|f| = f(S, T) \le \Sigma\, c(u, v) - 0 = c(S, T)$. □

What the theorem says agrees with intuition. All the flow leaving the source must cross the cut somewhere to reach the sink. If some flow comes back, it too has to cross forward at least one more time. In the end the net amount crossing the cut equals the amount sent into the network. Let us test the flow of Figure 8.2 on the cut $\{S, A\}$: the forward edges are $A \to B$ (4) and $S \to C$ (3), there is no backward edge; the net flow is 7. On the cut $\{S, C\}$ the forward edges are $S \to A$ (4) and $C \to D$ (5), and the backward edge is $B \to C$ (2); the net flow is $4 + 5 - 2 = 7$. The two cuts differ, but the net flow is the same.

The second half of the theorem says that every cut is an upper bound for every flow. The value of any flow cannot exceed the capacity of any cut. Hence the maximum flow cannot exceed the capacity of the minimum cut. This is an inequality. The real surprise is that the inequality is always an equality.

8.3.3 The max-flow min-cut theorem

Theorem 8.3 (Max-flow min-cut)

For a flow f the following three statements are equivalent (Ford and Fulkerson, 1956):

1. f is a maximum flow.
2. The residual network G_f has no augmenting path.
3. For some cut (S, T), $|f| = c(S, T)$.

Consequently, the value of the maximum flow equals the capacity of the minimum cut.

Proof. **(1) ⇒ (2).** If there were an augmenting path in the residual network, then by Theorem 8.1 the flow could be increased by the bottleneck. This contradicts the maximality of f.

(2) ⇒ (3). Let S be the set of vertices reachable from the source in the residual network, and $T = V - S$. The source is in S. Since there is no augmenting path, the sink is not in S; (S, T) is an s–t cut. For an edge (u, v) with $u \in S$ and $v \in T$ we must have $f(u, v) = c(u, v)$; otherwise $c_f(u, v) > 0$ and v would also be

reachable. For the same reason, on an edge (v, u) going from T to S we must have $f(v, u) = 0$; otherwise the backward edge $u \to v$ would be in the residual network. By Theorem 8.2, $|f| = f(S, T) = \Sigma\, c(u, v) - 0 = c(S, T)$.

(3) ⇒ (1). No flow can exceed the capacity of any cut. If $|f| = c(S, T)$, then no flow can be larger than $c(S, T)$, and therefore than $|f|$. □

The theorem has three separate uses. First, it proves the correctness of augmenting-path algorithms: it is safe to stop when no path is left. Second, it gives an easily checked *certificate* that a flow is maximum. A flow whose value equals the capacity of some cut is maximum without any other computation. Third, it shows how to find a minimum cut: the vertices reachable from the source in the final residual network form the S side.

The flow found in our example network has value 7, and the capacity of the cut $\{S\}$ is also 7. The third statement of the theorem holds; the flow is maximum. In the residual network built in Example 8.2 nowhere could be reached from the source. The reachable set is only $\{S\}$, and this is exactly the cut in the proof.

8.3.4 Finding the minimum cut

The proof of the theorem is also an algorithm. After a maximum flow has been found, a search is run from the source in the final residual network. The visited vertices form the S side. The original edges with one endpoint visited and the other not are the edges of the minimum cut. The following Java method does this. It uses the `residual` matrix holding the residual capacities and a copy of the original capacities.

```
List<int[]> minCut(int source, int[][] capacity) {
  boolean[] visited = new boolean[V];
  Queue<Integer> queue = new LinkedList<>();
  queue.add(source);
  visited[source] = true;
  while (!queue.isEmpty()) {          // BFS in the final residual network
    int u = queue.poll();
    for (int v = 0; v < V; v++)
      if (!visited[v] && residual[u][v] > 0) {
        visited[v] = true;
        queue.add(v);
      }
  }
  List<int[]> cut = new ArrayList<>();
  for (int u = 0; u < V; u++)          // original edges from S to T
    for (int v = 0; v < V; v++)
      if (visited[u] && !visited[v] && capacity[u][v] > 0)
        cut.add(new int[]{u, v});
  return cut;
}
```

The first half of the method is an ordinary BFS. The only difference is that the existence of an edge is tested with `residual[u][v] > 0`: the search moves only through the residual network. The second half scans all pairs of vertices and adds to the list those pairs with `visited[u]` true and `visited[v]` false that have an edge in the original network. The capacity condition matters. Backward edges occupy space in the residual matrix but do not exist in the original network; they cannot be cut edges.

With an adjacency matrix the BFS takes $O(V^2)$ and the pair scan takes $O(V^2)$. The total time is $O(V^2)$, which is negligible next to the cost of finding the maximum flow. The extra memory is $O(V + k)$ for the `visited` array, the queue, and the result list, where k is the number of edges in the cut. When the method is later run on the final residual network of the second Ford-Fulkerson example, it returns the edges $S \to 1$ (10) and $3 \to 4$ (9). The total, 19, equals the maximum flow in that network.

8.3.5 Overview of network flow algorithms

All the algorithms that find a maximum flow reach the same goal with the same concepts. They differ in the order and in the pieces in which they grow the flow. Table 8.2 summarizes the four algorithms treated in this chapter.

Algorithm	Approach	Time	Note
Ford-Fulkerson (1956)	any augmenting path (often DFS)	$O(E \cdot \lvert f^* \rvert)$	simple; the time depends on the flow value
Edmonds-Karp (1972)	shortest augmenting path (BFS)	$O(V \cdot E^2)$	the polynomial-time form of Ford-Fulkerson
Dinic (1970)	level graph + blocking flow	$O(V^2 \cdot E)$	faster than Edmonds-Karp; very efficient on unit capacities
Push–relabel (1988)	preflow + heights, local operations	$O(V^2 \cdot E)$	searches for no paths; the FIFO version runs in $O(V^3)$

Table 8.2: Network flow algorithms. $|f^*|$ is the value of the maximum flow; the running time of Ford-Fulkerson holds for integer capacities.

The first three algorithms rest on the idea of the augmenting path and maintain a valid flow at every step. The fourth is different: in its intermediate steps it temporarily violates conservation and repairs it only at the end. The years in the table also hold an interesting detail. Dinic's algorithm was published in 1970 in the Soviet Union, before Edmonds-Karp; the two works are independent.

8.4 The Ford-Fulkerson method

8.4.1 The idea of the method

The Ford-Fulkerson method rests on the work published by L. R. Ford Jr. and D. R. Fulkerson in 1956. It finds the maximum flow between two vertices in a directed graph with capacities. The idea is a natural consequence of the concepts built so far: the flow starts at zero, and as long as an augmenting

path is found in the residual network, the flow is increased along that path by the bottleneck. When no path is left, the loop stops; the max-flow min-cut theorem guarantees that the result is optimal.

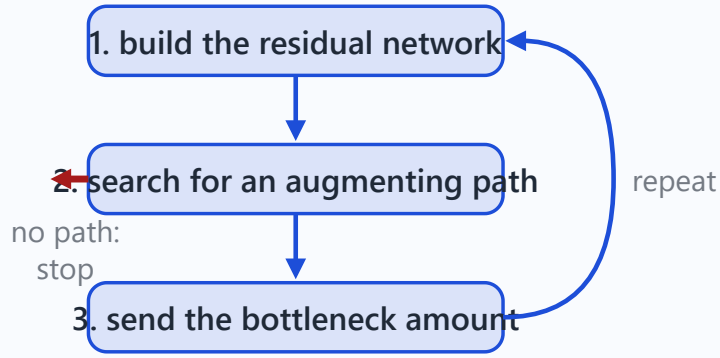


Figure 8.7: The Ford-Fulkerson loop: build the residual network, search for an augmenting path, send the bottleneck amount, and repeat. If no path is found, the loop ends.

Most sources call Ford-Fulkerson a **method** rather than an "algorithm." The reason is that it does not specify how the augmenting path is to be found. Any path is accepted. The version that searches for the path with DFS is usually known as Ford-Fulkerson; the version that uses BFS has a name of its own: Edmonds-Karp. That the choice of path affects the running time but not the result will be clear in the comparison of these two versions.

8.4.2 The steps of the algorithm

1. **Initialization.** $f(u, v) = 0$ on all edges. Since the flow is zero, the residual capacity of every edge is its own capacity; the residual network is identical to the original network.
2. **Path search.** An augmenting path p is searched for in the residual network from the source to the sink. Only edges with $c_f > 0$ are used.
3. **Bottleneck.** $\delta = \min c_f(u, v)$, $(u, v) \in p$ is computed. This is the most flow that can be sent along the path.
4. **Augmentation.** δ units of flow are sent along the path: on forward edges $f(u, v)$ += δ, on backward edges $f(v, u)$ −= δ; $|f|$ += δ. At least one edge on the path becomes saturated.
5. **Repeat.** Steps 2–4 are repeated until no augmenting path is left. When the loop ends, $|f|$ is maximum.

In an implementation the flow and the residual capacity need not be stored separately. Only the residual capacities can be kept. When flow is added, the residual capacity of the forward edge decreases by δ and that of the backward edge increases by δ. The flow on an edge is recomputed when needed as $f(u, v) = c(u, v) - c_f(u, v)$. The Java code on the slides follows this route.

8.4.3 Example 1: step by step

Let us run the method on the network of Figure 8.1. Initially all flows are zero and the residual network is the same as the original. The order of the paths in this example is the order that a search choosing the path with the fewest edges each time would give. Since Ford-Fulkerson accepts any path, this order is valid.

First augmentation. $p = S \to A \to B \to T$. The residual capacities are 4, 4, and 2; the bottleneck is 2 units and comes from $B \to T$. Two units are sent along the path: $|f| = 2$. $B \to T$ becomes saturated. In the residual network 2 units remain on each of $S \to A$ and $A \to B$, and backward edges of 2 units each appear in the opposite directions.

Second augmentation. $p = S \to C \to D \to T$. The residual capacities are 3, 6, and 6; the bottleneck is 3. $|f| = 2 + 3 = 5$. $S \to C$ becomes saturated. 3 units remain on each of $C \to D$ and $D \to T$.

Third augmentation. The only residual edge leaving S is $S \to A$ (2 units). From A one can go to B with 2 units. $B \to T$ is saturated, but $B \to C$ is empty (3 units). $C \to D$ and $D \to T$ have 3 units each. The path is $p = S \to A \to B \to C \to D \to T$, and the bottleneck is $\min(2, 2, 3, 3, 3) = 2$. $|f| = 7$.

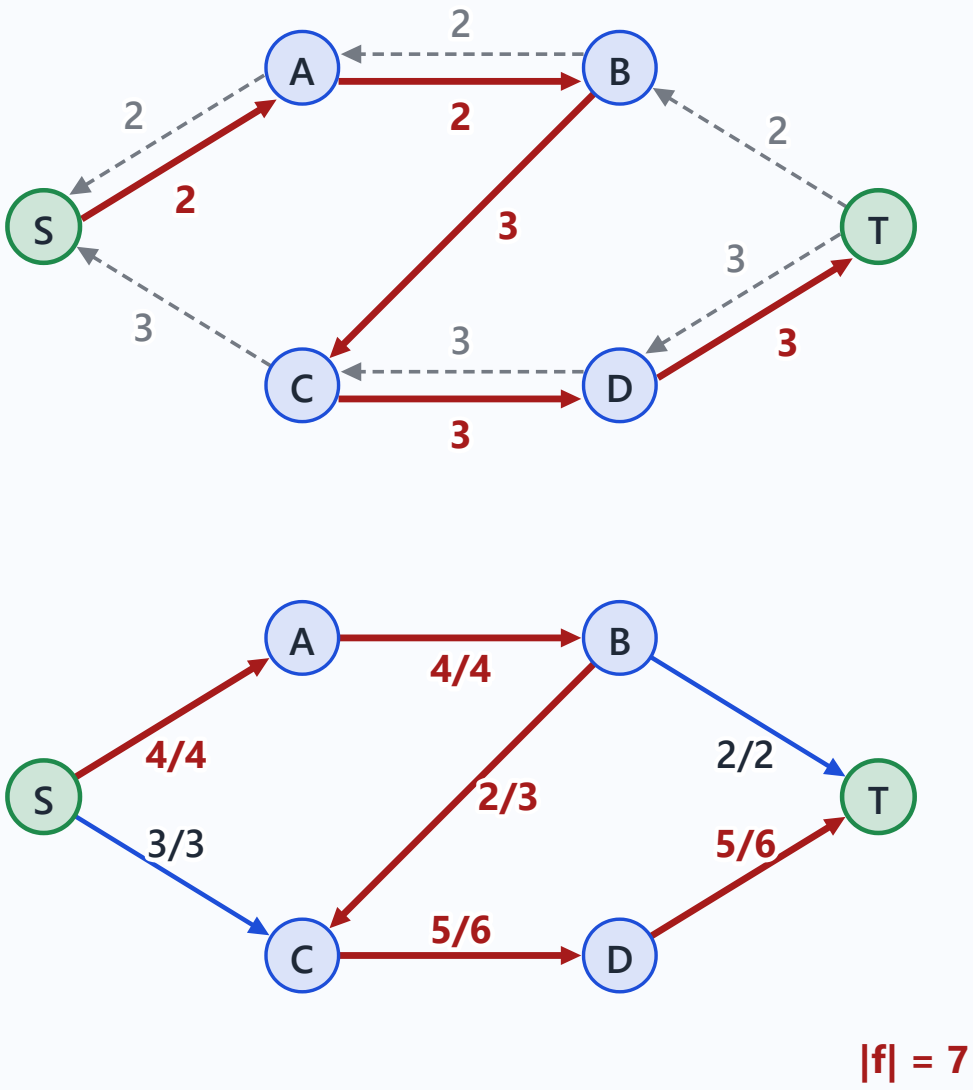


Figure 8.8: The third augmentation. (a) The path $S \to A \to B \to C \to D \to T$ in the residual network (highlighted); the dashed arrows are the backward edges of the previous two augmentations. (b) The flow after 2 units have been added along the path; $|f| = 7$.

After the third augmentation both edges leaving S are saturated. Nowhere can be reached from the source in the residual network; the loop ends.

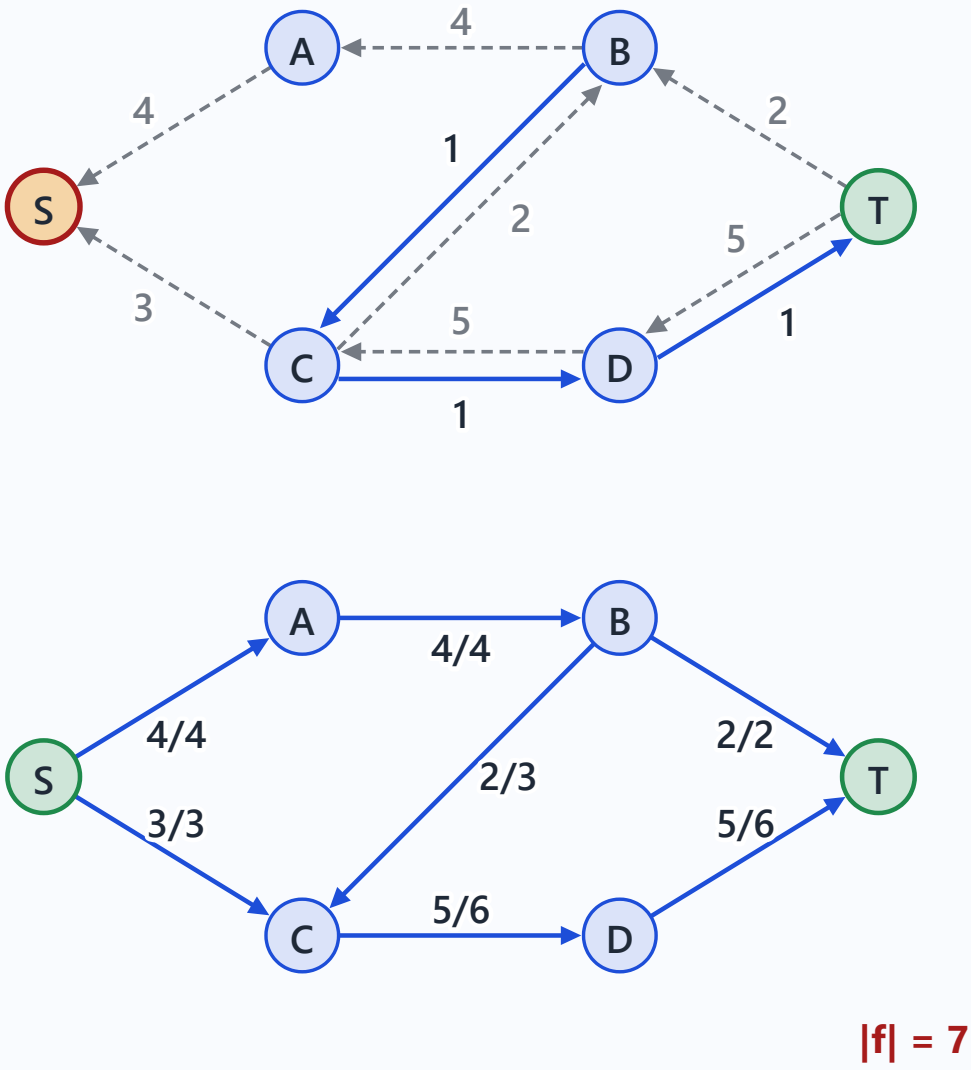


Figure 8.9: (a) The final residual network: no residual edge leaves S; only the backward edges $A \to S$ and $C \to S$ remain. (b) The maximum flow: $|f| = 7$. The minimum cut is $(\{S\}, \text{the rest})$, with capacity $4 + 3 = 7$.

Augmentation	Path	Residual capacities	Bottleneck	$\|f\|$	Saturated edge
1	S→A→B→T	4, 4, 2	2	2	B→T
2	S→C→D→T	3, 6, 6	3	5	S→C
3	S→A→B→C→D→T	2, 2, 3, 3, 3	2	7	S→A, A→B

Table 8.3: The augmentations in Example 1.

The final flow is the same as the flow in Figure 8.2: 2 units on $B \to C$ and 5 units on $C \to D$. This is a coincidence. The value of the maximum flow is unique, but its distribution over the edges need not be. In the same network, a flow carrying 1 unit on $B \to T$, 3 on $B \to C$, and 6 each on $C \to D$ and $D \to T$ is also valid, and its value is again 7. The only thing that does not change is that the two edges leaving the source are saturated; the minimum cut $\{S\}$ forces this.

8.4.4 Example 2: using a backward edge

In the first example no augmenting path used a backward edge. To see why backward edges are necessary, a somewhat larger network is needed. In the network of Figure 8.10 the source is S and the sink is T; the intermediate vertices are numbered 1 to 4.

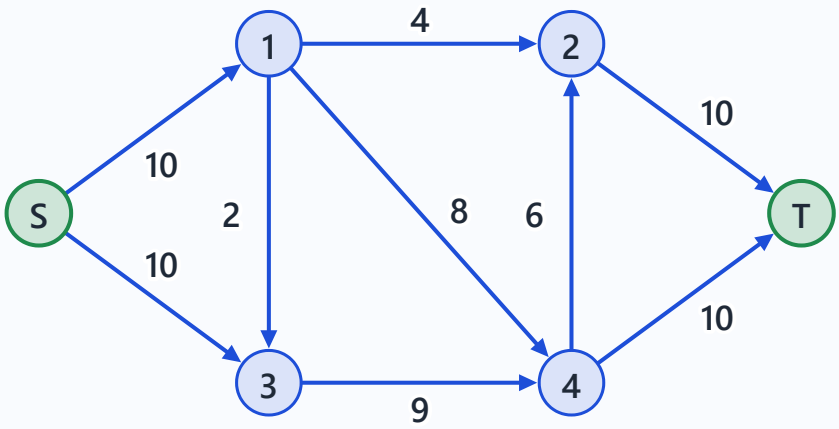


Figure 8.10: The second example network. Capacities: $S \to 1$ 10, $S \to 3$ 10, $1 \to 2$ 4, $1 \to 3$ 2, $1 \to 4$ 8, $3 \to 4$ 9, $4 \to 2$ 6, $2 \to T$ 10, $4 \to T$ 10.

This time the order of the paths is deliberately bad: the first path makes a decision that must be corrected later.

1. $p = S \to 1 \to 4 \to T$: residual capacities 10, 8, 10; the bottleneck is $1 \to 4$ (8). $|f| = 8$.
2. $p = S \to 3 \to 4 \to T$: only $10 - 8 = 2$ units are left on $4 \to T$. The bottleneck is 2; $|f| = 10$. $4 \to T$ becomes saturated.
3. $p = S \to 3 \to 4 \to 1 \to 2 \to T$: $3 \to 4$ has $9 - 2 = 7$, the backward edge $4 \to 1$ has 8 (the flow sent to $1 \to 4$ in the first step), $1 \to 2$ has 4, and $2 \to T$ has 10 units. The bottleneck is 4; $|f| = 14$.
4. $p = S \to 1 \to 4 \to 2 \to T$: 2 units were left on $S \to 1$. On $1 \to 4$ there is now room for $8 - 4 = 4$ units. The bottleneck is 2; $|f| = 16$.
5. $p = S \to 3 \to 4 \to 2 \to T$: the bottleneck is $3 \to 4$ ($9 - 6 = 3$ left). $|f| = 19$.

The third step is the heart of the example. When the backward edge $4 \to 1$ is used, the flow on $1 \to 4$ drops from 8 to 4. What this means is the following: the 4 units that previously went from 1 to 4 now arrive at 4 from 3. The 4 units freed up at 1 head for the sink along $1 \to 2$. Conservation is not broken at any vertex. The total flow into 4 and the total flow out of 1 do not change; only their routes change.

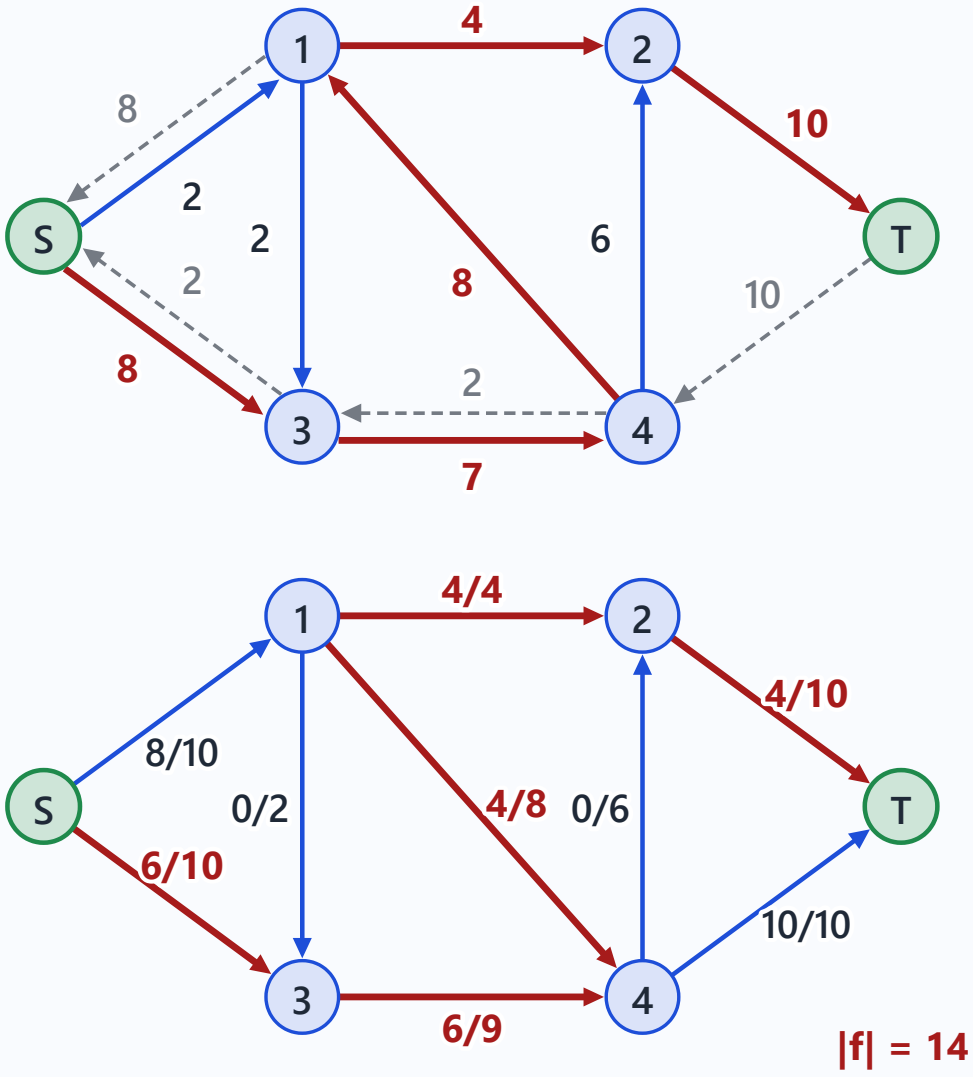


Figure 8.11: The third augmentation. (a) The path $S \to 3 \to 4 \to 1 \to 2 \to T$ in the residual network; $4 \to 1$ is a backward edge. (b) The flow afterward: the flow on $1 \to 4$ fell from 8 to 4, $1 \to 2$ filled up with 4 units; $|f| = 14$.

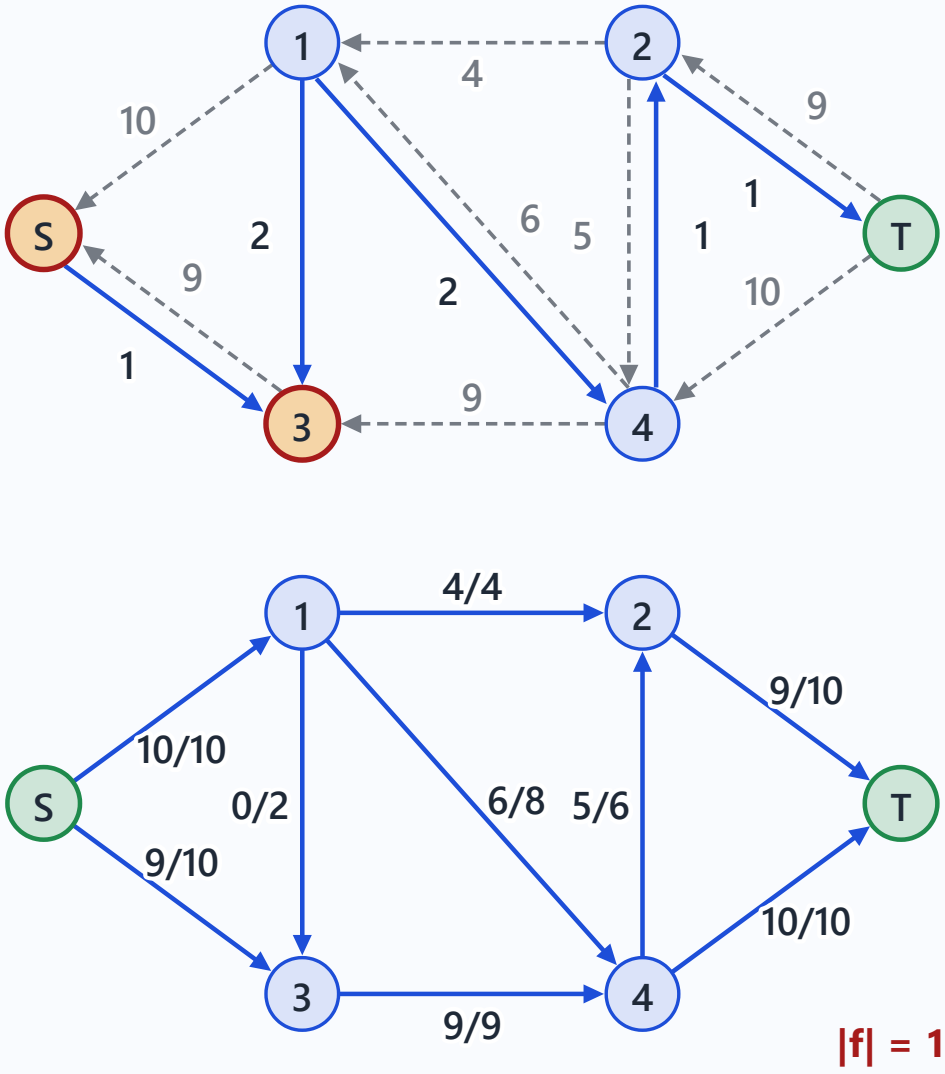


Figure 8.12: (a) The final residual network: from *S* only 3 can be reached (highlighted vertices). (b) The maximum flow, 19. The minimum cut *({S, 3}, the rest)*: *S→1* (10) + *3→4* (9) = 19.

In the final residual network $S \to 1$ is saturated; 1 unit is left on $S \to 3$, but the only edge leaving 3, $3 \to 4$, is saturated. The set reachable from the source is $\{S, 3\}$. The original edges leaving this set are $S \to 1$ and $3 \to 4$; both are full. The edge $1 \to 3$ goes from the T side to the S side and carries no flow. This is a perfect example of the conditions in the proof: forward edges full, backward edges empty.

8.4.5 Why are backward edges necessary?

Consider a greedy method that never uses backward edges in the same network. This method searches only for paths through forward edges that are not full.

Example 8.4

Let the first choice again be $S \to 1 \to 4 \to T$, and send 8 units. Then:

- $S \to 3 \to 4 \to T$: 2 units ($4 \to T$ becomes saturated). Total 10.
- $S \to 1 \to 2 \to T$: 2 units, since 2 units are left on $S \to 1$. Total 12.
- $S \to 3 \to 4 \to 2 \to T$: there are 7 units on $3 \to 4$, 6 on $4 \to 2$, and 8 on $2 \to T$; 6 units. Total 18.

Now $S \to 1$ is full. From S one can go only to 3, and from there to 4. Both exits of 4 ($4 \to 2$ and $4 \to T$) are full. The method gets stuck and stops at 18. Yet the maximum flow is 19.

The loss is a single unit, but the principle matters. The first path loaded 8 units onto the edge $1 \to 4$. Part of this flow should actually have gone through $1 \to 2$, which would have freed room on the edge $3 \to 4$. A method that works with forward edges only cannot undo this decision. In the residual network with backward edges, however, the edge $4 \to 1$ makes it possible to undo part of this decision: the flow coming from 1 is redirected to 2, and the flow coming from 3 to 4. The result is 19, equal to the minimum cut. There is a general rule here: in the residual network with backward edges, the method reaches the maximum whatever order the paths are chosen in.

Common mistake

Forgetting backward edges. Searching with forward edges only is a greedy algorithm and may miss the maximum (18 < 19 in the example). In code the backward edges are often a single line: `residual[v][u] += flow`. If this line is deleted, the program still runs and returns a number; the problem is noticed only when the result is verified.

8.4.6 Pseudocode

The pseudocode of the method is below. Here the flow values are kept in a separate `flow` table, and the residual capacity is computed when needed as `capacity - flow`.

```
FORD_FULKERSON(G, source, sink):
  flow[u, v] = 0, for each (u, v) in E     // 0 at the start
  maxFlow = 0
  while an augmenting path exists:
    path = DFS(G, source, sink, flow)           // edges with c − flow > 0
    delta = ∞                                   // smallest residual capacity on the path
    v = sink
    while v ≠ source:
      u = path.parent[v]
      delta = min(delta, capacity[u, v] − flow[u, v])
      v = u
    v = sink                                    // update the flows on the path
    while v ≠ source:
      u = path.parent[v]
      flow[u, v] += delta
      flow[v, u] −= delta                       // backward edge
      v = u
    maxFlow += delta
  return maxFlow, flow
```

The outer loop runs as long as an augmenting path is found. The `DFS` call searches for a path through edges of positive residual capacity and reports the path through a parent array (`parent[]` in the code) that stores the predecessor of each vertex. The path itself is not stored as a list; it is read back from the sink to the source by following `parent`.

The first inner loop finds the bottleneck. `v` starts at the sink; at each step `u = path.parent[v]`, the vertex from which `v` was reached. `delta` is limited by the smallest residual capacity seen so far. The loop ends when it reaches the source. The second inner loop walks the same path once more and updates the flow: `delta` is added in the forward direction and subtracted in the backward direction. The subtraction in the backward direction follows from keeping the flow in a skew-symmetric table: `flow[v, u] = −flow[u, v]`. In this representation the residual capacity of the backward edge is `capacity[v, u] − flow[v, u] = 0 − (−flow[u, v]) = flow[u, v]`, the same value as in the definition.

This skew-symmetric representation simplifies the code. There is no need for a separate “is it a backward edge?” question; forward and backward edges are handled by the same formula. There is one subtlety: if there are original edges in both directions between two vertices (for example both *1 → 2* and *2 → 1*), the flows of the two edges can cancel in the same cell. In the matrix representation this causes no problem, because only the net flow matters.

8.4.7 Java: an augmenting path with DFS

The Java implementation on the slides keeps the residual capacities in the matrix `residual[u][v]`. At the start this matrix is filled with the capacities; the residual capacity of backward edges is zero. The DFS that finds the augmenting path is as follows:

```
boolean dfs(int u, int sink,
            boolean[] visited, int[] parent) {
  visited[u] = true;
  if (u == sink) return true;
  for (int v = 0; v < V; v++) {
    if (!visited[v] && residual[u][v] > 0) {
      parent[v] = u;
      if (dfs(v, sink, visited, parent))
        return true;
    }
  }
  return false;   // no path to the sink from this branch
}
```

The method takes four parameters: the current vertex `u`, the sink, the visited marks, and the `parent` array that records the path. The first line marks `u` as visited. This way the same vertex is not entered again and the search does not fall into a cycle. If `u` is the sink, a path has been found and `true` is returned.

The loop tries all vertices in order; in an adjacency matrix, "neighbor" means a `v` with `residual[u][v] > 0`. When an unvisited neighbor with positive residual capacity is found, `parent[v] = u` is written and the search continues from `v`. If the recursive call returns `true`, this `true` is passed up at once; the rest of the loop is not tried. If all neighbors are tried and the sink cannot be reached, the method returns `false`: there is no path to the sink from this branch.

The visited marks are not cleared on return. This is important. If there is no path to the sink from a vertex, reaching that vertex by another route does not change the outcome; trying it again is wasted work. This way each vertex is visited at most once. With an adjacency matrix each visit scans V cells; one DFS takes $O(V^2)$. With an adjacency list the same search takes $O(V + E)$.

When flow is added, `residual[u][v] -= d` and `residual[v][u] += d` are performed. The backward edge is not created separately; the second line brings it into existence by itself. On the next call the DFS sees this cell as positive and uses the backward edge like an ordinary edge. This assignment is the only place where the code "knows" about backward edges.

8.4.8 Code trace: a dead end and a backward edge

Let us run the code on the network of Figure 8.1. Let the vertex order be S, A, B, C, D, T (indices 0 to 5); the DFS tries neighbors in this order. This order produces different paths from those in Example 1. The first DFS finds the path $S \to A \to B \to C \to D \to T$ (bottleneck 3). In the second DFS, $B \to C$ is saturated and the path $S \to A \to B \to T$ carries 1 unit. After two augmentations $|f| = 4$. Let us trace the third search in detail.

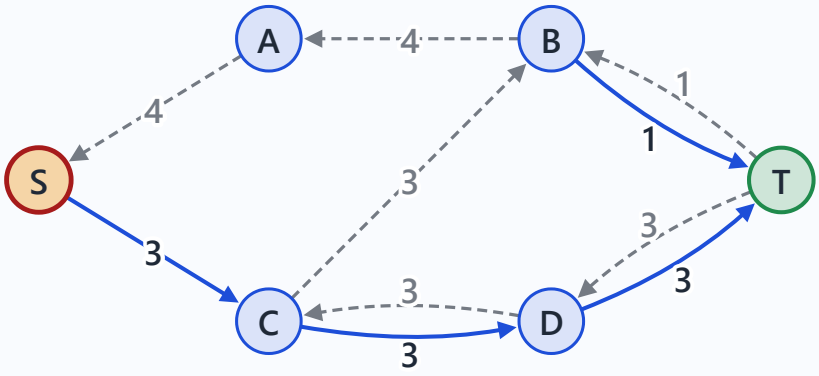


Figure 8.13: The residual network after two augmentations ($|f| = 4$). The dashed arrows are backward edges. $S \to A$ and $A \to B$ are saturated; the backward edge $C \to B$ has 3 units and the backward edge $B \to A$ has 4 units.

Step	Call	Event	`parent`
1	`dfs(S)`	$S \to A$: residual 0, skipped; $S \to B$: no edge. $S \to C$: residual 3 > 0, C not yet visited.	`parent[C] = S`
2	`dfs(C)`	$C \to S$: S visited; $C \to A$: 0. $C \to B$: residual 3 > 0 (backward edge).	`parent[B] = C`
3	`dfs(B)`	$B \to S$: 0. $B \to A$: residual 4 > 0 (backward edge).	`parent[A] = B`
4	`dfs(A)`	$A \to S$: residual 4, but S is visited. The others are 0. Dead end: `false`.	—
5	`dfs(B)` continues	$B \to C$: C visited; $B \to D$: no edge. $B \to T$: residual 1 > 0.	`parent[T] = B`
6	`dfs(T)`	$u = T$ is the sink: `true`. Every call passes `true` upward.	—

Table 8.4: Trace of the calls `dfs(S, T, visited, parent)` for the third augmenting path.

The dead end in the fourth step is characteristic behavior of DFS. A was reached by a backward edge, but there is no residual path from A to the sink. The only positive residual edge of A leads back to the source, and the source has already been visited. The search leaves A and continues from where it left off at B. The record `parent[A] = B` is not erased, but it is harmless: when the path is read back from the sink, A is never visited.

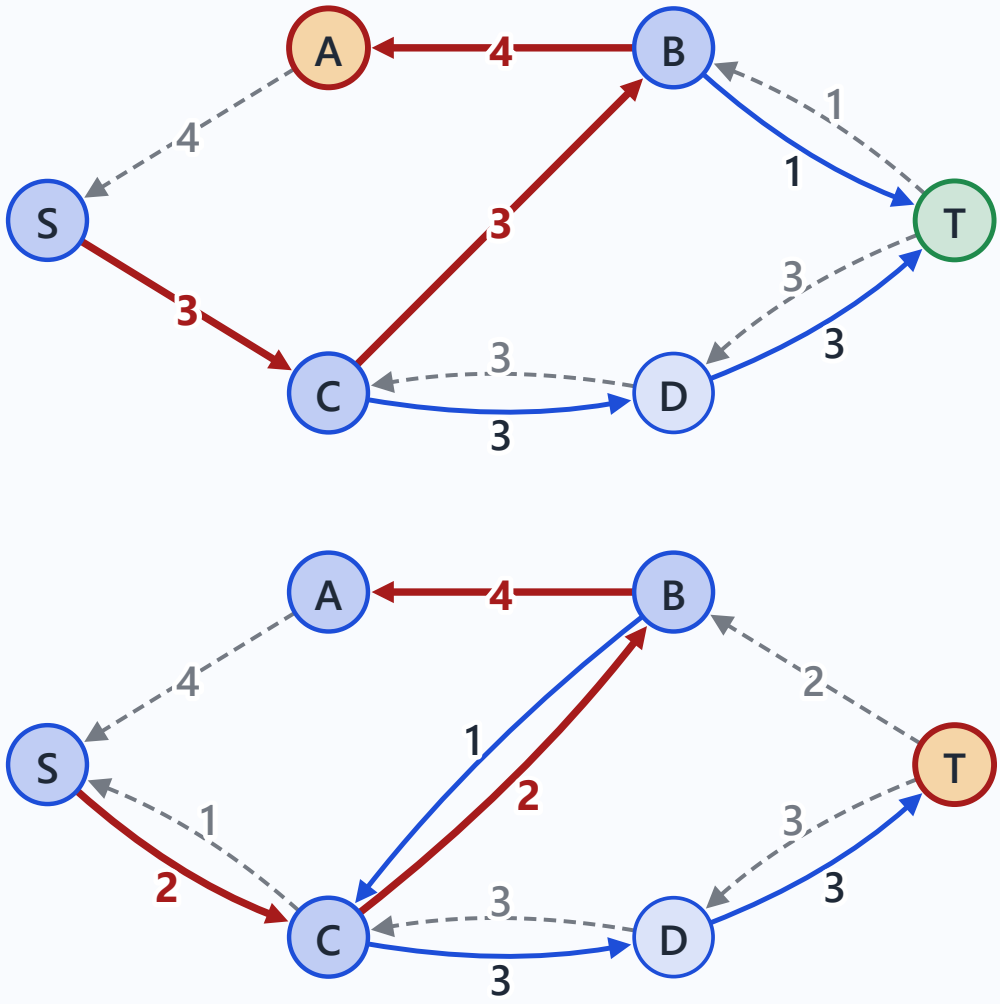


Figure 8.14: (a) A dead end at A: the search descended along $S \to C \to B \to A$ and cannot proceed from A. (b) The residual network after 1 unit has been added along the path found, $S \to C \to B \to T$; $|f| = 5$.

The path is read back from the sink: `parent[T] = B`, `parent[B] = C`, `parent[C] = S`. The path is $S \to C \to B \to T$. The bottleneck is $min(3, 3, 1) = 1$. This augmentation uses the backward edge $C \to B$: the flow on $B \to C$ drops from 3 to 2. What this means is that one unit of the flow going from B to C now goes directly to the sink along $B \to T$; the new unit from the source makes up the lost input of C.

The fourth DFS finds the path $S \to C \to D \to T$ and adds 2 units. $|f| = 7$. The fifth DFS cannot go anywhere from the source and returns `false`. This version makes 4 augmentations, whereas the order in Example 1 made 3. The final flow of the two versions is the same: the flow of Figure 8.2.

8.4.9 Complexity and integrality

The running time of Ford-Fulkerson depends on the structure of the network and on the capacities. Each round has two parts. Finding an augmenting path (DFS or BFS) and updating the flow take $O(E)$; an adjacency list is assumed. The real question is the number of rounds.

Theorem 8.4 (Integrality)

If all capacities are integers, then at every step of the Ford-Fulkerson method the flow and the residual capacities are integers. The method makes at most $|f^*|$ augmentations and finishes in $O(E \cdot |f^*|)$ time. The maximum flow found has an integer value.

Proof. By induction: at the start the flow is zero and the residual capacities equal the capacities; all are integers. If the residual capacities are integers, the bottleneck is also an integer and is at least 1. The update adds integers to or subtracts them from integers, so integrality is preserved. Every augmentation raises the value of the flow by at least 1, and the value cannot exceed $|f^*|$. So at most $|f^*|$ augmentations are made, each taking $O(E)$. □

The integrality property has an important application. In problems such as matching, each edge is either used or not; half an edge is meaningless. If the capacities are set to 1, the theorem guarantees that the maximum flow is 0 or 1 on every edge. The flow can then be read directly as a selection.

The bound has a flaw: $O(E \cdot |f^*|)$ is not polynomial. The running time of an algorithm is measured by the *length* of the input, that is, the number of bits needed to write it. A capacity of 10^9 is written with about 30 bits, but $|f^*|$ can be as large as the number itself. Bounds of this kind are called **pseudo-polynomial**. If the capacities are small integers there is no problem. With large capacities, however, the method can slow down badly if it chooses paths poorly.

If capacities may be irrational numbers, the situation is even worse. In known examples, with badly chosen paths the augmentations become smaller and smaller and the method never terminates; the flow value does not even converge to the maximum flow. Since computers store capacities with finite precision, this does not occur in practice. But it should be kept in mind as the price of the method's "any path" freedom.

8.4.10 A bad choice of paths

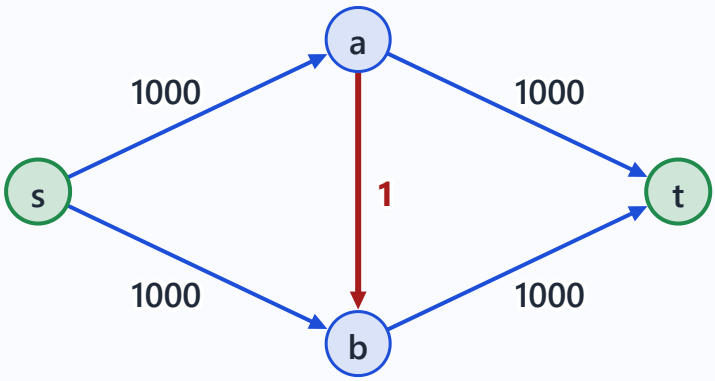


Figure 8.15: The worst-case network. The four outer edges have capacity 1000, and the middle edge $a \to b$ has

capacity 1. The maximum flow is 2000.

In the network of Figure 8.15 the maximum flow is 2000: each of the paths $s \to a \to t$ and $s \to b \to t$ carries 1000 units. Two augmentations suffice. Now consider a bad choice of paths. Let the first path be $s \to a \to b \to t$. Because of the middle edge the bottleneck is 1. This augmentation creates the backward edge $b \to a$. Let the second path be $s \to b \to a \to t$; because of the backward edge the bottleneck is again 1, and $a \to b$ becomes empty again. If these two paths continue to be chosen in turn, every augmentation adds only 1 unit. Reaching the maximum takes 2000 augmentations.

If the capacities were 10^9, the same choice would require $2 \cdot 10^9$ augmentations. Yet the network has only four vertices and five edges. DFS does not always make this bad choice. For example, a DFS that tries neighbors in index order finishes in 4 augmentations on this network: first $s \to a \to b \to t$ (1), then $s \to a \to t$ (999), $s \to b \to a \to t$ (1), and $s \to b \to t$ (999). But no rule prevents DFS from making the bad choice. For this reason the time bound remains $O(E \cdot |f^*|)$.

Edmonds-Karp, the version that works with BFS, chooses shortest paths. In this network the shortest paths have two edges: $s \to a \to t$ and $s \to b \to t$. Each carries 1000 units; 2 augmentations suffice. The three-edge path through the middle is never chosen. How BFS removes the dependence on the size of the capacities is the subject of the next section.

Common mistake

Thinking that Ford-Fulkerson runs in polynomial time. $O(E \cdot |f^*|)$ depends on the values of the capacities; in a network of the same structure, the time can grow as the capacities grow. Edmonds-Karp and Dinic remove this dependence.

8.5 The Edmonds-Karp algorithm

8.5.1 The idea: the shortest augmenting path

The Edmonds-Karp algorithm is a variant of the Ford-Fulkerson method. Jack Edmonds and Richard Karp published it in 1972; Dinic reached the same idea independently in 1970. The algorithm fixes the method's one free decision: the augmenting path is found by breadth-first search. Because BFS advances from the source layer by layer, the first path by which it reaches the sink is the one with the fewest edges. In every round Edmonds-Karp chooses this shortest augmenting path.

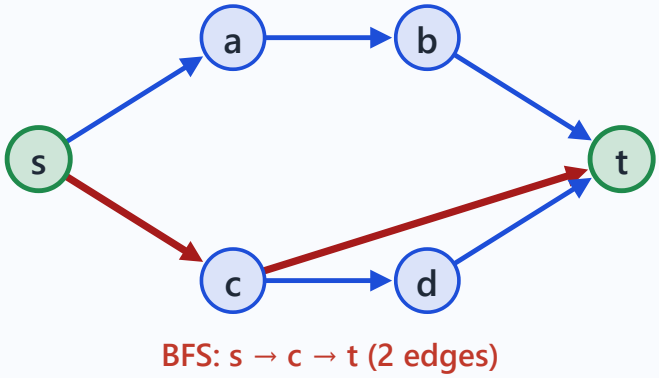


Figure 8.16: BFS chooses the augmenting path with the fewest edges. Here $s \to c \to t$ has two edges; the three-edge paths ($s \to a \to b \to t$, $s \to c \to d \to t$) are not chosen in that round.

Here "short" means the number of edges, not capacity or weight. The capacities of the edges do not affect the length of a path. This choice may look arbitrary at first. But the root of the trouble in the bad-path example was that a long, narrow path was chosen again and again. Choosing the shortest paths prevents such vicious circles. The steps of the algorithm are as follows:

1. Use BFS to find the shortest augmenting path from the source to the sink in the residual network.
2. Send the bottleneck amount of flow along the path and update the backward edges.
3. Repeat these two steps until BFS cannot reach the sink.

It is useful to note from the start that BFS has two different uses in network flow. Edmonds-Karp uses BFS directly to find an augmenting path, and after each BFS it processes a single path. Dinic's algorithm, by contrast, uses BFS to tell whether the sink is still reachable and to build a level graph; in that graph it processes many paths at once.

8.5.2 Complexity: O(V · E²)

Each round consists of one BFS and one path update: *O(E)*. The real contribution of Edmonds-Karp is that the number of augmentations has a bound independent of the capacities. The justification of this bound rests on two observations.

Let *d(v)* denote the number of edges on the shortest path from the source to a vertex *v* in the residual network. **First observation:** as Edmonds-Karp runs, the value *d(v)* of no vertex decreases. An augmentation removes some edges on the path from the residual network and adds some backward edges. Each added backward edge is the reverse of an edge *(u, v)* on a shortest path; that is, it goes from *v* to *u* in a pair with *d(v) = d(u) + 1*. Such an edge moves a vertex one step farther away and cannot shorten any path.

Second observation: The edge that determines the bottleneck in an augmentation is called a **critical edge**; it leaves the residual network after the augmentation. In every augmentation at least one edge is critical. When an edge *(u, v)* is critical, *d(v) = d(u) + 1*. For the same edge to return to the residual

network, an augmenting path must use its backward edge (v, u). At that moment $d'(u) = d'(v) + 1$. Since distances do not decrease, $d'(u) = d'(v) + 1 \geq d(v) + 1 = d(u) + 2$. So the distance of u grows by at least 2 between two times the edge is critical.

Theorem 8.5 (Edmonds-Karp)

The Edmonds-Karp algorithm makes at most $O(V \cdot E)$ augmentations and finishes in $O(V \cdot E^2)$ time. The time does not depend on the values of the capacities.

Proof. Each time an edge becomes critical, the distance of the vertex at its tail grows by at least 2. The distance starts at 0 and can be at most $V - 1$ for a reachable vertex; if the vertex becomes unreachable, the edge is never used again. So an edge becomes critical at most about $V/2$ times. The residual network has at most $2E$ edges. Since at least one edge is critical in every augmentation, the number of augmentations is $O(V \cdot E)$. Since each augmentation takes $O(E)$, the total time is $O(V \cdot E) \cdot O(E) = O(V \cdot E^2)$. □

The meaning of the bound is clear in the worst-case network. Whether the capacities are 1000 or 10^9, Edmonds-Karp finishes in 2 augmentations. The time no longer depends on $|f^*|$; the algorithm runs in polynomial time. The price is a full BFS in every augmentation. The bound is pessimistic: in practice the number of augmentations stays far below $V \cdot E$. In networks with $V = 6$ and $E = 9$ the bound allows about 27 augmentations; in the examples of these notes the number of augmentations is between 3 and 4.

8.5.3 Java: an augmenting path with BFS

The only difference between Edmonds-Karp and Ford-Fulkerson is the function that searches for the path; the main loop is the same. The BFS version is as follows:

```
boolean bfs(int source, int sink, int[] parent) {
  boolean[] visited = new boolean[V];
  Queue<Integer> queue = new LinkedList<>();
  queue.add(source);
  visited[source] = true;
  parent[source] = -1;
  while (!queue.isEmpty()) {
    int u = queue.poll();
    for (int v = 0; v < V; v++)
      if (!visited[v] && residual[u][v] > 0) {
        queue.add(v);
        parent[v] = u;
        visited[v] = true;
      }
  }
  return visited[sink];
}
```

In every call the method creates a new `visited` array. The queue starts with the source; the source counts as visited and `parent[source] = -1` is written. This −1 is a marker that tells, when the path is read back, that the source has been reached. In the loop the vertex `u` at the head of the queue is removed. Then all `v` are tried in order. Every `v` that is unvisited and has `residual[u][v] > 0` is added to the queue, has its parent recorded as `u`, and is immediately marked visited.

It matters that the mark is set when the vertex is added to the queue. If the vertex were marked when it leaves the queue, the same vertex could enter the queue from several different parents. The `parent` value would also change with the last parent to add it, and the BFS tree would be corrupted. Marking early ensures that every vertex enters the queue exactly once and through its parent closest to the source.

The method returns `visited[sink]`: if the sink was reached, an augmenting path exists. The `parent` array holds the BFS tree; following it from the sink to the source gives the path with the fewest edges. This code does not stop when it reaches the sink but continues until the queue is empty. The result does not change, because the sink's parent was written when it was first reached; stopping early only saves a little time. With an adjacency matrix one BFS takes $O(V^2)$, with an adjacency list $O(V + E)$.

The main loop is as follows:

```
int edmondsKarp(int s, int t) {
  int maxFlow = 0;
  int[] parent = new int[V];
  while (bfs(s, t, parent)) {
    int flow = Integer.MAX_VALUE;
    for (int v = t; v != s; v = parent[v])
      flow = Math.min(flow, residual[parent[v]][v]);
    for (int v = t; v != s; v = parent[v]) {
      int u = parent[v];
      residual[u][v] -= flow;  // forward
      residual[v][u] += flow;  // backward
    }
    maxFlow += flow;
  }
  return maxFlow;
}
```

The condition `while (bfs(s, t, parent))` runs a new BFS in every round. The first `for` loop walks from the sink toward the source. `parent[v]` is the previous vertex on the path; `residual[parent[v]][v]` is the residual capacity of that edge. `flow` falls to the smallest of these values. Its initial value is `Integer.MAX_VALUE`, so the first comparison always takes the value of a real edge. The second loop walks the same path once more: `flow` is subtracted from the forward residual capacity and added to the backward residual capacity. At the end of the round `maxFlow` grows. When BFS cannot reach the sink, the loop ends and the result is returned.

The memory use is $O(V^2)$ for the `residual` matrix and $O(V)$ for `parent`, `visited`, and the queue. A matrix is wasteful for sparse networks. For large sparse networks an adjacency list is used, with each edge holding a link to its backward edge; the Java code of Dinic's algorithm uses this representation.

8.5.4 Code trace: bfs(s, t, parent)

Let us trace BFS on the network of Figure 8.17. This network is the classic example from the book by Cormen and colleagues. The vertices *s, 1, 2, 3, 4, t* are numbered 0 to 5 in this order; the loop tries neighbors in this order. Initially all flows are zero and the residual capacity equals the capacity.

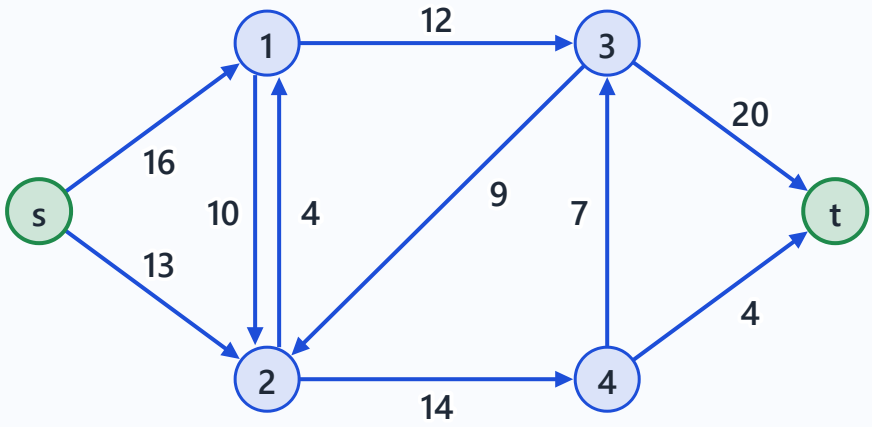


Figure 8.17: The trace network. Capacities: *s→1* 16, *s→2* 13, *1→2* 10, *2→1* 4, *1→3* 12, *3→2* 9, *2→4* 14, *4→3* 7, *3→t* 20, *4→t* 4.

Extracted `u`	Neighbors added (residual)	Skipped	Queue
—	start: `visited[s] = true`, `parent[s] = -1`	—	[s]
s	1 (16), 2 (13); parent s	—	[1, 2]
1	3 (12); parent 1	2 visited	[2, 3]
2	4 (14); parent 2	1 visited	[3, 4]
3	t (20); parent 3	2 visited	[4, t]
4	—	3 and t visited	[t]
t	—	no outgoing residual edge with positive capacity	[ ]

Table 8.5: The first call `bfs(s, t, parent)`. Each row is the removal of one vertex from the queue.

The sink is visited when 3 leaves the queue. The code nevertheless continues until the queue is empty. When the loop ends, the `parent` array is *[−1, s, s, 1, 2, 3]*. Read back from the sink, the path is *s → 1 → 3 → t*: only 3 edges. There is no s–t path shorter than 3 in this network; BFS has found the augmenting path with the fewest edges. The bottleneck is *min(16, 12, 20) = 12*.

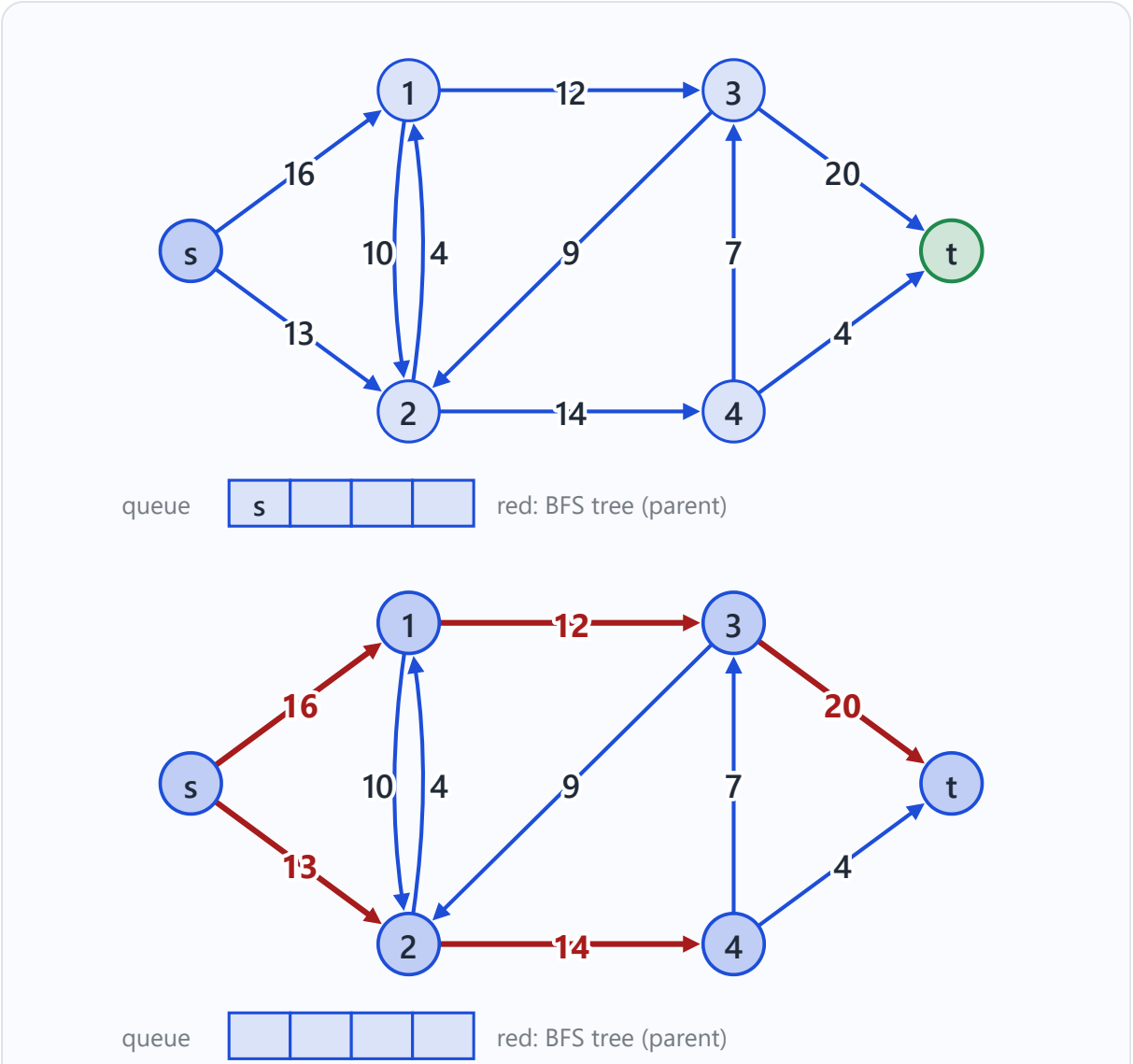


Figure 8.18: The beginning and the end of the BFS. (a) Only s is in the queue. (b) The queue is empty; the red edges are the BFS tree (the `parent` relation). The branch leading to the sink is $s \to 1 \to 3 \to t$.

8.5.5 Code trace: edmondsKarp(s, t)

The whole main loop can be seen in Table 8.6. For each round the path found, the bottleneck, and the updated residual capacities are given.

Round	BFS path	Residual capacities	`flow`	Updated cells	`maxFlow`
1	s→1→3→t (3 edges)	16, 12, 20	12	s→1: 4, 1→3: 0, 3→t: 8; backward: 1→s 12, 3→1 12, t→3 12	12
2	s→2→4→t (3 edges)	13, 14, 4	4	s→2: 9, 2→4: 10, 4→t: 0; backward: 2→s 4, 4→2 4, t→4 4	16
3	s→2→4→3→t (4 edges)	9, 10, 7, 8	7	s→2: 2, 2→4: 3, 4→3: 0, 3→t: 1; backward edges grow	23

Round	BFS path	Residual capacities	`flow`	Updated cells	`maxFlow`
4	none	—	—	BFS cannot reach t	23

Table 8.6: Trace of `edmondsKarp(s, t)`.

The path lengths are 3, 3, and 4; they do not decrease. This is a small example of the first observation in the complexity proof of Edmonds-Karp. After the second round $4 \to t$ is saturated. In the third round one cannot go from 4 directly to the sink; BFS finds the path that goes from 4 to 3 and is one edge longer. The bottleneck is 7, the capacity of $4 \to 3$.

In the fourth BFS one can go from the source to 1 (residual 4) and to 2 (residual 2). The edge from 1 to 3 is saturated. From 2 one can send 3 units to 4, but both exits of 4 ($4 \to 3$ and $4 \to t$) are full. Neither 3 nor the sink can be reached; the reachable set is $\{s, 1, 2, 4\}$. The method returns 23. The algorithm has made 3 augmentations and 4 BFS runs.

8.5.6 Example: the final flow and the minimum cut

Let us use cuts to verify why the flow stops at 23 in the same network. Even before the algorithm starts there is a simple upper bound. The capacity leaving the source is $16 + 13 = 29$, and the capacity entering the sink is $20 + 4 = 24$. Taking the smaller, $|f^*| \le 24$. The value found is 23, one below this bound. The bottleneck is neither at the source nor at the sink; it is in the middle of the network.

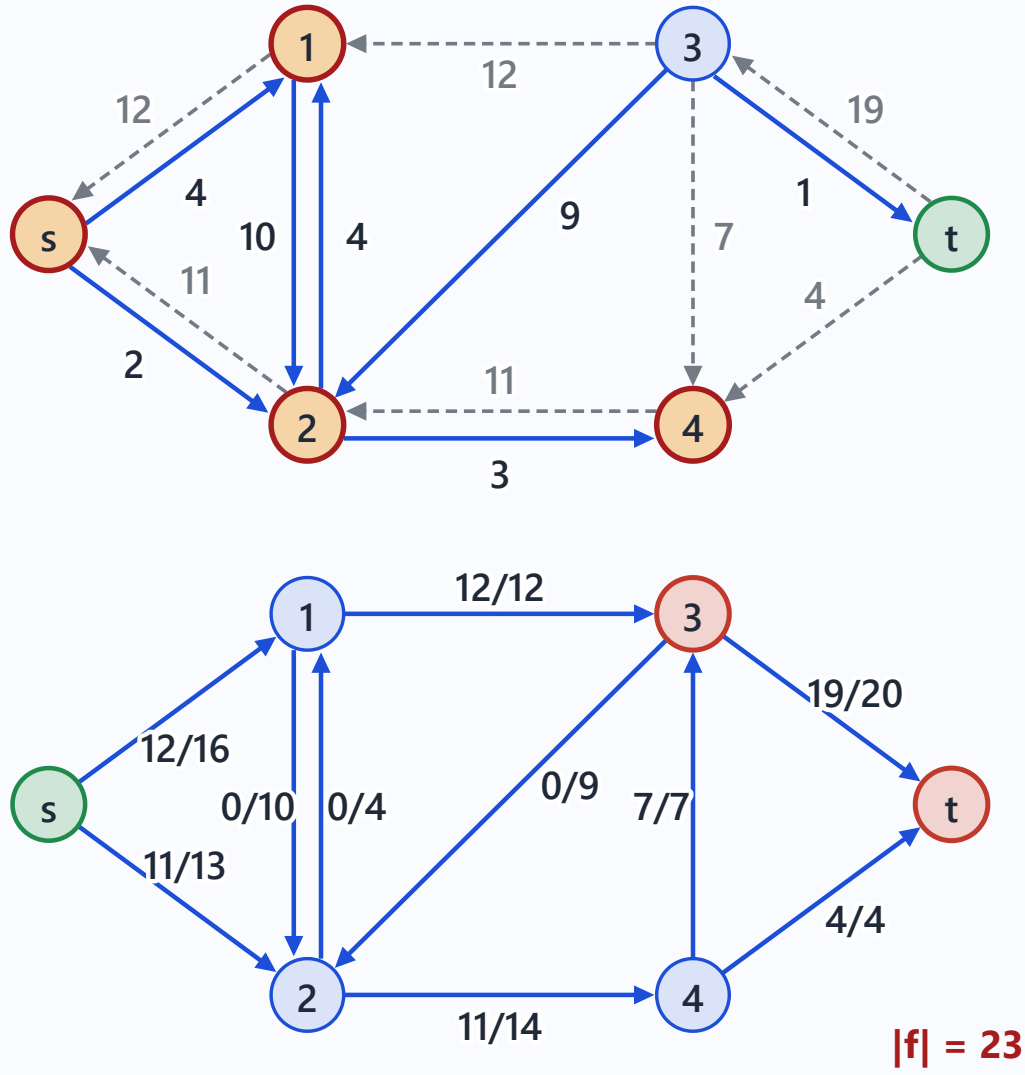


Figure 8.19: (a) The final residual network; the vertices reached from the source, *s, 1, 2, 4*, are highlighted. (b) The final flow and the minimum cut: the T side is *{3, t}* (red). The cut capacity is *12 + 7 + 4 = 23*.

Example 8.5

In the final residual network the set reachable from the source is $S = \{s, 1, 2, 4\}$, and the remaining set is $T = \{3, t\}$. The original edges going from S to T are:

- $1 \to 3$: capacity 12, flow 12 (full).
- $4 \to 3$: capacity 7, flow 7 (full).
- $4 \to t$: capacity 4, flow 4 (full).

The only edge going from T to S is $3 \to 2$, and its flow is 0. The cut capacity is $12 + 7 + 4 = 23$, equal to the value of the flow. Check from the sink side: the flow into t is $19 + 4 = 23$. The flow is maximum. It cannot be enlarged unless the capacity of one of the cut edges is increased.

It is also instructive to look at how the final flow is distributed over the edges. There is no flow on the edges $1 \to 2$ and $2 \to 1$; these two edges are not used at all in this solution. There are 12 units on $s \to 1$ and 11 on $s \to 2$. Even though spare capacity remains on the source edges, the flow cannot be increased, because the cut edges are full.

8.5.7 Example 2: the A–G network

The second example is a network in which BFS has to find long paths and uses a backward edge. The source is A and the sink is G. BFS tries neighbors in alphabetical order.

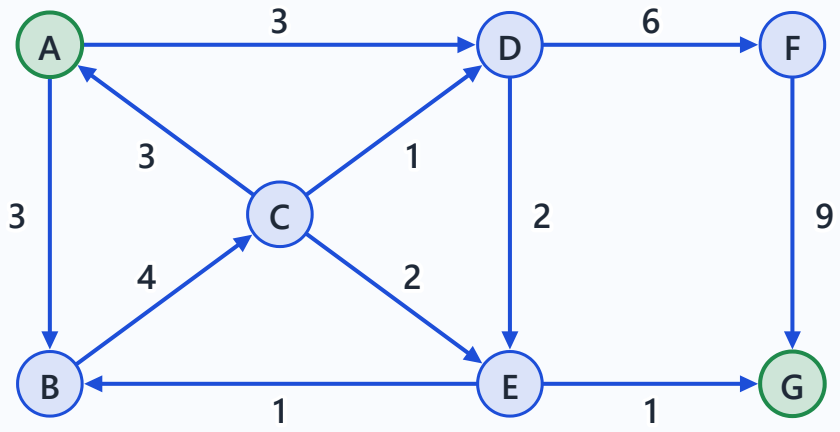


Figure 8.20: The A–G network. Capacities: $A \to B$ 3, $A \to D$ 3, $B \to C$ 4, $C \to A$ 3, $C \to D$ 1, $C \to E$ 2, $D \to E$ 2, $D \to F$ 6, $E \to B$ 1, $E \to G$ 1, $F \to G$ 9.

Round	Path	Edge	Bottleneck	Flow
1	A→D→E→G	3	E→G (1)	1
2	A→D→F→G	3	A→D (2 left)	3
3	A→B→C→D→F→G	5	C→D (1)	4
4	A→B→C→E→D→F→G	6	backward edge E→D (1)	5

Table 8.7: Augmenting paths found by Edmonds-Karp in the A–G network.

In the first round BFS puts B and D from A into the queue, C from B, and E and F from D. E leaves the queue before F and is the first to find G. The path is $A \to D \to E \to G$; since the capacity of $E \to G$ is 1, it carries only 1 unit. In the second round $E \to G$ is full. BFS finds $A \to D \to F \to G$; 2 units are left on $A \to D$. The flow becomes 3 and $A \to D$ is saturated.

In the third round the source can reach only B. The path stretches to five edges: $A \to B \to C \to D \to F \to G$. The bottleneck is $C \to D$ (1). The fourth round is the most interesting one. $C \to D$ is saturated; the search goes from C to E. There is no direct path from E to the sink, but in the first round 1 unit was

sent along $D \to E$. The residual network now has the backward edge $E \to D$, and the path uses it: $A \to B \to C \to E \to D \to F \to G$.

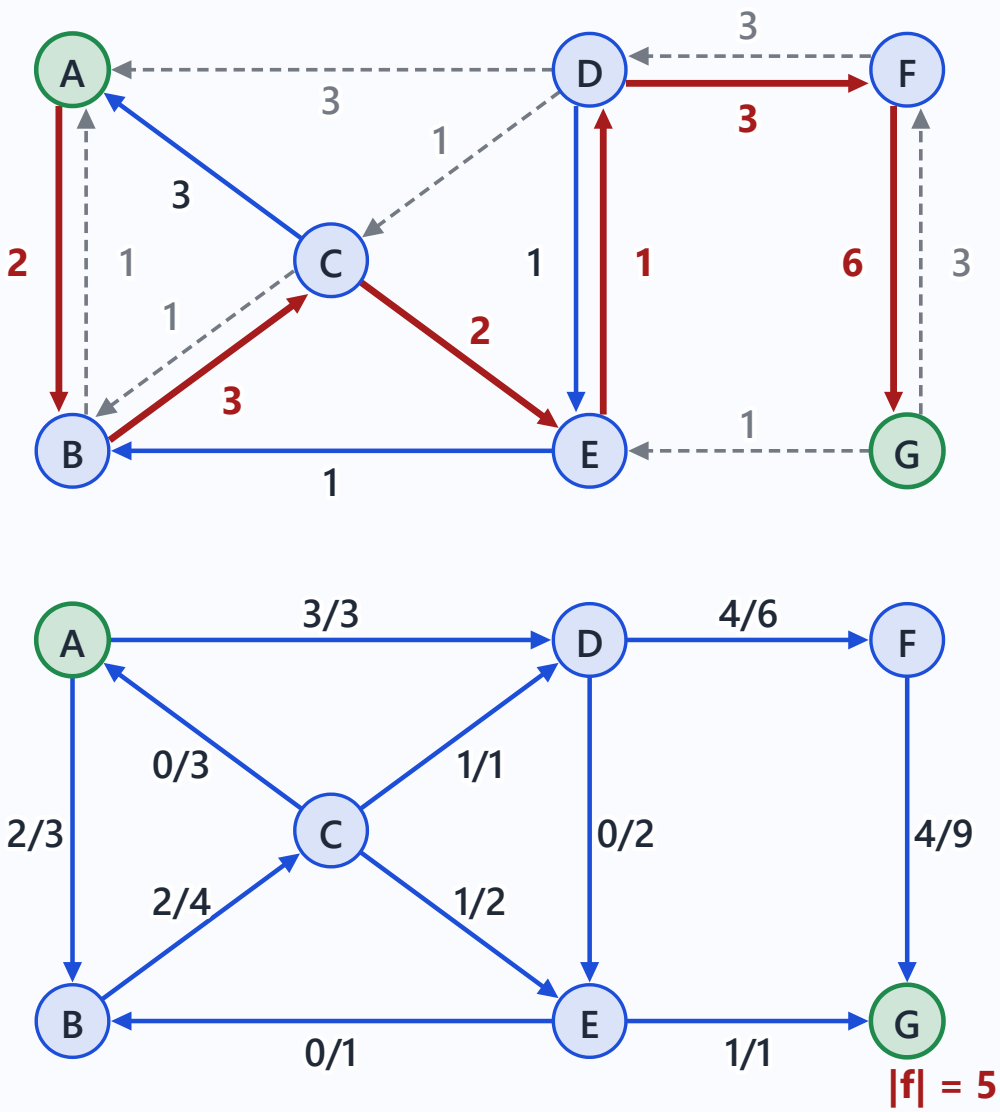


Figure 8.21: (a) The augmenting path of the fourth round; $E \to D$ is a backward edge. (b) The final flow: $|f| = 5$. The flow on $D \to E$ has dropped from 1 to 0.

The effect of the backward edge is this: the unit that in the first round passed from D to E and reached the sink along $E \to G$ now comes from C. The unit freed at D reaches the sink along $D \to F \to G$. In the final flow the edge $D \to E$ is empty. The path lengths are 3, 3, 5, and 6; again they do not decrease.

The fifth BFS cannot reach the sink. From the source, the vertices A, B, C, E are reached. The original edges leaving this set are $A \to D$ (3), $C \to D$ (1), and $E \to G$ (1). The cut capacity is $3 + 1 + 1 = 5 = |f^*|$. The edge $D \to E$ crosses the cut in the opposite direction and is empty in the final flow; the condition in the proof is again satisfied.

8.5.8 Example 3: on the residual network

In the third example the algorithm is followed directly on the residual network. One can see step by step how each augmentation changes the residual network.

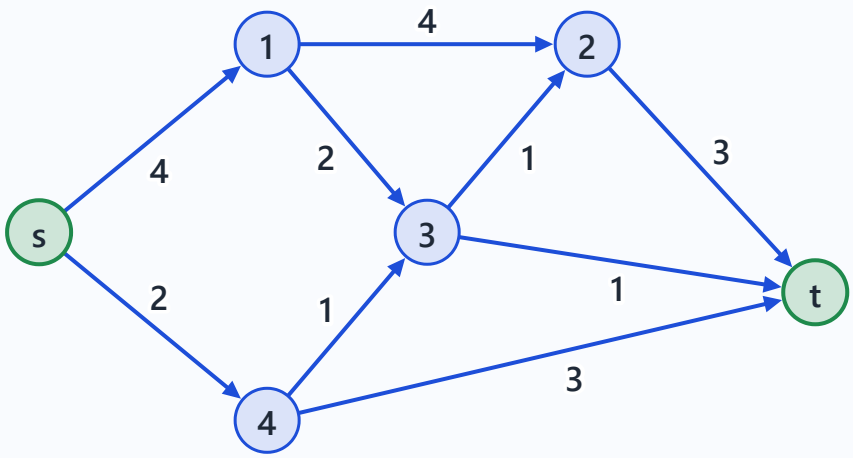


Figure 8.22: The third example network. Capacities: $s \to 1$ 4, $s \to 4$ 2, $1 \to 2$ 4, $1 \to 3$ 2, $3 \to 2$ 1, $2 \to t$ 3, $3 \to t$ 1, $4 \to 3$ 1, $4 \to t$ 3.

At the start the residual network equals the capacities. BFS first finds the two-edge path $s \to 4 \to t$. The bottleneck is $s \to 4$ (2); 2 units are pushed. The residual capacity of $s \to 4$ drops to 0, and the backward edge $4 \to s$ becomes 2. 1 unit is left on $4 \to t$.

Now there is no two-edge path. BFS finds the three-edge path $s \to 1 \to 2 \to t$; the bottleneck is $2 \to t$ (3). 3 units are pushed. 1 unit is left on $s \to 1$ and on $1 \to 2$; $2 \to t$ becomes saturated. In the next BFS the sink cannot be reached from 2. The path $s \to 1 \to 3 \to t$ is found; since 1 unit is left on $s \to 1$, 1 unit is pushed.

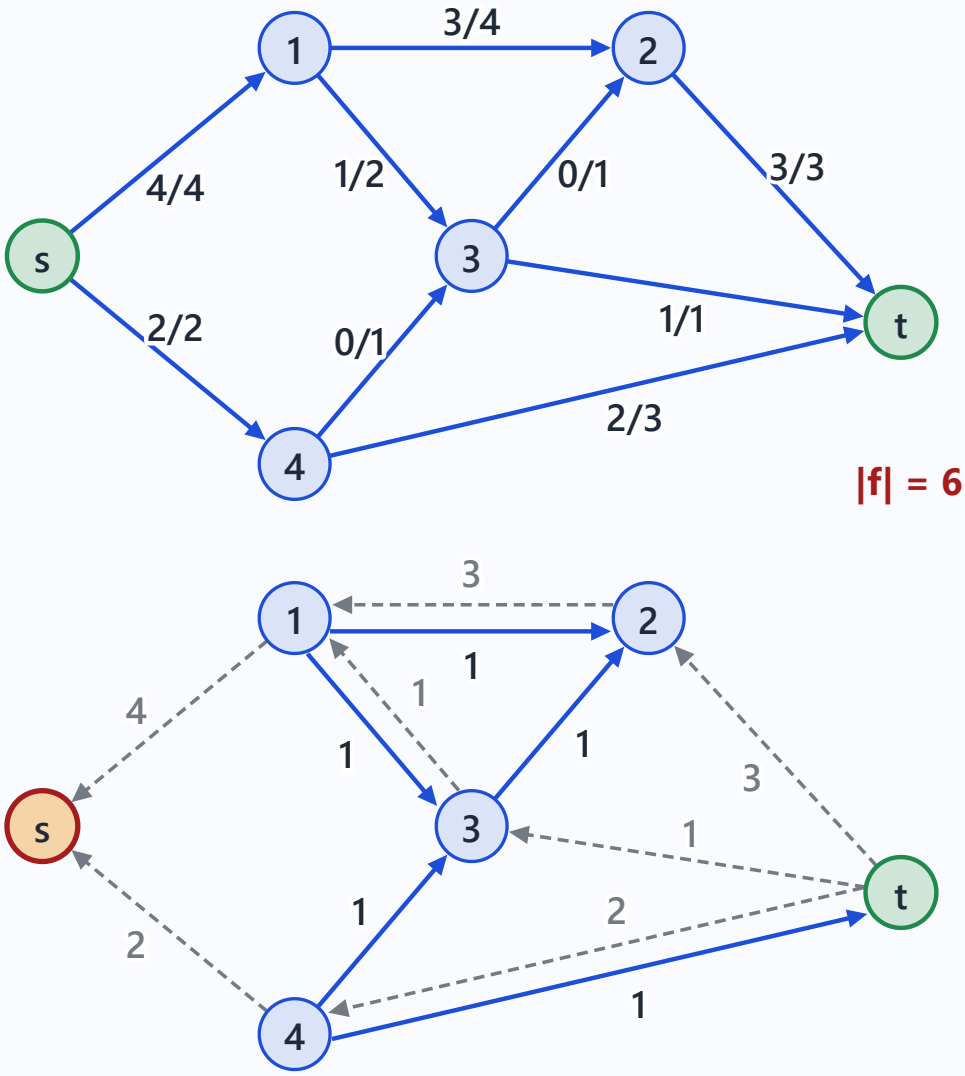


Figure 8.23: (a) The final flow (flow/capacity). (b) The final residual network: both edges leaving s are saturated, and nowhere can be reached from the source.

The edges leaving the source are saturated; no augmenting path is left. The maximum flow is $|f^*| = 2 + 3 + 1 = 6$: 3 units go along $s \to 1 \to 2 \to t$, 1 unit along $s \to 1 \to 3 \to t$, and 2 units along $s \to 4 \to t$. The minimum cut is *({s}, the rest)*, and its capacity is $4 + 2 = 6$. In this network the set *{s, 1, 2, 3}* also gives a cut of capacity 6; once again the minimum cut is not unique.

In this example no path used a backward edge. Some edges were never filled either: $3 \to 2$ and $4 \to 3$ stay empty, and there is 1 unit of spare room on $4 \to t$. A maximum flow does not fill every edge; it fills only the edges of a minimum cut.

8.6 Dinic's algorithm

Edmonds-Karp processes a single path after each BFS. Yet the information BFS finds is far richer than a single path: it gives the distance from the source to every vertex. Using these distances, all the shortest paths of the same length can be saturated in a single stage. Yefim Dinitz used this idea in the algorithm he published in 1970; the algorithm is known in the Western literature as Dinic's algorithm.

8.6.1 The level graph and the blocking flow

Definition 8.6

A BFS from the source in the residual network gives the **level** of every vertex v: *level(v)* is the number of edges on the shortest residual path from the source to v. The **level graph** is the subgraph of the residual network consisting only of the edges (u, v) that satisfy $level(v) = level(u) + 1$.

A flow in the level graph that leaves at least one saturated edge on every s–t path is called a **blocking flow**.

In the level graph every edge goes forward by one level. Edges between vertices of the same level, and edges that go backward, are discarded. Therefore every s–t path in the level graph is one of the shortest s–t paths in the residual network. Moreover the level graph is acyclic: the level increases at every step, so one cannot return to the same vertex.

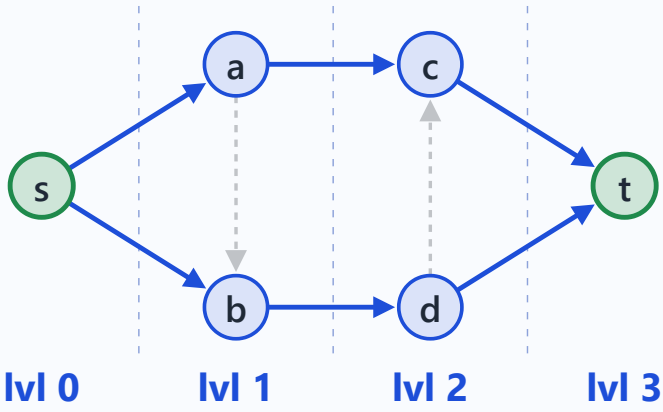


Figure 8.24: The level graph. The vertices are arranged in columns by their distance from the source (lvl 0, 1, 2, 3). Only edges going to the next column are kept; the edges $a \to b$ and $d \to c$ within a column (faded) are not used.

A blocking flow is not a maximum flow. It only blocks all the paths in the level graph; no s–t path is left in the level graph. But in the whole residual network, longer augmenting paths may remain through edges that do not skip a level and through newly created backward edges. This is why Dinic works in phases: in each phase it recomputes the levels and finds a new blocking flow.

8.6.2 The steps of the algorithm

1. **Initialization.** The flow on all edges is 0; the residual network is the same as the original network.
2. **BFS.** A BFS from the source determines the level of every vertex. BFS uses only edges with positive residual capacity. If the sink has no level, that is, if the sink cannot be reached, the flow is maximum and the algorithm stops.
3. **DFS.** Augmenting paths that go forward by one level at each step are searched for with DFS. Only edges with $level(v) = level(u) + 1$ are followed. Edges found to be dead ends are not tried again during that phase.
4. **Update.** The smallest residual capacity on each path found is added to the total flow; the residual capacities of the edges and their backward edges are updated. When no path remains, the blocking flow of this phase has been found.
5. **New phase.** Return to step 2.

In every phase the distance from the source to the sink grows by at least 1. Since the distance can be at most $V - 1$, there are at most $V - 1$ phases. This is the first factor in the algorithm's running-time bound.

8.6.3 Example: the phases

Let us run the algorithm on the network of Figure 8.25. This network is the same as that of the second Ford-Fulkerson example; only the names of vertices 2 and 3 are swapped. We therefore know that the maximum flow is 19.

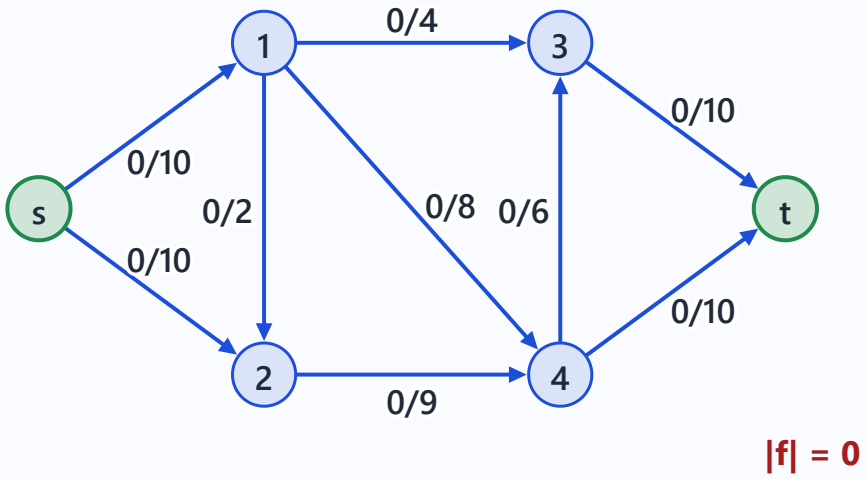


Figure 8.25: The Dinic example network (flow/capacity); initially all flows are 0. Capacities: $s \to 1$ 10, $s \to 2$ 10, $1 \to 2$ 2, $1 \to 3$ 4, $1 \to 4$ 8, $2 \to 4$ 9, $4 \to 3$ 6, $3 \to t$ 10, $4 \to t$ 10.

Phase 1, BFS. Levels: $s = 0$; $1, 2 = 1$; $3, 4 = 2$; $t = 3$. The edge $1 \to 2$ joins two level-1 vertices, and the edge $4 \to 3$ joins two level-2 vertices. Neither skips a level, and neither is used in this phase. The distance from the source to the sink is 3.

Phase 1, blocking flow. DFS finds three paths in the level graph: $\{s, 1, 3, t\}$ with 4 units (the capacity of $1 \to 3$), $\{s, 1, 4, t\}$ with 6 units (what is left on $s \to 1$), and $\{s, 2, 4, t\}$ with 4 units (what is left on $4 \to t$). The total is $|f| = 14$. After this phase no s–t path is left in the level graph: $s \to 1$, $1 \to 3$, and $4 \to t$ are saturated.

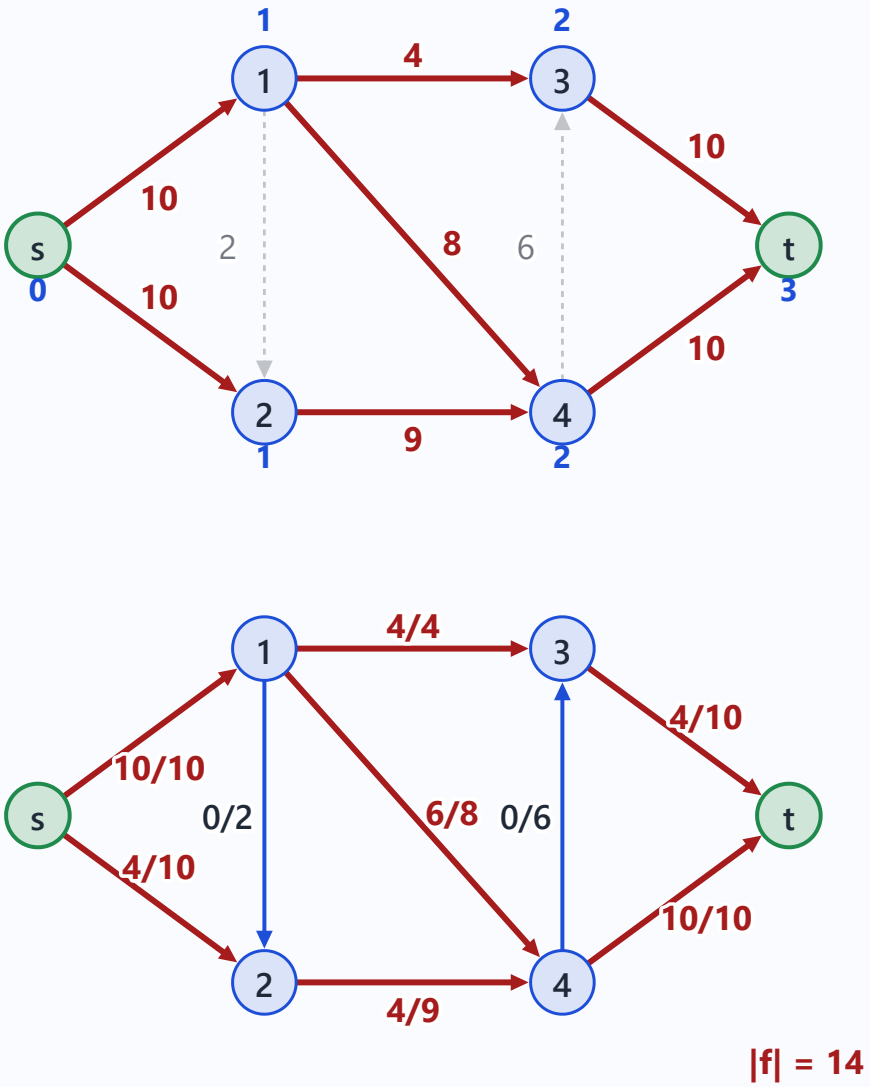


Figure 8.26: The first phase. (a) The level graph; the blue numbers next to the vertices are levels, and the faded edges skip no level. (b) The flow after the blocking flow; $|f| = 14$.

Phase 2, BFS. In the residual network the source can reach only 2 (6 units remain on $s \to 2$). From 2, 5 units can go to 4. From 4 one can go in two directions: the edge $4 \to 3$ and the backward edge $4 \to 1$ of the flow sent to $1 \to 4$ in the first phase. From 3 one also reaches the sink. The levels are $s = 0$; $2 = 1$; $4 = 2$; $1, 3 = 3$; $t = 4$. The distance from the source to the sink has risen from 3 to 4.

Phase 2, blocking flow. DFS first descends from 4 to 1. Vertex 1 has no edge to a vertex at level 4; the edge $4 \to 1$ is a dead end. DFS abandons this edge and continues with $4 \to 3 \to t$. The path is $\{s, 2, 4, 3, t\}$, and the bottleneck is $2 \to 4$ ($9 - 4 = 5$). $|f| = 14 + 5 = 19$.

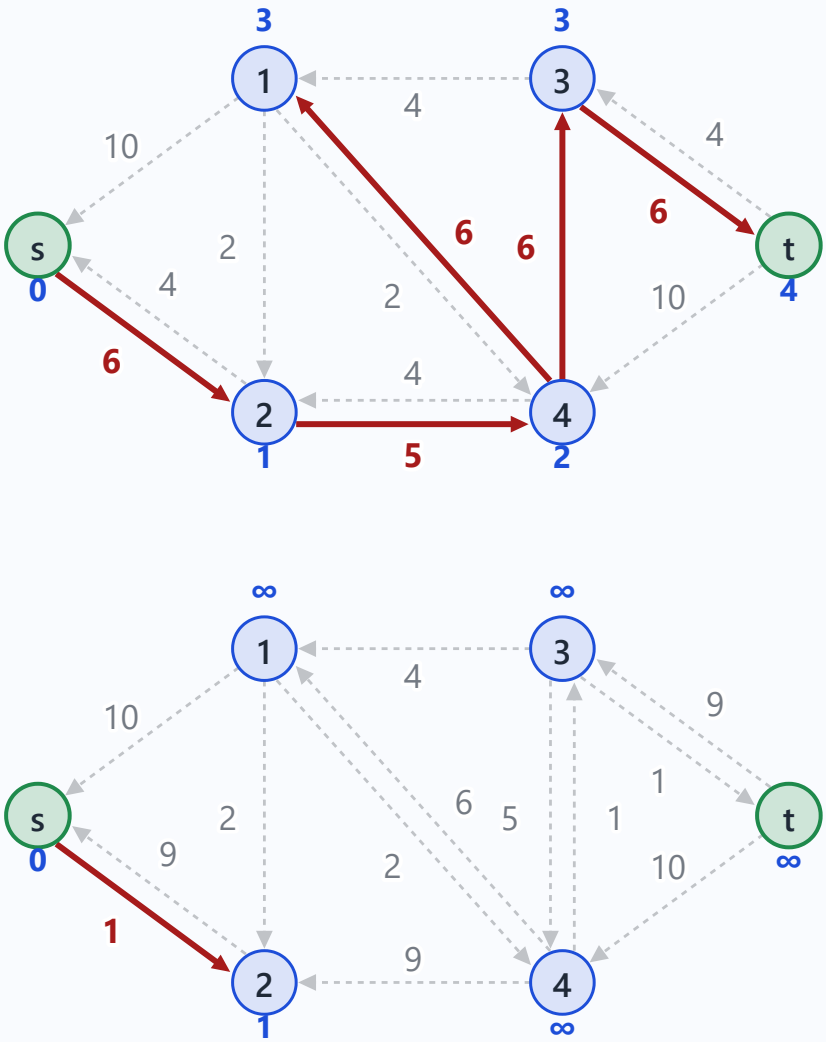


Figure 8.27: (a) The level graph of the second phase; the dashed edges are backward edges of the first phase. (b) The third BFS: from the source only 2 is reached, and the sink has no level (∞).

Phase 3, BFS. In the residual network only 2 is reached from the source (1 unit remains on $s \to 2$); $2 \to 4$ is saturated. The sink has no level, and the algorithm ends. The minimum cut is *({s, 2}, the rest)*: $s \to 1$ (10) + $2 \to 4$ (9) = 19.

Result: the maximum flow is 19, and only 2 phases, that is, $2 + 1 = 3$ BFS runs, were enough. On the same network the second Ford-Fulkerson example used 5 augmenting paths. Edmonds-Karp makes 4 augmentations and 5 BFS runs on this network. Dinic's gain comes from processing the three paths of length 3 in the first phase with the information of a single BFS.

8.6.4 Pseudocode

The main loop and the BFS of Dinic's algorithm are below.

```
DINIC(G, source, sink):
  flow = [ ]       // 0 for every edge
  maxFlow = 0
  while BFS(G, source, sink, flow):
    // level graph built
    while a blocking flow remains:
      delta = DFS(G, source, sink,
                  flow, level, ∞)
      maxFlow += delta
  return maxFlow

BFS(G, source, sink, flow):
  level[v] = -1, for each v
  level[source] = 0
  queue.add(source)
  while queue is not empty:
    u = queue.remove()
    for each (u, v) in G[u]:
      if level[v] < 0 and
           capacity[u,v] - flow[u,v] > 0:
        level[v] = level[u] + 1
        queue.add(v)
  return level[sink] ≥ 0
```

The outer loop runs as long as BFS reaches the sink; each round is a phase. The inner loop calls DFS again and again until the blocking flow is complete. Each DFS finds a path and returns the flow of that path; when it returns 0 the phase ends. The BFS is very similar to the BFS in Edmonds-Karp. The difference is that the level is recorded instead of the parent. The level of unreached vertices is −1. BFS returns `true` if the level of the sink is not negative.

```
DFS(G, u, sink, flow, level, limit):
  if u == sink: return limit
  for each (u, v) in G[u]:
    remaining = capacity[u,v] - flow[u,v]
    if level[v] == level[u] + 1
         and remaining > 0:
      delta = DFS(G, v, sink, flow, level,
                  min(limit, remaining))
      if delta > 0:
        flow[u, v] += delta
        flow[v, u] -= delta   // backward
        return delta
  return 0
```

The `limit` parameter of DFS is the bottleneck of the path from the source to `u`. It starts at ∞ at the source and shrinks at every edge to `min(limit, remaining)`. When the sink is reached this value is returned; it is the bottleneck of this path. On the way back each edge increases its flow by `delta` and

the flow of the backward edge decreases. The level condition (`level[v] == level[u] + 1`) ensures that the search advances only in the level graph.

8.6.5 Java: level-based DFS and the current-arc array

The DFS in the pseudocode tries all edges from the beginning on every call. Even when an edge is known to be a dead end, it is tried again on the next call. This needlessly lengthens the phase. The Java implementation avoids this with a small array:

```
int dfs(int u, int t, int pushed, int[] ptr) {
  if (u == t) return pushed;
  for (; ptr[u] < adj[u].size(); ptr[u]++) {
    Edge e = adj[u].get(ptr[u]);
    if (level[e.to] == level[u] + 1 && e.flow < e.cap) {
      int d = dfs(e.to, t, Math.min(pushed, e.cap - e.flow), ptr);
      if (d > 0) {
        e.flow += d;                          // forward edge
        adj[e.to].get(e.rev).flow -= d;       // backward edge
        return d;
      }
    }
  }
  return 0;   // no level path left from u to t
}
```

The edges are kept in adjacency lists. Each `Edge` object carries the target vertex, the capacity, the flow, and a `rev` field. `rev` is the position of the reverse edge in the target vertex's list. So the expression `adj[e.to].get(e.rev)` reaches the reverse edge in *O(1)*. When an edge is added, two objects are created together: the forward edge with its capacity and the backward edge with capacity 0.

`ptr[u]` is the index of the next edge to try at `u` ; this structure is called the *current arc*. The `for` loop starts at `ptr[u]` , and the counter is incremented directly in this array. If no flow reaches the sink through an edge (`d == 0`), the loop moves on to the next edge with `ptr[u]++` , and this edge is never tried again during that phase. If flow does reach the sink, the method returns without incrementing `ptr[u]` . The edge may still have room; the next call starts from the same edge.

The condition `e.flow < e.cap` says that the residual capacity is positive. For backward edges the capacity is 0 and the flow is negative: $-f$. The condition becomes $-f < 0$, that is, $f > 0$, and the residual capacity comes out as $0 - (-f) = f$. Forward and backward edges are handled by the same line. At the start of every phase the `ptr` array is reset and the levels are recomputed by BFS.

Common mistake

Incrementing `ptr[u]` in every case. When flow gets through, the edge may not be saturated; if the counter is incremented, the remaining capacity of the edge goes unused in that phase. The algorithm still gives the correct result, but the blocking flow is left incomplete and the number of phases grows. Conversely, if the counter is never used, dead-end edges are tried again and again and the per-phase bound of $O(V \cdot E)$ is lost.

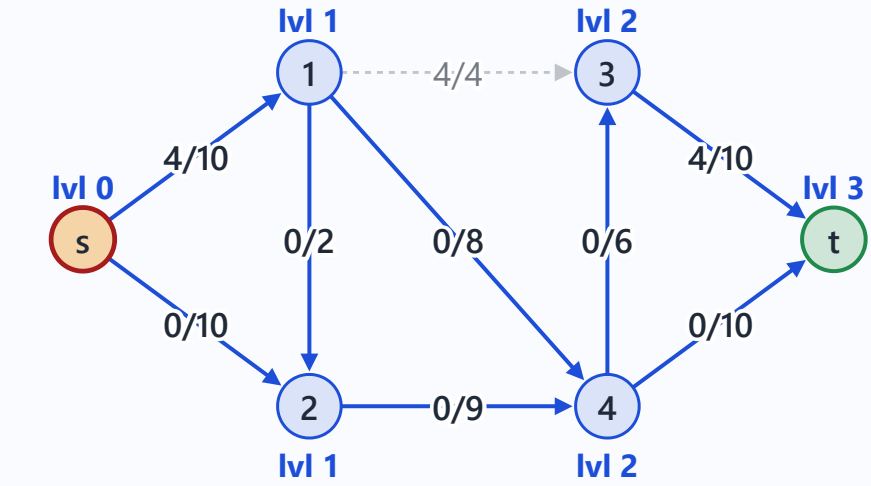
8.6.6 Code trace: the blocking flow in the first phase

Let us trace the DFS calls of the first phase. In the adjacency lists the edges are in the order they were added: in the list of 1, after the backward edge from s come $1 \to 2$, $1 \to 3$, and $1 \to 4$; in the list of 4, after two backward edges come $4 \to 3$ and $4 \to t$. Backward edges do not satisfy the level condition in this phase.

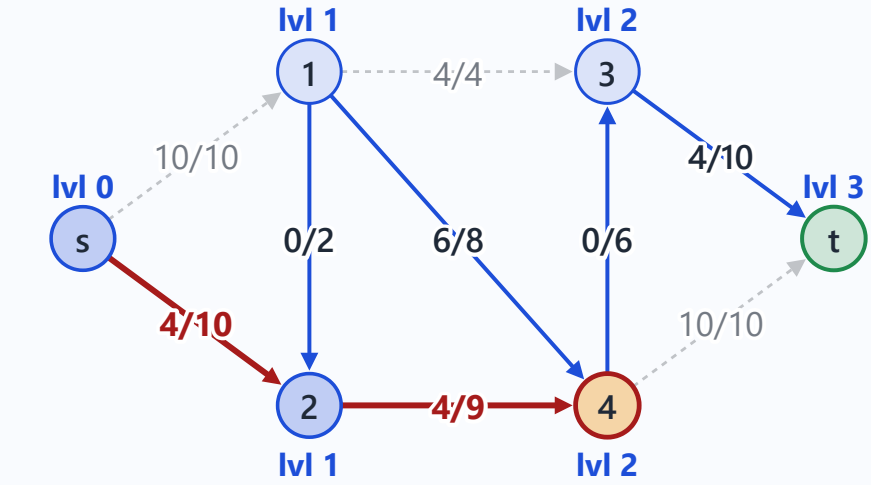
Call	Path descended (remaining)	Event	Returned `d`	$\lvert f \rvert$
1	s→1 (10), 1→3 (4), 3→t (10)	$1 \to 2$ is at the same level, `ptr[1]++`. The flow is limited to $\min(10, 4, 10) = 4$.	4	4
2	s→1 (6), 1→4 (8), 4→t (10)	$1 \to 3$ is saturated, `ptr[1]++`. At 4, $4 \to 3$ is at the same level, `ptr[4]++`.	6	10
3	s→2 (10), 2→4 (9), 4→t (4)	$s \to 1$ is saturated, `ptr[s]++`. 4 units remain on $4 \to t$.	4	14
4	s→2 (6), 2→4 (5)	At 4, $4 \to t$ is saturated, `ptr[4]++`; the list is finished: 0. At 2, `ptr[2]++`: 0. At s, `ptr[s]++`: 0.	0	14

Table 8.8: The calls `dfs(s, t, ∞, ptr)` in the first phase.

The fourth call shows that the blocking flow is complete. In this call no level path is left from 4: $4 \to 3$ is at the same level and $4 \to t$ is saturated. As the value 0 is passed upward, every vertex advances its counter. Finally `ptr[s]` reaches the end of the list and the phase ends with $\lvert f \rvert = 14$.



calls: dfs(s, ∞)



calls: dfs(s, ∞) → dfs(2, 6) → dfs(4, 5)

Figure 8.28: Two moments of the level-based DFS. (a) As the flow of the first path returns, 4 is added to each edge; $|f| = 4$. (b) The fourth call: the descent went along $s \to 2 \to 4$, and no level path is left from 4.

In the table the total number of increments of the `ptr` counters is small: each edge is declared a "dead end" at most once during a phase. This is the value of the current-arc array. Since a dead-end edge is never looked at again, the total work of all the DFS calls in a phase stays bounded.

8.6.7 Complexity: $O(V^2 \cdot E)$

Theorem 8.6 (Dinic)

Dinic's algorithm makes at most $V - 1$ phases. With the current-arc array each phase takes $O(V \cdot E)$. The total time is $O(V^2 \cdot E)$.

Proof. **Number of phases.** At the end of a phase every s–t path in the level graph contains a saturated edge. The new edges in the residual network of the next phase are only backward edges, and these go back one level in the level graph. An s–t path of the same length as before could consist only of edges of the level graph; all of these paths are blocked. So the distance from the source to the sink strictly increases. Since the distance lies between 1 and $V - 1$, there are at most $V - 1$ phases.

Time per phase. BFS takes $O(V + E)$. The work of the DFS runs splits into two parts. Each successful DFS follows a path of at most $V - 1$ edges and saturates at least one edge on it. A saturated edge is not used again in that phase; so there are at most E successful DFS runs, and their total cost is $O(V \cdot E)$. In unsuccessful attempts the counter advances one step; at most once per edge, $O(E)$ in total. Per phase, $O(V \cdot E)$.

Total: $O(V) \cdot O(V \cdot E) = O(V^2 \cdot E)$. □

Comparing Dinic's bound with Edmonds-Karp's makes the gain clear. In networks with $E \geq V$ we have $V^2 \cdot E \leq V \cdot E^2$; the gap grows as the network gets denser. For $V = 1000$ and $E = 10{,}000$, the value of $V \cdot E^2$ is 10^{11} and the value of $V^2 \cdot E$ is 10^{10}.

In special networks the bound is even better. If all capacities are 1, the number of phases is bounded by $O(\sqrt{E})$ and each phase takes $O(E)$: in total $O(E \cdot \sqrt{E})$. In unit networks, such as bipartite matching, where at most one unit of flow passes through each vertex, the time drops to $O(E \cdot \sqrt{V})$. This is the bound of the Hopcroft-Karp algorithm; Hopcroft-Karp is in fact a form of Dinic's algorithm adapted to matching.

In practice Dinic's algorithm is the maximum-flow algorithm most used in competitive programming. Its code is short and it usually runs far below its bound. In most networks the number of phases is far smaller than V; in the example of these notes 2 phases were enough.

8.7 Push–relabel

8.7.1 The idea: letting water run downhill

All the algorithms so far work globally: at every step they look for a complete path from the source to the sink. The **push–relabel** algorithm, published by A. Goldberg and R. Tarjan in 1988, takes a completely different route. It does not search for augmenting paths. Each operation concerns a single vertex and its neighbors; the algorithm reaches the maximum flow by repeating these local operations.

It helps to think of the algorithm as a water system. At the start, all the pipes leaving the source are filled to the brim. The water collects at the source's neighbors. Every vertex has a **height**, and water flows only from a higher vertex to a lower one. A vertex that cannot pass its accumulated water to any neighbor is raised; this is called a **relabel**. Passing water to a neighbor is called a **push**. Excess water

that cannot reach the sink returns to the source once its height exceeds the height of the source. When no vertex holds any accumulated water, the water that has reached the sink is the maximum flow.

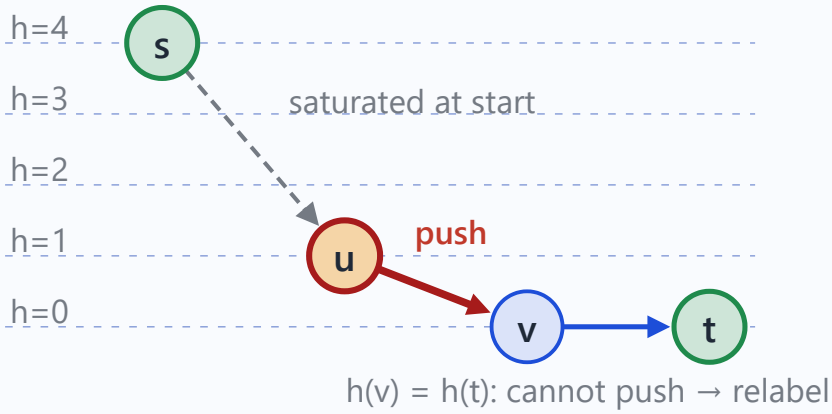


Figure 8.29: The idea of push–relabel. The source is at height $h = 4$ and its edges are saturated at the start. u ($h = 1$) pushes its excess to a lower neighbor (v, $h = 0$). Since v and t are at the same height, v cannot push to the sink; it must be relabeled first.

8.7.2 Preflow and excess flow

The structure in the intermediate steps of the algorithm is not a flow; the conservation constraint is violated. More water may enter a vertex than leaves it. This relaxed structure is called a preflow.

Definition 8.7

A **preflow** is a function f that obeys the capacity constraint and, in place of conservation, satisfies the following condition: for every vertex v other than the source,

$$e(v) = \Sigma_u f(u, v) - \Sigma_w f(v, w) \geq 0$$

$e(v)$ is called the **excess flow** of v: the total flow in minus the total flow out. A vertex other than the source and sink with $e(v) > 0$ is an **active vertex**.

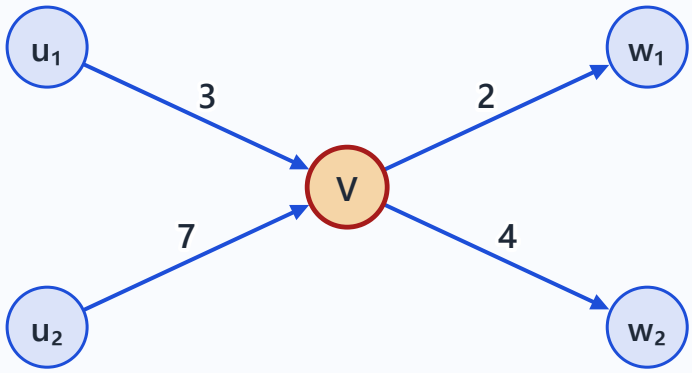


Figure 8.30: Vertex V receives 3 units from u_1 and 7 from u_2; 2 units leave V toward w_1 and 4 toward w_2. The excess is $e(V) = (3 + 7) - (2 + 4) = 4$.

In a preflow, conservation is relaxed in only one direction: a vertex can accumulate flow, but it cannot give away flow it does not have. The excess is never negative. Conservation means that $e(v) = 0$ at all intermediate vertices. So when no active vertex remains, the preflow has become a true flow. The algorithm runs until that moment.

8.7.3 The height function and initialization

The algorithm assigns each vertex an integer $h(v)$. The initialization is as follows:

- The flow on all edges is 0.
- $h(s) = |V|$, the number of vertices; all other vertices have $h = 0$.
- Every edge leaving the source is saturated: $f(s, v) = c(s, v)$ and $e(v) = c(s, v)$.

Two invariants are maintained throughout the algorithm. First, $h(s) = |V|$ and $h(t) = 0$ never change; the source and the sink are not relabeled. Second, for every edge (u, v) in the residual network, $h(u) \leq h(v) + 1$. This second invariant says that the height can drop by at most 1 along an edge. Slopes cannot be steep.

The second invariant has an important consequence: $h(u)$ is a lower bound on the residual-network distance from u to the sink. If there is a residual path with k edges from u to the sink, the height drops by at most 1 on each edge; since the height of the sink is 0, $h(u) \leq k$.

Theorem 8.7 (No s–t path)

As long as the invariants hold, there is no path from the source to the sink in the residual network. Hence, if the algorithm stops when no active vertex remains, the flow obtained is maximum.

Proof. If there were a path from the source to the sink in the residual network, as a simple path it would have at most $|V| - 1$ edges. By the observation just made, we would need $h(s) \le |V| - 1$. But $h(s) = |V|$. A contradiction. When no active vertex remains, the preflow is a flow and its residual network has no augmenting path; by the max-flow min-cut theorem it is a maximum flow. □

This viewpoint shows that the algorithm works the other way around from the Ford-Fulkerson family. Ford-Fulkerson maintains a valid flow at every moment and runs until no augmenting path is left. Push–relabel maintains the condition "no augmenting path" at every moment and runs until the preflow is a valid flow. The initial value $h(s) = |V|$ establishes this condition. Because the edges leaving the source are saturated, no edge leaves the source in the residual network; the invariant holds at the start as well.

8.7.4 The push and relabel operations

Push: push(u, v). The condition has three parts: $e(u) > 0$; $c_f(u, v) > 0$, that is, the edge has a gap between capacity and flow; and $h(u) = h(v) + 1$. Under these conditions $\delta = \min(e(u), c_f(u, v))$ units of flow are pushed: $f(u, v)$ increases (or $f(v, u)$ decreases if it is a backward edge), $e(u) \mathrel{-}= \delta$, $e(v) \mathrel{+}= \delta$. If the push saturates the edge it is called a **saturating push**, otherwise a **non-saturating push**. In a non-saturating push, $\delta = e(u)$ and the excess of u is emptied completely.

Relabel: relabel(u). The condition: $e(u) > 0$, but $h(u) \le h(v)$ for every edge (u, v) in the residual network; that is, it cannot push to any neighbor. The operation is $h(u) = 1 + \min h(v)$, with the minimum taken over the edges (u, v) in the residual network. With this new value at least one push becomes possible: the lowest neighbor is exactly one step below.

Both operations preserve the invariants. A push is made only on an edge with $h(u) = h(v) + 1$. On the backward edge (v, u) that may be created, $h(v) = h(u) - 1 \le h(u) + 1$; the invariant holds. In a relabel, $h(u)$ is chosen exactly one more than the smallest of its residual neighbors; the invariant is preserved on the edges leaving u. On the edges entering u, an increase of $h(u)$ only relaxes the condition.

Flow is pushed from high to low, and heights only increase. A vertex with excess flow always has a path to the source in the residual network: this flow came from the source along some path, and the backward edges of that path are in the residual network. So the *min* in the relabel is always defined. Excess that cannot reach the sink is pushed back toward the source once it rises above $|V|$, the height of the source.

> **Common mistake**
>
> Mixing up the direction of the push or the initial height. Flow is pushed only from high to low, when $h(u) = h(v) + 1$. Pushing to a neighbor of equal height breaks the invariant. At the start $h(s) = |V|$ must hold; if the source is left at 0, the excess cannot return to the source and the "no augmenting path" guarantee is lost.

8.7.5 Example: step by step

The network in Figure 8.31 has six vertices: $|V| = 6$. Active vertices may be chosen in any order; the order in the example was chosen to show the operations clearly.

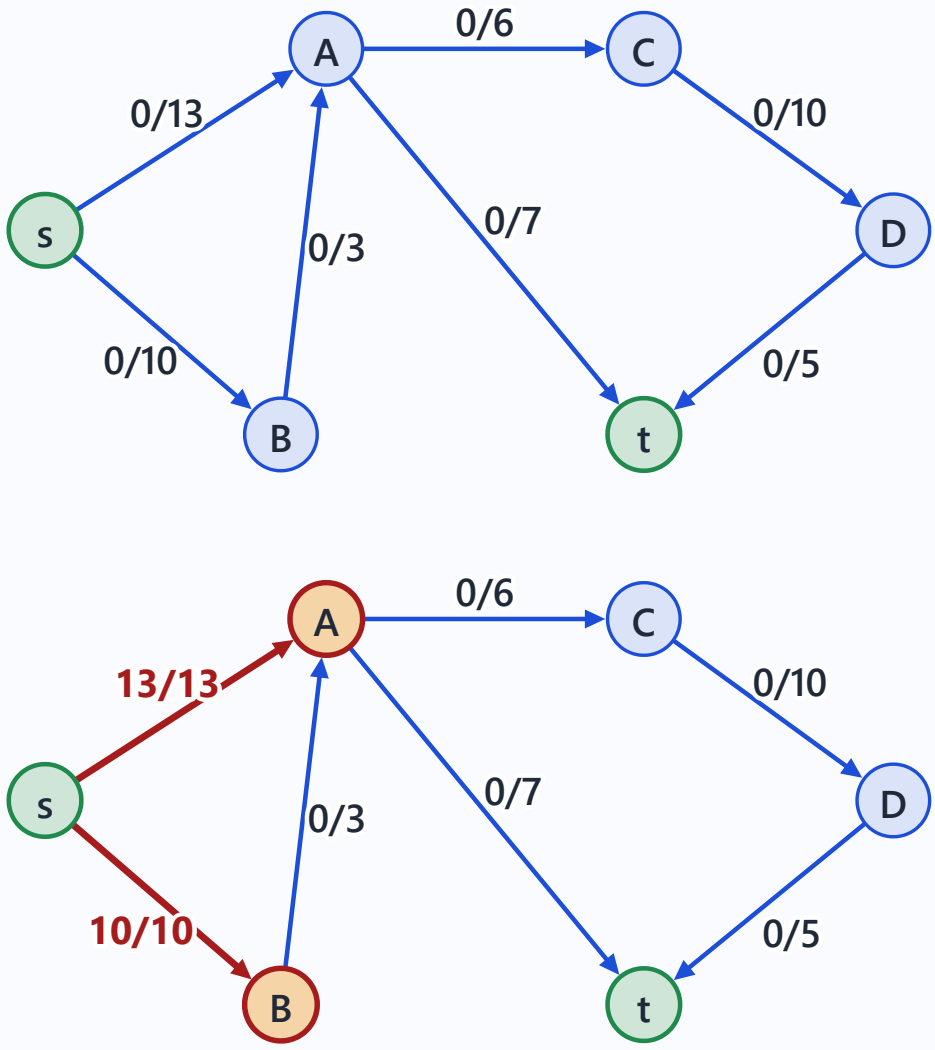


Figure 8.31: (a) The push–relabel example network (flow/capacity): $s \to A$ 13, $s \to B$ 10, $B \to A$ 3, $A \to C$ 6, $A \to t$ 7, $C \to D$ 10, $D \to t$ 5. (b) Start: $h(s) = 6$, the two edges of the source are saturated; $e(A) = 13$, $e(B) = 10$.

Step	Operation	A (h, e)	B (h, e)	C (h, e)	D (h, e)	e(t)
0	start: $s \to A$ and $s \to B$ are saturated	0, 13	0, 10	0, 0	0, 0	0
1	relabel(B): $1 + \min(h(A) = 0, h(s) = 6)$	0, 13	1, 10	0, 0	0, 0	0
2	push(B, A): $\delta = \min(10, 3) = 3$	0, 16	1, 7	0, 0	0, 0	0
3	relabel(A): $1 + \min(0, 0, 1, 6)$	1, 16	1, 7	0, 0	0, 0	0
4	push(A, C) 6, push(A, t) 7 (both saturate)	1, 3	1, 7	0, 6	0, 0	7

Step	Operation	A (h, e)	B (h, e)	C (h, e)	D (h, e)	e(t)
5	relabel(C): *1 + h(D)*	1, 3	1, 7	1, 6	0, 0	7
6	relabel(A): neighbors B (1), s (6)	2, 3	1, 7	1, 6	0, 0	7
7	push(A, B) 3 (cancels the flow on $B \to A$)	2, 0	1, 10	1, 6	0, 0	7
8	relabel(B) → 3, push(B, A) 3	2, 3	3, 7	1, 6	0, 0	7
9	Alternating relabel and push between A and B: *h(A) = 4*, *h(B) = 5*, *h(A) = 6*, *h(B) = 7*; B pushes 3 units to A and 7 units to the source	6, 3	7, 0	1, 6	0, 0	7
10	relabel(A): *1 + min(6, 7) = 7*; push(A, s) 3	7, 0	7, 0	1, 6	0, 0	7
11	push(C, D) 6	7, 0	7, 0	1, 0	0, 6	7
12	relabel(D) → 1, push(D, t) 5	7, 0	7, 0	1, 0	1, 1	12
13	relabel(D) → 2, push(D, C) 1	7, 0	7, 0	1, 1	2, 0	12
14	Alternating relabel and push between C and D: C 3, 5, 7, 8; D 4, 6, 8; C pushes 1 unit to A	7, 1	7, 0	8, 0	8, 0	12
15	push(A, s) 1	7, 0	7, 0	8, 0	8, 0	12

Table 8.9: Trace of push–relabel. The heights (h) and excess flows (e) after each row; *h(s) = 6* and *h(t) = 0* do not change.

The first steps show the conditions of the operations. In the first step B is active, but of its residual neighbors A is at the same height (0); the source is far higher. B cannot push and is relabeled. It then looks at A from one step above and pushes 3 units, the capacity of the edge $B \to A$. In the third step the neighbors of A are C (0), t (0), B (1; the backward edge of the flow $B \to A$), and s (6; the backward edge of the flow $s \to A$). The smallest is 0, and *h(A) = 1*.

In the fourth step A distributes its excess to the sink and to C. The edge to the sink saturates at 7 and the edge of C at 6. 3 units remain at A. These 3 units have no way to reach the sink: both exits of A are full. Most of the following steps are this excess returning to the source.

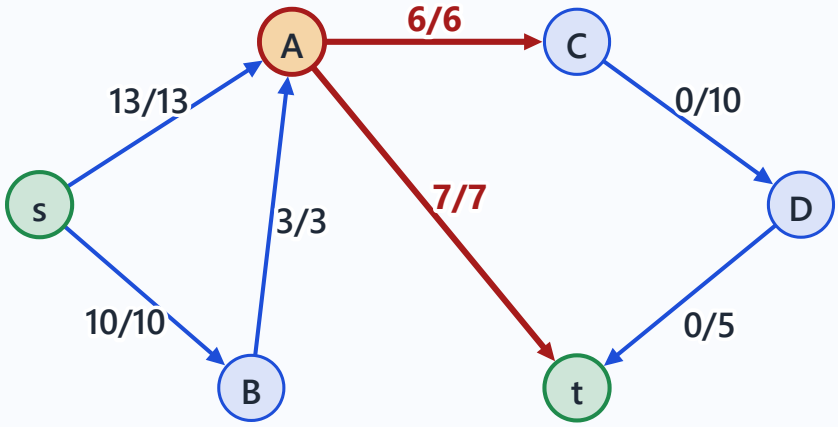


Figure 8.32: The fourth step: A pushes 6 units to C and 7 to the sink. Both edges saturate; *e(A) = 3*, *e(C) = 6*, and 7 at the sink.

The part from the sixth step to the ninth shows the most criticized aspect of the algorithm. A and B push the excess back and forth; each time one of them rises a step. This "playing catch" continues until both heights exceed 6, the height of the source. When $h(B) = 7$, B is higher than the source. Of the 10 units of excess at B, 3 go to A and 7 go to the source (the flow on $s \to B$ drops from 10 to 3). In the tenth step A also rises to 7 and pushes its own 3 units to the source.

In the following steps the same thing happens between C and D. The 6 units of C go to D; D pushes 5 of them to the sink, because the capacity of $D \to t$ is 5. The remaining 1 unit goes back and forth between C and D. When the heights exceed 7, the height of A, C pushes this unit to A (the flow on $A \to C$ drops from 6 to 5); A then pushes it to the source.

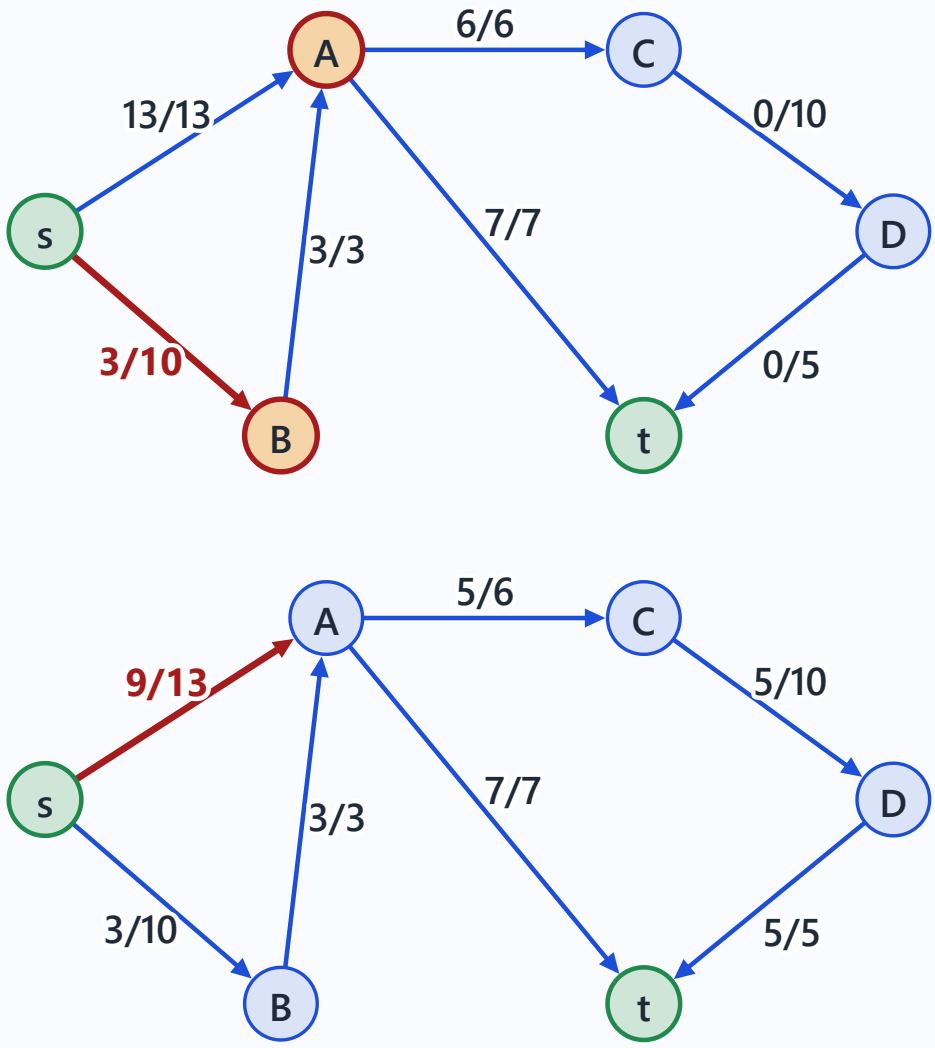


Figure 8.33: (a) The ninth step: the excess of B has returned to the source; the flow $s \to B$ has dropped to 3. (b) The final state: no active vertex. $s \to A$ 9, $s \to B$ 3; the flow into the sink is 12.

When no active vertex remains, the maximum flow is the flow into the sink: $7 + 5 = 12$. The flow out of the source is also $9 + 3 = 12$. The minimum cut is $S = \{s, A, B, C, D\}$, $T = \{t\}$: $A \to t$ (7) + $D \to t$ (5) = 12.

The order of selection in the example is not the only option. The Java functions on the slides can also be run with a main loop that picks the first active vertex in index order each time. On the same network this loop performs 38 operations (relabels and pushes) and again finds 12. The order changes the number of operations, not the result.

8.7.6 Pseudocode

The initialization part of the algorithm is as follows:

```
PUSH_RELABEL(G, source, sink):
  height[source] = |V|
  height[v] = 0, v ≠ source
  excess[v] = 0, for each v
  flow[e] = 0, for each edge e
  // preflow: saturate the source edges
  for each (source, v) ∈ E:
    c = capacity[source, v]
    flow[source, v] = c
    flow[v, source] = -c
    excess[v] = c
    excess[source] -= c
```

The flow is kept in a skew-symmetric table: when a source edge is saturated, `flow[v, source] = -c` is written. The excess of the source becomes negative; the source is a special vertex in this respect. The main loop is as follows:

```
while some u has excess[u] > 0:
  // u ≠ source, u ≠ sink
  if residual(u,v) > 0 and
       height[u] == height[v]+1:
    // push
    delta = min(excess[u], residual(u,v))
    flow[u,v] += delta
    flow[v,u] -= delta
    excess[u] -= delta
    excess[v] += delta
  else:
    // relabel
    m = ∞
    for each (u, v) in G[u]:
      if flow[u,v] < capacity[u,v]:
        m = min(m, height[v])
    height[u] = m + 1
return excess[sink]
// residual = capacity - flow
```

The loop runs as long as an active vertex `u` is found. If the push condition holds for some neighbor `v` of `u`, a push is made; if not, `u` is relabeled. The relabel part finds the smallest height among the neighbors in the residual network, that is, among the edges with `flow < capacity`, and writes one more than it. When the loop ends, the excess of the sink is returned; this is the total flow that has reached the sink.

8.7.7 Java: push() and relabel()

In the Java implementation each `Node` object holds its own `height` and `excess` fields and the list of its outgoing edges. The push operation is as follows:

```
boolean push(Node n) {
  for (Edge e : n.edges) {
    if (n.height > e.to.height
        && e.flow != e.cap) {
      int flow = Math.min(
        e.cap - e.flow, n.excess);
      n.excess -= flow;
      e.to.excess += flow;
      e.flow += flow;
      updateReverse(e, flow);
      return true;   // pushed
    }
  }
  return false;
}
```

The method tries the edges of the vertex in order. The expression `n.height > e.to.height` in the condition looks looser than the theoretical condition *h(u) = h(v) + 1*. But because of the invariant the two are the same: on an edge in the residual network *h(u) ≤ h(v) + 1*, so *h(u) > h(v)* is possible only when *h(u) = h(v) + 1*. `e.flow != e.cap` says that the edge is in the residual network.

When the condition holds, the amount pushed is the smaller of the residual capacity and the excess. The vertex's excess decreases, that of the target vertex increases, the flow on the edge grows, and `updateReverse` reduces the flow of the backward edge by the same amount. Backward edges are stored as edges with capacity 0 and negative flow. The method returns `true` after the first successful push. If no edge is suitable it returns `false` and the vertex is relabeled:

```
void relabel(Node n) {
  int minH = Integer.MAX_VALUE;
  for (Edge e : n.edges) {
    // only edges in the residual network
    if (e.flow != e.cap
        && e.to.height < minH) {
      minH = e.to.height;
      n.height = minH + 1;
    }
  }
}
```

The loop finds the lowest among the neighbors in the residual network and sets the height to one more than that. The assignment is made inside the loop, but since the last assignment made corresponds to the smallest height, the result is correct. The main loop combines the two methods: while there is an active vertex, `if (!push(n)) relabel(n);`. The result is the excess of the sink.

A push takes $O(1)$; a relabel takes time proportional to the degree of the vertex. The code scans the edge list from the start on every call. This is a choice made for simplicity; efficient implementations keep a "next edge" pointer similar to the `ptr` array in Dinic's algorithm.

8.7.8 Complexity

The running time is found by counting how many times each type of operation can be performed. Table 8.10 summarizes the results.

Operation	Total count	Reason
Relabel	$O(V^2)$	the height of every vertex is $\le 2\|V\| - 1$ and only increases
Saturating push	$O(V \cdot E)$	for an edge to be saturated again, the heights at both endpoints must rise by ≥ 2
Non-saturating push	$O(V^2 \cdot E)$	via the potential function $\Sigma\, h(v)$ (active v)

Table 8.10: The total number of push–relabel operations.

Relabels. An active vertex has a path to the source in the residual network. This path has at most $|V| - 1$ edges, and the height drops by at most 1 on each edge. Hence $h(u) \le h(s) + |V| - 1 = 2|V| - 1$. Since every relabel raises the height by at least 1, a vertex is relabeled at most $2|V| - 1$ times; in total $O(V^2)$. In our example, for $|V| = 6$ the bound is 11; the highest label reached was 8.

Saturating pushes. When an edge (u, v) is saturated, $h(u) = h(v) + 1$. For it to be saturated again in the same direction, a push must first be made from v to u; for that $h(v)$ must rise to $h(u) + 1$, that is, increase by at least 2. Then u must rise by 2 as well. Since heights cannot exceed $2|V|$, each edge is saturated $O(V)$ times; in total $O(V \cdot E)$.

Non-saturating pushes. The sum of the heights of the active vertices, Φ, is used as a potential. A non-saturating push makes u inactive and decreases Φ by at least 1 (even if v becomes active, $h(v) = h(u) - 1$). Relabels raise Φ by $O(V^2)$ in total, and saturating pushes by at most $O(V \cdot E) \cdot 2V$. Hence the number of non-saturating pushes is $O(V^2 \cdot E)$.

The generic algorithm takes $O(V^2 \cdot E)$ in the worst case. A relabel scans at most as many edges as the degree; the total scan is $O(V \cdot E)$ and does not change the dominant term. The rule for choosing the active vertex improves the bound. If the active vertices are kept in a FIFO (first-in first-out) queue, the time drops to $O(V^3)$. If the active vertex with the highest label is chosen each time, the time is $O(V^2 \cdot \sqrt{E})$.

In practice two heuristics speed the algorithm up a great deal. The **gap** heuristic notices that no vertex is left at some height value. In that case the vertices above that value cannot reach the sink; the heuristic moves them directly above $|V|$. The long "playing catch" stretches of the example are shortened to a single step this way. **Global relabeling** occasionally runs a BFS backward from the sink and sets all heights to the true distances. With these heuristics push–relabel is one of the fastest general-purpose maximum-flow methods.

8.8 Comparison and applications

8.8.1 Comparing the algorithms

When the four algorithms are run on the same network, the differences become concrete. Table 8.11 summarizes the results on the second example network, whose maximum flow is 19.

Algorithm	Basic operation	Time	On the example network
Ford-Fulkerson	any augmenting path (DFS)	$O(E \cdot \lvert f^* \rvert)$	5 augmenting paths (the order in Example 2)
Edmonds-Karp	shortest augmenting path (BFS)	$O(V \cdot E^2)$	4 augmenting paths, 5 BFS runs
Dinic	level graph + blocking flow	$O(V^2 \cdot E)$	2 phases, 3 BFS runs
Push–relabel	local push / relabel	$O(V^2 \cdot E)$, FIFO $O(V^3)$	searches for no paths

Table 8.11: Comparison of the algorithms. The last column is the behavior on the example network with $|f^*| = 19$.

Edmonds-Karp finds these paths in this network: $S \to 1 \to 2 \to T$ (4), $S \to 1 \to 4 \to T$ (6), $S \to 3 \to 4 \to T$ (4), and $S \to 3 \to 4 \to 2 \to T$ (5). The first three paths have three edges, the last has four. The fifth BFS cannot reach the sink. Dinic's first phase processes the first three paths after a single BFS; its second phase finds the fourth path.

A rough rule for choosing can be given. If the capacities are small integers, Ford-Fulkerson is enough and is the shortest code. As a general-purpose implementation Dinic is a good default: its code is short, its bound is good, and it is fast in practice. On very large and dense networks push–relabel,

together with its heuristics, is usually the fastest. All the algorithms find the same value $|f^*|$. The distribution of the flow over the edges may differ. The minimum cut is in every case read from the final residual network.

8.8.2 All the steps on one network

All the concepts introduced in this chapter can be applied together on a small network.

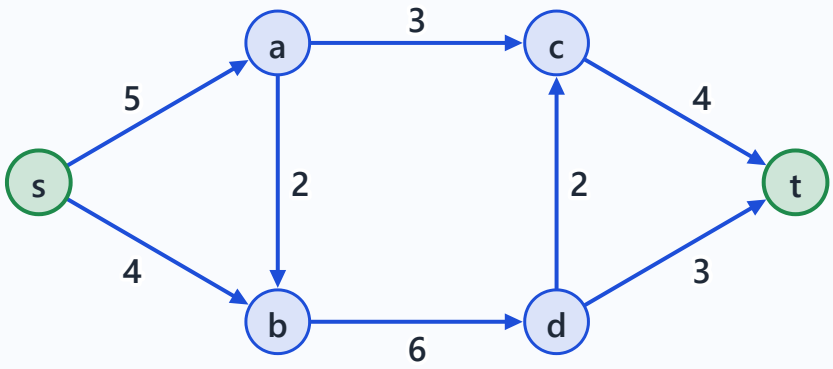


Figure 8.34: The problem network. Capacities: $s \to a$ 5, $s \to b$ 4, $a \to b$ 2, $a \to c$ 3, $b \to d$ 6, $d \to c$ 2, $c \to t$ 4, $d \to t$ 3.

Example 8.6

Upper bound. The capacity leaving the source is $5 + 4 = 9$, and the capacity entering the sink is $4 + 3 = 7$. Hence $|f^*| \le 7$.

Edmonds-Karp. BFS tries neighbors alphabetically. In the first BFS the queue order is a, b, c, d; c is the first to find the sink. The path is $s \to a \to c \to t$, with bottleneck 3 ($a \to c$). In the second BFS $a \to c$ is full; $s \to b \to d \to t$ is found, with bottleneck 3 ($d \to t$). In the third BFS $d \to t$ is full; $s \to b \to d \to c \to t$ is found. The bottleneck is $min(1, 3, 2, 1) = 1$. The first two paths have 3 edges, the last has 4. The total flow is $3 + 3 + 1 = 7$.

Minimum cut. In the final residual network there are 2 units on $s \to a$. $a \to b$ is empty; 2 units are left on $b \to d$ and 1 unit on $d \to c$. Every vertex other than the sink is reachable from the source: $S = \{s, a, b, c, d\}$. The cut edges are $c \to t$ (4) and $d \to t$ (3); the capacity is 7. The flow is maximum.

Residual capacities. $c_f(s, a) = 5 - 3 = 2$. $c \to d$ is not an edge of the original network; since there is 1 unit of flow on $d \to c$, $c_f(c, d) = f(d, c) = 1$ (a backward edge).

Start of push–relabel. There are six vertices: $h(s) = 6$, the others 0. $s \to a$ and $s \to b$ are saturated: $e(a) = 5$, $e(b) = 4$, and the other excesses are 0.

8.8.3 Applications

The power of maximum flow comes from the fact that many problems with no apparent connection to flow can be turned into a flow network. After the transformation, any of the algorithms described in this chapter is used.

Bipartite matching. Consider two groups, such as workers and jobs, or students and projects. Each worker is joined by an edge to the jobs he or she can do. Each worker can take at most one job, and each job at most one worker. The goal is the largest possible number of matched pairs. This problem is turned into a flow network as follows: a source is joined to all the workers, and all the jobs are joined to a sink; the worker–job edges stay as they are. All capacities are 1.

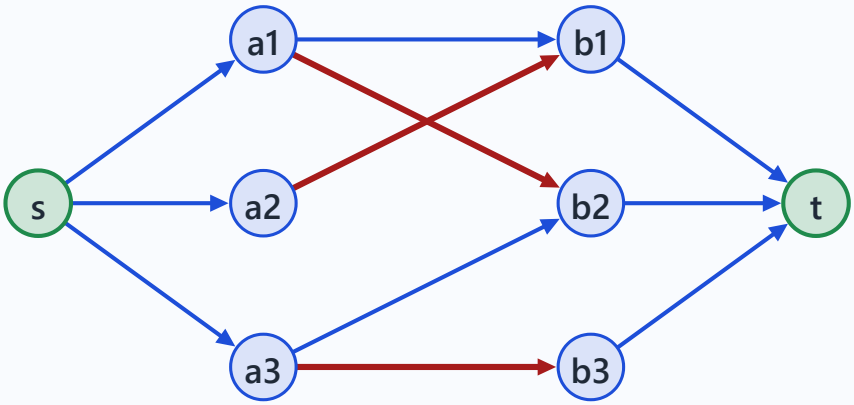


Figure 8.35: A bipartite matching network. All edges have unit capacity. The highlighted edges are a maximum matching: a_1–b_2, a_2–b_1, a_3–b_3.

Since the edge from the source to a worker has capacity 1, at most one unit of flow leaves that worker; a worker is assigned to at most one job. The edge from a job to the sink imposes the same limit on jobs. The integrality theorem guarantees that the maximum flow is 0 or 1 on every edge. The worker–job edges that carry flow form a matching, and the value of the flow is the size of the matching. In the network of Figure 8.35, worker a_1 can do b_1 and b_2, a_2 can do only b_1, and a_3 can do b_2 and b_3. The maximum flow is 3; everyone gets a job.

Edge-disjoint paths. What is the largest number of paths between two vertices that share no edge? This question concerns the robustness of networks: how many links must fail before communication is cut? Every edge is given capacity 1; the maximum flow equals the number of edge-disjoint paths. Menger's theorem says that this number equals the smallest number of edges that must be deleted to separate the two vertices. This is the unit-capacity special case of the max-flow min-cut theorem.

Vertex capacities. In some problems the constraint is on vertices rather than edges: at most a certain number of vehicles can pass through an intersection per hour. Each vertex v is split into two vertices: an input v_{in} and an output v_{out}. An edge $v_{in} \to v_{out}$ with capacity $c(v)$ is placed between them. The edges entering v are attached to v_{in}, and those leaving v are attached to v_{out}. All the flow passing through the vertex is forced to use this single edge.

Image segmentation. To separate foreground from background in a photograph, each pixel becomes a vertex. The source stands for "foreground" and the sink for "background." The probability that a pixel is foreground becomes the capacity of the edge from the source, and the probability that it is background the capacity of the edge to the sink. Edges between neighboring pixels reward keeping similar pixels on the same side. The minimum cut gives the partition that minimizes the total "mislabeling cost."

Network reliability, transportation, and logistics. The smallest total capacity that must be cut to split a communication network in two is a minimum cut. Maximum flow or minimum cut is also a basic tool in pipelines, traffic planning, airline scheduling, and project selection. In most of these problems the real work is building the right network; the algorithm is ready.

8.9 Common mistakes

Most mistakes in network flow problems come from a few confusions of concepts. The list below gathers them.

- **Forgetting backward edges.** Searching with forward edges only is a greedy algorithm and may miss the maximum; on the second example network the result is 18 instead of 19.
- **Computing the bottleneck from capacities.** The bottleneck is the smallest of the residual capacities, not of the capacities: $c - f$ on a forward edge and f on a backward edge. After the second augmentation some edges are partly full, so looking at capacities gives a wrong and too large bottleneck.
- **Taking the minimum cut to be the one with the fewest edges.** A minimum cut cuts the smallest total capacity, not the fewest edges.
- **Counting edges from T to S.** Only edges going from S to T enter the cut capacity; backward ones are not added. In computing the *net flow* across a cut, however, flows in the backward direction are subtracted. The two computations must not be confused.
- **Taking the value of a flow to be the sum of the edge flows.** The value is only the net flow leaving the source; the same unit is not counted again on every edge of its path.
- **Treating Ford-Fulkerson as polynomial.** $O(E \cdot |f^*|)$ depends on the values of the capacities. Edmonds-Karp, which works with BFS, removes this dependence.
- **Measuring "shortest" in Edmonds-Karp by capacity.** The shortest path is the one with the fewest edges. Choosing the path of largest capacity is a separate heuristic, and the bound of Edmonds-Karp does not hold for it.
- **Relaxing the level condition in Dinic's algorithm.** DFS must follow only edges with $level(v) = level(u) + 1$. If edges within the same level are allowed, DFS can enter a cycle and the bound on the number of phases becomes invalid.
- **Mixing up the direction of the push.** Flow is pushed from high to low, only when $h(u) = h(v) + 1$; at the start $h(s) = |V|$ must hold.

- **Reading the minimum cut from the wrong residual network.** The cut is read from the vertices reachable from the source in the final residual network, after the algorithm has finished. A set read from the residual network of an intermediate step may give a cut, but not necessarily a minimum one.

8.10 Summary

- **Flow network:** a directed graph of a source, a sink, and edges with capacities. A flow obeys the capacity constraint ($0 \le f \le c$) and conservation at intermediate vertices (in = out). The value $|f|$ is the net flow leaving the source. Multiple sources and sinks are reduced to a single-source problem with a supersource and a supersink joined by edges of infinite capacity.
- **Residual network:** the residual capacity of a forward edge is $c - f$ and that of a backward edge is f. The backward edge makes it possible to undo flow already sent. An augmenting path is an s–t path in the residual network; the bottleneck is the smallest residual capacity on the path. Augmenting along the path by the bottleneck gives a valid flow and increases its value by the bottleneck.
- **Cut:** $c(S, T)$ is the sum of the capacities of the edges going from S to T. The net flow across every cut equals $|f|$; hence $|f| \le c(S, T)$. The maximum flow equals the minimum cut. The S side of the minimum cut is the set of vertices reachable from the source in the final residual network.
- **Ford-Fulkerson:** uses any augmenting path. With integer capacities it terminates, finds a flow of integer value, and takes $O(E \cdot |f^*|)$. A bad choice of paths can require as many augmentations as the capacity.
- **Edmonds-Karp:** chooses the shortest augmenting path with BFS. Distances do not decrease, and each edge becomes critical $O(V)$ times; $O(V \cdot E)$ augmentations and $O(V \cdot E^2)$ time.
- **Dinic:** in each phase builds a level graph with BFS and finds a blocking flow with DFS and the current-arc array. At most $V - 1$ phases, $O(V \cdot E)$ per phase; in total $O(V^2 \cdot E)$. It is much faster on unit capacities.
- **Push–relabel:** starts with $h(s) = |V|$ and builds a preflow by saturating the source edges. Flow is pushed from high to low; if it cannot be pushed, the vertex is relabeled. When no active vertex remains, the preflow is a maximum flow. The generic version runs in $O(V^2 \cdot E)$, the FIFO version in $O(V^3)$.
- **Applications:** bipartite matching, edge-disjoint paths, networks with vertex capacities, image segmentation, and network reliability are modeled as maximum flow or minimum cut.

8.11 Key terms

flow network source / sink capacity flow conservation maximum flow residual network

residual capacity augmenting path bottleneck cut (s-t cut) max-flow min-cut theorem

Ford-Fulkerson method Edmonds-Karp algorithm Dinic's algorithm level graph blocking flow

push-relabel excess flow bipartite matching

8.12 Test Yourself

1. In the flow of Figure 8.2, which edges and capacities do the edges $B \to C$ (2/3) and $C \to D$ (5/6) produce in the residual network? Is there an augmenting path in the residual network for this flow?

   Answer

   $B \to C$: forward $c_f(B, C) = 3 - 2 = 1$, backward $c_f(C, B) = 2$. $C \to D$: forward $c_f(C, D) = 1$, backward $c_f(D, C) = 5$. There is no augmenting path: the two edges leaving the source are saturated, and no edge leaves S in the residual network.

2. In the second example network (Figure 8.10), what is the capacity of the cut $S = \{S, 1, 3\}$? Why is this cut not a minimum cut?

   Answer

   The edges going from S to T are $1 \to 2$ (4), $1 \to 4$ (8), and $3 \to 4$ (9): $4 + 8 + 9 = 21$. The edges $S \to 1$, $S \to 3$, and $1 \to 3$ have both endpoints on the S side. Since the maximum flow is 19, a cut of capacity 21 cannot be minimum; the minimum cut is $\{S, 3\}$ (19).

3. Which paths does Edmonds-Karp find on the first example network (Figure 8.1)? How many augmentations and how many BFS runs does it make?

   Answer

   $S \to A \to B \to T$ (2), $S \to C \to D \to T$ (3), and $S \to A \to B \to C \to D \to T$ (2). The first two paths have 3 edges, the third 5. There are 3 augmentations; the fourth BFS cannot reach the sink. In total 4 BFS runs, maximum flow 7.

4. Which paths does Ford-Fulkerson with a DFS that tries neighbors in index order ($s, 1, 2, 3, 4, t$) find on the network of Figure 8.17? How many augmentations does Edmonds-Karp make on the same network?

   Answer

   Six augmentations are made: $s \to 1 \to 2 \to 4 \to 3 \to t$ (7), $s \to 1 \to 2 \to 4 \to t$ (3), $s \to 1 \to 3 \to 2 \to 4 \to t$ (1), $s \to 1 \to 3 \to t$ (5), $s \to 2 \to 1 \to 3 \to t$ (6), $s \to 2 \to 3 \to t$ (1). The flow grows as $7, 10, 11, 16, 22, 23$. The edge $2 \to 1$ on the fifth path is an original edge; $2 \to 3$ on the sixth path is the backward edge of $3 \to 2$. Edmonds-Karp reaches 23 on the same network with 3 augmentations.

5. How many phases does Dinic's algorithm make on the first example network? What are the levels and the blocking flow in each phase?

   Answer

   Phase 1: the levels are $S = 0$; $A, C = 1$; $B, D = 2$; $T = 3$. $B \to C$ is not used because it goes back a level. The blocking flow is $S \to A \to B \to T$ (2) and $S \to C \to D \to T$ (3); $|f| = 5$. Phase 2: the levels are $S = 0$, $A = 1$, $B = 2$, $C = 3$, $D = 4$, $T = 5$; the path is $S \to A \to B \to C \to D \to T$ (2), $|f| = 7$. The third BFS cannot reach the sink. 2 phases, 3 BFS runs.

6. In the first example network, if the capacity of the edge $B \to T$ is raised from 2 to 5, does the maximum flow change? In how many phases does Dinic finish?

   Answer

   It does not change; it stays 7. The capacity of the cut *({S}, the rest)* is still $4 + 3 = 7$, and this edge is not in that cut. Dinic finishes in a single phase: in the level graph $S \to A \to B \to T$ carries 4 and $S \to C \to D \to T$ carries 3 units. The second BFS cannot reach the sink.

7. In the first example network, if at most 3 units of flow can pass through vertex B, how is the problem modeled and what is the maximum flow?

   Answer

   B is split into the vertices B_{in} and B_{out}; an edge of capacity 3 is placed between them. The edges become $A \to B_{in}$ (4), $B_{out} \to C$ (3), and $B_{out} \to T$ (2). The maximum flow is 6: for example 3 units through B (2 directly to the sink, 1 to C), and 3 units through $S \to C$. The minimum cut consists of the edges $S \to C$ (3) and $B_{in} \to B_{out}$ (3): 6.

8. What is the largest number of edge-disjoint paths from S to T in the second example network? How is this number reached with maximum flow?

   Answer

   Set all capacities to 1 and compute the maximum flow. The result is 2; for example $S \to 1 \to 2 \to T$ and $S \to 3 \to 4 \to T$. Since only two edges leave the source (the cut {S} has capacity 2), a third edge-disjoint path is impossible.

9. Push–relabel is started on the first example network. What are the initial heights and excesses? If

A is chosen as the active vertex, what are the first two operations?

Answer

$h(S) = 6$, and $h = 0$ at all other vertices. $S \to A$ and $S \to C$ are saturated: $e(A) = 4$, $e(C) = 3$. The residual neighbors of A are B ($h = 0$) and S (a backward edge, $h = 6$); neither is one step below. First a relabel: $h(A) = 1 + \min(0, 6) = 1$. Then a push: $\delta = \min(e(A), c_f(A, B)) = \min(4, 4) = 4$; $e(A) = 0$, $e(B) = 4$.

8.13 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 24: presents flow networks, the proof of the max-flow min-cut theorem, the Edmonds-Karp analysis, and bipartite matching with detailed proofs; the Edmonds-Karp trace network in these notes is from this book.
- J. Kleinberg and É. Tardos, *Algorithm Design* (2006), Chapter 7: treats maximum flow as a modeling tool; presents applications such as image segmentation, project selection, and airline scheduling with worked examples.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 6.4: codes the shortest-path version of Ford-Fulkerson in Java with edge objects and shows the residual network in an object-oriented design.
- A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem" (1988): gives the push–relabel algorithm, its invariants, and the $O(V^3)$ analysis of the FIFO version from the original source.

8.14 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 24: Maximum Flow).
2. E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation", *Soviet Math. Doklady* 11, 1970.
3. J. Edmonds, R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of the ACM* 19(2), 1972.
4. L. R. Ford, D. R. Fulkerson, "Maximal Flow Through a Network", *Canadian Journal of Mathematics* 8, 1956.
5. A. V. Goldberg, R. E. Tarjan, "A New Approach to the Maximum-Flow Problem", *Journal of the ACM* 35(4), 1988.
6. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2006 (Chapter 7: Network Flow).
7. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 6.4: Maximum Flow).

PART III

String Algorithms

Chapters 9–11

String Matching and Compression

Searching for a word with Ctrl+F in a text editor, a virus scanner looking for known signatures, and a gene being located in a DNA sequence are all instances of one problem: find where a short string occurs inside a long text. These notes present four classical solutions. The brute-force algorithm tries every position in turn. The Knuth-Morris-Pratt algorithm stores what it learns from the matched part in a table and never moves backward in the text. The Boyer-Moore algorithm compares the pattern from right to left and skips most of the text without reading it. The Rabin-Karp algorithm compares hash values instead of characters. The second half of the notes is about string compression. Run-length encoding (RLE) writes consecutive repeats as a count. Lempel-Ziv-Welch (LZW) replaces repeated substrings with short codes from a dictionary that grows as it goes. Every algorithm is run step by step on an example, its Java code is traced line by line, and its cost is derived.

LEARNING OUTCOMES

Concepts
Define text, pattern, shift, prefix, and suffix; state the string matching problem formally.

Matching
Run the brute-force, KMP, Boyer-Moore, and Rabin-Karp algorithms by hand on an example and trace their Java code.

Analysis
Explain the time saved by the LPS table, the bad-character table, and the rolling hash; compare best, average, and worst cases; choose an algorithm to fit the problem.

Compression
Compress losslessly with RLE and LZW, decompress the result, handle the special case in LZW decoding, and compute the compression ratio.

9.1 The string matching problem

Most of the data computers process is text: e-mail, messages, social media posts, news articles, program source code, log files. The operation performed most often on such text is search. We find where a word occurs, replace one phrase with another, or measure how similar two texts are. Algorithms that do this kind of work are called **string algorithms**. This chapter has two main topics: finding a pattern in a text (string matching) and storing a text in less space (string compression). This section defines the matching problem and sets up the notation.

9.1.1 Text search is everywhere

String matching is built into everyday software. Ctrl+F in a text editor, Unix's `grep` command, and the indexes of search engines are the best-known examples. There are less visible uses as well. Virus scanners look for the byte sequences (signatures) of known malware inside files. Plagiarism detectors find long passages that two documents share. In bioinformatics, one asks where a gene or a short DNA fragment occurs in a sequence of millions of letters.

Compression is just as common. ZIP archives, GIF and PNG images, PDF documents, and fax machines all rely on string algorithms to store data in fewer bits. Most of these methods are also a kind of search at heart: they find a repeat of something seen earlier and replace it with a short reference.

In these applications the texts are large. A log file can run to gigabytes, and a human genome is about three billion letters long. The number of times an algorithm reads each character therefore determines its running time directly. All the algorithms in this chapter answer the same question. They differ in how many comparisons they need to do so.

9.1.2 Basic concepts and notation

Strings are built from a set of characters called the **alphabet**, written Σ. For lowercase English letters $\Sigma = \{a, b, \ldots, z\}$, for DNA $\Sigma = \{A, C, G, T\}$, and for binary data $\Sigma = \{0, 1\}$. The size of the alphabet is written $\sigma = |\Sigma|$. The speed of some algorithms depends on this number.

The long string being searched is the **text**, and the short string being sought is the **pattern**. We write the text as $T[0..n-1]$ and the pattern as $P[0..m-1]$. So n is the length of the text, m is the length of the pattern, and usually $m \le n$. Indices start at 0, as in Java. In code the text is called `t` and the pattern `p`; `t.charAt(i)` corresponds to $T[i]$.

Definition 9.1

Aligning the pattern so that its first character lies under position s of the text is called a **shift**. If $T[s..s+m-1] = P$, then P **occurs** in T with shift s, and s is a **valid shift**. Since the pattern cannot stick out past the end of the text, the possible shifts are $0 \le s \le n - m$; there are $n - m + 1$ of them in all.

The **string matching** problem is to find all the shifts (or only the first one) at which P occurs in T.

Most of these algorithms are built on the beginning and the end of the pattern. A piece taken from the start of a string is a **prefix**, and a piece taken from the end is a **suffix**. The prefix of P of length k is $P[0..k-1]$, and its suffix of length k is $P[m-k..m-1]$. The string itself is both a prefix and a suffix of itself. A prefix that is not equal to the whole string is a **proper prefix**, and a suffix that is not equal to the whole string is a **proper suffix**.

Example 9.1

Let P = "ababd" ($m = 5$). Its proper prefixes are a, ab, aba, and abab; its proper suffixes are d, bd, abd, and babd. The empty string also counts as both a prefix and a suffix, but it is mostly useless in exercises. This pattern has no nonempty piece that is both a proper prefix and a proper suffix: every suffix ends in d, and no proper prefix does.

If the same pattern is searched for in the text T = "ababcababd" ($n = 10$), the possible shifts are $0, 1, \ldots, 5$. Only at $s = 5$ does $T[5..9]$ = "ababd" = P hold. The only valid shift is 5.

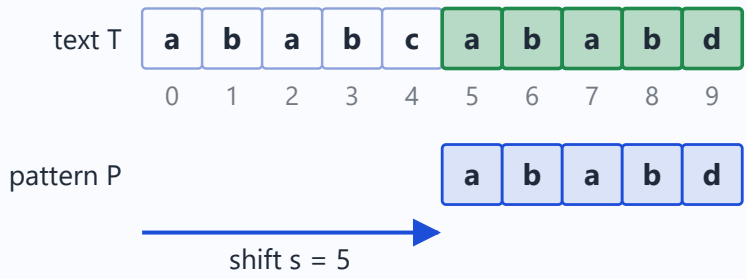


Figure 9.1: The text T = "ababcababd" and the pattern P = "ababd". At shift $s = 5$ the pattern matches the text completely; at every smaller shift at least one character differs.

Comparing two characters has one of two outcomes. If the characters are equal there is a **match**; if they differ there is a **mismatch**. In the figures of these notes, matching characters are drawn in green boxes and mismatching ones in red boxes. The cost of an algorithm is usually measured by the total

number of character comparisons. This is a countable quantity that does not depend on the machine or the language.

9.1.3 A map of the algorithms

The chapter covers six algorithms. The first four are for matching and the last two are for compression:

- **Brute force** (also called naive): every position in the text is checked separately for a match with the pattern. This method can make the most comparisons.
- **Knuth-Morris-Pratt (KMP)**: before searching, a table (the LPS table) is built from the pattern. During the search the algorithm never moves backward in the text; even in the worst case the running time is linear.
- **Boyer-Moore**: comparison goes from right to left. The bad-character and good-suffix rules give long jumps, which makes it very effective on long patterns.
- **Rabin-Karp**: hash values are compared instead of characters. As the window slides, the hash is updated in *O(1)* time; the algorithm is fast in the expected case.
- **Run-length encoding** (RLE): consecutive repeated values are stored as one value and a repeat count.
- **Lempel-Ziv-Welch (LZW)**: dictionary-based compression. Repeated substrings are represented by short codes from a dictionary that grows during compression. The GIF format uses this method.

All the matching algorithms produce the same output. What separates them is what they do at a mismatch. Brute force learns nothing from a mismatch and slides the pattern by one position. KMP uses the structure of the part already matched to skip useless shifts. Boyer-Moore looks at where the mismatched text character occurs in the pattern. Rabin-Karp makes the comparison itself cheaper. These four points of view also explain why the same problem has more than one solution.

9.2 The brute-force algorithm

The brute-force algorithm is the simplest way to search for a pattern in a text. Its other name, "naive," comes from this simplicity. The algorithm does no preprocessing and uses no extra memory. That makes it both the yardstick against which the other algorithms are compared and a practical method still used for short patterns.

9.2.1 How it works

The algorithm tries every possible shift in order, from left to right:

1. **Alignment.** For each shift $s = 0, 1, \ldots, n - m$, the pattern is placed under the text with its first character under $T[s]$.

2. **Comparison.** *P[0]* is compared with *T[s]*, *P[1]* with *T[s+1]*, and so on from left to right. The check of this alignment ends at the first mismatch.
3. **Decision.** If all *m* characters match, *s* is reported. Otherwise the pattern is shifted one position to the right and the next shift is tried.

The number of comparisons depends on the text and the pattern. In the worst case all *m* characters are compared at every alignment: $(n - m + 1) \cdot m$ comparisons, that is, $O(n \cdot m)$. In the best case the first character mismatches at every alignment: $n - m + 1$ comparisons, that is, $O(n)$.

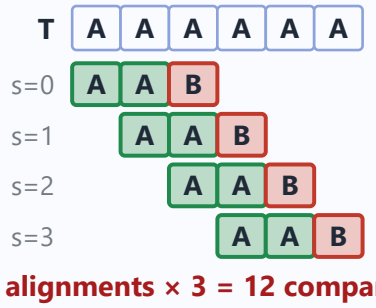


Figure 9.2: The worst case of brute force. When searching for "AAB" in "AAAAAA", the first two characters match at every alignment and the mismatch always comes at the last character: 4 alignments × 3 = 12 comparisons.

9.2.2 Example: searching for SIMPLE

Figure 9.3 shows the search for the pattern "SIMPLE" ($m = 6$) in the text "THIS IS A SIMPLE EXAMPLE" ($n = 24$). Each row is one shift. The space character is shown with the symbol ␣.

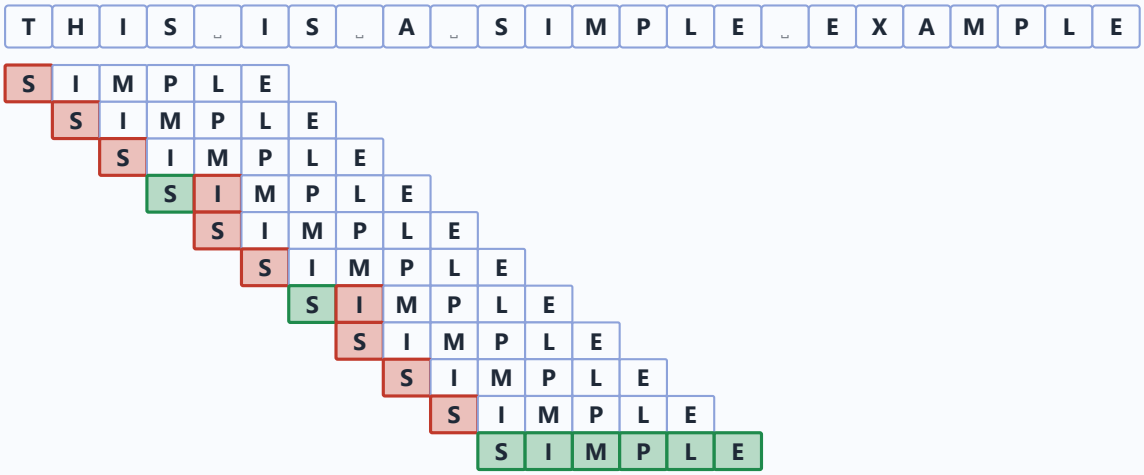


Figure 9.3: Brute-force search for SIMPLE. Each row is one shift; green boxes are matching comparisons and red boxes are mismatches. The pattern is found at the 11th alignment ($s = 10$).

The algorithm starts at *s = 0*. At shifts *s = 0, 1, …, 9*, either the first character S of the pattern mismatches the text character, or only the S matches. At *s = 3*, *T[3]* = S matches, and then *T[4]* = ␣ mismatches I. The same happens at *s = 6*: S matches and ␣ mismatches. All other shifts end at the first comparison. At *s = 10*, all six characters match. Table 9.1 counts the comparisons shift by shift.

Shift *s*	Aligned part of the text	Result	Comparisons
0, 1, 2	THI…, HIS…, IS␣…	first character ≠ S	3 × 1 = 3
3	S␣IS␣A	S matches, ␣ ≠ I	2
4, 5	␣IS…, IS␣…	first character ≠ S	2 × 1 = 2
6	S␣A␣SI	S matches, ␣ ≠ I	2
7, 8, 9	␣A…, A␣…, ␣SI…	first character ≠ S	3 × 1 = 3
10	SIMPLE	6 characters match: found	6
Total (11 alignments)			18

Table 9.1: Shifts and comparison counts in the search for SIMPLE.

The total of 18 comparisons is close to the length of the text (24). The reason is that in natural-language text two random characters are usually different, so a mismatch typically occurs at the first character. This observation is why brute force often runs at acceptable speed in practice.

9.2.3 Java code and trace

The Java code of the algorithm consists of two nested loops. The outer loop runs over the shifts and the inner loop walks over the characters of one alignment. The method returns the position of the first occurrence, or −1 if the pattern does not occur.

```
int bruteForce(String t, String p) {
  int n = t.length(), m = p.length();
  for (int i = 0; i <= n - m; i++) {
    int j = 0;
    while (j < m &&
      t.charAt(i + j) == p.charAt(j))
      j++;
    if (j == m) return i;
  }
  return -1;
}
```

The code reads line by line as follows:

- **Outer loop:** `i` is the shift and runs from 0 to `n - m`. The condition is `i <= n - m` because the last valid shift is *n − m*. The value `i + j` is at most *n − m + m − 1 = n − 1*, so the code never reads outside the string.

- **Inner loop:** `j` is the number of characters matched at this alignment. It increases as long as `j < m` and the characters are equal. The order of the two conditions matters: the `&&` operator evaluates the second condition only if the first is true. So when `j == m`, `p.charAt(m)` is never called.
- **Decision:** if the inner loop ended with `j == m`, every character matched and `i` is returned. If it ended earlier, there was a mismatch and the outer loop moves on to the next shift.

Let us trace the code with the call `bruteForce("HELLO WORLD", "LO")`. Here *n = 11* and *m = 2*, and the shifts are *i = 0, 1, ..., 9*. Table 9.2 lists every comparison in order.

Step	`i`	`j`	Comparison	Result
1	0	0	`t[0]` = 'H' with `p[0]` = 'L'	mismatch; pattern shifts by 1
2	1	0	`t[1]` = 'E' with `p[0]` = 'L'	mismatch; pattern shifts by 1
3	2	0	`t[2]` = 'L' with `p[0]` = 'L'	match; `j = 1`
4	2	1	`t[3]` = 'L' with `p[1]` = 'O'	mismatch; pattern shifts by 1
5	3	0	`t[3]` = 'L' with `p[0]` = 'L'	match; `j = 1`
6	3	1	`t[4]` = 'O' with `p[1]` = 'O'	match; `j = 2 = m`
7	3	2	—	`j == m`: returns 3

Table 9.2: Trace of `bruteForce("HELLO WORLD", "LO")`.

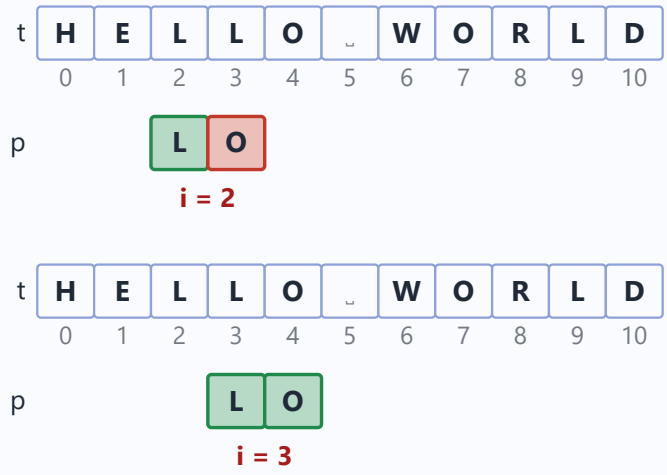


Figure 9.4: Two moments of the trace. On the left, *i = 2*: L has matched and at the second character L and O mismatch. On the right, *i = 3*: both characters have matched and the pattern is found at position 3.

In all, 6 comparisons are made. Look at steps 4 and 5: *T[3]* = 'L' is read twice. The first time it is compared with the second character of the pattern, and then with the first. This is the basic flaw of brute force: when the pattern shifts, the algorithm goes back in the text and reads the same characters again.

The memory cost is $O(1)$; only `i`, `j`, `n`, and `m` are stored. If all occurrences are wanted, `i` is added to a list instead of executing `return i`, and the loop continues; the complexity does not change.

Common mistake

Writing the outer loop condition as `i < n - m`. This skips the last shift ($s = n - m$). For example, in the call `bruteForce("kalem", "em")` the correct shifts are 0, 1, 2, 3. The faulty condition tries only 0, 1, 2. Since "em" occurs only at $s = 3$, it is not found and the method returns −1. If the pattern is not at the end of the text the bug stays invisible, so it easily escapes testing.

9.2.4 Best, average, and worst case

The number of comparisons in brute force depends on how many characters match at each alignment. Table 9.3 summarizes the three cases.

Case	When?	Comparisons	Time
Best	$P[0]$ matches no character of the text (e.g. P = "xyz", T = "aaaa…")	$n - m + 1$	$O(n)$
Average	Random text, large alphabet: the mismatch usually comes within the first character or two	$\approx (n - m + 1) \cdot \sigma/(\sigma - 1)$	$O(n)$
Worst	T = "aaaaaaaaaa", P = "aaab": m comparisons at every alignment	$(n - m + 1) \cdot m$	$O(n \cdot m)$

Table 9.3: Best, average, and worst case of brute force.

The factor $\sigma/(\sigma - 1)$ in the average case comes from a simple probability calculation. Suppose the characters of the text are chosen independently and uniformly from the alphabet. At each alignment the first comparison is always made. The second is made only if the first character matches, which has probability $1/\sigma$. The third requires the first two to match: probability $1/\sigma^2$. The expected number of comparisons at one alignment is therefore at most $1 + 1/\sigma + 1/\sigma^2 + \ldots = \sigma/(\sigma - 1)$. For English letters ($\sigma = 26$) this is about 1.04; for the binary alphabet ($\sigma = 2$) it is 2. In both cases the work per alignment is constant and the total time is $O(n)$.

Example 9.2

Find the number of comparisons brute force makes for T = "aaaaaaaaaa" ($n = 10$) and P = "aaab" ($m = 4$).

1. **Number of alignments.** The shifts are $0, 1, \ldots, 6$: $n - m + 1 = 7$ alignments.
2. **Work per alignment.** At every alignment the first three characters "aaa" of the pattern match the a's in the text. The fourth character, b, mismatches the a in the text. Each alignment costs 4 comparisons.

3. **Total.** $7 \times 4 = 28$ comparisons. The mismatch always comes at the last character and the pattern does not occur in the text.

The same structure gives $9 \times 4 = 36$ comparisons for $n = 12$, $m = 4$. The general formula is $(n - m + 1) \cdot m$. With $m = n/2$ this is about $n^2/4$: the running time grows with the square of the text length.

It is worth examining carefully what happens in the worst case. At $s = 0$ the first three characters have matched. This tells us that $T[0..2]$ = "aaa". When the algorithm moves on to $s = 1$ it reads $T[1]$ and $T[2]$ again, although these characters were already known. This is the real flaw of brute force: the information learned at a mismatch is thrown away, the algorithm goes back in the text, and the same characters are compared again. The KMP algorithm stores and uses exactly this information.

9.3 The Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt (KMP) algorithm is the first linear-time matching algorithm; it keeps the information that brute force throws away. James H. Morris and Vaughan Pratt discovered it together in 1970. Donald Knuth reached the same idea independently, from automata theory. The three published their results in a joint paper in 1977. The name of the algorithm comes from the surnames of the three.

9.3.1 Not forgetting the matched part

The basic idea of KMP is this: at a mismatch, use what is already known about the part of the pattern that has matched. Suppose that at some alignment $P[0..q-1]$ has matched the text and $P[q]$ has not. Then the last q characters of the text read are exactly $P[0..q-1]$. There is no need to read them again; we know what they are by looking at the pattern itself.

This knowledge gains two things. First, the text pointer never moves backward. The text is read once, from left to right. Second, alignments that cannot possibly match are never tried. The pattern is shifted directly to the farthest position that the structure of the matched part allows.

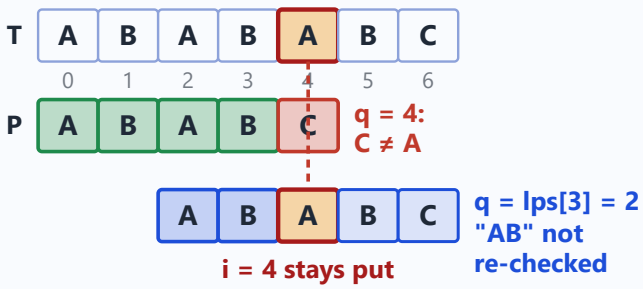


Figure 9.5: Searching for P = "ABABC" in the text T = "ABABABC". "ABAB" has matched ($q = 4$), and C mismatches A. Because the "AB" at the end of the matched "ABAB" is also a prefix, the pattern resumes from

position $q = lps[3] = 2$. "AB" is not checked again and the text pointer stays at $i = 4$.

Figure 9.5 shows the idea on a small example. Let the text be "ABABABC" and the pattern "ABABC". At $s = 0$ the first four characters "ABAB" match, and $T[4]$ = A mismatches $P[4]$ = C. Brute force would shift the pattern by one and restart from $T[1]$. But we know that the last four characters of the text are "ABAB". The suffix "AB" of "ABAB" is also a prefix of the pattern. So if the pattern is shifted by two positions, its leading "AB" already matches the "AB" in the text. The comparison continues between $P[2]$ and $T[4]$. Indeed, at this alignment the rest of the pattern, "ABC", matches too, and the pattern is found at $s = 2$.

How far to shift depends only on the pattern, not on the text. So it can be computed in advance for every possible q and written into a table. This table is called the **LPS table** or the **prefix function**, and sources also denote it by π.

9.3.2 The LPS table

LPS stands for "longest proper prefix which is also a suffix." Each cell of the table is the length of the longest piece of a prefix of the pattern that resembles the prefix itself.

Definition 9.2

For $0 \le q < m$, $lps[q]$ is the length of the longest piece of the string $P[0..q]$ that is both a proper prefix and a suffix of it. If there is no such nonempty piece, $lps[q] = 0$. Since the proper prefix of a single character is the empty string, $lps[0] = 0$ always.

The word "proper" in the definition matters. The string itself is also both a prefix and a suffix of itself. If the definition did not exclude it, every cell would be $q + 1$ and the table would carry no information. The properness condition also yields $lps[q] \le q$. This inequality is used in the proof that the algorithm makes progress at every fallback.

Example 9.3

Let the text be "ababcababcababababd" and the pattern "ababd". The LPS table of the pattern is given cell by cell, with reasons, in Table 9.4.

q	$P[q]$	$lps[q]$	Reason ($P[0..q]$)
0	a	0	"a": no proper prefix
1	b	0	"ab": a ≠ b
2	a	1	"aba": "a" is both prefix and suffix
3	b	2	"abab": "ab" is both prefix and suffix

q	$P[q]$	$lps[q]$	Reason ($P[0..q]$)
4	d	0	"ababd": no prefix ends in d

Table 9.4: The LPS table of the pattern "ababd".

The result is $lps = [0, 0, 1, 2, 0]$. In the row $q = 3$, "aba" may look like a candidate. But "aba" is not a suffix of "abab"; the three-letter suffix of "abab" is "bab". So the value is 2, not 3.

The meaning of the table shows up in the search. Suppose the first q characters of the pattern have matched and the $(q + 1)$th has not. The matched part is $P[0..q-1]$, and the longest piece of it that resembles its own prefix has length $lps[q - 1]$. The pattern is shifted so that this piece lies under the same piece in the text. The shift is $q - lps[q - 1]$, and the new pattern position from which comparison continues is $lps[q - 1]$.

Theorem 9.1 (Safety of the KMP shift)

Suppose $P[0..q-1]$ has matched the text at shift s and $P[q]$ has mismatched. Then the pattern cannot match the text at any shift strictly between s and $s + q - lps[q - 1]$.

Proof. **Proof.** Suppose the pattern could match at some shift s' with $s < s' < s + q$. At this alignment the first $k = s + q - s'$ characters of the pattern lie under the part $T[s'..s+q-1]$ of the text. This part is the last k characters of the matched $P[0..q-1]$. For a match, $P[0..k-1]$ would have to equal this part. That is, $P[0..k-1]$ would be both a proper prefix and a suffix of $P[0..q-1]$ ($0 < k < q$). By definition such a piece has length at most $lps[q - 1]$. Hence $k \le lps[q - 1]$, that is, $s' \ge s + q - lps[q - 1]$. Smaller shifts violate this condition, so they cannot match. ■

The theorem says that KMP never misses an occurrence. The skipped shifts contain a definite contradiction between the pattern and the text. The correctness of the algorithm rests on this observation.

9.3.3 Building the LPS table

The LPS table can be computed straight from the definition: for each q, try all prefix lengths and compare with the suffix. This takes $O(m^3)$ time. KMP's way of building the table is cleverer: the table is built by searching the pattern within itself, that is, by KMP itself. The total time is $O(m)$.

```
int[] buildLps(String p) {
  int m = p.length(), k = 0;
  int[] lps = new int[m];
  for (int q = 1; q < m; q++) {
    while (k > 0 &&
      p.charAt(k) != p.charAt(q))
      k = lps[k - 1];
    if (p.charAt(k) == p.charAt(q))
      k++;
    lps[q] = k;
  }
  return lps;
}
```

In the code, `k` is the length of the prefix matched so far. At each step of the loop `q` increases by one and the character `p[q]` is processed:

- **Fallback:** if `k > 0` and `p[k]` differs from `p[q]`, the prefix of length `k` cannot be extended. The next candidate is the longest piece of this prefix that resembles its own prefix: `k = lps[k - 1]`. This step is the same as the mismatch step in the search.
- **Extension:** if `p[k] == p[q]`, the matched prefix grows by one character and `k` increases by one.
- **Writing:** `k` is the length of the longest proper prefix-suffix of *P[0..q]*, and it is written to `lps[q]`.

Here is why the fallback is `lps[k - 1]`. The prefix-suffix pieces of *P[0..q]* must be tried in decreasing order of length. Each of them is a prefix-suffix piece of *P[0..q−1]* extended by one character. The prefix-suffix pieces of *P[0..q−1]* form a chain too. The longest is *k*, the next is *lps[k − 1]*, the next *lps[lps[k − 1] − 1]*, and so on. Each length in the chain is the longest prefix-suffix piece inside the previous piece. The algorithm follows the chain from large to small and stops at the first candidate that can be extended.

Example 9.4

Trace the call `buildLps("ACABACACD")`. The pattern's indices are 0–8. Table 9.5 gives every comparison and the changes in `k`.

`q`	`k` (before)	Comparison	Action	`lps[q]`
0	—	—	always 0	0
1	0	`p[0]` = A, `p[1]` = C	different; `k = 0` stays	0
2	0	`p[0]` = A, `p[2]` = A	same; `k = 1`	1
3	1	`p[1]` = C, `p[3]` = B	different; fall back: `k = lps[0] = 0`	
3	0	`p[0]` = A, `p[3]` = B	different; `k = 0` stays	0
4	0	`p[0]` = A, `p[4]` = A	same; `k = 1`	1
5	1	`p[1]` = C, `p[5]` = C	same; `k = 2` ("AC")	2

`q`	`k` (before)	Comparison	Action	`lps[q]`
6	2	`p[2]` = A, `p[6]` = A	same; `k = 3` ("ACA")	3
7	3	`p[3]` = B, `p[7]` = C	different; fall back: `k = lps[2] = 1`	
7	1	`p[1]` = C, `p[7]` = C	same; `k = 2` ("AC")	2
8	2	`p[2]` = A, `p[8]` = D	different; fall back: `k = lps[1] = 0`	
8	0	`p[0]` = A, `p[8]` = D	different; `k = 0` stays	0

Table 9.5: Trace of `buildLps("ACABACACD")`. Each row is one comparison.

The result is *lps = [0, 0, 1, 0, 1, 2, 3, 2, 0]*. The most instructive row is *q = 7*. The longest prefix-suffix piece of "ACABACA" is "ACA" (*k = 3*). The character after "ACA", *P[3]* = B, mismatches *P[7]* = C; "ACAB" and "ACAC" differ. The next candidate is the longest prefix-suffix piece inside "ACA", which is "A": *lps[2] = 1*. The character after that "A", *P[1]* = C, matches *P[7]*, and "AC" is found. Three fallbacks were made in all.

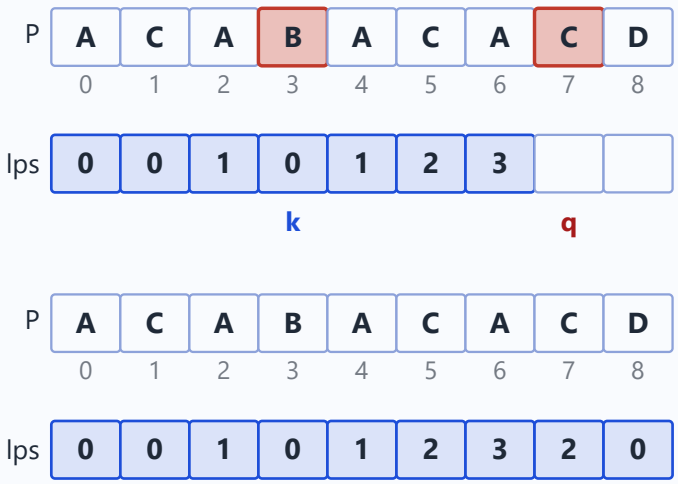


Figure 9.6: On the left, the moment *q = 7*, *k = 3*: B after "ACA" mismatches C; *k* will fall back to *lps[2] = 1*. On the right, the completed table: *[0, 0, 1, 0, 1, 2, 3, 2, 0]*.

The construction takes *O(m)* time. The reasoning is the same as in the linearity proof for the search: `k` increases by at most one per step, so the total increase cannot exceed *m − 1*. Every fallback decreases `k` by at least one, and `k` cannot be negative. So the total number of fallbacks cannot exceed *m − 1* either. In the example there are 3 fallbacks against 8 opportunities to increase. The table needs *O(m)* extra memory.

Common mistake

Computing LPS while counting the string itself. For "aaaa" the correct table is *[0, 1, 2, 3]*, not *[1, 2, 3, 4]*. For example, the longest proper prefix-suffix piece of "aaaa" is "aaa", of length 3. If the table is built wrongly, the fallback never shrinks q and the search loops forever.

9.3.4 The three phases of the search

The KMP search works with two pointers. i is the position in the text and q is the position in the pattern; q is also the number of characters matched so far. The behavior of the algorithm can be divided into three phases:

1. **Looking for the first character.** Here $q = 0$. The pointer i advances until a text character matching the first character of the pattern is found.
2. **Advancing together.** As long as characters match, both pointers advance. If $q = m$, the end of the pattern has been reached and the position $i - m + 1$ is returned.
3. **Falling back with LPS.** At a mismatch only the pattern pointer falls back: $q = lps[q - 1]$. The text pointer i stays where it is. When a match is found, we return to phase 2; if q drops to 0, we return to phase 1.

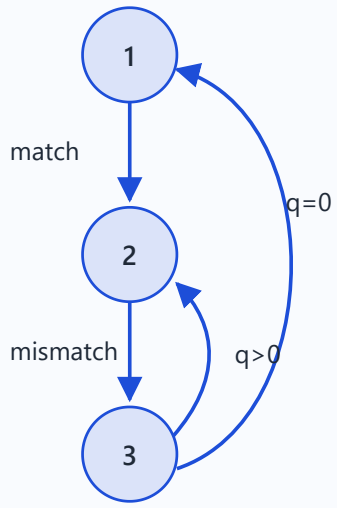


Figure 9.7: Transitions between the phases of the KMP search. A match takes us from 1 to 2; a mismatch takes us from 2 to 3. From phase 3, we return to 2 if a match is found while $q > 0$, and to 1 if q drops to 0.

In none of the phases does i decrease. This is the most important difference from brute force. Brute force pulls the text pointer back to s at every new alignment. In KMP the text is read like a stream: each character arrives once, is processed, and is never requested again.

9.3.5 Example: searching for ABCDABD

Let the pattern be "ABCDABD" and the text "ABCABCDABABCDABCDABDE" (*m = 7*, *n = 21*). The LPS table of the pattern is *[0, 0, 0, 0, 1, 2, 0]*: the "A" at the end of "ABCDA" and the "AB" at the end of "ABCDAB" are also prefixes of the pattern. Figure 9.8 shows the five alignments of the search.

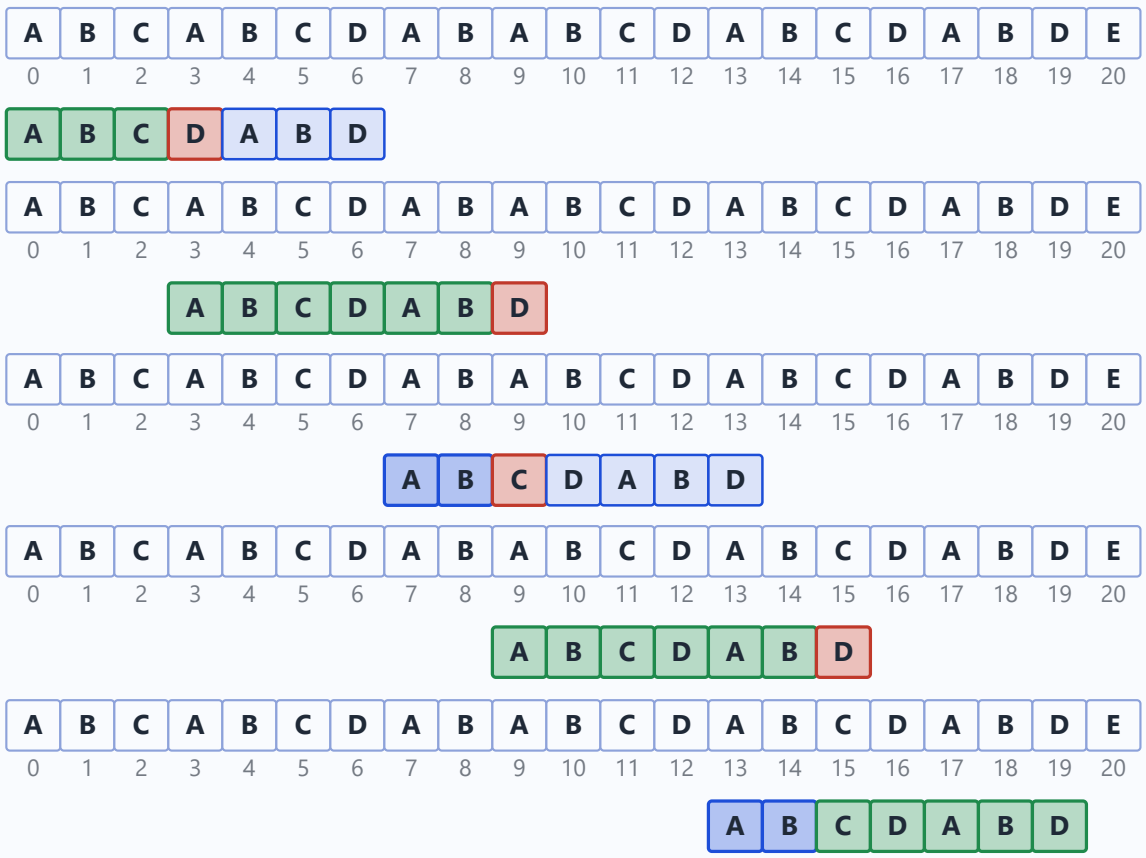


Figure 9.8: Searching for "ABCDABD" with KMP, alignments *s = 0, 3, 7, 9, 13* from top to bottom. Green is matching, red is mismatching, and blue marks characters that are known to match thanks to LPS and are not compared again.

1. ***s = 0.*** "ABC" matches; *T[3]* = A mismatches *P[3]* = D. Here *q = 3*, and *q = lps[2] = 0*. The pattern shifts by 3; this time *T[3]* is compared with *P[0]*.
2. ***s = 3.*** "ABCDAB" matches; *T[9]* = A mismatches D. Here *q = 6*, and *q = lps[5] = 2*. Because the "AB" at the end of the matched "ABCDAB" is a prefix, the pattern shifts by 4.
3. ***s = 7.*** "AB" is known to match and is not compared again. *T[9]* = A mismatches *P[2]* = C; *q = lps[1] = 0*.
4. ***s = 9.*** *T[9]* = A matches *P[0]*. "ABCDAB" matches; *T[15]* = C mismatches D. *q = 6 → lps[5] = 2*; the pattern shifts by 4.
5. ***s = 13.*** "AB" is known; then "CDABD" matches as well. The pattern is found at position 13.

During the search the text pointer never moves backward. The total number of character comparisons is 24. Twenty characters of the text are read once each. At the three mismatch points (*i = 3, 9, 15*) the same text character is compared once more with a different character of the pattern. At *i = 9* this happens twice. Brute force tries 14 alignments on the same text and makes 37 comparisons.

9.3.6 Java code and trace

The Java code of the search is almost identical to the LPS construction code. The only difference is that the comparison is with the text rather than with the pattern itself.

```
int kmpSearch(String t, String p) {
  int n = t.length(), m = p.length();
  int[] lps = buildLps(p);
  int q = 0;  // matched length
  for (int i = 0; i < n; i++) {
    while (q > 0 &&
      p.charAt(q) != t.charAt(i))
      q = lps[q - 1];
    if (p.charAt(q) == t.charAt(i))
      q++;
    if (q == m)
      return i - m + 1;
  }
  return -1;
}
```

- **Setup:** the LPS table is built (*O(m)*) and `q = 0` is set. `q` is the length of the matched prefix.
- **Outer loop:** `i` walks through the text once, from left to right. No line in the code decreases `i`.
- **The `while` loop:** as long as there is a mismatch, `q` falls back along the LPS chain. The loop stops when `q == 0`; there is nowhere further back to go.
- **The `if` (match):** if `p[q]` equals `t[i]`, the matched prefix grows by one.
- **`if (q == m)`:** the whole pattern has matched. Since the last matched character is at `i`, the start is `i - m + 1`.

We trace the code with the call `kmpSearch("ababcababcababababd", "ababd")`. Here *n = 17*, *m = 5*, and *lps = [0, 0, 1, 2, 0]*. Table 9.6 gives the change in `q` for each text character.

`i`	`t[i]`	`q` (before)	Fallback	Match	`q` (after)
0–3	a b a b	0 → 3	—	each matches	4
4	c	4	d ≠ c: `q = lps[3] = 2`; a ≠ c: `q = lps[1] = 0`	`p[0]` = a ≠ c	0
5–8	a b a b	0 → 3	—	each matches	4
9	c	4	d ≠ c: `q = 2`; a ≠ c: `q = 0`	`p[0]` = a ≠ c	0
10–13	a b a b	0 → 3	—	each matches	4

`i`	`t[i]`	`q` (before)	Fallback	Match	`q` (after)
14	a	4	d ≠ a: `q = lps[3] = 2`	`p[2]` = a = a	3
15	b	3	—	`p[3]` = b = b	4
16	d	4	—	`p[4]` = d = d	5 = m

Table 9.6: Trace of `kmpSearch("ababcababcababd", "ababd")`. The "Fallback" column shows what the `while` loop does.

At *i = 16*, `q == m` holds and *16 − 5 + 1 = 12* is returned. The text has been scanned only once. The row *i = 14* is the step that best shows the gain of KMP. At the moment of the mismatch the last four characters of the text are "abab". The "ab" at the end of it is a prefix of the pattern; with *q = 2* the pattern continues from two positions ahead, and *T[14]* = a matches at once. At this point brute force would have gone back to *T[11]*.

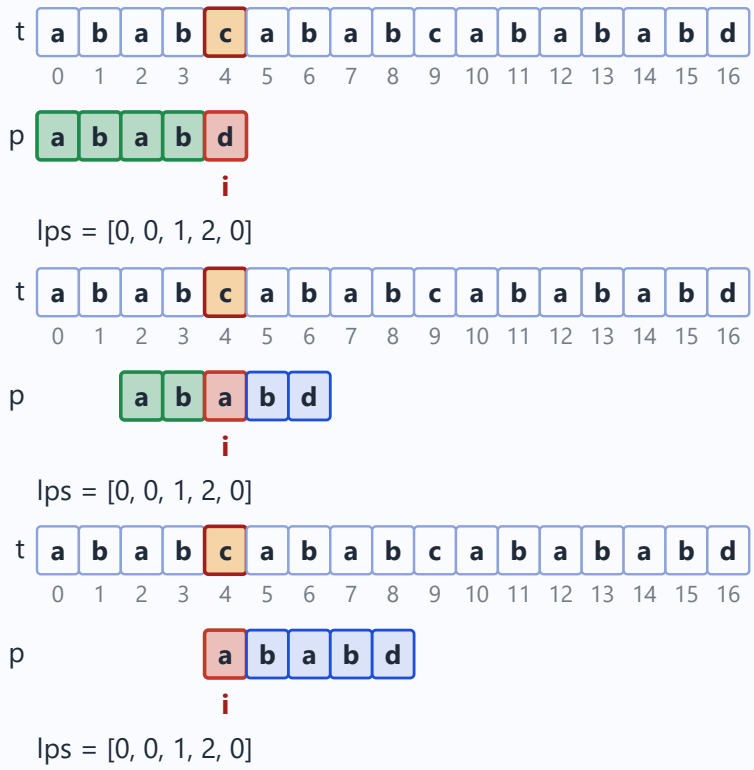


Figure 9.9: The fallback chain at *i = 4*. From top to bottom: at *q = 4*, d ≠ c and *q* drops to 2; at *q = 2*, a ≠ c and *q* drops to 0; at *q = 0* a ≠ c again and *i* advances. Through these three steps the text pointer stays in place.

This trace makes 22 character comparisons in all: each of the 17 text characters is read at least once, and the mismatch points are read several times. On the same search, brute force tries 13 alignments and makes 33 comparisons.

9.3.7 Why O(n + m)?

The KMP code has two nested loops. At first glance this looks like $O(n \cdot m)$ time. In fact, the total number of iterations of the `while` loop is bounded. The kind of analysis in which operations are counted in total rather than one by one is called **amortized analysis**.

> **Theorem 9.2 (Linearity of KMP)**
>
> The KMP search makes at most $2n$ character comparisons and runs in $O(n)$ time. Together with building the LPS table, the total time is $O(n + m)$ and the extra memory is $O(m)$. These bounds hold in the worst case as well.

Proof. **Proof.** Follow the value of q. In each iteration of the outer loop, q increases at most once, by one (`q++`). Since the outer loop runs n times, the total increase is at most n. Each iteration of the `while` loop lowers q to $lps[q - 1]$. Because $lps[q - 1] \le q - 1$, each iteration decreases q by at least one. q starts at 0 and never becomes negative. So the total decrease cannot exceed the total increase. Consequently the `while` loop runs at most n times over the whole search.

Each `while` iteration makes one comparison. To these we add the at most one comparison in the `if` for each i. The total is at most $n + n = 2n$. The same argument gives $O(m)$ for `buildLps` . ■

The idea in the proof can also be told as a bank-account analogy. Each match deposits one lira into the account, and each fallback withdraws one lira. The account can never go negative. So the total withdrawn cannot exceed the total deposited. For a single i the `while` loop can run many times; in the row $i = 4$ of Table 9.6 it runs twice. But this is possible only if the same amount of matching was accumulated in earlier steps.

> **Common mistake**
>
> Using `lps[q]` instead of `lps[q - 1]` at a mismatch. If `q` characters have matched, the matched part is $P[0..q-1]$ and its table value is `lps[q - 1]` . For example, the table for the pattern "abcabd" is $[0, 0, 0, 1, 2, 0]$. When "abcab" has matched ($q = 5$), the correct value is $lps[4] = 2$ and the pattern shifts by 3. If `lps[5] = 0` is used, the pattern shifts by 5; if an occurrence lies among the alignments this shift skips, it is missed.

9.3.8 Finding all occurrences; pros and cons

The `kmpSearch` above stops at the first occurrence. If all occurrences are wanted, the position is added to a list when `q == m` and the search continues. Before continuing, q is set back to $lps[m - 1]$. This step follows the same logic as the mismatch step: the whole pattern has matched, and the next possible occurrence starts from the longest prefix-suffix piece of the matched pattern. In this way overlapping occurrences are found too.

```
List<Integer> kmpAll(String t, String p) {
  int n = t.length(), m = p.length();
  int[] lps = buildLps(p);
  List<Integer> positions = new ArrayList<>();
  int q = 0;
  for (int i = 0; i < n; i++) {
    while (q > 0 &&
      p.charAt(q) != t.charAt(i))
      q = lps[q - 1];
    if (p.charAt(q) == t.charAt(i))
      q++;
    if (q == m) {
      positions.add(i - m + 1);
      q = lps[q - 1];  // overlapping hits
    }
  }
  return positions;
}
```

The code has been run in Java. The call `kmpAll("abababa", "aba")` returns *[0, 2, 4]*. The occurrences overlap: the last a of the "aba" at 0 is the first a of the "aba" at 2. Since the LPS table of "aba" is *[0, 0, 1]*, after every hit $q = 1$ and that single a is not read again. `kmpAll("aaaaa", "aa")` gives *[0, 1, 2, 3]*. If the line $q = lps[q - 1]$ is forgotten, `q` exceeds m on the next turn and `p.charAt(q)` reads outside the string.

The strengths and weaknesses of KMP can be summarized as follows:

- **Linear even in the worst case.** The time is $O(n + m)$ regardless of the structure of the input. The worst input for brute force, T = a…a, P = a…ab, is no problem for KMP.
- **Never moves backward in the text.** This matters for **stream** data, where the text arrives piece by piece from a network or a disk. Earlier parts of the text need not be kept in memory.
- **Usually slower than Boyer-Moore in practice.** KMP reads every character of the text at least once. Boyer-Moore skips many characters without reading them.
- **The gain is limited on small alphabets.** On four-letter alphabets such as DNA, LPS values can be large and the shifts stay short. The bound is still linear, but the constant factor grows.

9.4 The Boyer-Moore algorithm

With the algorithm they published in 1977, Robert S. Boyer and J Strother Moore showed a surprising result: to find a pattern, it is not necessary to read every character of the text. KMP looks at every character of the text at least once. Boyer-Moore, on natural-language text, skips most characters without reading them. The search in text editors and in tools like `grep` is based on this family of algorithms.

9.4.1 Comparing from right to left

The first difference of the algorithm is the direction of comparison. As in brute force, the pattern is slid along the text from left to right. But the characters at one alignment are compared from right to left, starting at the last character of the pattern. The reason for this choice is that a mismatch gives the most information at the right end. If the text character at the right end does not occur in the pattern at all, no alignment covering that character can match. The pattern can be shifted completely beyond this character, that is, by m positions.

The algorithm works in four steps:

1. **Preprocessing.** Shift tables are built from the pattern: the last occurrence of each character in the pattern (the bad-character table) and the shift amount for each suffix (the good-suffix table).
2. **Search.** The pattern is aligned under the text. Comparison starts at the last character of the pattern and moves left.
3. **Match check.** If all characters match, the position is reported.
4. **Shifting.** At a mismatch, two rules each propose a shift: the bad-character rule and the good-suffix rule. The pattern is shifted by the larger of the two.

Both rules are **safe**: the shifts they propose never skip an alignment at which a match could occur. So taking the larger of the two is safe too.

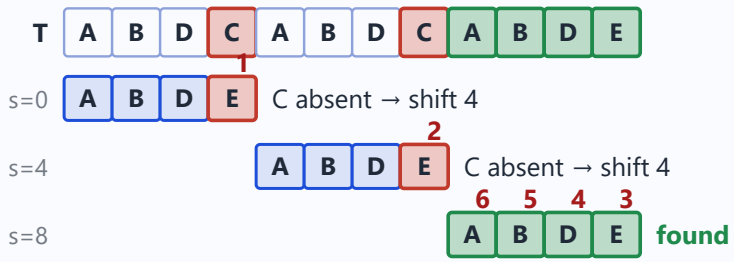


Figure 9.10: Searching for "ABDE" in the text "ABDCABDCABDE". The red numbers are the order of comparison. At the first two alignments the C at the right end is not in the pattern, and the pattern jumps 4 positions. At the third alignment four characters match from right to left: only 6 comparisons in a text of 12 characters.

9.4.2 The bad-character rule

Suppose that while comparing from right to left at some alignment a mismatch occurs at position j: $P[j] \neq T[s + j]$. The mismatched text character $c = T[s + j]$ is called the **bad character**. The rule shifts the pattern according to where c occurs in the pattern.

Definition 9.3

The **last-occurrence table** holds, for each character c, the value $last[c]$: the rightmost position of c in the pattern, or -1 if it does not occur. The **bad-character rule** advances the shift by $s \leftarrow s + max(1, j - last[c])$.

The rule has three cases:

- ***c* occurs in the pattern to the left of *j*.** The pattern is shifted so that the rightmost c comes under the c in the text. The shift is $j - last[c]$.
- ***c* does not occur in the pattern at all.** Since $last[c] = -1$, the shift is $j + 1$. The pattern moves completely past c.
- **The last occurrence of *c* is to the right of *j*.** Then $j - last[c] \leq 0$. The rule shifts by at least 1 to keep the pattern from moving backward.

The safety of the rule follows from the definition of the last occurrence. Since $last[c]$ is the rightmost occurrence, there is no c in the pattern positions between $last[c] + 1$ and j. Each of the smaller shifts $1, 2, \ldots, j - last[c] - 1$ would bring one of these positions under the c in the text. A mismatch there is certain, so those alignments can be skipped.

```
int[] lastOccurrence(String p) {
  int[] last = new int[256];
  Arrays.fill(last, -1);
  for (int i = 0; i < p.length(); i++)
    last[p.charAt(i)] = i;
  return last;
}
```

The `lastOccurrence` method builds an array of 256 cells and sets all of them to −1. The array has one cell for each of the 256 single-byte values (0–255; the first 128 are ASCII). Then the pattern is scanned from left to right and each character's cell is set to the current position. If a character occurs more than once, the last value written, the rightmost position, remains. The time is $O(m + \sigma)$ and the memory is $O(\sigma)$, where $\sigma = 256$.

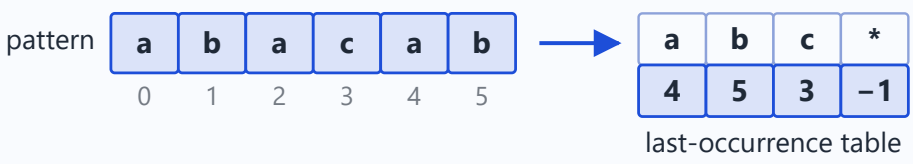


Figure 9.11: The last-occurrence table of the pattern "abacab": $last[a] = 4$, $last[b] = 5$, $last[c] = 3$, and −1 for all other characters.

Example 9.5

Let the text be "GCAATGCCTATGTGACC" and the pattern "TATGTG" ($m = 6$). Find the first shift using the bad-character rule alone.

1. **Table.** The pattern is T A T G T G. $last[T] = 4$, $last[A] = 1$, $last[G] = 5$; C does not occur in the pattern.
2. **Comparison.** At $s = 0$, from right to left: $P[5] = T[5] =$ G matches, $P[4] = T[4] =$ T matches. $P[3] =$ G mismatches $T[3] =$ A; $j = 3$ and the bad character is A.
3. **Shift.** $last[A] = 1$; the shift is $max(1, 3 - 1) = 2$. At the new alignment the A of the pattern (position 1) lies under the A of the text (position 3).

If the search continues, at the second alignment ($s = 2$) the C at the right end is not in the pattern, so the pattern jumps 6 positions. At the third alignment ($s = 8$) all six characters match. Three alignments and 10 comparisons suffice in all.

Common mistake

Writing the shift as just `j - last[c]` without `max(1, …)` . If the last occurrence of c is to the right of j, this difference is zero or negative. For example, in "abacab" $last[a] = 4$; if the mismatch occurs at $j = 3$, the difference is $3 - 4 = -1$. The pattern moves backward or stays in place, and the algorithm may loop forever.

9.4.3 The good-suffix rule

The bad-character rule looks only at the mismatched character; it makes no use of the characters matched so far. The second rule fills this gap. The suffix $u = P[j+1..m-1]$ that matched before the mismatch has also been found in the text. This suffix is called the "good suffix." The **good-suffix rule** shifts the pattern according to another occurrence of u in the pattern:

1. **Case 1.** If u occurs elsewhere in the pattern, preceded by a different character, the pattern is shifted so that this occurrence comes under the u in the text. The preceding character must be different; the same character would repeat the same mismatch.
2. **Case 2.** If there is no such occurrence, the longest prefix of P that equals a suffix of u is aligned.
3. **Case 3.** If neither exists, the pattern is shifted by m positions.

The good-suffix table precomputes this shift for every j. The table is built in $O(m)$ time by a method similar to KMP's LPS idea, this time over suffixes. The details of the construction are long; for this course it is enough to understand the meaning of the rule and to apply it by hand.

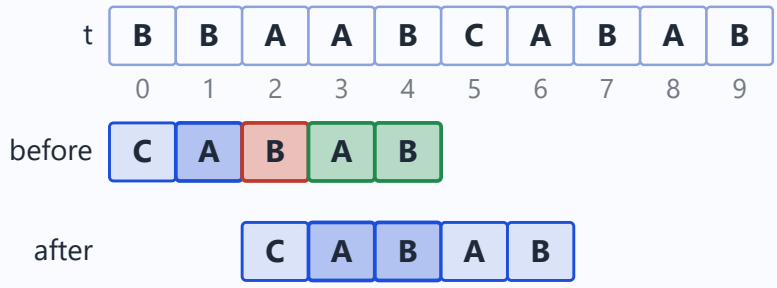


Figure 9.12: The good-suffix rule. The last two characters "AB" of the pattern "CABAB" have matched, and *T[2]* = A mismatches *P[2]* = B. "AB" also occurs in the pattern at positions 1–2, preceded by a different character (C). The pattern is shifted 2 positions so that this occurrence comes under the "AB" in the text.

Figure 9.12 shows the difference between the two rules. The bad character is A, and $last[A] = 3 > j = 2$; the bad-character rule proposes a shift of only 1. The good-suffix rule proposes 2. The algorithm picks the larger and shifts the pattern by 2. Full Boyer-Moore uses the two tables together.

On small alphabets the good-suffix rule matters even more. In DNA ($\sigma = 4$), each of the four letters almost certainly occurs somewhere near the right end of a long pattern. The bad-character rule therefore usually proposes shifts of only 1–2 positions. The good-suffix rule looks at the structure of the matched part, so it can find long shifts.

9.4.4 Example: searching for HELLO

Let the text be "LO LOELLO O HELLO" ($n = 17$) and the pattern "HELLO" ($m = 5$). The example is run with the bad-character rule only. The last-occurrence table is given in Table 9.7; since L occurs twice, the rightmost position (3) is recorded.

c	H	E	L	O	other
last[c]	0	1	3	4	−1

Table 9.7: The last-occurrence table of the pattern "HELLO".

Alignment	*s*	Right-to-left comparison	Bad character	Shift	Cmp.
1	0	O, L match; *T[2]* = ␣ ≠ *P[2]* = L	␣ (absent)	*2 − (−1) = 3*	3
2	3	*T[7]* = L ≠ *P[4]* = O	L (3)	*4 − 3 = 1*	1
3	4	O, L, L, E match; *T[4]* = O ≠ *P[0]* = H	O (4)	*0 − 4 < 1* → 1	5
4	5	*T[9]* = ␣ ≠ *P[4]* = O	␣ (absent)	*4 − (−1) = 5*	1
5	10	*T[14]* = L ≠ *P[4]* = O	L (3)	1	1
6	11	*T[15]* = L ≠ *P[4]* = O	L (3)	1	1
7	12	all five characters match	—	found	5

Alignment	*s*	Right-to-left comparison	Bad character	Shift	Cmp.
Total					17

Table 9.8: Searching for HELLO with Boyer-Moore (bad-character rule). The shift is *max(1, j − last[c])*.

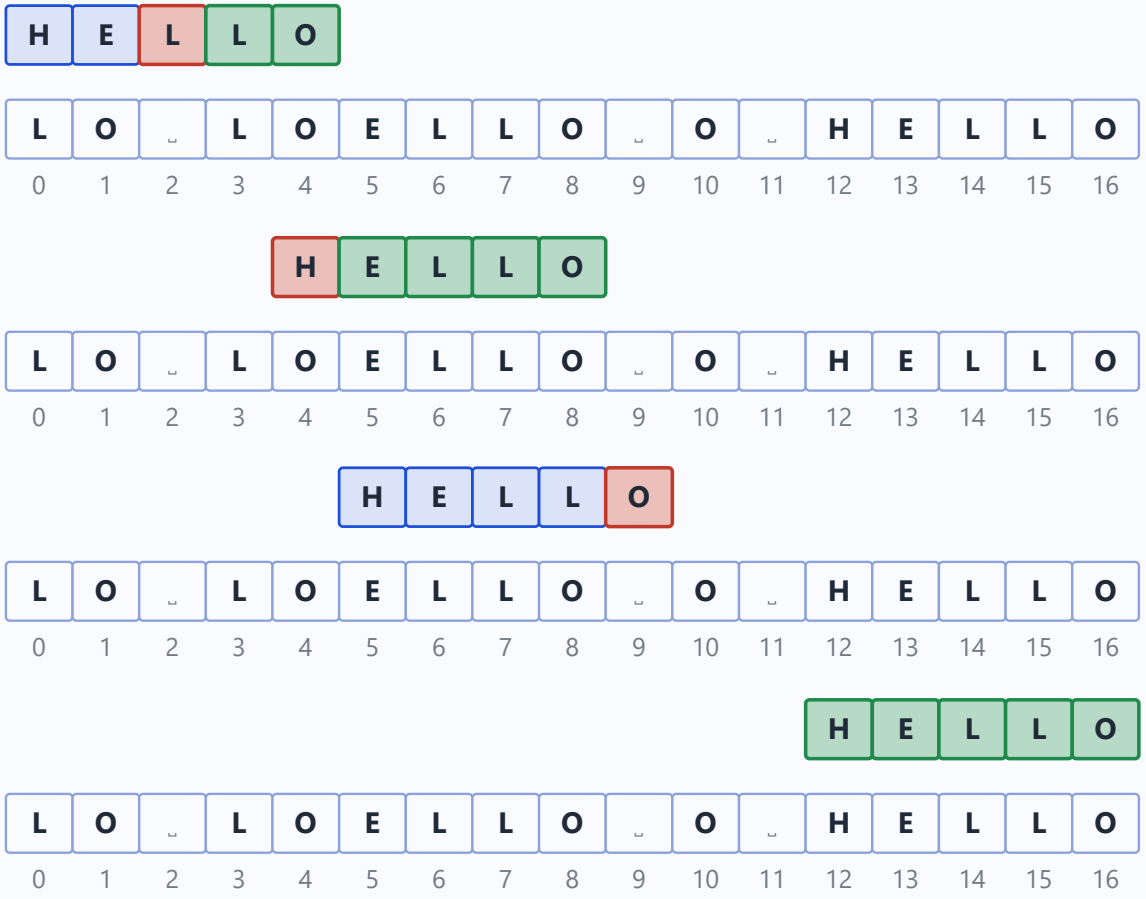


Figure 9.13: Four alignments of the HELLO search: alignments 1, 3, 4, and 7, from top to bottom. Green is matching and red is the bad character. At alignment 4, the space character, which is not in the pattern, gives a jump of 5 positions.

The pattern is found at *s = 12* after 7 alignments and 17 comparisons. Alignment 3 is noteworthy: after four characters match, the mismatch occurs at the left end. The last occurrence of the bad character O lies to the right of *j = 0*, so the rule shifts by only 1. At this alignment the good-suffix rule would have proposed more: the matched "ELLO" occurs nowhere else in the pattern and overlaps no prefix, so the shift would be 5. This example shows clearly how the rules complement each other.

9.4.5 The Horspool simplification

In 1980 R. Nigel Horspool proposed a simplification of Boyer-Moore that is widely used in practice. The good-suffix table is dropped entirely. The bad-character rule is changed too: the shift is determined by the text character under the last character of the alignment, regardless of where the

mismatch occurred.

Definition 9.4

The **Horspool shift table** holds, for each character c, the value $shift[c] = m - 1 - r$, where r is the rightmost position of c in $P[0..m-2]$. If c does not occur in this part, $shift[c] = m$. After each alignment the shift becomes $s \leftarrow s + shift[T[s + m - 1]]$.

The last character of the pattern is left out of the table. If it were included, the shift for the last character would be $m - 1 - (m - 1) = 0$ and the pattern would stand still. For "HELLO" the table is: H → 4, E → 3, L → 1, O → 5, other → 5. L occurs twice (positions 2 and 3); the rightmost, 3, is used and the shift is 1. Since O occurs only at the last position, it never enters the table and gets 5, like "other".

Alignment	s	Comparison	Last character	Shift
1	0	O, L match; $T[2] = ␣ \neq P[2]$	$T[4] =$ O	5
2	5	$T[9] = ␣ \neq P[4]$	$T[9] = ␣$	5
3	10	$T[14] =$ L $\neq P[4]$	$T[14] =$ L	1
4	11	$T[15] =$ L $\neq P[4]$	$T[15] =$ L	1
5	12	all match	—	found

Table 9.9: The same HELLO search with Horspool.

On the same example, Horspool gets by with 5 alignments and 11 comparisons; the basic bad-character rule needed 7 alignments and 17 comparisons. The difference comes from the first alignment. The basic rule looks at the mismatched space character and shifts by 3. Horspool looks at the last character, O. Since O does not occur in the first four characters of the pattern, the pattern shifts by 5. The simplification does not always do better, but with its single table and short code it is very fast in practice.

```
int horspool(String t, String p) {
  int n = t.length(), m = p.length();
  int[] shift = new int[256];
  Arrays.fill(shift, m);
  for (int i = 0; i < m - 1; i++)
    shift[p.charAt(i)] = m - 1 - i;
  int s = 0;
  while (s <= n - m) {
    int j = m - 1;
    while (j >= 0 &&
      p.charAt(j) == t.charAt(s + j))
      j--;
    if (j < 0) return s;
    s += shift[t.charAt(s + m - 1)];
  }
  return -1;
}
```

- **Table:** the 256 cells of the array `shift` are first filled with `m`. Then `P[0..m−2]` is scanned from left to right and `m - 1 - i` is written for each character. Because the scan goes left to right, the rightmost occurrence is written last and stays.
- **Comparison:** `j` starts at the right end and decreases as characters match. If `j < 0`, all characters have matched and `s` is returned.
- **Shift:** wherever the mismatch occurred, `s` increases by the table value of the last character of the alignment, `t[s + m - 1]`. Since this value is at least 1, the pattern never moves backward.

The call `horspool("LO LOELLO O HELLO", "HELLO")` follows the steps in Table 9.9 exactly and returns 12. The code was run in Java, and the counts of 5 alignments and 11 comparisons were confirmed.

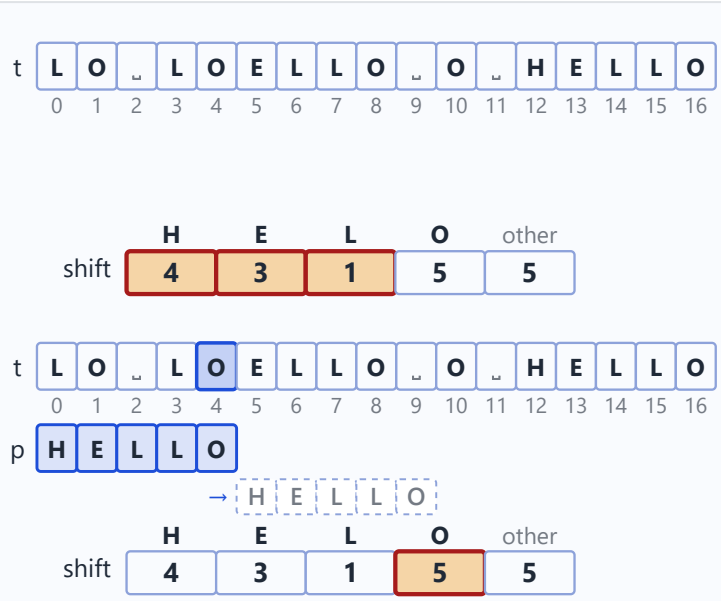


Figure 9.14: On the left, the Horspool shift table. On the right, the shift after the first alignment: the mismatch occurred at $j = 2$, but the shift is determined by the last character of the alignment, $T[4] = $ O, and s becomes 5.

9.4.6 Example: all occurrences of abacab

This example shows all the cases of the bad-character rule together. Let the text be "abacaabadcabacabaabb" ($n = 20$) and the pattern "abacab" ($m = 6$). The last-occurrence table is $a = 4$, $b = 5$, $c = 3$, and -1 for the others. All occurrences are sought; after a match the pattern is shifted by 1 and the search continues.

Alignment	s	Mismatch	Shift	Cmp.
1	0	$T[5]$ = a ≠ $P[5]$ = b	$5 - last[a] = 5 - 4 = 1$	1
2	1	b, a match; $T[4]$ = a ≠ $P[3]$ = c	$last[a] = 4 \geq j = 3 \rightarrow 1$	3
3	2	$T[7]$ = a ≠ $P[5]$ = b	1	1
4	3	$T[8]$ = d ≠ $P[5]$ = b	d absent: $j + 1 = 6$	1
5	9	$T[14]$ = a ≠ $P[5]$ = b	1	1
6	10	six characters match: occurrence, $s = 10$	1 (search continues)	6
7	11	$T[16]$ = a ≠ $P[5]$ = b	1	1
8	12	$T[17]$ = a ≠ $P[5]$ = b	1	1
9	13	b, a match; $T[16]$ = a ≠ $P[3]$ = c	$last[a] = 4 \geq 3 \rightarrow 1$	3
10	14	b matches; $T[18]$ = b ≠ $P[4]$ = a	$last[b] = 5 \geq 4 \rightarrow 1$	2
Total (the search ends when $s = 15 > n - m$)				20

Table 9.10: Searching for all occurrences of "abacab" with Boyer-Moore (bad character).

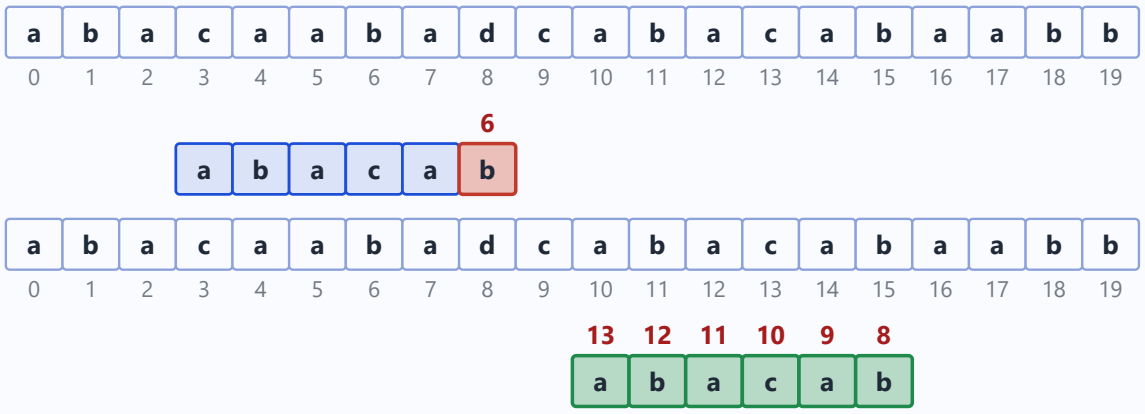


Figure 9.15: Top, alignment 4: the d at the right end is not in the pattern and the pattern jumps 6 positions. Bottom, alignment 6: all six comparisons match. The numbers above the cells are the order of comparison.

The search ends after 10 alignments and 20 comparisons; the pattern occurs only at *s = 10*. Most shifts are 1, because the characters a and b in the text occur near the right end of the pattern. Only d gives a big jump. This table shows that the speed of Boyer-Moore depends on the alphabet and the text: the more frequent the characters absent from the pattern, the longer the jumps.

9.4.7 Java code and trace

Boyer-Moore code that uses only the bad-character rule is very similar to the Horspool code. The only difference is the shift line: the shift is computed from the mismatched character rather than from the last character.

```
int boyerMoore(String t, String p) {
  int n = t.length(), m = p.length();
  int[] last = lastOccurrence(p);
  int s = 0;
  while (s <= n - m) {
    int j = m - 1;
    while (j >= 0 &&
      p.charAt(j) == t.charAt(s + j))
      j--;
    if (j < 0) return s;
    s += Math.max(1,
      j - last[t.charAt(s + j)]);
  }
  return -1;
}
```

We trace the code with the call `boyerMoore("merhaba naber", "abe")`. Here *n = 13* and *m = 3*. The last-occurrence table is *a = 0*, *b = 1*, *e = 2*, and −1 for the others.

`s`	Alignment	Comparison	Shift
0	mer	`p[2]` = e ≠ `t[2]` = r	`last['r'] = -1` : *max(1, 2 − (−1)) = 3*
3	hab	`p[2]` = e ≠ `t[5]` = b	`last['b'] = 1` : *max(1, 2 − 1) = 1*
4	aba	`p[2]` = e ≠ `t[6]` = a	`last['a'] = 0` : *max(1, 2 − 0) = 2*
6	a␣n	`p[2]` = e ≠ `t[8]` = n	`last['n'] = -1` : 3
9	abe	e, b, a match; `j = -1`	found: 9 is returned

Table 9.11: Trace of `boyerMoore("merhaba naber", "abe")`.

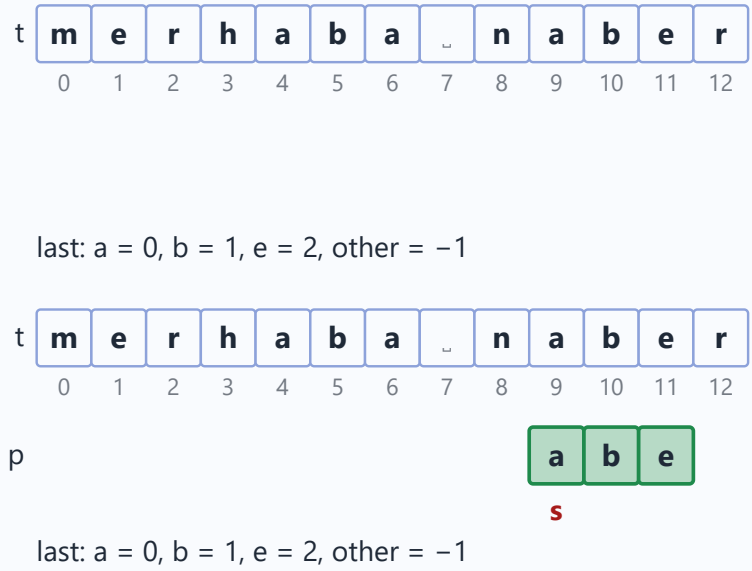


Figure 9.16: On the left, the start: the last-occurrence table is built, *s = 0*. On the right, the final state: "abe" is found at position 9 (inside the word "naber").

The search ends after 5 alignments and 7 comparisons. Brute force tries 10 alignments and makes 15 comparisons on the same search. Only 7 of the 13 characters of the text were looked at. The characters at positions 0, 1, 3, 4, 7, and 12 were never read. This is the property that sets Boyer-Moore apart from brute force and from KMP.

9.4.8 Analysis

The performance of Boyer-Moore varies over a wide range:

- **Best case:** none of the characters of the text occurs in the pattern. Each alignment makes a single comparison and the pattern jumps m positions. About n/m comparisons suffice in total: $O(n/m)$. The running time *shrinks* as the pattern gets longer. In brute force and KMP the opposite happens.
- **Average case:** in natural-language text over a large alphabet, most mismatches occur at the right end and the shifts are long. The time is sublinear; that is, fewer characters are read than the length of the text.

- **Worst case (bad character only):** T = aaaa…a, P = baaa…a. At each alignment $m - 1$ a's match from the right and the b at the left end mismatches. The last occurrence of the bad character a is at the right end, so the shift is 1. The time is $O(n \cdot m)$. For example, with $n = 20$ and $m = 5$, each of the 16 alignments makes 5 comparisons, 80 in all.
- **Worst case (full algorithm):** if the good-suffix rule and the improvement proposed by Zvi Galil in 1979 are used together, the worst case drops to $O(n + m)$.

Example 9.6

Let the text be "abccbaabccbaabcbcabbabcabc" ($n = 26$) and the pattern "abcbcabbabcabc" ($m = 14$). Find the last-occurrence table and search using the bad-character rule only.

1. **Table.** The pattern is a b c b c a b b a b c a b c. The rightmost a is at position 11, b at position 12, and c at position 13: $last[a] = 11$, $last[b] = 12$, $last[c] = 13$.
2. **First alignment.** At $s = 0$, $P[13]$ = c mismatches $T[13]$ = b. The shift is $max(1, 13 - 12) = 1$.
3. **Later alignments.** The search takes 10 alignments. The shifts are 1, 1, 1, 1, 2, 1, 2, 1, 2; none exceeds 2.
4. **Result.** The pattern is found at $s = 12$. In all, 31 comparisons are made.

The alphabet has three letters and each letter occurs at the right end of the pattern. This is why the bad-character rule stays weak here. The example shows why the good-suffix rule is needed on a small alphabet.

The preprocessing cost of Boyer-Moore is $O(m + \sigma)$ and its extra memory is $O(m + \sigma)$. Because the bad-character table keeps one cell for each letter of the alphabet, for large alphabets (e.g. all Unicode characters) a hash table is used instead of an array.

9.5 The Rabin-Karp algorithm

The algorithm published by Michael O. Rabin and Richard M. Karp in 1987 approaches the problem from another side. Instead of choosing shifts cleverly, it makes the comparison at each alignment cheaper. The pattern and the windows of the text are turned into numbers; instead of testing two strings for equality, two numbers are tested.

9.5.1 Comparing numbers instead of characters

Rabin-Karp computes the **hash** value of the pattern and of every window of length m in the text. A hash is a function that maps a string to a small integer; it is a kind of digest of the string. Comparing two numbers is a single operation, while comparing two strings can take up to m operations.

This idea has two problems. First, computing the hash of every window from scratch takes *O(m)*, and then nothing is gained. The solution is the **rolling hash**: when the window slides one position to the right, the new hash is computed from the old one in *O(1)* time. Second, different strings can have the same hash value. So hash equality does not guarantee a match. A window whose hash comes out equal is only a candidate and must be verified character by character.

The steps of the algorithm are as follows:

1. **Hash computation.** The hash values of the pattern and of the first window of the text are computed.
2. **Match check.** A window whose hash value equals that of the pattern is accepted as a candidate.
3. **Verification.** The candidate window is compared with the pattern character by character.
4. **Shift and recompute.** If there is no match, the window slides one position to the right and the hash is updated.
5. **Repeat.** The steps continue to the end of the text.

9.5.2 The sum hash and spurious hits

Let us see the idea with the simplest hash: the sum of the character values. Give the letters the values U = 1, T = 2, S = 3, A = 4, V = 5. We search for "TS" in the text "VUATS"; the hash of the pattern is *2 + 3 = 5*.

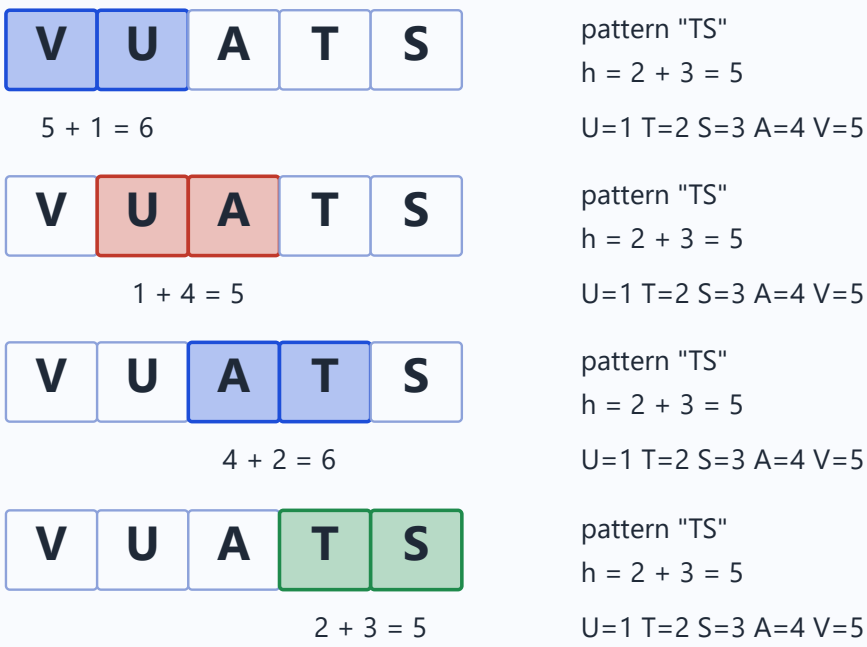


Figure 9.17: Searching for "TS" in "VUATS" with the sum hash, four windows from top to bottom. The hash of the window "UA" equals that of the pattern but its characters differ: a spurious hit. At the window "TS" both the hash and the characters agree.

- **"VU":** *5 + 1 = 6 ≠ 5*. The hash differs; the characters are never looked at.
- **"UA":** *1 + 4 = 5*. The hash is equal, but "UA" ≠ "TS". This is a **spurious hit**; the verification is wasted.
- **"AT":** *4 + 2 = 6 ≠ 5*. The new hash is obtained from the old one: the value of the outgoing U is subtracted and that of the incoming T is added: *5 − 1 + 2 = 6*. The operation is independent of the window length; it is *O(1)*.
- **"TS":** *2 + 3 = 5*. The hash is equal and the character comparison succeeds too: a match at *s = 3*.

The sum hash shows well what a rolling update is, but it is a weak hash. It is insensitive to the order of the letters: "ST" also gives 5. On long windows the values cluster in a narrow range and many windows give the same sum. Real implementations therefore use a position-weighted polynomial hash and a large prime modulus.

9.5.3 Polynomial hash and the rolling update

A polynomial hash reads the string like a number written in base *B*. The character codes are the digits; the leftmost character is the most significant digit. Since the number can grow very large, the remainder modulo a prime *Q* is taken.

Definition 9.5

Let *B* be the base and *Q* a prime modulus. The **polynomial hash** of the window at shift *s* is

$$h_s = (T[s]\cdot B^{m-1} + T[s+1]\cdot B^{m-2} + \ldots + T[s+m-1]) \bmod Q$$

When the window slides one position to the right, the new value is found with the **rolling update** formula:

$$h_{s+1} = (B \cdot (h_s - T[s] \cdot B^{m-1}) + T[s+m]) \bmod Q$$

The formula does three steps in one line. First the contribution of the outgoing character, $T[s] \cdot B^{m-1}$, is subtracted. Then the remaining digits are multiplied by *B*, which shifts them one digit to the left. Finally the incoming character is added in the lowest digit. A decimal analogy shows this clearly: to go from "314" to "141", drop the hundreds digit 3 (14), shift one digit left (140), and add 1 (141).

The value $B^{m-1} \bmod Q$ is needed in every update, and since the window length does not change, it is computed once and stored. The rules of modular arithmetic allow taking remainders at every intermediate step. The values $(a + b) \bmod Q$ and $(a \cdot b) \bmod Q$ can also be computed from the remainders of *a* and *b*. In this way the numbers never grow.

Example 9.7

Let the base be $B = 10$, $Q = 13$, and the characters digits. For T = "31415" and $m = 3$, find the hashes of the windows with the rolling formula.

1. **Constant.** $B^{m-1} \bmod Q = 100 \bmod 13 = 9$.
2. **First window.** $h(\text{"314"}) = 314 \bmod 13 = 2$ ($13 \cdot 24 = 312$).
3. **Second window.** $h(\text{"141"}) = (10 \cdot (2 - 3 \cdot 9) + 1) \bmod 13 = (-249) \bmod 13 = 11$. Direct computation: $141 = 13 \cdot 10 + 11$ ✓.
4. **Third window.** $h(\text{"415"}) = (10 \cdot (11 - 1 \cdot 9) + 5) \bmod 13 = 25 \bmod 13 = 12$. Direct computation: $415 = 13 \cdot 31 + 12$ ✓.

In the second step the intermediate result is negative. In mathematics $(-249) \bmod 13 = 11$, because $-249 = 13 \cdot (-20) + 11$. Java's `%` operator, however, gives a negative remainder for a negative number: the expression `-249 % 13` is −2. Adding 13 to the result gives the correct value, 11.

9.5.4 Java code and trace

The code below uses ASCII codes as digits: the base is $B = 256$ and the modulus is $Q = 101$. The helper method `matchAt(t, p, s)` compares the candidate window with the pattern character by character.

```
int rabinKarp(String t, String p) {
  int n = t.length(), m = p.length();
  int B = 256, Q = 101, h = 1;
  for (int i = 0; i < m - 1; i++)
    h = (h * B) % Q;  // B^(m-1) mod Q
  int ph = 0, th = 0;
  for (int i = 0; i < m; i++) {
    ph = (B * ph + p.charAt(i)) % Q;
    th = (B * th + t.charAt(i)) % Q;
  }
  for (int s = 0; s <= n - m; s++) {
    if (ph == th && matchAt(t, p, s))
      return s;
    if (s < n - m) {
      th = (B * (th - t.charAt(s) * h)
            + t.charAt(s + m)) % Q;
      if (th < 0) th += Q;
    }
  }
  return -1;
}
```

- **First loop:** `h = B^(m-1) mod Q` is computed. This is the weight of the outgoing character.

- **Second loop:** `ph` (the pattern hash) and `th` (the hash of the first window) are computed with Horner's method: at each step the previous value is multiplied by `B` and the new character is added. This loop takes *O(m)*.
- **Main loop:** for each shift, the hashes are compared first. Thanks to `&&`, `matchAt` is called only if the hashes are equal. If there is no match and the end of the text has not been reached, `th` is updated with the rolling formula.
- **The line** `if (th < 0) th += Q;` fixes Java's negative-remainder problem. The expression `th - t.charAt(s) * h` is often negative, and the `%` operator returns a negative remainder.

We trace the code with the call `rabinKarp("abracadabra", "cad")`. Here $n = 11$ and $m = 3$. $h = 256^2 \bmod 101 = 88$. The pattern hash is $ph = (99 \cdot 256^2 + 97 \cdot 256 + 100) \bmod 101 = 11$ (c = 99, a = 97, d = 100). The hash of the first window "abr" is 4.

`s`	Window	`th`	Decision	Out / in	Raw	`% 101`	New `th`
0	abr	4	4 ≠ 11, skip	a (97) / a (97)	−2,184,095	−71	30
1	bra	30	30 ≠ 11, skip	b (98) / c (99)	−2,199,965	−84	17
2	rac	17	17 ≠ 11, skip	r (114) / a (97)	−2,563,743	−60	41
3	aca	41	41 ≠ 11, skip	a (97) / d (100)	−2,174,620	−90	11
4	cad	11	11 = 11: `matchAt` verifies	"cad" = "cad": 4 is returned			

Table 9.12: Trace of `rabinKarp("abracadabra", "cad")` (*B = 256, Q = 101, h = 88, ph = 11*). The "Raw" column is the value of `B * (th - out * h) + in`.

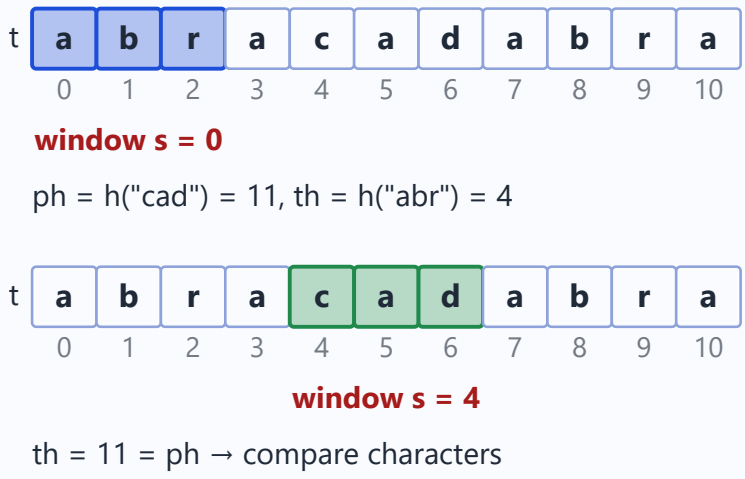


Figure 9.18: On the left, the start: *ph = 11* and *th = 4* for the first window "abr". On the right, *s = 4*: the hashes are equal, the character comparison succeeds too, and "cad" is found.

In the trace, the hashes differ in four windows and the characters are never looked at. A character comparison, over three characters, is made only in the fifth window. All the "raw" values in the table are negative. Java's `%` operator therefore gives a negative remainder at every step, and the correction

line runs at every step. If this line were forgotten, the hash of the second window would be −71 instead of 30. The later hashes would come out wrong too, and the hash of the "cad" window might never equal 11.

Common mistake

Choosing a large Q but staying with the `int` type. In the code above $Q = 101$, and the intermediate values do not exceed about $\pm 6.6 \cdot 10^6$; `int` is enough. To reduce spurious hits one can choose a prime such as $Q = 10^9 + 7$. Then even the product `B * th` reaches $2.56 \cdot 10^{11}$, while the `int` limit is about $2.1 \cdot 10^9$. Overflow is silent; the hashes come out wrong and real matches may be missed. In that case the arithmetic must be done in `long` .

9.5.5 Analysis and multiple patterns

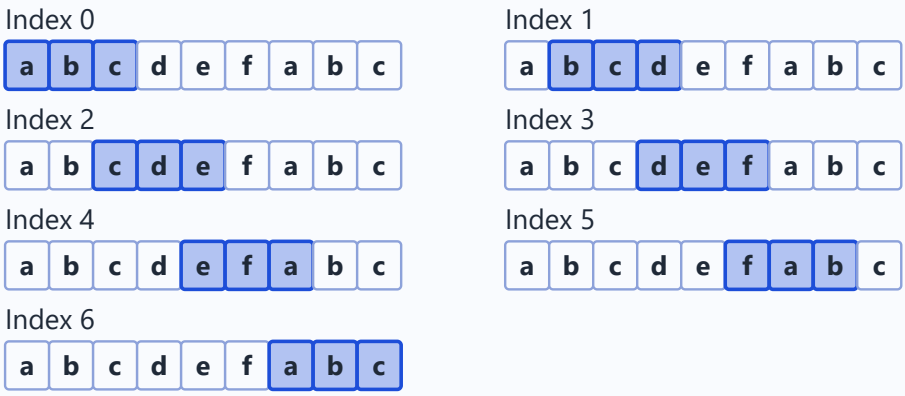


Figure 9.19: A window of length 3 sliding from 0 to 6 in the text "abcdefabc". The hash of each of the $n - m + 1 = 7$ windows is obtained from the previous one in $O(1)$ time.

The cost of Rabin-Karp has two parts: the hash computations and the verifications. The hash computations always total $O(m) + (n - m) \cdot O(1) = O(n + m)$. The cost of verification depends on how many windows become candidates.

- **Best and average case:** if Q is a large prime, the probability that a random window's hash accidentally equals that of the pattern is about $1/Q$. The expected number of spurious hits is $(n - m + 1)/Q$. For $n = 10^6$ and $Q \approx 10^9$ this is about 0.001: most likely no spurious hit occurs in the whole text. If there are k real occurrences, the expected time is $O(n + m + k \cdot m)$; for a small number of occurrences, $O(n + m)$.
- **Worst case:** if a bad Q is chosen, or the input is deliberately crafted to produce collisions, every window becomes a candidate. For example, for T = a…a and P = a…a every window is a real match and each is verified with m characters. The time is $O(n \cdot m)$.

The extra memory is *O(1)*; only a few integers are kept. The real strength of Rabin-Karp shows not with a single pattern but with **many patterns**. The hashes of k patterns of the same length are put into a set (`HashSet`). The text is scanned once and the hash of each window is looked up in the set. Since a set lookup is *O(1)* on average, all the patterns are found in expected $O(n + k \cdot m)$ time. The same job with KMP or Boyer-Moore would require scanning the text k times.

```
List<Integer> rkMulti(String t, List<String> patterns) {
  int n = t.length(), m = patterns.get(0).length();
  int B = 256, Q = 101, h = 1;
  for (int i = 0; i < m - 1; i++)
    h = (h * B) % Q;
  Set<Integer> hashes = new HashSet<>();
  Set<String> patternSet = new HashSet<>(patterns);
  for (String p : patterns) {
    int ph = 0;
    for (int i = 0; i < m; i++)
      ph = (B * ph + p.charAt(i)) % Q;
    hashes.add(ph);
  }
  List<Integer> positions = new ArrayList<>();
  int th = 0;
  for (int i = 0; i < m; i++)
    th = (B * th + t.charAt(i)) % Q;
  for (int s = 0; s <= n - m; s++) {
    if (hashes.contains(th)
        && patternSet.contains(t.substring(s, s + m)))
      positions.add(s);
    if (s < n - m) {
      th = (B * (th - t.charAt(s) * h)
            + t.charAt(s + m)) % Q;
      if (th < 0) th += Q;
    }
  }
  return positions;
}
```

The code was run in Java: the call `rkMulti("abracadabra", List.of("cad", "bra", "xyz"))` returns *[1, 4, 8]*. "bra" occurs at 1 and 8, "cad" at 4; "xyz" is not in the text. The verification checks whether the window itself is in the set of patterns. This eliminates windows whose hash collides with that of some pattern but which equal no pattern.

This technique is the basis of plagiarism detection and of document similarity measurement. The hashes of all k-word pieces of one document are extracted and compared with the pieces of another document. The same idea extends to two dimensions: a small image block can be searched for inside an image using row and column hashes.

9.6 Comparing the matching algorithms

The four algorithms answer the same question, but each is best under different assumptions. Table 9.13 puts their basic properties side by side.

Algorithm	Key idea	Preprocessing	Average	Worst	Extra memory
Brute force	Tries every shift	none	$O(n)$*	$O(n \cdot m)$	$O(1)$
KMP	LPS table, never moves back in the text	$O(m)$	$O(n + m)$	$O(n + m)$	$O(m)$
Boyer-Moore	Right to left; bad character and good suffix	$O(m + \sigma)$	sublinear	$O(n \cdot m)$**	$O(m + \sigma)$
Rabin-Karp	Rolling hash comparison	$O(m)$	$O(n + m)$	$O(n \cdot m)$	$O(1)$

Table 9.13: Comparison of the string matching algorithms. σ is the alphabet size.

* For random text and a large alphabet; on repetitive text it approaches $O(n \cdot m)$. ** With the bad-character rule only; $O(n + m)$ with the good-suffix rule and Galil's improvement.

Asymptotic bounds are only part of the picture. Table 9.14 gives the character comparison counts of the four methods on the examples used in these notes. The counts were obtained by running the algorithms in Python and Java. In KMP each pair of characters is counted once; re-checking the same pair in the `if` after leaving the `while` loop on equality is not counted separately. The Boyer-Moore column uses only the bad-character rule.

Text / pattern	n, m	Brute force	KMP	Boyer-Moore	Horspool
"THIS IS A SIMPLE EXAMPLE" / "SIMPLE"	24, 6	18	18	8	8
"ABCABCDABABCDABCDABDE" / "ABCDABD"	21, 7	37	24	12	12
"ababcababcababababd" / "ababd"	17, 5	33	22	8	8
"LO LOELLO O HELLO" / "HELLO"	17, 5	17	17	17	11
"merhaba naber" / "abe"	13, 3	15	14	7	7
"aaaaaaaaaa" / "aaab" (no occurrence)	10, 4	28	17	7	7

Table 9.14: Character comparisons made until the first occurrence is found.

Three observations follow from the table. First, KMP is never worse than brute force. Its gain grows on repetitive texts, where the matched parts are long. On natural-language text (the SIMPLE row) the two methods do almost the same work. Second, Boyer-Moore is far ahead in most rows. In the last row the worst input for brute force is one of the easiest for Boyer-Moore: the b at the right end of the pattern mismatches at the first comparison of every alignment, and the pattern moves ahead each time. Third, no method is best on every input. In the HELLO row the basic bad-character rule is no better than brute force, while Horspool is better.

A rough roadmap for choosing:

- **KMP when a guarantee is needed.** It is preferred for stream data (data arriving over a network that cannot be read back) or in real-time systems where the worst-case bound matters.
- **Boyer-Moore or Horspool for long patterns and natural-language text.** This is the fastest option in practice in text editors and command-line tools.
- **Rabin-Karp for many patterns.** Many patterns of the same length are searched in a single pass: plagiarism checking, virus signatures.
- **Brute force for short patterns and short texts.** It needs no preprocessing, its code is short, and it needs no extra memory. For patterns of a few characters the setup cost of the other methods can exceed their gain.

9.7 String compression

The second topic of the chapter is storing and transmitting text, or data in general, with fewer bits. Compression saves disk space and network bandwidth. If the size of a file is halved, its download time is roughly halved as well. Compression algorithms find the **redundancy** in data, that is, the repeated or predictable parts, and write them in a shorter form.

9.7.1 Lossless and lossy compression

Definition 9.6

In **lossless compression**, when the compressed data is decompressed the original data is recovered exactly, bit for bit. In **lossy compression**, details considered perceptually unimportant are discarded and the decompressed data is only close to the original.

The **compression ratio** is the ratio of the compressed size to the original size. The smaller the ratio, the better the compression; a ratio above 1 means the data has grown.

Lossy compression is used in JPEG images and MP3 audio. Discarding color transitions in a photograph that the eye cannot notice gives a large saving. For text, however, lossy compression is unsuitable. Changing a single character in a program breaks it; changing a single digit in a contract changes its meaning. This is why text, programs, and PNG and GIF images are compressed losslessly. The two methods in this chapter, RLE and LZW, are lossless.

An example of using the ratio: if a text of 15 characters shrinks to 8 characters, the ratio is $8/15 \approx 0.53$; the data has roughly halved. In some sources the ratio is given the other way round (original / compressed) or as a percentage saving. These notes use the definition given in the slides.

9.7.2 No method shrinks every input

Compression has an insurmountable limit. No lossless method can shrink every input. This result rests on a simple counting argument, the **pigeonhole principle**: if $k + 1$ pigeons are placed in k holes, at least one hole contains two pigeons.

Theorem 9.3 (Limit of lossless compression)

No lossless compression method can map all inputs of length N bits to shorter bit strings.

Proof. **Proof.** The number of distinct inputs of N bits is 2^N. The number of bit strings shorter than N (including the empty string) is $2^0 + 2^1 + \ldots + 2^{N-1} = 2^N - 1$. If every input were mapped to a shorter string, 2^N inputs would be spread over $2^N - 1$ outputs. By the pigeonhole principle, at least two different inputs would land on the same output. The decompressor could not tell which input an output came from, and the method could not be lossless. ■

For example, there are 256 distinct 8-bit bytes, but only 255 distinct bit strings shorter than 8 bits. The practical consequence of the theorem is this: compression algorithms target a particular kind of redundancy. They shrink inputs that contain that redundancy and enlarge slightly those that do not. When evaluating RLE and LZW, the question to ask is which redundancy they capture and on which inputs they fail. The fact that compressing an already compressed file gains almost nothing has the same cause.

9.8 Run-length encoding (RLE)

Run-length encoding (RLE) is one of the simplest compression methods. The redundancy it captures is the consecutive repetition of the same symbol.

9.8.1 Runs

Definition 9.7

A longest stretch of consecutive repeats of the same character in a string is called a **run**. RLE replaces each run with a pair consisting of the run's length and the character.

The steps of the method are: scan the text from left to right, count consecutive repeats, write the count and the character side by side for each run, and store the result as the compressed text. Since the text is scanned once, the time is $O(n)$.

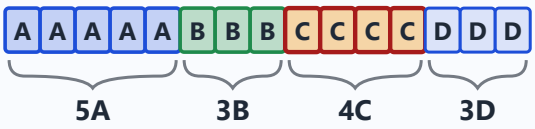


Figure 9.20: "AAAAABBBCCCCDDD" splits into four runs: 5 A, 3 B, 4 C, and 3 D. The compressed text is "5A3B4C3D".

The 15-character text "AAAAABBBCCCCDDD" becomes "5A3B4C3D": 8 characters, ratio $8/15 \approx 0.53$. If there are long runs the gain is large: a text of 1000 a's becomes "1000a", that is, 5 characters.

If there is little repetition, RLE does not help and can even hurt. In "ABCD" every run is a single character and the result is "1A1B1C1D": 4 characters grow to 8, a ratio of 2. As the pigeonhole principle predicts, RLE enlarges inputs that contain no runs. An input that is repetitive but has no consecutive repeats, such as "abababab", becomes "1a1b1a1b1a1b1a1b"; 8 characters grow to 16. RLE cannot see this kind of repetition.

The natural uses of RLE are data with long flat regions. Fax machines encode the lines of black-and-white pages with RLE; most of a line is white. Older image formats such as BMP, PCX, and TIFF, and simple icons, also use RLE.

9.8.2 Java code and trace

```
String rleEncode(String s) {
  StringBuilder result = new StringBuilder();
  int n = s.length();
  for (int i = 0; i < n; i++) {
    char c = s.charAt(i);
    int count = 1;
    while (i + 1 < n &&
      s.charAt(i + 1) == c) {
      count++;
      i++;
    }
    result.append(count).append(c);
  }
  return result.toString();
}
```

- **Outer loop:** `i` stops at the first character of each new run. `c` is this character and `count` starts at 1.
- **Inner loop:** as long as the next character is the same as `c`, `count` and `i` increase together. The condition `i + 1 < n` is checked first; at the end of the text `charAt` does not read outside the string.

- **Writing:** when the inner loop finishes, `i` is at the last character of the run. The count and the character are appended to the result. The `i++` of the outer loop carries `i` to the start of the next run.

Because the inner loop also increments `i`, the code is linear despite the nested loops: each character is visited once, either in the outer loop or in the inner loop. The total time is $O(n)$. The result is built with a `StringBuilder`; `String` concatenation creates a new object at every append and could lead to $O(n^2)$ time.

Run	`i` at start	`c`	Inner iterations	`i`, `count` at end	`result`
1	0	A	4	4, 5	"5A"
2	5	B	2	7, 3	"5A3B"
3	8	C	3	11, 4	"5A3B4C"
4	12	D	2	14, 3	"5A3B4C3D"

Table 9.15: Trace of `rleEncode("AAAAABBBCCCCDDD")` ($n = 15$).

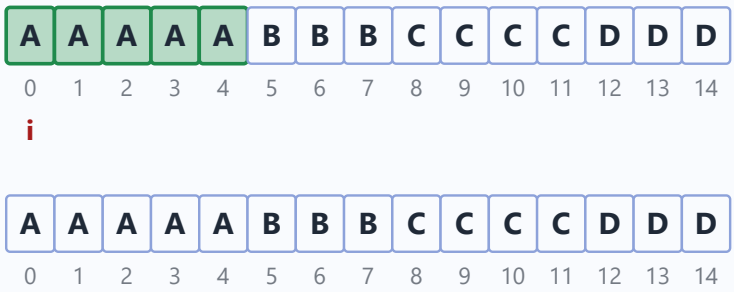


Figure 9.21: On the left, the end of the first run: $i = 4$, $count = 5$; "5A" has been written. On the right, the final state: 8 characters instead of 15.

9.8.3 Decoding and multi-digit counts

Decoding RLE is the reverse of compression: for each (count, character) pair, the character is written count times. Text-form RLE has one subtlety: counts can have several digits. When decoding "12A3B", if "1" is read alone, the output turns into something meaningless like "one 2". The correct reading is to accumulate the digits until a letter arrives and form a single number.

```
String rleDecode(String s) {
  StringBuilder result = new StringBuilder();
  int num = 0;
  for (char c : s.toCharArray()) {
    if (Character.isDigit(c)) {
      num = num * 10 + (c - '0');
    } else {
      for (int k = 0; k < num; k++)
        result.append(c);
      num = 0;
    }
  }
  return result.toString();
}
```

In the code, `num` is the number formed from the digits read so far. Whenever a digit arrives, the old value is multiplied by 10 and the digit added, as in the decimal system: for "12", first 1, then $1 \cdot 10 + 2 = 12$. When a letter arrives, it is written `num` times and `num` is reset. The code was run in Java: `rleDecode("12A3B1C")` produces twelve A's, three B's, and one C; `rleDecode("5A3B4C3D")` returns the original text. The time is linear in the length of the output.

This method has a limit: if the text itself contains digits, counts and characters get mixed up. Twelve repeats of the character "3" would be written "123", and the question "123 of what?" would go unanswered. Byte-level RLE solves this problem by keeping the counts in separate bytes.

9.8.4 Byte-level RLE and the flag byte

In real files RLE works on bytes. In its simplest form each run is written in two bytes: first the count, then the code of the character. The count must fit in one byte; runs longer than 255 are split into several runs. This format eliminates the digit problem, but in regions without repetition it doubles every character to two bytes.

RLE with a flag reduces this loss. Only long runs are written, as a three-byte group that starts with a special **flag byte**: flag, count, character. The characters in short runs are left as they are. When the decoder sees a flag byte it reads the next two bytes as the count and the character; when it sees any other byte it writes it directly to the output.

Example 9.8

Let the text be "aaaaaaaaaabbbbbbeceeceeceecddddddddddddddddecb" (46 bytes). The flag byte is 255; a = 97, b = 98, c = 99, d = 100, e = 101.

1. **Runs.** 10 a's and 6 b's are followed by twelve single-character runs (e c e c … e c). Then there are 15 d's and the single characters e, c, b: 18 runs in all.

2. **RLE without a flag.** Each run takes 2 bytes: *18 · 2 = 36* bytes. The sequence begins: 10 97 06 98 01 101 01 99 …; every character in the non-repeating region takes 2 bytes.
3. **RLE with a flag.** The three long runs are written in three bytes each: 255 10 97, 255 06 98, and 255 15 100. That is 9 bytes. The remaining 15 single characters stay as they are: 15 bytes. Total *9 + 15 = 24* bytes.
4. **Ratios.** Without a flag *36/46 ≈ 0.78*, with a flag *24/46 ≈ 0.52*.

The full sequence of the flagged format: 255 10 97 255 06 98 101 99 101 99 101 99 101 99 101 99 101 99 255 15 100 101 99 98.

The value of the flag byte can also occur in the text. If a character with value 255 were written directly, the decoder would mistake it for a flag. So this character too must be written in an escape form, for example as the triple 255 1 255. How long a run must be to count as "long" is also a design decision. The three-byte group pays off only for runs of four or more. For a run of three characters the two ways of writing have the same length; for a run of two, writing the characters as they are is shorter.

9.8.5 RLE in images

RLE is most effective on images with large flat areas. In a black-and-white image each row is scanned from left to right and pixels of the same color are counted until the color changes.

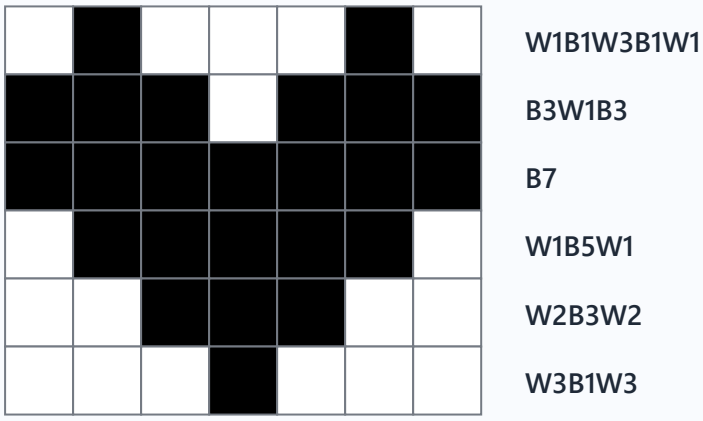


Figure 9.22: A 7 × 6 pixel black-and-white heart image and its row-by-row RLE codes. W is a white run and B is a black run; the number after the letter is the run length.

The 42-pixel image in Figure 9.22 is written with 18 runs. The rows are encoded separately; the first row becomes "W1B1W3B1W1" and consists of five runs. The run lengths of each row add up to 7. This provides a check: if the decoder does not produce 7 pixels in every row, the data is corrupt.

Since there are only two colors, the color name need not even be written. The runs are assumed to alternate between white and black; if a row starts with black, the first white run is written with length 0. The fax standard uses this idea and additionally compresses the run lengths with variable-length codes. RLE is very effective on images with large flat areas (drawings, text scans). In photographs, however, neighboring pixels are rarely exactly the same color and RLE is ineffective.

In color images, runs are defined at the pixel level. The color of each pixel is a three-byte (R, G, B) triple, and a run consists of consecutive pixels of the same color.

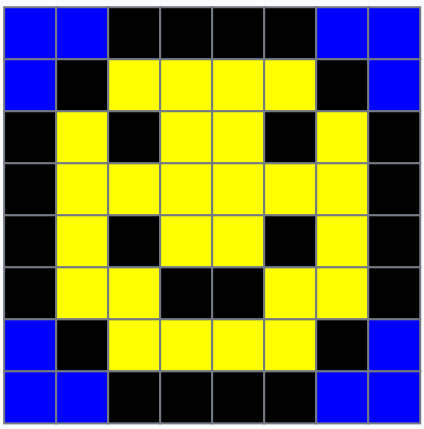


Figure 9.23: An 8 × 8 pixel smiley face. The colors are blue (0, 0, 255), black (0, 0, 0), and yellow (255, 255, 0). The pixels are read row by row, from left to right.

Example 9.9

Compare the raw and RLE sizes of the image in Figure 9.23. Each run is written with one count byte and three color bytes.

1. **Raw size.** 64 pixels, 3 bytes each: $64 \cdot 3 = 192$ bytes.
2. **Runs.** The pixels are read row by row. If the same color continues at the end of a row, the run spills into the next row. The last two blues of the first row and the first blue of the second row form a single run of "3 blue". The image splits into 33 runs. The first runs are 2 blue, 4 black, 3 blue, 1 black, 4 yellow, 1 black.
3. **RLE size.** $33 \cdot (1 + 3) = 132$ bytes. The ratio is $132/192 \approx 0.69$.

If runs were cut at row boundaries, the number of runs would rise to 38 and the RLE size would be $38 \cdot 4 = 152$ bytes. On a small, detailed image the gain is limited; if the same face were drawn at 800 × 800 pixels, the runs would be a hundred times longer and the ratio would be much smaller.

9.9 Lempel-Ziv-Welch (LZW)

RLE sees only consecutive repetition. In natural-language text, however, repeats are mostly not consecutive: words such as "the", "and", "of", or suffixes such as "-ing" and "-ed" are scattered throughout the text. LZW captures this kind of repetition.

9.9.1 The dictionary idea

In 1977 and 1978 Abraham Lempel and Jacob Ziv published two dictionary-based compression methods (LZ77 and LZ78). In 1984 Terry Welch simplified LZ78 and brought it to a form that runs fast in hardware. The resulting method was named LZW and was used in the GIF, TIFF, and PDF formats and in Unix's `compress` program.

The essence of the method is this: the compressor numbers the strings it has seen before in a dictionary. When it meets the same string again, it writes the string's number in the dictionary, that is, its code, instead of the string itself. The dictionary is **dynamic**: it grows as compression proceeds. The dictionary is not sent along with the output. The decompressor rebuilds the same dictionary itself, applying the same rules as the compressor.

Definition 9.8

LZW compression starts with an initial dictionary containing all the single characters of the text. w starts as the empty string. For each character c of the text:

- If $w + c$ is in the dictionary, set $w \leftarrow w + c$; the match is extended.
- Otherwise the code of w is written to the output, $w + c$ is added to the dictionary under the next free code, and $w \leftarrow c$.

When the text ends, if w is not empty, its code is written to the output as well.

The rule is greedy: at each step the compressor writes the longest piece found in the dictionary. Then it adds that piece extended by one character to the dictionary. In this way the strings in the dictionary grow longer over time. Frequently repeated pieces are written with ever longer dictionary entries and a single code.

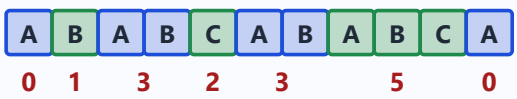


Figure 9.24: The LZW pieces and codes of the text "ABABCABABCA": A 0, B 1, AB 3, C 2, AB 3, ABC 5, A 0. Seven codes are written instead of 11 symbols.

9.9.2 Compression: Java code and trace

```
List<Integer> lzw(String s) {
  Map<String, Integer> dict =
      initialDict(s);  // A=0, B=1, C=2
  String w = "";
  List<Integer> codes = new ArrayList<>();
  for (char c : s.toCharArray()) {
    String wc = w + c;
    if (dict.containsKey(wc)) {
      w = wc;
    } else {
      codes.add(dict.get(w));
      dict.put(wc, dict.size());
      w = "" + c;
    }
  }
  if (!w.isEmpty())
    codes.add(dict.get(w));
  return codes;
}
```

- **Dictionary:** `Map<String, Integer>` maps each string to its code. `initialDict(s)` numbers the distinct characters of the text starting from 0; in this example A = 0, B = 1, C = 2.
- **Extension:** if `wc = w + c` is in the dictionary, only `w` grows; nothing is written to the output.
- **Writing and adding:** if `wc` is not in the dictionary, the code of `w` is written. `wc` is added with the code `dict.size()`, which is always the next free code. `w` goes back to the string consisting of `c` alone.
- **The final `w`:** when the loop ends, `w` still holds a piece that has not been written, and its code is appended.

We trace the code with the call `lzw("ABABCABABCA")`. Table 9.16 gives the state after each character.

Step	`c`	`w + c`	In dictionary?	Output	Added	New `w`
1	A	A	yes (0)	—	—	A
2	B	AB	no	0	3 = AB	B
3	A	BA	no	1	4 = BA	A
4	B	AB	yes (3)	—	—	AB
5	C	ABC	no	3	5 = ABC	C
6	A	CA	no	2	6 = CA	A
7	B	AB	yes (3)	—	—	AB
8	A	ABA	no	3	7 = ABA	A
9	B	AB	yes (3)	—	—	AB

Step	`c`	`w + c`	In dictionary?	Output	Added	New `w`
10	C	ABC	yes (5)	—	—	ABC
11	A	ABCA	no	5	8 = ABCA	A
end	—	—	—	0	—	—

Table 9.16: Trace of `lzw("ABABCABABCA")`. The initial dictionary is A = 0, B = 1, C = 2.

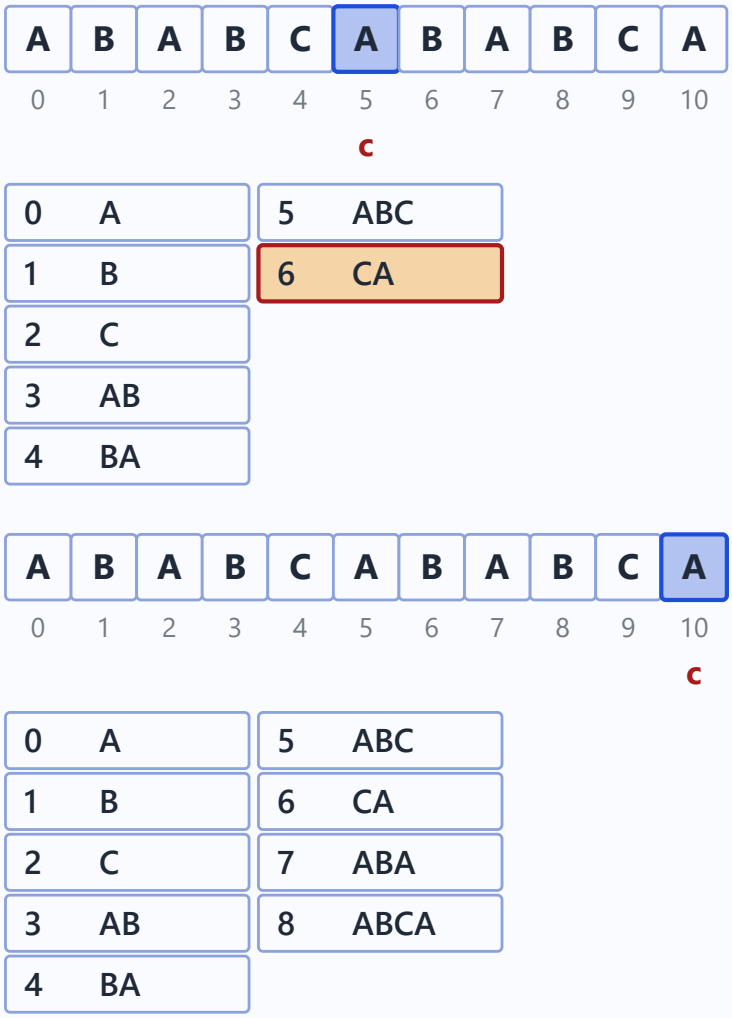


Figure 9.25: On the left, step 6: "CA" is not in the dictionary; the code 2 for C is written and 6 = CA is added. On the right, the final state: the dictionary has grown to codes 0–8 and the output is 0 1 3 2 3 5 0.

The output is *0 1 3 2 3 5 0*: 7 codes instead of 11 symbols. Step 10 shows how the dictionary pays off. The "ABC" added at step 5 is seen again in the second half of the text and is written with a single code (5). In so short a text the dictionary can be used only a few times. In long texts the strings in the dictionary grow longer and the gain increases.

The running time works out as follows: for each character one dictionary lookup and at most one insertion are done. If the dictionary is kept in a hash table, each operation takes $O(1)$ on average and the total time is $O(n)$. There is a subtlety in this count: the string `w + c` is rebuilt at every step and its

hash is computed; the cost is proportional to the length of `w`. Practical implementations therefore keep the dictionary as a trie, or store each entry as a pair (prefix code, last character). Then each step really takes constant time.

9.9.3 Decompression

The decompressor knows only the initial dictionary. It builds the rest of the dictionary itself, in the same order as the compressor, as it reads the codes. The rule is: for each new code k, write $e = dict[k]$ to the output, add $w + e[0]$ to the dictionary, and set $w \leftarrow e$. Here $e[0]$ is the first character of e.

Why the added string is $w + e[0]$ follows from the compressor's behavior. When the compressor wrote w, it added $w + c$ to the dictionary; c was the first character of the next piece. At the moment the decompressor decodes w, it does not yet know the next piece. When it decodes the next code it learns the first character of that piece and completes the missing entry. The decompressor therefore always stays one step behind in the dictionary.

Code	Output (e)	Added ($w + e[0]$)
0	A	—
1	B	3 = AB
3	AB	4 = BA
2	C	5 = ABC
3	AB	6 = CA
5	ABC	7 = ABA
0	A	8 = ABCA

Table 9.17: Decoding the codes *0 1 3 2 3 5 0* (A = 0, B = 1, C = 2).

Writing the outputs one after another gives A·B·AB·C·AB·ABC·A = ABABCABABCA; the original text is back. The codes 3–8 built by the decompressor are exactly the same as the compressor's.

In one situation this rule gets stuck. The decompressor can read a code that is not yet in its dictionary. This can happen in only one way: the code is the one the decompressor would add next.

Theorem 9.4 (The special case of LZW)

If the decompressor reads a code k that is not yet defined in its dictionary, then k is the next free code and $e = w + w[0]$. Here w is the decoding of the previous code.

Proof. **Proof.** Because the decompressor is one step behind, the only code that the compressor has defined but the decompressor has not yet is the code the compressor added most recently. The compressor added this code as $w + c$ when it wrote w; c is the first character of the next piece. The

next piece is exactly this new code: $e = w + c$. Since e begins with w, its first character is $w[0]$. And c is the first character of the next piece, that is, of e. Hence $c = e[0] = w[0]$ and $e = w + w[0]$. ■

This case arises when the compressor uses an entry immediately after adding it. For that, the text must contain a piece of the form $c \cdot S \cdot c \cdot S \cdot c$ with cS in the dictionary. Here c is a character and S is a string. For this reason the literature calls it the "cScSc case."

Example 9.10

Compress "ABABABA" with the dictionary A = 0, B = 1, and then decompress it.

1. **Compression.** AB is not in the dictionary: 0 is written and 2 = AB is added. BA is not: 1 is written and 3 = BA is added. AB is present but ABA is not: 2 is written, 4 = ABA is added, and w = A. In the last three characters AB and ABA are found in the dictionary; w grows to ABA. When the text ends, the code 4 of the remaining w is written. The output is *0 1 2 4*.
2. **Decompression.** 0 → A. 1 → B; 2 = AB is added. 2 → AB; 3 = BA is added.
3. **The special case.** The next code is 4, but the decompressor's dictionary goes only up to 3. By the theorem, $e = w + w[0]$ = AB + A = ABA. ABA is written to the output and 4 = ABA is added.
4. **Result.** A·B·AB·ABA = ABABABA ✓.

9.9.4 Example: a DNA sequence

LZW does not depend on the alphabet. For DNA sequences the initial dictionary is the four nucleotides: A = 0, T = 1, G = 2, C = 3. Let the text be "ATGATCATGAG" (11 symbols).

Code	Added to the dictionary	Output
4	AT	0 (A)
5	TG	1 (T)
6	GA	2 (G)
7	ATC	4 (AT)
8	CA	3 (C)
9	ATG	4 (AT)
10	GAG	6 (GA)
—	(final w = G)	2 (G)

Table 9.18: LZW compression of the text "ATGATCATGAG". Each row shows the string added to the dictionary and the code written at that moment.

The output is *0 1 2 4 3 4 6 2*: 8 codes instead of 11 symbols. When "AT" is seen the second and third time, it is written with a single code (4); "GA" is written with a single code (6) when seen the second time. The code (2) of the *w* = G left after the loop is appended at the very end. In real genomes the repeated sequences are very long, so the dictionary entries grow long too.

9.9.5 The ASCII dictionary and the special case

In practice the initial dictionary is all 256 single-byte values (0–255; the first 128 are ASCII); which characters occur in the text is not looked at. For example a = 97, b = 98, c = 99. New strings are numbered from 256. This also removes the need to send the initial dictionary.

Output	Added to the dictionary
97 (a)	256 = aa
97 (a)	257 = ab
98 (b)	258 = ba
256 (aa)	259 = aac
99 (c)	260 = ca
257 (ab)	261 = aba
261 (aba)	262 = abac
99 (c)	263 = cb
98 (b)	(final *w* = b)

Table 9.19: LZW compression with an ASCII dictionary of the text "aabaacababacb" (13 characters).

The text shrinks to 9 codes: 97 97 98 256 99 257 261 99 98. The sixth and seventh rows of the table contain the special case. When "ab" (257) is written, 261 = "aba" is added to the dictionary. The next piece is exactly "aba", and 261 is used immediately. When the decompressor reads 261, its dictionary still has codes only up to 260.

The decompression code handles the special case with a single conditional expression:

```
String lzwDecode(List<Integer> codes) {
  List<String> dict =
      initialDict();      // 0..255: byte values
  String w = dict.get(codes.get(0));
  StringBuilder s = new StringBuilder(w);
  for (int i = 1; i < codes.size(); i++) {
    int k = codes.get(i);
    String e = k < dict.size()
        ? dict.get(k)
        : w + w.charAt(0); // special case
    s.append(e);
    dict.add(w + e.charAt(0));
    w = e;
  }
  return s.toString();
}
```

- **Start:** the dictionary is filled with the 256 single-byte values (0–255; the first 128 are ASCII). The first code is always a single character; it is decoded directly and written to the output.
- **Decoding:** if `k < dict.size()`, the code is defined and `e = dict.get(k)`. Otherwise it is the special case and `e = w + w.charAt(0)`.
- **Adding:** at each step `w + e.charAt(0)` is appended to the end of the dictionary; in a `List` the code of the new entry is automatically the next index.

`k`	In dictionary?	`e`	Added (`w + e[0]`)	Output (cumulative)
97	ASCII	a	—	a
97	ASCII	a	256 = a + a = aa	aa
98	ASCII	b	257 = a + b = ab	aab
256	yes	aa	258 = b + a = ba	aabaa
99	ASCII	c	259 = aa + c = aac	aabaac
257	yes	ab	260 = c + a = ca	aabaacab
261	no (largest is 260)	ab + a = aba	261 = ab + a = aba	aabaacababa
99	ASCII	c	262 = aba + c = abac	aabaacababac
98	ASCII	b	263 = c + b = cb	aabaacababacb

Table 9.20: Trace of `lzwDecode([97, 97, 98, 256, 99, 257, 261, 99, 98])`.

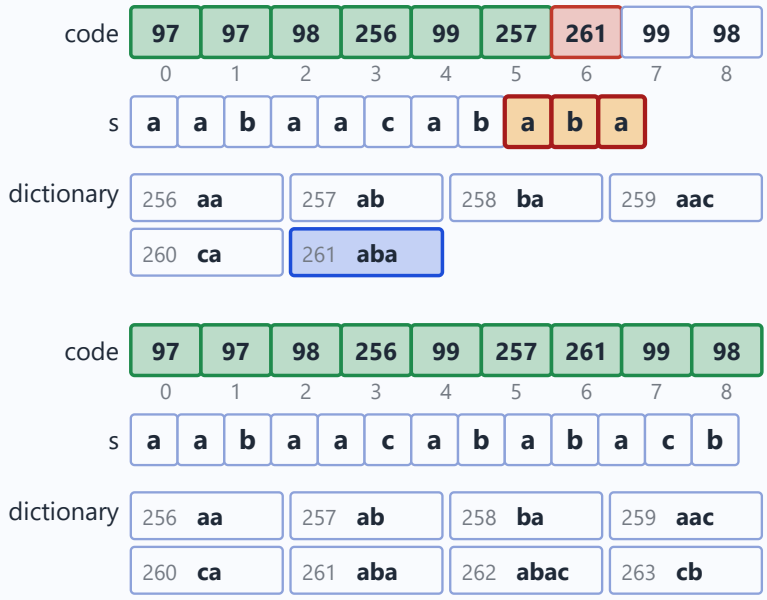


Figure 9.26: On the left, the special case: when 261 is read, the decompressor's dictionary ends at 260; *e = w + w[0]* = "aba". On the right, the final state: 9 codes have turned back into the 13-character text, and the dictionary 256–263 has been built the same as the compressor's.

The code was run in Java and the output "aabaacababacb" was confirmed. The time for decompression is linear in the length of the output.

9.9.6 Dictionary size and code width

How many bits are used to write the codes depends on the size of the dictionary. The 256 single-byte values (0–255; the first 128 are ASCII) can be written with 8 bits. When new entries are added to the dictionary, the codes no longer fit in 8 bits. So LZW implementations start with 9-bit codes and increase the code width by one bit each time the dictionary passes 512, 1024, and so on.

This calculation shows why a reduction in the number of codes is not directly a gain in bits. In the ASCII example the text is 13 bytes, that is, 104 bits. The 9 codes at 9 bits each take $9 \cdot 9 = 81$ bits; the ratio is $81/104 \approx 0.78$. If the "ABABCABABCA" example were encoded with ASCII, 7 codes (63 bits) would be written instead of 11 bytes (88 bits). On short texts the gain is small. On long, repetitive texts a single code stands in for dozens of characters and the gain is large.

As the dictionary grows, two problems arise: the code width (number of bits) increases and the memory taken by the dictionary grows. In practice the dictionary is limited. In GIF the largest code width is 12 bits; the dictionary holds at most $2^{12} = 4096$ entries. When the dictionary is full it is either frozen (no new entries are added) or reset and rebuilt. Resetting prevents old entries from taking up space when the content of the text changes (for example, when moving from the text part of a file to a numeric part).

Common mistake

Two mistakes are very common in LZW. The first is forgetting to write the code of the *w* left after the loop in compression. In the "ABABCABABCA" example this loses the last A (0), and the decompressed text becomes "ABABCABABC". The second is not handling the not-yet-defined code in decompression. `dict.get(k)` throws an `IndexOutOfBoundsException` when `k == dict.size()`. The bug appears only on texts of the cScSc form, so it can slip past simple tests.

9.9.7 LZ78: the ancestor of LZW

In LZ78, on which LZW is based, the dictionary starts empty. The output consists not of codes but of (index, character) pairs: the index of the longest match in the dictionary (0 if there is no match) and the character that follows it. Each pair also adds a new entry to the dictionary.

```
List<String> lz78(String s) {
  Map<String, Integer> dict =
      new HashMap<>();
  List<String> out = new ArrayList<>();
  String w = "";
  for (char c : s.toCharArray()) {
    if (dict.containsKey(w + c)) {
      w = w + c;               // extend
    } else {
      int k = w.isEmpty() ? 0
                          : dict.get(w);
      out.add("(" + k + ", " + c + ")");
      dict.put(w + c, dict.size() + 1);
      w = "";
    }
  }
  if (!w.isEmpty())        // last piece
    out.add("(" + dict.get(w) + ")");
  return out;
}
```

The code is very similar to the LZW code. The differences are these: if `w + c` is not in the dictionary, both the index of `w` and the character `c` are written; then `w` goes back to the empty string. Indices start at 1, because 0 is reserved for the empty prefix. When the text ends, `w` may hold an unwritten piece; this piece is in the dictionary and is written after the loop with its index alone. For example, in the text "aabaacababacb" the final b is entry number 6 in the dictionary and "(6)" is appended to the end of the output. If this line is forgotten, the end of the text is lost. In the example in the table the text ends exactly with a pair, `w` is left empty, and the last line does not run.

Position	`c`	`w + c`	In dictionary?	Output	Added
0	a	a	no	(0, a)	1 = a
1	a	a	yes	—	—
2	b	ab	no	(1, b)	2 = ab
3	a	a	yes	—	—
4	a	aa	no	(1, a)	3 = aa
5	c	c	no	(0, c)	4 = c
6–7	a, b	a, ab	yes, yes	—	—
8	c	abc	no	(2, c)	5 = abc
9–11	a, b, c	a, ab, abc	yes ×3	—	—
12	b	abcb	no	(5, b)	6 = abcb

Table 9.21: Trace of `lz78("aabaacabcabcb")`. A "—" marks steps where no output is written.

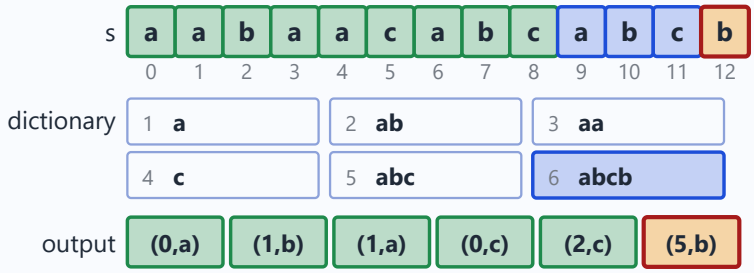


Figure 9.27: The final state of LZ78: the text has been split into the pieces a | ab | aa | c | abc | abcb, six pairs have been written, and the dictionary has grown to entries 1–6.

The text becomes 6 pairs: (0, a) (1, b) (1, a) (0, c) (2, c) (5, b). The pieces are a, ab, aa, c, abc, and abcb; each piece is a piece already in the dictionary extended by one character.

Welch's change comes down to a single idea: start the dictionary with all the single characters. In LZ78, when a character is seen for the first time a pair (0, c) is needed, because it is not in the dictionary. In LZW every single character is in the dictionary from the start. So there is no need to write characters in the output; only codes are written, and the character that follows each code is recovered from the first character of the next code.

Lempel and Ziv's other method, LZ77 from 1977, keeps no dictionary. Instead it looks back in a sliding window of the last few kilobytes and writes repeats as (backward distance, length) pairs. The LZ77 family is the basis of the DEFLATE format used by ZIP and gzip, and of PNG images.

9.9.8 Comparing RLE and LZW

	RLE	LZW
Redundancy captured	consecutive repeats of the same symbol	repeated strings (even if not consecutive)
Extra structure	none	dictionary (hash table or trie)
Time	*O(n)*	*O(n)* (dictionary access *O(1)* on average)
Worst case	output doubles if there are no repeats	little gain on short or random data, code width grows
Use	fax, BMP/PCX, simple graphics	GIF, TIFF, PDF, Unix `compress`

Table 9.22: Comparison of RLE and LZW.

Both methods are lossless and both run in linear time; for LZW this requires keeping the dictionary as a trie or a table keyed by (code, character), not with String concatenation. The structure of the data decides the choice. On images with long flat areas RLE stands out for its simplicity. In natural-language text or programs, consecutive repeats are few, but repeated words and patterns are plentiful; here LZW is far more effective. The pigeonhole principle says that both will enlarge some inputs: "abababab" for RLE, and short random data for LZW, are of this kind.

9.10 Common mistakes

The mistakes mentioned throughout the chapter are collected here. Each one arises from a small detail in the code, yet produces a wrong result, a crash, or an infinite loop.

- **Loop bound.** Writing `i < n - m` in brute force skips the last shift ($s = n - m$). The correct form is `i <= n - m`. If the pattern occurs at the very end of the text, it is not found.
- **Defining LPS wrongly.** The pattern itself is not counted: *lps("aaaa") = [0, 1, 2, 3]*, not *[1, 2, 3, 4]*. In KMP, `lps[q - 1]` is used at a mismatch, not `lps[q]`.
- **Negative shift.** In Boyer-Moore $j - last[c]$ can be zero or negative: this happens when the last occurrence of c is to the right of j. If `max(1, …)` is forgotten, the pattern moves backward or the algorithm loops forever.
- **Including the last character in the Horspool table.** If the last character enters the table, its shift is 0 and the pattern stands still. The table is built from *P[0..m−2]* only.
- **Trusting hash equality.** In Rabin-Karp, equal hashes do not mean a match; a candidate window must be verified character by character. In Java `%` can give a negative result: the line `if (th < 0) th += Q;` is needed. With a large Q, `int` overflows; `long` must be used.
- **Multi-digit counts in RLE.** When decoding "12A", reading the count as a single digit is wrong (one '2'?). Digits must be accumulated until a letter arrives, or the counts must be written in separate bytes.

- **The final *w* and the special case in LZW.** Forgetting to write the code of the *w* left after the loop loses the end of the text; the same line is needed in LZ78. Not handling the not-yet-defined code (*w + w[0]*) when decoding causes a crash on texts of the cScSc form.
- **Miscounting comparisons.** A mismatch at an alignment is also a comparison. In brute force every alignment makes at least one comparison; the total is at least $n - m + 1$.

9.11 Summary

- String matching is the problem of finding all shifts at which a pattern P of length m occurs in a text T of length n. The possible shifts are $0 \le s \le n - m$.
- **Brute force** tries every shift from left to right. It has no preprocessing and no extra memory. It is $O(n)$ on average and $O(n \cdot m)$ in the worst case, with $(n - m + 1) \cdot m$ comparisons. Its flaw is that it throws away what it learns at a mismatch.
- **KMP** builds, in $O(m)$, the table where *lps[q]* is the length of the longest proper prefix-suffix of *P[0..q]*. At a mismatch $q = lps[q - 1]$ is set and the text pointer never moves back. It is $O(n + m)$ even in the worst case; the amortized argument gives at most $2n$ comparisons.
- **Boyer-Moore** compares from right to left and shifts by the larger of *s += max(1, j − last[c])* (bad character) and the good-suffix rule. It is $O(n/m)$ in the best case and the fastest in practice. Horspool is a one-table simplification that shifts according to the last character of the alignment.
- **Rabin-Karp** updates a polynomial rolling hash in $O(1)$. If the hashes are equal the window is verified. The expected time is $O(n + m)$ and the worst time $O(n \cdot m)$. It can search for many patterns of the same length in one pass.
- No lossless compression method can shrink every input (the pigeonhole principle). The compression ratio is the compressed size divided by the original size.
- **RLE** writes each run as a (count, symbol) pair and takes $O(n)$. It doubles the data if there are no repeats; a flag byte reduces this loss. It is effective on images with flat areas.
- In **LZW**, if $w + c$ is in the dictionary, w is extended; otherwise the code of w is written and $w + c$ is added to the dictionary. The decompressor builds the dictionary itself and stays one step behind. When an undefined code is read, $e = w + w[0]$. LZ78 differs in that its dictionary starts empty and it writes (index, character) pairs.

9.12 Key terms

string alphabet substring pattern string matching prefix suffix brute force (naive) Knuth-Morris-Pratt (KMP) LPS table amortized analysis Boyer-Moore algorithm bad character rule good suffix rule Rabin-Karp algorithm rolling hash compression run-length encoding (RLE) Lempel-Ziv-Welch (LZW)

Further terms defined in these notes: shift, proper prefix, last-occurrence table, Horspool shift table, spurious hit, polynomial hash, lossless and lossy compression, compression ratio, run, flag byte, LZ78.

9.13 Test Yourself

1. What is the LPS table of the pattern "abaabab"?

   Answer

   [0, 0, 1, 1, 2, 3, 2]. At $q = 3$ the longest prefix-suffix piece of "abaa" is "a". At $q = 5$, "aba" (3) is found for "abaaba". At $q = 6$, "abaa" and "abab" mismatch; k falls back to $lps[2] = 1$ and "ab" (2) is found.

2. What value does `bruteForce("aaaab", "ab")` return and how many comparisons does it make? How many comparisons does Boyer-Moore make with the bad-character rule alone?

   Answer

   It returns 3. At $s = 0, 1, 2$ brute force matches a and mismatches b ($3 \times 2 = 6$), and at $s = 3$ both characters match (2): 8 in total. At each alignment Boyer-Moore compares the b at the right end with the a in the text; since $last[a] = 0$, it shifts by 1. That is 1 comparison each at $s = 0, 1, 2$ and 2 at $s = 3$: 5 in total.

3. In a KMP search for the pattern "aabaab", "aabaa" has matched ($q = 5$) and the next text character is not b. What value does q take, and by how many positions has the pattern shifted?

   Answer

   The LPS table is *[0, 1, 0, 1, 2, 3]*. $q = lps[4] = 2$: the "aa" at the end of "aabaa" is also a prefix. The shift is $5 - 2 = 3$. Then the same text character is compared with $P[2]$ = b; if that mismatches too, $q = lps[1] = 1$.

4. What is the last-occurrence table of the pattern "ANANAS"? In the text "BANANAS ANANAS", what is the first shift with the bad-character rule alone?

   Answer

   $last[A] = 4$, $last[N] = 3$, $last[S] = 5$, and −1 for the others. At $s = 0$, $P[5]$ = S mismatches $T[5]$ = A; the shift is $max(1, 5 - 4) = 1$. At $s = 1$ all six characters match: 7 comparisons in total.

5. What is the Horspool shift table for "ANANAS"? If S lies under the last character of the alignment,

by how much does the pattern shift?

Answer

Only *P[0..4]* = ANANA enters the table: A → *5 − 4 = 1*, N → *5 − 3 = 2*, other → 6. Since S occurs only at the last position, it is not in the table and shifts by 6.

6. Let *B = 10*, *Q = 7*, and the characters digits. We search for "18" in the text "27182818". Which windows become candidates, and which are spurious hits?

Answer

h("18") = 18 mod 7 = 4. The window hashes are: 27 → 6, 71 → 1, 18 → 4, 82 → 5, 28 → 0, 81 → 4, 18 → 4. The candidates are *s = 2, 5, 6*. The "81" at *s = 5* is a spurious hit; *s = 2* and *s = 6* are real occurrences.

7. How is the text "WWWWWWWWWWWWBWWW" (16 characters) written with character RLE? How many bytes do the byte-level formats take, without a flag and with a flag (runs of four or more written with the flag)?

Answer

"12W1B3W": 7 characters, ratio *7/16 ≈ 0.44*. Without a flag: 3 runs × 2 = 6 bytes. With a flag: 3 bytes for the 12 W's, and the B and three W's as they are, 4 bytes: 7 bytes. Since this input has no non-repeating region, the format without a flag is shorter.

8. With the dictionary A = 0, which codes come out when "AAAAAAA" (7 A's) is compressed with LZW? How many times does the decompressor meet the special case when decoding these codes?

Answer

The output is *0 1 2 0*; 1 = AA, 2 = AAA, 3 = AAAA are added to the dictionary. When decoding: 0 → A. 1 is undefined: *e* = A + A = AA. 2 is also undefined: *e* = AA + A = AAA. 0 → A. The special case arises twice; the output is A·AA·AAA·A = 7 A's.

9. Into which pairs does LZ78 turn the string "aaaaaaaaaa" of ten a's?

Answer

(0, a) (1, a) (2, a) (3, a). The pieces are a, aa, aaa, aaaa; their lengths sum to *1 + 2 + 3 + 4 = 10*. Since the piece lengths grow by one at each step, a run of length *n* is written with about $\sqrt{(2n)}$ pairs.

10. In a text of a million characters, a pattern of 10 letters made up of characters that never occur in the text is searched for. About how many comparisons does Boyer-Moore make? What is the minimum number of comparisons for brute force and KMP?

    Answer

    In Boyer-Moore the character at the right end mismatches at every alignment and the pattern jumps 10 positions: about $n/m = 10^5$ comparisons. Brute force and KMP look at every text character at least once: about 10^6 comparisons. In this best case Boyer-Moore does ten times less work.

9.14 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 32: gives Rabin-Karp, matching with finite automata, and the KMP prefix function with proofs; it also touches on suffix arrays.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Sections 5.3 and 5.5: present the automaton version of KMP, Java code for Boyer-Moore and Rabin-Karp, and RLE, Huffman, and LZW compression.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (1997): treats in detail how to build the good-suffix table in $O(m)$ time, the Z algorithm, and bioinformatics applications.
- K. Sayood, *Introduction to Data Compression* (5th ed., 2017): covers dictionary-based methods (LZ77, LZ78, LZW) and the information-theoretic limits of lossless compression.

9.15 References

1. R. S. Boyer, J S. Moore, “A Fast String Searching Algorithm”, *Communications of the ACM* 20(10), 1977.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 32: String Matching).
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014 (Chapter 13).
4. R. N. Horspool, “Practical Fast Searching in Strings”, *Software: Practice and Experience* 10(6), 1980.
5. R. M. Karp, M. O. Rabin, “Efficient Randomized Pattern-Matching Algorithms”, *IBM Journal of Research and Development* 31(2), 1987.
6. D. E. Knuth, J. H. Morris, V. R. Pratt, “Fast Pattern Matching in Strings”, *SIAM Journal on Computing* 6(2), 1977.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 5.3: Substring Search, 5.5: Data Compression).
8. T. A. Welch, “A Technique for High-Performance Data Compression”, *IEEE Computer* 17(6), 1984.

9. J. Ziv, A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding”, *IEEE Transactions on Information Theory* 24(5), 1978.

PART III · CHAPTER 10

Advanced String Algorithms

Text is among the data types computers process most. A dictionary's words are put in order, and an email address typed into a form is validated. A spell checker suggests the correct word closest to a misspelled one. A compression program rearranges the letters of a file before shrinking it. All of these tasks rest on algorithms that work on strings. These notes take up four families of problems in turn. The first is sorting: the lexicographic order of strings, comparison-based sorting, and the LSD and MSD radix sorts, which make no comparisons at all. The second is parsing: regular expressions and their theoretical counterpart, finite state machines. The third is similarity: the Levenshtein distance, its dynamic programming table, and other measures such as the Hamming distance and the longest common subsequence. The fourth is transformation: the suffix array and the Burrows–Wheeler transform. For each algorithm the notes give Java code, a step-by-step trace, and a cost analysis.

LEARNING OUTCOMES

Sorting
Compare strings in lexicographic order; carry out LSD and MSD radix sort step by step and explain the $O(W \cdot (N + R))$ cost.

Parsing
Write and read regular expressions; design and draw a deterministic finite automaton (DFA) and run it on an input.

Similarity
Compute the Levenshtein distance with a dynamic programming table and read the edit sequence back from the table; compare it with the Hamming distance, the LCS, and the longest common substring.

Transformation
Build a suffix array and use it for searching; compute the Burrows–Wheeler transform and invert it.

10.1 Four families of problems

A **string** is a finite sequence of characters drawn from an alphabet. The **alphabet** is the set of characters that may be used and is written Σ. For binary strings $\Sigma = \{0, 1\}$, and for DNA sequences $\Sigma = \{A, C, G, T\}$. In Java, each character of a `String` is a 16-bit `char` value. The size of the alphabet is written R: $R = 128$ for ASCII, or $R = 256$ for its extended form.

The problems posed on strings are many and varied, but most fall into one of four families. Each family answers a different question and uses a different algorithmic technique.

Family	Question	Algorithms	Core technique
Sorting	In what order should the strings be arranged?	lexicographic comparison, LSD and MSD radix sort	comparison, counting
Parsing	Does the string fit a given format? What parts does it consist of?	regular expressions, finite state machines	automaton
Similarity	How close are two strings to each other?	Levenshtein, Hamming, LCS, longest common substring	dynamic programming
Transformation	How can a string be brought into a form that makes searching or compression easier?	suffix array, Burrows–Wheeler transform	sorting, binary search

Table 10.1: The four families of string algorithms.

In the sorting family, strings are arranged as in a dictionary. Methods that compare strings character by character are general-purpose. Radix sort makes no comparisons at all; it treats each character as a digit and distributes the strings into groups. In the parsing family, a search pattern is written as a **regular expression**, and a **finite state machine** that recognizes this expression reads the input character by character. In the similarity family, the difference between two strings is measured by a number. The best known is the Levenshtein distance, the minimum number of single-character edits that turn one string into the other. In the transformation family, the string is rearranged: the **suffix array** speeds up searching by keeping all suffixes in sorted order, while the **Burrows–Wheeler transform** gathers equal letters together and so prepares the text for compression.

The four families share one thing: all of them exploit the *structure* of a string. Lexicographic order uses the left-to-right significance of the characters. A finite state machine summarizes the part of the input read so far in a single state. The Levenshtein distance is built from the distances between prefixes of the two strings. The suffix array and the BWT rest on the order that appears when all suffixes or rotations of a string are sorted. The terms prefix, suffix, and substring occur often in these notes. A **prefix** is a part taken from the beginning of a string and a **suffix** a part taken from its end. A **substring** is any part made of consecutive characters. For “banana”, for example, “ban” is a prefix, “ana” is a suffix, and “nan” is a substring.

10.2 Lexicographic order

In a dictionary, words are arranged in a fixed order: “armut” comes before “elma”, and “elma” before “elmas”. The rule is simple, and computers use it unchanged. The only difference is that the computer learns the order of the letters not from a memorized alphabet but from character codes.

10.2.1 The comparison rule

Definition 10.1

Let a and b be two strings. We say a is lexicographically smaller than b ($a < b$) if one of two cases holds. (1) The first position where the strings differ is some k, and $a[k] < b[k]$. (2) The strings are equal up to the end of a, and a is shorter than b; that is, a is a prefix of b. This order is called the **lexicographic order**.

The rule starts the comparison with the first characters. If they are equal, it looks at the second characters, and if those are equal, at the third. The first differing character decides the result, and no later character is examined. If one string runs out, the shorter one comes first. Table 10.2 shows what decides the outcome in four comparisons.

Comparison	Decided by	Result
apple · banana	$a < b$ (1st character)	apple first
apple · apricot	$p < r$ (3rd character)	apple first
car · cart	“car” ran out	car first
Zebra · apple	'Z' (90) < 'a' (97)	Zebra first

Table 10.2: Four lexicographic comparisons and the character that decides each.

The last row is surprising at first glance: a person would write “apple” before “Zebra”. The computer, however, finds the code of capital Z (90) smaller than the code of lowercase a (97) and takes no separate account of whether a letter is upper or lower case.

Lexicographic order is a **total order**: of any two different strings, one is smaller than the other, and the order is transitive. Strings can therefore be sorted like integers, and every sorting algorithm works on strings too. Only the cost of a comparison is no longer constant. Two integers are compared in a single operation. Two strings, in the worst case, require as many character comparisons as the length of the shorter one.

10.2.2 Code trace: apple and apricot

A Java method that applies the rule directly is shown below. If the method returns a negative number, *a* comes first; if it returns a positive number, *b* comes first; zero means the two strings are equal. Java's `String.compareTo` works the same way.

```
int compare(String a, String b) {
  int n = Math.min(a.length(), b.length());
  for (int k = 0; k < n; k++) {
    char x = a.charAt(k), y = b.charAt(k);
    if (x != y)
      return x - y;   // first difference
  }
  return a.length() - b.length();
}
```

`n` is the length of the shorter string, so the loop can compare at most that many characters. On each iteration, the `k`-th characters of the two strings are read as `x` and `y`. If the characters differ, `x - y` is returned. Subtracting two `char` values in Java yields the difference of their character codes, and the sign tells which character comes first. If the loop ends without finding a difference, the shorter string is a prefix of the longer one. In that case `a.length() - b.length()` is returned: the shorter string comes first, and the result is zero if the lengths are equal.

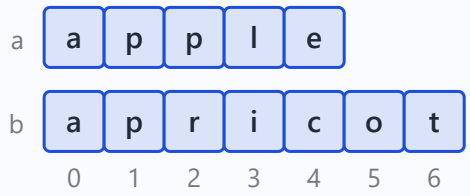


Figure 10.1: The start of the call `compare("apple", "apricot")`. `n = min(5, 7) = 5`, so at most five characters can be compared.

k	x	y	Comparison	Result
0	'a' (97)	'a' (97)	equal	move on to the next character
1	'p' (112)	'p' (112)	equal	move on to the next character
2	'p' (112)	'r' (114)	different	`return 112 - 114 = -2`

Table 10.3: Trace of `compare("apple", "apricot")`.

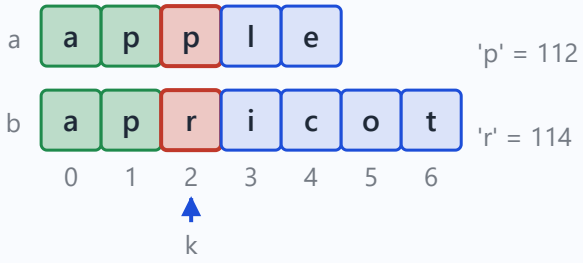


Figure 10.2: The first differing character is at position 2: 'p' (112) versus 'r' (114). The result is *−2 < 0*, so "apple" comes before "apricot".

Because the result is negative, "apple" comes before "apricot". The comparison ended after three characters; the last two letters of "apple" and the last four letters of "apricot" were never examined. The same method returns the following values on other inputs. For `compare("car", "cart")` the first three characters are equal and the shorter string runs out, so the result is *3 − 4 = −1*. `compare("Zebra", "apple")` returns *90 − 97 = −7* at the first character. `compare("kale", "kaba")` returns *108 − 98 = 10* at the third character; the positive result shows that "kaba" comes first.

The cost of the method is proportional to the number of characters read up to the first differing one. In the worst case the two strings share a long common prefix and the cost is *O(min(|a|, |b|))*. No extra memory is used.

Common mistake

Assuming the return value can only be −1, 0, or 1. `compareTo` and `compare` return the difference of character codes or the difference of lengths: values such as −2, −7, or 10 are perfectly normal. The result should therefore be tested with `< 0`, not with `== -1`.

10.2.3 Character codes and Turkish

To a computer, the "alphabet" is the table formed by character codes: ASCII, or Unicode, which contains it. In this table all capital letters (A = 65 … Z = 90) come before all lowercase letters (a = 97 … z = 122). Digits (0 = 48 … 9 = 57) come even before the letters. The Turkish letters ç, ğ, ı, ö, ş, ü have codes outside the ASCII table: for example ç = 231, ğ = 287, ı = 305, ş = 351. Sorted by code, they therefore all come after z.

As a result, sorting with `compareTo` differs from Turkish dictionary order. When "şeker" is compared with "zeytin", 'ş' (351) is greater than 'z' (122), so "zeytin" comes first. In a Turkish dictionary, ş comes before z. Likewise "Zebra" comes before "apple", and "Çorum" comes after "Zonguldak". When Turkish dictionary order is needed, Java's `java.text.Collator` class is used with the Turkish locale (`Collator.getInstance(new Locale("tr", "TR"))`). This class takes the difference between upper

and lower case and the position of the Turkish letters in the alphabet into account. To remove only the upper/lower-case difference, `compareToIgnoreCase` is enough, but that method does not fix the position of the Turkish letters.

Common mistake

Taking code order for alphabetical order. When the array `{"b2", "B10", "ab", "Ab", "a"}` is sorted with `Arrays.sort`, the result is `Ab, B10, a, ab, b2`: strings that start with a capital letter jump to the front. “a” comes before “ab” because it is a prefix of it. For an order that matches the human alphabet, a `Collator` or a normalization that first puts the letters into the same case is needed.

10.2.4 Words and numbers

For two words with a long common prefix, a comparison can take several characters. Figure 10.3 compares the words “EDUCATIVE” and “EDUCATED”. The first six letters (EDUCAT) are equal. At the seventh letter, 'I' (73) meets 'E' (69); because 'E' is smaller, “EDUCATED” comes first. The length of the words does not affect the result: “EDUCATIVE” has nine letters and “EDUCATED” has eight, but the decision is made at the seventh letter.

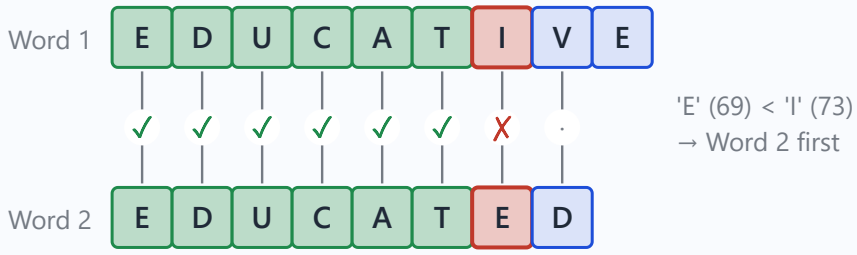


Figure 10.3: EDUCATIVE versus EDUCATED: the first six letters are equal, and at the seventh letter 'E' (69) < 'I' (73), so EDUCATED comes first.

When numbers are stored as strings, lexicographic order departs from numeric order. The comparison is again done from left to right, character by character. When “11” is compared with “2”, the first characters are '1' and '2'; because '1' is smaller, “11” comes first. The number of digits is not taken into account at all. Figure 10.4 shows the primes between 2 and 29 in two orders.

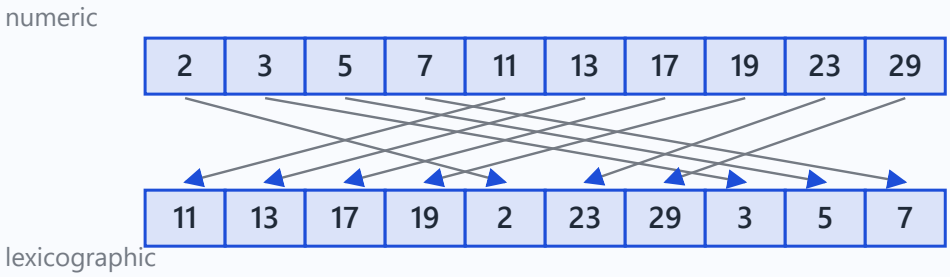


Figure 10.4: The primes from 2 to 29. The top row is numeric order; the bottom row is the lexicographic order of the same numbers as strings: 11, 13, 17, 19, 2, 23, 29, 3, 5, 7.

For the same reason, the strings “123”, “45”, “6”, “789” are in exactly this order lexicographically: their first characters satisfy 1 < 4 < 6 < 7. If numeric order is wanted, there are two ways. The first is to convert the strings to integers and sort them as numbers. The second is to pad all the numbers on the left with zeros to the same length: then “02” < “11”, because now the first characters are '0' and '1'. This is why file names take the form “image01, image02, …, image10”.

Example 10.1

Sort the strings “10”, “9”, “100”, “2” in lexicographic order. The first characters are '1', '9', '1', '2'. “10” and “100”, which start with '1', come first. Of these two, “10” is a prefix of “100” and comes first. Next comes “2”, which starts with '2', and last “9”, which starts with '9'. The result: *“10” < “100” < “2” < “9”*. The numeric order is 2, 9, 10, 100.

10.2.5 Comparison-based string sorting

Now that lexicographic comparison is available, any known sorting algorithm can be applied to strings. The example in the slides uses the simplest method: for each position, the lexicographically smallest of the remaining strings is brought there by swaps.

```
void lexSort(String[] a) {
  int n = a.length;
  for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++)
      if (a[i].compareTo(a[j]) > 0) {
        String tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}
```

The outer loop moves position `i` from left to right. For each `j` after `i`, the inner loop compares `a[i]` with `a[j]`. If `a[i].compareTo(a[j]) > 0`, then `a[j]` is smaller and the two items are swapped. When the inner loop finishes, position `i` holds the smallest of the remaining items. The method resembles selection sort; the difference is that it swaps every time it finds a smaller item, not once at the end. Table 10.4 gives the output of the code on two inputs.

Input	Output
eabcdf, abcde, bcdea, cdeab, deabcg	abcde, bcdea, cdeab, deabcg, eabcdf
banana, orange, grape, apple, kiwi	apple, banana, grape, kiwi, orange

Table 10.4: `lexSort` on two inputs.

The cost has two factors. The two nested loops make $N(N - 1)/2$ comparisons, which is $O(N^2)$. Each `compareTo` call reads at most W characters, where W is the length of the longest string. The total time is $O(N^2 \cdot W)$. Swaps only exchange references; the characters of the strings are not copied. A swap therefore takes constant time, and the extra memory is $O(1)$.

A better comparison sort, such as merge sort, makes $O(N \log N)$ comparisons. For strings that means $O(W \cdot N \log N)$ time. Java's `Arrays.sort(String[])` also uses a variant of merge sort and is stable. Every sort based on comparisons must make at least $\Omega(N \log N)$ comparisons in the worst case. The reason for this lower bound is the number of yes/no questions needed to distinguish each of the $N!$ possible orders: $\log_2(N!) \approx N \log_2 N$. Radix sort does not compare characters; it counts them. The lower bound therefore does not apply to it.

A numerical example makes the difference concrete. Take $N = 10^6$ strings of $W = 10$ characters each. Merge sort makes about $N \log_2 N \approx 2 \cdot 10^7$ comparisons; if each comparison reads at most 10 characters, the upper bound is $2 \cdot 10^8$ character operations. LSD radix sort finishes in $W \cdot (N + R) = 10 \cdot (10^6 + 256) \approx 10^7$ steps. These figures give only orders of magnitude; constant factors and cache behavior change the real running time.

10.3 Radix sort

A comparison sort compares two strings from the beginning every time. For strings with long common prefixes, the same characters are read over and over. **Radix sort** takes a different route: it handles the string character by character and uses each character as a digit of a number. In each pass, the strings are distributed into buckets by a single character position. The distribution is done by counting, not by comparison.

There are two directions. **LSD** (*least significant digit*) starts from the right, that is, from the last character, and moves left. **MSD** (*most significant digit*) starts from the left, from the first character, and sorts each bucket separately. Both methods use the same subroutine in each pass: a stable counting sort on a single character position.

10.3.1 Counting sort and stability

Counting sort sorts items whose keys are small integers. Here the key is the `d`-th character of the string, a code between 0 and 255. The method has three steps: count how many times each key occurs, turn the counts into positions with a prefix sum, and place the items at those positions.

> **Definition 10.2**
>
> A sorting method is **stable** if it preserves, in the output, the relative order that items with equal keys had in the input. For example, if x and y have equal keys and appear in this order in the input, then x also comes before y in the output.

The LSD code in the slides consists of three methods. `lsdSort` manages the passes, `countingSort` performs a stable counting sort on a single character position, and `key` reads the `d`-th character of a string safely:

```
void lsdSort(String[] a) {
  int W = maxLen(a);              // longest string
  for (int d = W - 1; d >= 0; d--)
    countingSort(a, d);           // right to left
}
void countingSort(String[] a, int d) {
  int[] count = new int[256];     // ASCII
  for (String s : a) count[key(s, d)]++;
  for (int r = 1; r < 256; r++) count[r] += count[r - 1];
  String[] aux = new String[a.length];
  for (int i = a.length - 1; i >= 0; i--)   // stable
    aux[--count[key(a[i], d)]] = a[i];
  System.arraycopy(aux, 0, a, 0, a.length);
}
int key(String s, int d) {        // short string → 0
  return d < s.length() ? s.charAt(d) : 0;
}
```

The lines of `countingSort` do the following, in order.

1. **Counting.** The `count` array has 256 cells; each cell corresponds to one character code. The line `for (String s : a) count[key(s, d)]++;` writes into `count[c]` the number of strings whose `d`-th character is c.
2. **Prefix sum.** After the loop `count[r] += count[r - 1]`, `count[c]` is the total number of strings whose key is less than or equal to c. This number gives the position where bucket c *ends* in the output array: bucket c starts at `count[c - 1]` and ends at `count[c] - 1`.
3. **Placement.** The array is scanned *from back to front*. Each string is written into the last free cell of its own bucket: `aux[--count[key(a[i], d)]] = a[i]`. The counter is decremented first, and then the item is written at that position.

4. **Copying back.** `System.arraycopy` copies the sorted `aux` array into `a`.

Placing items from back to front is what guarantees stability. Of the strings in a bucket, the one that comes last in the input is written to the very end of the bucket. The one before it is written to the previous cell. The order within the bucket thus stays the same as in the input. The `key` method returns 0 if `d` exceeds the length of the string. Since 0 is smaller than the code of every letter, a short string goes ahead of the strings that have a character at that position. So “car” comes before “cart”; this corresponds to the rule “a prefix comes first” in lexicographic order.

Example 10.2

Given ten strings: `cab, dab, bad, dad, fad, ace, bee, bed, add, ebb`. All have three characters, so *W = 3*. The first pass of LSD is `countingSort(a, 2)`, that is, a sort on the last character.

1. **Counting.** The last characters are b, b, d, d, d, e, e, d, d, b in turn. `count['b'] = 3`, `count['d'] = 5`, `count['e'] = 2`; the other cells are 0.
2. **Prefix sum.** `count['b'] = 3`, `count['c'] = 3`, `count['d'] = 8`, `count['e'] = 10`. Bucket b occupies positions 0–2, bucket d positions 3–7, and bucket e positions 8–9.
3. **Placement (back to front).** `i = 9`: ebb, bucket b → `count['b']` drops to 2, `aux[2] = ebb`. `i = 8`: add → `aux[7]`. `i = 7`: bed → `aux[6]`. `i = 6`: bee → `aux[9]`. `i = 5`: ace → `aux[8]`. `i = 4`: fad → `aux[5]`. `i = 3`: dad → `aux[4]`. `i = 2`: bad → `aux[3]`. `i = 1`: dab → `aux[1]`. `i = 0`: cab → `aux[0]`.
4. **Result.** `cab, dab, ebb, bad, dad, fad, bed, add, ace, bee`. In each bucket the strings keep their input order: cab, dab, ebb in bucket b; bad, dad, fad, bed, add in bucket d.

The cost of a single pass is easy to see. The counting and placement loops take *N* steps and the prefix-sum loop takes *R = 256* steps. A pass uses *O(N + R)* time and *O(N + R)* extra memory for the `aux` and `count` arrays.

10.3.2 LSD: right to left

`lsdSort` calls `countingSort` *W* times, lowering `d` from *W − 1* to 0. It sorts first by the last character, then by the middle character, and finally by the first character. In each pass, strings with equal keys keep the order established by the previous pass. The figures show the three passes of the previous example.

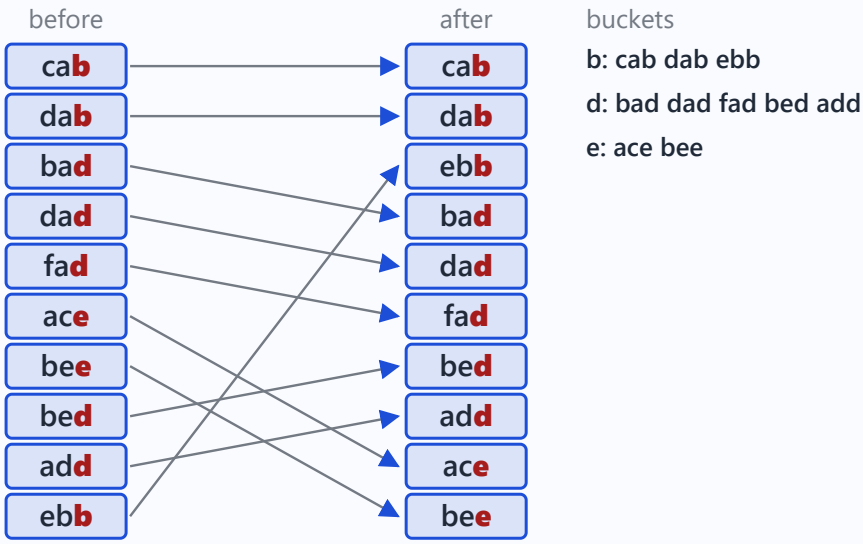


Figure 10.5: Pass 1 (3rd character). The strings are distributed into buckets b, d, and e by their last characters; those in the same bucket (e.g., cab, dab) keep their input order.

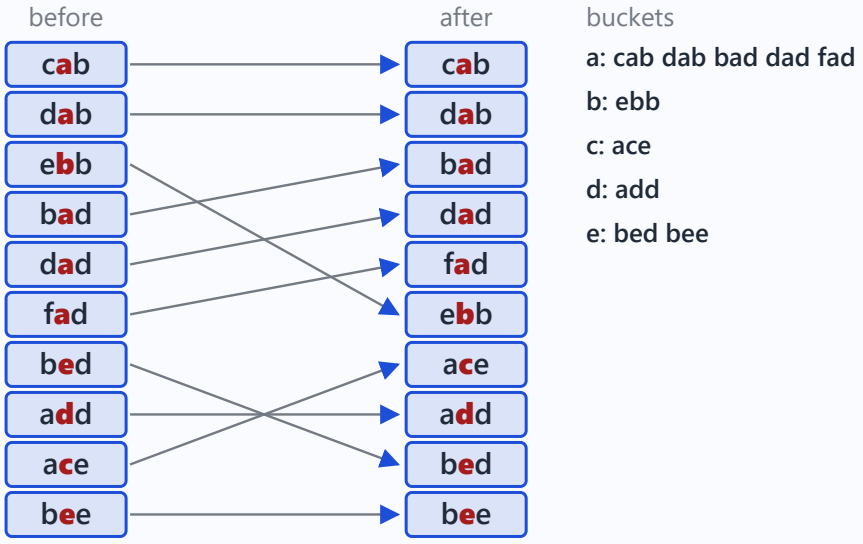


Figure 10.6: Pass 2 (2nd character). In bucket 'a', cab, dab, bad, dad, fad stay in their earlier order: those with last letter b come before those with d.

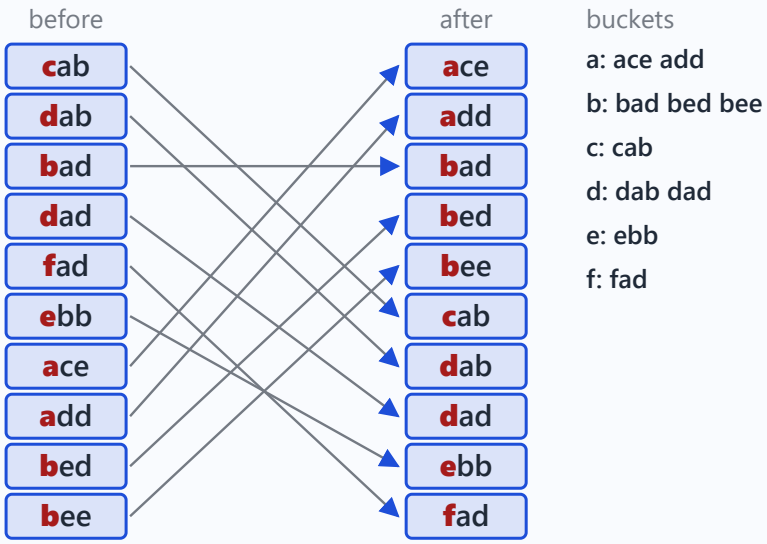


Figure 10.7: Pass 3 (1st character). The array is now lexicographically sorted: ace, add, bad, bed, bee, cab, dab, dad, ebb, fad.

Pass	Key	Array
input	—	cab, dab, bad, dad, fad, ace, bee, bed, add, ebb
1 (`d = 2`)	3rd character	cab, dab, ebb, bad, dad, fad, bed, add, ace, bee
2 (`d = 1`)	2nd character	cab, dab, bad, dad, fad, ebb, ace, add, bed, bee
3 (`d = 0`)	1st character	ace, add, bad, bed, bee, cab, dab, dad, ebb, fad

Table 10.5: The array after each pass of LSD radix sort.

It is worth looking closely at the result of the second pass. The array is now sorted by the last two characters: first those ending in “ab” (cab, dab), then those ending in “ad” (bad, dad, fad), then “bb”, “ce”, “dd”, “ed”, “ee”. The third pass groups by the first character. Strings with the same first letter, for example bad, bed, and bee, stay in the order left by the second pass. That order is already correct with respect to the last two characters.

If the strings have different lengths, `key` returns 0 for missing positions, and a short string behaves as if the smallest character had been appended to it. For `car, cart, ca, cat`, for example, $W = 4$. In the first pass, on the fourth character, the keys of “car”, “ca”, and “cat” are 0 and that of “cart” is 't'; the array becomes `car, ca, cat, cart`. In the pass on the third character, the key of “ca” is 0 and it moves to the front: `ca, car, cart, cat`. The remaining two passes do not change this order. The result is the lexicographic order.

10.3.3 Why is LSD correct?

It seems backwards at first that LSD starts from the least significant character. The key to its correctness is an invariant: after every pass, the array is sorted by the last characters processed so far.

Theorem 10.1 (LSD invariant)

Apply LSD to strings of W characters each, and let every pass be stable. After k passes, the array is lexicographically sorted by the suffixes of the strings consisting of their last k characters. When $k = W$, the array is completely sorted.

Proof. The proof is by induction on k. For $k = 1$, the first pass sorts by the last character, so the claim holds. Assume the claim holds after $k - 1$ passes. Pass k sorts by the k-th character from the end. Take two strings x and y. If their characters at this position differ, the pass puts them in the correct order by this character; this character also decides the order of the length-k suffixes. If their characters at this position are equal, the order of the length-k suffixes is decided by the remaining $k - 1$ characters. By the assumption, x and y are in the correct order by those $k - 1$ characters in the input to this pass. Since the pass is stable, the relative order of x and y, whose keys are equal, does not change. In both cases x and y are in the correct order by their last k characters. □

The proof also shows why stability is indispensable. The last pass does not decide the order of two strings with equal keys; the earlier passes decided it. An unstable subsort ruins that decision. If the placement loop wrote from front to back instead of back to front, for example, every bucket would be filled in reverse. On the ten strings from the slides, this broken version produces `add, ace, bed, bee, bad, cab, dab, dad, ebb, fad`. The first letters are in the right order, but strings with the same first letter are scrambled.

Common mistake

Using an unstable method such as quicksort or heapsort as the subsort in LSD. Even if each pass sorts correctly by its own key, the order established by earlier passes is destroyed; “bed” and “bee” may swap places in the last pass. In counting sort, the placement must run from back to front.

10.3.4 MSD: left to right

MSD radix sort starts from the first character. All strings are first split into buckets by their first characters. Then each bucket is sorted *separately* and recursively by the next character. The recursion stops at a bucket with a single item or when the end of a string is reached. In the slides' notation the end of a string is marked with the character `\0` and is considered smaller than every letter.

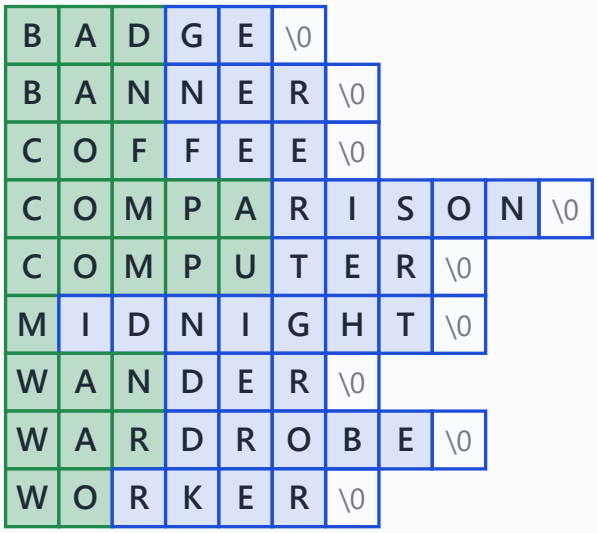


Figure 10.8: Nine words sorted by MSD. The green cells are the characters that must be read to determine each word's place (the distinguishing prefix).

The nine words in Figure 10.8 are split by first character into buckets B, C, M, and W. Bucket M holds only MIDNIGHT; a bucket with one item finishes immediately, and a single character suffices for MIDNIGHT. Bucket B holds BADGE and BANNER. Their second characters are the same (A), so the third character is examined: D < N. Three characters are read for these words. Bucket C holds COFFEE, COMPARISON, and COMPUTER. COFFEE separates at the third character (F < M). COMPARISON and COMPUTER separate only at the fifth character (A < U); both need five characters. In bucket W, WORKER separates at the second character (O), and WANDER and WARDROBE at the third (N < R).

The total number of characters read is the number of green cells: *3 + 3 + 3 + 5 + 5 + 1 + 3 + 3 + 2 = 28*. The nine words have 63 characters in total. MSD finishes the sort having read fewer than half of the characters. LSD, by contrast, would process every word as many times as the length of the longest word, that is, 10 times. This is the real advantage of MSD: it examines only the **distinguishing prefixes**.

The slides do not give code for MSD; they leave the recursive version as an exercise. The version below applies the slides' idea directly. `msdKey` returns 0 if the string has ended and the character code plus one otherwise. Bucket 0 thus means “the string ended here”, and this bucket falls at the very front of the sorted order.

```
void msdSort(String[] a) {
  String[] aux = new String[a.length];
  msdSort(a, aux, 0, a.length - 1, 0);
}
int msdKey(String s, int d) {          // 0 past the end, else code + 1
  return d < s.length() ? s.charAt(d) + 1 : 0;
}
void msdSort(String[] a, String[] aux, int lo, int hi, int d) {
  if (lo >= hi) return;            // bucket of 0 or 1 items
  int[] count = new int[258];
  for (int i = lo; i <= hi; i++) count[msdKey(a[i], d) + 1]++;
  for (int r = 0; r < 257; r++) count[r + 1] += count[r];
  for (int i = lo; i <= hi; i++) aux[count[msdKey(a[i], d)]++] = a[i];
  for (int i = lo; i <= hi; i++) a[i] = aux[i - lo];
  for (int r = 0; r < 256; r++)      /* each bucket on the next char */
    msdSort(a, aux, lo + count[r], lo + count[r + 1] - 1, d + 1);
}
```

The code performs LSD's counting step on a subarray (`lo` – `hi`). The `count` array has 258 cells: one bucket for the end of the string, 256 buckets for the 256 characters, and one extra cell for the prefix sum. After the prefix sum, `count[r]` is the starting position of bucket `r` within the subarray. The distribution step writes the items into the auxiliary array and copies them back. The last loop sorts each character bucket recursively with `d + 1`. Bucket 0 (finished strings) is not sorted again; its members are equal to one another. The code puts the nine words from the slides in the order `BADGE, BANNER, COFFEE, COMPARISON, COMPUTER, MIDNIGHT, WANDER, WARDROBE, WORKER`. It also sorts the input `car, cart, ca, cat, a, ""` as `"", a, ca, car, cart, cat`.

MSD has weaknesses too. Every recursive call creates and scans a `count` array of 258 cells. For small buckets of a few items, this scan is much more expensive than the items themselves. Real implementations therefore switch to insertion sort for small subarrays (say, below 15 items). The second weakness is strings with long common prefixes. If all strings start with the same long prefix, MSD also has to read all of those characters. In a list of URLs, for example, the prefix “https://www.” is shared by every string.

10.3.5 Complexity and choosing a method

LSD makes W passes, where W is the length of the longest string. Each pass handles N strings and R buckets. The total time is $O(W \cdot (N + R))$. If the alphabet is fixed ($R = 256$) and N is much larger than R, the cost drops to $O(W \cdot N)$: the length of the longest string times the number of items. That is the same order as the total number of characters in the input. Since even reading the input takes this long, nothing better can be expected.

Method	Time	Extra memory	Stable
Exchange-based sort + `compareTo`	$O(N^2 \cdot W)$	$O(1)$	no
Merge sort + `compareTo`	$O(W \cdot N \log N)$	$O(N)$	yes

Method	Time	Extra memory	Stable
LSD radix sort	$O(W \cdot (N + R))$	$O(N + R)$	yes
MSD radix sort	$O(N \cdot W)$ worst case	$O(N + W \cdot R)$	yes

Table 10.6: Comparison of string sorting methods (N strings, longest string W characters, alphabet R).

The $W \cdot R$ term in the extra memory of MSD comes from the recursion. The recursion depth is at most W, and one `count` array is alive at each level. The time of MSD is $O(N \cdot W)$ in the worst case, but on random strings it is much less, because only the distinguishing prefixes are read. For small buckets when R is large, however, the cost of the counting array dominates.

The choice depends on the structure of the keys. LSD is ideal when the keys are short and of fixed length: license plates, postal codes, IP addresses, three-letter airport codes. For example, $N = 10^6$ three-letter codes are sorted by LSD in $3 \cdot (10^6 + 256) \approx 3 \cdot 10^6$ steps, whereas merge sort makes about $2 \cdot 10^7$ comparisons. For strings of very different lengths, LSD processes every string as many times as the longest one and does needless work. In that case MSD or a comparison sort is preferred.

Common mistake

Believing that radix sort is faster than comparison sorting in every case. $O(W \cdot N)$ is good when W is small. When W is larger than $\log N$, for example a few hundred strings of a thousand characters each, a comparison sort may do less work. And when R is large (say $R = 65{,}536$ for all of Unicode), the $O(R)$ term in each pass dominates the cost on small arrays.

10.4 Regular expressions

You may need to find all the dates in a text, check whether a phone number typed into a form has the right format, or pick the error lines out of a log file. Writing a separate loop for each of these jobs is tiring and error-prone. Regular expressions let you describe such patterns with a short string.

10.4.1 What is a regular expression?

Definition 10.3

A **regular expression** (*regex* for short) is a string that describes a search pattern. A regular expression describes a set of strings: all strings that “match” the expression are elements of this set.

Regular expressions are used to find parts of a text that match a pattern, to verify that a whole string matches a pattern, to extract parts, or to replace them with other text. Log analysis, form validation, data cleaning, and the lexer in a compiler are foremost among these uses. The lexer is the first compiler stage; it splits source code into pieces such as keywords, variable names, and numbers.

Regular expressions also have a theoretical foundation. **Kleene's theorem** says that every regular expression has an equivalent finite state machine and that the converse also holds. The sets of strings that can be described by regular expressions are called **regular languages**. This link explains how regular expressions are run quickly: the expression is first converted into an automaton, and then the text is scanned with that automaton.

Pattern	Text	Matches
`colou?r`	color colour	color, colour
`gr(a|e)y`	gray grey groy	gray, grey
`\d+`	20.99 TL, 5 items	20, 99, 5
`\bkod\b`	kod kodlama	only “kod”

Table 10.7: Four regular expressions and their matches in a text.

In the first row, `u?` says the letter u may or may not be present; American and British spellings are caught together. In the second row, `(a|e)` requires one of two alternatives; “groy” does not match. In the third row, `\d+` finds every part made of one or more digits. Since the period is not a digit, “20.99” is split into two separate parts. In the fourth row, `\b` is a word boundary; the “kod” inside “kodlama” is not taken because it is not a whole word.

In Java, regular expressions are used through the `Pattern` and `Matcher` classes of the `java.util.regex` package. The `matches`, `replaceAll`, and `split` methods of the `String` class also take regular expressions. Because the backslash is an escape character in a Java string, every backslash in a regular expression is written twice: the expression `\d+` appears in code as `"\\d+"`.

10.4.2 Basic operators

Ordinary characters in a regular expression match themselves: `abc` matches only the substring “abc”. Some characters, however, have special meanings. Table 10.8 gives the most commonly used operators with an example of each. The last two columns of the table show whether the given string *contains* a part that matches the pattern; this is the behavior of `Matcher.find` in Java.

Op.	Meaning	Example	Found in	Not found in
`.`	any single character (except newline)	`a.c`	abc, a7c	ac
`*`	0 or more repetitions of the previous item	`ab*c`	ac, abbc	adc
`+`	1 or more repetitions of the previous item	`ab+c`	abc, abbc	ac
`?`	the previous item occurs 0 or 1 times	`ab?c`	ac, abc	abbc

Op.	Meaning	Example	Found in	Not found in
`[ ]`	one of the characters in the set	`[ck]ar`	car, kar	bar
`^`	start of the string	`^ab`	abc	cab
`$`	end of the string	`ab$`	cab	abc
`( )`	grouping; defines a subexpression	`^(ab)+$`	ab, abab	aba

Table 10.8: Basic operators. In the strings in the "Found in" column, `find` finds a match; in those in the "Not found in" column it does not.

`*`, `+`, and `?` apply only to the item immediately before them. This item can be a single character, a set, or a group built with parentheses. In the expression `ab*c`, the star belongs only to b: an a, then zero or more b's, then a c. So "ac" (zero b's) and "abbc" match, while "adc" does not. If a whole part is to be repeated, the part is grouped: `(ab)+` matches "ab", "abab", "ababab", and so on.

`^` and `$` specify a *position*, not a character. `^ab` matches only an "ab" at the very beginning of the string; the "ab" inside "cab" is not taken because it is not at the start. `^(ab)+$` accepts strings made up entirely of repetitions of "ab" from start to end. "aba" has an extra a at the end and does not match.

Common mistake

Assuming that `ab?c` will find a match in "abbc". `?` allows only one b. In "abbc" there are two b's between the a and the c, and there is no part in which a c follows an a directly. So `find` finds no match.

10.4.3 Special characters and escaping

Besides the basic operators, there are a few commonly used shorthands. These shorthands begin with a backslash and define a character class or a position.

Expression	Meaning
`a`	a single 'a' character
`abc`	exactly "abc"
`{ }`	a specific number of repetitions: `\d{4}` is four digits
`|`	alternatives: `kedi|köpek`
`\b`	word boundary (start or end of a word)
`\d`	a digit: `[0-9]`
`\w`	a word character: `[a-zA-Z0-9_]`
`\s`	a whitespace character: space, tab, newline

Table 10.9: Special characters and shorthands.

The upper-case forms of the shorthands are complements. `\D` matches a non-digit, `\W` a non-word character, and `\S` a non-whitespace character. `\w` is a single character; to match a whole word, write `\w+`.

If you are searching for a character that itself has a special meaning, a backslash is put in front of it. This is called **escaping**. `\.` matches a real period, `\+` a plus sign, and `\(` an opening parenthesis. If the escape is forgotten, the expression catches more than expected: `a.b` also matches “a+b”, because the dot means any character.

One detail in Java matters for Turkish text. By default `\w` covers only ASCII letters. When `\w+` is applied to the word “şeker”, the first match is “eker”; 'ş' is not counted as a word character. For a letter class that also covers Turkish letters, the Unicode property `\p{L}` is used: `\p{L}+` matches all of “şeker”. Another way is to compile the pattern with the `Pattern.UNICODE_CHARACTER_CLASS` flag, which extends `\w` to Unicode as well.

10.4.4 Character classes

Characters written inside square brackets form a **character class**. The class matches *any one* of the characters inside it. A hyphen (-) declares a range: `[a-z]` matches every lowercase letter whose code lies between a and z.

Class	Characters matched
`[abc]`	'a', 'b', or 'c'
`[a-z]`	any lowercase letter (ASCII)
`[A-Z]`	any uppercase letter (ASCII)
`[0-9]`	any digit
`[a-zA-Z_]`	a letter or underscore
`[^abc]`	any character other than 'a', 'b', 'c'
`[^0-9]`	a non-digit character
`[^0-9]+`	“ab” and “cd” in “ab12cd”

Table 10.10: Character classes and their complements.

The `^` sign has two different meanings. Inside square brackets, at the start, it means “not”; outside, it marks the start of the string. Inside a set, `.`, `*`, and `+` are not special: `[.+]` matches a period or a plus. Because a hyphen declares a range, it is written at the start or end of the set when it is itself being searched for: `[-+]`.

Common mistake

Assuming that the class `[a-z]` also covers Turkish lowercase letters. The range is based on character codes; ç, ğ, ı, ö, ş, ü lie outside it. When `[a-z]+` is applied to the word “çiçek”, only the parts “i” and “ek” are found. For Turkish letters, either add them to the set explicitly (`[a-`

`zçğıöşü]`) or use `\p{L}` .

10.4.5 Quantifiers and grouping

Operators that state how many times an item is repeated are called **quantifiers**. `*` , `+` , and `?` are the shortest quantifiers. Curly braces allow finer control.

Expression	Meaning
`a*`	zero or more 'a'
`a+`	one or more 'a'
`a?`	zero or one 'a'
`a{3}`	exactly three 'a'
`a{2,4}`	between two and four 'a'
`a{2,}`	at least two 'a'
`(abc)`	exactly “abc”; captured as a group
`(a|b|c)`	'a', 'b', or 'c'; captured as a group
`(ab)+`	“ab”, “abab”, …

Table 10.11: Quantifiers and grouping.

Parentheses have two jobs. The first is to let a quantifier apply to a whole part. The second is to **capture** the matched part: after a match is found, the text that fell inside the parentheses can be read separately. Groups are numbered from left to right, in the order of their opening parentheses. In Java, group 1 is read with `m.group(1)` , and the whole match with `m.group()` or `m.group(0)` .

Quantifiers are **greedy** by default: they take as long a part as possible. In “aaaaa”, `a{2,4}` takes the longest possible part, “aaaa”. If a `?` is written after the quantifier, the quantifier becomes **lazy** and picks as short a part as possible. The difference is best seen with HTML tags.

Example 10.3

Let the text be `<b>x</b>` . The greedy `<.+>` starts at the first `<` ; `.+` takes everything up to the end of the line. Then it backs off to find a `>` and stops at the *last* `>` . The first match is the whole text: `<b>x</b>` . The lazy `<.+?>` , on the other hand, first takes a single character (b) with `.+?` and then checks whether a `>` follows. It does; the first match is `<b>` . If `find` continues the search, the second match is `</b>` . In both patterns the match starts at the leftmost suitable position (0). Laziness affects only the length of a match, not its start.

10.4.6 Lookahead and lookbehind

Sometimes we look for a part that occurs only in a particular context, but the context itself must not be part of the match. For example, we may want the numbers that come before “TL”, but “TL” must not appear in the result. Lookaround expressions do this job.

Pattern	Name	Match condition	“ab ac ba”
`a(?=b)`	lookahead	'a' if followed by 'b'	0
`a(?!b)`	negative lookahead	'a' if not followed by 'b'	3, 7
`(?<=b)a`	lookbehind	'a' if preceded by 'b'	7
`(?<!b)a`	negative lookbehind	'a' if not preceded by 'b'	0, 3

Table 10.12: Lookaround expressions. The last column gives the positions (from 0) of the a's that match in the text “ab ac ba”.

The text “ab ac ba” has three a's, at positions 0, 3, and 7. The a at 0 is followed by b; the lookahead selects this a. The a at 3 is followed by c and the a at 7 by the end of the text; the negative lookahead selects these two. The a at 7 is preceded by b; the lookbehind selects only that one. The other two a's are preceded by a space or by the start of the text.

Lookaround expressions test a *position* and consume no characters. The text of a match of `a(?=b)` is just “a”; the b does not take part in the match and can be used again in the next search. So in the text “abacab”, `a(?=b)` matches twice, at positions 0 and 4, each time as “a”.

The most common use of lookaround is to test several conditions at once. The expression `^(?=.*\d)(?=.*[A-Z]).{8,}$` checks a password rule. The first lookahead tests that there is a digit somewhere in the string; the second, that there is an uppercase letter. Because neither consumes characters, the position is still at the start of the string. The last part, `.{8,}$`, demands that the string have at least 8 characters. “Parola12” satisfies all three conditions. “parola12” has no uppercase letter and “PAROLAXY” has no digit; “Pa1” is too short.

10.4.7 Validation expressions

Regular expressions are written most often to check the format of data coming from users. Table 10.13 gives three examples. All begin with `^` and end with `$`; the *whole* string, not just a part of it, is required to fit the format.

Purpose	Regular expression	Matches	Does not match
Email	`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`	ali.veli@giresun.edu.tr	ali@tr; a b@x.com
Phone	`^(\+\d{1,3}[- ]?)?\d{10}$`	+90 5321234567; 5321234567	0532 123 45 67

Purpose	Regular expression	Matches	Does not match
URL	`^https?:\/\/[^\s/$.?#].[^\s]*$`	https://giresun.edu.tr	ftp://x.com; http:// x

Table 10.13: Validation expressions and test strings.

The email expression has three parts. `[a-zA-Z0-9._%+-]+` is the user name: a part of at least one character made of letters, digits, and a few symbols. Then comes the @. `[a-zA-Z0-9.-]+` is the domain name. `\.[a-zA-Z]{2,}` is a real period and an extension of at least two letters. “ali@tr” does not match because there is no period and extension after the @. “a b@x.com” does not match because the space is not in the user-name set.

In the phone expression, `(\+\d{1,3}[- ]?)?` is the optional country code. The group consists of a plus, one to three digits, and an optional hyphen or space; the `?` at the end makes the whole group optional. Exactly 10 digits must then follow. “0532 123 45 67” does not match: it has 11 digits and also has spaces inserted.

In the URL expression, `s?` accepts both “http” and “https”. Then comes `:\/\/`. In Java the slash is not special and needs no escaping; this spelling, which comes from other languages, is harmless all the same. `[^\s/$.?#]` says that the first character cannot be a space or one of these symbols. The last part is any sequence of characters containing no whitespace. “ftp://x.com” does not match because the scheme is neither http nor https. “http:// x” does not match because the first character after // is a space.

These expressions are practical approximations; they do not cover the full standards. The complete syntax of email addresses is defined in RFC 5322 and the URI syntax in RFC 3986, and both are much broader. In real applications the regular expression does a first filter; the real validation, such as sending a confirmation message to the address, is done by other means.

10.4.8 Grammars and parse trees

Another way to describe the structure of a string is to write a **grammar**. A grammar uses rules to say what smaller parts a larger structure consists of. As an example, consider addresses of the form `http://didit.csail.mit.edu:4949/`:

```
url    ::= 'http://' host
           ':' port '/'
host   ::= word '.' host
         | word
word   ::= [a-z]+
port   ::= [0-9]+
```

The first rule says that a `url` consists of "http://", a host, a colon, a port, and a slash. The second rule is recursive: a host is either of the form "word . host" or a single word. The last two rules define the word and the port with character classes. Applying the grammar to a string produces a **parse tree**.

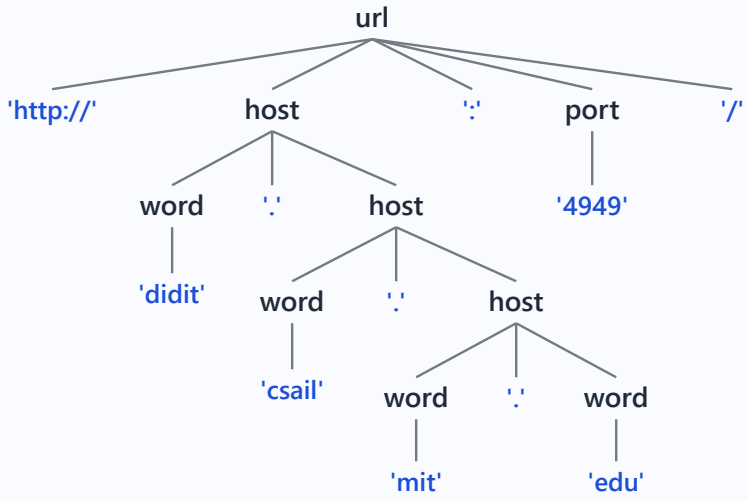


Figure 10.9: The parse tree of the address `http://didit.csail.mit.edu:4949/`. The root is `url`; the `host` rule is expanded recursively three times and produces the words didit, csail, mit, edu.

Although the grammar looks recursive, the recursion is at the very end of the rule; it is tail recursion. Such a recursion can be turned into a loop. The same language can therefore also be described by a regular expression: `http://([a-z]+\.)*[a-z]+:[0-9]+/`. This expression does indeed match the example address.

This is not possible for every structure. Balanced parentheses, that is, strings such as "(()())" in which everything opened is closed, cannot be described by a regular expression. The only memory of a finite state machine is its current state, and it cannot count parentheses nested to unbounded depth. Nested structures need a **context-free grammar** and a **parser** that processes it. Compilers for programming languages use this pair: the lexer works with regular expressions, the parser with a context-free grammar.

Common mistake

Trying to parse nested structures such as HTML or JSON with regular expressions alone. Because `<.*>` is greedy, it swallows the whole line "x y" as a single match. The lazy version finds the tags one by one, but it cannot tell which of the nested tags closes which. A parser is used for this job.

10.4.9 Pattern and Matcher in Java

In Java, a regular expression is first compiled with `Pattern.compile`. Compilation turns the expression into an automaton-like structure internally; if the same expression will be applied to many texts, the compiled object is stored and reused. From the compiled pattern, the `matcher` method yields a `Matcher` for a specific text.

```
String input = "Name: Ali Veli, Age: 30, Email: ali.veli@example.com";
Pattern p = Pattern.compile("Age: (\\d+),");  // compile
Matcher m = p.matcher(input);
if (m.find()) {                           // find first match
  int age = Integer.parseInt(m.group(1)); // group 1: "30"
  System.out.println("Age: " + age);
}
```

The code extracts the age from a one-line record. The pattern `"Age: (\\d+),"` looks for the text "Age: ", then one or more digits, then a comma. Because the digits are in parentheses, they are captured as group 1. `m.find()` looks for the first match in the text and returns `true` if it finds one. `m.group(1)` gives the string "30"; `Integer.parseInt` converts it to a number. The code prints `Age: 30`.

Pattern	`group(1)`	Explanation
Name: (.*?),	Ali Veli	lazy `.*?` : stops at the first comma
`Age: (\d+),`	30	one or more digits
`Email: (.*?)$`	ali.veli@example.com	up to the end of the line

Table 10.14: Three patterns on the same record and the captured group 1.

The laziness in the first row matters. If the greedy `Name: (.*),` were written, `.*` would stretch to the last comma and the group would be "Ali Veli, Age: 30". In the third row, although `.*?` is lazy, `$` forces it to go to the end of the line.

The two basic methods of `Matcher` answer different questions. `find()` searches for a *part* of the text; each call moves on to the next match. `matches()`, on the other hand, tries to match the *whole* text; it behaves as if `^` had been put at the start of the expression and `$` at the end. `String.matches` is of this second kind. For example, `"abc123".matches("\\d+")` returns `false` because the string does not start with a digit. With the same pattern, `find()` finds the part "123" and returns `true`. Similarly, `"2024".matches("\\d{4}")` is `true` and `"20245".matches("\\d{4}")` is `false`: the fifth digit is extra.

The same job can also be done without a regular expression. The start marker (e.g., "Age: ") is found with `indexOf`, and the part up to the end marker (",") is cut out with `substring`. For a simple extraction, this route can be faster and more readable. As the pattern becomes more complicated, for example as optional parts and alternatives multiply, the regular expression stays shorter and more reliable.

Common mistake

Using nested quantifiers carelessly. When Java's regular expression engine fails to find a match, it goes back (backtracking) and tries other options. In an expression such as `(a+)+$`, there is an exponential number of ways to split the text "aaaa…ab". Backtracking engines may try all of these ways in the worst case, and the program appears to hang. This is called **catastrophic backtracking**. Avoiding nested `+` and `*`, and not forgetting the escape character, is enough in most cases.

10.5 Finite state machines

Imagine a turnstile. When it is locked, dropping in a token unlocks it; when it is unlocked, someone passing through locks it again. How many tokens the turnstile has swallowed in the past does not matter; its behavior is determined only by its current state, locked or unlocked. A finite state machine is the mathematical form of this idea and the simplest model for processing strings character by character.

10.5.1 Formal definition

Definition 10.4

A **finite state machine** (FSM) is defined by five elements: $M = (Q, \Sigma, \delta, q_0, F)$. Q is a finite set of **states**. Σ is the input **alphabet**. δ is the **transition function**: $\delta(q, c)$ gives the state the machine goes to when it reads the symbol c while in state q. $q_0 \in Q$ is the **start state**. $F \subseteq Q$ is the set of **accepting states**. The machine processes a string by reading it one symbol at a time from left to right; when the string ends, it **accepts** the string if it is in a state in F and **rejects** it otherwise.

The machine's only memory is its current state. Because the number of states is finite, the machine can keep only a bounded summary of the input it has read. At first glance this restriction looks like a weakness. Even so, many real problems, such as lexical analysis, network protocols, and pattern search, are solved with this much memory. And because the machine does a single operation per character, it is very fast.

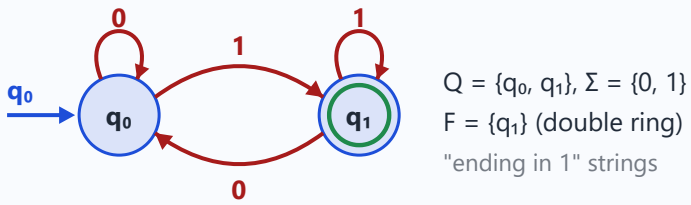


Figure 10.10: A two-state DFA: $Q = \{q_0, q_1\}, \Sigma = \{0, 1\}, F = \{q_1\}$ (double ring). The machine accepts binary strings that end in 1.

In a state diagram, each state is a circle and each transition a labeled arrow. An arrow comes in from outside to the start state; accepting states are drawn with a double ring. In the machine of Figure 10.10, $\delta(q_0, 0) = q_0$, $\delta(q_0, 1) = q_1$, $\delta(q_1, 1) = q_1$, and $\delta(q_1, 0) = q_0$. The meaning of the states is clear: q_1 says "the last symbol read was 1", and q_0 says "the last symbol read was 0, or nothing has been read yet".

Example 10.4

Run the machine on the input "0101". The start is q_0. Read 0: stay in q_0. Read 1: move to q_1. Read 0: return to q_0. Read 1: move to q_1. The input is finished and $q_1 \in F$, so the string is accepted. The string "0110", on the other hand, ends in q_0 after its final 0 and is rejected.

10.5.2 DFA, NFA, and ε-NFA

There are three common kinds of finite state machines. They differ in how many targets the transition function can give for a (state, symbol) pair.

- **Deterministic finite automaton** (DFA): for every (state, symbol) pair there is *exactly one* transition. The machine is in a single state at any moment. It is simple to run, and an input of length n is processed in $O(n)$ time.
- **Nondeterministic finite automaton** (NFA): for a (state, symbol) pair, zero, one, or several transitions may be defined. The machine is in a *set* of states at the same time. If the set contains an accepting state when the input ends, the string is accepted.
- **ε-NFA**: an NFA that also allows empty transitions (epsilon, ε) made without reading a symbol. It can be built directly from a regular expression, piece by piece. This construction is called Thompson's construction.

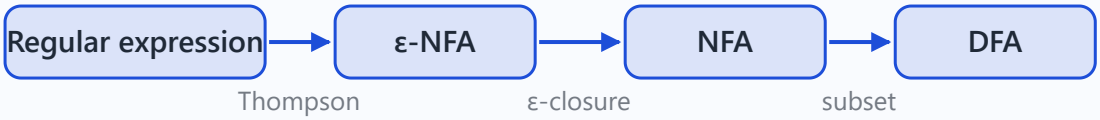


Figure 10.11: From regular expression to DFA: Thompson's construction yields an ε-NFA, ε-closure yields an NFA, and the subset construction yields a DFA.

Theorem 10.2 (Equivalence)

DFAs, NFAs, ε-NFAs, and regular expressions recognize the same class of languages, namely the regular languages. Every NFA can be converted by the **subset construction** into a DFA that recognizes the same language; the number of states of the DFA obtained from an NFA with n states is 2^n in the worst case.

Proof. The idea of the subset construction also supplies the proof. An NFA is in a set of states at any moment. Let each state of the DFA correspond to a *set* of states of the NFA. The start state of the DFA is $\{q_0\}$. When the DFA is in a set S and reads c, it moves to the union of the states that each state in S can reach on c. This transition is unique for every (set, symbol) pair, so the resulting machine is a DFA. The state of the DFA is the exact set of states the NFA could be in after each prefix of the input. The DFA therefore accepts the strings the NFA accepts, and only those. Since a set with n elements has 2^n subsets, the number of states cannot exceed this bound. □

The exponential growth can indeed occur. The NFA that recognizes the strings over $\{a, b\}$ whose third symbol from the end is a has four states. The start state stays in itself on every symbol, or moves to the state “a seen” when it reads an a. Two more symbols are then read and the accepting state is reached. A DFA recognizing this language, however, has to remember all of the last three symbols and needs $2^3 = 8$ states. For “the k-th symbol from the end”, the NFA has $k + 1$ states and the DFA 2^k. In practice this blow-up is rarely seen. `grep`, `lex` / `flex`, and the lexers of compilers use the chain in Figure 10.11: they convert the regular expression to an NFA and, if needed, to a DFA, and scan the text with that automaton.

10.5.3 Simulating a DFA

Running a DFA takes three steps. Move to the start state. For each character, update the state from the transition table. When the string ends, check whether the final state is an accepting state. The transition function is stored in a table: rows are states and columns are symbols. States are numbered from 0, and 0 is the start state.

```
boolean accepts(int[][] next, boolean[] accept,
                String s) {
  int state = 0;                    // start state
  for (char c : s.toCharArray())
    state = next[state][c - '0'];   // one transition
  return accept[state];             // final state accepting?
}
```

The `next` table has size `int[|Q|][|Σ|]`. In this example the alphabet is *{0, 1}*; the expression `c - '0'` turns the character '0' into 0 and '1' into 1 and is used as the column index. The `accept` array records whether each state is an accepting state. The loop does a single table read per character. A string of length *n* is processed in *O(n)* time without backing up; the only extra memory is the `state` variable. The table itself takes *O(|Q| · |Σ|)* space.

Never backing up is an important property. Some regular expression engines backtrack, as in the `(a+)+$` example, and can spend exponential time in bad cases. A DFA reads each character exactly once; it runs in linear time even in the worst case.

10.5.4 Example: a recognizer for “nice”

A machine that recognizes one specific word is the simplest example of a finite state machine. The machine in Figure 10.12 accepts only the string “nice”.

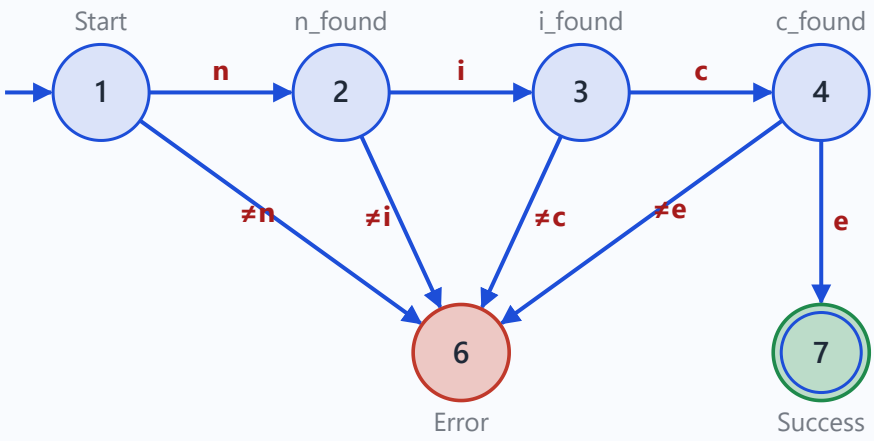


Figure 10.12: A machine that recognizes the word “nice”. From the states Start (1), n_found (2), i_found (3), c_found (4), it advances on the expected character; on e it reaches the Success (7) state. Every unexpected character leads to the Error (6) state.

Each state represents the prefix read so far: 1 the empty prefix (ε), 2 “n”, 3 “ni”, 4 “nic”, and 7 “nice”. If the expected character arrives, the machine moves to the next state. If it does not, for example if a character other than i is read in state 2, the machine goes to the Error state. There is no exit from the Error state; every character keeps the machine there. Such a state is called a **trap state**. In the

diagram the trap state's self-loops are not drawn, but they are assumed to exist so that the machine is a DFA. Likewise, if one more character arrives after the Success state, the machine moves to Error; “nicer” is rejected.

The input “nice” follows the path 1 → 2 → 3 → 4 → 7 and is accepted. The input “nine” follows 1 → 2 → 3 and then reads 'n' as its third character. The expected character is 'c'; the machine goes to 6 and the fourth character keeps it there. The result is rejection.

10.5.5 Example: a DFA over the binary alphabet

A larger DFA has six states: A, B, C, D, E, and F. The start state is A and the only accepting state is D. The transitions are given in Table 10.15 and Figure 10.13. Each cell of the table holds a single state, so the machine is a DFA.

δ	0	1
→ A	E	B
B	B	C
C	B	D
D *	D	A
E	E	F
F	F	A

Table 10.15: Transition table of the six-state DFA. → marks the start state and * an accepting state.

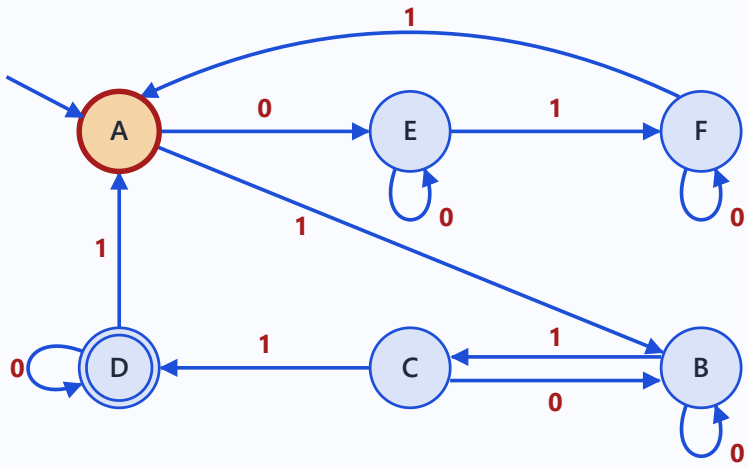


Figure 10.13: A six-state DFA over the binary alphabet. Start A (highlighted), accepting D (double ring).

Example 10.5

Let the input be “10011”.

1. Start: A.
2. Symbol 1 is 1: $\delta(A, 1) = B$.
3. Symbol 2 is 0: $\delta(B, 0) = B$.
4. Symbol 3 is 0: $\delta(B, 0) = B$.
5. Symbol 4 is 1: $\delta(B, 1) = C$.
6. Symbol 5 is 1: $\delta(C, 1) = D$. The input is finished; D is an accepting state and the string is accepted.

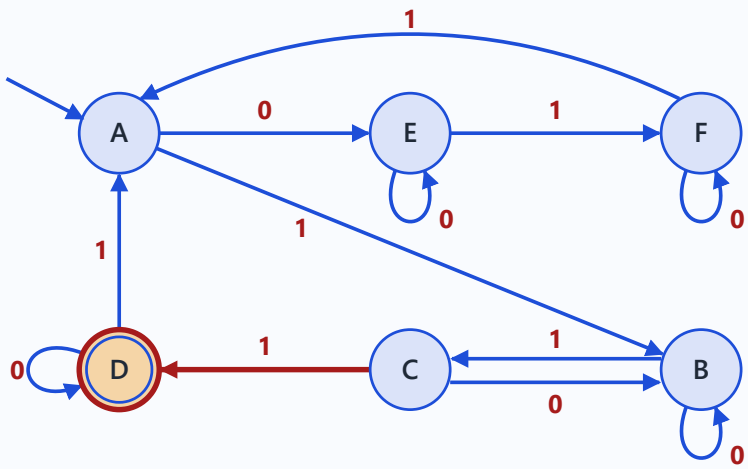


Figure 10.14: After reading “10011” the machine is in D; the last transition $\delta(C, 1) = D$ is shown in red.

The accept decision depends only on the *final* state. The input “1011010” reaches D after the path A → B → B → C → D, but the input is not finished. The next 0 keeps the machine in D, the 1 takes it to A, and the final 0 takes it to E. Since E is not an accepting state, the string is rejected. Having passed through an accepting state along the way changes nothing. Similarly, “111” is accepted via A → B → C → D, while “11” ends in C and is rejected.

The table is easy to give to the `accepts` method: number the states A–F as 0–5, so the table becomes `{{4, 1}, {1, 2}, {1, 3}, {3, 0}, {4, 5}, {5, 0}}`, and in the `accept` array only cell 3 is `true`. The method returns `true` for “10011” and `false` for “1011010”.

10.5.6 The pattern-matching automaton

Finite state machines are also used to search for a pattern inside a text. Let the pattern be "ACACAGA". While the text is read from left to right, the state of the machine shows which prefix of the pattern the end of the text read so far agrees with.

Definition 10.5

For a pattern P of length m, the states of the **pattern-matching automaton** are $0, 1, \ldots, m$. After the first i characters of the text have been read, the state q is the length of the longest prefix of P that is a suffix of the text read. Reaching state $q = m$ means that the pattern ends at that point in the text.

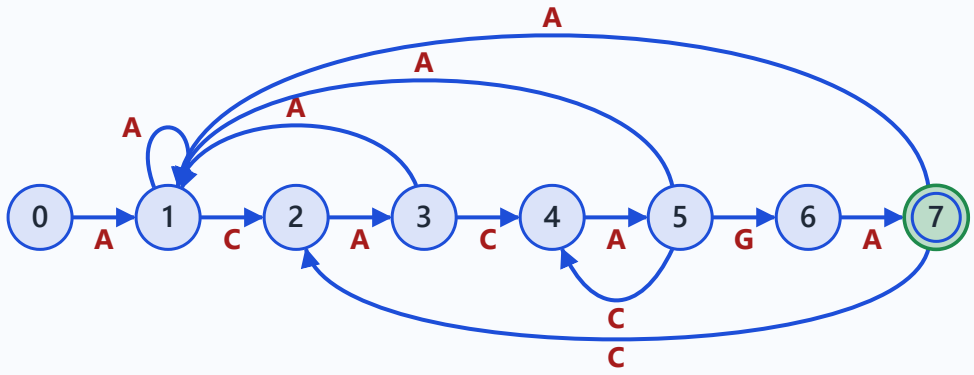


Figure 10.15: The matching automaton for the pattern "ACACAGA". The forward transitions from 0 to 7 are the characters of the pattern. The backward transitions show the return to the longest matching prefix. All transitions that are not drawn go to 0.

The forward transitions are easy: in state q, if the q-th character of the pattern (counting from 0) is read, the machine moves to $q + 1$. The interesting part is the backward transitions. For example, in state 5 (the last five characters read are "ACACA"), let C be read. The expected character is G; the match is broken. But the end of the text read is now "ACACAC", and its suffix "ACAC" is a prefix of the pattern of length 4. The machine returns not to 0 but to 4; part of the match is preserved. All the nonzero backward transitions in the automaton are: from states 1, 3, 5, and 7 on A to 1; from 5 on C to 4; from 7 on C to 2.

i	0	1	2	3	4	5	6	7	8	9
text	G	A	C	A	C	A	C	A	G	A
state	0	1	2	3	4	5	4	5	6	7

Table 10.16: The state of the automaton while reading the text "GACACACAGA". State 7 is reached at character number 9.

This idea is the automaton form of the prefix function in the Knuth–Morris–Pratt (KMP) algorithm. On a mismatch, KMP reads from the prefix function how far to shift the pattern. The automaton instead precomputes this information for every (state, character) pair and writes it into a table. Building the table takes $O(m \cdot |\Sigma|)$ and the search takes $O(n)$. The slides do not show how the automaton is built; the method below does this construction:

```
int[][] buildDfa(String p) {        // pattern DFA, O(m · 256)
  int m = p.length();
  int[][] dfa = new int[m + 1][256];
  dfa[0][p.charAt(0)] = 1;
  for (int x = 0, j = 1; j <= m; j++) {
    for (int c = 0; c < 256; c++)
      dfa[j][c] = dfa[x][c];         // mismatch: as in state x
    if (j < m) {
      dfa[j][p.charAt(j)] = j + 1;   // match: advance
      x = dfa[x][p.charAt(j)];       // restart state
    }
  }
  return dfa;
}
```

The key to the method is the variable `x`: it is the state reached when the part `p[1..j-1]` of the pattern is fed to the automaton. When there is a mismatch in state `j`, the machine knows that "what was read last is the `j`-character prefix of the pattern, followed by the new character". The longest matching prefix is the state reached from the state obtained by dropping the first character of this text, that is, from `x`, on the new character. So row `j` is first copied from row `x`. Then the match transition is written over it as `j + 1`, and `x` is advanced by one step. For "ACACAGA" the method produces exactly the transitions in Figure 10.15.

10.5.7 Code trace: dfaSearch

Once the automaton is built, the search is a few lines:

```
int dfaSearch(int[][] dfa, String t, int m) {
  int q = 0;                      // state
  for (int i = 0; i < t.length(); i++) {
    q = dfa[q][t.charAt(i)];
    if (q == m)
      return i - m + 1;           // match
  }
  return -1;
}
```

The rows of the `dfa` table are states and its columns are character codes; `dfa[q][t.charAt(i)]` gives the next state directly. When `q == m`, the last character of the pattern is at position `i` of the text. The pattern starts at position `i - m + 1`, and this value is returned. If the loop ends without finding a match, `-1` is returned. Let us trace the code with the call `dfaSearch(dfa, "GACACACAGA", 7)`.

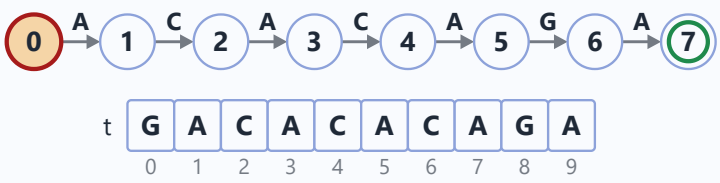


Figure 10.16: Start: the automaton has been built in advance for "ACACAGA", `q = 0`.

`i`	`t[i]`	`q`	**Explanation**
0	G	0 → 0	G does not continue any prefix of the pattern
1	A	0 → 1	next character of the pattern
2	C	1 → 2	next character of the pattern
3	A	2 → 3	next character of the pattern
4	C	3 → 4	next character of the pattern
5	A	4 → 5	next character of the pattern
6	C	5 → 4	not the expected G; the end of "ACACAC", "ACAC", is a prefix
7	A	4 → 5	next character of the pattern
8	G	5 → 6	next character of the pattern
9	A	6 → 7	`q == m`: `return 9 - 7 + 1 = 3`

Table 10.17: Trace of `dfaSearch(dfa, "GACACACAGA", 7)`.

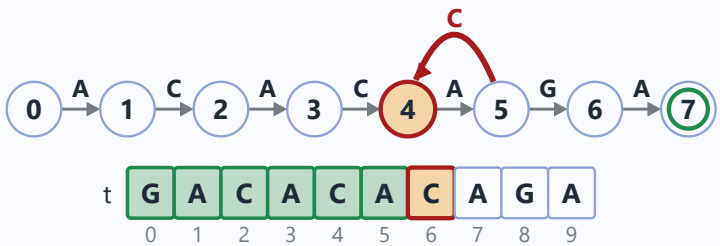


Figure 10.17: `t[6] = 'C'`: the expected character is not G, but `q` is not reset; the transition 5 → 4 is made. There is no going back in the text.

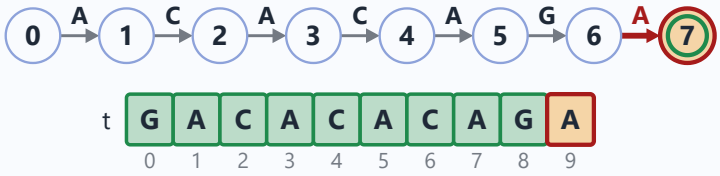


Figure 10.18: With `t[9] = 'A'`, 6 → 7 = `m`: the pattern is found and *9 − 7 + 1 = 3* is returned.

The pattern starts at position 3 of the text: `t[3..9]` = "ACACAGA". The remarkable step is `i = 6`. A naive search method would shift the pattern by one here and go back in the text to compare again. The automaton instead moves to state 4 and never reads any character of the text a second time. A total of ten characters is processed with ten table reads.

Common mistake

Always returning to state 0 on a mismatch. Let the automaton that searches for the pattern "abab" read the text "abaa". The first three characters bring it to state 3. The fourth character a is not the expected b. But the longest suffix of "abaa" that is a prefix of "abab" is "a"; the correct state is 1. A machine that returned to 0 would miss the match starting at 3 in a text such as "abaabab".

10.5.8 Parsing numbers with an FSM

Finite state machines are also used to extract parts from a text. The method below finds and prints the decimal numbers in a text. The machine has three states: START (we are not inside a number), DIGIT (the integer part is being read), and DECIMAL (a dot has been seen and the fractional part is being read). The characters of the number being read are accumulated in a `StringBuilder` buffer.

```
void findNumbers(String s) {
  State st = State.START;
  StringBuilder buf = new StringBuilder();
  for (char c : s.toCharArray()) {
    boolean digit = Character.isDigit(c);
    if (digit) {
      if (st == State.START) st = State.DIGIT;
      buf.append(c);
    } else if (c == '.' && st != State.DECIMAL) {
      st = State.DECIMAL;
      buf.append(c);
    } else if (st != State.START) {  // number ended
      System.out.println(buf);
      buf.setLength(0);
      st = State.START;
    }
  }
  if (st != State.START) System.out.println(buf);
}
```

For each character, the loop applies one of three conditions. If the character is a digit and the machine is in START, it moves to DIGIT; in every case the digit is added to the buffer. If the character is a dot and the machine is not yet in DECIMAL, it moves to DECIMAL and the dot is also added to the buffer. If neither condition holds and the machine is inside a number (anything other than START), the number has ended: the buffer is printed, emptied, and the machine returns to START. After the loop, if the machine is still inside a number, that is, the input ends with a number, the last number is printed too.



Figure 10.19: Input “$20.99 or 5.”; state START, buffer empty.

Character	Branch taken	State	Buffer	Output
$	none (START)	START	“”	
2	digit	DIGIT	“2”	
0	digit	DIGIT	“20”	
.	dot	DECIMAL	“20.”	
9	digit	DECIMAL	“20.9”	
9	digit	DECIMAL	“20.99”	

Character	Branch taken	State	Buffer	Output
␣	number ended	START	""	20.99
o, r, ␣	none (START)	START	""	
5	digit	DIGIT	"5"	
.	dot	DECIMAL	"5."	
(end)	`if` after the loop	—	"5."	5.

Table 10.18: Trace of `findNumbers("$20.99 or 5.")` (␣ stands for the space character).

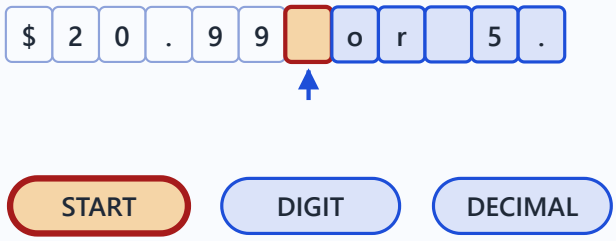


Figure 10.20: The space character is not a number character: the number has ended, 20.99 is printed, and the machine returns to START.



Figure 10.21: The input has ended; because the state is DECIMAL, the buffer is printed. The trailing dot has also joined the number.

The method prints two lines: "20.99" and "5.". The second is a flaw: the period at the end of the sentence has been taken for part of the number. The cause of the flaw is that the machine enters the DECIMAL state without knowing whether a digit will follow the dot. The same design gives unexpected results on other inputs too. In the input "v1.2.3 and 7", when the second dot arrives the machine is already in DECIMAL. The dot condition does not hold and the third branch runs: "1.2" is printed, the dot is discarded, and the machine returns to START. Then "3" and "7" are printed as separate numbers. The version number has been split into two numbers.

The flaw can be fixed in two ways. The first is to delete the last character if the buffer ends with a dot before printing. The second is to include the dot in the number only if a digit follows it; for that, a new state is added to the machine meaning "dot seen but no fractional digit yet". The version below follows the first way and accepts the dot only after a digit:

```
void findNumbersFixed(String s) {
  State st = State.START;
  StringBuilder buf = new StringBuilder();
  for (char c : (s + " ").toCharArray()) {        // append a separator
    if (Character.isDigit(c)) {
      if (st == State.START) st = State.DIGIT;
      buf.append(c);
    } else if (c == '.' && st == State.DIGIT) {     // dot only after a digit
      st = State.DECIMAL;
      buf.append(c);
    } else if (st != State.START) {
      if (buf.charAt(buf.length() - 1) == '.')
        buf.setLength(buf.length() - 1);     // trailing dot is not part of the number
      System.out.println(buf);
      buf.setLength(0);
      st = State.START;
    }
  }
}
```

Appending a space to the end of the input makes the separate `if` after the loop unnecessary: the last number also ends with the space and is printed in the same branch. The corrected method prints “20.99” and “5” for “$20.99 or 5.”. For “v1.2.3 and 7” the behavior does not change: “1.2”, “3”, and “7”. Accepting the dot only after a digit has a side effect: a number written without its leading zero, such as “.5”, is read as “5”. Which behavior is right depends on the rules of the application. That is the real benefit of FSM design: every rule corresponds to a state or a transition and is plainly visible.

10.6 Edit distance

Someone types “algortima” into a search box and the search engine suggests “algoritma”. For this, a number that measures how close two words are to each other is needed. An equality check says only “same” or “different”; it does not distinguish “one letter missing” from “a completely different word”. The edit distance makes this distinction.

10.6.1 The Levenshtein distance

Definition 10.6

The **Levenshtein distance** between two strings is the minimum number of single-character edits needed to transform the first string into the second. Three edits are allowed, and each counts as 1: **insertion**, **deletion**, and **substitution**.

The measure takes its name from the Soviet mathematician Vladimir Levenshtein, who defined the concept in 1965; the English translation of his paper appeared in 1966. Its general name is **edit distance**. This name also covers the variants in which the edits are given different costs. The areas of use are wide: spell checking and autocorrection, predictive typing on keyboards, comparison of DNA sequences, and fuzzy search, which tolerates small spelling differences.

An example can be given for each of the three edits. “sittin” → “sitting”: a g is *inserted* at the end. “shine” → “shin”: the final e is *deleted*. “kitten” → “sitten”: k is *substituted* by s. Using these three operations one after another, “kitten” is transformed into “sitting” in three steps: kitten → sitten (k → s) → sittin (e → i) → sitting (+g).

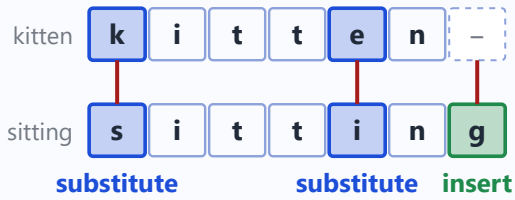


Figure 10.22: Alignment of “kitten” and “sitting”: two substitutions (k → s, e → i) and one insertion (g). The distance is 3.

There is no transformation shorter than three. To see this, there is no need to try every possibility; the dynamic programming table proves it. Still, a short argument can be given. “sitting” is one letter longer than “kitten”, so at least one insertion is needed. The first letters (k and s) differ and cannot be fixed by aligning them with other letters; one more substitution is needed. The e/i difference in the middle requires a third operation.

The Levenshtein distance is a **metric** in the mathematical sense. It has three properties. The distance of a string to itself is zero: $d(a, a) = 0$, and the distance is positive for different strings. The distance is symmetric: $d(a, b) = d(b, a)$, because every insertion becomes a deletion when reversed, and every substitution becomes a substitution again. The triangle inequality holds: $d(a, c) \le d(a, b) + d(b, c)$; applying the edits from a to b and then those from b to c one after the other gets from a to c.

10.6.2 Edit examples

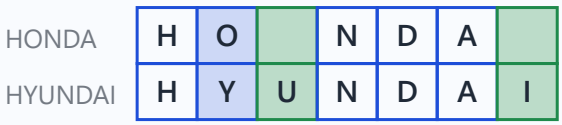


Figure 10.23: HONDA → HYUNDAI: substitute O → Y, insert U, insert I; $d = 3$.

The easiest way to see a transformation is to align the two strings one under the other. Equal letters are written in the same column. A gap is placed opposite a letter that is in one string but not in the other. In Figure 10.23, the letters H, N, D, A of HONDA are aligned with their counterparts in HYUNDAI. O is substituted by Y; there is a gap opposite U and I, so these two letters are inserted. A total of three edits is made.

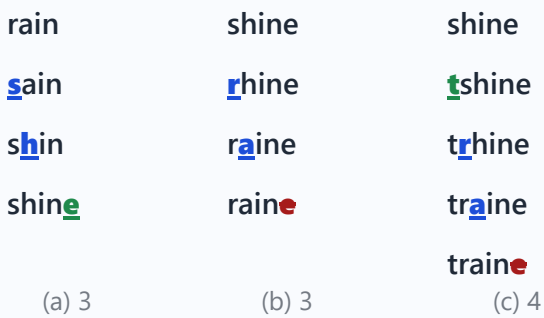


Figure 10.24: (a) rain → sain → shin → shine; (b) shine → rhine → raine → rain (e is deleted); (c) shine → tshine → trhine → traine → train (e is deleted). The underlined blue letter is the one substituted, the underlined green letter is the one inserted, and the orange letter with a strike-through is the one to be deleted.

(a) and (b) are inverses of each other. From “rain” to “shine” takes two substitutions and one insertion; going back from “shine” to “rain” makes the same two substitutions in reverse and the insertion becomes a deletion. Both directions take 3 edits; this is a concrete example of symmetry. In (c), “shine” and “train” have the same length, but the distance is 4: t is inserted, s → r and h → a are substituted, and the final e is deleted.

10.6.3 The dynamic programming recurrence

It is impossible to try all edit sequences; their number grows exponentially with the string length. The solution is to split the problem into small subproblems on prefixes. Let string a have m characters and string b have n characters.

Definition 10.7

$D[i][j]$ is the minimum cost of transforming the first i characters of a into the first j characters of b ($0 \le i \le m$, $0 \le j \le n$). The distance sought is $D[m][n]$.

Theorem 10.3 (Levenshtein recurrence)

> D[i][0] = i and D[0][j] = j
> if $a_i = b_j$ then D[i][j] = D[i−1][j−1]
> otherwise D[i][j] = 1 + min(D[i−1][j], D[i][j−1], D[i−1][j−1])

Here a_i is the i-th character of a (counting from 1). The three terms of the minimum correspond to deletion, insertion, and substitution, in that order.

Proof. The boundary values are clear: from a prefix of i characters to the empty string takes i deletions, and from the empty string to a prefix of j characters takes j insertions. In the general case we look at what the best transformation does with the last characters. There are three possibilities. (1) a_i is deleted; the remaining work is to turn the first $i - 1$ characters of a into the first j characters of b: cost $1 + D[i-1][j]$. (2) b_j is inserted at the end; the remaining work is to turn the first i characters into the first $j - 1$ characters: $1 + D[i][j-1]$. (3) a_i is aligned opposite b_j. If the characters are equal, this step is free and the cost is $D[i-1][j-1]$. If they differ, a substitution is needed: $1 + D[i-1][j-1]$. The best transformation uses one of these three options, so the value is the smallest of the options. It can also be shown that when the characters are equal, the free diagonal is always the best: since neighboring cells of D differ by at most 1, the other two options cannot give a value smaller than $D[i-1][j-1]$. □

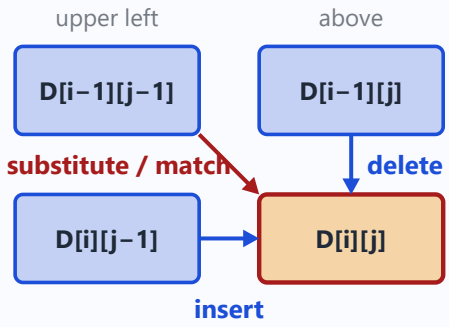


Figure 10.25: *D[i][j]* is computed from three neighbors: upper left (substitution or match), above (deletion), and left (insertion).

This is a typical example of the **dynamic programming** (DP) method. Each subproblem is solved once and stored in a table of size $(m + 1) \times (n + 1)$; the answer is in the bottom-right corner of the table. Because a cell depends only on its upper, left, and upper-left neighbors, the table can be filled row by row, left to right within each row. A constant amount of work is done per cell. The time is $O(m \cdot n)$ and the memory is $O(m \cdot n)$. If only the distance is needed, the memory can be reduced to two rows, that is, $O(\min(m, n))$.

10.6.4 The DP table: CARS → PAIRS

Write a = CARS along the rows and b = PAIRS along the columns. Row 0 and column 0 of the table correspond to the empty prefix (ε).

Example 10.6

1. **Initialization.** The first row is 0, 1, 2, 3, 4, 5: the number of insertions from the empty string to the prefixes P, PA, ... The first column is 0, 1, 2, 3, 4: the number of deletions from the prefixes C, CA, ... to the empty string.
2. **Row 1 (C).** C equals none of the letters of PAIRS. *D[1][1] = 1 + min(0, 1, 1) = 1*: C is substituted by P. The rest of the row grows by one at each step: 1, 2, 3, 4, 5.
3. **Row 2 (A).** *D[2][1] = 1 + min(1, 2, 1) = 2*. Since A = A, *D[2][2] = D[1][1] = 1*; no cost is added. A ≠ I: *D[2][3] = 1 + min(above 3, left 1, upper left 2) = 2*. The row becomes 2, 1, 2, 3, 4.
4. **Row 3 (R).** Since R = R, *D[3][4] = D[2][3] = 2*. The row becomes 3, 2, 2, 2, 3.
5. **Row 4 (S).** Since S = S, *D[4][5] = D[3][4] = 2*. The row becomes 4, 3, 3, 3, 2.

	ε	**P**	**A**	**I**	**R**	**S**
ε	0	1	2	3	4	5
C	1	**1**	2	3	4	5
A	2	2	**1**	**2**	3	4
R	3	3	2	2	**2**	3
S	4	4	3	3	3	**2**

Table 10.19: The filled DP table for CARS → PAIRS. The result is in the bottom-right corner: *D[4][5] = 2*.

The table gives not only the distance but also the edit sequence. Start at the bottom-right corner and walk back to the upper-left corner, looking at each cell to see which neighbor the value came from. This process is called **traceback**. *D[4][5] = 2* comes from the diagonal because S = S. *D[3][4] = 2* again comes from the diagonal because R = R. *D[2][3] = 2* comes from *D[2][2] = 1* on the left by an insertion: I is inserted. *D[2][2] = 1* comes from the diagonal with A = A; *D[1][1] = 1* comes from the diagonal with the substitution C → P. The bold cells show this path. The result: substitute C → P, A matches, insert I, R and S match. The transformation is CARS → PARS → PAIRS.

Common mistake

Confusing the table index with the string index. The table has size *(m + 1) × (n + 1)* because row 0 and column 0 are for the empty prefix. When *D[i][j]* is computed, the characters compared are `a.charAt(i - 1)` and `b.charAt(j - 1)`. Forgetting to initialize the first row

and column also spoils the result: since Java initializes the array with zeros, the borders stay 0 and the distance comes out too small.

10.6.5 Java code and trace

The recurrence translates directly into code:

```
int levenshtein(String a, String b) {
  int m = a.length(), n = b.length();
  int[][] D = new int[m + 1][n + 1];
  for (int i = 0; i <= m; i++) D[i][0] = i;  // deletion
  for (int j = 0; j <= n; j++) D[0][j] = j;  // insertion
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1))
        D[i][j] = D[i - 1][j - 1];            // equal
      else
        D[i][j] = 1 + Math.min(D[i - 1][j - 1], // substitute
                  Math.min(D[i][j - 1],         // insert
                           D[i - 1][j]));       // delete
  return D[m][n];
}
```

The first two loops fill the borders of the table. The two nested loops fill the remaining cells row by row. If the characters are equal, the upper-left value is copied; otherwise 1 is added to the smallest of the three neighbors. On the examples in the slides the code gives these values: kitten → sitting = 3, ercan → sercan = 1 (the initial s is inserted), apple → banana = 5. Let us trace the code with the call “kar” → “ark”. Write *a* = “kar” along the rows and *b* = “ark” along the columns.

	ε	a	r	k
ε	0	1	2	3
k	1			
a	2			
r	3			

Figure 10.26: `levenshtein("kar", "ark")` : the borders are filled. Column 0 shows *i* deletions and row 0 shows *j* insertions.

Cell	Characters	Computation	Value
D[1][1]	k ≠ a	1 + min(substitute 0, insert 1, delete 1)	1
D[1][2]	k ≠ r	1 + min(substitute 1, insert 1, delete 2)	2

Cell	Characters	Computation	Value
D[1][3]	k = k	upper left	2
D[2][1]	a = a	upper left	1
D[2][2]	a ≠ r	1 + min(substitute 1, insert 1, delete 2)	2
D[2][3]	a ≠ k	1 + min(substitute 2, insert 2, delete 2)	3
D[3][1]	r ≠ a	1 + min(substitute 2, insert 3, delete 1)	2
D[3][2]	r = r	upper left	1
D[3][3]	r ≠ k	1 + min(substitute 2, insert 1, delete 3)	2

Table 10.20: Trace of `levenshtein("kar", "ark")`. "substitute", "insert", and "delete" are the values of the upper-left, left, and upper neighbors, respectively.



Figure 10.27: `a[0] = b[2] = 'k'`: no cost; *D[1][3]* takes the 2 from the upper left.

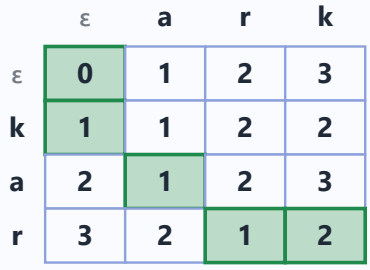


Figure 10.28: The completed table, *D[3][3] = 2*. The green path shows the two edits: the initial k is deleted (ar), and a k is inserted at the end (ark).

The result is 2. Traceback starts at *D[3][3]*. The value came from *D[3][2] = 1* on the left by an insertion: a k is inserted at the end. *D[3][2]* comes from *D[2][1]* with r = r, and that from *D[1][0] = 1* with a = a. *D[1][0]* is the deletion of k. With substitutions alone, three edits would be needed (k → a, a → r, r → k); because the last two letters of "kar" equal the first two letters of "ark", the delete–insert path is shorter.

This example also reminds us that the Levenshtein distance does not count a *swap* of letters as a single operation. In the transformation "saat" → "asat", the first two letters have exchanged places; the distance is 2. The measure that counts a swap of two adjacent letters as one operation is called the

Damerau–Levenshtein distance. Because a significant share of spelling errors are swaps of adjacent letters, spell checkers often use this variant.

10.6.6 Two-row memory

Filling a row requires only the previous row. If only the distance is wanted, storing the whole table is unnecessary. Two arrays suffice: `prev` and `curr`. When each row is finished, the two arrays swap places.

```
int levenshteinTwoRows(String a, String b) {
  if (a.length() < b.length()) { String t = a; a = b; b = t; }  // b is the shorter
  int n = b.length();
  int[] prev = new int[n + 1], curr = new int[n + 1];
  for (int j = 0; j <= n; j++) prev[j] = j;
  for (int i = 1; i <= a.length(); i++) {
    curr[0] = i;
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1))
        curr[j] = prev[j - 1];
      else
        curr[j] = 1 + Math.min(prev[j - 1], Math.min(curr[j - 1], prev[j]));
    int[] t = prev; prev = curr; curr = t;   // swap the rows
  }
  return prev[n];
}
```

The method first makes the shorter string `b`; the length of the arrays is then *min(m, n) + 1*. `curr[0] = i` is the boundary value. The inner loop applies the same recurrence: the upper left is `prev[j - 1]`, the upper is `prev[j]`, and the left is `curr[j - 1]`. When a row is finished, the references are swapped and the new row becomes `prev`. After the loop the answer is in `prev[n]`. On all the examples in the slides, the method gives the same result as the full table.

The saving is large. Let each of two strings have 10,000 characters. The full table holds $10{,}001^2 \approx 10^8$ integers, about 400 MB. The two-row version uses $2 \cdot 10{,}001$ integers, about 80 KB. The time is still $O(m \cdot n)$. The price is that traceback is no longer possible; if the edit sequence is also wanted, the full table or a more advanced technique (Hirschberg's algorithm) is needed.

10.6.7 Other similarity measures

Levenshtein is not the only similarity measure. Depending on the structure of the problem, a simpler or different measure may fit better. Table 10.21 compares four measures on the same pair of strings, AGGTAB and GXTXAYB.

Measure	What it measures	Time	AGGTAB · GXTXAYB
Hamming	number of differing positions in two strings of equal length	$O(n)$	undefined (6 ≠ 7)
Levenshtein	number of insertions, deletions, and substitutions	$O(m \cdot n)$	4
LCS	length of the longest common subsequence	$O(m \cdot n)$	4 (GTAB)
Longest common substring	length of the longest common contiguous part	$O(m \cdot n)$	1 (e.g., “G”)

Table 10.21: Comparison of four similarity measures.

The **Hamming distance** is defined only for strings of equal length and counts the differing characters at corresponding positions. It is computed in a single pass, in $O(n)$ time. For example, the Hamming distance between “karolin” and “kathrin” is 3 (r/t, o/h, l/r). The Hamming measure is suitable when there are no lost or extra characters: fixed-length binary codes, bit errors in communication.

```
int hamming(String a, String b) {
  if (a.length() != b.length())
    throw new IllegalArgumentException("lengths differ");
  int d = 0;
  for (int i = 0; i < a.length(); i++)
    if (a.charAt(i) != b.charAt(i)) d++;
  return d;
}
```

For two strings of equal length, the Levenshtein distance cannot exceed the Hamming distance. Every differing position that Hamming counts can be fixed by one substitution, which is a transformation Levenshtein allows. Levenshtein, which also uses insertions and deletions, sometimes finds a shorter path. For the pair “karolin” and “kathrin”, both measures are 3.

Definition 10.8

A sequence obtained by deleting some characters of a string is called a **subsequence** of that string; the relative order of the remaining characters is preserved, but they need not be consecutive. The **longest common subsequence** (LCS) of two strings is the longest sequence that is a subsequence of both. The **longest common substring**, on the other hand, is the longest part that occurs *consecutively* in both.

The LCS of AGGTAB and GXTXAYB is GTAB. The letters G, T, A, B occur in this order in both strings: at positions 2, 4, 5, 6 in AGGTAB and at positions 1, 3, 5, 7 in GXTXAYB. It does not matter that other letters lie in between. The longest common substring is only 1 character long: there is no two-letter

part that occurs consecutively in both strings. The LCS is also found with a DP table similar to Levenshtein's:

```
int lcs(String a, String b) {
  int m = a.length(), n = b.length();
  int[][] L = new int[m + 1][n + 1];            // row 0 and column 0 are 0
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1))
        L[i][j] = L[i - 1][j - 1] + 1;          // diagonal + 1
      else
        L[i][j] = Math.max(L[i - 1][j], L[i][j - 1]);
  return L[m][n];
}
```

The recurrence is this: if the characters are equal, *L[i][j] = L[i−1][j−1] + 1*, that is, 1 is added to the diagonal value; if they are not equal, *L[i][j] = max(L[i−1][j], L[i][j−1])*. Where Levenshtein looks for the minimum, LCS looks for the maximum. The method returns 4 for AGGTAB · GXTXAYB, 3 for CARS · PAIRS (ARS), and 4 for kitten · sitting (ittn).

There is a simple link between the LCS and the edit distance. If only insertion and deletion are allowed and substitution is forbidden, the distance is *m + n − 2 · LCS*. Every character that is not in the LCS is either deleted or inserted. For AGGTAB · GXTXAYB this is *6 + 7 − 8 = 5*. Levenshtein, which also allows substitution, combines one deletion and one insertion into a single substitution and lowers the distance to 4.

For the longest common substring the recurrence is simpler still. If the characters are equal, *L[i][j] = L[i−1][j−1] + 1*; if they are not, *L[i][j] = 0*, because the contiguity is broken. The answer is not in the bottom-right corner but at the largest value in the table:

```
String longestCommonSubstring(String a, String b) {
  int m = a.length(), n = b.length(), best = 0, end = 0;
  int[][] L = new int[m + 1][n + 1];
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      if (a.charAt(i - 1) == b.charAt(j - 1)) {
        L[i][j] = L[i - 1][j - 1] + 1;          // the run grows
        if (L[i][j] > best) { best = L[i][j]; end = i; }
      }                                         // no match: stays 0
  return a.substring(end - best, end);
}
```

The method returns “ercan” for “ercan” · “sercan” and “itt” for kitten · sitting. If there are several parts of equal length, it gives the first one found; for the pair AGGTAB · GXTXAYB it returns “A”. Plagiarism detection and finding the common blocks of two files are places where this measure is used.

Distances can also be converted into a comparable **similarity ratio**. A common definition is $1 - d / \max(m, n)$: it gives 1 for identical strings and a value close to 0 for strings with nothing in common. For kitten · sitting this is $1 - 3/7 \approx 0.57$. A spell checker might, for example, show dictionary words with a similarity above 0.7 as suggestions.

Common mistake

Confusing a subsequence with a substring. The LCS does not require contiguity; the longest common substring does. For the pair AGGTAB · GXTXAYB, one is 4 and the other 1. Applying the Hamming distance to strings of different lengths is another mistake: by definition it cannot be computed, and the `hamming` method above should throw an error in this case.

10.7 The suffix array

If the same pattern will be searched for many times in a text, scanning the text from the beginning on every search is expensive. The index at the back of a book solves this problem: the words are in a sorted list, and the pages on which each occurs are written next to it. The suffix array builds such an index for all the substrings of a text.

10.7.1 Definition

Definition 10.9

The i-th suffix of a string t of n characters is the substring that starts at position i and runs to the end of the string: $t[i..n-1]$. The **suffix array** (SA) of t consists of the starting indices of all the suffixes arranged in lexicographic order: $SA[k]$ is the position at which the k-th suffix in the sorted list begins.

i	suffix	rank	SA	sorted suffix
0	banana	0	5	a
1	anana	1	3	ana
2	nana	2	1	anana
3	ana	3	0	banana
4	na	4	4	na
5	a	5	2	nana

Table 10.22: The suffixes of the string “banana” (left, by starting position) and the sorted list (right). The SA column on the right is the suffix array.

“banana” has six suffixes. When they are sorted, “a” comes first, because it is a prefix of “ana” and “anana”. Then come “ana”, “anana”, “banana”, “na”, “nana”. The suffix array is *SA = [5, 3, 1, 0, 4, 2]*. The array stores only the starting indices, not the suffixes. A suffix itself is obtained when needed with `t.substring(SA[k])`.

The power of the suffix array rests on a simple observation: every substring of a string is a prefix of some suffix. The substring “nan” is a prefix of the suffix “nana”, which starts at position 2. So finding where a pattern occurs in the text means finding the suffixes that begin with that pattern. Because the suffixes are sorted, these suffixes sit next to each other in the list.

An older structure that does the same job is the **suffix trie**: a tree that inserts all the suffixes into a trie. A suffix trie searches very fast, but its number of nodes is proportional to the number of distinct substrings and is $O(n^2)$ in the worst case. Even a six-letter string such as “banana” has 15 distinct substrings, and hence 16 nodes counting the root. The suffix array holds only n integers: $O(n)$ memory. String search, sorting, genome sequence analysis, the BWT, and data compression are the main uses of the suffix array.

10.7.2 Construction methods

The easiest way to build the suffix array is to sort all the suffixes with a comparison sort:

```
int[] suffixArray(String t) {        // brute force: O(n² log n)
  int n = t.length();
  Integer[] ix = new Integer[n];
  for (int i = 0; i < n; i++) ix[i] = i;
  Arrays.sort(ix, (x, y) -> t.substring(x).compareTo(t.substring(y)));
  int[] sa = new int[n];
  for (int i = 0; i < n; i++) sa[i] = ix[i];
  return sa;
}
```

This brute-force method makes $O(n \log n)$ comparisons. Each comparison reads two suffixes character by character and takes $O(n)$ in the worst case; in addition, `substring` copies a new string on every call. The total time is $O(n^2 \log n)$. This is adequate for short texts but unusable for a genome sequence of a few million characters. Faster methods avoid comparing suffixes character by character.

Method	Idea	Time
Brute force	all suffixes are created and sorted with `Arrays.sort`; each comparison is $O(n)$	$O(n^2 \log n)$
Manber–Myers (1990)	prefix doubling: the ranks for $2k$ characters are found from the ranks for k characters by radix-sorting the rank pairs	$O(n \log n)$
Larsson–Sadakane (1999)	the same doubling; only groups that have not yet been separated are re-sorted	$O(n \log n)$

Method	Idea	Time
DC3 (2003), SA-IS (2009)	some of the suffixes are sorted recursively, and the rest are derived from them and merged	*O(n)*

Table 10.23: Suffix array construction methods.

Prefix-doubling methods run in *O(n log n)* time, not linear. Linear-time construction became possible only with later algorithms such as DC3 and SA-IS. SA-IS is still one of the fastest methods in practice today and is used in many bioinformatics tools.

10.7.3 Prefix doubling

The idea of Manber and Myers is this. First the suffixes are sorted by their first character only, and each suffix is assigned a **rank**: suffixes with the same first character get the same rank. Then they are sorted by the first two characters, the first four, the first eight. In each round the prefix length considered doubles. An observation that speeds up a round is this: the first *2k* characters of the *i*-th suffix consist of two parts. The first *k* characters are the prefix of the suffix starting at *i*, and the next *k* characters are the prefix of the suffix starting at *i + k*. The *k*-character ranks of both are known from the previous round. So the sort on *2k* characters is done without comparing letters, by sorting the number pairs *(rank[i], rank[i + k])*.

Example 10.7

Three rounds on “banana”:

1. **1 character.** The suffixes are sorted by first character: a → 0, b → 1, n → 2. The ranks: *rank[0..5] = 1, 0, 2, 0, 2, 0*. The suffixes starting at 1, 3, 5 (all starting with a) have not yet been separated.
2. **2 characters.** The key is *(rank[i], rank[i + 1])*; if *i + 1 ≥ n*, the second item is −1. 5: (0, −1); 1: (0, 2); 3: (0, 2); 0: (1, 0); 2: (2, 0); 4: (2, 0). After sorting, the new ranks are 5 → 0, 1 and 3 → 1, 0 → 2, 2 and 4 → 3. The two suffixes starting with “an” and the two starting with “na” are still equal.
3. **4 characters.** The key is *(rank[i], rank[i + 2])*. 5: (0, −1); 3: (1, 0); 1: (1, 1); 0: (2, 3); 4: (3, −1); 2: (3, 3). All the keys are distinct; the new ranks spread from 0 to 5. The sort is complete: *SA = [5, 3, 1, 0, 4, 2]*.

i	first 4 characters	key	new rank
5	a	(0, −1)	0
3	ana	(1, 0)	1
1	anan	(1, 1)	2
0	bana	(2, 3)	3
4	na	(3, −1)	4

i	first 4 characters	key	new rank
2	nana	(3, 3)	5

Table 10.24: The last round of prefix doubling (first 4 characters). The key is *(rank[i], rank[i + 2])*; no letters are compared, only number pairs are sorted.

One row of the third round sums up the method. “ana”, starting at 3, and “anana”, starting at 1, have the same first two characters; both have 2-character rank 1. The difference lies in the next two characters: the rank of “a”, starting at 3 + 2 = 5, is 0, and the rank of “an…”, starting at 1 + 2 = 3, is 1. Since (1, 0) < (1, 1), “ana” comes before “anana”. No letter has been read again.

The number of rounds is at most $\lceil \log_2 n \rceil$, because when $2^k \geq n$, the prefix considered covers every suffix from beginning to end and all the ranks must be distinct. In each round, the n number pairs can be sorted in $O(n)$ with a two-pass radix sort. The total time is $O(n \log n)$. To keep the idea clear, the version below sorts the number pairs with `Arrays.sort`; each round therefore takes $O(n \log n)$ and the total $O(n \log^2 n)$:

```
int[] prefixDoubling(String t) {          // sorted with Arrays.sort
  int n = t.length();
  Integer[] sa = new Integer[n];
  int[] rank = new int[n], tmp = new int[n];
  for (int i = 0; i < n; i++) { sa[i] = i; rank[i] = t.charAt(i); }
  for (int k = 1; ; k *= 2) {
    final int kk = k;
    final int[] r = rank;
    Comparator<Integer> cmp = (x, y) -> {
      if (r[x] != r[y]) return Integer.compare(r[x], r[y]);
      int rx = x + kk < n ? r[x + kk] : -1, ry = y + kk < n ? r[y + kk] : -1;
      return Integer.compare(rx, ry);
    };
    Arrays.sort(sa, cmp);            // key: (rank[i], rank[i + k])
    tmp[sa[0]] = 0;
    for (int i = 1; i < n; i++)
      tmp[sa[i]] = tmp[sa[i - 1]] + (cmp.compare(sa[i - 1], sa[i]) < 0 ? 1 : 0);
    System.arraycopy(tmp, 0, rank, 0, n);
    if (rank[sa[n - 1]] == n - 1) break;   // all ranks distinct
  }
  int[] s = new int[n];
  for (int i = 0; i < n; i++) s[i] = sa[i];
  return s;
}
```

The `rank` array initially holds the character codes. In each round the comparator `cmp` looks first at `rank[x]` and, if equal, at `rank[x + k]`. After sorting, the new ranks are assigned: a suffix's rank increases by one if it differs from the one before it, and stays the same otherwise. When the rank of

the last suffix is $n - 1$, all ranks are distinct and the loop ends. The method returns *[5, 3, 1, 0, 4, 2]* for “banana” and *[10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]* for “mississippi”; both are the same as with brute force.

10.7.4 Searching with a suffix array

Because the suffixes are sorted, the suffixes that start with a pattern *P* form a contiguous range in the SA. The two ends of the range are found by binary search. The first end is the first suffix that is not smaller than *P*. The second end is the first suffix after that point that does not start with *P*.

```
int[] search(String t, int[] sa, String p) {
  int lo = 0, hi = sa.length;
  while (lo < hi) {   // first >= p
    int mid = (lo + hi) / 2;
    if (t.substring(sa[mid])
          .compareTo(p) < 0) lo = mid + 1;
    else hi = mid;
  }
  int first = lo;
  hi = sa.length;
  while (lo < hi) {   // not starting with p
    int mid = (lo + hi) / 2;
    if (t.startsWith(p, sa[mid]))
      lo = mid + 1;
    else hi = mid;
  }
  return new int[] { first, lo };
}
```

The first loop is a classic “lower bound” binary search. If the middle suffix is smaller than *P*, the end being sought lies to the right: `lo = mid + 1`. Otherwise the end is at `mid` or to its left: `hi = mid`. When the loop finishes, `lo` is the rank of the first suffix not smaller than *P* and is written into `first`. The second loop leaves `lo` where it is and pulls `hi` back to the end of the array. This time the question is “does this suffix start with *P*?”. If it does, the end of the range lies to the right. When the loop finishes, `lo` points just past the range. The method returns the half-open range `[first, lo)`.



Figure 10.29: The start of `search("banana", sa, "an")`: `lo = 0`, `hi = 6` (end of the array). The suffixes

starting with “an” are shown with a blue prefix.

Loop	`mid`	Suffix	Question and answer	New range
1	3	banana	“banana” < “an”? no → `hi = 3`	[0, 3)
1	1	ana	“ana” < “an”? no → `hi = 1`	[0, 1)
1	0	a	“a” < “an”? yes → `lo = 1`	[1, 1)
—	—	—	`first = 1`, `hi = 6`	[1, 6)
2	3	banana	starts with “an”? no → `hi = 3`	[1, 3)
2	2	anana	starts with “an”? yes → `lo = 3`	[3, 3)

Table 10.25: Trace of `search("banana", sa, "an")`, *SA = [5, 3, 1, 0, 4, 2]*.



Figure 10.30: The first loop has finished: the first suffix not smaller than “an” is at rank 1, `first = 1`. For the second search, `hi` is pulled back to the end of the array.



Figure 10.31: The result is `[1, 3)`: ranks 1 and 2 are selected; their SA values are 3 and 1.

The result is `[1, 3)`: ranks 1 and 2. The SA values at these ranks are 3 and 1. “an” occurs in “banana” at positions 1 and 3. A total of 5 comparisons were made. Each binary search makes at most $\lfloor \log_2 6 \rfloor + 1 = 3$ comparisons. Each comparison reads at most $m = |P|$ characters. The total time is $O(m \log n)$. Even in a text of a billion characters, each search finishes with about 60 suffix comparisons.

If an **LCP array** (longest common prefix) is added next to the suffix array, more questions can be answered. The LCP array holds the length of the longest common prefix of two neighboring suffixes in the sorted list. For “banana”, the common prefix of “a” and “ana” is 1 character, and that of “ana” and “anana” is 3. The longest substring that occurs twice in the text is the largest value of the LCP array. In “banana” this is “ana”, and it is found by one scan of the LCP array, in *O(n)*.

Common mistake

Confusing the values of the suffix array with rank numbers. The range `[1, 3)` returned by `search` consists of *ranks* in the sorted list; the positions of the pattern in the text are the `sa[k]` values at those ranks (3 and 1). Another mistake is to assume the positions of the range are in order: the SA holds the sorted suffixes, while the positions come in mixed order.

10.8 The Burrows–Wheeler transform

Compression algorithms feed on repetition: “AAAAA” can be shortened to the information that the letter “A” is written five times. But in natural text, equal letters are usually not adjacent. The Burrows–Wheeler transform rearranges the letters so that equal letters are gathered together. Moreover, it does so in a reversible way.

10.8.1 Definition and steps

Definition 10.10

The **Burrows–Wheeler transform** (BWT) of a string *s* is computed in three steps. First, all the cyclic rotations of *s* are created. Then these rotations are sorted lexicographically. Finally, the last characters of the sorted rotations are read from top to bottom. The resulting string is called the **L column**. For the inverse transform, either the row number of the original string in the sorted list is stored, or a *$* marker, smaller than every character of the alphabet and occurring only once, is appended to *s*.

A rotation is obtained by moving the first character of the string to the end; when this is done *n* times, the string returns to the start. A string of *n* characters has *n* rotations. The BWT does not change the characters, only their positions: the L column is another arrangement of the letters of *s*. Michael Burrows and David Wheeler published the transform in a technical report in 1994.

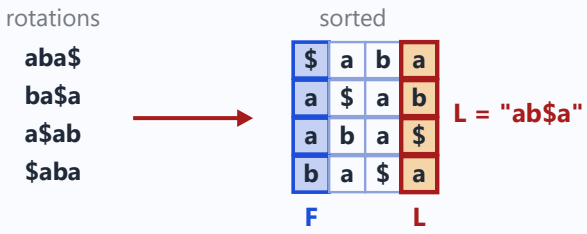


Figure 10.32: The rotations of the string “aba$” (left) are sorted (right). The first column is F and the last column is L; the BWT output is *L = “ab$a”*.

In the small example in Figure 10.32, the four rotations of “aba$” are aba$, baa, aab, and $aba. Because *$* is the smallest character, $aba comes first in the sorted list, followed by aab, aba, and ba$a. The last characters are a, b, $, a: *L = “ab$a”*. The first column of the sorted list is the **F column**: $, a, a, b.

10.8.2 Examples: BANANA and abraca

How the BWT groups letters is clearly visible in the “BANANA” example. In this example *$* is not used; the number of the original row is stored instead.

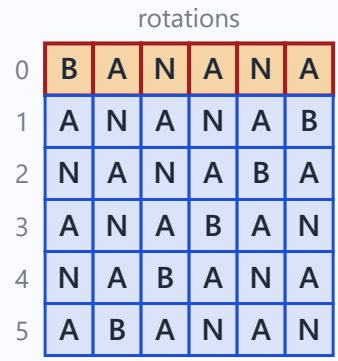


Figure 10.33: Step 1: the six rotations of BANANA. Each row is obtained by moving the first character of the previous one to the end.

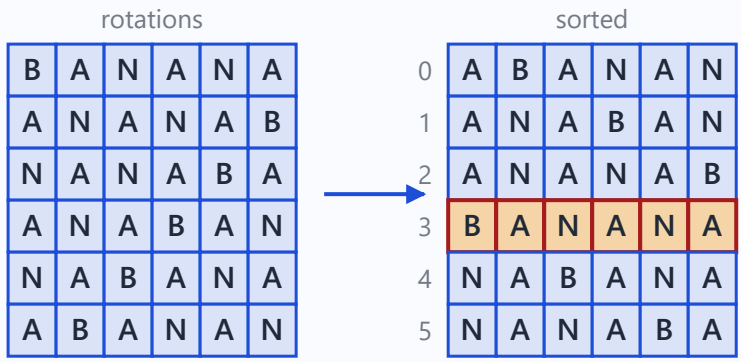


Figure 10.34: Step 2: the rotations are sorted alphabetically. The original string BANANA falls in row 3, counting from 0.

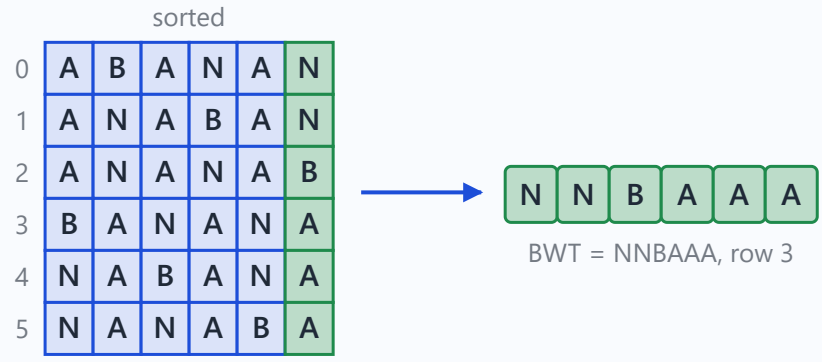


Figure 10.35: Step 3: the last characters (the L column) are read. The output is (NNBAAA, 3); equal letters have been grouped.

The rotations are BANANA, ANANAB, NANABA, ANABAN, NABANA, and ABANAN. The sorted list is ABANAN, ANABAN, ANANAB, BANANA, NABANA, NANABA. The last characters are N, N, B, A, A, A. The output is (NNBAAA, 3). In the original string no letter was equal to its neighbor; after the transform, two N's and three A's are adjacent.

The reason for the grouping is the structure of sorting. In the sorted list, rotations that begin with the same characters sit next to each other. For example, the two rotations that begin with “ANA…” are neighbors. The L column holds the character that comes just *before* the first character of each rotation. In natural text, the same character usually comes before the same context: in English the character before the context “he” is often t (the), and in Turkish the character before the context “ğı” is often a (ağı, dağı). When similar contexts come together in the sorted order, the equal characters that precede them also fall next to each other in L.

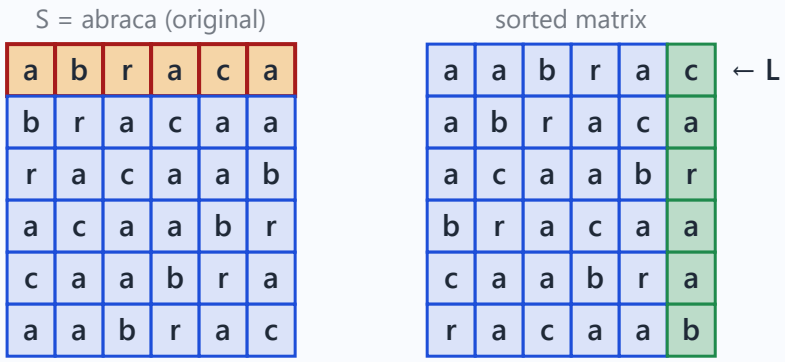


Figure 10.36: "abraca": the initial matrix (left, rotations) and the sorted matrix (right). The last column is *L = "caraab"*.

For "abraca" the sorted rotations are aabrac, abraca, acaabr, bracaa, caabra, racaab. The last column is *L = "caraab"*. The original "abraca" is in row 1 of the sorted matrix, counting from 0; this index is stored for the inverse transform. The first column is F = aaabcr and is the sorted version of the letters of L. This is always true: since every rotation contains the same letters, F and L carry the same multiset of letters, and F is sorted. Once L is known, sorting L is enough to find F.

The direct method explicitly produces the n rotations and sorts them:

```
String bwt(String s) {              // direct method; s ends with $
  int n = s.length();
  String[] rot = new String[n];
  for (int i = 0; i < n; i++)
    rot[i] = s.substring(i) + s.substring(0, i);
  Arrays.sort(rot);                 // O(n² log n) time, O(n²) space
  StringBuilder L = new StringBuilder();
  for (String r : rot) L.append(r.charAt(n - 1));
  return L.toString();
}
```

This method uses $O(n^2)$ memory, because each of the n rotations has n characters. The sort makes $O(n \log n)$ comparisons and each comparison can take $O(n)$: $O(n^2 \log n)$ time in total. A one-megabyte block would need a matrix of 10^{12} characters; this is clearly impractical. In practice the BWT is computed from the suffix array. If the text ends with $, the order of the rotations is the same as the order of the suffixes. Two rotations separate no later than when they reach $; what follows $ takes no part in the comparison. The k-th sorted rotation starts at $SA[k]$; its last character is the character just before that. So $L[k] = T[SA[k] - 1]$, and if $SA[k] = 0$ then $L[k] = \$$. The formula is for the BWT with $; the BWT with $ and the cyclic BWT give different results: the cyclic BWT of BANANA is NNBAAA, while that of BANANA$ is ANNB$AA. If the suffix array is built in linear time, the BWT is also obtained in $O(n)$.

10.8.3 Inversion: the LF mapping

The truly surprising thing about the BWT is that the original text can be recovered from the L column alone (and from *$* or the row index). As an example, take *T = "banana$"*. The *$* is the smallest and a unique character.

i	F	...	L
0	$	banan	a
1	a	$bana	n
2	a	na$ba	n
3	a	nana$	b
4	b	anana	$
5	n	a$ban	a
6	n	ana$b	a

Table 10.26: The sorted rotation matrix for "banana$". F is the first column and L the last; the characters in between are unknown during the inverse transform.

The BWT output is *L = "annb$aa"*. F is the sorted version of L: *F = "$aaabnn"*. Since each row is a rotation, the last character of a row, *L[i]*, comes in the text just before the first character *F[i]*. The inverse transform uses this adjacency as a chain. The only missing piece of information is which row of F a character in L corresponds to. When the same letter occurs several times, this is not obvious. The theorem below removes this ambiguity.

Theorem 10.4 (LF mapping)

If a character c occurs for the k-th time in L (counting from 0), then the same character in the text is also the k-th occurrence of c in F. Accordingly, the row of *L[i]* in F is *LF(i) = C[c] + occ[i]*. Here *c = L[i]*, *C[c]* is the number of characters in the text smaller than c (that is, the row where c starts in F), and *occ[i]* is how many c's precede *L[i]* in L, that is, which c it is, counting from 0.

Proof. Two rows that end with c in L have the forms $x \cdot c$ and $y \cdot c$; let $x \cdot c$ come first in the sorted list, that is, $x \cdot c < y \cdot c$. Rotating these two rows once to the right gives the rows $c \cdot x$ and $c \cdot y$; these are also rows of the matrix and begin with c in F. When two strings begin with the same character, the rest decides their order. $x \cdot c$ and $y \cdot c$ have equal length, and the first position where they differ lies within

x and y. Hence $x < y$, and $c \cdot x < c \cdot y$. So the order of the rows ending with c in L is the same as the order of their rotated versions in F. The k-th c in L is the k-th c in F. Since F is sorted, the first row of c in F is $C[c]$, and the k-th c is in row $C[c] + k$. □

The inverse transform is now simple. The row beginning with $ is row 0; this row is a rotation of the text itself, and its last character $L[0]$ is the last character of the text before the $. $L[0]$ is read and the walk goes to row $LF(0)$. The L character of that row is the previous character in the text. As the walk continues, the text is read from end to beginning. For "banana$" the rows are visited in the order $0 \to 1 \to 5 \to 2 \to 6 \to 3 \to 4$. The L characters read are a, n, a, n, a, b. Reversed, they give "banana".

If the occurrence counts and the C table are computed beforehand in one pass, each step takes constant time and the inverse transform takes $O(n)$. If the text has no $, as in the BANANA example, the original row index (3) is stored and the walk is started from that row instead of 0.

10.8.4 Code trace: inverseBwt

```
String inverseBwt(String L) {   // with $
  int n = L.length();
  int[] occ = new int[n];
  int[] count = new int[256];
  int[] C = new int[256];
  for (int i = 0; i < n; i++)
    occ[i] = count[L.charAt(i)]++;
  for (int c = 1; c < 256; c++)
    C[c] = C[c - 1] + count[c - 1];
  StringBuilder s = new StringBuilder();
  int i = 0;                    // the $ row
  for (int k = 0; k < n - 1; k++) {
    char c = L.charAt(i);
    s.insert(0, c);
    i = C[c] + occ[i];          /* LF(i) */
  }
  return s.toString();
}
```

The method has four parts. The first loop computes `occ[i]` for each i: `count[c]` holds how many c's have been seen so far in L and then increases by one. The second loop builds the `C` table with a prefix sum: `C[c]` is the number of all characters smaller than c. The third part is the walk. It starts at `i = 0`, the $ row. At each step `L.charAt(i)` is read and added to the *front* of the text being built (`s.insert(0, c)`); since the text is read from end to beginning, no reversal is needed. Then `i` is updated to $LF(i)$. The loop runs $n - 1$ rounds; the $ is not added to the text.

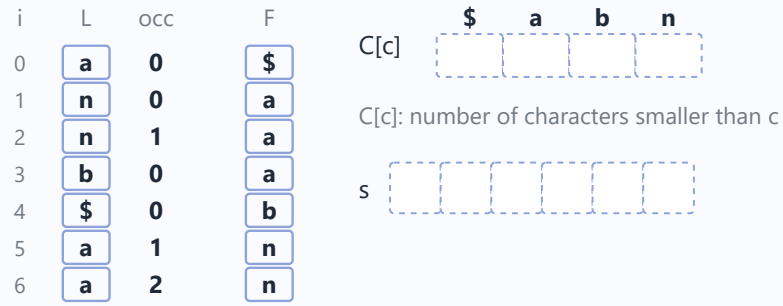


Figure 10.37: `inverseBwt("annb$aa")` : L, the occurrence counts, and F. `occ[6] = 2`, because L[6] is the third a.

The results of the preparation steps are these. *occ = [0, 0, 1, 0, 0, 1, 2]*: the a's in L (positions 0, 5, 6) get the order 0, 1, 2 and the n's (1, 2) get 0, 1; b and *$* are 0 because they are unique. The C table: *C[$] = 0*, *C[a] = 1*, *C[b] = 4*, *C[n] = 5*. These values are the rows where each letter starts in the F column.

k	i	L[i]	s	*LF(i) = C[c] + occ[i]*
0	0	a	a	1 + 0 = 1
1	1	n	na	5 + 0 = 5
2	5	a	ana	1 + 1 = 2
3	2	n	nana	5 + 1 = 6
4	6	a	anana	1 + 2 = 3
5	3	b	banana	4 + 0 = 4

Table 10.27: The walk of `inverseBwt("annb$aa")`.

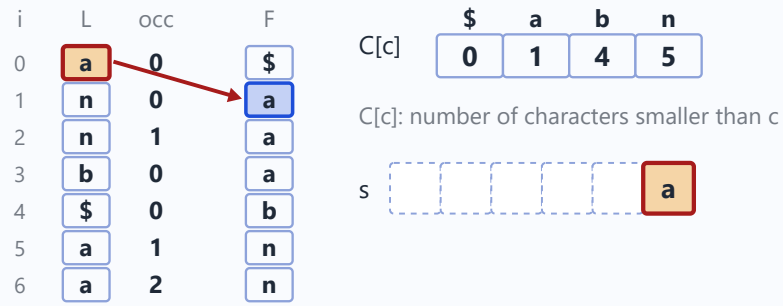


Figure 10.38: First step: `L[0] = 'a'` is added to the text; *LF(0) = C[a] + occ[0] = 1*. The arrow shows that the first a in L corresponds to the first a in F.

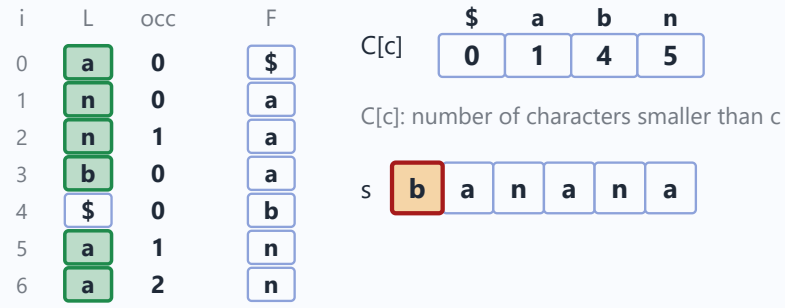


Figure 10.39: The rows 0 → 1 → 5 → 2 → 6 → 3 → 4 were followed. Because the next row would end with *$*, the loop stops after *n – 1 = 6* rounds. Result: banana.

After the sixth round, `i = 4`. The L character of row 4 is *$*; this row is "banana$" itself, and the walk has come back to the start. The loop ends at exactly this point. The time is *O(n + R)* and the extra memory is *O(n + R)* for the `occ`, `count`, and `C` arrays. There is an implementation detail: because `s.insert(0, c)` shifts the whole string on every call, it actually takes *O(n)*. For large inputs, the characters are written into an array from back to front, or appended at the end and reversed once at the very end.

Common mistake

Storing the BWT output as just the L column. If neither the *$* marker nor the original row index is stored, the transform cannot be undone: all the rotations of "BANANA" give the same L column (NNBAAA). Another mistake is not choosing *$* as the smallest character of the alphabet. In Java, '$' (36) is smaller than letters and digits, but if the text contains a space (32), this condition breaks and the LF mapping goes to the wrong rows.

10.8.5 Role in compression

The BWT is not compression by itself; its output has the same length as its input. It prepares the data for compression. The widely used `bzip2` program uses the BWT at the head of a four-stage chain:

1. **BWT.** The text is split into blocks and each block is transformed. Equal letters are gathered together.
2. **Move-to-front** (MTF). Each character is encoded by its position in a list and is then moved to the front of the list. Recently seen letters are encoded by small numbers; repeated equal letters become 0's.
3. **Run-length encoding** (RLE). Consecutive equal values are written as (value, repeat count) pairs. The long runs of 0's produced by MTF shrink here.
4. **Huffman coding.** Frequent values are given short bit strings and rare values long ones.

Each link in the chain takes advantage of the structure the previous one prepared. Without the BWT, the output of MTF would not contain so many 0's; without MTF, RLE would find no long runs. The BWT is also important in bioinformatics. A structure called the FM-index combines the BWT with the LF mapping to search for short reads in a genome of billions of letters directly on compressed data.

10.9 Common mistakes

The warnings scattered through the notes and the list of mistakes in the slides are collected here.

Common mistake

Taking code order for the alphabet. `compareTo` compares character codes: “Zebra” < “apple” and “şeker” > “zeytin”. For upper/lower-case differences and Turkish letters, a `Collator` or a normalization beforehand is needed.

Common mistake

Sorting numbers as strings. “10” < “9”, because '1' < '9'. For numeric order, convert the strings to integers or pad them on the left with zeros to the same length.

Common mistake

An unstable subsort in LSD. Every pass must be stable; otherwise the order established by earlier passes is destroyed. In counting sort, placement runs from back to front. A version that places from front to back puts the ten strings from the slides in the order `add, ace, bed, bee, bad, …`.

Common mistake

An unescaped dot and a greedy `.*`**.** `a.b` also matches “a+b”; `<.*>` swallows the whole line. The right choices are `\.` and the lazy `.*?`. Nested quantifiers such as `(a+)+$` can lead to an exponential number of attempts in backtracking engines.

Common mistake

Confusing `find` **with** `matches`**.** `find` looks for a part; `matches` requires the whole string. For validation, either use `matches` or wrap the expression in `^` and `$`; otherwise an input such as “abc123” may be accepted as a “number”.

Common mistake

Taking an intermediate state of an FSM for the result. A DFA's decision depends only on the state when the input ends. Passing through an accepting state along the way does not make the string accepted. In a pattern automaton, returning to 0 on a mismatch misses matches; the correct state is the longest matching prefix.

Common mistake

An index shift in the DP table. The table is $(m + 1) \times (n + 1)$; for $D[i][j]$, `charAt(i - 1)` and `charAt(j - 1)` are compared. Not initializing the first row and column spoils the result.

Common mistake

Confusing a subsequence with a substring. The LCS does not require contiguity; the longest common substring does. The Hamming distance is defined only for strings of equal length.

Common mistake

Not storing the row index or *$* in the BWT. All the rotations of a string give the same L column; without extra information the transform cannot be undone. In the suffix array, ranks and text positions (SA values) must not be confused with each other.

10.10 Summary

- **Lexicographic order:** the first differing character decides; if one string is a prefix of the other, the shorter comes first. Computers use character codes: capital letters come before lowercase ones, and Turkish letters come after z. One comparison takes $O(W)$.
- **Comparison sorting:** $O(N^2 \cdot W)$ with the simple method, $O(W \cdot N \log N)$ with merge sort. Every comparison-based method makes $\Omega(N \log N)$ comparisons.
- **LSD radix sort:** right to left, a stable counting sort on that character in each pass; $O(W \cdot (N + R))$ time, $O(N + R)$ memory. Its correctness rests on the invariant "sorted by the last k characters after k passes" and on stability.
- **MSD radix sort:** left to right, each bucket sorted recursively on its own; only the distinguishing prefixes are read. Switch to insertion sort on small buckets.
- **Regular expressions:** `. * + ? [ ] ^ $ ( ) { } |` and `\d \w \s \b`; quantifiers are greedy by default and become lazy with a `?`. Lookaround expressions test a position and consume no characters. In Java, `Pattern` / `Matcher`; `find` searches for a part, `matches` for the whole string.

- **Finite state machine:** *(Q, Σ, δ, q_0, F)*. A DFA has one transition per (state, symbol), and scanning takes *O(n)*. DFAs, NFAs, and regular expressions recognize the same languages; the subset construction produces 2^n states in the worst case.
- **Pattern-matching automaton:** the state is the length of the longest prefix of the pattern that matches the end of the text. Construction *O(m · |Σ|)*, search *O(n)*; it is the automaton form of KMP.
- **Levenshtein:** upper left if equal, otherwise *1 + min(delete, insert, substitute)*; *O(m · n)* time. The edit sequence is read back from the table; two rows suffice for the distance alone.
- **Other measures:** Hamming for equal lengths, *O(n)*. LCS and longest common substring *O(m · n)*. With insertion and deletion only, the distance is *m + n − 2 · LCS*.
- **Suffix array:** the starting indices of the sorted suffixes; *O(n)* memory. Built in $O(n^2 \log n)$ by brute force, *O(n log n)* by prefix doubling, and *O(n)* with SA-IS. Search takes *O(m log n)* with two binary searches.
- **BWT:** the last column of the sorted rotations and the row index (or *$*). It groups equal letters; it is inverted in *O(n)* with the LF mapping. It is the first step of `bzip2`: BWT → move-to-front → RLE → Huffman.

10.11 Key terms

string alphabet prefix suffix substring lexicographic order stable sort counting sort radix sort

LSD / MSD radix sort regular expression quantifier lookahead / lookbehind

finite state machine (FSM) deterministic finite automaton (DFA) subset construction trap state

edit distance Levenshtein distance dynamic programming traceback

longest common subsequence (LCS) longest common substring suffix array prefix doubling

Burrows–Wheeler transform (BWT) move-to-front (MTF)

10.12 Test Yourself

1. What is the result when the array `{"kedi", "Kedi", "kediler", "ked", "kafa", "çay"}` is sorted with `Arrays.sort`? Where does it differ from Turkish dictionary order?

   Answer

   The result is `Kedi, kafa, ked, kedi, kediler, çay`. 'K' (75) comes before all the lowercase letters. “kafa” goes ahead at the second character with a < e. “ked” is a prefix of “kedi”, and “kedi” is a prefix of “kediler”. Because 'ç' (231) is even larger than z, “çay” falls to the very end. In a Turkish dictionary, “çay” comes before the words starting with k, and “Kedi” and “kedi” come next to each other.

2. If LSD radix sort is applied to the strings `dog, cat, dot, cab, cog`, what does the array look like after each pass?

   Answer

   By the 3rd character: `cab, dog, cog, cat, dot` (b < g < t; dog and cog keep their input order). By the 2nd character: `cab, cat, dog, cog, dot` (in bucket a, cab, cat; in bucket o, dog, cog, dot in the earlier order). By the 1st character: `cab, cat, cog, dog, dot`.

3. Which of the strings "A12", "B1234", "c123", "Z999x", and "Q7y" does the expression `^[A-Z]\d{2,3}[a-z]?$` match, in the sense of `matches`?

   Answer

   Only "A12" and "Z999x". The expression requires one capital letter, two or three digits, and an optional single lowercase letter. "B1234" has four digits. "c123" starts with a lowercase letter. "Q7y" has a single digit.

4. At which positions does the pattern `(?<=a)b` match in the text "abcabb"? In the text "abbb", which parts does a `find` loop find with `b+` and with `b+?`?

   Answer

   The lookbehind selects the b's that have an a before them: positions 1 and 4. The b at 5 has a b before it. Because the lookaround consumes no characters, the text of both matches is just "b". In the text "abbb", the greedy `b+` finds "bbb" as a single match. The lazy `b+?` takes a single b each time and gives three separate "b" matches.

5. What is the transition table of the pattern-matching automaton for the pattern "aab" (alphabet {a, b})? How do the states change while the text "aaab" is read, and where is the pattern found?

   Answer

   State 0: a → 1, b → 0. State 1: a → 2, b → 0. State 2: a → 2, b → 3. State 3: a → 1, b → 0. If an a arrives in state 2, the end of what has been read is "aaa" and the longest matching prefix is still "aa"; so the automaton stays in 2. For "aaab" the states are 1, 2, 2, 3. State 3 is reached at character number 3; the pattern starts at position *3 − 3 + 1 = 1*.

6. What is `levenshtein("kitap", "katip")`? What is the last row of the table, and which edit

sequence gives this distance?

Answer

The distance is 2. The last row (p) is *5, 4, 3, 3, 3, 2*. The letters k, t, and p are at the same positions in the two strings; only the 2nd and 4th letters differ. One solution is two substitutions: i → a (katap), a → i (katip).

7. What are the LCS length, the insertion-and-deletion-only distance, and the Levenshtein distance for “ABCBDAB” and “BDCABA”?

Answer

The LCS length is 4 (e.g., BCBA). The insertion-and-deletion-only distance is $7 + 6 - 2 \cdot 4 = 5$. The Levenshtein distance is also 5; for this pair, substitutions give no shorter path.

8. What is the suffix array of the string “abaab”? At which ranks are the suffixes that begin with “ab”, and at which positions does the pattern occur in the text?

Answer

The sorted suffixes are aab, ab, abaab, b, baab; *SA = [2, 3, 0, 4, 1]*. Those beginning with “ab” are at rank 1 (ab) and rank 2 (abaab). Since the SA values are 3 and 0, “ab” occurs in the text at positions 0 and 3.

9. What is the BWT of the string “abab$”, and how is it inverted with the LF mapping?

Answer

The sorted rotations are $abab, ab$ab, abab$, b$aba, bab$a; *L = “bb$aa”*. *F = “$aabb”*, *C[$] = 0*, *C[a] = 1*, *C[b] = 3*; the occurrence counts are *0, 1, 0, 0, 1*. The walk starts at 0: *L[0] = b*, *LF = 3*; *L[3] = a*, *LF = 1*; *L[1] = b*, *LF = 4*; *L[4] = a*, *LF = 2*. The L character of row 2 is *$* and the walk stops. What was read is b, a, b, a; prepending each character gives “abab”.

10. How is a DFA built that accepts binary strings ending in “00”? Which states does the input “10100” pass through?

Answer

Three states suffice: q_0 (the last symbol is not 0, or the input is empty; the start state), q_1 (the last symbol is a single 0), and q_2 (the last two symbols are 00; accepting). The transitions: q_0: $0 \to q_1$, $1 \to q_0$; q_1: $0 \to q_2$, $1 \to q_0$; q_2: $0 \to q_2$, $1 \to q_0$. For “10100” the path is $q_0 \to q_0 \to q_1 \to q_0 \to q_1 \to q_2$ and the string is accepted.

10.13 Further reading

- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Chapter 5: covers string sorting (LSD, MSD, three-way quicksort), substring search, regular expressions, and data compression with Java code and experimental measurements.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (Cambridge University Press, 1997): treats suffix trees, edit distance, and alignment algorithms in depth with examples from computational biology.
- J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (3rd ed., 2006): DFAs, NFAs, the subset construction, and a proof of Kleene's theorem.
- R. Cox, “Regular Expression Matching Can Be Simple And Fast” (2007, swtch.com): an article that compares automaton-based matching via Thompson's construction with backtracking engines and shows the exponential slowdown with measurements.

10.14 References

1. M. Burrows, D. J. Wheeler, “A Block-sorting Lossless Data Compression Algorithm”, DEC SRC Research Report 124, 1994.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (string-matching automata).
3. R. Cox, “Regular Expression Matching Can Be Simple And Fast”, 2007, https://swtch.com/~rsc/regexp/regexp1.html.
4. J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed., O'Reilly, 2006; the documentation of Java `java.util.regex.Pattern`.
5. D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
6. J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed., 2006.
7. V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals”, *Soviet Physics Doklady* 10(8), 1966.
8. U. Manber, G. Myers, “Suffix Arrays: A New Method for On-Line String Searches”, *SIAM Journal on Computing* 22(5), 1993.
9. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (5.1 String Sorts, 5.4 Regular Expressions).
10. R. A. Wagner, M. J. Fischer, “The String-to-String Correction Problem”, *Journal of the ACM* 21(1), 1974.

Tries

When “an” is typed into a search box, suggestions such as “and”, “ant” and “answer” appear below it. Finding these suggestions in milliseconds does not require trying every word in the dictionary one by one. If the words are stored in a tree in which they share their common beginnings, all the words that start with “an” are gathered under a single node. This tree is called a **trie**, also known as a prefix tree. The notes first introduce the structure of the trie and its node class in Java. Then insertion, search, prefix queries, autocomplete and deletion are presented with code and step-by-step traces. Applications such as wildcard search, spell checking and the longest common prefix are built on top of these basic operations. The analysis section works out the running times of the operations and the memory cost of the trie with numeric examples, and compares the trie with the hash table and the binary search tree. The notes end with the other members of the trie family: the binary trie, the ternary search trie, the compressed trie, the suffix trie and Aho–Corasick.

LEARNING OUTCOMES

Structure
Explain the structure of a trie made up of a root, nodes, edges and an end-of-word marker; draw the trie of a set of words and count its nodes and null references.

Operations
Write insertion, search, prefix queries, autocomplete and deletion in Java and trace them step by step. Build derived operations such as prefix counts, wildcard search and the longest common prefix on this foundation.

Analysis
Justify the $O(L)$ running time and the $O(N \cdot M)$ memory of the operations; weigh the ways of storing children; compare the trie with the hash table and the balanced binary search tree.

Variants
Explain which question the binary trie, the ternary search trie, the compressed trie, the suffix trie and the Aho–Corasick algorithm each solve, and trace them on small examples.

11.1 The trie concept

Most programs that work with strings keep asking the same question: is this word in the dictionary? A spell checker asks it for every word. A compiler looks up every identifier in its table of keywords. A router looks up the address of every packet in its routing table. For many of these questions a hash table is a good answer. But some questions concern not the whole word but its beginning: “which words start with an?”, “which is the longest matching prefix of this address?” For questions of this kind we need a structure that keeps the common beginnings of words together.

Two terms used often in these notes should be made clear from the start. A **prefix** of a string is a part taken from the beginning of the string: the prefixes of the string “and” are “”, “a”, “an” and “and”. A **suffix** is a part taken from the end of the string: the suffixes of “and” are “and”, “nd”, “d” and “”. Every piece taken from inside a string as consecutive characters is a **substring**. Every substring is a prefix of some suffix. This observation is the basis of the suffix trie.

11.1.1 Definition

Consider a dictionary: and, ant, dad, do. Of these four words, two start with “a” and two with “d”. “and” and “ant” also share the prefix “an”. A trie shows this sharing explicitly. Each step down from the root reads one character. Words that begin with the same character enter the same branch and part ways only at the first different character.

Definition 11.1

A **trie** (prefix tree) is a data structure that stores a set of strings in a rooted tree. Each edge of the tree carries a character; the characters on the edges leading to the children of a node are all different. The root represents the empty string. Each node represents the concatenation of the characters on the path from the root down to that node, that is, a prefix. The node at which a word of the set ends is additionally marked as an **end of word**. A word is in the set only if there is a path from the root for it and the last node of the path is marked.

The last sentence of the definition is important. The existence of a path alone is not enough. The path for “an” exists in the trie, but “an” is not in the set; the “an” node is unmarked. The path is merely a prefix of words in the set. The name trie comes from the word *retrieval*. It was proposed by E. Fredkin in his 1960 paper “Trie Memory”. In these notes the names trie and prefix tree refer to the same structure.

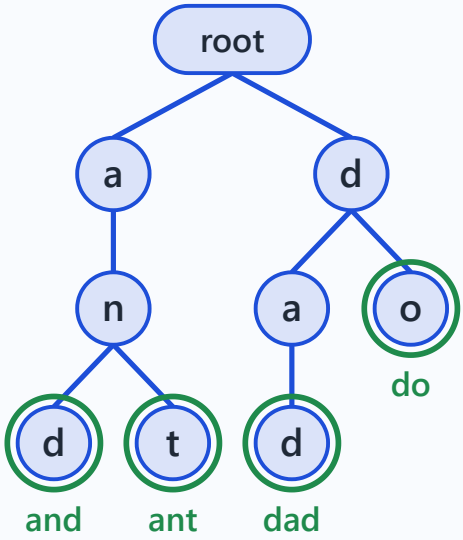


Figure 11.1: The trie of the words and, ant, dad and do. A word ends at each node drawn with a double circle; the word is written below the node.

In Figure 11.1 there are two edges out of the root: “a” and “d”. The “a” node has a single child, “n”. At the “an” node the path forks: “d” leads to “and” and “t” to “ant”. On the right side the “d” node forks too: “a” to “da” and “o” to “do”. “da” has a single child, “d”, and “dad” ends there. The total number of characters in the four words is *3 + 3 + 3 + 2 = 11*, but apart from the root there are only 8 nodes. The 3 characters in between were saved by sharing common prefixes: “and” and “ant” share two characters (“a”, “n”), and “dad” and “do” share one (“d”).

The uses of the trie follow from this structure. In autocomplete, we descend to the node of the typed prefix and suggest the words below it. In spell checking, a word is looked up in the trie; if it is not found, words close to it are also searched for on the trie itself. In IP routing, addresses are stored as bit strings and the route whose prefix matches the packet's address for the longest stretch is found.

11.1.2 Why a trie?

The question is this: all the words that start with “an” are to be found in a dictionary. The dictionary has n words, and the words are at most L long. Different data structures answer this question at different costs.

Array or list. Each word is taken in turn and its first two characters are compared with “an”. One comparison reads at most L characters, so the total work is $O(n \cdot L)$. As the dictionary grows, the time grows linearly. If the array is sorted, the first matching word can be found by binary search in $O(L \cdot \log n)$, but adding a new word becomes expensive because of the shifting in the array.

Hash table. It finds a whole word in $O(L)$ time on average: first the hash value is computed from the L characters of the word, then the relevant bucket is examined. But a hash table cannot answer the prefix question. There is no relation between the hash values of “an” and “and”. A good hash function

is in fact designed to scatter similar keys to distant places in the table. So the question “words starting with an” can be answered only by scanning the whole table.

Trie. From the root we first descend into the “a” branch, then into the “n” branch; two steps suffice. Every word below the node reached starts with “an”, and no word outside this subtree starts with “an”. The cost of finding the prefix node depends not on the number of words in the dictionary but on the length of the prefix.

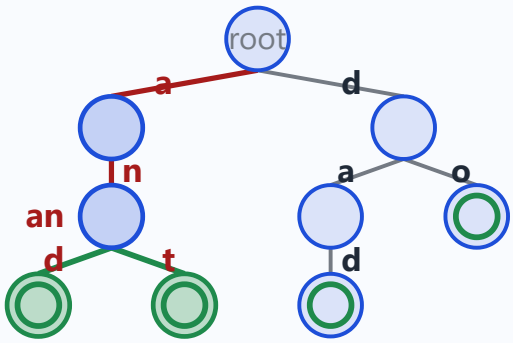


Figure 11.2: The node of the prefix “an”. Every word in the subtree of this node (and, ant) starts with “an”; the remaining branches are never looked at.

This observation is the real contribution of the trie: the running time depends not on n but on L. Whether the dictionary has four words or a million, “dad” is found in at most three steps. A balanced binary search tree needs $log_2\ 10^6 \approx 20$ comparisons, and every comparison reads the characters of the word again.

11.1.3 Basic components

A trie consists of four components. Each of these components corresponds to an element in the code.

- **Root:** the node that represents the empty string; it corresponds to no character. Every operation starts at the root. An empty trie consists of the root alone.
- **Nodes:** the path from the root to a node is a prefix. Each node holds the links to its children. A node can have at most M children, where M is the size of the alphabet. For lowercase English letters $M = 26$.
- **Edges:** an edge joins a node to its child and carries a character. Two edges leaving the same node cannot carry the same character; this rule ensures that each prefix has at most one node in the trie.
- **End-of-word marker:** its name in the code is `isWord`. It states that a word ends at the node. It is shown with a double circle in the figures.

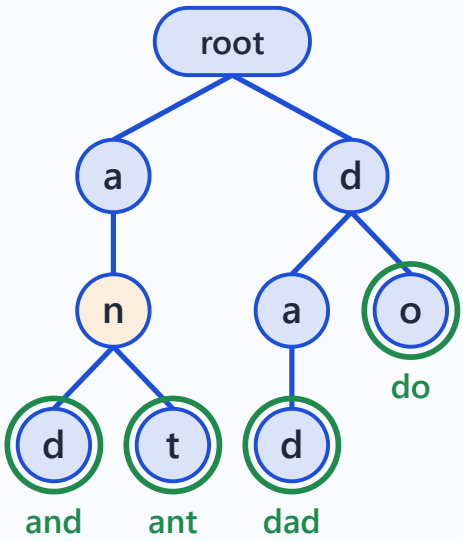


Figure 11.3: The components of a trie. The colored node represents the prefix “an”. “an” is not a word (it is unmarked), but it is a prefix of “and” and “ant”.

The relation among the components can be summarized as follows. There is a one-to-one mapping between nodes and prefixes: each prefix is represented by at most one node. Edges carry characters, while nodes stand for prefixes. The end-of-word marker says which prefixes are also words of the set. Leaf nodes are always marked: an unmarked leaf would be a useless node lying on the path of no word. That is why deletion leaves no unmarked leaf behind.

11.1.4 Example: a trie of seven words

A somewhat bigger example shows why the end-of-word marker is indispensable. The words are ab, aba, abc, ad, ba, bad and bag. The total length of the seven words is $2 + 3 + 3 + 2 + 2 + 3 + 3 = 18$ characters.

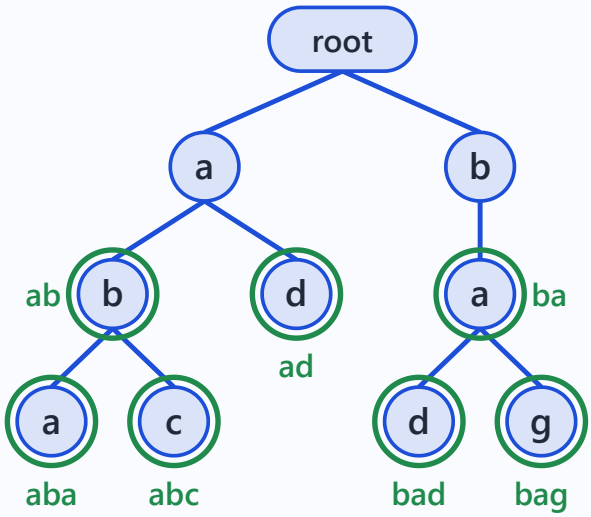


Figure 11.4: The trie of the words ab, aba, abc, ad, ba, bad, bag. The "ab" and "ba" nodes are both an end of word and on the path of other words.

Example 11.1

Question. How many nodes does the trie of seven words have? Which nodes are marked?

Solution. The nodes are all the distinct prefixes of the words. Those starting with "a": a, ab, aba, abc, ad (5 nodes). Those starting with "b": b, ba, bad, bag (4 nodes). That makes 9 nodes, 10 including the root. The 18 characters have fitted into 9 nodes. The marked nodes are the words themselves: ab, aba, abc, ad, ba, bad, bag. Two unmarked nodes remain: a and b. These are only prefixes.

The "ab" node cannot be decided by looking at the figure alone. The node has two children (a and c); that is, "ab" is a prefix of other words. At the same time it is itself a word. The structure cannot tell these two cases apart; only the `isWord` marker makes the distinction. Without the marker, "ab" and "a" would be nodes of the same kind and the answer to the question "is ab in the dictionary?" would be lost.

Common mistake

Confusing the end of a word with being a leaf. The rule "a word ends at a leaf" is wrong. "ab" and "ba" are internal nodes, but they are words. The correct rule goes the other way: every leaf is an end of word, but not every end of word is a leaf.

11.1.5 A trie that carries values

A trie can store not only a set of words but also a map. For this, a value is kept in the node instead of the `isWord` flag. If the value is `null`, the node is not the end of a key; otherwise it is the value of the key that ends at that node. In this way the trie works like a dictionary whose keys are strings.

Key	Value	Key	Value
A	15	tea	3
i	11	ted	4
in	5	ten	12
inn	9	to	7

Table 11.1: The keys and values of a trie that carries values.

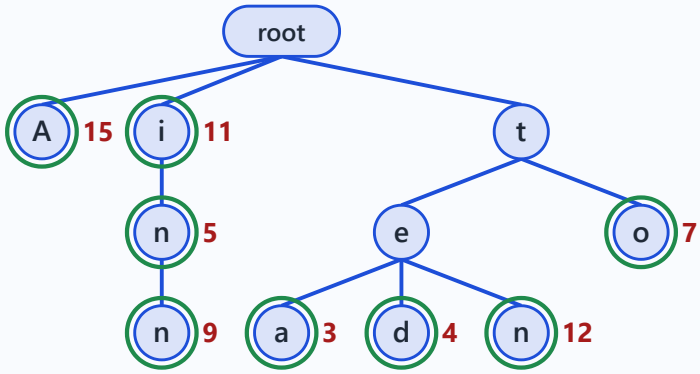


Figure 11.5: A trie that carries values. The numbers are the values stored in the nodes. The “t” and “te” nodes carry no value; they are only prefixes.

There are eight keys in the figure, and each ends at a node. “i”, “in” and “inn” form a chain in which each is a prefix of the next; all three carry values. The “t” node splits into two branches: “to” and “te”. “te” has three children: tea, ted and ten. The number of keys is 8, and the number of nodes other than the root is 10: A, i, in, inn, t, te, tea, ted, ten, to. The two nodes that carry no value (t and te) merely form the path.

There is one more detail in this example: “A” is an uppercase letter. An ordinary trie that holds its children in an array of 26 elements can store only lowercase letters. For a key with an uppercase letter, either a wider alphabet is needed or the children are kept in a `HashMap`. As the alphabet grows, the memory per node grows too; this cost is the weakest point of the trie.

11.2 The node class

11.2.1 The TrieNode class

The most direct way to build a trie in Java is to keep, in each node, a child reference for every letter of the alphabet. If the alphabet consists of lowercase English letters, an array of 26 elements in each node suffices. Cell *i* of the array points to the child reached from the node by the *i*-th letter; if there is no continuation with that letter, the cell is `null`.

```
public class TrieNode {
  TrieNode[] child;  // 'a'..'z'
  boolean isWord;      // does a word end here?

  public TrieNode() {
    isWord = false;
    child = new TrieNode[26];
  }
}
```

The class has two fields. The `child` array holds 26 references; Java initializes all reference cells of a new array to `null`. `isWord` says whether the path from the root to this node is a word of the set; it is set to `false` in the constructor. Java gives this value by default as well, but writing it out makes the intent clear.

The class does not store the character itself. Which character a node represents is known from the index at which it sits in its parent's array. The node in cell `child[0]` of the root is “a”, and the node in cell `child[3]` is “d”. Storing the character separately buys nothing but wasted memory. In some variants, however, such as the ternary search trie, the character has to be kept inside the node.

The trie itself is a class that holds the root. All the methods given in these notes are to be taken as written inside this class and use the `root` field:

```
public class Trie {
  private final TrieNode root = new TrieNode();
  // insert, search, startsWith, complete, delete ... live here

}
```

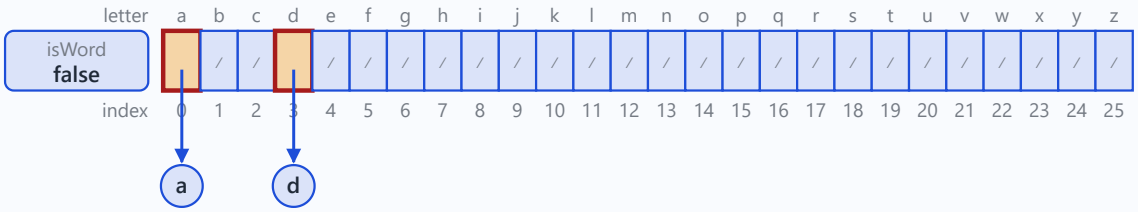


Figure 11.6: The `child` array of the root after and, ant, dad and do have been inserted. Of the 26 cells only 0 (a) and 3 (d) are filled; the / mark is `null`.

Figure 11.6 shows the root's array in full. Only two of the 26 cells are filled. This table is the first sign of the trie's memory problem. At the nodes below the root the situation is even sparser: the “an” node has 2 filled cells and the “a” node only 1. In the leaves all 26 cells of the array are empty, but the array is allocated all the same.

11.2.2 From character to index

To find the index of a character in the array, we use the fact that Java's `char` type is a number. A `char` stores the UTF-16 code value of the character and is converted to `int` in arithmetic. The codes of the lowercase letters are consecutive: 'a' is 97 and 'z' is 122. So the expression `ch - 'a'` maps 'a' to 0 and 'z' to 25.

`ch`	**Unicode**	`ch - 'a'`
'a'	97	0
'd'	100	3
'n'	110	13
'o'	111	14
't'	116	19
'z'	122	25

Table 11.2: The codes of some characters and their array indices.

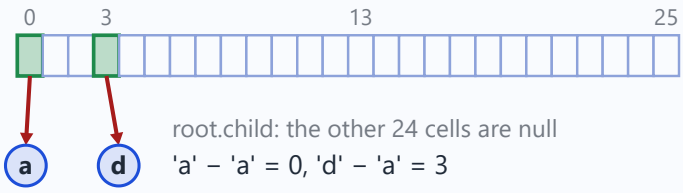


Figure 11.7: In the `root.child` array, the cells 'a' − 'a' = 0 and 'd' − 'a' = 3 are filled; the other 24 cells are `null`.

Because a child is reached through an array index, access is *O(1)*: `node.child[ch - 'a']`. For the word “and”, `child[0]`, `child[13]` and `child[3]` are used in turn. This speed is the greatest advantage of the array representation. Finding a child needs neither a comparison nor a hash computation.

The same formula does not work for characters outside the alphabet. The code of the uppercase 'A' is 65, and `'A' - 'a'` = *65 − 97 = −32*. The code of the Turkish letter 'ç' is 231: `'ç' - 'a'` = *231 − 97 = 134*. For 'ş' the result is *351 − 97 = 254*. None of these indices is in the range 0–25, and the array access throws an `ArrayIndexOutOfBoundsException`. There are two solutions: the input is first converted to lowercase and to the a–z alphabet, or the children are kept in a `HashMap` keyed by character.

Common mistake

Using `ch - 'a'` without checking the input. A word such as “Ankara” or “çay” causes an exception at the first character. A more insidious problem is a half-built path left in the trie when an insertion is interrupted: while “kaç” is being inserted, the 'k' and 'a' nodes are created and an exception is thrown at 'ç'. An unmarked “ka” path remains in the trie. The input must be checked before it is processed, or the alphabet must be chosen wide from the start.

11.3 Insertion

11.3.1 Algorithm and code

Inserting a word into a trie means walking the word's path from the root and completing the part that is missing. If the beginning of the path is shared with other words, that part already exists; insertion creates new nodes after the first missing character. When the path ends, the last node is marked.

1. Start at the **root**.
2. For each character of the word, look at the relevant child: `child[ch – 'a']`.
3. If the child is `null`, create a **new node**; if it exists (a common prefix), use the existing node.
4. Move on to that child.
5. When the characters run out, set `isWord = true` at the last node.

```
public void insert(String word) {
  TrieNode node = root;
  for (char ch : word.toCharArray()) {
    int i = ch - 'a';
    if (node.child[i] == null)
      node.child[i] = new TrieNode();
    node = node.child[i];
  }
  node.isWord = true;
}
```

The code follows these five steps line by line. The variable `node` starts at the root and goes down one level on each iteration of the loop. `toCharArray()` turns the word into a character array; the for-each loop yields the characters from left to right. For each character `i = ch - 'a'` is computed. If `node.child[i] == null`, there is no path continuing with that character yet; a new `TrieNode` is created and attached to the cell. If the condition is false, the path already exists and nothing is created. In both cases we descend to the child with `node = node.child[i]`. When the loop ends, `node` is the node of the last character of the word, and it is marked.

One property of the code stands out: insertion never changes an existing node; it only attaches new nodes and sets the last node's marker to `true`. If the same word is inserted a second time, the loop finds all the nodes in place, creates no new node and sets the already `true` marker to `true` again. The structure does not change. For this reason a trie behaves like a set: repeated words are stored once.

11.3.2 Step by step: and, ant, dad, do

Let and, ant, dad and do be inserted in order into an empty trie. At the start the trie consists of the root alone: all 26 children of the root are `null` and `isWord = false`. Table 11.3 gives the state of the relevant cell at each character and the work done.

Word	ch	i	child[i]	Action	Node
and	'a'	0	`null`	new node	a
	'n'	13	`null`	new node	an
	'd'	3	`null`	new node	and
	—	—	—	`isWord = true`	and ✓
ant	'a'	0	filled	move on (common prefix)	a
	'n'	13	filled	move on (common prefix)	an
	't'	19	`null`	new node	ant
	—	—	—	`isWord = true`	ant ✓

Word	`ch`	`i`	`child[i]`	Action	Node
dad	'd'	3	`null`	new node	d
	'a'	0	`null`	new node	da
	'd'	3	`null`	new node	dad
	—	—	—	`isWord = true`	dad ✓
do	'd'	3	filled	move on (common prefix)	d
	'o'	14	`null`	new node	do
	—	—	—	`isWord = true`	do ✓

Table 11.3: Inserting the words and, ant, dad, do in order. The "Node" column gives the prefix that `node` points to after the operation.

When "and" is inserted the trie is empty; all three characters create new nodes and the last node is marked. From the root the path `child[0] → child[13] → child[3]` has been built.

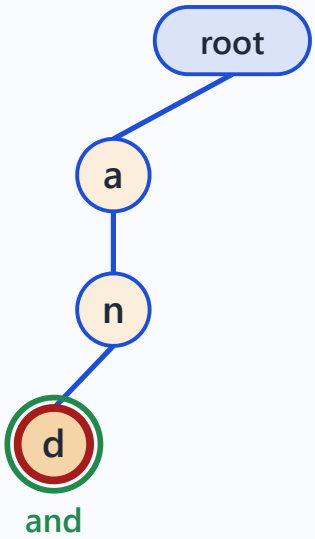


Figure 11.8: The trie after "and" is inserted: the root, a, an and the marked node and.

When "ant" is inserted, the first two characters find a ready path. For 'a', `child[0]` is filled, and for 'n', `child[13]` is filled; the loop only moves on. For 't', `child[19]` is empty and a single new node is created. The prefix "an" is shared between the two words. This sharing is the source of the trie's memory saving.

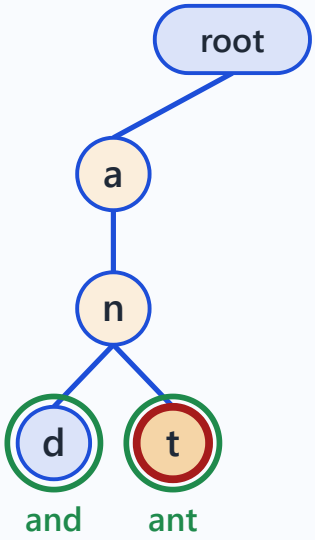


Figure 11.9: “ant” inserted. Only 1 new node (t) was needed; the prefix “an” was shared.

“dad” meets an empty cell (`child[3]`) at the root and builds its own branch with three new nodes. Note that the two 'd's in “dad” are different nodes: one is the child of the root, the other the child of “da”. The same character corresponds to different nodes under different prefixes. Finally “do” shares the 'd' node and creates a single new node for 'o'.

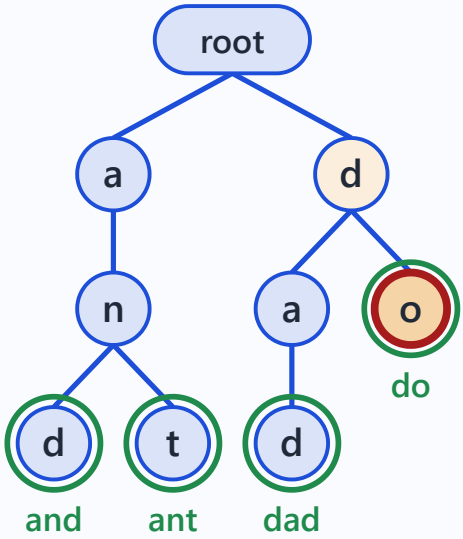


Figure 11.10: The final state: 4 words, 11 characters, 8 nodes not counting the root.

The numbers of new nodes created in the four insertions are 3, 1, 3 and 1, in that order; 8 in total. Since the total length of the words is 11, a saving of three characters has been achieved. Had the words been inserted in another order, the intermediate states would differ, but the final trie would be

the same. The shape of a trie depends only on the set of words, not on the insertion order. This is an important difference from the binary search tree, where the insertion order determines the height of the tree.

11.3.3 Code trace: insert("ant")

To see how the code works at the level of lines, we trace the call `insert("ant")` when the trie contains only "and". Table 11.4 gives the values of the variables at each step.

Step	Line	`ch`	`i`	Condition / action	`node`
1	`node = root`	—	—	start	root
2	loop	'a'	0	`child[0] == null` is false; no new node	a
3	loop	'n'	13	`child[13] == null` is false	an
4	loop	't'	19	`child[19] == null` is true → a new node is attached	ant
5	after the loop	—	—	`node.isWord = true`	ant ✓

Table 11.4: Trace of `insert("ant")`; at the start the trie contains only "and".

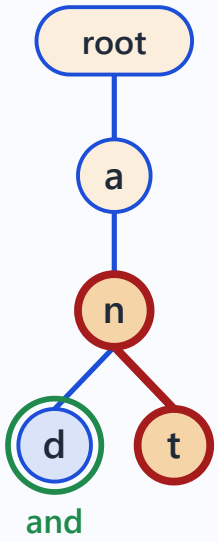


Figure 11.11: The fourth step: `child[19]` was found empty and the new "t" node (colored) was attached; `node` is still at "an".

The trace shows three points clearly. First, the condition is true only once; that is, only one node is created. Second, at the moment a new node is created its `isWord` value is `false`; the marker is set only after the loop has ended, and only on the last node. The intermediate nodes ("a", "an") stay unmarked. Third, the loop always walks over all the characters of the word; insertion never ends early. That is why insertion takes exactly L iterations in every case.

11.3.4 The cost of insertion

The loop runs once for each character of the word. In each iteration an index is computed, an array is accessed and, if needed, a node is created and an assignment is made. Creating a node means allocating an array of 26 cells. This takes time proportional to the alphabet size, but since the alphabet is taken as constant it is a constant cost. The total time is $O(L)$, where L is the length of the word. The time is completely independent of the number of words in the trie.

As for memory, one insertion creates at most L new nodes; this worst case occurs when no prefix of the word is in the trie. Each node carries 26 references and a boolean, so one insertion allocates at most $O(L \cdot M)$ memory. If the word lies entirely on an existing path (for example, “ka” inserted when “kalem” is present), no node is created at all; only the marker changes.

Example 11.2

Question. The trie contains only “kalem” and `insert("ka")` is called. How many nodes are created? Then what does `search("ka")` return?

Solution. For 'k' the cell `child[10]` of the root is filled; the loop moves on. For 'a' the cell `child[0]` of the “k” node is filled too. The loop ends and the “ka” node is marked. No node is created. The “ka” path existed before but was not a word; now it is marked, and `search("ka")` returns `true`. “kalem” is not affected by this operation.

Common mistake

Putting the marking inside the loop. If the line `node.isWord = true` is moved into the loop body, “a” and “an” are marked as well when “and” is inserted. Then `search("an")` returns `true` and the trie “finds” words that were never inserted. The marker must be set only on the last node.

11.4 Search and prefix queries

11.4.1 The search algorithm

Search is the read-only counterpart of insertion. The same path is walked; the difference is that when a missing branch is met, no node is created and the search ends at once.

1. Start at the root.
2. For each character, go to the relevant branch (child).
3. If the branch does not exist (`null`), the word is not in the trie.

4. On reaching the end of the word, check `isWord`: if it is `true`, the word has been found; if it is `false`, the word is absent and is only a prefix of an existing word.

```
public boolean search(String word) {
  TrieNode node = root;
  for (char ch : word.toCharArray()) {
    int i = ch - 'a';
    if (node.child[i] == null)
      return false;
    node = node.child[i];
  }
  return node.isWord;
}
```

The code is almost the same as insertion. Two lines differ. First: if the child is `null`, no new node is created; we leave with `return false`. Second: after the loop ends the marker is not set but read: `return node.isWord`. This last line is the most important line of the search. Even if all the characters of the path have been found, the word is not in the trie unless the last node is marked.

Search also takes *O(L)*. If the word is not in the trie, the loop may end earlier: if the path breaks at the *k*-th character, only *k* iterations run. Even with a million words in the trie, “dad” is found, or found absent, in at most 3 steps.

11.4.2 Code trace: search("dad")

Let the trie consist of the words and, ant, dad and do. The trace of the call `search("dad")` is in Table 11.5.

Iteration	`ch`	`i`	`child[i]`	`node` (after)
—	—	—	—	root
1	'd'	3	filled, continue	d
2	'a'	0	filled, continue	da
3	'd'	3	filled, continue	dad
end	the word has ended; `node.isWord` = `true`			result `true`

Table 11.5: Trace of `search("dad")`; trie: and, ant, dad, do.

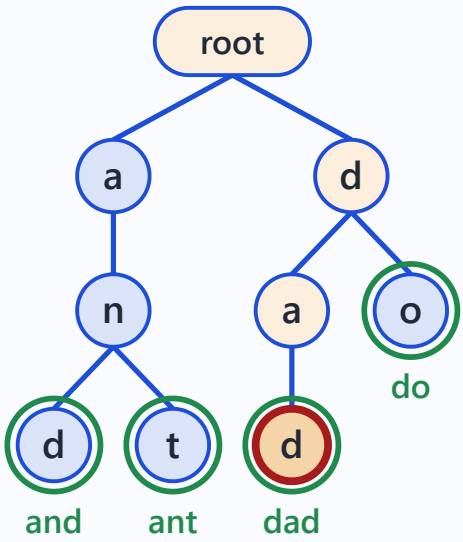


Figure 11.12: The last step of `search("dad")` : `node` is at the “dad” node and the node is marked; the method returns `true` .

The same index (3) is used in the first and third iterations, but in the arrays of different nodes: the first is the root's array, the second is that of “da”. Each iteration of the search works one level further down. After three iterations the path is complete and the last node's marker is read.

11.4.3 The three outcomes of a search

A search can end in three different ways. Telling the three apart is also the way to understand the difference between search and a prefix query.

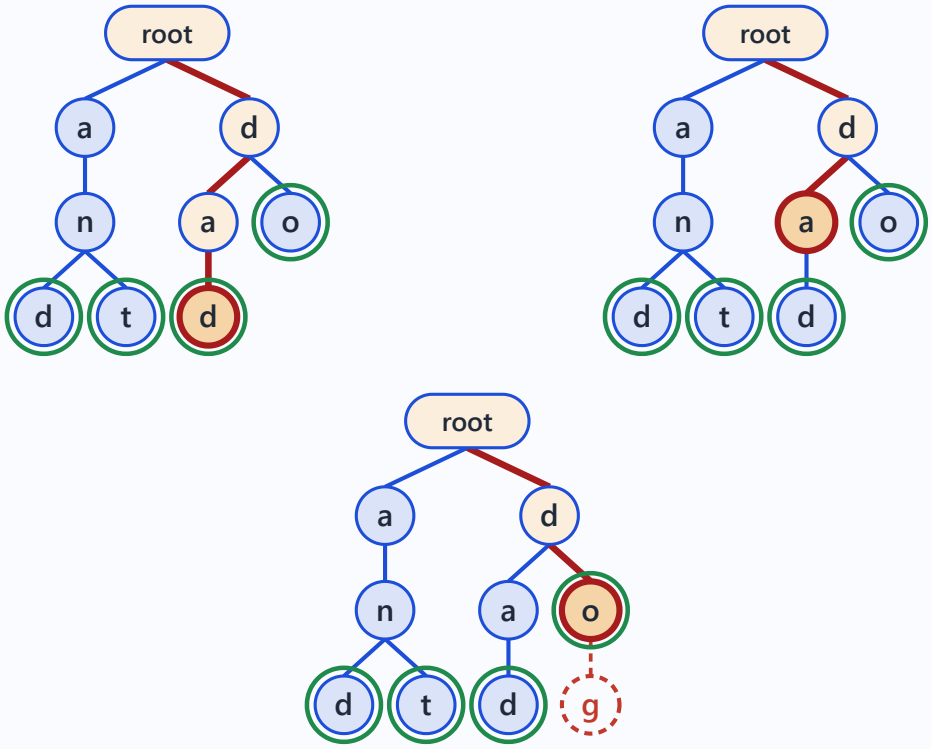


Figure 11.13: From left to right: `search("dad")`, `search("da")` and `search("dog")`. The dashed red circle is a node that was looked for but is not in the trie.

- `search("dad")` → `true`. All the characters are found and the last node is marked.
- `search("da")` → `false`. The path exists, but at the "da" node `isWord = false`. "da" is only a prefix of "dad". The loop completes two iterations and the last line returns `false`.
- `search("dog")` → `false`. The cell `child[6]` of the "do" node ('g' − 'a' = 6) is `null`. There is no branch; the loop exits early in the third iteration. Even if "do" is a word, this does not change the result.

The second and third cases return the same value (`false`), but for different reasons. In the second the path is complete and what is missing is the marker. In the third the path is broken. This difference is decisive in the prefix query: the prefix query returns `true` in the second case and `false` in the third.

11.4.4 Prefix query

A **prefix query** asks whether the trie contains at least one word that starts with the given prefix. It is the counterpart, over a set, of the `startsWith` method of Java's `String` class. In a trie this question is the same as asking whether the path of the prefix exists. Every node in the trie lies on the path of at least one word, because nodes are created only during insertion and deletion leaves no unmarked leaf. So if the path exists, there is also a word that starts with that prefix.

```java
public boolean startsWith(String prefix) {
  TrieNode node = root;
  for (char ch : prefix.toCharArray()) {
    int i = ch - 'a';
    if (node.child[i] == null)
      return false;
    node = node.child[i];
  }
  return true;  // search: node.isWord
}
```

The code is the same as `search`; the only difference is the last line. If `startsWith` finishes the loop without the path breaking, it returns `true` and does not look at the last node's marker. The comment line spells out the difference: at this point `search` returns `node.isWord`. The time is *O(P)*, where *P* is the length of the prefix.

Query	`search`	`startsWith`	Explanation
"dad"	`true`	`true`	path complete, last node marked
"da"	`false`	`true`	path complete, last node unmarked
"an"	`false`	`true`	path complete, last node unmarked
"dog"	`false`	`false`	no 'g' branch after “do”

Table 11.6: Results of `search` and `startsWith`; trie: and, ant, dad, do.

The results in the table show the one difference between the two methods: for queries that end at an unmarked node, `search` returns `false` and `startsWith` returns `true`. For queries where the path breaks, both return `false`.

For a code trace, take two calls: `startsWith("da")` and `startsWith("dog")`. In the first call `node` is set to the root. For 'd', `i = 3`, `child[3]` is filled and we descend to the “d” node. For 'a', `i = 0`, `child[0]` is filled and we descend to “da”. The prefix has ended and the path has not broken; the method returns `true`. Because the “da” node is unmarked, `search("da")` would have given `false` here.

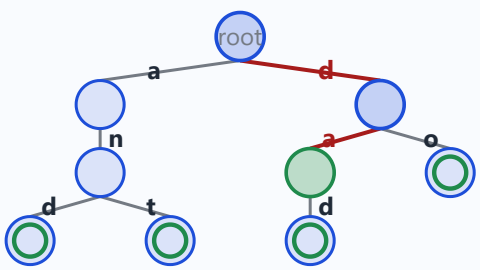


Figure 11.14: `startsWith("da")`: we descended to the “da” node without the path breaking; the result is `true`.

In the second call `node` is set to the root again. For 'd' (`i = 3`) and 'o' (`i = 14`) the cells are filled; we descend to the “do” node. For 'g', `i = 6`, and the cell `child[6]` of the “do” node is `null`. The method returns `false`. “do” is a word, but there is no word that starts with “dog”.

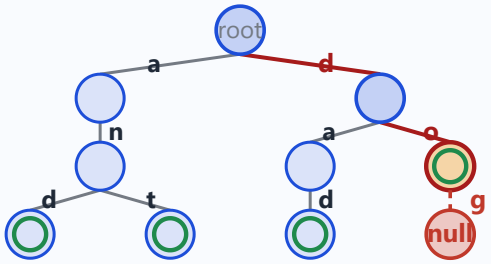


Figure 11.15: `startsWith("dog")` : the “do” node has no 'g' branch; the result is `false`.

Common mistake

Confusing `search` with `startsWith`. If `return true` is written on the last line of `search`, the method silently turns into a prefix query. It still returns `false` when a branch is not found, but every prefix that has a path counts as a word. This bug may not show in small tests: tests with real words such as “dad” and “and” give correct results. The bug appears only when it is tried with an unmarked prefix such as “da”. Search tests should always include at least one unmarked prefix.

11.4.5 Prefix counts

`startsWith` says whether there is a word that starts with a prefix, but not how many. The question “how many words start with an?” requires counting the marked nodes in the subtree of the prefix node. This count walks over all the nodes of the subtree. If the subtree has S nodes and 26 cells are checked at each node, the query takes $O(P + S \cdot M)$. If the dictionary is large and the prefix is short (for example a single letter), this means walking a large part of the dictionary.

A better way is to do the counting during insertion. A field `int count` is added to the `TrieNode` class. This field holds the number of words passing through the node's path. Insertion starts at the root and increments the counter of every node it passes through. During a query we descend to the prefix node and read the counter.

```
void insert(String word) {
  if (search(word)) return;  // don't recount
  TrieNode node = root;
  node.count++;
  for (char ch : word.toCharArray()) {
    int i = ch - 'a';
    if (node.child[i] == null) node.child[i] = new TrieNode();
    node = node.child[i];
    node.count++;       // one more word passed
  }
  node.isWord = true;
}
int prefixCount(String prefix) {
  TrieNode node = root;
  for (char ch : prefix.toCharArray()) {
    node = node.child[ch - 'a'];
    if (node == null) return 0;
  }
  return node.count;
}
```

If the word already exists, `insert` returns at once because of the `search` check; the counters grow only for a new word. The root's counter is incremented before the loop; this way the root holds the total number of words in the trie. The line `node.count++` in the loop increments the counter of every node, newly created or existing. In `prefixCount`, if the path breaks, 0 is returned; when the path ends, the prefix node's counter is returned. The loop descends to the child first and then checks for `null`; this order differs from that in `search` but does the same job.

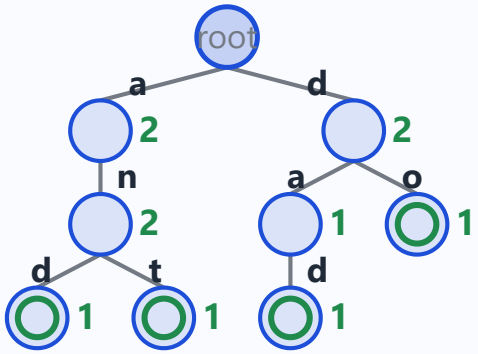


Figure 11.16: The `count` counters (green) in the trie of and, ant, dad, do. The root has 4, “a” and “an” have 2, “d” has 2, and “da” and “do” have 1.

Query	Result	Reason
`prefixCount("an")`	2	and and ant pass through the “an” node
`prefixCount("da")`	1	only dad passes through

Query	Result	Reason
`prefixCount("x")`	0	the cell `child[23]` of the root is empty

Table 11.7: `prefixCount` queries; trie: and, ant, dad, do.

The counter method reduces the query to *O(P)*. Its price is one `int` per node; in a node of 26 references this extra memory is very small. In deletion, however, all the counters along the path must be decremented by one. If the decrement is forgotten, the counters keep counting deleted words. If the same word were inserted twice, the counters would be incremented twice; that is why `insert` begins with the check `search(word)` and returns without recounting a word that is already in the trie.

Common mistake

Incrementing the counter only at the end of the word. If the line `node.count++` is taken out of the loop and run only at the last node, the counter turns into "the number of words ending at this node". The value at the "an" node becomes 0 and `prefixCount("an")` gives a wrong result. The counter must be incremented at every node of the path.

11.4.6 Wildcard search

A program that solves puzzles or plays word games may ask: "is there a word of the form d, an unknown letter, d?" In the pattern, a dot (`.`) stands for any letter. This kind of query is called **wildcard search**. An ordinary letter reduces the search to a single path. A dot, however, requires trying every filled branch at that level. The search is therefore written recursively.

```
boolean wildcard(TrieNode node, String pat, int j) {
  if (node == null) return false;
  if (j == pat.length()) return node.isWord;
  char ch = pat.charAt(j);
  if (ch != '.')
    return wildcard(node.child[ch - 'a'], pat, j + 1);
  for (int i = 0; i < 26; i++)
    if (wildcard(node.child[i], pat, j + 1)) return true;
  return false;
}
```

The method takes three parameters: the current node `node`, the pattern `pat` and the index `j` of the next character in the pattern. The first call has the form `wildcard(root, pattern, 0)`. If the node is `null`, this branch does not exist and `false` is returned. If the pattern has ended (`j == pat.length()`), the result is the node's marker, just as on the last line of `search`. If the next character

is a letter, we descend only into that letter's child. If it is a dot, each of the 26 children is tried in turn; if a match is found in one, `true` is passed straight up and the remaining branches are not tried. A call made for an empty child returns at once on the first line.

Pattern	Result	Match / reason
"d.d"	`true`	dad
".n."	`true`	and (ant is never looked at)
".o"	`true`	do
"b.."	`false`	the root has no 'b' branch
"...."	`false`	no word with 4 letters

Table 11.8: Results of wildcard search; trie: and, ant, dad, do.

Example 11.3

Question. Which nodes does the call `wildcard(root, "d.d", 0)` pass through?

Solution. With `j = 0`, the character 'd' is a letter; we descend only to `root.child[3]`, the "d" node. With `j = 1` the character is a dot; the 26 children of "d" are tried in turn. For `i = 0` the "da" node exists and `wildcard(da, "d.d", 2)` is called. In this call the character is 'd' and we descend to the "dad" node. `j = 3` equals the length of the pattern, and since "dad" is marked, `true` is returned. The value is passed up; the loop at the "d" node never reaches `i = 1`. The "do" branch (`i = 14`) is not tried.

The cost depends on the pattern and the trie. If the pattern has k dots, at most 26^k branches are tried in the worst case. But because empty branches return at once, the real cost is bounded by the paths that actually exist in the trie. The pattern "...." walks the whole trie down to the fourth level at most and finds no 4-letter word. A hash table can answer this query only by trying all the words one by one: there is no way to turn the pattern into a known key.

11.4.7 Spell checking: edit distance on a trie

A spell checker offers suggestions for a word that is not in the dictionary. The suggestions are dictionary words "close" to the word. The measure of closeness is usually the **edit distance**: the smallest number of single-character insertions, deletions and substitutions needed to turn one string into the other. This measure is also called the Levenshtein distance. The distance between "dod" and "dad" is 1 (o → a). The distance between "dod" and "do" is also 1 (the last d is deleted).

The distance between two strings is computed by dynamic programming, filling in a table. Let the query string be q and the dictionary word w. Row i of the table holds the distances between the first i characters of w and all prefixes of q. A row is computed from the previous row by the following rule:

> $S_i[0] = i,\ \ S_i[j] = \min(\, S_i[j-1] + 1,\ \ S_{i-1}[j] + 1,\ \ S_{i-1}[j-1] + (w_i \neq q_j\ ?\ 1 : 0)\,)$

The three terms are, in order, the cost of an insertion, a deletion and a substitution (or of a free move if the characters are equal). The last cell of the last row is the distance between the two strings.

The naive method fills a separate table for each word in the dictionary. But row i of the table depends only on the first i characters of the word. Words that have a common prefix share these rows. A trie provides exactly this sharing: each node is a prefix, and a node's row is computed from its parent's row in one step. The trie is walked depth first; one row is produced at each node. The row of the "d" node is computed once and used for both "dad" and "do".

Pruning is added to the walk. If the smallest value of a row exceeds k, no word below this node can be within distance k of the query. Every cell of a new row is obtained, directly or through its left neighbor, by adding 0 or more to a cell of the previous row. So the smallest value of a row does not decrease as the depth increases. The subtree of such a node is not entered.

Node	"" d o d	Minimum	Last	Decision
(root)	0 1 2 3	0	—	initial row
a	1 1 2 3	1	—	continue
an	2 2 2 3	2	—	prune (minimum 2 > 1)
d	1 0 1 2	0	—	continue
da	2 1 1 2	1	—	continue
dad	3 2 2 1	1	1	suggest
do	2 1 0 1	0	1	suggest

Table 11.9: Query "dod", $k = 1$; trie: and, ant, dad, do. The columns of a row are the prefixes "", d, o, d, in order. The "Last" column is meaningful only at word nodes.

Example 11.4

Question. How are the rows of "a" and "dad" computed, starting from the root's row?

Solution. The root's row holds the distances between the empty prefix and the prefixes of "dod": 0, 1, 2, 3. In the row of "a" the first cell is 1. The second cell is $\min(1 + 1, 1 + 1, 0 + [a \neq d]) = \min(2, 2, 1) = 1$. The third cell is $\min(1 + 1, 2 + 1, 1 + [a \neq o]) = 2$. The fourth cell is $\min(2 + 1, 3 + 1, 2 + [a \neq d]) = 3$. The row is 1, 1, 2, 3. The row of "dad" is computed from the row of "da" (2, 1, 1, 2): the first cell is 3; then $\min(4, 2, 2 + 0) = 2$; $\min(3, 2, 1 + 1) = 2$; $\min(3, 3, 1 + 0) = 1$. The row is 3, 2, 2, 1, and the last cell, 1, is the distance between "dad" and "dod".

At the end of the walk the suggestions are dad (o → a) and do (the last d is deleted). The minimum of the "an" row is 2, which exceeds 1; the "and" and "ant" nodes are never entered. Indeed the distance between "and" and "dod" is 2, and that with "ant" is 3. The nodes visited are a, an, d, da, dad and do.

Each row costs $O(|q|)$ work. Thanks to pruning, most of a large dictionary is never visited. The same walk also works with a weighted distance that counts neighboring letters on the keyboard as cheaper. But as k grows, pruning weakens and the number of visited nodes rises quickly; in applications k is mostly chosen as 1 or 2.

11.5 Autocomplete and the longest common prefix

11.5.1 complete and collect

Autocomplete suggests possible words according to the prefix typed. In a trie this is done in two phases. First the prefix node is found; this step is the same as the prefix query. Then the subtree of this node is walked depth first, and the word of every marked node is added to the list.

```
List<String> complete(String prefix) {
  List<String> result = new ArrayList<>();
  TrieNode node = findNode(prefix);
  if (node != null)
    collect(node, new StringBuilder(prefix), result);
  return result;
}
void collect(TrieNode node, StringBuilder path,
             List<String> result) {
  if (node.isWord) result.add(path.toString());
  for (int i = 0; i < 26; i++)
    if (node.child[i] != null) {
      path.append((char) ('a' + i));
      collect(node.child[i], path, result);
      path.setLength(path.length() - 1);
    }
}
```

`complete` first looks for the prefix node with the helper `findNode`. The helper is the version of `startsWith` that returns the node itself instead of a `boolean`:

```
TrieNode findNode(String prefix) {
  TrieNode node = root;
  for (char ch : prefix.toCharArray()) {
    node = node.child[ch - 'a'];
    if (node == null) return null;  // path broke
  }
  return node;
}
```

If there is no prefix node, `findNode` returns `null` and `complete` gives an empty list. If the node exists, `collect` is called. The `path` parameter of `collect` is a `StringBuilder` that holds the string from the root to the current node; at the start it is the prefix itself. The method first looks at the node's marker: if the node is an end of word, the current value of `path` is added to the result. Then the 26 indices are walked in turn. For each filled child three things are done: the child's character is appended to `path`, `collect` is called recursively for the child, and when the call returns, the appended character is removed.

The last step, `path.setLength(path.length() - 1)`, is a **backtracking** step. All calls share the same `StringBuilder` object. If, when returning from a subtree, the character appended for that subtree is not removed, the next sibling's word is built with a wrong prefix. Each call leaves `path` as it was handed to it.

11.5.2 Code trace: complete("te")

Let the trie consist of the words a, to, tea, ted, ten, i, in and inn. This is the lowercase version of the trie that carries values. We trace the call `complete("te")`.

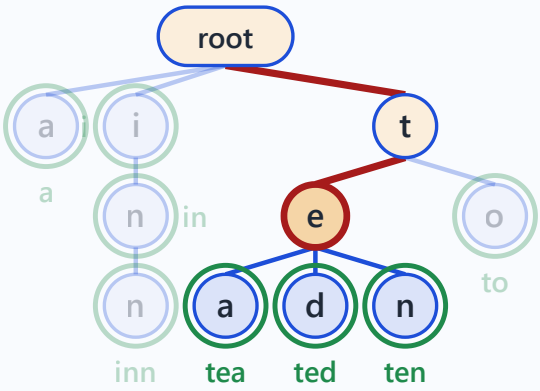


Figure 11.17: `findNode("te")` finds the "te" node by descending t → e from the root.

Step	Call / action	`path`	`result`
1	`findNode("te")` → the "te" node	te	[]
2	`collect(te)` : unmarked, not added	te	[]
3	`i = 0` ('a') filled → append, `collect(tea)`	tea	[]
4	"tea" marked → added; no children	tea	[tea]
5	return, last letter removed	te	[tea]
6	`i = 3` ('d') filled → `collect(ted)`	ted	[tea, ted]
7	return, last letter removed	te	[tea, ted]
8	`i = 13` ('n') filled → `collect(ten)`	ten	[tea, ted, ten]

Step	Call / action	`path`	`result`
9	return, last letter removed; the loop ends	te	[tea, ted, ten]

Table 11.10: Trace of `complete("te")`. The indentation shows the recursion depth.

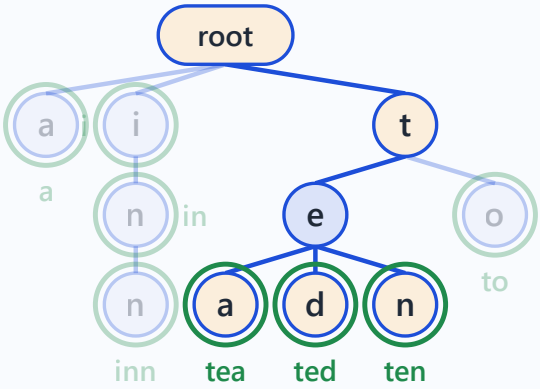


Figure 11.18: The result of `complete("te")`: [tea, ted, ten]. Only the subtree of the "te" node was walked; the faded nodes were never looked at.

The result is `[tea, ted, ten]` and is in alphabetical order. This order is no coincidence: the loop walks the indices from 0 to 25, and index order is letter order (a < d < n). Moreover, a node is added to the list before its children; so a prefix comes before the words that extend it. The output is the same whatever order the words were inserted into the trie.

The cost has two parts. Finding the prefix node takes $O(P)$. If the subtree has S nodes, each is visited once and 26 cells are checked at each visit: $O(S \cdot M)$. The total is $O(P + S \cdot M)$. In addition, `toString()` makes a copy for each word found; the total length of these copies is the size of the output. If the prefix is short, the subtree can be large. That is why applications usually settle for the first few suggestions. To show the most frequently used words first, a usage frequency is also stored in the nodes.

Example 11.5

Question. The words bal, ba, bas, bat and baba were inserted in that order. What do `complete("ba")`, `complete("bx")` and `complete("bas")` return?

Solution. "ba" is marked and is added to the list first. Then the children are walked in index order: b (1), l (11), s (18), t (19). The result is `[ba, baba, bal, bas, bat]`. The "bx" node does not exist, `findNode` returns `null` and the result is an empty list. "bas" is marked and has no children: `[bas]`.

Common mistake

Forgetting to backtrack. If the line `path.setLength(...)` is deleted, the characters 'a', 'd', 'n' pile up at the "te" node. The result becomes `[tea, tead, teadn]`. After every recursive call, shared state must be restored to what it was before the call.

11.5.3 The longest common prefix

The **longest common prefix** of a set of strings is the longest beginning that all the strings in the set share. In a trie this prefix is a chain of single branches starting at the root. We descend from the root; if the node has a single child and is not the end of a word, we move on. The characters passed are the answer.

```
String longestCommonPrefix() {
  StringBuilder sb = new StringBuilder();
  TrieNode node = root;
  while (!node.isWord) {
    int only = -1, kids = 0;
    for (int i = 0; i < 26; i++)
      if (node.child[i] != null) { kids++; only = i; }
    if (kids != 1) break;  // fork or leaf
    sb.append((char) ('a' + only));
    node = node.child[only];
  }
  return sb.toString();
}
```

The loop condition is `!node.isWord`: we stop on reaching a marked node. In the loop body the 26 cells are counted; `kids` holds the number of filled cells and `only` the last filled index seen. If `kids` is not 1 (if it is 0 we have reached a leaf, if 2 or more a fork), the loop is broken. If there is a single child, its character is appended and we descend to the child.

Node `node`	`isWord`	`kids`, `only`	**Action**	`sb`
root	`false`	1, 10 ('k')	'k' appended	k
k	`false`	1, 0 ('a')	'a' appended	ka
ka	`false`	1, 17 ('r')	'r' appended	kar
kar	`true`	—	the loop ends	kar

Table 11.11: Trace of `longestCommonPrefix()`; trie: kar, kart, karton.

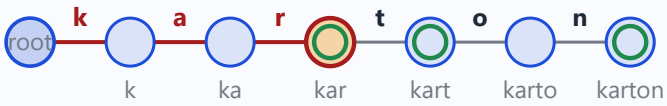


Figure 11.19: The trie of kar, kart, karton is a single chain. The descent stops at the “kar” node, because “kar” is a word.

The “kar” node has a single child (t), but the descent stops there. Why? A common prefix must be a prefix of *every* word in the set. The word “kar” itself cannot have a prefix longer than “kar”. Without the marker check in the condition, the descent would continue to “karton” and wrongly return “karton”. If the trie held only kart and karton, the descent would continue to “kart” and there `isWord` would stop it: the answer would be “kart”.

The cost is $O(P \cdot M)$, where P is the length of the answer: 26 cells are counted at each level. If the root has more than one child, the loop breaks in the first iteration and the answer is the empty string; for example, the set and, ant, dad, do has no common prefix.

11.6 Deletion

11.6.1 Three cases

Deletion is the trie operation that needs the most care. The path of a word may be shared with other words. Deleting the path from start to end would also destroy the other words that share the prefix. Deletion must therefore remove only the nodes that are *specific* to the word. Three cases arise, depending on the word's position in the trie.

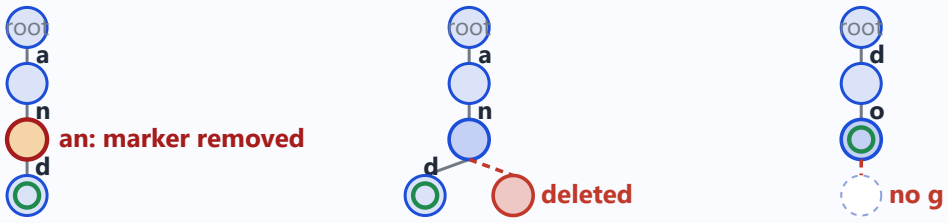


Figure 11.20: The three cases of deletion. Left, “an” is deleted: only the marker is removed. Middle, “ant” is deleted: the t node is removed and we stop at the prefix “an”. Right, “dog” is looked for: the word is not in the trie.

1. **The word is a prefix of another word.** The last node has children. Only `isWord = false` is set; the node is not deleted, because it lies on the path of other words.
2. **The word has a tail of its own.** From the end back toward the start, nodes that have no children and are unmarked are deleted. We stop on reaching a shared prefix or the end of another word.

3. **The word is not in the trie.** A branch is not found or the last node is unmarked. Nothing changes.

The “tail” in the second case is the chain of nodes, going up from the word's last node, that serve only this word. A node can be deleted if two conditions hold together: it has no children and it is unmarked. If it has children, it is on the path of another word. If it is marked, it is itself a word. The tail ends at the first node that breaks one of these two conditions.

Example 11.6

Question. The trie holds “apple” and “app”. How does the structure change when “app” is deleted?

Solution. The second 'p' node, the last node of “app”, has a child: 'l'. “apple” continues below this node. The node cannot be deleted; only its marker is removed (case 1). After that `search("app")` returns `false` and `search("apple")` returns `true`. The number of nodes does not change.

11.6.2 Step by step: app, apple, apt

Let the trie consist of the words app, apple and apt. Apart from the root there are 6 nodes: a, ap, app, appl, apple, apt. Both scenarios below start from this initial trie.

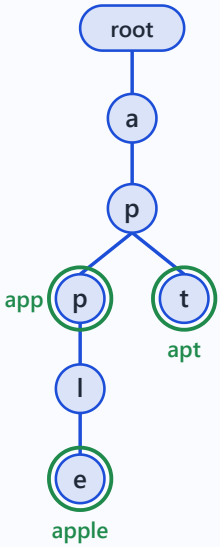


Figure 11.21: The trie of the words app, apple and apt.

Scenario 1: `delete("app")`. The path a → p → p is followed. The “app” node is marked, that is, the word exists. The marker is removed: `isWord = false`. Because the node has an 'l' child, it is not deleted. This is case 1. “apple” can still be found; the number of nodes stays at 6.

Scenario 2: `delete("apple")`. The path a-p-p-l-e is followed and “apple” is found. The marker of the 'e' node is removed. This node has no children; its parent deletes it (`child[4] = null`). Then the 'l' node is examined: it is now childless and unmarked, so it is deleted too. Next we come to the “app” node. This node is marked, because “app” is another word. The deletion stops here.

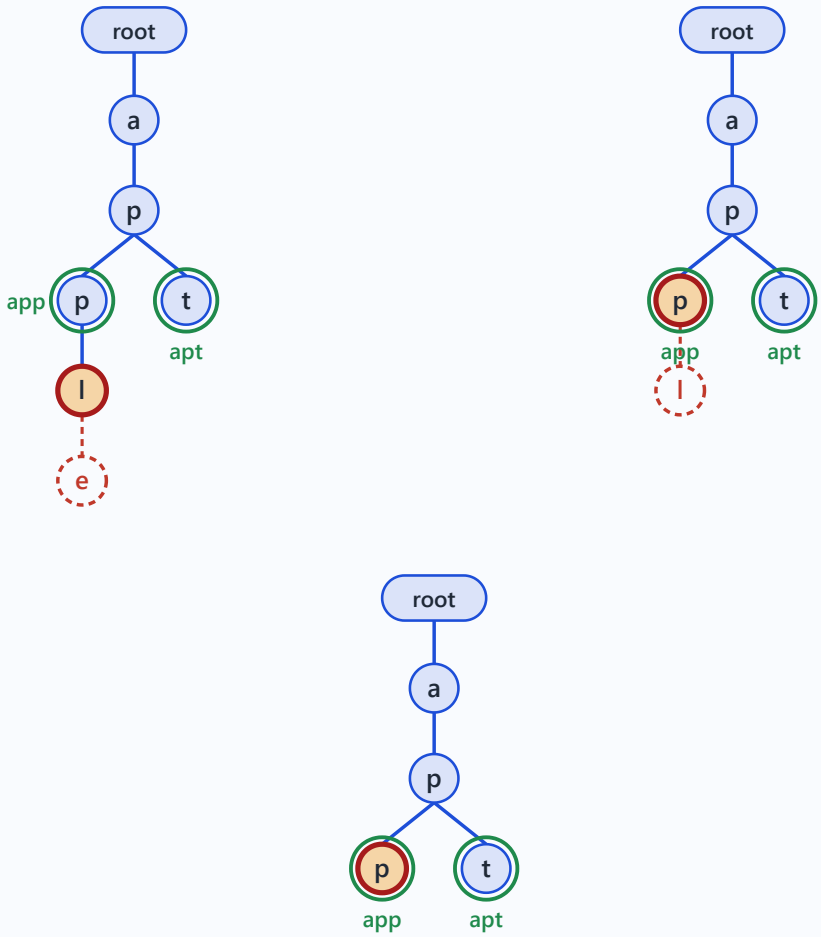


Figure 11.22: `delete("apple")` : first the 'e', then the 'l' node is deleted; we stop at the marked “app” node. Result: app and apt, 4 nodes not counting the root.

The two scenarios show that deletion decisions are made *from the bottom up*. Whether a node is deleted depends on whether its children are deleted. The 'l' node could not have been deleted at first, because it had the child 'e'. After 'e' was deleted, 'l' became deletable. This dependence is why deletion is naturally written recursively: first we descend to the end of the word, and the decisions are made on the way back.

11.6.3 Recursive code

```
// true: parent may drop node
boolean delete(TrieNode node, String word, int j) {
  if (j == word.length()) {    // word ended
    if (!node.isWord) return false;
    node.isWord = false;
    return isEmpty(node);       // no children?
  }
  int i = word.charAt(j) - 'a';
  TrieNode next = node.child[i];
  if (next == null) return false;  // no such word
  if (delete(next, word, j + 1)) {
    node.child[i] = null;       // unlink child
    return !node.isWord && isEmpty(node);
  }
  return false;
}
```

The method's return value is a decision passed to the parent: `true` means that the parent may delete this node. The call has the form `delete(root, word, 0)`; the root's return value is ignored, because the root is never deleted. `j` is the index of the next character in the word.

The method has two parts. The first is the base case: if `j == word.length()`, we have reached the word's last node. If the node is unmarked, the word is not in the trie and `false` is returned (case 3). If it is marked, the marker is removed and we check whether the node has children. If it has none, `true` is returned: the parent may delete this node. If it has children, `false` is returned (case 1).

The second part is the recursive case. The child `next` for the next character is found. If `next` is `null`, the word is not in the trie and `false` is returned (case 3). If it exists, `delete(next, word, j + 1)` is called. If the call returns `true`, the child can be deleted: the link is cut with `node.child[i] = null`. Then the same question is asked about this node: is it unmarked and now childless? The answer is passed up one level. If the call returns `false`, nothing is deleted and `false` is passed up. Once the chain is broken at one level, no node above is deleted. In a trie with prefix counters, the word's existence is first checked with `search`; if the word exists, `node.count--` is done at every node during the descent.

The code uses the helper `isEmpty`. The helper checks whether all 26 cells are `null`:

```
boolean isEmpty(TrieNode node) {
  for (int i = 0; i < 26; i++)
    if (node.child[i] != null) return false;
  return true;
}
```

In Java, “deleting” a node means setting the last reference to it to `null`. The detached node is no longer reachable and the garbage collector reclaims its memory. That is why the code only empties the cell in the parent.

11.6.4 Code trace: delete("dad")

Let the trie consist of the words and, ant, dad and do. We trace the call `delete(root, "dad", 0)`. The recursion first descends three levels, and decisions are made on the way back. Table 11.12 gives the descent and return steps in order.

Step	Frame	`j`	Action	Returns
1	root	0	'd' → `i = 3`, `next` = “d”; `delete(d, "dad", 1)`	—
2	d	1	'a' → `i = 0`, `next` = “da”; `delete(da, "dad", 2)`	—
3	da	2	'd' → `i = 3`, `next` = “dad”; `delete(dad, "dad", 3)`	—
4	dad	3	`j == word.length()`; marked → `isWord = false`; `isEmpty(dad) = true`	`true`
5	da	2	`child[3] = null`; “da” is unmarked and childless	`true`
6	d	1	`child[0] = null`; “d” is unmarked but has the child 'o'	`false`
7	root	0	the call returned `false`; the root deletes nothing	`false`

Table 11.12: Trace of `delete(root, "dad", 0)`; trie: and, ant, dad, do. The frame is the call's node.

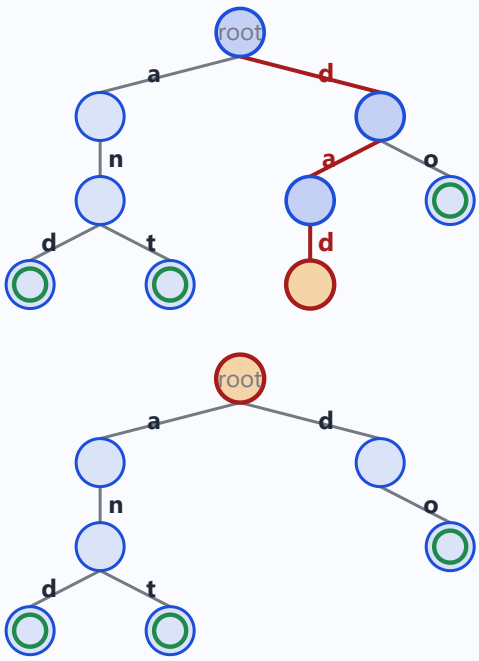


Figure 11.23: Left, the fourth step: the marker of the “dad” node has been removed. Right, the result: the “dad” and “da” nodes have been detached; the remaining words are and, ant, do.

Two nodes are deleted in the trace: “dad” and “da”. The “d” node is unmarked, but it has the child 'o' for “do”; `isEmpty` returns `false` and the chain breaks there. The call at the root gets `false` and the root's return value is ignored. The remaining trie holds the words and, ant and do; apart from the root there are 6 nodes.

The cost of deletion is computed as follows. The descent takes L levels. On the way back at most one `isEmpty` call is made per level, and each scans 26 cells. The total is $O(L \cdot M)$. If each node held a counter for its number of filled children, `isEmpty` would answer in constant time and deletion would drop to $O(L)$. This counter is incremented when a cell is filled during insertion and decremented when a cell is emptied during deletion.

Common mistake

Deleting a shared node. Deletion is sometimes written as “every node on the word's path, from the root on, is set to `null` ”. In a trie that holds app, apple and apt, this version empties the root's 'a' cell on the call `delete("app")` . All three words that begin with a are lost. A correct deletion only removes the marker and deletes, from the bottom up, nodes that are childless and unmarked. Because “app” has an 'l' child, no node is deleted.

Common mistake

Skipping the marker check on the way back. If the condition `!node.isWord &&` in the recursive part is removed, the call `delete("apple")` deletes the “app” node too, because after 'l' is detached “app” is left childless. The word “app” is thereby lost. A node can be deleted only if it is childless *and* unmarked.

11.7 Analysis

11.7.1 Complexity

The cost of trie operations depends on two quantities: the length of the string being processed and the size of the alphabet. L is the length of the word and P the length of the prefix. M is the alphabet size; for lowercase English letters it is 26. S is the number of nodes in the subtree of the prefix node.

Operation	Time	Explanation
`insert(word)`	$O(L)$	one step per character; at most L new nodes
`search(word)`	$O(L)$	ends earlier if a branch is missing
`startsWith(prefix)`	$O(P)$	the existence of the path is enough
`prefixCount(prefix)`	$O(P)$	with the `count` counter
`complete(prefix)`	$O(P + S \cdot M)$	depth-first walk from the prefix node

Operation	Time	Explanation
`delete(word)`	*O(L · M)*	`isEmpty` scans 26 cells; *O(L)* with a child counter

Table 11.13: Running times of the operations in an array-based trie.

The most important feature of the table is that a quantity is missing: in no row is there an n, the number of stored words. The *log n* factor of the binary search tree does not appear in a trie. Even if the trie stores a million words, the search for "dad" takes 3 steps. This also explains why it is wrong to state the cost of trie operations as *O(log n)* or *O(n)*.

It should not be forgotten, though, that L has a limit too. If a dictionary has n distinct words and the alphabet has M letters, at least some of the words must have a length of order $\log_M n$. With a short alphabet, a large number of distinct words can be built only from long words. Still, in real dictionaries L is small (most English and Turkish words are shorter than 15 letters) and the advantage of the trie holds in practice.

The memory requirement is $O(N \cdot M)$. N is the number of nodes, and is at most the total length of all the words plus 1 (the root). This bound is reached when no word shares a prefix with another. Since each node carries an array of M references, the total memory is proportional to $N \cdot M$.

11.7.2 Memory: a numeric example

Let us count the memory one item at a time for the trie of the words and, ant, dad and do.

Quantity	Value
Total characters	*3 + 3 + 3 + 2 = 11*
Number of nodes N	*1 (root) + 8 = 9*
Child references	*9 × 26 = 234*
Filled references	8 (the number of edges)
Empty (`null`) references	226 (≈ 97%)

Table 11.14: Memory breakdown of the trie of and, ant, dad, do.

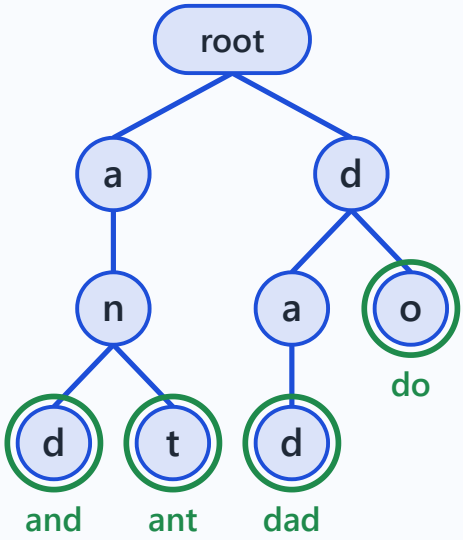


Figure 11.24: The trie of and, ant, dad, do: the root and 8 nodes, 8 edges.

The number of filled references equals the number of edges, because each edge is represented by exactly one filled cell. Since every node other than the root has a single parent in the tree, the number of edges is $N - 1 = 8$. The remaining $234 - 8 = 226$ cells are empty; the ratio $226 / 234 \approx 0.966$, about 97%. Common prefixes have reduced the 11 characters to 8 nodes, but arrays of 26 elements fill almost all of the memory with `null`.

At a larger scale the picture is more striking. A trie of a million nodes holds 26 million references. On a 64-bit Java virtual machine a reference is 8 bytes, or 4 bytes if compressed references are on. The reference arrays alone are $26 \cdot 10^6 \cdot 8 \approx 208 \cdot 10^6$ bytes, about 200 MB; with 4-byte references about 100 MB. To this is added the header information of each object. The problem worsens as the alphabet grows. The basic plane of Unicode has 65,536 characters; allocating an array of that size in every node is clearly impossible.

11.7.3 Strengths and weaknesses

The strengths of the trie are as follows:

- **Fast prefix-based search.** Prefix queries, prefix counts and autocomplete are natural operations.
- **Word search takes *O(L)*** and the time is independent of the number of words.
- **Common prefixes are stored once.** In sets with a lot of prefix sharing (dictionaries, URLs, file paths) the number of nodes stays far below the total number of characters.
- **Order information is preserved.** A depth-first walk yields the words in lexicographic order. There is no such thing as a hash collision; the worst case is also $O(L)$.

Its weaknesses are also clear:

- **High memory consumption,** especially with large alphabets. The space complexity is $O(N \cdot M)$.

- **No gain for sets with few common prefixes.** With random keys almost every character creates a new node.
- **Many small objects.** A search jumps between nodes that are far apart in memory. The processor's cache copes poorly with this access pattern. For this reason, between two structures of the same order, the trie may be slower in practice.

11.7.4 Ways of storing children

The logic of a trie is independent of how the children are stored. All that changes is how the operation "give me the child with character *ch*" is carried out. There are four common options.

Representation	Child access	Memory per node	Note
`TrieNode[26]` array	*O(1)*	*M* references	simple, fast; wasteful for sparse nodes
`HashMap<Character, TrieNode>`	*O(1)* avg.	as many as there are children	works for any alphabet (Turkish, Unicode); order is not preserved
`TreeMap` / sorted list	*O(log M)*	as many as there are children	alphabetical traversal is preserved
Ternary search trie (TST)	*O(log M)* avg.	3 references	smaller / equal / larger links in each node

Table 11.15: Ways of storing children.

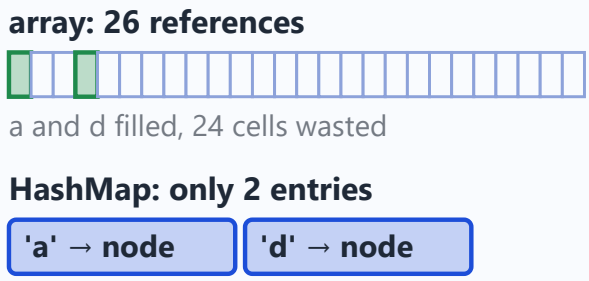


Figure 11.25: The array representation allocates 26 cells in every node; a `HashMap` stores only the children that exist.

The `HashMap` representation also solves the problem of characters outside the a–z alphabet, such as Turkish letters. The key is the character itself; there is no index computation. The insertion code becomes shorter:

```
class MapNode {
  Map<Character, MapNode> child = new HashMap<>();
  boolean isWord;
}

void insert(String word) {
  MapNode node = root;
  for (char ch : word.toCharArray())
    node = node.child.computeIfAbsent(ch, x -> new MapNode());
  node.isWord = true;
}
```

`computeIfAbsent` creates a new value with the given function and adds it if the key is not in the map, and returns the value; if the key exists, it returns the existing value. So the two steps of the array version, "create if `null`, then descend", shrink to a single call. "çay", "çam", "şeker" and "Ankara" can be inserted into this trie; the root's map then has the keys 'A', 'ç' and 'ş'. The price is a map object and a hash computation at every node. Also, because a `HashMap` keeps its keys unordered, the alphabetical output of autocomplete is lost. If sorted output is required, a `TreeMap` is used; access rises to *O(log M)*.

The choice is made by three questions: how large is the alphabet, how tight is the memory, is sorted output needed? For a small, fixed alphabet an array is fastest. For a large alphabet or one not known beforehand, a map is needed. If memory is the most important constraint, a ternary search trie or a compressed trie is preferred.

11.7.5 Trie, hash table and balanced BST

Three structures are often compared for storing string keys. Table 11.16 gives the typical costs for a set of n words of length L.

Operation	Trie	Hash table	Balanced BST
Exact word search	$O(L)$	$O(L)$ average	$O(L \cdot \log n)$
Prefix query	$O(P)$	not supported	$O(P \cdot \log n)$
k words starting with a prefix	$O(P + S \cdot M)$	scan the whole table	$O(P \cdot \log n + k \cdot L)$
Sorted listing	DFS, $O(N \cdot M)$	sorting needed	inorder traversal, $O(n)$
Worst case	$O(L)$	$O(n \cdot L)$ (collisions)	$O(L \cdot \log n)$
Memory	high	low	low

Table 11.16: Trie, hash table and balanced binary search tree (BST). *S*: the number of nodes in the subtree of the prefix node.

For exact word search the trie and the hash table are of the same order. This may look surprising at first, since a hash table is often quoted as "*O(1)*". But that *O(1)* holds when the key's hash value is taken as already computed. To compute the hash value of a string, all L characters are read. The

equality check also compares L characters. So the hash table is also $O(L)$. In a balanced BST, $\log n$ nodes are visited and two strings are compared at each visit; this comparison reads at most L characters.

The real difference of the trie is in prefix and order queries. A hash table supports prefixes not at all. Because a BST is sorted, it keeps the words that start with a prefix in a consecutive range and finds the start of the range in $O(P \cdot \log n)$. A trie answers the same question without the $\log n$ factor. In return the trie is the most expensive structure in memory. The choice depends on the kind of queries to be made: if only exact matches are needed, a hash table is enough; if prefix and order queries are frequent, the trie comes to the fore.

Example 11.7

Question. A trie holds $n = 1{,}000{,}000$ words; the word searched for has length $L = 8$. How many node steps does `search` take in the worst case? What is the situation in a balanced BST?

Solution. In the trie one descends to a child for each character; at most 8 steps are taken. In a balanced BST, $\log_2 10^6 \approx 19.9$, that is about 20 nodes are visited. At each visit at most 8 characters are compared; in the worst case about $20 \cdot 8 = 160$ characters are read. In a plain list the words are tried one by one and the work is $O(n \cdot L)$.

11.7.6 Sorting with a trie

The trie's property of yielding words in lexicographic order can be turned into a sorting algorithm. Let the input order be dog, ant, do, and, dad. All the words are inserted into the trie. Then a **preorder** traversal is made from the root: first the node itself is processed, then its children are visited in order from 'a' to 'z'. At every marked node the word is written. This traversal is the same as calling `collect` from the root.

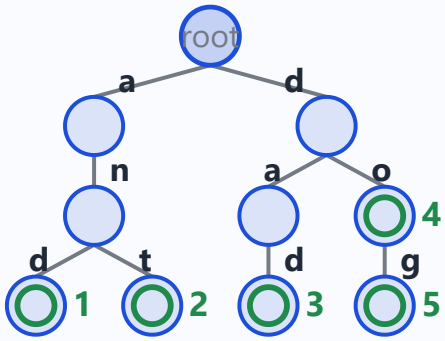


Figure 11.26: The trie of the words dog, ant, do, and, dad. The green numbers are the order in which the words are written in a preorder traversal.

The output is and, ant, dad, do, dog. Two rules guarantee this order. First, siblings are visited in index order, that is, in alphabetical order: all words in the “a” branch come before those in the “d” branch. Second, a node is written before its children: “do” comes before “dog”. This agrees with the definition of lexicographic order: a prefix is smaller than every string that extends it.

For the cost, let the total number of characters be *N*. Insertion takes *O(N)* in all. The traversal checks 26 cells at each node: *O(N · M)*. With a fixed alphabet this is *O(N)*. Comparison-based sorting, on the other hand, makes *O(n log n)* comparisons and each comparison can take *O(L)*; with words that have long common prefixes, the same characters are read again and again. The trie removes this repetition. The same idea underlies “burst sort” and the string versions of MSD radix sort. MSD radix sort also splits the words into buckets by the first character and divides each bucket by the second character; this tree of divisions is in fact a trie.

11.8 Applications and variants

11.8.1 Areas of use

The basic operations of the trie are the core of many real applications. Four areas stand out.

- **Autocomplete.** Search boxes and integrated development environments (IDEs) descend to the node of the typed prefix and suggest the words in the subtree. If usage frequency is also stored in the nodes, frequently used words are shown first.
- **Spell checking.** The dictionary is kept in a trie. If a word is not found, close words are searched for on the trie with small edits (insertion, deletion, substitution); the edit distance rows are computed while the trie is walked.
- **IP routing.** Longest prefix match is done with a **binary trie** whose alphabet is *{0, 1}*. A packet's address is directed to the longest matching one among the prefixes in the routing table.
- **Word games and T9.** A program that solves Boggle or a word puzzle builds paths from letters. If the path being built is a prefix of no word, the search in that direction is cut off at once; this pruning is done with `startsWith`. The T9 system on old phones also uses a trie-like structure to suggest words from a sequence of keys.

11.8.2 The binary trie and longest-prefix match

Internet routers decide where to send a packet by looking at the packet's destination address. The routing table stores not individual addresses but address prefixes. A route of the form “0*” covers all addresses that begin with 0, and “011*” those that begin with 011. An address may match more than one prefix; the rule is to choose the **longest prefix match**. The longer prefix is a more specific route and takes priority over the general route.

Let the routes be $0^* \to A$, $01^* \to B$, $011^* \to C$, $1^* \to D$ and $110^* \to E$. Each route is stored in the binary trie at the node that represents the prefix. The node class carries an array with two children and a route field; adding a route is the same as ordinary insertion:

```
class BitNode {
  BitNode[] child = new BitNode[2];  // '0' and '1'
  String route;                      // null: no route
}

void addRoute(String prefix, String route) {
  BitNode node = root;
  for (char b : prefix.toCharArray()) {
    int i = b - '0';
    if (node.child[i] == null) node.child[i] = new BitNode();
    node = node.child[i];
  }
  node.route = route;
}
```

The longest match for an address is found as follows. We start at the root and follow the bits of the address in order. At every node along the path that carries a route, the route is remembered. When the path breaks or the address ends, the last remembered route is the answer. The last route seen is the deepest among those seen on the path, that is, the longest prefix.

```
String longestMatch(String addr) {
  BitNode node = root;
  String best = null;
  for (int k = 0; k < addr.length(); k++) {
    if (node.route != null) best = node.route;  // longer prefix
    node = node.child[addr.charAt(k) - '0'];
    if (node == null) return best;
  }
  return node.route != null ? node.route : best;

}
```

In each iteration of the loop, the current node's route is checked first; if there is one, `best` is updated. Then we descend to the child according to the k-th bit of the address: `addr.charAt(k) - '0'` turns the character '0' into 0 and '1' into 1. If there is no child, the best route remembered is returned. If the loop processes all the bits, the last node's route is also checked: if it has one, that is returned, otherwise `best`.

Address	Nodes visited (route)	Reason for stopping	Result
0110	root, 0 (A), 01 (B), 011 (C)	`k = 3`: no '0' branch	C
1001	root, 1 (D)	`k = 1`: no '0' branch	D

Address	Nodes visited (route)	Reason for stopping	Result
1101	root, 1 (D), 11 (—), 110 (E)	`k = 3`: no '1' branch	E

Table 11.17: Trace of `longestMatch`; routes 0* → A, 01* → B, 011* → C, 1* → D, 110* → E.

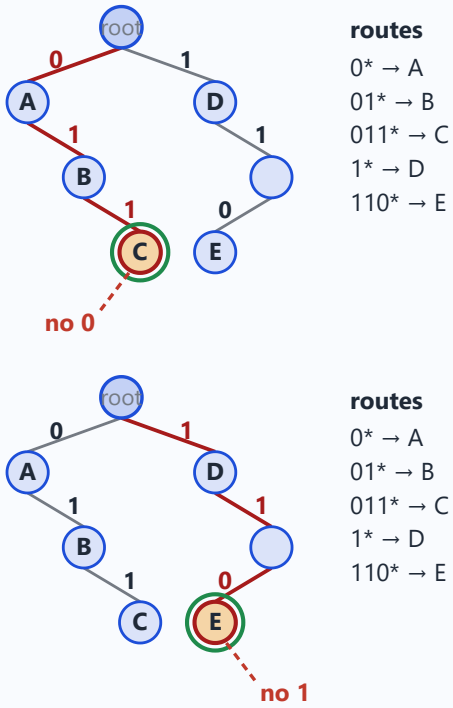


Figure 11.27: Left, the address 0110: we descend 0 → 01 → 011, there is no '0' branch; the last route seen is C. Right, the address 1101: "11" is an intermediate node with no route; the E at "110" is chosen.

For the address 0110, `best` becomes A, B and C in turn. Each new route is a longer prefix than the previous one and takes its place. For the address 1001 only D is seen; since there is no more specific route, the general route is used. For the address 1101 the "11" node is only on the path and does not change `best`. The number of steps is bounded by the length of the address: for a 32-bit IPv4 address at most 32 steps are taken, however many routes the routing table holds.

Real routers speed this idea up with compressed binary tries (Patricia) and hardware support, but the basic principle is the same. Had the prefix table been kept in a hash table, a separate lookup would have been needed for every possible prefix length.

11.8.3 The maximum XOR pair

A less well-known use of the binary trie concerns bit operations. The question is: in an array of numbers, which pair makes the value $a \oplus b$ largest? $\oplus$ is the bitwise exclusive or (XOR): if two bits differ the result is 1, if they are the same it is 0. Let the array be [3, 10, 5, 25, 2, 8]. Brute force tries all pairs: $n(n-1)/2$ pairs for n numbers.

The trie method inserts the numbers, starting from the most significant bit, into a binary trie of fixed length (here 5 bits). Then for each x we descend from the root and at every bit choose the branch that is the *opposite* of that bit of x; if there is no such branch, the same bit is chosen of necessity. At every level where the opposite bit is chosen, that bit of the result is 1.

```
int maxXor(int[] a, int B) {
  BitNode root = new BitNode();
  for (int x : a) {                       // insert with B bits
    BitNode node = root;
    for (int b = B - 1; b >= 0; b--) {
      int bit = (x >> b) & 1;
      if (node.child[bit] == null) node.child[bit] = new BitNode();
      node = node.child[bit];
    }
  }
  int best = 0;
  for (int x : a) {                       // seek opposite bits
    BitNode node = root;
    int y = 0;
    for (int b = B - 1; b >= 0; b--) {
      int bit = (x >> b) & 1;
      if (node.child[1 - bit] != null) { node = node.child[1 - bit]; y |= (1 - bit) << b; }
      else { node = node.child[bit]; y |= bit << b; }
    }
    best = Math.max(best, x ^ y);
  }
  return best;
}
```

The correctness of the greedy choice rests on a property of binary numbers. In a number, a 1 in a high bit is worth more than all the lower bits together: $2^b > 2^{b-1} + \ldots + 2 + 1 = 2^b - 1$. So getting a 1 in a high bit pays off whatever is lost in the lower bits. At every level, choosing the opposite branch when possible is the right decision.

number	binary (5 bits)
3	**00011**
10	**01010**
5	**00101**
25	**11001**
2	**00010**
8	**01000**
$5 \oplus 25$	**11100 = 28**

Figure 11.28: The 5-bit representation of six numbers. The XOR of 5 and 25 is 11100 = 28, the largest value in the array.

Example 11.8

Question. For $x = 5 = 00101$, which path is followed in the trie and which number is reached?

Solution. The bits wanted are the opposite of 5: 11010. At bit 4 a 1 is wanted; only 25 (11001) has this branch and we descend. At bit 3 a 1 is wanted; the bit of 25 is 1, so it exists. At bit 2 a 0 is wanted; the bit of 25 is 0, it exists. At bit 1 a 1 is wanted; only 25 is left and its bit is 0, so we are forced to descend to 0. At bit 0 a 0 is wanted; the bit of 25 is 1, so we are forced to descend to 1. The number reached is 11001 = 25, and $5 \oplus 25 = 00101 \oplus 11001 = 11100 = 28$. Doing the same for all the numbers, the largest value found is 28.

Method	Time	Amount of work
All pairs	$O(n^2)$	$n(n-1)/2 \approx 5 \cdot 10^9$ operations
Binary trie	$O(n \cdot B)$	insertion and query, $2 \cdot n \cdot B \approx 6 \cdot 10^6$ steps

Table 11.18: Cost of the two methods for maximum XOR; $n = 10^5$, $B = 30$ bits.

The trie method walks B levels twice for each number: once when inserting, once when querying. The difference is about a thousandfold. Here the trie's "length" parameter is the number of bits of the number, B.

11.8.4 The ternary search trie

One solution to the memory problem of the array representation is the **ternary search trie** (TST). In a TST each node carries a character and three links: `left` (smaller), `mid` (equal, move on to the next character) and `right` (larger). The left and right chains of a node form a small binary search tree for the sibling children of an ordinary trie. The middle link corresponds to the trie's step "go down one level". The end of a word is marked at the node.

```
class TstNode {
  char c;                    // the node's character
  boolean end;               // does a word end here?
  TstNode left, mid, right;
  TstNode(char c) { this.c = c; }
}

TstNode insert(TstNode node, String s, int i) {
  char ch = s.charAt(i);
  if (node == null) node = new TstNode(ch);
  if (ch < node.c) node.left = insert(node.left, s, i);
  else if (ch > node.c) node.right = insert(node.right, s, i);
  else if (i < s.length() - 1) node.mid = insert(node.mid, s, i + 1);
  else node.end = true;
  return node;
}
```

Insertion is recursive. If there is no node, a new node is created with the next character. If the character is smaller than the node's, we go left; if larger, right; in these two cases `i` does not change, because a sibling for the same character is being sought. If the characters are equal and the word has not ended, we go to the middle and `i` increases by one. If they are equal and it is the last character, the node is marked.

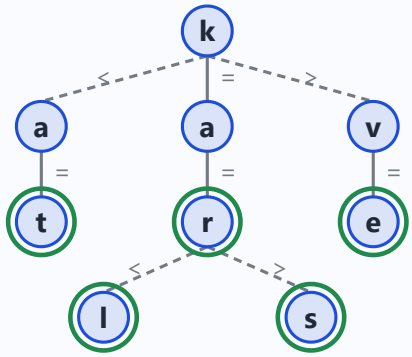


Figure 11.29: The ternary search trie of the words kar, kal, at, ve, kas. The solid line is the middle (=) link, the dashed lines are the left (<) and right (>) links.

The TST in the figure was built by inserting the words in the order kar, kal, at, ve, kas. The first word “kar” forms the k → a → r spine. “kal” follows this spine up to “ka”; since 'l' < 'r', it is attached to the left of 'r'. “at” goes to the left of the root because 'a' < 'k' at the root, and “ve” to the right because 'v' > 'k'. “kas” descends to 'r' and is attached to the right since 's' > 'r'. The insertion order changes the shape: the word inserted first becomes the spine.

Structure	Nodes	Links
Trie with 26-element arrays (including root)	10	*10 × 26 = 260*

Structure	Nodes	Links
TST	9	*9 × 3 = 27*

Table 11.19: Memory comparison of the two representations for kar, kal, at, ve, kas.

The array trie has the root and the nodes k, ka, kar, kal, kas, a, at, v, ve; 10 nodes allocate 260 references. The TST has no empty root; each character is a node, and 9 nodes carry only 27 links. The difference is about tenfold.

```
boolean search(String s) {
  TstNode node = root;
  int i = 0;
  while (node != null) {
    char ch = s.charAt(i);
    if (ch < node.c) node = node.left;
    else if (ch > node.c) node = node.right;
    else if (i == s.length() - 1) return node.end;
    else { node = node.mid; i++; }
  }
  return false;
}
```

Search is written iteratively. `i` is the index of the next character in the word. If the character is smaller than the node's, we go left; if larger, right, and `i` does not change. If equal there are two cases: if it is the last character the node's marker is returned, otherwise we descend to the middle and `i` increases. When a link is `null` the loop ends and `false` is returned.

Call	`node.c`	`ch , i`	Decision
`search("kas")`	k	k, 0	equal, not last → mid, `i = 1`
	a	a, 1	equal → mid, `i = 2`
	r	s, 2	's' > 'r' → right, `i` unchanged
	s	s, 2	equal and last index → `node.end` = `true`
`search("al")`	k	a, 0	'a' < 'k' → left
	a	a, 0	equal, not last → mid, `i = 1`
	t	l, 1	'l' < 't' → left; the link is `null` → `false`

Table 11.20: Trace of `search("kas")` and `search("al")` in the TST.

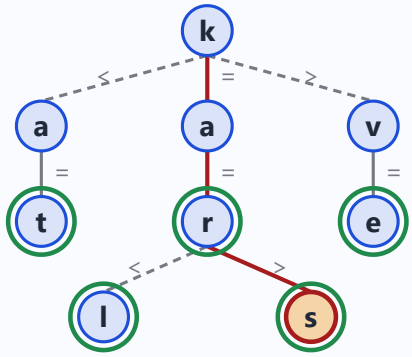


Figure 11.30: The last step of `search("kas")` : k → a → r → s; 4 nodes were visited for 3 characters.

In the search for “kas”, 4 nodes are visited for 3 characters: three are middle steps, and one is the right step taken to find 's' among the siblings of “r”. In general, a search in a TST is the sum of L middle steps and the binary search steps among siblings at each level. If the words are inserted in random order, the average cost is $O(L + \log n)$. If the words are inserted in sorted order, the sibling trees degrade: in a TST built by inserting at, kal, kar, kas, ve in that order, 'k' is placed to the right of 'a', 'v' to the right of 'k', and the siblings 'l', 'r', 's' form a chain. This is the same problem as inserting sorted keys into a binary search tree. In their 1997 work, Bentley and Sedgewick studied the TST in detail as a structure that keeps a balance between memory and speed for large alphabets (for example Unicode).

11.8.5 The compressed trie

In a standard trie most nodes have a single child. The part of the word “karton” after “kar” is a chain of the nodes “t”, “o”, “n”, each with a single child. These chains contain no decisions; they only carry characters in order. A **compressed trie** (radix tree) merges chains of single-child, unmarked nodes into a single edge. An edge now carries not a single character but a substring.

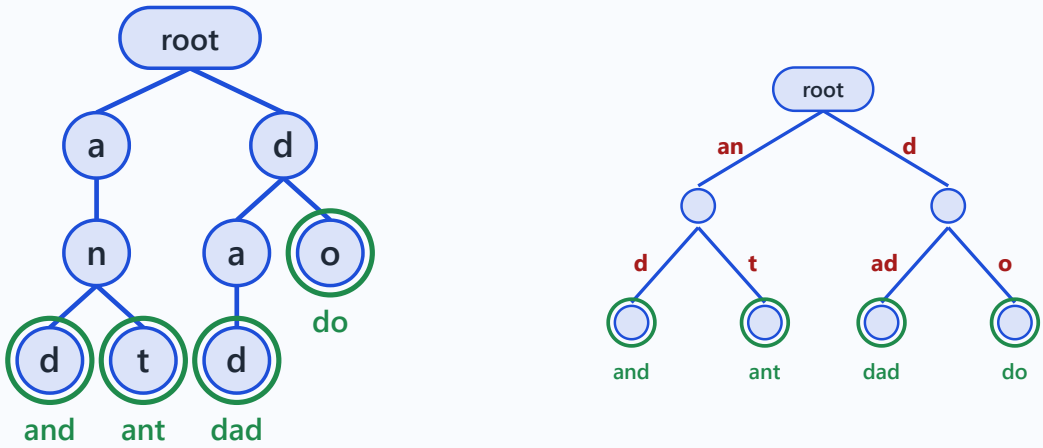


Figure 11.31: The words and, ant, dad, do. Left, the standard trie: 8 nodes not counting the root. Right, the

compressed trie: 6 nodes; the edges carry substrings such as “an” and “ad”.

In the figure the “a” node has a single child and is unmarked; “a” and “an” merge into a single “an” edge. The “an” node stays because it has two children. The “da” node is also single-child and unmarked; the path from “d” to “dad” becomes the edge “ad”. The “d” node stays because it has two children. In the end, apart from the root, the nodes an, and, ant, d, dad and do remain.

Theorem 11.1 (Size of a compressed trie)

A compressed trie of n words has at most $2n - 1$ nodes, not counting the root.

Proof. After compression, every node other than the root is either marked or has at least two children; single-child unmarked nodes have been merged. The number of marked nodes equals the number of words: n. Every leaf is marked; so the number of leaves is at most n. In any rooted tree, the number of nodes with at least two children is smaller than the number of leaves: each such node adds at least one to the leaf count. Hence the unmarked nodes with at least two children number at most $n - 1$. The total is at most $n + (n - 1) = 2n - 1$. □

The theorem says that the number of nodes is $O(n)$ regardless of the length of the words. In a standard trie, by contrast, the number of nodes is proportional to the total number of characters. In the example and, ant, dad, do, for $n = 4$ the bound is 7 and the actual value is 6. The edge labels still carry the characters in total; but these labels can be stored not as separate strings but as (start, length) pairs pointing into the original words. In this way memory too stays bounded by $O(n)$ nodes and the words themselves.

Example 11.9

Question. How many nodes, not counting the root, does the standard trie of the words car, cart, care, cat and dog have? How many remain in the compressed trie?

Solution. The nodes of the standard trie are c, ca, car, cart, care, cat, d, do, dog: 9 nodes. In compression “c” is single-child and unmarked; “c” and “ca” merge and a “ca” edge results. “ca” stays because it has two children (r, t). “car” is marked and stays. cart, care and cat are leaves. “d” and “do” are single-child and unmarked; the “dog” edge reduces to a single node. The remaining nodes are ca, car, cart, care, cat, dog: 6 not counting the root, 7 including it.

In 1968 Morrison published the binary-alphabet version of this idea under the name **Patricia**. The Patricia trie is used in IP routing tables and in file systems. The price of compression is that insertion and deletion become more complicated. If a new word diverges in the middle of an edge, the edge is split in two and a new node is placed between. In deletion, a node that is left with a single child is merged again with its parent edge.

11.8.6 The suffix trie

In the examples so far the trie stored a set of words. To search inside a text, a different set is chosen: all the suffixes of the text. The suffixes of the text T = “abab” are abab, bab, ab and b. If these four suffixes are inserted into a trie, we get a **suffix trie**.

The power of the suffix trie comes from a chain of equivalences:

> *P is a substring of T ⇔ P is a prefix of some suffix of T ⇔ P is a path from the root in the suffix trie.*

The first equivalence comes from the definition of a substring: if P begins at position i in T, it is a prefix of the suffix that begins at position i. The second holds because every prefix in a trie is a path. As a result, the question “does P occur in T?” is answered by `startsWith(P)`. The time is $O(m)$, where $m = |P|$, independent of the length of the text. The text is processed once, and then any number of patterns can be searched quickly.

```
TrieNode suffixTrie(String t) {
  TrieNode root = new TrieNode();
  for (int s = 0; s < t.length(); s++) {
    TrieNode node = root;
    for (int j = s; j < t.length(); j++) {
      int i = t.charAt(j) - 'a';
      if (node.child[i] == null)
        node.child[i] = new TrieNode();
      node = node.child[i];
    }
    node.isWord = true;  // suffix ends
  }
  return root;
}
```

In the outer loop the code walks over the start `s` of each suffix. The inner loop, starting from `s`, inserts the characters of the suffix into the trie like an ordinary insertion. When the suffix ends, the last node is marked. The marker is not needed for the substring query, but can be used to ask whether a string is a *suffix*.

`s`	Suffix	Nodes created	Marked
0	abab	a, ab, aba, abab (4)	abab
1	bab	b, ba, bab (3)	bab
2	ab	none; the path already exists	ab
3	b	none; the path already exists	b

Table 11.21: Trace of `suffixTrie("abab")`.

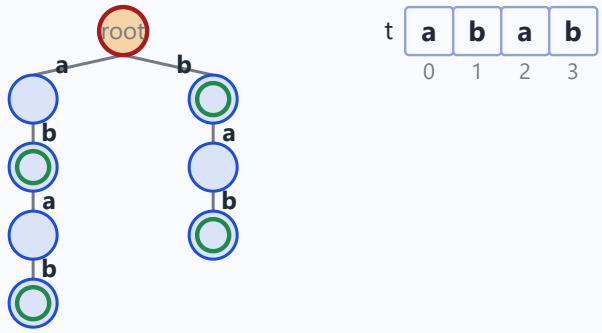


Figure 11.32: The suffix trie of the text “abab”: 8 nodes including the root. The suffixes abab, bab, ab and b are marked.

The first two suffixes open new branches; the last two stay on the paths of earlier suffixes and only get a marker. In all, 7 nodes were created; with the root there are 8. Queries are answered as follows: “ba” is a path from the root (b → a), so it is a substring of “abab”. For “bb” there is no 'b' branch after “b”; “bb” is not a substring. “bab” is found in three steps, “bb” is rejected in two.

The cost of the suffix trie is its size. A text of length n has at most $n(n + 1)/2$ distinct non-empty substrings, and every node of the suffix trie is a different substring. In the text “abcd”, whose characters are all different, this bound is reached: 10 nodes. Of the 21 possible substrings of the text “banana”, 15 are distinct, and the suffix trie has 15 nodes besides the root. In general the memory is $O(n^2)$; for a text of a million characters this is unacceptable.

The remedy is again compression. A compressed suffix trie is called a **suffix tree**. Since a suffix tree has n suffixes, by the compressed trie theorem it has $O(n)$ nodes. The edge labels are stored as (start, end) position pairs in the text. The algorithm published by Ukkonen in 1995 builds a suffix tree in $O(n)$ time. An option that needs less memory is the **suffix array**: an array holding the starting indices of the suffixes in lexicographic order. Substring search in a suffix array takes $O(m \log n)$ with binary search.

11.8.7 Aho–Corasick

An antivirus scanner looks for thousands of signatures in a file at the same time. Scanning the text separately for each signature means as many scans as there are patterns. The **Aho–Corasick algorithm** puts all the patterns into a trie and reads the text only once, from left to right. During reading it moves from one node of the trie to another. This approach is the multi-pattern generalization of the single-pattern KMP algorithm.

Let the patterns be he, she, his and hers. The trie has the paths h → he → her → hers, h → hi → his and s → sh → she. Besides the trie, a **failure link** is computed for each node.

Definition 11.2

In a pattern trie, the **failure link** of a node u is the link to the node of the longest *proper* suffix (a suffix shorter than the string itself) of the string that u represents, among those that have a node in the trie. If there is no such suffix, the link goes to the root.

In the example, the suffixes of “she” are “he” and “e”; “he” is in the trie, so the failure link goes to “he”. The suffix “h” of “sh” is in the trie. The suffixes of “hers” are “ers”, “rs” and “s”; only “s” is in the trie. The failure link of “his” goes to “s” as well. The links of the remaining nodes (h, s, he, hi, her) go to the root.

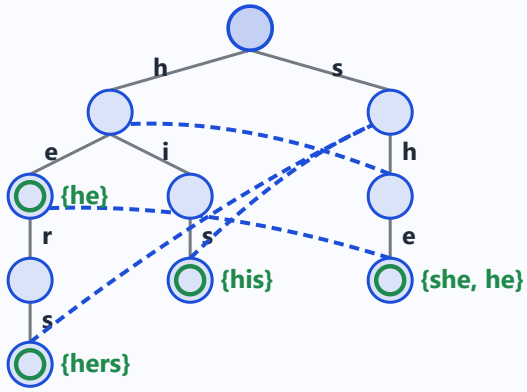


Figure 11.33: The trie of the patterns he, she, his, hers. The dashed lines are failure links; the braces show a node's output set.

While the text is read, if the current node has a branch for the character read, we move to that branch. If there is none, we fall back along the failure link and try the same character at the new node; this corresponds to falling back with the prefix table in KMP. If the root has no branch either, we stay at the root. At each node the node's **output set** is reported. The output set is the union of the node's own pattern and the patterns reached along the failure chain. The output of the “she” node is {she, he}, because its failure link goes to “he” and “he” is a pattern.

Index	Character	State (node)	Found
0	u	root (no 'u' branch)	—
1	s	s	—
2	h	sh	—
3	e	she	she, he
4	r	she → (failure) he → her	—
5	s	hers	hers

Table 11.22: Scanning the text “ushers” with Aho–Corasick.

At the character at index 4 (r), the “she” node has no 'r' branch. We fall back by the failure link to “he”; “he” has an 'r' branch and we move to the “her” node. This fallback keeps the last two characters read with “she” (h, e) from going to waste: “he” already represents a matched prefix. The text is read in a single pass and three matches are found: she and he ending at index 3, hers ending at index 5.

The failure links are computed on the trie by breadth-first traversal: the link of a node is found by starting from its parent's link and looking for a branch for the same character. This preprocessing is proportional to m, the sum of the pattern lengths. The scan takes $O(n)$ transitions for a text of length n, plus the cost of reporting z matches. The total time is $O(n + m + z)$ (Aho and Corasick, 1975). The algorithm is used in virus signature scanning, in network intrusion detection systems and in banned-word filters. The first versions of the Unix `fgrep` command were also based on this algorithm.

11.8.8 The trie family

The variants of the trie are different forms of a common idea. Table 11.23 summarizes which question is answered best by which structure.

Structure	What it stores	Question it is strong at	Query
Standard trie	a set of words	exact word, prefix, autocomplete	$O(L)$
Trie with prefix counts	+ the number of words passing through a node	“how many words have this prefix?”	$O(P)$
Compressed trie	single-child chains merged	keys with long common prefixes, memory	$O(L)$
Ternary search trie	3 links per node	large alphabet, little memory	$O(L + \log n)$ avg.
Binary trie	bit strings (IP, integers)	longest-prefix match, maximum XOR	$O(B)$
Suffix trie / array	all suffixes (array: $O(n)$ memory)	“is P a substring of T?”	$O(m)$ / $O(m \log n)$
Aho–Corasick	pattern trie + failure links	finding many patterns in a text at once	$O(n + z)$

Table 11.23: The trie family: which structure for which question?

The common idea is always the same: the key is read character by character (or bit by bit) and common prefixes are shared. The differences lie in what is stored in a node and how the children are reached. In choosing a structure, the kind of problem comes first: exact match, prefix, substring, or many patterns? Then the size of the alphabet and the memory budget are considered.

11.9 Common mistakes

Trie code is short, but a few subtle points are often overlooked. Most of the mistakes below do not show up in small tests and appear only with special inputs.

Common mistake

Forgetting `isWord`. Looking only at the existence of the path in a search counts every prefix as a word. In the trie of and, ant, dad, do, "da" would seem to be found. Without the end-of-word marker, a trie stores not a set of words but all the prefixes of the set.

Common mistake

Confusing `search` with `startsWith`. A prefix query does not look at the last node's marker; an exact word search must. The two methods differ only in the last line, and writing that line wrongly does not cause a compile error.

Common mistake

Characters outside the alphabet. `'A' - 'a'` = −32 and `'ş' - 'a'` = 254; the array bound is exceeded. The input must be converted to lowercase and to the a–z alphabet, or the children must be kept in a `HashMap`. Care is also needed when lowercasing Turkish text: in Java, `"I".toLowerCase()` gives "i" in the English locale and "ı" in the Turkish locale; a call without a parameter depends on the machine's locale.

Common mistake

Deleting a shared node during deletion. Deleting the path from start to end destroys the other words that share the prefix. Only childless, unmarked nodes are deleted, from the bottom up.

Common mistake

Expressing the complexity in terms of *n*. Trie operations are $O(L)$, not $O(\log n)$ or $O(n)$: they are independent of the number of words. On the other hand, counting a search in a hash table as "$O(1)$, faster than a trie" is also wrong; computing the hash value reads L characters as well.

Common mistake

Underestimating memory. Each node carries 26 references; with millions of nodes this comes to hundreds of MB. For a sparse trie a `HashMap`, a ternary search trie or compression is needed.

Common mistake

Not undoing shared state when backtracking. If recursive walks such as `collect` use a shared `StringBuilder`, the appended character must be removed after each call. Otherwise the words of the following siblings come out corrupted.

11.10 Summary

- **Structure:** a trie stores a set of strings in a rooted tree. The root is the empty string, each edge carries a character, each node is a prefix. A word is a path from the root whose last node is marked. Common prefixes are stored once; the shape of a trie does not depend on the insertion order.
- **Node:** `TrieNode[26] child` and `boolean isWord`. The character is not stored in the node; the index is found with `ch - 'a'`. Characters outside the alphabet exceed the array bound.
- **Insertion:** if the child does not exist it is created, if it does we move on; at the end the last node is marked. $O(L)$; at most L new nodes.
- **Search and prefix query:** `false` if a branch is missing; search returns `isWord` at the end, the prefix query returns `true`. $O(L)$ and $O(P)$. A `count` counter gives the number of words that start with a prefix in $O(P)$.
- **Wildcard and spell checking:** a wildcard character tries every filled branch. The edit distance rows are computed on the trie from parent to child; the row of a common prefix is computed once, and branches whose row minimum exceeds the bound are pruned.
- **Autocomplete:** the prefix node is found and the subtree is walked depth first; the result is in alphabetical order. $O(P + S \cdot M)$. The longest common prefix is found by descending from the root along single-child, unmarked nodes.
- **Deletion:** the marker is removed; childless, unmarked nodes are deleted from the bottom up. Three cases: if the word is a prefix only the marker is removed, if it has a tail the tail is deleted, if the word is absent nothing changes. $O(L \cdot M)$, $O(L)$ with a child counter.
- **Memory:** $O(N \cdot M)$; in the trie of and, ant, dad, do, 226 of the 234 references are `null`. For sparse nodes a `HashMap`, a `TreeMap`, a ternary search trie or a compressed trie is used.
- **Comparison:** for exact search the trie and the hash table are $O(L)$; the trie's difference is in prefix and order queries. A preorder traversal yields the words in lexicographic order.
- **Variants:** the binary trie (longest-prefix match, maximum XOR), the ternary search trie (3 links per node), the compressed trie and Patricia (at most $2n - 1$ nodes for n words), the suffix trie → suffix tree and suffix array, Aho–Corasick (multi-pattern search with failure links, $O(n + m + z)$).

11.11 Key terms

trie (prefix tree) prefix suffix substring end of word alphabet reference (child pointer) prefix query wildcard search edit distance autocomplete backtracking depth-first search (DFS) longest common prefix lexicographic order hash table radix sort binary trie longest prefix match ternary search trie (TST) compressed trie (radix tree) suffix trie / suffix tree suffix array Aho–Corasick algorithm failure link

11.12 Test Yourself

Questions 1–7 use the same trie: the words sel, selam, sen, ses and su have been inserted into a trie with 26-element arrays.

1. How many nodes does the trie have, counting the root? How many child references are `null` in total?

   Answer

   The nodes are s, se, sel, sela, selam, sen, ses and su: 8 nodes, 9 with the root. The number of references is $9 \times 26 = 234$. The number of filled references equals the number of edges (8); $234 - 8 = 226$ references are `null`.

2. What do `search("se")`, `startsWith("se")`, `search("sela")` and `startsWith("sela")` return?

   Answer

   `false`, `true`, `false`, `true`, in that order. The paths of “se” and “sela” exist, but both nodes are unmarked; both are only prefixes.

3. Which list does `complete("se")` return? Why is the list in this order?

   Answer

   `[sel, selam, sen, ses]`. Since “se” is unmarked it is not added. The children are walked in index order: l (11), n (13), s (18). The “sel” node, being itself marked, is added before “selam”.

4. Suppose the words were inserted with the prefix-counting `insert`. What is the value of `count` at

the root, at the “se” node and at the “sel” node?

Answer

5 at the root, 4 at “se” (all except su), 2 at “sel” (sel and selam). `prefixCount("se")` returns 4.

5. If each deletion is applied separately to the initial trie, how many nodes do `delete("sel")`, `delete("selam")` and `delete("su")` remove?

Answer

`delete("sel")`: 0; “sel” has an 'a' child, only the marker is removed. `delete("selam")`: 2; “selam” and “sela” are deleted, and we stop at the marked “sel” node. `delete("su")`: 1; “su” is deleted, and we stop because the “s” node has the child 'e'.

6. What does `longestCommonPrefix()` return? What would the result be if the word “su” were not in the trie?

Answer

“s”. The root's only child is 's', but the “s” node has two children (e, u) and the descent stops. Without “su”, the only child of “s” would be 'e'; the “se” node has three children (l, n, s) and the descent stops there: “se”.

7. For the query “sez” and $k = 1$, which words does the edit distance search on the trie suggest? At which node is pruning done?

Answer

The rows (with columns “”, s, e, z) are: s → 1 0 1 2, se → 2 1 0 1, sel → 3 2 1 1, sen and ses → 3 2 1 1, su → 2 1 1 2, sela → 4 3 2 2. The words whose last cell is 1 are sel, sen and ses; all three are suggested. The last cell of “su” is 2, so it is not suggested. The minimum of the “sela” row is 2, and it is pruned; “selam” is not entered.

8. How many nodes, not counting the root, does the compressed trie of the same five words have? What is the upper bound from the theorem?

Answer

s (two children), se (three children), sel, selam (with the edge “am”), sen, ses and su: 7 nodes. The “sela” node of the standard trie has been merged into the “am” edge. The bound is $2 \cdot 5 - 1 = 9$.

9. In the array [2, 9, 12, 5], which pair has the largest XOR value? In a 4-bit binary trie, which number is reached for 2?

   Answer

   The representations are 0010, 1001, 1100, 0101. For 2 the wanted bits are 1101: a 1 at bit 3 (9 and 12 have it), a 1 at bit 2 (only 12), a 0 at bit 1 (the bit of 12 is 0), a 1 is wanted at bit 0 but the bit of 12 is 0. The number reached is 12 and $2 \oplus 12 = 1110 = 14$. The largest value is 14 (2 and 12).

10. How many nodes, not counting the root, does the suffix trie of the text “aab” have? Are “ba” and “ab” substrings?

    Answer

    The suffixes are aab, ab, b. The nodes are a, aa, aab, ab, b: 5 nodes; this number is the number of distinct substrings of the text (“a” occurs twice and is counted once). “ab” is the path a → b from the root, so it is a substring. For “ba”, “b” has no children at all; it is not a substring.

11.13 Further reading

- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 5.2: gives Java code, deletion and memory measurements for R-way tries and ternary search tries in detail.
- D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (1997): covers suffix trees, Ukkonen's algorithm and Aho–Corasick together with applications to biological sequences.
- J. L. Bentley and R. Sedgewick, “Fast Algorithms for Sorting and Searching Strings”, *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1997: analyzes the ternary search trie and the three-way string quicksort based on it.
- A. V. Aho and M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search”, *Communications of the ACM* 18(6), 1975: the original paper on the multi-pattern matching algorithm with failure links.

11.14 References

1. A. V. Aho, M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search”, *Communications of the ACM* 18(6), 1975.
2. J. L. Bentley, R. Sedgewick, “Fast Algorithms for Sorting and Searching Strings”, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
4. E. Fredkin, “Trie Memory”, *Communications of the ACM* 3(9), 1960.

5. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
6. U. Manber, G. Myers, "Suffix Arrays: A New Method for On-Line String Searches", *SIAM Journal on Computing* 22(5), 1993.
7. D. R. Morrison, "PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric", *Journal of the ACM* 15(4), 1968.
8. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 5.2: Tries).
9. E. Ukkonen, "On-line Construction of Suffix Trees", *Algorithmica* 14(3), 1995.

APPENDIX A

Glossary of Terms

The terms used in the book, each with a short definition. The number next to a term is the chapter where it is explained in detail. A searchable version of the glossary is on the course page.

A

A* search 6
A search algorithm that expands nodes in order of f = g + h and is guided by a heuristic to find a shortest path.

adaptive (sort) 3
Describes a sorting algorithm that runs faster when the input is already partially sorted.

adjacency list 5
A graph representation that stores a list of neighbors for each vertex, using O(V + E) memory.

adjacency matrix 5
A graph representation that records the presence (or weight) of each edge in a V × V matrix.

admissible heuristic 6
A heuristic that never overestimates the true remaining cost to the goal.

Aho–Corasick algorithm 11
An algorithm that finds many patterns in a text in a single pass, using a trie of the patterns and failure links.

algorithm 1
A method that solves a problem in a finite number of clearly defined steps.

algorithm analysis 1
The study of an algorithm's running time and memory requirements as a function of the input size.

all-pairs shortest paths 6
The shortest paths between every pair of vertices (Floyd-Warshall).

alphabet 9
The set of characters that can appear in strings (Σ).

amortized analysis 2
Finding the cost per operation by dividing the total cost of a sequence of operations by the number of operations.

ancestor 5
Any node on the path from a given node up to the root of a tree: its parent, its parent's parent, and so on.

articulation point 5
A vertex whose removal disconnects a connected undirected graph.

asymptotic notation 2
The notations O, Ω and Θ, which describe growth for large n by discarding constant factors and lower-order terms.

augmenting path 8
A path from the source to the sink in the residual network along which the flow can be increased.

autocomplete 11
Suggesting likely words from the prefix typed so far, a typical use of a trie.

auxiliary space 2
The memory an algorithm needs in addition to the memory that holds its input; space complexity is usually stated for this.

average case 2
The expected number of operations over a given distribution of inputs.

B

back edge 5
In a DFS, an edge to an ancestor that is still on the stack; it signals a cycle.

bad character rule 9
A shifting rule that aligns the mismatched text character with its last occurrence in the pattern.

base case 2
The smallest case that a recursion solves directly, without calling itself.

Bellman-Ford algorithm 6
An algorithm that relaxes all edges V − 1 times and allows negative edge weights.

best case 2
The input of a given size that requires the fewest operations.

bidirectional search 6
A search that runs BFS from the source and from the target at the same time and stops when the two waves meet.

Big O notation 2
f(n) = O(g(n)): for large enough n, f is bounded above by a constant multiple of g.

Big Omega (Ω) 2
f(n) = Ω(g(n)): for large enough n, f is bounded below by a constant multiple of g.

Big Theta (Θ) 2
f(n) = Θ(g(n)): f is both O(g(n)) and Ω(g(n)), so the two grow at the same rate.

binary search 4
A search on a sorted array that compares with the middle element and halves the range at each step: O(log n).

binary trie 11
A trie over the alphabet {0, 1} that stores keys bit by bit.

bipartite graph 5
A graph whose vertices can be split into two groups so that every edge joins the two groups.

bipartite matching 8
A set of edges in a bipartite graph with no common endpoints; it can be found with maximum flow.

blocking flow 8
A flow that blocks every path in the level graph by saturating at least one of its edges.

Borůvka's algorithm 7
An MST algorithm that adds the lightest outgoing edge of every component in each round.

bottleneck 8
The smallest residual capacity on an augmenting path, which is the amount of flow added in that round.

Boyer-Moore algorithm 9
A search that compares the pattern from right to left and makes large shifts.

breadth-first search (BFS) 5
A traversal that visits vertices layer by layer in order of distance from the start, using a queue.

bridge 7
An edge whose removal splits the graph into more components.

brute force (naive) 9
An approach that tries every possibility one by one.

bubble sort 3
A sort that compares neighboring items and swaps them when out of order; each pass moves the largest item to the end.

bucket 4
Each cell of the array in a hash table.

bucket sort 3
A sort that distributes items into buckets by value range and then sorts each bucket.

Burrows–Wheeler transform (BWT) 10
A reversible transform that takes the last column of the sorted rotations of a string and makes compression easier.

C

call stack 2
The stack of frames that holds the parameters, local variables and return address of each method call.

capacity 8
The largest amount of flow that can pass through an edge.

catastrophic backtracking 10
When a backtracking regular-expression engine tries exponentially many alternatives because of nested quantifiers and appears to hang.

collision 4
Two different keys hashing to the same value.

comparison sort lower bound 3
Every comparison-based sort needs $\Omega(n \log n)$ comparisons in the worst case.

complete graph 5
A graph in which every pair of vertices is joined by an edge; with n vertices it has $n(n-1)/2$ edges.

complexity class 2
A group of algorithms with the same growth rate (constant, logarithmic, linear, and so on).

complexity theory 2
The field that studies the resources an algorithm uses, such as time and memory, as a function of the input size.

compressed trie (radix tree) 11
A trie that merges chains of single-child nodes into a single edge.

compression 9
Representing data with fewer bits.

compression ratio 9
The compressed size divided by the original size; the smaller it is, the better the compression.

connected component 7
A maximal set of vertices that are joined to one another by paths.

consistent heuristic 6
A heuristic that satisfies $h(u) \le w(u, v) + h(v)$ for every edge (u, v) and $h(\text{goal}) = 0$.

constant time 1
A running time that does not depend on the input size: $O(1)$.

correctness 1
The property that an algorithm produces the correct output for every valid input.

counting sort 3
A non-comparison sort that counts small integer keys and places them using prefix sums.

cubic 1
$O(n^3)$ growth; for example, straightforward matrix multiplication.

cut (s-t cut) 8
A split of the vertices into an s side and a t side; its capacity is the sum of the capacities of the edges going from the s side to the t side.

cut property 7
The lightest edge crossing a cut belongs to some MST.

cycle 5
A closed path that returns to its starting vertex without repeating an edge.

cycle property 7
The heaviest edge of a cycle (if it is unique) belongs to no MST.

D

decision tree 3
A tree that models a comparison-based algorithm: internal nodes are comparisons and leaves are the resulting orderings.

decrease and conquer 4
An approach that solves a problem by reducing it to a single smaller subproblem (for example, binary search).

definiteness 1
The property that every step is defined precisely and unambiguously.

degree 5
The number of edges attached to a vertex; in a directed graph, in-degree and out-degree are counted separately.

depth-first search (DFS) 5
A traversal that goes as deep as possible along one branch and then backtracks (using a stack or recursion).

deterministic algorithm 1
An algorithm that follows the same steps and gives the same output every time it runs on the same input.

deterministic finite automaton (DFA) 10
A finite state machine with exactly one transition for every (state, symbol) pair.

Dijkstra's algorithm 6
A shortest-path algorithm for non-negative weights that repeatedly picks the vertex with the smallest distance and relaxes its edges.

Dinic's algorithm 8
A maximum-flow algorithm that finds blocking flows in level graphs: $O(V^2 \cdot E)$.

directed acyclic graph (DAG) 5
A directed graph with no cycles; it plays a special role in task scheduling.

directed graph 5
A graph whose edges have a direction.

distance 6
The length of a shortest path to a vertex; for strings, a measure of the difference between two strings.

dynamic array 2
An array that grows by copying its contents into a larger array when it becomes full (for example, Java's ArrayList).

E

edge 5
A connection between two vertices; it may be directed and may carry a weight.

edit distance 10
The minimum number of insertions, deletions and substitutions needed to turn one string into another.

Edmonds-Karp algorithm 8
Ford-Fulkerson with augmenting paths chosen by BFS (fewest edges): $O(V \cdot E^2)$.

effectiveness 1
The property that every step is basic enough to be carried out in finite time with pencil and paper.

end-of-word marker 11
A mark on a trie node showing that the path from the root to that node spells a word.

Euclid's algorithm 1
An algorithm that finds the greatest common divisor of two numbers by repeated division with remainder.

excess flow 8
The amount by which the flow entering a vertex exceeds the flow leaving it.

exponential 1
Growth such as $O(2^n)$, where the running time doubles each time n increases by one.

exponential search 4
A method that grows the range as 1, 2, 4, … and then applies binary search.

F

factorial 1
O(n!) growth, such as trying all permutations.

failure link 11
In Aho–Corasick, a link from a node to the node for the longest proper suffix of its string that is present in the trie.

finite state machine (FSM) 10
A model of computation made up of a finite number of states and transitions that depend on the input.

finiteness 1
The property that an algorithm terminates after a finite number of steps for every input.

flow conservation 8
At every vertex other than the source and the sink, the incoming flow equals the outgoing flow.

flow network 8
A directed graph whose edges have capacities and which carries flow from a source to a sink.

Floyd-Warshall algorithm 6
A dynamic-programming algorithm that finds all-pairs shortest paths by adding intermediate vertices one at a time: $O(V^3)$.

Ford-Fulkerson method 8
A general method that keeps increasing the flow until no augmenting path remains.

G

good suffix rule 9
A shifting rule based on another occurrence in the pattern of the suffix that has just matched.

graph 5
A structure made of vertices and the edges that connect them: G = (V, E).

graph traversal 5
Systematically visiting all reachable vertices of a graph.

greedy best-first search 6
A search that looks only at the estimated remaining cost to the goal; it rushes toward the goal but ignores the cost of the path so far.

growth rate 2
How fast a function grows as n grows; the basis for comparing algorithms.

H

hash function 4
A function that maps a key to a table index.

hash table 4
A table that stores keys at positions given by a hash function, with O(1) search on average.

heap 3
A complete binary tree in which every node is greater than its children (max-heap) or smaller than its children (min-heap).

heap sort 3
A sort that builds a max-heap from the array and then repeatedly moves the root to the end.

heuristic 6
A rule or function that gives a good result or estimate without any guarantee of exactness.

I

in-place 3
Using only O(1) (or very little) extra memory besides the input.

input 1
The values, drawn from a specified set, that an algorithm receives at the start.

input size 1
The value n that measures the size of a problem (array length, number of vertices, and so on).

insertion sort 3
A sort that inserts each item into its correct place in the sorted part.

interpolation search 4
A search that estimates the position of the target from the end values; O(log log n) on uniformly distributed data.

inversion 3
A pair of indices with i < j and a[i] > a[j].

iterated logarithm 2
log* n: the number of times $\log_2$ must be applied to n before the result is 1 or less.

iterative deepening DFS (IDDFS) 5
A series of DFS runs with the depth limit raised to 0, 1, 2, and so on; it combines DFS's small memory use with BFS's guarantee of finding the shallowest solution.

J

jump search 4
A method that jumps ahead in steps of √n in a sorted array and then searches linearly within the block.

K

Knuth-Morris-Pratt (KMP) 9
An algorithm that searches in O(n + m) without moving backward in the text, using the LPS table.

Kruskal's algorithm 7
An MST algorithm that scans edges in order of weight and adds those that do not form a cycle.

L

Lempel-Ziv-Welch (LZW) 9
A compression method that adds seen strings to a dictionary and encodes them as dictionary indices.

level graph 8
The subgraph made of the edges that go from one BFS level to the next.

Levenshtein distance 10
The edit distance in which each insertion, deletion and substitution costs 1.

lexicographic order 10
Comparing strings character by character in alphabetical order.

linear 1
Growing in proportion to n: O(n).

linear probing 4
Resolving a collision by searching for the next empty cell and placing the key there.

linear search 4
A search that examines the items one by one from the beginning: O(n).

linearithmic 2
O(n log n) growth; the class of efficient comparison-based sorts.

load factor 4
The ratio of the number of keys in the table to the number of cells (α = n/m).

logarithmic 4
Growing in proportion to log n; for example, binary search.

longest common prefix 11
The longest beginning shared by all strings in a set of strings.

longest prefix match 11
Choosing the longest of the prefixes that match an address; the basic rule of IP routing.

lookahead / lookbehind 10
A regular-expression construct that tests whether the text to the right or left matches a pattern without advancing the position.

loop invariant 3
A statement that is true at the start of every iteration of a loop and is used to show that an algorithm is correct.

lossless / lossy compression 9
With lossless compression the decompressed data is bit-for-bit identical to the original; with lossy compression perceptually unimportant detail is discarded and the result is only close to the original.

lower bound 2
A limit that a value cannot go below; Ω notation gives a lower bound.

LPS table 9
For each position, the length of the longest proper prefix that is also a suffix.

LSD / MSD radix sort 3
Radix sort that starts from the least significant digit (right) or the most significant digit (left).

M

master theorem 2
A rule that solves recurrences of the form T(n) = aT(n/b) + f(n) in three cases.

max-flow min-cut theorem 8
The value of a maximum flow equals the capacity of a minimum cut.

maximum flow 8
The largest total flow that can be sent from the source to the sink.

median-of-three 3
A rule that picks the median of the first, middle and last items as the pivot.

merge 3
The operation that combines two sorted arrays into one sorted array.

merge sort 3
A sort that splits the array in half, sorts each half and then merges the sorted halves.

minimum spanning forest 7
A minimum spanning tree for each component of a graph that is not connected.

minimum spanning tree (MST) 7
A spanning tree with the smallest possible total edge weight.

N

negative cycle 6
A cycle whose weights add up to a negative number; if one exists, shortest paths are undefined.

nondeterministic finite automaton (NFA) 10
A finite state machine that allows zero, one or several transitions for a (state, symbol) pair.

O

open addressing 4
A hashing method in which a colliding key is placed in another empty cell of the table.

output 1
The result an algorithm produces, which has a specified relationship to the input.

P

parent 5
The node directly above a node in a tree; BFS and DFS trees, Prim's algorithm and union-find store it in a parent array.

partition 3
Splitting an array into items smaller than the pivot and items larger than it (Lomuto, Hoare).

path 5
A sequence of vertices obtained by moving along one edge at a time from vertex to neighbor; its length is the number of edges it contains.

path compression 7
Linking the nodes visited during a find operation directly to the root.

pattern 9
The string being searched for in the text (P).

prefix 9
A part taken from the beginning of a string.

prefix doubling 10
Building a suffix array by doubling the length of the sorted prefixes in each round.

prefix function 9
In KMP, a table that depends only on the pattern and gives the shift for each possible match length; another name for the LPS table, also written π.

prefix query 11
Finding all keys that start with a given prefix.

preflow 8
A flow that respects the capacity constraints but allows excess flow at vertices other than the source instead of conservation; push-relabel works with it.

Prim's algorithm 7
An MST algorithm that grows a tree from one vertex by adding the lightest outgoing edge.

primary clustering 4
In linear probing, the tendency of occupied cells to form long consecutive blocks.

priority queue 6
A structure that quickly returns the item with the smallest (or largest) key, usually implemented with a heap.

probe 4
The operation of looking at an array cell and comparing it with the value being searched for.

proper prefix 9
A prefix that is not equal to the string itself; likewise, a proper suffix is a suffix not equal to the string itself.

pseudocode 5
A way of describing an algorithm step by step without depending on a programming language.

push-relabel 8
A method that finds a maximum flow by pushing excess flow at vertices according to their heights.

Q

quadratic 3
$O(n^2)$ growth; typically two nested loops.

quantifier 10
An element of a regular expression (*, +, ?, {m,n}) that says how many times an item may repeat.

quick sort 3
An algorithm that partitions around a pivot and sorts both sides recursively.

R

Rabin-Karp algorithm 9
A search algorithm that compares the hash values of windows.

radix sort 3
A method that sorts keys digit by digit using a stable sort (LSD, MSD).

random-access machine (RAM) 2
An abstract model of computation in which every basic operation and every memory access takes one unit of time.

recurrence (relation) 2
An equation that defines T(n) in terms of the values of T on smaller inputs.

recursion 2
A function calling itself to solve a smaller instance of the problem.

recursion tree 2
A tree that shows the recursive calls and the work done at each level; used to solve recurrences.

reference (child pointer) 11
A link from a node to one of its children.

regular expression 10
A formal expression that describes a set of strings (a search pattern).

regularity condition 2
The condition $a \cdot f(n/b) \le k \cdot f(n)$ ($k < 1$) required for case 3 of the master theorem.

relaxation 6
If $d[u] + w(u, v) < d[v]$, updating the distance estimate and the parent of v.

residual capacity 8
The amount of flow that can still be sent through an edge: $c - f$.

residual network 8
A network that shows the remaining capacity on each edge and the flow that can be undone (the reverse edge).

reverse-delete algorithm 7
An MST algorithm that examines edges from heaviest to lightest and deletes those whose removal does not disconnect the graph.

rolling hash 9
A hash that can be updated in O(1) when the window slides by one position.

run-length encoding (RLE) 9
A compression method that writes consecutive repeats as (value, count) pairs.

running time 2
The number of basic operations an algorithm performs, or the time it takes, on a particular input.

S

saturated edge 8
An edge whose flow equals its capacity, that is, $f(u, v) = c(u, v)$; no more flow can be sent through it.

search problem 4
Finding whether a value occurs in a data set and, if so, where.

selection sort 3
A sort that selects the smallest item of the unsorted part in each pass and moves it to the front.

sentinel 4
A special value placed at the end to remove the bounds check from a loop.

#

(separate) chaining 4
Resolving collisions by keeping the keys that fall in the same cell in a list.

S

shell sort 3
An insertion sort with a gradually shrinking gap (Donald Shell).

shortest path 6
A path between two vertices whose edge weights have the smallest sum.

shortest-path tree 6
A tree rooted at the source in which the only path from the root to each vertex is a shortest path to that vertex.

single-source shortest paths 6
Shortest paths from one source to all vertices (Dijkstra, Bellman-Ford).

sorting 3
Putting items into ascending or descending order by a key.

source / sink 8
The vertices where the flow starts (s) and ends (t).

space complexity 2
How the extra memory an algorithm uses beyond its input grows with the input size.

spanning tree 7
An acyclic subgraph of a connected graph that contains all of its vertices (V − 1 edges).

sparse / dense graph 5
A graph is sparse if its number of edges is far below the maximum V(V − 1)/2 and dense if it is close to that value.

spurious hit 9
In Rabin-Karp, a window whose hash equals the pattern's hash although the characters differ; the verification is wasted.

stable (sort) 3
Describes a sort that preserves the relative order of items with equal keys.

stack frame 2
The space opened on the call stack for the parameters, local variables and return address when a method is called; it is removed when the method returns.

string 9
A finite sequence of characters from an alphabet.

string matching 9
The problem of finding all occurrences of a pattern in a text.

strongly connected component (SCC) 5
A maximal set of vertices in a directed graph in which there is a path from every vertex to every other vertex.

subset construction 10
A method that converts an NFA into a DFA that recognizes the same language; the DFA states are sets of NFA states.

substring 9
A part of a string made of consecutive characters.

suffix 9
A part taken from the end of a string.

suffix array 10
The starting indices of all suffixes of a string, listed in lexicographic order of the suffixes.

suffix trie / suffix tree 11
A trie containing all suffixes of a string; its compressed form is the suffix tree.

swap 3
Exchanging the positions of two items.

T

ternary search trie (TST) 11
A kind of trie whose nodes each hold one character together with less-than, equal and greater-than links.

tight bound 2
A bound that is both a lower and an upper bound and so gives the growth exactly (Θ).

time complexity 1
How the number of basic operations grows with the input size.

Timsort 3
A hybrid sort that processes naturally sorted runs using insertion sort and merging.

topological sort 5
An ordering of the vertices of a directed acyclic graph in which every edge goes from left to right.

transitive closure 6
A table that tells, for every pair i, j, whether there is a path from i to j; computed with the Boolean (and/or) version of Floyd-Warshall.

trap state 10
A non-accepting state of an automaton that cannot be left once entered, whatever the input.

trie (prefix tree) 11
A structure that stores strings in a tree whose character-labeled edges are shared by strings with a common prefix.

U

undirected graph 5
A graph whose edges can be traversed in both directions.

union by rank 7
A merging method that attaches the root of the shorter tree to the root of the taller tree.

union-find (disjoint sets) 7
A structure that manages disjoint sets with find and union operations.

upper bound 2
A limit that a value cannot exceed; O notation gives an upper bound.

V

vertex / node 5
A point of a graph.

W

weighted graph 5
A graph in which every edge is assigned a number (a weight or cost).

wildcard search 11
A search in which a wildcard character (.) in the pattern matches any letter.

worst case 2
The input of a given size that requires the most operations.

APPENDIX B

Prerequisites for Algorithms

ALGORITHMS · PREREQUISITES

The Algorithms course brings together programming in Java, basic data structures, discrete mathematics, proof techniques, sums and logarithms, probability and counting. The 28 topics on this page collect the background the course builds on in six groups.

Each card has a short explanation, a small example, links to the slides where the topic is used and a check question; clicking the question opens its answer.

Java and programming (9) **Basic data structures (6)**

Discrete mathematics and proof (3) **Analysis tools (5)**

Probability and counting (1) **Other mathematics and concepts (4)**

Java and programming

What you need to read and trace the code in the slides and to write solutions in Java.

J

Java basics: types, conditions, loops

int · long · double · if · for · while

The code in the slides is written in Java, so you need to be able to trace on paper how many times a loop runs and how each variable changes. The int type's limit of roughly ±2.1 billion and integer division (7 / 2 = 3) are common sources of bugs.

WHY IT MATTERS

All the code-trace slides use these constructs; the classic overflow bug in binary search also comes from the int limit.

WHERE IT IS USED **Chapter 3 · Insertion: Code** **Chapter 2 · Counting Steps**

Chapter 4 · Implementation Details

CHECK QUESTION

▶ **int a = 7, b = 2; double d = a / b; What is the value of d?**

EXAMPLE (JAVA)

```
int sum = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) sum += i;       // even numbers are added
}
System.out.println(sum);            // 30
long big = 50_000L * 50_000;        // would overflow as an int
System.out.println(big);            // 2500000000
```

[ ] Arrays and two-dimensional arrays

indices start at 0 · length · int[][]

Sorting, searching and dynamic programming tables all work on arrays. You should know that indices run from 0 to length − 1, that an array is passed to a method by reference, and that a two-dimensional array is traversed row by row.

WHY IT MATTERS

Sorting algorithms work in place on an array, DP tables are two-dimensional arrays, and an adjacency matrix is a two-dimensional array too.

WHERE IT IS USED **Chapter 3 · Bubble: Code** **Chapter 5 · Storing a Graph**

CHECK QUESTION

▶ **What happens if a[3] is read after the line int[] a = {3, 1, 2};?**

EXAMPLE (JAVA)

```
int[] a = new int[5];               // [0, 0, 0, 0, 0]
a[a.length - 1] = 9;                // last element
int[][] dp = new int[3][4];         // 3 rows, 4 columns
for (int i = 0; i < dp.length; i++)
    for (int j = 0; j < dp[i].length; j++)
        dp[i][j] = i + j;
System.out.println(dp[2][3]);       // 5
```

Aa Strings and characters

String · charAt · length() · char codes · compareTo

The Java code of string algorithms works on String objects. s.length() gives the length and s.charAt(i) gives the i-th character; indices start at 0 as in arrays. A String is immutable: s + c builds a new object every time, so StringBuilder is used to accumulate a string inside a loop. The

char type is a 16-bit integer; ch - 'a' converts a lowercase letter to an index in the range 0–25, and a character can be used directly as an index into a counting array. The contents of two strings are compared with equals, not ==, and lexicographic order with compareTo.

WHY IT MATTERS

The brute-force, KMP, Boyer-Moore and Rabin-Karp code compares characters with charAt; trie nodes pick a child with ch - 'a', and hash functions and radix sort use character codes like numbers.

WHERE IT IS USED **Chapter 9 · Brute Force: Trace** **Chapter 11 · Insertion Algorithm**

Chapter 4 · The Hash Function **Chapter 10 · Example: apple · apricot**

CHECK QUESTION

▶ **Does "kedi".compareTo("kek") return a negative or a positive value?**

EXAMPLE (JAVA)

```
String s = "algorithm";
System.out.println(s.length());                // 9
System.out.println(s.charAt(0));               // a
int idx = s.charAt(2) - 'a';                   // 'g' → 6
StringBuilder sb = new StringBuilder();
for (int i = s.length() - 1; i >= 0; i--) sb.append(s.charAt(i));
System.out.println(idx + " " + sb);            // 6 mhtirogla
System.out.println("abc".equals("ab" + "c"));  // true
```

f( ) Methods and parameter passing

return value · pass by value · array reference

Algorithms are usually written as methods: the input is the parameter and the result is the return value. Java passes everything by value; but for arrays and objects the copied value is a reference, so a swap inside the method changes the caller's array.

WHY IT MATTERS

All sorting algorithms use helper methods that swap or shift elements; the partition method returns the final position of the pivot.

WHERE IT IS USED **Chapter 3 · Bubble: Code** **Chapter 3 · Partitioning Procedure**

CHECK QUESTION

▶ **Does writing a = new int[10]; inside a method change the caller's array?**

EXAMPLE (JAVA)

```
static void swap(int[] a, int i, int j) {
    int t = a[i]; a[i] = a[j]; a[j] = t;     // the caller's array changes
}
static void increment(int x) { x++; }        // the caller's variable does not
change

int[] d = {5, 7};
swap(d, 0, 1);               // d = [7, 5]
int y = 1;
increment(y);                // y is still 1
System.out.println(d[0] + " " + y);   // 7 1
```

↻

Recursion and the call stack

base case · recursive call · depth

Merge sort, quicksort, DFS, backtracking and divide-and-conquer algorithms are recursive. You should know that each call is pushed onto the call stack with its own local variables, that forgetting the base case causes a StackOverflowError, and that recursion depth consumes memory.

WHY IT MATTERS

Fast exponentiation, merge sort, recursive DFS and the backtracking template are written directly with recursion.

WHERE IT IS USED

Chapter 5 · Recursive DFS: Pseudocode

CHECK QUESTION

▶ **In the call power(2, 10), how many times does the power method run, counting the first call?**

EXAMPLE (JAVA)

```
static int power(int a, int n) {          // a to the power n, n >= 0
    if (n == 0) return 1;                    // base case
    int half = power(a, n / 2);              // the problem is halved
    return (n % 2 == 0) ? half * half : half * half * a;
}

System.out.println(power(2, 10));         // 1024; depth is about log2(n)
```

NYP

Classes, objects and comparators

class · Comparable · Comparator · generics

Concepts such as an edge, a node, a job or an item are represented by small classes. Sorting and priority queues need a compareTo method or a Comparator to compare elements: in Kruskal's algorithm edges are sorted by weight, and in activity selection activities are sorted by finish time.

WHY IT MATTERS

Most greedy algorithms start with a sort; Dijkstra and Prim put comparable objects into a priority queue.

WHERE IT IS USED **Chapter 7 · Kruskal: Steps and Pseudocode**

CHECK QUESTION

▶ **Why is writing return w - o.w; inside compareTo risky?**

EXAMPLE (JAVA)

```
static class Edge implements Comparable<Edge> {
    final int u, v, w;
    Edge(int u, int v, int w) { this.u = u; this.v = v; this.w = w; }
    public int compareTo(Edge o) { return Integer.compare(w, o.w); }
}

List<Edge> edges = new ArrayList<>();
edges.add(new Edge(0, 1, 7));
edges.add(new Edge(1, 2, 3));
Collections.sort(edges);                              // ascending by weight
edges.sort((a, b) -> Integer.compare(b.w, a.w));   // descending: a Comparator
System.out.println(edges.get(0).w);                   // 7
```



Java collections

ArrayList · HashMap · ArrayDeque · PriorityQueue

The Java code in the slides often uses ready-made structures: ArrayList for adjacency lists, ArrayDeque for the BFS queue and the DFS stack, PriorityQueue for Dijkstra and Prim, HashMap for memoization. Knowing the cost of each basic operation is a prerequisite for computing an algorithm's total running time correctly.

WHY IT MATTERS

BFS is written with a queue, Dijkstra with a priority queue, memoization with a map.

WHERE IT IS USED **Chapter 5 · BFS: Pseudocode** **Chapter 6 · Dijkstra: Java Trace**

CHECK QUESTION

▶ **How long do add and poll take in a PriorityQueue? How long do contains(x) and remove(x) take?**

EXAMPLE (JAVA)

```
Deque<Integer> queue = new ArrayDeque<>();
queue.offer(1); queue.offer(2);
int first = queue.poll();                          // 1 (first in, first out)

PriorityQueue<int[]> pq = new PriorityQueue<>((x, y) -> Integer.compare(x[1],
y[1]));
pq.add(new int[] {7, 40});
pq.add(new int[] {3, 15});
int[] smallest = pq.poll();                        // [3, 15]

Map<Integer, Long> memo = new HashMap<>();
memo.put(10, 55L);
System.out.println(first + " " + smallest[0] + " " + memo.get(10));   // 1 3
55
```



Reading and tracing pseudocode

← assignment · indices starting at 1 · trace table

Slides and textbooks often give algorithms in pseudocode: ← is assignment, arrays may start at 1, and “for j ← 2 to n” includes the final value. When translating to Java you need to account for these differences and be able to write the value of every variable at every step into a trace table.

WHY IT MATTERS

Graph, flow and branch-and-bound algorithms are given first in pseudocode and then in Java.

WHERE IT IS USED **Chapter 3 · Insertion: Code** **Chapter 6 · Dijkstra: Pseudocode**

Chapter 8 · Pseudocode

CHECK QUESTION

▶ **How are the loops written when the pseudocode above is translated to Java with indices starting at 0?**

EXAMPLE

```
INSERTION-SORT(A, n)
  for j ← 2 to n
      key ← A[j]
      i ← j − 1
      while i > 0 and A[i] > key
          A[i + 1] ← A[i]
          i ← i − 1
      A[i + 1] ← key
```

✓

Testing and debugging

edge cases · assertions · small inputs

A correct idea must also be implemented carefully and tested. Testing with empty input, a single element, repeated values, sorted and reverse-sorted arrays, disconnected graphs and large values that may cause overflow should become a habit.

WHY IT MATTERS

The “common mistakes” slide of every chapter collects such edge cases.

WHERE IT IS USED **Chapter 3 · Common Mistakes** **Chapter 4 · Implementation Details**

CHECK QUESTION

▶ **When does mid = (low + high) / 2 give a wrong result in binary search?**

EXAMPLE (JAVA)

```
int[] empty = {}, one = {4}, equal = {2, 2, 2}, rev = {5, 4, 3, 2, 1};
for (int[] t : new int[][] {empty, one, equal, rev}) {
    int[] k = t.clone();
```

```
    Arrays.sort(k);
    for (int i = 1; i < k.length; i++)
        if (k[i - 1] > k[i]) throw new AssertionError("not sorted");
}
System.out.println("all tests passed");
```

Basic data structures

The structures algorithms work on and the cost of their basic operations. These structures are covered in detail in the prerequisite course Data Structures.

→ Linked lists

node · next · O(1) insertion at the head

Like an array, it stores a sequence, but the elements are not contiguous in memory; each node points to the next one. Insertion at the head takes O(1), and access to the i-th element takes O(n). Adjacency lists, chaining in hash tables and the O(1) implementation of an LRU cache rely on linked lists.

WHY IT MATTERS

Chained hash tables and the O(1) LRU cache use linked lists.

WHERE IT IS USED **Chapter 4 · Insertion with Chaining**

CHECK QUESTION

▶ **Why does deleting the last element of a singly linked list take O(n)?**

EXAMPLE (JAVA)

```
static class Node {
    int value; Node next;
    Node(int value, Node next) { this.value = value; this.next = next; }
}

Node head = null;
for (int x = 1; x <= 3; x++) head = new Node(x, head);    // 3 → 2 → 1
for (Node n = head; n != null; n = n.next) System.out.print(n.value + " ");
```

⇅ Stacks and queues

LIFO · FIFO · push/pop · offer/poll

A stack works by the last-in, first-out (LIFO) principle and a queue by first-in, first-out (FIFO); the basic operations of both take O(1). DFS works with a stack (or with the call stack of recursion) and BFS with a queue; backtracking undoes choices in stack order.

WHY IT MATTERS
Iterative DFS uses an explicit stack; BFS and Edmonds-Karp use a queue.

WHERE IT IS USED **Chapter 5 · Tracing Stack DFS** **Chapter 5 · BFS: The Idea**

Chapter 8 · Augmenting Paths with BFS (Java)

CHECK QUESTION

▶ **1, 2, 3 are added in this order to both a queue and a stack. Which element comes out first from each?**

EXAMPLE (JAVA)

```
Deque<Character> stack = new ArrayDeque<>();
for (char c : "abc".toCharArray()) stack.push(c);
StringBuilder sb = new StringBuilder();
while (!stack.isEmpty()) sb.append(stack.pop());
System.out.println(sb);                    // cba
```

▲

Heaps and priority queues

complete binary tree · array layout · O(log n)

In a min-heap every node is smaller than its children, so the smallest element is at the root. Stored as an array, the children of node i are at 2i + 1 and 2i + 2 and its parent is at (i − 1) / 2. Insertion and extracting the minimum take O(log n); heap sort, Dijkstra, Prim, Huffman and the online median rely on this structure.

WHY IT MATTERS
Heap sort, the complexity of Dijkstra's algorithm, building the Huffman tree and the online median rely on the heap.

WHERE IT IS USED **Chapter 3 · Heapsort** **Chapter 6 · Dijkstra: Complexity**

CHECK QUESTION

▶ **Building a heap by inserting n elements one at a time takes O(n log n). How long does bottom-up heap construction take?**

EXAMPLE

```
array:  [2, 5, 3, 9, 6, 4]
index:   0  1  2  3  4  5
children of 2: 5 and 3   (index 1, 2)
```

```
children of 5:  9 and 6   (index 3, 4)
child of 3:     4         (index 5)
```

Hash tables

hash function · collision · average O(1)

A hash table converts a key into an array index with a hash function; insertion and search take O(1) on average. When two keys land in the same cell, the collision is resolved by chaining or linear probing; performance drops as the load factor grows. Memoization, “visited” sets and Rabin-Karp's hash comparison rely on this idea.

WHY IT MATTERS

Search by hashing, Rabin-Karp and memoization use a hash table.

WHERE IT IS USED **Chapter 4 · Hashing: The Idea** **Chapter 4 · Load Factor**

Chapter 9 · Rabin-Karp

CHECK QUESTION

▶ **What is the worst-case search time in a hash table, and when does it occur?**

EXAMPLE (JAVA)

```
int m = 7;
int[] keys = {10, 17, 24};
for (int k : keys) System.out.print(k % m + " ");   // 3 3 3 → collision
Set<String> visited = new HashSet<>();
visited.add("A");
System.out.println(visited.contains("A"));          // true
```

A Trees and binary search trees

root · parent · child · leaf · height

A tree with n nodes has n − 1 edges and exactly one path between any two nodes. The terms root, parent, child, leaf, depth and height are used unchanged in DFS/BFS trees, MSTs, the union-find structure, tries and state-space trees. A binary search tree keeps smaller keys in the left subtree and larger keys in the right subtree; if the tree is balanced, search takes O(log n).

WHY IT MATTERS

The BFS tree, spanning trees, tries and state-space trees are described with the same terms.

WHERE IT IS USED **Chapter 5 · BFS Tree and Shortest Paths**

Chapter 7 · Spanning Trees and MSTs **Chapter 11 · Basic Components**

CHECK QUESTION

▶ **At most how many nodes can a binary tree of height h contain?**

EXAMPLE

```
        8            height 2; leaves 1, 6, 14
      /   \
     3     10        parent of 3 is 8, children are 1 and 6
    / \      \
   1   6      14     inorder traversal: 1 3 6 8 10 14
```

◇ Graph representations

G = (V, E) · adjacency list · adjacency matrix

A graph consists of vertices and edges; edges may be directed or undirected, weighted or unweighted. An adjacency matrix uses V^2 memory and answers an edge query in O(1); an adjacency list uses O(V + E) memory and traverses sparse graphs in O(V + E) time. The terms degree, path, cycle, connectivity and bipartiteness should be familiar.

WHY IT MATTERS

All graph chapters, and the TSP examples in branch and bound, assume a graph representation.

WHERE IT IS USED **Chapter 5 · Storing a Graph** **Chapter 5 · Kinds of Graphs**

Chapter 6 · Graphs and Shortest Paths

CHECK QUESTION

▶ **Which representation suits a road network with 10,000 vertices and 30,000 edges?**

EXAMPLE (JAVA)

```
int V = 4;
List<List<Integer>> adj = new ArrayList<>();
for (int i = 0; i < V; i++) adj.add(new ArrayList<>());
int[][] edges = {{0, 1}, {0, 2}, {2, 3}};
for (int[] e : edges) {                    // undirected: both directions are
added
    adj.get(e[0]).add(e[1]);
    adj.get(e[1]).add(e[0]);
}
System.out.println(adj.get(0));            // [1, 2]
```

Discrete mathematics and proof

The language used to define algorithms and prove them correct.

∈ Sets, functions and relations

∪ ∩ \ · |A| · f: A → B · relation

Algorithms are described in the language of sets: G = (V, E), the set of visited vertices, the cut (S, V − S) in an MST, the subset of items chosen in the knapsack. You should know union, intersection, difference, Cartesian product and subsets; the domain and codomain of a function; and the reflexivity, symmetry and transitivity of relations.

WHY IT MATTERS

The cut property, Floyd-Warshall's transitive closure and the feasible solutions in branch and bound are defined with sets.

WHERE IT IS USED **Chapter 7 · Cut and Cycle Properties**

Chapter 6 · The Floyd-Warshall Algorithm

CHECK QUESTION

▶ **How many subsets does a set with n elements have, and which brute-force method is that costly?**

EXAMPLE

```
A = {1, 2, 3},  B = {2, 3, 4}
A ∪ B = {1, 2, 3, 4}    A ∩ B = {2, 3}    A \ B = {1}
|A × B| = 3 · 3 = 9     number of subsets of A: 2³ = 8
The relation "reachable" is reflexive and transitive; in an undirected graph
it is also symmetric.
```

∀ Logic and quantifiers

∧ ∨ ¬ ⇒ · ∀ ∃ · De Morgan

Loop conditions, pruning conditions and definitions are logical statements. You need to be able to read quantified sentences such as “for every $n \geq n_0$, $f(n) \leq c \cdot g(n)$,” to negate a statement correctly (De Morgan), and to tell “p ⇒ q” from “q ⇒ p.”

WHY IT MATTERS

Big-O is defined with quantifiers; the inner loop of insertion sort stops on a compound condition.

WHERE IT IS USED **Chapter 2 · Definition of Big O** **Chapter 3 · Insertion: Code**

CHECK QUESTION

▶ **What is the negation of the statement “every vertex has at least one neighbor”?**

EXAMPLE

```
¬(a ∧ b)  ≡  ¬a ∨ ¬b                              (De Morgan)
while (i > 0 && A[i] > x) loop exit condition:  i <= 0 || A[i] <= x
f(n) = O(g(n))  ⇔  ∃c > 0, ∃n₀, ∀n ≥ n₀ : f(n) ≤ c · g(n)
```

n+1 Induction and loop invariants

base case · inductive step · initialization–maintenance–termination

Induction proves a statement for every n by showing that P(1) is true and that P(k) ⇒ P(k + 1). A loop invariant is its adaptation to algorithms: a statement that is true before the loop (initialization), is preserved in every iteration (maintenance) and gives the result when the loop ends (termination). The correctness of recursive algorithms is also proved by induction.

WHY IT MATTERS

The correctness of sorting algorithms and the solutions of recurrences are shown by induction.

WHERE IT IS USED **Chapter 3 · Insertion: Steps** **Chapter 2 · f3: Expanding the Recurrence**

CHECK QUESTION

▶ **How is the equality $1 + 3 + 5 + \ldots + (2n - 1) = n^2$ shown by induction?**

EXAMPLE

```
Insertion sort, invariant: "at the start of iteration j, A[0..j-1] is sorted."
  Initialization: j = 1 → A[0..0] is a single element, so it is sorted.
  Maintenance:    A[j] is shifted into place → A[0..j] is sorted.
  Termination:    j = n → A[0..n-1], that is, the whole array is sorted.
```

Analysis tools

Sums, logarithms and growth rates for computing running times.

Σ Summation notation and series

arithmetic · geometric · harmonic

The cost of nested loops and recursion trees is written as a sum. Three series are fundamental: the arithmetic series grows quadratically; a geometric series with ratio less than 1 converges to a constant, while one with ratio greater than 1 is of the order of its largest term; the harmonic

series grows logarithmically.

WHY IT MATTERS

Counting loop iterations, adding up the levels of a recursion tree and the expected number of comparisons of randomized quicksort use these series.

WHERE IT IS USED **Chapter 2 · f2: The Sum** **Chapter 2 · Loop Patterns**

CHECK QUESTION

▶ **How many times does the body of the loop for (i = 1; i < n; i *= 2) for (j = 0; j < i; j++) run?**

EXAMPLE

```
Σ (i = 1..n) i       = n(n + 1) / 2          = Θ(n²)
Σ (i = 0..k) 2^i     = 2^(k+1) − 1             (1 + 2 + 4 + … + 2^k)
Σ (i = 0..∞) (1/2)^i = 2
H(n) = Σ (i = 1..n) 1/i ≈ ln n                = Θ(log n)
```

log Logarithm and exponent rules

$\log_2 n$ · exponent rules · change of base

Every algorithm that halves its input (binary search, merge sort, heaps) produces logarithmic depth: reducing n to 1 means halving $\log_2 n$ times. You should know the rules of exponents and logarithms, that the base of a logarithm is asymptotically irrelevant, and the approximation $2^{10} \approx 10^3$.

WHY IT MATTERS

The number of steps of binary search, the n^(log_b a) term in the master theorem and Karatsuba's exponent n^1.585 are computed with logarithms.

WHERE IT IS USED **Chapter 4 · How Many Steps?** **Chapter 2 · Three Cases**

CHECK QUESTION

▶ **At most about how many comparisons does binary search make on a sorted array of a billion elements?**

EXAMPLE

```
log2(1024) = 10        log2(10^6) ≈ 20       log2(10^9) ≈ 30
a^(m+n) = a^m · a^n     (a^m)^n = a^(m·n)     log(x·y) = log x + log y
n^(log_b a) = a^(log_b n)              appears in the master theorem
log_3 n = log_2 n / log_2 3            the base changes only a constant factor
```

⌊⌋ Floor and ceiling

⌊x⌋ · ⌈x⌉ · integer division

When an array is split in two, the halves hold ⌊n/2⌋ and ⌈n/2⌉ elements; in Java n / 2 gives the floor for positive numbers. Floor and ceiling appear in the middle index of binary search, the parent index in a heap, the height of a binary tree and approximation ratios.

WHY IT MATTERS

The split step of merge sort, the middle index of binary search and the index computations of a heap use floor and ceiling.

WHERE IT IS USED **Chapter 3 · Merge Sort: Recursion**

Chapter 4 · The Binary Search Algorithm **Chapter 3 · Heapsort: Example**

CHECK QUESTION

▶ **What is −7 / 2 in Java? What is ⌊−7/2⌋?**

EXAMPLE

```
⌊7/2⌋ = 3      ⌈7/2⌉ = 4      ⌊n/2⌋ + ⌈n/2⌉ = n
height of a heap with n nodes is ⌊log2 n⌋:  n = 10 → 3
Java: 7 / 2 == 3,  (7 + 2 − 1) / 2 == 4     for positive a, b: ⌈a/b⌉ = (a + b
− 1) / b
```

O Asymptotic growth

$1 < \log n < n < n \log n < n^2 < 2^n < n!$

For large n the dominant term decides: $3n^2 + 100n + 7 = \Theta(n^2)$. You should be able to order the growth classes and, assuming roughly 10^8–10^9 simple operations per second, estimate which complexity is practical for which n. O is the upper bound, Ω the lower bound and Θ the tight bound.

WHY IT MATTERS

Every chapter compares algorithms by their complexity; growth classes determine which method suits which input size.

WHERE IT IS USED **Chapter 2 · Growth Curves** **Chapter 2 · Asymptotic Rules**

Chapter 2 · Common Growth Classes

CHECK QUESTION

▶ **Which eventually grows faster: n^{100} or 1.01^n?**

EXAMPLE

```
approximate operation counts for n = 10^6
  n log2 n ≈ 2 · 10^7     → under a second
  n²       = 10^12        → hours
  2^n      : even n = 60 is ≈ 10^18 → infeasible
```

T(n)

Recurrence relations

T(n) = aT(n/b) + f(n) · recursion tree

The running time of a recursive algorithm is written in terms of itself: merge sort solves two halves and merges in linear time, so T(n) = 2T(n/2) + n. Being used to solving a recurrence by unrolling it, or by adding up the work at each level of the recursion tree, makes the master theorem much easier to follow.

WHY IT MATTERS

Recurrences are solved with the master theorem; every divide-and-conquer algorithm is analyzed with a recurrence.

WHERE IT IS USED **Chapter 2 · Recurrence Relations** **Chapter 2 · f4: Recursion Tree**

Chapter 2 · Divide-and-Conquer Recurrence

CHECK QUESTION

▶ **What is the recurrence of binary search, and what is its solution?**

EXAMPLE

```
T(n) = 2T(n/2) + n
  level 0:  n                   = n
  level 1:  n/2 + n/2           = n
  level 2:  4 · n/4             = n
  … log2 n levels               ⇒ T(n) = Θ(n log n)
T(n) = T(n − 1) + 1  ⇒ Θ(n)       T(n) = 2T(n − 1) + 1  ⇒ Θ(2^n)
```

Probability and counting

For measuring the size of search spaces and the behavior of randomized algorithms.

n!

Combinatorics: permutations and combinations

multiplication rule · n! · C(n, k) · 2^n

The cost of brute force, backtracking and branch and bound is the size of the search space; counting it takes the multiplication rule, n! permutations, C(n, k) combinations and 2^n subsets. For example, in a TSP with n cities, once the starting city is fixed there are (n − 1)! tours.

WHY IT MATTERS

Generating permutations, the brute-force solution of TSP and the size of the n-queens search tree are found by counting.

WHERE IT IS USED **Chapter 1 · O(n!): Permutations**

CHECK QUESTION

▶ **In how many ways can 8 queens be placed on an 8 × 8 board with one queen in each row? In how many ways if the columns must also be distinct?**

EXAMPLE

```
P(n, k) = n! / (n − k)!          C(n, k) = n! / (k! · (n − k)!)
C(5, 2) = 10       5! = 120        2^10 = 1024
n = 10 cities: 9! = 362,880 tours      n = 20 cities: 19! ≈ 1.2 · 10^17 tours
```

Other mathematics and concepts

Matrices, modular arithmetic, bit operations, and the basics of optimization and concurrency used in particular chapters.

M Matrix operations

row–column product · identity matrix · block form

Directly multiplying two n × n matrices performs n^3 multiplications in three nested loops; Strassen reduces this number with block matrices. For adjacency matrices, Floyd-Warshall's distance table and the matrix–vector product in power iteration, you need to know the row–column product, the identity matrix and how to split a matrix into blocks.

WHY IT MATTERS

Matrix multiplication is the $O(n^3)$ example in complexity analysis, the subject of Strassen's algorithm and the basic step of power iteration.

WHERE IT IS USED **Chapter 1 · $O(n^3)$: Matrix Multiplication**

Chapter 6 · Floyd-Warshall: Steps

CHECK QUESTION

▶ **What is the product [[1, 2], [3, 4]] · [[0, 1], [1, 0]]?**

EXAMPLE (JAVA)

```
static int[][] multiply(int[][] a, int[][] b) {
    int n = a.length;
```

```
    int[][] c = new int[n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];    // n³ multiplications in
total
    return c;
}

int[][] c = multiply(new int[][] {{1, 2}, {3, 4}}, new int[][] {{0, 1}, {1,
0}});
System.out.println(Arrays.deepToString(c));      // [[2, 1], [4, 3]]
```

xy

Plane geometry and distance measures

Euclidean distance · Manhattan distance · triangle inequality · cross product

The Euclidean distance between two points in the plane is $\sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$, while the Manhattan distance, measured on a grid with horizontal and vertical steps only, is $|x_1 - x_2| + |y_1 - y_2|$. If a distance measure satisfies the triangle inequality, then d(a, c) ≤ d(a, b) + d(b, c): visiting a third point never shortens the route. The cross product of the vectors from A to B and from A to P, (B.x − A.x)(P.y − A.y) − (B.y − A.y)(P.x − A.x), is computed with integers; its sign tells whether P is to the left of line AB, to the right, or on it. To merely compare distances there is no need to take square roots; comparing their squares gives the same result.

WHY IT MATTERS

The heuristics of the A* algorithm are the Euclidean and Manhattan distances; the convex hull relies on the orientation test, the closest pair of points on Euclidean distance, and the 2-approximation of metric TSP on the triangle inequality.

WHERE IT IS USED

Chapter 6 · Heuristic Functions

CHECK QUESTION

▶ **What are the Euclidean and Manhattan distances for A = (0, 0) and B = (3, 4)? On which side of line AB is the point P = (1, 5)?**

EXAMPLE (JAVA)

```
static double euclid(int x1, int y1, int x2, int y2) {
    return Math.hypot(x1 - x2, y1 - y2);
}
static int manhattan(int x1, int y1, int x2, int y2) {
    return Math.abs(x1 - x2) + Math.abs(y1 - y2);
}
static long cross(int ax, int ay, int bx, int by, int px, int py) {
    return (long) (bx - ax) * (py - ay) - (long) (by - ay) * (px - ax);
```

```
}
System.out.println(euclid(0, 0, 3, 4));          // 5.0
System.out.println(manhattan(0, 0, 3, 4));      // 7
System.out.println(cross(0, 0, 3, 4, 1, 5));   // 11 (P is on the left)
```

mod Modular arithmetic and GCD

a mod m · (a · b) mod m · GCD · prime numbers

Hash functions, Rabin-Karp's rolling hash, pseudorandom number generators and the Miller–Rabin test work with modular arithmetic. You need to know that taking the remainder at every step of additions and multiplications does not change the result, how the GCD is computed with Euclid's algorithm, and the definition of a prime number.

WHY IT MATTERS

Euclid's algorithm, rolling hashes, Miller–Rabin and pseudorandom numbers use modular arithmetic.

WHERE IT IS USED

Chapter 1 · Euclid's Algorithm **Chapter 9 · Rabin-Karp: How It Works**

CHECK QUESTION

▶ **How is (17 · 23) mod 5 computed by taking remainders before multiplying?**

EXAMPLE (JAVA)

```
static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

System.out.println(gcd(48, 18));                          // 6
long M = 1_000_000_007L, h = 0;
for (char c : "abc".toCharArray()) h = (h * 31 + c) % M;  // the basis of a
rolling hash
System.out.println(h);                                     // 96354
```

01 Binary numbers and bit operations

& | ^ ~ << >> · powers of 2 · subset masks

Computers store numbers in binary; powers of 2, shifts and masks shorten many algorithms. The bits of an integer can represent a subset; the expression 1 << k gives 2^k, and x & (x − 1) clears the lowest 1 bit. In the n-queens problem an O(1) safety test, and iterating over all subsets for small n, are done with bit masks.

WHY IT MATTERS

Radix sort works digit by digit; the n-queens solution uses bit masks.

WHERE IT IS USED **Chapter 3 · Radix Sort**

CHECK QUESTION

▶ **What is the binary representation of 13? What are 13 >> 1 and 13 & 1?**

EXAMPLE (JAVA)

```
int n = 3;
for (int mask = 0; mask < (1 << n); mask++) {           // 2³ = 8 subsets
    StringBuilder s = new StringBuilder("[");
    for (int i = 0; i < n; i++)
        if ((mask & (1 << i)) != 0) s.append(i);
    System.out.print(s.append("] "));
}
// [] [0] [1] [01] [2] [02] [12] [012]
System.out.println((16 & (16 - 1)) == 0);             // true: 16 is a power
of 2
```

APPENDIX C

Important People in Algorithms

ALGORITHMS · PEOPLE

The 46 people behind the algorithms and ideas in the eleven chapters of the course: from Euclid's algorithm and the Turing machine to Dijkstra and Kruskal, from network flow to KMP and data compression.

Each card gives the person's years, country and institution, best-known contribution and a short summary; the links lead to the slides where that contribution is covered. The cards can be filtered by chapter, searched, and sorted by year of contribution or by surname.

EU

Euclid

c. 300 BCE · Ancient Greek · Alexandria

GCD algorithm · Elements

In the Elements he described how to find the greatest common divisor of two numbers by repeated subtraction; today we write the same method with division and remainder. Euclid's algorithm, based on the identity gcd(a, b) = gcd(b, a mod b), is one of the oldest algorithms still in use. Because the larger number at least halves every two steps, the number of steps is bounded by O(log min(a, b)).

IN THE COURSE **Chapter 1 · Euclid's Algorithm** **Chapter 1 · Euclid: Complexity**

GCD termination logarithmic number of steps

AK

Al-Khwarizmi

c. 780 – c. 850 · Baghdad · House of Wisdom

Arithmetic with Hindu-Arabic numerals · algebra

A mathematician working at the House of Wisdom in Baghdad. His book on calculating with Hindu-Arabic numerals reached Europe in a Latin translation (Algoritmi de numero Indorum). The word “algorithm” comes from the Latinized form of his name, and the word “algebra” from

the title of his work al-Kitāb al-mukhtaṣar fī ḥisāb al-jabr wa-l-muqābala. He is the emblem of the idea of solving problems in clear, repeatable steps.

IN THE COURSE **Chapter 1 · What Is an Algorithm?**

origin of the word “algorithm” step-by-step method

LF

Leonardo Fibonacci

c. 1170 – c. 1250 · Italian · Pisa

Liber Abaci · 1202

With Liber Abaci (1202) he introduced Hindu-Arabic numerals to Europe; the book's rabbit problem generates the sequence 1, 1, 2, 3, 5, 8, …. Solving the recurrence F(n) = F(n − 1) + F(n − 2) by plain recursion takes exponential time, because the same subproblems are computed again and again. This example is the best-known starting point for dynamic programming and the memoization technique.

IN THE COURSE **Chapter 1 · $O(2^n)$: Fibonacci**

recurrence relation overlapping subproblems

LE

Leonhard Euler

1707–1783 · Swiss · St. Petersburg, Berlin

Bridges of Königsberg · 1736

In 1736 he proved that no walk crosses each of Königsberg's seven bridges exactly once; treating the land masses as vertices and the bridges as edges is regarded as the beginning of graph theory. He showed that such a walk requires the number of odd-degree vertices to be 0 or 2. The Euler tour, which uses every edge once, is also a step of Christofides' algorithm.

IN THE COURSE **Chapter 5 · Graphs Are Everywhere**

graph theory Euler tour

CG

Carl Friedrich Gauss

1777–1855 · German · Göttingen

Arithmetic series · 1 + 2 + … + n

According to a well-known story from his school years, he added the numbers from 1 to 100 in an instant by pairing terms from the two ends, arriving at n(n + 1)/2. This sum shows up every time nested loops are counted as $O(n^2)$. The trick of multiplying two complex numbers with three real multiplications instead of four is also attributed to him; the Karatsuba method applies a similar idea to large integers.

IN THE COURSE **Chapter 2 · f2: The Sum**

arithmetic series counting loop iterations

GL

Gabriel Lamé

1795–1870 · French · Paris

Number of steps of Euclid's algorithm · 1844

In 1844 he proved that the number of divisions Euclid's algorithm performs cannot exceed five times the number of decimal digits of the smaller number. He used Fibonacci numbers in the proof: the smallest input that needs a given number of divisions is a pair of consecutive Fibonacci numbers, because every quotient in these inputs is 1 and the numbers shrink as slowly as possible. The theorem is regarded as one of the earliest studies of an algorithm's worst case and is the precise form of the O(log min(a, b)) bound.

IN THE COURSE **Chapter 1 · Euclid: Complexity**

Lamé's theorem Euclid's algorithm Fibonacci numbers

AC

Arthur Cayley

1821–1895 · British · Cambridge

Cayley's formula · 1889

In 1889 he published the formula showing that the complete graph on n vertices has exactly n^{n-2} spanning trees; Carl Wilhelm Borchardt had found the same number with a determinant in 1860. The number grows very fast: there are 16 spanning trees on 4 vertices and 100 million on 10. Finding the cheapest spanning tree by trying every tree is therefore impractical; greedy algorithms such as Kruskal's and Prim's reach the same result in polynomial time. Cayley is also one of the founders of matrix algebra and of the abstract notion of a group.

IN THE COURSE **Chapter 7 · Spanning Trees and MSTs**

Cayley's formula number of spanning trees complete graph

OB

Otakar Borůvka

1899–1995 · Czech · Brno

First MST algorithm · 1926

In 1926 he published the first minimum spanning tree (MST) algorithm, in order to build Moravia's electrical network at the lowest cost. In each round every component picks the lightest edge leaving it; since the number of components at least halves each round, O(log V) rounds suffice. Because the choices within a round are independent of one another, the method also suits parallel and distributed settings.

IN THE COURSE **Chapter 7 · Borůvka's Algorithm**

minimum spanning tree component

AT

Alan Turing

1912–1954 · British · Cambridge, Manchester

Turing machine · 1936

In his 1936 paper “On Computable Numbers” he formalized the notion of an algorithm with the Turing machine and showed that no algorithm can solve the halting problem. Together with the result Alonzo Church obtained the same year with the lambda calculus, this definition is accepted as the meaning of “effectively computable” (the Church–Turing thesis). The classes P and NP are likewise defined with a time-bounded form of the same model.

IN THE COURSE **Chapter 1 · What Is an Algorithm?**

computability halting problem Church–Turing thesis

JN

John von Neumann

1903–1957 · Hungarian-born American · Princeton IAS

Merge sort · 1945

In the programs he wrote for the EDVAC in 1945 he described merge sort; Knuth counts this program among the first sorting programs written for a computer. A classic example of the divide-and-conquer approach, the method runs in O(n log n) time even in the worst case and is stable. He helped develop the Monte Carlo method together with Stanisław Ulam, and today's computer architecture bears his name.

IN THE COURSE **Chapter 3 · Merge Sort**

divide and conquer stable sorting Monte Carlo

CS

Claude Shannon

1916–2001 · American · Bell Labs, MIT

Information theory · 1948

With his 1948 paper “A Mathematical Theory of Communication” he founded information theory and defined entropy as the lower bound on the average number of bits per symbol in lossless compression. The average length of a Huffman code exceeds the entropy by at most 1 bit. His 1937 master's thesis, which applied Boolean algebra to switching circuits, is also among the foundations of digital circuit design.

IN THE COURSE **Chapter 9 · String Compression**

entropy compression limit

RH

Richard Hamming

1915–1998 · American · Bell Labs

Hamming distance · 1950

In his 1950 paper “Error Detecting and Error Correcting Codes,” published in the Bell System Technical Journal, he introduced error-correcting codes and the measure now called Hamming distance. The Hamming distance is the number of positions at which two strings of equal length differ, and it is computed in O(n) time with a single scan; for example, the distance between “karolin” and “kathrin” is 3. It is undefined for strings of different lengths; in that case the Levenshtein distance, which also allows insertions and deletions, is used. He received the Turing Award in 1968.

IN THE COURSE **Chapter 10 · Similarity Measures**

Hamming distance error-correcting code string similarity

SK

Stephen C. Kleene

1909–1994 · American · University of Wisconsin

Regular expressions · 1951

In a 1951 RAND report (published in 1956) he defined regular expressions and showed that they describe exactly the languages recognized by finite automata (Kleene's theorem). The * operation, meaning “zero or more repetitions,” is called the Kleene star after him. Together with

recursive function theory, he is one of the founders of computability theory.

IN THE COURSE **Chapter 10 · Regular Expressions** **Chapter 10 · Quantifiers and Grouping**

regular expression Kleene star

DH David A. Huffman

1925–1999 · American · MIT, UC Santa Cruz

Huffman coding · 1952

As a graduate student at MIT, he solved the problem of finding the shortest prefix code as a term paper that replaced the final exam in Robert Fano's course, and published the result in 1952. The greedy method of repeatedly merging the two least frequent nodes minimizes the expected number of bits per character and beats his professor's Shannon–Fano method. Huffman coding is today a stage of formats such as DEFLATE and JPEG.

IN THE COURSE **Chapter 9 · String Compression**

Huffman coding prefix code greedy choice

HL Hans Peter Luhn

1896–1964 · German-born American · IBM

Distributing keys into buckets · 1953

In 1953 at IBM he proposed speeding up search by distributing keys into “buckets” with a function; this is one of the earliest known examples of hash tables and chaining. The Luhn check-digit algorithm, still used to validate credit card numbers, and keyword-in-context (KWIC) indexing are also his.

IN THE COURSE **Chapter 4 · Hashing: The Idea** **Chapter 4 · Insertion with Chaining**

hash table chaining

RB Richard Bellman

1920–1984 · American · RAND Corporation

Dynamic programming · 1950s

In the 1950s at the RAND Corporation he coined the name “dynamic programming” for multistage decision problems and stated the principle of optimality: every part of an optimal solution is itself optimal for its own subproblem. In 1958 he published the single-source shortest path method that allows negative edge weights (Bellman-Ford). The phrase “curse of dimensionality” is also his.

IN THE COURSE **Chapter 6 · The Bellman-Ford Algorithm**

dynamic programming optimal substructure Bellman-Ford

GA Gene Amdahl

1922–2015 · American · IBM

Linear probing (1954) · Amdahl's law (1967)

In 1954, in an assembler written for the IBM 701 together with Elaine McGraw and Arthur Samuel, he used linear probing, which moves to the next empty cell when a collision occurs. He was one of the chief architects of the IBM System/360. In a conference talk in 1967 he argued that the sequential part of a program limits parallel speedup; this observation is known as Amdahl's law.

IN THE COURSE **Chapter 4 · Linear Probing: Insertion**

linear probing speedup Amdahl's law

FF L. R. Ford Jr. ve D. R. Fulkerson

1927–2017 · 1924–1976 · American · RAND Corporation

Maximum flow · 1956

In 1956 at the RAND Corporation they published the Ford-Fulkerson method, which increases the flow until the residual network has no augmenting path, together with the max-flow min-cut theorem. In the same years Ford also gave the shortest path method that relaxes edges repeatedly, which is why that algorithm is called Bellman-Ford. Their 1962 book Flows in Networks made network flow a field of its own.

IN THE COURSE **Chapter 8 · The Ford-Fulkerson Method** **Chapter 8 · Max-Flow Min-Cut** **Chapter 6 · The Bellman-Ford Algorithm**

Ford-Fulkerson augmenting path minimum cut

JK

Joseph Kruskal

1928–2010 · American · Princeton, Bell Labs

Kruskal's algorithm · 1956

In a short 1956 paper he gave the MST algorithm that examines the edges in order of weight and adds those that do not form a cycle; the same paper also describes the reverse-delete method. Today the cycle check is done with the union-find structure, and the algorithm runs in O(E log E) time. He also holds an important place in statistics through contributions such as multidimensional scaling.

IN THE COURSE **Chapter 7 · Kruskal's Algorithm**

Chapter 7 · The Reverse-Delete Algorithm

Kruskal's algorithm union-find reverse-delete

EM

Edward F. Moore

1925–2003 · American · Bell Labs

Shortest path through a maze (BFS) · 1957

At Bell Labs he described breadth-first search (BFS) for finding the shortest path through a maze ("The Shortest Path Through a Maze," presented in 1957, published in 1959). The work shows that shortest paths in unweighted graphs can be found layer by layer; because of the method he gave for the weighted case, Bellman-Ford is sometimes called Bellman–Ford–Moore. The Moore machine type of finite state machine also bears his name.

IN THE COURSE **Chapter 5 · BFS: The Idea** **Chapter 5 · BFS in a Maze**

Chapter 10 · Kinds of FSM

BFS Moore machine

RP

Robert C. Prim

1921–2021 · American · Bell Labs

Prim's algorithm · 1957

In 1957 at Bell Labs he published the minimum spanning tree algorithm that starts from one vertex and grows the tree at each step by the lightest edge leaving it. Vojtěch Jarník had found the same method in 1930 and Dijkstra rediscovered it in 1959, which is why it is also called the Jarník–Prim algorithm. With a priority queue it runs in O(E log V) time, and its structure is very similar to Dijkstra's algorithm.

IN THE COURSE **Chapter 7 · Prim's Algorithm** **Chapter 7 · Prim and Dijkstra**

Prim's algorithm cut property

DS

Donald L. Shell

1924–2015 · American · General Electric

Shell sort · 1959

In the method he published in 1959, the elements are first passed through insertion sort at wide gaps and then at progressively narrower gaps. Distant inversions are thereby fixed early, and the $O(n^2)$ behavior of insertion sort improves greatly in practice. The complexity depends on the gap sequence and remains an open question for some sequences.

IN THE COURSE **Chapter 3 · Shell Sort** **Chapter 3 · Shell: Steps**

Shell sort inversion

ED

Edsger W. Dijkstra

1930–2002 · Dutch · Amsterdam, Eindhoven; Austin, USA

Shortest paths · 1959

By his own account he designed the shortest path algorithm in about twenty minutes in Amsterdam in 1956 and published it in 1959; the same paper independently gives Prim's method for the minimum spanning tree. Finalizing the vertex with the smallest tentative distance and relaxing its edges gives the correct result for non-negative weights and takes $O((V + E) \log V)$ time with a priority queue. He is also known for structured programming and semaphores, and received the Turing Award in 1972.

IN THE COURSE **Chapter 6 · Dijkstra's Algorithm** **Chapter 6 · Relaxation**

Chapter 7 · Prim and Dijkstra

Dijkstra's algorithm relaxation priority queue

MR

Michael O. Rabin

b. 1931 · Israeli · Hebrew University of Jerusalem; Harvard

Miller–Rabin (1980) · Rabin-Karp (1987)

In 1959, together with Dana Scott, he defined nondeterministic finite automata; this work earned the 1976 Turing Award. In 1980 he turned Gary Miller's primality test (1976) into an unconditional probabilistic test: in the Miller–Rabin test a composite number passes a round with probability at most 1/4. With Karp he published the Rabin-Karp string matching algorithm, which uses a rolling hash, in 1987; he is one of the pioneers of using randomness deliberately in algorithm design.

IN THE COURSE **Chapter 9 · Rabin-Karp** **Chapter 10 · FSM Definition**

primality test rolling hash finite automaton

TH C. A. R. (Tony) Hoare

1934–2026 · British · Oxford

Quicksort · 1959–1961

He designed quicksort in 1959–1960 as a visiting student at Moscow State University; the algorithm was published in 1961. Partitioning around a pivot and sorting the two sides recursively takes O(n log n) on average and O(n^2) with bad pivots. In the same year he published the Find method for the k-th smallest element (today's QuickSelect); he founded Hoare logic for program correctness and received the Turing Award in 1980.

IN THE COURSE **Chapter 3 · Quicksort** **Chapter 3 · Partitioning Procedure**

quicksort partitioning QuickSelect

EF Edward Fredkin

1934–2023 · American · MIT, Carnegie Mellon

Trie · 1960

In 1960 he published the idea of storing strings in a tree that shares common prefixes and named the structure “trie,” from the middle of the word “retrieval”; René de la Briandais had described the same structure a year earlier (1959). Fredkin is also known for reversible computing and the Fredkin gate, and he directed Project MAC at MIT.

IN THE COURSE **Chapter 11 · Trie (Prefix Tree)** **Chapter 11 · Why a Trie?**

trie prefix query

RF

Robert W. Floyd

1936–2001 · American · Carnegie Mellon, Stanford

Floyd-Warshall · 1962

In 1962 he published the algorithm that finds shortest paths between all pairs (all-pairs shortest paths) in $O(V^3)$ time; this dynamic program, which adds intermediate vertices one at a time, is known today as Floyd-Warshall. In 1964 he gave the method that builds a heap in O(n) time, and the “tortoise and hare” method for detecting cycles in lists is also attributed to him. He received the Turing Award in 1978 for his pioneering work in program verification.

IN THE COURSE **Chapter 6 · The Floyd-Warshall Algorithm**

Chapter 3 · Heapsort: Building the Heap

all-pairs shortest paths heap construction

SW

Stephen Warshall

1935–2006 · American

Transitive closure · 1962

In his 1962 paper “A Theorem on Boolean Matrices” he published the method that computes the transitive closure of a relation on a Boolean matrix in $O(V^3)$ time: if there is a path from i to k and from k to j, then there is also a path from i to j. Because Floyd's shortest path algorithm of the same year uses the same triple loop, the two are cited together. Bernard Roy had published a similar result in 1959.

IN THE COURSE **Chapter 6 · Floyd-Warshall: Steps**

Chapter 6 · Floyd-Warshall: Pseudocode

transitive closure adjacency matrix

AK

Arthur B. Kahn

Communications of the ACM 5(11), 1962

Topological sorting · 1962

In his 1962 paper “Topological Sorting of Large Networks” he gave the algorithm that finds the topological order of a directed acyclic graph by repeatedly removing vertices of in-degree zero. Vertices with in-degree 0 are placed in a queue; each vertex taken from the queue is written to

the order and the in-degree of its successors is decreased by one, and a vertex whose in-degree drops to zero joins the queue. The algorithm runs in O(V + E) time. If not all vertices can be removed, the graph has a cycle, so the same method also performs cycle detection.

IN THE COURSE **Chapter 5 · Topological Sort** **Chapter 5 · Kahn's Algorithm: Code Trace**

topological sorting in-degree DAG

JW J. W. J. Williams

1930–2012 · British-Canadian

Heap sort · 1964

In 1964 he published heap sort, and the binary heap structure it rests on, in Communications of the ACM under the title "Algorithm 232: Heapsort." Turning the array into a max-heap and repeatedly moving the root to the end requires O(n log n) time in the worst case and O(1) extra memory. In the same year Robert Floyd gave the improvement that builds the heap in O(n) time.

IN THE COURSE **Chapter 3 · Heapsort** **Chapter 3 · Heapsort: Building the Heap**

heap in-place sorting

JE Jack Edmonds

b. 1934 · American-born · Waterloo, Canada

Blossom algorithm (1965) · Edmonds-Karp (1972)

In his 1965 paper "Paths, Trees, and Flowers" he gave the blossom algorithm, which finds a maximum matching in general graphs in polynomial time, and argued that "efficient algorithm" should mean polynomial-time algorithm. In 1972, with Karp, he showed that the Ford-Fulkerson method runs in $O(V \cdot E^2)$ time when augmenting paths are chosen by BFS. In 1971 he showed that the greedy algorithm is optimal exactly on matroids.

IN THE COURSE **Chapter 8 · The Edmonds-Karp Algorithm**

Edmonds-Karp polynomial time matroid

VL Vladimir Levenshtein

1935–2017 · Soviet-Russian · Keldysh Institute of Applied Mathematics

Levenshtein distance · 1965

In 1965 he defined the minimum number of insertions, deletions and substitutions needed to turn one string into another; the measure bears his name. His aim was error-correcting codes; the measure is used today in spell checking, in comparing DNA sequences and in approximate search. Computing the distance by dynamic programming in O(m·n) time (Wagner–Fischer, 1974) is the table method used in the course.

IN THE COURSE **Chapter 10 · Levenshtein Distance** **Chapter 10 · The DP Recurrence**

edit distance DP table

DK Donald E. Knuth

b. 1938 · American · Stanford

The Art of Computer Programming · 1968–

With the multivolume The Art of Computer Programming he turned algorithm analysis into a discipline; he popularized computing operation counts exactly and average-case analysis. In 1976 he proposed the present-day meanings of the O, Ω and Θ notations. With Morris and Pratt he published the KMP string matching algorithm (1977), developed the TeX typesetting system, and received the Turing Award in 1974.

IN THE COURSE **Chapter 2 · Ω and Θ** **Chapter 9 · Knuth-Morris-Pratt**

algorithm analysis asymptotic notation KMP

A* Peter Hart, Nils Nilsson ve Bertram Raphael

Nilsson: 1933–2019 · American · Stanford Research Institute (SRI)

A* algorithm · 1968

They published the A* algorithm in 1968 for the path planning of the Shakey robot at SRI. Raphael proposed adding the cost so far to Nilsson's heuristic search; Hart brought in the notions of admissible and consistent heuristics. A*, which proceeds by the value f = g + h, finds the shortest path with an admissible heuristic and often expands far fewer vertices than Dijkstra's algorithm.

IN THE COURSE **Chapter 6 · The A* Algorithm** **Chapter 6 · Heuristic Functions**

A* algorithm admissible heuristic

YD

Yefim Dinitz

living · Sovyet-Israeli · Ben-Gurion University

Dinic's algorithm · 1970

In 1969 in Moscow he designed the maximum flow algorithm that proceeds by finding blocking flows in a level graph, and published it in 1970. In the West it became known as “Dinic's algorithm” through Shimon Even's presentation. The algorithm runs in $O(V^2 \cdot E)$ time; on unit-capacity networks and in bipartite matching it is much faster.

IN THE COURSE **Chapter 8 · Dinic's Algorithm** **Chapter 8 · Complexity: $O(V^2 \cdot E)$**

level graph blocking flow

MP

James H. Morris ve Vaughan Pratt

b. 1941 · b. 1944 · American · Berkeley, Stanford (Pratt born in Australia)

Knuth-Morris-Pratt · 1970/1977

In 1970 Morris found a matching method that never moves backward in the text by exploiting the pattern's own prefixes, and published it with Pratt in a technical report; a few weeks later Knuth reached the same algorithm independently, from automata theory. The three published the paper “Fast Pattern Matching in Strings” together in 1977. Thanks to the LPS table, the text pointer never moves backward and the search takes $O(n + m)$ time.

IN THE COURSE **Chapter 9 · Knuth-Morris-Pratt** **Chapter 9 · The LPS Table**

KMP LPS table prefix

RT

Robert E. Tarjan

b. 1948 · American · Princeton

Strongly connected components · 1972

In 1972 he published the algorithm that finds the strongly connected components of a directed graph in linear time with a single DFS. In 1975 he proved that the union-find structure with path compression and union by rank takes nearly constant amortized time, and made amortized analysis systematic. With Sleator he developed splay trees and the analysis of online list update (1985), and with Goldberg the push–relabel method (1988); he shared the 1986 Turing Award with Hopcroft.

IN THE COURSE **Chapter 5 · Tarjan: Principles and Steps**

Chapter 7 · Union-Find (Disjoint Sets) **Chapter 2 · Amortized Analysis**

Chapter 8 · Push–Relabel

DFS union-find amortized analysis

RK

Richard M. Karp

b. 1935 · American · Berkeley

21 NP-complete problems · 1972

In 1972, starting from SAT, he showed through polynomial-time reductions that 21 problems, among them clique, vertex cover, Hamiltonian cycle and knapsack, are NP-complete, revealing how widespread NP-completeness is. He developed the Edmonds-Karp algorithm with Edmonds (1972), the bipartite matching algorithm with Hopcroft (1973), and the Rabin-Karp string matching algorithm with Rabin (1987). He received the Turing Award in 1985.

IN THE COURSE **Chapter 8 · The Edmonds-Karp Algorithm** **Chapter 9 · Rabin-Karp**

NP-complete reduction Rabin-Karp

JH

John Hopcroft

b. 1939 · American · Cornell

Hopcroft–Karp matching · 1973

With Tarjan he showed how powerful depth-first search (DFS) is in graph algorithms; together they solved planarity testing in linear time. In 1973, with Karp, he published the algorithm that finds a maximum matching in a bipartite graph in $O(E\sqrt{V})$ time. The book on automata and formal languages he wrote with Ullman was a standard reference for many years; he shared the 1986 Turing Award with Tarjan.

IN THE COURSE **Chapter 5 · DFS: Analysis and Uses** **Chapter 8 · Applications**

DFS bipartite matching

AC

Alfred V. Aho ve Margaret J. Corasick

Aho: b. 1941 · Bell Labs

Aho–Corasick algorithm · 1975

In their 1975 paper “Efficient String Matching: An Aid to Bibliographic Search,” published in Communications of the ACM, they gave the algorithm that searches for many patterns in a text in a single pass. The patterns are placed in a trie; each node gets a failure link pointing to the

longest proper suffix to fall back to when a match breaks. The search time is linear in the text length, the total length of the patterns and the number of matches found. The algorithm is the basis of Unix's fgrep command. Aho is also among the authors of the AWK language and of the compiler books known as the “Dragon Book”; he shared the 2020 Turing Award with Jeffrey Ullman.

IN THE COURSE **Chapter 11 · Aho–Corasick** **Chapter 11 · The Trie Family**

multi-pattern search failure link trie

BM Robert S. Boyer ve J Strother Moore

Boyer: b. 1946 · American · University of Texas at Austin

Boyer-Moore · 1977

In 1977 they published the algorithm that compares the pattern from right to left and makes large jumps in the text with the bad character and good suffix rules; with long patterns most text characters are never examined. In 1980 Horspool gave a simpler version that uses only the bad character rule. The two also developed the Boyer–Moore theorem prover (an ancestor of ACL2) and the majority vote algorithm.

IN THE COURSE **Chapter 9 · Boyer-Moore** **Chapter 9 · Bad-Character Rule**

Chapter 9 · Horspool's Simplification

bad character rule good suffix rule

LZ Abraham Lempel ve Jacob Ziv

1936–2023 · 1931–2023 · Israeli · Technion

LZ77 · LZ78

At the Technion they published LZ77 in 1977, which refers back to earlier repetitions within a sliding window, and LZ78 in 1978, which collects the phrases seen so far in a dictionary. In 1984 Terry Welch simplified LZ78 to obtain LZW; the GIF format uses LZW, while DEFLATE, used by ZIP and PNG, combines LZ77 with Huffman coding. These methods are the basis of “universal” compression, which needs no probability table in advance.

IN THE COURSE **Chapter 9 · Lempel-Ziv-Welch (LZW)**

Chapter 9 · LZ78: The Ancestor of LZW

dictionary-based compression LZW

SK

S. Rao Kosaraju

living · Indian-born American · Johns Hopkins

Kosaraju's algorithm · 1978

In 1978 he proposed the method that finds the strongly connected components of a directed graph with two DFS passes but did not publish it; Micha Sharir found the same algorithm independently and published it in 1981. The first DFS produces the finishing order of the vertices. The second DFS runs on the graph with its edges reversed and starts its searches in decreasing order of finishing times; each tree of this DFS is a strongly connected component. With adjacency lists the time is O(V + E). It is easier to understand than Tarjan's single-DFS method, but traverses the graph twice.

IN THE COURSE **Chapter 5 · Complexity and Kosaraju** **Chapter 5 · Kosaraju: Code Trace**

strongly connected components two DFS reversed graph

JB

Jon Bentley

b. 1953 · American · Carnegie Mellon, Bell Labs

Divide-and-conquer recurrences · 1980

With Haken and Saxe he published in 1980 a general method for solving divide-and-conquer recurrences of the form T(n) = aT(n/b) + f(n); the master theorem became widespread through this work. With Yao he gave in 1976 a nearly optimal method for unbounded search, which is the basis of exponential search. He introduced the k-d tree in 1975 and brought efficient programming to a wide audience with his Programming Pearls columns.

IN THE COURSE **Chapter 2 · Divide-and-Conquer Recurrence**

Chapter 4 · Exponential Search: The Idea

master theorem exponential search

MM

Udi Manber ve Gene Myers

living · b. 1953 · American · University of Arizona

Suffix array · 1990

In 1990 (journal version 1993) they introduced the suffix array, which uses far less memory than the suffix tree, and gave the method that sorts the suffixes in O(n log n) time by doubling the prefix length in each round. Myers later contributed to the development of BLAST and to the

sequencing of the human genome; Manber worked on search technologies at Yahoo!, Amazon and Google.

IN THE COURSE **Chapter 10 · Suffix Array** **Chapter 10 · Prefix Doubling**

suffix array prefix doubling

BW Michael Burrows ve David Wheeler

b. 1963 · 1927–2004 · British · Cambridge; DEC SRC, USA

Burrows–Wheeler transform · 1994

Burrows and Wheeler published the transform Wheeler had found in 1983 as research report 124 of the DEC Systems Research Center in 1994. The last column of the sorted rotations of a string gathers similar characters together and is reversible; the bzip2 compressor and FM-index-based DNA alignment tools rely on this transform. Earlier, at Cambridge, Wheeler had developed one of the first forms of the subroutine call for the EDSAC.

IN THE COURSE **Chapter 10 · Burrows–Wheeler Transform**

Chapter 10 · Inverting the BWT

BWT compression

Dates and contributions are based on the original publications and encyclopedic sources. “Living” marks people who are alive and whose year of birth is not given in published sources. For people whose years of life could not be found in reliable sources, only the year of the contribution and the journal of publication are given.