

COURSE TEXTBOOK

Advanced Algorithms

Sercan Külcü

Giresun University · Department of Computer Engineering

Advanced 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/advanced-algorithms/`

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

© Sercan Külcü. All rights reserved.

Generated from the notes of 22 November 2026.

Preface

This book is made up of the notes for the Advanced Algorithms course taught in the Department of Computer Engineering at Giresun University, and it continues the *Algorithms* book. Algorithm complexity, sorting, searching, graph and string algorithms are covered there and are assumed here. The book studies the main methods of algorithm design, ways of coping with hard problems, and models of computation beyond the sequential, deterministic one. The chapters follow the order of the course; the examples are written in Java.

The book has three parts. In order, they are **Design Paradigms** (Chapters 1–3), **Hard Problems** (Chapters 4–6), and **Other Models of Computation** (Chapters 7–9).

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. Traces show, step by step, how an algorithm builds its solution. 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/advanced-algorithms-java-en`, and the programs use the same names as the listings in this book. Slides, study questions and simulators are on the course page.

Sercan Külcü
Giresun

Contents

Course Map 1

PART I · DESIGN PARADIGMS

1 Divide and Conquer 4

1.1 The divide-and-conquer paradigm 5
1.2 Recurrence relations 11
1.3 Basic examples 15
1.4 Convex hull and Quickhull 21
1.5 The closest pair of points 32
1.6 Karatsuba multiplication 40
1.7 Strassen matrix multiplication 44
1.8 Assessment 47
1.9 Common mistakes 48
1.10 Summary 48
1.11 Key terms 49
1.12 Test Yourself 49
1.13 Further reading 51
1.14 References 52

2 Greedy Algorithms 53

2.1 The greedy approach 54
2.2 The coin change problem 59
2.3 Activity selection 65
2.4 Job sequencing with deadlines 71
2.5 The fractional knapsack 75
2.6 The greedy choice in the minimum spanning tree 80
2.7 Huffman coding 83
2.8 Other greedy algorithms 93
2.9 Comparison and common mistakes 94
2.10 Summary 95
2.11 Key terms 96
2.12 Test Yourself 96
2.13 Further reading 99
2.14 References 99

3 Dynamic Programming 100

3.1 Basic concepts 101
3.2 Fibonacci: memoization and tabulation 107
3.3 0/1 knapsack 112
3.4 Unbounded knapsack 122
3.5 Subset sum 126

3.6 Longest increasing subsequence 130
3.7 Longest common substring 138
3.8 Longest common subsequence 141
3.9 Other classic DP problems 145
3.10 Common mistakes 151
3.11 Summary 152
3.12 Key terms 153
3.13 Test Yourself 153
3.14 Further reading 155
3.15 References 156

PART II · HARD PROBLEMS

4 Backtracking 158

4.1 Hard problems: P, NP, and NP-hard 159
4.2 The basic idea of backtracking 162
4.3 The n-queens problem 174
4.4 The min-conflicts heuristic 188
4.5 Sudoku 193
4.6 Graph coloring 200
4.7 Hamiltonian path 206
4.8 Word search 211
4.9 Other applications 215
4.10 Analysis 217
4.11 Common mistakes 219
4.12 Summary 220
4.13 Key terms 221
4.14 Test Yourself 221
4.15 Further reading 224
4.16 References 224

5 Branch and Bound 225

5.1 The idea of branch and bound 226
5.2 The paradigm 229
5.3 The 0/1 knapsack 236
5.4 The traveling salesman problem 247
5.5 The assignment problem 258
5.6 Integer linear programming 261
5.7 Maximum clique 265
5.8 Other problems and the relation to A* 268
5.9 Comparing the design paradigms 270
5.10 Common mistakes 271
5.11 Summary 271
5.12 Key terms 272
5.13 Test Yourself 272
5.14 Further reading 275

5.15 References 275

6 Approximation Algorithms 276

6.1 Approximation algorithms 277
6.2 The traveling salesman problem 282
6.3 Local search: 2-opt 293
6.4 Job scheduling 301
6.5 Vertex cover 307
6.6 Knapsack 315
6.7 Power iteration 318
6.8 Comparison and common mistakes 324
6.9 Summary 325
6.10 Key terms 325
6.11 Test Yourself 326
6.12 Further reading 328
6.13 References 328

PART III · OTHER MODELS OF COMPUTATION

7 Randomized Algorithms 330

7.1 Randomness and algorithms 331
7.2 Las Vegas and Monte Carlo 335
7.3 Shuffling 339
7.4 BogoSort 343
7.5 Randomized QuickSort 347
7.6 Randomized selection (QuickSelect) 355
7.7 Estimating π by Monte Carlo 357
7.8 The Miller–Rabin primality test 361
7.9 Karger's minimum cut algorithm 367
7.10 Other applications 373
7.11 Pseudorandom numbers 374
7.12 Common mistakes 376
7.13 Summary 377
7.14 Key terms 378
7.15 Test Yourself 378
7.16 Further reading 380
7.17 References 381

8 Online Algorithms 382

8.1 The online problem model 383
8.2 Competitive analysis 387
8.3 Ski rental 389
8.4 Paging 396
8.5 List update 413
8.6 Data streams 418
8.7 Incremental learning 427

8.8 Problems and results 429
8.9 Common mistakes 430
8.10 Summary 431
8.11 Key terms 431
8.12 Test Yourself 432
8.13 Further reading 434
8.14 References 434

9 Parallel Algorithms 435

9.1 The idea of a parallel algorithm 436
9.2 Models of parallel computation 439
9.3 Performance measures 443
9.4 Summation on a PRAM 449
9.5 Parallel prefix sums 455
9.6 The pipeline 460
9.7 The pipelined tree 465
9.8 Kinds of parallel algorithms and Java 468
9.9 Shared state and races 476
9.10 Common mistakes 481
9.11 Summary 482
9.12 Key terms 483
9.13 Test Yourself 483
9.14 Further reading 485
9.15 References 486

APPENDICES

A Glossary of Terms 487

B Prerequisites for Advanced Algorithms 492

C Important People in Advanced Algorithms 507

ADVANCED 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.

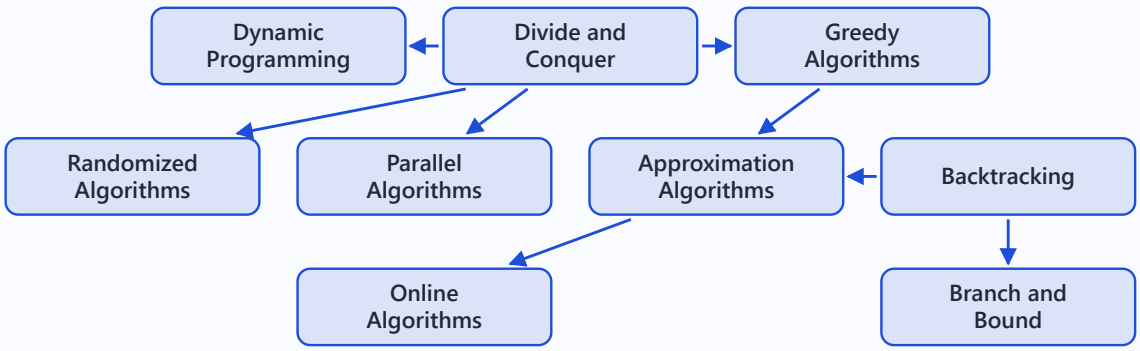


Figure 1: The course map. Each arrow points from a topic to one that builds on it. The top row holds the design paradigms, the right side the hard problems, and the left side the other models of computation.

Figure 1 draws only the most direct connections. Table 1 lists every chapter that each chapter depends on, together with the idea that links them.

Topic (chapter)	Builds on	Connection
Divide and Conquer (1)	—	The common template of merge sort and quicksort; recurrence relations and the Master Theorem (Algorithms, Chapters 2–3).
Greedy Algorithms (2)	Divide and Conquer	Take the choice that looks best at each step; Prim, Kruskal, and Dijkstra are greedy graph algorithms (Algorithms, Chapters 6–7).
Dynamic Programming (3)	Divide and Conquer	Divide and conquer with overlapping subproblems: memoization and tabulation; LCS and edit distance on strings (Algorithms, Chapter 10).
Backtracking (4)	—	Depth-first search of the state-space tree (Algorithms, Chapter 5); an introduction to P, NP, and NP-hard problems.
Branch and Bound (5)	Backtracking	Backtracking gains a bound: branches that cannot contain the best solution are pruned.
Approximation Algorithms (6)	Greedy Algorithms, Backtracking	Solutions with a proven ratio for NP-hard problems: the MST-based 2-approximation, LPT, vertex cover.
Randomized Algorithms (7)	Divide and Conquer	Quicksort with a random pivot and QuickSelect; the Las Vegas and Monte Carlo types.

Topic (chapter)	Builds on	Connection
Online Algorithms (8)	Greedy Algorithms, Approximation Algorithms	Deciding without knowing the future: the competitive ratio is the online counterpart of the approximation ratio; paging.
Parallel Algorithms (9)	Divide and Conquer	Independent subproblems are solved in parallel: prefix sums, parallel merge sort, work and depth.

Table 1: Connections on the map.

The backbone of the map can be read as **design paradigms → hard problems → other models**. The course builds on the analysis, sorting, searching, graph, and string algorithms of the Algorithms course. Divide and conquer grows out of sorting and opens onto dynamic programming and onto randomized and parallel algorithms. The greedy method is familiar from graph algorithms. Backtracking and branch and bound search for the best solution in exponential time; approximation algorithms settle instead for a proven ratio. Online algorithms apply the same idea of a ratio to an unknown future.

PART I

Design Paradigms

Chapters 1–3

PART I · CHAPTER 1

Divide and Conquer

Solving a large problem head-on is usually hard. Half of the same problem is easier, and a quarter is easier still. At some point the problem becomes so small that its solution is obvious. Divide and conquer builds on this observation: split the problem into smaller pieces of the same kind, solve the pieces by the same method, and combine their solutions to answer the original question. Several algorithms you already know from sorting and searching fit this pattern. These notes first present the steps of the approach, its general template, and the way a recurrence relation gives the running time. They then take a brief paradigm-level look at the familiar algorithms. Most of the weight falls on four new algorithms: Quickhull for the convex hull, the strip method for the closest pair of points, Karatsuba integer multiplication, and Strassen matrix multiplication. For each algorithm the notes give code, a step-by-step trace, and a cost analysis.

LEARNING OUTCOMES

Paradigm
Describe the steps of divide and conquer, and decide whether a problem suits the approach, in particular by checking whether its subproblems are independent or overlapping.

Analysis
Set up the recurrence of a divide-and-conquer algorithm; solve it with the master theorem or a recursion tree; recognize the cases where the master theorem does not apply.

Geometric algorithms
Find the convex hull with Quickhull and the closest pair of points with the strip method, step by step; compute the orientation test with integer multiplication alone; justify why at most seven comparisons per strip point suffice.

Fast multiplication
Show how Karatsuba (integer) and Strassen (matrix) multiplication reduce the running time by removing one recursive call; run both algorithms by hand on small examples.

1.1 The divide-and-conquer paradigm

Designing an algorithm involves two things: the problem itself and the general strategy used to solve it. Patterns at the level of strategy are called **paradigms**. Brute force, the greedy approach, dynamic programming, and backtracking are such patterns. Divide and conquer is one of the oldest and most efficient of them. The name comes from a military maxim: if a large army cannot be defeated as a whole, split it into small units and defeat the units one at a time. In algorithms the "army" is the large problem and the "units" are its smaller copies.

1.1.1 Divide, break down, conquer, combine

Definition 1.1

Divide and conquer is an algorithm design approach that splits a problem into smaller, mutually independent subproblems of the same kind. The subproblems are solved recursively by the same method, and their solutions are combined to build the solution of the original problem. Splitting stops when a subproblem is small enough to be solved directly; this situation is the **base case**.

The approach is usually described in four steps. Most textbooks list only three (divide, conquer, combine). The four-step description additionally stresses that splitting is not done once but repeatedly, until the base case is reached:

1. **Divide:** The complex problem is split into smaller, manageable subproblems. For example, an array of n elements is split into two halves of $n/2$ elements.
2. **Break down:** The subproblems keep being split until they are small enough to solve directly. Recursion provides this repetition: the same function is applied to the subproblem, and that function splits its own input in turn.
3. **Conquer:** Each subproblem is solved separately. Base-case subproblems are solved directly; the others are solved by a recursive call.
4. **Combine:** The solutions of the subproblems are combined to build the solution of the original problem.

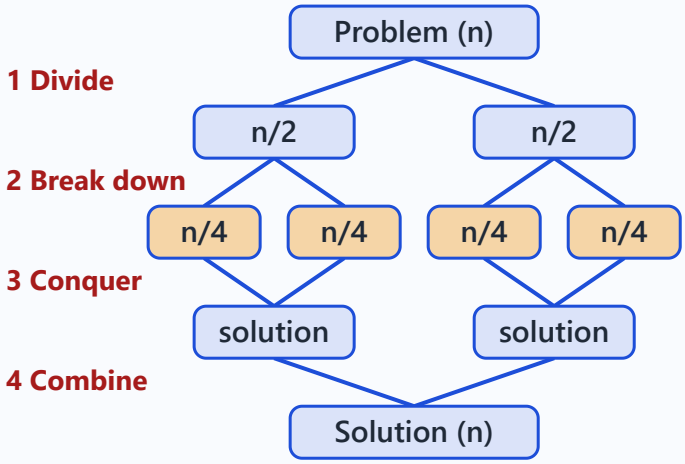


Figure 1.1: The general picture of divide and conquer. A problem of size n is split first into two subproblems of size $n/2$, then into four of size $n/4$. The smallest subproblems are solved directly. The solutions are combined from the bottom up to obtain the final answer.

Figure 1.1 shows this flow as a tree. The root of the tree is the original problem. Each internal node is a subproblem, and its children are the pieces it is split into. The leaves are base cases. The algorithm descends by splitting from the top and climbs back by combining from the bottom. In a program, the counterpart of this tree is the **recursion tree**: each node is a function call.

For the approach to work, the subproblems must have two properties. First, a subproblem is *of the same kind* as the original problem; it is merely smaller. Sorting an array is the same kind of task as sorting half of it. That is why the same function can call itself, and no separate "sort a half-array" function is needed. Second, the subproblems are *independent of one another*. Sorting the left half needs no information from sorting the right half. If this independence fails, that is, if the subproblems share common sub-subproblems, the same work is done over and over. The recursive Fibonacci computation is the best-known example.

1.1.2 Why divide and conquer?

At first glance divide and conquer looks like merely a way of organizing work: split the job into pieces and do the pieces one by one. The real strength of the approach, however, is that on suitable problems it lowers the running time by an order of magnitude. Almost all of the algorithms discussed in these notes are asymptotically faster than the naive method for the same problem. Examples: $O(n \log n)$ instead of $O(n^2)$ for sorting, $O(\log n)$ instead of $O(n)$ for searching a sorted array, and about $O(n^{1.585})$ instead of $O(n^2)$ for integer multiplication.

The approach has other benefits as well:

- **Clarity and reuse.** A complex problem is easier to understand once it is split into small, manageable pieces. The method for solving subproblems is the same function at every level, so the code stays short and is reused at every level.

- **Parallelism.** Because the subproblems are independent, they can be solved at the same time on different processor cores. Sorting the left and right halves do not wait for each other.
- **Memory locality.** As subproblems shrink, they fit in the cache. A small subarray is loaded into the cache once and all the work on it finishes there. On large data this gives a visible speedup.
- **Easy correctness proofs.** The recursive structure makes it natural to prove correctness by induction: if the base case is correct and the combine step builds a correct solution from correct subsolutions, the algorithm is correct.

The costs of the approach should also be known:

- **Recursion overhead.** Every call opens a frame on the call stack, which costs time and memory. Some algorithms also need extra memory; merge sort, for example, uses an $O(n)$ auxiliary array.
- **Overlapping subproblems.** If the subproblems are not independent, the same work is repeated and the running time can become exponential.
- **Constant factors on small inputs.** The asymptotic gain shows up at large n. At small n, the overhead of splitting, combining, and calling dominates. In practice a **cutoff** is therefore chosen, and once a subproblem falls below it, a simple method takes over. Karatsuba and Strassen implementations work this way.

Divide and conquer is used in many areas. Sorting and searching, computational geometry, and big-number and matrix arithmetic come first. The fast Fourier transform and game-tree search follow the same pattern.

1.1.3 The general template

All divide-and-conquer algorithms follow the same skeleton. The Java code below writes this skeleton independent of any particular problem. The types `Problem` and `Result` and the methods `split()`, `solveDirectly()`, and `combine()` take on a different meaning in each algorithm.

```
Result divideAndConquer(Problem p) {
  if (p.size() <= CUTOFF)
    return solveDirectly(p);        // base case
  Problem[] parts = split(p);       // a subproblems, size n/b
  Result[] sols = new Result[parts.length];
  for (int i = 0; i < parts.length; i++)
    sols[i] = divideAndConquer(parts[i]); // conquer: recursion
  return combine(sols);              // combine
}
```

The lines of the template match the steps of the paradigm one to one:

- `if (p.size() <= CUTOFF) return solveDirectly(p);` is the base case. If the problem is no larger than the cutoff, splitting stops and the problem is solved directly. The cutoff is often 1: an array of one element is already sorted, and the maximum of one element is the element itself. A

larger cutoff can also be chosen; in the closest pair of points it is 3.

- `Problem[] parts = split(p);` is the divide step. The problem is split into a subproblems, each of size about n/b.
- The `for` loop is the conquer step. For each subproblem the template itself is called. The breakdown step is not a separate line; it happens inside this recursive call, when the subproblem is split again.
- `return combine(sols);` is the combine step. The subsolutions are turned into the final solution.

The running time of the template can be read off the same structure. Let $T(n)$ be the time to solve a problem of size n. The algorithm solves a subproblems of size n/b recursively, and their total time is $a \cdot T(n/b)$. The time spent on splitting and combining is written $f(n)$. This gives

$$T(n) = a \cdot T(n/b) + f(n)$$

In this recurrence a is the number of subproblems, n/b is the subproblem size, and $f(n)$ is the cost of splitting and combining. The recurrence summarizes the cost of a divide-and-conquer algorithm in three numbers. The ways to speed an algorithm up can be read off these same three numbers:

- **Without a base case, the recursion never ends.** The recurrence holds only for large n; for small n, $T(n)$ is a constant. If the base case is forgotten, the program recurses forever and stops with a `StackOverflowError`.
- **The split must be balanced.** A split of the form $n/2 + n/2$ is good: the problem halves at every level, and the base case is reached after about $\log_2 n$ levels. A split of the form $1 + (n - 1)$ is bad: the problem shrinks by only one element per level, and n levels are needed.
- **The speedup most often comes from reducing a or shrinking $f(n)$.** The entire gain of Karatsuba and Strassen comes from reducing a by one. Lowering the combine cost from $O(n \log n)$ to $O(n)$ is likewise decisive in the closest pair of points.

1.1.4 Applying the template: the maximum element

To see the template on a concrete example, we choose a very simple problem: finding the maximum element of an array. A single loop also solves this problem, and divide and conquer does not shorten the running time here. The purpose of the example is to show what each part of the template looks like in code.

Divide and conquer sets the problem up as follows. The maximum of the array is the larger of the maximum of the left half and the maximum of the right half. This sentence reduces the problem to two smaller problems of the same kind. In a range of one element, the maximum is that element itself; this is the base case.

```
int findMax(int[] a, int lo, int hi) {
  if (lo == hi) return a[lo];     // base
  int mid = (lo + hi) / 2;
  int left = findMax(a, lo, mid);
  int right = findMax(a, mid + 1, hi);
  return Math.max(left, right);     // combine
}
```

The method works on the range `a[lo..hi]` of the array. For the whole array one calls `findMax(a, 0, n - 1)`. The lines do the following:

- `if (lo == hi) return a[lo];` is the base case. The range holds one element, and that element is the maximum.
- `int mid = (lo + hi) / 2;` is the divide step. The range is split in the middle: `[lo..mid]` and `[mid+1..hi]`. Because integer division rounds down, the left half gets one extra element when the number of elements is odd.
- The two recursive calls are the conquer step: `left` is the maximum of the left half and `right` is the maximum of the right half.
- `return Math.max(left, right);` is the combine step and is a single comparison.

Splitting creates no new array; it only computes two indices. Splitting and combining together therefore take constant time: $f(n) = O(1)$. The recurrence is $T(n) = 2T(n/2) + O(1)$, and its solution is $\Theta(n)$.

1.1.5 Code trace: findMax(a, 0, 4)

In the trace the array is `a = [5, 2, 9, 4, 7]` and the call is `findMax(a, 0, 4)`. The range holds five elements. The middle index is $(0 + 4)/2 = 2$, so the left half is `[0..2]` and the right half is `[3..4]`. The left half gets three elements and the right half two. Table 1.1 gives the order of the calls and the value each call returns.

Step	Event	Call stack	Return
1	`(0, 4)` : mid = 2, left half `[0..2]` first	(0,4)	—
2	`(0, 2)` : mid = 1, first `[0..1]`, then `[2..2]`	(0,4), (0,2)	—
3	`(0, 1)` : mid = 0, first `[0..0]`, then `[1..1]`	(0,4), (0,2), (0,1)	—
4	`(0, 0)` : single element, base case	..., (0,1), (0,0)	5
5	`(1, 1)` : single element, base case	..., (0,1), (1,1)	2
6	`(0, 1)` : left = 5, right = 2 → max	(0,4), (0,2), (0,1)	5
7	`(2, 2)` : single element, base case	(0,4), (0,2), (2,2)	9
8	`(0, 2)` : left = 5, right = 9 → max	(0,4), (0,2)	9
9	`(3, 4)` : mid = 3, first `[3..3]`, then `[4..4]`	(0,4), (3,4)	—

Step	Event	Call stack	Return
10	`(3, 3)` : single element, base case	(0,4), (3,4), (3,3)	4
11	`(4, 4)` : single element, base case	(0,4), (3,4), (4,4)	7
12	`(3, 4)` : left = 4, right = 7 → max	(0,4), (3,4)	7
13	`(0, 4)` : left = 9, right = 7 → max	(0,4)	9

Table 1.1: Trace of `findMax(a, 0, 4)` with `a = [5, 2, 9, 4, 7]`. Calls are shown as `(lo, hi)` pairs; the call stack grows from left to right.

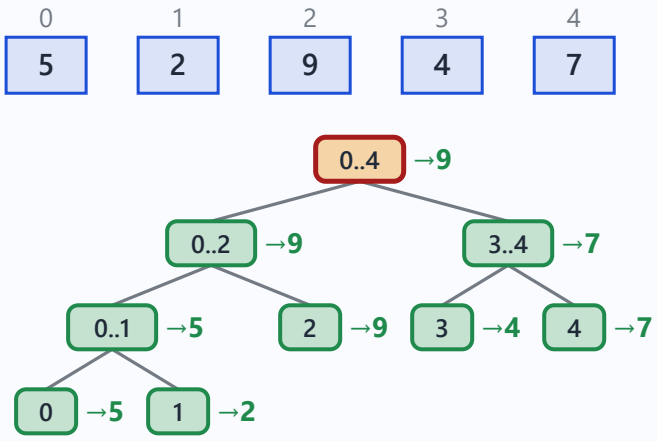


Figure 1.2: The recursion tree of the call `findMax(a, 0, 4)`. Each node is a `lo..hi` range; the number beside the arrow is the value that call returns. The five leaves are base cases; each of the four internal nodes makes one comparison.

The order of the calls in the trace deserves attention. A call does not move on to the right half until the work on the left half is completely finished. At step 3 the call `(0, 1)` starts; at steps 4 and 5 it solves two leaves, and only at step 6 does it return its result. The right half `(3, 4)` does not start at all until step 9. This order is a *depth-first* traversal of the recursion tree. The call stack never holds more frames than the tree has levels; here there are at most four frames (`(0,4)`, `(0,2)`, `(0,1)`, and one leaf).

The result is 9. The total number of calls is 9: 5 leaves and 4 internal nodes. The number of comparisons equals the number of internal nodes, that is, 4. In general, for an array of n elements the recursion tree has n leaves. Since every internal node has exactly two children, the number of internal nodes is $n - 1$. The algorithm makes $n - 1$ comparisons and $2n - 1$ calls.

A simple loop also finds the maximum with $n - 1$ comparisons by scanning the array from start to end. Divide and conquer has not reduced the number of comparisons, and it has even added call overhead. What, then, is the benefit of this version? The answer is parallelism. The two halves are independent and can be solved at the same time. With an unlimited number of processors, each level of the tree finishes in one step. Since the height of the tree is $\lceil \log_2 n \rceil$, the result is found in $O(\log n)$ parallel steps. In the loop version each comparison waits for the result of the previous one, and this chain cannot be shortened.

Common mistake

Believing that divide and conquer shortens the running time on every problem. In the maximum-element problem both halves must be examined in full; no element can be skipped. The total work therefore cannot drop below n. The gain appears in three situations. One of the subproblems can be discarded (binary search). The number of subproblems can be reduced (Karatsuba, Strassen). Or the combine step can be made cheaper than the naive method (merge sort, closest pair of points).

1.1.6 Correctness and induction

The correctness of a divide-and-conquer algorithm is proved by an induction that follows the structure of the recursion. The induction is on the problem size. First one shows that the algorithm is correct in the base case. Then one assumes it is correct for all smaller inputs and shows that the combine step produces the right result. For `findMax` this goes as follows.

Theorem 1.1 (findMax is correct)

For every range with $lo \le hi$, `findMax(a, lo, hi)` returns the maximum element of `a[lo..hi]`.

Proof. The induction is on the number of elements in the range, $k = hi - lo + 1$. If $k = 1$, then $lo = hi$ and the method returns `a[lo]`, which is the maximum of a one-element range. Now let $k \ge 2$ and assume the method is correct on every range with fewer than k elements. For `mid = ⌊(lo + hi)/2⌋` we have $lo \le mid < hi$. Both halves are therefore nonempty, and each contains fewer than k elements. By the induction hypothesis, `left` is the maximum of the left half and `right` is the maximum of the right half. Every element of the range lies in one of the two halves; hence the maximum of the range is `max(left, right)`. ∎

The inequality $lo \le mid < hi$ used in the proof is important. If the split leaves one of the halves empty, or leaves a half as large as the whole range, the induction hypothesis cannot be applied and the program can recurse forever. For example, if `mid` were rounded up, with the left half `[lo..mid]` and the right half `[mid+1..hi]`, then on a two-element range `mid = hi` and the right half would be empty. The same question must be asked of every divide-and-conquer algorithm: is each subproblem really smaller, and how is an empty subproblem handled?

1.2 Recurrence relations

The running time of a recursive algorithm is written in terms of itself: the time for a problem of size n is expressed through the times of smaller problems and the cost of splitting and combining. An equation of this kind is called a **recurrence relation**. This section briefly recalls what divide and

conquer needs from the setting up and solving of recurrences with the master theorem.

1.2.1 Setting up the recurrence

A recurrence is read off the code of the algorithm by asking three questions:

1. How many recursive calls are made? This number is a. Only calls that actually run are counted. Binary search contains two separate call lines, but only one of them runs in each execution; hence $a = 1$.
2. How large is the input of each call? If the input is n/b, read off b. If the input shrinks by a constant amount, such as $n - 1$, the recurrence does not fit the form of the master theorem.
3. How much work is done outside the calls? The total cost of splitting and combining is $f(n)$. Loops, array copying, and building sublists belong to this cost.

Example 1.1

The recurrences of the following algorithms are written down by looking at their code.

- `findMax` : two calls, each on a half range; one comparison outside: $T(n) = 2T(n/2) + O(1)$.
- Merge sort: two calls, each on a half array; outside, a linear loop that merges the two halves: $T(n) = 2T(n/2) + O(n)$.
- Binary search: one call (left or right half), a constant number of comparisons: $T(n) = T(n/2) + O(1)$.
- Fast exponentiation: `power(x, n)` calls `power(x, n/2)` once and performs at most two multiplications: $T(n) = T(n/2) + O(1)$. Here n is the value of the exponent, not the length of the input.
- Naive recursive Fibonacci: two calls, but on inputs $n - 1$ and $n - 2$: $T(n) = T(n - 1) + T(n - 2) + O(1)$. This recurrence is not in the form of the master theorem.

1.2.2 The master theorem

The master theorem solves recurrences of the form $T(n) = aT(n/b) + f(n)$ without computation. It compares $f(n)$, which stands for the work at the root, with $n^{\log_b a}$, which stands for the total work at the leaves. The second value is called the leaf work, and its exponent $\log_b a$ is the **critical exponent**: the recursion tree has $\log_b n$ levels, and the number of nodes multiplies by a at each level; the number of leaves is $a^{\log_b n} = n^{\log_b a}$.

Theorem 1.2 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n) = aT(n/b) + f(n)$. With $c = \log_b a$:

1. If $f(n) = O(n^{c-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^c)$. (Case 1: the leaves dominate.)
2. If $f(n) = \Theta(n^c)$, then $T(n) = \Theta(n^c \log n)$. (Case 2: every level contributes equally.)
3. If $f(n) = \Omega(n^{c+\varepsilon})$ for some $\varepsilon > 0$, and the regularity condition $a \cdot f(n/b) \leq k \cdot f(n)$ holds for some constant $k < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. (Case 3: the root dominates.)

The language of the theorem looks heavy at first, but using it is short: compare $f(n)$ with $n^{\log_b a}$; the larger one wins; if the two are of the same order, a factor of $\log n$ is added to the result. "Polynomially smaller" means that the gap is at least a power such as n^{ε}. A gap of only a $\log n$ factor does not count as a polynomial gap.

Example 1.2

Three recurrences are solved with the master theorem.

1. $T(n) = 4T(n/2) + n$: $a = 4$, $b = 2$, $n^{\log_2 4} = n^2$. $f(n) = n$ is polynomially smaller than n^2 ($\varepsilon = 1$). Case 1: $T(n) = \Theta(n^2)$.
2. $T(n) = 2T(n/2) + n^2$: $n^{\log_2 2} = n$. $f(n) = n^2$ is polynomially larger. Regularity condition: $2 \cdot (n/2)^2 = n^2/2 \leq \tfrac{1}{2} \cdot n^2$; it holds with $k = \tfrac{1}{2}$. Case 3: $T(n) = \Theta(n^2)$.
3. $T(n) = 2T(n/2) + 1$ (`findMax`): $n^{\log_2 2} = n$, $f(n) = 1 = n^0$. The gap is n^1. Case 1: $T(n) = \Theta(n)$. This agrees with the count of $n - 1$ comparisons and $2n - 1$ calls.

1.2.3 Level sums in the recursion tree

The three cases of the master theorem become clear on their own once one looks at the total work on each level of the recursion tree. At level i there are a^i subproblems, each of size n/b^i. The total work done at this level is $a^i \cdot f(n/b^i)$. The total running time of the algorithm is the sum over all levels. From the root to the leaves, the level sums either grow, stay constant, or shrink. Table 1.2 gives the level sums of five recurrences for $n = 16$.

Recurrence	0	1	2	3	4	Total	Trend
$2T(n/2) + n$	16	16	16	16	16	80	constant (case 2)
$3T(n/2) + n$	16	24	36	54	81	211	×1.5 (case 1)
$4T(n/2) + n$	16	32	64	128	256	496	×2 (case 1)
$7T(n/2) + n^2$	256	448	784	1372	2401	5261	×1.75 (case 1)

Recurrence	0	1	2	3	4	Total	Trend
$2T(n/2) + n^2$	256	128	64	32	16	496	×0.5 (case 3)

Table 1.2: Level sums $a^i \cdot f(16/2^i)$ of the recursion tree for $n = 16$. Level 0 is the root and level 4 the leaves.

The table shows the three behaviors side by side. In the first row every level does the same work: 16. Since there are $\log_2 n + 1$ levels, the total is of order $n \log n$; this is case 2 and the recurrence of merge sort. In the second, third, and fourth rows the level sums grow geometrically. In a geometric series the total is a constant multiple of the last term, and the last term is the leaf level. For $3T(n/2) + n$ the leaf level is $3^4 = 81 = 16^{\log_2 3}$. This is the Karatsuba recurrence, and its running time is $\Theta(n^{\log_2 3})$. In the last row the level sums halve. This time the total is a constant multiple of the first term, that is, of the root: $\Theta(n^2)$.

This view also explains where the speedup in divide and conquer comes from. The only difference between $4T(n/2) + n$ and $3T(n/2) + n$ is that one subproblem is missing at each level. But this difference compounds at every level: at $n = 16$ the leaf level has 81 units of work instead of 256. As n grows, the ratio grows like $(4/3)^{\log_2 n} = n^{2 - 1.585}$.

1.2.4 The algorithms of this chapter and the master theorem

Table 1.3 solves the recurrences of the algorithms discussed in these notes with the master theorem. What the calculations have in common is that a, b, and $f(n)$ are read off the code and the critical exponent is compared with the exponent of $f(n)$.

Algorithm	a	b	f(n)	$n^{\log_b a}$	Case	T(n)
Binary search, fast exponentiation	1	2	1	$n^0 = 1$	2	$\Theta(\log n)$
Merge sort, closest pair, balanced quicksort	2	2	n	n	2	$\Theta(n \log n)$
Maximum element	2	2	1	n	1	$\Theta(n)$
Karatsuba	3	2	n	$n^{1.585}$	1	$\Theta(n^{1.585})$
Naive block matrix multiplication	8	2	n^2	n^3	1	$\Theta(n^3)$
Strassen	7	2	n^2	$n^{2.807}$	1	$\Theta(n^{2.807})$

Table 1.3: Recurrences of divide-and-conquer algorithms and their solutions by the master theorem.

The last three rows of the table carry a common lesson. The entire gain of Karatsuba and Strassen comes from reducing a by one: from 4 to 3 and from 8 to 7. The cost of splitting and combining, $f(n)$, does not shrink; it even grows slightly. In case 1 the running time depends only on the critical exponent, which is $\log_b a$. When a gets smaller, the exponent gets smaller too: $\log_2 4 = 2$ but $\log_2 3 \approx 1.585$; $\log_2 8 = 3$ but $\log_2 7 \approx 2.807$.

1.2.5 Recurrences the master theorem cannot handle

The master theorem is powerful but fits a narrow pattern: all subproblems must have the same size, and the size must shrink by a *ratio*. Some divide-and-conquer algorithms do not fit this pattern:

- **Shrinking by a constant amount.** The recurrence $T(n) = T(n - 1) + n$ has no b. Unbalanced quicksort and worst-case Quickhull behave this way. Unrolling the recurrence gives $n + (n - 1) + \ldots + 1 = n(n + 1)/2 = \Theta(n^2)$.
- **Unequal splits.** The recurrence of Quickhull is $T(n) = T(n_1) + T(n_2) + O(n)$, where n_1 and n_2 depend on the input. Splits in a fixed but unequal ratio, such as $T(n) = T(n/3) + T(2n/3) + n$, also fall outside the theorem. These are solved with a recursion tree: in the last example every level sums to at most n and the longest branch has $\log_{3/2} n$ levels; the result is $\Theta(n \log n)$.
- ***f(n)* that falls between the cases.** In $T(n) = 2T(n/2) + n/\log n$, $f(n) = n/\log n$ is smaller than n but not polynomially smaller: the gap is only a factor of $\log n$. No case of the theorem applies. Similarly, the recurrence $2T(n/2) + n \log n$ of the flawed version of the closest pair of points lies outside the three cases of the master theorem; its solution is $\Theta(n \log^2 n)$.

> **Common mistake**
>
> Forcing the master theorem onto a recurrence that does not fit the pattern. For $T(n) = T(n - 1) + n$, saying "$a = 1$, $f(n) = n$, case 3, result $\Theta(n)$" is wrong; the correct result is $\Theta(n^2)$. Before applying the theorem, check that the recurrence has the form $aT(n/b) + f(n)$, that $b > 1$, and that $f(n)$ falls into one of the three cases.

1.3 Basic examples

Merge sort, quicksort, and binary search are familiar algorithms. This section does not explain them again; it merely sets them side by side as three different faces of the same paradigm. Two new examples follow: fast exponentiation, which brings exponentiation down to logarithmic time, and the recursive Fibonacci computation, where divide and conquer gives a *bad* result.

1.3.1 Classic divide-and-conquer algorithms

Table 1.4 collects the divide and combine steps and the recurrences of the algorithms named in these notes in a single table. It is instructive to read the table column by column rather than row by row. In the "Combine" column two algorithms have "none": in binary search and quicksort the work is either done at splitting time or the result comes directly from a single subproblem.

Algorithm	Divide	Combine	Recurrence	Time
Binary search	inspect the middle element, discard one half	none	$T(n/2) + O(1)$	$O(\log n)$

Algorithm	Divide	Combine	Recurrence	Time
Merge sort	split the array in the middle	merge the sorted halves	$2T(n/2) + O(n)$	$O(n \log n)$
Quicksort	partition around a pivot	none (in place)	$T(k) + T(n-k-1) + O(n)$	avg. $O(n \log n)$
Fast exponentiation	$x^n = (x^{\lfloor n/2 \rfloor})^2 \cdot (x)$	square the result	$T(n/2) + O(1)$	$O(\log n)$
Quickhull	build a triangle with the farthest point	add the hull pieces	$T(n_1) + T(n_2) + O(n)$	avg. $O(n \log n)$
Closest pair of points	split in the middle by x	scan the middle strip	$2T(n/2) + O(n)$	$O(n \log n)$
Karatsuba	split the number into two halves	add up 3 products	$3T(n/2) + O(n)$	$O(n^{1.585})$
Strassen	split the matrix into 4 blocks	add up 7 products	$7T(n/2) + O(n^2)$	$O(n^{2.807})$

Table 1.4: Classic divide-and-conquer algorithms.

1.3.2 Merge sort: the work is in the merge

Merge sort is the plainest example of divide and conquer. The **divide** step splits the array in the middle and takes $O(1)$, since it computes only a middle index. In the **conquer** step the two halves are sorted recursively. In the **combine** step the two sorted halves are merged in a single pass, at each step taking the smaller of the two front elements; this step takes $\Theta(n)$.

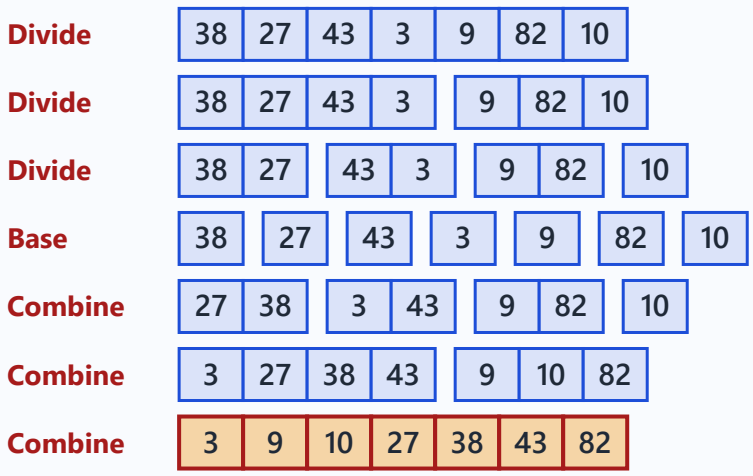


Figure 1.3: Merge sort of the array `[38, 27, 43, 3, 9, 82, 10]`. The top three rows show the splitting, the "Base" row the single-element arrays, and the bottom three rows the merges. The last row is the sorted array: `[3, 9, 10, 27, 38, 43, 82]`.

Figure 1.3 shows the sorting of a seven-element array level by level. First, with $mid = (0 + 6)/2 = 3$, the array is split into the halves `[38 27 43 3]` and `[9 82 10]`. The halves are split again: `[38 27]`, `[43 3]`, `[9 82]`, `[10]`. One-element arrays are already sorted and are the base case. Merging works from the bottom up. First the pairs become `[27 38]`, `[3 43]`, `[9 82]`. Then `[3 27 38 43]`

and `[9 10 82]` are obtained; finally these two give `[3 9 10 27 38 43 82]`. Each merge takes as many steps as the sum of the lengths of the two halves. All the merges on one level together touch at most n elements. The number of levels is $\lceil \log_2 7 \rceil = 3$.

The recurrence is $T(n) = 2T(n/2) + \Theta(n)$ and falls under case 2 of the master theorem: $\Theta(n \log n)$. Because the split is always made in the middle without looking at the input, this running time is the same in the best, average, and worst case. The most important observation from the paradigm's point of view is this: **the real work is done in the merge**. All the comparisons are in the merge loop; the split looks at no element.

1.3.3 Quicksort: the work is in the partition

Quicksort solves the same problem from the opposite direction. The **divide** step picks a pivot element and separates the array into the elements smaller and larger than the pivot. This operation is called **partitioning** and takes $\Theta(n)$. In the **conquer** step the two parts are sorted recursively. There is no **combine** step: because the parts are sorted in place, the two sorted parts already sit side by side in the right order. Unlike merge sort, **the real work is done in the partition**.

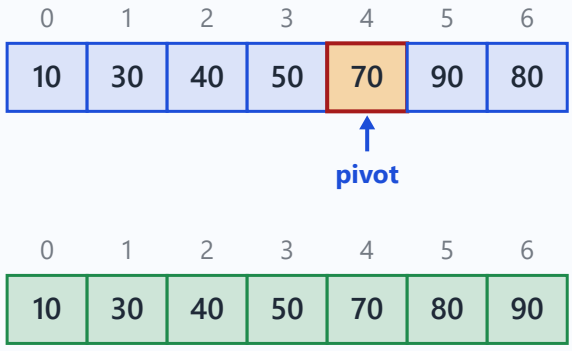


Figure 1.4: Quicksort with Lomuto partitioning on the array `[10, 80, 30, 90, 40, 50, 70]`. Left: the state after the first partition; the pivot 70 has settled at index 4. Right: the sorted array.

The example in Figure 1.4 uses Lomuto partitioning: the last element of the range is the pivot. Table 1.5 gives the order of the calls.

Call	Pivot	New index	Left part	Right part	Array
`partition(arr, 0, 6)`	70	4	[10 30 40 50]	[90 80]	10 30 40 50 70 90 80
`partition(arr, 0, 3)`	50	3	[10 30 40]	[—]	10 30 40 50 70 90 80
`partition(arr, 0, 2)`	40	2	[10 30]	[—]	10 30 40 50 70 90 80
`partition(arr, 0, 1)`	30	1	[10]	[—]	10 30 40 50 70 90 80

Call	Pivot	New index	Left part	Right part	Array
`partition(arr, 5, 6)`	80	5	[—]	[90]	10 30 40 50 70 80 90

Table 1.5: Partition calls for `[10, 80, 30, 90, 40, 50, 70]` (pivot: the last element of the range).

The first partition can be called balanced: parts of four and two elements result. But the left part `[10 30 40 50]` is already sorted. Because the last element is the pivot, each partition separates only one element. The pivots 50, 40, and 30 settle at the end one after another, and the right part stays empty each time. This is the worst split in divide and conquer: $1 + (n - 1)$.

The running time of quicksort depends on the balance of the split. If the pivot is near the median, the recurrence is $T(n) = 2T(n/2) + O(n)$ and the time is $O(n \log n)$; this is the average case. On sorted input, if the last element is chosen as the pivot, each partition separates one element; the recurrence becomes $T(n) = T(n - 1) + O(n)$ and the time $O(n^2)$. The remedy is to choose the pivot independently of the input: a random pivot, or the median of the first, middle, and last elements.

Comparing the two sorting algorithms shows the flexibility of the paradigm. The work must be done somewhere. Quicksort divides cleverly first, and the combine comes for free. Merge sort divides blindly and combines cleverly. Merge sort guarantees balance but needs an extra array; quicksort works in place but leaves the balance to the choice of pivot.

1.3.4 Binary search: decrease and conquer

In a sorted array, binary search compares the middle element with the target value and discards one half of the array entirely. The search continues only in the remaining half. The recurrence is $T(n) = T(n/2) + O(1)$; here $a = 1$, $b = 2$, $n^{\log_2 1} = n^0 = 1$ and $f(n) = 1$ are of the same order. Case 2 gives $\Theta(\log n)$. In an array of a million elements, at most 20 probes suffice.

Binary search has a single subproblem and no combine step. For this reason some sources call it not divide and conquer but **decrease and conquer**: the problem is not split, only reduced. The distinction is terminological; the analysis uses the same recurrence. The real observation is this: if one of the subproblems can be discarded, the gain is large. Linear search takes $O(n)$, binary search $O(\log n)$.

1.3.5 Fast exponentiation

The naive way to compute x^n is to multiply x by itself $n - 1$ times. For $n = 10^6$ that means 999,999 multiplications. With divide and conquer, far fewer multiplications suffice. The observation is simple:

> $$x^n = (x^{n/2})^2 \ (n \text{ even}), \quad x^n = x \cdot (x^{\lfloor n/2 \rfloor})^2 \ (n \text{ odd}).$$

For example, $3^{13} = 3 \cdot (3^6)^2$, $3^6 = (3^3)^2$, and $3^3 = 3 \cdot (3^1)^2$. The problem x^n reduces to a single subproblem whose exponent is half as large. The result of the subproblem is computed once and multiplied by itself. The base case is $x^0 = 1$.

```
long power(long x, int n) {
  if (n == 0) return 1;          // base
  long y = power(x, n / 2);
  if (n % 2 == 0) return y * y; // even
  return x * y * y;              // odd
}
```

The most important line of the code is `long y = power(x, n / 2);`. The value for the half exponent is stored in a variable and used twice: for even n the method returns `y * y`, for odd n it returns `x * y * y`. Each call therefore makes only *one* recursive call and adds at most two multiplications. The recurrence is $T(n) = T(n/2) + O(1)$ and the result is $O(\log n)$. The number of calls is one more than the number of digits in the binary representation of n; the number of multiplications is at most $2(\lfloor \log_2 n \rfloor + 1)$.

1.3.6 Code trace: power(3, 13)

When `power(3, 13)` is called, the exponent halves at each call: $13 \to 6 \to 3 \to 1 \to 0$. The calls first descend along this chain, and the multiplications are performed as the values come back. Table 1.6 gives the return order.

Call	y	n even?	Returned value	Multiplications
`power(3, 0)`	—	base	1	0
`power(3, 1)`	1	odd	$3 \cdot 1 \cdot 1 = 3$	2
`power(3, 3)`	3	odd	$3 \cdot 3 \cdot 3 = 27$	2
`power(3, 6)`	27	even	$27 \cdot 27 = 729$	1
`power(3, 13)`	729	odd	$3 \cdot 729 \cdot 729 = 1{,}594{,}323$	2

Table 1.6: Trace of `power(3, 13)`. The rows are in return order: the deepest call returns first.

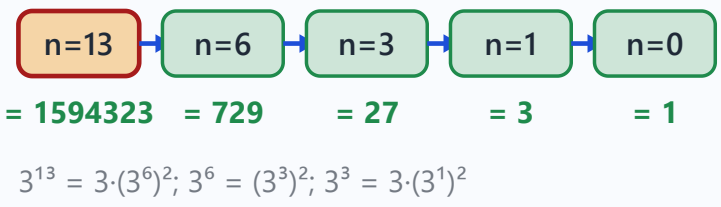


Figure 1.5: The call chain of `power(3, 13)` and its return values. The chain does not branch: each call makes a single subcall. The equalities in the bottom row show the decomposition $3^{13} = 3 \cdot (3^6)^2$.

The total number of multiplications is $2 + 2 + 1 + 2 = 7$; the naive method would make 12. The difference is small for small exponents and striking for large ones. At $n = 10^6$, fast exponentiation makes 21 calls and 27 multiplications; the naive method makes 999,999 multiplications.

A warning: the `long` type holds at most $2^{63} - 1 \approx 9.22 \cdot 10^{18}$. The value $3^{39} \approx 4.05 \cdot 10^{18}$ fits, but $3^{40} \approx 1.22 \cdot 10^{19}$ does not, and Java overflows without raising an error. In real applications, fast exponentiation is used either with `BigInteger` or by taking the remainder modulo some number after every multiplication ($x^n \bmod m$). The second form is the basis for computing very large exponents in public-key cryptography.

Common mistake

Not using the variable `y` and recomputing the half exponent at every use, for example writing `return power(x, n / 2) * power(x, n / 2);` for even n and `return x * power(x, n / 2) * power(x, n / 2);` for odd n. The code still gives the right result, but each call now makes two recursive calls. The recurrence becomes $T(n) = 2T(n/2) + O(1)$, and case 1 of the master theorem gives $\Theta(n)$: the logarithmic gain is lost completely. This version makes 31 calls for `power(3, 11)` and 63 for `power(2, 16)`; the correct version makes 5 and 6. Solving the same subproblem twice is an expensive habit in divide and conquer.

1.3.7 An unsuitable case: overlapping subproblems

The Fibonacci numbers are defined by $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n - 1) + F(n - 2)$. The definition is recursive and can be turned directly into code:

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

The code has the divide-and-conquer form: the problem is split into two smaller subproblems, the subproblems are solved by the same function, and the results are combined by addition. The result is also correct. But this is a bad example, because the subproblems are not independent of each other. To compute $F(n - 1)$, $F(n - 2)$ is already computed; then $F(n - 2)$ is computed again from scratch in the second call.

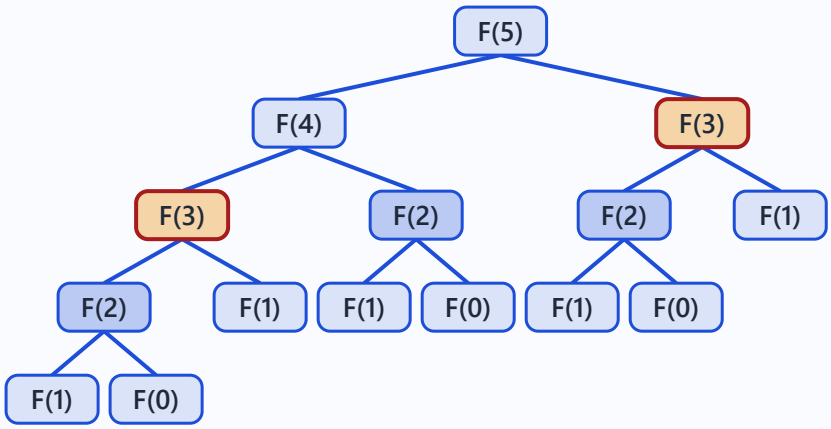


Figure 1.6: The call tree of `fib(5)`. The highlighted nodes are repetitions of the same subproblem: *F(3)* is computed twice and *F(2)* three times.

In the tree of Figure 1.6, `fib(5)` makes 15 calls. *F(3)* is computed twice, *F(2)* three times, *F(1)* five times, and *F(0)* three times. The number of calls grows by the formula $C(n) = 2F(n + 1) - 1$: `fib(30)` makes 2,692,537 calls and `fib(40)` makes 331,160,281. The recurrence is $T(n) = T(n - 1) + T(n - 2) + O(1)$ and its solution is $O(\varphi^n)$, where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio.

The remedy is to store the result of each subproblem when it is first computed, and to read it from a table when it is needed a second time. This approach is called **dynamic programming**; for Fibonacci it brings the time down to $O(n)$. The question to ask when choosing a paradigm is this: are the subproblems independent of each other, or do they contain shared sub-subproblems? If they are independent, divide and conquer is suitable; if they overlap, dynamic programming is. If the locally best choice at each step leads to the global optimum, the greedy approach gives an even simpler solution.

> **Common mistake**
>
> Assuming that a problem can be solved directly by divide and conquer because it has a recursive definition. A recursive definition gives a correct algorithm; it may not give a fast one. If the call tree shows several calls with the same argument, the subproblems overlap and the results should be stored.

1.4 Convex hull and Quickhull

Computational geometry studies algorithms that work on points, lines, and polygons. One of the most basic problems of this field is the convex hull. Quickhull, the divide-and-conquer solution of the problem, takes its name from its resemblance to quicksort: a "pivot" is chosen, the points are partitioned around it, and the parts are processed recursively.

1.4.1 The convex hull problem

Definition 1.2

A polygon is **convex** if the line segment joining any two points inside it lies entirely inside the polygon. The **convex hull** of a set S of n points in the plane is the smallest convex polygon that contains all the points of S.

There is an easy way to keep the definition in mind. Imagine nails driven into a board at the positions of the points, with a rubber band stretched around all of them. When released, the band wraps around the outer nails and tightens. The shape the band takes is the convex hull. The analogy also shows two important facts. First, the vertices of the hull are points of S; the band touches only nails. Second, the inner nails have no effect on the shape of the band.

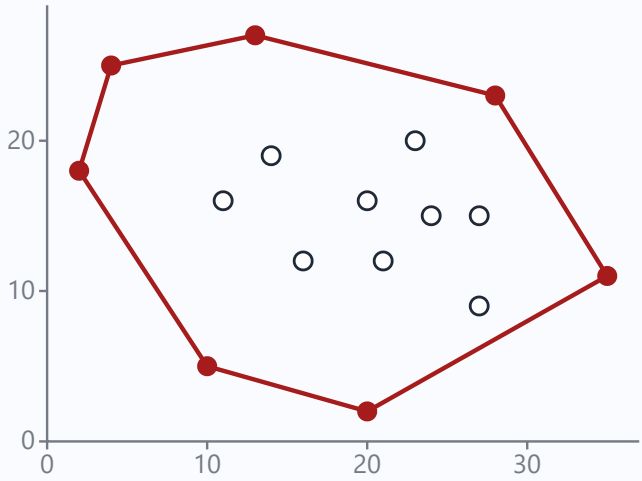


Figure 1.7: A set of 16 points and its convex hull with 7 vertices. Filled points are the vertices of the hull; hollow points lie inside it.

The output of the algorithm is the ordered list of the vertices of the hull; by convention they are listed counterclockwise. It is not enough to answer only "which points are on the hull?", because the order of the vertices is needed to draw the polygon or compute on it. The hull of the set in Figure 1.7 has seven vertices; the remaining nine points are inside.

The convex hull is the first step of many applications. In games and robotics, collision detection speeds up by using the hull of a complex object in its place: if two hulls do not intersect, the objects do not intersect either. Geographic information systems draw the boundary of the observation points in a region with a hull. In image processing the hull of a shape summarizes its form. The two points of a set that are farthest from each other are always vertices of the hull; this observation shrinks the farthest-pair problem. In clustering, the boundary of each cluster is shown by its hull.

1.4.2 The Quickhull algorithm

Quickhull rests on this observation: the point farthest from a line is always part of the hull of the points on that side of the line. The algorithm works in the following steps:

1. The point A with the smallest x coordinate and the point B with the largest are found. Both are on the hull: the leftmost and rightmost points can never lie between other points.
2. The line AB separates the points into two subsets: above the line (to the left when looking from A to B) and below it (to the right).
3. In one subset the point C farthest from AB is found. C is on the hull.
4. The points inside the triangle ACB are eliminated: since the vertices of the triangle are on the hull, the interior of the triangle lies inside the hull, and these points cannot be vertices of the hull.
5. The same procedure is applied recursively to the points outside AC and the points outside CB.
6. When no points remain outside an edge, that edge is a hull edge and the recursion stops. The points found make up the hull.

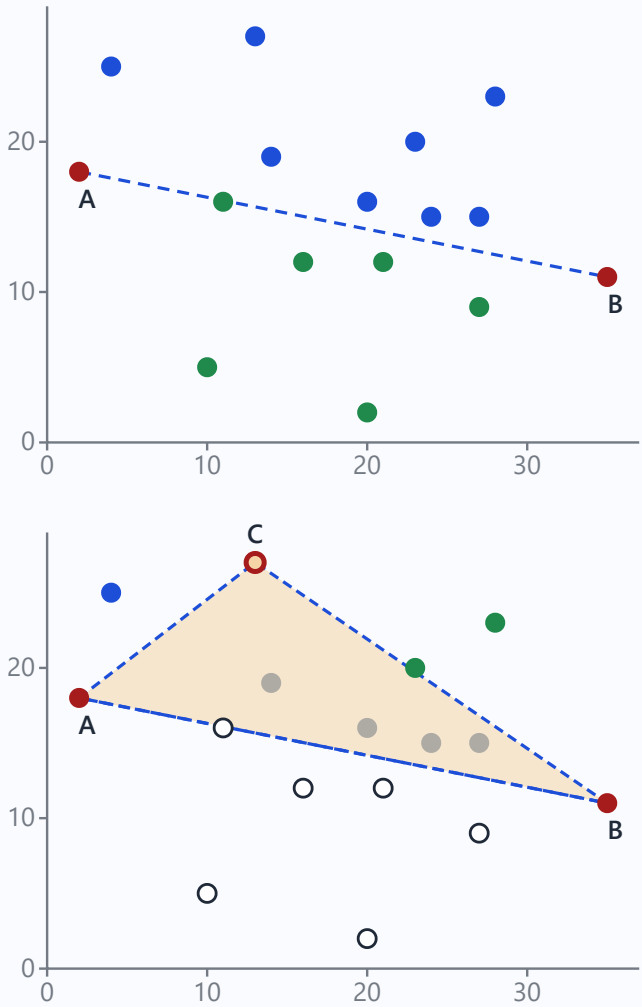


Figure 1.8: The first steps of Quickhull. Left: the leftmost point A and the rightmost point B have been found, and the line AB has split the points into an upper and a lower subset. Right: in the upper subset the point

farthest from AB is $C = (13, 27)$; the four points inside the triangle ACB are eliminated.

The resemblance to quicksort is clear. In quicksort the pivot splits the array into two parts; in Quickhull the farthest point C splits the set into two parts according to the two sides of the triangle. There is one difference: in Quickhull's partitioning some points fall into neither part. The points inside the triangle are discarded entirely, and this discarding is the main reason the algorithm is fast in practice.

The algorithm has almost no combine step. Each recursive call adds the point it finds to the hull list; the results of the two subcalls are not combined separately. The work is done at splitting time, as in quicksort: finding the farthest point and separating the points by the sides of the triangle.

1.4.3 The orientation test and distance

The algorithm keeps asking two geometric questions: is a point P to the left or to the right of the line $A \to B$? And how far is P from AB? Both questions are answered by a single formula. The formula uses only integer multiplication and subtraction; no angles, square roots, or division are needed.

```
long cross(Point A, Point B, Point P) {
  return (long)(B.x - A.x) * (P.y - A.y)
       - (long)(B.y - A.y) * (P.x - A.x);
}
```

The method `cross` computes the z component of the cross product of the vectors $A \to B$ and $A \to P$. The `(long)` casts prevent the `int` product from overflowing for large coordinates. The sign of the value gives the direction:

- `cross > 0`: P is to the left of $A \to B$ (a counterclockwise turn).
- `cross < 0`: P is to the right of $A \to B$ (a clockwise turn).
- `cross = 0`: A, B, and P lie on the same line.

The absolute size of the value carries meaning too. $|cross|$ is twice the area of the triangle with vertices A, B, and P. The area of the triangle is also $\frac{1}{2} \cdot |AB| \cdot h$, where h is the perpendicular distance of P from the line AB. This gives $|cross| = |AB| \cdot h$. For the same A and B, $|AB|$ is constant. The point farthest from the line is therefore the point with the largest `cross` value, and there is no need to compute the distance separately.

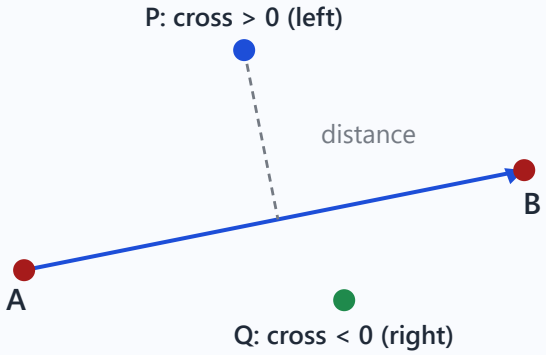


Figure 1.9: The orientation test. P is to the left of $A \to B$ and `cross` is positive; Q is to the right and `cross` is negative. The dashed line is the perpendicular distance of P from the line.

Example 1.3

Let $A = (2, 18)$ and $B = (35, 11)$. Then $B - A = (33, -7)$.

- For $C = (13, 27)$, $C - A = (11, 9)$: $cross = 33 \cdot 9 - (-7) \cdot 11 = 297 + 77 = 374$.
- For $(28, 23)$, $(26, 5)$: $cross = 33 \cdot 5 - (-7) \cdot 26 = 165 + 182 = 347$.
- For $(10, 5)$, $(8, -13)$: $cross = 33 \cdot (-13) - (-7) \cdot 8 = -429 + 56 = -373$; the point is below the line.

The first two points are above the line. Since 374 is larger than 347, $C = (13, 27)$ is farther from the line. The actual distance is $374 / |AB| = 374 / \sqrt{(33^2 + 7^2)} = 374 / \sqrt{1138} \approx 11.09$, but this division is unnecessary for the comparison.

Common mistake

Mixing up the point order in the orientation test. `cross(A, B, P)` and `cross(B, A, P)` have opposite signs. Quickhull finds the upper subset with `leftOf(P, A, B)` and the lower subset with `leftOf(P, B, A)` : the same test, applied to the other side by reversing the direction of the line. If the order is mixed up, one subset is processed twice and the other not at all.

1.4.4 Code: quickHull() and find()

```
void quickHull(List<Point> P) {
  Point A = leftmost(P), B = rightmost(P);
  hull.add(A); hull.add(B);
  find(leftOf(P, A, B), A, B); // upper set
  find(leftOf(P, B, A), B, A); // lower set
}
void find(List<Point> S, Point A, Point B) {
  if (S.isEmpty()) return;   // AB is a hull edge
  Point C = farthest(S, A, B); // max cross
  hull.add(C);
  find(leftOf(S, A, C), A, C); // outside AC
  find(leftOf(S, C, B), C, B); // outside CB
}  // inside ACB is dropped
```

The code consists of two methods. `quickHull` is the starter: it finds the leftmost and rightmost points, adds them to the hull, and calls `find` for the two halves. `find(S, A, B)` works on the set S lying to the left of the line $A \to B$ and finds the hull vertices in this set:

- `if (S.isEmpty()) return;` is the base case. If no points lie outside the line, AB is a hull edge.
- `Point C = farthest(S, A, B);` selects the point with the largest `cross` value, and `hull.add(C);` adds it to the hull.
- The two recursive calls work on the points to the left of $A \to C$ and to the left of $C \to B$. Points that fall into neither of these two sets are inside the triangle and are eliminated automatically; there is no separate code for them.

Processing the lower subset in the direction $B \to A$ involves a subtlety. The part below the line is to the right of $A \to B$, that is, to the left of $B \to A$. This lets `find` always work under the convention "points to the left of the line", so a single method serves both sides. The helper methods used in the code follow. `leftmost` and `rightmost` scan the list once and return the points with the smallest and largest x.

```
List<Point> leftOf(List<Point> S, Point A, Point B) {
  List<Point> result = new ArrayList<>();
  for (Point p : S)
    if (cross(A, B, p) > 0) result.add(p); // left of A→B
  return result;
}
Point farthest(List<Point> S, Point A, Point B) {
  Point best = null;
  long bestVal = -1;
  for (Point p : S) {
    long c = cross(A, B, p);
    if (c > bestVal) { bestVal = c; best = p; }
  }
  return best;
}
```

`leftOf` and `farthest` each scan the list once; both take $O(|S|)$. A `find` call therefore does work proportional to the size of its own set: one `farthest` scan and two `leftOf` scans.

The list `hull` keeps the vertices in the order they are found, and this is not the order around the perimeter of the polygon. For ordered output it suffices to move the line `hull.add(C)` between the two recursive calls. In that case, the vertices of the upper subset are added from A to B and those of the lower subset from B to A; if A is written first and B between the two subsets, the vertices come out in clockwise order. Reversing the list gives counterclockwise order.

1.4.5 Code trace: 16 points

The trace uses the 16 points of Figure 1.7: (2,18), (4,25), (10,5), (11,16), (13,27), (14,19), (16,12), (20,16), (20,2), (21,12), (23,20), (24,15), (27,9), (27,15), (28,23), (35,11). The leftmost point is $A = (2, 18)$ and the rightmost is $B = (35, 11)$; both are added to the hull. Of the remaining 14 points, 8 are to the left of $A \to B$ (the upper subset) and 6 are to the right (the lower subset). Table 1.7 gives the `find` calls in order.

Call	Line	$\lvert S \rvert$	Farthest C	Dropped	Outside AC	Outside CB
1	(2,18) → (35,11)	8	(13,27)	4	1	2
2	(2,18) → (13,27)	1	(4,25)	0	0	0
3	(2,18) → (4,25)	0	S empty: hull edge			
4	(4,25) → (13,27)	0	S empty: hull edge			
5	(13,27) → (35,11)	2	(28,23)	1	0	0
6	(13,27) → (28,23)	0	S empty: hull edge			
7	(28,23) → (35,11)	0	S empty: hull edge			
8	(35,11) → (2,18)	6	(20,2)	4	0	1
9	(35,11) → (20,2)	0	S empty: hull edge			

Call	Line	$\lvert S\rvert$	Farthest C	Dropped	Outside AC	Outside CB
10	(20,2) → (2,18)	1	(10,5)	0	0	0
11	(20,2) → (10,5)	0	S empty: hull edge			
12	(10,5) → (2,18)	0	S empty: hull edge			

Table 1.7: Quickhull trace on 16 points. "Dropped" is the number of points inside the triangle *ACB*.

The first row of the table is the algorithm's most efficient moment. Of the 8 points of the upper subset, the farthest is *C = (13, 27)*; its `cross` value is 374. The four points inside the triangle, (14,19), (20,16), (24,15), and (27,15), are eliminated in a single move. Only (4,25) remains outside *AC*, and (23,20) and (28,23) remain outside *CB*. The numbers add up: *8 = 1 + 4 + 1 + 2* (C, the dropped points, and the two outer sets).

Call 2 processes a one-point set. A single point is naturally the farthest: (4,25) joins the hull and the two subcalls return with empty sets. Call 5 has two points: for the line *(13,27) → (35,11)*, (28,23) is farther (`cross` = 152); (23,20) lies inside the triangle and is eliminated. In the lower subset, call 8 selects *C = (20, 2)* (`cross` = 402) and eliminates the points (11,16), (16,12), (21,12), and (27,9). The one remaining point, (10,5), joins the hull in call 10.

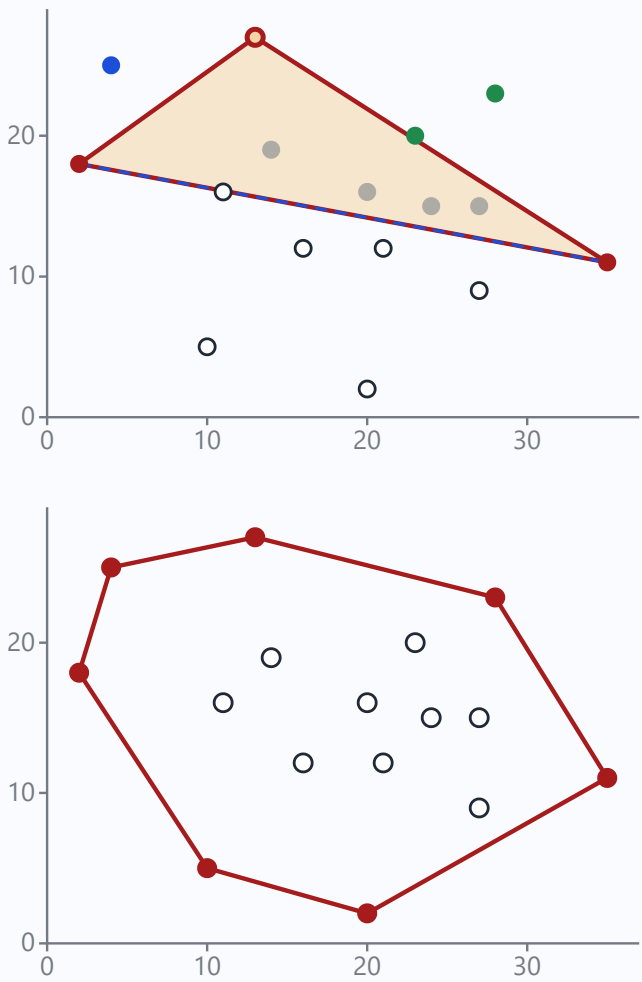


Figure 1.10: Two moments of the Quickhull trace. Left, call 1: in the upper subset the farthest point is (13,27), and the four points inside the shaded triangle are eliminated. Right, the end of the trace: the hull with 7 vertices.

At the end of the trace the hull has seven vertices: (2,18), (35,11), (13,27), (4,25), (28,23), (20,2), (10,5). Nine points have been eliminated, and 12 `find` calls have been made. The number of calls is determined by the structure: each vertex added to the hull inside `find` opens two subcalls, and each hull edge ends in an empty call. Here 5 vertices were found inside `find` (5 non-empty calls), and the 7 edges of the hull gave 7 empty calls.

1.4.6 An example with 104 points

The power of the algorithm shows more clearly on large sets. In a set of 104 points, the leftmost point is (−14, 46) and the rightmost is (106, 20). The line joining these two points has 71 points above it and 31 below. Table 1.8 gives only the non-empty calls, that is, the calls that add a point to the hull. The "Candidates" column is the number of points that have been neither added to the hull nor eliminated; at the start it is *71 + 31 = 102*.

Line	$\lvert S \rvert$	Farthest	Dropped	Candidates left
(−14,46) → (106,20)	71	(54,120)	56	45
(−14,46) → (54,120)	6	(8,84)	2	42
(−14,46) → (8,84)	1	(−4,69)	0	41
(8,84) → (54,120)	2	(42,111)	1	39
(54,120) → (106,20)	8	(104,74)	6	32
(54,120) → (104,74)	1	(77,103)	0	31
(106,20) → (−14,46)	31	(11,−25)	25	5
(106,20) → (11,−25)	5	(88,−5)	4	0

Table 1.8: The non-empty calls of Quickhull on the 104-point set.

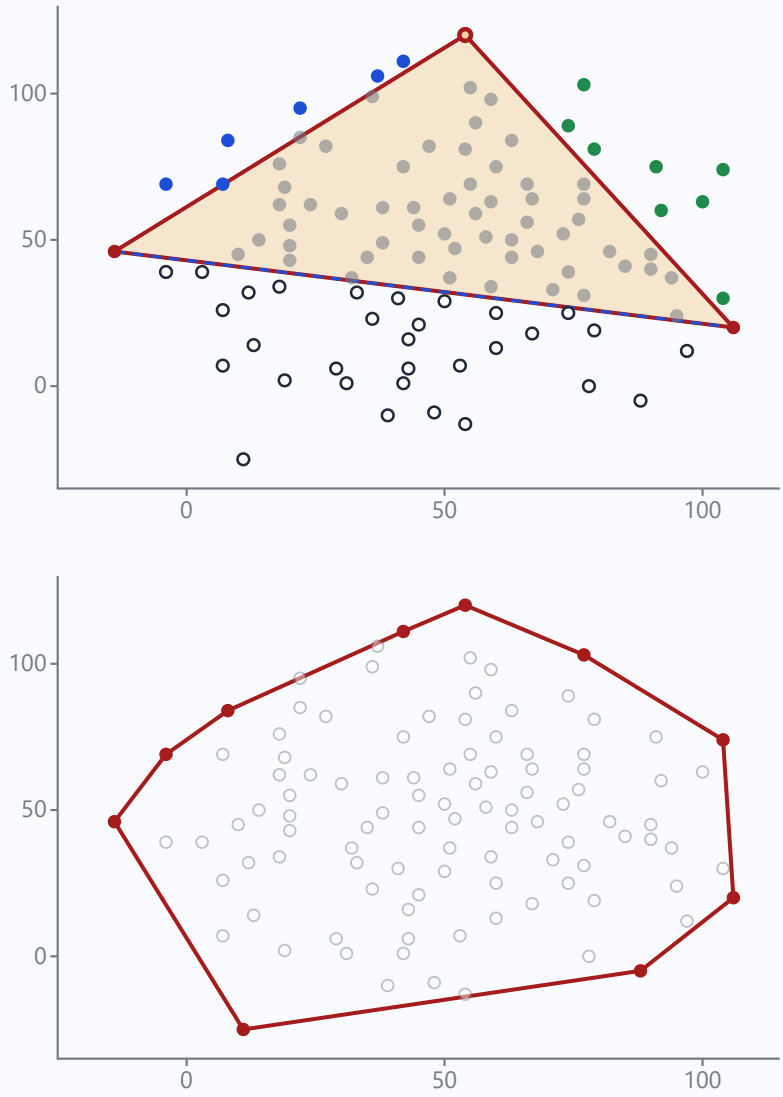


Figure 1.11: The 104-point example. Top, the first non-empty call: in the upper subset the farthest point is (54,120); the 56 points inside the large triangle are eliminated in one step. Bottom, the result: a hull with 10 vertices.

The first call alone eliminates 56 points, and the first call in the lower subset eliminates 25. Of the 94 eliminated points, 81 come from these two calls. The sets of the later calls are small: 6, 1, 2, 8, 1, 5 points. With 8 non-empty calls in all, the 10 vertices of the hull are found and 94 points are eliminated. Because most of the points are eliminated at every step, the work shrinks quickly. This is typical for uniformly distributed random points: the number of hull vertices is very small relative to n, and the vast majority of points fall inside the first few triangles.

1.4.7 Analysis and other hull algorithms

Each `find` call scans its own set a constant number of times: once to find the farthest point and twice to separate the two outer sets. The work of a call is $O(|S|)$. The recurrence has the form

$$T(n) = T(n_1) + T(n_2) + O(n), \quad n_1 + n_2 \le n - 1$$

Since n_1 and n_2 depend on the input, the master theorem does not apply, and the cases are examined separately:

- **Average case.** If the splits are balanced and many points are eliminated at each step, the time is $O(n \log n)$. On uniformly distributed random points in the plane, the algorithm is often even faster than this, because the eliminated points never descend into the deep levels of the recursion.
- **Worst case.** At each step the farthest point may separate only a single point. For example, the points may lie along a convex curve, with the farthest point always falling next to an end. In this case the recurrence becomes $T(n) = T(n - 1) + O(n)$ and the time is $O(n^2)$. On this input no point is eliminated; all the points are on the hull.
- **Output-sensitive bound.** Let h be the number of hull vertices. The sets on one level of the recursion tree are disjoint; the total work of a level is therefore at most n. Because the non-empty calls along every root-to-leaf path find different vertices, the depth of the tree does not exceed h. The total work is $O(n \cdot h)$. When the number of vertices is small, this bound is even better than $n \log n$.

There are other algorithms for the convex hull. Table 1.9 compares them by running time.

Algorithm	Time	Approach
Brute force (every candidate edge)	$O(n^3)$	a pair of points is a hull edge if all the other points lie on the same side of it
Jarvis march (gift wrapping)	$O(n \cdot h)$	start from one vertex and at each step wrap to the next outermost vertex
Graham scan	$O(n \log n)$	sort the points by angle and discard right turns with a stack
Andrew's monotone chain	$O(n \log n)$	sort the points by x and build the upper and lower chains separately
Quickhull	avg. $O(n \log n)$, worst $O(n^2)$	divide and conquer; farthest point and elimination inside the triangle

Algorithm	Time	Approach
Chan	$O(n \log h)$	hulls of small groups by Graham, merged by Jarvis

Table 1.9: Convex hull algorithms (h: the number of hull vertices).

It is not possible to go below these times. An algorithm that finds a convex hull can also solve the sorting problem: the numbers $x_1, \ldots, x_n$ to be sorted are placed on the parabola $y = x^2$ as the points (x_i, x_i^2). Since the parabola is convex, all the points are vertices of the hull, and the ordered vertex list of the hull gives the numbers in sorted order. Since sorting in the comparison model requires $\Omega(n \log n)$ time, the convex hull requires at least that much time as well. In this sense the Graham and Andrew algorithms are optimal. The $O(n \log h)$ time of Chan's algorithm does not contradict this lower bound, because in the parabola example $h = n$.

1.5 The closest pair of points

Finding the two closest points among points in the plane is one of the oldest problems of computational geometry. The brute-force solution looks at all pairs and takes quadratic time. The divide-and-conquer solution brings the time down to $O(n \log n)$. The interesting part of the solution is the combine step: a geometric observation is needed so as not to miss pairs that straddle the boundary between the two halves.

1.5.1 The problem and brute force

Definition 1.3

Let a set of $n \geq 2$ points in the plane be given. The **closest pair of points** problem asks for the pair $p \neq q$ with the smallest Euclidean distance

$$d(p, q) = \sqrt{((p_x - q_x)^2 + (p_y - q_y)^2)}$$

(or only for this smallest distance).

The problem arises in many applications. In air traffic control, the two aircraft that come closest to each other are monitored continuously. Clustering algorithms proceed by merging the two closest points or clusters. In collision detection, the two closest objects are the first candidate pair. In geographic data, facilities that lie close to each other are found.

The most direct solution is to compute the distance of all pairs and take the smallest:

```
double bruteForce(List<Point> p) {
  double min = Double.MAX_VALUE;
  for (int i = 0; i < p.size(); i++)
    for (int j = i + 1; j < p.size(); j++)
      min = Math.min(min, dist(p.get(i), p.get(j)));
  return min;
}
```

The two nested loops try each pair once: j is always greater than i, so neither both (p, q) and (q, p) are tried, nor is a point compared with itself. The number of pairs is $n(n - 1)/2$ and the time is $O(n^2)$. For 10 points, 45 distances are computed; for $n = 10^6$, about $5 \cdot 10^{11}$ distances are needed. On a modern computer this is a job that takes minutes.

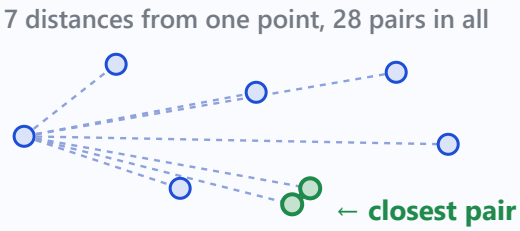


Figure 1.12: Brute force on eight points. Seven distances are computed from a single point; in all, 28 pairs are tried for all the points. The closest pair is highlighted.

In one dimension the problem is easy. The numbers are sorted and only consecutive numbers are compared: the two closest numbers are always adjacent in the sorted array. The time is that of sorting: $O(n \log n)$. In two dimensions this idea does not work directly. When the points are sorted by x, two points with close x values can be far apart because their y values differ greatly. Two points that are not adjacent in the array sorted by x can also be the closest pair. Divide and conquer overcomes this difficulty.

1.5.2 Divide and conquer: the middle strip

The algorithm works in the following steps:

1. The points are sorted once by x and once by y. These two sorted lists are called X and Y. Sorting is done only at the start, before the recursion.
2. **Divide:** The vertical line L through the x coordinate of the point in the middle of X splits the points into a left half and a right half.
3. **Conquer:** The closest distance in each half is found recursively: δ_L and δ_R. The smaller of the two is $\delta = min(\delta_L, \delta_R)$.

4. **Combine:** A closer pair may still exist: a pair with one point on the left and the other on the right. If such a pair exists, it crosses L and both of its points are closer than δ to L. Only the points with $|x - L| < \delta$ are therefore examined, that is, the points of the **strip** of width 2δ.
5. The strip is scanned in y order. Each point is compared only with the points that come after it and whose y difference is less than δ.

The reasoning behind the combine step is as follows. Pairs with both points in the same half have already been examined in the recursion, and none is closer than δ. Only the pairs that cross L remain. If the distance of such a pair is less than δ, then its x difference is also less than δ; both points are therefore at most δ away from L. Likewise, the y difference must be less than δ. Scanning the strip in y order is the cheap way to check this second condition: as soon as the y difference exceeds δ, the following points are even farther away and the inner loop stops.

1.5.3 Why at most seven neighbors in the strip?

The strip can be crowded. On a bad input all the points may fall in the strip. Trying every pair of strip points would again take quadratic time. The observation that makes the algorithm fast is that it suffices to compare each point in the strip with only a constant number of its neighbors.

Theorem 1.3 (Strip neighborhood)

Let the strip points be sorted by y. The number of strip points that come after a point p and whose y difference from it is less than δ is at most 7.

Proof. The points that come after p and whose y difference is less than δ lie inside a rectangle of width 2δ and height δ whose bottom edge is at the y level of p. The rectangle is split in two by L, and each half is a $\delta \times \delta$ square. Each square is divided into four small squares of side $\delta/2$, giving 8 small squares in all. A small square lies entirely within one half. If a small square contained two points, both would be in the same half, and the distance between them would be at most the diagonal of the square, that is, $\delta/\sqrt{2} < \delta$. Yet the closest distance in that half is at least δ; a contradiction. Hence each small square contains at most one point. One of these points is p itself; at most 7 points remain. ■

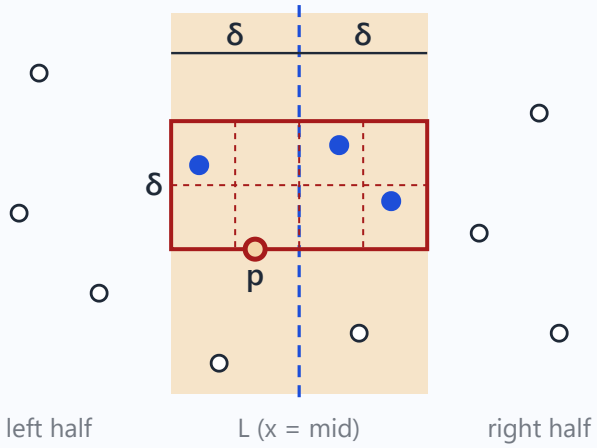


Figure 1.13: The strip of width 2δ around the line L. The $\delta \times 2\delta$ rectangle above p is divided into eight small squares; each small square can contain at most one point.

The theorem gives the bound; the code does not write this bound explicitly. The inner loop stops when the y difference exceeds δ. The theorem guarantees that under this condition the inner loop runs at most 7 iterations. Some sources lower the bound to 6 or even 5; the value of the constant does not change the order of the running time. What matters is that the strip scan takes $O(7n) = O(n)$.

Common mistake

Believing that the total number of points in the strip is bounded. The strip can contain all n points; what is bounded is the number of neighbors each point is compared with. Another mistake is to scan the strip points in x order. The neighbor bound holds only in y order; in a strip scanned in x order, one point may have to be compared with many points.

1.5.4 Code: closest()

```
double closest(List<Point> X, List<Point> Y) {
  int n = X.size(), mid = n / 2;
  if (n <= 3) return bruteForce(X);
  Point m = X.get(mid); // Yl/Yr: halves of Y
  double dl = closest(X.subList(0, mid), Yl);
  double dr = closest(X.subList(mid, n), Yr);
  double d = Math.min(dl, dr);
  List<Point> strip = new ArrayList<>();  // strip
  for (Point p : Y)                   // y order
    if (Math.abs(p.x - m.x) < d) strip.add(p);
  for (int i = 0; i < strip.size(); i++) {
    Point p = strip.get(i);
    for (int j = i + 1; j < strip.size()
         && strip.get(j).y - p.y < d; j++)
      d = Math.min(d, dist(p, strip.get(j)));
  }
  return d;
}
```

The method takes two lists: `X` is the list of points sorted by *x*, and `Y` is the list of the same points sorted by *y*. Before the first call, all the points are sorted once by the two separate criteria. The lines work as follows:

- `if (n <= 3) return bruteForce(X);` is the base case. For three or fewer points, brute force computes at most three distances; splitting is not worth it. There is one more reason for the cutoff of 3: in a half with a single point, the closest pair is undefined.
- `Point m = X.get(mid);` determines the dividing line: L is the line $x = m.x$. The left half is `X.subList(0, mid)` and the right half is `X.subList(mid, n)`.
- `Yl` and `Yr` are the points of `Y` that fall into the two halves, and they preserve the *y* order. The code skips over how they are built with a comment line; one way is given below.
- The two recursive calls give `dl` and `dr`; `d` is the smaller of them.
- The first `for` loop builds the strip. Because it walks over `Y`, the strip is automatically sorted by *y*; no further sorting is needed.
- The nested loops scan the strip. The condition of the inner loop is `strip.get(j).y - p.y < d`; as `d` shrinks, the loop stops earlier. If a closer pair is found, `d` is updated.

```
List<Point> Yl = new ArrayList<>(), Yr = new ArrayList<>();
Set<Point> leftHalf = new HashSet<>(X.subList(0, mid));
for (Point p : Y)                  // y order is kept
  if (leftHalf.contains(p)) Yl.add(p);
  else Yr.add(p);
```

This fragment builds `Yl` and `Yr` in a single pass. The points of the left half are put into a `HashSet`; `Y` is walked in order, and each point is appended to one of the two lists depending on whether it is in the left half. Because the `Point` class does not override `equals`, the set works by object identity; even two different points with the same coordinates do not get mixed up. The pass takes expected *O(n)* time, and the lists preserve the *y* order. No re-sorting is done.

1.5.5 Code trace: 10 points

The trace uses ten points: (2,3), (12,30), (40,50), (5,1), (12,10), (3,4), (7,7), (17,17), (27,27), (28,27). The list sorted by *x* is `X = [(2,3), (3,4), (5,1), (7,7), (12,30), (12,10), (17,17), (27,27), (28,27), (40,50)]`. The points (12,30) and (12,10), which have the same *x*, keep their input order because the sort is stable. The list sorted by *y* is `Y = [(5,1), (2,3), (3,4), (7,7), (12,10), (17,17), (27,27), (28,27), (12,30), (40,50)]`.

Step	Call	Event	Result
1	`X[0..9]`	*n = 10 > 3*; mid = 5, *m = (12, 10)*; 5 points left, 5 right	—
2	`X[0..4]`	*n = 5*; mid = 2, *m = (5, 1)*; 2 points left, 3 right	—
3	`X[0..1]`	brute force: (2,3), (3,4)	1.41
4	`X[2..4]`	brute force: (5,1), (7,7), (12,30); closest (5,1)–(7,7)	6.32
5	`X[0..4]`	*d = min(1.41; 6.32) = 1.41*	—
6	`X[0..4]`	strip $\lvert x - 5 \rvert < 1.41$: [(5,1)]; no comparison	1.41
7	`X[5..9]`	*n = 5*; mid = 2, *m = (27, 27)*; 2 points left, 3 right	—
8	`X[5..6]`	brute force: (12,10), (17,17)	8.60
9	`X[7..9]`	brute force: (27,27), (28,27), (40,50); closest (27,27)–(28,27)	1.00
10	`X[5..9]`	*d = min(8.60; 1.00) = 1.00*	—
11	`X[5..9]`	strip $\lvert x - 27 \rvert < 1$: [(27,27)]; no comparison	1.00
12	`X[0..9]`	*d = min(1.41; 1.00) = 1.00*	—
13	`X[0..9]`	strip $\lvert x - 12 \rvert < 1$: [(12,10), (12,30)]; *y* difference 20 ≥ 1, no comparison	1.00

Table 1.10: Trace of `closest(X, Y)`. Distances are rounded to two decimal places.

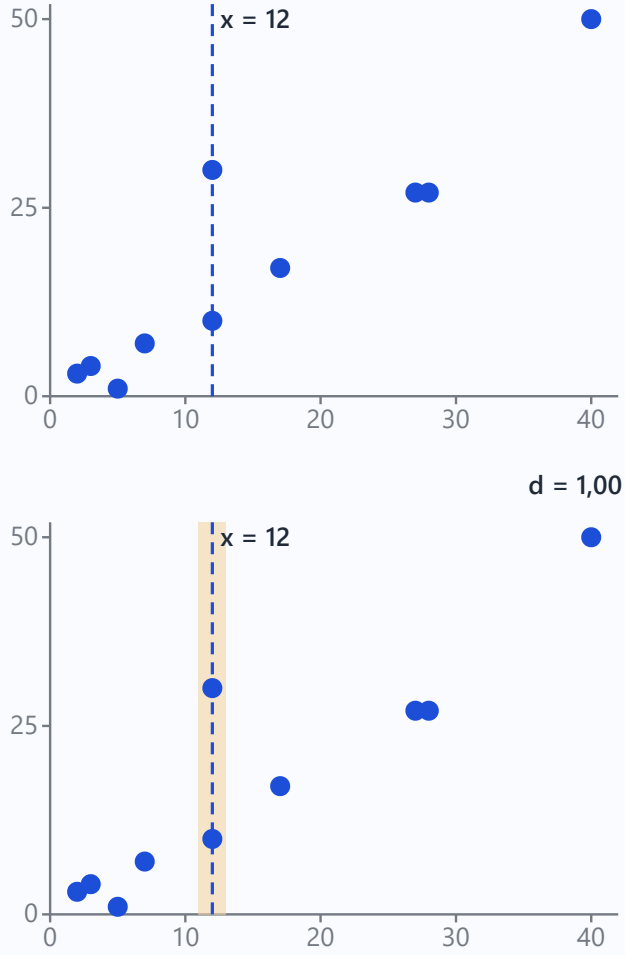


Figure 1.14: The ten-point trace. Left, the first split: the line L is at $x = 12$. Right, the last step: only (12,10) and (12,30) remain in the narrow strip of width $d = 1$.

The result is $d = 1.00$; the closest pair is (27,27)–(28,27). This pair was found by brute force at the base, in the right half of the right half. In this example the strips make no pair comparisons at all. The reason is that δ turns out small early, and the strips become very narrow. The algorithm's total work is 8 distance computations (1 at step 3, 3 at step 4, 1 at step 8, 3 at step 9). Brute force on the same input would compute 45 distances.

An example where the strip actually does work should also be seen. Let L be at $x = 10$, and let the two halves return $\delta_L = 5$ and $\delta_R = 4$. Let the list `Y` be (8,1), (13,2), (11,4), (7,6), (12,9), (9,10), (15,12).

Example 1.4

1. $d = min(5, 4) = 4$. The strip is built with the condition $|x - 10| < 4$: (15,12) is left out; the strip holds, in y order, (8,1), (13,2), (11,4), (7,6), (12,9), (9,10).
2. (8,1): the distance to (13,2) is $\sqrt{26} \approx 5.10$, so d does not change. To (11,4) it is $\sqrt{18} \approx 4.24$, no change. The y difference of (7,6) is $5 \geq 4$; the inner loop stops.

3. (13,2): to (11,4) it is $\sqrt{8} \approx 2.83$; $d = 2.83$. The y difference of (7,6) is $4 \geq 2.83$; it stops.
4. (11,4): to (7,6) it is $\sqrt{20} \approx 4.47$, no change. The y difference of (12,9) is 5; it stops.
5. (7,6): the y difference of (12,9) is $3 \geq 2.83$; no comparison is made.
6. (12,9): to (9,10) it is $\sqrt{10} \approx 3.16$, no change. End of the strip.

In all, 5 distances are computed. The result is $d \approx 2.83$ and the pair is (13,2)–(11,4). This pair crosses L: one point is on the right and the other on the left. Searching the two halves separately would not have found it; the combine step caught it.

1.5.6 Analysis

Step	Cost	Note
Presorting by x and by y	$O(n \log n)$	only once, before the recursion
Splitting (`X` in two, `Y` into `Yl` / `Yr` preserving order)	$O(n)$	no re-sorting
Building the strip	$O(n)$	`Y` is already sorted by y
Scanning the strip	$O(7n) = O(n)$	at most 7 neighbors per point
Total	$O(n \log n)$	$T(n) = 2T(n/2) + O(n)$

Table 1.11: Step-by-step cost of the closest-pair algorithm.

At every level of the recursion, splitting, building the strip, and scanning the strip together take $O(n)$. The recurrence is the same as that of merge sort: $T(n) = 2T(n/2) + O(n)$, with result $O(n \log n)$. Since the two initial sorts also take $O(n \log n)$, the total time is $O(n \log n)$. This time is optimal. In the algebraic decision tree model, even testing whether all the points are distinct requires $\Omega(n \log n)$. Whether the closest distance is zero answers this very question.

Common mistake

Re-sorting the strip in every call. If, instead of using the list `Y`, the strip points are taken in x order and sorted by y at every call, the combine cost becomes $O(n \log n)$. The recurrence turns into $T(n) = 2T(n/2) + O(n \log n)$ and the result is $O(n \log^2 n)$. The difference looks small, but for $n = 2^{20}$, with $\log_2 n = 20$, it means 20 times as much extra work. The right way is to compute the y order once and preserve it while splitting.

The choice of the base case also affects the algorithm. If the cutoff were $n \leq 1$ instead of $n \leq 3$, a half with a single point would return `Double.MAX_VALUE`, effectively infinity. The result would still be correct. A two-point range is split into two single points and d stays infinite. The strip contains both points, and the strip scan finds the pair that would escape. But the number of calls grows, and on small inputs unnecessary recursion overhead arises.

1.6 Karatsuba multiplication

Multiplying two integers looks like something the computer does in a single instruction. This is true only as long as the numbers fit in a machine word, for example 64 bits. In cryptography, scientific computing, and computer algebra, numbers with hundreds or thousands of digits are multiplied. These numbers are stored as digit arrays, and the time of multiplication depends on the number of digits. In 1960 Kolmogorov conjectured that multiplying two n-digit numbers requires $\Omega(n^2)$ operations. Anatoly Karatsuba, then still a student, soon found an algorithm showing this conjecture to be false.

1.6.1 The schoolbook method and naive splitting

In the multiplication method learned in primary school, each digit of the second number is multiplied by each digit of the first, and the shifted partial results are added up. For two n-digit numbers, n^2 single-digit multiplications are performed: the time is $O(n^2)$.

The first attempt at divide and conquer is to split the numbers in two. The n-digit numbers x and y are split at the point $m = \lfloor n/2 \rfloor$:

$$x = a \cdot 10^m + b, \quad y = c \cdot 10^m + d$$

Here a and c are the upper halves of the numbers, and b and d the lower halves. For example, $1234 = 12 \cdot 100 + 34$. Expanding the product gives

$$x \cdot y = ac \cdot 10^{2m} + (ad + bc) \cdot 10^m + bd$$

Multiplying by 10^m merely appends m zeros on the right and takes $O(n)$; the additions also take $O(n)$. The real work is the four half-size products: ac, ad, bc, bd. The recurrence is $T(n) = 4T(n/2) + O(n)$. In the master theorem $n^{\log_2 4} = n^2$ dominates and the result is $O(n^2)$. Naive splitting yields no gain: each of the four products takes $n^2/4$, and the total is again n^2.

1.6.2 Three products instead of four

Karatsuba's observation is that the middle term $ad + bc$ does not need two separate products. The following identity holds:

$$(a + b)(c + d) = ac + (ad + bc) + bd \quad \Rightarrow \quad ad + bc = (a + b)(c + d) - ac - bd$$

ac and bd will be computed anyway. Once one more product, $(a + b)(c + d)$, is made, the middle term is obtained with two subtractions. The three products are named as follows:

$$z_2 = ac,\quad z_0 = bd,\quad z_1 = (a+b)(c+d),\quad x \cdot y = z_2 \cdot 10^{2m} + (z_1 - z_2 - z_0) \cdot 10^m + z_0$$

The price of getting rid of one product is a few extra additions and subtractions. The additions take $O(n)$; the products are each a half-size problem and are solved recursively. The recurrence becomes $T(n) = 3T(n/2) + O(n)$.

Example 1.5

Compute $1234 \cdot 5678$. Here $n = 4$, $m = 2$, and $10^m = 100$.

1. Split: $a = 12$, $b = 34$, $c = 56$, $d = 78$.
2. $z_2 = 12 \cdot 56 = 672$, $z_0 = 34 \cdot 78 = 2652$.
3. $z_1 = (12 + 34)(56 + 78) = 46 \cdot 134 = 6164$.
4. Middle term: $z_1 - z_2 - z_0 = 6164 - 672 - 2652 = 2840$. Check: $ad + bc = 12 \cdot 78 + 34 \cdot 56 = 936 + 1904 = 2840$.
5. Result: $672 \cdot 10^4 + 2840 \cdot 10^2 + 2652 = 6{,}720{,}000 + 284{,}000 + 2652 = 7{,}006{,}652$.

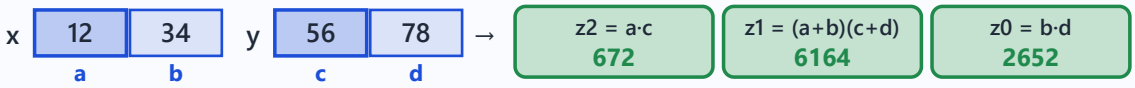


Figure 1.15: The first level of Karatsuba: 1234 and 5678 are each split in two; the three products $z_2 = a \cdot c$, $z_1 = (a + b)(c + d)$, $z_0 = b \cdot d$ are computed.

1.6.3 Code: karatsuba()

```
long karatsuba(long x, long y) {
  if (x < 10 || y < 10) return x * y; // base
  int n = Math.max(digits(x), digits(y));
  long p = (long) Math.pow(10, n / 2);
  long a = x / p, b = x % p;   // x = a·p + b
  long c = y / p, d = y % p;   // y = c·p + d
  long z0 = karatsuba(b, d);
  long z1 = karatsuba(a + b, c + d);
  long z2 = karatsuba(a, c);
  return z2 * p * p + (z1 - z2 - z0) * p + z0;
}
```

The code is a direct counterpart of the formula:

- `if (x < 10 || y < 10) return x * y;` is the base case. If one of the factors has a single digit, the product is computed directly. It matters that the condition uses "or": in a product like $7 \cdot 15$, splitting is meaningless, because 7 has no upper half.
- `n` is the larger of the digit counts of the two numbers; `p = 10^(n/2)` is the split point. `Math.pow` returns a `double`; since small powers of 10 are represented exactly in `double`, the `(long)` cast is safe.
- `a = x / p` and `b = x % p` are the upper and lower halves; `c` and `d` are the same for `y`.
- The three recursive calls compute `z0`, `z1`, and `z2` in turn. The last line combines the results.

The helper method that gives the digit count is simple:

```
int digits(long x) {
  return String.valueOf(Math.abs(x)).length();
}
```

This Java code is meant to show the algorithm. Since the `long` type holds numbers of at most 19 digits, it cannot be used for large numbers. Real implementations store numbers as digit arrays; in Java this is done with the `BigInteger` class. The correctness of the code can still be checked: on 100,000 randomly chosen pairs of numbers, `karatsuba(x, y)` and `x * y` gave the same result.

1.6.4 Code trace: 1234 · 5678

In the call `karatsuba(1234, 5678)`, $n = 4$ and $p = 100$. The call makes three subcalls for *(34, 78)*, *(46, 134)*, and *(12, 56)*. Table 1.12 gives all the calls in the order they are made.

Call	*n*, *p*	a, b, c, d	Combine	Return
(1234, 5678)	4, 100	12, 34, 56, 78	$672 \cdot 10^4 + (6164 - 672 - 2652) \cdot 100 + 2652$	7,006,652
(34, 78)	2, 10	3, 4, 7, 8	$21 \cdot 100 + (105 - 21 - 32) \cdot 10 + 32$	2652
4 · 8, 7 · 15, 3 · 7	base		direct product	32, 105, 21
(46, 134)	3, 10	4, 6, 13, 4	$52 \cdot 100 + (170 - 52 - 24) \cdot 10 + 24$	6164
6 · 4	base		direct product	24
(10, 17)	2, 10	1, 0, 1, 7	$1 \cdot 100 + (8 - 1 - 0) \cdot 10 + 0$	170
0 · 7, 1 · 8, 1 · 1	base		direct product	0, 8, 1
4 · 13	base		direct product	52
(12, 56)	2, 10	1, 2, 5, 6	$5 \cdot 100 + (33 - 5 - 12) \cdot 10 + 12$	672
2 · 6, 3 · 11, 1 · 5	base		direct product	12, 33, 5

Table 1.12: Trace of `karatsuba(1234, 5678)`. Indentation shows call depth.

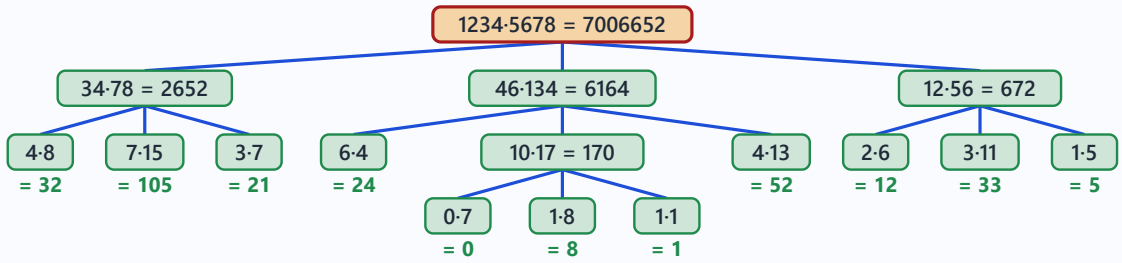


Figure 1.16: The call tree of `karatsuba(1234, 5678)`. Each node is a product; the number below it is the value it returns. Green nodes are base cases.

The interesting point of the trace is the call *(46, 134)*. When the sums of the halves are taken for z_1, the numbers can grow by one digit: *12 + 34 = 46* has two digits, but *56 + 78 = 134* has three. The code handles this automatically: n is the larger of the digit counts of the two numbers, and $p = 10^{\lfloor 3/2 \rfloor} = 10$ is chosen. The split becomes $46 = 4 \cdot 10 + 6$ and $134 = 13 \cdot 10 + 4$. This one-digit growth does not change the order of the running time.

In all, 16 calls are made: 1 root, 3 children, 9 grandchildren, and the 3 children of *(10, 17)*. Of these, 11 calls are base cases. The schoolbook method makes 16 single-digit multiplications for the same product. In this small example the gain is not evident, because some of the base-case products ($7 \cdot 15$, $4 \cdot 13$, $3 \cdot 11$) are larger than single-digit products. The gain shows up on large numbers.

1.6.5 Analysis and practice

The recurrence is $T(n) = 3T(n/2) + O(n)$. Here $n^{\log_2 3} \approx n^{1.585}$ is polynomially larger than $f(n) = n$. Case 1 of the master theorem gives $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$. The same result is found by counting. For numbers of $n = 2^k$ digits, the number of products triples at every level, and after k levels one reaches the single-digit products. The number of single-digit products is $3^k = 3^{\log_2 n} = n^{\log_2 3}$. For example, for $n = 8$, $3^3 = 27$ products are made; the schoolbook method makes 64.

n (digits)	n^2	$n^{1.585}$	Ratio
10	100	38	2.6×
100	10,000	1,479	6.8×
1,000	$1.0 \cdot 10^6$	56,871	17.6×
10,000	$1.0 \cdot 10^8$	$2.2 \cdot 10^6$	45.7×
$1.0 \cdot 10^6$	$1.0 \cdot 10^{12}$	$3.2 \cdot 10^9$	309.2×

Table 1.13: Comparing n^2 with $n^{1.585}$ (constant factors ignored).

The ratio grows like $n^{0.415}$: about 18 times at a thousand digits and about 300 times at a million digits. However, the table does not show constant factors. At every level Karatsuba uses extra additions, subtractions, and temporary numbers. On small numbers this overhead outweighs the gain, and

Karatsuba is slower than the schoolbook method. Implementations therefore use a cutoff: when the number drops below a certain digit count, they switch to the schoolbook method.

Java's `BigInteger.multiply` method follows this approach: it uses the schoolbook method on small numbers, Karatsuba on large ones, and the Toom-Cook method on larger ones still. Toom-Cook is a generalization of Karatsuba's idea: it splits the number into three parts and makes five products instead of nine, giving a time of $O(n^{\log_3 5}) \approx O(n^{1.465})$. Even faster algorithms are based on the fast Fourier transform. The Schönhage–Strassen algorithm takes $O(n \log n \log\log n)$. The algorithm announced by Harvey and van der Hoeven in 2019 takes $O(n \log n)$. This last algorithm is of theoretical importance; because its constants are very large, it is not used in practice.

1.7 Strassen matrix multiplication

Karatsuba's idea also applies to matrices. In 1969 Volker Strassen showed that two matrices can be multiplied faster than by the classical $\Theta(n^3)$ method. The method again rests on removing one recursive call: seven block products instead of eight.

1.7.1 Classical and block matrix multiplication

Each cell of the product $C = A \cdot B$ of two $n \times n$ matrices A and B is the inner product of a row of A with a column of B: $C[i][j] = \Sigma_k A[i][k] \cdot B[k][j]$. The classical method writes this definition as three nested loops:

```
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];
```

For each of the n^2 cells, n products are made: n^3 products in all. The code assumes `C` starts out filled with zeros; in Java a newly created array is already filled with zeros.

The first attempt at divide and conquer is to split each matrix into four blocks of size $n/2 \times n/2$:

$$A = [A_{11}\, A_{12};\, A_{21}\, A_{22}],\quad B = [B_{11}\, B_{12};\, B_{21}\, B_{22}],\quad C = [C_{11}\, C_{12};\, C_{21}\, C_{22}]$$

Blocks can be multiplied and added like numbers; only the order of the factors in a product cannot be changed. The blocks of the result are:

- $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
- $C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$
- $C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
- $C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$

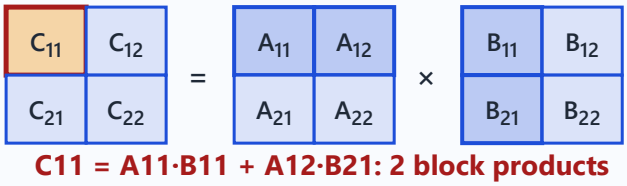


Figure 1.17: Block matrix multiplication. C_{11} is computed from the first block row of A and the first block column of B: two block products and one block addition.

Eight block products and four block additions are needed. A block addition adds $(n/2)^2$ cells and takes $\Theta(n^2)$. The recurrence is $T(n) = 8T(n/2) + \Theta(n^2)$; $n^{\log_2 8} = n^3$ dominates and the result is $\Theta(n^3)$. As with naive splitting in Karatsuba, splitting alone is not enough; the number of products must be reduced.

1.7.2 Seven products instead of eight

Strassen showed that all four result blocks can be obtained from suitable sums of seven intermediate products. Table 1.14 gives these products and the result blocks.

Product	Definition	Result block
P_1	$A_{11} \cdot (B_{12} - B_{22})$	$C_{11} = P_5 + P_4 - P_2 + P_6$
P_2	$(A_{11} + A_{12}) \cdot B_{22}$	$C_{12} = P_1 + P_2$
P_3	$(A_{21} + A_{22}) \cdot B_{11}$	$C_{21} = P_3 + P_4$
P_4	$A_{22} \cdot (B_{21} - B_{11})$	$C_{22} = P_5 + P_1 - P_3 - P_7$
P_5	$(A_{11} + A_{22}) \cdot (B_{11} + B_{22})$	
P_6	$(A_{12} - A_{22}) \cdot (B_{21} + B_{22})$	
P_7	$(A_{11} - A_{21}) \cdot (B_{11} + B_{12})$	

Table 1.14: Strassen's seven products and the result blocks.

The correctness of the formulas is checked by expanding them. C_{12} is the easiest: $P_1 + P_2 = A_{11}B_{12} - A_{11}B_{22} + A_{11}B_{22} + A_{12}B_{22} = A_{11}B_{12} + A_{12}B_{22}$. For C_{11}, expanding P_5, P_4, P_2, and P_6 gives twelve terms. Ten of them cancel in pairs, leaving $A_{11}B_{11} + A_{12}B_{21}$. The order of the factors must be preserved here: for matrices, $A_{11}B_{11}$ and $B_{11}A_{11}$ are generally different. Strassen's formulas never use the commutativity of multiplication; they are therefore valid for blocks as well.

Besides the seven products, 18 block additions or subtractions are made: 10 in the definitions of the P's and 8 in the result blocks. The blocks are in turn multiplied recursively by Strassen; the base case is a 1×1 matrix, that is, the product of two single numbers.

1.7.3 Step by step: a 2 × 2 example

Example 1.6

Let $A = [[1, 3], [7, 5]]$ and $B = [[6, 8], [4, 2]]$. Since the blocks are 1×1, each block is a single number: $A_{11} = 1, A_{12} = 3, A_{21} = 7, A_{22} = 5$; $B_{11} = 6, B_{12} = 8, B_{21} = 4, B_{22} = 2$.

1. $P_1 = 1 \cdot (8 - 2) = 6$
2. $P_2 = (1 + 3) \cdot 2 = 8$
3. $P_3 = (7 + 5) \cdot 6 = 72$
4. $P_4 = 5 \cdot (4 - 6) = -10$
5. $P_5 = (1 + 5) \cdot (6 + 2) = 48$
6. $P_6 = (3 - 5) \cdot (4 + 2) = -12$
7. $P_7 = (1 - 7) \cdot (6 + 8) = -84$
8. $C_{11} = 48 + (-10) - 8 + (-12) = 18$, $C_{12} = 6 + 8 = 14$, $C_{21} = 72 + (-10) = 62$, $C_{22} = 48 + 6 - 72 - (-84) = 66$.

The result is $C = [[18, 14], [62, 66]]$. Check by the classical method: $1 \cdot 6 + 3 \cdot 4 = 18$, $1 \cdot 8 + 3 \cdot 2 = 14$, $7 \cdot 6 + 5 \cdot 4 = 62$, $7 \cdot 8 + 5 \cdot 2 = 66$. Strassen used 7 products; the classical method uses 8.

Common mistake

Making a sign error in Strassen's formulas. $C_{22} = P_5 + P_1 - P_3 - P_7$; P_7 is subtracted. If "$+ P_7$" is written, the program produces a wrong matrix without raising any error: in the example above, C_{22} comes out as $48 + 6 - 72 + (-84) = -102$ instead of 66. This kind of error cannot be caught by the compiler or the runtime. Comparing the implementation with the classical method on small random matrices exposes the bug at once.

1.7.4 Analysis and practice

The recurrence is $T(n) = 7T(n/2) + \Theta(n^2)$. Here $n^{\log_2 7} \approx n^{2.807}$ is polynomially larger than n^2, and case 1 of the master theorem gives $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$. Counting shows the same thing: for $n = 2^k$ the number of scalar products is 7^k; the classical method makes $8^k = n^3$ products. On 16×16 matrices, Strassen makes 2401 products and the classical method 4096. For $n = 1024 = 2^{10}$, $7^{10} \approx 2.8 \cdot 10^8$ and $8^{10} \approx 1.07 \cdot 10^9$; Strassen makes about 3.8 times fewer products.

The algorithm has some problems in practice:

- **Size.** If n is not a power of 2, the matrix is padded with zeros up to the nearest power. The zeros do not change the result, but they add work.

- **Cutoff.** Because of the 18 extra additions and the temporary matrices, Strassen is slower than the classical method at small n. Libraries switch to the classical method once the submatrix size drops to about 64–128.
- **Numerical stability.** With floating-point numbers, the subtractions can amplify rounding error. The classical method is more reliable in this respect.
- **Memory.** Temporary blocks are created at every level; the extra memory is $O(n^2)$.

After Strassen, smaller upper bounds were found for the exponent ω of matrix multiplication. As of 2024 the best known bound is $\omega < 2.372$. The constants of these algorithms are so large that they are not used in practice. Strassen's real lesson is theoretical: trading a multiplication for additions lowers the exponent by reducing a by one. The same idea appears in Karatsuba's drop from 4 to 3 and in Strassen's from 8 to 7.

1.8 Assessment

Table 1.15 sets the naive method and the divide-and-conquer solution side by side for the problems discussed in these notes.

Problem	Naive method	Divide and conquer	Recurrence
Sorting	$O(n^2)$ (bubble sort)	$O(n \log n)$	$2T(n/2) + n$
Searching a sorted array	$O(n)$	$O(\log n)$	$T(n/2) + 1$
Exponentiation x^n	$O(n)$ products	$O(\log n)$	$T(n/2) + 1$
Convex hull	$O(n^3)$	avg. $O(n \log n)$	$T(n_1) + T(n_2) + n$
Closest pair of points	$O(n^2)$	$O(n \log n)$	$2T(n/2) + n$
Integer multiplication	$O(n^2)$	$O(n^{1.585})$	$3T(n/2) + n$
Matrix multiplication	$O(n^3)$	$O(n^{2.807})$	$7T(n/2) + n^2$

Table 1.15: The naive method and divide and conquer.

The gain comes from three sources. First, the work is split in a balanced way and combined in $O(n)$ at each level: sorting and the closest pair of points. Second, the number of subproblems is reduced: Karatsuba and Strassen. Third, one of the subproblems is discarded entirely, or the result of a single subproblem is reused: binary search and fast exponentiation. Quickhull is a mix of the first and third kinds: the points inside the triangle are discarded, and the remaining work is split into two unbalanced parts.

When deciding whether to solve a problem by divide and conquer, ask the following questions. Can the problem be split into smaller pieces of the same kind? Are the pieces independent of each other? Can the solutions of the pieces be combined more cheaply than by the naive method? If the answers to the first two are yes and the third is no, divide and conquer gives a correct but slow algorithm, as in the maximum-element example. If the pieces overlap, dynamic programming is more suitable; if the locally best choice at each step suffices, the greedy approach is.

1.9 Common mistakes

- **Forgetting the base case** or writing a base case that is never reached. The result is infinite recursion and a `StackOverflowError`. In Karatsuba the base case is `x < 10 || y < 10`. If this line is deleted, then on single-digit numbers $p = 10^0 = 1$. `karatsuba(4, 8)` calls `karatsuba(0, 0)` in its first subcall; that call in turn calls itself with the same arguments, and the recursion never ends.
- **Unbalanced splits.** Choosing the last element as the pivot on a sorted array, or separating only one point at each step in Quickhull, gives $T(n) = T(n - 1) + n = O(n^2)$.
- **Not noticing overlapping subproblems.** Computing Fibonacci by plain recursion takes exponential time; dynamic programming, which stores the results, is needed.
- **Calling the same subproblem twice.** Calling `power(x, n / 2)` twice in fast exponentiation raises $\Theta(\log n)$ to $\Theta(n)$.
- **Underestimating the combine cost.** Sorting the strip at every call in the closest pair of points makes the time $O(n \log^2 n)$; $2T(n/2) + n \log n \neq \Theta(n \log n)$.
- **Using the master theorem in the wrong place.** $T(n) = T(n - 1) + n$, unequal splits, and functions that fall between the cases, such as $f(n) = n/\log n$, are outside the scope of the theorem.
- **Sign and direction errors.** A single sign error in $C_{22} = P_5 + P_1 - P_3 - P_7$ of Strassen produces a wrong matrix without any error message. In Quickhull, if the point order passed to `cross` is mixed up, the upper and lower subsets swap places. Comparing with the naive method on a small example catches these errors.
- **Believing that divide and conquer always speeds things up.** In the maximum-element problem divide and conquer also makes $n - 1$ comparisons; the gain is parallelism, not running time.

1.10 Summary

- Divide and conquer splits the problem into smaller, independent subproblems of the same kind, solves the subproblems recursively, and combines the results. A base case is essential.
- The running time is expressed by the recurrence $T(n) = aT(n/b) + f(n)$. The master theorem compares $f(n)$ with $n^{\log_b a}$: if the leaves dominate, the result is $\Theta(n^{\log_b a})$; if they are equal, a factor of $\log n$ is added; if the root dominates, $\Theta(f(n))$.
- For unequal splits or splits that shrink by a constant amount, the master theorem does not apply; the recursion tree is used.
- In merge sort the real work is in the merge; in quicksort it is in the partition. Binary search and fast exponentiation take $O(\log n)$ with a single subproblem.
- If subproblems overlap (Fibonacci), divide and conquer takes exponential time; dynamic programming is needed.

- Quickhull starts from the leftmost and rightmost points, at each step adds the point farthest from the line (the largest `cross`) to the hull, and eliminates the inside of the triangle. It takes $O(n \log n)$ on average, $O(n^2)$ in the worst case, and $O(n \cdot h)$ output-sensitively.
- In the closest pair of points, the points are split in two by x, $\delta = \min(\delta_L, \delta_R)$ is taken, and the strip of width 2δ is scanned in y order. Each point is compared with at most 7 neighbors; the time is $O(n \log n)$.
- Karatsuba uses the identity $ad + bc = (a + b)(c + d) - ac - bd$ to make three products instead of four: $\Theta(n^{1.585})$.
- Strassen makes seven block products instead of eight: $\Theta(n^{2.807})$. In both algorithms the gain comes from reducing a by one; on small inputs, below a cutoff, they switch to the naive method.

1.11 Key terms

divide and conquer base case subproblem recursion recurrence (relation) recursion tree master theorem merge sort quicksort partition binary search overlapping subproblems dynamic programming convex hull closest pair of points brute force lower bound Karatsuba multiplication Strassen's algorithm

---

1.12 Test Yourself

1. What are the solutions of the following recurrences by the master theorem? (a) $T(n) = 3T(n/3) + n$, (b) $T(n) = T(n/2) + n$, (c) $T(n) = 16T(n/4) + n^2$.

   Answer

   (a) $n^{\log_3 3} = n = f(n)$; case 2: $\Theta(n \log n)$. (b) $n^{\log_2 1} = 1$, and $f(n) = n$ is polynomially larger; regularity $f(n/2) = n/2 \le \tfrac{1}{2} \cdot n$ holds; case 3: $\Theta(n)$. (c) $n^{\log_4 16} = n^2 = f(n)$; case 2: $\Theta(n^2 \log n)$.

2. The method `findMax` is called as `findMax(a, 0, 5)` on the array `[3, 8, 1, 9, 4, 6]`. How many calls and how many comparisons are made in total, and what is the result?

   Answer

   For six elements the recursion tree has 6 leaves and 5 internal nodes: 11 calls and 5 comparisons. The first split is `[0..2]` and `[3..5]`; the maximum of the left half is 8 and that of the right half is 9. The result is 9.

3. How many recursive calls and how many multiplications does `power(2, 25)` make? How many

multiplications does the naive method make?

Answer

The sequence of exponents is $25 \to 12 \to 6 \to 3 \to 1 \to 0$: 6 calls. Multiplications: $n = 1$ (odd) 2, $n = 3$ (odd) 2, $n = 6$ (even) 1, $n = 12$ (even) 1, $n = 25$ (odd) 2; total 8. The result is $2^{25} = 33{,}554{,}432$. The naive method makes 24 multiplications.

4. Let $A = (0, 0)$, $B = (10, 0)$, and let the upper subset contain the points (2,3), (5,6), (8,2), (5,2). Which point does Quickhull choose as C? Which points are eliminated, and with which does the recursion continue?

Answer

$cross(A, B, P) = 10 \cdot P_y$: the values are 30, 60, 20, 20; $C = (5, 6)$. $cross(A, C, (2,3)) = 5 \cdot 3 - 6 \cdot 2 = 3 > 0$: (2,3) is outside AC. For (8,2), $cross(C, B, \cdot) = 5 \cdot (-4) - (-6) \cdot 3 = -2$; for (5,2), $cross(A, C, \cdot) = -20$ and $cross(C, B, \cdot) = -20$: both are inside the triangle and are eliminated. The recursion continues on the AC side with (2,3) alone; the CB side is empty.

5. In the closest-pair algorithm the line L is at $x = 10$ and $\delta = 3$ has been returned from the two halves. The list `Y` is (9,1), (11,2), (14,4), (10,5), (8,6), (12,7). Which points make up the strip, how many distances are computed, and what is the result?

Answer

For (14,4), $|14 - 10| = 4 \geq 3$; it is outside the strip. The strip: (9,1), (11,2), (10,5), (8,6), (12,7). The distance (9,1)–(11,2) is $\sqrt{5} \approx 2.24$ and d becomes 2.24; the y difference of (10,5) is 4, so the inner loop stops. After (11,2), the y difference of (10,5) is 3; it stops. (10,5)–(8,6) = 2.24 and (10,5)–(12,7) ≈ 2.83 are computed. (8,6)–(12,7) ≈ 4.12 is computed. In all, 4 distances; the result is $d = \sqrt{5} \approx 2.24$.

6. Which subcalls are made in the call `karatsuba(56, 78)`? How many calls are made, and how many of them are base cases?

Answer

$p = 10$; $a = 5$, $b = 6$, $c = 7$, $d = 8$. $z_0 = 6 \cdot 8 = 48$ (base). $z_1 = karatsuba(11, 15)$: the inner split gives $1, 1, 1, 5$; the inner products are $1 \cdot 5 = 5$, $2 \cdot 6 = 12$, $1 \cdot 1 = 1$; the result is $100 + (12 - 1 - 5) \cdot 10 + 5 = 165$. $z_2 = 5 \cdot 7 = 35$ (base). The result is $35 \cdot 100 + (165 - 35 - 48) \cdot 10 + 48 = 4368$. In all, 7 calls, 5 of them base cases.

7. What are $P_1, \ldots, P_7$ and C when the matrices $A = [[1, 0], [2, 1]]$ and $B = [[3, 1], [0, 2]]$ are multiplied

with Strassen's method?

Answer

$P_1 = 1 \cdot (1 - 2) = -1$, $P_2 = (1 + 0) \cdot 2 = 2$, $P_3 = (2 + 1) \cdot 3 = 9$, $P_4 = 1 \cdot (0 - 3) = -3$, $P_5 = (1 + 1)(3 + 2) = 10$, $P_6 = (0 - 1)(0 + 2) = -2$, $P_7 = (1 - 2)(3 + 1) = -4$. $C_{11} = 10 - 3 - 2 - 2 = 3$, $C_{12} = -1 + 2 = 1$, $C_{21} = 9 - 3 = 6$, $C_{22} = 10 - 1 - 9 + 4 = 4$. $C = [[3, 1], [6, 4]]$; the classical method gives the same result.

8. Karatsuba's idea is generalized: numbers are split into three parts, and five products are made instead of nine partial products. If splitting and combining take $O(n)$, what is the running time of the algorithm? Is it faster than Karatsuba?

Answer

$T(n) = 5T(n/3) + O(n)$. $n^{\log_3 5} \approx n^{1.465}$ dominates; case 1: $\Theta(n^{1.465})$. It is asymptotically faster than Karatsuba's $n^{1.585}$; because its constants are larger, it pulls ahead only on larger numbers. This is the Toom-Cook method.

9. How many calls does the naive recursive `fib(6)` make? Why is divide and conquer unsuitable here?

Answer

$C(n) = 1 + C(n - 1) + C(n - 2)$, $C(0) = C(1) = 1$: $C(5) = 15$, $C(4) = 9$, $C(6) = 25$. The subproblems are not independent: while $F(4)$ is computed, $F(3)$, $F(2)$, … are already computed, and they are recomputed from the other branch. If the results are stored (dynamic programming), each $F(k)$ is computed once and the time is $O(n)$.

1.13 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 4: gives Strassen's algorithm, the recursion-tree and substitution methods, and the proof of the master theorem. Chapter 33 treats the convex hull and the closest pair of points.
- J. Kleinberg, É. Tardos, *Algorithm Design* (2005), Chapter 5: explains divide-and-conquer recurrences intuitively; contains readable treatments of the closest pair of points and integer multiplication.
- C. B. Barber, D. P. Dobkin, H. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls" (1996): the paper that generalizes Quickhull to higher dimensions and forms the basis of the widely used Qhull software.
- V. Strassen, "Gaussian Elimination is not Optimal" (1969): the original paper, only a few pages long, that introduced the seven-product idea.

1.14 References

1. C. B. Barber, D. P. Dobkin, H. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls", *ACM Transactions on Mathematical Software* 22(4), 1996.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 4: Divide-and-Conquer; Chapter 33: Computational Geometry).
3. A. Karatsuba, Yu. Ofman, "Multiplication of Multidigit Numbers on Automata", 1962.
4. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005 (Chapter 5: Divide and Conquer).
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.
6. V. Strassen, "Gaussian Elimination is not Optimal", *Numerische Mathematik* 13, 1969.

PART I · CHAPTER 2

Greedy Algorithms

When a cashier gives change, they hand over the largest banknote first, then the largest coin that still fits. An attendant filling a meeting room's schedule for the day writes in the meeting that ends earliest first. Neither computes the future; each takes the step that looks best at that moment and never revisits the decision. Algorithms of this kind are called greedy algorithms. The greedy approach is fast and its code is short, but it does not give the best result on every problem. These notes first present the general pattern of the approach and the conditions under which it gives the correct result, namely the greedy-choice property and optimal substructure. The exchange argument, the basic tool for proving correctness, is treated in detail. Five classic problems then follow: coin change, activity selection, job sequencing with deadlines, the fractional knapsack, and Huffman coding. The minimum spanning tree algorithms are then re-examined through the cut property from a greedy point of view. For each problem the notes give Java code, a step-by-step trace, a cost analysis, and counterexamples for the cases where the greedy approach fails.

LEARNING OUTCOMES

The approach

Define the greedy strategy; explain when the greedy-choice property and optimal substructure guarantee a correct result; build a simple correctness proof with the exchange argument.

Classic problems

Solve the coin change, activity selection, job sequencing, and fractional knapsack problems greedily; trace their code step by step; give counterexamples for the cases where the greedy approach fails.

Spanning trees

Explain why the cut property makes the greedy choice of Kruskal, Prim, and Borůvka correct; compare the three algorithms by selection rule, data structure, and running time.

Data compression

Build a Huffman tree, derive prefix-free codes, encode and decode a text, and compute the saving in bits; justify why the Huffman code is optimal.

2.1 The greedy approach

Many algorithmic problems consist of a sequence of decisions. Which coin to hand over, which activity to select, which job to put in which hour, which edge to add to the tree? These decisions have to be made in order, and each affects the ones that follow. Trying every sequence of decisions gives the correct result, but the number of sequences grows exponentially with the input. The greedy approach follows just a single branch of this tree: at each step it takes the option that looks best at that moment and moves on.

2.1.1 Deciding one step at a time

Definition 2.1

A **greedy algorithm** builds the solution step by step and, at each step, selects the option that looks best *at that moment* according to some criterion. A decision once made is never undone; the algorithm does not backtrack.

A greedy algorithm resembles a treasure hunter. At every fork in the road the hunter turns toward the direction that shows the most gold. The hunter does not look at the whole map, only at the options directly ahead. Choosing, while planning a trip, the road that looks shortest to the destination at every junction is the same behavior. Sometimes that short-looking road leads to a dead end or a long detour. Because the algorithm does not turn back, it cannot correct this wrong step.

Both faces of the approach are plain in this example. On one side are speed and simplicity. At each step only one option is selected, and the options not selected are never examined again. Most greedy algorithms sort the candidates once and then decide in a single pass. On the other side is the lack of a guarantee. The choice that looks best at the moment does not always lead to the best solution of the whole. On many problems a greedy algorithm produces a good approximate solution, but that it finds the optimal solution can be shown only by a proof.

In these notes "best solution", "optimal solution", and "optimum" are used in the same sense. A problem can have more than one best solution. For example, two different sets of activities with the same number of activities are both best if both reach the maximum number. It is enough for a greedy algorithm to find any one of them.

2.1.2 The general template

Most greedy algorithms follow the same skeleton. First one determines which choices do not violate the constraints, that is, what the **feasible** solutions are. Then a **greedy criterion** is chosen: cheapest, earliest finishing, most profitable, most frequent, and so on. The candidates are sorted by this

criterion. Then each candidate is examined in turn. If adding the candidate to the current solution does not violate the constraints, it is added; if it does, it is skipped. When the solution is complete, the algorithm stops. The template code in the slides shows this skeleton in Java:

```
Solution greedy(Candidate[] cands) {
  Solution s = new Solution();   // empty solution
  sort(cands);                   // by the criterion
  for (Candidate c : cands) {
    if (s.isFeasible(c))         // constraints hold
      s.add(c);                  // never undone
  }
  return s;
}
```

`Solution` and `Candidate` in the code are abstract names; they carry a different meaning in each problem. In coin change, a candidate is a coin and the solution is the list of coins handed over. In activity selection, a candidate is an activity and the solution is the set of selected activities. The call `sort` orders the candidates by the greedy criterion. `isFeasible` checks whether the new candidate violates the constraints. The loop considers each candidate once, and a candidate that joins the solution with `add` is never removed again. The comments in the code stress these two properties: the check protects the constraints, and the addition is not undone.

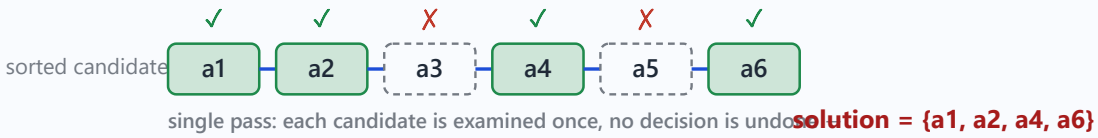


Figure 2.1: The single pass of the greedy template. The candidates are sorted by the criterion; each candidate is examined once. Feasible ones (a1, a2, a4, a6) are added to the solution, those that violate the constraint (a3, a5) are skipped. No decision is undone.

Figure 2.1 shows this single pass. Six candidates arrive in sorted order. a1 and a2 do not violate the constraints and are added. a3 conflicts with those selected so far and is skipped. a4 is added, a5 is skipped, and a6 is added. The result is the set *{a1, a2, a4, a6}*. The skipping of a3 is permanent: after a4 and a6 are added, no one goes back to a3 to ask "is it feasible now?".

The cost of the template has two parts. Sorting the n candidates takes $O(n \log n)$. The scan that follows performs one `isFeasible` check for each candidate. If the check finishes in constant time, the scan takes $O(n)$ and the sort determines the total cost: $O(n \log n)$. In some problems the candidates are not sorted up front; at each step the best candidate is taken from a priority queue. This is the case in Huffman coding and in Prim's algorithm. Since each queue operation takes $O(\log n)$, the total cost is again mostly of order $O(n \log n)$. If the check is expensive, for example when a free time slot has to be searched for, as in job sequencing, the dominant part of the cost shifts to the check.

Common mistake

Believing that applying the template correctly makes the algorithm correct. The template is only a skeleton; the criterion determines the result. The same skeleton gives the best result in activity selection with the "earliest finish" criterion, and a very bad result with the "earliest start" criterion. In every new problem, the correctness of the chosen criterion must be proved separately.

2.1.3 The greedy-choice property and optimal substructure

For a greedy algorithm to give the best result, the problem must have two properties together. The first concerns the first step: the first choice the algorithm makes must not close off the path to the best solution. The second concerns the rest of the work. After the first choice, the remaining problem must be a smaller problem of the same kind. Its best solution, combined with the first choice, must give the best solution of the whole.

Definition 2.2

A problem has the **greedy-choice property** if there is *at least one* best solution that contains the locally best choice according to the greedy criterion. In other words, the first greedy choice never sends the algorithm down the "wrong path".

Definition 2.3

A problem has **optimal substructure** if, after a choice has been made, the best solution of the remaining subproblem together with that choice gives the best solution of the whole problem.

What the two properties give together can be seen by induction. The greedy-choice property says that there is a best solution containing the first choice. After this choice has been made, the remaining problem is of the same kind. Optimal substructure says that the best solution of the remaining problem combined with the first choice is the best solution of the whole. Because the greedy-choice property also holds in the remaining problem, the algorithm's second choice does not send it down the wrong path either. This continues: at every step, the partial solution at hand remains part of some best solution. When the algorithm finishes, the partial solution is complete and optimal.

The words "at least one" in the definition are important. The greedy-choice property does not say that *every* best solution contains the greedy choice. Activity selection is a good example: in the ten-activity example in the slides the maximum size is 6, and there are two different sets of this size. Only one of them contains the earliest-finishing activity. For the property this is enough.

Optimal substructure is not unique to greedy algorithms. Dynamic programming also rests on this property. What separates the two approaches is the greedy-choice property. Dynamic programming does not know which choice is best at the next step; it tries all the options and stores the results of the subproblems in a table. A greedy algorithm knows in advance which choice is safe and makes only that one. This is why, on the problems where it applies, a greedy algorithm is much faster than dynamic programming.

If the two properties do not hold together, a greedy algorithm is only a **heuristic**. A heuristic is fast and gives a reasonable result on most inputs. But it does not by itself say how much worse than the best its result can be. Coin change with the coins *{4, 3, 1}* is the simplest example: for the amount 6 the greedy algorithm gives 3 coins, although 2 suffice.

2.1.4 The exchange argument

The most common way to prove the greedy-choice property is the **exchange argument**. The proof starts from an arbitrary best solution. Call it O. If O already contains the greedy choice, there is nothing more to say. If it does not, a suitable element of O is removed and the greedy choice is put in its place. The real work of the proof is to show two things:

1. After the change the solution is still **feasible**; no constraint is violated.
2. After the change the solution is **no worse**; the objective value either stays the same or improves.

Once these two points are shown, a best solution containing the greedy choice has been found. Together with optimal substructure, induction is applied and the result extends to all the steps.

The strength of the argument is that it does not require knowing what the best solution looks like. All that is known about O is that it is feasible and optimal. The proof makes O resemble the greedy solution step by step and shows that quality does not drop at any step. Most greedy proofs fit this pattern. In activity selection the first activity is exchanged. In the fractional knapsack a portion of one item trades places with a portion of another. In the cut property one edge is exchanged for another, and in Huffman coding two leaves are swapped with each other.

Example 2.1

A student can read at most k of n exam papers; each paper has a score, and the total score should be as large as possible. The greedy criterion is obvious: the k papers with the highest scores are selected. The exchange argument is set up as follows. Let a be the paper with the highest score, and let O be a best selection that does not contain it. Any paper b is removed from O and a is put in its place. Since the number of papers has not changed, the selection is feasible. Since $score(a) \geq score(b)$, the total does not decrease. So there is a best selection that contains a. After a is selected, the remaining problem is to select $k - 1$ papers from the other $n - 1$, and it is of the same kind.

The example looks easy because the constraint concerns only the number of selected items. As constraints grow more complicated, it gets harder to show that the exchange does not violate feasibility. In activity selection this step requires comparing finish times, and in Huffman coding it requires comparing depths in the tree.

> **Common mistake**
>
> In the exchange argument, showing only the "no worse" part and skipping feasibility. If the changed solution is not feasible, that is, if it violates one of the constraints, the proof shows nothing. In the 0/1 knapsack, putting a high-ratio item in place of a low-ratio item can raise the profit, but if the new item is heavier, the capacity is exceeded. This is exactly why the ratio criterion fails in this problem.

2.1.5 Strengths and weaknesses

The strengths of the greedy approach can be listed briefly:

- **Clear, short code.** The whole algorithm is often a sort and a loop. None of the Java code given in these notes exceeds twenty lines.
- **Speed.** Sorting and a single pass take $O(n \log n)$ in total. Solving the same problem with dynamic programming usually requires filling a table and a higher cost.
- **Little memory.** Only the current partial solution is kept. No table storing the results of subproblems is needed.
- **The best result on the right problem.** On problems such as the minimum spanning tree, Huffman coding, and activity selection, a greedy algorithm is provably optimal.
- **A good start on hard problems.** On problems where finding the best solution is very expensive, a greedy algorithm gives a fast approximate solution or a starting point for better methods.

The weaknesses are at least as important:

- **There is no guarantee.** Coin change with the coins $\{1, 3, 4\}$ and the 0/1 knapsack are known examples where the greedy result is worse than the best.
- **Each problem needs its own proof.** That a greedy algorithm is correct on one problem does not show that it will be correct on another that looks similar.
- **The choice of criterion is decisive.** Of the many criteria that come to mind for the same problem, usually only one is correct. A wrong criterion makes an algorithm wrong even when its code is written perfectly.
- **Mistakes cannot be corrected.** Because a decision is never undone once made, the effect of an early bad choice is carried to the end.

2.1.6 The greedy approach and divide and conquer

Divide and conquer also reduces a problem to smaller problems. This resemblance can lead to confusing the two approaches. The difference lies in how the reduction is done. Divide and conquer splits the problem by a predetermined rule, for example the array in the middle. There is no choice in this split. It then solves all the pieces and combines the results. The greedy approach, by contrast, makes a choice, and after this choice a single subproblem remains. That subproblem is solved; there is no other piece. Table 2.1 collects the differences under five headings.

	Greedy	**Divide and conquer**
Decision	a single, locally best choice at each step; never undone	split by a fixed rule; no choice is made
Subproblems	one subproblem remains after the choice	several mutually independent subproblems
Correctness	the greedy-choice property and optimal substructure must be proved	depends on the correctness of the combine step
Cost	usually $O(n \log n)$: sorting and a single pass	found from the recurrence and the master theorem
Example	activity selection, Kruskal, Huffman	merge sort, Karatsuba

Table 2.1: Comparison of the greedy approach and divide and conquer.

The cost row reflects this difference as well. The running time of merge sort is found from the recurrence $T(n) = 2T(n/2) + O(n)$ with the master theorem, and it is $O(n \log n)$. For the running time of a greedy algorithm there is usually no need to set up a recurrence; the sorting and scanning costs are simply added. The correctness row shows the most important difference between the two approaches. In divide and conquer, if the subproblems are solved correctly and the combine step is correct, the result is correct automatically. In the greedy approach, the correctness of the result depends on the safety of the choice made, and this must be proved separately.

2.2 The coin change problem

The most familiar example of the greedy approach is giving change. If a cashier wants to pay 68 kuruş with as few coins as possible, they hand over the 50 first, then the 10, then the 5, and finally three 1s. At each step they select the largest coin that fits the remaining amount. This method works perfectly in everyday currency systems. But if the denominations are chosen differently, the same method can miss the best result. This section examines both the cases where the method works and the cases where it fails.

2.2.1 The problem and the greedy criterion

> **Definition 2.4**
>
> The **coin change problem**: given k coin denominations with positive integer values $p_1 > p_2 > \ldots > p_k = 1$ and an amount T. Any number of coins of each denomination may be used. The goal is to find the fewest coins whose sum is exactly T.

That the smallest coin is 1 guarantees that every amount can be paid. Without this condition, some amounts cannot be paid at all; for example, 7 cannot be paid with only 3 and 5 coins. The greedy criterion is this: select the **largest** coin that is not greater than the remaining amount; repeat this step until the amount is zero. The Java code in the slides applies this rule with two nested loops:

```
// coins[] sorted from largest to smallest
static int makeChange(int[] coins, int amount) {
  int count = 0;
  for (int c : coins) {
    while (amount >= c) {
      amount -= c;
      count++;
    }
  }
  return count;
}
```

The code assumes that the coin array arrives sorted from largest to smallest. The outer loop considers each denomination once, in this order. The inner `while` loop uses that coin as long as it fits the remaining amount: it subtracts it from the amount and increments the counter. When the coin no longer fits, the inner loop ends and the next, smaller denomination is taken up. Since the smallest coin is 1, the amount is always zero after the last denomination.

The cost has two parts. The outer loop runs k times and tests at least one `while` condition per iteration; this amounts to $O(k)$. Each time the body of the inner loop runs, one coin is selected; the body runs *count* times in total. The total time is $O(k + count)$. If the coin array does not arrive sorted, an $O(k \log k)$ sort is added. The inner loop can also be written with division and remainder: `count += amount / c; amount %= c;`. This version takes $O(k)$ regardless of the number of coins and is faster on large amounts. The result is the same.

2.2.2 Step by step in a canonical system

In the first example in the slides the coins are *{50, 25, 10, 5, 1}* and the amount is 68. Table 2.2 gives each step.

Step	Largest coin that fits	Remaining amount	Coins used
start	—	68	0
1	50	68 − 50 = 18	1
2	10 (25 does not fit)	18 − 10 = 8	2
3	5	8 − 5 = 3	3
4	1	3 − 1 = 2	4
5	1	2 − 1 = 1	5
6	1	1 − 1 = 0	6

Table 2.2: Greedy payment of the amount 68 with the coins *{50, 25, 10, 5, 1}*.

The result is *68 = 50 + 10 + 5 + 1 + 1 + 1*, that is, 6 coins. This is indeed the fewest coins for 68. That fewer is impossible can be seen as follows. All the coins other than 1 are multiples of 5, and the remainder of 68 divided by 5 is 3. Every solution therefore contains at least three 1 coins. If there were a solution with at most 5 coins, at most two coins other than the 1s would have to pay 65. But 65 cannot be paid with two coins: 50 + 10 = 60, 50 + 25 = 75, and 25 + 25 = 50; no sum of two coins equals 65. It cannot be paid with one coin or with none, either.

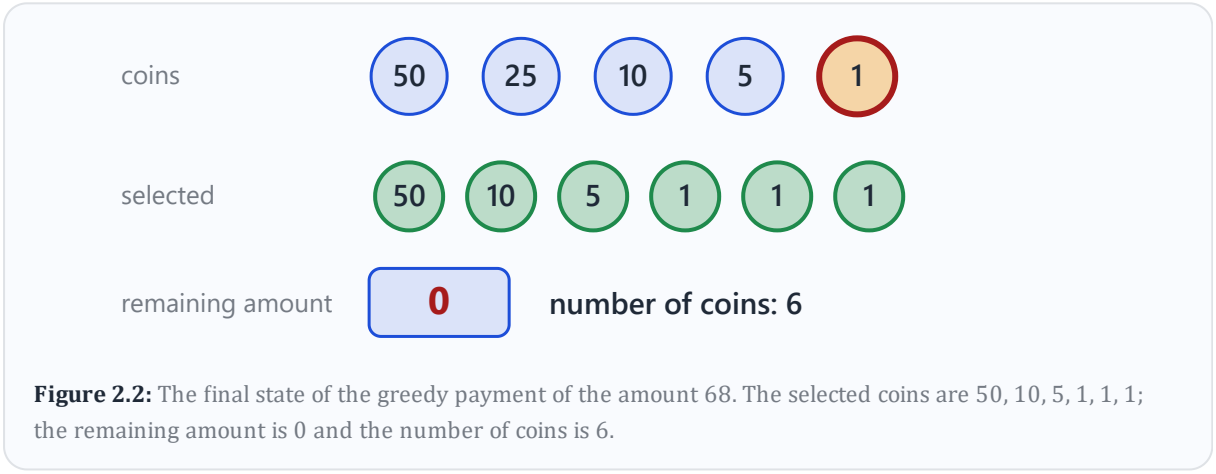


Figure 2.2: The final state of the greedy payment of the amount 68. The selected coins are 50, 10, 5, 1, 1, 1; the remaining amount is 0 and the number of coins is 6.

Coin systems in which the greedy result is optimal for every amount are called **canonical** coin systems. *{50, 25, 10, 5, 1}* and *{25, 10, 5, 1}* are canonical systems. The *{100, 50, 25, 10, 5, 1}* kuruş corresponding to the coins of the Turkish lira is also canonical. So is the sequence *{200, 100, 50, 20, 10, 5, 2, 1}* cents of the euro system. The verification script for these notes compared every amount from 1 to 499 in each of these systems with dynamic programming and found no difference. That everyday currency systems are canonical is no accident; the systems are designed with this property in mind.

2.2.3 Code trace: {4, 3, 1} and 10

Now let the denominations be *{4, 3, 1}* and call `makeChange` with the amount 10. The coins are given from largest to smallest. This system is not canonical; whether the greedy result will be optimal is not known in advance. Table 2.3 shows the values of the variables at each step of the code.

Step	`c`	Condition `amount >= c`	Action	`amount`	`count`
1	4	10 ≥ 4 true	subtract 4	6	1
2	4	6 ≥ 4 true	subtract 4	2	2
3	4	2 ≥ 4 false	`while` ends	2	2
4	3	2 ≥ 3 false	the 3 coin is never used	2	2
5	1	2 ≥ 1 true	subtract 1	1	3
6	1	1 ≥ 1 true	subtract 1	0	4
7	1	0 ≥ 1 false	`while` ends, `return 4`	0	4

Table 2.3: Trace of `makeChange({4, 3, 1}, 10)`.

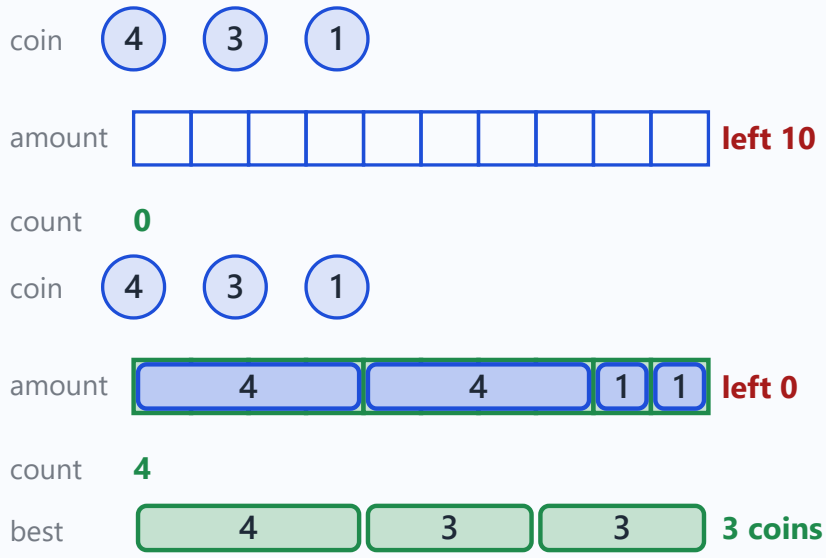


Figure 2.3: `makeChange({4, 3, 1}, 10)`: the initial state (top) and the final state (bottom). Greedy uses 4 coins with 4 + 4 + 1 + 1; the optimal solution is 3 coins with 4 + 3 + 3.

The code returns 4: *10 = 4 + 4 + 1 + 1*. Yet *10 = 4 + 3 + 3* uses only 3 coins. The mistake is made at the second step. After the first 4 is selected, for the remaining amount 6 the greedy rule again selects 4, because 4 still fits. But 6 could have been paid with two 3 coins. The moment the second 4 is selected, the 3 coin becomes useless for the remaining 2, and 2 is paid with two 1 coins. The fourth row of the trace shows this outcome: the 3 coin is never used.

2.2.4 Why does the greedy choice fail?

In the same system the amount 6 is an even shorter counterexample. The greedy algorithm first selects 4 and pays the remaining 2 with two 1 coins: *6 = 4 + 1 + 1*, 3 coins. The optimal solution is *6 = 3 + 3*, 2 coins. Figure 2.4 shows these two results.

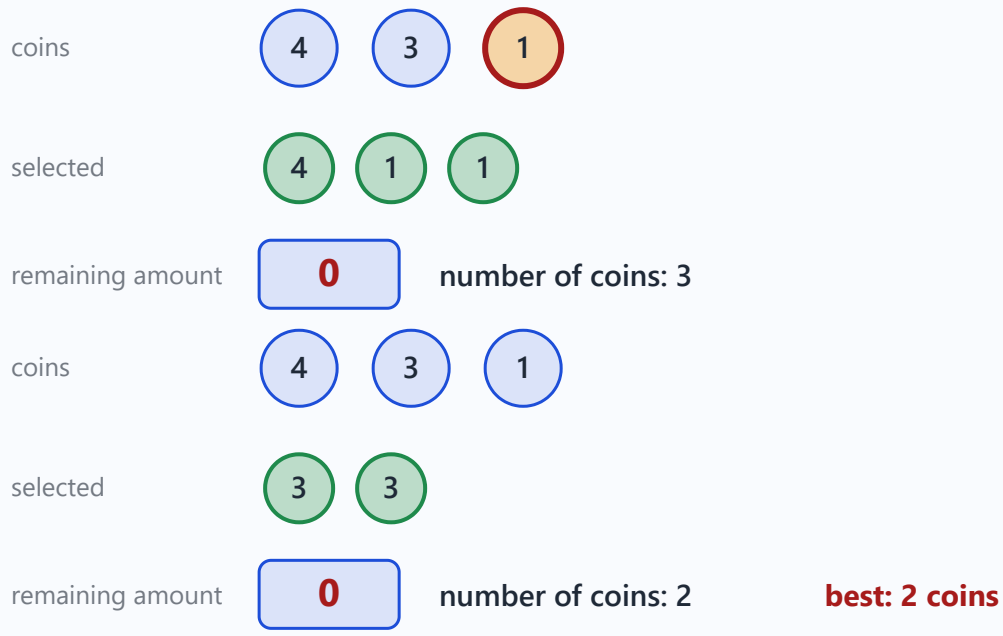


Figure 2.4: The amount 6 with the coins *{4, 3, 1}*. Greedy uses 3 coins with 4 + 1 + 1 (top); the optimal solution is 2 coins with 3 + 3 (bottom).

Here the greedy-choice property is broken. The first greedy choice is 4, but 3 + 3, the only optimal solution for 6, does not contain 4. Every solution that contains 4 uses at least 3 coins. The exchange argument does not work in this system either: if one of the 3 coins in the optimal solution is exchanged for a 4, the total becomes 7 and the solution loses feasibility.

Trying a single amount is not enough to tell whether a coin system is canonical. In the system *{4, 3, 1}*, among the amounts from 1 to 10, greedy errs only at 6 and 10; for the amounts 1–5, 7, 8, and 9 the greedy result is optimal. Up to 20, the amounts with errors are 6, 10, 14, and 18. The second counterexample system in the slides is *{10, 7, 1}*. In this system the first erroneous amount is 14: greedy uses 5 coins with *10 + 1 + 1 + 1 + 1*, and the optimal solution is 2 coins with *7 + 7*. For 15, greedy gives 6 coins (10 plus five 1s), and the optimal solution is 3 coins with *7 + 7 + 1*.

Example 2.2

Which coin system with the smallest largest coin makes greedy change fail? If the largest coin is 2, the system is *{2, 1}*, and greedy is optimal for every amount: as many 2s as possible, then at most one 1. If the largest coin is 3, the possible systems are *{3, 1}* and *{3, 2, 1}*. In both, any amount is paid in the form $3q + r$ ($r \le 2$), and nothing does better. The systems with largest

coin 4 are *{4, 1}*, *{4, 2, 1}*, *{4, 3, 1}*, and *{4, 3, 2, 1}*. Of these, only *{4, 3, 1}* is not canonical, and its first counterexample is 6. The verification script found this result by scanning all the small systems. So *{1, 3, 4}* with the amount 6 is the smallest example on which greedy fails.

Whether a system is canonical is generally not easy to see. That neighboring coins are multiples of each other is a sufficient condition: in a system like *{100, 50, 10, 5, 1}*, each larger coin can stand in for the smaller ones. This condition is not necessary, however; in the system *{25, 10, 5, 1}*, 25 is not a multiple of 10, but the system is still canonical. As Kozen and Zaks showed, the smallest counterexample of a non-canonical system is smaller than the sum of the two largest coins. To check a system it therefore suffices to compare the amounts up to this bound with dynamic programming. For *{4, 3, 1}* the bound is 7, and 6 lies in this range. For *{10, 7, 1}* the bound is 17, and 14 lies in this range.

Common mistake

Trying greedy change on a few amounts and assuming it is correct. In the system *{4, 3, 1}* all the amounts from 1 to 5 give the correct result; the first error appears at 6. A few successful trials are not a proof. The correctness of a greedy algorithm is either shown by a proof or, as in coin change, checked by comparing a whole finite range against the optimal method.

2.2.5 The optimal result: dynamic programming

If the coin system is not canonical, dynamic programming gives the best result. The idea is to compute the minimum number of coins for every amount, from smallest to largest. Let *minCoins[t]* be the minimum number of coins needed to pay the amount *t*. We have *minCoins[0] = 0*. For *t > 0*, if the last coin used is *p*, the amount *t − p* remains, and it too must be paid in the best way. All the coins are tried and the best is chosen:

$$\text{minCoins}[t] = 1 + \min \{ \text{minCoins}[t - p] : p \le t \}$$

This recurrence is an instance of optimal substructure: if the last coin is removed from the best solution for *t*, the remaining coins form the best solution for *t − p*. The difference from the greedy algorithm is that it is not known which coin will be last, and all the options are tried. Table 2.4 gives the table from 0 to 10 for the system *{4, 3, 1}* next to the greedy result.

t	0	1	2	3	4	5	6	7	8	9	10
greedy	0	1	2	1	1	2	**3**	2	2	3	**4**
minCoins[t]	0	1	2	1	1	2	**2**	2	2	3	**3**

Table 2.4: The greedy result and *minCoins[t]* in the system *{4, 3, 1}*.

Computing two cells of the table in detail shows how the recurrence works. For $t = 6$ there are three options: if the last coin is 1, $1 + minCoins[5] = 3$; if it is 3, $1 + minCoins[3] = 2$; if it is 4, $1 + minCoins[2] = 3$. The smallest is 2, and it is obtained with the 3 coin. For $t = 10$ the options are $1 + minCoins[9] = 4$, $1 + minCoins[7] = 3$, and $1 + minCoins[6] = 3$; the smallest is 3. The greedy algorithm follows only the option of the 4 coin and finds 4 coins for 10.

The price of dynamic programming is time. The table has $T + 1$ cells, and each cell tries the k coin denominations; $O(k \cdot T)$ time and $O(T)$ memory are needed in total. The greedy algorithm finishes in $O(k + count)$ time and uses no extra memory. The interactive slide in the presentation runs the two methods side by side: the desired set of coins and an amount are entered, and the greedy result is compared with the optimal result of dynamic programming. One of the ready-made buttons loads the example $\{1, 7, 10\}$ with 14, and the other loads the canonical Turkish kuruş system with the amount 68.

Common mistake

Calling `makeChange` with an unsorted coin array. The code assumes the array is sorted from largest to smallest and does not check it. If it is called with the array `{1, 5, 10}`, the first denomination is 1, so the whole amount is paid with 1 coins; for 68 it returns 68 coins. Either the caller must sort the array, or a sort must be added at the start of the method.

2.3 Activity selection

Many activities want to use a conference hall. Each activity has a known start time and finish time. Only one activity can take place in the hall at a time. The management wants to accept as many mutually non-overlapping activities as possible. It is not the duration or importance of the activities that matters, only their number. This problem is a classic example in which a greedy algorithm gives the best result and this can be shown by a short proof.

2.3.1 The problem definition

Definition 2.5

Activity selection: n activities are given; activity i occupies the half-open interval $[s_i, f_i)$ ($s_i < f_i$). Two activities $[s_i, f_i)$ and $[s_j, f_j)$ are **compatible** if $s_j \geq f_i$ or $s_i \geq f_j$. The goal is to find a set consisting of the largest possible number of mutually compatible activities.

That the intervals are half-open means that another activity can start at the moment one ends. For example, $[3, 4)$ and $[4, 6)$ do not overlap. The finish instant is not part of the activity. The example in the slides has ten activities (Table 2.5).

Activity	1	2	3	4	5	6	7	8	9	10
start s	1	2	4	6	5	8	0	3	7	10
finish f	3	5	6	8	7	9	2	4	9	12

Table 2.5: The ten activities in the slides.

Trying all the subsets by brute force means 2^n sets: 1024 sets for ten activities, and more than a trillion for forty. The greedy algorithm sorts the activities once and selects in a single pass.

2.3.2 Which criterion?

To build a greedy algorithm one must first answer the question "which activity should be selected at each step?". Four criteria come to mind:

1. **Earliest start.** Let the hall fill up as early as possible.
2. **Shortest duration.** Short activities keep the hall busy for less time.
3. **Fewest conflicts.** Select the one that blocks the fewest other activities.
4. **Earliest finish.** Let the hall become free as early as possible.

Each looks reasonable, but only the last gives the best result on every input. A single counterexample suffices to show that a criterion is wrong. Figure 2.5 shows the counterexamples from the slides for the first two criteria.

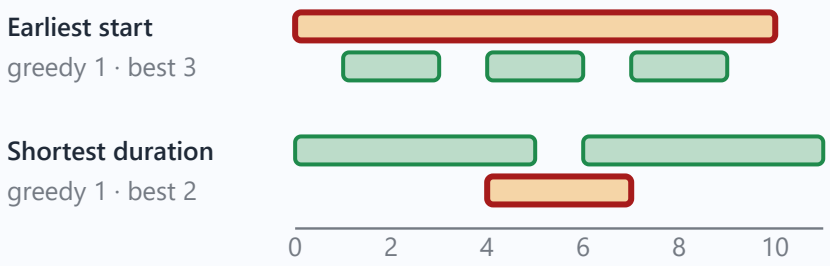


Figure 2.5: Counterexamples to wrong criteria. Orange: the activity selected by greedy; green: the optimal solution. The earliest-start criterion selects 1 activity, while the best is 3. The shortest-duration criterion selects 1 activity, while the best is 2.

In the upper example, the activity *[0, 10)* starts earliest. The greedy algorithm selects it and discards the activities *[1, 3)*, *[4, 6)*, and *[7, 9)* that overlap with it; the result is 1 activity. Yet these three short activities do not overlap one another and can be selected together. In the lower example, *[4, 7)* is the shortest activity (3 units). Greedy selects it, but this activity overlaps both *[0, 5)* and *[6, 11)*. The result is again 1 activity; the best solution selects the two long activities together.

The fewest-conflicts criterion needs a subtler counterexample, because on small examples it often gives the right result. Table 2.6 gives a counterexample with eleven activities. The best solution consists of A, B, C, and D and contains 4 activities. But the activity with the smallest number of

conflicts is M: it overlaps only B and C. Greedy selects M first and so loses both B and C. At most two more of the remaining activities can be selected; the result is 3 activities. The verification script tried every way of breaking ties and showed that this criterion cannot exceed 3 on this input.

Activity	Interval	Number of conflicting activities
A	*[0, 3)*	3
B	*[4, 7)*	4
C	*[8, 11)*	4
D	*[12, 15)*	3
X_1, X_2, X_3	*[2, 5)* (three copies)	4 each
Y_1, Y_2, Y_3	*[10, 13)* (three copies)	4 each
M	*[6, 9)*	**2**

Table 2.6: A counterexample to the "fewest conflicts" criterion.

The thinking behind the earliest-finish criterion is this: the activity that frees the hall earliest leaves the most time for the remaining activities. The longer the remaining time, the more activities can fit there. The criterion looks not at the duration or the start of the activity, only at its finish.

2.3.3 Code and trace

The `select` method in the slides sorts the activities by finish time, selects the first, and then compares each activity with the finish of the last one selected:

```
static void select(Activity[] a) {
  Arrays.sort(a, (x, y) -> x.finish - y.finish);
  int i = 0;                 // last selected
  System.out.print(a[i].id + " ");
  for (int j = 1; j < a.length; j++) {
    if (a[j].start >= a[i].finish) {
      System.out.print(a[j].id + " ");
      i = j;
    }
  }
}
```

In the code, the `Activity` class has the fields `id`, `start`, and `finish`. The expression `(x, y) -> x.finish - y.finish` in the `Arrays.sort` call is a comparator: if the finish of `x` is smaller, a negative value is returned and `x` moves ahead. The sort Java uses for object arrays is **stable**; two activities with equal finish times keep their original order in the array. The variable `i` holds the position of the last selected activity in the array. The first element of the sorted array is the earliest-finishing activity and is selected unconditionally. The loop starts at `j = 1`. The condition `a[j].start >= a[i].finish` tests whether activity `j` starts after the last selected one, that is, whether it is compatible with it. If the condition is true, the activity is printed and `i` is updated.

There is an important observation here: the new activity is compared only with the *last* selected one, not with all the selected ones. This suffices because the activities are sorted by finish. The last selected is the latest-finishing among the selected. If the new activity starts after it, it also starts after all the selected activities that finished earlier.

After sorting, the array is as follows: *#7 [0, 2)*, *#1 [1, 3)*, *#8 [3, 4)*, *#2 [2, 5)*, *#3 [4, 6)*, *#5 [5, 7)*, *#4 [6, 8)*, *#6 [8, 9)*, *#9 [7, 9)*, *#10 [10, 12)*. #6 and #9 have the same finish (9); the stable sort keeps #6 in front. Table 2.7 gives each step of the loop.

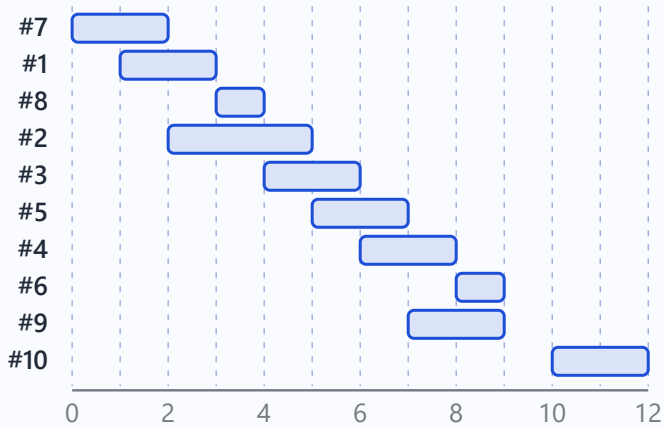


Figure 2.6: The ten activities sorted by finish time (top to bottom). The horizontal axis is time.

`j`	**Activity**	`a[j].start`	`a[i].finish`	**Decision**	**New `i`**
0	#7 [0, 2)	0	—	first activity, selected	0
1	#1 [1, 3)	1	2	1 < 2, overlaps, skipped	0
2	#8 [3, 4)	3	2	3 ≥ 2, selected	2
3	#2 [2, 5)	2	4	2 < 4, skipped	2
4	#3 [4, 6)	4	4	4 ≥ 4, selected	4
5	#5 [5, 7)	5	6	5 < 6, skipped	4
6	#4 [6, 8)	6	6	6 ≥ 6, selected	6
7	#6 [8, 9)	8	8	8 ≥ 8, selected	7
8	#9 [7, 9)	7	9	7 < 9, skipped	7
9	#10 [10, 12)	10	9	10 ≥ 9, selected	9

Table 2.7: Trace of `select(a)`. `a[i].finish` is the finish of the last selected activity.

The program prints `7 8 3 4 6 10`; 6 activities have been selected. This number is indeed the largest: the verification script tried all 1024 subsets and found no compatible set of 7 activities. There are two sets of 6 activities: the one greedy found, *{7, 8, 3, 4, 6, 10}*, and *{1, 8, 3, 4, 6, 10}*. The second

starts with #1 instead of #7. This shows concretely that the greedy-choice property speaks of "at least one" best solution: there is a best solution that does not contain #7, the earliest finisher, but there is also one that does.

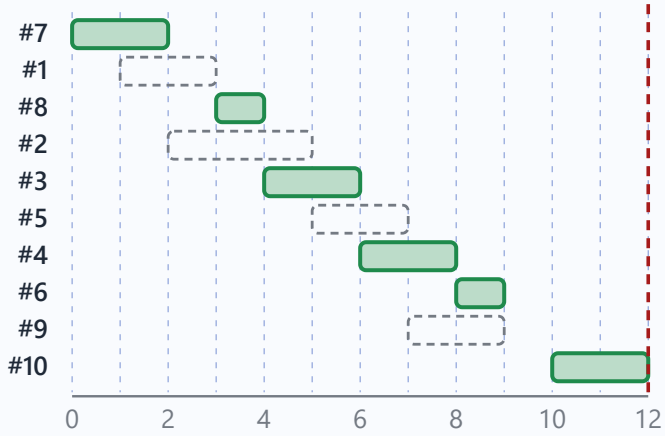


Figure 2.7: The end of the selection. The green activities are selected (#7, #8, #3, #4, #6, #10); those with dashed outlines are skipped. The vertical dashed line is the finish of the last selected activity (12).

Three steps in the trace show the importance of half-open intervals. #3 starts exactly at 4 and #8 ends exactly at 4; #4 starts exactly at 6 and #3 ends exactly at 6; #6 starts exactly at 8 and #4 ends exactly at 8. If the condition were written `>` instead of `>=`, the activities starting at the moment the last selected one ends would be skipped. After #8, #5 [5, 7) would be selected instead of #3; then #6 [8, 9) and #10 [10, 12) would follow, and the result would stay at 5 activities.

The cost is clear. Sorting takes $O(n \log n)$ and the single pass $O(n)$; the total is $O(n \log n)$. If the activities arrive sorted by finish, the sort is unnecessary and the algorithm finishes in $O(n)$ time. As extra memory only the variable `i` is kept.

Example 2.3

Let eleven activities be given as (s, f) pairs: (1, 4), (3, 5), (0, 6), (5, 7), (3, 9), (5, 9), (6, 10), (8, 11), (8, 12), (2, 14), (12, 16). The activities are already sorted by finish. The first activity (1, 4) is selected and the last finish becomes 4. (3, 5) and (0, 6) start before 4 and are skipped. (5, 7) is selected and the last finish becomes 7. (3, 9), (5, 9), and (6, 10) start before 7 and are skipped. (8, 11) is selected and the last finish becomes 11. (8, 12) and (2, 14) are skipped. (12, 16) is selected. The result is 4 activities: (1, 4), (5, 7), (8, 11), (12, 16).

Common mistake

Comparing the new activity not with the last selected one but with the previous activity in the array; that is, writing `a[j].start >= a[j - 1].finish`. The previous activity may have been skipped. #2 [2, 5) at $j = 3$ in the trace shows this: the finish of the previous activity #8 is 4, and

since 2 < 4 the result comes out right. But #3 [4, 6) at $j = 4$ would be compared with 5, the finish of the skipped #2; since 4 < 5 it would be skipped by mistake. The comparison must always be made with the last *selected* activity.

2.3.4 Why is earliest finish correct?

Theorem 2.1 (The greedy choice in activity selection)

Let a_1 be the earliest-finishing activity. The activity selection problem has a best solution that contains a_1. Hence the algorithm "select the earliest finisher, discard the activities that overlap it, repeat with the rest" selects the largest number of activities.

Proof. The proof uses the exchange argument. Let $O = \{b_1, b_2, \ldots, b_k\}$ be a best solution sorted by finish. If $b_1 = a_1$, there is nothing more to say. Otherwise form the set $O' = (O - \{b_1\}) \cup \{a_1\}$. Since a_1 is the earliest-finishing activity, $f(a_1) \le f(b_1)$. Since O is compatible, $f(b_1) \le s(b_2)$. Combining the two inequalities gives $f(a_1) \le s(b_2)$. Since the activities $b_2, \ldots, b_k$ all start after b_2 does, a_1 overlaps none of them. So O' is feasible. The number of elements has not changed either: $|O'| = |O| = k$. Hence O' is also a best solution, and it contains a_1.

For optimal substructure, a_1 and the activities overlapping it are discarded. The remaining activities are those that start after $f(a_1)$, and they form a problem of the same kind. When a_1 is removed from O', the remaining set is a solution of this smaller problem. If the smaller problem had a larger solution, adding a_1 to it would give a solution larger than O'; this contradicts the optimality of O'. By induction, the algorithm's choice at every step remains part of a best solution. □

In the proof the earliest-finishing activity has a special role: it does not finish *later* than the b_1 it replaces. So when it takes b_1's place, it does not obstruct the later activities. The same argument cannot be set up for the earliest-starting activity: the earliest-starting activity may finish very late and may overlap b_2. *[0, 10)* in Figure 2.5 is in exactly this situation.

The same problem can also be solved in the opposite direction. The activities are sorted by start from largest to smallest, and at each step the *latest-starting* activity is selected. This is the same as the earliest-finish criterion with the time axis reversed, and it gives the best result by the same proof. In *weighted* activity selection, a generalization of the problem, each activity has a value and the total value must be as large as possible. In this version no simple greedy criterion is correct; the problem is solved by dynamic programming.

2.4 Job sequencing with deadlines

A workshop has a single machine and several orders waiting in the queue. Each order takes one day on the machine. Each order has a delivery deadline and a profit; if the order is finished by its deadline the profit is earned, and if not, the order is lost. Which orders should the workshop put on which days to make the total profit as large as possible? This problem resembles activity selection, but here all the jobs take the same time, and what matters is not the count but the profit.

2.4.1 The problem definition

Definition 2.6

Job sequencing with deadlines: n jobs are given. Each job takes 1 unit; job i has a deadline d_i and a profit k_i. Time is divided into unit slots numbered 1, 2, 3, ..., and at most one job is done in each slot. If a job is finished by the end of slot d_i, the profit k_i is earned. The goal is to find the set of jobs and the slot assignment with the largest total profit.

The deadline d_i of a job means the job can be placed in any of the slots from 1 to d_i. A job with deadline 1 can be placed only in the first slot. A job with deadline 3 can be placed in the first, second, or third slot. The example in the slides has six jobs (Table 2.8).

Job	1	2	3	4	5	6
deadline	4	1	1	1	3	2
profit	20	10	40	30	50	30

Table 2.8: The six jobs in the slides.

At first glance, three jobs (2, 3, and 4) have deadline 1, and only one of them can be done. Which one to do, and where to put the other jobs, must be considered together.

2.4.2 The greedy criterion and the latest free slot

The greedy algorithm makes two decisions. The first is the order in which the jobs are considered: the jobs are examined in **decreasing order of profit**, that is, the most profitable job first. The second is which slot each job goes in: the job is placed in the **latest free** slot at or before its deadline. If no free slot remains up to its deadline, the job is skipped and never considered again.

The reasoning for the second decision is this. Putting a job in an earlier slot than necessary can block the place of a job whose deadline is closer. For example, if a job with deadline 3 is put in the first slot, the only place that a later job with deadline 1 could go is already taken. Had the same job been put in the third slot, the first slot would have stayed free. The latest free slot leaves the early slots to jobs with closer deadlines and preserves flexibility.

2.4.3 Code and trace

The `sequenceJobs` method in the slides applies these two rules and returns the total profit:

```
static int sequenceJobs(Job[] jobs, int n) {
  Arrays.sort(jobs, (a, b) -> b.profit - a.profit);
  int[] slot = new int[n];
  Arrays.fill(slot, -1);
  int total = 0;
  for (int i = 0; i < n; i++) {
    int j = Math.min(n, jobs[i].deadline) - 1;
    for (; j >= 0; j--) {
      if (slot[j] == -1) {
        slot[j] = jobs[i].id;
        total += jobs[i].profit;
        break;
      }
    }
  }
  return total;
}
```

The `Job` class has the fields `id`, `deadline`, and `profit`. The comparator `b.profit - a.profit` in the `Arrays.sort` call moves the job with the larger profit forward. Thanks to the stable sort, jobs with equal profit keep their original order. The array `slot` holds n slots: `slot[j]` is the number of the job placed in slot $j + 1$, or −1 if it is empty. Because arrays start at 0 in Java, the first slot is in `slot[0]`. n jobs can use at most n slots; deadlines larger than n are therefore lowered to n. The expression `Math.min(n, jobs[i].deadline) - 1` does this and gives the index of the latest slot the job can be put in. The inner loop searches backward from this slot for the first free slot. If a free slot is found, the job is placed, the profit is added to the total, and the search ends with `break`. If the loop descends to $j = -1$, there is no suitable slot and the job is silently skipped.

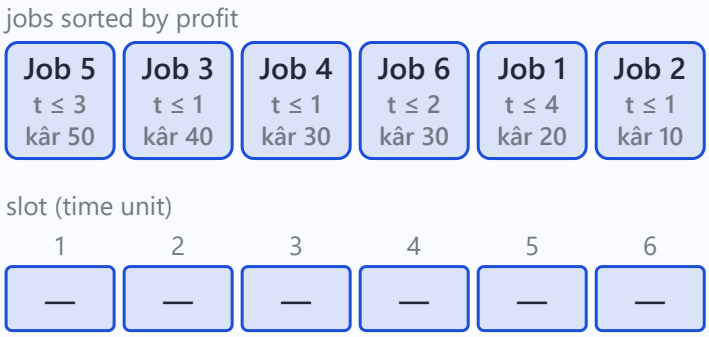


Figure 2.8: The jobs in decreasing order of profit; the six slots are empty. Job 4 and Job 6, with equal profit, keep their original order.

The array sorted by profit is Job 5 (50), Job 3 (40), Job 4 (30), Job 6 (30), Job 1 (20), Job 2 (10). Table 2.9 gives the placement of each job step by step. In the table the slots are numbered from 1.

Job	Deadline	Profit	Slots checked	Decision	Slots 1–6	`total`
5	3	50	3	into slot 3	– – 5 – – –	50
3	1	40	1	into slot 1	3 – 5 – – –	90
4	1	30	1 (full)	skipped	3 – 5 – – –	90
6	2	30	2	into slot 2	3 6 5 – – –	120
1	4	20	4	into slot 4	3 6 5 1 – –	140
2	1	10	1 (full)	skipped	3 6 5 1 – –	140

Table 2.9: Trace of `sequenceJobs(jobs, 6)`.

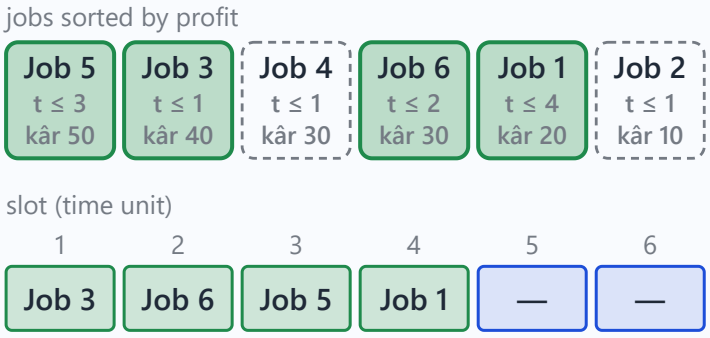


Figure 2.9: The result: slots 1–4 hold Job 3, Job 6, Job 5, Job 1 in turn; the total profit is 140. Job 4 and Job 2 cannot be done.

The method returns 140. The machine does Job 3 on the first day, Job 6 on the second, Job 5 on the third, and Job 1 on the fourth. The fifth and sixth slots stay empty; since no job's deadline exceeds 4, using these slots is not possible anyway. That 140 is the largest total profit was shown in the verification script by trying all subsets of the six jobs.

The interesting step in the trace is Job 4. Its profit is 30, equal to Job 6's, but its deadline is 1, and the first slot has been taken by the more profitable Job 3. There is no other slot for Job 4. Job 6, however, fits in the second slot because its deadline is 2. That one of two jobs with equal profit gets done and the other does not comes from their different deadlines.

Example 2.4

Let five jobs be given as (number, deadline, profit): (1, 2, 100), (2, 1, 19), (3, 2, 27), (4, 1, 25), (5, 3, 15). The order by profit is Job 1, Job 3, Job 4, Job 2, Job 5. Job 1's deadline is 2; slot 2 is free and the job goes there (total 100). Job 3's deadline is also 2. Slot 2 is full, so the one

before it is checked: slot 1 is free and the job goes there (total 127). The deadline of Job 4 and of Job 2 is 1; slot 1 is full and both are skipped. Job 5's deadline is 3; slot 3 is free (total 142). Result: the machine does Job 3, Job 1, and Job 5 in turn, and the total profit is 142.

2.4.4 Wrong criteria

Either of the two decisions can be chosen wrongly. Consider the order of the jobs first. Sorting the jobs by deadline looks natural: let the urgent jobs be done first. But this criterion ignores profit. Consider two jobs with deadline 1: one has profit 10 and the other 40. Only one can be done. In a sort by deadline (with a stable sort) the job that comes first is selected; if the first one is the job with profit 10, the result is 10. Sorting by profit selects the job with 40. On the six jobs in the slides, sorting by deadline with the latest-free-slot rule gives a profit of 110; the profit order gives 140.

Now consider the choice of slot. Suppose the job is put in the *earliest* free slot at or before its deadline. In the six jobs of the slides, Job 5 (deadline 3) goes to the first slot. The only option for the following Job 3 is the first slot, but it is full; the job with profit 40 is lost. At the end the algorithm does Job 5, Job 6, and Job 1 and earns only 100. With the latest-free-slot rule the same jobs yield 140. A two-job example is shorter: let X have deadline 2 and profit 50, and Y have deadline 1 and profit 40. The earliest-free-slot rule puts X in the first slot and loses Y: 50. The latest-free-slot rule puts X in the second slot, and Y fits in the first: 90.

Common mistake

Putting a job in the earliest free slot at or before its deadline. This rule leaves no room for later jobs with close deadlines. The correct rule is the latest free slot. In the code this is ensured by the inner loop starting `j` at the deadline and counting backward; if the loop starts from 0 and counts forward, the algorithm is wrong.

2.4.5 Correctness and cost

The correctness of the algorithm is shown in two parts. The first part is that the latest-free-slot rule rejects a job only if it really does not fit.

Theorem 2.2 (The latest free slot)

Let A be the set of jobs the algorithm has accepted so far and x the next job. If the latest-free-slot rule cannot find a free slot for x, then the jobs in $A \cup \{x\}$ cannot all be completed by their deadlines under any slot assignment.

Proof. Since no free slot was found for x, all the slots from 1 to $d(x)$ are full. Let $t \geq d(x)$ be the largest number such that all the slots from 1 to t are full; slot $t + 1$ is empty, or $t = n$. Every job in slots 1–t has a deadline less than or equal to t. The reason is this: at the moment a job with a deadline greater than t was placed, slot $t + 1$ was empty, because slots are never emptied. The latest-free-slot rule would have put that job in $t + 1$ or a later slot, not between 1 and t. Hence the t jobs in slots 1–t and x, that is, all $t + 1$ jobs, have deadlines of at most t. $t + 1$ jobs do not fit in t slots. □

The second part is that the decreasing profit order is correct, and it is shown with an exchange argument. Let a be the most profitable job, and suppose it can meet its deadline alone. Let O be a best solution that does not contain a. If one of the slots 1–$d(a)$ in O is empty, a is added there and the profit increases; this contradicts the optimality of O. If all are full, the job b in one of these slots is removed and a is put in its place. Since a is placed in a slot at or before its own deadline, the solution is feasible. Since $k(a) \geq k(b)$, the profit does not decrease. So there is a best solution containing a. The slides also recall this result in a more general language: the sets of jobs that can be completed together by their deadlines form a **matroid**, and in matroids the greedy selection in decreasing order of weight is always optimal. The correctness of Kruskal's algorithm is also an instance of the same general result.

The cost is computed as follows. Sorting takes $O(n \log n)$. For each job, the slot search looks at at most $\min(n, d_i)$ slots; if the largest deadline is d_{max}, the search takes $O(n \cdot d_{max})$ in all. Since d_{max} can go up to n, the worst case is $O(n^2)$. For example, if all the jobs have deadline n, job i looks at i slots, and $1 + 2 + \ldots + n = n(n + 1)/2$ probes are made in total. This cost can be reduced with a disjoint-set (union-find) structure. Each full slot is linked to a set that points to the first free slot on its left. The latest free slot of a job is found with a single `find` call; when a slot fills, a `union` is made with the set on the left. With path compression and union by rank, each operation takes nearly constant time, and the slot search drops to nearly $O(n)$ in total. The total cost is then determined by the sort.

2.5 The fractional knapsack

A traveler's backpack can carry at most 50 kilograms. In front of the traveler are several sacks of spices; each sack has a known weight and value. The spice is divisible: if half of a sack is taken, half of its value is gained. What should the traveler put in the pack to carry the largest value? This question is the fractional knapsack problem. When the items cannot be divided, that is, when each item is either taken whole or not at all, the problem is called the 0/1 knapsack. The two problems are very similar, but the greedy approach gives the best result only in the first.

2.5.1 The problem and the ratio criterion

Definition 2.7

Fractional knapsack: n items and a capacity W are given; item i has profit k_i and weight w_i. A portion $0 \le x_i \le 1$ of each item may be taken. The goal is to find the selection that satisfies the condition $\Sigma\, x_i\, w_i \le W$ on the total weight and maximizes the total profit $\Sigma\, x_i\, k_i$.

How "efficient" an item is shows in the profit it brings per unit of weight. This value is called the item's **ratio**: k_i / w_i. The greedy criterion is to start from the item with the largest ratio and take as much of it as fits. If the whole item fits, it is taken whole. If it does not fit, a portion equal to the remaining capacity is taken and the pack is full. In the example in the slides the capacity is $W = 50$ and there are three items (Table 2.10).

Item	Profit	Weight	Ratio
1	60	10	6
2	100	20	5
3	120	30	4

Table 2.10: The three items in the slides ($W = 50$).

The ratios are 6, 5, and 4; the items are already in ratio order. Item 1 is taken whole (10 kg, profit 60). Item 2 is taken whole (20 kg, profit 100). A capacity of 20 kg remains, but item 3 weighs 30 kg. $20/30$ of item 3 is taken, earning $120 \cdot 20/30 = 80$. The total is $60 + 100 + 80 = 240$.

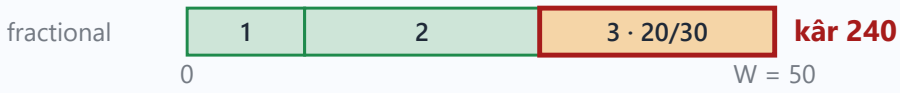


Figure 2.10: Filling the 50 kg pack: items 1 and 2 whole, and 20/30 of item 3. The total profit is 240.

Note that the ratio criterion does not take the item with the largest profit first. Item 3 has the largest profit (120), but it also has the largest weight and brings the least profit per kilogram. Taking in decreasing order of profit selects all of item 3 (30 kg, 120) and all of item 2 (20 kg, 100) and stops at 220. Taking in increasing order of weight gives the same result as the ratio order in this example, but that is a coincidence: it fails on an input containing a heavy, very profitable item and a light but worthless one.

2.5.2 Code and trace

The `fractional` method in the slides sorts the items by decreasing ratio and fills the pack:

```
static double fractional(Item[] s, int W) {
  Arrays.sort(s, (a, b) -> Double.compare(b.ratio(), a.ratio()));
  double profit = 0;
  int left = W;
  for (Item x : s) {
    if (x.weight <= left) {        // fits whole
      left -= x.weight;
      profit += x.profit;
    } else {                         // fractional part
      profit += x.profit * (double) left / x.weight;
      break;
    }
  }
  return profit;
}
```

The `Item` class has the fields `profit` and `weight` and the method `ratio()`; `ratio()` returns `(double) profit / weight`. The comparator `Double.compare(b.ratio(), a.ratio())` moves the item with the larger ratio forward. Here `Double.compare` is used instead of subtraction, because the ratios are fractional, and if the difference of two ratios is converted to an integer, differences like 0.5 are lost. `left` is the capacity remaining in the pack. For each item the loop first checks whether the whole item fits. If it does, it subtracts the weight from the capacity and adds the profit. If it does not, the `else` branch takes a portion of `left / weight` of the item. The pack is now exactly full and the loop exits with `break`. The `(double)` cast matters when computing the profit of the fractional portion: if `left / x.weight` were written as integer division, 20/30 would be rounded down to zero.

Item (ratio)	Condition `weight <= left`	Action	`left`	`profit`
start	—	sort: 1, 2, 3	50	0
1 (6)	10 ≤ 50 true	taken whole	40	60
2 (5)	20 ≤ 40 true	taken whole	20	160
3 (4)	30 ≤ 20 false	20/30 taken: *120 · 20/30 = 80*; `break`	20	240

Table 2.11: Trace of `fractional(s, 50)`.

In the last row `left` stays at 20, because the `else` branch does not update the capacity; since the loop is broken right after, this value is never used again. The method returns `240.0`.



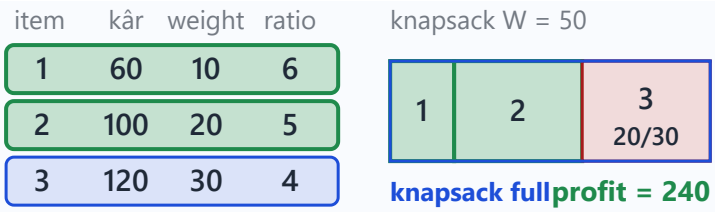


Figure 2.11: The fractional knapsack: the empty pack at the start (top) and the full pack (bottom). Items 1 and 2 are taken whole, and 20/30 of item 3; the pack is full, the profit is 240.

The dominant part of the cost is the sort: $O(n \log n)$. The scan looks at at most n items and takes $O(n)$. Instead of sorting, the item with the largest ratio can be taken from a heap at each step; if the pack fills early, only a few items are removed from the heap, and the total time drops to $O(n + m \log n)$ (m is the number of items taken). There is also a linear-time method based on median finding.

Example 2.5

Let the capacity be $W = 10$ and the items (profit, weight) be (10, 5), (40, 4), (30, 6), (50, 3). The ratios are 2, 10, 5, and about 16.7 respectively. The ratio order is (50, 3), (40, 4), (30, 6), (10, 5). (50, 3) is taken whole: 7 left, profit 50. (40, 4) is taken whole: 3 left, profit 90. (30, 6) does not fit; 3/6 of it is taken and $30 \cdot 3/6 = 15$ profit is added. The total is 105. (10, 5) is not taken at all.

2.5.3 Why is the ratio criterion optimal?

Theorem 2.3 (The ratio criterion in the fractional knapsack)

The greedy algorithm that takes the items in decreasing order of ratio, as much as fits, finds the optimal solution of the fractional knapsack problem.

Proof. Number the items so that $k_1/w_1 \geq k_2/w_2 \geq \ldots \geq k_n/w_n$. If the total weight is less than W, everything is taken and there is nothing to say; so assume the pack gets full. Let G be the greedy solution and O a best solution. If O is different, let i be the first item at which the two solutions differ. Since the greedy method took as much of i as could fit, O has taken less of i. Since the pack of O is also full, this shortfall has been made up by taking extra of some item j that comes after i. In O, δ kilograms of j are removed and δ kilograms of i are put in their place. The weight does not change and the solution is feasible. The profit changes by $\delta \cdot (k_i/w_i - k_j/w_j)$, and this value is nonnegative, because the ratio of i is not smaller than that of j. The change moves O one step closer to G without lowering its profit. When this operation is repeated a finite number of times, O turns into G; so the profit of G is not less than that of O. □

The crucial step of the proof is that weight can be exchanged in small portions. δ kilograms can be removed from one item and added to another, because the items are divisible. Every kilogram of the highest-ratio item brings at least as much profit as any other choice of the same weight. There is therefore no reason not to take it to the end.

2.5.4 The ratio criterion in the 0/1 knapsack

If the items cannot be divided, the exchange step in the proof collapses. Consider the same three items and the 50 kg pack, but this time each item is either taken whole or not at all. The ratio criterion takes item 1 and item 2 (30 kg, profit 160). Item 3 does not fit in the remaining 20 kg and is skipped. The result is 160. Yet items 2 and 3 together weigh exactly 50 kg and bring 220. Comparing all the selections shows that this is the best. The single items bring 60, 100, and 120. The pairs bring *1 + 2 = 160*, *1 + 3 = 180*, and *2 + 3 = 220*. All three items together weigh 60 kg and do not fit.

Method	Selection	Weight	Profit
fractional, ratio criterion	1, 2, and 20/30 of 3	50	240 (optimal)
0/1, ratio criterion	1, 2	30	160
0/1, optimal	2, 3	50	220

Table 2.12: The fractional and 0/1 knapsack with the same items ($W = 50$).

In the 0/1 case the ratio criterion cannot fill the 20 kg gap. In the fractional case this gap is filled by a portion of item 3. In the 0/1 case, portions cannot be taken and the gap stays worthless. The exchange argument fails too: if item 3 is removed from the best solution and replaced by item 1, the weight drops to 30 kg but the profit falls from 220 to 160. Because the weights are not equal, the exchange cannot be made in the form "same weight, more profit". The 0/1 knapsack is solved by dynamic programming in $O(n \cdot W)$ time.

The value of the fractional solution is an upper bound on the value of the 0/1 solution: a 0/1 solution is also a feasible solution of the fractional problem, so it cannot exceed the value of the fractional optimum. In the example $220 \le 240$. This bound is used for pruning in the algorithms that solve the 0/1 knapsack by branch and bound.

Common mistake

Applying the ratio criterion, which is optimal for the fractional problem, to the 0/1 knapsack. The input of the two problems is the same; only the answer to "can an item be divided?" differs. This small difference removes the greedy-choice property. In the 0/1 problem the ratio criterion can be used only as a heuristic.

2.6 The greedy choice in the minimum spanning tree

The minimum spanning tree is one of the most successful applications of the greedy approach. A **spanning tree** of a connected, weighted, undirected graph is a connected, acyclic subgraph that contains all the vertices of the graph; in a graph with V vertices it has exactly $V - 1$ edges. The spanning tree with the smallest sum of edge weights is called the **minimum spanning tree** (MST). Connecting a network of cities with the least cable is a typical application of this problem. The algorithms of Kruskal, Prim, and Borůvka find the MST, and all three are greedy. This section does not describe the algorithms from scratch; it explains why the greedy choice of all three is safe through a single property.

2.6.1 The cut property

Let the vertices of a graph be divided into two nonempty sets: S and $V - S$. This division is called a **cut**. Edges with one endpoint in S and the other in $V - S$ **cross** the cut. Every spanning tree must contain at least one edge crossing the cut; otherwise no connection can be made between S and $V - S$.

Theorem 2.4 (Cut property)

Take any cut of a connected, weighted, undirected graph. If e is the edge of smallest weight among the edges crossing the cut, then there is an MST that contains e. If the lightest edge crossing the cut is unique, this edge is in every MST.

Proof. The proof is again an exchange argument. Let T be an MST that does not contain $e = (u, v)$, with $u \in S$ and $v \in V - S$. Since T is a tree, there is a single path from u to v. Since this path starts in S and ends in $V - S$, it crosses the cut at least once; let e' be an edge it crosses by. e is added to T and e' is removed. Removing e' splits the tree into two parts, and e reconnects these two parts; the result is again a spanning tree. The weight changes by $w(e) - w(e')$. Since e is the lightest edge crossing the cut, this difference is not greater than zero. Since the new tree is no heavier than T, it too is an MST and contains e. If e is the unique lightest edge, the difference is strictly negative; in that case T could not be an MST, and there is no MST that does not contain e. □

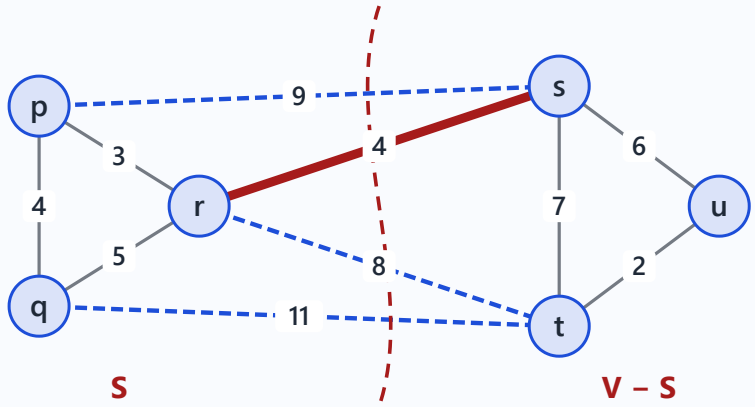


Figure 2.12: The cut property. The dashed curve separates the vertices into $S = \{p, q, r\}$ and $V - S = \{s, t, u\}$. The edges crossing the cut (blue dashed and red) are 9, 4, 8, and 11. The lightest, r–s (4), is in the MST.

In the graph of Figure 2.12, four edges cross the cut: p–s (9), r–s (4), r–t (8), and q–t (11). The lightest is r–s, and since it is the unique lightest edge, it is in every MST. The MST of the graph indeed consists of the edges t–u (2), p–r (3), p–q (4), r–s (4), and s–u (6); the total weight is 19. Of the edges crossing the cut, only r–s enters the tree. This is no accident: it suffices for the MST to connect S and $V - S$ by exactly one edge, and that edge must be the cheapest.

The meaning of the cut property for greedy algorithms is this. With the edges it has selected so far, the algorithm has built some components. A cut is chosen that separates one of these components from the others. None of the selected edges crosses this cut. If the lightest edge crossing the cut is added, the result, together with the selected edges, is still part of an MST. The exchange in the proof works in this case too, because the removed e' crosses the cut and is therefore not one of the selected edges. For this reason the algorithm never has to undo any of its choices.

If the weights are all different, the lightest edge of every cut is unique and the MST is unique as well. In the graph in the figure, p–q and r–s both have weight 4, so the weights are not all different; even so, the MST of this graph is unique. That all weights are different is sufficient for uniqueness but not necessary. Equal weights can lead to more than one MST, but all the MSTs have the same total weight.

Example 2.6

In a graph with four vertices, the vertices are split into $\{A, B\}$ and $\{C, D\}$; the edges crossing the cut are A–C 5, B–C 3, and B–D 4. The lightest edge crossing the cut is B–C (3) and it is the unique lightest edge; by the cut property it is in every MST. For A–C and B–D the cut property alone says nothing. Whether they are in the MST depends on the remaining edges of the graph. For example, if the edges A–B and C–D also exist and are light, the MST crosses the cut with B–C alone.

Common mistake

Reading the cut property as "the lightest edges are always safe". What is safe is the lightest edge *crossing a cut*. If the lightest edges of the whole graph are taken in order, a cycle can form: the third edge of the cycle reconnects two vertices that the first two already connect, and it crosses no cut. Kruskal's algorithm therefore discards the edge that forms a cycle; every cut that separates the endpoints of the discarded edge is already crossed by one of the selected edges.

2.6.2 Kruskal, Prim, and Borůvka: three cuts

The three MST algorithms use the same property with different cuts. The difference is which cut is chosen at each step. Table 2.13 sets the three algorithms side by side.

	Kruskal	Prim	Borůvka
Greedy choice	the lightest edge in the whole graph that forms no cycle	the lightest edge leaving the tree	the lightest edge leaving each component
Growth	forest → tree	a single tree, vertex by vertex	a forest, merging in rounds
Cut used	the two components that the edge's endpoints separate	the tree versus the remaining vertices	each component versus the rest
Data structure	sorting and disjoint sets	priority queue	disjoint sets
Time	$O(E \log V)$	$O(E \log V)$ (binary heap)	$O(E \log V)$
Suited to	sparse graphs, edge lists	dense graphs, adjacency structures	parallel or distributed computing

Table 2.13: The greedy choice of Kruskal, Prim, and Borůvka.

Kruskal examines the edges in increasing order of weight. If the next edge connects two different components, it is added; if it connects two vertices of the same component, it is skipped because it would form a cycle. The cut used for the added edge is the one between the component of one of its endpoints and the remaining vertices. If a lighter edge crossed this cut, it would have been examined earlier and added; so the added edge is the lightest of the cut. The component check is done with a disjoint-set structure. The sort determines the time: $O(E \log E)$. Since $E \le V^2$, $\log E \le 2 \log V$, and this time is also written $O(E \log V)$.

Prim grows a single tree starting from one vertex. At each step the lightest edge joining a vertex in the tree to a vertex outside it is added. The cut used is the one between the vertices of the tree and the remaining vertices. The cheapest connection of each vertex outside the tree to the tree is kept in a priority queue. With a binary heap the time is $O((V + E) \log V)$; in a connected graph this equals $O(E \log V)$. On dense graphs a version with array scanning can be faster, at $O(V^2)$.

Borůvka works in rounds. In each round every component finds the lightest edge leaving it, and all these edges are added together. Each component uses its own cut, namely the cut "this component versus everything else". Since each round at least halves the number of components, there are at most $\lceil \log_2 V \rceil$ rounds, and each round takes $O(E)$; the total is $O(E \log V)$. Because the components' choices are independent of one another, the algorithm suits parallel execution. If weights can be equal, a consistent tie-breaker is needed; otherwise two components may pick different edges of equal weight and form a cycle.

The three algorithms find the same MST on the same graph, provided the weights are distinct. If the weights are distinct, the MST is unique and it is the tree all of them find. If there are equal weights, they may find different trees, but their total weights are the same. They differ in the order in which they consider the edges and in the data structure they need. When all the edges have the same weight, for instance, every spanning tree is an MST; Kruskal in this case produces some spanning tree depending on the order of the edge list.

The weight of an edge alone does not say whether it will be in the MST. The edges that are in every MST are those that are the unique lightest edge of some cut. The edges that are in no MST are those that are the unique heaviest edge of some cycle; this second rule is called the **cycle property**, and the reverse-delete algorithm is based on it. Adding the same constant to all edges does not change the MST, because the order of the edges is preserved and, since every spanning tree has $V - 1$ edges, all the totals increase by the same amount. If the edges are processed in decreasing order, Kruskal in the same way finds the maximum spanning tree.

2.7 Huffman coding

A text is stored on a computer as bits. Each character is assigned a bit string, that is, a **codeword**, and the text is formed by writing these codes one after another. In encodings such as ASCII every character's code has the same length. In real texts, however, characters do not occur equally often. In a Turkish text the letters "a" and "e" appear very often and "ğ" and "j" very rarely. If frequent characters are given short codes and rare characters long codes, the total length of the text shrinks. Huffman coding is a greedy algorithm that applies this idea in the best possible way.

2.7.1 Fixed-length and variable-length codes

Huffman coding is a **lossless** data compression method: when the compressed data is decoded, the original data comes back bit for bit. David Huffman developed the method in 1952, as a student, for a term paper. The example text in the slides is 25 characters long and contains only six distinct characters: F, O, R, G, E, and T. To tell six characters apart, a fixed-length code needs $\lceil \log_2 6 \rceil = 3$ bits; 2 bits can distinguish only four characters. In this case the text takes $25 \times 3 = 75$ bits. If every character were stored in 8-bit ASCII, it would take 200 bits.

In a variable-length code the codewords have different lengths. The total length of the text is the sum, over characters, of frequency times code length:

> Total bits = Σ freq(c) · length(c)

To reduce this sum, the codes of the most frequent characters must be kept short. But codes cannot be shortened arbitrarily; the shortened codes must still be uniquely decodable.

2.7.2 Prefix-free codes

Let the codes be a = 0, b = 01, c = 1. Given the bit string "01", it cannot be told whether it is "ab" or just "b". The source of the problem is that the code of a, 0, is at the beginning of the code of b, 01. The problem can be solved by placing a separator between codewords, but the separator also spends bits. A better solution is for no code to be the beginning of another code.

Definition 2.8

A code in which no codeword is a prefix of another codeword is called a **prefix-free code**. A bit string written with a prefix-free code is decoded uniquely by reading from left to right, without using a separator.

Prefix-free codes are represented by binary trees. One starts from the root; a left edge means 0 and a right edge means 1. Each character sits at a leaf, and the bits on the path from the root to that leaf are the character's code. If a character stood at an internal node, its code would be a prefix of the codes of the leaves below it. Since characters stand only at leaves, no code can be a prefix of another. The length of a code is the depth of the character in the tree.

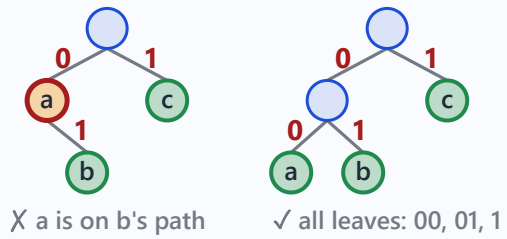


Figure 2.13: On the left, a is at an internal node; its code 0 is a prefix of the code 01 of b, and decoding is ambiguous. On the right, all characters are at leaves; the codes 00, 01, and 1 are prefix-free.

Decoding is done on the tree. One starts at the root; as each bit is read, one descends to the left child if it is 0 and to the right child if it is 1. When a leaf is reached, the character at that leaf is written and one returns to the root. The moment a codeword ends is the moment a leaf is reached, and this moment is known exactly. For example, with the tree on the right of Figure 2.13, the string "00110" is read as 00 | 1 | 1 | 0…: the first two bits give a, the next bit gives c, and the next gives c again. The last 0 does not reach a leaf by itself; the string came in incomplete.

The tree of an optimal prefix-free code is a **full** binary tree: every internal node has two children. If there were an internal node with a single child, that node could be skipped and its child moved up; all the codes below it would become one bit shorter and the codes would remain prefix-free. In the left tree in the figure, the node a, the left child of the root, has a single child; the right tree is better for this reason.

Example 2.7

Consider four sets of codes. *{0, 10, 110, 111}* is prefix-free: the only code beginning with 0 is 0, the only code beginning with 10 is 10, and 110 and 111 are not prefixes of each other. This set corresponds to a tree with four leaves, extending to the right. *{0, 01, 110, 111}* is not prefix-free; 0 is a prefix of 01. *{1, 10, 00, 01}* is not prefix-free; 1 is a prefix of 10. *{00, 01, 0, 11}* is not prefix-free; 0 is a prefix of both 00 and 01.

2.7.3 The steps of the algorithm

The Huffman algorithm builds the tree from the bottom up. Each character starts as a single-node tree with its own frequency. At each step the two trees with the smallest frequencies are merged: they become the children of a new internal node, and the frequency of the new node is the sum of its children's frequencies. When a single tree remains, that tree is the Huffman tree. The steps are listed as follows:

1. The frequency of each character is computed.
2. A leaf is created for each character. The two nodes with the smallest frequencies are merged into a new internal node; this step is repeated until a single node remains.
3. In the tree, the code of each character is read off the path from the root to its leaf: left edge 0, right edge 1.
4. The text is encoded with these codes.

The greedy choice is in the second step: at each merge, the two trees with the smallest frequencies at that moment are selected. The leaves of these two trees sink one level deeper after the merge, that is, their codes grow by one bit. Lengthening the codes of the rarest characters is the choice that increases the total number of bits the least. The frequencies in the example in the slides are given in Table 2.14.

Character	**F**	**O**	**R**	**G**	**E**	**T**
Frequency	2	3	4	4	5	7

Table 2.14: Character frequencies in the example text (25 characters in all).

For *n* characters exactly *n − 1* merges are made, because each merge reduces the number of trees by one and one goes from *n* trees to one tree. The Huffman tree therefore has *n* leaves and *n − 1* internal nodes. In the example there are six characters and five merges.

2.7.4 Building the tree: code and trace

The `huffman` method in the slides builds the tree with a priority queue:

```
static Node huffman(char[] ch, int[] fr) {
  PriorityQueue<Node> queue =
    new PriorityQueue<>((a, b) -> a.f - b.f);
  for (int k = 0; k < ch.length; k++)
    queue.add(new Node(ch[k], fr[k]));
  while (queue.size() > 1) {
    Node left = queue.poll();
    Node right = queue.poll();
    Node parent = new Node('\0', left.f + right.f);
    parent.left = left;
    parent.right = right;
    queue.add(parent);
  }
  return queue.poll();
}
```

The `Node` class has four fields: the character `ch`, the frequency `f`, and the left and right child. In leaves the children are `null`. The queue is a `PriorityQueue` built with the comparator `(a, b) -> a.f - b.f`. This Java class is a min-heap; each call of `poll()` removes the node with the smallest frequency. The first loop creates a leaf for each character and adds it to the queue. The `while` loop runs as long as the queue holds more than one node. In each round `poll()` is called twice; the first node removed becomes the left child and the second the right child. The character of the new internal node is `'\0'`, because internal nodes carry no character; its frequency is the sum of the two children. The new node is added back to the queue. When the loop ends, a single node remains in the queue, and it is the root of the tree.

Round	Left (1st `poll`)	Right (2nd `poll`)	New node	Queue at end of round
start	—	—	—	F2, O3, R4, G4, E5, T7
1	F2	O3	(5) = 2 + 3	R4, G4, E5, (5), T7
2	G4	R4	(8) = 4 + 4	E5, (5), T7, (8)
3	(5)	E5	(10) = 5 + 5	T7, (8), (10)
4	T7	(8)	(15) = 7 + 8	(10), (15)
5	(10)	(15)	(25) = 10 + 15	(25)

Table 2.15: Trace of `huffman(ch, fr)`. The numbers in parentheses are internal nodes.

The queue contents in the table are written in frequency order; the actual layout of the heap in its array is different. There are ties in the second and third rounds. In the second round G4 and R4 have the same frequency, and in the third round (5) and E5 do. Which comes out first depends on the internal layout of the heap, and therefore on the order in which the characters were added to the queue. The table matches the trace in the slides. When the characters are added in the order F, O, R, G, E, T, Java's `PriorityQueue` removes E5 first in the third round, and E becomes the left child. In that case the codes become E = 00, F = 010, O = 011; the remaining codes do not change. The code lengths and the total number of bits are the same in both cases: E has 2 bits, F and O have 3 bits.

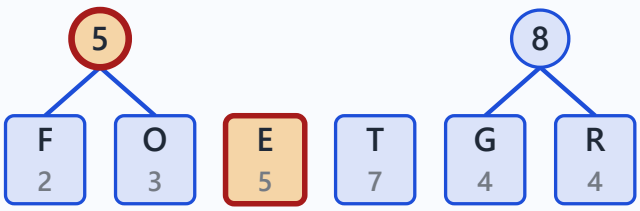


Figure 2.14: The start of the third round: the internal nodes (5) and (8) have been built. The next two smallest nodes are (5) and E5 (highlighted).

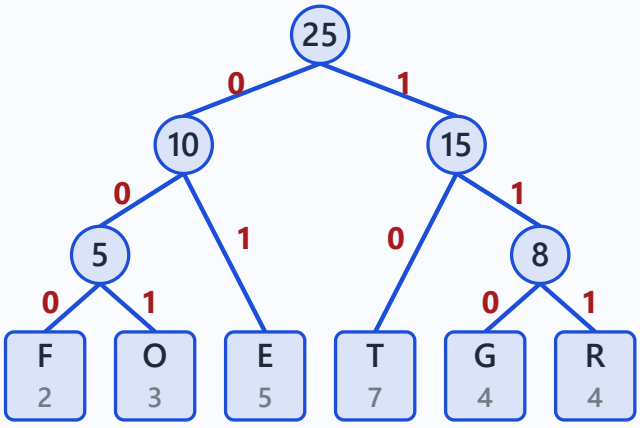


Figure 2.15: The completed Huffman tree. The root is 25; left edges are labeled 0 and right edges 1.

The structure of the tree can be read from the table. The left child of the root is (10) and the right child is (15). The children of (10) are (5) and E, and the children of (15) are T and (8). Under (5) are F and O, and under (8) are G and R. E and T are at depth 2; F, O, G, and R are at depth 3. The two most frequent characters have received the shortest codes.

The total bit cost of the tree also equals the sum of the frequencies of the internal nodes: *5 + 8 + 10 + 15 + 25 = 63*. The reason is this: a character contributes once to the frequency of every internal node on the path from the root to its leaf. The number of internal nodes on the path is the depth of the character. Thus the frequency of each character is counted as many times as its depth, and the total equals *Σ frequency · depth*. This observation also explains why Huffman picks the two smallest frequencies at every merge: the cost of each merge is the frequency of the new internal node, and the algorithm keeps this cost as small as possible at every step.

The cost is as follows. Adding the *n* leaves to the queue takes *O(n log n)*. The loop runs *n − 1* times and in each round does two `poll` calls and one `add`; each is *O(log n)*. The total is *O(n log n)*. Here *n* is the number of distinct characters, not the length of the text. If the frequencies are sorted in advance, a version that runs in *O(n)* time with two ordinary queues can also be written: the first queue holds the sorted leaves, and the second holds the internal nodes in the order they are created. Since the frequencies of the internal nodes created arrive in increasing order, the second queue stays sorted by itself.

Example 2.8

Let the frequencies of six characters be a:5, b:9, c:12, d:13, e:16, f:45 (100 in all). The merges are, in order: a5 + b9 = 14; c12 + d13 = 25; (14) + e16 = 30; (25) + (30) = 55; f45 + (55) = 100. The left child of the root 100 is f and the right child is (55). The codes are f = 0, c = 100, d = 101, a = 1100, b = 1101, e = 111. The total number of bits is *45 · 1 + 12 · 3 + 13 · 3 + 16 · 3 + 5 · 4 + 9 · 4 = 224*. The sum of the internal nodes gives the same result: *14 + 25 + 30 + 55 + 100 = 224*. With a fixed 3-bit code the text would take 300 bits; the saving is about 25%. In this example f is so dominant that it receives a one-bit code.

2.7.5 Generating the codes: code and trace

Once the tree is built, the code of every character is found by traversing the tree once. The recursive `genCodes` method in the slides does this:

```
static void genCodes(Node d, String code) {
  if (d.left == null) {      // leaf
    System.out.println(d.ch + " = " + code);
    return;
  }
  genCodes(d.left, code + "0");  // left: 0
  genCodes(d.right, code + "1"); // right: 1
}
```

The method takes a node and the bit string accumulated on the way to that node. If the node is a leaf, that is, if its left child is `null`, the character and its code are printed. Because the Huffman tree is a full binary tree, it suffices to look at the left child alone: internal nodes have both children, and leaves

have none. If the node is internal, the method descends first to the left child with "0" appended to the code, then to the right child with "1" appended. This is a preorder traversal; the leaves are printed in order from left to right. The first call has the form `genCodes(root, "")`.

Call	Node	`code`	What happens?
1	(25), root	""	internal node; descend left with "0"
2	(10)	"0"	internal node; descend left with "00"
3	(5)	"00"	internal node; descend left with "000"
4	F	"000"	leaf: `F = 000` is printed
5	O	"001"	leaf: `O = 001` is printed
6	E	"01"	leaf: `E = 01` is printed
7	(15)	"1"	internal node; descend left with "10"
8	T	"10"	leaf: `T = 10` is printed
9	(8)	"11"	internal node; descend left with "110"
10	G	"110"	leaf: `G = 110` is printed
11	R	"111"	leaf: `R = 111` is printed

Table 2.16: Trace of `genCodes(root, "")` (in call order).

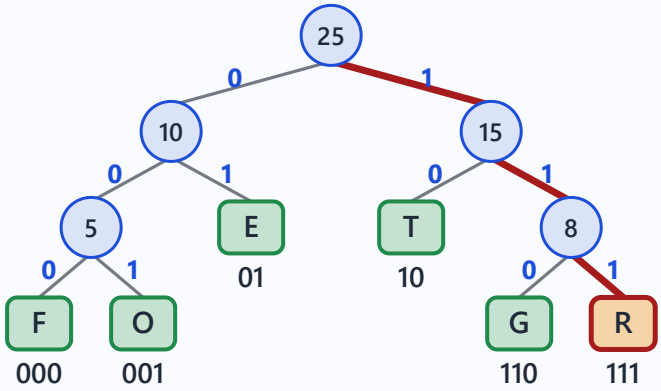


Figure 2.16: The end of code generation. The code is written under each leaf; the highlighted path is that of R = 111, the last one printed.

The method is called 11 times in all: once each for 6 leaves and 5 internal nodes. Since each node is visited once, the traversal makes $O(n)$ node visits. The recursion depth equals the height of the tree; in the example there are 3 edges from the root to the deepest leaf. The string concatenation `code + "0"` creates a new string at each call and takes time proportional to its length. If this cost is also counted, the total work is proportional to the sum of the lengths of all the codes. If the codes are kept in a `StringBuilder` or in a character array, this extra cost is avoided.

Common mistake

If the text has a single distinct character (for example "AAAA"), the Huffman tree consists of a single node: the root is also a leaf. In this case `genCodes(root, "")` prints the empty string as the code of the character, and the text looks as if it were encoded in 0 bits. The decoder cannot know how many characters there are. In this special case the character must be given a code of at least 1 bit (for example "0").

2.7.6 Encoding, decoding, and the saving

Table 2.17 gives the code of each character and its total contribution in the text.

Character	Frequency	Code	Length	Bits (frequency · length)
F	2	000	3	6
O	3	001	3	9
E	5	01	2	10
T	7	10	2	14
G	4	110	3	12
R	4	111	3	12
Total	25			**63**

Table 2.17: Huffman codes and the bit count.

The Huffman code writes the text in 63 bits; the fixed 3-bit code uses 75 bits. The saving is *(75 − 63) / 75 = 16%*. On average *63 / 25 = 2.52* bits fall to each character. The saving depends on how unbalanced the frequencies are. If all the characters occurred equally often, the saving would be very small; if one character is very dominant, as in the a–f example, the saving grows.

For **encoding**, the code of each character is written in order. The word "FORTE" becomes *000 | 001 | 111 | 10 | 01*, that is, 13 bits. With the fixed code the same word would take *5 × 3 = 15* bits. No separator is placed between the bits during encoding; the vertical bars here are only to make reading easier.

For **decoding**, the tree is used. Take the bit string "1100110". One starts at the root. With 1, go right to (15); with 1, go right to (8); with 0, go left to G. G is written and one returns to the root. With 0, go left to (10); with 1, go right to E. E is written and one returns to the root. With 1, go right to (15); with 0, go left to T. T is written. The result is *110 | 01 | 10*, that is, "GET". Decoding descends one edge per bit; a string of *m* bits is decoded in *O(m)* time.

The placement of the left and right children is arbitrary. For example, T could be made the right child of (15). In that case the codes change, but the code lengths and the total of 63 bits do not. For the same reason, which of the nodes with equal frequencies is chosen first can change the code lengths, but not the total number of bits. A text can have more than one optimal prefix-free code.

Compressed data cannot be decoded on its own; the same tree or code table must also be in the decoder's hands. The tree (or the code length of each character) is therefore sent along with the compressed data. On short texts this extra information can eat up the whole gain. In the 25-character example, the code table for six characters takes far more space than the 12 bits gained. Huffman coding therefore makes sense on long texts and large files.

There is also a theoretical lower bound on how good a code can be. If the proportion of the character c in the text is p_c, the average number of bits per character cannot fall below $H = -\Sigma\, p_c \log_2 p_c$. This value is called the **entropy** of the text. The entropy of the example text is about 2.48 bits; the Huffman code uses 2.52 bits. The average of a Huffman code always lies between the entropy and the entropy plus one bit.

Common mistake

Trying to decode Huffman-compressed data without the tree or code table. The bit string does not say which character was written with which code. The encoder and the decoder must use the same tree. Two trees that differ in the placement of left and right children, or in how ties are broken, produce different codes from the same frequencies.

2.7.7 Why is the Huffman code optimal?

The cost of a prefix-free code tree T is $B(T) = \Sigma\, f(c) \cdot d_T(c)$, where $f(c)$ is the frequency of the character and $d_T(c)$ its depth in the tree. That the tree Huffman builds minimizes this cost is shown in two steps: the greedy choice and optimal substructure.

Theorem 2.5 (The greedy choice in Huffman coding)

Let x and y be the two characters with the smallest frequencies. There is an optimal code tree in which x and y are sibling leaves at the deepest level.

Proof. Let T be an optimal tree. An optimal tree is a full binary tree; there are therefore two leaves at the deepest level that are siblings of each other. Let them be a and b. Let the positions of x and a be swapped. The cost changes by:

$$B(T') - B(T) = f(x) \cdot d(a) + f(a) \cdot d(x) - f(x) \cdot d(x) - f(a) \cdot d(a) = -(f(a) - f(x)) \cdot (d(a) - d(x))$$

Since x is the character with the smallest frequency, $f(a) - f(x) \geq 0$. Since a is at the deepest level, $d(a) - d(x) \geq 0$. The product is nonnegative, and with the minus sign in front, the difference cannot be greater than zero: $B(T') \leq B(T)$. The same exchange is done for y and b. In the end x and y are siblings at the deepest level and the cost has not increased; the tree is still optimal. □

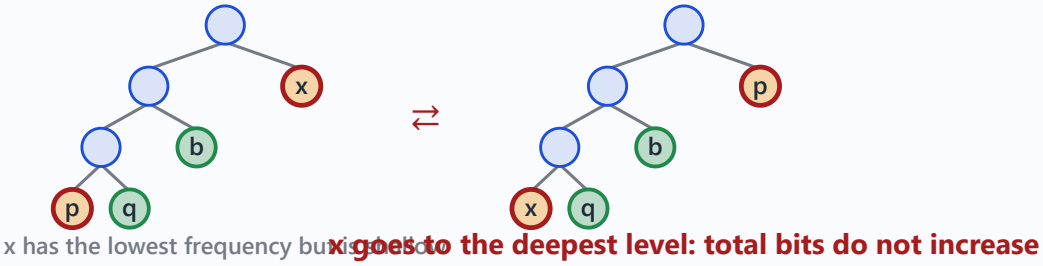


Figure 2.17: The exchange argument. On the left, the lowest-frequency x is at a shallow place and p is at the deepest. On the right, the two have been swapped; x sinks to the deepest level and the total number of bits does not increase.

Theorem 2.6 (Optimal substructure in Huffman coding)

Let x and y be replaced by a new character z with frequency $f(z) = f(x) + f(y)$, and let T' be an optimal tree for the reduced alphabet. If the leaf z in T' is replaced by an internal node whose children are x and y, an optimal tree T for the original alphabet is obtained.

Proof. x and y are one level deeper than z. Hence $B(T) = B(T') - f(z) \cdot d(z) + (f(x) + f(y)) \cdot (d(z) + 1) = B(T') + f(x) + f(y)$. Suppose T is not optimal: there is a T'' with smaller cost. By the previous theorem, x and y can be assumed to be siblings in T''. If these two leaves are deleted in T'' and their parent is made the leaf z, a tree for the reduced alphabet results whose cost is $B(T'') - f(x) - f(y) < B(T')$. This contradicts the optimality of T'. □

Together, the two theorems give the correctness of the Huffman algorithm by induction. In the first step the algorithm merges the two smallest frequencies; the greedy-choice theorem says there is an optimal tree containing this choice. The two merged characters behave like a single character z, and the problem shrinks by one character. The optimal-substructure theorem says that an optimal tree for the original problem is obtained from the optimal tree of the smaller problem. Thus the Huffman code is, among prefix-free codes, the one that minimizes the sum Σ *frequency · depth*.

2.7.8 Cost and applications

The cost of all the stages of Huffman coding is as follows. In a text of m characters, let there be n distinct characters. Counting frequencies reads the text once: $O(m)$. Building the tree takes $O(n \log n)$ and generating the codes takes $O(n)$ node visits. Encoding reads the text once more and takes each character's code from the table: $O(m)$ characters and work proportional to the number of output bits. Usually n is far smaller than m (for example, $n \le 256$ for one-byte characters); the total time is therefore in practice directly proportional to the length of the text.

Huffman coding is used not alone but often together with other methods. The DEFLATE format, used by ZIP and gzip files and PNG images, first shortens repeated strings with backward pointers, then encodes the resulting symbols with Huffman. In the JPEG image and MP3 audio formats too, the data

is first transformed and quantized; the entropy coding of the last stage is done with Huffman codes. LZW is dictionary-based: it gives codes to repeated strings. Huffman is frequency-based: it gives codes to single characters according to their frequencies. The difference between the two approaches can be seen by comparing the results of RLE and LZW with those of Huffman on the same texts.

2.8 Other greedy algorithms

The greedy idea underlies many other algorithms as well. Some of them give a provably optimal result. Others do not guarantee the best result, but it is known how far the result can be from the best.

Dijkstra and Prim. Both algorithms take the node with the smallest key from a priority queue at each step and select it permanently. In Dijkstra the key is the shortest known distance from the source to that node; in Prim it is the weight of the cheapest edge connecting the node to the tree. A selected node is never changed again. Dijkstra finds shortest paths correctly in graphs with no negative-weight edges; if there is a negative edge, a distance counted as final can later become shorter and the greedy choice errs. Prim, since it rests on the cut property, is correct regardless of the sign of the weights.

Graph coloring. Coloring a graph with the fewest colors, subject to neighboring vertices getting different colors, is an NP-hard problem. Greedy coloring takes the vertices in some order and gives each vertex the smallest color not used by its neighbors. It is fast, but the number of colors used depends on the order. Consider a cycle of six vertices: a1 – b2 – a3 – b1 – a2 – b3 – a1. This cycle can be colored with two colors: the a's one color, the b's the other. If the vertices are colored in the order a1, a2, a3, b1, b2, b3, two colors indeed suffice. But if the order is a1, b1, a2, b2, a3, b3, then since a1 and b1 are not neighbors, both get color 1. Since a2 is a neighbor of b1, it gets 2, and since b2 is a neighbor of a1, it also gets 2. a3 is a neighbor of both b1 (1) and b2 (2) and gets 3; b3 gets 3 for the same reason. A bad order has caused one extra color to be used. Greedy coloring uses at most "the maximum degree plus one" colors; this bound always holds.

Bin packing. Items of given sizes are to be placed into the fewest bins of fixed capacity. This problem is also NP-hard. The *first fit* rule puts each item into the first bin it fits in; if it fits in none, it opens a new bin. The *best fit* rule puts the item into the bin that is left with the least space after it is placed. Let the capacity be 10 and the items 2, 5, 4, 7, 1, 3, 8. First fit in this order opens four bins: [2, 5, 1], [4, 3], [7], [8]. If the items are first sorted from largest to smallest (*first fit decreasing*), three bins suffice: [8, 2], [7, 3], [5, 4, 1]. Since the total size is 30, fewer than three bins is impossible. It has been proved that first fit decreasing uses at most about $11/9 \cdot OPT + 1$ bins, where OPT is the minimum number of bins.

Maximum coverage. A universe of elements and subsets of these elements are given; at most k sets are to be chosen so as to cover as many elements as possible. The greedy algorithm at each step chooses the set that covers the most elements not yet covered. It has been proved that this method

covers at least $1 - 1/e \approx 0.632$ of what the optimum covers, that is, about 63% of the best. An algorithm guaranteeing a better ratio in polynomial time does not exist unless P = NP.

These last examples show the role of greedy algorithms in NP-hard problems. Finding the best solution is very expensive, but a greedy algorithm quickly produces a solution and gives a proven bound on the quality of this solution. Such algorithms are called **approximation algorithms**; the worst-case ratio of the solution to the optimum is called the **approximation ratio**.

2.9 Comparison and common mistakes

Table 2.18 compares the problems studied in the notes by greedy criterion, time, and optimality.

Problem	Greedy criterion	Time	Optimal?
Coin change	the largest coin that fits	$O(k + count)$	only in canonical systems
Activity selection	earliest finish	$O(n \log n)$	yes
Job sequencing	the most profitable job, latest free slot	$O(n^2)$	yes
Fractional knapsack	largest profit/weight ratio	$O(n \log n)$	yes (no in the 0/1 version)
MST	the lightest edge crossing a cut (Kruskal, Prim, Borůvka)	$O(E \log V)$	yes
Huffman	the two lowest-frequency trees	$O(n \log n)$	yes

Table 2.18: Comparison of greedy algorithms.

The last column of the table sums up the most important lesson of the greedy approach. In five problems the greedy algorithm is optimal, and this is proved by an exchange argument in each. In coin change the result depends on the coin system. In the 0/1 knapsack the ratio criterion misses the best. The $O(n^2)$ in job sequencing is the worst case of the direct slot search in the slides; with disjoint sets this too drops to $O(n \log n)$.

The mistakes treated in detail in different places in these notes can be gathered as follows:

- **Counting greedy as correct without a proof.** With the coins $\{1, 3, 4\}$ and the amount 6, greedy uses 3 coins and the best solution 2. For every problem the greedy-choice property must be shown; a few successful trials are not a proof.
- **A wrong criterion.** In activity selection, choosing the earliest-starting, the shortest, or the least-conflicting activity does not guarantee the best result. The correct criterion is earliest finish.
- **The ratio criterion in the 0/1 knapsack.** The ratio criterion, which is optimal for the fractional problem, does not guarantee the best when the items cannot be divided.
- **The earliest free slot.** In job sequencing, putting a job in the earliest free slot at or before its deadline leaves no room for jobs with close deadlines. The latest free slot must be chosen.

- **Misreading the cut property.** What is safe is the lightest edge crossing a cut. If the lightest edges of the whole graph are taken blindly, a cycle can form; Kruskal therefore discards the edge that forms a cycle.
- **Decoding without the tree in Huffman coding.** Without the code table or the tree, the data cannot be decoded. In a text with a single character the root is a leaf; the code must be given at least 1 bit.

2.10 Summary

- **Greedy algorithm:** builds the solution step by step, at each step makes the choice that looks best at that moment according to a criterion, and does not undo its decisions. The typical structure is a sort and a single pass; the time is mostly $O(n \log n)$.
- **Conditions for correctness:** by the greedy-choice property, there is a best solution that contains the first choice. By optimal substructure, the best solution of the remaining problem, together with the first choice, gives the best solution of the whole. If the two properties hold together, the greedy algorithm is optimal; if not, it is only a heuristic.
- **The exchange argument:** a part of an arbitrary best solution is replaced by the greedy choice; the solution is shown to remain feasible and not get worse. The proofs for activity selection, job sequencing, the fractional knapsack, the cut property, and Huffman use this pattern.
- **Coin change:** the largest coin that fits is chosen; $O(k + count)$. It is optimal in canonical systems (e.g. $\{50, 25, 10, 5, 1\}$); the amounts 6 and 10 in the system $\{4, 3, 1\}$ and the amount 14 in the system $\{10, 7, 1\}$ are counterexamples. The best result is found by the recurrence $minCoins[t] = 1 + min\ minCoins[t - p]$ in $O(k \cdot T)$ time.
- **Activity selection:** the activities are sorted by finish; every activity that starts after the finish of the last selected is selected. $O(n \log n)$; 6 of the ten activities in the slides are selected.
- **Job sequencing:** the jobs are placed in decreasing order of profit, each in the latest free slot at or before its deadline. By direct search the worst case is $O(n^2)$; with disjoint sets, nearly the sorting time. On the six jobs in the slides the total profit is 140.
- **Fractional knapsack:** the items are taken in decreasing order of profit/weight ratio, as much as fits; a portion of the last item may be taken. $O(n \log n)$; 240 in the example. In the 0/1 version the same criterion gives 160, and the best is 220.
- **MST and the cut property:** the lightest edge crossing a cut belongs to an MST. Kruskal (a cut separating two components), Prim (the tree versus the rest), and Borůvka (each component versus the rest) use this property with different cuts; all three are $O(E \log V)$.
- **Huffman coding:** the two lowest-frequency trees are merged repeatedly; left edge 0, right edge 1. The codes are prefix-free and the sum $\Sigma\ frequency \cdot depth$ is minimal. In the example, 63 bits (75 with a fixed code); the tree is built in $O(n \log n)$ time and must be sent to the decoder.

- **Approximation algorithms:** on NP-hard problems such as graph coloring, bin packing, and maximum coverage, greedy algorithms do not guarantee the best but produce good solutions with proven ratios.

2.11 Key terms

greedy algorithm greedy-choice property optimal substructure exchange argument heuristic coin change problem canonical coin system activity selection job sequencing with deadlines matroid fractional knapsack 0/1 knapsack minimum spanning tree (MST) cut cut property Huffman coding prefix-free code compression entropy priority queue graph coloring approximation algorithm approximation ratio

---

2.12 Test Yourself

1. Let the coin system be *{9, 6, 5, 1}*. How many coins does greedy change use for the amounts 11 and 12? What are the best solutions? Is this system canonical?

   Answer

   For 11 greedy uses 3 coins with *9 + 1 + 1*; the best is 2 coins with *6 + 5*. For 12 greedy uses 4 coins with *9 + 1 + 1 + 1*; the best is 2 coins with *6 + 6*. The system is not canonical. 11 is the smallest amount on which greedy fails in this system, and it is smaller than 15, the sum of the two largest coins.

2. Eight activities are given as *(s, f)* pairs and numbered 1 to 8: (0, 3), (1, 2), (2, 5), (3, 4), (4, 7), (5, 6), (6, 9), (8, 10). Which activities does the earliest-finish criterion select?

   Answer

   Order by finish: #2 (1, 2), #1 (0, 3), #4 (3, 4), #3 (2, 5), #6 (5, 6), #5 (4, 7), #7 (6, 9), #8 (8, 10). #2 is selected (last finish 2). #1 starts at 0 and is skipped. #4 starts at 3 and is selected (4). #3 is skipped. #6 starts at 5 and is selected (6). #5 is skipped. #7 starts at 6 and is selected (9). #8 starts at 8, and since 8 < 9 it is skipped. The result is 4 activities: #2, #4, #6, #7. If all subsets are tried, no compatible set of 5 activities is found.

3. In activity selection, does the algorithm that selects the *latest-starting* activity at each step give the

best result? Justify your answer.

Answer

Yes. If the time axis is reversed (each *[s, f)* interval becomes *[−f, −s)*), the latest-starting activity turns into the earliest-finishing one, and the compatibility relation does not change. The exchange argument for earliest finish holds unchanged. The latest-starting a can be put in place of the last activity of a best solution. a does not start earlier than that activity and does not overlap the earlier ones.

4. Seven jobs are given as (name, deadline, profit): (A, 3, 35), (B, 4, 30), (C, 4, 25), (D, 2, 20), (E, 3, 15), (F, 1, 12), (G, 2, 5). How does greedy job sequencing fill the slots, and what is the total profit? What would the result be if jobs were placed in the earliest free slot?

Answer

The profit order is A, B, C, D, E, F, G. A → slot 3; B → slot 4; C's deadline is 4, slots 4 and 3 are full, so it goes to slot 2; D's deadline is 2, slot 2 is full, so it goes to slot 1. No suitable free slot is left for E, F, and G. Slots 1–4: D, C, A, B; the total profit is $20 + 25 + 35 + 30 = 110$, the best result. With the earliest-free-slot rule, A goes to slot 1, B to slot 2, C to slot 3; there is no room for D, E, F, and G, and the total stays at 90.

5. Let the capacity be $W = 20$ and the items (profit, weight) be (48, 12), (30, 10), (28, 10), (5, 1). What is the value of the fractional knapsack? If the items could not be divided, what would the ratio criterion find, and what would the best result be?

Answer

The ratios are 4, 3, 2.8, and 5. The ratio order is (5, 1), (48, 12), (30, 10), (28, 10). (5, 1) and (48, 12) are taken whole: 13 kg, profit 53. For the remaining 7 kg, 7/10 of (30, 10) is taken: 21. The fractional value is 74. In the 0/1 case the ratio criterion takes (5, 1) and (48, 12); neither of the next two items fits in the remaining 7 kg, and the result is 53. The best 0/1 selection is (30, 10) with (28, 10): exactly 20 kg, profit 58.

6. Let the frequencies be A:1, B:1, C:2, D:3, E:5, F:8. Find the merges of the Huffman tree, the code lengths, and the total number of bits. Compare with a fixed-length code.

Answer

The merges: $1 + 1 = 2$; $2 + 2 = 4$; $3 + 4 = 7$; $5 + 7 = 12$; $8 + 12 = 20$. Because each merge includes the result of the previous merge, the tree extends in a single direction. The code lengths are F 1, E 2, D 3, C 4, and A and B 5 bits. The total is $8 \cdot 1 + 5 \cdot 2 + 3 \cdot 3 + 2 \cdot 4 + 1 \cdot 5 + 1 \cdot 5 = 45$ bits; the sum of the internal nodes also gives $2 + 4 + 7 + 12 + 20 = 45$. For six characters a fixed code needs 3 bits: $20 \times 3 = 60$ bits.

7. Let the codes be F = 000, O = 001, E = 01, T = 10, G = 110, R = 111. To what text does the bit string 11001101001111 decode? How many bits would the same text take with a fixed 3-bit code?

   Answer

   Reading from the left: 110 → G, 01 → E, 10 → T, 10 → T, 01 → E, 111 → R. The text is "GETTER". It takes 14 bits with Huffman and $6 \times 3 = 18$ bits with the fixed code. Since no code is a prefix of another, the place where each code ends is known exactly.

8. Are the following two statements true? (a) In a connected graph with distinct weights, the lightest edge is in every MST. (b) The heaviest edge is in no MST.

   Answer

   (a) True. A cut is formed by putting one endpoint of the lightest edge in S and the other in $V - S$. This edge is the lightest of the edges crossing the cut, and since it is the unique lightest, it is in every MST by the cut property. (b) False. The heaviest edge can be a bridge: if the graph splits in two when it is deleted, it has to be in every spanning tree. For an edge to be excluded from the MST, it must be the unique heaviest edge of some cycle.

9. At which step does the exchange argument given for the fractional knapsack fail in the 0/1 knapsack?

   Answer

   The proof removes δ kilograms of a low-ratio item from the best solution and puts δ kilograms of a high-ratio item in its place. This exchange preserves the weight and does not reduce the profit. In the 0/1 case items cannot be taken in portions; only whole items can be exchanged. When two items of different weights are exchanged, either the capacity is exceeded or a worthless gap is left in the pack. Either the feasibility condition or the "no worse" condition breaks.

10. Let all of $n = 10^6$ jobs have deadline n. How many slot probes does job sequencing with direct slot search make? About how many with disjoint sets?

    Answer

    Job i starts at the latest slot and searches backward for the first free slot; because the previous $i - 1$ jobs have filled the last slots, it looks at i slots. The total is $1 + 2 + \ldots + n = n(n + 1)/2 \approx 5 \cdot 10^{11}$ probes. With disjoint sets, each job is placed with a few `find` and `union` calls, and the total work is nearly linear, that is, on the order of a few million steps. In this case the time is determined by the sort, which makes about $n \log_2 n \approx 2 \cdot 10^7$ comparisons.

2.13 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 15: activity selection, the elements of the greedy strategy, and the optimality of Huffman codes, given in full form for the proofs summarized here.
- J. Kleinberg and É. Tardos, *Algorithm Design* (2006), Chapter 4: interval scheduling, the "greedy stays ahead" and exchange-argument proof techniques, and counterexamples to the fewest-conflicts criterion are discussed in detail.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 5.5: a complete Java implementation of Huffman compression, including writing and reading the tree to a file.
- D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes" (1952): the short and readable original paper in which the method was first published.

2.14 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 15: Greedy Algorithms; Chapter 21: Minimum Spanning Trees).
2. D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE* 40(9), 1098–1101, 1952.
3. J. Kleinberg, É. Tardos, *Algorithm Design*, Addison-Wesley, 2006 (Chapter 4: Greedy Algorithms).
4. D. Kozen, S. Zaks, "Optimal Bounds for the Change-Making Problem", *Theoretical Computer Science* 123(2), 377–388, 1994.
5. J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proceedings of the American Mathematical Society* 7(1), 48–50, 1956.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 4.3: Minimum Spanning Trees; Section 5.5: Data Compression).

Dynamic Programming

Some recursive solutions are correct but unbearably slow. A method that computes Fibonacci numbers straight from the definition makes more than three hundred million calls for fib(40), although it has only 41 distinct questions to answer. The same questions are asked again and again, and the answers are recomputed from scratch every time. Dynamic programming removes this waste: each subproblem is solved once, its answer is written into a table, and later it is simply read back. These notes first introduce the two conditions of the method, its design steps, and its two ways of implementation (memoization and tabulation). Then the classic problems are taken up one by one: 0/1 and unbounded knapsack, subset sum, longest increasing subsequence, longest common substring, and longest common subsequence. For each problem we write the state and recurrence, fill the table by hand, trace the Java code line by line, and read the solution back from the table. The notes close with other classic examples such as edit distance, coin change, maximum subarray, and matrix-chain multiplication.

LEARNING OUTCOMES

Recognition

Recognize overlapping subproblems and optimal substructure in a problem; decide when dynamic programming is appropriate and when divide and conquer or a greedy approach is enough.

Formulation

Define a subproblem (a state), write the recurrence and its base cases, and determine the size of the table and the order in which it is filled.

Implementation

Fill in the tables for Fibonacci, 0/1 and unbounded knapsack, subset sum, longest increasing subsequence, longest common substring, and longest common subsequence by hand, and implement them in Java.

Analysis

Compare memoization with tabulation; justify the running times $O(n \cdot W)$, $O(m \cdot n)$, $O(n^2)$, and $O(n \log n)$; explain pseudo-polynomial time; reconstruct the solution itself from the table.

3.1 Basic concepts

3.1.1 What is dynamic programming?

A common idea in algorithm design is to split a big problem into smaller problems of the same kind. Divide and conquer uses this idea when the pieces are independent: merge sort splits the array in two, sorts the two halves separately, and merges them. The two halves share no work. In other problems the pieces are entangled. The subproblems of the big problem share subproblems of their own. If such a problem is solved by plain recursion, the same small question is asked many times and answered from scratch each time.

Dynamic programming (DP for short) removes this repetition. The solution of each subproblem is written into a table. When the same subproblem is met again, nothing is computed; the answer is read from the table. The method is used especially for **optimization problems**, which look for the largest or smallest of something: the largest value, the shortest path, the fewest operations. But the same idea also works for counting and decision problems: "how many different paths are there?", "is there such a subset?"

Definition 3.1

Dynamic programming is an algorithm design method that splits a problem into smaller **subproblems**, solves each subproblem only once, and reuses the solutions by storing them in a table. In short: DP = recursion + remembering.

This definition has an important consequence. Behind every dynamic programming solution stands a recursive solution. DP does not solve the problem in a different way; it runs the same recursion without the needless repetition. Therefore the first and hardest step of DP design is finding the right recursion. The table is merely a device that stores the values of that recursion.

The name of the method comes from Richard Bellman in the 1950s. The word "programming" in the name does not mean writing code. In those years the word meant planning and scheduling; the name linear programming comes from the same root. Programming in dynamic programming means planning decisions step by step with a table.

3.1.2 Overlapping subproblems

Two properties are needed for a problem to benefit from dynamic programming. The first is that the subproblems overlap.

Definition 3.2

A problem has **overlapping subproblems** if its recursive solution calls the same subproblem more than once. In other words, the number of distinct subproblems is small, but the number of times the recursion calls them is large.

The best-known example is the Fibonacci numbers. If the definition $F(n) = F(n - 1) + F(n - 2)$ is translated directly into code, computing fib(5) calls fib(4) and fib(3). fib(4) in turn calls fib(3), so fib(3) is computed twice. The same thing grows exponentially at smaller arguments: fib(2) is computed three times and fib(1) five times. Figure 3.1 shows these repetitions.

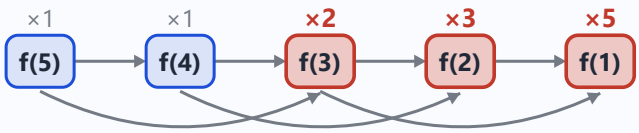


Figure 3.1: How many times each subproblem is called while computing fib(5). Arrows mean "calls"; the red boxes are subproblems that are solved more than once.

The gap between two numbers decides the gain of DP. For fib(5) there are only six distinct subproblems, fib(0) through fib(5), yet plain recursion makes 15 calls in total. As n grows, the number of distinct subproblems stays at $n + 1$ while the number of calls grows exponentially. As long as there is repetition to save, the table closes this gap.

If the subproblems do not overlap, a table is useless. In merge sort the two halves of the array are disjoint pieces. No work done to sort the left half is repeated on the right half. A subarray is sorted only once, so an answer written to a table would never be read a second time. Such problems belong to divide and conquer.

Common mistake

It is often assumed that adding a table to any recursive solution speeds it up. A table helps only when the same subproblem is called again. In binary search or merge sort every subproblem is solved once, and a table merely wastes memory.

3.1.3 Optimal substructure

The second property is that an optimal solution of the big problem can be built from optimal solutions of the smaller problems. Without it, the subproblem answers in the table are not enough to build the answer to the big problem.

Definition 3.3

A problem has **optimal substructure** if an optimal solution of the problem can be built from optimal solutions of its subproblems. Bellman stated this property as the **principle of optimality**: every sub-solution contained in an optimal solution is itself optimal for its own subproblem.

An example makes this property clear. Suppose we look for the **maximum-sum subarray** of an array, where a subarray is a piece made of adjacent elements. Let $V(i)$ be the largest sum of a piece that ends exactly at element i. The best piece ending at i either consists of $a[i]$ alone or is obtained by appending $a[i]$ to a piece that ends at $i - 1$. In the second case the piece ending at $i - 1$ must also be the best one; if a better one existed, using it would increase the sum. The recurrence then follows by itself:

$$V(i) = \max(a[i],\ \ V(i - 1) + a[i])$$

Relations of this kind are also called **Bellman equations**: the value of a state is computed from the optimal value of a previous subproblem. The answer is the largest of all $V(i)$ values. This method is Kadane's algorithm, and it runs in $O(n)$ time.

Optimal substructure does not hold in every problem. Suppose we look for the *longest simple path* (a path with no repeated vertex) between two vertices of a graph. If the longest simple path from u to w passes through a vertex v, its u–v part need not be the longest simple path from u to v. A longer u–v path may use vertices from the second half of the path. Joining two "best" pieces then produces a path that is not simple. The subproblems cannot be solved independently, which is why no efficient DP is known for this problem. For the shortest-path problem the situation is different: every piece of a shortest path is itself a shortest path. Bellman–Ford and Floyd–Warshall rely on this property.

Common mistake

It is not enough to spot overlapping subproblems and build a table right away. One must first show that combining optimal solutions of the subproblems yields an optimal solution of the big problem. Without optimal substructure, the table produces a fast but wrong answer.

3.1.4 Design steps

At first glance dynamic programming solutions look very different from one another: one fills a two-dimensional table, another gets by with a single array. Yet all of them are designed with the same steps. If the steps are followed in order, the solution of a new problem often emerges by itself.

1. **Examine the structure of an optimal solution.** What is the last decision in an optimal solution? Which smaller problem remains once that decision is made?

2. **Write the state and the recurrence.** Express the subproblem as a **state** defined by a few integers, e.g. "the first i items and capacity c". Write the value of a state in terms of smaller states (the recurrence). Give separately the values of the smallest states on which the recurrence rests (the base cases).
3. **Determine the order of computation.** When a cell is computed, the cells it needs must already be ready.
4. **Determine the size of the table.** How many dimensions are needed, and how many values does each dimension take? In knapsack, for example, the table is $(n + 1) \times (W + 1)$.
5. **Fill the table.** Either with bottom-up loops (tabulation) or with top-down recursion (memoization).
6. **Reconstruct the solution.** In most problems we want not only the optimal value but also the choices that produce it. The choices are found by reading the table backwards.

These steps have a simple consequence for running time. Each state is computed once, and computing a state tries a few options. Hence the total time is roughly the number of states times the time to compute one state:

> *total time ≈ (number of states) × (time per state)*

Every complexity in these notes follows from this rule. In knapsack there are $(n + 1)(W + 1)$ states, each computed in $O(1)$ from two options: $O(n \cdot W)$. In longest increasing subsequence there are n states, but each looks at all earlier elements: $O(n^2)$. In matrix-chain multiplication each of the $O(n^2)$ states tries $O(n)$ split points: $O(n^3)$.

3.1.5 Memoization and tabulation

A DP recurrence can be run in two ways. Both solve each subproblem once; the difference is in the order in which, and the tool with which, the subproblems are solved.

Memoization works **top-down**. The recursive solution is written as it is; only the results are kept in a cache. The cache can be an array or a `HashMap`. At the start of every call the table is consulted: if the answer is there, it is returned at once; otherwise it is computed, stored, and then returned. The computation starts at the big problem and descends to the small problems it needs, so only the subproblems that are really needed are solved. The price is recursion: every call has a cost, and deep recursion can cause a stack overflow (`StackOverflowError`).

Tabulation works **bottom-up**. It starts from the smallest subproblem and fills the table in order with loops. There is no recursion and no call overhead, so tabulation is usually faster in practice. In return, the filling order must be chosen correctly in advance: when a cell is reached, the cells it depends on must be ready. Tabulation has one more advantage. If the recurrence uses only the previous row, a single row can be stored instead of the whole table. In knapsack this reduces memory from $O(n \cdot W)$ to $O(W)$.

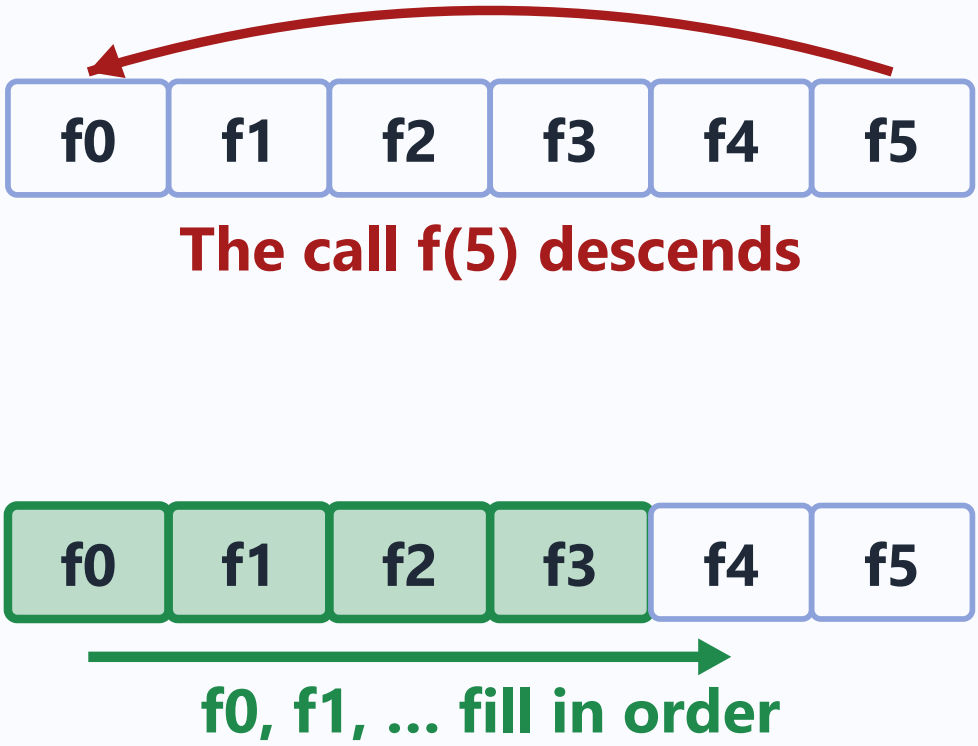


Figure 3.2: Left, memoization: the call fib(5) descends top-down and values are written on the way back. Right, tabulation: f0, f1, … fill from left to right in order.

	Memoization (top-down)	Tabulation (bottom-up)
Style	recursive solution + cache	nested loops
Subproblems solved	only those needed	every cell of the table
Order	determined by the recursion itself	must be chosen correctly in advance
Extra cost	call overhead, risk of stack overflow	none
Reducing memory	hard	easy if the previous row suffices
Asymptotic time	the same: each subproblem is solved once	

Table 3.1: Comparison of memoization and tabulation.

Which one to choose depends on the problem. If the state space is large but few states are actually used, memoization does less work. 0/1 knapsack is an example: on an instance with four items and capacity 6, the memoized version computes only 12 of the 28 states. If the whole table is needed anyway and the filling order is clear, tabulation is both faster and safer.

Common mistake

It is sometimes said that "memoization is always faster than tabulation" or the other way round. Their asymptotic times are the same. Tabulation is often faster in practice because it makes no calls; memoization can come out ahead when it skips unnecessary states.

3.1.6 DP, divide and conquer, and greedy algorithms

Dynamic programming is understood better when it is considered together with two familiar design methods. All three split the problem into smaller pieces; they differ in the relation between the pieces and in how the choice is made.

	Divide and conquer	Dynamic programming	Greedy
Subproblems	independent	overlapping	a single choice at each step
Choice	none; split and combine	all options are tried, the best goes into the table	the best option at the moment is taken, never revisited
Optimality	—	guaranteed if optimal substructure holds	only if the greedy-choice property holds
Examples	merge sort, binary search	0/1 knapsack, LCS, Bellman–Ford	fractional knapsack, Dijkstra, Kruskal
Cost	recurrence, master theorem	number of states × work per state	usually $O(n \log n)$: sorting + one pass
Memory	recursion stack	a table (often $O(n \cdot W)$ or $O(m \cdot n)$)	usually $O(1)$ extra

Table 3.2: Divide and conquer, dynamic programming, and greedy algorithms.

A greedy approach can be seen as a dynamic program that looks at only one option at each step. DP tries all options in a state and writes the best into the table. A greedy algorithm takes the option that looks best at the moment and never examines the others. If it can be proved that this choice always leads to an optimal solution (the greedy-choice property), no table is needed and the algorithm runs much faster. If it cannot be proved, a greedy algorithm may give a wrong answer. 0/1 knapsack is the typical example.

3.1.7 Applications

Beyond being a theoretical tool, dynamic programming runs inside many programs that are used every day.

- **Shortest paths.** Maps and routing protocols: Bellman–Ford works with the "shortest path using at most k edges" subproblem, and Floyd–Warshall with the "shortest path using only the first k vertices as intermediates" subproblem.
- **String comparison.** `diff` and `git` find the difference between two files with the longest common subsequence; spell checkers generate suggestions using edit distance.

- **Bioinformatics.** Alignment of DNA and protein sequences (Needleman–Wunsch, Smith–Waterman) and determining the folding structure of RNA.
- **Resource allocation.** Knapsack, budget and investment planning, production-line scheduling.
- **Compilers and numerical computing.** The order of a matrix-chain product, optimal binary search trees, CYK parsing for context-free grammars.
- **Artificial intelligence.** The Viterbi algorithm in game-playing programs, speech recognition, and machine translation; the Bellman equation in reinforcement learning.

3.2 Fibonacci: memoization and tabulation

The recursive definition of the Fibonacci sequence and the method that translates it directly into code are well known. The goal here is not to explain recursion again. Fibonacci is used to show the two forms of dynamic programming at the smallest scale: the same recurrence is run first with memoization and then with tabulation, and their costs are compared.

3.2.1 The cost of plain recursion

The Fibonacci sequence runs 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... The base cases are $F(0) = 0$ and $F(1) = 1$; for $n \geq 2$, $F(n) = F(n - 1) + F(n - 2)$. A method that copies the definition word for word is correct but very slow.

```
static long fib(int n) {
  if (n <= 1)
    return n;         // base cases
  return fib(n - 1) + fib(n - 2);
}
```

Let $T(n)$ be the number of calls the method makes. Each call counts as 1 for itself and makes two subcalls: $T(n) = T(n - 1) + T(n - 2) + 1$, $T(0) = T(1) = 1$. The solution of this recurrence is $T(n) = 2 \cdot F(n + 1) - 1$, which is easily verified by induction. Because Fibonacci numbers grow like powers of the golden ratio, the number of calls is $\Theta(\varphi^n)$, where $\varphi \approx 1.618$. Table 3.3 shows how quickly the gap opens.

n	F(n)	Plain recursion: 2 · F(n + 1) − 1	Memoization: 2n − 1
5	5	15	9
10	55	177	19
20	6,765	21,891	39
30	832,040	2,692,537	59

n	F(n)	Plain recursion: 2 · F(n + 1) − 1	Memoization: 2n − 1
40	102,334,155	331,160,281	79

Table 3.3: Number of calls made while computing fib(n) with plain recursion and with memoization (counted in Java).

fib(40) makes more than three hundred thirty million calls. For fib(50) the number grows to about $4 \cdot 10^{10}$, which takes minutes on an ordinary computer. Yet computing fib(n) involves only $n + 1$ distinct subproblems: fib(0), fib(1), …, fib(n). The overwhelming majority of the calls ask a question that has already been answered.

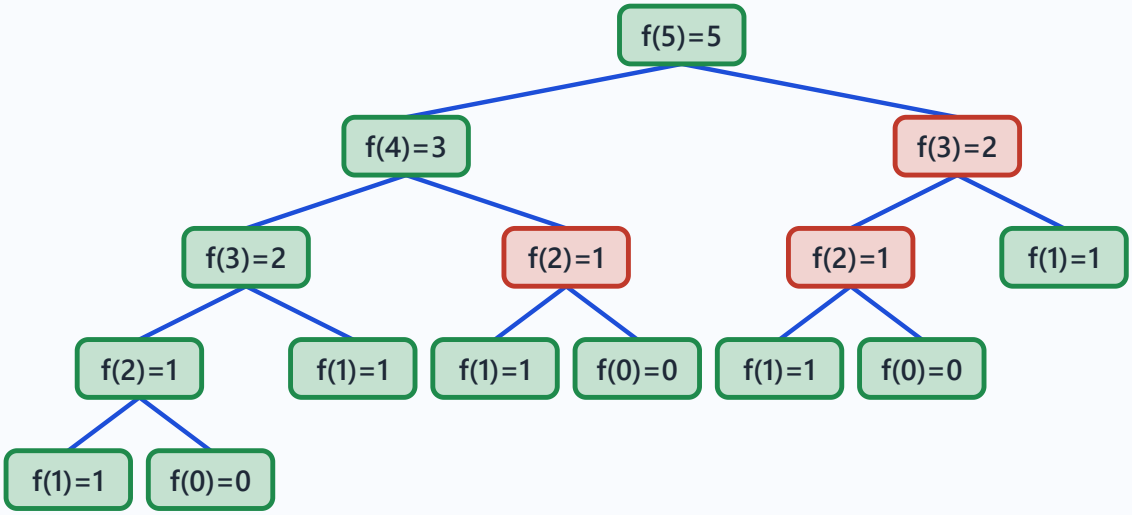


Figure 3.3: The recursion tree of fib(5) after all calls have finished. There are 15 calls; the red boxes mark places where a subproblem that was already solved is computed again.

In the tree of Figure 3.3, fib(3) appears twice, fib(2) three times, fib(1) five times, and fib(0) three times. These counts are themselves Fibonacci numbers: in the tree of fib(n), fib(k) ($k \geq 1$) is called exactly $F(n - k + 1)$ times. The amount of repetition grows as fast as the number being computed. This observation is the heart of dynamic programming: when the number of distinct subproblems is small but the recursion calls them many times, storing the results brings a large gain.

3.2.2 Memoization

Memoization adds two lines to the body of the method. Every computed value is written into the `memo` array. Before computing, a call looks into the array; if the value is there, it returns at once.

```
static long[] memo = new long[50];

static long fib(int n) {
  if (n <= 1) return n;          // base
  if (memo[n] != 0) return memo[n];
  long a = fib(n - 1);
  long b = fib(n - 2);
  memo[n] = a + b;
  return memo[n];
}
```

Because the `memo` array is declared `static`, all calls share the same array. In Java all elements of a new `long` array are 0, and the code uses 0 to mean "not yet computed". This choice is safe because *F(n)* is never 0 for $n \geq 2$, and the base cases are never written into the array. Since the array has length 50, the code can compute at most fib(49). Choosing `long` as the return type is also deliberate: *F(47) = 2,971,215,073* exceeds `int`'s largest value, $2^{31} - 1 = 2{,}147{,}483{,}647$.

The lines work as follows. The first `if` handles the base cases. The second `if` is the memo check: if `memo[n]` is nonzero, the value has been computed before and is returned as is. If it is missing, two subcalls are made, the sum is written to `memo[n]`, and it is returned. Table 3.4 gives a trace of the call fib(5).

Step	Call	Event	memo[2..5]
1	fib(5)	not in the table; fib(4) is called first	— — — —
2	fib(4)	not in the table; fib(3) is called first	— — — —
3	fib(3)	not in the table; fib(2) is called first	— — — —
4	fib(2)	not in the table; fib(1) is called first	— — — —
5	fib(1)	base case, returns 1	— — — —
6	fib(0)	base case, returns 0; `memo[2] = 1 + 0 = 1`	1 — — —
7	fib(1)	from within fib(3); base case, returns 1; `memo[3] = 1 + 1 = 2`	1 2 — —
8	fib(2)	from within fib(4); `memo[2] = 1` is ready, not recomputed; `memo[4] = 2 + 1 = 3`	1 2 3 —
9	fib(3)	from within fib(5); `memo[3] = 2` is ready; `memo[5] = 3 + 2 = 5`	1 2 3 5

Table 3.4: Trace of fib(5) with memoization. The "memo" column gives the values of `memo[2..5]` after the step (— means not computed).

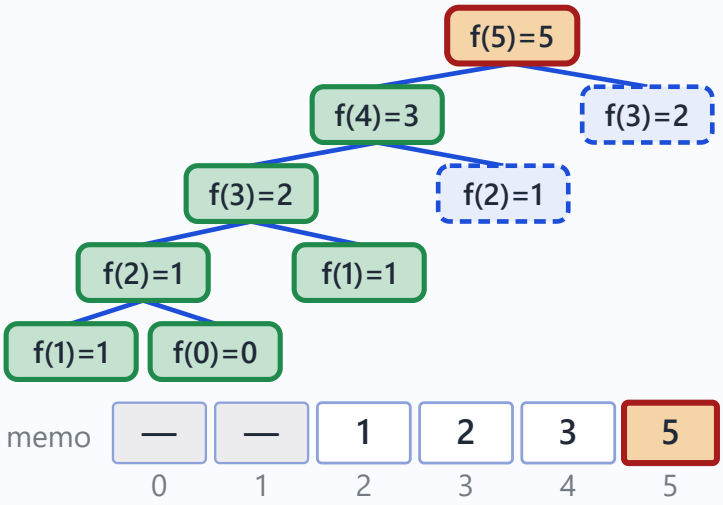


Figure 3.4: fib(5) with memoization: the calls that are actually expanded and the filled `memo` array. The dashed boxes are the two calls answered straight from the table.

The tree in Figure 3.4 is now a chain. The left arm descends from fib(5) to fib(1). At every level the right child is either a base case or is read directly from the table. In total 9 calls are made. In general every value from fib(2) to fib(n) is computed once. The calls are counted as follows: fib(n) and fib(n − 1) are called once each. For $1 \le k \le n - 2$, fib(k) is called twice: once it is expanded, and once it is answered immediately from the table or as a base case. fib(0) is called once. The total is $2 + 2(n - 2) + 1 = 2n - 1$. The exponential time has dropped to linear. The extra memory is $O(n)$: the `memo` array plus a call stack of n frames at its deepest.

Example 3.1

How many calls do fib(6) with plain recursion and with memoization make?

Plain recursion: $2 \cdot F(7) - 1 = 2 \cdot 13 - 1 = 25$ calls. In the tree, fib(2) appears five times and fib(1) eight times. Memoization: fib(6) and fib(5) are called once each. fib(4), fib(3), fib(2), and fib(1) are called twice each; the second calls return immediately from the table or as a base case. fib(0) is called once. The total is $2 + 8 + 1 = 11 = 2 \cdot 6 - 1$ calls.

Common mistake

The cache is created as a local variable inside the method. The line `long[] memo = new long[50];` allocates a new, empty array on every call. No value is shared, and the running time stays exponential. The cache must live where all calls can see it: a `static` field, an object field, or an array passed as a parameter.

3.2.3 Tabulation

The same recurrence can also be run with a loop. fib(i) depends only on fib(i − 1) and fib(i − 2). If the array is filled from left to right, the two cells each cell needs are ready when it is reached.

```
static int fibonacci(int n) {
  if (n <= 1) return n;   // n = 0 and n = 1
  int[] fib = new int[n + 1];
  fib[0] = 0;   // base cases
  fib[1] = 1;
  for (int i = 2; i <= n; i++)
    fib[i] = fib[i - 1] + fib[i - 2];
  return fib[n];
}
```

The method first creates an array of *n + 1* elements; in Java all elements start at 0. After the base cases are written, the loop fills every cell from *i = 2* to *n* with one addition. For fibonacci(9) the cells become 1, 2, 3, 5, 8, 13, 21, and 34 in turn. The last cell, `fib[9] = 34`, is returned. Each cell is computed once: the time is *O(n)* and the memory is *O(n)*.



Figure 3.5: The `fib` array after fibonacci(9) finishes. The cells were filled from left to right, one addition each.

The first line is a small but necessary detail. Without it, the array would have a single element for *n = 0*, and the line `fib[1] = 1` would throw an `ArrayIndexOutOfBoundsException`. `if (n <= 1) return n;` handles the two base cases before the array is built, so the code works for every *n ≥ 0*. Also, since the array is `int`, fibonacci(47) overflows and returns a negative number; in Java this result is −1,323,752,223.

3.2.4 O(1) space and a comparison

In the tabulated version, computing `fib[i]` reads only the last two cells of the array. Older cells are never looked at again. So there is no need to store the whole array; two variables holding the last two values are enough.

```
static long fib(int n) {
  if (n <= 1) return n;
  long prev = 0, curr = 1;
  for (int i = 2; i <= n; i++) {
    long next = prev + curr;
    prev = curr;
    curr = next;
  }
  return curr;
}
```

At each turn of the loop, the variables `prev` and `curr` are $F(i-2)$ and $F(i-1)$. The new value is computed and the variables are shifted by one step. When the loop ends, `curr` is $F(n)$. The time is still $O(n)$ and the extra memory is $O(1)$. The `long` type suffices up to $F(92) \approx 7.5 \cdot 10^{18}$; $F(93)$ overflows.

Method	Time	Extra memory
Plain recursion	$\Theta(\varphi^n)$	$O(n)$ stack
Memoization	$O(n)$	$O(n)$ table + stack
Tabulation	$O(n)$	$O(n)$ table
Two variables	$O(n)$	$O(1)$

Table 3.5: Four ways to compute Fibonacci numbers.

The idea of reducing memory rests on a general rule. If a table cell depends on only a bounded number of earlier cells, the cells that will no longer be used can be discarded. In the knapsack table every row depends only on the previous row, so the two-dimensional table can be cut down to a single row. This gain has a price: the discarded cells may be needed to reconstruct the solution. In Fibonacci there is no choice to reconstruct, so nothing is lost.

3.3 0/1 knapsack

3.3.1 The problem

A traveler is going to pack items into a bag that can carry only a limited weight. Each item has a weight and a value. The goal is to maximize the total value in the bag without exceeding the weight limit. Items cannot be divided: an item either goes into the bag whole or not at all. The 0/1 in the name of the problem describes these two options.

Definition 3.4

The 0/1 knapsack problem: given n items and a capacity W, item i has value v_i and weight w_i. Choose $x_i \in \{0, 1\}$ for every item so that $\Sigma\, v_i\, x_i$ is maximized subject to $\Sigma\, w_i\, x_i \le W$.



Figure 3.6: A bag with a capacity of 15 kg and five items: 4 ₺ and 12 kg, 2 ₺ and 2 kg, 2 ₺ and 1 kg, 1 ₺ and 1 kg, 10 ₺ and 4 kg.

Example 3.2

Which items should go into the bag of Figure 3.6?

The four items other than the 12 kg one weigh $2 + 1 + 1 + 4 = 8$ kg in total; all of them fit and are worth $2 + 2 + 1 + 10 = 15$ ₺. If the 12 kg item is chosen, 3 kg remain. The best way to use those 3 kg is the 2 kg and 1 kg items worth 2 ₺ each, for a total of $4 + 2 + 2 = 8$ ₺. No choice that takes the 12 kg item can reach 15 ₺. The answer is 15 ₺. Notably, the most expensive item is not part of the optimal solution.

3.3.2 Why brute force and greedy choice are not enough

The most direct solution of the problem is to try every selection. Each item has two options; n items give 2^n subsets. For $n = 50$ this number is $2^{50} \approx 1.13 \cdot 10^{15}$. Even at a billion subsets per second, it would take more than thirteen days. Brute force is usable only on very small inputs.

The second approach that comes to mind is the greedy choice. The items are sorted by value-to-weight ratio, and starting with the highest ratio, every item that fits is taken. This method sometimes gives the right answer, but not always.

Example 3.3

W = 50 kg; items (60 ₺, 10 kg), (100 ₺, 20 kg), and (120 ₺, 30 kg). What does the greedy choice give, and what is the optimal solution?

The ratios are 6, 5, and 4 ₺/kg. The greedy algorithm first takes the first item (10 kg full), then the second (30 kg full). The third item of 30 kg does not fit into the remaining 20 kg. The result is 160 ₺. Yet the second and third items together weigh exactly 50 kg and bring 220 ₺. By taking the most efficient item the greedy choice makes a correct local decision, but this decision leaves a gap of 20 kg in the bag that cannot be used.

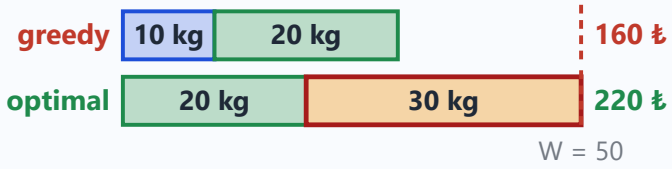


Figure 3.7: The greedy choice stops at 160 ₺; the optimal choice is 220 ₺.

The greedy choice is optimal if the items can be divided. That problem is called the **fractional knapsack**. In the example, if two thirds of the third item could be taken, the greedy algorithm would reach the optimal result with *60 + 100 + 80 = 240* ₺. In the 0/1 case, taking an item changes the later options. The only way to know which choice is better is to try both. That trial means a recursion that branches in two at every item, and the branches meet at the same subproblems. Overlapping subproblems and optimal substructure together open the door to dynamic programming.

3.3.3 State and recurrence

The first design step is to look at the last decision. Number the items from 1 to n. In an optimal solution there are only two options for the last item: it is either taken or not taken. If it is not taken, the remaining problem is to pack the first $n - 1$ items optimally with the same capacity. If it is taken, its value v_n has been gained, and the remaining problem is to pack the first $n - 1$ items optimally with capacity $W - w_n$. In both cases the remaining problem is of the same kind. Only two numbers change: the number of items and the capacity. The state is defined by these two numbers.

Definition 3.5

K[i][c] is the largest value obtainable with the first *i* items and capacity *c* ($0 \le i \le n$, $0 \le c \le W$). Base cases and recurrence:

> $K[0][c] = 0, \quad K[i][0] = 0$
> *if* $w_i > c$: $K[i][c] = K[i-1][c]$
> *otherwise:* $K[i][c] = \max(K[i-1][c], \; v_i + K[i-1][c-w_i])$

The answer is *K[n][W]*.

The two branches of the recurrence are the values of the two decisions. In the *skip* branch the capacity stays the same and the best value obtained with the earlier items is used: $K[i-1][c]$. In the *take* branch v_i is gained, and the remaining capacity $c - w_i$ is filled optimally with the earlier items: $v_i + K[i-1][c-w_i]$. If the item does not fit, there is no take branch. The base cases also agree with intuition: if there are no items, or the capacity is zero, the value is 0.

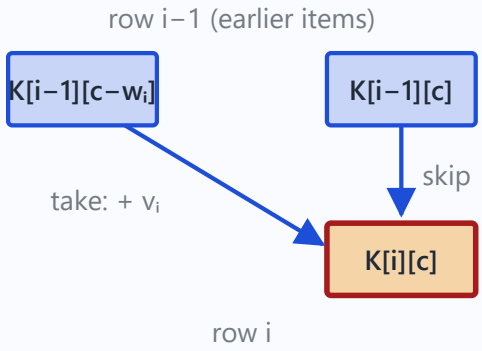


Figure 3.8: Cell *K[i][c]* is computed from two cells in the row above: $K[i-1][c]$ directly above it (skip) and $K[i-1][c-w_i]$, w_i columns to the left (take).

Figure 3.8 also fixes the order of computation. A cell looks only at the row above. So if the table is filled row by row from top to bottom, the cells a cell needs are ready when it is reached. Within a row the order of the columns does not matter.

3.3.4 Filling the table

The recurrence translates directly into two nested loops. The code keeps the items in arrays that start at 0; row *i* of the table uses item *i*, that is, the elements `v[i - 1]` and `w[i - 1]` of the arrays.

```
int knapsack(int[] v, int[] w, int W) {
  int n = v.length;
  int[][] K = new int[n + 1][W + 1];
  for (int i = 1; i <= n; i++)
    for (int c = 1; c <= W; c++) {
      int vi = v[i - 1], wi = w[i - 1];
      if (wi > c)            // does not fit
        K[i][c] = K[i - 1][c];
      else                   // skip / take
        K[i][c] = Math.max(K[i - 1][c],
                    vi + K[i - 1][c - wi]);
    }
  return K[n][W];
}
```

`new int[n + 1][W + 1]` initializes all cells to 0, so row 0 and column 0 supply the base cases by themselves. This is why the loops start at 1. The inner `if` tests whether the item fits. If it does not, the value above is copied; if it does, the larger of the two branches is written.

As an example take four items and $W = 6$. The items, as (value, weight), are (5, 4), (4, 3), (3, 2), and (2, 1). The table has size $(n + 1) \times (W + 1) = 5 \times 7$. Table 3.6 gives the filled table.

i	v_i	w_i	c = 0	1	2	3	4	5	6
0	—	—	0	0	0	0	0	0	0
1	5	4	0	0	0	0	5	5	5
2	4	3	0	0	0	4	5	5	5
3	3	2	0	0	3	4	5	7	8
4	2	1	0	2	3	5	6	7	**9**

Table 3.6: The table *K[i][c]*; items (5, 4), (4, 3), (3, 2), (2, 1), $W = 6$.

Let us follow the rows one by one.

- **Row 1** (5, 4). For $c = 1, 2, 3$ the item does not fit; the 0 above is copied. At $c = 4$, $max(0, 5 + K[0][0]) = 5$. At $c = 5$ and 6 the same computation gives 5; a single item can be taken only once.
- **Row 2** (4, 3). At $c = 3$ the second item fits for the first time: $max(0, 4 + K[1][0]) = 4$. At $c = 4$ the skip branch gives 5 and the take branch gives $4 + K[1][1] = 4$; 5 stays. At $c = 5$ and 6 it also stays 5: the two items together weigh 7 kg and do not fit.
- **Row 3** (3, 2). At $c = 2$, 3 is written. At $c = 3$ skipping gives 4 and taking gives $3 + K[2][1] = 3$; 4 stays. At $c = 5$ the take branch gives $3 + K[2][3] = 3 + 4 = 7$, which beats the 5 of the skip branch. At $c = 6$, $3 + K[2][4] = 3 + 5 = 8$ is written.

- **Row 4** (2, 1). The lightest item fits in every column. *c = 1*: *max(0, 2) = 2*. *c = 3*: *max(4, 2 + K[3][2]) = max(4, 5) = 5*. *c = 4*: *max(5, 2 + 4) = 6*. *c = 5*: *max(7, 2 + 5) = 7*; the two branches are equal and skipping is kept. *c = 6*: *max(8, 2 + K[3][5]) = max(8, 9) = 9*.

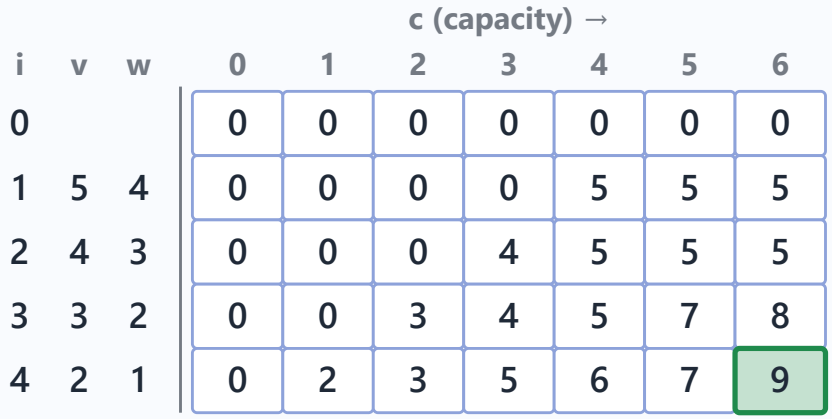


Figure 3.9: The table is filled. The answer is *K[4][6] = 9*; 24 cells were computed, each in *O(1)*.

The answer is in the bottom-right cell: *K[4][6] = 9*. Each row of the table answers a question: "with only the first *i* items available, what is the best value for each capacity?" Row 3, for example, says that with items (5, 4), (4, 3), and (3, 2), a 5 kg bag can carry a value of at most 7 Ł.

> **Common mistake**
>
> In row *i*, the item data is read with `v[i]` and `w[i]`. The table has *n + 1* rows but the arrays have *n* elements. If `v[i]` is written, every row uses the next item, and the last row causes an `ArrayIndexOutOfBoundsException`. The correct form is `v[i - 1]`.

3.3.5 Reconstructing the solution

The table gives only the largest value. Which items were chosen is hidden inside the table and is found by reading it from the end back to the start. This process is called **traceback**.

The rule is simple. If *K[i][c] ≠ K[i − 1][c]*, the value can have increased only by taking item *i*; the item was taken, and the capacity becomes $c \leftarrow c - w_i$. If they are equal, the same value is obtainable without the item; the item counts as not taken and *c* does not change. In both cases we move up one row.

Cell	Comparison	Decision	New c
K[4][6] = 9	≠ K[3][6] = 8	item 4 taken (v = 2, w = 1)	6 − 1 = 5
K[3][5] = 7	≠ K[2][5] = 5	item 3 taken (v = 3, w = 2)	5 − 2 = 3
K[2][3] = 4	≠ K[1][3] = 0	item 2 taken (v = 4, w = 3)	3 − 3 = 0

Cell	Comparison	Decision	New c
K[1][0] = 0	= K[0][0] = 0	item 1 not taken	0

Table 3.7: Traceback from cell *K[4][6] = 9*.

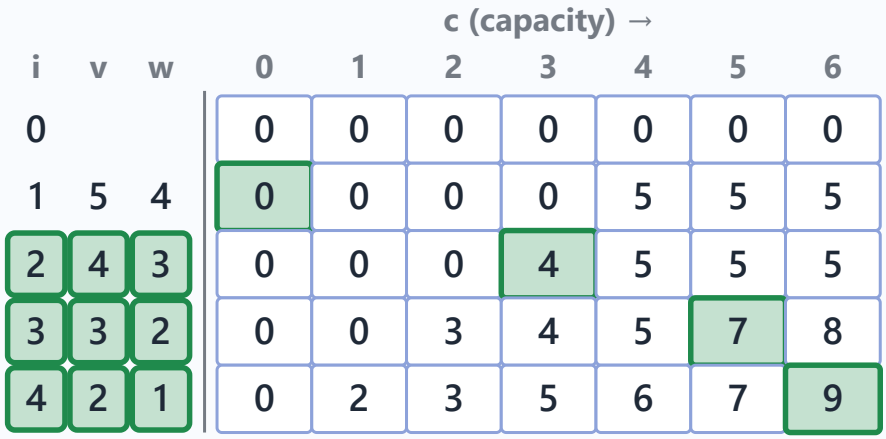


Figure 3.10: The path followed by the traceback. The chosen items are 2, 3, and 4: value *4 + 3 + 2 = 9*, weight *3 + 2 + 1 = 6 = W*.

The traceback makes one comparison per row; its running time is *O(n)*. The same rule translates easily into code. The following method takes the filled table and the weights and returns the numbers of the chosen items (starting from 1).

```
List<Integer> traceback(int[][] K, int[] w, int W) {
  List<Integer> chosen = new ArrayList<>();
  int c = W;
  for (int i = K.length - 1; i >= 1; i--)
    if (K[i][c] != K[i - 1][c]) {   // value grew: item i taken
      chosen.add(0, i);
      c -= w[i - 1];
    }
  return chosen;
}
```

For the example table the method returns `[2, 3, 4]`. Counting an item as not taken in case of equality means that, if there are several optimal solutions, one of them is picked. For example, in cell *K[4][5] = 7* the two branches were equal; a traceback starting from that cell does not take item 4 and finds items (4, 3) and (3, 2).

3.3.6 Complexity and pseudo-polynomial time

The table has *(n + 1)(W + 1)* cells, and each cell is computed in constant time. The time and memory are *O(n · W)*. At first sight this looks like polynomial time. But there is an important subtlety here.

The running time of an algorithm is measured against the *size* of the input, that is, the number of bits needed to write it down. Writing n items takes space proportional to n. But writing the number W takes only $\log_2 W$ bits. $W = 10^9$ is written with thirty bits, yet the table needs 10^9 columns. When the number of bits of W grows by one, the value of W and the width of the table double. The running time is exponential in the length of the input.

Definition 3.6

An algorithm whose running time is polynomial not in the length of the input but in the *value* of a number in the input is called a **pseudo-polynomial** time algorithm. The $O(n \cdot W)$ DP for 0/1 knapsack is of this kind.

This distinction has a practical consequence. 0/1 knapsack is an NP-hard problem; no known algorithm solves it in polynomial time on every input. DP does not change this fact. It only makes the problem practically solvable when W is small, for example a few thousand or a few million. If the weights are in grams and the capacity is a few tons, the table reaches billions of columns and cannot be built.

Common mistake

$O(n \cdot W)$ is taken to be polynomial, and it is said that "0/1 knapsack is solved in polynomial time by DP". W is a number; it occupies $\log W$ bits in the input. The running time is exponential in this number of bits.

3.3.7 A one-dimensional table

Row i uses only row $i-1$. This observation reduces the memory to $O(W)$: a one-dimensional `dp` array is kept. The items are processed in order; for each item the array is updated in place, from the old row to the new row.

```
int[] dp = new int[W + 1];
for (int i = 0; i < n; i++)
  for (int c = W; c >= w[i]; c--)
    dp[c] = Math.max(dp[c],
                     v[i] + dp[c - w[i]]);
return dp[W];
```

The most important detail of the code is the direction of the inner loop: the capacity is scanned **from large to small**. The reason is this. When `dp[c]` is updated, the `dp[c - w[i]]` that is read must be the value of the old row; that is, item i must not have been taken into account yet. When scanning from large to small, the cells with $c - w_i < c$ have not yet been updated in this round and still carry their old values. If the scan went from small to large, `dp[c - w[i]]` would already have been

updated in this round, and the same item could be taken twice. The lower bound of the loop, `c >= w[i]`, never visits capacities that cannot hold the item. In those cells the value does not change; it counts as copied from the old row.

Item (v, w)	Changed cells (from c = 6 downward)	dp[0..6]
initial	—	0 0 0 0 0 0 0
0: (5, 4)	dp[6]: 0 → 5, dp[5]: 0 → 5, dp[4]: 0 → 5	0 0 0 0 5 5 5
1: (4, 3)	dp[3]: 0 → 4	0 0 0 4 5 5 5
2: (3, 2)	dp[6]: 5 → 8, dp[5]: 5 → 7, dp[2]: 0 → 3	0 0 3 4 5 7 8
3: (2, 1)	dp[6]: 8 → 9, dp[4]: 5 → 6, dp[3]: 4 → 5, dp[1]: 0 → 2	0 2 3 5 6 7 9

Table 3.8: One-dimensional 0/1 knapsack; items (5, 4), (4, 3), (3, 2), (2, 1), $W = 6$. Each row is `dp[0..6]` after that item has been processed.

Each row is identical to the row with the same number in the two-dimensional table. For example, while item 0 is processed, `dp[6]`, `dp[5]`, and `dp[4]` become 5 in turn. The cells read in the meantime, `dp[2]`, `dp[1]`, and `dp[0]`, have not changed yet and are 0; the item is taken once. The result is `dp[6] = 9`. Since the code numbers items from 0, the selection reads "items 1, 2, 3": *4 + 3 + 2*. In the numbering of the two-dimensional table, these are items 2, 3, and 4.



Figure 3.11: The single-row table after four items have been processed: `dp[6] = 9`, the same as the bottom-right cell of the two-dimensional table.

Example 3.4

For the same four items, what happens if the inner loop is mistakenly written from small to large (`for (int c = w[i]; c <= W; c++)`)?

While item 3 (2, 1) is processed, `dp[1]` becomes 2; then `dp[2] = max(3, 2 + dp[1]) = 4`. This computation uses the `dp[1]` that was just updated, that is, it takes the same item a second time. The chain continues: `dp[3] = 6`, `dp[4] = 8`, `dp[5] = 10`, `dp[6] = 12`. The result is 12, as if six copies of the lightest item had been taken. This is not a 0/1 solution; it is the answer of the unbounded knapsack.

The one-dimensional version has a price. Since the intermediate rows are deleted, traceback is no longer possible. If the chosen items are needed, a separate `boolean[][] take` table is kept that stores the decision for each item and capacity. This table is also of size $n \cdot W$, but it takes less space because it stores `boolean` instead of `int`.

3.3.8 The top-down solution

The same recurrence can also be written with memoization. Here the subproblem is defined as "the largest value obtainable from item i onward with capacity c"; the items are scanned from first to last. This definition is the mirror image of the one in tabulation and gives the same answer.

```
Integer[][] memo;   // memo[i][c]: best from item i onward, capacity c

int f(int[] v, int[] w, int i, int c) {
  if (i == v.length) return 0;              // no items left
  if (memo[i][c] != null) return memo[i][c];
  int best = f(v, w, i + 1, c);             // skip
  if (w[i] <= c)                            // take
    best = Math.max(best, v[i] + f(v, w, i + 1, c - w[i]));
  memo[i][c] = best;
  return best;
}

int knapsackMemo(int[] v, int[] w, int W) {
  memo = new Integer[v.length][W + 1];
  return f(v, w, 0, W);
}
```

The cache is of type `Integer[][]`. `Integer` objects start as `null`, and `null` means "not yet computed". If `int[][]` were used, the value 0 would mean both "not computed" and "the real answer is 0", and states whose answer is 0 would be recomputed again and again. The method first checks the base case: if no items are left, the value is 0. Then it checks the cache. If the value is missing, it computes the skip branch, computes the take branch too if the item fits, and stores the larger one.

In the example `knapsackMemo(v, w, 6)` returns 9. The cache has $4 \times 7 = 28$ cells, but only 12 of them are filled. The recursion starting from the root descends only to capacities that can really be reached: at item 0 only $c = 6$; at item 1, $c = 6$ and 2; at item 2, 6, 3, and 2; at item 3, 6, 4, 3, 1, 2, and 0. When the weights are large and few, this difference grows even more. For example, if all weights are multiples of 1000, only capacities that are multiples of 1000 are reached.

3.4 Unbounded knapsack

3.4.1 Problem and recurrence

In some problems the items are given not one by one but as types, and any number of each type may be taken. A shipping company can load as many boxes of the same size as it wishes; a cutting machine can cut as many pieces of the same length as it wishes. This problem is called the **unbounded knapsack**. Everything else is the same as in 0/1 knapsack: the total profit is maximized without exceeding the capacity.

The state is again defined by two numbers, but this time the item types are numbered from 0: *dp[i][c]* is the largest profit obtainable with item types 0 through *i* and capacity *c*. The structure of the last decision is similar. If no copy of type *i* is taken, the answer is *dp[i − 1][c]*. If at least one is taken, $profit_i$ is gained and a capacity of $c - weight_i$ remains. In this remaining capacity *the same type can be taken again*. So the remaining problem lies not in row $i - 1$ but again in row *i*.

> $dp[i][c] = max(\, dp[i-1][c],\ \ profit_i + dp[i][c - weight_i] \,) \quad (if\ weight_i \le c)$

If the item does not fit, only the first branch remains. Since row indices start at 0, we take *dp[−1][c] = 0*: with no item types the profit is 0. The only difference from 0/1 knapsack is the row that the "take" branch looks at.

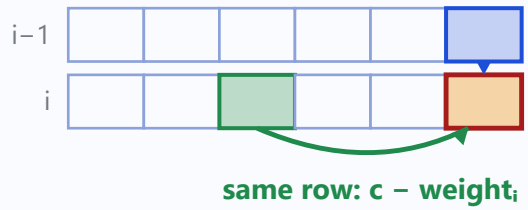


Figure 3.12: In unbounded knapsack the take branch looks at the cell $c - weight_i$ in the same row: after an item is taken it can be taken again.

This difference also affects the order of computation. A cell now depends on a cell to its left in the same row. If the row is filled from left to right, that cell is ready. Rows are still scanned from top to bottom.

3.4.2 Filling the table

As an example take four item types: profits *[15, 50, 60, 90]*, weights *[1, 3, 4, 5]*, capacity 8. The table has size 4×9. In the first row only type 0 (profit 15, 1 kg) can be used.

i	profit	weight	c = 0	1	2	3	4	5	6	7	8
0	15	1	0	15	30	45	60	75	90	105	120
1	50	3	0	15	30	50	65	80	100	115	130
2	60	4	0	15	30	50	65	80	100	115	130
3	90	5	0	15	30	50	65	90	105	120	**140**

Table 3.9: The unbounded knapsack table; profits *[15, 50, 60, 90]*, weights *[1, 3, 4, 5]*, capacity 8.

The rows are computed as follows.

- **Row 0.** The capacity 0 column is 0. At *c = 1* the only option is to take type 0: *15 + dp[0][0] = 15*. At *c = 2*, *15 + dp[0][1] = 30*: the same row was used again, that is, two items of type 0 were taken. The row thus fills with multiples of 15: *dp[0][c] = 15c*.
- **Row 1** (50, 3 kg). At *c = 1, 2* the item does not fit and the row above is copied. *c = 3*: *max(45, 50 + dp[1][0]) = 50*. *c = 4*: *max(60, 50 + dp[1][1]) = 65*. *c = 6*: *max(90, 50 + dp[1][3]) = max(90, 100) = 100*; here *dp[1][3]* is itself a cell that contains type 1. 100 is two items of 50. *c = 8*: *max(120, 50 + dp[1][5]) = 130*.
- **Row 2** (60, 4 kg). The item of 60 cannot beat the row above in any column. At *c = 8*, for example, *60 + dp[2][4] = 60 + 65 = 125 < 130*. The row is a copy of the row above.
- **Row 3** (90, 5 kg). *c = 5*: *max(80, 90 + 0) = 90*. *c = 6*: *max(100, 90 + dp[3][1]) = 105*. *c = 7*: *max(115, 90 + 30) = 120*. *c = 8*: *max(130, 90 + dp[3][3]) = max(130, 140) = 140*.

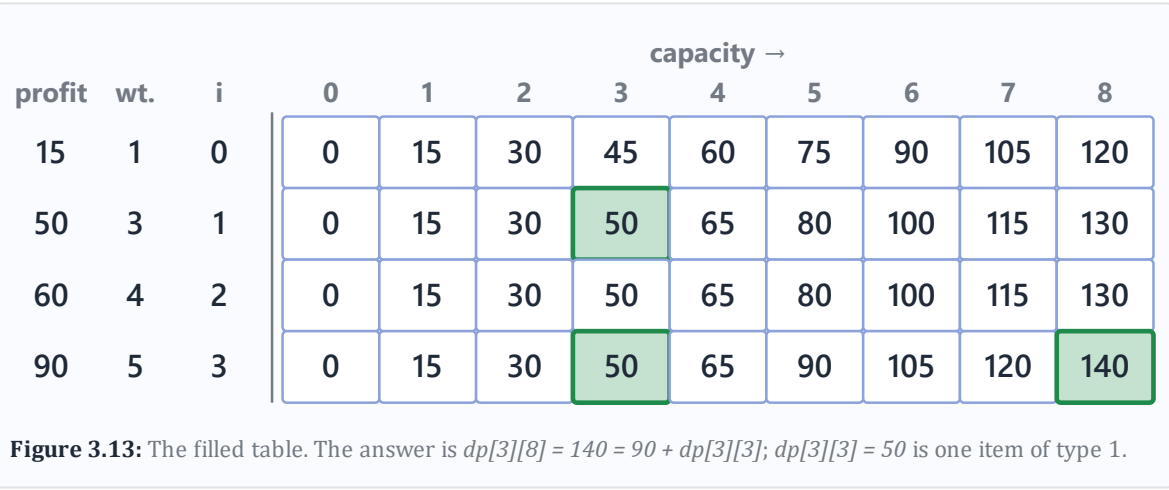


Figure 3.13: The filled table. The answer is *dp[3][8] = 140 = 90 + dp[3][3]*; *dp[3][3] = 50* is one item of type 1.

The answer is *dp[3][8] = 140*. Traceback is done as in 0/1, with one difference: when a type is taken, we stay in the same row. Since *dp[3][8] = 140 ≠ dp[2][8] = 130*, one item of type 3 has been taken and *c = 3*. Since *dp[3][3] = dp[2][3] = 50*, we move up. *dp[2][3] = dp[1][3]*, up again. *dp[1][3] = 50 ≠ dp[0][3] = 45*: one item of type 1 has been taken and *c = 0*. The solution is one item of 90 (5 kg) and one item of 50 (3 kg).

3.4.3 One-dimensional solution and loop direction

In unbounded knapsack the rows can be merged completely. Since any number of each type may be taken, the best value for capacity c can be written by looking at the last item taken. If the last item is j, the remaining capacity is $c - w_j$, and there too all types are free:

> $dp[c] = max\ \{\ dp[c - w_j] + v_j : w_j \le c\ \},\quad dp[0] = 0$

```
static int unboundedKnapsack(int[] w, int[] v, int W) {
  int[] dp = new int[W + 1];       // dp[c]: best for capacity c
  for (int c = 1; c <= W; c++)
    for (int j = 0; j < w.length; j++)
      if (w[j] <= c)
        dp[c] = Math.max(dp[c], dp[c - w[j]] + v[j]);
  return dp[W];
}
```

The outer loop raises the capacity from 1 to W, and the inner loop tries every type. When $dp[c]$ is computed, all smaller capacities are ready. For the example the resulting array is $dp = [0, 15, 30, 50, 65, 90, 105, 120, 140]$, the same as the last row of the two-dimensional table. The time is $O(n \cdot W)$ and the memory is $O(W)$.

The relation between the one-dimensional 0/1 knapsack and this code is instructive. Both perform the following update on a single array: `dp[c] = max(dp[c], v + dp[c - w])`. The difference is in the loop direction. If the capacity is scanned **from small to large**, `dp[c - w]` has already been updated in this round and may contain the item; the item can be taken again, and the result is unbounded knapsack. If the capacity is scanned **from large to small**, `dp[c - w]` carries its old value; the item is taken at most once, and the result is 0/1. The same one-line update solves two different problems, merely by the direction of the loop.

Example 3.5

There is a single item: value 10, weight 3; capacity 9. What is `dp[9]` if the one-dimensional table is filled in the two directions?

From large to small: at $c = 9$, `dp[6]` is still 0, so `dp[9] = 10`; 10 is also written at $c = 6$ and $c = 3$. The result is 10; the item is taken once. From small to large: `dp[3] = 10`, `dp[6] = 10 + dp[3] = 20`, `dp[9] = 10 + dp[6] = 30`. The item has been taken three times. 10 is the answer of 0/1; 30 is the answer of unbounded knapsack.

Common mistake

In one-dimensional 0/1 knapsack the capacity is scanned from small to large. The code compiles, runs, and returns a plausible-looking number, but that number is the answer of unbounded knapsack. The easiest way to catch the bug is to test a small example with a single item and check whether the result exceeds the item's value.

3.4.4 Code trace

A trace of the one-dimensional code shows which option wins at each capacity. The version below is the tracing code from the slides and does the same job as the one above.

```
static int unboundedKnapsack(
    int[] w, int[] v, int W) {
  int[] dp = new int[W + 1];  // dp[c]
  for (int c = 1; c <= W; c++)
    for (int j = 0; j < w.length; j++)
      if (w[j] <= c)
        dp[c] = Math.max(dp[c],
                  dp[c - w[j]] + v[j]);
  return dp[W];
}
```

c	Candidates $dp[c - w_j] + v_j$	dp[c]
1	dp[0] + 15 = 15	15
2	dp[1] + 15 = 30	30
3	dp[2] + 15 = 45, dp[0] + 50 = 50	50
4	dp[3] + 15 = 65, dp[1] + 50 = 65, dp[0] + 60 = 60	65
5	dp[4] + 15 = 80, dp[2] + 50 = 80, dp[1] + 60 = 75, dp[0] + 90 = 90	90
6	dp[5] + 15 = 105, dp[3] + 50 = 100, dp[2] + 60 = 90, dp[1] + 90 = 105	105
7	dp[6] + 15 = 120, dp[4] + 50 = 115, dp[3] + 60 = 110, dp[2] + 90 = 120	120
8	dp[7] + 15 = 135, dp[5] + 50 = 140, dp[4] + 60 = 125, dp[3] + 90 = 140	140

Table 3.10: Trace of `unboundedKnapsack(w, v, 8)`; profits *[15, 50, 60, 90]*, weights *[1, 3, 4, 5]*. In each row, the candidates from the types that fit and the winning value.

The row *c = 2* shows the source of unboundedness: `dp[1]` had been updated in this round and contained type 0 once; adding 15 to it again means two items of the same type have been taken. In the last row two candidates reach 140: *dp[5] + 50* and *dp[3] + 90*. Both describe the same selection: one item of 50 (3 kg) and one item of 90 (5 kg). The selection 15 + 15 + 50 + 60, which also gives 140, weighs 9 kg and does not fit into the bag. The table has tried all combinations in $O(n \cdot W)$ time without counting them one by one.

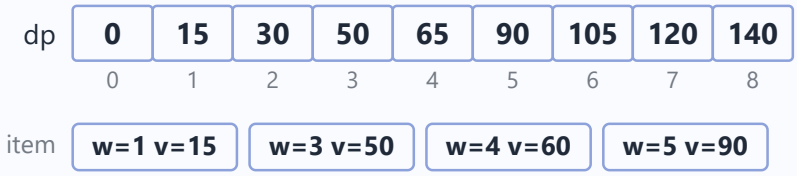


Figure 3.14: The `dp` array at the end of the trace; `dp[8] = 140`.

3.5 Subset sum

3.5.1 Problem and recurrence

Suppose we are given an array of non-negative integers and a target value ($target \geq 0$). Is there a subset whose sum is exactly equal to the target? This question is not an optimization but a **decision problem**: the answer is yes or no. The problem is called **subset sum**.

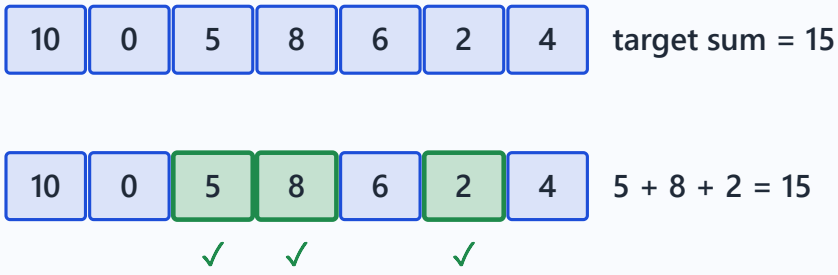


Figure 3.15: In the array 10, 0, 5, 8, 6, 2, 4 with target 15: for example, choosing 5, 8, and 2 gives a sum of 15.

The last decision is again whether an element is chosen. If the last element is not chosen, the question is whether the same target can be reached with the first $n - 1$ elements. If it is chosen, the target decreases by the element. The state is defined by two numbers, and its value is a Boolean.

Definition 3.7

dp[i][j] is true if a subset with sum exactly *j* can be formed from the first *i* elements, and false otherwise.

> $dp[i][0] = true, \quad dp[0][j] = false \ \ (j > 0)$
> $dp[i][j] = dp[i-1][j] \ \text{ or } \ (a_i \leq j \ \text{ and } \ dp[i-1][j-a_i])$

Column 0 is true in every row, because the empty set gives the sum 0. In row 0 no sum greater than zero can be obtained. The recurrence has the same form as that of 0/1 knapsack: a logical or replaces the `max` of knapsack, and instead of adding up values there is only reachability. Subset sum can also be seen as the special case of knapsack in which "value = weight". Let the value of each element equal its weight and let the capacity be the target. If the answer of the knapsack equals the target, the desired subset exists.

3.5.2 Table and reconstruction

Let the array be $a = 10, 0, 5, 8, 6, 2, 4$ and the target 15. The table has size 8×16. Instead of writing every row with all its cells, it is more readable to list the sums reachable in that row. A row keeps all the sums of the previous row and also adds new sums by adding a_i to each of them (those exceeding the target are left out).

i	a_i	Reachable sums (dp[i][j] true)	New
0	—	0	—
1	10	0, 10	10
2	0	0, 10	none
3	5	0, 5, 10, 15	5, 15
4	8	0, 5, 8, 10, 13, 15	8, 13
5	6	0, 5, 6, 8, 10, 11, 13, 14, 15	6, 11, 14
6	2	0, 2, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15	2, 7, 12
7	4	0, 2, 4, 5, 6, …, 15 (all except 1 and 3)	4, 9

Table 3.11: The subset sum table row by row; $a = 10, 0, 5, 8, 6, 2, 4$, target 15. The columns that are true in the row are listed.

Row 2 is interesting: adding 0 changes no sum, and the row is a copy of the row above. The target 15 becomes reachable for the first time in row 3: 10 and 5 together. The last cell $dp[7][15]$ is true; the answer is yes.

The traceback starts from the last cell. If $dp[i-1][j]$ is true, the sum j can be formed without a_i; the element is not chosen and we move up. If it is false, the sum can be formed only with a_i; the element is chosen and $j \leftarrow j - a_i$.

- $dp[7][15] = dp[6][15] =$ true: 4 is not chosen. By the same reasoning 2, 6, and 8 are not chosen either; we move up to row 4.
- $dp[3][15]$ is true but $dp[2][15]$ is false: 5 is chosen; $j = 15 - 5 = 10$.
- $dp[2][10] = dp[1][10] =$ true: 0 is not chosen.
- $dp[1][10]$ is true but $dp[0][10]$ is false: 10 is chosen; $j = 0$ and the reading ends.

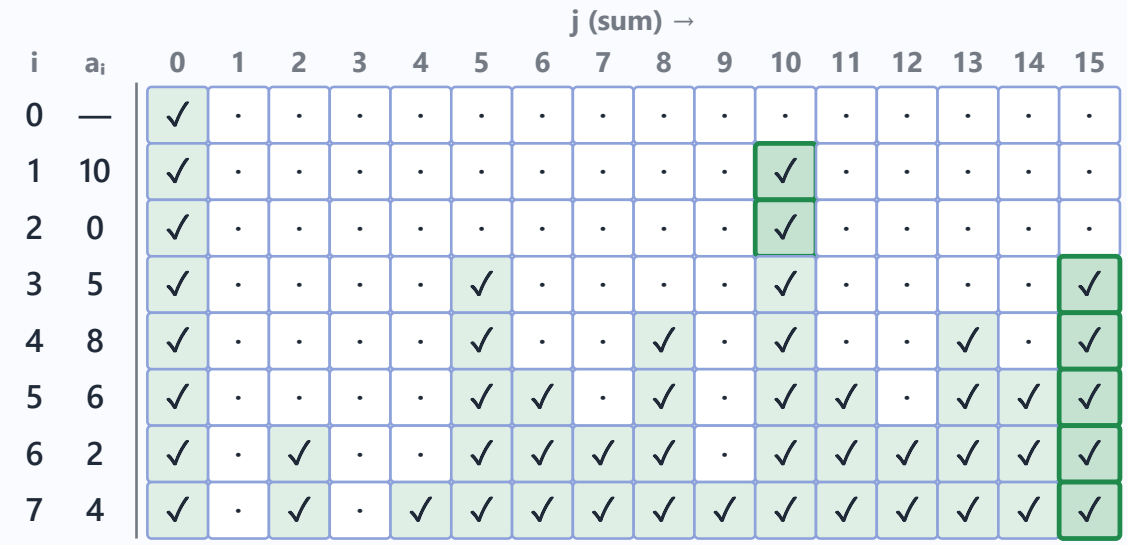


Figure 3.16: The filled table and the traceback path. The traceback finds the subset {5, 10}: *10 + 5 = 15*.

In this array there are six subsets whose sum is 15: {10, 5}, {5, 8, 2}, {5, 6, 4}, and the versions of each with 0 added. The traceback gives only one. Which one it gives depends on the tie-breaking rule. Since the rule here is "do not choose if it is possible without the element", the reading leans toward the solution that uses elements from the start of the array.

3.5.3 Code and analysis

```
boolean subsetSum(int[] a, int target) {
  int n = a.length; // a[i] ≥ 0, target ≥ 0
  boolean[][] dp = new boolean[n + 1][target + 1];
  for (int i = 0; i <= n; i++)
    dp[i][0] = true;           // empty set
  for (int i = 1; i <= n; i++) {
    int x = a[i - 1];
    for (int j = 1; j <= target; j++)
      dp[i][j] = dp[i - 1][j]  // x not taken
        || (x <= j && dp[i - 1][j - x]);
  }
  return dp[n][target];
}
```

In Java a new `boolean` array starts as `false`. The first loop makes column 0 true. The expression in the inner loop is a literal translation of the recurrence. The short-circuit behavior of the `||` operator is useful here: if `dp[i - 1][j]` is true, the right-hand side is never evaluated. The test `x <= j` on the right, thanks to the short circuit of `&&`, prevents accessing the array when `j - x` is negative.

This guarantee rests on the precondition that the elements and the target are non-negative: since `x ≥ 0`, `j - x` never exceeds `j`, and since `target ≥ 0`, the table can be built. For negative elements a separate table with a shifted range of sums is needed.

The time and memory are *O(n · target)*. This too is a pseudo-polynomial time: if the target is large, the table cannot be built. Subset sum is an NP-complete problem; DP makes it practical only when the target is small. As in 0/1 knapsack, the memory can be reduced to one dimension. For this, *j* is scanned from large to small so that each element is used at most once.

Example 3.6

The partition problem. Can the array *{1, 5, 11, 5}* be split into two parts with equal sums?

The total is *S = 22*. If one part has sum *S/2 = 11*, the other part has sum 11 as well. The question turns into subset sum with target 11. {11} or {1, 5, 5} gives this sum; the answer is yes. If *S* were odd, the answer would be no without any computation.

Common mistake

Column 0 is not initialized to true. The table stays `false` throughout and no sum looks reachable; the answer is "no" on every input. The sum of the empty set is 0, and this holds in every row.

3.5.4 Code trace

Let us trace the line-by-line behavior of the code on a smaller example: `subsetSum({3, 34, 4, 12, 5, 2}, 9)`. The table has *(n + 1) × (target + 1) = 7 × 10 = 70* cells.

i	x	j with dp[i][j] true	Explanation
0	—	0	only the empty set
1	3	0, 3	new sum: 3
2	34	0, 3	*34 > 9*: does not fit for any j, the row is copied
3	4	0, 3, 4, 7	new sums: 4, 7
4	12	0, 3, 4, 7	*12 > 9*: the row is copied
5	5	0, 3, 4, 5, 7, 8, 9	new: 5, 8, 9; *dp[5][9] = dp[4][4] → 4 + 5 = 9*
6	2	0, 2, 3, 4, 5, 6, 7, 8, 9	new: 2, 6

Table 3.12: Trace of `subsetSum({3, 34, 4, 12, 5, 2}, 9)`; the columns that are true in each row.

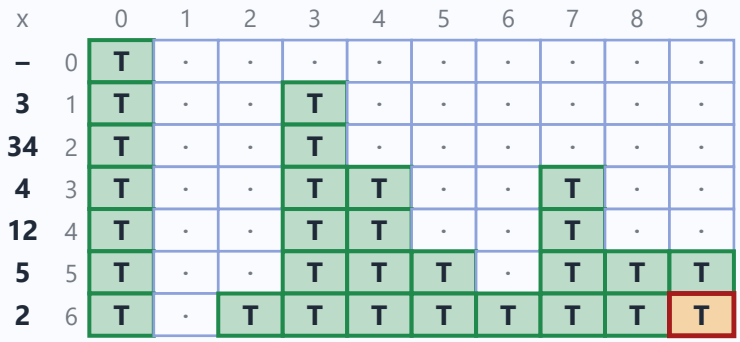


Figure 3.17: The end of the trace: `dp[6][9] = true`. T marks true cells and dots mark false cells.

The method returns `true`. There are two subsets with sum 9: {4, 5} and {3, 4, 2}. The traceback starts at *dp[6][9]*; since *dp[5][9]* is true, 2 is not chosen. Since *dp[4][9]* is false, 5 is chosen and $j = 4$. 12 is not chosen; since *dp[2][4]* is false, 4 is chosen and $j = 0$. The subset found is {4, 5}. If the target 30 were asked for the same array, the answer would be no: 34 can never be used, and the sum of the other elements is $3 + 4 + 12 + 5 + 2 = 26 < 30$.

3.6 Longest increasing subsequence

3.6.1 Subsequence and substring

The three problems in this and the next two sections (LIS, common substring, LCS) concern pieces of a sequence. The word "piece" is used in two different senses, and confusing the two is the leading source of mistakes on this topic.

Definition 3.8

A **subsequence** of a sequence is a sequence obtained by skipping some elements and keeping the order of the remaining ones. The chosen elements need not be adjacent.

A **substring** (or contiguous subarray) of a sequence is a piece made of consecutive elements.

In the string "GXTXAYB" the elements G, T, A, B occur at positions 0, 2, 4, and 6. Since their relative order is kept, "GTAB" is a subsequence. Since the elements are not adjacent, it is not a substring. In contrast, "XAY" occurs contiguously at positions 3, 4, and 5; it is both a substring and a subsequence. Every substring is also a subsequence; the converse is not true. A sequence of length n has 2^n subsequences, but only $n(n + 1)/2$ nonempty contiguous pieces.

Definition 3.9

The **longest increasing subsequence** (LIS) is the longest among the subsequences whose elements are strictly increasing.

As an example take the following arrangement of the numbers 0 through 15: 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15. One LIS of this sequence is 0, 2, 6, 9, 11, 15, and its length is 6.



Figure 3.18: The elements 0, 2, 6, 9, 11, 15 are picked while skipping the others; their relative order is kept and their values increase.

The same sequence has other LISs as well. For example, 0, 4, 6, 9, 13, 15 and 0, 2, 6, 9, 13, 15 are also increasing subsequences of length 6; this sequence has four different LISs in total. The LIS need not be unique, but its length is. DP first finds this length; if the sequence itself is needed, it is read back with extra information.

3.6.2 The O(n^2) dynamic program

Choosing the right state is the key to this problem. "The LIS length of the first *i* elements" is not a good state: knowing this value does not tell whether element *i + 1* can extend that sequence, because we do not know which number the sequence ends with. The state must fix the endpoint.

Definition 3.10

dp[i] is the length of the longest increasing subsequence that ends *exactly* at *a[i]*.

> $dp[i] = 1 + \max \{ dp[j] : j < i,\ \ a[j] < a[i] \}$

If there is no such *j*, *dp[i] = 1*. The answer is *max dp[i]*.

The meaning of the recurrence is this. In an increasing subsequence ending at *a[i]*, the element before *a[i]* is some *a[j]* that lies to the left and is smaller than *a[i]*. *a[i]* is appended to the longest sequence ending at that element. All candidates are tried and the longest is chosen. If there is no smaller element before it, *a[i]* alone is a sequence of length 1. The answer is not the last cell: the LIS can end anywhere in the sequence.

Table 3.13 gives all *dp* values of the 16-element example. The "prev" row is the *j* that gives the largest *dp[j]*; in case of a tie the first *j* from the left was chosen.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[i]	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
dp[i]	1	2	2	3	2	3	3	4	2	4	3	5	3	5	4	6
prev	—	0	0	1	0	1	2	3	0	6	2	9	4	9	6	11

Table 3.13: *dp* and predecessor indices for the 16-element sequence.

Let us write out the computation of a few cells in detail. At *i = 3*, *a[3] = 12*; the candidates on the left are 0, 8, and 4, with dp values 1, 2, 2. The largest is the 2 at *j = 1*: *dp[3] = 3* (0, 8, 12). At *i = 9*, *a[9] = 9*; the candidates are 0, 8, 4, 2, 6, and 1. Of these, the one ending the longest sequence is 6 (*dp[6] = 3*): *dp[9] = 4* (0, 4, 6, 9). At *i = 15*, *a[15] = 15* is larger than all elements; the largest dp value among the candidates is 5, first seen at *j = 11*: *dp[15] = 6*.



Figure 3.19: The largest dp is 6 (*i = 15*). The predecessor indices are followed: 15 → 11 → 9 → 6 → 2 → 0; the LIS is 0, 4, 6, 9, 13, 15.

The sequence itself is read back from the predecessor indices. We start at the index holding the largest dp value and jump along predecessor indices: 15 → 11 → 9 → 6 → 2 → 0. The elements at these indices, from last to first, are 15, 13, 9, 6, 4, 0; reversed, this gives 0, 4, 6, 9, 13, 15. This differs from the LIS of Figure 3.18 but has the same length. The cause of the difference is the tie-breaking rule: at *i = 15* both *j = 11* and *j = 13* are candidates with dp value 5, and the first one seen is chosen.

3.6.3 Code and trace

```
static int lis(int[] a) {
  int n = a.length, best = 0;
  int[] dp = new int[n];
  for (int i = 0; i < n; i++) {
    dp[i] = 1;               // alone
    for (int j = 0; j < i; j++)
      if (a[j] < a[i] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;   // extend
    best = Math.max(best, dp[i]);
  }
  return best;
}
```

For each *i*, the outer loop first writes `dp[i] = 1`: the element alone is a sequence. The inner loop tries every *j* to the left. The condition `a[j] < a[i]` ensures that the sequence is increasing, and `dp[j] + 1 > dp[i]` ensures that only a longer sequence is accepted. The strict greater-than keeps the first *j* on a tie. At the end of each round `best` is updated. The inner loop runs $0 + 1 + \dots + (n-1) = n(n-1)/2$ times in total; the time is $O(n^2)$ and the memory is $O(n)$.

Tracing the code on a shorter sequence shows the effect of the lines clearly: `lis({5, 2, 8, 6, 3, 6, 9, 7})`.

i	a[i]	Smaller on the left	Chosen j (dp[j])	dp[i]	best
0	5	none	—	1	1
1	2	none	—	1	1
2	8	5, 2	0 (1)	2	2
3	6	5, 2	0 (1)	2	2
4	3	2	1 (1)	2	2
5	6	5, 2, 3	4 (2)	3	3
6	9	5, 2, 8, 6, 3, 6	5 (3)	4	4
7	7	5, 2, 6, 3, 6	5 (3)	4	4

Table 3.14: Trace of `lis({5, 2, 8, 6, 3, 6, 9, 7})`.

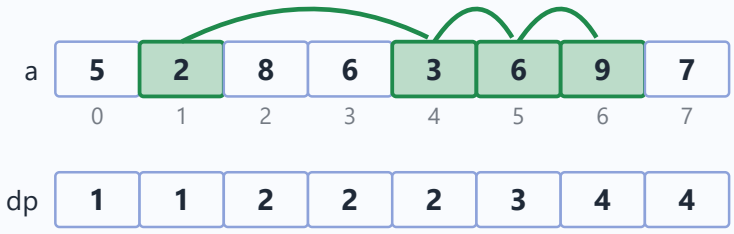


Figure 3.20: The end of the trace: `best = 4`. One LIS is 2, 3, 6, 9 (the chain with dp values 1, 2, 3, 4).

The row *i = 5* shows why the state is defined as "ending at *a[i]*". There are three elements smaller than *a[5] = 6*: 5, 2, and 3. 5 and 2 are each the end of a sequence of their own (dp = 1). 3 is the end of the sequence 2, 3 (dp = 2). 6 extends the longest chain: 2, 3, 6. The dp value of the last element 7 is 4 and equals `best`; this is a coincidence. If a 1 were appended to the sequence, the last cell would be 1 and the answer would still be 4.

To build the sequence, the predecessor index of each element is stored. The following method adds a `prev` array to `lis` and follows the chain backwards from where the longest sequence ends.

```
List<Integer> lisSeq(int[] a) {
  int n = a.length, last = 0;
  int[] dp = new int[n], prev = new int[n];
  for (int i = 0; i < n; i++) {
    dp[i] = 1; prev[i] = -1;
    for (int j = 0; j < i; j++)
      if (a[j] < a[i] && dp[j] + 1 > dp[i]) {
        dp[i] = dp[j] + 1; prev[i] = j;
      }
    if (dp[i] > dp[last]) last = i;
  }
  LinkedList<Integer> lis = new LinkedList<>();
  for (int i = last; i != -1; i = prev[i])
    lis.addFirst(a[i]);
  return lis;
}
```

For the example sequence the method returns `[2, 3, 6, 9]`, and for the 16-element sequence `[0, 4, 6, 9, 13, 15]`. `LinkedList.addFirst` offsets the fact that the traceback proceeds from the end to the start: since elements are added at the front, the list comes out in the correct order.

Common mistake

The method returns `dp[n - 1]` instead of `best`. In the sequence {5, 1, 6, 2, 7, 0}, *dp = [1, 1, 2, 2, 3, 1]*. The last element 0 cannot extend any sequence; `dp[5] = 1` is returned. The correct

answer is 3 (5, 6, 7 or 1, 2, 7).

3.6.4 Active lists

The $O(n^2)$ algorithm looks at all elements to the left at every element. For a faster method we use the following observation: if there are two increasing sequences of the same length, the one with the smaller last element is always better. If a number that comes later can extend the one with the larger last element, it can also extend the one with the smaller last element; the converse is not true. So it suffices to keep a single list for each length: among the increasing lists of that length, the one **with the smallest last element**. These lists are called **active lists**.

When a new element x arrives there are three cases:

1. If x is smaller than the last elements of all active lists, a new list $[x]$ of length 1 starts and replaces the old list of length 1.
2. If x is larger than all last elements, the longest list is copied and x is appended; the number of lists grows by one.
3. Otherwise, the longest list whose last element is smaller than x is found, copied, and x is appended. The new list replaces the old list of the same length; since the old list's last element is larger, it is no longer needed.

x	Event	Number of lists	Longest active list
0	first list [0]	1	0
8	larger than all: extend	2	0, 8
4	extends [0] → [0, 4]; [0, 8] is redundant	2	0, 4
12	larger than all: extend	3	0, 4, 12
2	extends [0] → [0, 2]; [0, 4] is redundant	3	0, 4, 12
10	extends [0, 2] → [0, 2, 10]; [0, 4, 12] is redundant	3	0, 2, 10
6	extends [0, 2] → [0, 2, 6]; [0, 2, 10] is redundant	3	0, 2, 6
14	larger than all: extend	4	0, 2, 6, 14
1	extends [0] → [0, 1]; [0, 2] is redundant	4	0, 2, 6, 14
9	extends [0, 2, 6] → [0, 2, 6, 9]; [0, 2, 6, 14] is redundant	4	0, 2, 6, 9
5	extends [0, 1] → [0, 1, 5]; [0, 2, 6] is redundant	4	0, 2, 6, 9
13	larger than all: extend	5	0, 2, 6, 9, 13
3	extends [0, 1] → [0, 1, 3]; [0, 1, 5] is redundant	5	0, 2, 6, 9, 13
11	extends [0, 2, 6, 9] → [0, 2, 6, 9, 11]; [0, 2, 6, 9, 13] is redundant	5	0, 2, 6, 9, 11
7	extends [0, 1, 3] → [0, 1, 3, 7]; [0, 2, 6, 9] is redundant	5	0, 2, 6, 9, 11
15	larger than all: extend	6	0, 2, 6, 9, 11, 15

Table 3.15: Active lists for the 16-element sequence. Each row shows the longest list and the number of lists after the element is processed.

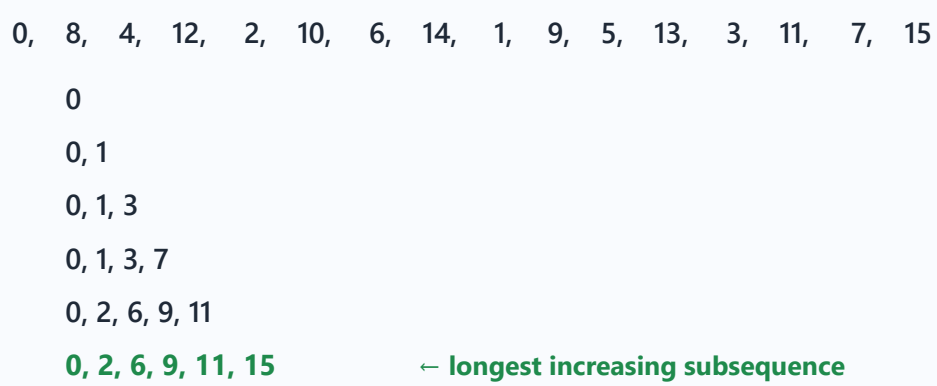


Figure 3.21: The six active lists when the sequence ends. The longest is 0, 2, 6, 9, 11, 15; the LIS length is 6.

When the sequence ends, the number of lists is the LIS length and the longest active list is an LIS. The correctness of the method rests on the following invariant.

> **Theorem 3.1 (The ends are increasing)**
>
> After every step, the last element of the active list of length k is strictly smaller than the last element of the active list of length $k + 1$.

Proof. The first k elements of the active list of length $k + 1$ form an increasing list of length k, and its last element is smaller than the last element of the $(k + 1)$-list. The active list of length k is the one with the smallest last element among lists of that length; its last element cannot exceed this element. Hence the last element of the k-list is less than or equal to the second-to-last element of the $(k + 1)$-list, which is strictly smaller than the last element of the $(k + 1)$-list. □

The theorem has two important consequences. First, which list the new element extends can be found by searching a sorted array: the array of last elements is increasing, and the place sought is the first list whose last element is greater than or equal to x. Binary search finds it in $O(\log n)$ time. Second, there is no need to store the lists themselves; storing only their last elements is enough.

3.6.5 The O(n log n) code

The code below stores only the last elements of the active lists. `tail[k]` is the smallest last element among increasing subsequences of length $k + 1$. `tail[0..len - 1]` is always increasing; initially `len = 0`.

```
static int lisFast(int[] a) {
  int[] tail = new int[a.length];
  int len = 0;               // number of active lists
  for (int x : a) {
    int lo = 0, hi = len;    // first place with tail[lo] >= x
    while (lo < hi) {
      int mid = (lo + hi) / 2;
      if (tail[mid] < x) lo = mid + 1;
      else hi = mid;
    }
    tail[lo] = x;            // replace / extend
    if (lo == len) len++;
  }
  return len;
}
```

For each *x*, the `while` loop finds by binary search the first position satisfying `tail[lo] >= x`. The search range `[lo, hi)` is half-open; `hi` is initially `len`. If `tail[mid] < x`, the place sought is to the right of mid; otherwise mid may also be a candidate and the range is narrowed to the left. When the loop ends, `lo` is the place where *x* is to be written. If `lo == len`, *x* is larger than all last elements and the number of lists grows; otherwise *x* takes the place of a larger last element. This latter case is the third rule of the active-lists method: the list of the same length now ends with a smaller number.

x	lo	cmp.	Action	len	tail[0..len − 1]
5	0	0	list empty; grows	1	5
2	0	1	2 replaces 5	1	2
8	1	1	larger than all; grows	2	2, 8
6	1	2	6 replaces 8	2	2, 6
3	1	2	3 replaces 6	2	2, 3
6	2	1	larger than all; grows	3	2, 3, 6
9	3	2	larger than all; grows	4	2, 3, 6, 9
7	3	2	7 replaces 9	4	2, 3, 6, 7

Table 3.16: Trace of `lisFast({5, 2, 8, 6, 3, 6, 9, 7})`; "cmp." is the number of comparisons in the binary search.

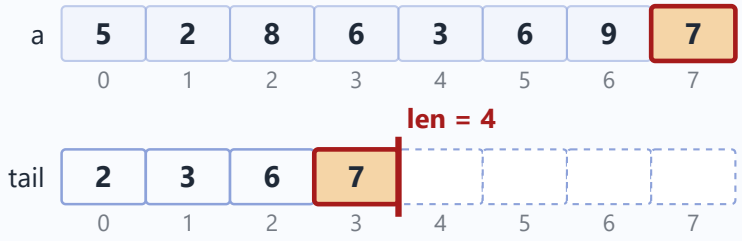


Figure 3.22: The end of the trace: `len = 4`, `tail = [2, 3, 6, 7]`.

The method returns 4, the same result as the $O(n^2)$ algorithm. For each element the binary search makes at most $\lfloor \log_2(len + 1) \rfloor$ comparisons. The total time is $O(n \log n)$ and the extra memory is $O(n)$. On an array of one million elements the $O(n^2)$ algorithm makes about $5 \cdot 10^{11}$ comparisons, while this method finishes with around twenty million comparisons.

One must be careful about the contents of the `tail` array. It need not be an LIS. In the 16-element example, when the process ends `tail = [0, 1, 3, 7, 11, 15]`. The length is correct (6), but these numbers do not occur in the sequence in this order: 7 is at position 14, while 11 is at position 13, that is, before 7. If the sequence itself is needed, whenever an element is written into `tail`, the index of the element at the position to its left (`lo - 1`) is stored as its predecessor index; the traceback is done as in the $O(n^2)$ version.

Common mistake

In the binary search, `tail[mid] <= x` is written instead of `tail[mid] < x`. Then an equal element is written not in place of the equal last element but to its right, and the list grows. The algorithm now finds the longest non-decreasing (equality allowed) subsequence rather than the strictly increasing one. For {2, 2, 2} the correct answer is 1; the faulty code answers 3.

3.7 Longest common substring

3.7.1 Problem and recurrence

Two strings are given. What is the longest piece that occurs contiguously in both? This problem is called the **longest common substring**. In plagiarism detection the longest passage that is identical in two texts, and in version comparison the common block of two files, are found with this question.

As an example take the strings "abdca" and "cbda". Their common substrings are "a", "b", "d", "c", and "bd". The longest is "bd", with length 2.

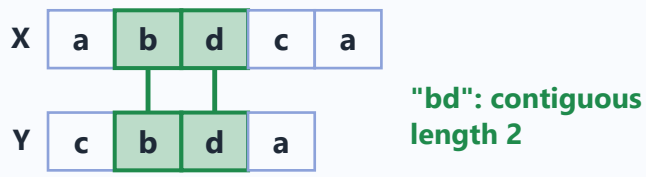


Figure 3.23: The longest common substring of "abdca" and "cbda" is "bd".

The brute-force method searches for every substring of the first string in the second string. A string of length m has $O(m^2)$ substrings; if each is searched for in a string of length n, the time is $O(m^2 \cdot n)$ or worse. DP answers the same question in $O(m \cdot n)$ time.

The choice of state is similar to that of LIS. The place where the common piece ends must be fixed. Let the strings be X (length m, the columns of the table) and Y (length n, the rows of the table).

Definition 3.11

$L[i][j]$ is the length of the longest common piece that ends *exactly at the end* of the first i characters of Y and the first j characters of X.

> *if* $Y[i-1] = X[j-1]$: $L[i][j] = L[i-1][j-1] + 1$
> *otherwise:* $L[i][j] = 0$

Row 0 and column 0 are zero. The answer is the largest value in the table.

If the two characters are equal, the common piece ending at the previous characters grows by one character: this means adding 1 to the diagonal cell. If the characters differ, there is no common piece ending at these two positions; contiguity is broken and the counter is reset to zero. The answer is not the last cell, because the common piece can end anywhere in the strings.

3.7.2 Filling the table

Let the columns be X = "abdca" and the rows Y = "cbda". The table is 5×6 and is filled row by row.

	j = 0	1 (a)	2 (b)	3 (d)	4 (c)	5 (a)
i = 0	0	0	0	0	0	0
1 (c)	0	0	0	0	1	0
2 (b)	0	0	1	0	0	0
3 (d)	0	0	0	**2**	0	0
4 (a)	0	1	0	0	0	1

Table 3.17: The longest common substring table; columns X = "abdca", rows Y = "cbda".

Most of the table is zero; only the cells where matching characters intersect are filled. In row 1, 'c' matches only $X[3]$ = 'c': $L[1][4] = L[0][3] + 1 = 1$. In row 2, 'b' matches $X[1]$: $L[2][2] = 1$. In row 3, 'd' matches $X[2]$ and extends the $L[2][2] = 1$ on its diagonal: $L[3][3] = 2$. This is the common piece that starts with "b" and ends with "d". In row 4, 'a' matches in two places. $L[4][1] = L[3][0] + 1 = 1$. $L[4][5] = L[3][4] + 1 = 0 + 1 = 1$; since 'd' and 'c' on the diagonal do not match, the piece does not grow.

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	0
b	2	0	0	1	0	0	0
d	3	0	0	0	2	0	0
a	4	0	1	0	0	0	1

Figure 3.24: The largest value is *L[3][3] = 2*; going 2 steps back along the diagonal gives *X[1..2]* = "bd". The last cell *L[4][5] = 1* is not the answer.

The piece itself is read from the largest cell. The largest value is *L[3][3] = 2*; the piece ends at the 3rd character of *X* and has length 2. So the piece is *X[1..2]* = "bd". No separate walk backwards is needed here: the end position and the length determine the piece on their own.

3.7.3 Code

The slides give only the recurrence and the table for this problem. The code below translates the same recurrence into Java. The code tracks the largest value and its end position in *X* while the table is filled; at the end it returns the piece with `substring`.

```
String longestCommonSubstring(String x, String y) {
  int m = x.length(), n = y.length();
  int[][] L = new int[n + 1][m + 1];
  int best = 0, end = 0;            // end: where the piece ends in X
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      if (y.charAt(i - 1) == x.charAt(j - 1)) {
        L[i][j] = L[i - 1][j - 1] + 1;
        if (L[i][j] > best) { best = L[i][j]; end = j; }
      }                                 // on a mismatch L[i][j] stays 0
  return x.substring(end - best, end);
}
```

Nothing is written to mismatching cells; Java's initial value 0 stays there. The condition `L[i][j] > best` is a strict greater-than; if another piece of equal length is found, the first one found is kept. For "abdca" and "cbda" the method returns "bd", and for "ababc" and "babca" it returns "babc". The time and memory are $O(m \cdot n)$. Since each cell depends only on the cell on its diagonal, the memory can be reduced to two rows, hence to $O(\min(m, n))$.

Common mistake

The answer is read from the last cell. The last cell is the length of the common piece that ends *at the end of both* strings. In the example $L[4][5] = 1$, but the answer is 2. For the common substring, the answer is the largest value of the table.

3.8 Longest common subsequence

3.8.1 Problem and recurrence

When the contiguity requirement is removed, the problem changes. We look for the longest sequence that occurs in both sequences in the same relative order; the characters in between may be skipped. This problem is called the **longest common subsequence** (LCS). `diff` and `git` find the unchanged lines of two files with LCS; the changed lines are those left outside the LCS. The similarity of DNA sequences and plagiarism detection are also applications of LCS.

For "abdca" and "cbda" the LCS is "bda" with length 3; the common substring had only 2. Another example: the LCS of "AGGTAB" and "GXTXAYB" is "GTAB", with length 4. "GTAB" is a subsequence of both strings but a substring of neither. The LCS need not be unique; its length is. The sequences "ABCBDAB" and "BDCABA" have three different LCSs: "BCBA", "BDAB", and "BCAB".

The recurrence is found by looking at the last characters of the two sequences. The state is defined in the same form as for the common substring, but its meaning is different: $L[i][j]$ is the LCS length of the first i characters of Y and the first j characters of X. This time the piece need not end at the end.

Theorem 3.2 (Optimal substructure of LCS)

> *if* $Y[i-1] = X[j-1]$: $L[i][j] = L[i-1][j-1] + 1$
> *otherwise:* $L[i][j] = \max(L[i-1][j],\ L[i][j-1])$

Row 0 and column 0 are zero; the answer is the last cell $L[n][m]$.

Proof. Let the last characters be equal; call this character c. Any common subsequence that does not end with c can be extended by appending c. Hence an LCS ends with c, and the part before c is a common subsequence of the two sequences with their last characters removed. If this part could be longer, the LCS would be longer too; so it is itself an LCS: $L[i-1][j-1] + 1$. If the last characters differ, a common subsequence does not use at least one of these two characters; that character can be dropped. If the dropped character is Y's, the answer is $L[i-1][j]$; if it is X's, it is $L[i][j-1]$. Since we do not know which one to drop, the larger is taken. □

The recurrences of the two problems differ only in the mismatch case. Table 3.18 summarizes the differences.

	Match	Mismatch	Answer
Common substring	diagonal + 1	0	largest value of the table
Common subsequence (LCS)	diagonal + 1	max(up, left)	last cell

Table 3.18: Comparison of the recurrences of the common substring and the common subsequence (LCS).

The difference in the mismatch case reflects the essence of the two problems. In the common substring, a mismatching character breaks the chain; the counter is reset. In LCS, a mismatching character is merely skipped; the common subsequence found so far is not lost and is carried over from above or from the left. Hence the values of the LCS table never decrease going right and down; the last cell is the largest value.

3.8.2 Code and trace

```
static int lcs(String x, String y) {
  int m = x.length(), n = y.length();
  int[][] L = new int[n + 1][m + 1];
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      if (y.charAt(i - 1) == x.charAt(j - 1))
        L[i][j] = L[i - 1][j - 1] + 1;
      else
        L[i][j] = Math.max(L[i - 1][j],
                           L[i][j - 1]);
  return L[n][m];
}
```

The code applies the recurrence with two nested loops. The characters of the strings are accessed with `charAt(i - 1)` and `charAt(j - 1)`, because row i of the table represents the i-th character of Y, that is, the character with 0-based index $i - 1$. For the same "abdca" and "cbda" example, the table fills as follows:

	j = 0	1 (a)	2 (b)	3 (d)	4 (c)	5 (a)
i = 0	0	0	0	0	0	0
1 (c)	0	0	0	0	**1**	1
2 (b)	0	0	**1**	1	1	1
3 (d)	0	0	1	**2**	2	2
4 (a)	0	**1**	1	2	2	**3**

Table 3.19: The LCS table; columns X = "abdca", rows Y = "cbda". Bold cells are matches.

The computation of some cells is as follows. *L[1][5]*: 'c' ≠ 'a'; up is 0, left is 1; the value is 1. The common subsequence found by the character 'c' has been carried to the right. *L[3][3]*: 'd' = 'd'; 1 is added to the diagonal *L[2][2] = 1* and it becomes 2 ("bd"). *L[4][3]*: 'a' ≠ 'd'; up is 2, left is 1; the value is 2. *L[4][5]*: 'a' = 'a'; 1 is added to the diagonal *L[3][4] = 2*: 3. In the common substring table this diagonal cell was 0; in LCS, "bd" had been carried to the right and was 2. Once the contiguity requirement is lifted, 'a' can be appended to the end of "bd".

		0	1	2	3	4	5
			a	b	d	c	a
	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1
b	2	0	0	1	1	1	1
d	3	0	0	1	2	2	2
a	4	0	1	1	2	2	3

Figure 3.25: The table is filled: *L[4][5] = 3*. 20 cells were computed, each in *O(1)*.

The time and memory are $O(m \cdot n)$. If only the length is needed, two rows suffice and the memory drops to $O(\min(m, n))$. If the sequence itself is also wanted, the whole table must be kept.

3.8.3 Reconstruction

The LCS itself is read back starting from the last cell. At each cell we look at which branch of the recurrence was used:

- If the characters are equal, the character is added to the LCS and we move to the diagonal cell.
- If they are not equal, we go in the direction the value came from: left if the left cell is larger than the one above, up otherwise.
- We stop when we reach row 0 or column 0. Since the characters are found from last to first, the result is reversed.

Cell	Characters	Decision	Found (from the end)
L[4][5] = 3	'a' = 'a'	append, diagonal	a
L[3][4] = 2	'd' ≠ 'c'	left (2) > up (1): go left	a
L[3][3] = 2	'd' = 'd'	append, diagonal	da
L[2][2] = 1	'b' = 'b'	append, diagonal	bda
L[1][1] = 0	'c' ≠ 'a'	up (0) ≥ left (0): go up	bda

Cell	Characters	Decision	Found (from the end)
L[0][1]	—	boundary; stop	bda

Table 3.20: LCS traceback for "abdca" and "cbda".

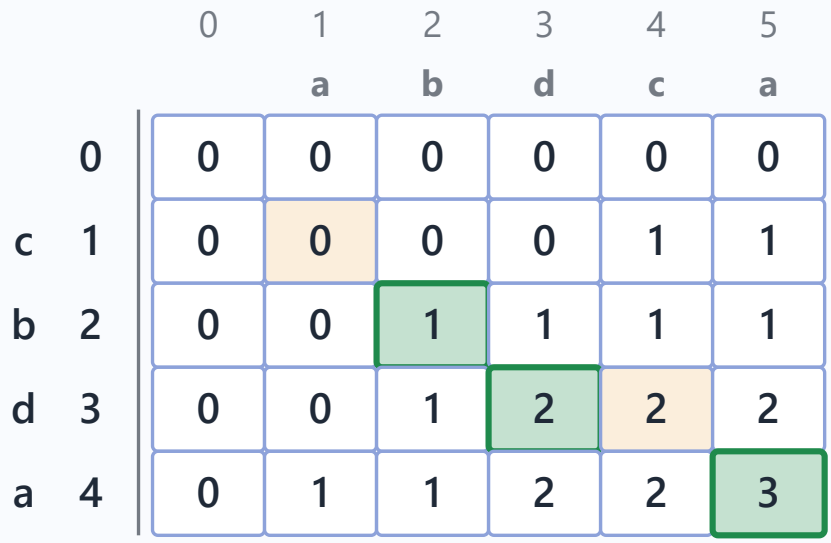


Figure 3.26: The traceback path. The green cells are matches: LCS = "bda", length 3. The traceback takes $O(m + n)$ steps.

At each step either the row, the column, or both decrease by one; the traceback takes at most $m + n$ steps. The following method fills the table and returns the LCS itself.

```
String lcsString(String x, String y) {
  int m = x.length(), n = y.length();
  int[][] L = new int[n + 1][m + 1];
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      L[i][j] = y.charAt(i - 1) == x.charAt(j - 1) ? L[i - 1][j - 1] + 1
              : Math.max(L[i - 1][j], L[i][j - 1]);
  StringBuilder sb = new StringBuilder();
  int i = n, j = m;
  while (i > 0 && j > 0)
    if (y.charAt(i - 1) == x.charAt(j - 1)) {   // match: diagonal
      sb.append(x.charAt(j - 1)); i--; j--;
    } else if (L[i][j - 1] > L[i - 1][j]) j--;  // left larger: go left
    else i--;                                   // otherwise: go up
  return sb.reverse().toString();
}
```

The method returns "bda" for "abdca" and "cbda", and "GTAB" for "AGGTAB" and "GXTXAYB". The rule of going up on a tie determines which LCS is found when there are several. For "ABCBDAB" (columns) and "BDCABA" (rows), the method finds "BDAB"; if the rule is changed so that ties go left, a different

LCS may be found.

3.8.4 A DNA example

In bioinformatics, how similar two DNA sequences are can be measured by the length of their LCS. The sequences in Figure 3.27 are ACCGGTGGACAATTCA and GGAAAGAGATATGCAC. Both consist of 16 bases.

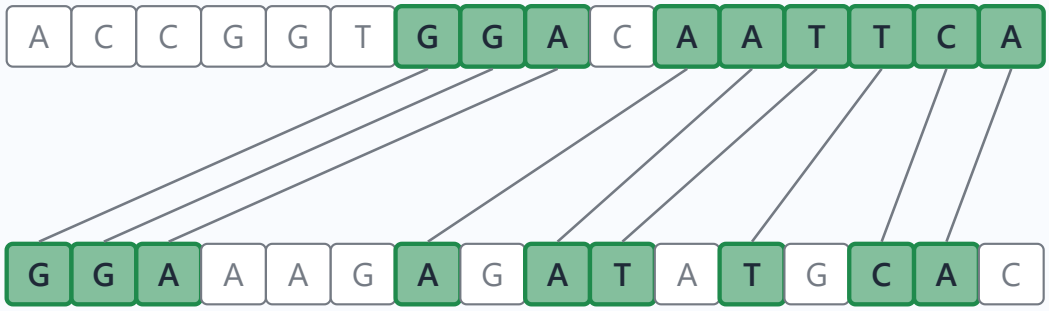


Figure 3.27: Two DNA sequences and an LCS: GGAAATTCA, length 9. The lines joining matched bases do not cross.

The lines in the figure are the visual counterpart of a common subsequence. The lines joining matched characters do not cross, because the relative order is kept in both sequences. Every matching built with non-crossing lines is a common subsequence. The LCS is the matching with the most lines that can be drawn this way. The DP result is GGAAATTCA, of length 9. If the sequences are swapped, the traceback may find a different LCS such as GGGGAATCA; the length is still 9.

Matching by eye is misleading. For example, AAAATTCA is also a valid common subsequence: in the first sequence the A's are at positions 0, 8, 10, and 11, followed by T, T, C, A; the same order is found in the second sequence. But its length is 8 and it is not the longest. Only the DP table, which implicitly tries all options, gives the right answer. A 16×16 table finishes this job with 256 cells; trying all possible subsequences would mean searching $2^{16} = 65{,}536$ subsequences in the other sequence.

3.9 Other classic DP problems

The problems studied in these notes follow a pattern: the state is defined by a few integers, the recurrence tries the options of the last decision, and the table is filled in a suitable order. The same pattern solves many other classic problems. Table 3.21 summarizes seven of them; then four are worked through with small examples.

Problem	State and recurrence	Time
Edit distance	$D[i][j] = \min(D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + [a_i \neq b_j])$; $D[i][0] = i$, $D[0][j] = j$	$O(m \cdot n)$

Problem	State and recurrence	Time
Coin change	fewest coins: $P[x] = 1 + \min P[x - c]$ $(c \le x)$; $P[0] = 0$	$O(k \cdot X)$
Maximum subarray	$V(i) = \max(a[i], V(i-1) + a[i])$ (Kadane); answer $\max V(i)$	$O(n)$
Matrix-chain multiplication	$M[i][j] = \min \{ M[i][k] + M[k+1][j] + p_{i-1} \cdot p_k \cdot p_j : i \le k < j \}$	$O(n^3)$
Traveling salesman (Held–Karp)	$dp[S][i]$: shortest path that visits the set S and ends at i; $dp[S][i] = \min dp[S - i][j] + d(j, i)$	$O(n^2 \cdot 2^n)$
Bellman–Ford	$d_k(v)$: shortest path with at most k edges; $d_k(v) = \min(d_{k-1}(v), d_{k-1}(u) + w(u, v))$	$O(V \cdot E)$
Optimal binary search tree	$e[i][j] = \min \{ e[i][r-1] + e[r+1][j] : i \le r \le j \} + w(i, j)$	$O(n^3)$

Table 3.21: Other classic dynamic programming problems.

3.9.1 Edit distance

The **edit distance** (Levenshtein distance) between two strings is the smallest number of single-character operations needed to turn one into the other. There are three kinds of operations: deleting a character, inserting a character, and replacing a character with another one. A spell checker suggests dictionary words whose distance to a misspelled word is small.

$D[i][j]$ is the minimum cost of turning the first i characters of a into the first j characters of b. There are three options for the last character. The last character of a is deleted: $D[i - 1][j] + 1$. The last character of b is inserted: $D[i][j - 1] + 1$. The last characters are matched: free if they are equal, otherwise with one replacement, $D[i - 1][j - 1] + [a_i \ne b_j]$. The square brackets mean 1 if the condition is true and 0 if it is false. The base cases are clear: from the empty string to a string of j characters takes j insertions, and from a string of i characters to the empty string takes i deletions.

```
int editDistance(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;     // i deletions
  for (int j = 0; j <= n; j++) D[0][j] = j;     // j insertions
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++) {
      int diff = a.charAt(i - 1) == b.charAt(j - 1) ? 0 : 1;
      D[i][j] = Math.min(Math.min(D[i - 1][j] + 1,   // delete
                                  D[i][j - 1] + 1),  // insert
                         D[i - 1][j - 1] + diff);    // replace / same
    }
  return D[m][n];
}
```

Example 3.7

What is the edit distance between "kitap" and "kayak"?

	""	k	a	y	a	k
""	0	1	2	3	4	5
k	1	0	1	2	3	4
i	2	1	1	2	3	4
t	3	2	2	2	3	4
a	4	3	2	3	2	3
p	5	4	3	3	3	**3**

The result is *D[5][5] = 3*. Reading along the diagonal, the characters k and a match; i → a, t → y, and p → k are replaced. For example *D[2][2]*: 'i' ≠ 'a'; the options are *D[1][2] + 1 = 2*, *D[2][1] + 1 = 2*, and *D[1][1] + 1 = 1*; the smallest is 1. *D[4][4]*: 'a' = 'a'; the diagonal *D[3][3] = 2* is carried over for free.

Edit distance is closely related to LCS. If only insertion and deletion are allowed, the minimum number of operations is $m + n - 2 \cdot LCS$: the characters not in the LCS are deleted from one string and inserted into the other.

3.9.2 Coin change

Let the coin denominations be $c_1, \ldots, c_k$ and an amount X be given. This problem has two famous questions: what is the fewest number of coins that pay the amount, and in how many different ways can it be paid? Since there is an unlimited supply of each coin, both are relatives of unbounded knapsack.

Fewest coins. *P[x]* is the fewest number of coins needed to pay the amount *x*. If the last coin given is *c*, then $x - c$ remains: $P[x] = 1 + \min P[x - c]$ $(c \le x)$, $P[0] = 0$. Small amounts must be ready first; *x* is filled from 1 to *X*. The table has $X + 1$ cells, and *k* coins are tried in each cell: $O(k \cdot X)$ time, $O(X)$ memory. This too is a pseudo-polynomial time.

```
int minCoins(int[] coins, int X) {
  int[] P = new int[X + 1];            // P[0] = 0
  for (int x = 1; x <= X; x++) {
    P[x] = Integer.MAX_VALUE;
    for (int c : coins)
      if (c <= x && P[x - c] != Integer.MAX_VALUE)
        P[x] = Math.min(P[x], P[x - c] + 1);
  }
  return P[X];                         // MAX_VALUE: cannot be paid
}

long countWays(int[] coins, int X) {
  long[] d = new long[X + 1];
  d[0] = 1;                            // the only way to pay 0: no coins
  for (int c : coins)                   // outer loop over coins
    for (int x = c; x <= X; x++)
      d[x] += d[x - c];
  return d[X];
}
```

Example 3.8

Coins {1, 3, 4}, amount 6. How many coins are needed at least? How many does the greedy method use?

Table: *P = [0, 1, 2, 1, 1, 2, 2]*. For example *P[5] = 1 + min(P[4], P[2], P[1]) = 1 + 1 = 2* (4 + 1). Last cell: *P[6] = 1 + min(P[5], P[3], P[2]) = 1 + min(2, 1, 2) = 2*; the payment is 3 + 3. The greedy method first takes the largest coin: 4, then 1, then 1. It uses three coins and is not optimal. In most currency systems used every day the greedy method gives the right result, but this is not true for every currency system.

Number of ways. *d[x]* is the number of different ways to pay the amount *x*; *d[0] = 1* (giving no coins). Here the order of the loops is decisive. If the outer loop is over coins and the inner loop over amounts, each way is counted once: coins are added in a fixed order, and "2 + 1" and "1 + 2" are considered the same. If the loops are swapped, ordered sequences are counted. For {1, 2, 5} and the amount 5, the correct answer is 4: 5, 2 + 2 + 1, 2 + 1 + 1 + 1, and 1 + 1 + 1 + 1 + 1. With the loops reversed, the result would be 9.

Common mistake

It is assumed that the greedy choice is always right in coin change. {1, 3, 4} with amount 6 is a counterexample: greedy gives 3 coins, DP gives 2. As in knapsack, unless the correctness of the greedy choice is proved separately, DP is needed.

3.9.3 Maximum subarray

We look for the contiguous piece of an array with the largest sum. The recurrence was written while discussing optimal substructure: $V(i) = \max(a[i], V(i-1) + a[i])$. At each step there are two options: extend the previous piece or restart with $a[i]$. If the sum of the previous piece is negative, carrying it is a loss; restarting is better. Since $V(i)$ depends only on $V(i-1)$, a single variable suffices.

```
int kadane(int[] a) {
  int V = a[0], best = a[0];
  for (int i = 1; i < a.length; i++) {
    V = Math.max(a[i], V + a[i]);   // restart or extend
    best = Math.max(best, V);
  }
  return best;
}
```

Example 3.9

What is the maximum subarray of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$?

The V values are −2, 1, −2, 4, 3, 5, 6, 1, 5 in turn. At index 1 the previous sum is −2, so we restart: $V = \max(1, -1) = 1$. At index 3 the previous sum is −2; we restart with 4. Then the piece grows: $4 - 1 = 3$, $3 + 2 = 5$, $5 + 1 = 6$. The −5 lowers the piece to 1 but does not push it below zero. The largest value is 6: $[4, -1, 2, 1]$. The time is $O(n)$ and the memory is $O(1)$.

3.9.4 Matrix-chain multiplication

The ordinary product of a $p \times q$ matrix and a $q \times r$ matrix requires $p \cdot q \cdot r$ scalar multiplications. Matrix multiplication is associative: $(AB)C = A(BC)$. The result is the same, but the cost can differ greatly depending on the parenthesization. The problem is to find the parenthesization that computes a chain of matrices with the fewest scalar multiplications.

Let matrix A_i have size $p_{i-1} \times p_i$. $M[i][j]$ is the minimum cost of the chain $A_i \dots A_j$. The last multiplication of this chain splits it in two at some point k: first $A_i \dots A_k$ and $A_{k+1} \dots A_j$ are computed, then the two results of sizes $p_{i-1} \times p_k$ and $p_k \times p_j$ are multiplied. All values of k are tried. Here the states are filled by chain length: short chains must be ready first.

```
long matrixChain(int[] p) {          // matrix A_i is p[i-1] × p[i]
  int n = p.length - 1;
  long[][] M = new long[n + 1][n + 1];
  for (int len = 2; len <= n; len++)      // chain length
    for (int i = 1; i + len - 1 <= n; i++) {
      int j = i + len - 1;
      M[i][j] = Long.MAX_VALUE;
      for (int k = i; k < j; k++)
        M[i][j] = Math.min(M[i][j],
            M[i][k] + M[k + 1][j] + (long) p[i - 1] * p[k] * p[j]);
    }
  return M[1][n];
}
```

Example 3.10

A_1 is 10 × 30, A_2 is 30 × 5, and A_3 is 5 × 60 (p = *10, 30, 5, 60*). Which is the best parenthesization?

$(A_1A_2)A_3$: $10 \cdot 30 \cdot 5 + 10 \cdot 5 \cdot 60 = 1500 + 3000 = 4500$ multiplications. $A_1(A_2A_3)$: $30 \cdot 5 \cdot 60 + 10 \cdot 30 \cdot 60 = 9000 + 18{,}000 = 27{,}000$ multiplications. The first is six times cheaper; the method returns 4500.

The number of states is $O(n^2)$ (the pairs $i \le j$); in each state $O(n)$ split points are tried. The time is $O(n^3)$. Trying every parenthesization one by one would mean as many options as the Catalan numbers: 4862 for 10 matrices and more than 1.7 billion for 20 matrices.

3.9.5 Traveling salesman, Bellman–Ford, and optimal binary search trees

The last three problems in the table show that a state can take different forms: a set and a vertex, a vertex and an edge count, a range of keys.

- **Traveling salesman (Held–Karp).** The state is a set and a vertex: *dp[S][i]* is the shortest path that starts at the origin, visits all vertices of the set S, and ends at i. The set is stored as a bit mask. There are $2^n \cdot n$ states, and each tries $O(n)$ predecessors: $O(n^2 \cdot 2^n)$. This time is still exponential, but much smaller than the $(n-1)!$ of trying every tour. For 20 vertices, $20^2 \cdot 2^{20} \approx 4.2 \cdot 10^8$ against $19! \approx 1.2 \cdot 10^{17}$.
- **Bellman–Ford.** The state is a vertex and an edge count: $d_k(v)$ is the shortest path using at most k edges. In each round all edges are relaxed; $V - 1$ rounds suffice. The table is reduced to one dimension and the time is $O(V \cdot E)$.
- **Optimal binary search tree.** If the access probabilities of the keys are known, we look for the tree with the smallest expected search cost. The state is a range of keys; the recurrence tries every root choice for the range. The structure is the same as in matrix-chain multiplication and the time

is $O(n^3)$.

3.10 Common mistakes

Dynamic programming mistakes usually do not produce a compile error. The code runs, returns a number, and the number is wrong. The mistakes below, together with the list in the slides, collect those met throughout these notes.

Common mistake

Forgetting the base cases. If *K[0][c]*, *dp[i][0]*, or fib(0) and fib(1) are initialized wrongly, the whole table shifts. In subset sum, if column 0 is not made true, every answer is "no"; in coin change, if infinity is written instead of *P[0] = 0*, no amount can be paid.

Common mistake

Index shift. The table has size *n + 1* but the array has *n* elements: row *i* uses `v[i - 1]`, and row *i* of the LCS table uses `charAt(i - 1)`. Writing `v[i]` shifts every row to the next element and runs out of the array in the last row.

Common mistake

Wrong loop direction. In one-dimensional 0/1 knapsack, scanning the capacity from small to large takes an item more than once; the result becomes the answer of unbounded knapsack. For the number of ways in coin change, putting the outer loop over amounts counts different orders of the same payment separately.

Common mistake

Confusing substring with subsequence. In the common substring, a mismatch writes 0 and the answer is the largest value of the table. In LCS, a mismatch writes *max(up, left)* and the answer is the last cell. In LIS too, the answer is not the last cell but the largest *dp[i]*.

Common mistake

Assuming the greedy choice is enough. In 0/1 knapsack (the 60, 100, 120 Ł example) and in coin change ({1, 3, 4} with 6) the greedy choice is not optimal. The greedy method is used only if its correctness has been proved.

Common mistake

Taking $O(n \cdot W)$ to be polynomial. W is a numeric value; the input size is $\log W$ bits. For large W the table cannot be built.

Common mistake

Choosing the wrong "not computed" marker in the cache. In an `int` table, 0 means both "empty" and "answer 0"; states whose answer is 0 are recomputed every time. `Integer` with `null`, or a table filled with −1, avoids this confusion.

Common mistake

Losing the traceback while saving memory. A one-dimensional table gives the optimal value, but the rows needed to read the choices back have been deleted. If the choices are wanted too, either the whole table or a separate decision table is kept.

3.11 Summary

- Dynamic programming = overlapping subproblems + optimal substructure. Each subproblem is solved once and its result is stored in a table.
- Design steps: the structure of an optimal solution, the state and recurrence (with base cases), the order of computation, the table size, filling the table, and reconstructing the solution.
- Total time ≈ number of states × work per state. This rule explains every complexity in these notes.
- Memoization works top-down and solves only the states that are needed; tabulation works bottom-up, has no call overhead, and lends itself to memory reduction. Their asymptotic times are the same.
- A greedy approach looks at a single option at each step; it is correct only if the greedy-choice property has been proved. In divide and conquer the subproblems are independent and no table is needed.

Problem	State / recurrence	Time · memory
Fibonacci	$F(n) = F(n-1) + F(n-2)$; memoization or table	$O(n) \cdot O(1)$
0/1 knapsack	$K[i][c] = \max(K[i-1][c], v_i + K[i-1][c-w_i])$	$O(nW) \cdot O(W)$
Unbounded knapsack	$dp[i][c] = \max(dp[i-1][c], p_i + dp[i][c-w_i])$	$O(nW) \cdot O(W)$
Subset sum	$dp[i][j] = dp[i-1][j] \vee dp[i-1][j-a_i]$	$O(n \cdot target)$
LIS	$dp[i] = 1 + \max dp[j]$ ($j < i$, $a[j] < a[i]$); or active lists	$O(n^2)$ / $O(n \log n)$

Problem	State / recurrence	Time · memory
Longest common substring	if equal, diagonal + 1, otherwise 0; answer is the largest value in the table	$O(mn)$
LCS	if equal, diagonal + 1, otherwise max(up, left); answer is the last cell	$O(mn)$

Table 3.22: Summary of the problems studied in these notes.

- 0/1 knapsack, subset sum, and coin change are pseudo-polynomial: the time depends on the value of a number in the input, not on its bit length.
- In a one-dimensional table the loop direction determines the problem: large to small gives 0/1, small to large gives unbounded.
- The solution itself is read back from the table: the rows where the value changes in knapsack, the diagonal steps in LCS, the chain of predecessor indices in LIS.

3.12 Key terms

dynamic programming (DP) subproblem overlapping subproblems optimal substructure memoization tabulation top-down / bottom-up traceback greedy-choice property 0/1 knapsack fractional knapsack unbounded knapsack pseudo-polynomial subset sum longest increasing subsequence (LIS) longest common substring longest common subsequence (LCS) edit distance coin change matrix chain multiplication

3.13 Test Yourself

Questions 2 and 3 use the same items: (value, weight) = (12, 2), (17, 3), (30, 5); capacity 6.

1. In a plain recursive call of fib(6), how many times is fib(2) called? With memoization, how many times is fib(2) computed, and what is the total number of calls?

   Answer

   In the tree of fib(n), fib(k) is called $F(n - k + 1)$ times: for fib(2), $F(5) = 5$ times. Plain recursion makes $2 \cdot F(7) - 1 = 25$ calls in total. With memoization fib(2) is computed once and its second call returns from the table; the total number of calls is $2 \cdot 6 - 1 = 11$.

2. What is the last row of the 0/1 knapsack table? What is the maximum value, and which items are

chosen?

Answer

The rows are: *[0, 0, 12, 12, 12, 12, 12]*, *[0, 0, 12, 17, 17, 29, 29]*, *[0, 0, 12, 17, 17, 30, 30]*. The answer is *K[3][6] = 30*. Traceback: *30 ≠ K[2][6] = 29*, so item 3 is taken and *c = 1*; no other item fits into 1 kg. Only the item worth 30 is chosen. 12 + 17 = 29 falls short by one.

3. What would the answer be if any number of copies of the same items could be taken? What number would be returned if the inner loop of the one-dimensional 0/1 code were written from small to large?

Answer

Unbounded: *dp = [0, 0, 12, 17, 24, 30, 36]*; the answer is 36, three items worth 12 (2 kg). The one-dimensional 0/1 code scanned from small to large also returns 36, because that code now solves the unbounded knapsack.

4. In the 0/1 knapsack DP, how does the running time change if *W* doubles? What happens if one bit is added to the binary representation of *W*?

Answer

In both cases the table width roughly doubles, and so does the running time. A one-bit increase grows the input size by only one bit, yet the running time doubles. For a *b*-bit *W* the time is $O(n \cdot 2^b)$; this is what pseudo-polynomial means.

5. Does the set {2, 3, 7, 8, 10} have a subset with sum 11? Which sums are reachable in the last row of the table?

Answer

Yes: {3, 8}; this is the only solution. The reachable sums in the last row are 0, 2, 3, 5, 7, 8, 9, 10, and 11 (1, 4, and 6 cannot be obtained). The traceback starts at *dp[5][11]*: since 11 can be formed without 10, 10 is not chosen; 8 is chosen (*j = 3*); 7 is not chosen; 3 is chosen (*j = 0*).

6. For the array [3, 10, 2, 1, 20, 4, 6, 7], what is the dp array of the $O(n^2)$ algorithm? What is the `tail` array when `lisFast` finishes?

Answer

dp = [1, 2, 1, 1, 3, 2, 3, 4]; the LIS length is 4, for example 3, 4, 6, 7. `tail` goes through [3], [3, 10], [2, 10], [1, 10], [1, 10, 20], [1, 4, 20], [1, 4, 6], [1, 4, 6, 7] in turn. In this example the final array is an LIS (1, 4, 6, 7), but that is a coincidence; the array need not be an LIS.

7. What are the longest common substring and LCS lengths for "kalem" and "kelam"?

   Answer

   The common substring has length 1: none of the two-letter pieces (ka, al, le, em versus ke, el, la, am) is shared. The LCS has length 3; "kam", "kem", and "klm" are three different LCSs. Lifting the contiguity requirement triples the length.

8. What is the edit distance between "kedi" and "kadın"?

   Answer

   It is 3: e → a and i → ı are replaced, and n is appended. The last row of the table is *[4, 3, 3, 2, 2, 3]*; *D[4][5] = 3*.

9. Coins {1, 5, 6, 9}, amount 11. How many coins are needed at least? How many does the greedy method use?

   Answer

   DP: 2 coins (5 + 6). Greedy: 9 + 1 + 1, 3 coins. In this currency system too, the greedy choice is not optimal.

10. A_1 is 5 × 10, A_2 is 10 × 3, and A_3 is 3 × 12. What is the fewest number of scalar multiplications needed?

    Answer

    $(A_1A_2)A_3 = 5 \cdot 10 \cdot 3 + 5 \cdot 3 \cdot 12 = 150 + 180 = 330$. $A_1(A_2A_3) = 10 \cdot 3 \cdot 12 + 5 \cdot 10 \cdot 12 = 360 + 600 = 960$. The minimum is 330.

3.14 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 14: presents the principles of DP, with proofs, through rod cutting, matrix-chain multiplication, LCS, and optimal binary search trees.
- J. Kleinberg and É. Tardos, *Algorithm Design* (2005), Chapter 6: discusses the choice of state step by step through weighted interval scheduling, knapsack, sequence alignment, and Bellman–Ford.
- S. Dasgupta, C. Papadimitriou, and U. Vazirani, *Algorithms* (2006), Chapter 6: interprets DP as shortest paths in a directed acyclic graph; treats LIS, edit distance, knapsack, and the traveling salesman problem briefly and clearly.
- R. Bellman, *Dynamic Programming* (1957): the first comprehensive account of the method and the principle of optimality.

3.15 References

1. R. Bellman, *Dynamic Programming*, Princeton University Press, 1957.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 14: Dynamic Programming).
3. S. Dasgupta, C. Papadimitriou, U. Vazirani, *Algorithms*, McGraw-Hill, 2006 (Chapter 6).
4. J. Kleinberg, É. Tardos, *Algorithm Design*, Pearson, 2005 (Chapter 6).

PART II

Hard Problems

Chapters 4–6

Backtracking

For some problems no fast, exact algorithm is known. Placing eight queens on a board so that none attacks another, filling in a Sudoku, and coloring a map with three colors are all of this kind. For such problems a solution is found by trying choices. Backtracking is a method for organizing that search. The solution is built step by step; as soon as a step breaks a rule, the last decision is undone and the next option is tried. These notes first explain how hard problems are classified. They then develop the general template of backtracking, the state-space tree, and pruning through maze and permutation examples. The n-queens problem is treated in detail with Java code and a line-by-line trace, and the min-conflicts heuristic approaches the same problem in a completely different way. Sudoku, graph coloring, Hamiltonian paths, word search, and subset sum are further applications of the template. The notes end with the complexity analysis, heuristics that strengthen pruning, and a comparison of backtracking with the other design methods.

LEARNING OUTCOMES

Concepts

Explain backtracking in terms of partial solutions, candidates, constraints, and pruning; draw the state-space tree of a small problem; tell the classes P, NP, NP-hard, and NP-complete apart.

Template

Write the recursive “choose → explore → unchoose” template in Java and adapt it to a new problem. Distinguish the version that stops at one solution from the version that counts all solutions.

Applications

Solve the n-queens, Sudoku, maze, graph coloring, Hamiltonian path, word search, and subset sum problems step by step and trace their code.

Analysis

Show why the worst case is exponential; evaluate with numbers the effect of pruning, of an O(1) constraint test, and of heuristics such as MRV and min-conflicts.

4.1 Hard problems: P, NP, and NP-hard

Most of the algorithms seen so far in the course are fast. Merge sort sorts an array in $O(n \log n)$ time. Dijkstra's algorithm finds shortest paths, and Kruskal's algorithm finds a minimum spanning tree, in polynomial time. But not every problem is so well behaved. Is there a path that passes through every vertex of a graph exactly once? Can the countries of a map be colored with three colors so that neighboring countries get different colors? For both questions, no polynomial-time algorithm has ever been found, and none has been proved impossible. Backtracking is a search method developed for problems of this kind. To see why it is needed, it helps first to look briefly at the classes of problem difficulty.

4.1.1 Decision problems and polynomial time

Complexity classes are usually defined in terms of **decision problems**, whose answer is only "yes" or "no." "Does this graph contain a Hamiltonian path?" is a decision problem. "What is the length of the shortest Hamiltonian tour in this graph?" is an **optimization problem**. An optimization problem can often be turned into a decision problem: "Is there a tour of length at most 100?" If the decision version could be solved quickly, binary search on the threshold would also yield the optimal value. The hardness of the decision version therefore says something about the hardness of the optimization version too.

An algorithm runs in **polynomial time** if its running time is $O(n^k)$ for some constant k, where n is the input size. $O(n)$, $O(n \log n)$, and $O(n^3)$ are polynomial; $O(2^n)$ and $O(n!)$ are not. The difference is barely visible on small inputs, but a chasm opens as the input grows. For $n = 60$, $n^3 = 216{,}000$ operations take a computer an instant, whereas $2^{60} \approx 1.15 \cdot 10^{18}$ operations take more than thirty years on a machine doing a billion operations per second. That is why polynomial time is taken to mean "solvable in practice."

4.1.2 P and NP

Definition 4.1

P is the class of decision problems that can be *solved* in polynomial time. **NP** is the class of decision problems for which a proposed solution can be *verified* in polynomial time. When the answer is "yes," there is a witness, also called a certificate, that shows it, and the correctness of the witness can be checked in polynomial time.

Two examples from P are two weighted-graph questions: "Is there a path from s to t of length at most 50?" and "Is there a spanning tree of weight at most 30?" Dijkstra's algorithm answers the first and Kruskal's algorithm the second in polynomial time.

The definition of NP is about checking, not solving. The Hamiltonian path problem is a good example. Finding a Hamiltonian path in a graph is hard. But if someone proposes an ordering of the vertices, checking that the ordering contains every vertex once and that consecutive vertices are adjacent takes $O(n)$ steps. Sudoku is of the same kind. Checking a filled-in grid is easy: each row, column, and box is scanned once. Filling in an empty grid requires search.

A problem that can be solved in polynomial time can also be verified in polynomial time: the verifier can ignore the proposed witness and solve the problem from scratch. So P lies inside NP. The real question goes the other way: is every problem that is easy to verify also easy to solve? This is known as the $P = NP$ question, and it is the most important open problem in computer science. Most researchers believe that $P \neq NP$, but this has not been proved.

4.1.3 Reductions, NP-hard, and NP-complete

The tool for comparing the difficulty of two problems is the **reduction**. If every instance of problem A can be transformed in polynomial time into an instance of problem B, and the two instances have the same answer, then A reduces to B. In that case a fast algorithm for B also solves A quickly: transform first, then run B's algorithm. So B is at least as hard as A.

Definition 4.2

A problem B is **NP-hard** if every problem in NP can be reduced to B in polynomial time. If B is both in NP and NP-hard, it is **NP-complete**. An NP-hard problem is at least as hard as the hardest problems in NP; it need not itself be in NP.

NP-complete problems are the “hardest” members of NP. If one of them could be solved in polynomial time, then by the reductions every problem in NP could be solved in polynomial time, and $P = NP$ would hold. The first NP-complete problem is Boolean satisfiability (SAT): given a formula built from Boolean variables, is there an assignment of values that makes it true? 3-SAT, a restricted form of SAT, is also NP-complete. Hamiltonian path and Hamiltonian cycle, graph 3-coloring, and the decision version of subset sum are likewise among the NP-complete problems.

The **optimization** versions of the traveling salesman problem (TSP) and of the knapsack problem, on the other hand, are NP-hard. These versions are not decision problems; they return a number or a tour. Checking that a proposed tour is the shortest one is not easy either. So they are not counted as being in NP, but because their decision versions are NP-complete, they are at least that hard.

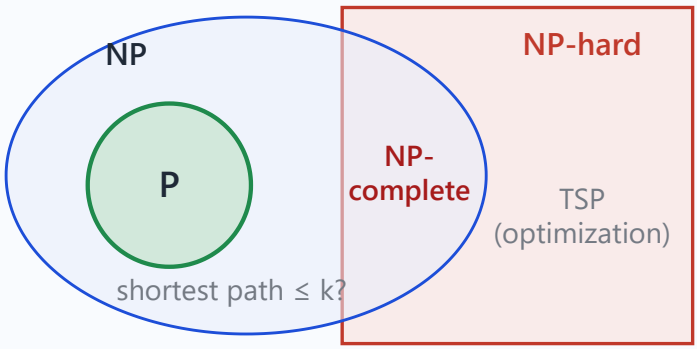


Figure 4.1: The relationship of the classes under the assumption $P \neq NP$. P lies inside NP. The intersection of the NP-hard region with NP consists of the NP-complete problems. The optimization version of TSP is NP-hard but not in NP.

The picture in Figure 4.1 rests on an assumption. If $P = NP$ held, the regions P, NP, and NP-complete would coincide (apart from a few trivial exceptions). The distinction the picture shows is the common belief, not our current knowledge.

Problem	Version	Class
Shortest path, minimum spanning tree	decision	P
SAT, 3-SAT	decision	NP-complete
Hamiltonian path, Hamiltonian cycle	decision	NP-complete
Graph 3-coloring	decision	NP-complete
Subset sum	decision	NP-complete
Traveling salesman, 0/1 knapsack	optimization	NP-hard

Table 4.1: Classes of the problems that appear in these notes.

4.1.4 What can be done about exact solutions?

If $P \neq NP$, there is no polynomial-time **exact** algorithm for NP-complete and NP-hard problems. Three routes remain.

- **Exponential but clever search.** If an exact solution is required, the possible solutions are searched. The search space is exponential, but it is often enough to visit a small part of it. The aim is to discard early the parts that are seen to be hopeless. This is called **pruning**. Backtracking and branch and bound follow this route.
- **Approximate solutions.** If polynomial time is required, the best solution is given up. Approximation algorithms produce a solution that is within a provable ratio of the optimum.
- **Heuristic search.** Methods are used that give no guarantee but often work well in practice. The min-conflicts heuristic is of this kind.

The subject of these notes is the first route. In the worst case backtracking takes exponential time. But with good pruning it solves most of the instances met in practice in reasonable time. The n-queens problem shows the difference in numbers: for 8 queens, instead of trying all 16.7 million complete placements one by one, it is enough to generate only 2056 partial placements.

4.2 The basic idea of backtracking

4.2.1 Trying and turning back

To solve a problem it is sometimes necessary to try different paths and to turn back from wrong ones. Someone who has forgotten the three-digit code of a padlock tries every combination in turn. Someone solving a puzzle writes a digit in a cell, runs into a contradiction a few steps later, erases the digit, and tries another. **Backtracking** turns this intuitive behavior into a systematic algorithm.

Definition 4.3

Backtracking is a search method that builds a solution step by step as a **partial solution**. At each step a **candidate** is added to the partial solution. If the added candidate violates the **constraints** of the problem, that path is abandoned. If the partial solution is complete, a solution has been found. If no candidate is left to try at some step, the last decision is undone and the next candidate of the previous step is tried.

The four terms in the definition recur throughout these notes. The **partial solution** is the choices made so far: the queens already placed in the first few rows in n-queens, the filled cells in Sudoku. A **candidate** is a choice that can be made at the next step: one of the columns of the next row, one of the digits 1–9 that can be written in an empty cell. A **constraint** is a rule that a valid solution must obey: no two queens attack each other, no digit appears twice in a row. **Undoing** means removing the most recently added candidate and restoring the previous state.

This is what separates backtracking from **brute force**. Brute force generates all complete candidate solutions and tests each one afterward. Backtracking stops as soon as a partial solution becomes invalid. No extension of an invalid partial solution is ever generated. If the first two queens attack each other, no placement of the remaining six queens can repair that mistake. Brute force tries all $8^6 = 262{,}144$ placements of these six queens one by one; backtracking tries none of them.

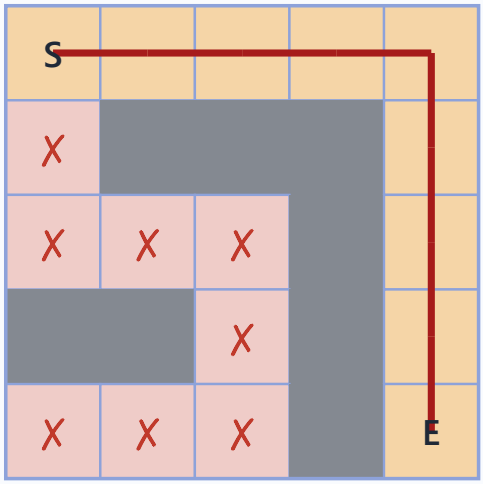


Figure 4.2: Backtracking in a maze. S is the start and E is the exit. The orange cells show the path found; the cells marked X are dead ends that were tried and abandoned.

4.2.2 The traveler in the maze

A backtracking algorithm can be compared to a traveler looking for a way through a maze. At every junction the traveler makes a decision and enters a corridor. If the corridor leads to the exit, the job is done. If it ends in a dead end, the traveler returns to the last junction and tries a corridor not yet tried. If all the corridors at that junction are dead ends, the traveler goes back to the previous junction. This trial and error continues until the exit is found or all options are exhausted.

The analogy also shows the most important requirement of the algorithm. To be able to turn back, one must remember where one came from. The traveler leaves a mark at each junction or writes the junctions entered in a list. In backtracking algorithms, recursion provides this list automatically. Each recursive call is a junction. The frames stacked on the call stack are the list of junctions that have been entered but not yet finished. When a call ends with `return`, control goes back to the previous frame. That is the traveler returning to the previous junction.

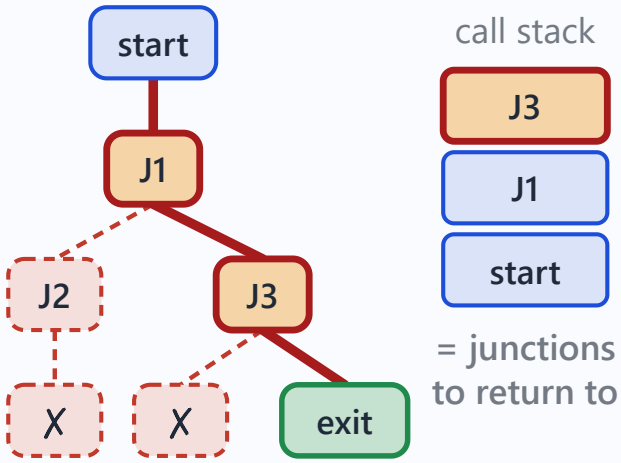


Figure 4.3: Junctions and the call stack. We have gone from the start to J1 and from there to J3. The J2 branch and the first branch of J3 are dead ends. The stack holds the start, J1, and J3: these are the junctions to which we can return.

The maze example has the structure common to artificial intelligence and puzzle problems. In Sudoku every empty cell is a junction and the digits that can be written are corridors. In chess every move is a junction. In all of them the same question is asked: is it possible to continue with this decision, and if not, which decision should we return to?

4.2.3 Backtracking in the maze, step by step

The maze is a *5 × 5* grid. Cells are written as (row, column) pairs; the start is cell (0, 0) and the exit is cell (4, 4). The gray cells are walls: (1, 1), (1, 2), (1, 3), (2, 3), (3, 0), (3, 1), (3, 3), and (4, 3). In every cell the algorithm tries the four directions in a fixed order: down, right, up, left. Every visited cell is marked, so that no cell is entered twice. Without the marks the traveler could go back and forth between two cells forever.

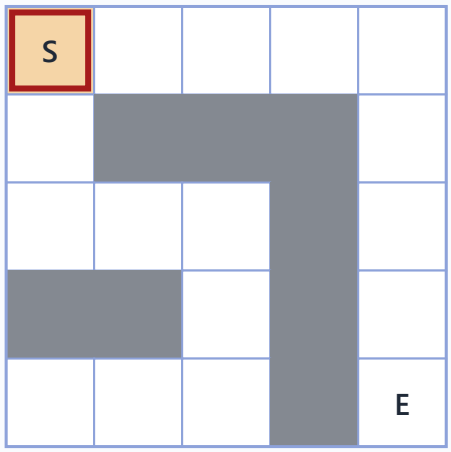


Figure 4.4: The start: we are in the start cell (S). The direction order is down, right, up, left.

The search proceeds as follows. From the start, two steps down are taken: (1, 0), (2, 0). Since (3, 0) is a wall, the down direction is closed, and the next direction, right, is tried. Two steps to the right reach (2, 1) and (2, 2). Cell (2, 2) is a junction: below it is open, while its right, (2, 3), and its top, (1, 2), are walls. Down is tried first: (3, 2), (4, 2). Below (4, 2) is outside the grid and to its right, (4, 3), is a wall, so we turn left: (4, 1), (4, 0). In cell (4, 0) all four directions are closed: below and left are the boundary, above is a wall, and to the right is a visited cell. This is a dead end.

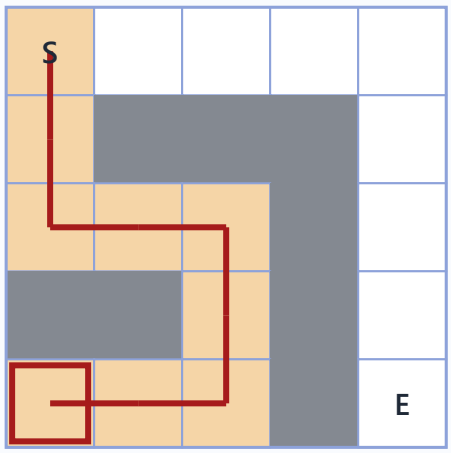


Figure 4.5: The corridor that runs down and to the left ends in a dead end at cell (4, 0).

Now the return begins. Cells (4, 0), (4, 1), (4, 2), and (3, 2) are removed from the path, because they have no untried open direction. We are back at junction (2, 2). The remaining directions of this junction are closed as well: right and up are walls, and left is visited. The return continues: (2, 2), (2, 1), (2, 0), and (1, 0) are also left behind as dead ends. Finally we are back at the start. At the start the down direction is exhausted; the next direction, right, is tried.

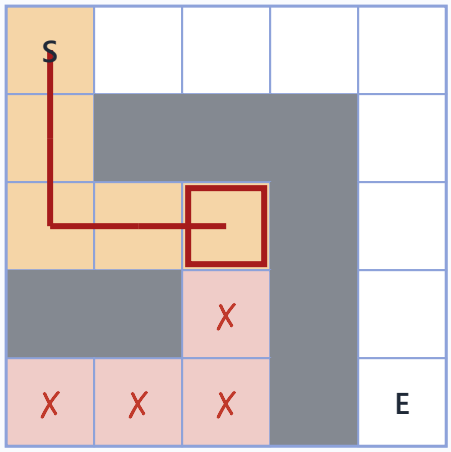


Figure 4.6: The first return: (4, 0), (4, 1), (4, 2), and (3, 2) are removed from the path (*X*). Junction (2, 2) is revisited, but its remaining directions are closed too.

We go right along the top corridor: (0, 1), (0, 2), (0, 3), (0, 4). Below (0, 1), (0, 2), and (0, 3) there are walls; below (0, 4) it is open. Through (1, 4), (2, 4), (3, 4) we reach the exit, (4, 4).

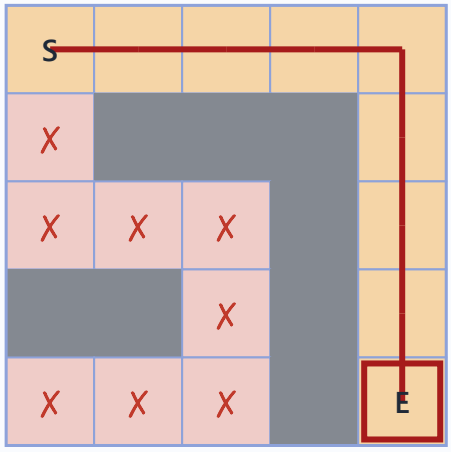


Figure 4.7: The exit has been reached. The path is 9 cells long; 8 cells were tried and abandoned as dead ends.

Example 4.1

Question. How many different simple paths lead from the start to the exit in the maze? (In a simple path a cell is used at most once.) If the direction order were "right, down, up, left," how many cells would the search mark as dead ends?

Solution. The lower-left region, that is, cells (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), and (4, 0)–(4, 2), is connected to the rest of the maze only through the start. Because (2, 3), (3, 3), (4, 3), and (1, 2) are walls, there is no way from this region to the right-hand column. The only connection to the exit is the top corridor and the right-hand column. So there is only 1 simple path: the 9-

cell orange path. With a direction order starting with right, the first direction tried at the start would be the top corridor. At every cell the search goes right first and reaches (0, 4). There the right direction is outside the grid; it goes down and follows the right-hand column to the exit. The search never enters a dead end: 0 cells are marked as dead ends. In the same maze, the direction order reduces the cost of the search from 8 dead-end cells to 0. Which option is tried first is an important decision that determines the speed of backtracking.

4.2.4 The state-space tree

It helps to think of the search done by backtracking as taking place on a tree. At the root no choice has been made yet. The children of the root are the options for the first choice. The children of each child are the options for the second choice, and so on. This tree is called the **state-space tree**.

Definition 4.4

The **state-space tree** is a rooted tree in which partial solutions are the nodes and choices are the edges. The root is the empty partial solution. The path from the root to a node is the choices made so far. The children of a node are the choices that can be made at the next level. The leaves are either complete solutions or partial solutions that cannot be extended.

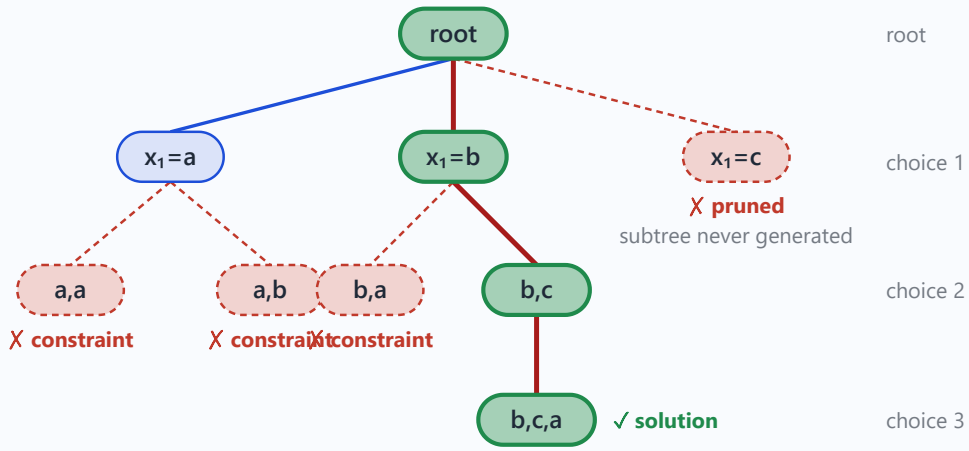


Figure 4.8: The state-space tree of a problem with three choices. The node $x_1 = c$ has been pruned; its subtree is never generated. The branches (a, a), (a, b), and (b, a), which violate the constraint, are cut as well. The thick path leads to the solution, (b, c, a).

Figure 4.8 shows the terms of the definition together. A **partial solution** is a path from the root to a node: the node $x_1 = b$, $x_2 = c$ is the partial solution (b, c). The **candidates** are the children of a node. If a **constraint** is violated, the subtree of the node is pruned: the partial solution (a, a) is invalid and no node below it is generated. The **goal test** asks whether a complete and valid solution has been reached; in this tree (b, c, a) is a leaf and a solution.

Two properties of the tree are important. First, the tree is not built in advance. The algorithm keeps in memory only the node it is currently at and the path to the root. The tree is generated as it is traversed, and the branches already visited are forgotten. Second, the traversal order is **depth-first** (DFS). The algorithm goes down a branch to the bottom. When the branch ends or is pruned, it climbs to the parent and moves on to the next child. Backtracking is a depth-first search of the state-space tree. The only difference is that the subtrees of invalid nodes are never generated.

> **Common mistake**
>
> Taking the state-space tree to be a data structure built in memory. The tree is only a model for thinking about the search. At any moment memory holds only the single path from the root to the current node. In n-queens this path has at most n nodes, whereas the tree itself may contain thousands of nodes.

4.2.5 The general template: choose, explore, unchoose

Almost all backtracking algorithms are written with the same skeleton. The skeleton is a recursive method. The method takes a partial solution, checks whether it is a complete solution, and if not, tries each candidate in turn.

```
void backtrack(State partial) {
  if (isSolution(partial)) {
    save(partial);
    return;
  }
  for (Candidate c : candidates(partial)) {
    if (isValid(partial, c)) {
      choose(partial, c);     // choose
      backtrack(partial);     // explore
      unchoose(partial, c);   // unchoose
    }
  }
}
```

The code has four parts.

1. **Goal test.** `isSolution(partial)` checks whether the partial solution is complete. If it is, the solution is saved and the method returns. If a single solution is enough, the search can be stopped completely here.
2. **Choose.** `candidates(partial)` gives the options for the next step. A valid candidate is added to the partial solution with `choose(partial, c)`.
3. **Explore.** `backtrack(partial)` descends one level by recursion. This call traverses the entire subtree below the added candidate.

4. **Unchoose.** When the call returns, `unchoose(partial, c)` removes the candidate from the partial solution. The state returns to what it was before the call, and the loop continues with the next candidate.

Pruning happens inside `isValid`. For an invalid candidate the recursive call is never made. In this way the whole subtree below that candidate is skipped with a single `if`. The power of pruning comes from here: one check eliminates a subtree of possibly thousands of nodes.

To see why the undo step is essential, look at how the partial solution is stored. `partial` is a single object shared by all calls: a board, an array, or a list. Every call modifies this object. When a call returns, the object stays in the state the call left it in. If this change is not erased before the next candidate is tried, the new candidate is tested together with the leftovers of the old one. Primitive parameters passed by value, such as an `int` sum or an index, return to their old values on their own, because each call has its own copy.

If a single solution is enough, the template returns a `boolean` with a small change. The goal test returns `true`. If the recursive call returns `true`, the caller also returns `true` at once. So when the first solution is found, all the calls end in turn. The n-queens, Sudoku, graph coloring, and Hamiltonian path code is of this form.

4.2.6 Applying the template: permutations

The cleanest application of the template is generating all permutations of a set. Every ordering of the numbers 1, 2, and 3 is a permutation. The partial solution is the numbers written in the first k positions. A candidate is a number not yet used. There is a single constraint: a number cannot be used twice.

```
void permute(int[] p, boolean[] used, int k) {
  if (k == p.length) {               // solution
    System.out.println(Arrays.toString(p));
    return;
  }
  for (int a = 1; a <= p.length; a++) {
    if (!used[a]) {                  // valid
      p[k] = a; used[a] = true;     // choose
      permute(p, used, k + 1);       // explore
      used[a] = false;               // unchoose
    }
  }
}
```

The array `p` holds the partial solution; `p[0..k-1]` is filled. The array `used` records which numbers have been used. Its length is *n + 1*, because the numbers start at 1 and `used[0]` stays empty. `k == p.length` is the goal test: if all positions are filled, the permutation is printed. The loop tries every

number from 1 to *n*. `!used[a]` is the validity test. If the number is unused, it is written to `p[k]` and marked (choose). The recursive call fills the next position (explore). When the call returns, the mark is removed (unchoose).

The unchoose line contains only `used[a] = false`; `p[k]` is not reset. The reason is that `p[k]` will be overwritten by the next candidate anyway. The validity test looks at `used`, not at `p`. So the old value in `p[k]` influences no decision. The structure that checks the constraint, however, must always be undone.

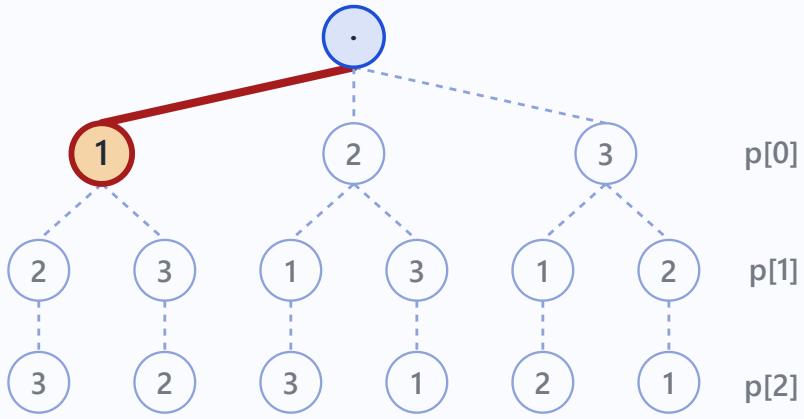


Figure 4.9: The first step: at `k = 0` the candidate 1 has been chosen. The dashed nodes have not been generated yet.

Step	k	Action	`p`	Used	Output
1	0	a = 1 is chosen	[1, –, –]	{1}	
2	1	a = 1 is used; a = 2 is chosen	[1, 2, –]	{1, 2}	
3	2	only 3 is unused; it is chosen	[1, 2, 3]	{1, 2, 3}	
4	3	`k == p.length`: solution	[1, 2, 3]	{1, 2, 3}	[1, 2, 3]
5	1	3 and 2 are undone; a = 3 is chosen	[1, 3, 3]	{1, 3}	
6	2	only 2 is unused; it is chosen	[1, 3, 2]	{1, 2, 3}	
7	3	solution	[1, 3, 2]	{1, 2, 3}	[1, 3, 2]
8	0	2, 3, 1 are undone; a = 2 is chosen	[2, 3, 2]	{2}	
9	1–3	a = 1, then 3	[2, 1, 3]	{1, 2, 3}	[2, 1, 3]
10	1–3	undo; a = 3, then 1	[2, 3, 1]	{1, 2, 3}	[2, 3, 1]
11	0–3	the branch starting with 3 proceeds the same way	[3, 1, 2], [3, 2, 1]	—	[3, 1, 2], [3, 2, 1]

Table 4.2: Trace of the call `permute(p, used, 0)` (*n* = 3). The “Used” column shows the numbers that are `true` after the step.

The value of `p` in step 5 deserves attention: [1, 3, 3]. `p[2]` still holds the 3 left over from the previous permutation. This value breaks nothing, because we are at `k = 1` and `p[2]` is not yet part of the partial solution. In step 6 a 2 is written over it. The output of the program is:

```
[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]
```

Because the candidates are tried from smallest to largest, the permutations come out in **lexicographic order**. This observation holds for all backtracking algorithms: solutions are found in the same order in which the candidates are tried.

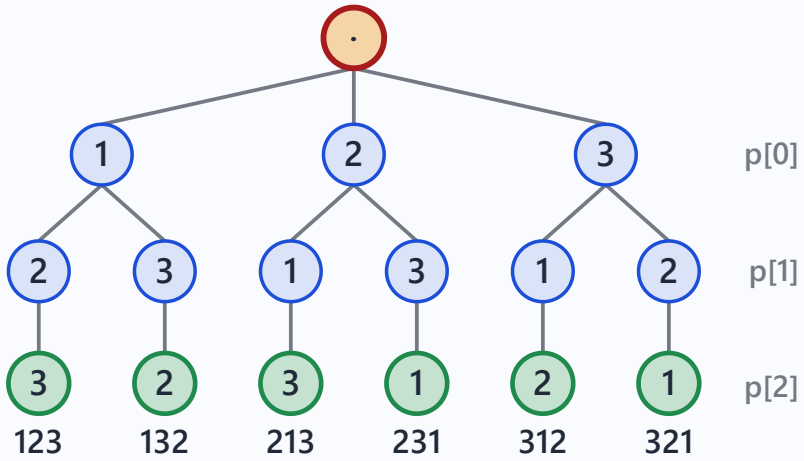


Figure 4.10: The complete state-space tree when the search ends. The six leaves are the six permutations; under each leaf is written the reading of the path from the root to that leaf.

Example 4.2

Question. For $n = 3$, how many nodes does the state-space tree have, counting the root? How many times is "choose" done, and how many times "unchoose"? How are the same questions answered for $n = 4$?

Solution. It is enough to count the levels separately. At the root there is 1 node. At the first level there are 3 options, at the second level 2 options remain under each of them, and at the third level 1 option. The number of nodes is $1 + 3 + 3 \cdot 2 + 3 \cdot 2 \cdot 1 = 1 + 3 + 6 + 6 = 16$. Every node other than the root is a "choose" operation, and every choose has an "unchoose": 15 chooses and 15 unchooses. The number of leaves is $3! = 6$. For $n = 4$ the nodes number $1 + 4 + 12 + 24 + 24 = 65$; there are 64 chooses, 64 unchooses, and 24 leaves.

In general, the permutation tree of n items has $n \cdot (n - 1) \cdot \ldots \cdot (n - k + 1) = n!/(n - k)!$ nodes at level k. The last two levels have equal numbers of nodes ($n!$), and the total is a small constant multiple of $n!$ (close to $e \cdot n!$). Printing a permutation at each leaf takes $O(n)$. So generating all permutations takes

$O(n \cdot n!)$ time. Here pruning does not reduce the work, because all the leaves are solutions anyway. What pruning gains is that the branches that violate the constraint are never opened: in an unpruned tree that allowed the same number to be chosen again, there would be $4^4 = 256$ leaves for $n = 4$.

> **Common mistake**
>
> Adding the partial solution itself to the list when collecting all permutations. If `results.add(path)` were written in the code below, the reference to the *same* object would be added to the list six times. Because `path` will be empty when the search ends, the list would consist of six empty lists. The right thing is to store a copy at the moment of the solution.

```
List<List<Integer>> results = new ArrayList<>();

void collect(List<Integer> path, boolean[] used, int n) {
  if (path.size() == n) {
    results.add(new ArrayList<>(path));   // store a copy
    return;
  }
  for (int v = 1; v <= n; v++) {
    if (!used[v]) {
      used[v] = true; path.add(v);          // choose
      collect(path, used, n);                // explore
      path.remove(path.size() - 1);          // unchoose
      used[v] = false;
    }
  }
}
```

This version keeps the partial solution in a `List`. The choose step appends to the list, and the unchoose step deletes the last element. `new ArrayList<>(path)` is an independent copy of the current contents; later changes do not affect the copy. After the call `collect(new ArrayList<>(), new boolean[4], 3)`, `results` holds the six permutations in lexicographic order.

4.2.7 Brute force and pruning

How much work pruning saves can be shown numerically with the 8-queens problem. The problem is this: place 8 queens on an 8×8 chessboard so that no two attack each other. There are several ways to solve it, and each examines a different number of candidates.

Approach	Candidates examined	Explanation
Choose 8 of the 64 squares	4,426,165,368	$C(64, 8)$: generate every placement, then test it
One queen per row	16,777,216	8^8 complete placements

Approach	Candidates examined	Explanation
One queen per row, distinct columns	40,320	*8!* permutations
Backtracking, all solutions	2056	number of safe placements (92 solutions)
Backtracking, first solution	113	safe placements; 105 undone

Table 4.3: Number of candidates examined by different approaches to the 8-queens problem.

The first three rows of the table are increasingly clever versions of brute force. The first row is the purest approach: there are $C(64, 8) = 4{,}426{,}165{,}368$ ways to choose 8 of the 64 squares. The second row uses an observation: there must be exactly one queen in each row, because two queens in the same row attack each other. The queen in each row has 8 column choices, which leaves $8^8 = 16{,}777{,}216$ placements. The third row applies the same logic to columns: the columns must be distinct as well. A placement is then a permutation of the columns, and $8! = 40{,}320$ candidates remain. What these three approaches have in common is that every candidate is a *complete* placement that is tested only at the end.

Backtracking, on the other hand, generates not complete placements but *partial* ones. The queens are placed row by row. If a queen conflicts with one of the earlier ones, no extension of that partial placement is ever tried. The result is striking: to find all 92 solutions only 2056 safe placements are made. The first solution is found in 113 placements. Compared with the 8^8 complete placements, even finding all the solutions takes about $16{,}777{,}216 / 2056 \approx 8160$ times less work.

Pruning does not change the worst case; backtracking is still an exponential algorithm. In practice, though, only a very small part of the state-space tree is visited. The power of pruning depends on how early the constraint is checked. The constraint must be checked on the partial solution, at every new candidate. A check left to the end is no different from brute force.

4.2.8 Strengths and weaknesses

The strengths of backtracking come from its generality.

- **Wide range of application.** It applies to any problem whose solution can be expressed as a sequence of choices. The same template is used unchanged from permutation generation to Sudoku.
- **Often efficient.** For problems whose constraints can be checked early, most of the tree is pruned.
- **Short, readable code.** The algorithm is a recursive method a few lines long.
- **Exhaustive search.** Backtracking finds all solutions or shows with certainty that none exists. If the search traverses the whole tree and finds nothing, there is no solution. This property is called **completeness**.

The weaknesses are just as clear.

- **Exponential worst case.** The time complexity is bounded by exponential functions such as 2^n, $n!$, or 9^m.

- **Slow on large search spaces.** When the number of possible solutions is very large, good pruning and heuristics are needed.
- **Stack memory.** Deep recursion uses one frame per level: *O(depth)* memory. In very deep searches the Java stack can overflow.
- **Repeated work.** The same subproblem may be solved again and again along different paths. Dynamic programming stores these results in a table; backtracking does not.

4.3 The n-queens problem

4.3.1 The problem

In chess the queen is the most powerful piece. It can move any number of squares along its row, its column, and both diagonals, and it attacks every piece on these four lines. The **eight queens problem** is an old puzzle posed in a chess magazine in 1848: can 8 queens be placed on an 8×8 board so that no queen attacks another?

Definition 4.5

The **n-queens problem**: n queens are placed on an $n \times n$ board. No two queens may be in the same row, the same column, or the same diagonal. A solution is such a placement.

Some small cases of the general problem are interesting. For $n = 1$, a single queen is placed on the single square; this is a solution. For $n = 2$ and $n = 3$ there is no solution. On the 2×2 board every pair of squares lies in the same row, the same column, or the same diagonal. For $n \geq 4$ there is always at least one solution. The 8-queens problem has 92 solutions. If solutions that can be turned into one another by rotating and reflecting the board are counted as the same, 12 **fundamental solutions** remain.

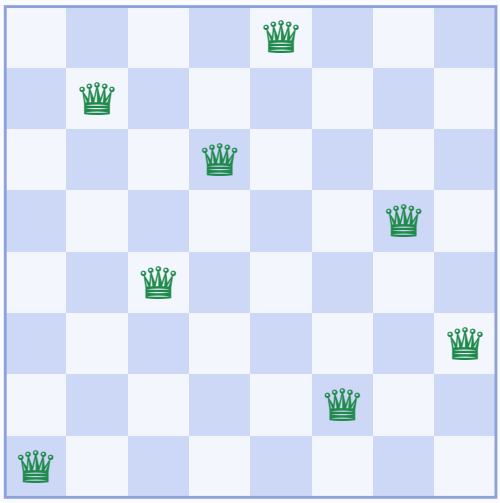


Figure 4.11: A solution to the 8-queens problem: a1, b7, c4, d6, e8, f2, g5, h3. In chess notation the letter gives the file (column) and the number the rank (row); rank 1 is at the bottom of the board.

The solution in Figure 4.11 is given in chess notation. a1 is rank 1 of file a, that is, the bottom-left corner. The correctness of the solution can be checked: the eight letters are distinct (distinct columns), and the eight numbers are distinct (distinct rows). For the diagonal check, the absolute values of the row difference and the column difference are compared for each pair; they are equal for no pair. This check is a small example of why the problem is in NP: finding a solution requires search, while checking one ends with comparing $C(8, 2) = 28$ pairs.

4.3.2 Placing queens row by row

The backtracking solution proceeds in the following steps.

1. Every cell of the board is initially empty.
2. The first queen is placed in some column of the first row.
3. In the next row each column is tried in turn. When a queen is put in that column, we check whether it conflicts with the earlier queens: is there a queen in the same column or on either of the two diagonals?
4. If there is no conflict, the position is accepted as valid and we move on to the next row.
5. If all the columns of a row have been tried and no safe place is found, the queen in the previous row is removed and the next column is tried in that row. This is the backtrack.
6. When all the queens have been placed, a solution has been found.

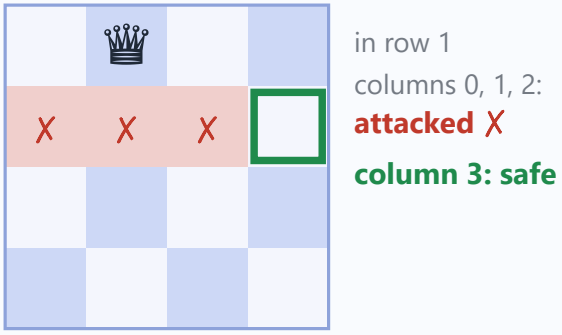


Figure 4.12: Row 1 on a 4 × 4 board with a queen in column 1 of row 0: columns 0, 1, and 2 are under attack, and column 3 is safe.

Placing the queens row by row is a deliberate design decision. There must be exactly one queen in every row. This observation builds the row constraint into the structure. Queen 0 goes to row 0 and queen 1 to row 1; two queens in the same row become impossible. Each placement therefore checks only the column and the two diagonals. The search space also falls from 64^8 (64 squares for each queen) to 8^8. Guaranteeing constraints through the structure instead of checking them is a technique often used in backtracking.

4.3.3 Four queens: the board and the state-space tree

The way the algorithm works is best seen on a small example. For $n = 4$ the queens are placed row by row; in each row columns 0, 1, 2, 3 are tried in that order. In the state-space tree each level below the root is a row; a node shows the column chosen in that row. The red cells on the board are the squares attacked by the queens already placed.

1. Row 0: column 0 is safe; the queen is placed and we descend to row 1.
2. Row 1: columns 0 and 1 are under attack; column 2 is safe. The queen is placed and we descend to row 2.
3. Row 2: all four columns are under attack. This is a dead end. The queen in row 1 (column 2) is removed; the next column is tried in row 1.

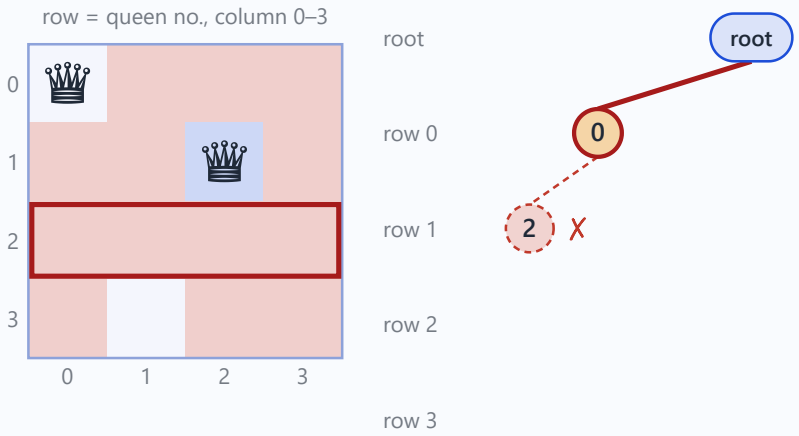


Figure 4.13: The first dead end. With queens at (0, 0) and (1, 2), every square of row 2 is under attack. In the tree the node for 2 is drawn dashed: this branch will be undone.

4. Row 1: column 3 is safe; the queen is placed and we descend to row 2.
5. Row 2: column 0 is under attack, column 1 is safe. The queen is placed and we descend to row 3.
6. Row 3: all four columns are under attack; dead end. The queen in row 2 (column 1) is removed.
7. Row 2: the remaining columns, 2 and 3, are also under attack; dead end. The queen in row 1 (column 3) is removed.
8. Row 1 has no columns left to try. The queen in row 0 (column 0) is removed as well.

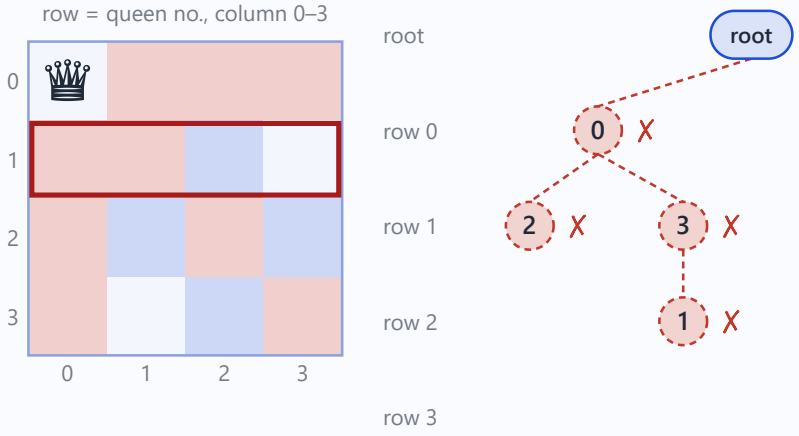


Figure 4.14: All the branches starting with column 0 have ended in dead ends. The return reaches all the way to the root; the next column will be tried in row 0.

9. Row 0: column 1 is safe; the queen is placed.
10. Row 1: columns 0, 1, 2 are under attack, column 3 is safe.
11. Row 2: column 0 is safe.

12. Row 3: columns 0 and 1 are under attack, column 2 is safe. The fourth queen is placed and `row == n` holds. A solution has been found: the columns are [1, 3, 0, 2].

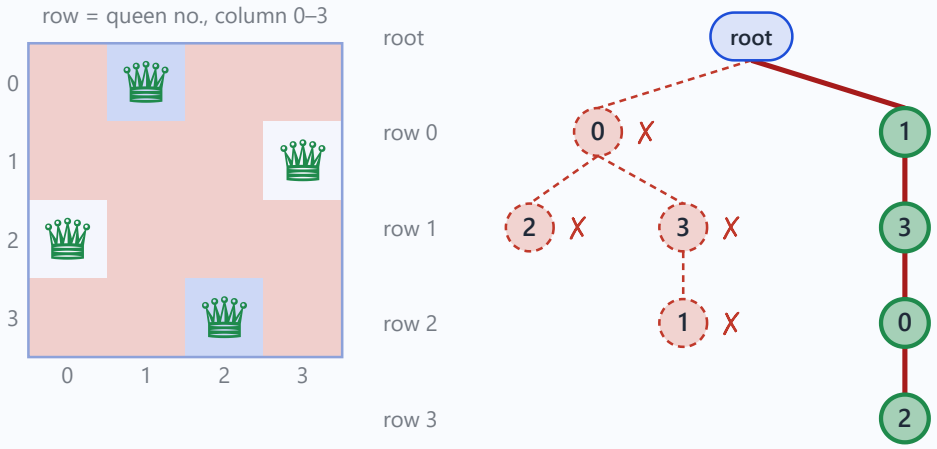


Figure 4.15: The solution [1, 3, 0, 2]. In the tree the green path is the solution and the dashed nodes are the branches that were backed out of. In all, 8 placements and 4 backtracks were made.

Counting the nodes in the tree gives the breakdown of the work. Apart from the root there are 8 nodes: 4 nodes on the solution path and 4 nodes that were undone (0, 2, 3, and 1). Each node is a placement; each undone node is a backtrack. Because no queen is ever put on an attacked square, those squares do not even appear as nodes in the tree. That is exactly the gain from pruning.

Note the representation of the solution. The *i*-th element of the array [1, 3, 0, 2] is the column of the queen in row *i*. The queen in row 0 is in column 1, the one in row 1 in column 3, the one in row 2 in column 0, and the one in row 3 in column 2. This representation is the natural result of building the row constraint into the structure: a single number per row is enough.

Example 4.3

Question. How many solutions does the 4-queens problem have? How is the second solution obtained from the first?

Solution. If the search is not stopped at the first solution, columns 2 and 3 are also tried in row 0. The column 2 branch gives the second solution: [2, 0, 3, 1]. The column 3 branch ends in a dead end. The 4-queens problem has only 2 solutions. The second solution is the mirror image of the first: if each column *c* is replaced by *3 − c*, the array [1, 3, 0, 2] becomes [2, 0, 3, 1]. When the board is reflected about the vertical axis, the attack relations are preserved. So the reflection of a solution is also a solution.

4.3.4 Java code

The algorithm is written with two methods. `isSafe` checks whether a queen put on square *(r, c)* conflicts with the queens placed earlier. `solve` is the backtracking method that proceeds row by row. The board is the array `int[][] b`; `b[i][j] == 1` means there is a queen on square *(i, j)*. `n` is a field of the class.

```
boolean isSafe(int[][] b,
               int r, int c) {
  // same column
  for (int i = 0; i < r; i++)
    if (b[i][c] == 1) return false;
  // upper-left diagonal
  for (int i = r-1, j = c-1;
       i >= 0 && j >= 0; i--, j--)
    if (b[i][j] == 1) return false;
  // upper-right diagonal
  for (int i = r-1, j = c+1;
       i >= 0 && j < n; i--, j++)
    if (b[i][j] == 1) return false;
  return true;
}
```

`isSafe` looks in only three directions: up, the upper-left diagonal, and the upper-right diagonal. There is no need to look at the rows below, because the queens are placed row by row, and when a queen is put in row *r* the lower rows are empty. The same row is not examined either, because each row has at most one queen. The first loop scans the *r* cells of the column. The second loop starts at *(r − 1, c − 1)* and moves toward the upper-left corner, decreasing the row and the column together. The third loop decreases the row while increasing the column. The loop conditions keep the scan from leaving the board: `j >= 0` guards the left edge and `j < n` the right edge. If any loop finds a queen, the method returns `false` at once.

```
boolean solve(int[][] b, int row) {
  if (row == n)
    return true;
  for (int col = 0; col < n;
       col++) {
    if (isSafe(b, row, col)) {
      b[row][col] = 1; // place
      if (solve(b, row + 1))
        return true;
      b[row][col] = 0; // undo
    }
  }
  return false;
}
```

`solve` is a direct application of the template. `row == n` is the goal test: if n queens have been placed, a solution has been found. The loop tries every column of the row. `isSafe` is the validity test and does the pruning. A queen is placed on a safe square (choose), and `solve` is called for the next row (explore). If the call returns `true`, a solution has been found. This `true` is passed up at once; the board stays as it is and contains the solution. If the call returns `false`, there is no way to continue with this column; the queen is removed (unchoose) and the loop continues with the next column. If the loop ends, no column worked in this row and `false` is returned. This `false` forces the `solve` one level up to remove its own queen.

The cost has two parts. `isSafe` runs at most three loops, each of at most n steps, so one call takes $O(n)$. For `solve` the worst case is traversing the whole state-space tree. The first row has n safe columns and the second at most $n - 1$, since at least one column is taken. With this bound the number of nodes is $O(n!)$. At each node, `isSafe` at $O(n)$ is called for each of the n columns, giving a crude upper bound of $O(n^2 \cdot n!)$. The real number of nodes is much smaller than $n!$. Memory use is $O(n^2)$ for the board and $O(n)$ for the call stack.

Common mistake

Not propagating the result of the recursive call. If only `solve(b, row + 1);` is written instead of `if (solve(b, row + 1)) return true;`, the loop continues even after a solution has been found. On the next line of code `b[row][col] = 0` runs, and the solution is erased from the board. The search traverses the whole tree and finally returns `false`; the board is left empty.

4.3.5 Code trace: solve(b, 0), n = 4

Tracing the code line by line reveals a detail that does not show up in the tree trace: every safety test is a call to `isSafe`, and the method returns at the *first* attack it finds. The method checks the column first, then the upper-left diagonal, and the upper-right diagonal last. If a square is attacked by more than one queen, only the attack found by the first check is visible. For example, square (2, 2) is both in the column of the queen at (1, 2) and on the diagonal of the queen at (0, 0). Because the method scans the column first, it finds the attack at (1, 2).

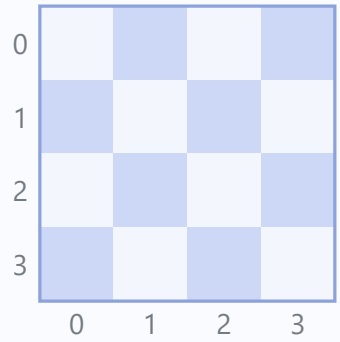


Figure 4.16: `solve(b, 0)` has been called; the board is empty, `row = 0`.

#	Square	`isSafe`	Reason / action	Queens	Stack depth
1	(0, 0)	true	`b[0][0] = 1`, `solve(b, 1)`	[0, –, –, –]	2
2	(1, 0)	false	same column: (0, 0)	[0, –, –, –]	2
3	(1, 1)	false	upper-left diagonal: (0, 0)	[0, –, –, –]	2
4	(1, 2)	true	`b[1][2] = 1`, `solve(b, 2)`	[0, 2, –, –]	3
5	(2, 0)	false	same column: (0, 0)	[0, 2, –, –]	3
6	(2, 1)	false	upper-right diagonal: (1, 2)	[0, 2, –, –]	3
7	(2, 2)	false	same column: (1, 2)	[0, 2, –, –]	3
8	(2, 3)	false	upper-left diagonal: (1, 2)	[0, 2, –, –]	3
—	—	—	the loop in row 2 ends, `return false`; undo: `b[1][2] = 0`	[0, –, –, –]	2
9	(1, 3)	true	`b[1][3] = 1`, `solve(b, 2)`	[0, 3, –, –]	3
10	(2, 0)	false	same column: (0, 0)	[0, 3, –, –]	3
11	(2, 1)	true	`b[2][1] = 1`, `solve(b, 3)`	[0, 3, 1, –]	4
12	(3, 0)	false	same column: (0, 0)	[0, 3, 1, –]	4
13	(3, 1)	false	same column: (2, 1)	[0, 3, 1, –]	4
14	(3, 2)	false	upper-left diagonal: (2, 1)	[0, 3, 1, –]	4

#	Square	`isSafe`	Reason / action	Queens	Stack depth
15	(3, 3)	false	same column: (1, 3)	[0, 3, 1, –]	4
—	—	—	row 3 is a dead end; undo: `b[2][1] = 0`	[0, 3, –, –]	3
16	(2, 2)	false	upper-left diagonal: (0, 0)	[0, 3, –, –]	3
17	(2, 3)	false	same column: (1, 3)	[0, 3, –, –]	3
—	—	—	row 2 is a dead end; undo: `b[1][3] = 0`; column 4 = n, so the loop in row 1 ends too	[0, –, –, –]	2
—	—	—	row 1 is a dead end; undo: `b[0][0] = 0`	[–, –, –, –]	1
18	(0, 1)	true	`b[0][1] = 1`, `solve(b, 1)`	[1, –, –, –]	2
19	(1, 0)	false	upper-right diagonal: (0, 1)	[1, –, –, –]	2
20	(1, 1)	false	same column: (0, 1)	[1, –, –, –]	2
21	(1, 2)	false	upper-left diagonal: (0, 1)	[1, –, –, –]	2
22	(1, 3)	true	`b[1][3] = 1`, `solve(b, 2)`	[1, 3, –, –]	3
23	(2, 0)	true	`b[2][0] = 1`, `solve(b, 3)`	[1, 3, 0, –]	4
24	(3, 0)	false	same column: (2, 0)	[1, 3, 0, –]	4
25	(3, 1)	false	same column: (0, 1)	[1, 3, 0, –]	4
26	(3, 2)	true	`b[3][2] = 1`, `solve(b, 4)` → `row == n`, `true`	[1, 3, 0, 2]	5

Table 4.4: Trace of the call `solve(b, 0)` (*n = 4*). The "Queens" column gives the placement after the step, in row order; – is an empty row.

The table shows the whole behavior of the code. `isSafe` is called 26 times. Of these, 8 return `true`; these 8 calls are placements. There are 4 undo steps. The stack depth is the number of `solve` frames open at that moment. When the depth reaches 5, the frame `solve(b, 4)` is open and the goal test returns `true`. This value is carried up through four frames. No frame removes its own queen; the board contains the solution.

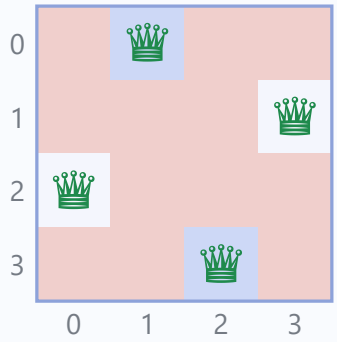


Figure 4.17: When `row == n`, the value `true` is carried up the stack; the solution [1, 3, 0, 2] stays on the board.

Example 4.4

Question. How does `solve(b, 0)` find the first solution for $n = 5$? How many placements and how many backtracks are made?

Solution. In row 0, column 0 is safe. In row 1, column 0 is in the same column and column 1 is on a diagonal; column 2 is safe. In row 2, columns 0, 1, 2, and 3 are under attack; column 4 is safe. In row 3, column 0 is under attack and column 1 is safe. In row 4, columns 0, 1, and 2 are under attack and column 3 is safe. The solution is [0, 2, 4, 1, 3]. 5 placements are made and there is no backtracking: in every row the first safe column is the right choice. With 4 queens, 8 placements and 4 backtracks were needed. A larger problem can need less work; the cost of finding the first solution does not grow smoothly with n.

4.3.6 Eight queens: the first solution found

When `solve(b, 0)` is run for $n = 8$, the columns are tried from smallest to largest, so the first solution in lexicographic order is always found. This solution is [0, 4, 7, 5, 2, 6, 1, 3]: the queen in row 0 is in column 0, the one in row 1 in column 4, and so on.

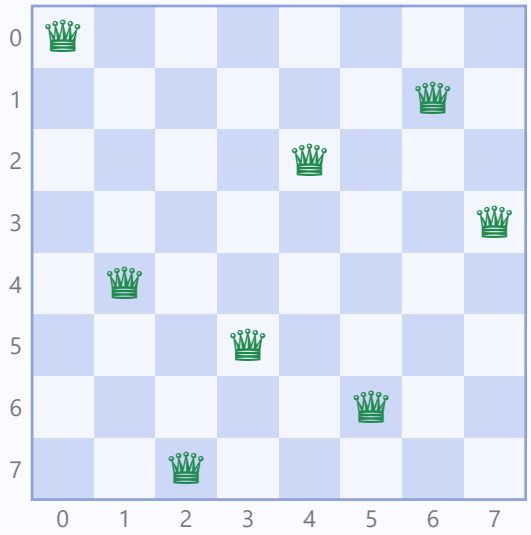


Figure 4.18: The first solution the algorithm finds for 8 queens. The columns for rows 0–7 are [0, 4, 7, 5, 2, 6, 1, 3].

Up to this solution, `isSafe` is called 876 times, 113 queens are placed, and there are 105 backtracks. The relation between the numbers is striking: *113 − 105 = 8*. Every placement is either undone or stays in the solution; since 8 queens stay in the solution, the remaining 105 placements were undone. This equality holds for every n-queens search that stops at the first solution: *placements − undos = n*.

The placement a1, b7, c4, d6, e8, f2, g5, h3 in Figure 4.11 is also a solution, but it is not the first one the algorithm finds. All 92 solutions are valid; the algorithm stops only at the first one, which is determined by the candidate order. To find all the solutions, the solution is recorded in the goal test and `false` is returned. `false` means that the search continues: the call above removes its queen and tries the next column.

4.3.7 Solution and placement counts

To see how the algorithm behaves as n grows, two numbers are measured: the number of safe placements made to find all the solutions, and the number of placements made up to the first solution. A “placement” is a queen found safe and put on the board; it equals the number of non-root nodes in the state-space tree.

n	Solutions	n^n (brute force)	Placements (all solutions)	Up to the first solution
1	1	1	1	1
2	0	4	2	—
3	0	27	5	—
4	2	256	16	8
5	10	3,125	53	5
6	4	46,656	152	31

n	Solutions	n^n (brute force)	Placements (all solutions)	Up to the first solution
7	40	823,543	551	9
8	92	16,777,216	2056	113
9	352	387,420,489	8393	41
10	724	10,000,000,000	35,538	102

Table 4.5: For n-queens, the number of solutions, the number of complete placements of brute force, and the placements made by backtracking.

Three conclusions can be drawn from the table.

- For $n = 2$ and $n = 3$ there is no solution. The algorithm traverses the whole tree and returns `false`. For $n = 3$ the tree has only 5 non-root nodes; all 5 placements are undone. This is the proof of the answer "no solution": the search is complete and has skipped no possibility.
- The cost of finding all the solutions is far smaller than n^n, but it still grows quickly. From $n = 9$ to 10 the number of placements grows by a factor of about 4.2. The growth is roughly like $n!$: about one percent of $10! = 3{,}628{,}800$.
- The work up to the first solution does not grow smoothly with n. For $n = 6$, 31 placements are needed, while for $n = 7$ only 9 suffice. The lexicographically first solution for $n = 6$, [1, 3, 5, 0, 2, 4], starts with column 1; all the branches starting with column 0 are explored in vain. The first solution for $n = 7$, [0, 2, 4, 6, 1, 3, 5], needs almost no backtracking: only 2 undos are made.

4.3.8 An O(1) safety test

At every call `isSafe` scans the column and the two diagonals and takes $O(n)$. Considering that for 8 queens 876 calls are made up to the first solution, this cost is not negligible. If instead a flag is kept for each column and each diagonal, the test drops to $O(1)$. For this we need to know how to number the diagonals.

Denote a square by (r, c). On an **anti-diagonal**, which runs from the upper right down to the lower left, moving from one square to the next increases the row by one and decreases the column by one. So the sum $r + c$ stays constant. On a **diagonal** running from the upper left down to the lower right, the row and the column increase together; the difference $r - c$ is constant. On an $n \times n$ board $r + c$ takes values from 0 to $2n - 2$; there are $2n - 1$ anti-diagonals. $r - c$ lies between $-(n - 1)$ and $n - 1$. If $n - 1$ is added to this value, the index starts at 0 and again takes $2n - 1$ values.

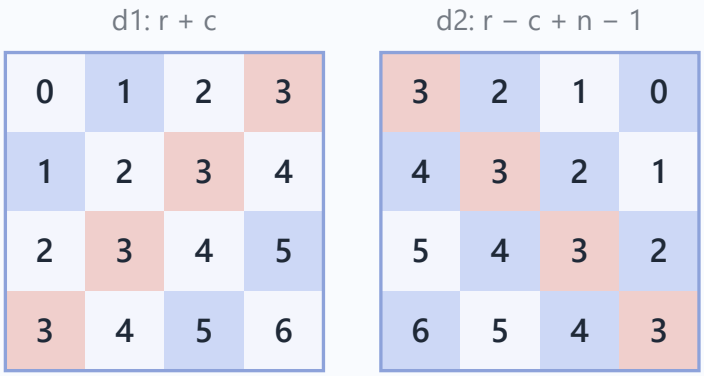


Figure 4.19: The indices `d1` ($r + c$) and `d2` ($r - c + n - 1$) of each square on a 4 × 4 board. The highlighted squares share an index: those on the left form anti-diagonal number 3, those on the right form diagonal number 3.

```
boolean[] cols = new boolean[n];
boolean[] d1 = new boolean[2*n-1]; // r+c
boolean[] d2 = new boolean[2*n-1]; // r-c+n-1

void solve(int r) {
  if (r == n) { count++; return; }
  for (int c = 0; c < n; c++) {
    int a = r + c, b = r - c + n - 1;
    if (cols[c] || d1[a] || d2[b])
      continue;
    cols[c] = d1[a] = d2[b] = true;
    solve(r + 1);
    cols[c] = d1[a] = d2[b] = false;
  }
}
```

The code has three `boolean` arrays. `cols[c]` says whether column c has a queen; `d1[a]` says whether the anti-diagonal numbered $a = r + c$ has one; `d2[b]` says whether the diagonal numbered $b = r - c + n - 1$ has one. Whether a square is safe is learned with three array accesses. If any one of the flags is `true`, the square is under attack and `continue` moves to the next column. Otherwise all three flags are set to `true` (choose), the next row is solved (explore), and then all three flags are set to `false` (unchoose).

This version differs in two ways. First, there is no need for a board array. The flags alone check for conflicts. If the solution itself is needed, it is enough to keep an array `place[row] = col`. Second, the method is `void` and does not stop at the first solution. In the goal test `count` is incremented and the method returns; the search goes on. So the code counts all the solutions. For $n = 8$, `count` becomes 92 and 2056 placements are made; these numbers are the same as for the board-based version, because the two versions traverse the same tree in the same order. Only the work at each node changes.

Example 4.5

Question. In the flag version for $n = 8$, there is a queen only at square (3, 1). Which flags are `true`? Are the squares (5, 3), (1, 3), and (4, 4) safe?

Solution. `cols[1]`, `d1[3 + 1] = d1[4]`, and `d2[3 - 1 + 7] = d2[9]` are `true`. For (5, 3), $a = 8$ and $b = 5 - 3 + 7 = 9$; `d2[9]` is occupied. The square is not safe: (3, 1), (4, 2), (5, 3) are on the same diagonal. For (1, 3), $a = 4$; `d1[4]` is occupied. (3, 1), (2, 2), (1, 3) are on the same anti-diagonal; the square is not safe. For (4, 4), `cols[4]`, `d1[8]`, and `d2[7]` are empty; the square is safe.

Common mistake

Forgetting to add $n - 1$ to the diagonal index. If `d2[r - c]` is written, the index for square (0, 3) is -3 and an `ArrayIndexOutOfBoundsException` is thrown. Taking the length of the array to be n gives the same error: the number of diagonals is $2n - 1$, not n.

4.3.9 Counting all solutions

The board-based `solve` can also be changed to count all the solutions. The change is only in the goal test.

```
int count = 0;

boolean solve(int[][] b, int row) {
  if (row == n) {
    count++;          // count the solution
    return false;     // keep searching
  }
  for (int col = 0; col < n; col++) {
    if (isSafe(b, row, col)) {
      b[row][col] = 1;
      solve(b, row + 1);
      b[row][col] = 0;   // undo
    }
  }
  return false;
}
```

In the goal test `count` is incremented and `false` is returned. The value `false` tells the calling frame, “this path is finished; keep searching.” The caller removes its queen and tries the next column. Because the method never returns `true`, there is no need to look at the result of the recursive call. For $n = 4, 5, 6, 7, 8$ the code counts 2, 10, 4, 40, and 92 solutions respectively.

Returning `true` in the goal test would be wrong. All the calls would then end at the first solution and `count` would stay at 1. Deleting the undo line would be wrong as well. The board would fill up with the queens of earlier attempts, and later attempts would be pruned incorrectly.

When all solutions are searched for, the symmetry of the board can be exploited. Reflecting a solution about the vertical axis gives another solution. If the queen in the first row is in column c, its mirror image is in column $n - 1 - c$. So the first queen is tried only in the left half, and every solution found is counted twice. If n is odd, the mirror image of a solution that starts in the middle column also starts in the middle column; those are counted once. For $n = 8$ this method reaches the same result of 92 with 1028 placements instead of 2056: the work is cut exactly in half. For $n = 9$ the placements fall from 8393 to 4703; because the middle column is counted on its own, the gain is somewhat smaller.

4.4 The min-conflicts heuristic

4.4.1 From complete placement to repair

Backtracking builds the solution starting from an empty board and keeps it valid at every step. The n-queens problem can also be approached in a completely different way. First all the queens are put on the board, one in each column. This placement is very likely invalid: some queens attack each other. Then the conflicts are repaired one by one with small local changes. This approach is called the **min-conflicts heuristic**.

Definition 4.6

The **min-conflicts heuristic** is a local search method that starts from a complete assignment, which may be invalid. At each step one of the variables that violate constraints is chosen. This variable is given the value that creates the fewest conflicts with the other variables. The search stops when no conflicts remain or when the step limit is reached.

For n-queens the variables are the columns, and the value of each variable is the row of the queen in that column. `queen[k] = r` means that the queen in column k is in row r. The column constraint is built into the structure; there is exactly one queen in each column. Conflicts arise in rows and on diagonals.

```
minConflicts(n, maxSteps):
  // start: greedy placement
  for k = 0 .. n-1:
    queen[k] = row with the fewest conflicts
  for step = 1 .. maxSteps:
    if no conflicts: return queen
    k = random conflicted column
    queen[k] = argmin_r conflicts(k, r)
               // break ties at random
  return failure
```

The pseudocode has two phases. The **initialization** phase places the queens column by column. In each column the row that conflicts least with the queens placed so far is chosen; if there is a tie, one of the rows is taken at random. This is a greedy start and often leaves only a few conflicts. The **repair** phase picks one of the conflicted queens at random at each step. The chosen queen is moved, within its own column, to the row that creates the fewest conflicts. Here too ties are broken at random. When no conflicts remain, the placement is a solution. If no solution is found within `maxSteps` steps, the search counts as failed.

4.4.2 The number of conflicts

The basic computation of the algorithm is this: if the queen in column k were moved to row r, how many queens would it conflict with? This number is the count of the other queens in the same row and on the same diagonals.

```
int conflicts(int[] queen, int k, int r) {
  int c = 0;
  for (int j = 0; j < queen.length; j++) {
    if (j == k) continue;                      // the queen itself
    if (queen[j] == r                          // same row
        || Math.abs(queen[j] - r) == Math.abs(j - k))  // diagonal
      c++;
  }
  return c;
}
```

The method scans all the columns. The queen's own column is skipped. If the queen in column j is in the same row, the counter is incremented. If the absolute value of the row difference equals the absolute value of the column difference, the two squares lie on the same diagonal; the counter is incremented again. The computation takes $O(n)$. Computed for all the rows of one column it is $O(n^2)$. For large n, counters are kept for how many queens are in each row, each diagonal, and each anti-diagonal. Then a single conflict count is read in $O(1)$.

Example 4.6

Question. On the *4 × 4* board, `queen = [1, 3, 0, 3]`. Which queens conflict? If column 3 is chosen for repair, what is the conflict count of this queen for each row? Where is the queen moved?

Solution. The queens are at (row, column) positions (1, 0), (3, 1), (0, 2), and (3, 3). The queens in columns 1 and 3 are in the same row (3); there is no other conflict. The computation for column 3:

- Row 0: (0, 2) is in the same row: 1.
- Row 1: (1, 0) is in the same row; (0, 2) is on a diagonal (row difference 1, column difference 1); (3, 1) is on a diagonal (2 and 2): 3.
- Row 2: it conflicts with no queen: 0.
- Row 3: (3, 1) is in the same row: 1.

The counts are 1, 3, 0, 1. The smallest value, 0, occurs only in row 2. The queen is moved there and the placement becomes [1, 3, 0, 2]. No conflicts remain; the 4-queens solution has been found with a single repair step.

4.4.3 Repair steps

One run of the algorithm on 8 queens goes as follows. The greedy start produces the placement `queen = [7, 2, 4, 1, 4, 0, 3, 6]`. In this placement only one pair conflicts: the queens in columns 2 and 4 are in the same row (4). In the figures, the numbers in the highlighted column show how many queens the red queen would conflict with if it were moved to that row; the queen's own row is left blank.

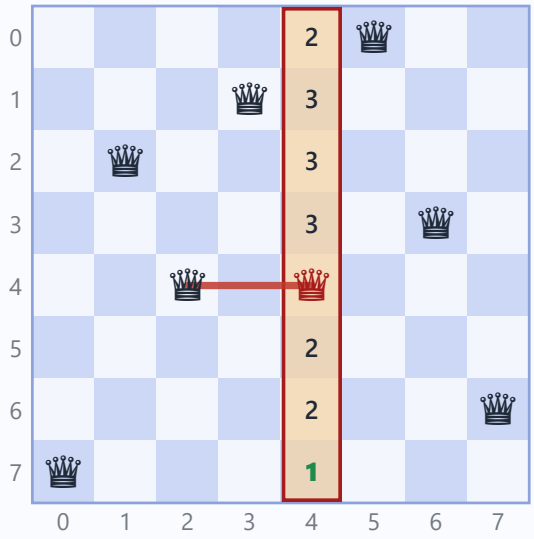


Figure 4.20: The greedy start. The queen in column 4 has been chosen. The conflict counts in its column, from row 0 to row 7, are 2, 3, 3, 3, (its own row), 2, 2, 1; the smallest is in row 7.

Step	`queen`	Conflicting	Chosen	Minimum value	Decision
1	[7, 2, 4, 1, 4, 0, 3, 6]	2 and 4 (row 4)	column 4	1 (row 7)	to row 7
2	[7, 2, 4, 1, 7, 0, 3, 6]	0 and 4 (row 7)	column 0	1 (rows 0, 5)	one at random
…	a few repair steps				
3	[2, 5, 3, 0, 7, 3, 1, 6]	2 and 5 (row 3)	column 5	1 (rows 4, 5)	to row 4
4	[2, 5, 3, 0, 7, 4, 1, 6]	5 and 7 (diagonal)	column 5	1 (rows 3, 5)	plateau; to row 5
5	[2, 5, 3, 0, 7, 5, 1, 6]	1 and 5 (row 5)	column 1	1 (row 7)	to row 7
6	[2, 7, 3, 0, 7, 5, 1, 6]	1 and 4 (row 7)	column 4	1 (rows 0, 2)	to row 2
7	[2, 7, 3, 0, 2, 5, 1, 6]	0 and 4 (row 2)	column 0	0 (row 4)	to row 4
8	[4, 7, 3, 0, 2, 5, 1, 6]	none	—	—	solution

Table 4.6: One run of the min-conflicts heuristic ($n = 8$). `queen[k]` is the row of the queen in column k.

The table shows two important behaviors of local search. First, a repair does not always remove a conflict; sometimes it only moves it. In step 1 the queen in column 4 is moved to row 7. The conflict with column 2 ends, but the queen in column 0 is also in row 7, and a new conflict arises. The number of conflicting pairs stays at 1. The algorithm never increases the number of conflicts, but it does not guarantee decreasing it either.

Second, the algorithm can get stuck on **plateaus**. In step 4 the queen in column 5 is in row 4 and is on the same diagonal as the queen at square (6, 7). In rows 3, 4, and 5 the conflict count is 1. No move improves the situation; this is a plateau. The heuristic breaks the tie at random and the queen goes to row 5. There, this time, it is in the same row as column 1. The queen in column 5 may go back and

forth a few times among rows 3, 4, and 5 on the plateau. Thanks to the random choice, another conflicted queen, in column 1, is eventually chosen, and the search leaves the plateau. In step 7 there are 0 conflicts in row 4 for column 0; the queen is moved there and the solution is found.

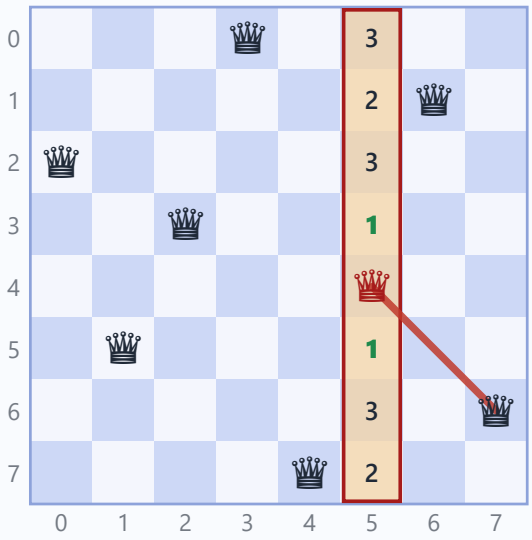


Figure 4.21: A plateau. For the queen in column 5, the conflict count is 1 in each of rows 3, 4, and 5; a move does not improve the situation.

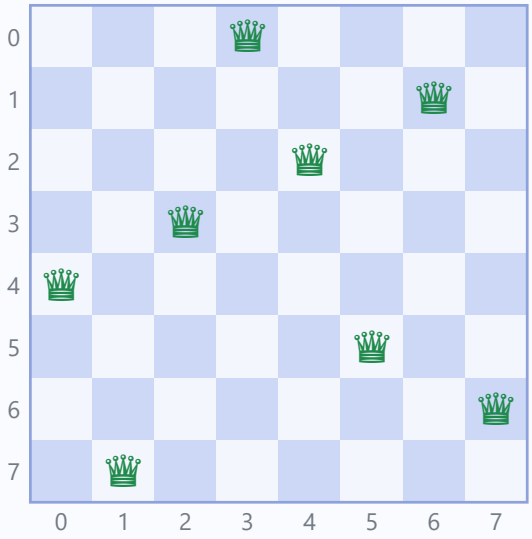


Figure 4.22: No conflicted queens are left: the solution is [4, 7, 3, 0, 2, 5, 1, 6].

The solution found is written in column order, not in row order. This difference should be kept in mind when comparing it with the solution [0, 4, 7, 5, 2, 6, 1, 3] of the backtracking algorithm. Both representations are permutations; one is the mapping from row to column, the other from column to row.

4.4.4 Local search and completeness

Min-conflicts is a **local search** method. It differs from backtracking in three basic ways.

- **State.** In backtracking the state is partial but always valid. In local search the state is complete but may be invalid.
- **Memory.** Local search has no backtrack stack. The algorithm keeps only the current placement; it does not remember past decisions.
- **Completeness.** Local search is **not complete**. It can wander on plateaus and enter a cycle, which is why a step limit is needed. It finds only a solution and cannot prove that none exists. Reaching the step limit does not mean “no solution”; it means only “not found in this run.”

On the other hand, min-conflicts is surprisingly fast for large n. According to the 1992 work of Minton and colleagues, with a good greedy start even a problem with a million queens is solved in an average of a few tens of repair steps. Backtracking, by contrast, slows down rapidly as n grows. The strengths of the two methods complement each other. To count all the solutions or to show that none exists, backtracking is needed. If a single solution to a very large instance is wanted quickly, local search is more suitable.

Common mistake

Taking local search to be complete. For $n = 3$, min-conflicts always reaches the step limit, because there is no solution. But for $n = 100$ reaching the step limit is also possible: the limit may have been chosen too small or the search may have fallen onto a bad plateau. The two cases cannot be told apart from the output of the algorithm. Local search cannot prove the answer “no solution.”

4.5 Sudoku

4.5.1 Sudoku as a constraint satisfaction problem

The goal of Sudoku is to fill in a 9×9 grid. In every row, every column, and each of the nine 3×3 boxes, the digits 1 to 9 are used once each. Some cells are given at the start; these are called **clues**. The remaining cells are filled in so as to obey these rules. A well-formed puzzle has a single solution.

Sudoku is an example of a more general kind of problem.

Definition 4.7

A **constraint satisfaction problem** (CSP) has three parts. The first is a set of variables. The second is, for each variable, the set of values it may take (its **domain**). The third is the **constraints** that the variables must obey together. A solution is an assignment that gives each variable a value from its domain and satisfies all the constraints.

In Sudoku the variables are the empty cells. The domain of each variable is *{1, ..., 9}*. The constraints are of the form "all different within a unit": cells in the same row, the same column, and the same box must get different digits. N-queens is also a constraint satisfaction problem: the variables are the rows and the domains are the columns. In graph coloring the variables are the vertices and the domains are the colors. Backtracking is the general method for solving constraint satisfaction problems.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 4.23: The example puzzle. 30 cells are clues and 51 cells are empty.

4.5.2 The algorithm

The backtracking algorithm solves Sudoku in the following steps.

1. A digit is assigned to an empty cell.
2. If the assigned digit is valid, that is, it does not already appear in the same row, column, or box, we move on to the next empty cell.
3. If it is not valid, the next digit is tried.
4. If all the digits have been tried in a cell and none worked, we return to the previous cell and try a different digit there.
5. The steps are repeated until all the cells are filled.

This is a direct application of the general template. The partial solution is the filled cells. The candidates are the digits 1–9. The constraint test is `isValid`. The goal is that no empty cell remains.

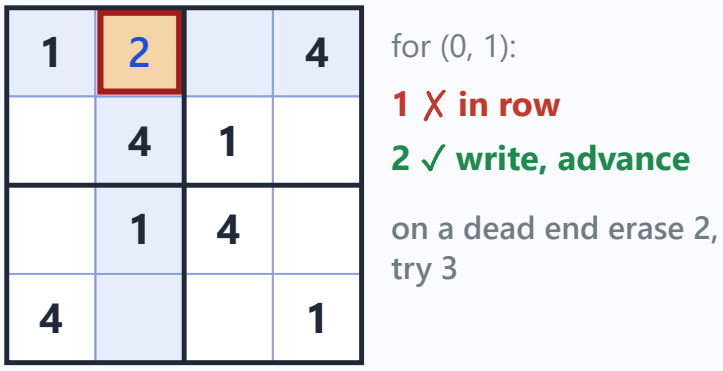


Figure 4.24: Cell (0, 1) in a small 4 × 4 Sudoku: 1 is already in the row and is rejected; 2 is valid, is written, and we move on to the next cell. If a dead end is reached later, the 2 is erased and 3 is tried.

The small puzzle in the figure is played with *2 × 2* boxes and the digits 1–4. In this puzzle no dead end is ever reached: 2 is written to (0, 1), 3 to (0, 2), and the remaining six empty cells are also filled with the first valid digit. The solution is found in eight placements. In a large puzzle the situation is very different.

4.5.3 Java code

```
boolean solve(int[][] b) {
  for (int r = 0; r < 9; r++)
    for (int c = 0; c < 9; c++)
      if (b[r][c] == 0) {
        for (int d = 1; d <= 9; d++)
          if (isValid(b, r, c, d)) {
            b[r][c] = d;
            if (solve(b)) return true;
            b[r][c] = 0;   // undo
          }
        return false;  // no digit fits
      }
  return true;  // no empty cell
}
```

At each call `solve` scans the grid from the beginning, looking for the first empty cell. When it finds an empty cell *(r, c)*, it tries every digit from 1 to 9. A valid digit is written and `solve` is called again. That call finds the next empty cell. If the call returns `true`, a solution has been found. If it returns `false`, the digit is erased and the next digit is tried. If none of the nine digits works, `return false` runs. This line triggers the backtrack: the calling frame erases the digit in its own cell.

There is a subtle point in the structure of the code. `return false` is *inside* the two loops, at the end of the block that handles the first empty cell. After handling the first empty cell, the method does not look at the remaining cells. If the first empty cell cannot be filled, there is no point in trying to fill the others. If the loops finish without finding any empty cell, the `return true` at the bottom runs: the grid is full.

```
boolean isValid(int[][] b,
    int r, int c, int d) {
  for (int i = 0; i < 9; i++)
    if (b[r][i] == d || b[i][c] == d)
      return false;
  int br = r / 3 * 3;
  int bc = c / 3 * 3;
  for (int i = br; i < br + 3; i++)
    for (int j = bc; j < bc + 3; j++)
      if (b[i][j] == d)
        return false;
  return true;
}
```

`isValid` checks whether the digit *d* can be written in cell *(r, c)*. The first loop scans both the row (`b[r][i]`) and the column (`b[i][c]`) in a single pass. Then the cell's box is scanned. The top-left corner of the box is found with `br = r / 3 * 3` and `bc = c / 3 * 3`. Integer division rounds down; multiplying by 3 converts the box number back to an index. For cell (7, 4), *7 / 3 = 2*, *2 · 3 = 6*; *4 / 3 = 1*, *1 · 3 = 3*. The box covers rows 6–8 and columns 3–5. The method looks at no more than 27 cells, 18 for the row and column and 9 for the box; this is constant work.

Common mistake

Computing the box corner as `r / 3` . This expression gives the number of the box (0, 1, or 2), not the index of its first row. For cell (7, 4) the box comes out as (2, 1), and rows 2–4 and columns 1–3 are scanned. That region is the wrong box. Forgetting the box check altogether causes a similar error: “solutions” are produced that obey the row and column rules but have invalid boxes.

4.5.4 Step-by-step solution

On the example puzzle the algorithm fills the empty cells row by row, from left to right; in each cell 1–9 are tried in order. In row 0 the clues are 5, 3, and 7. The first five empty cells, (0, 2), (0, 3), (0, 5), (0, 6), and (0, 7), receive 1, 2, 4, 8, and 9 respectively. Each is the smallest valid digit for that cell. In cell (0, 8) no digit is valid: the only digit still missing from the row is 6, and it is already in column 8, at cell (5, 8). This is the first dead end; up to this point 5 placements have been made and none undone.

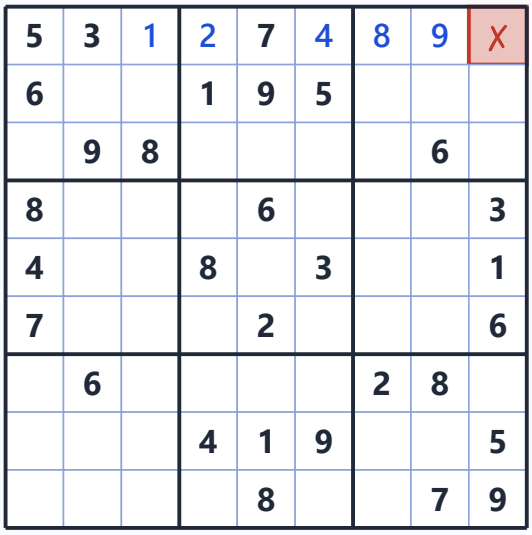


Figure 4.25: The first dead end. The blue digits are those the algorithm has written. No digit can be written in cell $(0, 8)$ (X).

The backtrack begins. The 9 in (0, 7) is erased; there is no digit larger than 9 in that cell, so (0, 7) is also a dead end. The 8 in (0, 6) is erased and 9 is tried. This time no digit fits in (0, 7); the 9 is erased too. (0, 6) is also a dead end. The backtrack reaches (0, 5), where 6 is tried instead of 4. At this point 7 placements and 4 undos have been made. Then 4 is written to (0, 6), 9 to (0, 7), and 8 to (0, 8), and row 0 is completed for the first time: 5 3 1 2 7 6 4 9 8. This row is valid in itself, but it is not yet known whether it leads to a solution.

In fact it does not. All the attempts made in row 1 end in dead ends. The backtrack reaches row 0 and the 8 in (0, 8) is erased; by then 31 placements and 26 undos have been made. A decision made early can turn out to be wrong only much later. The algorithm finds this out only after trying everything that depends on that decision.

Row	Placements	Undos	Note
0	10	4	5 3 1 2 7 6 4 9 8 (will change later)
1	229	218	
2	377	360	the first three rows are now the same as in the final solution
3	395	372	
4	1520	1492	
5	4185	4151	most of the work is done by this row
6	4197	4157	
7	4202	4157	no more undos
8	4208	4157	solution

Table 4.7: Total numbers of placements and undos made at the moment each row is completed for the first time.

The table shows the irregular progress of the search. The first three rows settle into place within 377 placements. Between row 4 and row 5, however, 2665 placements are made. In this interval the algorithm changes the choices in rows 3–5 over and over. After row 6 almost no backtracking remains: none of the last 11 placements is undone. In the end all 51 empty cells are filled and all the constraints are satisfied. In total 4208 placements and 4157 undos were made. The difference is *4208 − 4157 = 51*: the digits that remain in the solution.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4.26: The solution has been found. Black digits are clues and blue digits are those the algorithm wrote; the orange cell is the last cell written.

4.5.5 Cost and MRV

If brute force wrote one of 9 digits into each of the 51 empty cells and only then checked, $9^{51} \approx 4.6 \cdot 10^{48}$ assignments would be needed. Pruning reduces this number to 4208. The upper bound for the worst case is still $O(9^m)$, where m is the number of empty cells. In practice, though, only a small part of the tree is visited.

The same puzzle can be solved with much less work. It is not compulsory to fill the first empty cell in row-by-row order. Which cell to fill first is a choice, and this choice determines the height of the search. The **MRV** (minimum remaining values) heuristic fills, at each step, the empty cell that has *the fewest candidates left*.

```
boolean solveMRV(int[][] b) {
  int br = -1, bc = -1, fewest = 10;
  for (int r = 0; r < 9; r++)
    for (int c = 0; c < 9; c++)
      if (b[r][c] == 0) {
        int cands = 0;
        for (int d = 1; d <= 9; d++)
          if (isValid(b, r, c, d)) cands++;
        if (cands < fewest) { fewest = cands; br = r; bc = c; }
      }
  if (br < 0) return true;              // no empty cell
  for (int d = 1; d <= 9; d++)
    if (isValid(b, br, bc, d)) {
      b[br][bc] = d;
      if (solveMRV(b)) return true;
      b[br][bc] = 0;                    // undo
    }
  return false;
}
```

The method first scans all the empty cells and counts how many valid digits are left for each. The cell with the fewest candidates, *(br, bc)*, is chosen. If there is no empty cell, the grid is full. In the chosen cell the valid digits are tried in order; the structure is the same as in the other backtracking code. If a cell has 0 candidates, the loop never runs and the method returns `false` at once. This is an early detection of a contradiction.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4.27: The solution of the puzzle. Both strategies reach the same grid.

Strategy	Placements	Undos
Row by row, first empty cell	4208	4157

Strategy	Placements	Undos
MRV: cell with the fewest candidates	51	0

Table 4.8: The cost of two cell-selection strategies on the example puzzle.

In this puzzle MRV finds a cell with a single candidate at every step. At the start, 4 of the 51 empty cells have a single candidate. When one of them is filled, the candidates of the neighboring cells shrink and new single-candidate cells appear. The chain never breaks. All 51 placements are correct and no backtracking is needed. Because the scan at each step calls `isValid` for 81 cells and 9 digits, the work per step grows. But since the number of steps falls from 4208 to 51, the total work still drops a great deal.

MRV does not eliminate backtracking in every puzzle. In hard puzzles there comes a point at which every empty cell has at least two candidates left. Then MRV picks a cell with two candidates, and if it tries the wrong candidate it has to backtrack. Even so, starting with the cell with the fewest candidates keeps the branching smallest and exposes contradictions as early as possible.

4.6 Graph coloring

4.6.1 The problem

We want the neighboring countries on a map to be painted in different colors. If the countries are thought of as vertices and the shared borders as edges, this question becomes a graph problem. The same problem appears in other guises. In an exam schedule, two exams taken by the same student cannot be placed at the same time; the exams are vertices, the clashes are edges, and the time slots are colors. In a compiler, two variables that are live at the same time cannot share a register; the variables are vertices and the registers are colors.

Definition 4.8

The **m-coloring problem**: can the vertices of a graph be assigned colors from 1 to m so that no two vertices joined by an edge get the same color? The smallest value of m that suffices to color a graph is called its **chromatic number**.

Backtracking fits this problem naturally. The vertices are colored in the order 0, 1, 2, ... The partial solution is the colors of the first k vertices. The candidates are the m colors. The constraint is that the color of a vertex must differ from the color of each of its neighbors. The decision version of 3-coloring is NP-complete; for $m = 2$, on the other hand, the problem is solved in polynomial time, because 2-colorability is equivalent to being bipartite and can be tested with a breadth-first search.

4.6.2 Step by step: a graph with six vertices

The graph has six vertices and nine edges: 0–1, 0–2, 0–5, 1–3, 1–4, 2–3, 2–5, 3–4, and 4–5. Three colors are used: 1 blue, 2 green, 3 orange. At each vertex the colors are tried in the order 1, 2, 3.

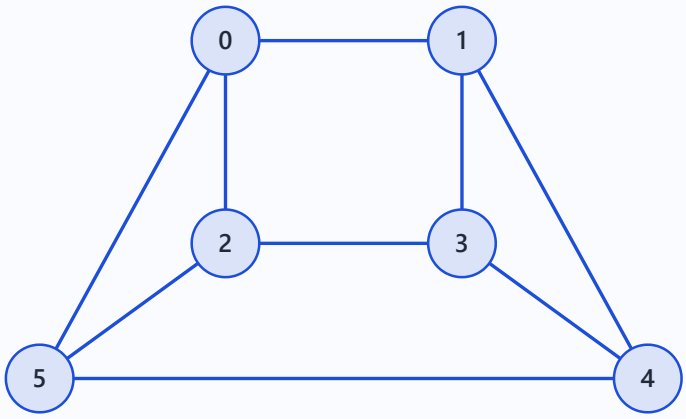


Figure 4.28: The six-vertex graph to be colored, *m = 3*.

Step	Vertex	Neighbor colors	Decision	`colors[0..5]`
1	0	—	color 1	[1, 0, 0, 0, 0, 0]
2	1	0: 1	1 is taken; color 2	[1, 2, 0, 0, 0, 0]
3	2	0: 1	1 is taken; color 2	[1, 2, 2, 0, 0, 0]
4	3	1: 2, 2: 2	color 1	[1, 2, 2, 1, 0, 0]
5	4	1: 2, 3: 1	1 and 2 are taken; color 3	[1, 2, 2, 1, 3, 0]
6	5	0: 1, 2: 2, 4: 3	dead end; the color of vertex 4 is erased	[1, 2, 2, 1, 0, 0]
7	4	—	no color left to try; the color of vertex 3 is erased	[1, 2, 2, 0, 0, 0]
8	3	1: 2, 2: 2	2 is taken; color 3	[1, 2, 2, 3, 0, 0]
9	4	1: 2, 3: 3	color 1	[1, 2, 2, 3, 1, 0]
10	5	0: 1, 2: 2, 4: 1	1 and 2 are taken; color 3	[1, 2, 2, 3, 1, 3]

Table 4.9: A 3-coloring of the six-vertex graph. The “Neighbor colors” column gives the colors of the neighbors that are colored at that moment.

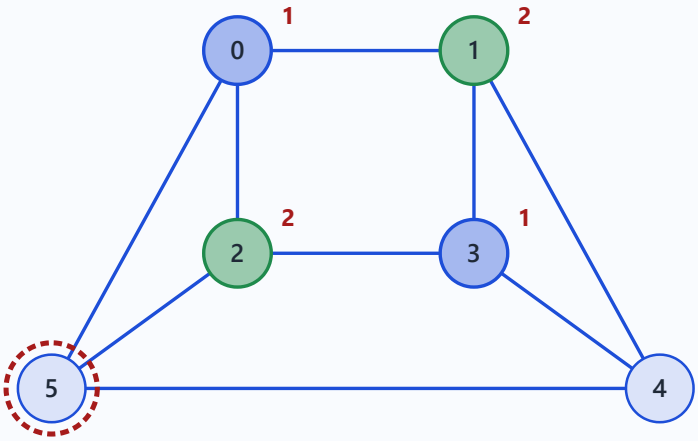


Figure 4.29: Before step 6: the neighbors of vertex 5, namely 0, 2, and 4, use all three colors. No color is possible; dead end.

The source of the dead end is the decision in step 4. When vertex 3 is painted blue, vertex 4 is forced to be orange. Vertex 5 is adjacent to the blue vertex 0, the green vertex 2, and the orange vertex 4. The backtrack goes first to vertex 4. Vertex 4 has no color greater than 3 and it, too, hits a dead end. The backtrack extends to vertex 3. When vertex 3 becomes orange, vertex 4 can be blue; the neighbors of vertex 5 are then blue, green, and blue, and vertex 5 is painted orange. The valid 3-coloring [1, 2, 2, 3, 1, 3] is found.

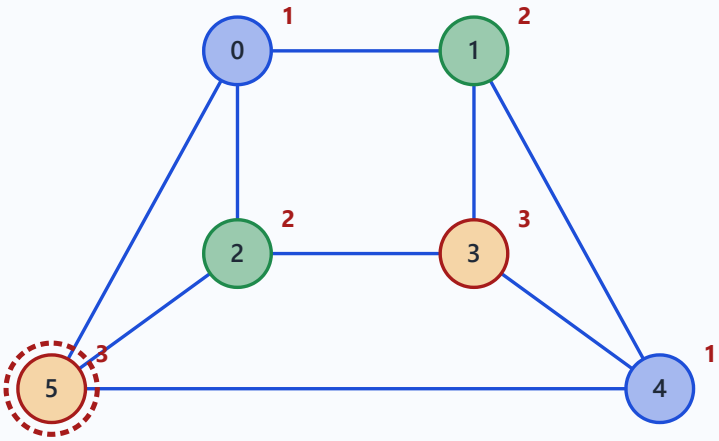


Figure 4.30: All six vertices are colored. The number beside each vertex is its color: [1, 2, 2, 3, 1, 3].

4.6.3 Java code and code trace

```
boolean colorGraph(int v) {
  if (v == n) return true;          // done
  for (int c = 1; c <= m; c++) {
    if (canUse(v, c)) {
      colors[v] = c;                // choose
      if (colorGraph(v + 1)) return true; // explore
      colors[v] = 0;                // unchoose
    }
  }
  return false;                     // dead end
}
// canUse: true if v has no neighbor of color c
```

`colorGraph(v)` assumes that vertices 0 through *v* – *1* have been colored and continues coloring from vertex *v*. If `v == n`, all the vertices have been colored. The loop tries the colors from 1 to *m*. A suitable color is written to `colors[v]` and we move on to the next vertex. If the recursive call fails, the color is set back to 0; 0 means "uncolored." If all the colors fail, `false` is returned.

The slides give the body of `canUse` only as a one-line description. A version written with an adjacency matrix is as follows:

```
boolean canUse(int v, int c) {
  for (int u = 0; u < n; u++)
    if (adj[v][u] == 1 && colors[u] == c)
      return false;   // a neighbor has color c
  return true;
}
```

The method looks at all the neighbors of *v*. If one of the neighbors has color *c*, it returns `false`. Neighbors not yet colored have color 0, which matches no *c*. So only neighbors colored earlier affect the constraint. With an adjacency matrix this check takes *O(n)*; with an adjacency list it takes *O(degree)*.

For the code trace, let us use a smaller graph: five vertices and six edges, 0–1, 0–3, 1–3, 1–4, 2–4, and 3–4, with *m = 3*.

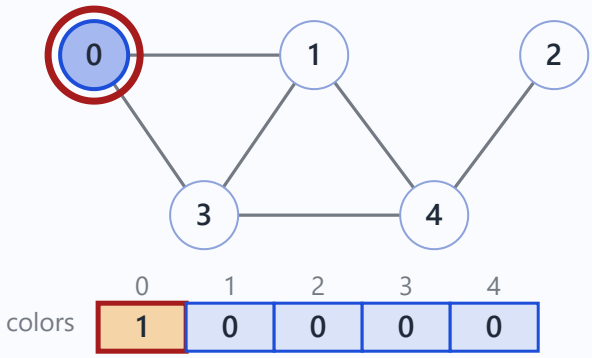


Figure 4.31: `colorGraph(0)` begins. Vertex 0 has been given color 1; the other cells of the `colors` array are 0.

Call	Colors tried	Result	`colors`
`colorGraph(0)`	1 fits	`colors[0] = 1`	[1, 0, 0, 0, 0]
`colorGraph(1)`	1 (used by 0), 2 fits	`colors[1] = 2`	[1, 2, 0, 0, 0]
`colorGraph(2)`	1 fits	`colors[2] = 1`	[1, 2, 1, 0, 0]
`colorGraph(3)`	1 (used by 0), 2 (used by 1), 3 fits	`colors[3] = 3`	[1, 2, 1, 3, 0]
`colorGraph(4)`	1 (used by 2), 2 (used by 1), 3 (used by 3)	dead end, `false`; `colors[3] = 0`	[1, 2, 1, 0, 0]
`colorGraph(3)` resumes	no color left	`false`; `colors[2] = 0`	[1, 2, 0, 0, 0]
`colorGraph(2)` resumes	2 fits	`colors[2] = 2`	[1, 2, 2, 0, 0]
`colorGraph(3)`	1, 2 are taken; 3 fits	`colors[3] = 3`	[1, 2, 2, 3, 0]
`colorGraph(4)`	1 fits	`colors[4] = 1`	[1, 2, 2, 3, 1]
`colorGraph(5)`	—	`v == n`, `true`	[1, 2, 2, 3, 1]

Table 4.10: Trace of the call `colorGraph(0)` (five vertices, $m = 3$).

The interesting point of the trace is vertex 2. The only neighbor of vertex 2 is vertex 4, and when 2 is colored, vertex 4 is not yet colored. So vertex 2 takes color 1 on the first try. This choice is locally correct, but it forces the three neighbors of vertex 4 (1, 2, and 3) into three different colors. The mistake is discovered only when vertex 4 is reached. The backtrack passes through vertex 3 and reaches vertex 2. When vertex 2 becomes green, vertex 4 can be painted blue. The result is [1, 2, 2, 3, 1].

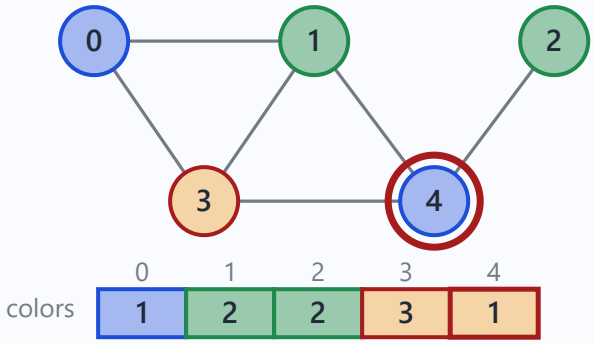


Figure 4.32: `v = 5 = n` : all the vertices are colored; the valid 3-coloring is [1, 2, 2, 3, 1].

In terms of cost, in the worst case every vertex tries all m colors. The tree has at most m^n leaves. At each node `canUse` takes $O(n)$. The total time is $O(n \cdot m^n)$. Memory is $O(n)$ for the `colors` array and the call stack.

The symmetry of the colors is a source of unnecessary work. If the names of the colors in a coloring are permuted, another valid coloring results: [1, 2, 2, 3, 1] and [2, 1, 1, 3, 2] give the same structure. So fixing the color of vertex 0 to 1 shrinks the search by roughly a factor of m. When a dead end is found, it is pointless to try vertex 0 with another color; even with the color names changed, the same dead end is reached.

4.6.4 Odd cycles and the minimum number of colors

Example 4.7

Question. We try to color the five-vertex cycle graph C_5 (edges 0–1, 1–2, 2–3, 3–4, 4–0) with $m = 2$ colors using `colorGraph(0)` . What does the search return? What is the minimum number of colors needed?

Solution. Vertex 0 takes color 1. The neighbor of vertex 1, vertex 0, has color 1; it takes color 2. Vertex 2 takes color 1 and vertex 3 takes color 2. The neighbors of vertex 4 are 0 (color 1) and 3 (color 2). Both colors are taken; dead end. The backtracks do not change the outcome. With two colors on a cycle, the colors necessarily alternate: 1, 2, 1, 2, ... Even if vertex 0 is tried with color 2, the sequence becomes 2, 1, 2, 1 and vertex 4 is again adjacent to both colors. The search traverses the whole tree and returns `false` . This result is a proof that there is no solution; the search is complete. With $m = 3$, `colorGraph(0)` finds the coloring [1, 2, 1, 2, 3] on the first attempt. The chromatic number of C_5 is 3.

The example is a special case of a general rule. A cycle of odd length cannot be colored with two colors. Cycles of even length can be colored with two colors: the colors are distributed alternately and the last vertex ends up with a color different from the first. More generally, a graph is 2-colorable if

and only if it contains no odd cycle, that is, if it is bipartite. To find the chromatic number of a graph, `colorGraph` can be run with increasing values of *m*: 1, 2, 3, … The first *m* that succeeds is the chromatic number. Note that every failed attempt traverses the whole tree; the most expensive step is usually the last failed attempt.

4.7 Hamiltonian path

4.7.1 The problem and a step-by-step solution

> **Definition 4.9**
>
> A **Hamiltonian path** is a path that visits every vertex of a graph exactly once. If the last vertex of the path is also adjacent to the first, the path closes and a **Hamiltonian cycle** is obtained.

A Hamiltonian path looks similar to an Euler path, which traverses every *edge* exactly once, but the two are very different in difficulty. The existence of an Euler path is decided in linear time by looking at the vertex degrees. The Hamiltonian path problem, on the other hand, is NP-complete; all known exact algorithms take exponential time.

In backtracking the partial solution is a path: `path[0..pos - 1]`. The candidate is the next vertex to be appended to the path. The constraint has two parts: the new vertex must be adjacent to the last vertex of the path, and it must not already be on the path. The goal is for the path to reach *n* vertices.

The example graph has five vertices and six edges: 0–1, 0–2, 1–3, 1–4, 2–3, and 3–4. The path starts at vertex 0; `path[0] = 0`. At each position the vertices are tried in the order 0, 1, 2, ….

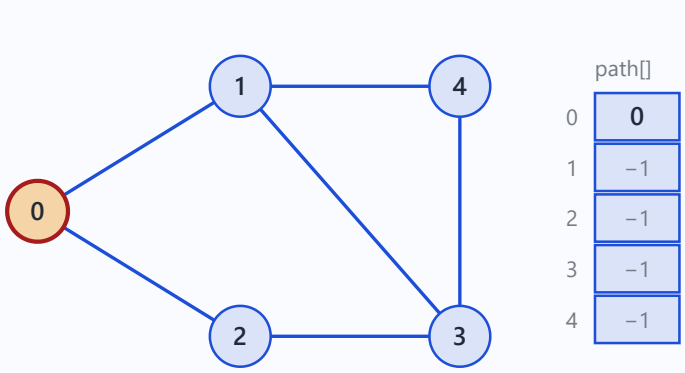


Figure 4.33: The start: `path[0] = 0`, the other positions are −1 (empty).

1. Position 1: vertex 1 is adjacent to 0; `path[1] = 1`.

2. Position 2: the neighbors of 1 are 0, 3, and 4. 0 is on the path; the first suitable vertex is 3. `path[2] = 3`.
3. Position 3: the neighbors of 3 are 1, 2, and 4. The first suitable vertex is 2. `path[3] = 2`.
4. Position 4: the neighbors of the last vertex, 2, are 0 and 3; both are on the path. Dead end. Vertex 2 is removed.
5. Position 3: the next suitable vertex is 4. `path[3] = 4`.
6. Position 4: the neighbors of 4, namely 1 and 3, are on the path. Dead end. Vertex 4 is removed.
7. Position 3: vertex 3 has no untried neighbor left. Vertex 3 is removed too.
8. Position 2: the next suitable neighbor of 1 is 4. `path[2] = 4`.
9. Position 3: the neighbor of 4 that is not on the path is 3. `path[3] = 3`.
10. Position 4: the neighbor of 3 that is not on the path is 2. `path[4] = 2`. The path has reached five vertices.

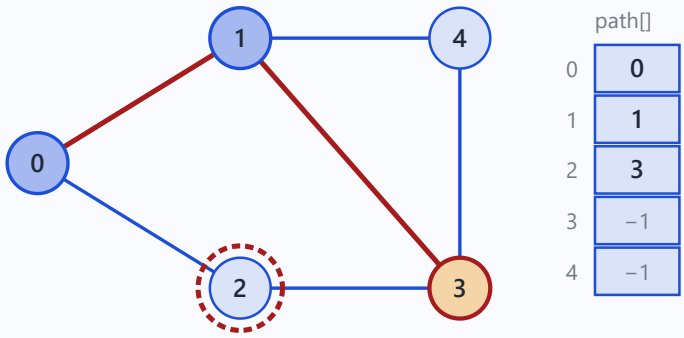


Figure 4.34: Step 4: the path is 0 → 1 → 3 → 2. All the neighbors of the last vertex, 2, are on the path; 2 will be removed.

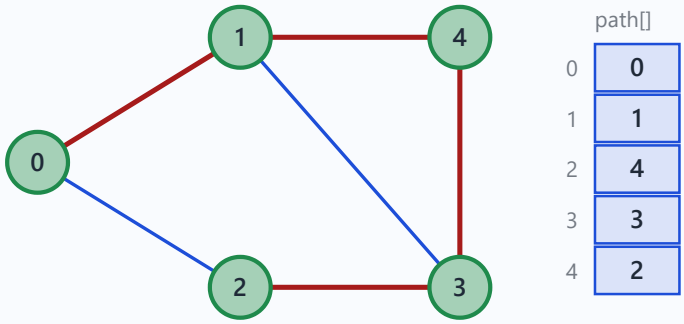


Figure 4.35: The Hamiltonian path has been found: 0 → 1 → 4 → 3 → 2.

4.7.2 Java code

```
boolean find(int[][] g, int[] path, int pos) {
  if (pos == g.length) return true;
  for (int v = 0; v < g.length; v++) {
    if (canExtend(v, g, path, pos)) {
      path[pos] = v;
      if (find(g, path, pos + 1)) return true;
      path[pos] = -1;     // undo
    }
  }
  return false;
}

boolean canExtend(int v, int[][] g,
                  int[] path, int pos) {
  if (g[path[pos - 1]][v] == 0) return false;
  for (int i = 0; i < pos; i++)
    if (path[i] == v) return false;
  return true;
}
```

`g` is the adjacency matrix; `g[u][v] == 1` means there is an edge between *u* and *v*. The calling code fills the `path` array with −1, sets `path[0] = 0`, and calls `find(g, path, 1)`. `pos == g.length` is the goal test: the path has *n* vertices. The loop tries every vertex as a candidate. `canExtend` checks the two constraints. The first line checks whether the candidate is adjacent to the last vertex, that is, to `path[pos - 1]`. The loop scans whether the candidate is already on the path. The undo line is `path[pos] = -1`.

Here too the observation from the permutation code applies: the undo line is not essential for correctness. `canExtend` looks only at `path[0..pos - 1]`, and the old value in `path[pos]` is overwritten in the next attempt. Even so, writing −1 keeps the state clean and makes debugging easier.

The cost analysis is similar to permutations. Each position of the path is one of the remaining vertices; in the worst case *(n − 1)!* orders are tried. At each node, for *n* candidates, `canExtend` takes *O(n)*. The crude bound is *O(n · n!)*. If membership in the path is kept in a `boolean` array, `canExtend` drops to *O(1)*.

4.7.3 Code trace: the second graph

Let us trace the code on a second graph. It has five vertices and six edges: 0–1, 0–2, 1–2, 1–3, 2–3, and 2–4. The only neighbor of vertex 4 is 2. So vertex 4 can be on a Hamiltonian path only at an end.

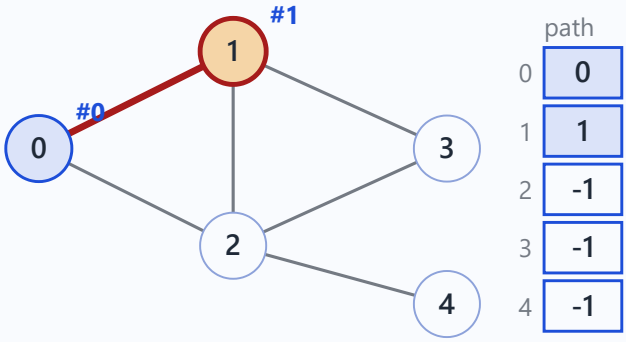


Figure 4.36: The second graph. `find(g, path, 1)` has been called with `path[0] = 0`; at position 1 the first vertex adjacent to 0, vertex 1, has been chosen.

Position	Last vertex	Decision	`path`
1	0	0 is on the path; first suitable is 1	[0, 1, −1, −1, −1]
2	1	first suitable is 2	[0, 1, 2, −1, −1]
3	2	first suitable is 3	[0, 1, 2, 3, −1]
4	3	the neighbors of 3, namely 1 and 2, are on the path; dead end, 3 is removed	[0, 1, 2, −1, −1]
3	2	next suitable is 4	[0, 1, 2, 4, −1]
4	4	the only neighbor of 4, vertex 2, is on the path; dead end, 4 is removed	[0, 1, 2, −1, −1]
3	2	no candidates left; dead end, 2 is removed	[0, 1, −1, −1, −1]
2	1	next suitable is 3	[0, 1, 3, −1, −1]
3	3	first suitable is 2	[0, 1, 3, 2, −1]
4	2	first suitable is 4	[0, 1, 3, 2, 4]
5	—	`pos == n`, `true`	Hamiltonian path 0 → 1 → 3 → 2 → 4

Table 4.11: Trace of the call `find(g, path, 1)` (second graph).

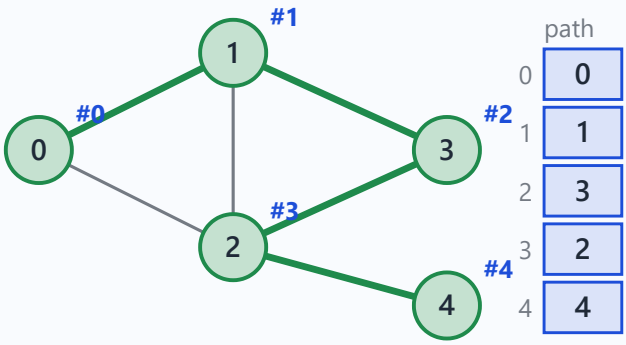


Figure 4.37: The path found: 0 → 1 → 3 → 2 → 4. The labels #0–#4 beside the vertices show the order along

the path.

In the trace the algorithm does not know that vertex 4 can be only at an end. It tries 3 after 2, and then 4. Both attempts end in dead ends. A person looking at this graph says, "4 must be the last vertex, and 2 the one before it," and narrows the search at once. Reasoning of this kind can be turned into pruning rules. For example, if a vertex not on the path has no neighbor off the path and is not adjacent to the last vertex either, the branch can be cut at once.

4.7.4 Hamiltonian cycle

To search for a Hamiltonian cycle, the only change in the code is in the goal test. When the path reaches *n* vertices, we also check whether the last vertex is adjacent to the starting vertex.

```
boolean cycle(int[][] g, int[] path, int pos) {
  if (pos == g.length)
    return g[path[pos - 1]][path[0]] == 1;   // last adjacent to start?
  for (int v = 0; v < g.length; v++) {
    if (canExtend(v, g, path, pos)) {
      path[pos] = v;
      if (cycle(g, path, pos + 1)) return true;
      path[pos] = -1;     // undo
    }
  }
  return false;
}
```

If the goal test returns `false`, the search continues; this path has not closed into a cycle, and other paths are tried. In the first example graph, the last vertex of the path $0 \to 1 \to 4 \to 3 \to 2$, vertex 2, is adjacent to the starting vertex 0. So `cycle` finds the same path and yields a Hamiltonian cycle closed by the edge $2 \to 0$. The second graph has no Hamiltonian cycle. Vertex 4 has only one neighbor; in a cycle each vertex uses two neighbors. `cycle` traverses the whole tree and returns `false`.

Common mistake

Looking only at the test `pos == n` when searching for a Hamiltonian cycle. This test finds a Hamiltonian *path*. If it is not checked separately that the last vertex is adjacent to the first, the path $0 \to 1 \to 3 \to 2 \to 4$ in the second graph is wrongly taken to be a cycle.

4.8 Word search

4.8.1 The problem

A word is searched for in a grid of letters. The word must be read along a path through adjacent cells. Adjacency is four-directional: up, down, left, and right. A cell may be used at most once in the same path. The question is: is there such a path that spells the word?

The example grid has three rows: ABCE, SFCS, and ADEE. Let the word searched for be “SEE.” Every cell is tried in turn as the starting point. The neighboring directions are tried in the order up, down, left, right.

A	B	C	E
S	F	C	S
A	D	E	E

Figure 4.38: The letter grid: ABCE / SFCS / ADEE.

1. (1, 0) = 'S' matches the first letter of the word and is marked as visited. Its neighbors are A, A, and F; none is 'E'. We backtrack and the mark is removed.
2. (1, 3) = 'S' matches and is marked. Its upper neighbor (0, 3) = 'E' matches and is marked.
3. The neighbors of (0, 3): above is outside the grid, below is (1, 3), which is visited, left is (0, 2) = 'C', right is outside the grid. The third letter cannot be found; the mark is removed.
4. The lower neighbor of (1, 3), (2, 3) = 'E', matches and is marked.
5. The left neighbor of (2, 3), (2, 2) = 'E', matches. The last letter matches too: “SEE” is found.

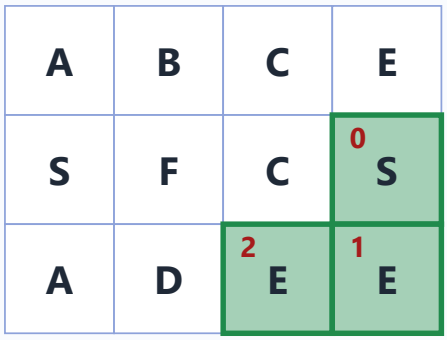


Figure 4.39: "SEE" has been found. The small numbers are the index of the letter in the word: (1, 3) S, (2, 3) E, (2, 2) E.

4.8.2 Java code

```
boolean search(char[][] g, String word, int i,
               int r, int c, boolean[][] seen) {
  if (i == word.length()) return true;
  if (r < 0 || r >= g.length || c < 0
      || c >= g[0].length || seen[r][c]
      || g[r][c] != word.charAt(i))
    return false;
  seen[r][c] = true;
  for (int d = 0; d < 4; d++)
    if (search(g, word, i + 1, r + dr[d],
               c + dc[d], seen))
      return true;
  seen[r][c] = false;   // undo
  return false;
}
```

`search` tells whether the part of the word after its *i*-th letter can be found starting from cell *(r, c)*. `i == word.length()` is the goal test: all the letters have matched. The second `if` rejects three cases at once: the cell is outside the grid, it has already been used on this path, or its letter is not the expected one; then `false` is returned. This is a style in which the constraint test is made at the *start* of the recursive call. In the n-queens code, by contrast, the test was made before the call, with `isSafe`. Both styles do the same pruning.

A cell that passes the tests is marked (choose). `search` is called for the four neighbors with the next letter (explore). `dr` and `dc` are the direction arrays. If one of the neighbors succeeds, `true` is passed straight up. If none succeeds, the mark is removed (unchoose) and `false` is returned. The outer code tries every cell as the starting point:

```
int[] dr = {-1, 1, 0, 0};   // up, down, left, right
int[] dc = {0, 0, -1, 1};

boolean exists(char[][] g, String word) {
  for (int r = 0; r < g.length; r++)
    for (int c = 0; c < g[0].length; c++) {
      boolean[][] seen = new boolean[g.length][g[0].length];
      if (search(g, word, 0, r, c, seen))
        return true;
    }
  return false;
}
```

There is a reason the goal test comes before the boundary check. If the last letter of the word matches in a cell on the edge of the grid, some of the four neighbor calls fall outside the grid. But in these calls *i* already equals the length of the word, and the first line returns `true`. If the order were reversed, two problems would arise. The calls falling outside the grid would return `false`. The calls inside the grid would try to read beyond the end of the word with `word.charAt(i)` and throw an exception. For example, in a one-cell grid a one-letter word could never be found.

Word	Result	Path or reason
ABCCED	found	(0, 0) A → (0, 1) B → (0, 2) C → (1, 2) C → (2, 2) E → (2, 1) D
SEE	found	(1, 3) S → (2, 3) E → (2, 2) E
ABCB	not found	B cannot be used twice
SADE	found	(1, 0) S → (2, 0) A → (2, 1) D → (2, 2) E

Table 4.12: Some words in the grid ABCE / SFCS / ADEE.

The search for “SADE” also involves a backtrack. From the S at (1, 0), the A above, at (0, 0), is tried first. The neighbors of this A are B, S (visited), and outside the grid; 'D' cannot be found. The mark is removed and the A below, at (2, 0), is tried. From there, with (2, 1) = 'D' and (2, 2) = 'E', the word is completed.

4.8.3 Code trace: “ABCB”

The word “ABCB” is not in the grid, but to see why, the search must be traced. The search starts at (0, 0).

1. (0, 0) = 'A' = word[0]. The cell is marked.
2. word[1] = 'B' is sought. Up is outside the grid, down (1, 0) = 'S' ≠ 'B', left is outside the grid. The cell to the right, (0, 1) = 'B', matches and is marked.
3. word[2] = 'C' is sought. Up is outside the grid, down (1, 1) = 'F' ≠ 'C', left (0, 0) is visited. The cell to the right, (0, 2) = 'C', matches and is marked.

4. word[3] = 'B' is sought. Up is outside the grid, down (1, 2) = 'C' ≠ 'B', left (0, 1) is visited, right (0, 3) = 'E' ≠ 'B'. All four directions fail. The only B in the grid is at (0, 1), and it has been used on this path.
5. Undo for (0, 2): `seen[0][2] = false`, `false` is returned.
6. The remaining directions of (0, 1) and (0, 0) fail as well. The marks of both are removed. `search(…, 0, 0, 0, …)` returns `false`.
7. The outer loop tries the other cells. The only other cell starting with A is (2, 0); its neighbors are S, D, and outside the grid, and there is no B. "ABCB" is not in the grid.

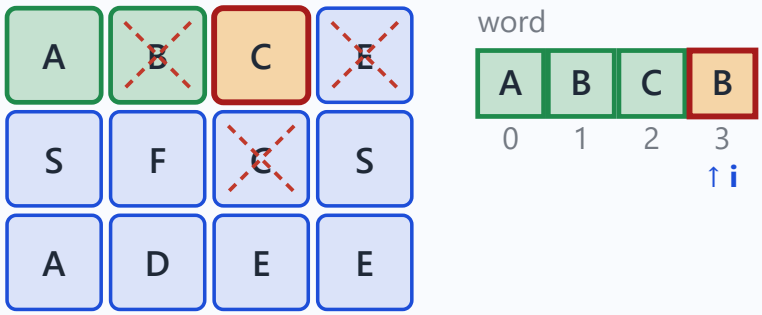


Figure 4.40: Step 4: 'B' is sought from (0, 2). The crossed cells are the neighbors that were rejected; (0, 1), where the B is, has already been used on this path.

For the cost, let the length of the word be L and the dimensions of the grid $R \times C$. Every cell is a starting point. After the first letter there are at most 3 new directions at each step, because the cell we came from is visited. So the number of paths traversed from one starting point is bounded by $O(3^L)$ and the total time is $O(R \cdot C \cdot 3^L)$. Memory is $O(R \cdot C)$ for the `seen` array and $O(L)$ for the stack. In practice, letter matching cuts the search very early; the search for "ABCB" marks only 4 cells.

Common mistake

Not removing the visited mark when backtracking. If the mark stays, cells passed through in a failed attempt look "used" in later attempts. In the search for "SADE", even if the mark on the A at (0, 0) stayed after it was tried and abandoned, the result would not change. But if the S at (1, 0) stayed marked and the word needed to use this S on another path, a word that exists would not be found. The correct order is: the cell is marked first, then the neighbors are explored, and finally the mark is removed.

4.9 Other applications

4.9.1 Subset sum

Given a set of positive integers and a target value, is there a subset whose sum is exactly the target? This question is the **subset sum** problem, and its decision version is NP-complete. This time the state-space tree is a binary tree. At level i there are two options for the i-th item: it is taken or it is not. For n items the tree has 2^n leaves; each leaf is a subset.

```
int[] a = {3, 5, 6, 7};
int TARGET = 15;
boolean[] chosen = new boolean[a.length];

boolean subset(int i, int sum) {
  if (sum == TARGET) return true;
  if (i == a.length || sum > TARGET) return false;   // prune
  chosen[i] = true;                    // take
  if (subset(i + 1, sum + a[i])) return true;
  chosen[i] = false;                   // skip
  return subset(i + 1, sum);
}
```

`subset(i, sum)` is called when the first i items have been decided and the sum of those chosen is `sum`. If the target has been reached, `true` is returned. If no items are left or the sum has exceeded the target, `false` is returned. The second condition is the pruning: because the items are positive, once the sum has exceeded the target it can never decrease again. Then the two branches are tried in turn. First the i-th item is taken; if this branch fails, the item is left out and the second branch is tried. In this code `sum`, which is part of the partial solution, is passed by value, so no undo is needed for it. Only the `chosen` array is undone.

#	Call	Meaning	Result
1	`subset(0, 0)`	root	descends into the branch that takes 3
2	`subset(1, 3)`	3 was taken	descends into the branch that takes 5
3	`subset(2, 8)`	5 was taken	descends into the branch that takes 6
4	`subset(3, 14)`	6 was taken	descends into the branch that takes 7
5	`subset(4, 21)`	7 was taken	$21 > 15$: pruned, `false`
6	`subset(4, 14)`	7 was not taken	no items left, $14 \neq 15$: `false`
7	`subset(3, 8)`	6 was not taken	descends into the branch that takes 7
8	`subset(4, 15)`	7 was taken	`sum == TARGET` : `true`

Table 4.13: The order of the calls of `subset(0, 0)` for `a = {3, 5, 6, 7}`, `TARGET = 15`.

The search ends in 8 calls and `chosen = [true, true, false, true]`: the subset found is {3, 5, 7}. The whole tree has $2^5 - 1 = 31$ nodes. Call 5 is where the pruning acts: the sum of 3, 5, 6, and 7 is 21, which exceeds the target.

This pruning cuts only "excess" sums. A pruning in the opposite direction is also possible: if the target cannot be reached even by taking all the remaining items, the branch is cut. For this, `remaining[i]`, the sum of the items from the *i*-th to the end (a suffix sum), is computed in advance, and the following line is added:

```
if (sum + remaining[i] < TARGET) return false;
```

Example 4.8

Question. Let the target be 22 in the same set. How many calls does `subset(0, 0)` make without and with the remaining-sum pruning?

Solution. The sum of the items is $3 + 5 + 6 + 7 = 21$; no subset can reach 22. In the code without the extra pruning, no sum exceeds 22, so the "excess" pruning never fires. The search traverses all 31 nodes of the tree and returns `false`. With the remaining-sum pruning, in the first call $0 + remaining[0] = 21 < 22$. The search ends in a single call. When the target in the same set is 15, on the other hand, the search makes 8 calls in both versions; the pruning does not help in this example. The gain from pruning always depends on the input.

4.9.2 Other problems solved by backtracking

The template can be adapted to a large number of problems. In each adaptation two questions are answered: what choice is made at each level, and when is a branch cut?

Problem	Choice at each level	Pruning (constraint)
Subset sum / knapsack	take the *i*-th item or not? (2 branches)	the sum or the weight limit is exceeded
Traveling salesman (TSP)	the next city	the path is longer than the best known tour
Bin packing	which bin does the item go in?	the bin capacity is exceeded
Maximum coverage	pick the set or not?	no improvement is possible with the remaining sets
Boolean satisfiability (SAT)	is the variable true or false?	a clause has become false
Minesweeper	is the cell a mine?	a contradiction with the revealed numbers
Permutation / combination generation	the next item	the item is already used

Table 4.14: Some problems solved by backtracking.

Some of the problems in the table are decision or counting problems and some are optimization problems. In traveling salesman and knapsack the goal is not *a* solution but the *best* solution. For these problems a second kind of pruning is added next to constraint pruning. The best solution found so far is stored during the search. If the best value a branch can reach is worse than the best known solution, the branch is cut. In traveling salesman, if the length of a partly built tour already exceeds the best known tour, there is no need to complete it. This idea is the basis of the **branch and bound** method.

The SAT row is also important. The foundation of the SAT solvers used today is backtracking too: values are given to the variables in turn, and when a clause becomes false we backtrack. Modern solvers add unit propagation, conflict-driven learning, and smart variable ordering to this skeleton. In this way they can solve industrial formulas with millions of variables.

4.10 Analysis

4.10.1 Complexity

The running time of a backtracking algorithm is the product of two factors:

> *Time = number of nodes visited in the state-space tree × work per node*

The work per node is usually polynomial: a constraint test, a copy, a printout. The number of nodes depends on the shape of the tree. If there are b branches at each level and d levels, the tree has at most b^d leaves. If the number of options shrinks as the level increases (as with permutations), the number of leaves grows like $n!$.

Problem	Worst-case time	Explanation
n-queens	$O(n^2 \cdot n!)$	$O(n!)$ nodes × n columns × `isSafe` at $O(n)$
Sudoku	$O(9^m)$	m: number of empty cells; 9 candidates in each cell
Graph m-coloring	$O(n \cdot m^n)$	n vertices, m colors each; the check is $O(n)$ or $O(degree)$
Hamiltonian path	$O(n \cdot n!)$	the remaining vertices for each position on the path
Word search	$O(R \cdot C \cdot 3^L)$	at most 3 new directions after the first letter
Subsets / permutations	$O(2^n)$ / $O(n \cdot n!)$	number of leaves in the tree

Table 4.15: Worst-case running times of the problems examined in these notes.

These bounds do not take pruning into account. Pruning often shrinks the number of nodes a great deal, but it usually does not change the worst case. Bad inputs can always be constructed on which pruning never acts. For example, a subset sum input in which no subset reaches the target and the sums never exceed the target makes the search traverse the whole tree.

In terms of memory, backtracking is very economical. Only a single partial solution and the frames on the call stack are kept at any time. The depth of the stack equals the depth of the tree: n in n-queens, m in Sudoku, L in word search. So memory is $O(depth)$ plus the size of the partial solution itself. However wide the tree may be, memory does not change. Compared with methods that search breadth-first, this is an important advantage.

Example 4.9

Question. If brute force tested the 8^8 complete placements for 8 queens, and each test compared 28 pairs of queens, how many comparisons would be made? How many `isSafe` calls does backtracking make for the first solution?

Solution. $16{,}777{,}216 \times 28 = 469{,}762{,}048$ pair comparisons, that is, about 470 million operations, would be needed. Backtracking calls `isSafe` 876 times up to the first solution, and each call looks at no more than $3 \cdot 8 = 24$ cells. The upper bound is $876 \times 24 = 21{,}024$ cell reads. The difference is more than four orders of magnitude.

4.10.2 Heuristics that strengthen pruning

The speed of backtracking depends on two things: how much of the tree is pruned and how long each node takes. The techniques seen in these notes shrink these two factors.

- **Variable ordering (MRV).** The variable with the fewest options left is filled first. Contradictions appear early and the branching stays small. In the example Sudoku the number of backtracks falls from 4157 to 0.
- **Value ordering.** Among the values of a variable, the least constraining value is tried first. This is the counterpart in backtracking of the idea behind min-conflicts. Value ordering does not reduce the work when all solutions are searched for, but the first solution is found earlier. Changing the direction order in the maze reduced the number of dead-end cells from 8 to 0.
- **Forward checking.** After an assignment, conflicting values are removed from the domains of neighboring variables. If some variable's domain becomes empty, we backtrack at once. In this way a dead end is noticed before that variable's turn comes.
- **Fast constraint tests.** In n-queens, column and diagonal flags are used; in Sudoku, bit masks for rows, columns, and boxes. The test drops from $O(n)$ to $O(1)$.
- **Symmetry breaking.** Only one of the symmetric solutions is searched for. In n-queens the first queen is tried only in the left half and the solutions found are reflected: for 8 queens the number of placements falls from 2056 to 1028. In graph coloring the color of the first vertex is fixed.
- **Bounding.** In optimization, if the best value a partial solution can reach is worse than the best known solution, the branch is cut. This is the branch and bound method.

These techniques share a lesson: the cost of a search depends less on the size of the space searched than on how the search is organized. A space of size 9^{51} is solved in 51 steps with the right cell order.

4.10.3 What sets backtracking apart?

The place of backtracking is better understood when it is compared with the other algorithm design methods of the course.

Method	Idea	Difference from backtracking
Divide and conquer	Split into independent subproblems, combine	No choosing and trying; each subproblem is solved exactly once
Greedy	Locally best at every step; no turning back	Follows a single branch; a wrong choice cannot be corrected
Dynamic programming	Solve overlapping subproblems once and store them	All options are tried but the results are kept in a table; the number of subproblems must be polynomial
Backtracking	Extend the partial solution, undo it if invalid	Choices are tried and undone; constraint satisfaction, all solutions, decision problems

Table 4.16: Algorithm design methods and how they differ from backtracking.

The comparison with the greedy method is instructive. A greedy algorithm follows a single path from the root to a leaf of the state-space tree. At each node it descends to the child that looks best and never turns back. Backtracking does a depth-first search of the same tree. If a greedy choice is wrong, backtracking notices it and corrects it. The greedy start of min-conflicts shows this: the start is almost correct, but repair is still needed to resolve a single conflict.

The comparison with dynamic programming is also important. Both methods try all the options. The difference is whether the subproblems are stored. In subset sum the same pair (i, `sum`) can arise several times along different paths. For example, in the set {1, 2, 3, 4}, after the first four decisions the choices {1, 4} and {2, 3} give the same sum, 5. Backtracking solves this pair from scratch every time. Dynamic programming stores the result in a table. If the target is small, dynamic programming runs in $O(n \cdot TARGET)$ time. But in problems like n-queens or Hamiltonian path, the state that defines a subproblem is the set of columns used or of vertices visited. The number of such sets is exponential and a table stops being practical.

In short, backtracking is a depth-first search of the state-space tree. The tree is not built in advance; it is generated as it is traversed. Invalid branches are never generated.

4.11 Common mistakes

Backtracking code is short, but small mistakes lead to very different results. Each of the mistakes below is treated in detail somewhere in these notes.

Common mistake

Forgetting to undo. If `b[r][c] = 0` or `seen[r][c] = false` is not done on returning from the recursion, the old choice corrupts the later branches. In n-queens the board fills up with

the queens of earlier attempts, and valid columns are wrongly rejected.

Common mistake

Not propagating the result. If only `solve(...)` is written instead of `if (solve(...)) return true;`, the search continues even after a solution is found, and the board is undone. The program shows an empty board even though it has found a solution.

Common mistake

An early `return false`. Leaving the loop at the first invalid candidate. If `if (!isSafe(...)) return false;` is written, the remaining columns are never tried. With 4 queens, when column 0 is rejected in row 1, the method returns and column 2 is never tried.

Common mistake

Storing the shared list. When collecting all solutions, a copy, `new ArrayList<>(path)`, must be added instead of `results.add(path)`. Otherwise the list fills with references that point to the same object.

Common mistake

An incomplete constraint check. Forgetting to check one of the diagonals in n-queens, or the 3×3 box in Sudoku. The algorithm quickly finds a "solution," but the placements it finds are invalid.

Common mistake

Taking local search to be complete. Min-conflicts cannot prove that there is no solution. Reaching the step limit does not mean "no solution." For a definite "no" answer, a backtracking search that traverses the whole tree is needed.

4.12 Summary

- **Hard problems:** P consists of the decision problems solved in polynomial time, NP of those verified in polynomial time. NP-complete problems are both in NP and NP-hard. If $P \neq NP$, these problems have no polynomial-time exact algorithm; the known exact algorithms take exponential time in the worst case, and the aim is to shrink the search by pruning.

- **The idea:** the solution is built step by step as a partial solution. If a candidate violates a constraint, that branch is abandoned; if no candidate is left at a step, the last decision is undone and the next candidate is tried. No extension of an invalid partial solution is ever generated.
- **The template:** a goal test, then for each candidate: if valid, choose, recurse, unchoose. For a single solution a `boolean` is returned and `true` is passed up; for all solutions, count at the goal and return `false`.
- **State-space tree:** partial solutions are the nodes and choices are the edges. Backtracking is a depth-first search of this tree; the tree is generated as it is traversed, and pruning skips invalid subtrees.
- **n-queens:** queens are placed row by row; the column and the two diagonals are checked. For 4 queens the first solution is [1, 3, 0, 2] (8 placements, 4 backtracks); for 8 queens there are 92 solutions, and the first is [0, 4, 7, 5, 2, 6, 1, 3] (113 placements, 105 backtracks). With flags ($r + c$, $r - c + n - 1$) the safety test becomes $O(1)$.
- **Min-conflicts:** start from a complete placement and move a conflicted queen to the row with the fewest conflicts. It is fast but not complete: it can wander on plateaus and cannot prove that no solution exists.
- **Sudoku:** 1–9 are tried in an empty cell; the row, column, and box are checked. In row-by-row order the example puzzle needs 4208 placements and 4157 undos; with MRV, 51 placements and 0 undos.
- **Other applications:** graph coloring (color constraint, $O(n \cdot m^n)$), Hamiltonian path and cycle (adjacency and repetition constraints, $O(n \cdot n!)$), word search (undoing the visited mark, $O(R \cdot C \cdot 3^L)$), subset sum (a binary take/skip tree, pruning of excess and insufficient sums).
- **Cost:** time is the product of the number of nodes visited and the work per node; the worst case is exponential (2^n, $n!$, 9^m). Memory is only $O(depth)$. Variable and value ordering, forward checking, fast constraint tests, and symmetry breaking shrink the tree that is visited.

4.13 Key terms

backtracking partial solution pruning state-space tree brute force depth-first search (DFS)
decision problem polynomial time reduction NP-hard NP-complete N-queens problem
min-conflicts heuristic local search heuristic constraint satisfaction problem
minimum remaining values (MRV) forward checking graph coloring Hamiltonian path subset sum
lexicographic order branch and bound

4.14 Test Yourself

1. The permutation code is called as `permute(new int[4], new boolean[5], 0)`. Which is the 7th

permutation printed? How many times is “choose” done during the search?

Answer

The permutations come out in lexicographic order: [1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], and the 7th is [2, 1, 3, 4]. The number of chooses is the number of non-root nodes of the tree: $4 + 12 + 24 + 24 = 64$.

2. A search generates the subsets of a four-element set by “take / skip” decisions. How many nodes (counting the root) and how many leaves does its state-space tree have?

Answer

The number of nodes doubles at each level: $1 + 2 + 4 + 8 + 16 = 31$ nodes. The number of leaves is $2^4 = 16$; each leaf is a subset.

3. For $n = 4$ in the flag version, the queens are at squares (0, 1) and (1, 3). Which cells of `cols`, `d1`, and `d2` are `true`? Are the squares (2, 0) and (2, 2) safe?

Answer

`cols[1]`, `cols[3]`; `d1[1]`, `d1[4]`; `d2[0 - 1 + 3] = d2[2]`, `d2[1 - 3 + 3] = d2[1]`. For (2, 0), `cols[0]`, `d1[2]`, and `d2[5]` are empty: safe. For (2, 2), `d1[4]` is occupied: it is on the same anti-diagonal as (1, 3), so it is not safe.

4. For $n = 6$, `solve(b, 0)` places 31 queens before it finds the first solution. How many undos are made? Why does the first solution not start with column 0?

Answer

Every placement is either undone or stays in the solution; the solution has 6 queens. $31 - 6 = 25$ undos are made. None of the four solutions of the 6-queens problem uses column 0 in row 0. All the branches starting with column 0 end in dead ends, and the lexicographically first solution is [1, 3, 5, 0, 2, 4].

5. On a 4 × 4 board, `queen = [1, 3, 0, 0]` (`queen[k]` is the row of the queen in column k). Column 2 is chosen for repair. What are the conflict counts of this queen for rows 0–3? Does

moving it to the row with the smallest value solve the problem?

Answer

The counts are 1, 2, 1, 2. Row 0 is the queen's current position (the same row as column 3). Row 2 also has 1 conflict: it is on the same diagonal as (3, 1). If the queen is moved to row 2, `queen = [1, 3, 2, 0]` and this time columns 1 and 2 conflict; the number of conflicting pairs is again 1. This is a plateau. Had column 3 been chosen, the counts would have been 1, 3, 0, 1, and moving the queen to row 2 would have given the solution [1, 3, 0, 2].

6. In the example Sudoku puzzle, which rows and columns does the box of cell (4, 7) cover? What are the candidates of this cell at the start?

Answer

4 / 3 · 3 = 3, *7 / 3 · 3 = 6*: the box is rows 3–5 and columns 6–8. Row 4 contains 4, 8, 3, 1; column 7 contains 6, 8, 7; the box contains 3, 1, 6. The unused digits are 2, 5, and 9. In the solution, 9 is written in the cell.

7. The graph with edges 0–1, 1–2, 2–0, and 2–3 is colored with `colorGraph(0)` using $m = 2$ and $m = 3$ colors. What are the results?

Answer

Vertices 0, 1, 2 form a triangle; all three are adjacent to each other and two colors are not enough. For $m = 2$ the search traverses the whole tree and returns `false`. For $m = 3$ the first attempt finds [1, 2, 3, 1]: the only neighbor of vertex 3 is 2 (color 3), so color 1 is suitable.

8. In a star graph with vertex 0 at the center and vertices 1, 2, and 3 connected only to the center, what does `find(g, path, 1)` return? Would another starting vertex change the result?

Answer

`false`. When we go from 0 to 1, vertex 1 has no neighbor that is not on the path; the same happens for 2 and 3. Changing the starting vertex does not help either: a path has at most two end vertices, but this graph has three vertices of degree 1. There is no Hamiltonian path.

9. Do the words “ASFB” and “ECE” occur in the grid ABCE / SFCS / ADEE?

Answer

“ASFB” does occur: (0, 0) A → (1, 0) S → (1, 1) F → (0, 1) B. “ECE” does not. The letter C is at (0, 2) and (1, 2). The only E neighbor of (0, 2) is (0, 3); it was used at the start of the path. The only E neighbor of (1, 2) is (2, 2); if the path starts there, no other neighbor is left for the second E. In both cases an E cell would have to be used twice.

10. For `a = {2, 4, 5, 9}` and target 13, which subset does the subset sum search (taking first) find? How does the number of calls change if remaining-sum pruning is added?

    Answer

    All the branches starting with 2 fail: 2 + 4 + 5 = 11, and adding 9 gives 20; 2 + 4 + 9 = 15; 2 + 5 + 9 = 16; 2 + 9 = 11. Then 2 is skipped and 4 is taken. Adding 9 to 4 + 5 = 9 gives 18 and the branch is pruned. When 5 is skipped and 9 is taken, 4 + 9 = 13 is found. The result is {4, 9}. The code without pruning makes 23 calls, and the code with remaining-sum pruning makes 21; the gain is small, because most sums exceed the target and are pruned anyway.

4.15 Further reading

- A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed., chapter 12: presents backtracking and branch and bound within the same framework. It works through the n-queens, Hamiltonian cycle, and subset sum examples.
- S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., chapter 6: treats constraint satisfaction problems, MRV, forward checking, and the min-conflicts heuristic in detail.
- D. E. Knuth, *The Art of Computer Programming*, Volume 4B: gives the history of backtrack programming, methods for estimating its cost, and the "dancing links" technique.
- S. W. Golomb and L. D. Baumert, "Backtrack Programming" (1965): one of the first papers to describe the method as a general programming technique.

4.16 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed., Pearson, 2012 (chapter 12: Backtracking).
3. S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., 2020 (chapter 6: constraint satisfaction, min-conflicts).
4. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, "Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems", *Artificial Intelligence* 58, 1992.
5. D. E. Knuth, *The Art of Computer Programming*, Volume 4B, Addison-Wesley, 2022 (Backtrack programming, Dancing Links).
6. S. W. Golomb and L. D. Baumert, "Backtrack Programming", *Journal of the ACM* 12(4), 516–524, 1965.
7. S. A. Cook, "The Complexity of Theorem-Proving Procedures", *Proceedings of the 3rd ACM Symposium on Theory of Computing (STOC)*, 151–158, 1971.

Branch and Bound

In many optimization problems the number of possible solutions grows exponentially or factorially with the input. Trying every candidate one by one becomes impossible even with a few dozen items. Branch and bound shortens this search intelligently. Solutions are built step by step on a tree. For each branch, an estimate is made in advance of how good that branch can possibly be. If the estimate cannot beat the best solution found so far, the branch is never entered. These notes first present the concepts of the method, its general scheme, how the bound is chosen, and the three search strategies. They then apply the method step by step to five problems: the 0/1 knapsack, the traveling salesman, assignment, integer linear programming, and maximum clique. In each application we show why the bound is valid, explain the code line by line, and run it with a trace table. The notes end with a comparison of design paradigms.

LEARNING OUTCOMES

Paradigm
Explain the steps of branch and bound (the state-space tree, branching, bounding, and pruning) and state how the method differs from backtracking.

Bound design
Build an upper bound for maximization and a lower bound for minimization; justify why a bound must be valid, that is, optimistic; obtain a bound through relaxation.

Search strategies
Trace the FIFO, LIFO, and least-cost (best-first) strategies on the same example and compare them in terms of the number of nodes generated and memory. Tell the design paradigms apart.

Applications
Solve the 0/1 knapsack, traveling salesman, assignment, integer linear programming, and maximum clique problems step by step with branch and bound.

5.1 The idea of branch and bound

Solving a complex problem is sometimes like finding the right direction at a fork in the road. At each fork there are several options, and each option leads to new forks. The goal is to find the path that finally reaches the best point. Walking every path to its end gives a sure result but takes far too long. Branch and bound proposes a middle way: at each fork, we estimate where we could get to in the best case in that direction. If the estimate is worse than the best result in hand, we never go that way.

5.1.1 Combinatorial optimization and the limits of brute force

The targets of branch and bound are **combinatorial optimization** problems. In these problems a solution is built from a finite number of discrete choices: which items to take, in what order to visit the cities, which person gets which job. The number of possible solutions is finite but very large. What we look for is the solution that maximizes or minimizes an objective function. The method aims at an exact, that is, optimal solution; it is not content with an approximate one.

The simplest solution to such problems is **brute force**: all candidates are generated, each is evaluated, and the best is chosen. The trouble is the growth rate of the number of candidates. In a knapsack problem with n items, each item is either taken or not; there are 2^n subsets. In a traveling salesman problem with n cities, once the starting city is fixed, every ordering of the remaining cities is a separate tour; there are $(n-1)!$ tours. Table 5.1 shows how these two numbers grow.

n	2^n subsets	$(n-1)!$ tours
5	32	24
10	1,024	362,880
20	1,048,576	$\approx 1.2 \cdot 10^{17}$
40	$\approx 1.1 \cdot 10^{12}$	$\approx 2.0 \cdot 10^{46}$

Table 5.1: The number of candidates brute force must try.

A computer that evaluates a billion tours per second would finish all the tours of 20 cities in about $1.2 \cdot 10^8$ seconds, which is more than three years. For 40 cities the time required is far beyond the age of the universe. In the knapsack the growth is slower, but even with 40 items a trillion-scale number of subsets is tried. Most of these problems are **NP-hard**: no known algorithm can solve them in polynomial time in the worst case. Branch and bound does not change the worst case either. What it changes is the number of candidates examined in practice.

The basic idea of the method can be stated in one sentence: how good a branch can possibly be is estimated in advance, and if the estimate is not good enough, that branch is never entered. If this estimate is built correctly, the optimal solution is never lost, but most candidates are eliminated without ever being generated.

5.1.2 Branching, bounding, pruning

Definition 5.1

Branch and bound is an algorithm design method that splits the solution space of an optimization problem into subproblems in the form of a tree. For each subproblem, the best value that can be reached is estimated. Subproblems whose estimate is no better than the best solution found so far are eliminated without being examined.

The name of the method comes from its two basic operations. It can be summarized in four steps:

1. **Branching:** The problem is split into smaller subproblems. In the knapsack, the question "should item 1 be taken?" produces two subproblems: the cases in which item 1 is taken and in which it is not.
2. **Bounding:** For each subproblem, the best value that can be reached is estimated. This estimate is called the **bound**.
3. **Pruning:** Subproblems whose bound is no better than the best solution found so far are eliminated. No solution below an eliminated subproblem is generated.
4. **Repeat:** The process continues for all the remaining subproblems. As unnecessary branches are pruned, the search gets shorter.

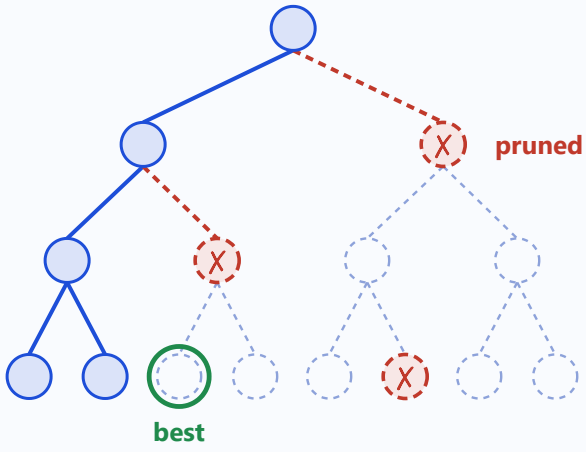


Figure 5.1: A tree that branches and is pruned. The nodes marked with a red cross have an insufficient bound; these nodes are pruned. The subtree of a pruned node (the faint dashed lines) is never generated. The green ring shows the best solution.

Figure 5.1 shows the method on a tree. We start at the root and descend by making a decision at each node. The node at the upper right has been pruned. The seven nodes below it are drawn faintly, because the algorithm never generates them. This saving grows exponentially as the depth of the tree increases. A node pruned just below the root eliminates almost half of the tree in one stroke.

The method is used in a wide range of areas. The traveling salesman, knapsack, scheduling, bin packing, and integer programming are among the first. In industry, branch and bound is also at the core of commercial optimization software that solves problems such as production planning, airline crew assignment, and logistics routing.

5.1.3 Basic concepts

Branch and bound algorithms work on a tree, and the nodes of this tree can be in various states. Putting the names of the concepts down at the start makes the later traces easier to read.

Definition 5.2

The **state-space tree** is a tree that shows all the partial and complete solutions of a problem. The root is the state in which no decision has been made. Each level of the tree corresponds to a decision; for example, at level i the question "should item i be taken?" is answered. The leaves are the complete solutions, in which all the decisions have been made.

The state-space tree is not built in full in the program. The algorithm generates nodes as needed and forgets the nodes it has finished with. It is therefore necessary to name the states of the nodes:

- **Branching and the E-node.** Generating the children of a node, that is, its subproblems, is called branching. The node whose children are being generated at that moment is called the **E-node** (expanding node).
- **Bound.** An optimistic value that no solution below the node can exceed. In maximization it is an upper bound; in minimization, a lower bound.
- **Live node.** A node that has been generated but not yet expanded and still looks promising. All the live nodes wait in a queue-like structure. This structure can be an ordinary queue, a stack, or a priority queue.
- **Dead node.** A node that has been expanded, found infeasible, or pruned because its bound is insufficient. A dead node is never considered again.
- **Incumbent.** The value of the best complete solution found so far. Pruning decisions are made against this value. As the search proceeds, `best` only improves: it increases in maximization and decreases in minimization.

A solution is **feasible** if it satisfies all the constraints of the problem. A node that violates one of the constraints is called an infeasible node. For example, a knapsack whose weight exceeds the capacity is infeasible.

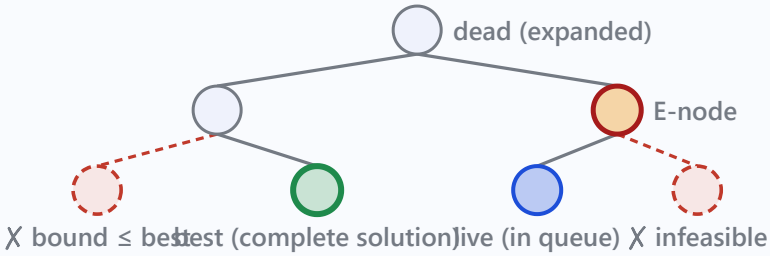


Figure 5.2: Node types in a state-space tree. The dead node at the top has been expanded. The orange node is the E-node being expanded at that moment. The blue node is a live node waiting in the queue. The green node is a complete solution and determines `best`. The red dashed nodes have been pruned: one because its bound cannot beat `best`, the other because it is infeasible.

Figure 5.2 shows these types in a single tree. The life cycle of a node is as follows. The node is generated and its bound is computed. If the bound is promising, it enters the queue as a live node; if not, it is pruned at once. When its turn comes it leaves the queue and becomes the E-node. After its children have been generated, it turns into a dead node.

5.2 The paradigm

All branch and bound algorithms follow the same skeleton. Three things change from problem to problem: how a node is represented, how the children are generated, and how the bound is computed. This section first gives the skeleton and then discusses the choice of the bound, the pruning rules, and the search strategy.

5.2.1 The general scheme

The code below is the branch and bound skeleton for a maximization problem. It is pseudocode close to Java: `queue`, `bound()`, `isFeasible()`, `isComplete()`, and `value()` take on a different meaning in each problem.

```
int best = -∞;
queue.add(root);
while (!queue.isEmpty()) {
  Node u = queue.remove();   // strategy
  if (bound(u) <= best) continue;
  for (Node c : u.children()) {
    if (!isFeasible(c)) continue;
    if (isComplete(c) && value(c) > best)
      best = value(c);      // new incumbent
    if (bound(c) > best)
      queue.add(c);         // live node
  }
}
return best;
```

The code is a single loop. Each pass of the loop has five steps:

1. **Select:** A live node is removed from the queue. Which node comes out is decided by the search strategy: first in (FIFO), last in (LIFO), or the one with the best bound.
2. **Test again:** The line `if (bound(u) <= best) continue;` tests the node once more. The node looked promising when it entered the queue. But while it waited, `best` may have grown. If its bound is now insufficient, the node is discarded without ever being expanded.
3. **Branch:** The `for` loop generates the children of the node. Infeasible children are eliminated at once by `continue`.
4. **Update:** If a child is a complete solution and its value is greater than `best`, `best` is raised. This solution is the new incumbent.
5. **Bound:** Only children whose bound can beat `best` are put in the queue. The others are pruned.

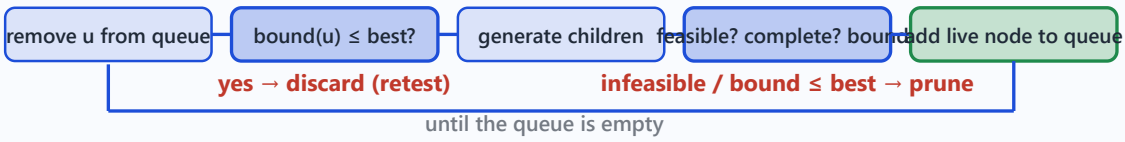


Figure 5.3: The branch and bound loop. A node is removed from the queue and its bound is tested again; if it is insufficient, it is discarded. Children are generated; those that are infeasible or whose bound is insufficient are pruned, and the rest are added to the queue. The loop continues until the queue is empty.

The loop stops when the queue is empty. At that moment `best` is the value of the optimal solution. The reason for this claim is simple. Every node has either been expanded or pruned. The solutions below pruned nodes cannot beat `best`, because the bound guarantees it. The solutions below expanded nodes have been examined one by one. So no solution better than `best` has been skipped.

The code is written for maximization. In **minimization** the directions are reversed. The bound is a lower bound, `best` starts at $+\infty$, and a node with $lower(c) \geq best$ is pruned. The traveling salesman and assignment problems are solved in this form.

The skeleton stores only the value of the solution, not the solution itself. If we also want to know which items were taken or in what order the tour visits the cities, each node carries its own partial solution. A `boolean[]` array or a bit mask is enough for this. When `best` is updated, this information is copied as well.

5.2.2 How is the bound chosen?

The whole success of branch and bound depends on the bound function. A good bound has four properties. One of them is mandatory; the other three concern efficiency.

> **Definition 5.3**
>
> In a maximization problem, the bound of a node u is **valid** if it is greater than or equal to the value of every feasible solution in the subtree of u: $bound(u) \geq value(x)$ for every x. In minimization the condition is reversed: $lower(u) \leq value(x)$. A valid bound is also called an **optimistic** bound, because it assumes the branch looks better than it can actually be.

- **It must be valid (optimistic).** This property is mandatory. If the bound falls below the true value (in maximization), the branch containing the optimum may be pruned as "not good enough." The algorithm then returns a wrong result without raising any error.
- **It must be tight.** The closer the bound is to the true value, the more branches are pruned. An infinitely large "bound" is valid but prunes nothing; the algorithm degenerates into brute force.
- **It must be cheap to compute.** The bound is computed at every node. A very expensive bound can cost more time than the nodes it prunes.
- **It is built by relaxation.** The general way to obtain a bound is **relaxation**. One of the hard constraints of the problem is removed and the easy problem that remains is solved. When a constraint is removed, the set of feasible solutions grows. So the optimal value of the relaxed problem cannot be worse than the optimal value of the original problem; it is a valid bound.

Three examples of relaxation are treated in detail in these notes. In the knapsack, the constraint "items cannot be split" is removed and the fractional knapsack is solved. In integer programming, the constraint "the variables must be integers" is removed and an ordinary linear program is solved. In the traveling salesman problem, the constraint "the chosen edges must form a single tour" is removed and each city is left by its cheapest edge.

There is a trade-off between tightness and cheapness. In minimization, the lower bound is the smallest value that can be obtained in the best scenario. A weak lower bound prunes almost nothing and lengthens the search. A very strong lower bound requires an expensive computation at every

node. The choice is made according to the balance between the number of nodes pruned and the work done per node.

> **Common mistake**
>
> Using a pessimistic value as the bound. In the knapsack, adding up the profits of just the items that fit completely seems plausible. But this value is the profit of a feasible solution, that is, a lower bound; it is not an upper bound. Let the items be (20, 2), (30, 5), (35, 7), (12, 3) and the capacity 9. At the root the items that fit completely give 20 + 30 = 50, whereas the optimum is 55 (the first and third items). The moment `best` is 50, this "bound" cuts the branch of the optimum and the algorithm returns 50.

5.2.3 When is a node pruned?

In branch and bound algorithms a node is pruned for three reasons. All three rules answer the same question: could there be a solution better than `best` below this node?

1. **Infeasibility.** If the node violates a constraint, there is no valid solution below it. Examples: the weight has exceeded the capacity, a city has been visited a second time, the linear program has no solution.
2. **Bound.** If *bound(u) ≤ best* in maximization, or *lower(u) ≥ best* in minimization, the solutions below cannot beat the best. The node is pruned even if it is feasible.
3. **Completeness.** If the node is a complete solution (a leaf), or if the solution of the relaxation is already feasible, there is no need to branch. For example, in integer programming, if the solution of the linear program turns out to be integral, there is no better integer solution below that node. Its value is compared with `best` and the node is closed.

Note the case of equality. A node with *bound(u) = best* is pruned too. Below it there is at most a solution as good as `best`, and a solution of that value is already in hand. If not only the value of the best solution but all the best solutions are wanted, nodes are not pruned on equality; that is a separate problem.

Backtracking uses only the first rule and completeness. What sets branch and bound apart from it is the second rule: even if the node is feasible, it is cut if it is "not good enough." So adding a single bound line to a search written as backtracking turns it into branch and bound.

5.2.4 Search strategies

In the general scheme we said "a live node is removed from the queue." Which live node comes out depends on the type of the queue structure. There are three common strategies (Table 5.2).

Strategy	Queue structure	How it traverses	Pros / cons
FIFO branch and bound	queue (`ArrayDeque`)	level by level, breadth-first	finds shallow solutions early; the number of live nodes can grow exponentially
LIFO branch and bound	stack (or recursion)	depth-first	memory proportional to the depth; an early complete solution raises `best` quickly
Least cost (LC, best-first)	priority queue (`PriorityQueue`)	the node with the most promising bound first	usually expands the fewest nodes; the queue consumes memory, and each operation costs *O(log k)*

Table 5.2: Strategies for selecting live nodes.

In the **FIFO** (first in, first out) strategy the live nodes wait in an ordinary queue. The tree is traversed level by level; this is breadth-first search. The children of the root are expanded, then the grandchildren, then their children. Solutions at shallow levels are found early. But before descending to the deep levels, all the shallow nodes accumulate in the queue. In a tree with branching factor 2, level k contains 2^k nodes; if pruning is weak, the queue grows exponentially.

In the **LIFO** (last in, first out) strategy the live nodes wait on a stack, or the search is written directly with recursion. The tree is traversed depth-first. The stack holds only the not-yet-tried siblings of the nodes on the current path. So memory is bounded by the product of the depth and the branching factor. Another benefit is that it reaches a leaf quickly. A complete solution found early gives `best` a value, and pruning starts in the very first steps.

In the **least cost** (LC) strategy the live nodes wait in a priority queue. At each step the node with the most promising bound is expanded: in maximization the node with the largest bound, in minimization the one with the smallest. This strategy is also called **best-first search**. Because the most promising branch is looked at first, a good solution is found early and most of the remaining nodes are pruned. The price is the memory consumption of the priority queue and the *O(log k)* cost of each insertion and removal, where k is the number of nodes in the queue.

As long as the bound is valid, all three strategies find the same optimal value. The strategy changes only the amount of work, that is, the number of nodes generated and expanded, and the memory. These differences will be seen with numbers in the knapsack example.

5.2.5 Least-cost search and cost estimation

In the general form of LC search, each node is assigned an estimated cost:

$$\hat{c}(x) = f(x) + \hat{g}(x)$$

Here $f(x)$ is the cost already paid from the root to x; for example, the depth of x or the length of the path traveled so far. $\hat{g}(x)$ is an estimate of the remaining cost from x to a solution. Of the live nodes, the one with the smallest $\hat{c}$ is expanded.

The two extreme choices give the other strategies. If $\hat{g} = 0$, only the cost paid is considered; this is uniform-cost search. If the steps have unit cost, that is, if f is the depth of the node, the shallowest node is always chosen and the search is FIFO, that is, breadth-first. If $f = 0$, only the estimate of the remaining cost is considered; this is greedy best-first search, which chooses the node that looks closest to the goal. The sum of the two weighs the cost paid and the estimate of what remains together.

A good property of the estimate is that it is never greater than the true remaining cost. Such an estimate is called an **admissible heuristic**. This property corresponds to the valid bound in branch and bound: the remaining cost is shown as less than it is, that is, the estimate is optimistic.

The classic example is the **8-puzzle**. In a 3 × 3 frame there are tiles numbered 1 to 8 and one blank. In one move, a tile adjacent to the blank slides into the blank. The goal is to bring the tiles to the arrangement 1 2 3 / 4 5 6 / 7 8, with the blank in the lower-right corner. In this problem $f(x)$ is the number of moves made so far and $\hat{g}(x)$ is the number of tiles not in place. Every misplaced tile needs at least one move. So $\hat{g}$ never overestimates the number of remaining moves.

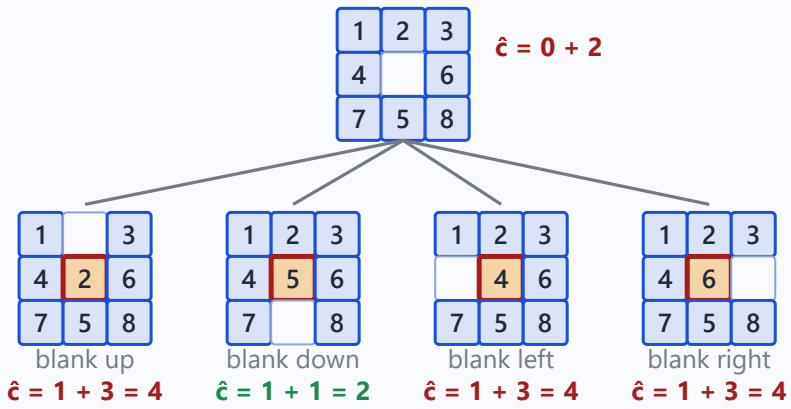


Figure 5.4: LC search on the 8-puzzle. In the initial state 5 and 8 are out of place: $\hat{c} = 0 + 2$. Of the four children, "blank down" has the smallest value, $\hat{c} = 1 + 1 = 2$; the other three have $\hat{c}$ equal to 4. Orange: the tile that moves.

Figure 5.4 shows the first step of the search. The initial state is 1 2 3 / 4 □ 6 / 7 5 8 (□ is the blank). Only 5 and 8 are out of place; the $\hat{c}$ value of the root is $0 + 2 = 2$. When the root is expanded, four children arise:

- **Blank up:** 2 moves down; 1 □ 3 / 4 2 6 / 7 5 8. Misplaced tiles 2, 5, 8: $\hat{c} = 1 + 3 = 4$.
- **Blank down:** 5 moves up; 1 2 3 / 4 5 6 / 7 □ 8. Only 8 is misplaced: $\hat{c} = 1 + 1 = 2$.
- **Blank left:** 4 slides right; 1 2 3 / □ 4 6 / 7 5 8. Misplaced tiles 4, 5, 8: $\hat{c} = 4$.
- **Blank right:** 6 slides left; 1 2 3 / 4 6 □ / 7 5 8. Misplaced tiles 6, 5, 8: $\hat{c} = 4$.

The child with the smallest $\hat{c}$ is "blank down," and it is the one expanded. Of its children, "blank right" slides the 8 to the left and reaches the goal: $\hat{c} = 2 + 0 = 2$. The solution takes 2 moves and only 2 expansions were made. FIFO, by contrast, does not look at the $\hat{c}$ values. It expands the nodes of the

first level in the order they were generated and wastes time on a hopeless branch such as “blank up.” The **A* algorithm** of path finding is the counterpart of this idea on weighted graphs; the relation is discussed separately at the end of the notes.

5.2.6 Comparison with backtracking

Branch and bound and **backtracking** traverse the same tree and their code is very similar. Both extend partial solutions and cut hopeless branches. They differ in what they look for and what they cut by (Table 5.3).

Property	Backtracking	Branch and bound
Problem type	decision and search: one or all valid solutions (n-queens, Sudoku)	optimization: the best solution (largest profit, shortest tour)
Pruning criterion	feasibility: is a constraint violated?	feasibility + bound: can it beat `best`?
Traversal	always depth-first (recursion)	depth-first, breadth-first, or best-first
Extra information	only the partial solution	partial solution + bound function + `best`
Memory	*O(depth)*	*O(depth)* in the depth-first version; the size of the queue in FIFO and LC
Worst case	exponential	still exponential; far fewer nodes in practice

Table 5.3: Backtracking and branch and bound.

When solving the n-queens problem, backtracking checks whether one queen attacks another. If there is an attack, it cuts the branch. All solutions or one solution are sought; there is no ordering of “better” among the solutions. Branch and bound is meaningful when there is an ordering among solutions. Even if a branch violates no constraint, it is cut if the best solution that can come out of it is worse than the one in hand.

The code counterpart of this difference is a single line. A search written for the traveling salesman or maximum clique that contains no bound test is really backtracking; it traverses the whole tree. When a single bound line such as `if (lowerBound(...) < best)` is added, the search turns into branch and bound.

5.2.7 Advantages and disadvantages

The strengths of the method are the following:

- It finds the exact optimal solution of complex optimization problems. Unlike approximate methods, the optimality of the result is proven.
- It narrows the search space and eliminates unnecessary possibilities. With a good bound, a very small fraction of the tree is traversed.
- It is a natural framework for problems that require finding the best route or selection.

- It applies to a wide variety of problems whose solutions can be represented as a tree.
- It can be stopped early. If the search is cut off midway, a solution (`best`) and the best bound in the queue are in hand. The difference between the two tells how far the solution in hand can be from the optimum at most. This information is a quality guarantee.

It has weaknesses as well:

- In the worst case the time is still exponential. In cases with a great many possibilities where the bound does not prune well, the time becomes unacceptably long.
- In the FIFO and best-first strategies, memory too can grow exponentially.
- Success depends on a good bound estimate; a weak bound prunes almost nothing.
- A bound and a branching rule specific to each problem must be designed. The implementation is more complex than backtracking.

5.3 The 0/1 knapsack

To see all the parts of the method, the 0/1 knapsack problem is chosen as the first application. The problem is familiar, its state-space tree is binary, and its bound is easy to compute. The same example will be run with three different strategies and the results compared.

5.3.1 The problem and its state-space tree

A brief reminder of the problem is enough. There are n items; item i has profit p_i and weight w_i. The capacity of the knapsack is W. A decision is made for each item: $x_i \in \{0, 1\}$. An item is either taken whole (1) or not taken at all (0); it cannot be split. The goal is to make the decisions so that

> $\Sigma\, p_i\, x_i$ is maximal subject to $\Sigma\, w_i\, x_i \le W$

These notes do not use the dynamic programming solution of the knapsack. Dynamic programming builds a table that depends on the capacity W and takes $O(nW)$. If W is very large or the weights are not integers, this table cannot be built; branch and bound is not affected by the size of the weights.

For branch and bound, the items are first sorted in decreasing order of **profit-to-weight ratio**. This order is needed for the bound computation to be valid. The state-space tree is binary. The root is the state in which no decision has been made. At level i the decision for item i is made: the left branch is "item i taken" (1), the right branch "not taken" (0). In a problem with four items the tree has five levels, and the full tree contains $2^5 - 1 = 31$ nodes. Of these, 16 are leaves; each leaf corresponds to a subset.

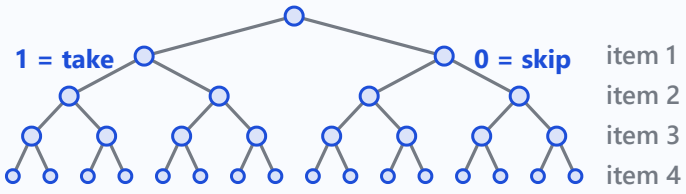


Figure 5.5: The state-space tree of a four-item knapsack. At each level the decision for one item is made; the left branch 1 = take, the right branch 0 = skip. The full tree has 31 nodes and 16 leaves.

The example used in these notes is given in Table 5.4. The capacity is $W = 16$. The items are already in decreasing order of ratio: 20, 6, 5, 2.

Item	p_i	w_i	p_i/w_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

Table 5.4: The knapsack example, $W = 16$.

The optimum is to take items 1 and 3: profit $40 + 50 = 90$, weight $2 + 10 = 12$. Brute force finds this result by trying all 16 subsets. Branch and bound reaches the same result without generating all 31 nodes. How few nodes are generated depends on the strategy used.

5.3.2 Upper bound: the fractional knapsack relaxation

The knapsack is a maximization problem; an upper bound is needed for every node. This bound is built by relaxation. When the constraint “items cannot be split” is removed, the problem becomes the **fractional knapsack**. In the fractional knapsack, any desired fraction of an item can be taken and the same fraction of its profit is obtained. The greedy method solves this problem optimally: the items are taken whole in ratio order as long as they fit; of the first item that does not fit, the fraction that fits is taken, and the knapsack is full.

The bound of a node is computed with the same idea. At node u the decisions already made are fixed: let the profit of u be k, its weight a, and its level i. The items not yet decided, $i + 1, i + 2, \ldots$, are added whole in ratio order as long as they fit. Let the first item that does not fit be m; the fraction of it that fits is added:

$$bound(u) = k + \Sigma_{j=i+1}^{m-1} p_j + (W - a - \Sigma_{j=i+1}^{m-1} w_j) \cdot p_m / w_m$$

If all the remaining items fit, there is no m and the last term is not added. If the node itself is infeasible, that is, $a > W$, the bound is taken to be 0. This guarantees that the node can never beat `best` and will not enter the queue.

Theorem 5.1 (The fractional bound is valid)

If the items are in decreasing order of p_i/w_i, then for every node u, *bound(u)* is greater than or equal to the profit of every feasible 0/1 solution in the subtree of u.

Proof. Any feasible 0/1 solution in the subtree of u keeps the decisions made at u and chooses a subset of the remaining items. This choice is also a solution of the fractional problem: the fraction of every item is 0 or 1. So the profit of the 0/1 solution cannot exceed the optimal value of the fractional problem built for the remaining items and the remaining capacity $W - a$. Because the items are in ratio order, the greedy method gives the optimal value of the fractional problem: every unit of remaining capacity is spent on the item that gives the highest profit per unit among the remaining items. Adding k to this value gives *bound(u)*. ■

The proof has two supports: every 0/1 solution is a fractional solution, and the greedy method is optimal for the fractional problem. The second support depends on the ratio order. If the items are not in ratio order, the greedy method does not find the optimum even for the fractional problem. In that case the computed value can fall below the true optimum and the bound loses its validity.

The bound of the root is computed in the example as follows. Items 1 and 2 fit completely: *40 + 30 = 70*, weight 7. Item 3 has weight 10 and does not fit into the remaining 9 units. *9/10* of item 3 is taken: *9 · 50/10 = 45*. So *bound(root) = 70 + 45 = 115*. The meaning of this number is: whichever subset of the four items is chosen, the profit cannot exceed 115. The true optimum is 90; the bound is optimistic but valid.

Common mistake

Forgetting to sort the items by ratio. Suppose the items are given in the order (10, 5), (40, 2), (30, 5), (50, 10) and the bound is computed in that order. At the root the first three items fit completely (weight 12, profit 80); for the remaining 4 units, *4/10* of item 4 is taken: *80 + 20 = 100*. This number is smaller than 115 and does not reflect the true fractional optimum. In this example 100 is still greater than 90, but in another example the unsorted computation can fall below the optimum and the right branch is pruned.

5.3.3 Code trace: bound(root)

The Java counterpart of the bound function is below. Arrays start at zero: `p[0]` and `w[0]` belong to item 1. `n` is the number of items and `W` the capacity. The class `Node` carries the fields `level`, `profit`, `weight`, and `bound`. The level of the root is −1; no item has been decided yet.

```
double bound(Node u) {
  if (u.weight > W) return 0;   // infeasible
  double result = u.profit;
  int totalW = u.weight;
  int j = u.level + 1;
  while (j < n && totalW + w[j] <= W) {
    totalW += w[j];
    result += p[j];
    j++;
  }
  if (j < n)                // fractional part
    result += (W - totalW) * (double) p[j] / w[j];
  return result;
}
```

The lines do the following:

- `if (u.weight > W) return 0;` catches an infeasible node. If the weight has exceeded the capacity, there is no feasible solution below the node; the bound of 0 keeps it out of the queue.
- `result` starts with the node's own profit and `totalW` with its own weight. `j` points to the first undecided item.
- The `while` loop adds items whole as long as they fit. The condition has two parts: there must be an item left (`j < n`) and item *j* must fit completely into the remaining capacity.
- When the loop ends, either no items are left or `j` is the first item that does not fit. In the second case a fractional profit of `(W - totalW) * (double) p[j] / w[j]` is added. The `(double)` cast prevents integer division.

For one node the function looks at no more than *n* items; its time is *O(n)*. It uses no extra memory. Table 5.5 is the trace of the call for the root.

Step	Test	`j`	`totalW`	`result`
Entry	*0 ≤ 16*: feasible	0	0	0
Loop 1	*0 + w[0] = 2 ≤ 16*: item 1 fits whole	1	2	40
Loop 2	*2 + w[1] = 7 ≤ 16*: item 2 fits whole	2	7	70
Loop end	*7 + w[2] = 17 > 16*: item 3 does not fit	2	7	70
Fraction	*j = 2 < 4*: *(16 − 7) · 50/10 = 45*	2	7	115

Table 5.5: Trace of `bound(root)` (*W = 16*; root: level −1, profit 0, weight 0).

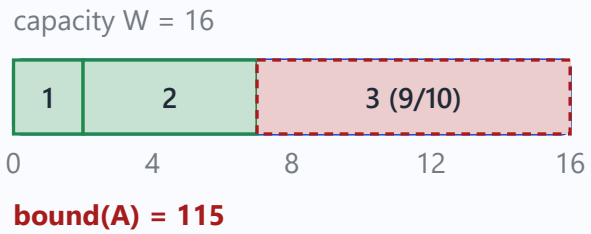


Figure 5.6: The bound of the root as a capacity bar. Item 1 (2 units) and item 2 (5 units) go in whole; the remaining 9 units are filled with *9/10* of item 3. *bound(A) = 115*.

Figure 5.6 shows the same computation on a bar. The green part of the bar is the items taken whole, and the hatched part is the fractional piece. The fractional piece cannot be taken in the 0/1 problem; this is why the bound is larger than the true value.

Example 5.1

Let the items (p, w) be (20, 2), (30, 5), (35, 7), (12, 3) and $W = 9$. The ratios are 10, 6, 5, 4; the items are in order.

1. **Root.** Items 1 and 2 fit whole: *20 + 30 = 50*, weight 7. Item 3 does not fit (*7 + 7 = 14 > 9*); for the remaining 2 units, *2 · 35/7 = 10*. *bound(root) = 60*.
2. **The node "item 1 skipped"** (level 0, profit 0, weight 0). Item 2 fits whole: 30, weight 5. Item 3 does not fit; for the remaining 4 units, *4 · 35/7 = 20*. The bound is 50.
3. **The node "item 1 taken, item 2 skipped"** (level 1, profit 20, weight 2). Item 3 fits whole: *20 + 35 = 55*, weight 9. The capacity is full; the fraction for item 4 is 0. The bound is 55.

The optimum is items 1 and 3: profit 55, weight 9. Greedy selection in ratio order takes 1 and 2 and stops at 50. After the optimum is found, the bound of the node "item 1 skipped" satisfies *50 ≤ 55* and the node is pruned: no solution without item 1 can exceed 55.

5.3.4 Best-first branch and bound, step by step

Now the main example ($W = 16$) is solved with the best-first strategy. Each node is named with a letter and is given with its "profit / weight" pair and its bound (s). The queue is a priority queue; at each step the node with the largest bound comes out. When a child is generated, if it is feasible and its profit is greater than `best`, `best` is updated at once. In the knapsack every feasible node is a valid solution made up of the items taken; there is no need to descend to a leaf to update.

1. **Start.** Root A: no item chosen; 0 / 0, *s = 115*. `best` = 0. Queue: A (115).
2. **A is removed and branched.** B (item 1 taken): 40 / 2, feasible; `best` = 40. The bound of B is the same as that of the root, because the bound of the root already took item 1 whole: *s = 115*. C (no item 1): 0 / 0. In the bound computation, items 2 and 3 fit whole: *30 + 50 = 80*, weight *5 + 10 = 15*.

Item 4 does not fit; for the remaining 1 unit, *1 · 10/5 = 2* is added. *s = 82*. Queue: B (115), C (82).

3. **B is removed and branched.** D (item 2 taken): 70 / 7; `best` = 70, *s = 115*. E (no item 2): 40 / 2; item 3 fits whole (90, weight 12), *4/5* of item 4 is added (8): *s = 98*. Queue: D (115), E (98), C (82).
4. **D is removed and branched.** F (item 3 taken): weight *7 + 10 = 17 > 16*; infeasible, its bound is 0, pruned. G (no item 3): 70 / 7; item 4 fits whole (80, weight 12), no more items: *s = 80*. Because *80 > 70*, G enters the queue. Queue: E (98), C (82), G (80).
5. **E is removed and branched.** H (item 3 taken): 90 / 12, feasible; `best` = 90. The bound of H is *90 + 4 · 2 = 98* and it enters the queue. I (no item 3): 40 / 2; item 4 fits whole: *s = 50 ≤ 90*, pruned. Queue: H (98), C (82), G (80).
6. **H is removed and branched.** J (item 4 taken): weight 17; infeasible. K (no item 4): 90 / 12, the last item; *s = 90 ≤ 90*, pruned. Queue: C (82), G (80).
7. **C and G are discarded.** C is removed: *82 ≤ 90*, discarded at the second test. G is removed: *80 ≤ 90*, discarded. The queue is empty.

The result is `best` = 90, found at node H: items 1 and 3. Of the 31 nodes of the full tree, only 11 were generated: A through K. Of these, 5 were expanded: A, B, D, E, H. The queue never held more than 3 nodes.

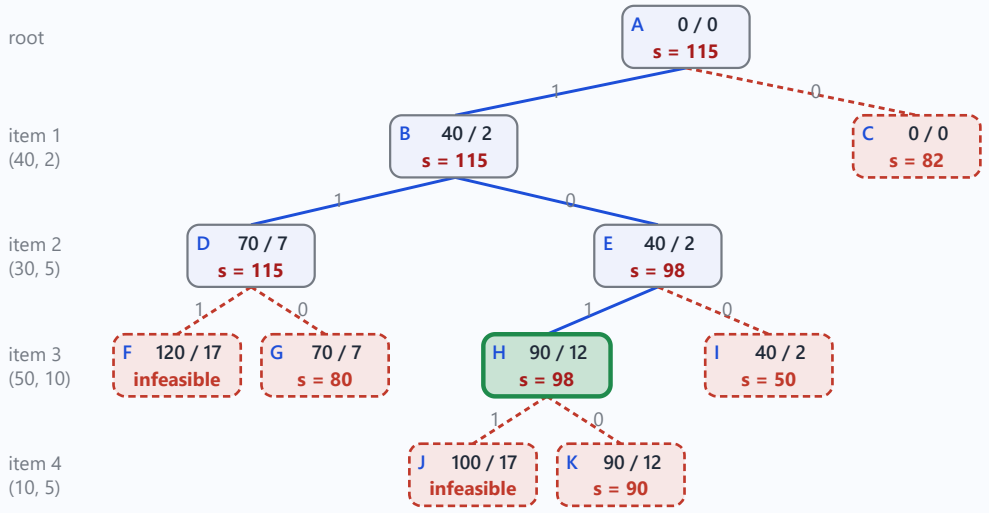


Figure 5.7: The final state of best-first branch and bound. Each box contains the node letter, "profit / weight," and the bound (*s*). The red dashed boxes are nodes that are infeasible or pruned by the bound. The green box H is the best solution: profit 90, weight 12. The labels on the left show the item decided at each level.

Figure 5.7 shows the finished tree of the search. Two moments of the trace are instructive. The first is step 4. The bound of D is 115, but the "take" child of D is infeasible and the bound of the "skip" child falls to 80. The bound approaches the truth as decisions are made. The second is the last step. When C and G entered the queue they looked promising: `best` was 40 when C entered and 70 when G entered. When their turn came, `best` was 90. Without the second test both would have been expanded; the result would not have changed, but work would have been done for nothing.

Common mistake

Updating `best` only at the leaves. In the knapsack every feasible node is a solution: the chosen items go into the knapsack and the rest are not taken. In the trace above `best` rises to 40 in step 2, to 70 in step 3, and to 90 in step 5. If the update were held back until the leaf, `best` would stay at 0 for a long time and nodes like I could not be pruned.

5.3.5 Java code: branchAndBound()

The Java code of best-first branch and bound is below. `PriorityQueue` is used as the priority queue.

```
int branchAndBound() {
  PriorityQueue<Node> queue = new PriorityQueue<>(
      (a, b) -> Double.compare(b.bound, a.bound));
  Node root = new Node(-1, 0, 0);
  root.bound = bound(root);
  queue.add(root);  int best = 0;
  while (!queue.isEmpty()) {
    Node u = queue.poll();           // largest bound
    if (u.bound <= best || u.level == n - 1) continue;
    int i = u.level + 1;
    // left child: take item i
    Node take = new Node(i, u.profit + p[i],
                         u.weight + w[i]);
    if (take.weight <= W && take.profit > best)
      best = take.profit;
    take.bound = bound(take);
    if (take.bound > best) queue.add(take);
    // right child: skip item i
    Node skip = new Node(i, u.profit, u.weight);
    skip.bound = bound(skip);
    if (skip.bound > best) queue.add(skip);
  }
  return best;
}
```

The important points of the code are these:

- **The comparator is reversed.** By default `PriorityQueue` returns the smallest element first. The comparator `(a, b) -> Double.compare(b.bound, a.bound)` reverses the order, so `poll()` returns the node with the largest bound.
- **The second test.** The line `if (u.bound <= best || u.level == n - 1) continue;` catches two things. The first is nodes that have gone stale while waiting in the queue (C and G in the example). The second is nodes at the last level; they have no children. The bound itself does not change in the queue; what changes is `best`.

- **The left child.** The node `take` adds item *i*. If the weight does not exceed the capacity and the profit is greater than `best`, `best` is updated. Then its bound is computed, and if the bound exceeds `best` it enters the queue. An infeasible `take` node has bound 0; it never enters the queue.
- **The right child.** The node `skip` carries its parent's profit and weight unchanged. If the parent is feasible, `skip` is feasible too and its profit does not change `best`. So only its bound is checked.
- **Changing the strategy.** For FIFO, an `ArrayDeque` with `add()` / `poll()` is used; for LIFO, `push()` / `pop()`. The rest of the code is the same.

The code returns only the best value. If the chosen items are wanted as well, a `boolean[]` array or a bit mask is added to the class `Node`. Each child copies its parent's array and records its own decision. When `best` is updated, this array is saved too.

Cost. In the worst case all the nodes of the full tree are generated: $O(2^n)$ nodes. At each node the bound computation takes $O(n)$ and the queue operation $O(\log k)$. The worst-case time is $O(n \cdot 2^n)$. Memory is proportional to the number of nodes in the queue, which in the worst case is also exponential. In practice, with a good bound, the number of nodes generated stays far below this.

5.3.6 FIFO branch and bound: code trace

The same example is now solved with a FIFO queue. The code uses an `ArrayDeque` instead of a priority queue. Children are generated by the helper method `child(u, take)`; the method builds the child and computes its bound as well. The loop first generates the “take” child (`take = 1`) and then the “skip” child (`take = 0`).

```
int fifoBranchAndBound() {
  ArrayDeque<Node> queue = new ArrayDeque<>();
  queue.add(root);  int best = 0;
  while (!queue.isEmpty()) {
    Node u = queue.poll();          // first in
    if (u.bound <= best || u.level == n - 1) continue;
    for (int take = 1; take >= 0; take--) { // take first
      Node c = child(u, take);        // includes bound(c)
      if (c.weight <= W && c.profit > best)
        best = c.profit;               // new incumbent
      if (c.bound > best) queue.add(c);
    }
  }
  return best;
}
```

In this code `root` has been built in advance and its bound computed. The structure of the loop is the same as in the best-first code; the only difference is that `poll()` returns the node that entered the queue first. Table 5.6 gives every step of the trace. The letters do not follow the order of generation; the children of C are named L and M.

Removed	"Take" child	"Skip" child	`best`	Queue (front to back)
—	—	—	0	A
A (115)	B 40/2, 115 → queued	C 0/0, 82 → queued	40	B, C
B (115)	D 70/7, 115 → queued	E 40/2, 98 → queued	70	C, D, E
C (82)	L 30/5, 82 → queued	M 0/0, 60 ≤ 70: pruned	70	D, E, L
D (115)	F 120/17: infeasible	G 70/7, 80 → queued	70	E, L, G
E (98)	H 90/12, 98 → queued	I 40/2, 50 ≤ 90: pruned	90	L, G, H
L (82)	82 ≤ 90: discarded at the second test		90	G, H
G (80)	80 ≤ 90: discarded at the second test		90	H
H (98)	J 100/17: infeasible	K 90/12, 90 ≤ 90: pruned	90	empty

Table 5.6: FIFO branch and bound trace ($W = 16$). Nodes are written as "profit / weight, bound."

The result is again 90. FIFO generated 13 nodes and expanded 6: A, B, C, D, E, H. The queue held at most 3 nodes. The source of the difference from the best-first strategy is node C. Best-first left C to the end, because C's bound (82) was smaller than that of the other nodes in the queue; by the time its turn came, `best` was 90 and C was never expanded. FIFO expands C third, in level order. At that moment `best` is only 70, and since C's bound is 82, C cannot be discarded. Expanding C generates two extra nodes (L and M) and means one extra expansion. L too ends up waiting in the queue for nothing.

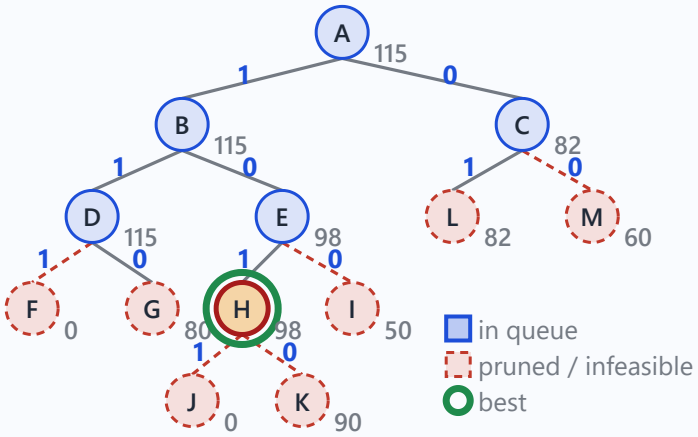


Figure 5.8: The final state of FIFO branch and bound. The numbers beside the nodes are bounds. The blue nodes are live nodes that entered the queue; the red dashed nodes are pruned or infeasible. The green ring is the best solution, H (90).

5.3.7 Comparing the three strategies

The same example can also be solved with the LIFO strategy. In LIFO the live nodes wait on a stack. If the children are pushed so that the "take" child comes out first, that is, the "skip" child is added first and then the "take" child, the search always descends the left branch. After A, B is expanded, then D. The "take" child of D is infeasible; the "skip" child, G (70/7, 80), is expanded. The "take" child of G

adds item 4: 80/12. This is a feasible solution and `best` rises to 80. The search then returns to E and finds 90 at H. Finally C (82), left on the stack, is discarded. LIFO also generates 13 nodes and expands 6: A, B, D, G, E, H.

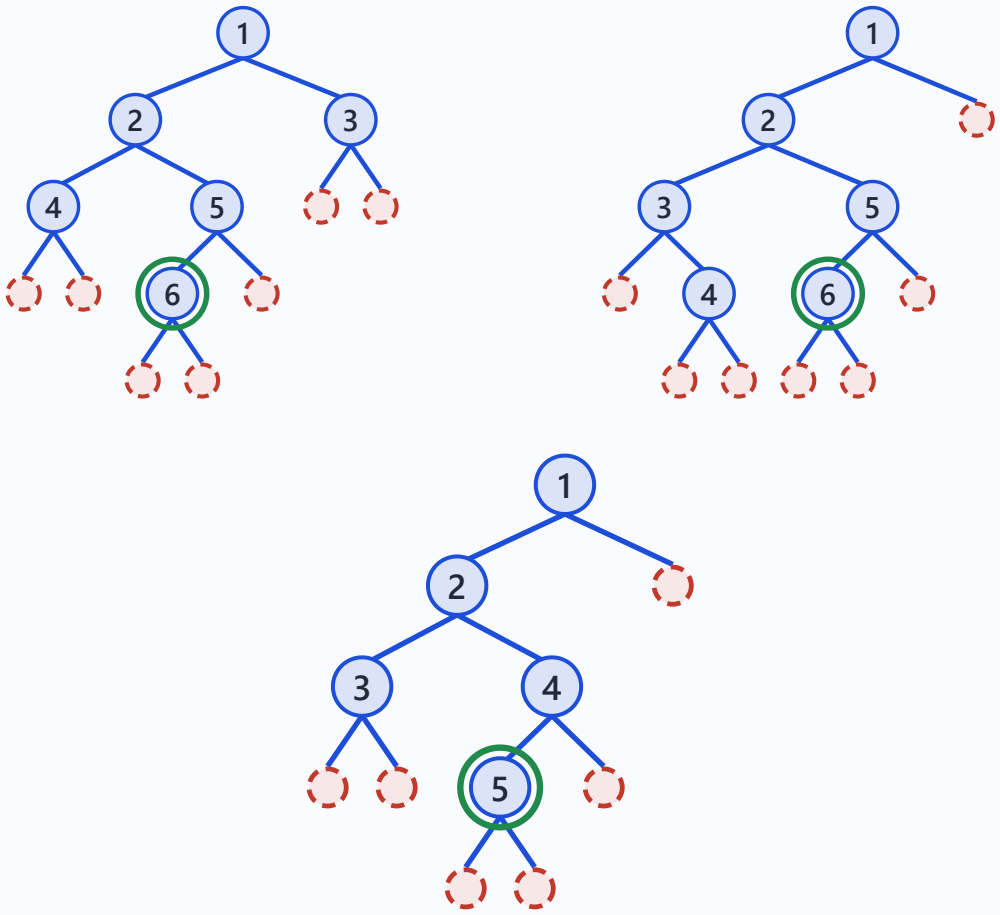


Figure 5.9: The three strategies on the same example: FIFO on the left (13 nodes, 6 expansions), LIFO in the middle (13 nodes, 6 expansions), best-first on the right (11 nodes, 5 expansions). The numbers give the order of expansion. The small circles are nodes that were generated but not expanded; the red dashed ones are pruned. The ring marks the optimum (90).

Figure 5.9 shows the three searches side by side; Table 5.7 summarizes the numbers.

Method	Nodes generated	Expanded	Largest queue
Brute force (full tree)	31	15	—
FIFO branch and bound	13	6	3
LIFO branch and bound (take first)	13	6	3
Best-first (LC)	11	5	3

Table 5.7: The amount of work of the methods on the same knapsack example.

Three conclusions follow from the table. First, all three strategies find the same optimum; the strategy affects only the amount of work. Second, best-first generated the fewest nodes in this example. This is a general tendency but not a guarantee. Third, in a small example the difference in memory is not visible: in all three the queue holds at most 3 nodes. In large examples the difference opens up. In the FIFO and best-first strategies the queue can grow exponentially; in LIFO the stack stays proportional to the depth.

The order in which the children are pushed onto the stack matters in LIFO. If the order is reversed, that is, the "skip" child comes out first, the search dives first into the branch in which item 1 is not taken. There are no good solutions in that branch; `best` rises slowly, first 40, then 50, 60, 80. This version generates 21 nodes and expands 10. In depth-first search, finding a good solution early speeds up the whole search. That is why the children are ordered so that the promising branch comes out first.

Example 5.2

The items (20, 2), (30, 5), (35, 7), (12, 3) and $W = 9$ of Example 5.1 are solved with the best-first strategy.

1. Root A: 0 / 0, $s = 60$. Queue: A (60).
2. A is expanded. B (item 1 taken): 20 / 2, $s = 60$; `best` = 20. C (no item 1): $s = 50$. Queue: B (60), C (50).
3. B is expanded. D (item 2 taken): 50 / 7, $s = 60$; `best` = 50. E (no item 2): 20 / 2, $s = 55$. Queue: D (60), E (55), C (50).
4. D is expanded. F (item 3 taken): weight 14 > 9, infeasible. G (no item 3): 50 / 7; *2/3* of item 4: $s = 50 + 8 = 58$. Queue: G (58), E (55), C (50).
5. G is expanded. H (item 4 taken): weight 10 > 9, infeasible. I (no item 4): $s = 50 \leq 50$, pruned. Queue: E (55), C (50).
6. E is expanded. J (item 3 taken): 55 / 9; `best` = 55. The bound of J is 55; $55 \leq 55$, so it does not enter the queue. K (no item 3): $s = 32 \leq 55$, pruned. Queue: C (50).
7. C is removed: $50 \leq 55$, discarded. The queue is empty.

The result is 55 (items 1 and 3). 11 nodes were generated and 5 were expanded. When C was added, `best` was 20; when it was removed, it was 55. In step 6 the value of J is written to `best` but J does not enter the queue. The bound of J equals its value; there is no better solution below it.

5.4 The traveling salesman problem

The **traveling salesman problem** (TSP) is a minimization problem. As a brief reminder: n cities are given, together with the distance between every pair of cities. We look for the shortest tour that visits every city exactly once and returns to the start. The distances are kept in a matrix d; $d[i][j]$ is the cost of going from i to j. If $d[i][j] = d[j][i]$, the problem is symmetric. The problem is NP-hard and brute force tries $(n - 1)!$ tours. In this section the same four-city example is examined with three different lower bounds.

5.4.1 State space and the cheapest-outgoing-edge bound

The tour always starts at city 0. Each node of the state-space tree is a **partial path** starting at 0. The root contains only 0. The children of a node are obtained by going from the last city of the path to a city not yet visited. Nodes at depth $n - 1$ contain all the cities; when we return from the last city to 0, the tour is complete.

Because the problem is a minimization, each node needs a **lower bound**: no tour that completes that partial path can cost less than this value. The simplest lower bound rests on the following observation. To complete the tour, we must leave the current last city and each unvisited city exactly once. Each departure costs at least the cheapest edge of that city. Let the cost of the cheapest outgoing edge of city v be `cheapest[v]`. These values are computed once before the search begins. The lower bound is:

$$lower(path) = cost(path) + cheapest[last\ city] + \Sigma_{v\ unvisited}\ cheapest[v]$$

This bound is a relaxation. In the original problem the outgoing edges must form a single tour. In the relaxation this condition is removed: each city leaves by its own cheapest edge, and whether the edges form a tour is not examined. Because of the removed constraint, the value of the relaxation cannot exceed the cost of any tour.

Theorem 5.2 (The cheapest-outgoing-edge bound is valid)

For every partial path starting at 0, the cost of every tour that completes the path is greater than or equal to $lower(path)$.

Proof. Any tour that completes the path contains the edges of the path unchanged; their sum is $cost(path)$. In the rest of the tour, the last city is left once and every unvisited city is left once; there is no other departure. The edge leaving city v costs at least `cheapest[v]`. Adding these inequalities, the cost of the tour is at least $lower(path)$. ■

The symmetric four-city matrix used in these notes is given in Table 5.8. The last column is the minimum of each row, that is, the array `cheapest`.

	0	1	2	3	cheapest
0	—	3	2	15	2
1	3	—	21	5	3
2	2	21	—	8	2
3	15	5	8	—	5

Table 5.8: The symmetric four-city distance matrix *d* and `cheapest`.

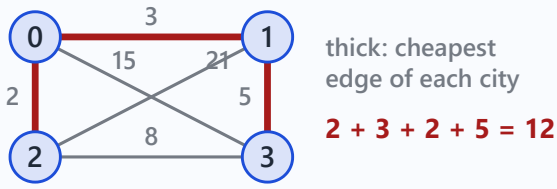


Figure 5.10: The four cities and the edge lengths. The thick edges are the cheapest outgoing edge of each city: 0-2, 1-0, 2-0, 3-1. The lower bound of the root is *2 + 3 + 2 + 5 = 12*.

At the root the path is just 0 and its cost is 0. The last city is 0, and 1, 2, 3 are unvisited. The lower bound is *0 + 2 + 3 + 2 + 5 = 12*. The four thick edges in Figure 5.10 do not form a tour: the edge 0-2 has been chosen twice and city 3 is never entered. This is why the value of the relaxation is smaller than a real tour.

The bound for the path 0 → 1 is as follows. The cost of the path is *d[0][1] = 3*. We will leave the last city, 1: `cheapest[1]` = 3. The unvisited cities are 2 and 3: *2 + 5*. The total is *3 + 3 + 7 = 13*. The bound is larger than that of the root; as decisions are made, the bound approaches the truth.

5.4.2 Code trace: tsp(0, 0, 1)

Branch and bound for the traveling salesman is written as a bounded depth-first search. It is a LIFO branch and bound; recursion plays the role of the stack.

```
void tsp(int city, int cost, int count) {
  if (count == n) {           // tour complete
    int tour = cost + d[city][0];
    if (tour < best) best = tour;
    return;
  }
  for (int c = 0; c < n; c++) {
    if (visited[c]) continue;
    int next = cost + d[city][c];
    visited[c] = true;
    if (lowerBound(c, next) < best) // bound
      tsp(c, next, count + 1);
    visited[c] = false;      // unchoose
  }
}
```

The parameters of the method are the last city of the path (`city`), the cost of the path (`cost`), and the number of cities on the path (`count`). The `visited` array marks the cities on the path; `best` is initially infinite. Before the first call, `visited[0] = true` is set and `tsp(0, 0, 1)` is called. The lines do the following:

- `if (count == n)` is the base case. All the cities are on the path; we return from the last city to 0 and the tour is complete. If the tour is shorter than `best`, `best` is updated.
- The `for` loop is the branching. Each unvisited city `c` is a child. Visited cities are skipped by `continue`; this is the infeasibility rule.
- `visited[c] = true;` adds the city to the path. The bound is computed after this marking, so that `lowerBound` sees city `c` as "visited" and counts it only as the last city.
- `if (lowerBound(c, next) < best)` is the bound rule. If the lower bound is less than `best`, the branch is entered; otherwise it is pruned.
- `visited[c] = false;` is the undo. The path returns to its old state before the loop moves on to the next city.

Table 5.9 gives all the steps of the search in order.

Step	Edge	`next`	`lower`	`best`	Decision
1	0 → 1	*0 + 3 = 3*	*3 + 3 + 2 + 5 = 13*	∞	branch
2	1 → 2	*3 + 21 = 24*	*24 + 2 + 5 = 31*	∞	branch
3	2 → 3	*24 + 8 = 32*	*32 + 5 = 37*	∞	branch
4	tour 0 → 1 → 2 → 3 → 0 = *32 + 15 = 47*			47	first candidate
5	1 → 3	*3 + 5 = 8*	*8 + 5 + 2 = 15*	47	branch
6	3 → 2	*8 + 8 = 16*	*16 + 2 = 18*	47	branch
7	tour 0 → 1 → 3 → 2 → 0 = *16 + 2 = 18*			18	new best
8	0 → 2	*0 + 2 = 2*	*2 + 2 + 3 + 5 = 12*	18	branch

Step	Edge	`next`	`lower`	`best`	Decision
9	2 → 1	*2 + 21 = 23*	*23 + 3 + 5 = 31*	18	pruned
10	2 → 3	*2 + 8 = 10*	*10 + 5 + 3 = 18*	18	pruned
11	0 → 3	*0 + 15 = 15*	*15 + 5 + 3 + 2 = 25*	18	pruned

Table 5.9: Trace of `tsp(0, 0, 1)`.

A few points in the trace stand out. The first tour (47) is bad, but it gives `best` a finite value. Right after it, 18 is found. In step 8 the bound of the branch 0 → 2 is 12; because *12 < 18*, the branch is entered. Two children of this branch are pruned. In step 10 the bound equals `best`: *18 ≥ 18*. The best tour in this branch can be at most 18, and a tour of 18 has already been found. In step 11 the last child of the root, 0 → 3, is pruned directly, and the two tours below it are never tried.

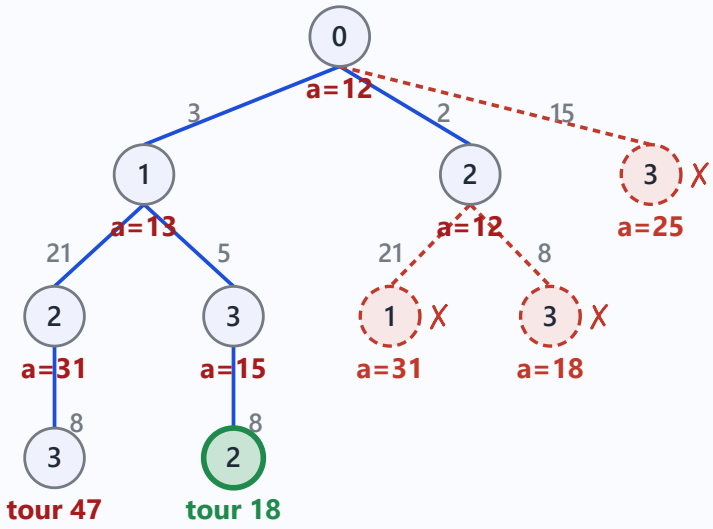


Figure 5.11: The finished tree of the `tsp` search. The value *a* under each node is the lower bound. The red dashed nodes are pruned. The green node completes the shortest tour (18).

The result is the shortest tour 0 → 1 → 3 → 2 → 0, of length *3 + 5 + 8 + 2 = 18*. The full state-space tree has *1 + 3 + 6 + 6 = 16* nodes. Branch and bound generated 10 nodes, counting the root, and called the method `tsp` 7 times. A search without pruning generates all 16 nodes and calls the method 16 times.

The code does not store the best tour itself. To store it, an `int[] path` array is kept. In the loop, `path[count] = c` is written before entering the branch. In the base case, if the tour is shorter than `best`, it is copied with `bestPath = path.clone()`. The copy is necessary, because the `path` array keeps changing during the rest of the search.

5.4.3 Code trace: lowerBound(3, 8)

The bound function is short:

```
int lowerBound(int last, int cost) {
  int lower = cost + cheapest[last];
  for (int v = 0; v < n; v++)
    if (!visited[v])       // not yet visited
      lower += cheapest[v];
  return lower;
}
```

`lower` first adds the cost of the path and the cheapest departure from the last city. The loop goes through all the cities and adds the `cheapest` value of every unvisited city. A visited city is of two kinds. Either it is inside the path and its outgoing edge is already included in the cost, or it is the last city and its departure was added at the start. In neither case should it be added again. The function takes *O(n)*.

The call in step 5 of the `tsp` trace is `lowerBound(3, 8)`. The path is 0 → 1 → 3, its cost is *3 + 5 = 8*, and `visited` = {0, 1, 3}.

Step	Explanation	`lower`
Start	*8 +* `cheapest[3]` *= 8 + 5*: the last city, 3, will be left once more	13
v = 0	visited; its outgoing edge (0 → 1) is in the cost	13
v = 1	visited; its outgoing edge (1 → 3) is in the cost	13
v = 2	not visited; it must be left by at least `cheapest[2]` = 2	15
v = 3	visited; its departure was added at the start	15

Table 5.10: Trace of `lowerBound(3, 8)`.

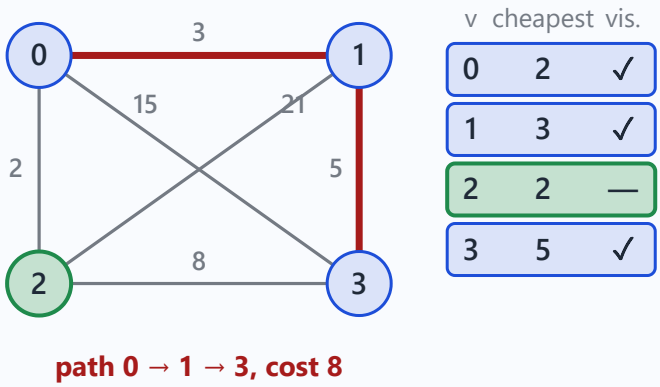


Figure 5.12: `lowerBound(3, 8)`. The red edges are the path 0 → 1 → 3 (cost 8). The table on the right shows the `cheapest` value of each city and whether it is visited; only city 2 contributes to the bound.

The function returns 15. Every tour in this branch costs at least 15. The real tour is 0 → 1 → 3 → 2 → 0 and costs 18. During the search `best` is 47, so *15 < 47* holds and the branch is entered.

Example 5.3

What is the lower bound of the path 0 → 3 → 1 in the same matrix? The cost of the path is *15 + 5 = 20*. The last city is 1: `cheapest[1]` = 3. The only unvisited city is 2: `cheapest[2]` = 2. The bound is *20 + 3 + 2 = 25*. Because *25 ≥ 18*, this path is pruned once the optimum has been found. The search never reaches this node anyway, because it already pruned 0 → 3 (bound 25) at the root.

5.4.4 A tighter bound: the two shortest edges

The cheapest-outgoing-edge bound counts only one edge per city. But in a tour each city is entered by one edge and left by one edge. In a symmetric problem there is no distinction between entering and leaving: every city touches exactly two edges of the tour. The sum of these two edges of a city cannot be smaller than the sum of the two shortest edges touching it. If this observation is summed over all the cities, each tour edge is counted twice: once at one end and once at the other. This gives a tighter lower bound:

> $$\textit{tour} \geq \tfrac{1}{2} \cdot \Sigma_v \textit{ (the sum of the two shortest edges touching } v\textit{)}$$

Theorem 5.3 (The two-shortest-edges bound)

In a symmetric traveling salesman problem, the cost of every tour cannot be smaller than half of the sum, over all cities, of the two shortest edges touching each city.

Proof. Let T be a tour. Every city v touches exactly two edges of T; if the sum of these two edges is called $t(v)$, then $\Sigma_v\, t(v) = 2 \cdot cost(T)$, because each edge is counted once at each of its ends. Let the sum of the two shortest edges touching v be $e(v)$. Both edges of T touching v are edges touching v; their sum cannot be smaller than $e(v)$: $t(v) \geq e(v)$. Summing gives $2 \cdot cost(T) \geq \Sigma_v\, e(v)$. ■

The code that computes this bound at the root is:

```
double twoEdgeBound() {
  int total = 0;
  for (int v = 0; v < n; v++) {
    int m1 = INF, m2 = INF;     // two smallest
    for (int u = 0; u < n; u++) {
      if (u == v) continue;
      if (d[v][u] < m1) { m2 = m1; m1 = d[v][u]; }
      else if (d[v][u] < m2) m2 = d[v][u];
    }
    total += m1 + m2;
  }
  return total / 2.0;  // each edge twice
}
```

The outer loop goes through the cities. The inner loop finds, among the edges touching the city, the two smallest values, with `m1` ≤ `m2`. If a new edge is smaller than `m1`, the old `m1` moves to second place and the new edge becomes first. If it is smaller only than `m2`, it is written to second place. This is the update of a two-element “smallest values list.” Because the total counts every edge twice, the result is divided by 2.0; `double` division keeps the half if the total is odd. The time is $O(n^2)$.

v	Edges	m1	m2	total
0	3, 2, 15	2	3	5
1	3, 21, 5	3	5	13
2	2, 21, 8	2	8	23
3	15, 5, 8	5	8	36

Table 5.11: Trace of `twoEdgeBound()`.

The row *v = 0* shows how the inner loop works. At the start `m1` = `m2` = ∞. At *u = 1*, *3 < ∞*: `m1` becomes 3 and the old `m1` (∞) moves to `m2`. At *u = 2*, *2 < 3*: `m2` = 3 and `m1` = 2. At *u = 3*, 15 is larger than both; there is no change. *2 + 3 = 5* is added to the total. In the end the total is 36 and the function returns *36 / 2 = 18*.

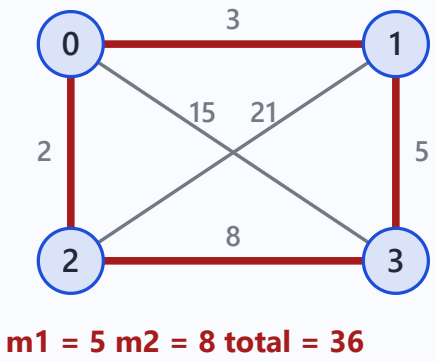


Figure 5.13: The two shortest edges of the last city, 3 (5 and 8), and the edges chosen by the four cities. The thick edges are 0-1, 1-3, 3-2, 2-0, and they form exactly the shortest tour. The total is 36, the bound is 18.

The result is striking: the bound is 18, equal to the length of the shortest tour. As Figure 5.13 shows, the two edges chosen by each city together form a real tour. The solution of the relaxation has turned out to be feasible; that is why the bound is tight. Since at the root the bound equals the optimum, as soon as the first complete tour is found all the remaining branches are pruned by the condition *lower ≥ best*. On partial paths the same idea is applied by fixing the edges on the path. One or both of the edges of a city on the path are already known. The shortest edges are sought only for those that are missing.

5.4.5 The reduced cost matrix

A third lower bound looks at the rows and columns of the cost matrix. The idea is this. Every tour uses exactly one cell from each row of the matrix (a departure from a city) and exactly one cell from each column (an arrival at a city). If the same number is subtracted from all the cells of a row, the cost of every tour decreases by exactly that number. The ordering of the tours does not change. The sum of the numbers subtracted from the rows and columns has been taken off the cost of every tour. Because the remaining matrix is nonnegative, this sum is a lower bound.

The procedure is as follows. The diagonal is set to ∞; one cannot go from a city to itself. From each row the minimum of that row is subtracted. Then from each column the minimum of that column is subtracted. In the end every row and every column contains at least one 0. This matrix is called the **reduced cost matrix**.

```
int reduce(int[][] m) {
  int total = 0;
  for (int i = 0; i < n; i++) {       // rows
    int mn = rowMin(m, i);
    if (mn > 0 && mn < INF) {
      subtractRow(m, i, mn); total += mn;
    }
  }
  for (int j = 0; j < n; j++) {       // columns
    int mn = colMin(m, j);
    if (mn > 0 && mn < INF) {
      subtractCol(m, j, mn); total += mn;
    }
  }
  return total;
}
```

`rowMin`, `subtractRow`, `colMin`, and `subtractCol` are simple helper methods. The subtraction is applied only to cells that are not ∞. The condition `mn > 0` skips subtracting zero. The condition `mn < INF` skips rows and columns that are entirely ∞; such rows arise during branching. The method modifies the matrix in place and returns the total subtracted. Both passes take $O(n^2)$.

Step	Minimum	Result	`total`
Row 0: ∞, 3, 2, 15	2	∞, 1, 0, 13	2
Row 1: 3, ∞, 21, 5	3	0, ∞, 18, 2	5
Row 2: 2, 21, ∞, 8	2	0, 19, ∞, 6	7
Row 3: 15, 5, 8, ∞	5	10, 0, 3, ∞	12
Columns 0, 1, 2	0	no change	12
Column 3: 13, 2, 6, ∞	2	11, 0, 4, ∞	14

Table 5.12: Trace of `reduce()` (the four-city example).

When the row reduction is finished, the total is 12. This number is the same as the cheapest-outgoing-edge bound; the row minima are exactly the `cheapest` values. The column step adds new information. After the row reduction there is no 0 in column 3: not every city's cheapest departure goes to 3. Yet in a tour city 3 must be entered, and this entry costs at least 2. The column step brings the constraint “every city is entered once” into the bound as well and raises the bound to 14. Table 5.13 is the final form of the reduced matrix.

	0	**1**	**2**	**3**
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4

	0	1	2	3
3	10	0	3	∞

Table 5.13: The reduced matrix (total subtracted: 14). Every row and column contains at least one 0.

With this method, branching is done by choosing an edge. Let the edge $i \to j$ be chosen in the reduced matrix of the parent node. The matrix of the child is built as follows. All the cells of row i and column j are set to ∞: there can be no other departure from i and no other arrival at j. The cell (j, i) is also set to ∞; going straight back from j to i would close a cycle too early. The matrix is reduced again. The bound of the child is computed as:

> *bound(child) = bound(parent) + reduced c[i][j] + the total of the new reduction*

Example 5.4

The bound of the root is 14. Three branches open from the root:

1. **0 → 1.** Reduced *c[0][1] = 1*. Row 0, column 1, and *(1, 0)* are set to ∞. The remaining rows: row 1 = ∞, ·, 18, 0 (minimum 0); row 2 = 0, ·, ∞, 4 (0); row 3 = 10, ·, 3, ∞ (3). 3 is subtracted from row 3: 7, ·, 0, ∞. Every column has a 0. The new reduction is 3. The bound is *14 + 1 + 3 = 18*.
2. **0 → 2.** Reduced *c[0][2] = 0*. Row 0, column 2, and *(2, 0)* are set to ∞. Row 2 = ∞, 19, ·, 4; its minimum is 4 and it is subtracted. The other rows and columns have a 0. The bound is *14 + 0 + 4 = 18*.
3. **0 → 3.** Reduced *c[0][3] = 11*. Row 0, column 3, and *(3, 0)* are set to ∞. Every row keeps a 0. Column 2 = 18, ∞, 3 (excluding row 0); its minimum is 3. The bound is *14 + 11 + 3 = 28*.

The bound of the 0 → 3 branch is 28. As soon as the optimum 18 is found, this branch is pruned. The bounds of the 0 → 1 and 0 → 2 branches are exactly 18, and both contain the shortest tour: 0 → 1 → 3 → 2 → 0 and its reverse 0 → 2 → 3 → 1 → 0.

The reduced matrix method rests on the paper published by Little and colleagues in 1963. The method has an important advantage: it does not require symmetry. It is valid for asymmetric problems with $d[i][j] \neq d[j][i]$ as well. One-way streets, or vehicles that burn different amounts of fuel uphill and downhill, are examples of asymmetric problems. The two-shortest-edges bound, by contrast, depends on symmetry and cannot be used directly in an asymmetric problem.

5.4.6 Comparing the three bounds

In the same four-city example the three lower bounds give three different values at the root (Figure 5.14).

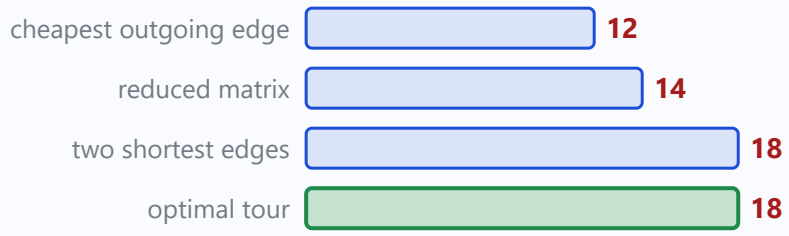


Figure 5.14: The three lower bounds of the root and the optimum in the four-city example: cheapest outgoing edge 12, reduced matrix 14, two shortest edges 18, optimal tour 18.

Bound	Cost per node	What it uses	Note
Cheapest outgoing edge	$O(n)$ (`cheapest[]` is computed in advance)	one departure from each city	cheapest, loosest
Reduced matrix	$O(n^2)$	one arrival + one departure for each city	valid for asymmetric problems too
Two shortest edges	$O(n^2)$	the two edges touching each city	for the symmetric problem

Table 5.14: Tightness and cost of the three TSP lower bounds.

A tight bound prunes more but is more expensive at each node. The cheapest-outgoing-edge bound is computed in $O(n)$ thanks to the `cheapest` array. The other two look at the whole matrix at every node. The ranking does not have to be the same in every example. The reduced matrix bound and the two-shortest-edges bound use different information, and in another example their order can change. The cheapest-outgoing-edge bound, on the other hand, cannot be larger than either of them. The row step of the reduction alone gives it. In the two-edge bound, too, the average of a city's two edges is not smaller than its cheapest edge.

For larger examples, even tighter bounds are used. The **1-tree** bound leaves one city out, builds a minimum spanning tree for the rest, and connects the outside city by its two shortest edges. Every tour is a 1-tree; so the smallest 1-tree is a valid lower bound. The linear programming relaxation is the basis of the software that today solves examples with thousands of cities exactly. Which bound to choose is decided by the balance between the number of nodes pruned and the work done per node.

5.4.7 A larger example: 6 cities

In the four-city example the effect of pruning is limited; the full tree is small anyway. To see the effect, a symmetric six-city example is taken (Table 5.15).

	0	1	2	3	4	5	cheapest
0	—	29	20	21	16	31	16
1	29	—	15	29	28	40	15
2	20	15	—	15	14	25	14
3	21	29	15	—	12	18	12
4	16	28	14	12	—	22	12
5	31	40	25	18	22	—	18

Table 5.15: The symmetric six-city distance matrix.

The cheapest-outgoing-edge bound of the root is *16 + 15 + 14 + 12 + 12 + 18 = 87*. The shortest tour is 0 → 1 → 2 → 3 → 5 → 4 → 0: *29 + 15 + 15 + 18 + 22 + 16 = 115*. There are other tours of the same length; the search keeps the first one it finds. The gap between the root bound and the optimum is large (87 against 115); the cheapest-outgoing-edge bound is loose in this example.

Version	Nodes generated	`tsp` calls
Without pruning (backtracking)	326	326
Cheapest-outgoing-edge bound	210	130

Table 5.16: The amount of work in the six-city example.

The search without pruning generates all the nodes of the state-space tree: *1 + 5 + 20 + 60 + 120 + 120 = 326*. The 120 nodes at the last level are the *(6 − 1)! = 120* tours. The bounded search generates 210 nodes; that is, `lowerBound` is computed 210 times (counting the root). Only in 130 of them is the branch entered and `tsp` called. The number of calls falls from 326 to 130, about two fifths. A tighter bound reduces these numbers markedly. For example, the two-shortest-edges bound gives 100.5 at the root for this matrix; that is much closer than 87.

5.5 The assignment problem

The assignment problem is suitable for showing the minimization direction of branch and bound on a small, easy-to-follow example. The code structure is very similar to the traveling salesman search; the bound is even simpler.

5.5.1 The problem and a lower bound

Definition 5.4

The **assignment problem**: given *n* people and *n* jobs, *c[i][j]* is the cost of person *i* doing job *j*. Each person is given a different job so that the total cost is minimized.

Each assignment is a permutation of the jobs; there are *n!* possible assignments. At level *i* of the state-space tree, person *i* is given a job that nobody has yet been given. The root is the state in which no person has a job; the leaves are complete assignments.

The lower bound rests on the following observation: every person who has not yet been given a job pays at least the smallest cost in their own row. This is a relaxation. The constraint "a single person per job" is removed; two people may choose the same cheap job. Because job clashes are ignored, the bound cannot rise above the true cost. To compute the bound quickly at every node, the suffix sums of the row minima are kept in advance:

> $$remainingMin[i] = \Sigma_{r \geq i} \min_j c[r][j], \quad remainingMin[n] = 0$$

After person *i* has been given a job, if the cost of the path is `next`, the lower bound of that node is *next + remainingMin[i + 1]*. The computation takes *O(1)*. If *lower ≥ best*, the branch is cut.

person \ job	0	1	2	minimum
0	5	8	4	4
1	7	3	6	3
2	2	9	5	2

Table 5.17: A three-person assignment example (row: person, column: job).

In the example the row minima are 4, 3, 2. The suffix sums are *remainingMin = [9, 5, 2, 0]*. The lower bound of the root is 9: whichever assignment is made, the total cannot fall below 9.

The assignment problem also has a polynomial-time algorithm: the **Hungarian method** solves the problem in $O(n^3)$ time. So branch and bound is not needed for the pure assignment problem; it is used here to demonstrate the method. For variants with extra constraints the situation changes. For example, if some jobs cannot be given to the same person, or to two people who are tied to each other, the Hungarian method cannot be applied directly. For such problems branch and bound is truly needed. In branching, the result of the Hungarian method can also be used as a lower bound.

5.5.2 Code trace: assign(0, 0)

```
void assign(int i, int cost) {
  if (i == n) {                    // complete assignment
    best = Math.min(best, cost);
    return;
  }
  for (int j = 0; j < n; j++) {
    if (used[j]) continue;
    int next = cost + c[i][j];
    if (next + remainingMin[i + 1] >= best)
      continue;                    // bound
    used[j] = true;
    assign(i + 1, next);
    used[j] = false;          // unchoose
  }
}
```

The structure of the method is the same as that of the `tsp` method. `i` is the next person and `cost` is the total of the assignments made so far. The `used` array marks the jobs already given; this is the infeasibility rule. The bound test is made before a job is added to the partial assignment. If the branch is cut, `used` never changes, so no undo is needed. In the base case `best` is updated with `Math.min`.

Step	Assignment	`next`	lower	`best`	Decision
1	p0 → job 0	5	*5 + 5 = 10*	∞	branch
2	p1 → job 1 (job 0 taken)	8	*8 + 2 = 10*	∞	branch
3	p2 → job 2 (the only free job)	13	13	13	complete assignment
4	p1 → job 2	11	*11 + 2 = 13*	13	cut
5	p0 → job 1	8	*8 + 5 = 13*	13	cut
6	p0 → job 2	4	*4 + 5 = 9*	13	branch
7	p1 → job 0	11	*11 + 2 = 13*	13	cut
8	p1 → job 1	7	*7 + 2 = 9*	13	branch
9	p2 → job 0	9	9	9	complete assignment

Table 5.18: Trace of `assign(0, 0)` (*remainingMin = [9, 5, 2, 0]*, with `best` = ∞ at the start).

The essence of the trace is this. The first complete assignment (jobs 0, 1, 2) costs 13 and becomes `best`. Then three branches are cut because their bound equals 13. The cut in step 5 is significant: below the branch p0 → job 1 there are two complete assignments and neither is tried. In step 6 the branch p0 → job 2 is opened with bound 9. In this branch the assignments p1 → job 1 and p2 → job 0 give *4 + 3 + 2 = 9*. The result equals the lower bound of the root; from this equality it is immediately clear that no better assignment can exist.



Figure 5.15: The cells chosen at the end of the search: p0 → job 2 (4), p1 → job 1 (3), p2 → job 0 (2). The total, 9, equals the lower bound of the root.

The search made 7 calls in all: `assign(0, 0)`, together with four intermediate nodes and two complete assignments. A search without pruning goes through all *3! = 6* assignments; that takes *1 + 3 + 6 + 6 = 16* calls. In a four-person example the difference is larger. In the matrix in Table 5.19 the row minima are 3, 2, 3, 2 and the lower bound of the root is 10.

person \ job	0	1	2	3
0	7	3	8	4
1	5	6	2	9
2	4	8	5	3
3	6	2	7	5

Table 5.19: A four-person assignment example.

The best assignment is p0 → job 3 (4), p1 → job 2 (2), p2 → job 0 (4), p3 → job 1 (2), with a total of 12. Branch and bound reaches only 4 complete assignments in 21 calls. A search without pruning tries all *4! = 24* complete assignments and makes 65 calls.

5.6 Integer linear programming

The most widespread industrial use of branch and bound is integer linear programming. The method was put forward in this field in the 1960 paper of Land and Doig. Today the commercial software that solves production planning, scheduling, and distribution problems uses advanced versions of the same idea.

5.6.1 The problem and the LP relaxation

Definition 5.5

Integer linear programming (ILP) is the problem of maximizing or minimizing a linear objective function subject to linear inequality constraints. All the variables are required to be integers.

Without the integrality condition the problem would be an ordinary **linear program** (LP). Linear programs are solved very fast in practice by the simplex method, and in polynomial time by interior-point methods. In small examples with two variables the solution can be found graphically: the constraints define a polygon in the plane (the feasible region), and the best value is attained at one of the corners of this polygon. The integrality condition is what makes the problem NP-hard.

Branch and bound combines these two facts:

- **Bound:** The integrality condition is removed and the **LP relaxation** is solved. The feasible region of the relaxation contains all the integer points. So in maximization the LP optimum is an upper bound for the ILP optimum.
- **Branching:** If a variable in the LP solution is fractional, for example $y = 1.5$, two subproblems are built: the constraint $y \le 1$ is added to one and $y \ge 2$ to the other. The fractional region in between, $1 < y < 2$, is thrown away. Since there are no integer points in this region, no integer solution is lost.
- **Completeness:** If the LP solution of a node is already integral, there is no better integer solution below that node. This solution becomes a candidate and the node is closed.

The example problem is:

> maximize $Z = 5x + 4y$; $6x + 4y \le 24$, $x + 2y \le 6$, $x, y \ge 0$ integers

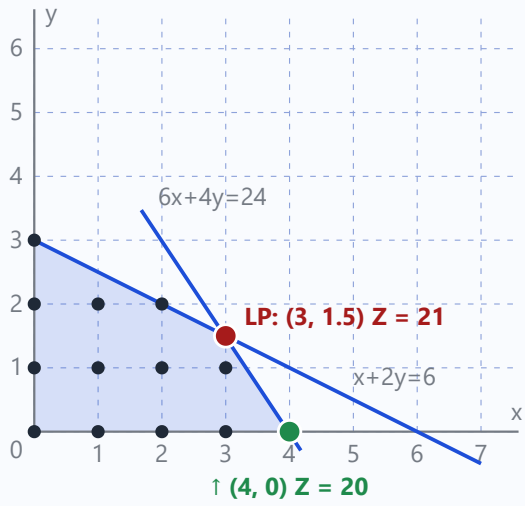


Figure 5.16: The shaded area is the feasible region of the LP relaxation; the dots are the integer solutions. The LP optimum is at the point (3, 1.5) and $Z = 21$. The integer optimum is at the point (4, 0) and $Z = 20$.

The optimum of the LP relaxation is at the intersection of the two constraint lines. $6x + 4y = 24$ and $x + 2y = 6$ are solved together. Multiplying the second equation by 2 and subtracting it from the first gives $4x = 12$, that is, $x = 3$; from this $y = 1.5$. The value is $Z = 15 + 6 = 21$. The other corners are (0, 0), (4, 0), and (0, 3); their values are 0, 20, and 12. Figure 5.16 shows the region and the two optima.

5.6.2 Branching with LP relaxations

The branch and bound tree is built step by step as follows:

1. **Root.** The LP optimum is (3, 1.5), $Z = 21$. This value is an upper bound: no integer solution can exceed 21. y is fractional; we branch on y.
2. **Branch $y \leq 1$.** In the new LP we press against the limit $y = 1$. From $6x + 4 \leq 24$ comes $x \leq 3.33$, and from $x + 2 \leq 6$ comes $x \leq 4$. The LP optimum is (3.33, 1), $Z = 20.67$. x is fractional; the branches $x \leq 3$ and $x \geq 4$ are opened.
3. **Branch $y \leq 1, x \leq 3$.** The LP optimum is (3, 1), $Z = 19$. The solution is integral: the first candidate, `best` = 19.
4. **Branch $y \leq 1, x \geq 4$.** The constraint $6 \cdot 4 + 4y \leq 24$ forces $y = 0$. The LP optimum is (4, 0), $Z = 20$. It is integral and $20 > 19$: `best` = 20.
5. **Branch $y \geq 2$.** The constraint $x + 2y \leq 6$ forces $x \leq 2$. The LP optimum is (2, 2), $Z = 18$. Because $18 \leq 20$, the branch is pruned. Even if the LP solution of this branch were integral, it could not beat `best`.

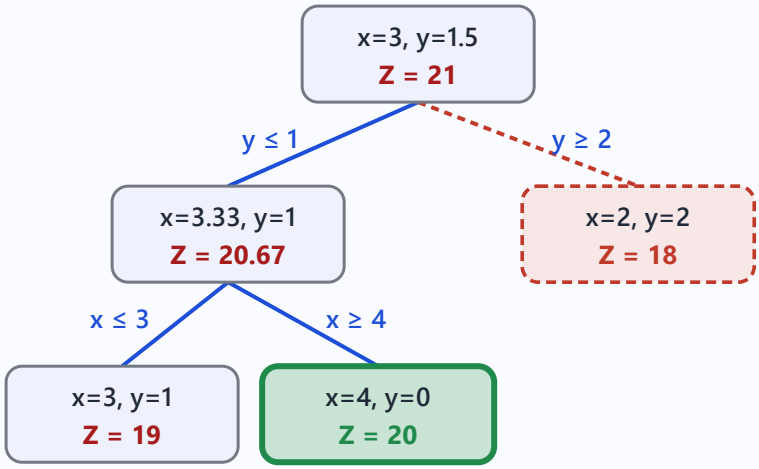


Figure 5.17: The ILP branch and bound tree. Each node shows the solution and value of its LP relaxation. The branch $y \geq 2$ is pruned with $Z = 18 \leq 20$. The green node is the integer optimum: $x = 4, y = 0, Z = 20$.

The optimum is $x = 4$, $y = 0$, $Z = 20$. This example also shows why rounding the LP solution is not enough. If (3, 1.5) is rounded down, (3, 1) is obtained and $Z = 19$; it is not the best. If it is rounded up, (3, 2) is obtained: $6 \cdot 3 + 4 \cdot 2 = 26 > 24$, which is infeasible. The true optimum (4, 0) is not near the LP point; it is found only by branching.

Common mistake

Branching wrongly on a fractional variable. For $y = 2.6$ the correct subproblems are $y \leq 2$ and $y \geq 3$. Branching as $y = 2$ and $y = 3$ loses all the other values of y. Branching as $y \leq 2.6$ and $y \geq 2.6$ leaves the existing LP solution in both subproblems and the search makes no progress. $y \leq 3$ and $y \geq 2$, in turn, give two overlapping subproblems; the same points are searched twice.

A second example shows the completeness rule: maximize $Z = 3x + 2y$; $x + y \leq 4$, $x \leq 3$, $y \leq 2$. Brute force tries all 12 points of the grid $0 \leq x \leq 3$, $0 \leq y \leq 2$ and finds $Z = 11$ at the point (3, 1). The optimum of the LP relaxation is also (3, 1): the corner resting on the limit of x (3) and the line $x + y \leq 4$. Because the solution is already integral, branch and bound ends at the root; a single LP has been solved.

Real software improves this basic scheme in several directions. The fractional variable to branch on is chosen intelligently. Extra constraints (cutting planes), which cut off the fractional solution without cutting off integer points, are added to the LP relaxation. This combination is called branch and cut. In addition, the LP of a subproblem is re-solved in a few steps starting from the solution of its parent.

5.7 Maximum clique

The last application is a graph problem. The clique problem is used to find tightly connected groups in social networks, clusters of interacting proteins in biology, and error-correcting codes in coding theory.

5.7.1 The problem and the bound

Definition 5.6

In an undirected graph, a set of vertices in which every two vertices are adjacent is called a **clique**. The **maximum clique** problem is the problem of finding the clique with the most vertices.

The problem is NP-hard. The search grows the current clique K step by step. The vertices that can be added to K are called **candidates**. For a vertex to be a candidate, it must be adjacent to all the vertices in K. To prevent the same clique from being generated again and again in different orders, vertices are added in increasing order of number: only vertices with a number greater than the last vertex added to K can be candidates.

The bound is a maximization bound and is very simple. In the best case K grows with all the candidates. So the largest clique that can be found in this branch cannot exceed $|K| + |candidates|$ vertices. If $|K| + |candidates| \leq |best|$, the current clique cannot beat `best` however far it is extended; the branch is pruned. The bound is valid, because every clique in the branch is the union of K and a subset of the candidates.

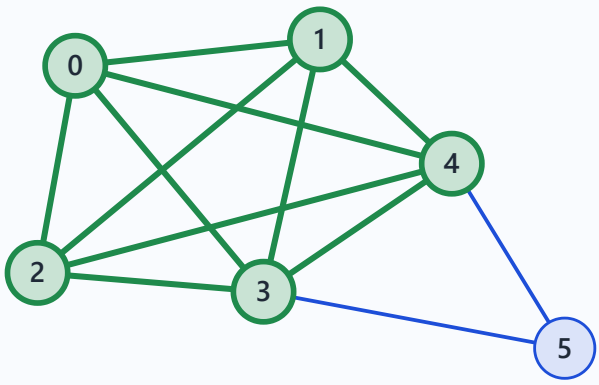


Figure 5.18: The example graph with six vertices. Every pair of vertices numbered 0, 1, 2, 3, and 4 is adjacent; the largest clique is {0, 1, 2, 3, 4}. Vertex 5 is adjacent only to 3 and 4.

```
void clique(List<Integer> K, int start) {
  if (K.size() > best.size())
    best = new ArrayList<>(K);
  List<Integer> cands = new ArrayList<>();
  for (int v = start; v < n; v++)
    if (compatible(v, K)) cands.add(v);  // adjacent to K
  if (K.size() + cands.size() <= best.size())
    return;                          // bound
  for (int v : cands) {
    K.add(v);
    clique(K, v + 1);
    K.remove(K.size() - 1);          // unchoose
  }
}
```

The lines of the method work as follows:

- The first two lines update `best`. A clique is a feasible solution at every moment; there is no need to descend to a leaf. `new ArrayList<>(K)` makes a copy, because `K` changes during the rest of the search.
- The `for` loop collects the candidates. `compatible(v, K)` checks whether v is adjacent to every vertex in `K`. The loop starts at `start`; smaller numbers have already been tried.
- `if (K.size() + cands.size() <= best.size()) return;` is the bound rule.
- The last loop is the branching. Each candidate is added to `K` in turn, the search continues from `v + 1`, and the candidate is removed again.

In one call the candidate list is built in $O(n \cdot |K|)$ time; if adjacency is kept in a matrix, each `compatible` check is $O(|K|)$. Without the bound line, the method is ordinary backtracking and traverses all the cliques.

5.7.2 Code trace: clique({}, 0)

The search starts with the empty clique and `start` = 0. Table 5.20 gives the calls in order.

Call	K	Candidates	$\lvert K\rvert + \lvert cands\rvert$	`best`	Decision
1	{}	{0, 1, 2, 3, 4, 5}	0 + 6	{}	continue
2	{0}	{1, 2, 3, 4}	1 + 4	{0}	continue
3	{0, 1}	{2, 3, 4}	2 + 3	{0, 1}	continue
4	{0, 1, 2}	{3, 4}	3 + 2	{0, 1, 2}	continue
5	{0, 1, 2, 3}	{4}	4 + 1	{0, 1, 2, 3}	continue
6	{0, 1, 2, 3, 4}	{}	5 + 0	{0, 1, 2, 3, 4}	no candidates
7	{0, 1, 2, 4}	{}	4 + 0 ≤ 5	5 vertices	pruned
8	{0, 1, 3}	{4}	3 + 1 ≤ 5	5 vertices	pruned

Call	*K*	Candidates	*|K| + |cands|*	`best`	Decision
9	{0, 1, 4}	{}	3 + 0 ≤ 5	5 vertices	pruned
10–12	{0, 2}, {0, 3}, {0, 4}	{3, 4}, {4}, {}	2 + 2, 2 + 1, 2 + 0	5 vertices	pruned
13–17	{1}, {2}, {3}, {4}, {5}	{2, 3, 4}, {3, 4}, {4, 5}, {5}, {}	4, 3, 3, 2, 1	5 vertices	pruned

Table 5.20: Trace of `clique({}, 0)`.

The first five calls dive deep, always adding the first candidate, and in call 6 reach the clique with five vertices. This is the largest clique of the graph. The bound of every branch opened afterward is at most 5, and all of them are pruned at once. For example, in call 8 *K = {0, 1, 3}*; the only candidate is 4. In the best case a clique of four vertices is obtained, and this cannot exceed 5. The remaining candidates 1–5 at the root are also tried one by one, and all are cut in the same way. For example, when *K = {3}*, the candidates are 4 and 5. 3, 4, 5 form a triangle, but the bound is *1 + 2 = 3 ≤ 5*.

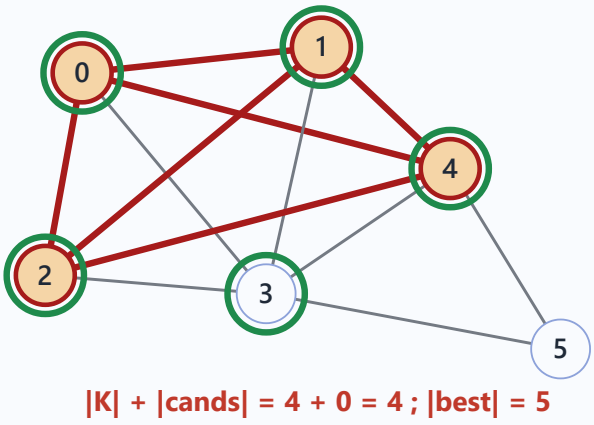


Figure 5.19: Call 7: *K = {0, 1, 2, 4}* (orange), no candidates. The green rings show the best clique (5 vertices). *|K| + |cands| = 4 + 0 ≤ 5*; the branch is pruned.

The result is {0, 1, 2, 3, 4}. The bounded search makes 17 calls. When the bound line is deleted, the search traverses all the cliques and makes 36 calls. This number is the number of cliques in the graph, counting the empty set: $2^5 = 32$ cliques in the complete five-vertex part, plus four cliques containing vertex 5: {5}, {3, 5}, {4, 5}, {3, 4, 5}.

5.7.3 A tighter bound: coloring

The bound *|K| + |cands|* assumes that every candidate can enter the clique. If there are non-adjacent pairs among the candidates, this assumption is very optimistic. A tighter bound comes from **graph coloring**. Every pair of vertices of a clique is adjacent; so all the vertices in a clique must receive different colors. If the candidates can be properly colored with *k* colors, at most *k* of the candidates can enter the same clique. The new bound is: if *|K| + colors(cands) ≤ |best|*, the branch is pruned.

Finding the minimum number of colors is also NP-hard. But the best coloring is not needed. Greedy coloring goes through the vertices in order and gives each vertex the smallest color not used among its neighbors. It takes $O(cands^2)$. It may use extra colors; this only loosens the bound, it does not invalidate it. The bound still cannot exceed $|cands|$, because in the worst case each vertex gets its own color.

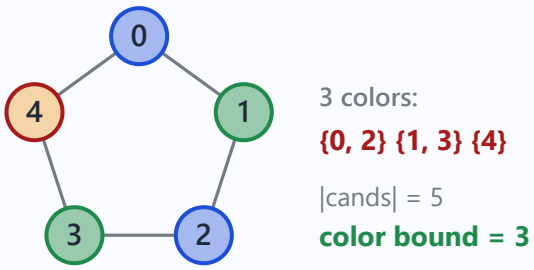


Figure 5.20: A ring of five candidates. Greedy coloring uses 3 colors: {0, 2}, {1, 3}, {4}. $|cands| = 5$, the color bound is 3.

In the example in Figure 5.20 the candidates form a ring of five vertices; K is empty and $|best| = 3$. The old bound gives $0 + 5 = 5 > 3$ and the branch is opened. Greedy coloring gives vertex 0 color 1, vertex 1 color 2, vertex 2 color 1, and vertex 3 color 2. Vertex 4 is adjacent to both 3 and 0; it takes color 3. The color bound gives $0 + 3 \le 3$ and the branch is pruned. The largest clique in the ring has only 2 vertices anyway. Most of the fastest exact clique algorithms rest on this idea. Tomita's MCQ and MCS algorithms try the candidates in color order and prune with the color bound.

5.8 Other problems and the relation to A*

5.8.1 Other problems solved by branch and bound

The method can be applied to almost any optimization problem whose solution is built from a sequence of decisions. The two questions to ask in each application are the same: how is the branching done, and which relaxation gives a valid bound?

- **Hamiltonian path.** We look for a path that visits every vertex exactly once. The weighted version is very similar to the traveling salesman and is solved with similar bounds.
- **Boolean satisfiability** (SAT). Truth values are assigned to the variables; partial assignments that contradict are pruned. This is the essence of the DPLL algorithm. In the maximum SAT version we want as many clauses as possible to be true; the bound is the largest number of clauses that can still be satisfied by the remaining assignments.
- **Scheduling.** Jobs are distributed among machines; the maximum load (makespan) or the total tardiness is minimized. The lower bound is the load the remaining jobs would create in the best case; for example, splitting the total of the remaining jobs equally among the machines.

- **Bin packing.** Items are placed into the smallest number of bins. The lower bound is *⌈Σ size / bin capacity⌉*: even if the bins were filled perfectly, this many bins would be needed.
- **Maximum coverage.** k sets are chosen that cover the most elements. The upper bound is the number of elements the largest remaining sets can add.
- **The assignment problem** and its variants with extra constraints. The lower bound is the smallest cost of each unassigned row.

Branch and bound is meaningful for NP-hard problems. For problems with a known polynomial-time algorithm, such as maximum flow, shortest path, or minimum spanning tree, there is no need for branch and bound. For these problems the polynomial algorithm is always preferred; branch and bound takes exponential time in the worst case.

5.8.2 Branch and bound, A*, and depth-first branch and bound

Best-first branch and bound and the A* algorithm of path finding are two faces of the same idea. Table 5.21 compares three closely related methods.

Property	Best-first branch and bound (LC)	A*	Depth-first branch and bound
Selection	best bound ($\hat{c}$)	smallest $f = g + h$	stack or recursion
Bound / estimate	valid (optimistic) bound	admissible h (never overestimates)	valid bound + `best`
Goal	optimization (the best leaf)	the shortest path from start to goal	optimization
Memory	live nodes: can be exponential	open list: can be exponential	*O(depth)*
First complete solution	usually late but good	the first goal found is optimal	early; a solution is in hand at every moment

Table 5.21: Best-first branch and bound, A*, and depth-first branch and bound.

A* is LC branch and bound looking for a path in a weighted graph. $g(x)$ is the cost paid from the start to x; it corresponds to $f(x)$ in LC search. $h(x)$ is a lower bound on the remaining cost from x to the goal; it corresponds to $\hat{g}(x)$. The number of misplaced tiles in the 8-puzzle is such an estimate. If h is admissible, the first goal node removed from the queue gives the shortest path. The reason is the same as in branch and bound. The cost of that goal is less than or equal to the f value of every node in the queue. And the f values are lower bounds for every path through those nodes.

Depth-first branch and bound is the method for cases where memory is limited. Memory is bounded by the depth, and at every moment of the search a solution is in hand. Its price is that when a bad branch is entered, the good solution is found late. There is also a middle way. IDA* (iterative deepening A*) runs a depth-first search with a threshold on f. It cuts nodes that exceed the threshold, and if the search fails, it raises the threshold and starts again. Memory stays bounded by the depth, and the order of best-first search is largely preserved.

5.9 Comparing the design paradigms

Branch and bound is among the last of the algorithm design paradigms to be treated, and it combines some of the ideas of the others. Table 5.22 puts six paradigms side by side by the same criteria.

Paradigm	Decision	Subproblems / traversal	When?	Example
Brute force	all candidates are generated, then tested	the full search space	if the space is very small	8^8 queen placements
Divide and conquer	fixed splitting rule; divide, solve, combine	independent; top-down	if the subproblems do not overlap	merge sort, Strassen
Greedy	the option that looks best at the moment is chosen; no turning back	a single subproblem after the choice	if the greedy-choice property is proven	Dijkstra, Huffman, Kruskal
Dynamic programming	all options are tried and the best is stored	overlapping; mostly bottom-up	optimal substructure, a polynomial number of subproblems	Fibonacci, 0/1 knapsack, longest common subsequence
Backtracking	the partial solution is extended; undone if invalid	depth-first on the state-space tree	constraint satisfaction, all solutions	n-queens, Sudoku
Branch and bound	backtracking + pruning by a bound	depth-first, breadth-first, or best-first	NP-hard optimization, an exact result	0/1 knapsack, traveling salesman

Table 5.22: A cumulative comparison of the design paradigms.

When choosing a paradigm for a problem, the questions are asked in a certain order. Are the subproblems independent? Then divide and conquer is suitable. Do the subproblems overlap, and is their number polynomial? Dynamic programming is suitable. Does the locally best choice lead to the global best, and can this be proved? The greedy approach is suitable. If none of these holds and the problem is NP-hard, two options remain. If an exact result is wanted, backtracking or branch and bound is used. If an approximate result of guaranteed quality in polynomial time is enough, an approximation algorithm is used.

The same problem can be solved with more than one paradigm. The 0/1 knapsack is a good example. Dynamic programming solves it in $O(nW)$; if the capacity is small, it is the best choice. If the capacity is very large or the weights are fractional, branch and bound comes to the fore. The greedy method, on the other hand, does not guarantee the optimum in the 0/1 version; it is correct only in the fractional version. Branch and bound takes this greedy method inside itself as a bound computation. The paradigms are not rivals of one another; they are often used as parts inside one another.

5.10 Common mistakes

- **Using an invalid (pessimistic) bound.** In the knapsack, adding up the profits of just the items that fit completely is not an upper bound; it is the value of a feasible solution. The branch containing the optimum can be pruned, and the algorithm returns a wrong result without raising any error.
- **Mixing up the direction.** In maximization a node is pruned when *bound* ≤ *best*, in minimization when *lower* ≥ *best*. Writing the reverse either misses the optimum or never prunes. Remember also that `best` must start at +∞ in minimization and at −∞ or 0 in maximization.
- **Not testing again after leaving the queue.** The node looked promising when it was added. If `best` grew while it waited, it is now unnecessary. In the knapsack example C and G are such nodes. Without the second test the result does not change, but unnecessary nodes are expanded.
- **Not sorting the items.** The fractional bound may not be a valid upper bound if the items are not in decreasing order of the ratio p_i/w_i.
- **Updating `best` only at the leaves.** In the knapsack and in clique, every feasible node is a solution. Updating early raises `best` quickly and brings more pruning.
- **Taking branch and bound to be the same as backtracking.** Code with no bound test, for example an unpruned TSP search, is not branch and bound; it traverses the whole tree.
- **Forgetting to undo.** In recursive versions, if the line `visited[c] = false`, `used[j] = false`, or `K.remove(...)` is forgotten, later branches run with a wrong state and feasible solutions are lost.
- **Storing `best` without copying.** In the clique search, writing `best = K` stores the list that keeps changing. When the search ends, `best` shows an empty list. A copy (`new ArrayList<>(K)`, `path.clone()`) is essential.

5.11 Summary

- Branch and bound traverses the solution space of an optimization problem as a state-space tree. The loop is: a live node is selected, its bound is tested again, the node is branched, the bounds of the children are computed, and the hopeless ones are pruned. `best` holds the value of the best complete solution found so far.
- The bound must be valid (optimistic), tight, and cheap. Validity is mandatory; tightness and cheapness determine efficiency. The bound is mostly built by relaxation. An upper bound is used in maximization and a lower bound in minimization.
- A node is pruned for three reasons: infeasibility, an insufficient bound (*bound* ≤ *best* or *lower* ≥ *best*), and completeness (a leaf, or an LP solution that turns out integral). The bound rule is what separates branch and bound from backtracking.
- FIFO traverses breadth-first and uses a queue. LIFO traverses depth-first, uses a stack or recursion, and needs little memory. LC (best-first) selects the most promising node with a priority queue and usually generates the fewest nodes. All three find the same optimum.

- In the knapsack the fractional knapsack relaxation gives the upper bound. In the example *bound(root) = 115* and the optimum is 90. Best-first generates 11 of the 31 nodes, FIFO and LIFO 13.
- In the traveling salesman problem, the cheapest-outgoing-edge bound adds to the cost of the path the cheapest departures of the last city and of the unvisited cities. With four cities 10 nodes are generated instead of 16. The reduced matrix (14) and the two shortest edges (18) are tighter but take $O(n^2)$ per node.
- In assignment the lower bound is the sum of the row minima of the remaining people. In integer programming the LP relaxation gives the upper bound, and a fractional variable is branched into two subproblems. In clique the bound $|K| + |cands| \le |best|$ is used; coloring gives a tighter bound.
- The method is exponential in the worst case. Its practical success depends on the quality of the bound. It is not used for problems that have a polynomial algorithm.
- Independent subproblems lead to divide and conquer, overlapping subproblems to dynamic programming, and a proven greedy choice to the greedy approach. For an exact result on an NP-hard problem, backtracking or branch and bound is used.

5.12 Key terms

branch and bound branching bound pruning state-space tree feasible relaxation upper bound

best-first search least-cost (LC) search depth-first branch and bound admissible heuristic

A* search backtracking 0/1 knapsack fractional knapsack traveling salesman problem

reduced cost matrix assignment problem Hungarian method integer linear programming

LP relaxation maximum clique graph coloring NP-hard

5.13 Test Yourself

1. Let the items (p, w) be (30, 3), (40, 5), (45, 9), (10, 4) and $W = 12$. What is the fractional upper bound of the root? Which nodes does best-first branch and bound expand, how many nodes does

it generate, and what is the optimum?

Answer

The ratios are 10, 8, 5, 2.5; the items are in order. At the root items 1 and 2 fit whole (70, weight 8); *4/9* of item 3 is taken: *4 · 45/9 = 20*. *bound(root) = 90*. The root is expanded: B (item 1 taken) 30/3, bound 90, `best` = 30; C (no item 1) bound *40 + 7 · 5 = 75*. B is expanded: D 70/8, bound 90, `best` = 70; E 30/3, bound 75. D is expanded: the "take" child 115/17 is infeasible; G (no item 3) 70/8, bound *70 + 10 = 80*. G is expanded: H (item 4 taken) 80/12, `best` = 80; the bound of H is 80 and it does not enter the queue. I (no item 4) has bound 70 and is pruned. E (75) and C (75) in the queue are discarded. 4 expansions (A, B, D, G), 9 nodes. The optimum is 80: items 1, 2, and 4.

2. If the same example is solved with a FIFO queue, how many nodes are generated and how many are expanded? Where does the difference come from?

Answer

FIFO generates 13 nodes and expands 6: A, B, C, D, E, and G. The result is again 80. FIFO expands C (bound 75) early, in level order; at that moment `best` is only 70 and C cannot be discarded. E is also expanded; its child 75/12 is not infeasible and raises `best` to 75. Best-first leaves C and E to the end; by then `best` is 80 and both are discarded.

3. Let the symmetric distances be *d(0,1) = 10*, *d(0,2) = 15*, *d(0,3) = 20*, *d(1,2) = 35*, *d(1,3) = 25*, *d(2,3) = 30*. What are the cheapest-outgoing-edge, reduced matrix, and two-shortest-edges bounds at the root? How long is the shortest tour?

Answer

The cheapest departures are 10, 10, 15, 20: the bound is 55. In the reduction the row minima are *10 + 10 + 15 + 20 = 55*. After the row reduction, the minimum in column 2 is 5 and in column 3 is 10: the bound is *55 + 5 + 10 = 70*. The sums of the two shortest edges are 25, 35, 45, 45: *150 / 2 = 75*. The shortest tour is 0 → 1 → 3 → 2 → 0: *10 + 25 + 30 + 15 = 80*. The order is again cheapest outgoing edge ≤ reduced matrix ≤ two edges ≤ optimum. With the cheapest-outgoing-edge bound, the `tsp` search generates 15 of the 16 nodes in this example; the loose bound prunes almost nothing.

4. For the assignment problem with cost matrix [[4, 7, 3], [2, 6, 1], [3, 9, 4]], what is `remainingMin`?

What is the optimal assignment, and how many calls does `assign(0, 0)` make?

Answer

The row minima are 3, 1, 3: *remainingMin = [7, 4, 3, 0]*; the lower bound of the root is 7. The search finds complete assignments of cost 14, 13, and 11 in turn. The optimum is p0 → job 1 (7), p1 → job 2 (1), p2 → job 0 (3), with a total of 11. The search makes 12 calls; a search without pruning makes 16. In this example the first branch (p0 → job 0) is bad, so pruning starts late.

5. Maximize $Z = 3x + 5y$; $x + 2y \le 7$, $3x + y \le 9$, $x, y \ge 0$ integers. How does the tree take shape when branch and bound is applied with LP relaxations (branching first on y, then on x if it is fractional)?

Answer

Root: the intersection of the two lines (2.2, 2.4), $Z = 18.6$; y is fractional. Branch $y \le 2$: (2.33, 2), $Z = 17$; x is fractional. Below it, $x \le 2$: (2, 2), $Z = 16$, integral, `best` = 16. $x \ge 3$: (3, 0), $Z = 9 \le 16$, pruned. Branch $y \ge 3$: (1, 3), $Z = 18$, integral, `best` = 18. The optimum is $x = 1$, $y = 3$, $Z = 18$. Rounding the LP solution down would have given (2, 2) and 16.

6. What is the largest clique in the graph with edges 0–1, 0–2, 0–3, 1–2, 1–3, 2–3, 3–4, 3–5, 4–5? How many calls does the method `clique` make with and without the bound?

Answer

The largest clique is {0, 1, 2, 3} (4 vertices). The bounded search makes 13 calls and the unbounded search 22. The number of unbounded calls is the number of cliques, counting the empty set: the 16 subsets of {0, 1, 2, 3} plus {4}, {5}, {3, 4}, {3, 5}, {4, 5}, {3, 4, 5}. In the bounded search, after {0, 1, 2, 3} has been found, for example the branch $K = \{3\}$, cands = {4, 5} is cut by $1 + 2 \le 4$.

7. If, in the knapsack bound, the profits of all the remaining items are added instead of the fractional piece, is the bound valid? What does it give at the root in the $W = 16$ example?

Answer

It is valid: no 0/1 solution can earn more than taking all the remaining items. But because it ignores the capacity entirely, it is very loose. At the root it gives $40 + 30 + 50 + 10 = 130$; the fractional bound is 115. A looser bound prunes less; the result stays correct but more nodes are generated.

8. In the best-first knapsack code, if the comparator is mistakenly written as

`Double.compare(a.bound, b.bound)`, does the result stay correct? What changes?

Answer

The result stays correct. As long as the pruning rule and the bound are valid, the optimum is not lost whatever the order of traversal. What changes is the order: at each step the node with the *smallest* bound comes out of the queue. The search busies itself with the most hopeless branches and `best` rises late. So more nodes are expanded; the strategy is no longer "best first" but "worst first."

5.14 Further reading

- E. Horowitz, S. Sahni, S. Rajasekaran, *Computer Algorithms* (2nd ed., 2008), chapter 8: treats LC search and the cost function $\hat{c} = f + \hat{g}$ in detail. The knapsack and traveling salesman applications of FIFO and LC branch and bound are given in language close to that of these notes.
- A. Levitin, *Introduction to the Design and Analysis of Algorithms* (3rd ed., 2012), section 12.2: presents branch and bound for assignment, knapsack, and traveling salesman with short, visual examples; it is suitable to read together with backtracking.
- J. D. C. Little, K. G. Murty, D. W. Sweeney, C. Karel, "An Algorithm for the Traveling Salesman Problem" (1963): the paper that first used the name "branch and bound" and introduced the reduced cost matrix method.
- R. Carraghan, P. M. Pardalos, "An Exact Algorithm for the Maximum Clique Problem" (1990): the classic clique algorithm based on the $|K| + |cands|$ bound; the starting point of the later algorithms with a color bound.

5.15 References

1. R. Carraghan, P. M. Pardalos, "An Exact Algorithm for the Maximum Clique Problem", *Operations Research Letters* 9(6), 1990.
2. E. Horowitz, S. Sahni, S. Rajasekaran, *Computer Algorithms*, 2nd ed., Silicon Press, 2008 (chapter 8: Branch-and-Bound).
3. A. H. Land, A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems", *Econometrica* 28(3), 1960.
4. A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed., Pearson, 2012 (section 12.2).
5. J. D. C. Little, K. G. Murty, D. W. Sweeney, C. Karel, "An Algorithm for the Traveling Salesman Problem", *Operations Research* 11(6), 1963.
6. R. Neapolitan, *Foundations of Algorithms*, 5th ed., Jones & Bartlett, 2015 (chapter 6).

Approximation Algorithms

A freight company plans the routes of vehicles that will visit hundreds of addresses every morning. A factory assigns jobs to machines at the start of the day. A network administrator wants to place enough monitors to watch every link. For all of these problems, every known algorithm that finds the best solution takes exponential time as the input grows, yet the decision has to be made in seconds. Approximation algorithms offer a middle way. They give up on the optimal solution, but they prove a bound on how much worse than optimal their answer can be. The notes first introduce the approximation ratio and what separates an approximation algorithm from a heuristic. They then treat, for the traveling salesman problem, the nearest-neighbor heuristic, the 2-approximation based on a minimum spanning tree, and 2-opt local search. For machine scheduling they cover list scheduling and the longest-processing-time-first rule. For vertex cover they present the edge-picking 2-approximation and the reduction from 3-SAT, and for the knapsack problem the ratio-ordered greedy rule and an approximation scheme. The last section covers power iteration, a typical example of numerical convergence. Every algorithm comes with Java code, a step-by-step trace, a proof of its ratio, and examples showing that the ratio is tight.

LEARNING OUTCOMES

Concepts

Define an approximation algorithm, the approximation ratio, and approximation schemes (PTAS, FPTAS); distinguish an approximation algorithm from heuristic and metaheuristic methods.

Traveling salesman

Apply the nearest-neighbor heuristic, the 2-approximation based on a minimum spanning tree for metric TSP, and 2-opt local search step by step; explain why general TSP has no constant-ratio approximation.

Scheduling and covering

Trace list scheduling, the longest-processing-time-first rule, and the vertex cover 2-approximation in Java; prove their ratios and construct tight examples; explain the reduction from 3-SAT to vertex cover.

Numerical convergence
Compute the dominant eigenvalue and eigenvector with power iteration; show how the convergence rate depends on the ratio $|\lambda_2/\lambda_1|$.

6.1 Approximation algorithms

Most of the algorithms studied so far find the best solution to a problem. A sorting algorithm sorts the array exactly, and a shortest-path algorithm really returns the shortest path. All of these run in polynomial time: when the input doubles, the running time grows by a constant factor. Yet for many optimization problems met in practice no such algorithm is known. For these problems the only known way to find the best solution is to try a very large fraction of the possible solutions, and the number of solutions grows exponentially or factorially with the input.

6.1.1 Why exact solutions are expensive

The traveling salesman problem (TSP), vertex cover, machine scheduling, and the knapsack problem are problems of this kind. All four are **NP-hard**. A brief reminder of the complexity classes is useful. P is the class of decision problems solvable in polynomial time. NP is the class of decision problems for which the correctness of a proposed solution can be checked in polynomial time. A problem is NP-hard if every problem in NP reduces to it in polynomial time; if it is both NP-hard and in NP, it is NP-complete. Whether P = NP is unknown, but the common belief is that they differ. If P ≠ NP, no NP-hard problem has an exact polynomial-time algorithm.

The practical meaning of this result is clear: one should not expect a method that guarantees the best route for a hundred cities, or the best schedule for a thousand jobs, in reasonable time. Three options remain. One can settle for small inputs and accept exponential time; branch and bound and dynamic programming reduce this time but do not eliminate it. One can use a heuristic that is fast but promises nothing. The third option is the subject of these notes: an algorithm that runs in polynomial time and whose result is proven to be within a bounded distance of the optimum.

Definition 6.1

An **approximation algorithm** for an optimization problem is an algorithm that runs in polynomial time and, on every input, produces a feasible solution that is within a provable ratio of the optimal solution.

All three parts of the definition matter. The algorithm runs in *polynomial time*; otherwise it would be no different from searching for an exact solution. The solution it produces is *feasible*, that is, it satisfies all the constraints of the problem: the tour visits every city, the schedule assigns every job to

a machine, the cover covers every edge. Finally, the quality of the solution lies within a proven bound *for every input*. Doing well on a few trials is not enough. The algorithm must not exceed this bound even on the worst input.

This is a trade-off. We give up a little optimality, and in return we obtain a guarantee on both efficiency and solution quality. When computing resources, that is, time and memory, are limited, this trade-off is often the most sensible choice. Moreover, the proven ratio is a worst-case bound. On inputs met in practice, the solutions found by approximation algorithms are usually much closer to the optimum than this bound.

6.1.2 The balance-scale analogy

Suppose the left pan of a two-pan balance holds a load of 37 kilograms. The weights available for the right pan are 20, 10, 5, and 1 kilograms, and each can be used only once. These weights cannot balance 37 kilograms exactly: even all of them together come to only 36 kilograms. The closest balance achievable is 36 kilograms, and the difference of 1 kilogram is about 2.7% of the load.

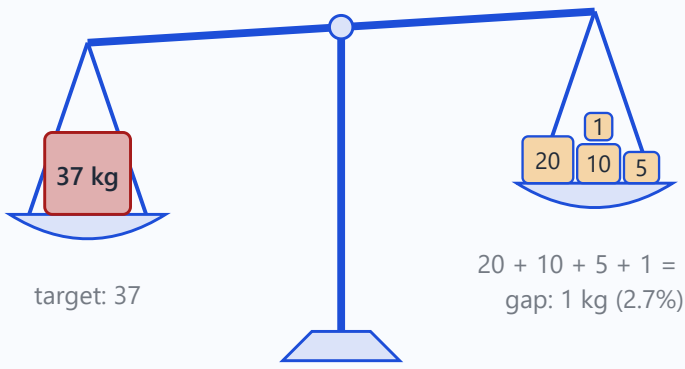


Figure 6.1: The balance-scale analogy. On the left a load of 37 kilograms, on the right 20 + 10 + 5 + 1 = 36 kilograms. Exact balance is impossible; the difference is 1 kilogram (2.7%).

Figure 6.1 shows this situation. The idea the analogy carries is this: instead of searching for exact balance, that is, the optimum, we accept a balance that is set up quickly and whose error is bounded. An approximation algorithm does the same. The solution it finds may not be the best, but the largest possible gap is known in advance. If the difference on the balance never exceeds 3% of the load, we can speak of an "error of at most 3%". For approximation algorithms this bound is expressed as a ratio.

6.1.3 The approximation ratio

For a given input, let C be the cost of the solution the algorithm finds and C^* the cost of the best (optimal) solution. For a minimization problem $C \geq C^*$; for a maximization problem $C \leq C^*$. To compare the two cases with a single measure, we divide the larger value by the smaller one.

Definition 6.2

An algorithm is a **ρ(n)-approximation algorithm** if, for every input of size n,

> max(C / C*, C* / C) ≤ ρ(n)

holds. We call $\rho(n)$ the **approximation ratio** of the algorithm. The ratio is always $\rho \geq 1$; if $\rho = 1$, the algorithm finds the optimal solution.

The *max* in the definition combines the two cases. For minimization, $C \geq C^*$, so the larger ratio is C / C^* and the condition takes the form $C \leq \rho \cdot C^*$. The cost the algorithm finds is at most ρ times the best cost. For example, a 2-approximation vertex cover algorithm picks at most twice as many vertices as the smallest cover. For maximization, $C \leq C^*$, so the larger ratio is C^* / C and the condition becomes $C \geq C^* / \rho$. The value the algorithm finds is at least a $1/\rho$ fraction of the best value. For example, a 2-approximation knapsack algorithm guarantees at least half of the best value.

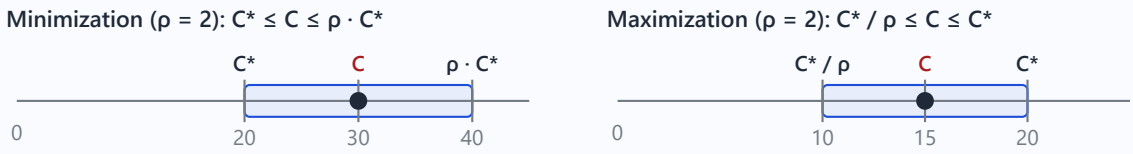


Figure 6.2: The approximation ratio on a number line ($\rho = 2$). Left, minimization: if $C^* = 20$, then C lies between 20 and 40. Right, maximization: if $C^* = 20$, then C lies between 10 and 20.

Figure 6.2 shows the two cases on a number line. For minimization the interval in which C can lie is $[C^*, \rho \cdot C^*]$, and for maximization it is $[C^* / \rho, C^*]$. The example in the figure takes $C^* = 20$ and $\rho = 2$. If a minimization algorithm finds 30, the ratio for this input is $30 / 20 = 1.5$. If a maximization algorithm finds 15, the ratio is $20 / 15 \approx 1.33$. Both are within the bound of 2.

Example 6.1

In a minimization problem the algorithm finds a solution of cost 30, and the cost of the best solution is 20. For this input the ratio is $C / C^* = 30 / 20 = 1.5$. If the algorithm is a 2-approximation, this result is within the expected bound, because $30 \leq 2 \cdot 20 = 40$.

We can also look from the other direction. If a 2-approximation algorithm found 30, what can be said about the best cost? From the inequality $C \le 2 \cdot C^*$ we get $C^* \ge 15$. Also, the best solution cannot be worse than the one found: $C^* \le 30$. So the best cost lies between 15 and 30. The approximation ratio also yields a lower bound on the optimum without knowing it.

Writing the ratio as a function of n is deliberate. Some algorithms have a constant ratio: 2 for vertex cover, 1.5 for metric TSP. For others it grows with the input: the ratio of the nearest-neighbor heuristic on metric TSP is $\Theta(\log n)$. A constant ratio is a much stronger guarantee. However large the input gets, the algorithm never strays from the optimum by more than a constant factor.

Common mistake

Writing the ratio upside down. For minimization the condition is $C \le \rho \cdot C^*$, for maximization $C \ge C^* / \rho$. Writing $C \ge \rho \cdot C^*$ for maximization would demand that the algorithm find a value better than the optimum, which is impossible. In both kinds of problem, ρ is at least 1. Some sources use a ratio below 1 for maximization ("at least 0.5 of the optimum"), but these notes, like the slides, always write $\rho \ge 1$.

6.1.4 Approximation algorithms, heuristics, and approximation schemes

The methods that can be applied to a hard optimization problem can be ranked by the guarantee they give. Table 6.1 summarizes this ranking.

Method	Guarantee	Example
Exact algorithm	Best solution; exponential time on NP-hard problems	Brute-force TSP: $(n-1)!/2$ tours
Approximation algorithm	Polynomial time and a provable ratio ρ	Vertex cover: $\rho = 2$
PTAS	Ratio $(1+\varepsilon)$ for every fixed $\varepsilon > 0$; time polynomial in n, possibly exponential in $1/\varepsilon$	Euclidean TSP (Arora, 1998)
FPTAS	Ratio $(1+\varepsilon)$; time polynomial in both n and $1/\varepsilon$	Knapsack: $O(n^3/\varepsilon)$
Heuristic	No guarantee; usually good in practice	Nearest neighbor, 2-opt
Metaheuristic	No guarantee; a search framework independent of the problem	Genetic algorithm, ant colony

Table 6.1: Types of methods for hard optimization problems and the guarantees they give.

An exact algorithm gives the strongest guarantee, but on NP-hard problems its price is exponential time. An approximation algorithm stays in polynomial time and secures its quality with a ratio. For some problems the user chooses the desired accuracy with a number ε, and the algorithm returns a solution with ratio $(1+\varepsilon)$. Such a family of algorithms is called an **approximation scheme**.

Definition 6.3

A family of algorithms that gives a $(1 + \varepsilon)$-approximation for every fixed $\varepsilon > 0$ and whose running time is polynomial in n is called a **polynomial-time approximation scheme** (PTAS). If the running time is also polynomial in $1/\varepsilon$, the family is called a **fully polynomial-time approximation scheme** (FPTAS).

The difference between the two schemes looks subtle at first, but in practice it is large. A PTAS with running time $O(n^{1/\varepsilon})$ is polynomial for every fixed ε. Yet choosing $\varepsilon = 0.1$ makes the time n^{10}, which is unusable even on an input of a hundred elements. For an FPTAS with running time $O(n^3/\varepsilon)$, on the other hand, $\varepsilon = 0.1$ multiplies the time by only ten. The knapsack problem has an FPTAS. Euclidean TSP for points in the plane has a PTAS; Arora and Mitchell showed this independently of each other.

The methods in the lower part of the table give no guarantee. A **heuristic** produces a good solution quickly with a problem-specific rule of thumb. Nearest neighbor and 2-opt are well-known TSP heuristics. A **metaheuristic** is a general search framework that is not tied to a particular problem; simulated annealing, genetic algorithms, and ant colony optimization belong to this group. These methods often give very good results in practice, but how far they can stray from the optimum in the worst case is unknown.

6.1.5 Applications and techniques

The problems that approximation algorithms address are part of everyday life. The **traveling salesman problem** appears in parcel delivery and in the order of drilling holes on circuit boards. The **knapsack problem** is used in budget and load planning, and **job scheduling** in machine and processor planning. **Vertex cover** places the fewest monitoring points in a network. Numerical methods that converge to the eigenvalues of large matrices by iteration are also mentioned under the same heading; Google's PageRank computation is the best-known example.

The approximation techniques applied to these problems come down to a few main ideas:

- **Simplification:** a simpler version of the problem is solved and its result is adapted to the original problem. The linear programming relaxation (LP relaxation), which drops the integrality constraint, and the minimum spanning tree, which gives a lower bound for TSP, are of this kind.
- **Greedy choice:** at each step the option that looks best at that moment is taken. Nearest neighbor, longest processing time first, and ratio-ordered knapsack are examples.
- **Local search:** the current solution is improved by small changes; the search stops when no better neighboring solution remains. 2-opt is an example.
- **Primal–dual method:** the primal and dual solutions of a linear program are grown together. It is often used for covering problems.
- **Randomization:** random rounding or sampling gives a ratio guarantee in expectation.

- **Metaheuristics:** simulated annealing, genetic algorithms, ant colony optimization. They give no guarantee, but they are flexible and powerful.

These techniques have one thing in common. Almost every ratio proof builds a *lower bound* on the unknown value of the optimal solution. The best tour is at least as long as a minimum spanning tree; the best schedule takes at least as long as the longest job; the smallest cover contains at least as many vertices as there are edges in a matching. The result of the algorithm is compared with this lower bound. In this way the ratio is proved without ever computing the optimal solution. All the ratio proofs in these notes follow the same pattern.

6.2 The traveling salesman problem

The traveling salesman problem is the most studied problem for approximation algorithms. Its definition fits in one sentence, but finding the best solution is surprisingly hard. This section first defines the problem and computes the cost of an exact solution. It then examines two approximate methods: the nearest-neighbor heuristic and the 2-approximation based on a minimum spanning tree.

6.2.1 Problem definition

> **Definition 6.4**
>
> **Traveling salesman problem** (TSP): given n cities and the distance d_{ij} between every pair of cities i, j, find, among all tours that visit every city exactly once and return to the starting city, the one with the smallest total distance. If $d_{ij} = d_{ji}$, the problem is called **symmetric TSP**.

TSP is an optimization problem: the goal is to minimize the total distance. It is also a combinatorial problem, because a solution is the order in which the cities are visited, that is, a permutation. In graph terms, symmetric TSP seeks the lightest **Hamiltonian cycle** in a complete graph with weighted edges. A Hamiltonian cycle passes through every vertex exactly once and returns to the start.

The problem is often confused with the shortest-path problem. The shortest path between two cities is found in polynomial time with Dijkstra's algorithm. TSP, by contrast, asks for the shortest closed tour through *all* the cities and is NP-hard. Finding the best route for hundreds of cities is very hard in practice.

This section uses the five-city example from the slides. The cities are numbered 0 to 4, and the distances are given in the symmetric matrix of Table 6.2.

d	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30

d	0	1	2	3	4
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

Table 6.2: Distance matrix d of five cities.

The matrix is symmetric: for example $d[1][3] = d[3][1] = 25$. The diagonal is zero, because a city has no distance to itself. The length of a tour is the sum of the distances between consecutive cities plus the return from the last city to the first. For example, the tour $0 \to 1 \to 2 \to 3 \to 4 \to 0$ has length $10 + 35 + 30 + 40 + 25 = 140$.

6.2.2 The cost of an exact solution

The most direct exact method is to try all tours by brute force. There are $n!$ orderings of n cities, but many of them write the same tour in different ways. Because the tour is closed, the starting city can be fixed: $0 \to 1 \to 3 \to 2 \to 4 \to 0$ and $1 \to 3 \to 2 \to 4 \to 0 \to 1$ are the same tour. This leaves $(n - 1)!$ orders. In symmetric TSP a tour and its reverse also have the same length. So the number of distinct tours is $(n - 1)!/2$.

n	$(n - 1)!/2$	Time
5	12	instantly
10	181,440	≈ 0.2 milliseconds
15	$\approx 4.4 \cdot 10^{10}$	≈ 44 seconds
20	$\approx 6.1 \cdot 10^{16}$	≈ 1.9 years
25	$\approx 3.1 \cdot 10^{23}$	$\approx 10^{7}$ years

Table 6.3: Number of distinct tours in symmetric TSP and the time needed when 10^9 tours are tried per second.

Table 6.3 shows how fast the factorial grows. With five cities there are only 12 tours, and all of them can even be tried by hand. With ten cities the number of tours rises to 181,440; a computer finishes this in an instant. With twenty cities, even a computer that tries a billion tours per second runs for about two years. With twenty-five cities the time needed is about ten million years. Adding five cities multiplies the time by millions.

A smarter exact method is dynamic programming. The Held–Karp algorithm tabulates the values “the shortest path that starts at the origin, visits every city in the set S, and ends at j” and runs in $O(n^2 \cdot 2^n)$ time. For 25 cities this is about $2 \cdot 10^{10}$ operations, which is far better than the factorial. Even so, both the time and the $O(n \cdot 2^n)$ memory are exponential; at sixty cities this method also becomes unusable.

6.2.3 The nearest-neighbor heuristic

The **nearest-neighbor** heuristic follows the rule a person would apply without looking at a map: always go to the nearest city not yet visited. Its steps are as follows:

1. Choose a starting city and mark it as visited.
2. From the current city, go to the nearest unvisited city. This is a greedy choice.
3. Repeat the second step until all cities have been visited.
4. Return from the last city to the starting city, closing the tour.

The Java code from the slides implements these steps directly:

```
int[] nearestNeighbor(int[][] d, int start) {
  int n = d.length, total = 0, current = start;
  boolean[] visited = new boolean[n];
  int[] tour = new int[n + 1];
  tour[0] = start; visited[start] = true;
  for (int k = 1; k < n; k++) {
    int nearest = -1, minDist = Integer.MAX_VALUE;
    for (int j = 0; j < n; j++)
      if (!visited[j] && d[current][j] < minDist) {
        minDist = d[current][j]; nearest = j; }
    // go to the nearest city
    tour[k] = nearest; visited[nearest] = true;
    total += minDist; current = nearest;
  }
  // return to the start
  tour[n] = start; total += d[current][start];
  return tour;
}
```

The code takes the distance matrix `d` and the starting city `start`. The array `visited` records whether each city is already in the tour. The array `tour` has $n + 1$ cells, because the starting city appears in both the first and the last cell. `current` is the city where we are now, and `total` is the length of the path so far.

The outer loop runs from $k = 1$ to $n - 1$ and adds one city per round. The inner loop scans the row of the `current` city. Every city that is unvisited and closer than the smallest distance seen so far becomes the candidate `nearest`. Comparing with `<` also introduces a tie-breaking rule: if two cities are at the same distance, the one with the smaller index is chosen. When the inner loop ends, the nearest city is written into the tour, marked, its distance is added to the total, and it becomes the new `current`. After the outer loop, the edge returning from the last city to the start is added.

The cost analysis is short. The outer loop runs $n - 1$ times and the inner loop runs n times each time. The body of the inner loop takes constant time. The total time is $O(n^2)$. The extra memory is $O(n)$ for the arrays `visited` and `tour`. Since the input itself is an n^2-sized matrix, this running time is on the

order of the time needed to read the input. Nothing faster can be expected.

6.2.4 Code trace: five cities

Let `nearestNeighbor(d, 0)` be run with the matrix of Table 6.2. Initially `visited[0] = true`, `tour = [0]`, and `total = 0`. Table 6.4 shows each step of the outer loop.



Figure 6.3: The start of the nearest-neighbor trace. The distance matrix; the tour consists only of city 0.

`k`	`current`	**Distances scanned (unvisited cities)**	`nearest` (`minDist`)	`total`	`tour`
1	0	*d[0][1] = 10, d[0][2] = 15, d[0][3] = 20, d[0][4] = 25*	1 (10)	10	0, 1
2	1	*d[1][2] = 35, d[1][3] = 25, d[1][4] = 30*	3 (25)	35	0, 1, 3
3	3	*d[3][2] = 30, d[3][4] = 40*	2 (30)	65	0, 1, 3, 2
4	2	*d[2][4] = 35*	4 (35)	100	0, 1, 3, 2, 4
return	4	*d[4][0] = 25*	—	125	0, 1, 3, 2, 4, 0

Table 6.4: Trace of `nearestNeighbor(d, 0)`.

In the first step row 0 is scanned. Among the four candidates the smallest distance is 10, belonging to city 1. The tour becomes *0 → 1* and the total is 10. In the second step only the unvisited columns 2, 3, and 4 of row 1 are examined; column 0 is skipped because it is marked. The smallest value is 25, city 3 is chosen, and the total becomes 35. In the third step, from row 3, city 2 (30) is chosen over 4 (40); the total is 65. In the fourth step only one candidate remains: 4. The total becomes *65 + 35 = 100*. Finally we return from 4 to 0 and add *d[4][0] = 25*.

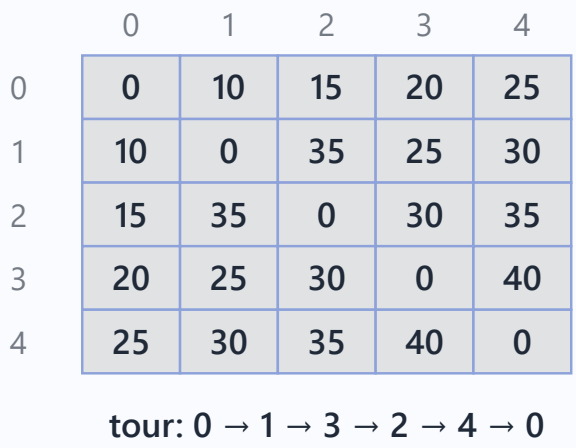


Figure 6.4: The end of the trace. The tour *0 → 1 → 3 → 2 → 4 → 0*, length 125.

The result is the tour *0 → 1 → 3 → 2 → 4 → 0* with length 125 (Figure 6.4). A verification script ran the same code from all five starting cities. The tour found from every start again has length 125. For example, starting from 2 gives the tour *2 → 0 → 1 → 3 → 4 → 2*: *15 + 10 + 25 + 40 + 35 = 125*.

6.2.5 Comparison with brute force

With five cities there are *(5 − 1)!/2 = 12* distinct tours. Once all of them have been tried, the result of nearest neighbor can be evaluated. Table 6.5 gives the twelve tours and their lengths.

Tour	Length	Tour	Length
0 → 1 → 3 → 2 → 4 → 0	125	0 → 1 → 2 → 3 → 4 → 0	140
0 → 1 → 3 → 4 → 2 → 0	125	0 → 1 → 2 → 4 → 3 → 0	140
0 → 1 → 4 → 2 → 3 → 0	125	0 → 2 → 1 → 3 → 4 → 0	140
0 → 1 → 4 → 3 → 2 → 0	125	0 → 2 → 1 → 4 → 3 → 0	140
0 → 2 → 3 → 1 → 4 → 0	125	0 → 3 → 1 → 2 → 4 → 0	140
0 → 2 → 4 → 1 → 3 → 0	125	0 → 3 → 2 → 1 → 4 → 0	140

Table 6.5: The twelve tours of the five-city example (start fixed at 0, a tour and its reverse counted as one).

The table gives two interesting results. First, half of the tours have length 125 and the other half length 140. The best length is 125, and six different tours reach this value. The tour found by nearest neighbor is one of them; that is, in this example the heuristic found the best result. This is luck and not the general case. Second, the matrix does not satisfy the **triangle inequality**: *d[1][2] = 35*, but going from 1 to 2 by way of 0 costs only *d[1][0] + d[0][2] = 10 + 15 = 25*. The direct path is longer than the detour. Such a matrix is called non-metric. This property is decisive for ratio guarantees.

The strengths and weaknesses of nearest neighbor can be summarized as follows. It is very simple and runs in $O(n^2)$ time. On random planar instances the tours it finds are on average about 25% above the best. It provides a good starting tour for local search. On the other hand, it has no constant approximation ratio. Rosenkrantz, Stearns, and Lewis showed that even when the triangle inequality holds, the ratio can reach the order of $\Theta(\log n)$: the upper bound is $O(\log n)$, and inputs approaching this bound can be constructed.

6.2.6 General TSP has no constant ratio

Nearest neighbor has no constant ratio, but perhaps a cleverer algorithm does. For general TSP, that is, with no condition at all on the distances, this hope is in vain. If P ≠ NP, there is no constant-ratio polynomial-time approximation algorithm for general TSP. The proof rests on the Hamiltonian cycle problem. Deciding whether a graph has a Hamiltonian cycle is NP-complete.

Theorem 6.1 (Inapproximability of general TSP)

If P ≠ NP, then for no constant $\rho \geq 1$ does general TSP have a polynomial-time ρ-approximation algorithm.

Proof. The proof is by contradiction. Suppose there is a polynomial-time ρ-approximation algorithm A. Let G be any graph with n vertices. From this graph we build a TSP instance as follows: the cities are the vertices of G; pairs joined by an edge in G get distance 1, and pairs not joined get distance $\rho n + 1$. The construction takes $O(n^2)$ time.

If G has a Hamiltonian cycle, that cycle is a tour all of whose edges have length 1. The best tour has length n. The tour of A has length at most ρn. This tour cannot contain even a single heavy edge, because that edge alone would cost $\rho n + 1 > \rho n$. Hence the tour of A uses only edges of G and is a Hamiltonian cycle of G.

If G has no Hamiltonian cycle, every tour contains at least one heavy edge and has length greater than ρn. So does the tour of A.

Therefore, running A and checking whether the length of its tour exceeds ρn tells us in polynomial time whether G has a Hamiltonian cycle. Since the Hamiltonian cycle problem is NP-complete, this means P = NP, contradicting the assumption. □

The strength of the construction comes from the heavy edges: even a single heavy edge ruins the ratio. But this construction violates the triangle inequality badly: while the heavy edge u–v has length $\rho n + 1$, the path through a common neighbor is only 2. For TSP with distances satisfying the triangle inequality the construction is invalid, and constant-ratio algorithms exist.

6.2.7 MST-based 2-approximation for metric TSP

Definition 6.5

A TSP instance is called **metric TSP** if the inequality $d_{ij} \le d_{ik} + d_{kj}$ holds for all cities i, j, k. This inequality is called the **triangle inequality**: going directly between two cities is never longer than stopping at a third city on the way.

Most real-life distances are metric. Euclidean distance between points in the plane, shortest-path distances in a road network, and flight times are of this kind. Metric TSP is still NP-hard, but a constant-ratio algorithm is now possible. The simplest one is based on the minimum spanning tree (MST). A minimum spanning tree is the tree connecting all the vertices of a graph whose total weight is smallest. It is found in $O(n^2)$ or $O(E \log V)$ time with Prim's or Kruskal's algorithm.

The algorithm has three steps:

1. Build a minimum spanning tree T in the complete graph of the cities.
2. Traverse T depth-first from a root city. Every edge is used once going and once returning. The length of this closed walk is $2 \cdot w(T)$.
3. Skip the cities already seen in the walk (**shortcutting**). What remains is the preorder of the cities; appending the starting city at the end closes the tour.

The example in the slides has seven cities in the plane, with Euclidean distances. A verification script recovered the coordinates of the cities from the drawing in the figure: A (10, 20), B (18, 68), C (42, 86), D (50, 46), E (76, 78), F (92, 42), G (62, 10). All distances are computed from these coordinates.

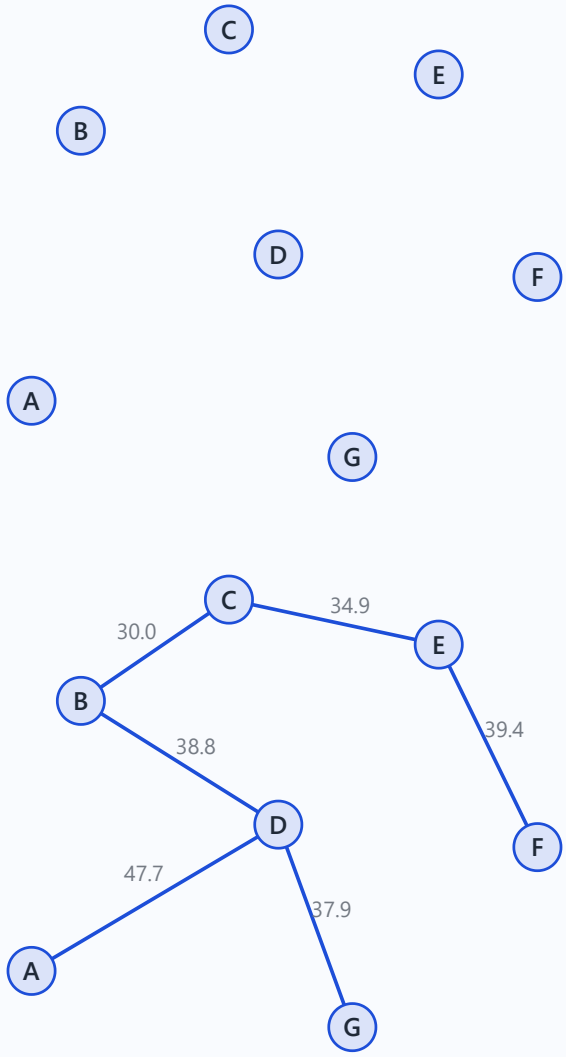


Figure 6.5: Left, the seven cities. Right, the minimum spanning tree built with Prim's algorithm; the edge weights are Euclidean distances and their total is $w(T) = 228.8$.

If Prim's algorithm is started from A, the edges are added in this order: A–D (47.7), D–G (37.9), D–B (38.8), B–C (30.0), C–E (34.9), E–F (39.4). The total weight is $w(T) = 228.8$ (Figure 6.5). It is easy to see why the tree is a lower bound for the best tour. If any edge is deleted from the best tour, a path connecting all the cities remains. A path is also a spanning tree. Since the minimum spanning tree is the lightest of all spanning trees, it cannot be longer than this path. Because edge weights are nonnegative, $w(T) \leq OPT$ follows.

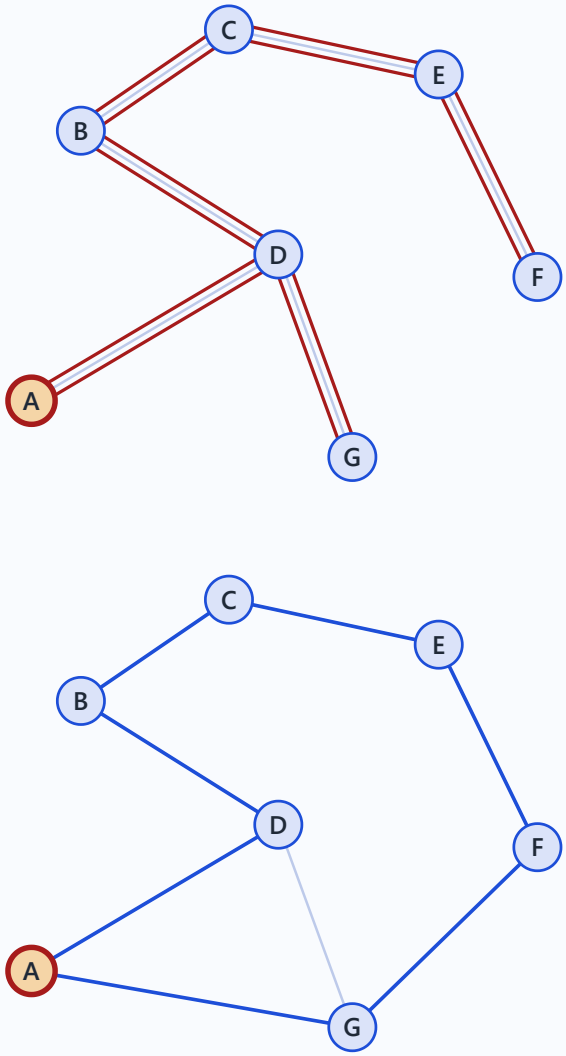


Figure 6.6: Left, the double traversal of the tree; every edge is used once going and once returning, and the walk has length $2 \cdot w(T) = 457.6$. Right, the tour with shortcuts $A \to D \to B \to C \to E \to F \to G \to A$, length 287.7.

In the second step the tree is traversed depth-first from A. The traversal in the slides visits B before G among the children of D: $A \to D \to B \to C \to E \to F \to E \to C \to B \to D \to G \to D \to A$. Every tree edge is used twice, and the length of the walk is $2 \cdot 228.8 = 457.6$. This walk is not a valid tour, because D, B, C, and E are passed more than once.

In the third step the repeated cities are skipped. The walk is read from the left, and every city seen for the first time is written into the tour: A, D, B, C, E, F; then E, C, B, D are skipped; G is written; D and A are skipped. Appending A at the end gives the tour $A \to D \to B \to C \to E \to F \to G \to A$. Its length is computed step by step in Table 6.6.

Edge	Length	Note
A → D	47.7	tree edge

Edge	Length	Note
D → B	38.8	tree edge
B → C	30.0	tree edge
C → E	34.9	tree edge
E → F	39.4	tree edge
F → G	43.9	shortcut: replaces the walk F → E → C → B → D → G (181.0)
G → A	53.0	shortcut: replaces the walk G → D → A (85.6)
Total 287.7		

Table 6.6: The edges of the tour with shortcuts. The two edges that are not tree edges are shortcuts.

The table shows that the shortcuts never lengthen the walk. The walk from F to G passes through five tree edges and costs 181.0, while the direct edge F–G is 43.9. What guarantees this is the triangle inequality: skipping an intermediate city, that is, using d_{ij} instead of $d_{ik} + d_{kj}$, does not increase the length. Applying the skip repeatedly, several intermediate cities can be skipped at once.

Theorem 6.2 (MST-based 2-approximation)

In metric TSP, the tour built from the preorder of a minimum spanning tree is at most twice as long as the best tour: $tour \le 2 \cdot OPT$.

Proof. The proof combines two inequalities. The first is the lower bound: deleting an edge from the best tour leaves a spanning path, this path is a spanning tree, and its weight is at least $w(T)$. Since the weight of the deleted edge is nonnegative, $w(T) \le OPT$.

The second is the upper bound. Traversing the tree twice uses every edge exactly twice, and its length is $2 \cdot w(T)$. The tour is obtained from this walk by shortcuts. Each shortcut replaces a subwalk between two cities with the direct edge. Applying the triangle inequality repeatedly, the direct edge cannot be longer than the subwalk. Hence $tour \le 2 \cdot w(T)$. Combining the two inequalities gives $tour \le 2 \cdot w(T) \le 2 \cdot OPT$. □

In the example, brute force tries all $6!/2 = 360$ tours. The best tour is $A \to B \to C \to E \to F \to G \to D \to A$ with length 282.5. The 287.7 found by the algorithm is only about 1.02 times the best. The guarantee is 2, but the practical result is much better. The bounds in the proof can also be checked: $w(T) = 228.8 \le 282.5 = OPT$ and $287.7 \le 457.6 = 2 \cdot w(T)$.

Example 6.2

In a depth-first traversal the algorithm does not specify the order in which the children of a vertex are visited. The traversal in the slides takes B before the other child G of D. Prim's algorithm, however, adds G to the tree before B (D–G 37.9, D–B 38.8). If the children were traversed in the order they were added to the tree, the walk would be $A \to D \to G \to D \to B \to C$

$\to E \to F \to E \to C \to B \to D \to A$. The tour with shortcuts comes out as $A \to D \to G \to B \to C \to E \to F \to A$. Its length is $47.7 + 37.9 + 72.8 + 30.0 + 34.9 + 39.4 + 84.9 = 347.7$. This tour is noticeably longer than the one in the slides, but it is still below the bound $2 \cdot w(T) = 457.6$. The ratio is $347.7 / 282.5 \approx 1.23$. The guarantee is independent of the order of the children; the actual result is sensitive to it.

The running time of the algorithm is determined by building the tree. On the complete graph, Prim's algorithm with an array-based implementation takes $O(n^2)$. The depth-first traversal and the shortcut step are $O(n)$. The total time is $O(n^2)$, the same order as nearest neighbor, but now there is a proven ratio.

Common mistake

Forgetting the triangle inequality. That the shortcut step does not lengthen the walk holds only for metric instances. In the matrix of Table 6.2, $d[1][2] = 35$ is greater than $d[1][0] + d[0][2] = 25$. In such an instance a shortcut from 1 to 2 lengthens the walk, and the bound $tour \le 2 \cdot OPT$ loses its validity. The algorithm still produces a tour, but without a guarantee.

6.2.8 Christofides and other bounds

The loss of the MST-based algorithm comes from one place: every tree edge is walked twice. In 1976 Christofides replaced this double walk with a cheaper structure. The vertices of odd degree in the tree are found; their number is always even. A minimum-weight perfect matching is built among these vertices. In the union of the tree and the matching every vertex has even degree, so an Euler tour exists. An Euler tour is a closed walk that passes through every edge exactly once. Shortcutting is applied to this walk. The weight of the matching is at most $OPT / 2$; the total is $w(T) + OPT/2 \le 1.5 \cdot OPT$. Because of the matching step, the running time of the algorithm is $O(n^3)$.

Christofides's bound of 1.5 could not be improved for more than forty years. In 2020 Karlin, Klein, and Oveis Gharan reduced the ratio to $3/2 - \varepsilon$ with a randomized algorithm; here ε is a very small constant greater than 10^{-36}. It makes no practical difference, but it shows that 1.5 can be beaten. Table 6.7 collects the main known results for TSP.

Case	Ratio	Note
General TSP	none	If P ≠ NP, there is no polynomial-time ρ-approximation for any constant ρ; one would solve the Hamiltonian cycle problem.
Metric TSP, MST-based	2	MST + preorder traversal + shortcuts; $O(n^2)$.
Metric TSP, Christofides	1.5	MST + minimum perfect matching on the odd-degree vertices + Euler tour (1976); $O(n^3)$.
Euclidean TSP	$1 + \varepsilon$	A PTAS exists for points in the plane (Arora, Mitchell).

Case	Ratio	Note
Nearest neighbor	$\Theta(\log n)$	Guarantees no constant ratio even in the metric case.

Table 6.7: Known approximation results for TSP, by whether the triangle inequality holds.

6.3 Local search: 2-opt

Nearest neighbor and the MST-based algorithm build a tour from scratch. Local search instead assumes that a tour is already in hand and improves it step by step. The starting tour can be random or can come from another heuristic.

6.3.1 The idea of local search

Definition 6.6

Local search is a search method that starts from a solution and improves it by small changes. The solutions reachable from a given solution by a single change are called its **neighbors**. A solution none of whose neighbors is better than itself is called a **local optimum**.

Local search resembles a climber ascending in fog. The climber sees only the area around the current position. The climber takes whichever step leads higher and stops when no direction goes up. The place where the climber stops is a peak, but perhaps not the highest peak of the mountain. Local search likewise stops at a local optimum. This point need not be the global optimum, that is, the best of all solutions.

6.3.2 The 2-opt move and its gain

Remove two edges from a tour: (a, b) and (c, d). The tour falls into two pieces. There is exactly one way to join the pieces into a tour again that differs from the old connections: connect a to c and b to d. After this connection the subpath between b … c is traversed in the opposite direction. This change is called a **2-opt move**.

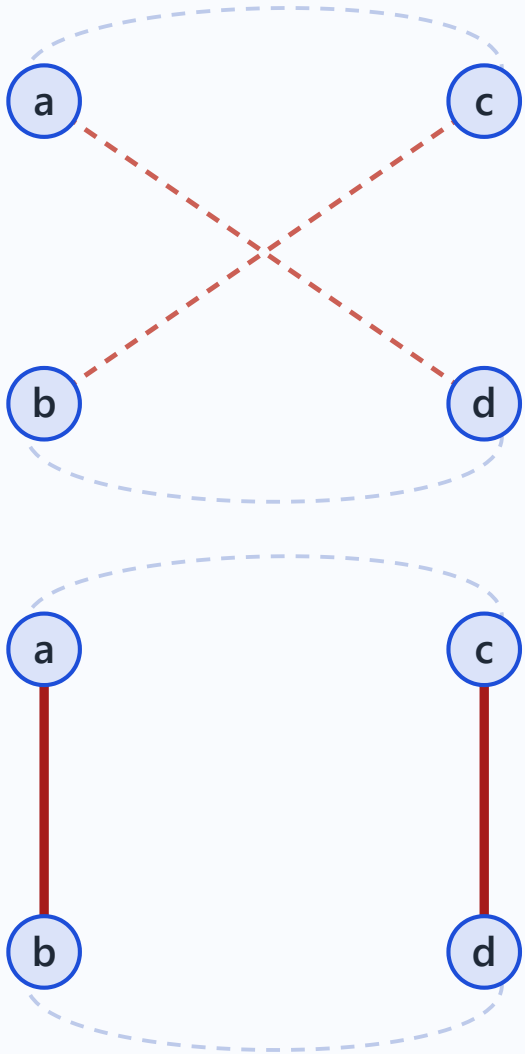


Figure 6.7: The 2-opt move. On the left the edges *(a, b)* and *(c, d)* cross (dashed). On the right they are replaced by *(a, c)* and *(b, d)*; the path in between is reversed. The dotted arc shows the rest of the tour.

Whether the move shortens the tour can be told by looking at just four edges. The inner edges of the reversed subpath stay the same, because in symmetric TSP the length of an edge does not depend on its direction. Only the two edges removed and the two edges added change. The gain of the move is written as follows:

$$\Delta = d(a, c) + d(b, d) - d(a, b) - d(c, d)$$

If $\Delta < 0$, the new tour is shorter than the old one by $|\Delta|$ and the move is made. If $\Delta \geq 0$, the move does not shorten the tour. The gain is computed in $O(1)$ time. The tour length is not re-summed.

In the Euclidean plane 2-opt has a geometric interpretation: two crossing edges can always be removed. Let the segments (a, b) and (c, d) cross at a point X. By the triangle inequality, $d(a, c) \leq d(a, X) + d(X, c)$ and $d(b, d) \leq d(b, X) + d(X, d)$. The sum of the right-hand sides is exactly $d(a, b) + d(c, d)$.

Hence $\Delta \le 0$, and the inequality is strict unless the points are collinear. Consequently, no edge of a Euclidean tour that is a 2-opt local optimum crosses another.

Example 6.3

Let the points be $a\ (0, 0)$, $b\ (4, 3)$, $c\ (4, 0)$, $d\ (0, 3)$, and let the tour contain the edges (a, b) and (c, d). These two edges are the diagonals of a rectangle and cross each other. Each has length $\sqrt{(16 + 9)} = 5$. The new edges (a, c) and (b, d) are the horizontal sides of the rectangle, of length 4. The gain is $\Delta = 4 + 4 - 5 - 5 = -2$. The tour becomes 2 units shorter and the move is made.

When no pair of edges remains that would give a 2-opt move, the tour is a **2-opt local optimum**. Such a tour is usually a good one, but it need not be the global optimum. To reach a better tour it may be necessary to change more than two edges at once.

6.3.3 Code and cost

The Java code from the slides follows the "first improvement" strategy. It scans the tour and immediately applies the first move whose gain is negative:

```
void twoOpt(int[] tour, double[][] d) {
  int n = tour.length;
  boolean improved = true;
  while (improved) {
    improved = false;
    for (int i = 0; i < n - 1; i++)
      for (int j = i + 2; j < n; j++) {
        int a = tour[i], b = tour[i + 1];
        int c = tour[j], e = tour[(j + 1) % n];
        double delta = d[a][c] + d[b][e]
                     - d[a][b] - d[c][e];
        if (delta < -1e-9) {   // shorter
          reverse(tour, i + 1, j);
          improved = true;
        }
      }
  }
}
```

The tour is kept in the array `tour` of city numbers and is closed: there is an edge from the last city to the first. The flag `improved` records whether at least one move was made during a scan. The outer `while` loop continues until a scan ends without making any move.

The inner loops run over pairs of edges. Edge `i` is the edge between `tour[i]` and `tour[i + 1]`; edge `j` is between `tour[j]` and `tour[(j + 1) % n]`. The modulo operation makes sure the last edge, the one returning from the last city to the first, is also taken into account. Starting `j` at `i + 2` prevents the choice of two adjacent edges. Adjacent edges share an endpoint, and a 2-opt move on them does not change the tour. Still, in the case `i = 0` and `j = n - 1` the two edges meet at the start of the tour: `e = tour[0] = a`, and the gain always comes out zero. The code does not exclude this case separately; since the gain is zero, no move is made.

The gain is computed in the variable `delta`. The comparison is made with `delta < -1e-9` rather than `delta < 0`. Because the distances are `double`, rounding errors can turn a difference that should be zero into a tiny negative number such as -10^{-16}. Such a "phantom gain" can make the algorithm go back and forth between the same two tours forever. The small threshold prevents this.

When a move is made, `reverse(tour, i + 1, j)` is called. This helper method is not given in the slides; it reverses in place the elements between `tour[i + 1]` and `tour[j]`, both ends included. It can be written as a loop that swaps elements from the two ends toward the middle:

```
void reverse(int[] t, int i, int j) {
  while (i < j) {
    int x = t[i]; t[i] = t[j]; t[j] = x;
    i++; j--;
  }
}
```

The cost can be examined at three levels. A scan looks at about $n(n - 3)/2$, that is, $O(n^2)$ pairs of edges. The gain of each pair is computed in $O(1)$. For a pair where a move is made, the reversal takes $O(n)$. How many scans will be made is not known in advance. In the worst case the number of scans can be exponential; theoretically, bad inputs can be constructed. In practice a few dozen scans usually suffice.

6.3.4 Code trace: four points

The trace in the slides works with four points: 0 (0, 0), 1 (3, 4), 2 (3, 0), and 3 (0, 4). The points are the corners of a 3 × 4 rectangle. The distances are Euclidean: the horizontal sides are 3, the vertical sides 4, and the diagonals 5 units. The starting tour is $0 \to 1 \to 2 \to 3 \to 0$. This tour uses both diagonals: $d(0, 1) + d(1, 2) + d(2, 3) + d(3, 0) = 5 + 4 + 5 + 4 = 18$.

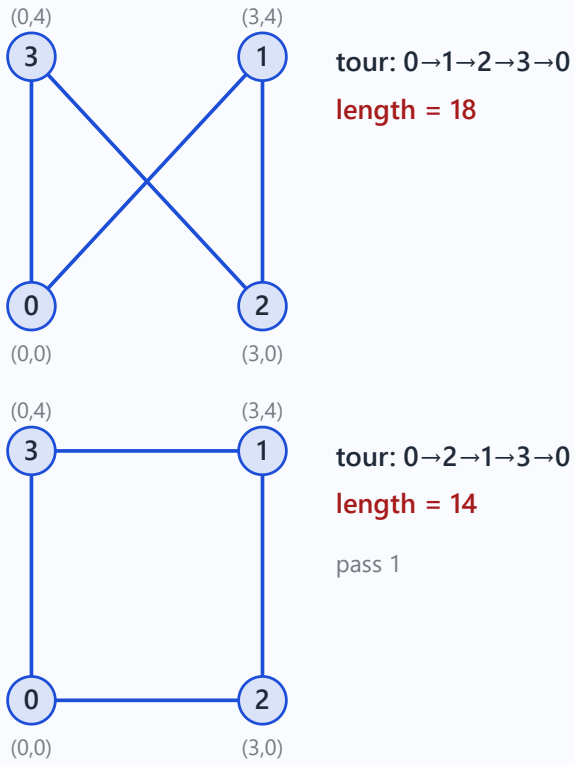


Figure 6.8: 2-opt on four points. Left, the starting tour $0 \to 1 \to 2 \to 3 \to 0$, length 18; the edges (0, 1) and (2, 3) cross. Right, the tour after the first move $0 \to 2 \to 1 \to 3 \to 0$, length 14.

Pass	`i, j`	`a, b, c, e`	`delta`	Result
1	0, 2	0, 1, 2, 3	*3 + 3 − 5 − 5 = −4*	move: `tour = [0, 2, 1, 3]`, length 14
1	0, 3	0, 2, 3, 0	*4 + 3 − 3 − 4 = 0*	adjacent edges (`e = a`); no move
1	1, 3	2, 1, 3, 0	*5 + 5 − 4 − 4 = 2*	no move; `improved = true`
2	0, 2	0, 2, 1, 3	*5 + 5 − 3 − 3 = 4*	no move
2	0, 3	0, 2, 3, 0	*4 + 3 − 3 − 4 = 0*	no move
2	1, 3	2, 1, 3, 0	*5 + 5 − 4 − 4 = 2*	no move; `improved = false`, the loop ends

Table 6.8: Trace of `twoOpt` on four points ($n = 4$).

Table 6.8 gives the trace. Since $n = 4$, `i` ranges from 0 to 2 and `j` from `i + 2` to 3. The valid pairs are (0, 2), (0, 3), and (1, 3). In the first pass the pair (0, 2) removes the diagonals *(0, 1)* and *(2, 3)* and proposes the edges *(0, 2)* and *(1, 3)*. These are the short edges of 3 units: *delta = 3 + 3 − 5 − 5 = −4*. The move is made, `reverse(tour, 1, 2)` swaps 1 and 2, and the tour becomes `[0, 2, 1, 3]`. The new length is *3 + 4 + 3 + 4 = 14*.

The scan continues on the same tour. For the pair (0, 3) the two edges share an endpoint (`e = a`) and the difference is zero. The pair (1, 3) brings the diagonals back and gives *+2*. Because a move was made in the first pass, the scan is repeated; in the second pass no pair gives a negative gain and the loop ends.

The result is the tour `[0, 2, 1, 3]` with length 14. This tour is the perimeter of the rectangle. With four points there are only three distinct tours (*3!/2 = 3*): the perimeter (14) and two tours using diagonals (18 and 16). In this example the 2-opt local optimum is also the global optimum.

6.3.5 2-opt on twenty-eight cities

A larger example shows the power of 2-opt more clearly. In the slides 28 cities are placed at random in the range 0–100 and a random starting tour is chosen. The tour edges cross each other constantly, and the length is 1269.8. A verification script read the city coordinates from the drawing in the slides and ran the same `twoOpt` code. All the values in the slides match this run exactly.

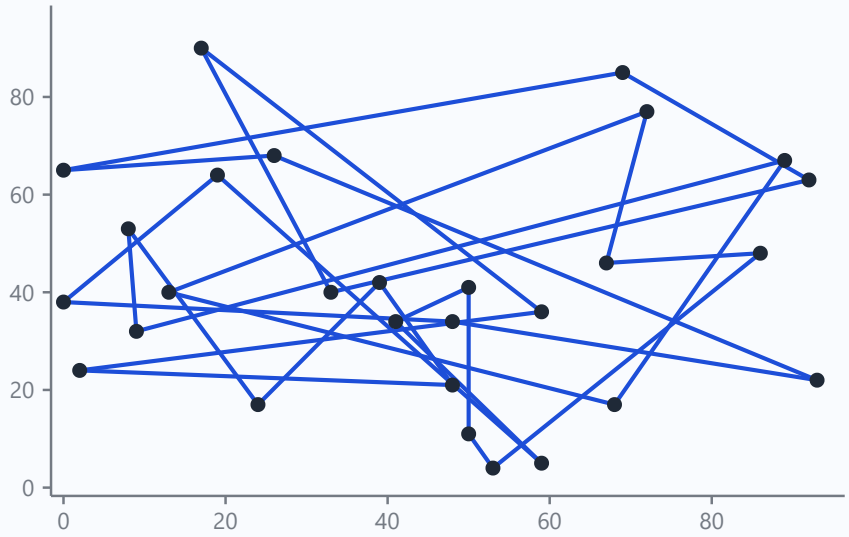


Figure 6.9: 28 cities and the random starting tour; length 1269.8. The edges cross one another at 66 points.

Move	Gain *Δ*	Tour length
start	—	1269.8
1	−36.9	1232.9
2	−8.9	1224.0
4	−27.6	1187.3
8	−32.6	1111.7
14	−14.9	1005.2
22	−37.0	852.8
32	−11.8	595.9
48 (last)	−13.2	430.8

Table 6.9: 2-opt with the first-improvement strategy on 28 cities; gain and tour length after selected moves.

The values in Table 6.9 show two things. First, the gain of an individual move is small. The gains in the table range from 9 to 37 units; even the largest of the 48 moves (70.4) does not exceed 10% of the tour length at that moment. The first-improvement strategy does not look for large gains; it takes the first gain it finds. Second, these small gains add up. After 48 moves the tour drops to 430.8. This is about 34% of the starting length.

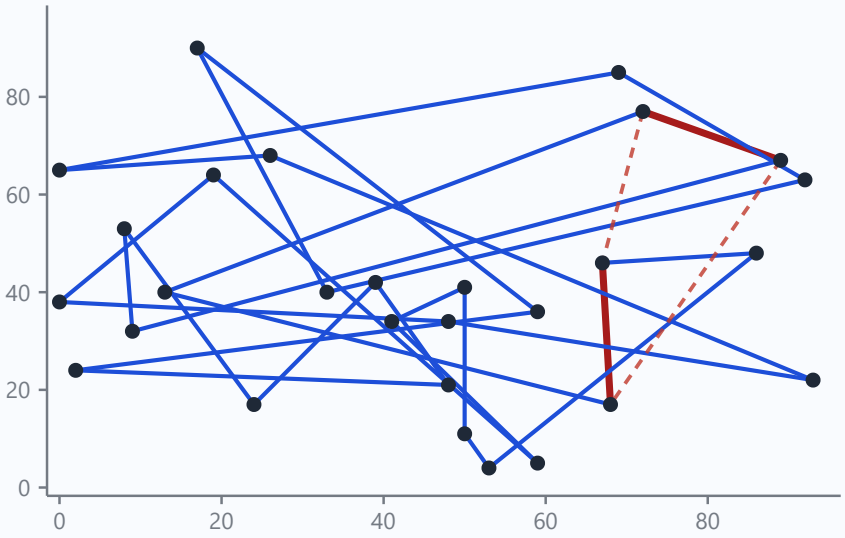


Figure 6.10: The first improving move. The dashed edges are removed and the bold edges added; $\Delta = -36.9$, new length 1232.9.

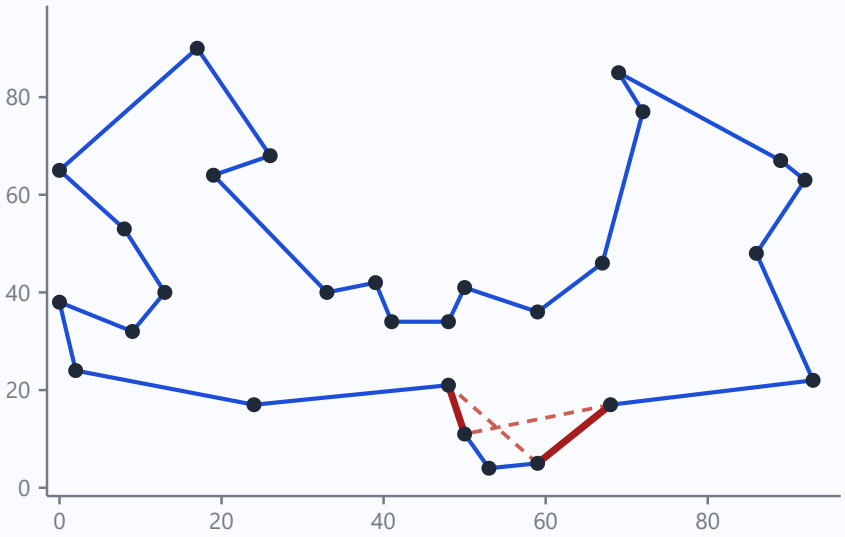


Figure 6.11: The 48th and last move. No 2-opt move shortens the tour any more; the tour is a local optimum, length 430.8. No crossing edges remain.

The final tour has no crossing edges (Figure 6.11). This is the result expected from the geometric interpretation of the 2-opt move: the edges of a Euclidean tour that is a 2-opt local optimum do not cross each other. But the absence of crossings does not mean the tour is the best. There are many tours without crossings, and their lengths differ from one another.

The strategy and the starting tour affect the result. A verification script made two more experiments. The "best improvement" strategy examines all pairs in each scan and applies the one with the largest gain. Starting from the same random tour, it stops after 23 moves and finds a local optimum of length 441.9. First improvement started from the nearest-neighbor tour (518.6), on the other hand, stops after only 10 moves and reaches 449.7. In this example the random start took longer but arrived at a better local optimum. No general rule can be drawn. The result of local search depends on the start and the strategy; in practice it is run several times from different starts and the best result is kept.

6.3.6 Escaping local optima: metaheuristics

The basic problem of local search is that it gets stuck in the first local optimum. An algorithm that accepts only improving moves cannot climb out once it has fallen into a pit. Metaheuristics overcome this problem in different ways: they sometimes accept worsening moves too, they improve many solutions together at the same time, or they learn from the past.

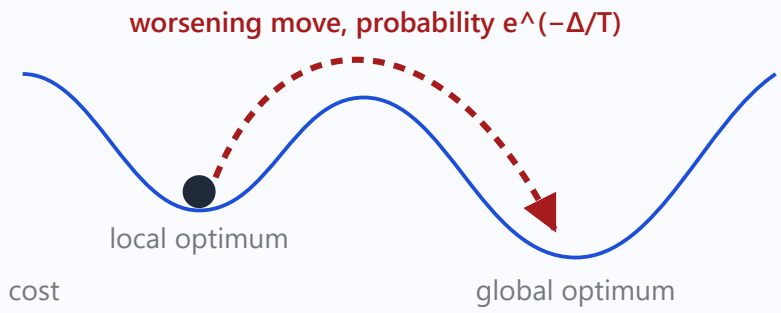


Figure 6.12: A cost landscape. Local search, which accepts only improving moves, stays in the pit on the left. Simulated annealing can climb over the hill and reach the global optimum by accepting a worsening move with probability $e^{-\Delta/T}$.

Simulated annealing takes its name from the annealing of metals. If hot metal is cooled slowly, its atoms settle into an ordered, low-energy structure. At each step the algorithm picks a random neighbor. If the neighbor is better, the move is made. If it is worse, that is, it increases the cost by $\Delta > 0$, the move is accepted anyway with probability $e^{-\Delta/T}$. Here T is a parameter called the "temperature", and it is lowered slowly. A typical setting starts with $T_0 = 100$ and sets $T \leftarrow 0.997 \cdot T$ at each step.

Temperature T	Δ / T	Acceptance probability
100	0.05	≈ 0.95
10	0.5	≈ 0.61

Temperature T	Δ / T	Acceptance probability
1	5	≈ 0.0067

Table 6.10: Acceptance probability $e^{-\Delta/T}$ of a move that worsens the solution by $\Delta = 5$.

Table 6.10 shows the effect of temperature. At high temperature even worsening moves are accepted almost always, and the search roams freely over the landscape. As the temperature falls, moves become more selective. At $T = 1$, a move that worsens the solution by 5 units is accepted about once in 150 trials. In the end the algorithm turns into a local search that accepts almost only improving moves. The cooling rate can also be computed. The inequality $100 \cdot 0.997^k < 1$ gives $k > \ln 100 / (-\ln 0.997) \approx 1532.8$; the temperature falls below 1 in 1533 steps.

Genetic algorithms work with a population of tours. Shorter tours are selected more often, the selected tours are combined by crossover, and mutation preserves diversity. A typical setting is a population of 100 individuals, crossover with a random cut point, and a mutation probability of 0.05.

Ant colony optimization is inspired by real ants. Artificial ants that build short tours leave more pheromone on edges; later ants prefer edges with dense pheromone, and the pheromone evaporates over time.

All three methods are widely used in TSP, vehicle routing, job scheduling, and assignment problems. None of them guarantees an approximation ratio. The result is sensitive to the starting solution, the parameters, and the seed of the random number generator.

Common mistake

Assuming the result of a metaheuristic is exact. It is theoretically known that if the temperature is lowered slowly enough, at a logarithmic rate, simulated annealing converges in probability to the global optimum. Every cooling schedule used in practice, however, is finite and gives no guarantee. The same algorithm is run several times with different seeds and parameters, and the best result is taken. To say how close the result found is to the best, a lower bound is needed, for example the weight of the minimum spanning tree.

6.4 Job scheduling

Job scheduling is the planning of jobs on given resources (workers, machines, processors) as efficiently as possible. The goal is to minimize the completion time of the jobs, or to find the best plan by criteria such as profit, lateness, and deadlines. This section separates two problems and examines in detail the one that is the subject of approximation algorithms.

6.4.1 Two scheduling problems

In the **machine scheduling** problem there are n jobs and m identical machines. Job j has processing time p_j. Each job runs on a single machine without interruption. The load of a machine is the sum of the processing times of the jobs assigned to it. The moment the last job finishes, that is, the load of the most loaded machine, is denoted C_{max}. This value is called the **makespan**. The goal is to minimize C_{max}. The problem is NP-hard for $m \geq 2$; even the case $m = 2$ is a version of the problem of splitting the jobs into two sets with equal sums.

In the **job sequencing with deadlines** problem every job takes one unit of time and has a profit and a deadline. The total profit of the jobs that finish before their deadlines must be maximized. This problem is solved optimally by a greedy algorithm that takes the jobs in decreasing order of profit and places each in the latest free slot before its deadline.



Figure 6.13: Two scheduling problems. Left, jobs distributed over three machines, with makespan $C_{max} = 5$. Right, unit-time jobs placed in four time slots; job 2, whose deadline is 1, does not fit, and the total profit is 90.

6.4.2 List scheduling

Definition 6.7

List scheduling takes the jobs in the given order and assigns the next job to the machine that becomes free earliest at that moment, that is, the least loaded machine. The algorithm is due to Graham (1966).

List scheduling is an online algorithm: when assigning a job it does not need to know the jobs that come later. It can therefore be used directly in settings where jobs arrive over time. The example in the slides shows this case. There are three identical machines M1, M2, and M3. J1 (length 2), J2 (length 8), and J3 (length 6) are ready at $t = 0$; J4 (length 3) arrives at $t = 3$.

Event	Decision	Reason
J1 (2), $t = 0$	M1, [0, 2]	All machines are idle; the first machine is chosen.
J2 (8), $t = 0$	M2, [0, 8]	M1 is busy until $t = 2$.
J3 (6), $t = 0$	M3, [0, 6]	M1 is busy until $t = 2$, M2 until $t = 8$.
$t = 2$	M1 idle	No job is waiting; M1 stays idle until $t = 3$.

Event	Decision	Reason
J4 (3), $t = 3$	M1, [3, 6]	M1 is idle; M2 is busy until $t = 8$, M3 until $t = 6$.

Table 6.11: A list scheduling example with arrival times.

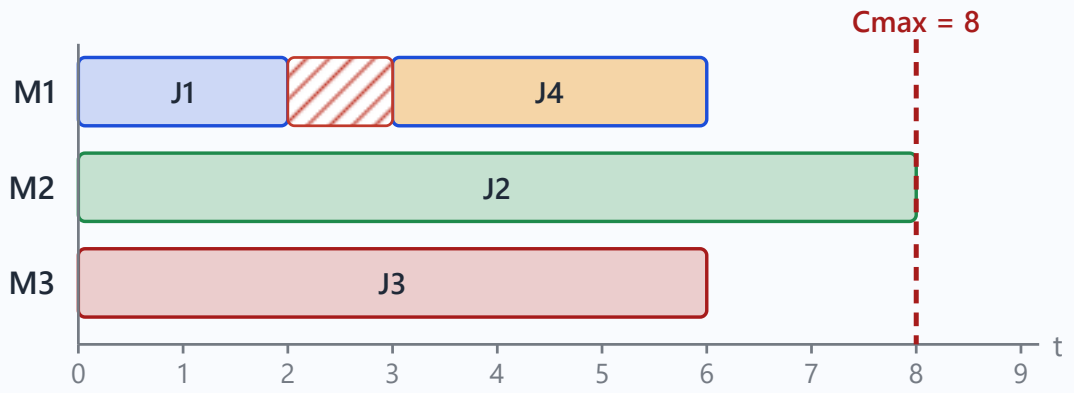


Figure 6.14: The result of list scheduling. On M1 there is a gap of one unit (hatched) between J1 and J4; the makespan is the load of M2: $C_{max} = 8$.

After all jobs are assigned, the makespan is 8 and belongs to M2 (Figure 6.14). This schedule is optimal, because no schedule can finish before the longest job. J2 alone takes 8 units, so the best makespan is at least 8. In general, however, list scheduling does not find the optimum; it guarantees only the ratio $2 - 1/m$.

If the least loaded machine is found by scanning an array, the time is $O(nm)$; if the machines are kept in a min-heap, it is $O(n \log m)$.

6.4.3 The ratio of list scheduling

The ratio proof rests on two lower bounds for the optimal solution. One concerns the workload. The total work is $\Sigma\, p_j$ and must be spread over m machines. Even in the best schedule some machine has a load at least equal to the average: $OPT \geq \Sigma\, p_j / m$. The other concerns the longest job. Each job runs on one machine without interruption; the longest job alone takes $\max p_j$: $OPT \geq \max p_j$.

Theorem 6.3 (Graham)

If all jobs are ready at the start, list scheduling produces, whatever the order of the jobs, a schedule satisfying $C_{max} \leq (2 - 1/m) \cdot OPT$.

Proof. Let j be the job that finishes last in the schedule and let it start at time s_j. Then $C_{max} = s_j + p_j$. At the moment j was assigned it went to the least loaded machine; so at that moment every machine had a load of at least s_j. In other words, before s_j all machines were busy; if a machine had been idle, j

would have started earlier. This busy time comes from jobs other than j: $m \cdot s_j \le \Sigma\, p - p_j$, that is, $s_j \le (\Sigma\, p - p_j) / m$.

$$C_{max} = s_j + p_j \le \Sigma\, p / m + (1 - 1/m) \cdot p_j \le OPT + (1 - 1/m) \cdot OPT = (2 - 1/m) \cdot OPT$$

The last step uses the two lower bounds: $\Sigma\, p / m \le OPT$ and $p_j \le max\, p \le OPT$. □

That the ratio is tight, that is, that no better bound can be proved, is shown by an example. Let $m = 2$ machines and three jobs of lengths 1, 1, 2 in this order. List scheduling gives the first job to M1 and the second to M2; for the third job both machines have load 1, and the job goes to M1. The loads become 3 and 1; $C_{max} = 3$. The best schedule puts the two short jobs on one machine and the long job on the other and finishes at 2. The ratio is $3/2 = 2 - 1/2$; the bound in the theorem is reached exactly.

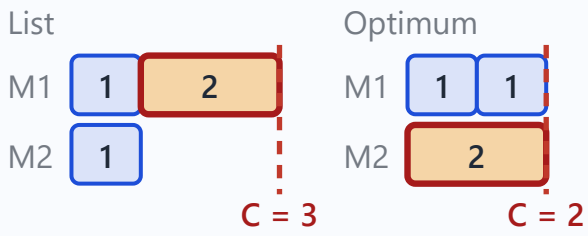


Figure 6.15: The tight example, $m = 2$ and jobs 1, 1, 2. List scheduling gets $C = 3$, the best schedule $C = 2$.

The example generalizes to every m: let $m(m - 1)$ jobs of 1 unit come first, then a single job of m units. The short jobs spread evenly over the machines and each machine gets load $m - 1$. The long job is added to one of them: $C_{max} = 2m - 1$. The best schedule gives the long job a machine of its own and spreads the short jobs over the remaining $m - 1$ machines, m each: $OPT = m$. The ratio is $(2m - 1)/m = 2 - 1/m$. For $m = 3$, six jobs of 1 unit and one job of 3 units give the ratio $5/3$.

6.4.4 Longest processing time first (LPT)

The tight examples show the same weakness: when a long job is left to the end, a single machine keeps working while the other machines have long since finished. The fix is simple: put the long jobs first. The **longest processing time first** (LPT) rule sorts the jobs in decreasing order of processing time and then applies list scheduling. The jobs left for the end are short; they correct the imbalance.

In the example in the slides the jobs are 2, 3, 4, 6, 2, 2 and $m = 3$. The total work is 19, so the best makespan is at least $\lceil 19/3 \rceil = 7$. List scheduling in the given order first distributes 2, 3, and 4 over the three machines. The job of 6 units goes to the least loaded machine, M1 with load 2, whose load becomes 8. The remaining two short jobs are added to M2 and M3. $C_{max} = 8$. The LPT order is 6, 4, 3, 2, 2, 2. The long jobs spread over the three machines first; the short jobs are then added to the least loaded machines and the loads become 6, 6, 7. $C_{max} = 7$, which is optimal because it equals the lower bound.

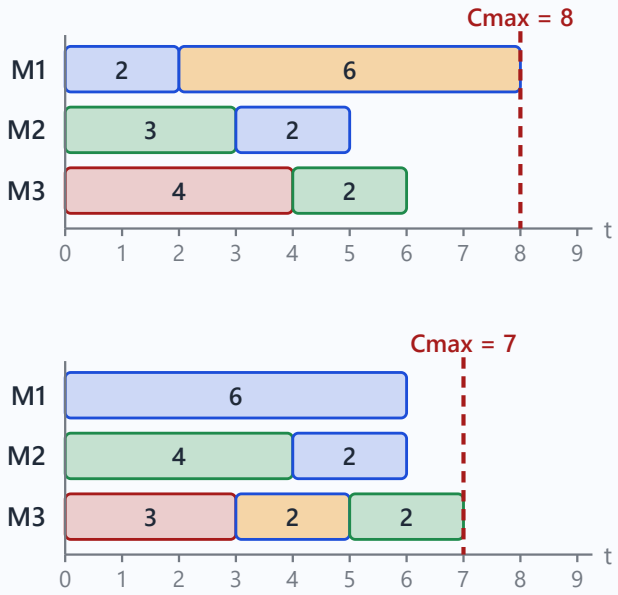


Figure 6.16: The same jobs, two orders. Left, list scheduling in the given order, $C_{max} = 8$. Right, the LPT order 6, 4, 3, 2, 2, 2, $C_{max} = 7 = OPT$.

Theorem 6.4 (Graham, 1969)

The LPT rule produces a schedule satisfying $C_{max} \le (4/3 - 1/(3m)) \cdot OPT$. Its running time is $O(n \log n)$ because of the sorting.

Proof. The outline of the proof is as follows. Suppose there is a counterexample that exceeds the bound, and take one with the fewest jobs. In this instance the job j that finishes last is the shortest job. Otherwise the jobs after j could be deleted, C_{max} would not change, OPT would not increase, and a smaller counterexample would be found.

The inequality from the list scheduling proof applies here too: $C_{max} \le \Sigma\, p / m + (1 - 1/m) \cdot p_j \le OPT + (1 - 1/m) \cdot p_j$. If $p_j \le OPT/3$, the right-hand side is $OPT + (1 - 1/m) \cdot OPT/3 = (4/3 - 1/(3m)) \cdot OPT$ and the bound holds. So in the counterexample $p_j > OPT/3$. Since j is the shortest job, all jobs are longer than $OPT/3$. In the best schedule at most two jobs fit on a machine. In this case it can be shown directly that LPT finds the best schedule: the longest m jobs are placed one per machine, and the rest are matched so that the shortest job comes next to the longest job. A contradiction is obtained. □

The LPT bound is also tight. Let $m = 3$ and the jobs 5, 5, 4, 4, 3, 3, 3. LPT first makes the loads 5, 5, 4. The next 4 and two 3s bring the loads to 8, 8, 8; the last 3 raises one machine to 11. The best schedule, $\{5, 4\}$, $\{5, 4\}$, $\{3, 3, 3\}$, finishes at 9: ratio $11/9 = 4/3 - 1/9$. The same pattern can be built for every m: $2m - 1, 2m - 1, \ldots, m + 1, m + 1$ and three jobs of m. A verification script checked for $m = 2, 3, 4$ that LPT gives $4m - 1$ and the best schedule gives $3m$.

LPT is not online: to sort the jobs it must know all of them in advance. In return its guarantee is much stronger. As m grows, the bound of list scheduling approaches 2 and that of LPT approaches $4/3$; for $m = 2$ the values are 1.5 and $7/6 \approx 1.17$, for $m = 4$ they are 1.75 and 1.25. Machine scheduling also has a PTAS (Hochbaum and Shmoys, 1987): every ratio $1 + \varepsilon$ can be reached in polynomial time, but the time is exponential in $1/\varepsilon$.

6.4.5 LPT code and trace

The Java code in the slides writes LPT in the simplest way:

```
int lpt(Integer[] jobs, int m) {
  Arrays.sort(jobs, Comparator.reverseOrder());
  int[] load = new int[m];
  for (int p : jobs) {
    int k = 0;              // least loaded
    for (int i = 1; i < m; i++)
      if (load[i] < load[k]) k = i;
    load[k] += p;
  }
  return Arrays.stream(load).max().getAsInt();
}
```

The first line sorts the jobs from largest to smallest with `Comparator.reverseOrder()`. Because this comparator works with objects, the array is of type `Integer[]`, not `int[]`. The array `load` holds the current load of each machine and starts at zero. The loop takes each job in turn. The inner loop finds the index of the least loaded machine in `k`. The comparison is strictly less than (`<`); so if loads are equal, the machine with the smaller index is chosen. The job is added to that machine's load. Finally the largest of the loads is returned.

If the first line of the code is deleted, list scheduling remains: jobs go to the least loaded machine in the order given. This ordering is the only difference between the two algorithms. In the code the least loaded machine is found by scanning an array, which takes $O(m)$ per job. The total time is $O(n \log n + nm)$. If the machines are kept in a heap, the time drops to $O(n \log n + n \log m) = O(n \log n)$.

Step	p	`load` (before)	Least loaded `k`	`load` (after)
1	6	[0, 0, 0]	0 (first machine on a tie)	[6, 0, 0]
2	4	[6, 0, 0]	1 (load 0)	[6, 4, 0]
3	3	[6, 4, 0]	2 (load 0)	[6, 4, 3]
4	2	[6, 4, 3]	2 (load 3)	[6, 4, 5]
5	2	[6, 4, 5]	1 (load 4)	[6, 6, 5]
6	2	[6, 6, 5]	2 (load 5)	[6, 6, 7]

Table 6.12: Trace of `lpt({2, 3, 4, 6, 2, 2}, 3)`; the jobs after sorting are 6, 4, 3, 2, 2, 2.



Figure 6.17: The end of the LPT trace. Machine loads 6, 6, and 7; $C_{max} = 7$.

Table 6.12 shows each step of the code. In the first step all loads are zero. The inner loop starts with `k = 0` and does not change it, because no load is less than 0; 6 goes to the first machine. In the second and third steps the empty machines are filled in turn. In the fourth step the least loaded machine is M3 with load 3 (`k = 2`). In the fifth step the loads are 6, 4, 5 and the job of 2 goes to the machine with load 4. In the last step the loads are 6, 6, 5 and the last job is added to the machine with load 5. The makespan is 7. Since the lower bound is $\lceil 19/3 \rceil = 7$, this schedule is optimal.

6.5 Vertex cover

Suppose every link in a campus network is to be monitored. When monitoring software is installed on a router, it sees all the lines connected to that router. On how few routers must the software be installed? This question is the classic vertex cover problem of graph theory.

6.5.1 Problem definition

Definition 6.8

Let $G = (V, E)$ be an undirected graph. A set $C \subseteq V$ that contains at least one endpoint of every edge is called a **vertex cover**. An edge is *covered* if one of its endpoints is in the set. The minimum vertex cover problem asks for the cover with the fewest elements.

A cover can be thought of as a guard arrangement: the guard at each vertex sees the edges touching that vertex, and every edge must be seen by at least one guard. The decision version of the problem, that is, the question “is there a cover of k vertices?”, is NP-complete.

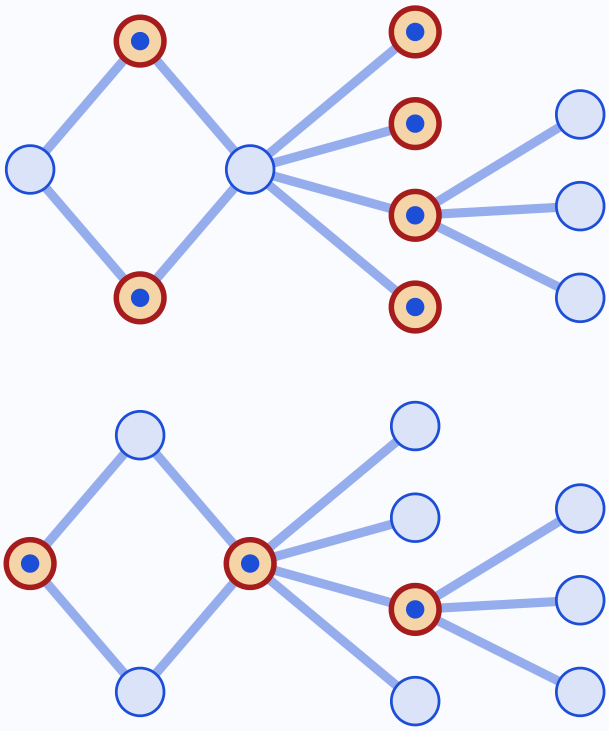


Figure 6.18: Two covers of the same graph. Left, a valid cover of size 6: every edge has a guard at at least one endpoint. Right, the minimum cover, of size 3.

In the graph of Figure 6.18 the six-vertex set on the left is valid, but the three vertices on the right do the same job. The leftmost vertex covers two edges of the quadrilateral, the central hub covers the other two edges of the quadrilateral and the four edges leading to the right, and the inner vertex on the right covers the edges going to the three leaves. It can also be shown that fewer than three do not suffice: one edge of the quadrilateral, one edge going from the hub to the right, and one leaf edge are three edges with no common endpoint. Each needs a separate guard.

The second example in the slides has four vertices. The vertices are A, B, C, D and the edges are *(A, B), (A, C), (B, C), (B, D), (C, D)*. The set *{A, B, C}* is a valid cover: every edge has at least one endpoint in the set. But it is not minimum, because *{B, C}* also covers all five edges. A single vertex does not suffice; for example, B does not cover the edge *(A, C)*. The best value is 2.

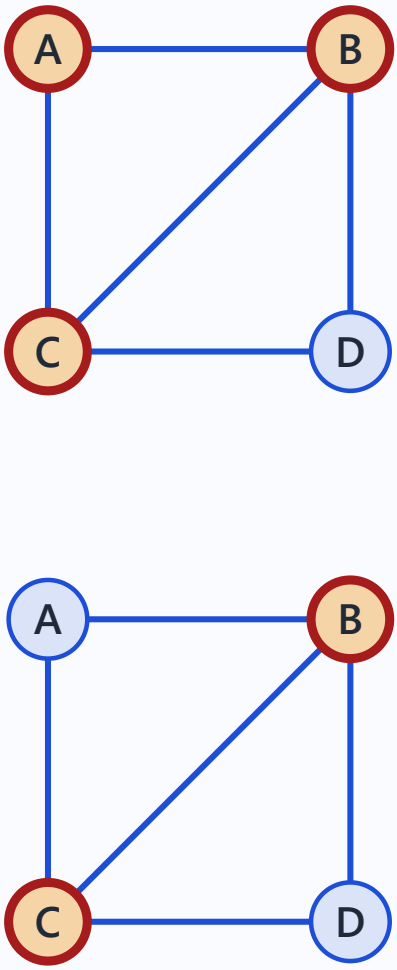


Figure 6.19: A graph with four vertices. Left, *{A, B, C}*, valid, of size 3. Right, *{B, C}*, minimum, of size 2.

6.5.2 Solution approaches

The exact solution checks each of the 2^n subsets in $O(m)$ time: $O(2^n \cdot m)$ in total. Branch and bound and linear programming speed up this search but do not remove the exponential time.

For approximate solutions two greedy ideas stand out: choosing the vertex of highest degree, and taking both endpoints of an uncovered edge at once. The first looks natural but has a weak guarantee; the second looks wasteful but guarantees the ratio 2.

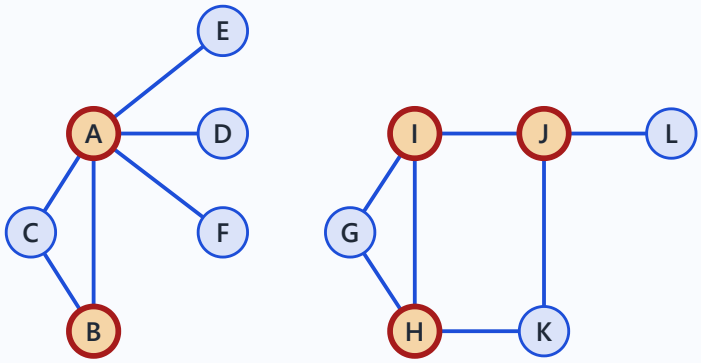


Figure 6.20: A graph with 12 vertices, A to L. One of the minimum covers is *{A, B, H, I, J}*, 5 vertices.

The graph in Figure 6.20 is used in the trace. Its edges are *(A, B), (A, C), (A, D), (A, E), (A, F), (B, C), (G, H), (G, I), (H, I), (H, K), (I, J), (J, K), (J, L)*. A verification script tried all subsets. The minimum cover has 5 vertices, and there are four different optimal solutions: *{A, B, H, I, J}*, *{A, B, G, H, J}*, *{A, C, H, I, J}*, and *{A, C, G, H, J}*. That fewer than five do not suffice can be seen as follows: the triangle A, B, C needs at least two vertices; so does the triangle G, H, I. The edge *(J, L)* also needs a vertex separate from these two triangles.

6.5.3 Why is adding one endpoint not enough?

The first algorithm that comes to mind is this. Initially the cover is empty. An uncovered edge is chosen and one of its endpoints is added to the set. This step is repeated until all edges are covered. The algorithm always produces a valid cover, but nothing can be said about its quality unless a rule is given for which endpoint to add.

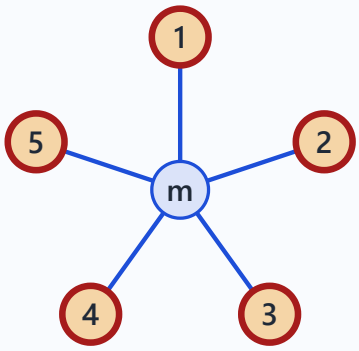


Figure 6.21: A star graph. If the leaf endpoint is chosen at every edge, 5 vertices are chosen; the best solution is just the center, *{m}*.

Figure 6.21 shows the bad case. The center *m* is connected to five leaves. If the algorithm chooses the leaf endpoint at every edge, the edges are covered one by one, and in the end all five leaves are chosen. Yet choosing only the center covers all the edges. On a star with *n* leaves the ratio of this algorithm is *n*, which grows without bound with the input.

The correct 2-approximation makes a small but decisive change. An uncovered edge *(u, v)* is chosen and *both* *u* and *v* are added to the set. All edges touching *u* or *v* count as covered. These steps are repeated until no edges remain. On the star this algorithm picks both endpoints of the first edge, that is, the center and one leaf. All edges are covered; the result is 2 vertices, twice the best.

6.5.4 The edge-picking 2-approximation: code and trace

The Java code in the slides takes the edges in an array and processes them in order:

```
Set<Character> vertexCover(char[][] edges) {
  Set<Character> cover = new HashSet<>();
  for (char[] e : edges) {
    char u = e[0], v = e[1];
    if (!cover.contains(u) && !cover.contains(v)) {
      cover.add(u); cover.add(v); }  // both endpoints
  }
  return cover;
}
```

Each edge is a two-character array: `e[0]` and `e[1]` are the endpoints. The cover is kept in a `HashSet`; the operations `contains` and `add` take *O(1)* on average. For each edge the loop checks that neither endpoint is in the cover. If so, the edge is not yet covered and both endpoints are added. If one of the endpoints is already in the cover, the edge is covered and is skipped. The code does not write a separate "mark as covered" operation; an edge becomes covered automatically the moment one of its endpoints enters the set, and it is skipped when its turn comes.

The cost is clear. Each edge is examined once, and each examination takes constant time on average. The total time is *O(V + E)*. The memory is *O(V)* for the cover.

Edge	Case	Cover (after)
(A, B)	both endpoints outside → A and B added	{A, B}
(A, C)	A in cover → skipped	{A, B}
(A, D), (A, E), (A, F)	A in cover → skipped	{A, B}
(B, C)	B in cover → skipped	{A, B}
(G, H)	both endpoints outside → G and H added	{A, B, G, H}
(G, I)	G in cover → skipped	{A, B, G, H}
(H, I), (H, K)	H in cover → skipped	{A, B, G, H}
(I, J)	both endpoints outside → I and J added	{A, B, G, H, I, J}

Edge	Case	Cover (after)
(J, K), (J, L)	J in cover → skipped	{A, B, G, H, I, J}

Table 6.13: Trace of `vertexCover` on the graph with 12 vertices.

Table 6.13 runs the code on the graph of Figure 6.20. The edges are given in the order above. Both endpoints of the first edge *(A, B)* are outside; A and B are added. The next five edges touching A or B are skipped. *(G, H)* is the second independent edge; G and H are added. *(G, I)*, *(H, I)*, and *(H, K)* are skipped. Both endpoints of *(I, J)* are outside; I and J are added. J covers the remaining two edges.

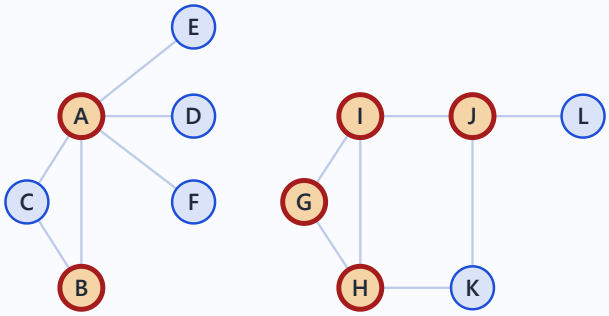


Figure 6.22: The end of the trace. The cover *{A, B, G, H, I, J}*, 6 vertices. The selected edges *(A, B)*, *(G, H)*, *(I, J)* form a matching, that is, they share no endpoints.

The result is the set *{A, B, G, H, I, J}* with 6 vertices. Since the best value is 5, the ratio is *6/5 = 1.2*, far below the guaranteed 2. The three edges the algorithm picked, *(A, B)*, *(G, H)*, and *(I, J)*, have no common endpoints. A set of edges with no common endpoints is called a **matching**. The key to the ratio proof is this matching.

6.5.5 Why is the ratio at most 2?

Theorem 6.5 (The edge-picking 2-approximation)

The set C found by the edge-picking algorithm is a valid vertex cover and $|C| \le 2 \cdot OPT$.

Proof. Validity is easy: when an edge is processed, either one of its endpoints is already in the set or both endpoints are added. In either case the edge is covered.

For the ratio, let M be the set of edges for which the algorithm adds both endpoints. An edge enters M only when neither endpoint is yet in the set. So two edges of M cannot share an endpoint; M is a matching. Now consider any cover, for example the best cover C^*. C^* must cover every edge of M. Because the edges of M share no endpoints, a vertex can cover at most one of them. Hence C^* contains a separate vertex for each edge of M: $OPT = |C^*| \ge |M|$. The algorithm picks exactly $|C| = 2\,|M|$ vertices. Combining, $|C| = 2\,|M| \le 2 \cdot OPT$. □

There are other approaches too. The **greedy algorithm that picks the vertex of highest degree** looks natural. At each step it takes the vertex touching the most uncovered edges. On the graph of Figure 6.20, if ties in degree are broken alphabetically, this algorithm picks A, H, J, B, G in turn and finds the optimum with 5 vertices. But the ratio of this algorithm can only be bounded by $O(\log n)$, and graphs can be constructed on which the ratio really grows logarithmically. It is a good heuristic, but not a constant-ratio approximation algorithm.

Rounding a linear program gives a second 2-approximation. Each vertex gets a variable $x_v \in \{0, 1\}$, and for each edge the constraint $x_u + x_v \geq 1$ is written; $\Sigma\, x_v$ is minimized. If the integrality condition is dropped and $0 \leq x_v \leq 1$ is written, the problem becomes a linear program, the **LP relaxation**, which is solved in polynomial time. The value of the relaxation is less than or equal to OPT. Then the vertices with $x_v \geq 1/2$ are chosen. Since in every edge constraint at least one of the two variables reaches $1/2$, the result is a valid cover. Since every chosen vertex has value at least $1/2$, the number of elements is at most twice the value of the relaxation.

No constant ratio noticeably better than 2 is known. If the Unique Games Conjecture is true, it has been shown that for no $\varepsilon > 0$ is there a polynomial-time algorithm with ratio $2 - \varepsilon$ (Khot and Regev, 2008). The seemingly simple edge-picking algorithm reaches the best possible constant ratio under this conjecture.

> **Common mistake**
>
> Adding only one endpoint of the edge. This makes the algorithm look less wasteful, but on the star graph the ratio grows to n. Adding both endpoints is the condition for being able to compare with the best solution: the matching proof works only when the selected edges share no endpoints. Conversely, if the edges whose two endpoints are added did not form a matching, no lower bound could be built.

6.5.6 NP-completeness: the reduction from 3-SAT

That the decision version of vertex cover is NP-complete is shown by a reduction. A reduction transforms every instance of a known NP-complete problem into an instance of vertex cover in polynomial time, so that the two instances have the same answer. 3-SAT is used as the source problem. In 3-SAT a logic formula is a conjunction ("and") of clauses, each consisting of three literals; each clause is a disjunction ("or") of literals. A literal is a variable or its negation. The question is whether there is an assignment of values that makes the formula true.

From a formula with l variables and m clauses, the reduction builds the following graph:

- For each variable x_i, two vertices, x_i and $\neg x_i$, with an edge between them. To cover this edge, one of the two vertices must be chosen; the choice determines the value of the variable.
- For each clause, a triangle with three vertices; the vertices are labeled with the literals of the clause. To cover a triangle, at least two of its three vertices must be chosen.

- Each triangle vertex is joined by an edge to the variable vertex with the same label.

The cover size sought is $k = l + 2m$: one vertex from each variable edge and two from each triangle. The graph has $2l + 3m$ vertices and $l + 6m$ edges; the construction takes polynomial time.

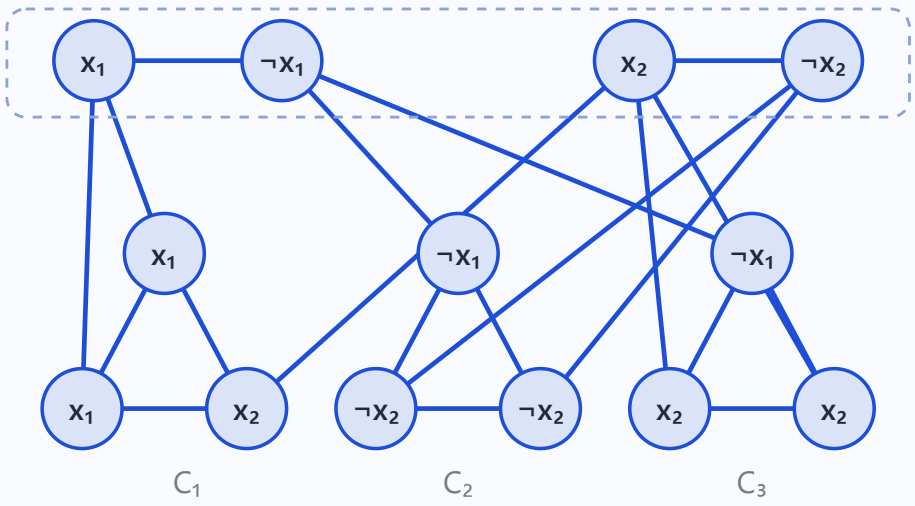


Figure 6.23: The graph built for $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$. In the upper box two variable edges, below three clause triangles; $l = 2$, $m = 3$, $k = 8$.

The example formula in the slides is $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$. It has two variables and three clauses; the graph has $4 + 9 = 13$ vertices and $2 + 9 + 9 = 20$ edges. The size sought is $k = 2 + 2 \cdot 3 = 8$.

The assignment that satisfies the formula is $x_1 = \mathit{false}$, $x_2 = \mathit{true}$. The cover is built from this assignment as follows. Because x_1 is false, $\neg x_1$ is chosen from its variable edge; because x_2 is true, x_2 is chosen. Every clause has a true literal. The triangle vertex of that literal is left out and the other two vertices are chosen. The connecting edge of the vertex left out is covered by the vertex chosen on the variable side. In C_1 the true literal is x_2; the two x_1 vertices are chosen. In C_2 the true literal is $\neg x_1$; the two $\neg x_2$ vertices are chosen. In C_3 the true literal is $\neg x_1$; the two x_2 vertices are chosen.

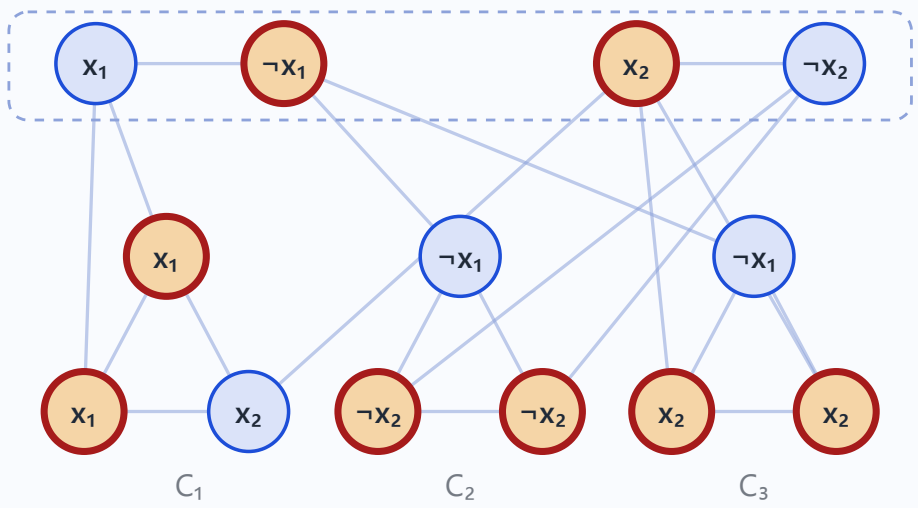


Figure 6.24: The cover built from the assignment: $\neg x_1$ and x_2 from the variable edges, and from each triangle the two vertices other than the true literal. The 8 vertices cover all 20 edges.

In the end 8 vertices are chosen and all 20 edges are covered (Figure 6.24). A verification script confirmed, by trying all possibilities, that the minimum cover of this graph really is 8 and that the only assignment satisfying the formula is $x_1 = false, x_2 = true$.

The reverse direction also holds. A cover of size $k = l + 2m$ contains exactly one vertex from each variable edge and exactly two from each triangle. These structures need at least one and at least two vertices, and the total is exactly k. The vertices chosen on the variable side define an assignment. In each triangle, the connecting edge of the vertex left out must be covered; this is possible only if its partner on the variable side has been chosen. So that literal is true under the assignment and the clause is satisfied. Conclusion: φ is satisfiable if and only if the graph has a vertex cover of size $l + 2m$.

6.6 Knapsack

In the 0/1 knapsack problem there are n items. Item i has value v_i and weight w_i. The capacity of the knapsack is W. We seek the set of items whose total weight does not exceed W and whose total value is largest. Each item is either taken whole or not at all. The problem is NP-hard. It is solved by dynamic programming in $O(nW)$ time, but this time depends on the numerical value of W. Since W is written in the input with $\log W$ bits, the time is exponential in the size of the input.

6.6.1 The ratio-ordered greedy rule

The most natural greedy rule is to sort the items by value per unit of weight. The items are put in decreasing order of the ratio v_i / w_i and examined in turn. An item that fits in the knapsack is taken, and one that does not fit is skipped. Sorting is the dominant cost; the time is $O(n \log n)$.

Item	Value	Weight	Ratio	Decision
6	20	7	2.86	taken (weight 7)
3	8	4	2.00	does not fit (7 + 4 = 11)
7	15	8	1.88	does not fit
2	5	3	1.67	taken (weight 10)
5	10	6	1.67	does not fit
1	3	2	1.50	does not fit
4	6	5	1.20	does not fit

Table 6.14: The ratio-ordered greedy rule for capacity $W = 10$.

Table 6.14 gives the seven-item example from the slides. The item with the highest ratio is 6 (20/7 ≈ 2.86) and it is taken; the knapsack now holds 7 kilograms. The next item, 3, weighs 4 kilograms and does not fit. Item 7 does not fit either. Item 2 weighs 3 kilograms and fills the knapsack to exactly 10 kilograms. After this no item fits. The greedy total value is $20 + 5 = 25$. A verification script tried all $2^7 = 128$ subsets; the best value is also 25, again consisting of items 2 and 6. In this example the greedy rule finds the optimum. Items 2 and 5 have equal ratios (5/3 = 10/6); in the table item 2 comes first.

If items could be divided, that is, if the problem were the fractional knapsack, this rule would always give the best result: a part of the first item that does not fit is taken, and the knapsack is exactly full. For the 0/1 version, however, the rule has no guarantee at all.

Example 6.4

Let the capacity be $W = 10$ and let there be two items: (value 2, weight 1) and (value 10, weight 10). The ratios are 2 and 1. The greedy rule takes the first item first. Room for 9 kilograms remains in the knapsack, and the second item does not fit. The greedy value is 2. The best solution takes only the second item and reaches value 10. The ratio is $10 / 2 = 5$. If the value and weight of the second item are made W, the ratio becomes $W / 2$; it gets arbitrarily bad as W grows.

6.6.2 The improved rule: a 2-approximation

The source of the bad example is a single valuable item. The ratio ordering leaves it behind, and once small items take up room in the knapsack that item no longer fits. The fix is simple. The greedy value is compared with the value of the most valuable single item that fits in the knapsack by itself, and the larger is taken:

```
result = max( greedy value, most valuable single item that fits )
```

On the bad example this rule finds $max(2, 10) = 10$ and reaches the optimum. The general guarantee is 2.

Theorem 6.6 (The improved greedy rule)

Suppose the items whose weight exceeds W have been discarded at the start. If the greedy value is G and the value of the most valuable single item that fits is v_{max}, then $max(G, v_{max}) \geq OPT / 2$.

Proof. Number the items $1, 2, \ldots, n$ in ratio order. Let k be the first item that does not fit when the items are taken in order. The optimal value of the fractional knapsack is obtained as $v_1 + \ldots + v_{k-1}$ plus a part of item k. A 0/1 solution is also a fractional solution; therefore

$$\mathrm{OPT} \leq v_1 + \ldots + v_{k-1} + v_k$$

can be written. The greedy rule has taken all of the first $k - 1$ items and may then take other items that fit: $G \geq v_1 + \ldots + v_{k-1}$. Since item k fits by itself, $v_k \leq v_{max}$. Combining, $OPT \leq G + v_{max} \leq 2 \cdot max(G, v_{max})$. If there is no item that does not fit, that is, if k does not exist, the greedy rule takes all the items and $G = OPT$. □

Example 6.5

Let $W = 15$ and let the items, as (value, weight), be E1 (30, 10), E2 (20, 8), E3 (18, 7), E4 (6, 4). The ratios are 3.00, 2.50, 2.57, and 1.50. The ratio order is E1, E3, E2, E4. E1 is taken (10 kg). E3 does not fit (17 kg), E2 does not fit (18 kg), E4 fits (14 kg). The greedy value is 36. The most valuable single item that fits is E1 (30); the improved rule finds 36. The best solution takes E2 and E3: exactly 15 kg and value 38. The ratio is $38/36 \approx 1.06$. The upper bound in the proof can also be checked: here k is E3 and $OPT = 38 \leq 30 + 18 = 48$. The fractional value is $30 + (5/7) \cdot 18 \approx 42.86$, which is also greater than 38.

6.6.3 A fully polynomial-time approximation scheme

The knapsack problem is among the problems that admit a ratio much better than 2. For every $\varepsilon > 0$ there is an algorithm that finds at least a $(1 - \varepsilon)$ fraction of the best value and runs in $O(n^3/\varepsilon)$ time. This is a fully polynomial-time approximation scheme (FPTAS).

The idea is to scale the values. The knapsack problem has a dynamic program built over values: for each value x, the table holds "the weight of the lightest set of items whose total value is exactly x". The table has as many entries as the sum of the values, and the time is $O(n \cdot \Sigma\, v_i)$. When the values are large, this time is large. The FPTAS divides the values by a number $K = \varepsilon \cdot v_{max} / n$ and rounds down: v'_i

$= \lfloor v_i / K \rfloor$. Each scaled value is at most n/ε and their sum is at most n^2/ε. With these small values the dynamic program runs in $O(n \cdot n^2/\varepsilon) = O(n^3/\varepsilon)$ time and finds the optimal solution of the scaled problem.

The loss caused by rounding is bounded. Each item loses at most K of its value; since the best set contains at most n items, the total loss is at most $nK = \varepsilon \cdot v_{max} \le \varepsilon \cdot OPT$. Therefore the true value of the set found is at least $(1 - \varepsilon) \cdot OPT$. The summary table in the slides writes this ratio as $1 + \varepsilon$, by analogy with the minimization form; for small ε the two ways of writing express the same guarantee.

6.7 Power iteration

The algorithms so far produced approximate solutions to combinatorial problems. In numerical computation, "approximation" takes another form: instead of finding the exact value in a finite number of steps, one produces a sequence that gets a little closer to it at each step. This behavior is called **convergence**. A typical example of convergence is power iteration, which finds the largest eigenvalue of a matrix.

6.7.1 Dominant eigenvalue and eigenvector

For a square matrix A, a nonzero vector v satisfying $A \cdot v = \lambda \cdot v$ is called an **eigenvector**, and the number λ its **eigenvalue**. An eigenvector is a vector that, when multiplied by the matrix, does not change direction but only stretches or shrinks by the factor λ. An $n \times n$ matrix has at most n eigenvalues. The one with the largest absolute value is called the **dominant eigenvalue** and is denoted λ_1; its eigenvector is v_1.

> **Definition 6.9**
>
> **Power iteration** is an iterative method that repeatedly multiplies a nonzero vector x_0 by the matrix A, normalizing at each step, and converges to the dominant eigenvector v_1 and the dominant eigenvalue λ_1.

Methods that compute all the eigenvalues exactly take $O(n^3)$ time. This is acceptable if the matrix has thousands of rows; it is not if it has billions. Power iteration gives only the dominant eigenvalue and eigenvector, and only approximately, but each step is a single matrix–vector product. This product takes $O(n^2)$ on a dense matrix and as many operations as there are nonzero entries on a sparse one. For huge sparse matrices it is often the only method that can be used.

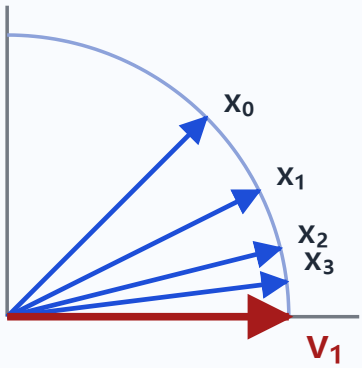


Figure 6.25: For $A = [[2, 0], [0, 1]]$ and $x_0 = (1, 1)$, the unit vectors x_0, x_1, x_2, x_3 turn step by step toward the direction $v_1 = (1, 0)$.

The simplest example is a diagonal matrix. The matrix $A = [[2, 0], [0, 1]]$ has eigenvalues 2 and 1 and eigenvectors $(1, 0)$ and $(0, 1)$. Take $x_0 = (1, 1)$. With each multiplication the first component doubles and the second stays the same: $A^k x_0 = (2^k, 1)$. The direction of the normalized vector gradually turns toward $(1, 0)$ (Figure 6.25). The ratio of the second component to the first is 2^{-k}. The error halves at each step. For this ratio to fall below 10^{-3} we need $2^k > 1000$; since $2^9 = 512$ and $2^{10} = 1024$, $k = 10$ steps suffice.

6.7.2 The steps and why it converges

The steps of the algorithm are as follows:

1. Fix a matrix A and choose a nonzero starting vector x_0. Choosing it at random is common.
2. Multiply the vector by the matrix: $y = A \cdot x_k$.
3. Normalize the new vector, that is, bring its norm (length) to 1: $x_{k+1} = y / \|y\|$.
4. Repeat the second and third steps for the desired number of steps, or until the change in the vector becomes small enough.

In the end x_k converges to v_1. There are two ways to estimate the eigenvalue. The length before normalization, $\|A \cdot x_k\|$, converges to $|\lambda_1|$. For symmetric matrices a faster-converging estimate is the **Rayleigh quotient**: when $\|x_k\| = 1$, $\lambda \approx x_k^T A x_k$.

Why the method works can be seen by writing the starting vector in terms of eigenvectors. Let the eigenvectors of A form a basis and let $x_0 = c_1 v_1 + c_2 v_2 + \ldots + c_n v_n$. Each multiplication by A multiplies each component by its own eigenvalue:

$$A^k x_0 = \lambda_1^k \left[c_1 v_1 + c_2 (\lambda_2/\lambda_1)^k v_2 + \ldots + c_n (\lambda_n/\lambda_1)^k v_n \right]$$

If $|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$, then all the factors $(\lambda_i/\lambda_1)^k$ inside the square brackets decay geometrically to zero. What remains is $c_1 v_1$. Normalization removes the leading factor λ_1^k, and the vector approaches the unit vector in the direction of v_1. The component that dies out most slowly is that of v_2. So the

error shrinks by a factor of about $|\lambda_2/\lambda_1|$ at each step. If the ratio is 0.1, one correct digit is gained per step; if the ratio is 0.99, about 230 steps are needed for one digit.

The formula shows both conditions clearly. First, $c_1 \neq 0$ must hold; if the starting vector is chosen orthogonal to v_1, the v_1 component is absent and in theory never appears. In practice rounding errors create a small v_1 component, but convergence is delayed. The probability that a randomly chosen vector is exactly orthogonal is zero. Second, $|\lambda_1| > |\lambda_2|$ must hold. If there are two eigenvalues equal in absolute value, for example $\lambda_1 = 3$ and $\lambda_2 = -3$, the second component does not die out and the vector may oscillate between two directions.

6.7.3 Convergence step by step

The example in the slides is a symmetric matrix: $A = [[2, 1], [1, 3]]$. The characteristic equation is $(2 - \lambda)(3 - \lambda) - 1 = \lambda^2 - 5\lambda + 5 = 0$, and its roots are $\lambda_1 = (5 + \sqrt{5})/2 \approx 3.618$ and $\lambda_2 = (5 - \sqrt{5})/2 \approx 1.382$. The dominant eigenvector, normalized to length 1, is $v_1 \approx (0.5257, 0.8507)$. The ratio is $\lambda_2/\lambda_1 \approx 0.382$. The starting vector is $x_0 = (1, 0)$ and its error is $\|x_0 - v_1\| \approx 0.974$.

k	$A \cdot x_{k-1}$	x_k	λ estimate	Error
0	—	(1, 0)	—	0.974
1	(2.000, 1.000)	(0.894, 0.447)	3.000	0.5465
2	(2.236, 2.236)	(0.707, 0.707)	3.500	0.2313
3	(2.121, 2.828)	(0.600, 0.800)	3.600	0.0899
4	(2.000, 3.000)	(0.5547, 0.8321)	3.61538	$3.4 \cdot 10^{-2}$
5	(1.942, 3.051)	(0.5369, 0.8437)	3.61765	$1.3 \cdot 10^{-2}$
6	(1.917, 3.068)	(0.5300, 0.8480)	3.61798	$5.0 \cdot 10^{-3}$
7	(1.908, 3.074)	(0.5274, 0.8496)	3.61803	$1.9 \cdot 10^{-3}$
8	(1.904, 3.076)	(0.5264, 0.8503)	3.61803	$7.3 \cdot 10^{-4}$
9	(1.903, 3.077)	(0.5260, 0.8505)	3.618034	$2.8 \cdot 10^{-4}$

Table 6.15: Power iteration with $A = [[2, 1], [1, 3]]$ and $x_0 = (1, 0)$. λ is the Rayleigh quotient; error $\|x_k - v_1\|$.

Table 6.15 gives the first nine steps. In the first step $A \cdot (1, 0) = (2, 1)$. Its length is $\sqrt{5} \approx 2.236$, and dividing gives $x_1 = (0.894, 0.447)$. The Rayleigh quotient is $x_1^T A\, x_1 = 3.0$. In the second step $A \cdot x_1 = (2.236, 2.236)$; the vector turns exactly to the diagonal direction and $\lambda \approx 3.5$. In the third step $x_3 = (0.6, 0.8)$ and $\lambda \approx 3.6$ are found. In the following steps the estimate of λ quickly settles at 3.618.

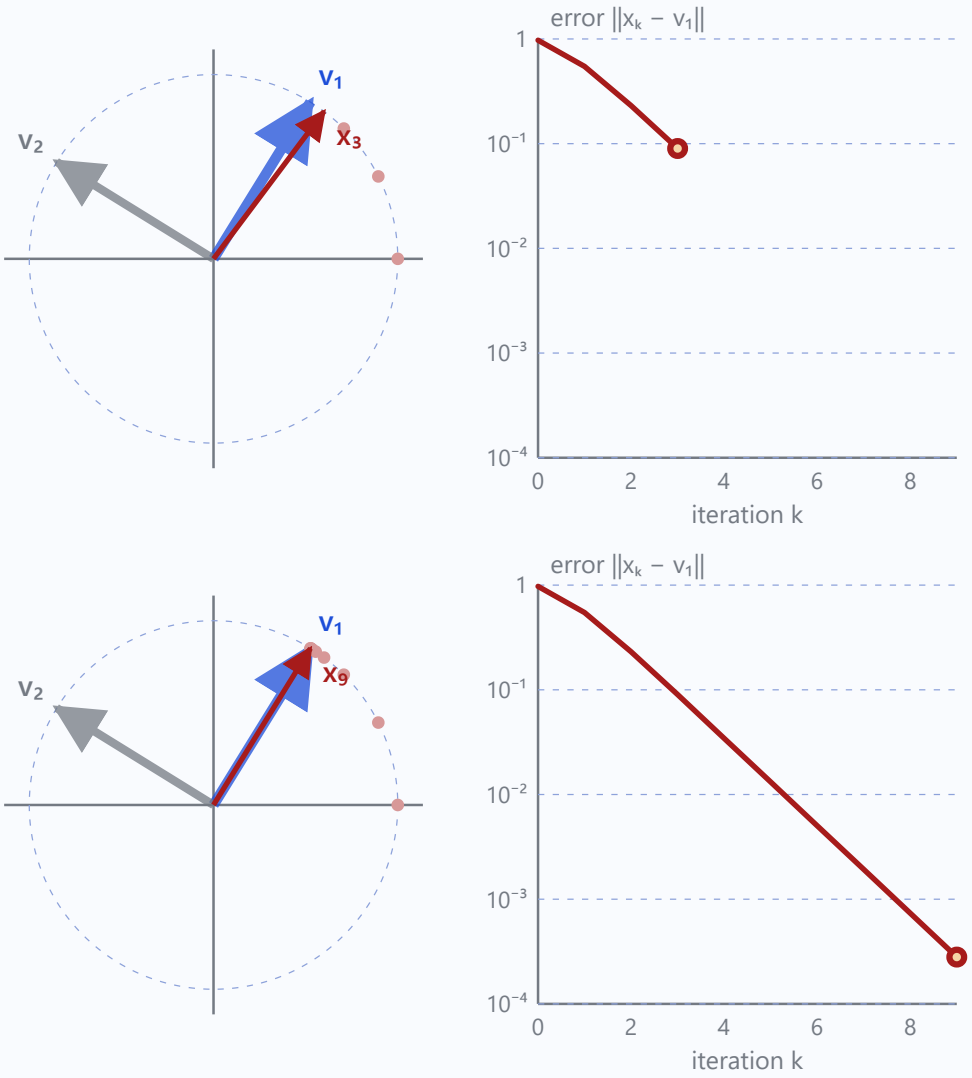


Figure 6.26: Top, the third iteration; bottom, the ninth. On the left the estimate vector x_k approaches v_1; on the right a log-scale error plot. The error plot settles on a straight line: convergence is at a linear (geometric) rate.

The error column shows the rate of convergence clearly. The ratio of two successive errors is 0.42 and 0.39 in the first steps; after the fourth step it settles at $0.382 \approx \lambda_2/\lambda_1$. The error shrinks by a factor of about 0.38 at each step. On a log-scale plot this behavior is a straight line with slope $\log_{10} 0.382 \approx -0.42$ (Figure 6.26). In numerical analysis such convergence is called linear or geometric convergence. Each step gains about 0.42 decimal digits; two and a half steps are needed for one digit. The eigenvalue estimate converges faster than the eigenvector: for a symmetric matrix the error of the Rayleigh quotient is proportional to the square of the vector error. At the ninth step, when the vector error is $2.8 \cdot 10^{-4}$, λ is correct to six digits.

6.7.4 Java code and a 3 × 3 example

The Java code in the slides runs the algorithm for a fixed number of steps:

```
double[] powerIteration(double[][] a, int steps) {
  int n = a.length;
  double[] x = new double[n];
  Arrays.fill(x, 1.0); // x₀ ≠ 0, not orthogonal to v₁
  for (int k = 0; k < steps; k++) {
    double[] y = new double[n];
    for (int i = 0; i < n; i++)  // y = A·x
      for (int j = 0; j < n; j++)
        y[i] += a[i][j] * x[j];
    double norm = 0;
    for (double yi : y) norm += yi * yi;
    norm = Math.sqrt(norm);
    if (norm == 0) break; // A·x = 0
    for (int i = 0; i < n; i++)  // ‖x‖ = 1
      x[i] = y[i] / norm;
  }
  return x;
}
```

The method takes the matrix `a` and the number of steps `steps`. The starting vector is the vector with all components equal to 1. This choice is simple and satisfies the condition $c_1 \neq 0$ for most matrices. In each round of the outer loop, the product $y = A \cdot x$ is first computed with two nested loops; `y[i]` is the dot product of row `i` of A with `x`. Then the Euclidean norm of `y` is found as the square root of the sum of the squares of its components. If the norm is zero ($A \cdot x = 0$), the division is undefined; `break` stops the loop at that point. Finally `x` is set to `y / norm`. The method returns the final `x`; the value `norm` is not returned, but it can be used as an estimate of λ_1.

In each step the product performs n^2 multiplications and additions; normalization takes $O(n)$. The total time for `steps` steps is $O(steps \cdot n^2)$. The memory is $O(n)$ for `x` and `y`; the matrix itself is not modified. The code uses no stopping criterion. In practical applications the loop is stopped when the difference between two successive `x` falls below a threshold.

The example matrix in the slides is the following:

> A = [[0.511, 0.32, 0.12], [0.211, 0.62, 0.12], [0.311, 0.12, 0.16]], x_0 = (1, 1, 1)

The eigenvalues of this matrix are about 0.9008, 0.3, and 0.0902. The ratio is $\lambda_2/\lambda_1 = 0.3 / 0.9008 \approx 0.33$; convergence is fast. One property of the matrix also stands out. The sums of the first two entries of the first and second rows are the same ($0.511 + 0.32 = 0.211 + 0.62 = 0.831$) and their third entries are equal. So if the first two components of x are equal, those of $A \cdot x$ stay equal. Since $x_0 = (1, 1, 1)$ has this property, $x_1 = x_2$ holds at every step of the iteration.

`k`	`y = A · x`	`norm` = ‖y‖	New `x`
0	(0.9510, 0.9510, 0.5910)	1.4690	(0.6474, 0.6474, 0.4023)
1	(0.5862, 0.5862, 0.3434)	0.8974	(0.6533, 0.6533, 0.3827)

`k`	`y = A · x`	`norm` = $\|y\|$	**New** `x`
2	(0.5888, 0.5888, 0.3428)	0.9005	(0.6539, 0.6539, 0.3807)
3	(0.5890, 0.5890, 0.3427)	0.9008	(0.6539, 0.6539, 0.3805)
4–7	(0.5891, 0.5891, 0.3427)	0.9008	(0.6539, 0.6539, 0.3805)

Table 6.16: Trace of `powerIteration(A, steps)`, 3 × 3 example. In the code the loop variable `k` starts from 0.

In the first step `y[0]` is the product of the first row with `x`: $0.511 \cdot 1 + 0.32 \cdot 1 + 0.12 \cdot 1 = 0.951$. `y[1]` is 0.951 for the same reason, and `y[2]` is $0.311 + 0.12 + 0.16 = 0.591$. The norm is $\sqrt{(0.951^2 + 0.951^2 + 0.591^2)} \approx 1.4690$. Dividing gives $x = (0.6474, 0.6474, 0.4023)$. This vector has length 1; the numbers now neither grow nor shrink.

In the second step $y = (0.5862, 0.5862, 0.3434)$. The ratio of the components is very close to that of the first step; the direction has changed little. The norm is 0.8974. In the third step the norm rises to 0.9005 and in the fourth to 0.9008, where it stays. We see that the norm converges to $\lambda_1 = 0.9008$. After the fourth step the vector no longer changes to four digits: $v_1 \propto (0.6539, 0.6539, 0.3805)$.

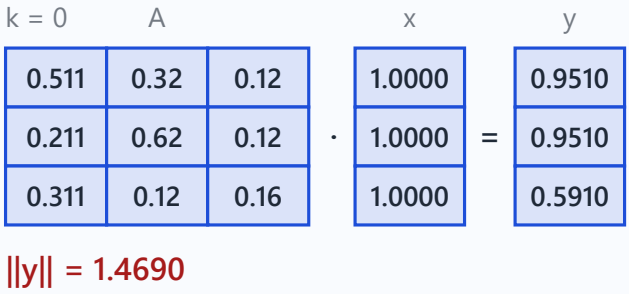


Figure 6.27: The product $y = A \cdot x$ and the norm in the first iteration: $y = (0.9510, 0.9510, 0.5910)$, $\|y\| = 1.4690$.

The trace in the slides stops at `steps = 3` and returns $x \approx (0.6539, 0.6539, 0.3807)$; the third component settles at 0.3805 one step later. The number of steps must be chosen according to the desired accuracy: for the error to fall by a factor of 10^{-d}, about $d \,/\, \log_{10}(\lambda_1/|\lambda_2|)$ steps are needed. Here the ratio is about 3, and each step gains slightly less than half a digit ($\log_{10} 3 \approx 0.48$).

Common mistake

Skipping normalization. If `x` is not normalized at each step, $A^k x_0$ is computed directly. If $|\lambda_1| > 1$, the components grow at rate λ_1^k and exceed the range of the `double` type within a few hundred steps. If $|\lambda_1| < 1$, as in the 3 × 3 example, the components shrink to zero and the significant digits are lost. Normalization corrects only the scale; it does not change the direction, that is, the information being sought.

6.7.5 Limitations and applications

The strengths of power iteration come from its simplicity. It uses only matrix–vector products. It does not modify the matrix itself, and as extra memory it needs only a few vectors, that is, $O(n)$ space. It can therefore be applied to sparse matrices with billions of rows.

Its limitations are also clear. It finds only the dominant eigenvalue. If $|\lambda_2|$ is close to $|\lambda_1|$ it converges very slowly; if they are equal it may not converge. If the starting vector is chosen orthogonal to v_1 ($c_1 = 0$), it fails in theory.

The best-known application of the method is Google's **PageRank** computation. The importance of a page is determined by the importance of the pages that link to it; this definition is an eigenvector equation. The importance values of the pages are the dominant eigenvector of the web link matrix. The matrix has billions of rows but is very sparse, and power iteration reaches sufficient accuracy in a few dozen steps. Its derivatives are also widespread: inverse iteration finds the smallest eigenvalue, shifted iteration the eigenvalue closest to a given value; the Lanczos and Arnoldi methods compute several eigenvalues together. In principal component analysis (PCA) the first principal component can also be found this way.

6.8 Comparison and common mistakes

Table 6.17 puts the algorithms studied in these notes side by side by problem, ratio, and running time.

Problem	Algorithm	Ratio	Time
TSP (general)	Nearest neighbor (heuristic)	no guarantee	$O(n^2)$
Metric TSP	MST + preorder + shortcuts / Christofides	2 / 1.5	$O(n^2)$ / $O(n^3)$
TSP	2-opt local search	no guarantee	$O(n^2)$ per pass
Machine scheduling	List scheduling / LPT	$2 - 1/m$ / $4/3 - 1/(3m)$	$O(n \log m)$ / $O(n \log n)$
Vertex cover	Pick an edge, add both endpoints	2	$O(V + E)$
0/1 knapsack	max(ratio-ordered greedy, best single item) / FPTAS	2 / $1 + \varepsilon$	$O(n \log n)$ / $O(n^3/\varepsilon)$
Dominant eigenvalue	Power iteration	error $\propto \lvert\lambda_2/\lambda_1\rvert^k$	$O(n^2)$ per step

Table 6.17: Comparison of the algorithms studied.

The common mistakes treated in various places in these notes are the following:

- **Mistaking a heuristic for an approximation algorithm.** Nearest neighbor and 2-opt give good results, but they have no ratio guarantee. An approximation algorithm requires a proven ρ.
- **Writing the ratio upside down.** For minimization $C \le \rho \cdot C^*$, for maximization $C \ge C^* / \rho$. ρ is always at least 1.

- **Adding one endpoint in a cover.** Adding only one endpoint of an edge can give a result n times worse on a star graph; the 2-approximation adds both endpoints.
- **Forgetting the triangle inequality.** The shortcut step of the MST-based 2-approximation does not lengthen the route only for metric TSP.
- **Assuming a metaheuristic is exact.** Simulated annealing and genetic algorithms do not guarantee reaching the optimum; they are run several times with different seeds and parameters and the best result is kept.
- **Skipping normalization.** In power iteration, if `x` is not normalized at each step, the numbers overflow ($|\lambda_1| > 1$) or shrink to zero.

6.9 Summary

- **Approximation algorithm:** for an NP-hard optimization problem it runs in polynomial time and, on every input, satisfies $\max(C/C^*, C^*/C) \le \rho$; for minimization $C \le \rho \cdot C^*$, for maximization $C \ge C^*/\rho$.
- **Types of methods:** exact algorithm, approximation algorithm, PTAS, FPTAS (time polynomial in $1/\varepsilon$ as well), heuristic and metaheuristic (no guarantee).
- **TSP:** brute force $(n-1)!/2$ tours, Held–Karp $O(n^2\,2^n)$. For general TSP, if P ≠ NP there is no constant-ratio approximation. Nearest neighbor runs in $O(n^2)$ but its ratio can be $\Theta(\log n)$.
- **Metric TSP:** $w(T) \le OPT$ and the tour with shortcuts is $\le 2 \cdot w(T)$; 287.7 / 282.5 on seven cities. Christofides gives the ratio 1.5.
- **2-opt:** gain $\Delta = d(a,c) + d(b,d) - d(a,b) - d(c,d)$; no crossing edges remain at a local optimum; from 1269.8 to 430.8 in 48 moves on 28 cities. Metaheuristics can escape local optima but give no guarantee.
- **Machine scheduling:** list scheduling $2 - 1/m$, LPT $4/3 - 1/(3m)$; lower bounds $\Sigma p/m$ and $\max p$; both bounds are tight.
- **Vertex cover:** both endpoints of an uncovered edge are added; the selected edges form a matching, $|C| = 2|M| \le 2 \cdot OPT$, time $O(V+E)$. In the 3-SAT reduction $k = l + 2m$.
- **Knapsack:** $\max$(ratio-ordered greedy, most valuable single item) is a 2-approximation; value scaling gives an FPTAS.
- **Power iteration:** $x_{k+1} = A\,x_k / \|A\,x_k\| \to v_1$; the error shrinks by a factor of about $|\lambda_2/\lambda_1|$ at each step; the conditions are $c_1 \ne 0$ and $|\lambda_1| > |\lambda_2|$.

6.10 Key terms

approximation algorithm approximation ratio approximation scheme (PTAS / FPTAS) heuristic metaheuristic traveling salesman problem (TSP) Hamiltonian cycle triangle inequality minimum spanning tree (MST) nearest neighbor local search local optimum simulated annealing

list scheduling makespan longest processing time first (LPT) vertex cover matching reduction LP relaxation 0/1 knapsack power iteration convergence

6.11 Test Yourself

1. An approximation algorithm with ratio 2 finds a solution of cost 50 on a minimization problem. In what range does the best cost lie? What would the range be if the same result were found on a maximization problem?

   Answer

   For minimization, since $50 \le 2 \cdot OPT$, $OPT \ge 25$; since the best solution cannot be worse than the one found, $OPT \le 50$. The range is $[25, 50]$. For maximization, since $50 \ge OPT / 2$, $OPT \le 100$, and $OPT \ge 50$. The range is $[50, 100]$.

2. Let the distance matrix of four cities be, row by row, 0: (0 1 15 6), 1: (1 0 7 3), 2: (15 7 0 2), 3: (6 3 2 0). What tour does nearest neighbor find when started from 0? What is the best tour? Is the matrix metric?

   Answer

   The nearest city to 0 is 1 (1). From 1 the unvisited cities are 2 (7) and 3 (3); 3 is chosen. From 3 the only candidate is 2 (2). From 2 we return to 0 (15). The tour is $0 \to 1 \to 3 \to 2 \to 0$, length $1 + 3 + 2 + 15 = 21$. With four cities there are three distinct tours: $0 \to 1 \to 2 \to 3 \to 0$ (16), $0 \to 1 \to 3 \to 2 \to 0$ (21), and $0 \to 2 \to 1 \to 3 \to 0$ (31). The best is 16. The matrix is not metric: $d[0][2] = 15 > d[0][1] + d[1][2] = 8$. The cheap choices left the expensive return edge for the end.

3. Let the points be $a\ (0, 0)$, $b\ (3, 4)$, $c\ (6, 0)$, $d\ (3, -4)$, and let the tour contain the edges (a, b) and (c, d). What is the 2-opt gain; is the move made? Do the edges cross?

   Answer

   $d(a, b) = d(c, d) = 5$, $d(a, c) = 6$, $d(b, d) = 8$. $\Delta = 6 + 8 - 5 - 5 = 4 > 0$; the move would lengthen the tour and is not made. The two edges do not cross: (a, b) lies above the axis and (c, d) below it. For pairs of edges that do not cross, the 2-opt gain can be positive; the geometric result guarantees an improvement only for crossing edges.

4. Jobs with processing times 1, 2, 3, 4, 5, 6 are given in this order to $m = 3$ machines by list

scheduling. What are the machine loads and C_{max}? What does LPT find? What is the best value?

Answer

List: 1, 2, 3 go to the three machines (loads 1, 2, 3). 4 goes to the least loaded, the first machine (5), 5 to the second (7), 6 to the third (9). The loads are 5, 7, 9; $C_{max} = 9$. The LPT order is 6, 5, 4, 3, 2, 1: 6, 5, 4 go to the three machines; 3 goes to the least loaded, the third (7), 2 to the second (7), 1 to the first (7). The loads are 7, 7, 7. Since the total work is 21, the lower bound is 7; LPT finds the optimum. The ratio of list scheduling is $9/7 \approx 1.29$, below the bound $2 - 1/3 \approx 1.67$.

5. The edges of the six-vertex cycle *1–2–3–4–5–6–1*, namely *(1,2), (2,3), (3,4), (4,5), (5,6), (6,1)*, are given in this order to the edge-picking 2-approximation. How many vertices are chosen? What would the result be if the edges were given in the order *(2,3), (5,6), (1,2), (3,4), (4,5), (6,1)*? What is the best value?

Answer

In the first order *(1,2)*, *(3,4)*, and *(5,6)* are chosen; the result is 6 vertices. In the second order *(2,3)* and *(5,6)* are chosen; one endpoint of each of the remaining four edges is in the cover, and the result is *{2, 3, 5, 6}*, 4 vertices. The best cover has 3 vertices, for example *{1, 3, 5}*. The ratios are 2 and *4/3*. The order of the edges changes the result, but the guarantee holds for every order.

6. Let $W = 50$ and let the items, as (value, weight), be (60, 10), (100, 20), (120, 30). What do the ratio-ordered greedy rule, the improved rule, and the best 0/1 solution find? What is the value of the fractional version?

Answer

The ratios are 6, 5, and 4. The greedy rule takes (60, 10) and (100, 20) (30 kg); (120, 30) does not fit (60 kg). The greedy value is 160. The most valuable single item that fits is 120; the improved rule finds $\max(160, 120) = 160$. The best 0/1 solution takes (100, 20) and (120, 30): exactly 50 kg, value 220. The ratio is $220/160 = 1.375 \le 2$. In the fractional version the first two items and $2/3$ of the third are taken: $60 + 100 + 80 = 240$.

7. Carry out the first step of power iteration for $A = [[2, 1], [1, 2]]$ and $x_0 = (1, 0)$. Given that the eigenvalues are 3 and 1, about how many steps are needed for the error to fall by a factor of 10^{-6}?

Answer

$y = A \cdot x_0 = (2, 1)$, $\|y\| = \sqrt{5} \approx 2.2361$, $x_1 = (0.8944, 0.4472)$. The Rayleigh quotient is $x_1^T A x_1 = 2.8$; the true value is 3. The error shrinks by a factor of about $1/3$ per step. For $(1/3)^k < 10^{-6}$, $k > 6 / \log_{10} 3 \approx 12.6$, so 13 steps are needed.

8. Is a TSP instance in which all distances are 1 or 2 metric? Can the MST-based 2-approximation be applied to it?

Answer

Yes, it is metric. For all i, j, k, $d_{ij} \le 2$; also $d_{ik} + d_{kj} \ge 1 + 1 = 2$. The triangle inequality always holds. So the MST-based algorithm can be applied and the guarantee of $2 \cdot OPT$ is valid. The construction in the Hamiltonian cycle proof, on the other hand, is not metric because it makes the heavy edge $\rho n + 1$; once the weights are restricted to 1 and 2, that proof does not work.

6.12 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 35: approximation algorithms for vertex cover, metric TSP, and set cover, with the full proof of the Hamiltonian cycle reduction.
- D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms* (2011): presents the techniques of greedy algorithms, local search, linear programming, and approximation schemes in an orderly sequence; the electronic version of the book is free on the authors' website.
- V. V. Vazirani, *Approximation Algorithms* (2001): gives covering problems, the knapsack FPTAS, and primal–dual methods with short and precise proofs.
- G. H. Golub and C. F. Van Loan, *Matrix Computations* (4th ed., 2013): convergence analysis of power iteration, inverse and shifted iteration, the Lanczos method.

6.13 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 35: Approximation Algorithms).
2. N. Christofides, “Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem”, Report 388, Carnegie Mellon University, 1976.
3. G. H. Golub, C. F. Van Loan, *Matrix Computations*, 4th ed., Johns Hopkins University Press, 2013.
4. R. L. Graham, “Bounds on Multiprocessing Timing Anomalies”, *SIAM Journal on Applied Mathematics* 17(2), 1969.
5. A. R. Karlin, N. Klein, S. Oveis Gharan, “A (Slightly) Improved Approximation Algorithm for Metric TSP”, *Proceedings of the 53rd ACM Symposium on Theory of Computing (STOC)*, 2021.
6. M. Sipser, *Introduction to the Theory of Computation*, 3rd ed., Cengage, 2012 (3-SAT → vertex cover reduction).
7. V. V. Vazirani, *Approximation Algorithms*, Springer, 2001.
8. D. P. Williamson, D. B. Shmoys, *The Design of Approximation Algorithms*, Cambridge University Press, 2011.

PART III

Other Models of Computation

Chapters 7–9

Randomized Algorithms

Not every step of an algorithm has to be determined by the input. An algorithm can flip a coin, roll a die, or pick a random element of an array. This small change has surprising consequences. Quicksort, which slows down on sorted input, no longer stays slow on any particular input once it chooses its pivot at random. Whether a number of several hundred digits is prime can be decided in milliseconds, without ever looking at its factors, with a vanishingly small probability of error. The weakest cut of a graph can be found by merging random edges. The notes first introduce randomized algorithms, expected running time, and the probability tools needed. They then present the two basic kinds, Las Vegas and Monte Carlo algorithms, and show how repetition reduces the error probability. After that the examples are studied one by one. The first are the Fisher–Yates shuffle, the bias of the naive shuffle, and BogoSort. They are followed by quicksort with a random pivot, the expected number of comparisons, and randomized selection. The last examples are π estimation by the Monte Carlo method, the Miller–Rabin primality test, and Karger's minimum cut algorithm. The notes end with pseudorandom number generators.

LEARNING OUTCOMES

Concepts

Define a randomized algorithm; distinguish expected running time from average-case running time; explain the difference between Las Vegas and Monte Carlo algorithms.

Shuffling and BogoSort

Write the Fisher–Yates shuffle and prove that it produces every permutation with equal probability; show by counting why the naive shuffle is biased; compute the expected $\Theta(n \cdot n!)$ running time of BogoSort.

Randomized QuickSort

Trace partitioning with a random pivot step by step; justify the expected $O(n \log n)$ and worst-case $O(n^2)$ running times; compute the expected number of comparisons with indicator variables; explain the expected $O(n)$ running time of randomized selection.

Monte Carlo
Compute the error probability in π estimation, the Miller–Rabin primality test, and Karger's minimum cut algorithm, and reduce it with independent repetitions.

7.1 Randomness and algorithms

7.1.1 Decisions based on chance

The word algorithm usually brings to mind a fully determined sequence of steps. Given the same input, the algorithm takes the same steps in the same order and produces the same result. Such algorithms are called **deterministic**. For some problems, however, it pays to introduce chance into the algorithm's decisions on purpose. At some point the algorithm flips a coin, rolls a die, or picks one of the elements of an array at random, and continues along its path according to that choice.

Games of chance are the most familiar example of this idea. Consider a game played with two dice. If the total is 7 or 11, the player wins; otherwise the dice are rolled again. Two dice have 36 equally likely outcomes. The outcomes with total 7 are (1, 6), (2, 5), (3, 4), (4, 3), (5, 2), and (6, 1); there are 6 of them. The outcomes with total 11 are (5, 6) and (6, 5); there are 2 of them.

> *P(7) = 6/36, P(11) = 2/36, P(win) = 8/36 = 2/9 ≈ 0.22*



Figure 7.1: Two dice: 3 and 4, total 7. The player wins on this roll.

The probability of winning on one roll is about 0.22. How many rolls does the player make before winning? This number differs from game to game: sometimes the first roll wins, sometimes ten are needed. But its average is known. Since every roll has the same winning probability and the rolls are independent of each other, the expected number of rolls is *1 / (2/9) = 4.5*. This result comes from the geometric distribution. In 200,000 games played with a fixed seed (`random.Random(42)`), the average number of rolls came out as 4.502, in agreement with the formula.

Two observations from the dice game recur throughout these notes. First, how long a single run will take is not known in advance; what is known is the expected value of the running time. Second, the expected value is found by a probability computation and can be checked with a long simulation.

7.1.2 Randomized algorithms and expected running time

Definition 7.1

A **randomized algorithm** is an algorithm that uses random numbers in addition to its input and makes some of its decisions according to these numbers. Two runs on the same input may follow different steps; the running time, and sometimes the output itself, varies from run to run.

Figure 7.2 shows this structure. The algorithm has two inputs: the input of the problem and a source of random bits. Even when the input stays the same, the random bits change; so the path the algorithm follows can change too.

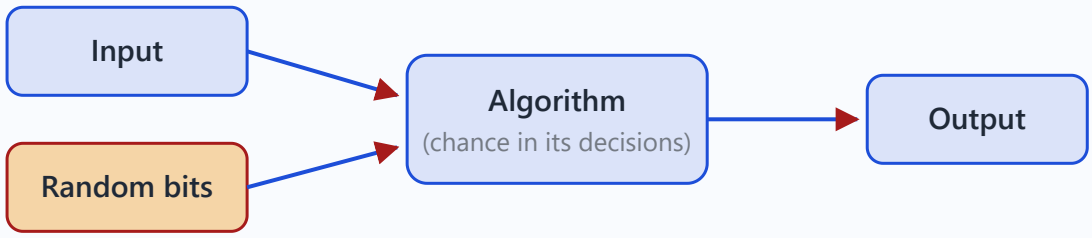


Figure 7.2: A randomized algorithm: the input and random bits enter the algorithm, and the output is produced. The algorithm's decisions involve an element of chance.

The running time of a deterministic algorithm on a fixed input is a single number. For a randomized algorithm, on the same input the time is a **random variable**: it can take a different value in each run. So the analysis examines the **expected value** $E[T(n)]$ of the time instead of a single number. There is an important detail here. The expectation is taken over the algorithm's own coin flips, not over the input. The input is fixed and may even be the worst one; the probability comes only from the random choices the algorithm makes.

Definition 7.2

The **expected running time** of a randomized algorithm, for each input of size n, is the expected time taken over the algorithm's random choices. By the expected running time of the algorithm we mean the largest of these values over all inputs:

$$T_{exp}(n) = \max_{|x| = n} E[\, T(x) \,]$$

This definition must not be confused with **average-case** analysis. Average-case analysis is applied to a deterministic algorithm and makes an assumption about the inputs: for example, it assumes that every ordering of the array is equally likely. If the assumption fails, the result is invalid too. In real life data often arrive sorted or nearly sorted, so the equal-probability assumption is often violated. The expected running time, on the other hand, makes no assumption about the input. It holds for each input separately, even for the worst one.

Example 7.1

Quicksort that always chooses the first element of the subarray as pivot runs in $O(n \log n)$ time on average if the input is in random order. This is an average-case result. If the same algorithm is given a sorted array of 10 elements, it makes exactly 45 comparisons on every run, because this input produces the worst case every time. The version that chooses the pivot at random, on the same sorted array, makes about 24.4 comparisons on average, and this value does not depend on the order of the input. For a sorted array, a reverse-sorted array, or any other arrangement, the expected value is the same. The value 24.4 comes from the formula for the expected number of comparisons, $2(n + 1)H_n - 4n$.

7.1.3 Why randomness?

Randomness looks at first like a source of uncertainty. Even so, it brings four important gains in algorithm design.

Protection against bad inputs. Deterministic quicksort that chooses the first element as pivot falls to $O(n^2)$ time on a sorted array. Sorted arrays are very common in practice. An attacker who knows the rules of the algorithm can deliberately prepare the worst input. When the pivot is chosen at random, no input is consistently bad. The bad case arises only if the algorithm's own choices go unluckily, and the probability of that is very small.

Simplicity. A randomized algorithm is often much shorter and easier to understand than the best known deterministic algorithm. Karger's minimum cut algorithm is a loop of a few lines. The randomized selection algorithm is a small modification of quicksort. The deterministic solutions to these problems are either long and intricate or slow in practice.

Speed. The Miller–Rabin test decides the primality of numbers with hundreds of digits within milliseconds. The error probability can be made as small as desired. Key generation in cryptography relies on this speed.

Sampling and symmetry breaking. Some tasks are either very hard or impossible without randomness. Monte Carlo simulations, drawing a random sample from a large dataset, and electing a leader among identical processes in a distributed system are examples. Two identical processes running the same program cannot be told apart by deterministic rules; a random number breaks this symmetry.

These gains have a price. The guarantee for time or accuracy is no longer exact but probabilistic. One cannot say "this algorithm is always $O(n \log n)$"; one says "its expected time is $O(n \log n)$" or "the probability of error is less than 10^{-12}". The algorithm also needs a good random number generator. Flaws in the generator break the assumptions of the analysis.

7.1.4 Probability tools

A few basic probability tools are used again and again in analyzing randomized algorithms. Table 7.1 summarizes them; each is then explained with a short example.

Concept	Definition / rule	Example
Expected value	$E[X] = \Sigma\, x \cdot P(X = x)$	the expected value of one die is 3.5
Linearity	$E[X + Y] = E[X] + E[Y]$ (independence not required)	total of two dice: 7
Indicator variable	$X_A = 1$ if A occurs, 0 otherwise; $E[X_A] = P(A)$	in QuickSort, "were i and j compared?"
Geometric distribution	in independent trials with success probability p, the expected number of trials until the first success is $1/p$	BogoSort: $p = 1/n! \to n!$ shuffles
Independent repetition	the probability that all of k independent trials, each failing with probability q, fail is q^k	Miller–Rabin: $(1/4)^k$
Markov's inequality	if $X \geq 0$, then $P(X \geq c \cdot E[X]) \leq 1/c$	Las Vegas → Monte Carlo

Table 7.1: Probability tools used in these notes.

Expected value. Each value a random variable can take is multiplied by its probability and the products are summed. Each face of a die comes up with probability 1/6: $E[X] = (1 + 2 + 3 + 4 + 5 + 6) / 6 = 3.5$. The expected value itself may never be observed; no die shows 3.5. The expected value is the value the average approaches when many trials are made.

Linearity. The expected value of the sum of two random variables is the sum of their expected values. For the total of two dice, $3.5 + 3.5 = 7$ is obtained; there is no need to add up the 36 outcomes one by one. The strongest feature of this rule is that it does not require independence. The rule holds even when the variables depend on each other. The analysis of quicksort relies on exactly this property: the comparisons are not independent of each other, but the expected total is still the sum of the individual expected values.

Indicator variable. For an event A, X_A takes the value 1 if the event occurs and 0 if it does not. Its expected value is $1 \cdot P(A) + 0 \cdot (1 - P(A)) = P(A)$. If we are to count how many times something happens, we write an indicator for each possible event, the count is the sum of the indicators, and by linearity the expected value reduces to the sum of the probabilities. In quicksort the question "were z_i and z_j compared?" is such an indicator.

Geometric distribution. Let each trial have success probability p and let the trials be independent. If X is the number of trials up to the first success, then $P(X = k) = (1 - p)^{k-1} p$ and $E[X] = 1/p$. This result can also be seen by a short argument. The first trial is always made. With probability p the job is done; with probability $1 - p$ the process starts over and the same expected value is spent again: $E = 1 + (1 - p) E$, hence $E = 1/p$. In the dice game $p = 2/9$ and the expected number of rolls is 4.5.

Independent repetition. If a trial fails with probability at most q, the probability that all of k independent trials fail is at most q^k. As long as q is less than 1, this value decreases exponentially in k. The error of Monte Carlo algorithms is reduced in this way.

Markov's inequality. A random variable that takes no negative values exceeds c times its expected value with probability at most $1/c$. The reason is simple: when X is at least $c \cdot E[X]$, it contributes at least $c \cdot E[X] \cdot P(X \geq c \cdot E[X])$ to the expected value. This contribution cannot exceed $E[X]$. The inequality gives a tail bound from the expected value alone, knowing nothing about the distribution; so it is crude but valid everywhere.

Common mistake

Looking for independence for linearity. $E[X + Y] = E[X] + E[Y]$ is always true. Independence is needed only for products: the equality $E[XY] = E[X]\,E[Y]$ holds in general only for independent variables. This is why summing dependent indicators in the comparison analysis of quicksort is no problem.

7.2 Las Vegas and Monte Carlo

7.2.1 Two kinds of randomized algorithms

Randomized algorithms fall into two groups according to what the randomness affects. In the first group the result is always correct and only the running time depends on chance. In the second group the running time is bounded, but the result may be wrong with a small probability. These groups are named after two cities famous for their casinos.

Definition 7.3

A **Las Vegas algorithm** always gives the correct result; its running time is random and its expected time is analyzed. A **Monte Carlo algorithm**, on the other hand, finishes within a predetermined time bound; its result may be wrong with a small probability.

	Las Vegas	Monte Carlo
Correctness	always the correct result	a wrong result with small probability
Running time	random; the expected time is analyzed	bounded (a deterministic upper bound)

	Las Vegas	Monte Carlo
What is random	how long it takes	whether the result is correct
Examples	randomized QuickSort, QuickSelect, BogoSort, Fisher–Yates	Miller–Rabin, Karger's minimum cut, π estimation

Table 7.2: Comparison of Las Vegas and Monte Carlo algorithms.

Quicksort with a random pivot is a Las Vegas algorithm. Whatever pivot is chosen, the array ends up correctly sorted. Unlucky pivots only prolong the work. BogoSort is also Las Vegas: when it stops, the array is certainly sorted, but when it will stop is not known. The Fisher–Yates shuffle is a slightly different case. Its running time is exactly $n - 1$ swaps on every run; what is random is the permutation produced. In this algorithm the "correct result" is not a single array but the desired probability distribution, and this distribution is achieved exactly every time. So it too is placed in the Las Vegas group.

The Miller–Rabin test, on the other hand, is Monte Carlo. The number of rounds is fixed in advance and each round performs a bounded number of multiplications. But the answer "prime" may be wrong with a small probability. Karger's algorithm also performs a fixed number of contractions and sometimes returns a cut that is not minimum. The π estimate throws the desired number of points and only approaches the true value.

7.2.2 One-sided and two-sided error

Monte Carlo algorithms are also classified by the direction of the error. In a decision problem, that is, a problem whose answer is "yes" or "no", if the algorithm can be wrong on only one of the answers, the error is **one-sided**. Miller–Rabin is the typical example. If the test says "composite", this is certain, because it has found concrete evidence that the number is not prime. Only the answer "prime" can be wrong. One-sided error has a great advantage: after repetitions, even a single "composite" answer settles the matter.

If the error can occur in both directions, it is **two-sided**. Numerical simulations such as the π estimate can deviate both above and below the true value. For such algorithms the margin of error shrinks as the number of samples grows, but the direction in which a single run deviates is not known. The error of Karger's algorithm is also one-sided, but in a different direction: the cut found can never be smaller than the minimum, only larger. So the smallest of several runs is taken.

7.2.3 Reducing the error probability

The error probability of a Monte Carlo algorithm can be large in a single run. But if the algorithm is run many times with independent random choices, the error shrinks quickly. This technique is called **amplification**.

Theorem 7.1 (Amplification by repetition)

Let a Monte Carlo algorithm with one-sided error have error probability at most q in a single round. Run the algorithm k times independently, and if one of the rounds gives the definite answer, choose that answer. The error probability of this combined algorithm is at most q^k.

Proof. The combined algorithm errs only when all rounds err. Because the rounds use independent random choices, this probability is the product of the individual error probabilities and is at most $q \cdot q \cdot \ldots \cdot q = q^k$. □

In Miller–Rabin, for a composite number the probability of error in one round is at most $1/4$. After twenty rounds the error probability drops to at most $4^{-20} \approx 9.1 \cdot 10^{-13}$. This is smaller than the probability of a spontaneous bit error occurring in computer hardware. Figure 7.3 shows on a logarithmic scale how the bound falls with the number of rounds: each new round divides the bound by 4.

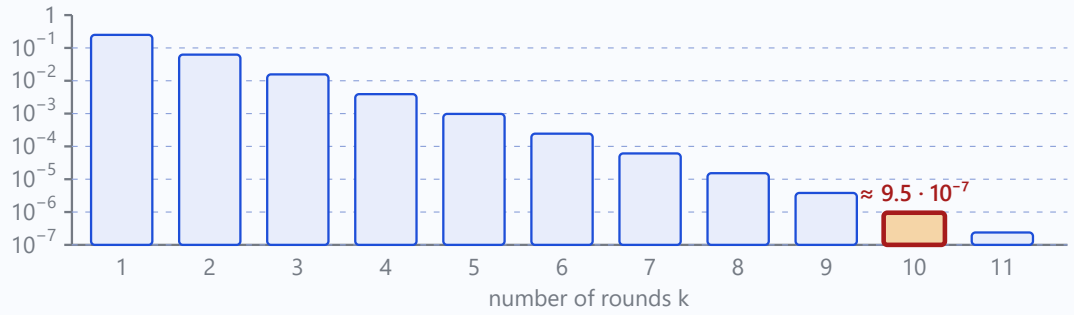


Figure 7.3: The Miller–Rabin error bound 4^{-k} on a logarithmic scale. At $k = 9$ the bound is $3.8 \cdot 10^{-6}$; at $k = 10$ it drops to $9.5 \cdot 10^{-7}$, falling below 10^{-6} for the first time.

Example 7.2

At least how many rounds are needed to push the error probability of Miller–Rabin below 10^{-9}? The condition is $4^{-k} < 10^{-9}$, that is, $4^k > 10^9$. $4^{14} = 268{,}435{,}456$ is not enough. $4^{15} = 2^{30} = 1{,}073{,}741{,}824$, however, exceeds 10^9. At least 15 rounds are needed. The same computation gives 10 rounds for 10^{-6}. Since the bound falls to a quarter, not a half, the number of rounds needed is about $\log_4 10 \approx 1.66$ times the number of decimal digits desired.

In Karger's algorithm the success probability of a single run is at least $2/(n(n-1))$. This is a small number; for a graph with 100 vertices it is about 0.0002. But if the algorithm is repeated $n(n-1)/2 \cdot \ln n$ times, the probability that all runs fail falls to at most $1/n$. However small the success probability is, if it has a known lower bound greater than zero, repetition raises it to the desired level.

7.2.4 Converting between the two kinds

Las Vegas and Monte Carlo algorithms can be converted into each other. Both conversions are used in practice.

From Las Vegas to Monte Carlo. A Las Vegas algorithm with expected running time $E[T]$ is stopped after $2 \cdot E[T]$ steps. If the algorithm has finished in this time, its answer is correct. If not, "no answer" is returned. By Markov's inequality, $P(T \geq 2\,E[T]) \leq 1/2$; that is, the new algorithm fails to answer with probability at most 1/2. Its running time is now strictly bounded. If the bound is taken as $c \cdot E[T]$, the probability of no answer is at most $1/c$.

Example 7.3

On an array of four distinct elements, the number of shuffles BogoSort makes follows a geometric distribution with success probability $p = 1/24$ and has expected value 24. Let the algorithm be stopped after 48 shuffles. Markov's inequality gives the bound 1/2 on the probability of no answer. The true probability is much smaller: the probability that all of the first 48 shuffles fail is $(23/24)^{48} \approx 0.130$. A simulation of 100,000 trials with a fixed seed gave 0.129. The Markov bound is correct but crude, because it uses only the expected value of the distribution.

From Monte Carlo to Las Vegas. If the answer of a Monte Carlo algorithm can be verified quickly, the algorithm is repeated until a verified answer is found. For example, the factors found by a randomized algorithm that tries to factor a number are easily checked by multiplying them. If the success probability of a single trial is p, the expected number of trials is $1/p$ by the geometric distribution. The result is always correct; only the running time becomes random. For $p = 1/3$ the expected number of repetitions is 3; in a simulation of 100,000 trials with seed 42 the average came out as 2.994.

Common mistake

Repeating an unverifiable answer "until it comes out right". The "prime" answer of Miller–Rabin cannot be verified quickly; no short proof of primality is at hand. So Miller–Rabin cannot be converted into Las Vegas this way. The conversion works only when the answer itself can be checked cheaply.

7.3 Shuffling

7.3.1 The problem and the correctness criterion

Putting the elements of an array into a random order, that is, **shuffling**, is a common task. In card games the deck is shuffled, music players play a playlist in shuffled order, and algorithm tests generate randomly ordered test data. Another form of quicksort with a random pivot is to shuffle the array first and then run deterministic quicksort.

What does it mean for a shuffle to be "correct"? A weak criterion is that each element is equally likely to be in each position. The strong and actual criterion is that each of the $n!$ different orderings of n elements, that is, each **permutation**, is produced with probability exactly $1/n!$. Such a shuffle is called **uniform**. The strong criterion includes the weak one; the converse is not true. For example, let a method turn the array [1, 2, 3] into only one of its three cyclic shifts: [1, 2, 3], [2, 3, 1], or [3, 1, 2]. Let the three results be equally likely. This method puts each element in each position with probability 1/3. But it never produces three of the six permutations.

n	n! permutations
3	6
4	24
10	3,628,800
52	$\approx 8.07 \cdot 10^{67}$

Table 7.3: The number of permutations $n!$ grows very fast.

The number of permutations of a 52-card deck is about $8 \cdot 10^{67}$. This number is so large that the order of a well-shuffled deck has most likely never been seen in any deck before. The shuffling algorithm must choose from this huge set with equal probability, yet it cannot list the permutations one by one. The solution is to break the choice into small steps.

7.3.2 The Fisher–Yates shuffle

The **Fisher–Yates shuffle**, also known as the Knuth shuffle, works like this: the element that will go to the last position of the array is chosen at random among all the elements and placed there. Then the element for the second-to-last position is chosen among the elements not yet placed. This continues down to the first position. At each step one position becomes final and never changes again.

```
static void shuffle(int[] a) {
  for (int i = a.length - 1; i > 0; i--) {
    int j = rng.nextInt(i + 1); // 0 ≤ j ≤ i
    int t = a[i]; a[i] = a[j]; a[j] = t;
  }
}
```

In the code, `rng` is a shared `java.util.Random` object created once in the class. The loop starts at $i = n - 1$ and goes down to $i = 1$. In each round `rng.nextInt(i + 1)` produces a uniformly distributed integer j between 0 and i (both included). Then `a[i]` and `a[j]` are swapped. $j = i$ can also come up; in that case the element is swapped with itself and stays in place. This possibility is important: an element staying where it started is also a valid outcome, and the shuffle must produce this outcome with the right probability too.

The loop does not run for $i = 0$. At that point a single element is left and there is nothing else to choose. A total of $n - 1$ swaps are made and each swap takes constant time. The running time is the same on every run: $\Theta(n)$. The extra memory is $O(1)$; the shuffle is done in place. What varies is the result itself, not the running time. Java's `Collections.shuffle(list)` method also uses this algorithm.

Example 7.4

Let the array $a = [1, 2, 3, 4, 5]$ be shuffled. Let the random number generator give the values $j = 1, 3, 0, 1$ in turn. Table 7.4 gives each step.

i	choice range	j	operation	array
—	—	—	start	[1, 2, 3, 4, 5]
4	a[0..4], 5 elements	1	a[4] ↔ a[1]	[1, 5, 3, 4, **2**]
3	a[0..3], 4 elements	3	j = i, no change	[1, 5, 3, **4**, **2**]
2	a[0..2], 3 elements	0	a[2] ↔ a[0]	[3, 5, **1**, **4**, **2**]
1	a[0..1], 2 elements	1	j = i, no change	[3, **5**, **1**, **4**, **2**]

Table 7.4: Fisher–Yates trace, $a = [1, 2, 3, 4, 5]$. Bold values are positions that are final.

In the first round each of the 5 elements could have gone to the last position with probability 1/5; the chosen $j = 1$ is the value 2. After the swap `a[4] = 2` is final. In the second round $j = 3 = i$ came up, so the value 4 stays in place. In the third round `a[0] = 1` is chosen among three elements and goes to `a[2]`; the old `a[2] = 3` moves to the front. In the last round $j = i$ comes up again. `a[0] = 3` is the only element left and settles by itself. The result is [3, 5, 1, 4, 2]. The code, run in Java with a generator that returns the same j values, gives this array as well.

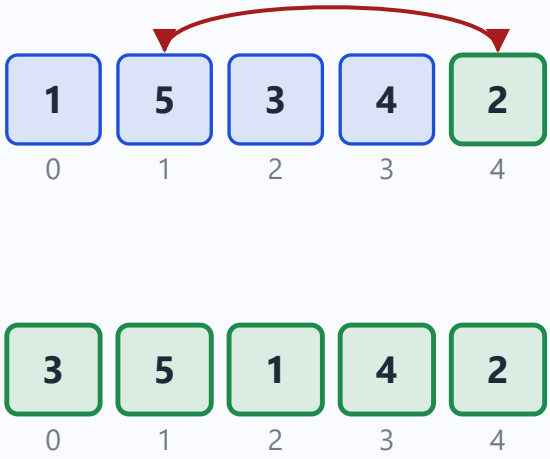


Figure 7.4: Two moments of the Fisher–Yates trace. Left, the array after the first swap: a[4] = 2 is final (green). Right, when the loop ends: [3, 5, 1, 4, 2]; all positions are final.

7.3.3 Proof of correctness

The correctness of Fisher–Yates is shown by a short counting argument. In the round $i = n - 1$ the algorithm chooses one of n values, in the round $i = n - 2$ one of $n - 1$ values, and so on; in the last round $i = 1$ one of 2 values is chosen. The number of possible sequences of choices is $n \cdot (n - 1) \cdot \ldots \cdot 2 = n!$, and every sequence of choices is equally likely.

Theorem 7.2 (Fisher–Yates is uniform)

The Fisher–Yates shuffle produces each of the $n!$ permutations of an array of distinct elements with probability exactly $1/n!$.

Proof. There is a one-to-one correspondence between sequences of choices and permutations. Let a desired permutation be given. The element in the last position is determined by this permutation. In the round $i = n - 1$ that element is at exactly one position of the array; that position must be j. After the swap the same argument is repeated for the remaining front part of $n - 1$ elements. Thus for every permutation there is a single sequence of choices that produces it. There are $n!$ sequences of choices and $n!$ permutations. Since the probability of each sequence of choices is $(1/n) \cdot (1/(n - 1)) \cdot \ldots \cdot (1/2) = 1/n!$, the probability of each permutation is also $1/n!$. □

The proof shows one thing clearly: the uniformity of the shuffle comes from the number of equally likely paths being equal to the number of permutations. In the five-element example the number of possible sequences of draws is $5 \cdot 4 \cdot 3 \cdot 2 = 120 = 5!$; each permutation is obtained exactly once. The array [1, 2, 3] was shuffled 600,000 times with a fixed seed (`new Random(42)`). Each of the six permutations came up between 99,754 and 100,315 times. The expected value is 100,000, and the deviations are at the level of random fluctuation expected for this many trials.

7.3.4 The naive shuffle is biased

The first way of writing a shuffle that comes to mind is this: swap each position with a random position of the array. The code looks short and innocent.

```
// WRONG: for each i, pick from 0..n-1
for (int i = 0; i < n; i++) {
  int j = rng.nextInt(n);
  swap(a, i, j);
}
```

In this code, for each i the index j is chosen not from a shrinking range as in Fisher–Yates but always from $0..n-1$, the range of the whole array. The array really does get mixed at every round, but the result is not uniform. The reason is seen by counting. Each of the n draws can take n different values. The number of equally likely paths is n^n. The number of permutations, however, is $n!$. If every permutation were equally likely, each would be produced by exactly $n^n / n!$ paths, and this number would be an integer.

For $n = 3$ there are $3^3 = 27$ paths and $3! = 6$ permutations. 27 is not divisible by 6; therefore the permutations cannot be equally likely. This argument holds in general for $n \geq 3$. The number $n - 1$ divides $n!$ but has no common divisor with n^n; so $n^n / n!$ is never an integer.

result	paths	probability	should be	simulation (600,000)
123	4	4/27 ≈ 0.148	1/6 ≈ 0.167	88,843
132	5	5/27 ≈ 0.185	1/6 ≈ 0.167	110,659
213	5	5/27 ≈ 0.185	1/6 ≈ 0.167	112,175
231	5	5/27 ≈ 0.185	1/6 ≈ 0.167	110,602
312	4	4/27 ≈ 0.148	1/6 ≈ 0.167	89,008
321	4	4/27 ≈ 0.148	1/6 ≈ 0.167	88,713

Table 7.5: The 27 sequences of draws of the naive shuffle on the array [1, 2, 3], counted one by one. The last column is the result of a simulation of 600,000 trials with seed 42.

In Table 7.5 the results 132, 213, and 231 are obtained by five paths each and the other three by four paths each. The expected counts are $600{,}000 \cdot 5/27 \approx 111{,}111$ and $600{,}000 \cdot 4/27 \approx 88{,}889$; the simulation is very close to these values. The difference may look small, but it is systematic: it does not disappear as the number of trials grows; on the contrary, it becomes more pronounced. In a card game or a statistical experiment this bias corrupts the results.

The naive shuffle does not satisfy even the weak criterion. Read off the positions of the value 3 from the table: the probability of landing in the first position is $(4 + 4)/27 = 8/27$, in the middle position $10/27$, and in the last position $9/27$. All three should have been 1/3. For an array of four elements the

situation is even messier: $4^4 = 256$ paths are distributed over 24 permutations. 256/24 ≈ 10.67 is not an integer; an exact count made in Java showed permutations produced by at least 8 and at most 15 paths.

> **Common mistake**
>
> **Checking the uniformity of a shuffle by eye.** The outputs of the naive shuffle look perfectly random to the eye. The bias shows up only by counting or with a large number of trials. The correct pattern is `rng.nextInt(i + 1)` : the range of choice shrinks in every round and stays limited to the elements not yet placed.

7.3.5 A one-character bug: nextInt(i)

What happens if `rng.nextInt(i)` is mistakenly written in the Fisher–Yates code instead of `rng.nextInt(i + 1)` ? In this case j can be at most $i - 1$; `a[i]` is always swapped with an element before it, and no element can stay in place. For the array [1, 2, 3], in the round $i = 2$ we have $j \in \{0, 1\}$, and in the round $i = 1$ we must have $j = 0$. There are only $2 \cdot 1 = 2$ paths.

j at i = 2	after the first swap	j at i = 1	result
0	[3, 2, 1]	0	[2, 3, 1]
1	[1, 3, 2]	0	[3, 1, 2]

Table 7.6: All paths for the array [1, 2, 3] with `nextInt(i)`.

Each of the two results comes up with probability 1/2. The other four permutations, including the sorted [1, 2, 3], are never produced. This modified method has a name of its own: **Sattolo's algorithm**. It produces, with equal probability, the $(n - 1)!$ permutations that consist of a single cycle. In some applications exactly this is wanted; for example, when a loop is to be built in which every element is sent to another element and nobody lands on itself. But it is not a uniform shuffle. A difference of a single character completely changes the probability distribution of the output.

7.4 BogoSort

7.4.1 The algorithm

BogoSort is one of the worst-performing sorting algorithms known. It is also called "stupid sort" or permutation sort. Its logic can be stated in one sentence: shuffle the list at random until it is sorted. It is not used in real applications. But it is the simplest example of Las Vegas algorithms and of the geometric distribution; that is why it is instructive.

1. Is the list sorted? If yes, stop.

2. If not, shuffle the list at random and go back to step one.

```
static void bogoSort(int[] a) {
  while (!isSorted(a))
    shuffle(a);         // Fisher–Yates
}
static boolean isSorted(int[] a) {
  for (int i = 0; i + 1 < a.length; i++)
    if (a[i] > a[i + 1]) return false;
  return true;
}
```

The method `isSorted` checks adjacent pairs of elements from left to right. If in some pair the left element is larger than the right, the array is not sorted and the method immediately returns `false`. If all pairs are in order, it returns `true`. In the worst case it makes $n - 1$ comparisons: $O(n)$. `bogoSort` shuffles an unsorted array with Fisher–Yates and repeats the check. It is important to use a correct method for shuffling; with the naive shuffle some permutations come up more rarely and the analysis becomes invalid.

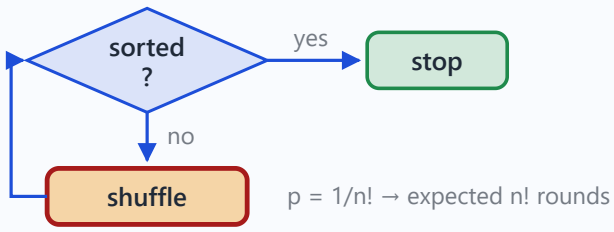


Figure 7.5: The BogoSort flow: if sorted, it stops; if not, it shuffles and asks again. The success probability of each round is $p = 1/n!$ and the expected number of rounds is $n!$.

7.4.2 One run

Example 7.5

Let the list [5, 2, 8, 4] be sorted with BogoSort. Each shuffle is done with Fisher–Yates. Table 7.7 gives all the steps of one run made with a seeded generator. In each row the first out-of-order adjacent pair found by `isSorted` is shown.

shuffle	array	first inverted pair	result
0 (start)	[5, 2, 8, 4]	a[0] = 5 > a[1] = 2	not sorted
1	[5, 4, 2, 8]	a[0] = 5 > a[1] = 4	not sorted
2	[8, 4, 5, 2]	a[0] = 8 > a[1] = 4	not sorted

shuffle	array	first inverted pair	result
3	[4, 2, 8, 5]	a[0] = 4 > a[1] = 2	not sorted
4	[2, 8, 4, 5]	a[1] = 8 > a[2] = 4	not sorted
5	[5, 8, 2, 4]	a[1] = 8 > a[2] = 2	not sorted
6	[8, 4, 2, 5]	a[0] = 8 > a[1] = 4	not sorted
7	[2, 8, 5, 4]	a[1] = 8 > a[2] = 5	not sorted
8	[2, 4, 5, 8]	none	sorted, the algorithm stops

Table 7.7: A BogoSort run for [5, 2, 8, 4].

This run needed 8 shuffles. Of the 24 permutations of four elements only one is sorted; the expected number of shuffles is 24. This run was luckier than expected. Another run could finish in 3 shuffles or in 80. Note also that in most rounds `isSorted` stops at the first or second comparison. In a random array the probability that the first pair is inverted is 1/2; so the check takes far less than $n - 1$ comparisons on average. The shuffle, on the other hand, makes exactly $n - 1$ swaps in every round.

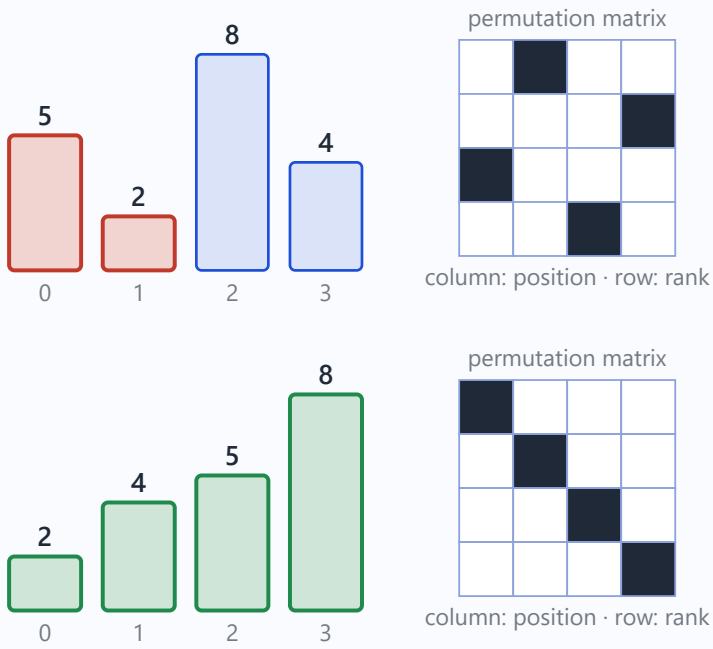


Figure 7.6: On the left the start [5, 2, 8, 4]; the red bars are the first inverted pair. On the right [2, 4, 5, 8] after the eighth shuffle. In the permutation matrix on the right, a dark cell shows the rank (row) and position (column) of a value; in the sorted array the diagonal fills in.

7.4.3 Analysis

Let the elements be distinct. Since Fisher–Yates produces every permutation with probability $1/n!$, the probability that a shuffle comes out sorted is $p = 1/n!$. The shuffles are independent of each other. The number of shuffles made until the sorted array is obtained follows the geometric distribution. Its expected value is $1/p = n!$.

Each round contains one shuffle ($O(n)$) and one check ($O(n)$). The expected running time is therefore $\Theta(n \cdot n!)$. In the literature this time is sometimes written as $O((n + 1)!)$. The two expressions are of the same magnitude, because $(n + 1)! = (n + 1) \cdot n!$ and $n + 1$ and n differ only by a constant factor.

n	expected shuffles n!	time
5	120	0.12 µs
10	3,628,800	3.6 ms
12	479,001,600	0.48 s
13	6,227,020,800	6.2 s
15	1,307,674,368,000	≈ 22 minutes
20	2,432,902,008,176,640,000	≈ 77 years

Table 7.8: The expected number of shuffles in BogoSort and the time on an imaginary machine that can make 10^9 shuffles per second.

The table shows what factorial growth means in practice. Twelve elements are sorted in half a second, fifteen elements take more than twenty minutes, and twenty elements take a human lifetime. The formula for the expected value was also checked by simulation. The trials were run with a fixed seed (`new Random(42)`) and from a shuffled, unsorted start. The average numbers of shuffles came out as follows: 23.9 for $n = 4$ (expected 24), 119.7 for $n = 5$ (expected 120), 714.5 for $n = 6$ (expected 720). 20,000 runs were made for each n.

Best case. If the array is already sorted, a single check is made and the algorithm finishes in $O(n)$ time.

Worst case. Here there is no upper bound. The probability that all of k shuffles come out unsorted is $(1 - 1/n!)^k$. This number approaches zero as k grows but is never zero for any k. So for every k a run lasting at least k rounds is possible. The expression $O((n + 1)!)$ is a bound only for the expected time, not for the worst case. Unlucky shuffles can go on forever; the probability of this goes to zero.

Example 7.6

What is the probability that BogoSort on an array of four elements has still not finished after 24 shuffles, that is, after the expected value? All of the first 24 shuffles must come out unsorted: $(23/24)^{24} \approx 0.36$. More than a third of the runs exceed the expected value. The probability of not having finished after 100 shuffles is $(23/24)^{100} \approx 0.014$. The tail of the geometric distribution thins out exponentially, but it never breaks off.

Common mistake

Mistaking the expected-time bound for a worst-case bound. The sentence "BogoSort is $O((n + 1)!)$" is true only for the expected time. For Las Vegas algorithms the worst case is often unbounded. Which running time is being discussed should always be stated explicitly.

7.5 Randomized QuickSort

7.5.1 Pivot choice and its effect

Quicksort works briefly as follows. A **pivot** element is chosen from the array. The array is **partitioned** so that the elements smaller than or equal to the pivot stay on the left and the larger ones on the right, and the pivot settles between the two groups, in its final place. Then the two sides are sorted separately, recursively. Partitioning takes $\Theta(n)$ time. The total running time of the algorithm depends entirely on how evenly the pivot splits.

If the pivot splits the array into two equal halves every time, there are $\log_2 n$ levels and the time is $\Theta(n \log n)$. If the pivot is every time the smallest or largest element of the subarray, one of the parts is empty and the other is just one element smaller. In this case there are n levels and the time rises to $\Theta(n^2)$. The rule for choosing the pivot is therefore the most important design decision of the algorithm. Table 7.9 compares the common strategies.

Pivot strategy	Bad input	Expected / typical	Worst case
First or last element	sorted or reverse-sorted array (very common)	$O(n \log n)$*	$O(n^2)$
Median of three	specially crafted arrays	$O(n \log n)$*	$O(n^2)$
Random element	none: badness comes only from bad luck	$O(n \log n)$, for every input	$O(n^2)$
True median (BFPRT)	none	$O(n \log n)$	$O(n \log n)$

Table 7.9: Pivot strategies. (*) For deterministic strategies, $O(n \log n)$ holds only if the input is in random order (average case).

Choosing the first or last element as pivot is the simplest rule, but it produces the worst case on an already sorted or reverse-sorted array. These inputs are very common in practice. The **median-of-three** rule chooses the median of the first, middle, and last elements as pivot. It works perfectly on a sorted array, but someone who knows the rule can craft special arrays that fool it. If the true median of the subarray is found and used as pivot, the split is always balanced. There is a deterministic algorithm that finds the median in linear time; it is known by the initials of its inventors, BFPRT (Blum, Floyd, Pratt, Rivest, Tarjan). But its constants are large and it is slow in practice.

The difference made by a random pivot is subtle but important. For deterministic strategies, $O(n \log n)$ rests on the assumption that the input is in random order. With a random pivot the expectation is over the algorithm's own choices; so it holds for every input. Still, the worst case does not disappear.

If the algorithm picks an unlucky pivot every time, the time is again $O(n^2)$. What changes is that this case no longer depends on a particular input and that its probability is very small. In practice a random pivot or the median of three is preferred.

Common mistake

"A random pivot eliminates the worst case." This is false; the worst case is still $O(n^2)$. What a random pivot gives is that no input is consistently bad and that the probability of the bad case is very small.

7.5.2 A run with random pivots

Let quicksort with a random pivot be run on the array [6, 5, 3, 1, 8, 7, 2, 4]. Suppose that in this run the pivots happen to be chosen as 3, 1, 4, 6, and 8, in that order. In the trace, partitioning is done with two pointers. *i* advances from the left and skips elements smaller than the pivot; it stops at the first element greater than or equal to it. *j* advances from the right and skips elements larger than the pivot; it stops at the first element smaller than or equal to it. If $i < j$, the two elements are swapped and the search continues. When the pointers meet, the pivot is in its final place. This method is based on Hoare's partitioning idea. The pivot itself may move during this process; its position is tracked.

subarray	pivot	pointers and swaps	array afterward	pivot position
a[0..7]	3	i = 0 (6), j = 6 (2): swap; i = 1 (5), j = 3 (1): swap; meet	[2, 1, 3, 5, 8, 7, 6, 4]	2
a[0..1]	1	i = 0 (2), j = 1 (1): swap; meet	[1, 2, 3, 5, 8, 7, 6, 4]	0
a[1..1]	—	single element, already sorted	unchanged	—
a[3..7]	4	i = 3 (5), j = 7 (4): swap; meet	[1, 2, 3, 4, 8, 7, 6, 5]	3
a[4..7]	6	i = 4 (8), j = 7 (5): swap; i = 5 (7), j = 6 (6): swap; meet	[1, 2, 3, 4, 5, 6, 7, 8]	5
a[4..4]	—	single element	unchanged	—
a[6..7]	8	i skips 7 and stops at 8; j is already at 8; no swap	[1, 2, 3, 4, 5, 6, 7, 8]	7
a[6..6]	—	single element	array sorted	—

Table 7.10: Quicksort with random pivots on the array [6, 5, 3, 1, 8, 7, 2, 4]. The pivots are 3, 1, 4, 6, 8.

Let us trace the first partitioning in detail. The pivot is 3 and is at index 2. *i* stops immediately at `a[0] = 6`, because 6 is greater than the pivot. *j* starts from the right: 4 is greater than the pivot and is skipped; it stops at `a[6] = 2`. Since $0 < 6$, 6 and 2 are swapped and the array becomes [2, 5, 3, 1, 8, 7, 6, 4]. The search continues: *i* skips 2 and stops at `a[1] = 5`; *j* skips 6, 7, and 8 and stops at `a[3] = 1`. 5 and 1 are swapped: [2, 1, 3, 5, 8, 7, 6, 4]. In the next step *i* skips 1 and stops at the pivot itself (index 2); *j* skips 5 and also arrives at index 2. The pointers have met. The pivot 3 is in its final place: only smaller elements (2, 1) are to its left and only larger ones to its right.



Figure 7.7: Three moments of quicksort with a random pivot. Top, the start: the pivot 3 was chosen at random (black box); the dashed frame is the active subarray. Middle, the first partitioning is done: 3 is in its final place (index 2, green). Bottom, all elements are final.

In this run the pivots fell fairly evenly. The first pivot 3 split the eight elements into two parts of 2 and 5 elements. The depth of the recursion tree stayed at 4. If the smallest element had been chosen as pivot every time, the depth would have been 7. The final position of a pivot equals the number of elements in the array smaller than it; this is so whichever partitioning method is used. For example, there are five elements smaller than 6, so the position of 6 is index 5.

7.5.3 Code and partitioning trace

The following code is the complete quicksort with a random pivot. For partitioning the **Lomuto** method is used: the randomly chosen pivot is first moved to the end of the subarray, and then in a single left-to-right pass the smaller elements are gathered on the left. It does the same job as the two-pointer method in the trace; the pivots go to the same final places, only the swaps in between differ.

```
static Random rng = new Random();

static void quickSort(int[] a, int lo, int hi) {
  if (lo < hi) {
    int p = partition(a, lo, hi); // random
    quickSort(a, lo, p - 1);
    quickSort(a, p + 1, hi);
  }
}

static int partition(int[] a, int lo, int hi) {
  int r = lo + rng.nextInt(hi - lo + 1);
  swap(a, r, hi);               // pivot to the end
  int pivot = a[hi], k = lo;
  for (int i = lo; i < hi; i++)
    if (a[i] <= pivot)
      swap(a, i, k++);          // small ones to the left
  swap(a, k, hi);               // pivot into place
  return k;
}
```

The method `quickSort` is the standard recursive skeleton. If the subarray has at least two elements, `partition` is called and the two sides of the returned pivot position are sorted separately. The randomness is only in the first line of `partition`: the expression `lo + rng.nextInt(hi - lo + 1)` gives a uniformly distributed index *r* between *lo* and *hi* (both included). This element is moved to the end by swapping it with `a[hi]` and becomes the pivot.

The loop invariant is this: at the start of each round, `a[lo..k − 1]` contains the elements less than or equal to the pivot and `a[k..i − 1]` contains the elements greater than the pivot. If `a[i]` is less than or equal to the pivot, it is swapped with `a[k]` and *k* increases by one; thus the "small" region grows by one element. When the loop ends, `a[k]` is the first element of the large region. The pivot is swapped with it to put it in its final place, and *k* is returned. `partition` makes *hi* − *lo* comparisons.

Example 7.7

Let `partition(a, 0, 7)` be called for *a = [6, 5, 3, 1, 8, 7, 2, 4]* and let `rng.nextInt(8)` return 2. The pivot is `a[2] = 3`. Table 7.11 gives each step of the partitioning.

step	i	a[i] ≤ 3?	operation	k	array
pivot to the end	—	—	a[2] ↔ a[7]	0	[6, 5, 4, 1, 8, 7, 2, 3]
1	0	6: no	—	0	unchanged
2	1	5: no	—	0	unchanged
3	2	4: no	—	0	unchanged
4	3	1: yes	a[3] ↔ a[0]	1	[1, 5, 4, 6, 8, 7, 2, 3]

step	i	a[i] ≤ 3?	operation	k	array
5	4	8: no	—	1	unchanged
6	5	7: no	—	1	unchanged
7	6	2: yes	a[6] ↔ a[1]	2	[1, 2, 4, 6, 8, 7, 5, 3]
pivot to its place	—	—	a[2] ↔ a[7]	2	[1, 2, 3, 6, 8, 7, 5, 4]

Table 7.11: Lomuto partitioning, $r = 2$ (pivot 3).

Seven comparisons were made and `partition` returned 2. The pivot 3 again settled at index 2; it had gone to the same place in the two-pointer trace. But the rest of the array differs: the two-pointer method had produced [2, 1, 3, 5, 8, 7, 6, 4], while Lomuto produces [1, 2, 3, 6, 8, 7, 5, 4]. The contract of partitioning fixes only the position of the pivot and the contents of the two sides; the order within the sides depends on the method. The code was run in Java and gave the same array. On 2000 random arrays the result of `quickSort` was compared with `Arrays.sort` and all matched.

7.5.4 Best, worst, and expected case

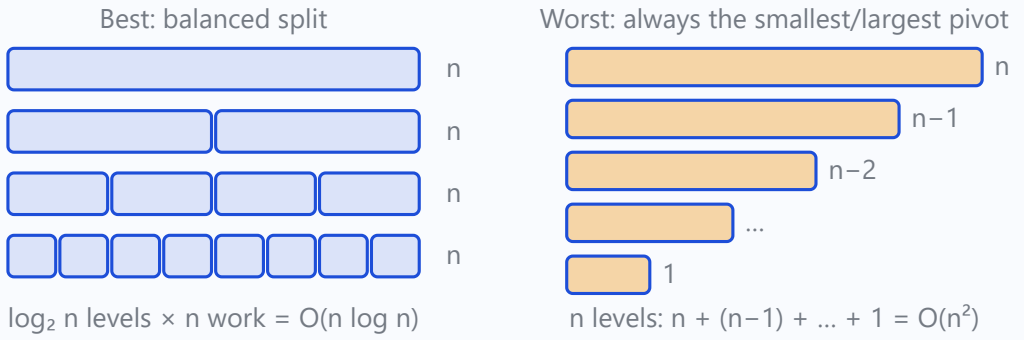


Figure 7.8: Left, balanced splitting: each of the $\log_2 n$ levels does a total of n work, time $O(n \log n)$. Right, the smallest or largest pivot every time: n levels and $n + (n - 1) + \ldots + 1 = O(n^2)$.

Case	Recurrence	Time
Best	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \log n)$
Expected (random pivot)	$T(n) = (1/n)\,\Sigma_q\,[T(q) + T(n - 1 - q)] + \Theta(n)$	$\Theta(n \log n)$
Worst	$T(n) = T(n - 1) + \Theta(n)$	$\Theta(n^2)$

Table 7.12: Recurrences for the three cases of quicksort.

The expected-case recurrence reads as follows. The random pivot chooses each of the n elements of the subarray with probability $1/n$. If the pivot is the q-th element in sorted order (counting from 0), the left part has q elements and the right part $n - 1 - q$. The recurrence averages over all values of q and adds the linear cost of partitioning. Its solution is $\Theta(n \log n)$. Instead of solving this recurrence directly, it is easier to compute the number of comparisons with indicator variables.

First an intuitive observation can be made. If the pivot is one of the elements falling in the middle half in sorted order (that is, it is neither in the smallest quarter nor in the largest quarter), the larger of the two parts has at most $3n/4$ elements. The probability of this "good pivot" event is 1/2. Every good pivot shrinks the part containing an element by at least a quarter. On the path of an element there can be at most $\log_{4/3} n$ good pivots, and good pivots come on average once in two tries. Each element takes part in $O(\log n)$ partitionings on average; the total work is $O(n \log n)$.

The probability of the worst case. For the worst case with a random pivot, the smallest or largest element of the subarray must be chosen in every partitioning. In a subarray of k elements the probability of this is $2/k$. Since the choices are independent, the probabilities are multiplied:

> $$P(\text{worst case}) = (2/n) \cdot (2/(n-1)) \cdot \ldots \cdot (2/2) = 2^{n-1} / n!$$

For $n = 5$ this value is $16/120 \approx 0.133$; for small arrays the worst case is not rare at all. In Python, with a fixed seed (`random.Random(42)`), an array of five elements was sorted 200,000 times. The fraction of runs in which the number of comparisons reached its maximum value, 10, came out as 0.134. For $n = 10$ the probability falls to $1.4 \cdot 10^{-4}$, and for $n = 20$ to $2^{19} / 20! \approx 2 \cdot 10^{-13}$. For arrays of realistic size the worst case is practically never seen.

7.5.5 Expected number of comparisons

The running time of quicksort is proportional to the number of comparisons it makes: each round of the partitioning loop makes exactly one comparison. Let this number be X. $E[X]$ is computed with indicator variables and linearity.

Let the elements be named by their sorted order: $z_1 < z_2 < \ldots < z_n$. Here z_i is the i-th smallest element of the array; it has nothing to do with its position in the array. For each pair $i < j$ define an indicator: $X_{ij} = 1$ if z_i and z_j are compared during the algorithm, 0 if not. Two elements can be compared at most once, because comparisons are made only with the pivot, and after that partitioning the pivot never enters any subarray again. Therefore $X = \Sigma_{i<j} X_{ij}$.

> **Theorem 7.3 (Probability of comparison)**
>
> In quicksort with a random pivot, z_i and z_j ($i < j$) are compared with probability exactly $2 / (j - i + 1)$.

Proof. Consider the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$; this set has $j - i + 1$ elements. As long as no element of this set is chosen as pivot, all the elements of the set stay in the same subarray, because every pivot chosen from outside is either smaller than all of them or larger than all of them. The moment the first pivot is chosen from the set is decisive. If this pivot is z_i or z_j, the other element is in the same subarray at that moment and is compared with the pivot. If the pivot is one of the elements in between, z_i falls to the left and z_j to the right; the two stay in separate parts and never meet. Since the

pivot is chosen uniformly among the elements of the subarray, the first element chosen from the set is equally likely to be any element of the set. The probability of being one of the two wanted elements is $2 / (j - i + 1)$. □

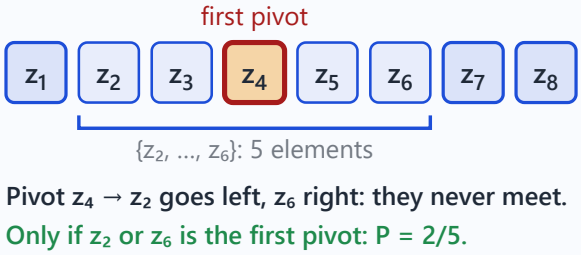


Figure 7.9: If the first pivot from the set $\{z_2, \ldots, z_6\}$ is z_4, then z_2 falls to the left and z_6 to the right, and they are never compared. They are compared only if z_2 or z_6 is the first pivot: probability 2/5.

The two extreme cases of the theorem are interesting. Two elements adjacent in sorted order, that is, $j = i + 1$, are compared with probability $2/2 = 1$, that is, always. This is natural: there is no other element between them, and no pivot can be found that separates them. The smallest and the largest element, on the other hand, meet only if the first pivot is one of the two: probability $2/n$. Random arrays of eight elements were sorted 100,000 times in Python (seed 42). The fraction of runs in which the pair z_2–z_6 was compared was 0.399 (expected 0.4), for the pair z_1–z_8 0.251 (expected 0.25), and for the adjacent pair z_3–z_4 exactly 1.

By linearity, the expected number of comparisons is the sum of the probabilities of the indicators:

> $$E[X] = \Sigma_{i<j}\, 2 / (j - i + 1)$$

The sum is brought to closed form by grouping by the distance $k = j - i + 1$. The number of pairs with distance k is $n - k + 1$ ($2 \le k \le n$). Thus

> $$E[X] = \Sigma_{k=2..n}\, (n - k + 1) \cdot 2/k = 2(n + 1)(H_n - 1) - 2(n - 1) = 2(n + 1)H_n - 4n$$

is obtained. Here $H_n = 1 + 1/2 + \ldots + 1/n$ is the n-th harmonic number and is about $\ln n + 0.577$. A crude upper bound is also easy to see: for each i the inner sum is less than $2(1/2 + 1/3 + \ldots)$, hence $E[X] < 2n\, H_n$. Since $2n \ln n = 2 \ln 2 \cdot n \log_2 n$, this bound is about $1.39\, n \log_2 n$. Quicksort with a random pivot makes only about 39% more comparisons than the $n \log_2 n$ comparisons of perfectly balanced splitting. That the closed formula equals the sum itself for $n \le 30$ in exact rational arithmetic was checked with Python.

n	$2(n + 1)H_n - 4n$	simulation average	sorted input, first element as pivot
10	24.4	24.4 (20,000 sorts)	45
100	647.9	648.0 (20,000)	4,950
1,000	10,985.9	11,003.6 (2,000)	499,500

n	$2(n+1)H_n - 4n$	simulation average	sorted input, first element as pivot
10,000	155,771.7	155,710.7 (200)	49,995,000
10^6	$\approx 2.48 \cdot 10^7$	—	$\approx 5.0 \cdot 10^{11}$

Table 7.13: The expected number of comparisons in quicksort with a random pivot (formula and simulation with `new Random(42)`) and the number of comparisons of the version that takes the first element as pivot on sorted input.

In the simulation a sorted array (1, 2, ..., n) was sorted each time. This is the worst input for the deterministic first-element pivot; for a random pivot it is an ordinary input. The averages are very close to the formula; the small differences are random fluctuation from a limited number of repetitions. The last column is $n(n-1)/2$: in a sorted array the first element is always the smallest, and at each level the part shrinks by just one. At $n = 1000$ the difference is about a factor of 45, and at $n = 10^6$ about a factor of 20,000.

Common mistake

Confusing average case with expected time. The average case makes an assumption about the input distribution; if the assumption fails, the guarantee fails. The expected time, on the other hand, is computed for each fixed input over the algorithm's coin flips. The value 24.4 in Table 7.13 holds for every input of 10 elements, including a sorted array.

7.5.6 Memory and repeated keys

Quicksort sorts the array in place; the extra memory is only the recursion stack. The stack depth is the height of the recursion tree. With a random pivot the expected depth is $O(\log n)$, but in the worst case it can reach n. A simple technique prevents this. Recurse first into the smaller part and process the larger part with a loop. Since the smaller part is at most half the size, the size at least halves with every recursive call. The stack depth stays $O(\log n)$ even in the worst case.

If the array has many equal keys, Lomuto partitioning behaves badly. In an array of all equal elements every element satisfies the condition "≤ pivot", the pivot always settles at the end, and one of the parts always stays empty. Even with a random pivot the time is $\Theta(n^2)$. The solution is **three-way partitioning**: the array is divided into three regions, the elements smaller than the pivot, those equal to the pivot, and those larger than the pivot. The region of equal elements is in its final place and does not enter the recursion.

7.6 Randomized selection (QuickSelect)

7.6.1 The problem and the code

The **selection problem** is to find the *k*-th smallest element of an array. Questions such as the median, or the threshold of the ten smallest or largest elements, are examples of this problem. Sorting the array and reading position *k* takes *O(n log n)* time. But most of the sorting is unnecessary: only the place of a single element is asked for. The **randomized selection** (quickselect) algorithm uses the partitioning of quicksort, but each time descends only into the side where the sought element lies.

```
static int select(int[] a, int lo, int hi, int k) {
  if (lo == hi) return a[lo];
  int p = partition(a, lo, hi);
  if (k == p) return a[p];
  if (k < p) return select(a, lo, p - 1, k);
  return select(a, p + 1, hi, k);
}
```

k is a 0-based rank: *k = 0* is the smallest element and *k = n − 1* the largest. The method first checks the base case. If a single element is left in the range (*lo = hi*), it is the one sought. Otherwise the same randomized `partition` is called and the final position *p* of the pivot is learned. The final position of the pivot is its position in the sorted array. If *k = p*, the pivot is the one sought. If *k < p*, the sought element is to the left of the pivot; if *k > p*, it is to the right. Whichever side is descended into, *k* does not change, because the indices are relative to the whole array. The other side is never examined again.

This algorithm too is a Las Vegas algorithm. The result is always correct; what is random is only how many partitionings will be made. It also has a side effect: the method modifies the array in place. After it returns, `a[k]` is the sought element; those to its left are less than or equal to it and those to its right are greater.

7.6.2 Code trace

Example 7.8

Let `select(a, 0, 7, 5)` be called on the array *a = [6, 5, 3, 1, 8, 7, 2, 4]*; the element sought is the 0-based 5th, that is, the 6th smallest. Let `partition` choose 3, 7, and 5 as pivots in turn. Table 7.14 gives the calls.

call	pivot	array after partition	p	decision
select(a, 0, 7, 5)	3	[1, 2, 3, 6, 8, 7, 5, 4]	2	k = 5 > 2: go right, [3, 7]

call	pivot	array after partition	p	decision
select(a, 3, 7, 5)	7	[1, 2, 3, 6, 4, 5, 7, 8]	6	k = 5 < 6: go left, [3, 5]
select(a, 3, 5, 5)	5	[1, 2, 3, 4, 5, 6, 7, 8]	4	k = 5 > 4: go right, [5, 5]
select(a, 5, 5, 5)	—	unchanged	—	lo = hi = 5: `return a[5]` = 6

Table 7.14: Trace of `select(a, 0, 7, 5)`.

In the first call the pivot 3 settles at index 2 by Lomuto partitioning. The position sought, 5, is to the right of the pivot; so only the range [3, 7] is descended into. The two elements on the left (1 and 2) are never examined again. In the second call the pivot is 7; since there are three elements smaller than 7 in the range 3..7 (6, 5, 4), 7 goes to index 6. The position sought is now to the left of the pivot. In the third call, in the three-element range [6, 4, 5], the pivot 5 is chosen and settles at index 4. In the last call a single element is left in the range: *lo = hi = 5*. The method returns `a[5] = 6`, and this value comes back unchanged along the chain of calls.



Figure 7.10: Top, the first call: the active range is [0, 7] and the position sought, 5, is marked. Bottom, the result: the element at position 5 is 6. In this example the array ended up completely sorted; this is a coincidence, since the algorithm partitions only the parts it needs.

The code, run in Java with a scripted generator that made `partition` choose the same pivots, returned 6; the intermediate arrays also matched those in the table. In addition, on 2000 random arrays and random values of *k*, the result of `select` was compared with the *k*-th element of the sorted array; all were equal.

7.6.3 Analysis

The expected running time of randomized selection is *O(n)*. It differs from quicksort in that the recursion descends into one branch instead of two. A good pivot, that is, one that falls in the middle half, reduces the range to at most *3/4* of its size and occurs with probability 1/2. On average, once in

two partitionings the range shrinks by at least a quarter. The range size falls from n to $3n/4$, then to $9n/16$. The expected cost of each stage is at most two partitionings, that is, twice the size at that stage. The total expected work is bounded by a geometric series:

> $E[T(n)] \le 2n\,(1 + 3/4 + (3/4)^2 + \ldots) = 2n \cdot 4 = 8n = O(n)$

This is a crude bound. The exact value of the expected number of comparisons when looking for the median is about $2(1 + \ln 2)\,n \approx 3.39\,n$. The median was searched for 2000 times in Java (seed 42): for $n = 1001$ the average came out as $3.33\,n$ comparisons and for $n = 10{,}001$ as $3.39\,n$. The difference at small n comes from lower-order terms in the formula.

The worst case is again $O(n^2)$. If the pivot falls at one end of the range every time, the range shrinks by only one at each step. Since partitioning already takes $\Theta(n)$, a time like $O(\log n)$ is impossible: every element must be read at least once. If linear time is wanted in the worst case as well, the pivot is chosen by the BFPRT method, that is, the "median of medians". In practice, however, the randomized version is faster.

Common mistake

Thinking the expected time of selection is $O(n \log n)$. Because it uses the same partitioning, one tends to expect the same time as quicksort. But selection continues with only one part at each level, and the work decreases geometrically: $n + 3n/4 + 9n/16 + \ldots$ The total is linear.

7.7 Estimating π by Monte Carlo

7.7.1 The method and the code

The Monte Carlo method estimates a number from the proportion of random samples. Its best-known example is the estimation of π. Consider the unit square with side 1 and a quarter circle of radius 1 centered at the lower-left corner of the square. The area of the square is 1 and the area of the quarter circle is $\pi/4$. If a uniformly random point is thrown into the square, the probability that it falls inside the quarter circle is the ratio of the areas: $p = \pi/4 \approx 0.785$.

A point (x, y) is inside the quarter circle exactly when its distance from the origin is at most 1: $x^2 + y^2 \le 1$. There is no need to take a square root. n points are thrown and those falling inside are counted. The fraction of inside points estimates $\pi/4$; therefore

> $\pi \approx 4 \cdot inside\,/\,n$

The method is a Monte Carlo algorithm with two-sided error. Its running time is exactly n steps, but its result can fall both below and above the true value.

```
static double estimatePi(int n) {
  int inside = 0;
  for (int t = 0; t < n; t++) {
    double x = rng.nextDouble(), y = rng.nextDouble();
    if (x * x + y * y <= 1) inside++;
  }
  return 4.0 * inside / n;
}
```

In each round `rng.nextDouble()` is called twice and produces two numbers uniformly distributed in *[0, 1)*. If the point is in the quarter circle, `inside` increases by one. After the loop `4.0 * inside / n` is returned. The `4.0` here is important: it moves the expression into `double` arithmetic. If `4 * inside / n` were written, integer division would be performed and the fractional part dropped; for example, if 4 of 6 points are inside, the result would be 2 instead of 2.667. The method runs in *O(n)* time and *O(1)* extra memory.

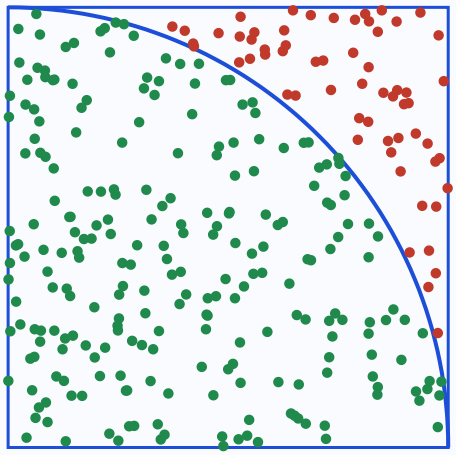


Figure 7.11: 300 points thrown into the unit square. 239 of them are inside the quarter circle (green): $\pi \approx 4 \cdot 239 / 300 \approx 3.187$.

7.7.2 Code trace

Example 7.9

Let `estimatePi(8)` be called with `rng = new Random(2024)`. Because the seed is fixed, the run produces the same points every time. Table 7.15 gives all eight points; the coordinates are rounded to four digits.

t	x	y	$x^2 + y^2$	result	inside
0	0.6185	0.1883	0.4179	inside	1
1	0.9310	0.0942	0.8756	inside	2
2	0.5821	0.4941	0.5830	inside	3
3	0.0008	0.4948	0.2448	inside	4
4	0.3028	0.7075	0.5921	inside	5
5	0.7837	0.7167	1.1279	outside	5
6	0.3506	0.9796	1.0825	outside	5
7	0.4963	0.3621	0.3774	inside	6

Table 7.15: `estimatePi(8)`, seed 2024.

Six of the eight points landed inside. The method returns *4.0 · 6 / 8 = 3.0*. The estimate is about 0.14 away from the true value. With so few points this is not surprising: the standard deviation of the estimate is about $1.64 / \sqrt{8} \approx 0.58$. By this measure the estimate 3.0 is quite good. With the same seed the same code always gives 3.0; if the seed is changed, a different result comes out.

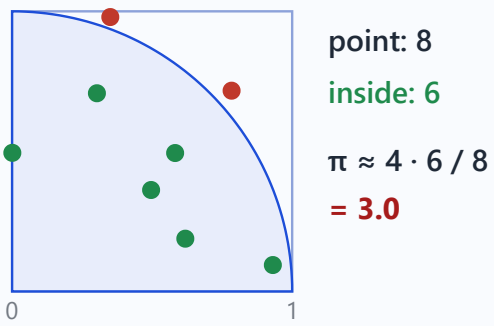


Figure 7.12: When `estimatePi(8)` finishes: 8 points, 6 inside (green) and 2 outside (red). The estimate is *4 · 6 / 8 = 3.0*.

7.7.3 Error and the number of points

How reliable the estimate is can be found by a simple probability computation. Each point is an independent indicator that takes the value 1 with probability $p = \pi/4$ and 0 otherwise. The expected value of `inside`, the sum of n indicators, is np and its variance is $np(1 - p)$. Since the estimate is $4 \cdot inside / n$, its expected value is exactly $4p = \pi$; the method gives the correct value on average. Its standard deviation is

$$\sigma = 4 \cdot \sqrt{(p(1 - p) / n)} = 1.642 / \sqrt{n}$$

The error decreases in inverse proportion to the square root of the number of points. The practical meaning of this is heavy: to reduce the error to a tenth, the number of points must grow a hundredfold. One more decimal digit of accuracy requires 100 times more work.

n	estimate	|estimate − π|	σ = 1.642/√n
100	3.16000	0.0184	0.164
1,000	3.10400	0.0376	0.0519
10,000	3.12240	0.0192	0.0164
100,000	3.14128	0.00031	0.0052
1,000,000	3.13863	0.00296	0.0016
10,000,000	3.14066	0.00093	0.00052

Table 7.16: Estimating π with different numbers of points. Each row was started from scratch with `rng = new Random(42)`.

The table shows that individual runs are noisy. The estimate with 100 points happened to come out very good; the estimate with 1,000,000 points is worse than that with 100,000. This is not a contradiction. σ gives not the error of a single run but the typical size of errors. To test this, the estimate with 1,000 points was repeated 2000 times (seed 42). The root mean square of the errors came out as 0.0514; the formula gives 0.0519. With the normal approximation, the error stays below $1.96\,\sigma$ in about 95% of runs. For the error to be below 0.01 with 95% confidence, about 104,000 points are needed; to be below 0.001, about 10.4 million points.

Example 7.10

In an estimate made with 10,000 points the error is about 0.016. How many points are needed to bring the error down to about 0.0016? Since the error decreases as $1/\sqrt{n}$, to reduce it to a tenth, $\sqrt{n}$ must grow tenfold and n a hundredfold: 1,000,000 points. Raising the number of points to 100,000 reduces the error only by a factor of $\sqrt{10} \approx 3.16$.

Estimating π is not a good way to compute π; there are much faster deterministic formulas. But the method itself is very valuable. Complicated shapes whose area or volume cannot be computed by a formula, multidimensional integrals, particle simulations in physics, and risk calculations in finance are all done with the same idea. The error of the Monte Carlo method decreases as $1/\sqrt{n}$ regardless of the number of dimensions. In high-dimensional problems this property makes it superior to grid-based methods.

Common mistake

Integer division. In the expression `4 * inside / n` all three operands are `int`, so the result is truncated to an integer. If four of six points are inside, the result is 2 instead of 2.667. Starting with `4.0` moves the division to `double`.

7.8 The Miller–Rabin primality test

7.8.1 The Fermat test and Carmichael numbers

The simplest way to tell whether a number is prime is to try dividing by all numbers up to $\sqrt{n}$. This method makes $\sqrt{n}$ divisions. In terms of the number of digits of the number this is exponential time: for a number of 300 digits, 10^{150} divisions are needed. Cryptography needs primes of exactly this size. A **primality test** checks the primality of a number without factoring it.

The idea comes from number theory. By **Fermat's little theorem**, if n is prime and $1 < a < n$, then

> $a^{n-1} \equiv 1 \pmod{n}$

holds. If this equality fails for some a, then n is certainly composite. This means that compositeness is proved without finding factors. If the equality holds, n is "maybe prime". The problem is this: some composite numbers satisfy the equality for every base coprime to themselves. These are called **Carmichael numbers**. The smallest is $561 = 3 \cdot 11 \cdot 17$. For example $2^{560} \bmod 561 = 1$. All bases coprime to 561, that is, all 320 numbers between 1 and 560, pass the Fermat test. The Fermat test can catch these numbers only by hitting a base that shares a factor. There are infinitely many Carmichael numbers.

7.8.2 The square-root property and the Miller–Rabin sequence

The Miller–Rabin test adds a second check to the Fermat test. This check relies on the square roots of 1. If n is prime, the congruence $x^2 \equiv 1 \pmod{n}$ has only two solutions: $x \equiv 1$ and $x \equiv -1 \equiv n - 1$. The reason is short. If the product $x^2 - 1 = (x - 1)(x + 1)$ is divisible by n and n is prime, then n divides one of the factors. For composite numbers, however, 1 can have other square roots. In 561, for example, $67^2 = 4489 = 8 \cdot 561 + 1$; 67 is a "hidden" square root of 1.

The test is set up as follows. Let n be an odd number. $n - 1$ is even and is written as $n - 1 = 2^s \cdot d$, where d is odd. For a random base a the following sequence is computed:

> $a^d,\ a^{2d},\ a^{4d},\ \ldots,\ a^{2^{s-1}d} \pmod{n}$

Each element is the square of the previous one. The next square after the sequence is $a^{2^s d} = a^{n-1}$. If n is prime, by Fermat this last value is 1. Looking back along the sequence, the element before any square that reaches 1 is a square root of 1; in a prime modulus this element must be either 1 or $n - 1$. Consequently, if n is prime, the sequence either starts with 1 or takes the value $n - 1$ somewhere.

Definition 7.4

Let an odd number n and $n - 1 = 2^s \cdot d$ (d odd) be given. If $a^d \bmod n \neq 1$ and, for $0 \le r < s$, no value $a^{2^r d}$ is congruent to $n - 1$, then a is a **witness** to the compositeness of n. If n is composite but a is not a witness, a is called a **strong liar** for n.

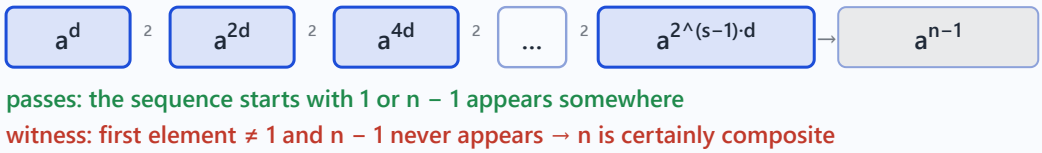


Figure 7.13: The Miller–Rabin sequence: successive squares starting from a^d. If the sequence starts with 1 or $n - 1$ appears somewhere, n passes the round. If the first element is not 1 and $n - 1$ never appears, a is a witness and n is certainly composite.

The power of the test comes from a number-theoretic result. Rabin showed that for an odd composite n, at most a quarter of the bases between 1 and $n - 1$ are strong liars. The proof of this theorem is not given here, but its consequence is easily tested on small numbers. For all odd composite numbers below 3000, the bases between 2 and $n - 2$ were tried one by one in Python. The highest liar fraction came out as $14/59 \approx 0.237$ for $n = 1891 = 31 \cdot 61$; it is below $1/4$. If the base is chosen at random, the probability that a composite number passes a round is at most $1/4$. In k independent rounds the probability of error is at most 4^{-k}.

7.8.3 Code

```
static boolean isPrime(long n, int k) {
  if (n < 4) return n == 2 || n == 3;
  if (n % 2 == 0) return false;
  long d = n - 1; int s = 0;     // n-1 = 2^s·d
  while (d % 2 == 0) { d /= 2; s++; }
  for (int round = 0; round < k; round++) {
    long a = 2 + rng.nextLong(n - 3); // [2, n-2]
    long x = powMod(a, d, n);     // a^d mod n
    if (x == 1 || x == n - 1) continue;
    boolean witness = true;
    for (int r = 1; r < s && witness; r++) {
      x = x * x % n;
      if (x == n - 1) witness = false;
    }
    if (witness) return false;     // certainly composite
  }
  return true;                  // probably prime
}
```

The first two lines filter out small cases. If $n < 4$, only 2 and 3 are prime. Even numbers other than 2 are composite; 2 has already been caught in the first line. The next two lines bring $n - 1$ to the form $2^s \cdot d$: while d is even it is halved and s is counted. The main loop makes k rounds. In each round the expression `2 + rng.nextLong(n - 3)` picks a uniform base from the range $[2, n - 2]$; `nextLong(n - 3)` gives a value between 0 and $n - 4$. (The bases 1 and $n - 1$ are useless because they pass the test for every number.)

`x = powMod(a, d, n)` is the first element of the sequence. If x is 1 or $n - 1$, this round is passed and `continue` goes to the next round. Otherwise `witness` is first set to `true`; the inner loop computes the remaining $s - 1$ elements of the sequence by successive squaring. If $n - 1$ appears, `witness = false` and the inner loop stops. If `witness` is still `true` when the inner loop ends, a is a witness and the method returns `false`. This answer is certain. If all rounds are passed, the method returns `true`: the number is "probably prime".

The code contains two details. First, the method `rng.nextLong(long bound)` was added in Java 17; it does not exist in older versions. Second, the product `x * x % n` is computed in the type `long`. Since $x < n$, the product is less than n^2. The largest value of `long` is about $9.22 \cdot 10^{18}$; its square root is 3,037,000,499. The code is therefore safe only for $n < 3 \cdot 10^9$. For larger numbers `BigInteger.modPow` or directly `BigInteger.isProbablePrime` is used.

7.8.4 Fast modular exponentiation

Computing $a^d \bmod n$ by multiplying a by itself $d - 1$ times is impossible for large d. The **square-and-multiply** method uses the binary representation of the exponent. The exponent is written in binary; at each step the base is squared to reach the next power of two. If that bit of the exponent is 1, that power is included in the result.

```
static long powMod(long a, long d, long n) {
  long result = 1;
  a %= n;
  while (d > 0) {
    if ((d & 1) == 1)          // if the bit is 1
      result = result * a % n;
    a = a * a % n;             // square
    d >>= 1;                   // next bit
  }
  return result;
}
```

The loop invariant is this: at the start of each round $result \cdot a^d$ is equal to the value sought (mod n). If the bit is 1, a is multiplied into the result and that bit of d is consumed. Then a is squared and d is shifted right by one bit, that is, halved. When $d = 0$, the invariant says that *result* is the value sought. The loop runs as many times as d has bits, that is, $\lfloor \log_2 d \rfloor + 1$ times. Each round makes at most two modular multiplications: the time is $O(\log d)$ multiplications.

round	d	rightmost bit	result	a (after squaring)	note
start	35	—	1	2	$a = 2^1$
1	35	1	$1 \cdot 2 = 2$	4	2^1 included
2	17	1	$2 \cdot 4 = 8$	16	2^2 included
3	8	0	8	256	square only
4	4	0	8	65,536 mod 561 = 460	square only
5	2	0	8	211,600 mod 561 = 103	square only
6	1	1	$8 \cdot 103 = 824 \bmod 561 = 263$	511	2^{32} included

Table 7.17: Trace of `powMod(2, 35, 561)`. $35 = 100011_2 = 32 + 2 + 1$.

The loop ends when $d = 0$ and returns 263: $2^{35} \bmod 561 = 263$. In the table the column a carries, in turn, the values $2^1, 2^2, 2^4, 2^8, 2^{16}, 2^{32}$ (mod 561). In the rounds where the bit is 1, 2^1, 2^2, and 2^{32} are included in the result and $2^1 \cdot 2^2 \cdot 2^{32} = 2^{35}$ is obtained. In total 6 squarings and 3 multiplications were made; plain multiplication would have needed 34 multiplications. The difference looks insignificant for small exponents. For an exponent of 300 digits, however, about 1000 squarings suffice, while the plain method needs about 10^{300} multiplications. The value $a = 511$ in the last round is computed but not used.

7.8.5 Trace for the number 561

Example 7.11

Let `isPrime(561, 1)` be called and let the random base come out as $a = 2$. 561 is odd and greater than 4; the small-case checks are skipped. Table 7.18 gives the remaining steps.

step	operation	values
1	$d = n - 1, s = 0$	d = 560, s = 0
2–5	while d is even, $d /= 2, s{+}{+}$	280/1, 140/2, 70/3, 35/4 → $560 = 2^4 \cdot 35$
6	round 0: the base is drawn	a = 2
7	`x = powMod(2, 35, 561)`	x = 263; neither 1 nor 560 → `witness = true`
8	r = 1: $x = 263^2 \bmod 561$	69,169 mod 561 = 166; not 560
9	r = 2: $x = 166^2 \bmod 561$	27,556 mod 561 = 67; not 560
10	r = 3: $x = 67^2 \bmod 561$	4,489 mod 561 = 1; not 560
11	the inner loop ends (r = 4 = s)	`witness` is still true → `return false`

Table 7.18: `isPrime(561, 1)`, base $a = 2$.

The sequence is 263, 166, 67, 1 and never reaches 560. The method returns `false`: 561 is certainly composite. The last step of the sequence is especially instructive. $67^2 \equiv 1$, but 67 is neither 1 nor 560. In a prime modulus there cannot be such a square root; so 67 is concrete evidence of compositeness. One square further along the sequence is $2^{560} \equiv 1$. The Fermat test is fooled because it looks only at this last value. Miller–Rabin looks at how 1 was reached and is not fooled.

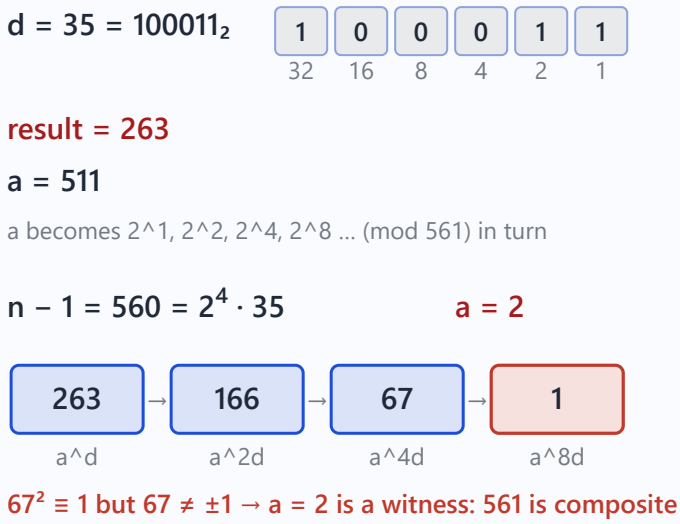


Figure 7.14: Left, `powMod(2, 35, 561)` finished: $35 = 100011_2$, result 263. Right, the Miller–Rabin sequence for 561: 263 → 166 → 67 → 1; $67^2 \equiv 1$ but 67 is neither 1 nor −1; therefore 2 is a witness and 561 is composite.

7.8.6 Examples and strong liars

n	n − 1	a	a^d, a^{2d}, … (mod n)	result
61	$2^2 \cdot 15$	2	11, 60	passed (may be prime)
221	$2^2 \cdot 55$	174	47, 220	passed (may be prime)
221	$2^2 \cdot 55$	137	188, 205	composite (a is a witness)
561	$2^4 \cdot 35$	2	263, 166, 67, 1	composite (a is a witness)
25	$2^3 \cdot 3$	7	18, 24	passed (may be prime)
25	$2^3 \cdot 3$	2	8, 14, 21	composite (a is a witness)

Table 7.19: Miller–Rabin examples. Each row is a single round.

61 is prime. $2^{15} = 32{,}768 = 537 \cdot 61 + 11$, so the sequence starts with 11. Its square is $121 = 61 + 60$, that is, 60, which is $n - 1$. Since the sequence reaches $n - 1$, the round is passed. A prime number passes every round; Miller–Rabin never says "composite" for a prime.

221 = 13 · 17 is composite. For $a = 174$ the sequence starts with 47; $47^2 = 2209 = 9 \cdot 221 + 220$, that is, 220, which is $n - 1$. 174 is a strong liar: it makes the composite 221 look prime in this round. For $a = 137$ the sequence is 188 and 205; 220 does not appear and 137 becomes a witness. If `isPrime(221,`

`2)` is called with these two bases, the first round is passed and the second returns `false`. If one round were enough and only 174 were drawn, the answer would wrongly be "prime". This is why several rounds are needed. Of the 218 bases of 221 between 2 and 219, only four (21, 47, 174, 200) are strong liars. The error probability of a single round is $4/218 \approx 0.018$. In Python, 100,000 random bases were tried with a fixed seed; the fraction came out as 0.0185.

For $\mathbf{25 = 5^2}$, $n - 1 = 24 = 2^3 \cdot 3$. With $a = 7$, $7^3 = 343 \equiv 18$ and $18^2 = 324 \equiv 24 = n - 1$; 7 is a strong liar for 25. Of the 22 bases of 25, only two (7 and 18) are liars. With $a = 2$ the sequence is 8, 14, 21; 24 never appears and the number is found composite.

These examples show that Rabin's bound is very pessimistic for most numbers. Of the 558 bases of 561, only 8 are strong liars, and of the 218 bases of 221, 4. Even the numbers that come closest to the bound do not exceed 1/4. So the true error probability is often even smaller than 4^{-k}; 4^{-k} is a guarantee for the worst case.

7.8.7 Error, cost, and overflow

Miller–Rabin is a Monte Carlo algorithm with one-sided error. The answer "composite" is certain, because a witness has been found. The answer "prime" is only probably correct, and in k rounds the probability of error is at most 4^{-k}. For a prime number the answer is always correct. The error can occur only in the form of calling a composite number "prime".

A round performs one modular exponentiation and at most $s - 1$ squarings. Together these are $O(\log n)$ modular multiplications. For numbers of b bits, one multiplication by the schoolbook method takes $O(b^2)$ bit operations. Since $b = \log n$, one round is $O(\log^3 n)$ and k rounds are $O(k \log^3 n)$ bit operations. This is polynomial in the number of digits of the number. Compared with the exponential time of trial division, the gap is enormous.

The code was tested in Java. With a generator seeded `new Random(42)`, `isPrime(n, 20)` was compared with `BigInteger.isProbablePrime` for every number between 2 and 200,000; no wrong answer appeared. In the same range with a single round ($k = 1$), 46 composite numbers were wrongly found "prime". The numbers 1,000,000,007, $2^{31} - 1 = 2{,}147{,}483{,}647$, and 2,999,999,929 were found prime with 20 rounds; `BigInteger` gives the same result.

Common mistake

Overflow in modular multiplication. If `x * x % n` is written with `int`, the product exceeds $2^{31} - 1$ when x reaches 46,341: $46{,}341^2 = 2{,}147{,}488{,}281$. So for $n > 46{,}341$ the result silently becomes wrong. With `long` the limit is $n < 3.037 \cdot 10^9$. For larger numbers, a 128-bit product via `Math.multiplyHigh` or `BigInteger.modPow` is used.

Common mistake

Treating a Monte Carlo answer as certain. The answer "prime" from Miller–Rabin means "probably prime". The number of rounds k determines the error bound and is chosen according to the needs of the application. If a certain answer is needed, a deterministic test is used (for example AKS, or fixed sets of bases proven for specific ranges of numbers).

7.9 Karger's minimum cut algorithm

7.9.1 The minimum cut problem

Consider an undirected, connected graph. Let the vertices of the graph be split into two nonempty groups. The edges remaining between the two groups, that is, the edges with one endpoint in one group and the other endpoint in the other group, form a **cut**. If these edges are deleted, the graph falls into two pieces. The **minimum cut** problem is to find the cut with the fewest edges. This number is the smallest number of edges that must be deleted to disconnect the graph; it is a measure of network reliability. It is used in computer networks, circuit design, and segmenting images into regions.

Definition 7.5

Let $G = (V, E)$ be an undirected multigraph. A partition of V into two nonempty, complementary subsets is called a **cut** $(S, V - S)$. The size of the cut is the number of edges with one endpoint in S and the other in $V - S$. A cut of smallest size is called a **minimum cut** (global minimum cut).

This problem differs from the s–t cut in network flow. There two special vertices are given and only cuts separating them are sought. Here, in contrast, which vertices are separated is free. It is possible to solve it deterministically: fix one vertex and solve an s–t maximum flow problem with each of the other vertices. But this means $n - 1$ flow computations and code that is both long and complicated. The algorithm Karger gave in 1993, on the other hand, consists of a loop of a few lines.

7.9.2 Edge contraction

The only operation of the algorithm is **edge contraction**. When an edge (u, v) is contracted, u and v merge into a single "supervertex". The edges between them now become self-loops from a vertex to itself and are deleted. Edges going from u or v to other vertices are attached to the supervertex. If several edges arise between the same two supervertices, all are kept; the graph becomes a multigraph. These parallel edges matter, because each is counted as a separate edge in the cut.

The algorithm is as follows: until two supervertices remain in the graph, a uniformly random edge is chosen among the remaining edges and contracted. The two supervertices that remain in the end give the partition of the vertices into two groups; the number of edges left between them is a cut. The algorithm always makes exactly $n - 2$ contractions. Its result is sometimes not minimum. That is why it is a Monte Carlo algorithm.

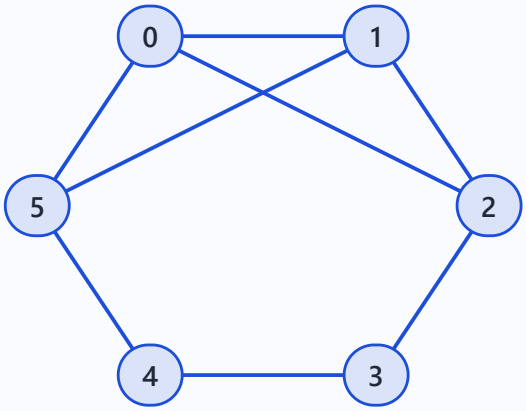
Example 7.12

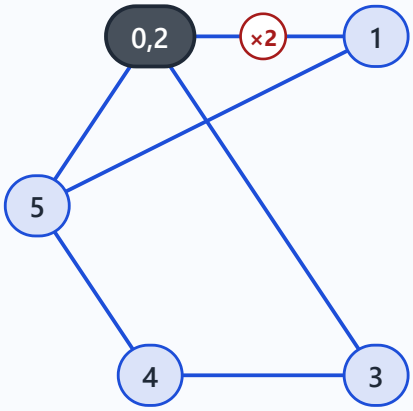
Consider a graph with six vertices and eight edges. The edges are (0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (0, 5), (0, 2), and (1, 5). The minimum cut is 2. Vertices 3 and 4 are connected to the graph only by the edges (2, 3) and (4, 5); the set {3, 4} is separated by these two edges. {3} alone or {4} alone is also a cut with 2 edges. This graph has three minimum cuts. Let the algorithm choose the edges (0, 2), (1, 5), (3, 4), and (0, 1) in turn. Table 7.20 gives the contractions.

step	edge contracted	supervertices	self-loops deleted	edges remaining
0	—	{0} {1} {2} {3} {4} {5}	—	8
1	(0, 2)	{0, 2} {1} {3} {4} {5}	(0, 2)	7
2	(1, 5)	{0, 2} {1, 5} {3} {4}	(1, 5)	6
3	(3, 4)	{0, 2} {1, 5} {3, 4}	(3, 4)	5
4	(0, 1)	{0, 1, 2, 5} {3, 4}	(0, 1), (1, 2), (0, 5)	2

Table 7.20: A successful Karger run. The remaining edges are counted including parallel edges.

After the first contraction, two parallel edges arise between the supervertex {0, 2} and 1: the old (0, 1) and (1, 2). After the third contraction there are three parallel edges between {0, 2} and {1, 5}: (0, 1), (1, 2), and (0, 5). In the fourth step one of them, (0, 1), is chosen. All three become self-loops at once and are deleted. Two supervertices remain: {0, 1, 2, 5} and {3, 4}. Between them are the edges (2, 3) and (4, 5). The cut found is 2; this run is successful.


0
1
5
2
4
3


0,2
×2
1
5
4
3

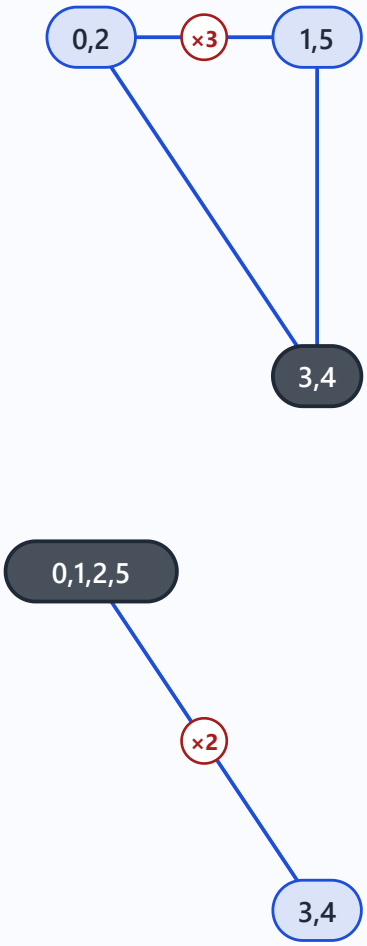


Figure 7.15: A Karger run. Top left, the starting graph (6 vertices, 8 edges). Top right, (0, 2) has been contracted; two parallel edges (×2) between {0, 2} and 1. Bottom left, three supervertices and 5 edges after three contractions. Bottom right, two supervertices remain; 2 edges between them: the minimum cut was found. The dark vertex is the supervertex formed in the last contraction.

Another sequence of random choices gives another result. Let the algorithm choose the edges (1, 2), then (4, 5), (3, 4), and (0, 1). This time the supervertices are {0, 1, 2} and {3, 4, 5}. The edges (2, 3), (0, 5), and (1, 5) remain between them: the cut is 3 and is not minimum. The error arose from contracting (4, 5), an edge of a minimum cut. When an edge of a minimum cut is contracted, the two sides of that cut merge and this cut can no longer be found.

7.9.3 Code

The code for this algorithm is not given in the slides. The following short version shows the whole algorithm on an edge list. The supervertices are tracked with a `group` array: each vertex is linked to a "representative" vertex, and the method `find` follows these links to find the representative. If two vertices have the same representative, they are in the same supervertex.

```
static int kargerCut(int n, int[][] edges) {
  int[] group = new int[n];             // group[v]: vertex v links to
  for (int v = 0; v < n; v++) group[v] = v;
  int remaining = n;                    // number of supervertices
  while (remaining > 2) {
    int[] e = edges[rng.nextInt(edges.length)];
    int g1 = find(group, e[0]), g2 = find(group, e[1]);
    if (g1 == g2) continue;             // self-loop: draw again
    group[g1] = g2;                     // contract the edge
    remaining--;
  }
  int cut = 0;
  for (int[] e : edges)
    if (find(group, e[0]) != find(group, e[1])) cut++;
  return cut;
}
static int find(int[] group, int v) {
  while (group[v] != v) v = group[v];
  return v;
}
```

The loop draws a uniformly random edge from the edge list. If both endpoints of the edge are already in the same supervertex, this edge is a self-loop; it is skipped and drawn again. Otherwise the two groups are merged and the number of supervertices decreases by one. This "redraw" method gives the correct distribution. Every edge that is not a self-loop is chosen with equal probability, and parallel edges between two supervertices are counted separately. This is the same as the choice in the definition of the algorithm. When two supervertices remain, the edge list is scanned once and the edges whose endpoints lie in different groups are counted.

```
static int kargerRepeat(int n, int[][] edges, int T) {
  int best = Integer.MAX_VALUE;
  for (int t = 0; t < T; t++)
    best = Math.min(best, kargerCut(n, edges));
  return best;
}
```

`kargerRepeat` runs the algorithm T times independently and returns the smallest cut found. Karger's error is one-sided: the cut found can never be smaller than the minimum. So taking the smallest result is correct. The code is written to be readable, not fast. In a simple implementation a single run takes $O(n^2)$ time.

The code was tested in Java on the example graph. With a scripted generator, the order (0, 2), (1, 5), (3, 4), (0, 1) returned 2; the order (1, 2), (4, 5), (3, 4), (0, 1) returned 3. Then 200,000 independent runs were made with `rng = new Random(42)`. 91,392 runs found a cut of 2 edges, 88,725 runs a cut of 3, and 19,883 runs a cut of 4. The success rate is 0.457.

7.9.4 Success probability

Theorem 7.4 (Karger)

In a connected multigraph with n vertices, a single run of Karger's algorithm finds a particular minimum cut with probability at least $2 / (n(n - 1))$.

Proof. Let a particular minimum cut be C and let it have c edges. If the algorithm contracts no edge of C, it finds exactly C in the end. Every vertex has degree at least c; otherwise the cut separating that vertex alone would be smaller than c. Contractions preserve cuts, so in the remaining graph too every supervertex has degree at least c. When k supervertices remain, the sum of the degrees is at least kc and the number of edges at least $kc/2$. At that moment the probability that an edge of C is chosen is at most $c / (kc/2) = 2/k$. The probability that C is left untouched is the product over the steps:

$$(1 - 2/n)(1 - 2/(n - 1)) \dots (1 - 2/3) = (n - 2)/n \cdot (n - 3)/(n - 1) \cdot \dots \cdot 1/3 = 2 / (n(n - 1))$$

In the product most numerators and denominators cancel; only $n(n - 1)$ remains in the denominator and only $2 \cdot 1$ in the numerator. □

For $n = 6$ the lower bound is $2/30 = 1/15 \approx 0.067$. On the example graph the true success probability is much higher than this. All contraction sequences were computed one by one in Python with rational arithmetic: the probability of finding any minimum cut is exactly $16/35 \approx 0.457$. This agrees with the 0.457 in the simulation. There are two reasons for the difference. First, the graph has three minimum cuts and finding any one counts as success. Second, the bound "every vertex has degree at least c" in the proof is loose on this graph: the degrees are between 2 and 3.

The theorem has an interesting corollary. The events of finding different minimum cuts are mutually exclusive and each has probability at least $2/(n(n - 1))$. Since the probabilities cannot sum to more than 1, a graph can have at most $n(n - 1)/2$ distinct minimum cuts. This bound is tight: in a cycle graph with n vertices, every choice of two edges gives a minimum cut, and there are exactly $n(n - 1)/2$ minimum cuts. In the six-vertex cycle, Karger finds a minimum cut in every run, but finds a particular cut with probability only 1/15.

Example 7.13

What is the lower bound for a graph with five vertices? The product in the proof is $(1 - 2/5)(1 - 2/4)(1 - 2/3) = 3/5 \cdot 2/4 \cdot 1/3 = 1/10$; the formula also gives $2/(5 \cdot 4) = 1/10$. The true success may be higher depending on the graph.

7.9.5 Amplification by repetition

The success probability of a single run, $p \geq 2/(n(n-1))$, is small. If the algorithm is run T times independently and the smallest result is taken, the probability that all runs fail is at most $(1-p)^T$. Using the inequality $1 - x \leq e^{-x}$, we get

$$(1 - 2/(n(n-1)))^T \leq e^{-2T/(n(n-1))}$$

is obtained. If $T = n(n-1)/2 \cdot \ln n$ is chosen, the right-hand side becomes $e^{-\ln n} = 1/n$. The total time is $O(n^2 \log n)$ runs × $O(n^2) = O(n^4 \log n)$. This is the running time of a simple implementation. The recursive version developed by Karger and Stein exploits the observation that the first steps of contraction are safe and the last steps are risky, and reduces the time to $O(n^2 \log^3 n)$.

Example 7.14

On a graph with six vertices the success probability is at least $1/15$. How many repetitions suffice to push the failure probability below $1/6$? With the exponential bound, the condition $(14/15)^T \leq e^{-T/15} \leq 1/6$ gives $T \geq 15 \ln 6 \approx 26.9$; 27 repetitions suffice. Computed directly without the bound, $(14/15)^{26} \approx 0.1663$ is already below $1/6 \approx 0.1667$; 26 repetitions also suffice, 25 do not. The exponential bound makes the computation easier, at the price of one extra repetition.

These bounds are for the worst graphs. Since the true success on the example graph is 16/35, the probability that all of 27 repetitions fail is $(19/35)^{27} \approx 6.9 \cdot 10^{-8}$. In Java, `kargerRepeat(6, G, 27)` was called 20,000 times (seed 42) and returned 2 every time. With only 3 repetitions the success rate came out as 0.836; the exact value is $1 - (19/35)^3 \approx 0.840$.

Common mistake

Not choosing the edge uniformly. If a random vertex is chosen first and then a random neighbor of it, the edges are not equally likely. Edges of vertices with small degree are chosen more often. These vertices, in turn, often carry exactly the edges of the minimum cut. The proof assumes each edge is chosen with equal probability; parallel edges must also be counted separately.

7.10 Other applications

Randomness is used in many areas of algorithm design. The following examples are a few of them.

Graph coloring. Different colorings are tried with random decisions. The goal is to find good colorings with few colors without giving adjacent vertices the same color. Local improvements made from a random start help escape the bad solutions in which a deterministic order gets stuck.

Randomized depth-first search. Neighbors are visited in a random order. Results such as connectivity and spanning trees do not change, but the tree obtained is different every time. Maze generation is based on this idea: a random spanning tree of a grid is a maze with a single path from every cell to every other cell.

Matching. Randomized incremental algorithms take the edges in a random order and produce solutions approaching the maximum matching. The random order prevents a bad input order from fooling the algorithm.

Approximate counting. Data structures such as the Count-Min Sketch work with randomly chosen hash functions. In a very large data stream they estimate how many times each element occurs, with little memory and a controlled error.

Matrix operations. Randomized matrix methods quickly produce low-rank approximations of a large matrix. Dimensionality reduction and data compression use these approximations.

Computational geometry. There are randomized incremental algorithms for the convex hull, the closest pair of points, and Delaunay triangulation. The points are added in a random order and the expected time is $O(n \log n)$.

Metaheuristics such as randomized scheduling (job assignment), simulated annealing, and genetic algorithms also exploit randomness in search. These methods find good solutions in large search spaces, but they do not guarantee reaching the optimum. In this respect they differ from Las Vegas and Monte Carlo algorithms: for those, the correctness or the error probability is given by a proven bound.

7.11 Pseudorandom numbers

Throughout these notes we kept saying "a random number is chosen". But a computer is a deterministic machine; it does not produce true randomness. Programs use **pseudorandom numbers** instead. These are numbers produced by a deterministic formula that nevertheless behave like random numbers in statistical tests. The starting value of the formula is called the **seed**.

The oldest and simplest method is the **linear congruential generator** (LCG):

> $X_{k+1} = (a \cdot X_k + c) \bmod m$

Example 7.15

Let $a = 5$, $c = 3$, $m = 16$, and seed $X_0 = 7$. The sequence runs 7, 6, 1, 8, 11, 10, 5, 12, 15, 14, 9, 0, 3, 2, 13, 4 and returns to 7 after 16 steps. For example, $5 \cdot 7 + 3 = 38$ and $38 \bmod 16 = 6$. This generator produces every value in 0..15 once; the sequence looks irregular at first sight. But looking at the last bits the weakness is seen at once: the numbers come alternately odd, even, odd, even. The low bits of LCGs have a short period.

`java.util.Random` is an LCG. Its parameters are $a = 25{,}214{,}903{,}917$, $c = 11$, and $m = 2^{48}$. Because of the weakness seen in the example, the class uses only the upper bits of its 48-bit state as output. Such a generator is sufficient for algorithm experiments and simulations. But from a few consecutive outputs its internal state can be computed and all later numbers predicted.

The same seed always gives the same sequence. This is not a weakness but a valuable property. A program started with `new Random(42)` makes the same "random" decisions on every run. In this way a bug can be reproduced when debugging, experiments can be repeated, and tests give the same result every time. All the simulations in these notes were also made with fixed seeds. The eight points in the π trace are obtained again every time with `new Random(2024)`.

Different generators are used depending on the purpose. In cryptography, that is, when generating keys, passwords, or session tokens, unpredictability is essential; for this `java.security.SecureRandom` is used. In multithreaded programs, sharing a single `Random` object makes the threads wait for each other; `ThreadLocalRandom.current()` gives each thread its own generator.

expression	range of values
`rng.nextInt(n)`	0 … n − 1
`lo + rng.nextInt(hi - lo + 1)`	lo … hi
`rng.nextDouble()`	[0, 1)
`rng.nextInt(6) + 1`	a die roll: 1 … 6

Table 7.21: Commonly used random ranges (`rng` is a `Random` object).

The second row of the table is the pattern used in choosing a random pivot. The range has $hi - lo + 1$ values; this number is given to `nextInt` and the result is shifted by *lo*. If the `+ 1` is forgotten, *hi* is never chosen. This is the same bug as `nextInt(i)` in Fisher–Yates.

Common mistake

`new Random()` **on every call.** Creating a new generator wherever a random number is needed produces unnecessary objects. Worse, the seeds of generators created one after another can be close to each other and correlated. A single `Random` object is shared

throughout the program; this is why `static Random rng` in the code of these notes is at class level.

7.12 Common mistakes

Mistakes in randomized algorithms often give no compile error and do not show in a single run. The program runs and produces plausible-looking results, but the probability distribution or the analysis is wrong. The following list collects the mistakes met throughout the notes.

Common mistake

The naive shuffle. Swapping with `nextInt(n)` for every i is biased: n^n equally likely paths cannot be distributed evenly over $n!$ permutations. The correct form is `nextInt(i + 1)`. `nextInt(i)` gives Sattolo's algorithm and produces only single-cycle permutations.

Common mistake

"A random pivot eliminates the worst case." The worst case is still $O(n^2)$. What changes is that no input is consistently bad and that the probability of the bad case is very small: about $2 \cdot 10^{-13}$ at $n = 20$.

Common mistake

Confusing average case with expected time. The average case assumes a distribution of inputs. The expected time is computed for each fixed input over the algorithm's coin flips and holds for the worst input too.

Common mistake

Treating a Monte Carlo answer as certain. In Miller–Rabin the answer "prime" is only probably correct; the number of rounds k sets the error bound at 4^{-k}. A single run of Karger's algorithm often gives a cut that is not minimum; the smallest of many runs is taken.

Common mistake

Overflow. `x * x % n` with `int` can overflow for $n > 46{,}341$; with `long` the limit is about $3 \cdot 10^9$. For larger moduli `Math.multiplyHigh` or `BigInteger.modPow` is used.

Common mistake

A new generator on every call, and a non-uniform choice. Creating `new Random()` on every call is wasteful and the seeds may be correlated; a single `Random` is shared. In Karger's algorithm the edge is chosen directly and uniformly; choosing a vertex first and then a neighbor is not uniform.

Common mistake

Off by one at the range boundary. If `lo + rng.nextInt(hi - lo)` is written, *hi* is never chosen. For a random pivot this means the last element can never be the pivot; the assumption "every element with probability *1/n*" in the analysis breaks.

7.13 Summary

- A randomized algorithm uses random bits in addition to its input. The running time and sometimes the output is a random variable. The expected time is taken, for every input, over the algorithm's own choices.
- The basic tools of the analysis are linearity of expectation, indicator variables, the geometric distribution (*1/p* trials), independent repetition (q^k), and Markov's inequality.
- A Las Vegas algorithm is always correct and its running time is random. A Monte Carlo algorithm has bounded running time and its answer is wrong with a small probability. A Monte Carlo algorithm with a verifiable answer is turned into Las Vegas by repetition; Las Vegas is turned into Monte Carlo by Markov's inequality.

Topic	What to remember
Fisher–Yates	`j = nextInt(i + 1)`, $\Theta(n)$, each of the $n!$ outcomes has probability $1/n!$; the naive shuffle is biased
BogoSort	$p = 1/n!$, expected $n!$ shuffles and $\Theta(n \cdot n!)$ time; best case $O(n)$, worst case unbounded
Randomized QuickSort	$P(z_i, z_j \text{ compared}) = 2/(j - i + 1)$; expected $2(n+1)H_n - 4n \approx 1.39\, n \log_2 n$ comparisons, for every input; worst case $O(n^2)$
QuickSelect	descends into one side; expected $O(n)$ ($\approx 3.39\, n$ for the median), worst case $O(n^2)$
π estimation	$4 \cdot \text{inside} / n$; standard deviation $1.642/\sqrt{n}$; 100 times the points for one more digit
Miller–Rabin	$n - 1 = 2^s d$; the sequence must start with 1 or contain $n - 1$; error $\le 4^{-k}$; "composite" is certain; $O(k \log^3 n)$
Karger	random edge contraction; success $\ge 2/(n(n-1))$; with $n(n-1)/2 \cdot \ln n$ repetitions failure $\le 1/n$

Table 7.22: Summary of the algorithms studied in these notes.

- The error probability of an algorithm is reduced exponentially fast by independent repetition. With one-sided error a single certain answer suffices; in Karger's algorithm the smallest result is taken.
- The numbers a computer produces are pseudorandom. A fixed seed makes experiments reproducible; cryptography needs `SecureRandom`.

7.14 Key terms

randomized algorithm expected value expected running time average case

indicator random variable geometric distribution Markov's inequality Las Vegas algorithm

Monte Carlo algorithm amplification Fisher–Yates shuffle pivot partition quickselect

primality test witness, strong liar Carmichael number minimum cut edge contraction

pseudorandom number seed

7.15 Test Yourself

1. The Fisher–Yates shuffle is applied to the array [1, 2, 3, 4, 5, 6]. For *i = 5, 4, 3, 2, 1* the drawn values of *j* are 2, 4, 0, 1, 0 in turn. What is the array after each step?

   Answer

   i = 5, j = 2: [1, 2, 6, 4, 5, 3]. *i = 4, j = 4*: unchanged. *i = 3, j = 0*: [4, 2, 6, 1, 5, 3]. *i = 2, j = 1*: [4, 6, 2, 1, 5, 3]. *i = 1, j = 0*: [6, 4, 2, 1, 5, 3]. Five swaps were made; one of them left the element in place.

2. If the naive shuffle is applied to an array of five elements, how many equally likely paths are there? How can one tell that the result cannot be uniform? How many paths does Fisher–Yates have?

   Answer

   There are $5^5 = 3125$ paths and $5! = 120$ permutations. $3125 / 120 \approx 26.04$ is not an integer; 3125 is odd and 120 is even. The permutations cannot be produced by equal numbers of paths. Fisher–Yates has $5 \cdot 4 \cdot 3 \cdot 2 = 120$ paths and each permutation is obtained exactly once.

3. What is the expected number of shuffles of BogoSort on an array of eight distinct elements? What

is the approximate probability that the algorithm has still not finished after this many shuffles?

Answer

$8! = 40{,}320$ shuffles. The probability of not having finished is $(1 - 1/40{,}320)^{40320} \approx e^{-1} \approx 0.368$. In the geometric distribution the probability of not being done by the expected value is, for large n, about $1/e$.

4. For $a = [9, 4, 7, 1, 6, 3, 8]$, `partition(a, 0, 6)` is called and `rng.nextInt(7)` returns 4. What are the array after partitioning, the return value, and the number of comparisons?

Answer

The pivot is `a[4] = 6` and is moved to the end: [9, 4, 7, 1, 8, 3, 6]. 4, 1, and 3 are moved to the left in turn: [4, 9, 7, 1, 8, 3, 6], [4, 1, 7, 9, 8, 3, 6], [4, 1, 3, 9, 8, 7, 6]. The pivot settles at $k = 3$: [4, 1, 3, 6, 8, 7, 9]. The return value is 3 and the number of comparisons is $hi - lo = 6$.

5. In quicksort with a random pivot, with what probability are z_3 and z_7 compared in an array of 10 elements? What is the expected number of comparisons for an array of five elements?

Answer

The set $\{z_3, \ldots, z_7\}$ has 5 elements; the first pivot must be z_3 or z_7: $2/5$. The probability does not depend on the size of the array. For five elements, $H_5 = 137/60$ and $E[X] = 2 \cdot 6 \cdot 137/60 - 20 = 7.4$. The worst case is $5 \cdot 4/2 = 10$ comparisons.

6. With the Monte Carlo method, 7,860 of 10,000 points fell inside the quarter circle. What is the estimate of π? What is the approximate standard deviation of the estimate at this number of points?

Answer

$4 \cdot 7{,}860 / 10{,}000 = 3.144$. The standard deviation is $1.642 / \sqrt{10{,}000} \approx 0.016$. The true error is $|3.144 - \pi| \approx 0.0024$; it is smaller than the typical error.

7. What does the Fermat test say for $n = 91$ and $a = 3$? What does Miller–Rabin say with the same base?

Answer

Since $3^{90} \equiv 1 \pmod{91}$, the Fermat test is passed; 3 is a Fermat liar for 91. In Miller–Rabin, $90 = 2^1 \cdot 45$, that is, $s = 1$ and $d = 45$. $3^{45} \equiv 27 \pmod{91}$; 27 is neither 1 nor 90, and since $s = 1$ no further squaring is done. 3 is a witness: 91 = 7 · 13 is certainly composite.

8. In the call `powMod(3, 13, 17)`, what are `result` and `a` after each round? What is the return value?

   Answer

   $13 = 1101_2$. Bit 1: result = 3, a = 9. Bit 0: result = 3, a = 81 mod 17 = 13. Bit 1: result = 39 mod 17 = 5, a = 169 mod 17 = 16. Bit 1: result = 80 mod 17 = 12, a = 256 mod 17 = 1. The return value is 12: $3^{13} \bmod 17 = 12$.

9. What is the lower bound on the success probability of a single run of Karger's algorithm on a graph with eight vertices? How many repetitions suffice, with the exponential bound, to bring the failure probability below $1/8$? How many repetitions does the exact computation give?

   Answer

   The lower bound is $2/(8 \cdot 7) = 1/28$. For $e^{-T/28} \le 1/8$, $T \ge 28 \ln 8 \approx 58.2$, that is, 59 repetitions. Solving $(27/28)^T \le 1/8$ directly gives $T \ge \ln 8 / \ln(28/27) \approx 57.2$, so 58 repetitions suffice.

10. A Las Vegas algorithm with expected running time 100 ms is stopped after 300 ms. What can be said about the probability of no answer? If the algorithm is restarted independently at most five times with this rule, what is the largest probability that no answer is obtained at all?

    Answer

    By Markov's inequality, $P(T \ge 300) \le 100/300 = 1/3$. The probability that all five independent trials give no answer is at most $(1/3)^5 \approx 0.0041$. The total time is at most 1.5 seconds; this new algorithm is a Monte Carlo algorithm.

7.16 Further reading

- R. Motwani and P. Raghavan, *Randomized Algorithms* (1995): the classic reference on randomized algorithms; covers the Las Vegas and Monte Carlo types, Karger's algorithm, and primality tests with proofs.
- M. Mitzenmacher and E. Upfal, *Probability and Computing* (2nd ed., 2017): teaches probability tools such as indicator variables and the Markov and Chernoff bounds through algorithm examples.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 5, 7, and 9: probabilistic analysis, a detailed analysis of quicksort with a random pivot, and selection in expected linear time.
- D. E. Knuth, *The Art of Computer Programming*, Volume 2 (3rd ed., 1997): examines pseudorandom number generators, the theory of linear congruential generators, and the shuffling algorithm in detail.

7.17 References

1. R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
2. M. Mitzenmacher, E. Upfal, *Probability and Computing*, 2nd ed., Cambridge University Press, 2017.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapters 5, 7, 9, 31).
4. C. A. R. Hoare, "Quicksort", *The Computer Journal* 5(1), 1962.
5. M. O. Rabin, "Probabilistic Algorithm for Testing Primality", *Journal of Number Theory* 12(1), 1980.
6. D. R. Karger, "Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm", SODA, 1993.
7. D. E. Knuth, *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms, 3rd ed., Addison-Wesley, 1997 (Section 3.4.2: shuffling).

Online Algorithms

Most of the algorithms in this course see their entire input before they start. A sorting algorithm holds the whole array, and Dijkstra's algorithm holds the whole graph. In real systems, however, input often arrives piece by piece. A cache must decide which page to evict without knowing the next page request. A trading program buys or sells today without knowing tomorrow's price. Such decisions cannot be undone once they are made. Online algorithms are the algorithms that work under these conditions. These notes first set up the online problem model and competitive analysis: an online algorithm is judged against the best solution that knows the future. Three classical problems follow: ski rental, paging, and list update. For each of them we prove both an algorithm and a lower bound that says no algorithm can do better. The second half of the notes turns to data streams. Moving averages, online median, reservoir sampling, and pattern search in a stream are algorithms that look at each item once and use little memory. The notes close with incremental learning and a general comparison of the topic.

LEARNING OUTCOMES

Model

Distinguish the online and offline problem models, and explain irrevocable decisions with examples.

Competitive analysis

Define the competitive ratio and prove lower bounds with the adversary model. For ski rental, prove that the break-even strategy achieves the ratio $2 - 1/B$ and that this is the best possible value for deterministic algorithms.

Paging and lists

Compute page faults step by step under FIFO, LRU, and OPT; exhibit Belady's anomaly; interpret why LRU is k-competitive and why move-to-front is 2-competitive.

Data streams

Write moving average, online median, reservoir sampling, and rolling-hash pattern search in Java; analyze time per item and memory.

8.1 The online problem model

An algorithm's input can arrive in two ways. In the first, the entire input is available before the algorithm starts. In the second, the input is revealed over time, piece by piece, and the algorithm must answer each piece immediately. This chapter studies the second setting. We first describe the model itself, and then how "a good algorithm" is measured in it.

8.1.1 When information arrives piece by piece

Most everyday decisions are made with incomplete information. A chess player moves without knowing the opponent's next move. Once made, a move cannot be taken back. After the opponent replies, the player may think "I should have moved the other piece", but the position on the board does not change. A driver in a traffic jam likewise chooses which road to turn onto using only what is known at that moment, and learns that the road is closed ahead only on reaching it.

A traveler walking through a maze is a good example. The maze as a whole is invisible; the traveler sees only the corridor ahead. At each junction a direction is chosen from what has been seen so far. Someone with a bird's-eye map of the same maze finds the shortest path without ever going wrong, while the traveler is forced into some dead ends. The question an online algorithm asks is this: in the worst case, how much farther does the traveler without a map walk than the one with a map?

The same structure is common in computer systems. A program that controls traffic lights measures the density at the intersection in real time and adjusts the green phase accordingly. It does not know how many cars will arrive from which direction a minute later. Waiting is not an option either: the light has to show some color now. An **online algorithm** is an algorithm that works under these conditions.

Definition 8.1

An **online algorithm** receives its input as a sequence of requests $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$, one at a time. When σ_t arrives, the algorithm makes a decision. This decision depends only on the requests $\sigma_1, \ldots, \sigma_t$ and cannot be changed afterwards. An **offline algorithm**, in contrast, sees the whole sequence from the start.

The two conditions in the definition matter together. First, the future is unknown: when σ_t arrives there is no information about σ_{t+1}. Second, a decision, once made, cannot be undone. With only the first condition, the algorithm could wait for the entire input and then decide. With only the second, an algorithm that knew the future would make the right decision every time. The difficulty comes from having both at once.

8.1.2 Online versus offline

The difference between the two models lies not in the *content* of the input but in *when* the algorithm sees it. The same request sequence can be processed in either model. An offline algorithm looks at the whole sequence, computes the best decisions, and then carries them out. An online algorithm answers each request at the moment it arrives. Table 8.1 compares the two models along five dimensions.

	Offline	**Online**
Input	Known in full from the start	Arrives piece by piece; when σ_t arrives, σ_{t+1} is unknown
Decision	Made by looking at the whole input	Made the moment σ_t arrives, irrevocable
Examples	Sorting, Dijkstra, knapsack (dynamic programming)	Cache management, ski rental, stock trading
Optimal solution	OPT can be computed	OPT is known only in hindsight
Measure	Time and space complexity	Competitive ratio: ALG / OPT

Table 8.1: Offline and online problem models.

The last row of the table carries the main idea of the chapter. For an offline algorithm, "good" usually means "fast": the algorithm finds the optimal solution, and the question is how long it takes. An online algorithm often cannot find the optimal solution at all, because the optimum depends on the future. Here the question is how far the total cost of the decisions made is from the optimum. The number that measures this distance is called the **competitive ratio**.

The word "online" here has nothing to do with an internet connection. It describes the input being revealed over time. Even a sorting algorithm can be online or not, and two familiar sorting algorithms are enough to see this.

Example 8.1

Question. Numbers arrive one at a time: 5, 2, 8, 1. After each number arrives, the sorted version of the numbers so far must be ready. Can insertion sort and selection sort satisfy this requirement?

Solution. Insertion sort places each new number into the sorted part. When 5 arrives the array is [5]. When 2 arrives, 5 shifts one place right and the array is [2, 5]. When 8 arrives the comparison ends at the first step: [2, 5, 8]. When 1 arrives, three items shift: [1, 2, 5, 8]. At every step the array is sorted, and the algorithm never needs to know the future. Selection sort, in contrast, picks the minimum of the whole input in its first pass. When 5 arrives it cannot write it to the first position, because it does not know whether a smaller number will come. Indeed, the smallest number (1) arrives last. Insertion sort can run online; selection sort cannot. Merge sort and quicksort also need the whole input to split it in two.

8.1.3 Bounded memory and work per item

The most common practical form of the online model is continuously arriving data. Sensors produce thousands of measurements per second, and millions of packets pass through a network router every second. Storing all of this data is usually impossible. Figure 8.1 shows the situation: data keeps arriving at the algorithm, the algorithm works with a small buffer, and it produces output immediately.

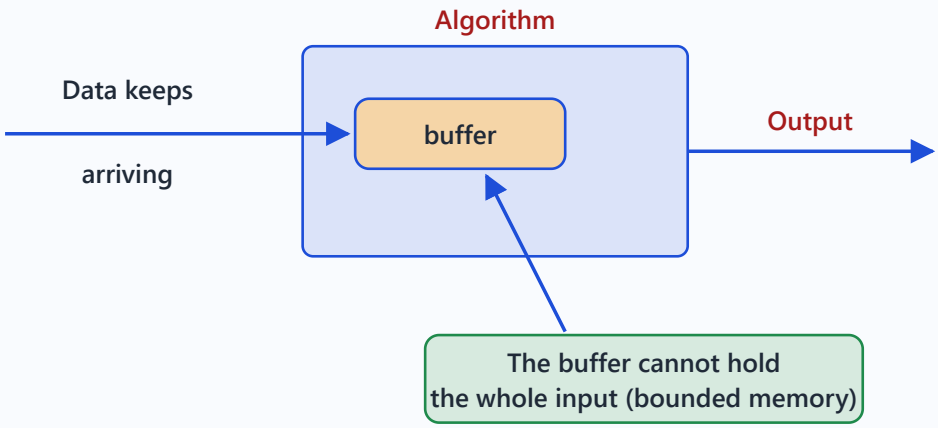


Figure 8.1: The online algorithm model. Data arrives continuously; the algorithm processes each piece as it arrives and produces output. The buffer cannot hold the whole input.

If the buffer cannot hold the whole input, the past can be kept only as a *summary*. A summary may be a sum, a counter, two heaps, or a small sample. Choosing the right summary is the heart of stream algorithm design. A summary needs two properties. First, it must be quickly updatable when a new item arrives. Second, the question being asked must be answerable from the summary. For example, to answer "what is the average so far?", the item count and the sum are enough. With each new item, the sum and the counter are each updated once, and the average is obtained with one division.

For this reason two costs are measured in stream algorithms: *work per item* and *memory*. We usually want the work per item to be $O(1)$ or $O(\log n)$. An algorithm that does $O(n)$ work per item does $O(n^2)$ work in total after n items arrive. In a stream delivering a million items per second this is unacceptable. Memory, too, must be independent of the stream length or very small relative to it.

8.1.4 Application areas

Online problems appear in almost every area of computer science. Table 8.2 collects the main examples. They share three features: the environment is dynamic, decisions are made in real time, and information about the future is incomplete.

Area	Online decision
Stock market	Buy or sell according to the current price; the future price is unknown.

Area	Online decision
Caching	Which page stays in memory in a processor, an operating system, or a web cache.
Routing	Which path a packet takes in a network whose links keep changing.
Resource allocation	Dynamic sharing of bandwidth and processor time.
Scheduling	Assigning arriving jobs to machines (job scheduling).
Advertising and recommendation	Which advertisement or product to show when a user enters a page.
Machine learning	Updating a model with every new example (incremental learning).
Security	Catching fraud and distributed denial-of-service (DDoS) attacks on the stream.
Autonomous vehicles	Real-time decisions as sensor data arrives.

Table 8.2: Main areas in which online decisions are made.

In some of these areas the decision is irrevocable, and in others undoing it is expensive. A page evicted from a cache can be loaded back, but loading has a cost. A displayed advertisement cannot be undone; the ad slot cannot be given to another ad. The model is always the same: every decision has a cost, and we want to minimize the total cost.

8.1.5 Strengths and weaknesses

The greatest strength of online algorithms is that they do not wait. They do not wait for the input to end, and they work in dynamic environments. Because decisions are made on the spot, latency is low. With little memory they can process very large, even infinite, streams. Thanks to the competitive ratio, a worst-case guarantee can also be given: for example, "whatever happens, the cost never exceeds twice the optimum."

The weaknesses come from the same place. A wrong decision in the early steps affects later steps. In chess, a single bad move can lose the game. An online algorithm does not guarantee finding the best global solution. It is sensitive to the initial input: the same algorithm may do very well on one sequence and badly on another. Its design and analysis are also harder than those of offline algorithms, since the analysis has to evaluate the algorithm's behavior against every possible future not yet seen.

8.1.6 Design approaches

Online algorithms are built on a few common ideas. They are not mutually exclusive; an algorithm may use several together.

- **Greedy approach**: serve the arriving request in the way that looks best at that moment, without considering global effects. Assigning each arriving request to the nearest idle server is of this kind.
- **Summarization**: instead of storing the entire past, keep a small summary. A moving average keeps a sum and a queue, an online median keeps two heaps, and reservoir sampling keeps a fixed-size sample.

- **Adaptive approach**: learn from past requests and change behavior over time. An LRU cache keeps recently used pages. The move-to-front rule brings frequently requested items to the head of the list. Incremental learning updates the model with every new example.
- **Deterministic approach**: always make the same decision on the same input. This predictability is also a weakness. A malicious input generator that knows the algorithm, an adversary, can easily trap it.
- **Randomized approach**: add chance to the decisions. The adversary cannot know in advance what the algorithm will do. Reservoir sampling and the randomized marking algorithm are of this kind.

The difference between deterministic and randomized approaches shows up in numbers in competitive analysis. In ski rental the best deterministic ratio is close to 2, while a randomized strategy comes down to about 1.58. In paging the deterministic ratio is k, while that of the randomized marking algorithm is around $2 \ln k$.

8.2 Competitive analysis

For offline algorithms the usual measure is time complexity. For online algorithms this measure is not enough. A cache policy may do $O(1)$ work per request and still produce a large number of unnecessary page faults. What really needs to be measured is the quality of the decisions. Measuring quality requires a point of comparison. That point is the best algorithm that knows the future.

8.2.1 The competitive ratio

Let a request sequence σ be given. The total cost of the online algorithm on this sequence is denoted $ALG(\sigma)$. The cost of the best offline algorithm, which knows the whole sequence in advance, is denoted $OPT(\sigma)$. OPT is less an algorithm than a yardstick: it gives the smallest cost attainable on each sequence. Whatever the online algorithm does, its cost cannot be below OPT. The question is how much larger than OPT it can be.

Definition 8.2

In a cost-minimization problem, an online algorithm ALG is **c-competitive** if, for every request sequence σ,

$$ALG(\sigma) \le c \cdot OPT(\sigma) + \alpha$$

holds. Here α is a constant independent of σ. If $\alpha = 0$, the algorithm is *strictly* c-competitive. The smallest c satisfying this condition is called the algorithm's **competitive ratio**. When $\alpha = 0$, the ratio is the supremum of $ALG(\sigma) / OPT(\sigma)$ over all sequences.

Three aspects of the definition deserve emphasis. First, the condition must hold for *every* sequence. Competitive analysis is a worst-case analysis; it contains no average-case or probabilistic assumption. Saying "it usually works well on real sequences" does not give a competitive ratio. Second, the closer the ratio is to 1, the less the lack of knowledge about the future hurts. An online algorithm with ratio 1 is as good as an algorithm that knows the future. Third, the additive constant α leaves start-up effects out. For example, if the cache is initially empty, the first few faults occur in every algorithm. These faults are insignificant on long sequences, and α absorbs them.

The definition is for cost problems: the goal is to minimize cost, and the ratio is always $c \geq 1$. In some problems, however, a gain is maximized. In ad placement, for example, total revenue is to be increased. For gain problems the ratio is defined by inverting it, as OPT / ALG, so the ratio is again at least 1.

Example 8.2

Question. An online algorithm costs 12, 30, 8, and 50 on four request sequences; on the same sequences OPT costs 6, 20, 8, and 25. If these were the only four sequences, what would the algorithm's competitive ratio be? What do these numbers say about the true ratio?

Solution. The ratios are $12/6 = 2$, $30/20 = 1.5$, $8/8 = 1$, and $50/25 = 2$. Because the competitive ratio measures the worst case, we look at the maximum rather than the average: 2. Although the average ratio is 1.625, this number gives no guarantee. The true competitive ratio is defined over all possible sequences. These four sequences show that the ratio is at least 2. To show that it does not exceed 2, a proof valid for every sequence is required.

8.2.2 The adversary model

To show that an algorithm's competitive ratio is *at least* c, a single bad sequence suffices. Showing that no algorithm can do better than c is harder: a separate bad sequence must be constructed against every algorithm. The tool that organizes this construction is the **adversary** model. The adversary is an imaginary opponent playing against the algorithm. It generates the request sequence and also solves the same sequence with OPT. Its goal is to make the ratio $ALG(\sigma) / OPT(\sigma)$ as large as possible.

What the adversary knows depends on the type of algorithm being analyzed.

- The **adversary against a deterministic algorithm** knows the algorithm's code. Because the algorithm is deterministic, the adversary can compute in advance what it will do at each step. At each step it generates the request that hurts the algorithm most.
- The **oblivious adversary** is used against randomized algorithms. It knows the algorithm's code but cannot see the coin flips. It therefore chooses the sequence in advance. The algorithm's *expected* cost is measured.

- The **adaptive adversary** sees the algorithm's random decisions and chooses the next request accordingly. Against this adversary, most of the benefit of randomization is lost.

Lower-bound proofs always follow the same pattern. To say "no deterministic algorithm can be better than c", take an arbitrary ALG. Against this ALG, construct a sequence σ with $ALG(\sigma) / OPT(\sigma) \geq c$. Because ALG was arbitrary, the result holds for all deterministic algorithms. The power of the pattern comes from the adversary knowing ALG. A deterministic algorithm cannot hide what it will do in the next step, and the adversary exploits exactly that step. In ski rental, the adversary ends the season on the day the algorithm buys the skis. In paging, it requests the page that is not in the algorithm's cache.

Common mistake

Confusing the competitive ratio with average-case behavior. The inference "LRU gives results very close to OPT on real programs, so its ratio is close to 1" is wrong. The competitive ratio is defined by the worst sequence. LRU's ratio is k, because there is a sequence that makes it fault on every request. Good performance on real sequences is a separate observation; it comes from locality and is not a guarantee.

8.3 Ski rental

Ski rental is the simplest example of an online algorithm. There is a single decision: when to buy. Even so, the problem contains every part of competitive analysis: an OPT that knows the future, a simple strategy, an upper bound for that strategy, and a lower bound that no strategy can beat.

8.3.1 The problem

A skier goes to the ski resort every morning throughout the season. A set of skis can be rented for 1 TL per day or bought for B TL. Bought skis are free for the rest of the season. The skier does not know how long the season will last. Each morning the skier learns only the answer to "will I ski today too?" If the weather turns bad one day, the season ends that day. The total number of skiing days is denoted d.

Definition 8.3

The **ski rental problem**: renting costs 1 per day, buying costs B units. At the start of each skiing day the algorithm chooses one of two options: rent for that day, or buy. After buying there is no further cost. The number of days d is not known in advance. The goal is to minimize the total cost.

For OPT, which knows the future, the problem is easy. If d is known, the two options are compared. Renting every day costs d, and buying on the first day costs B. Buying on some day in between is never better: paying rent for a few days and buying later costs more than buying on day one. Therefore

$$OPT(d) = \min(d, B)$$

holds. If $d < B$, OPT always rents; if $d \geq B$, it buys on the first day. The online algorithm, however, does not know d. Whichever day it picks, the adversary can end the season in the way that is worst for it. Figure 8.2 shows OPT's cost for $B = 10$ together with the cost of the break-even strategy examined below.

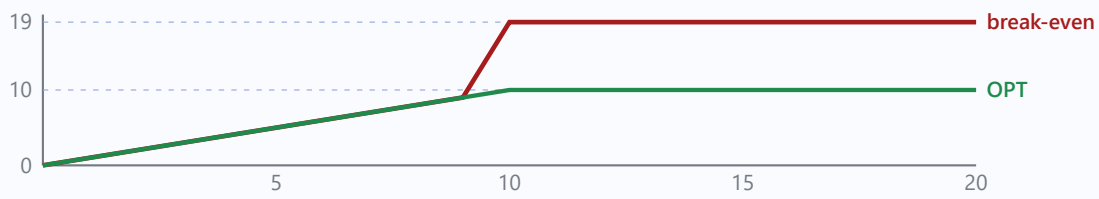


Figure 8.2: Costs for $B = 10$; the horizontal axis is the actual number of days d. OPT is $\min(d, 10)$. The break-even strategy equals OPT while $d < 10$ and then stays at 19. The worst ratio occurs at $d = B$: $19/10 = 2 - 1/B$.

The structure of the problem is not limited to skiing. A cloud server can be rented by the hour or bought for a year. A TCP connection can be kept open, or closed and reopened when needed. Keeping it open costs a little every second, while reopening is a single large cost. A thread can spin while waiting for a lock to be released, or go to sleep. Spinning is like renting; sleeping and being woken up is like buying. In all of them the question is the same: how long should the small recurring cost be paid, and when should we switch to the large one-time cost?

8.3.2 The cost of three strategies

A deterministic strategy is defined by a single number: the buying day k. The strategy rents for $k - 1$ days and buys on day k if the skier is still skiing. If the season ends before day k, it never buys. This family has three natural members.

- **Always rent** ($k = \infty$): the cost is always d. As d grows, OPT stays fixed at B while the cost increases without bound. The ratio is d / B, which goes to infinity as $d \to \infty$. This strategy is not competitive.
- **Buy on day one** ($k = 1$): the cost is always B. But if the season lasts a single day, OPT is 1. The ratio is B. If B is large, this is very bad.
- **Break-even** ($k = B$): rent for $B - 1$ days and buy on day B. It buys at the moment the total paid in rent approaches the purchase price.

Figure 8.3 plots the cost of these three strategies together with OPT as a function of the number of days, for $B = 10$. Table 8.3 gives the same values numerically for a few values of d.

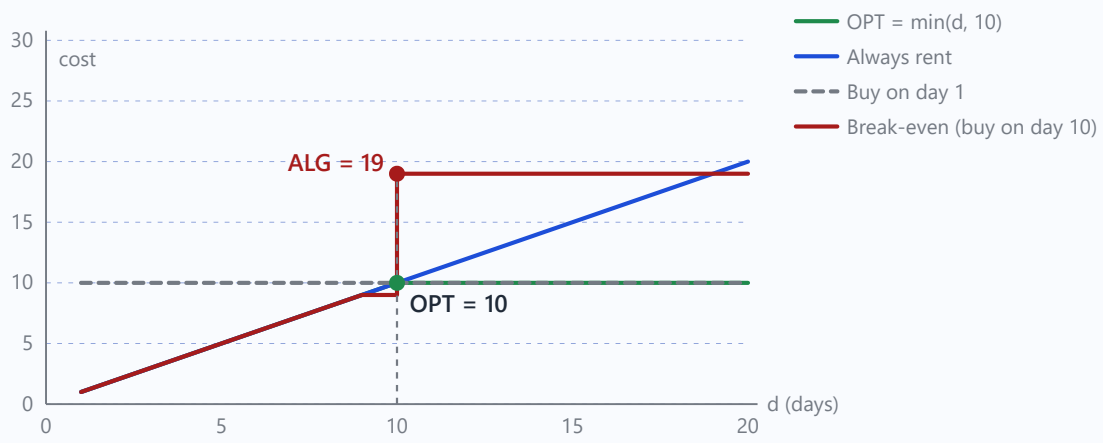


Figure 8.3: Cost of the strategies for $B = 10$. The always-rent line rises without bound. Buy-on-day-one is constant at 10. The break-even strategy coincides with OPT up to day 9 and stays at 19 from day 10 on. The worst case is $d = 10$: ALG = 19, OPT = 10.

d	OPT	Always rent	Buy on day 1	Break-even
1	1	1 (1)	10 (10)	1 (1)
5	5	5 (1)	10 (2)	5 (1)
9	9	9 (1)	10 (1.11)	9 (1)
10	10	10 (1)	10 (1)	19 (1.9)
15	10	15 (1.5)	10 (1)	19 (1.9)
50	10	50 (5)	10 (1)	19 (1.9)
1000	10	1000 (100)	10 (1)	19 (1.9)

Table 8.3: Cost of the three strategies for $B = 10$ and the ratio to OPT (in parentheses).

The table shows the weaknesses of the strategies clearly. Always-rent is perfect on short seasons and a disaster on long ones. Buy-on-day-one is perfect on long seasons and B times worse on short ones. Break-even is never perfect, but it is never worse than 1.9 times optimal. Competitive analysis looks for exactly this kind of balance: the strategy that makes the worst case best.

8.3.3 The break-even strategy: code and trace

The Java code of the break-even strategy is below. The method takes the actual season length d as a parameter. This is for simulation only: the algorithm does not know d. At the start of each day the loop only "sees" whether skiing happens that day; it makes no decision based on the value of d.

```
int breakEven(int B, int d) {  // d is unknown
  int cost = 0;
  for (int day = 1; day <= d; day++) {
    if (day < B) {
      cost += 1;               // rent
    } else {
      cost += B;               // buy
      break;                   // no more fees
    }
  }
  return cost;
}
```

The code works line by line as follows. `cost` starts at zero. The loop runs from day 1 to day *d*; each iteration stands for one skiing day. If `day < B`, the skis are rented for that day and the cost grows by 1. The first time `day` equals *B*, the `else` branch runs: *B* is added to the cost and `break` leaves the loop. The skis now belong to the skier; the remaining days are free. If the season ends before day *B*, the loop ends on its own and no purchase is made.

The method's time is *O(min(d, B))*: the loop runs at most *B* times. Memory is *O(1)*. For online algorithms, though, the interesting quantity is not this running time but the ratio of the returned `cost` to OPT. Table 8.4 gives a trace of the code for *B = 4* and *d = 6*. Figure 8.4 shows the final state of the same run.

`day`	`day < B`	**Decision**	`cost`
1	true	rent	1
2	true	rent	2
3	true	rent	3
4	false	buy, `break`	3 + 4 = 7
5	—	loop ended; skis are free	7
6	—	loop ended; skis are free	7

Table 8.4: Trace of the call `breakEven(4, 6)`. The season actually lasts 6 days; the algorithm does not know this.

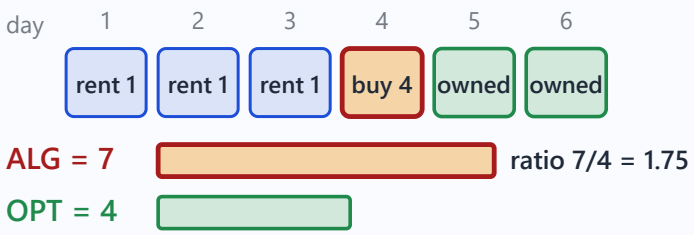


Figure 8.4: *B = 4, d = 6*: three days of rent, purchase on day 4, days 5 and 6 free. ALG = 7, OPT = *min(6, 4) = 4*, ratio *7/4 = 1.75 = 2 − 1/B*.

The result is ALG = 7. Because OPT knows the season will last 6 days, it buys on the first day and pays 4. The ratio is $7/4 = 1.75$, which equals $2 - 1/4$. Had the season ended on day 4, ALG would still be 7 and OPT 4. Had it ended on day 3, ALG would be 3, and OPT would also pay 3. A general version of the code takes the buying day as a separate parameter:

```
int skiCost(int B, int k, int d) {
  int cost = 0;
  for (int day = 1; day <= d; day++) {
    if (day == k) { cost += B; break; }  // buy
    cost += 1;  // rent
  }
  return cost;
}
```

This method computes the cost of the strategy that buys on day k. `skiCost(B, B, d)` gives the same value as `breakEven(B, d)` for every B and d. The two methods were run and compared for $1 \le B \le 15$ and $1 \le d \le 40$; all values were equal.

Common mistake

Forgetting `break`. Without `break` in the `else` branch the loop continues, and after day B another B is added every day. The skier ends up buying a new set of skis every morning. For $B = 4$ and $d = 6$ the cost comes out as $3 + 4 + 4 + 4 = 15$; as d grows the ratio grows without bound. Writing the condition as `day <= B` is also an off-by-one error. The strategy then rents for B days and buys on day $B + 1$. For $d = B + 1$ the cost is $2B$ and OPT is B, so the ratio rises to 2 instead of $2 - 1/B$.

8.3.4 Upper bound: the ratio of break-even

The ratio in the trace is no coincidence. The ratio of the break-even strategy never exceeds $2 - 1/B$ for any d.

Theorem 8.1 (Break-even strategy)

The strategy that rents for $B - 1$ days and buys on day B is strictly $(2 - 1/B)$-competitive. That is, $ALG(d) \le (2 - 1/B) \cdot OPT(d)$ for every $d \ge 1$, with equality when $d \ge B$.

Proof. There are two cases. If $d < B$, the strategy never buys; it rents for d days and pays d. OPT also pays $\min(d, B) = d$. The ratio is 1. If $d \ge B$, the strategy pays $B - 1$ days of rent and buys on day B: $ALG = (B - 1) + B = 2B - 1$. OPT buys on the first day and pays B. The ratio is $(2B - 1)/B = 2 - 1/B$. In both cases the ratio does not exceed $2 - 1/B$. □

The proof also explains why the break-even idea works. Up to the moment of buying, the algorithm never pays more than OPT would. At the moment of buying, it pays in total at most "one rent and one purchase", each of which is at most OPT. Therefore the total cannot exceed twice OPT. This same "at most twice" idea appears in many online problems.

8.3.5 Lower bound: why nothing does better

The break-even strategy achieves the ratio $2 - 1/B$. Could a cleverer deterministic strategy lower it? The adversary model shows that this is impossible.

Theorem 8.2 (Ski rental lower bound)

For the ski rental problem, no deterministic online algorithm has a competitive ratio smaller than $2 - 1/B$.

Proof. Every deterministic strategy is defined by a buying day k. The strategy that never buys is $k = \infty$; its ratio is unbounded. The adversary knows the strategy and ends the season exactly on the buying day: $d = k$. The algorithm then pays $k - 1$ days of rent and one purchase: $ALG = (k - 1) + B$. OPT pays $\min(k, B)$. There are two cases.

If $k < B$, the ratio is $(k - 1 + B)/k = 1 + (B - 1)/k$. Since $k \le B - 1$, we have $(B - 1)/k \ge 1$, so the ratio is at least 2. That is greater than $2 - 1/B$.

If $k \ge B$, the ratio is $(k - 1 + B)/B$. The numerator grows with k; the smallest value occurs at $k = B$: $(2B - 1)/B = 2 - 1/B$.

For every k, the ratio on the adversary's chosen sequence is at least $2 - 1/B$. □

Table 8.5 gives the ratio the adversary obtains against different buying days for $B = 10$. Buying too early and buying too late are both bad; the worst-case ratio is smallest exactly at $k = B$.

k	Rent	ALG	OPT	Ratio
1	0	10	1	10
2	1	11	2	5.5
5	4	14	5	2.8
9	8	18	9	2
10	9	19	10	1.9
11	10	20	10	2
15	14	24	10	2.4
20	19	29	10	2.9

Table 8.5: $B = 10$; the adversary ends the season on the buying day ($d = k$).

The adversary's choice $d = k$ really is the worst choice. If $d < k$, the algorithm never buys and pays only rent; the ratio is at most 1. If $d > k$, the algorithm's cost does not change while OPT's cost may grow or stay the same; the ratio does not grow. This observation was also verified by simulation. For $B = 10$, the strategies $k = 3, 5, 10, 14$ were run for $d = 1, \ldots, 60$. In each, the worst ratio occurred at $d = k$: 4, 2.8, 1.9, and 2.3 respectively.

Example 8.3

Question. In ski rental, $B = 20$. What is the competitive ratio of the break-even strategy? What is the worst ratio of the strategy that buys on day 5?

Solution. The ratio of the break-even strategy is $2 - 1/20 = 1.95$. The adversary ends the season on day 20: ALG = $19 + 20 = 39$, OPT = 20, ratio $39/20 = 1.95$. Against the strategy that buys on day 5, the adversary picks $d = 5$. The algorithm pays 4 days of rent and a purchase: $4 + 20 = 24$. Because OPT knows the season lasts 5 days, it rents throughout and pays 5. The ratio is $24/5 = 4.8$. The early-buying strategy is about 2.5 times worse than break-even.

8.3.6 A randomized strategy

The power of the lower-bound proof comes from the adversary knowing the buying day. If the algorithm picks its buying day at random, the adversary loses this information. The oblivious adversary must choose the sequence, that is, d, in advance. The algorithm's expected cost is measured.

Example 8.4

Question. Let $B = 2$. The ratio of the deterministic break-even strategy is $2 - 1/2 = 1.5$. If the algorithm buys on day 1 with probability p and on day 2 with probability $1 - p$, what is the expected ratio? What is the best p?

Solution. The adversary has two meaningful choices. If it picks $d = 1$, the strategy that buys on day 1 pays 2 and the one that buys on day 2 pays 1; OPT is 1. The expected ratio is $2p + (1 - p) = 1 + p$. If it picks $d \geq 2$, the day-1 buyer pays 2 and the other pays $1 + 2 = 3$; OPT is 2. The expected ratio is $(2p + 3(1 - p))/2 = (3 - p)/2$. The adversary picks the larger. The algorithm minimizes the worst case by equating the two values: $1 + p = (3 - p)/2$, so $p = 1/3$. The expected ratio is $4/3 \approx 1.33$. Randomization lowered the ratio from 1.5 to 1.33. $p = 0$ is the deterministic break-even strategy, with worst ratio 1.5; $p = 1/2$ also gives 1.5.

The same idea generalizes to every B. Karlin, Manasse, McGeoch, and Owicki (1994) found the best randomized strategy. In this strategy, the probability of buying on day i is proportional to $((B - 1)/B)^{B - i}$; that is, later days get larger probability. The resulting expected ratio is

$$1 / (1 - (1 - 1/B)^B)$$

For $B = 2$ this formula gives $1/(1 - 1/4) = 4/3$, matching the value in the example. As B grows, $(1 - 1/B)^B$ approaches $1/e$, and the ratio goes to $e/(e - 1) \approx 1.58$. Table 8.6 compares the deterministic and randomized ratios.

B	Deterministic $2 - 1/B$	Randomized (expected)
2	1.5	1.333
3	1.667	1.421
4	1.75	1.463
10	1.9	1.535
100	1.99	1.577
$B \to \infty$	2	$e/(e - 1) \approx 1.582$

Table 8.6: Ratios of the best deterministic and the best randomized strategy for ski rental.

For $B = 10$, the probabilities increase from 0.059 on day 1 to 0.154 on day 10. With this distribution the expected ratio was computed for every d; the worst value came out as 1.535. Randomization takes away the adversary's strongest weapon, knowing when the algorithm will buy. But there is a limit to this too. Karlin et al. showed that these ratios are also the best possible for randomized strategies. For large B, no randomized strategy can do noticeably better than $e/(e - 1)$.

8.4 Paging

Paging is the most studied online problem. It appears everywhere in the memory organization of computers: in the processor cache, in the operating system's virtual memory, in the database buffer, and in the web browser's cache. In all of them a small, fast memory holds part of a large, slow memory. Which part to keep is decided without knowing future requests.

8.4.1 The problem

Consider a two-level memory. The fast memory, the **cache**, can hold k pages at a time. The slow memory is unbounded and contains all pages. The processor requests pages one after another. If the requested page is in the cache, this is a **hit** and costs 0. If the page is not in the cache, a **page fault** occurs. The page is loaded from slow memory and the cost is 1. If the cache is full, a page must be evicted to make room for the new one. Figure 8.5 shows this moment.

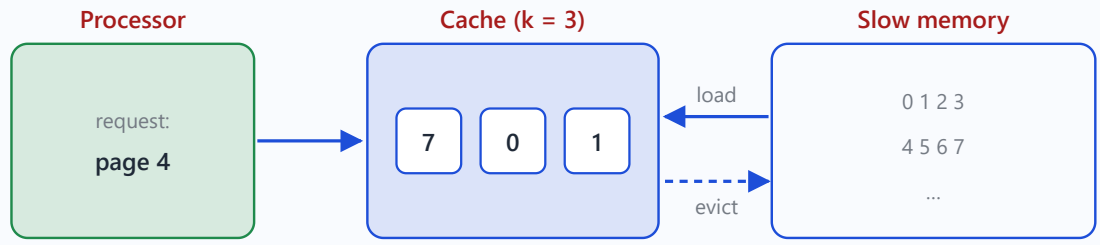


Figure 8.5: The processor requests page 4. The cache of capacity $k = 3$ holds 7, 0, and 1. Page 4 must be loaded from slow memory and one of the pages in the cache must be evicted.

Definition 8.4

The **paging problem**: given a cache of k pages and a sequence σ of page requests. Each request is either a hit (cost 0) or a page fault (cost 1). On a fault, if the cache is full, the algorithm chooses the page to evict. The goal is to minimize the total number of page faults. The rule that chooses the page to evict is called the **eviction policy**.

In this model only faults are counted. In a real system a hit also has a small cost, but loading from slow memory is hundreds or even millions of times more expensive. Reading a page from disk takes milliseconds, and reading from memory takes nanoseconds. For this reason the number of faults is the main determinant of performance. The model also loads a page only on a fault; there is no prefetching. This restriction does not lose generality: any algorithm that prefetches can be converted into one that postpones loading until it is really needed, without increasing the number of faults.

The problem is a typical online problem. Which page should be evicted depends on future requests. If we knew which page would be requested next, the right choice would be easy. But the algorithm sees only the past.

8.4.2 Eviction policies

Eviction policies use different pieces of information from the past. Table 8.7 lists the main ones with their competitive ratios.

Policy	Which page does it evict?	Online?	Competitive ratio
FIFO	The page loaded into the cache earliest (a queue)	yes	k
LRU	The page unused for the longest time (least recently used)	yes	k
FWF	Flushes everything on a fault when the cache is full (flush when full)	yes	k
LFU	The least frequently used page	yes	unbounded
LIFO	The most recently loaded page	yes	unbounded

Policy	Which page does it evict?	Online?	Competitive ratio
OPT (MIN)	The page whose next use is farthest in the future (Belady, 1966)	no	1

Table 8.7: Page eviction policies (*k*: cache size).

FIFO (first in, first out) manages the cache like a queue. The page at the head of the queue, that is, the one loaded earliest, is evicted. It ignores how often a page is used. **LRU** tracks the last use time of each page and evicts the oldest. It assumes that a page used in the recent past will soon be used again. **FWF** is a crude policy. If a fault occurs while the cache is full, it discards all pages and starts over. Crude as it looks, its competitive ratio is the same as that of FIFO and LRU: *k*.

LFU and **LIFO**, on the other hand, are not competitive. Why LIFO is bad can be seen on a small sequence. For $k = 3$, consider the sequence 1 2 3 4 3 4 3 4 .... The first three requests fill the cache. When 4 arrives, LIFO evicts 3, the most recently loaded. Then 3 is requested, and this time 4, the most recently loaded, is evicted. As the sequence continues like this, every request is a fault. OPT, in contrast, evicts 1 when 4 arrives, since 1 will never be requested again, and then never faults. As the sequence grows, LIFO's fault count grows while OPT's stays at 4. The ratio grows without bound. LFU is trapped in a similar way. If pages 1 and 2 are requested *m* times and then 3 and 4 are requested alternately, LFU keeps 1 and 2, whose counters are large. Pages 3 and 4 evict each other until their counters approach *m*. As *m* grows, so does the ratio.

OPT is the policy Belady defined in 1966; it is also known as MIN or "longest forward distance". On a fault it looks at the next use of every page in the cache and evicts the one used farthest in the future. If some page will never be used, it is evicted first. OPT requires knowing the future, so it cannot be implemented in a real system. But it is indispensable as a yardstick. It is proven to give the fewest page faults on every sequence. This is exactly the OPT in the competitive ratio.

8.4.3 FIFO step by step

To compare the three policies on the same sequence we use the request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2 and $k = 3$. In this sequence page 0 is requested often and the other pages rarely. Figure 8.6 shows FIFO's slot contents after each request; S1, S2, and S3 are the three slots of the cache. A newly loaded page is written into the slot of the evicted page, so positions within the slots do not show the loading order. The loading order is in the queue column of Table 8.8.



Figure 8.6: Paging with FIFO, request sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, $k = 3$. F: page fault, ✓: hit. 10 faults in total.

#	Request	Result	Evicted	Queue	Fault
1	7	F	—	[7]	1
2	0	F	—	[7, 0]	2
3	1	F	—	[7, 0, 1]	3
4	2	F	7	[0, 1, 2]	4
5	0	✓	—	[0, 1, 2]	4
6	3	F	0	[1, 2, 3]	5
7	0	F	1	[2, 3, 0]	6
8	4	F	2	[3, 0, 4]	7
9	2	F	3	[0, 4, 2]	8
10	3	F	0	[4, 2, 3]	9
11	0	F	4	[2, 3, 0]	10
12	3	✓	—	[2, 3, 0]	10
13	2	✓	—	[2, 3, 0]	10

Table 8.8: FIFO trace. The queue is written from oldest to newest; a hit does not change the queue.

The critical moment in the trace is the 6th request. Page 0 had just been used, at the 5th request. FIFO does not take this into account. Because 0 is at the head of the queue, it is evicted. Right after that, at the 7th request, 0 is requested again and an unnecessary fault occurs. The same happens at the 11th request: 0 was evicted at the 10th request and requested again one request later. FIFO makes 10 faults in total.

8.4.4 LRU step by step

On every hit, LRU updates the page's last use time. This small difference keeps frequently used pages in the cache. Figure 8.7 shows how LRU processes the same sequence, and Table 8.9 shows the usage order at each step.



Figure 8.7: Paging with LRU, same sequence and $k = 3$. 9 faults in total.

#	Request	Result	Evicted	Usage order	Fault
1	7	F	—	[7]	1
2	0	F	—	[7, 0]	2
3	1	F	—	[7, 0, 1]	3
4	2	F	7	[0, 1, 2]	4
5	0	✓	—	[1, 2, 0]	4
6	3	F	1	[2, 0, 3]	5
7	0	✓	—	[2, 3, 0]	5
8	4	F	2	[3, 0, 4]	6
9	2	F	3	[0, 4, 2]	7
10	3	F	0	[4, 2, 3]	8
11	0	F	4	[2, 3, 0]	9
12	3	✓	—	[2, 0, 3]	9
13	2	✓	—	[0, 3, 2]	9

Table 8.9: LRU trace. The usage order is written from oldest to newest; a hit moves the page to the end.

The difference starts at the 5th request. When 0 gets a hit, LRU moves it to the end of the order. At the 6th request, the evicted page is no longer 0 but 1, which has gone unused the longest. At the 7th request 0 gets a hit again. The unnecessary faults FIFO made at these two requests do not occur in LRU. There are moments when LRU is also helpless. At the 10th request 0 is the least recently used page and is evicted; at the 11th request it is requested again. LRU makes 9 faults in total.

8.4.5 OPT step by step

OPT looks to the future at every fault. For each page in the cache it finds the position of the next request, and evicts the farthest one. Figure 8.8 shows OPT's slots, and Table 8.10 gives the next-use positions computed at each eviction.

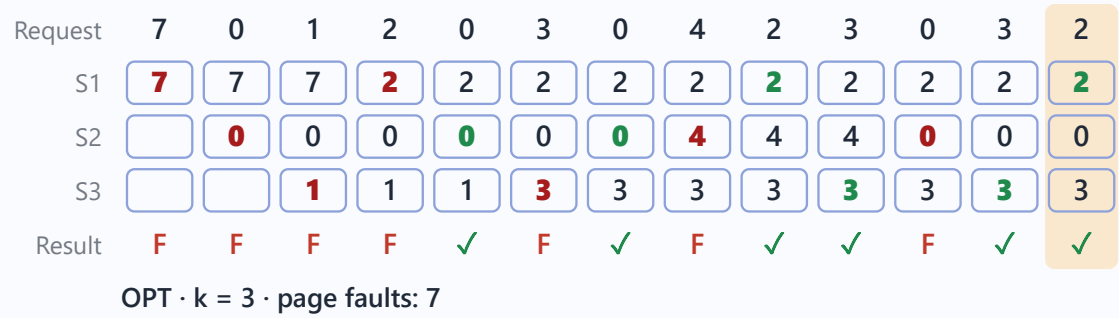


Figure 8.8: Paging with OPT, same sequence and $k = 3$. 7 faults in total.

#	Request	Cache	Next use	Evicted	Fault
4	2	{7, 0, 1}	7: none, 0: 5th, 1: none	7	4
6	3	{2, 0, 1}	2: 9th, 0: 7th, 1: none	1	5
8	4	{2, 0, 3}	2: 9., 0: 11., 3: 10.	0	6
11	0	{2, 4, 3}	2: 13th, 4: none, 3: 12th	4	7

Table 8.10: OPT's eviction decisions. The next-use column gives the position at which each page is next requested; "none" means the page is never requested again.

The first three requests are compulsory faults. At the 4th request the cache holds 7, 0, and 1. Neither 7 nor 1 is ever requested again; either can be evicted. OPT evicts 7. Evicting 1 would give the same number of faults: 1 would be evicted at the 6th request anyway. The decision at the 8th request is the most instructive. Page 0 is the most frequently requested page in the sequence, but its next use is the farthest away: the 11th request. OPT evicts it. Pages 2 and 3 get hits at the 9th and 10th requests. LRU had evicted 2 at the same moment and made two faults. OPT makes 7 faults in total.

Common mistake

Understanding OPT as "evict the page that will be used least". OPT looks not at how many times a page will be used in the future but at when its next use is. In the sequence above, page 0 is requested once more in the future and 4 never again. Even so, 0 is evicted at the 8th request, because the candidates in the cache are 2, 0, and 3; 4 is not in the cache at that moment. Among the candidates, 0 has the farthest next use.

8.4.6 Comparing the three policies

Table 8.11 gives the results of the three policies together.

Policy	Page faults	Hits	ALG / OPT	Information needed
FIFO	10	3	1.43	Loading order (queue)

Policy	Page faults	Hits	ALG / OPT	Information needed
LRU	9	4	1.29	Last use time
OPT	7	6	1	All future requests

Table 8.11: Three policies on the sequence 7 0 1 2 0 3 0 4 2 3 0 3 2, for $k = 3$.

The first $k = 3$ faults cannot be avoided in any policy. The cache is initially empty, and the first three distinct pages must be loaded no matter what. These faults are called **compulsory misses** or **cold misses**. Even OPT cannot make zero faults. The constant α in the definition of the competitive ratio exists precisely to absorb start-up effects like these.

The reason LRU beats FIFO is **locality of reference**. Programs do not access memory at random. A loop accesses the same variables again and again; an array is read from consecutive addresses. A page used in the recent past will be used again soon. LRU exploits this regularity, and FIFO does not. In the example sequence, because page 0 is requested often, FIFO evicted it twice unnecessarily.

In real systems LRU itself is expensive. Updating a timestamp on every memory access is a heavy burden for hardware. For this reason operating systems use cheap approximations close to LRU. The best known is the **clock algorithm**, also called second chance. Each page has a "referenced" bit; the hardware sets the bit to 1 when the page is accessed. Pages sit in a circular list, and a hand sweeps around it. When an eviction is needed, the hand clears the bit of a page whose bit is 1 and moves on, and evicts the first page whose bit is 0. Recently used pages thus get one more round of grace.

8.4.7 Java code and trace for LRU

The most direct implementation of LRU uses a linked list. The head of the list is the least recently used page and the tail the most recently used. The method takes the request sequence and the cache size and returns the total number of page faults.

```
int lruFaults(int[] requests, int k) {
  var cache = new LinkedList<Integer>();
  int faults = 0;
  for (int page : requests) {
    if (cache.remove((Integer) page)) {
      cache.addLast(page); // make newest
    } else {
      faults++;
      if (cache.size() == k)
        cache.removeFirst(); // drop oldest
      cache.addLast(page);
    }
  }
  return faults;
}
```

The core of the code is the call `cache.remove((Integer) page)`. This call does two jobs at once. If the page is in the list, it removes it and returns `true`; otherwise it returns `false`. In the `true` case the page was a hit. `addLast` puts it back at the end of the list, in the "newest" position. In the `false` case there is a page fault. The fault counter increases. If the cache is full, `removeFirst` drops the page at the head of the list, the least recently used. Finally the new page is appended at the end.

Table 8.12 traces the code with the sequence 1 2 3 1 4 2 and $k = 3$.

page	remove	Action	cache	faults
1	false	room left; append	[1]	1
2	false	room left; append	[1, 2]	2
3	false	room left; append	[1, 2, 3]	3
1	true	hit; append	[2, 3, 1]	3
4	false	full; drop the head, 2	[3, 1, 4]	4
2	false	full; drop the head, 3	[1, 4, 2]	5

Table 8.12: Trace of the call `lruFaults(new int[]{1, 2, 3, 1, 4, 2}, 3)`. `cache` is written from oldest to newest.



Figure 8.9: End of the trace: the list is [1, 4, 2]. In six requests there are 5 page faults and 1 hit.

The method returns 5 (Figure 8.9). The hit at the 4th request is important: 1 moves from the head of the list to the tail. Because of this, at the 5th request 2 is dropped, not 1. FIFO would have dropped 1 at the same point.

The cost of the code comes from the `remove` call. `LinkedList.remove(Object)` scans the list from the head and looks at no more than k items. Each request therefore takes $O(k)$, and a sequence of n requests takes $O(n \cdot k)$ time. Memory is $O(k)$. This suffices for small k. For caches of thousands of pages, $O(1)$ per request is needed.

Common mistake

Omitting the `(Integer)` cast. If you write `cache.remove(page)`, Java picks the `remove(int index)` overload because `page` is an `int`. The call deletes the item at *position* `page`, not the item whose *value* is `page`. When page 1 is requested, the second item of the list is deleted. When page 7 is requested, `IndexOutOfBoundsException` is thrown. The cast `(Integer) page`

makes Java call the `remove(Object)` overload.

8.4.8 An O(1) LRU cache

For *O(1)* per request, two structures are used together. A hash table finds in *O(1)* whether a page is in the cache and where it sits in the list. A doubly linked list keeps the pages in usage order. Because the node's position is known, removing it from the list and moving it to the end is *O(1)*. Java's `LinkedHashMap` class is exactly this combination.

```
class LruCache<K, V>
    extends LinkedHashMap<K, V> {
  private final int cap;

  LruCache(int cap) {
    super(16, 0.75f, true); // access order
    this.cap = cap;
  }

  @Override
  protected boolean removeEldestEntry(
      Map.Entry<K, V> eldest) {
    return size() > cap;    // drop when over cap
  }
}
```

The third parameter of the constructor call `super(16, 0.75f, true)` is `accessOrder`. When `true` is passed, every `get` and `put` moves the accessed item to the end of the list. The head of the list is therefore always the least recently used item. The first two parameters are the initial capacity and the load factor; their default values are given here. The method `removeEldestEntry` is called after every insertion. If it returns `true`, the class deletes the item at the head of the list on its own. Here it returns `true` when the size exceeds *cap*. Find, move, and delete are all *O(1)*; memory is *O(cap)*.

Example 8.5

Question. On a `new LruCache<Integer, String>(3)` object, `put(1)`, `put(2)`, `put(3)`, `get(1)`, `put(4)`, and `put(2)` are applied in order (the values are unimportant). What does `keySet()` print after each step?

Solution. After the three insertions the order is [1, 2, 3]. `get(1)` moves 1 to the end: [2, 3, 1]. After `put(4)` the size is 4, `removeEldestEntry` returns `true`, and the head, 2, is deleted: [3, 1, 4]. After `put(2)` the head, 3, is deleted: [1, 4, 2]. This is the same sequence as 1 2 3 1 4 2 in the earlier trace, and the final state agrees with Table 8.12. The code was run to verify this.

If the same class is constructed with `accessOrder` set to `false`, it becomes a FIFO cache. The list then keeps insertion order, and an access does not change the order. A single parameter separates the two policies: does a hit update the order or not?

8.4.9 Java code for FIFO and Belady's sequence

The Java code for FIFO is written with a queue and a set. The queue keeps the loading order. The set answers "is this page in the cache?" in *O(1)*.

```
int fifoFaults(int[] requests, int k) {
  Queue<Integer> queue = new LinkedList<>();
  Set<Integer> inCache = new HashSet<>();
  int faults = 0;
  for (int p : requests) {
    if (inCache.contains(p)) continue;  // hit
    faults++;
    if (inCache.size() == k)
      inCache.remove(queue.poll()); // oldest
    queue.offer(p); inCache.add(p);
  }
  return faults;
}
```

For each page the loop first checks the set. If the page is in the set, it is a hit, and `continue` moves to the next request. In FIFO a hit changes nothing. On a fault the counter increases. If the cache is full, `queue.poll()` takes the oldest page at the head of the queue and `inCache.remove` deletes it from the set too. Finally the new page is added both to the end of the queue and to the set. Each request is *O(1)*; memory is *O(k)*.

This time the code is traced with the sequence 1 2 3 4 1 2 5 1 2 3 4 5 and $k = 3$ (Table 8.13). This sequence is the classic example of Belady's anomaly.

p	Result	Evicted	queue	faults
1	F (compulsory)	—	[1]	1
2	F (compulsory)	—	[1, 2]	2
3	F (compulsory)	—	[1, 2, 3]	3
4	F	1	[2, 3, 4]	4
1	F	2	[3, 4, 1]	5
2	F	3	[4, 1, 2]	6
5	F	4	[1, 2, 5]	7
1	✓	—	[1, 2, 5]	7
2	✓	—	[1, 2, 5]	7
3	F	1	[2, 5, 3]	8

p	Result	Evicted	queue	faults
4	F	2	[5, 3, 4]	9
5	✓	—	[5, 3, 4]	9

Table 8.13: `fifoFaults` with 1 2 3 4 1 2 5 1 2 3 4 5, *k* = *3*. The queue is written from oldest to newest.



Figure 8.10: Final state of FIFO on Belady's sequence: queue [5, 3, 4], 9 faults.

The method returns 9 (Figure 8.10). Run with *k* = *4*, the same code returns 10. The cache grew, and the number of faults went up.

8.4.10 Belady's anomaly

Enlarging the cache should reduce the number of faults. This expectation looks so natural that it went unquestioned until the late 1960s. In 1969 Belady and his colleagues showed that FIFO can violate it. Figure 8.11 processes the same sequence with FIFO, first with 3 and then with 4 slots.

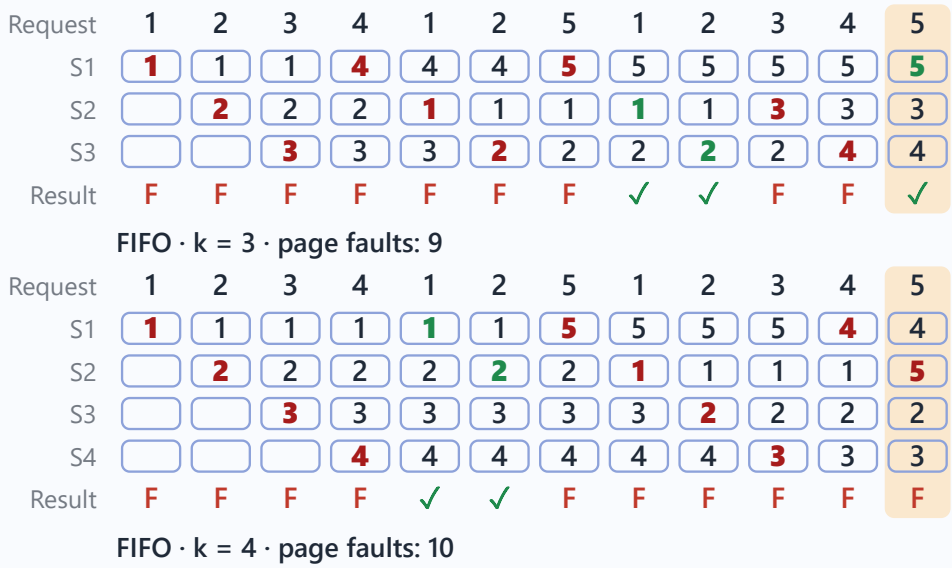


Figure 8.11: Belady's anomaly. Sequence 1 2 3 4 1 2 5 1 2 3 4 5. Top: FIFO with 3 slots, 9 faults. Bottom: FIFO with 4 slots, 10 faults.

Definition 8.5

Belady's anomaly: for a page eviction policy, the number of page faults on the same request sequence increases even though the number of slots increases.

The cause of the anomaly is visible in the 4-slot run. With 4 slots the first four requests fill the cache, and 1 and 2 get hits. When 5 arrives, FIFO evicts 1 from the head of the queue. But 1 is requested right afterwards. Loading 1 evicts 2, loading 2 evicts 3, and the chain continues. Every page is evicted just before it is requested. In the 3-slot run the loading order is different. When 5 is loaded, 1 and 2 remain in the cache, and the next two requests are hits.

This anomaly never occurs in LRU or OPT. On the same sequence LRU makes 10 faults with 3 slots and 8 with 4 slots. OPT makes 7 and 6. What the two policies have in common is that both are **stack algorithms**. In a stack algorithm, at every moment the pages in a cache of k slots are a subset of the pages in a cache of $k + 1$ slots. For LRU the reason is clear. LRU with k slots keeps the k most recently used distinct pages, and LRU with $k + 1$ slots keeps the $k + 1$ most recently used distinct pages. The first set lies inside the second. Every request that hits in the small cache also hits in the large one. So the fault count cannot increase when the number of slots grows.

FIFO is not a stack algorithm. After the 7th request, the cache is {5, 2, 3, 4} in the 4-slot run and {5, 1, 2} in the 3-slot run. Page 1 is in the small cache but not in the large one. The 8th request is for 1; the small cache gets a hit and the large cache faults.

Example 8.6

Question. On the sequence 1 2 3 4 1 2 5 1 2 3 4 5, what are the fault counts of FIFO, LRU, and OPT for $k = 1, \ldots, 5$? At which value of k does the anomaly occur?

Solution. All three policies were run in Java. For $k = 1$ all three make 12 faults: no page is requested twice in a row. For $k = 2$, FIFO and LRU make 12 faults and OPT makes 9. For $k = 3$, FIFO makes 9, LRU 10, and OPT 7. For $k = 4$, FIFO makes 10, LRU 8, and OPT 6. For $k = 5$ all five distinct pages fit, and only 5 compulsory faults remain. The fault counts of LRU and OPT never increase as k grows. FIFO's rises from 9 to 10 in going from 3 to 4. There is also an interesting detail: at $k = 3$, FIFO is better than LRU. A single result on one sequence does not determine the general ranking among policies.

8.4.11 Deterministic lower bound: k

On the example sequence, LRU was only 1.29 times worse than OPT. Does this ratio hold on every sequence? No. The adversary can construct a sequence that makes any deterministic policy k times worse.

Theorem 8.3 (Sleator and Tarjan, 1985)

No deterministic paging algorithm working with a cache of k pages has a competitive ratio smaller than k.

Proof. The adversary uses only $k + 1$ distinct pages. Because the cache holds k pages, at least one of these pages is outside the cache at every moment. Since the adversary knows the algorithm's code, it can compute at every step which page is outside. At each step it requests exactly that page. The algorithm therefore faults on *every* request: n faults in n requests.

Let OPT process the same sequence. When OPT faults, it evicts the page whose next use is farthest away. Each of the other $k - 1$ pages remaining in the cache will be requested before the evicted page. The evicted page is the only page outside the cache. So all requests up to the next request of the evicted page are hits. During this time all of the other $k - 1$ pages are requested. That is, there are at least $k - 1$ hits between two faults of OPT. OPT faults at most once in every k requests; the total number of faults is at most about n / k.

The ratio is $ALG / OPT \geq n / (n/k) = k$. As n grows, the additive constant becomes negligible; $ALG \leq c \cdot OPT + \alpha$ cannot hold with a smaller c. □

Figure 8.12 gives a concrete example of this proof for $k = 3$ and LRU. The sequence 1 2 3 4 1 2 3 4 1 2 3 4, built from 4 pages, requests exactly the page LRU has just evicted every time.

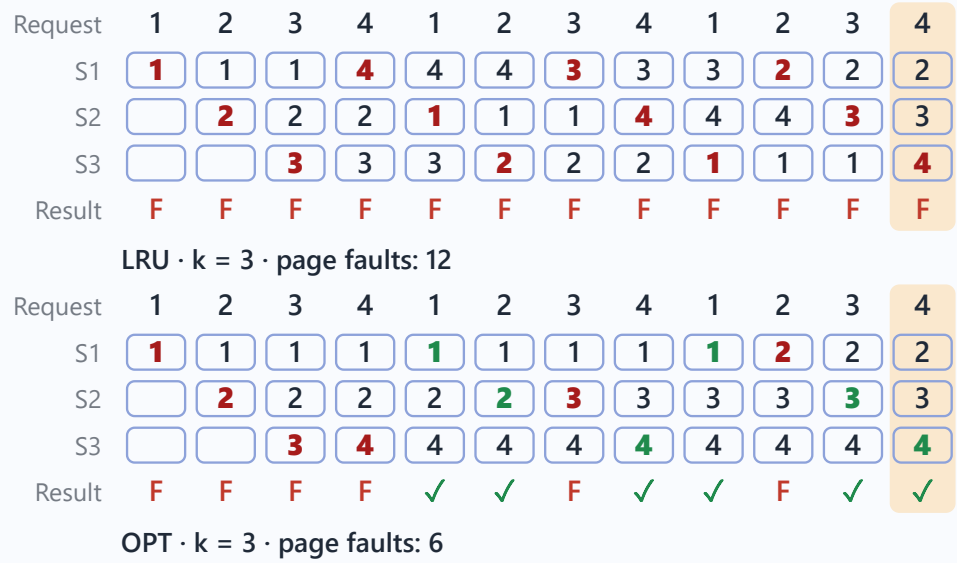


Figure 8.12: Adversary sequence 1 2 3 4 1 2 3 4 1 2 3 4, $k = 3$. Top, LRU: every request is a fault, 12 in total. Bottom, OPT: 6 faults in total.

LRU makes 12 faults in 12 requests, while OPT makes 6. If the first three compulsory faults are set aside, the ratio is 9 to 3, exactly $k = 3$. OPT behaves as in the proof. At the 4th request the cache is {1, 2, 3}, and 3, which will be requested farthest in the future, is evicted. Then 1 and 2 get hits. When 3 arrives at the 7th request, among 1, 2, and 4 in the cache, 2, which will be requested farthest in the future, is evicted. Two hits follow every fault.

FIFO also makes 12 faults on the same sequence. On this sequence FIFO and LRU behave identically, because there are no hits at all. As the sequence grows, the ratio approaches k. For example, with 5 pages and $k = 4$, if the sequence 1 2 3 4 5 is repeated three times, LRU makes 15 faults and OPT makes 7.

8.4.12 Upper bound: LRU is k-competitive

The lower bound says that no deterministic algorithm can do better than k. LRU and FIFO reach this bound. The proof divides the request sequence into phases.

Definition 8.6

The request sequence is divided from left to right into **phases**. Each phase is the longest segment containing at most k distinct pages. The first new page requested after k distinct pages have been requested in a phase, that is, the $(k + 1)$-th distinct page, starts a new phase.

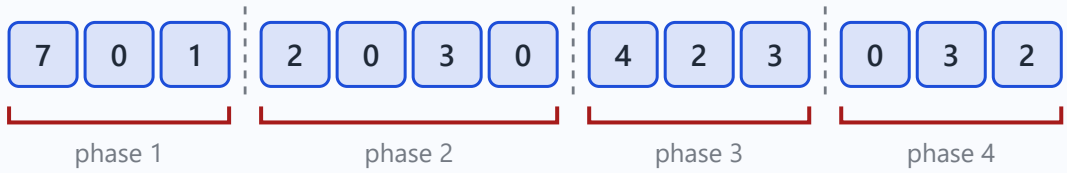


Figure 8.13: The phases of the sequence 7 0 1 2 0 3 0 4 2 3 0 3 2 for $k = 3$: 7 0 1 | 2 0 3 0 | 4 2 3 | 0 3 2.

Theorem 8.4 (LRU and FIFO are k-competitive)

For every request sequence σ, $LRU(\sigma) \le k \cdot OPT(\sigma) + k$. The same inequality holds for FIFO.

Proof. Divide the sequence into m phases. The proof has two steps.

LRU makes at most k faults in a phase. Let p be a page requested in the phase. At the moment p is requested, it is the most recently used page. Until the phase ends, at most k distinct pages are requested. So at most $k - 1$ pages can have been used more recently than p. When the cache is full and

p is in it, there is always at least one page used less recently than p. LRU therefore does not evict p before the phase ends. In a phase each distinct page faults at most once. Because a phase has at most k distinct pages, there are at most k faults.

OPT makes at least 1 fault per two consecutive phases. Consider the segment from the second request of phase i to the first request of phase $(i + 1)$. Let q be the first request of phase i. At the start of the segment q is in the cache, because it has just been requested. In the segment at least k pages different from q are requested: the remaining $k - 1$ pages of phase i and the page that starts the new phase. Apart from q, the cache has room for only $k - 1$ pages. At least one of these k pages faults within the segment. Since the segments are disjoint, over m phases OPT makes at least $m - 1$ faults.

Combining the two steps gives $LRU \le k \cdot m$ and $OPT \ge m - 1$. From these, $LRU \le k \cdot (OPT + 1) = k \cdot OPT + k$. For FIFO the first step is shown as follows. A page that faults in the phase enters at the tail of the queue. For it to reach the head and be evicted, k more new pages must be loaded behind it. Because a phase has at most k distinct pages, this cannot happen before the phase ends. The second step concerns OPT and holds unchanged. □

The proof can also be checked on the example sequence. The sequence in Figure 8.13 has 4 phases. LRU's 9 faults do not exceed $k \cdot m = 3 \cdot 4 = 12$. OPT's fault count must be at least $m - 1 = 3$; 7 faults satisfy this bound. As we see, the theory bounds only the worst case. On real sequences, thanks to locality, LRU is much closer to OPT than this bound suggests.

8.4.13 Randomized paging: marking

The key to the deterministic lower-bound proof is that the adversary knows the algorithm's cache. If the algorithm chooses the page to evict at random, an oblivious adversary cannot know which page is outside. This idea leads to the **marking algorithm**.

Definition 8.7

The **marking algorithm** keeps a mark bit for each page in the cache and works as follows:

1. At the start of a phase, the marks of all pages are cleared.
2. The requested page is marked, whether the request is a hit or a fault.
3. On a fault with a full cache, one of the unmarked pages is chosen at random and evicted.
4. If at the moment of a fault all pages in the cache are marked, a new phase begins and we return to step 1.

A marked page is a page that has been requested in this phase. The algorithm protects the pages requested in this phase and chooses only among the unmarked ones. The notion of phase is the same as in the LRU proof. Figure 8.14 shows the phases of the example sequence and the sets of pages requested in each phase.



Figure 8.14: The phases of the example sequence and the pages requested in each phase. Each phase has at most $k = 3$ distinct pages.

Fiat, Karp, Luby, McGeoch, Sleator, and Young (1991) showed that the randomized marking algorithm is $2H_k$-competitive against an oblivious adversary. Here H_k is the k-th harmonic number:

$$H_k = 1 + 1/2 + 1/3 + \ldots + 1/k \approx \ln k$$

The same work also shows that no randomized algorithm can do better than H_k. Table 8.14 compares the deterministic and randomized ratios. The gap widens as k grows: the deterministic ratio grows linearly, the randomized ratio logarithmically.

k	Deterministic (LRU, FIFO)	H_k (randomized lower bound)	$2H_k$ (marking)
2	2	1.5	3
3	3	1.83	3.67
8	8	2.72	5.44
16	16	3.38	6.76
100	100	5.19	10.37

Table 8.14: Deterministic and randomized competitive ratios for paging.

For small k, $2H_k$ can be larger than the deterministic ratio. At $k = 2$, $2H_2 = 3$ while the deterministic ratio is 2. But at $k = 8$, marking, at 5.44, beats the deterministic ratio. Because real caches hold thousands of pages, the gap is large. LRU is also a marking algorithm: the least recently used page that LRU evicts is always an unmarked page. LRU picks the oldest among the unmarked pages rather than a random one. Randomness defeats the adversary's strategy of "request the page that is not in the algorithm's cache"; the adversary cannot know which page was evicted.

8.4.14 Code trace of the marking algorithm

The Java code of the marking algorithm uses two sets: `cache` holds the pages in the cache and `marked` holds the marked pages. `rnd` is a `Random` object.

```
int marking(int[] requests, int k) {
  Set<Integer> cache = new HashSet<>();
  Set<Integer> marked = new HashSet<>();
  int faults = 0;
  for (int p : requests) {
    if (!cache.contains(p)) {
      faults++;
      if (cache.size() == k) {
        if (marked.size() == k) marked.clear(); // new phase
        List<Integer> cand = new ArrayList<>(cache);
        cand.removeAll(marked);       // unmarked
        int i = rnd.nextInt(cand.size());
        cache.remove(cand.get(i));
      }
      cache.add(p);
    }
    marked.add(p);
  }
  return faults;
}
```

If the page is not in the cache, the fault counter increases. If the cache is full, the phase is checked first: if the number of marked pages equals *k*, every page in the cache is marked. A new phase starts and `marked.clear()` erases all marks. Then the list `cand` is built: the marked pages are removed from the pages in the cache. `rnd.nextInt(cand.size())` picks a random index from this list, and that page is evicted from the cache. The new page is added to the cache. The last line, `marked.add(p)`, marks the requested page whether it was a hit or a fault.

Because the algorithm is random, a trace can be given only for particular draws. Table 8.15 gives one run on the example sequence step by step. The code was run with a fixed random number generator that returned the draws shown in this table, and it returned 9.

`p`	Result	New phase?	Candidates	Evicted	Cache	`faults`
7	F	—	—	—	7•	1
0	F	—	—	—	0• 7•	2
1	F	—	—	—	0• 1• 7•	3
2	F	yes	[0, 1, 7]	7	0 1 2•	4
0	✓	—	—	—	0• 1 2•	4
3	F	no	[1]	1	0• 2• 3•	5
0	✓	—	—	—	0• 2• 3•	5
4	F	yes	[0, 2, 3]	3	0 2 4•	6
2	✓	—	—	—	0 2• 4•	6
3	F	no	[0]	0	2• 3• 4•	7
0	F	yes	[2, 3, 4]	2	0• 3 4	8

p	Result	New phase?	Candidates	Evicted	Cache	faults
3	✓	—	—	—	0• 3• 4	8
2	F	no	[4]	4	0• 2• 3•	9

Table 8.15: `marking` on 7 0 1 2 0 3 0 4 2 3 0 3 2, $k = 3$. "Candidates" are the unmarked pages; "•" denotes a marked page.

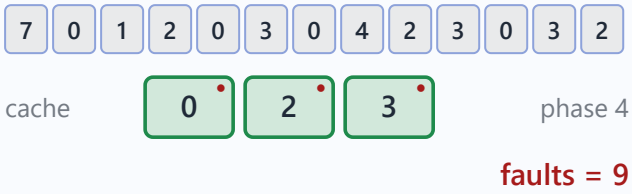


Figure 8.15: End of the trace: cache {0, 2, 3}, all three marked; phase 4, 9 faults.

In some rows of the table the choice is forced. If the candidate list has a single item, randomness changes nothing: as in rows 6, 10, and 13. Real choices are made only at the first fault of a new phase. With these draws the algorithm made 9 faults (Figure 8.15). On the same sequence LRU had made 9, FIFO 10, and OPT 7. With other draws the result changes. The code was run with 200,000 different seeds. The fault count varied between 7 and 12; the average came out at about 9.0.

Common mistake

Setting marks only on a fault. If the line `marked.add(p)` is moved inside the `if` block, pages that get hits are not marked. A page just used in this phase stays unmarked and can be evicted by the random choice. The algorithm loses its property of protecting recently used pages. In the trace above, after the hit at row 5, 0 would remain unmarked. At row 6 the candidate list would be [0, 1] and 0 could be evicted; 0 is requested again right after.

8.5 List update

In paging the cost was binary: hit or fault. In the list update problem, the cost depends on the item's position in the list. The problem rests on an old but still used idea: if frequently requested items are kept near the front of the list, searches become cheaper. Since it is not known in advance which items will be requested often, the problem is online.

8.5.1 The problem and three rules

Let a linked list of n items be given. To access an item, the list is scanned from the front. Accessing the item at position i costs i. After the access, the accessed item may be moved toward the front of the list for free. Exchanging two adjacent items in any other way costs 1 unit; such swaps are called *paid*

exchanges.

Definition 8.8

The **list update problem**: given a list of n items and a sequence of access requests. Accessing the item at position i costs i units. The accessed item may be moved to the front for free; other swaps cost one unit each. After each access the algorithm decides how to rearrange the list without knowing the future. The goal is to minimize the total cost.

There are three natural rules.

- **Move-to-front** (MTF): the accessed item is brought to the front of the list.
- **Transpose**: the accessed item is swapped with the item in front of it.
- **Frequency count**: an access count is kept for each item, and the list is kept in decreasing order of these counts.

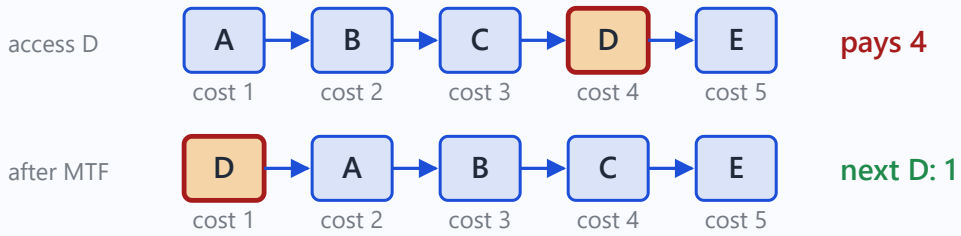


Figure 8.16: Accessing D in the list A B C D E costs 4 units. With move-to-front, D is moved to the front and the list becomes D A B C E. The next access to D costs only 1 unit.

Lists that follow these rules are called **self-organizing lists**. Their uses are wide. In a chained hash table the chain of each bucket is a short linked list, and move-to-front brings frequently searched keys to the head of the chain. Data compression uses the *move-to-front transform*: each symbol is encoded by its position in the current list and then moved to the front. The bzip2 compressor uses this transform. Consecutive repeats turn into small numbers, mostly 0, which compress well in the next stage. For example, the word "banana" with the alphabet a, b, n is encoded as 1, 1, 2, 1, 1, 1.

The Java code of the move-to-front rule is short with a linked list:

```
int mtfCost(LinkedList<Character> list, char[] requests) {
  int total = 0;
  for (char x : requests) {
    int pos = list.indexOf(x) + 1;  // 1-based
    total += pos;
    list.remove((Character) x);
    list.addFirst(x);  // move to front
  }
  return total;
}
```

`indexOf` finds the item's 0-based position; adding 1 gives the cost. `remove((Character) x)` removes the item from the list and `addFirst` puts it at the front. The `(Character)` cast here too makes Java call the `remove(Object)` overload instead of `remove(int index)`. Because `indexOf` and `remove` scan the list, each request takes *O(n)*. This time is consistent with the problem's own cost model: an access already costs as much as the position.

8.5.2 Move-to-front step by step

Let the initial list be A B C D E and the requests E D E A D E. Table 8.16 gives every step of move-to-front.

Request	Position (cost)	Total	List after
E	5	5	E A B C D
D	5	10	D E A B C
E	2	12	E D A B C
A	3	15	A E D B C
D	3	18	D A E B C
E	3	21	E D A B C

Table 8.16: Move-to-front on the list A B C D E with requests E D E A D E.

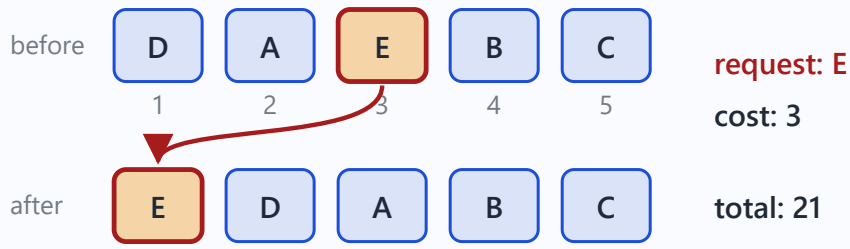


Figure 8.17: Last request: E is at position 3, cost 3. The list becomes E D A B C; total 21.

On the same sequence the other rules pay more (Table 8.17). The list that never moves anything pays 5 for every E and 4 for every D. Transpose does very badly on this sequence. E and D keep trading places at the end of the list: when E moves forward D drops back, and when D moves forward E drops back. Both are always requested from position 5.

Strategy	E	D	E	A	D	E	Total
Move-to-front	5	5	2	3	3	3	21
No moves	5	4	5	1	4	5	24
Transpose	5	5	5	1	5	5	26

Table 8.17: Cost per request of three strategies on the same request sequence.

An algorithm that knows the future pays even less. By trying all lists and paid exchanges, the best cost on this small example was computed: 17. This cost is reached as follows: at the start the list is turned into E D A B C with 7 paid exchanges. Then the accesses cost $1 + 2 + 1 + 3 + 2 + 1 = 10$.

Example 8.7

Question. On the list A B C D, what is the total cost of the requests D C D A under move-to-front?

Solution. D is at position 4: cost 4, list D A B C. C is at position 4: cost 4, list C D A B. D is at position 2: cost 2, list D C A B. A is at position 3: cost 3, list A D C B. The total is $4 + 4 + 2 + 3 = 13$. The list that never moves would have paid $4 + 3 + 4 + 1 = 12$. Move-to-front is not better than not moving on every sequence; on this one it is slightly worse.

8.5.3 Move-to-front is 2-competitive

Sleator and Tarjan showed in 1985 that the competitive ratio of move-to-front is 2. The same paper contains the result k for paging. Together they are regarded as the starting point of competitive analysis.

Theorem 8.5 (Sleator and Tarjan, 1985)

For every algorithm A starting from the same list and every sequence σ of m requests, $MTF(\sigma) \le 2 \cdot A(\sigma) - m$. A's cost includes its paid exchanges. In particular, $MTF(\sigma) \le 2 \cdot OPT(\sigma) - m$.

The proof rests on **amortized analysis**. The actual cost of an operation may fluctuate. Expensive operations are paid for by the "credit" that cheap operations accumulated. Credit is measured by a **potential function** Φ. The amortized cost of an operation is its actual cost plus the change in potential: $\text{actual} + \Delta\Phi$. If the potential is never negative and starts at 0, the sum of amortized costs cannot be smaller than the sum of actual costs.

Proof. The potential is the number of **inversions** between the MTF list and A's list. An inversion is a pair of items that appear in different orders in the two lists. At the start the two lists are identical and $\Phi = 0$.

Let the requested item be x. Split the items that precede x in the MTF list into two groups. Let a be the number of those that also precede x in A's list, and b the number of those that come after x in A's list. MTF's actual cost is $a + b + 1$. If x's position in A is i, then $i \geq a + 1$.

MTF moves x to the front. The inversions between the b items and x disappear. New inversions arise between the a items and x. Hence $\Delta\Phi \leq a - b$. The amortized cost is $(a + b + 1) + (a - b) = 2a + 1 \leq 2i - 1$.

A's own moves also change the potential. If A moves x to the front for free, x is already at the front of the MTF list; each such step destroys one inversion. A paid exchange raises the potential by at most 1 and costs A 1 unit. Summing over all requests, $MTF(\sigma) \leq \Sigma(2i - 1) + P = 2C - m + P$. Here C is A's access cost and P is the number of paid exchanges. Since $2C + P \leq 2(C + P) = 2 \cdot A(\sigma)$, the result follows. □

The inequality in the proof can be followed step by step on the example sequence. Table 8.18 takes as A the list A B C D E that never moves anything. At every step the amortized cost does not exceed $2i - 1$. In total, $21 \leq 2 \cdot 24 - 6 = 42$. The bound also holds against the true OPT: $21 \leq 2 \cdot 17 - 6 = 28$.

Request	MTF cost	i in the fixed list	Φ before → after	Amortized	$2i - 1$
E	5	5	0 → 4	9	9
D	5	4	4 → 6	7	7
E	2	5	6 → 7	3	9
A	3	1	7 → 5	1	1
D	3	4	5 → 5	3	7
E	3	5	5 → 7	5	9

Table 8.18: Move-to-front with the potential function; the list compared against is the fixed A B C D E. Φ is the number of inversions between the two lists.

Row 4 shows the power of the proof. Accessing A costs MTF 3 units but the fixed list only 1. By actual cost, MTF is 3 times worse at this step. But moving A to the front destroys two inversions and the potential drops by 2. The amortized cost falls to 1. The credit accumulated in earlier steps pays for this expensive step.

8.5.4 Why are transpose and frequency count not competitive?

The transpose rule looks cautious because it moves an item only one step. But the adversary exploits this caution. It requests the last two items of the list in alternation: $x_n, x_{n-1}, x_n, x_{n-1}, \ldots$ Each access swaps the two items, but both stay in positions n and $(n - 1)$. The requested item is at position n every time. The cost per request is n. MTF, in contrast, keeps the two items at the front of the list after the

first two accesses and pays 2 per request. This sequence was run with $n = 5$ and 20 requests: transpose paid 100, MTF 46, and the list that never moves 90. As n grows the ratio goes to $n/2$; it is unbounded.

Frequency count is also trapped. First item x_1 is requested n times, item x_2 $n - 1$ times, and so on. Then the same items are requested in reverse order, in larger numbers. The counters keep the old order for a long time. The new frequent items stay at the end of the list until the counters catch up. On the list A B C D, A was requested four times, B three, C two, and D once, followed by D five, C six, B seven, and A nine times. Frequency count paid 81 and MTF paid 49. On long lists the cost can be worse by a factor of $\Theta(n)$.

Common mistake

Reading "2-competitive" as "it pays less than twice on every sequence, so it is always good". The theorem compares MTF with the algorithm that knows the future. On some sequences MTF is worse even than the list that never moves anything. For example, on the list A B C D with the requests C A C D B A, MTF pays 19 and the non-moving list pays 14. The guarantee is only this: on no sequence does it exceed twice the best algorithm.

8.6 Data streams

In the earlier problems the online algorithm's trouble was making wrong decisions. In data streams the trouble is different: the input is so large that it cannot be stored. The algorithm looks at each item once and keeps a small summary. A question can be asked at any moment, and the answer must be given from the summary alone.

8.6.1 The data stream model

Definition 8.9

In the **data stream** model, items $x_1, x_2, \ldots$ arrive one at a time. The stream may be very long or infinite. Each item is looked at once (single pass), and the past cannot be stored in full. At every moment the answer to a question such as "what is the average, median, or sample so far?" must be ready.

Table 8.19 gives the problems treated in this section, the summary stored for each, and the costs.

Problem	Stored summary	Per item	Memory
Moving average (last k)	Queue and sum	$O(1)$	$O(k)$
Online median	Max-heap and min-heap	$O(\log n)$	$O(n)$

Problem	Stored summary	Per item	Memory
Reservoir sampling	Sample of k items	$O(1)$	$O(k)$
Pattern search in a stream	Last m characters and a rolling hash	$O(1)$	$O(m)$
Running mean	n and μ	$O(1)$	$O(1)$

Table 8.19: Stream problems and their summaries.

The online median is the exception in the table: its memory grows with the stream. For an exact median this is unavoidable. Every item in the stream is a candidate for becoming the median later. For very large streams, summaries that give an approximate median are therefore used.

8.6.2 Moving average

The **moving average** is the average of the last k items in the stream. It is used to smooth noisy sensor data. The 50-day average in the stock market is also a moving average. The naive method re-sums the last k items at every new item: $O(k)$. The better method is the **sliding window**. The sum is kept in a variable. When a new item enters, it is added to the sum, and when the oldest item leaves the window, it is subtracted:

$$sum \leftarrow sum - x_{t-k} + x_t, \quad average = sum / min(t, k)$$

```
class MovingAverage {
  private final int k;
  private final Queue<Integer> window
      = new LinkedList<>();
  private int sum = 0;

  MovingAverage(int k) { this.k = k; }

  double next(int value) {
    if (window.size() == k)
      sum -= window.poll();
    window.offer(value);
    sum += value;
    return sum / (double) window.size();
  }
}
```

The window is a queue. The `next` method first checks whether the window is full. If it is, `poll` takes the oldest item from the queue and subtracts it from the sum. Then the new item is added to the queue and to the sum. The average is the sum divided by the window size. If the window is not yet full, the divisor is the current number of items rather than k. The `(double)` cast prevents integer division. Each call is $O(1)$; memory is $O(k)$. Table 8.20 traces the stream 4, 8, 6, 2, 10 with $k = 3$.

`value`	Evicted	`window`	`sum`	Returned average
4	—	[4]	4	*4 / 1 = 4*
8	—	[4, 8]	12	*12 / 2 = 6*
6	—	[4, 8, 6]	18	*18 / 3 = 6*
2	4	[8, 6, 2]	*18 − 4 + 2 = 16*	*16 / 3 ≈ 5.33*
10	8	[6, 2, 10]	*16 − 8 + 10 = 18*	*18 / 3 = 6*

Table 8.20: The stream 4, 8, 6, 2, 10 with `new MovingAverage(3)`.

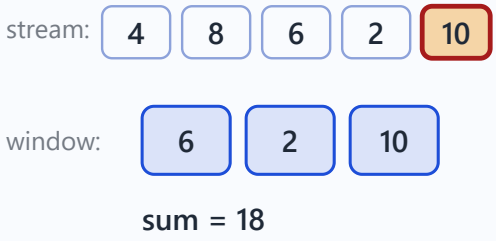


Figure 8.18: Last step: window [6, 2, 10], sum 18, average 6.

Common mistake

Mixing up the order of removing and adding. If we add first and then check `window.size() > k`, the code still works correctly. But if we add first and then check `size() == k`, the window is shrunk before it ever reaches k. Then only the average of the last $k - 1$ items is taken. Another mistake is keeping `sum` as a `double`. On long streams, adding and subtracting accumulates rounding error; for integer data the sum should be kept as an integer.

8.6.3 Online median

After every new item, the **median** of the items so far is requested. Sorting each time takes $O(n \log n)$, and inserting into a sorted array takes $O(n)$. With two heaps, insertion drops to $O(\log n)$ and reading the median to $O(1)$. This method is called the **running median**.

The idea is as follows. The items are split into two halves. The smaller half is kept in a max-heap, called `left`. The larger half is kept in a min-heap, called `right`. The median lies at the boundary of the two halves. The items at this boundary are exactly the heap tops. Two invariants are maintained:

- Every item in `left` is less than or equal to every item in `right`.
- The sizes are either equal or `left` has one more: $|left| = |right|$ or $|left| = |right| + 1$.

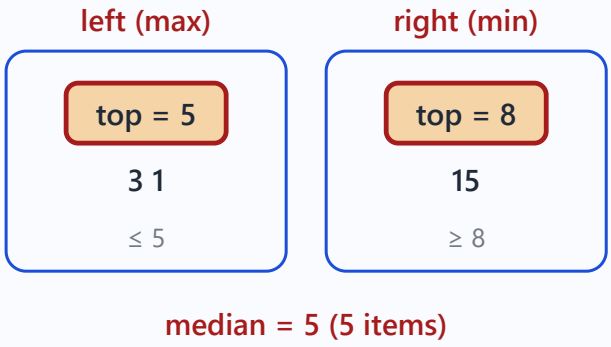


Figure 8.19: 5 items: `left` (max-heap) holds 5, 3, 1; `right` (min-heap) holds 8, 15. Because the item count is odd, the median is the top of `left` : 5.

If the item count is odd, the median is the top of `left`. If it is even, it is the average of the two tops. In Java, `PriorityQueue` is a min-heap by default; a max-heap is built with a reversed comparator:

```
PriorityQueue<Integer> left = new PriorityQueue<>((a, b) -> b - a); // max-heap
PriorityQueue<Integer> right = new PriorityQueue<>();               // min-heap
```

```
void add(int x) {
  if (left.isEmpty() || x <= left.peek())
    left.offer(x);          // smaller half
  else
    right.offer(x);         // larger half
  if (left.size() > right.size() + 1)
    right.offer(left.poll());
  else if (right.size() > left.size())
    left.offer(right.poll());
}
double median() {
  if (left.size() == right.size())
    return (left.peek() + right.peek()) / 2.0;
  return left.peek();
}
```

`add` works in two stages. First the item is put in the correct half: if `left` is empty or the item is not greater than the top of `left`, it goes to the smaller half; otherwise to the larger half. This step preserves the first invariant. Then the balance is restored. If `left` has become two larger, its top is moved to `right`. If `right` has become larger, its top is moved to `left`. The moved item is always an item at the boundary of one half, so the move does not break the first invariant. Each insertion does at most three heap operations: *O(log n)*. `median` looks only at the tops: *O(1)*. Table 8.21 traces the stream 5, 15, 1, 3, 8, 7.

x	Half entered	Rebalance	left	right	Median
5	left (empty)	ok	5	—	5
15	right (15 > 5)	ok	5	15	*(5 + 15)/2 = 10*
1	left (1 ≤ 5)	ok	5, 1	15	5
3	left (3 ≤ 5)	3 > 1 + 1: 5 → right	3, 1	5, 15	*(3 + 5)/2 = 4*
8	right (8 > 3)	3 > 2: 5 → left	5, 3, 1	8, 15	5
7	right (7 > 5)	ok	5, 3, 1	7, 8, 15	*(5 + 7)/2 = 6*

Table 8.21: Online median on 5, 15, 1, 3, 8, 7. Heap contents are written top first.



Figure 8.20: End of the trace: left 5, 3, 1; right 7, 8, 15; median 6.

The fourth and fifth steps show why the rebalancing is needed. After 3 is added, left holds 3 items and right holds 1. The median is no longer the top of left. Moving 5 to right evens the halves out again. At the fifth step right becomes the larger one this time, and 5 moves back. The same item has been moved twice; this is normal. When the code was run, the medians came out as 5.0, 10.0, 5.0, 4.0, 5.0, and 6.0 in order.

Example 8.8

Question. What are the online medians for the stream 2, 9, 4, 7?

Solution. 2 goes into left : median 2. Because 9 > 2, it goes into right : median *(2 + 9)/2 = 5.5*. Because 4 > 2, it goes into right . Now right is larger, and its top, 4, is moved to left . left holds 4, 2; right holds 9; median 4. Because 7 > 4, it goes into right . The sizes are 2 and 2; median *(4 + 7)/2 = 5.5*. The medians are 2, 5.5, 4, 5.5. This can be checked with sorted lists: [2, 9], [2, 4, 9], and [2, 4, 7, 9].

Common mistake

Forgetting the rebalancing. Putting the item in the correct half is not enough by itself. If the stream is 5, 1, 3, all three items go into left . Without the rebalancing, median() returns the top of left , which is 5; the correct answer is 3. When the size difference exceeds 1, the top

must be moved to the other heap. A second mistake is to build the max-heap with `(a, b) -> b - a`. With very large and very small numbers this subtraction overflows. The safe choice is `Comparator.reverseOrder()`.

8.6.4 Reservoir sampling

We want to keep a random sample of k items from a stream of unknown length. Every item must be in the sample with equal probability. If the stream length n were known, the job would be easy: pick k indices from $1, \ldots, n$. But n is unknown, and the items cannot be stored. **Reservoir sampling** solves this problem with $O(k)$ memory.

Definition 8.10

Algorithm R (Vitter, 1985): the first k items are put into the reservoir. Then, for each 0-based item $i \geq k$, a uniformly random j in the range $0 \ldots i$ is chosen. If $j < k$, x_i overwrites the j-th item of the reservoir; otherwise x_i is skipped.

According to the algorithm, item i enters the reservoir with probability $k / (i + 1)$. If it enters, it evicts the item at a random position. Late-arriving items have a small probability of entering. But early items also face a risk of being evicted that lasts a long time. The two effects balance exactly.

Theorem 8.6 (Algorithm R is uniform)

When a stream of $n \geq k$ items ends, each item is in the reservoir with probability k / n.

Proof. Consider an item x_i with $i \geq k$. It enters with probability $k/(i + 1)$. At each later step $t > i$, item x_t enters with probability $k/(t + 1)$ and, if it enters, picks one of the k positions with equal probability. The probability that x_i is evicted at this step is $(k/(t + 1)) \cdot (1/k) = 1/(t + 1)$. The probability that it stays is $t/(t + 1)$. Because the steps are independent, the probabilities multiply:

$$k/(i + 1) \cdot (i + 1)/(i + 2) \cdot (i + 2)/(i + 3) \cdot \ldots \cdot (n - 1)/n = k / n$$

In the product each numerator cancels the previous denominator, leaving k/n. For the first k items, with $i < k$, the probability of entering is 1. The product of staying is $k/(k + 1) \cdot \ldots \cdot (n - 1)/n = k/n$. The result is the same for every item. □



Figure 8.21: *k = 2, n = 5*, 0-based item 2: probability of entering 2/3, staying 3/4 and 4/5 in the next two steps. The product is *2/5 = k/n*.

```
List<Integer> reservoir(int[] stream, int k) {
  List<Integer> r = new ArrayList<>();
  for (int i = 0; i < Math.min(k, stream.length); i++)
    r.add(stream[i]);
  for (int i = k; i < stream.length; i++) {
    int j = rnd.nextInt(i + 1); // 0..i
    if (j < k)
      r.set(j, stream[i]);
  }
  return r;
}
```

The code works with a list. The first loop puts the first *k* items into the reservoir; if the stream is shorter than *k*, it puts in all items and the list is returned as is, because the second loop never runs. The second loop calls `rnd.nextInt(i + 1)` for each remaining item. This call returns a uniform integer in the range *0 … i*, endpoints included. If `j < k`, `r.set` overwrites the item at that position. It uses *O(1)* work per item and *O(k)* memory in total. On a real stream one works with an iterator instead of an array; the algorithm does not change.

Table 8.22 traces the code with the stream 10, 20, 30, 40, 50, 60 and *k = 3*. The random numbers are fixed for this trace: 1, 4, and 0.

`i`	Value	`j`	Action	`r`	Entry probability
0–2	10, 20, 30	—	added directly	[10, 20, 30]	1
3	40	1	`r[1] = 40` (20 is evicted)	[10, 40, 30]	3/4
4	50	4	*4 ≥ 3*: does not enter	[10, 40, 30]	3/5
5	60	0	`r[0] = 60` (10 is evicted)	[60, 40, 30]	3/6

Table 8.22: `reservoir` on 10, 20, 30, 40, 50, 60 with *k = 3*.

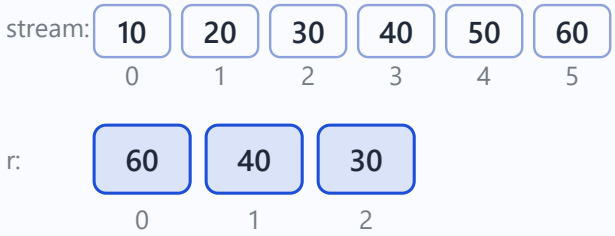


Figure 8.22: The stream has ended: the sample is [60, 40, 30].

Other values of `j` give another sample. But according to the theorem every item is in the sample with probability *3/6 = 1/2*. This claim was also checked by simulation. With the stream 1, ..., 10 and $k = 3$, the code was run 300,000 times. The frequency of each item appearing in the sample came out between 0.299 and 0.301; the expected value is *3/10 = 0.3*.

Example 8.9

Question. Let $k = 2$. With what probability does the 0-based item 9 of the stream enter the reservoir? If the stream ends with 10 items, with what probability is this item in the reservoir at the end?

Solution. The probability of entering is $k/(i + 1) = 2/10 = 1/5$. Because the stream ends with 10 items, item 9 is the last one and there is no risk of eviction after it. The probability of being in the reservoir at the end is also *1/5*. This equals $k/n = 2/10$ in the theorem.

Common mistake

Choosing the wrong range. If you write `nextInt(i)`, `j` comes from the range $0 \ldots i - 1$ and the item enters with probability k/i. The sample is not uniform. In particular at $i = k$ the probability rises to 1: the k-th item always enters. The faulty code was run 300,000 times with the stream 1, ..., 10 and $k = 3$. The frequency of the first three items dropped to 0.22, while that of the other seven rose to 0.33.

8.6.5 Pattern search in a stream: the rolling hash

Suppose a network monitoring program is looking for a certain signature in the packets passing by. The characters arrive as a stream. If the pattern has length m, looking at only the last m characters is enough. Comparing this window with the pattern at every new character takes $O(m)$. The **rolling hash** reduces the comparison to $O(1)$ most of the time.

The window's hash value treats its characters like the digits of a number in base B. When a new character enters, the value is not computed from scratch. The outgoing character's contribution is subtracted, the remaining value is shifted by one digit, and the incoming character is added:

$$h_{new} = (h_{old} - c_{out} \cdot B^{m-1}) \cdot B + c_{in}$$

```
void search(String stream, String pattern) {
  int m = pattern.length(), B = 101, h = 0, hp = 0;
  int pow = 1;                          // B^(m-1)
  for (int i = 0; i < m - 1; i++) pow *= B;
  for (int i = 0; i < m; i++)
    hp = hp * B + pattern.charAt(i);
  for (int t = 0; t < stream.length(); t++) {
    char c = stream.charAt(t);
    if (t >= m) h -= stream.charAt(t - m) * pow;
    h = h * B + c;          // add incoming
    if (t >= m - 1 && h == hp
        && stream.startsWith(pattern, t - m + 1))
      System.out.println("match: " + (t - m + 1));
  }
}
```

The code first computes B^{m-1} into the variable `pow` and the pattern's hash value into `hp`. The main loop runs for each character. If `t >= m`, a character leaves the window and its contribution is subtracted as `stream.charAt(t - m) * pow`. Then `h` is shifted by one digit and the incoming character is added. Once the window is full, if `h == hp`, the characters are also compared with `startsWith`. Two different strings can give the same hash value; this is called a collision. For this reason hash equality alone is not enough. If the hashes differ, the characters are never looked at. If collisions are rare, the work per item is *O(1)*; memory is *O(m)* for the window.

Table 8.23 searches for the pattern "AB" in the stream "CABDAB". $B = 101$, $m = 2$, and `pow` = 101. The character codes are A = 65, B = 66, C = 67, D = 68. The pattern's hash value is $65 \cdot 101 + 66 = 6631$.

`t`	Outgoing	Incoming	Window	`h`	Result
0	—	C	“C”	67	window not full
1	—	A	“CA”	$67 \cdot 101 + 65 = 6832$	≠ 6631
2	C	B	“AB”	$(6832 - 6767) \cdot 101 + 66 = 6631$	match: 1
3	A	D	“BD”	6734	≠ 6631
4	B	A	“DA”	6933	≠ 6631
5	D	B	“AB”	$(6933 - 6868) \cdot 101 + 66 = 6631$	match: 4

Table 8.23: Rolling hash in the call `search("CABDAB", "AB")`.

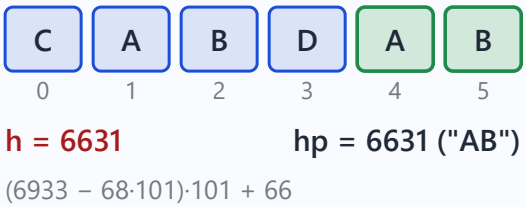


Figure 8.23: Last step: outgoing D, incoming B; $h = 6631 = hp$. The second match is at position 4.

When the code was run, it printed "match: 1" and "match: 4". As the pattern gets longer, `int` overflows. This does not harm correctness. In Java, `int` operations are done modulo 2^{32}, and subtraction, multiplication, and addition are consistent in this modulus. Because `h` and `hp` are computed in the same modulus, equal windows give equal hashes. Overflow can only increase collisions; the character comparison filters them out. In practice one usually works modulo a large prime.

8.7 Incremental learning

In machine learning the model is usually trained once on all the data. If data arrives continuously, this approach is expensive. Retraining the model from scratch at every new piece of data means reprocessing all the data that has arrived so far. The online view helps here too: the model is updated with the new data only.

8.7.1 Updating a model with new data

Definition 8.11

Incremental learning is the updating of a model with each new piece of data without reprocessing the past data. The updated model represents the knowledge in both the old and the new data.

Figure 8.24 shows the process. Initially the model is trained on the original data set, giving MODEL 1. When new data 1 arrives, the model is updated with this data alone and becomes MODEL 2. MODEL 2 carries the knowledge of two data sets, but the original data has not been reprocessed. The same step is repeated with new data 2. The model is up to date at every moment, and the cost of an update is only the size of the new data.

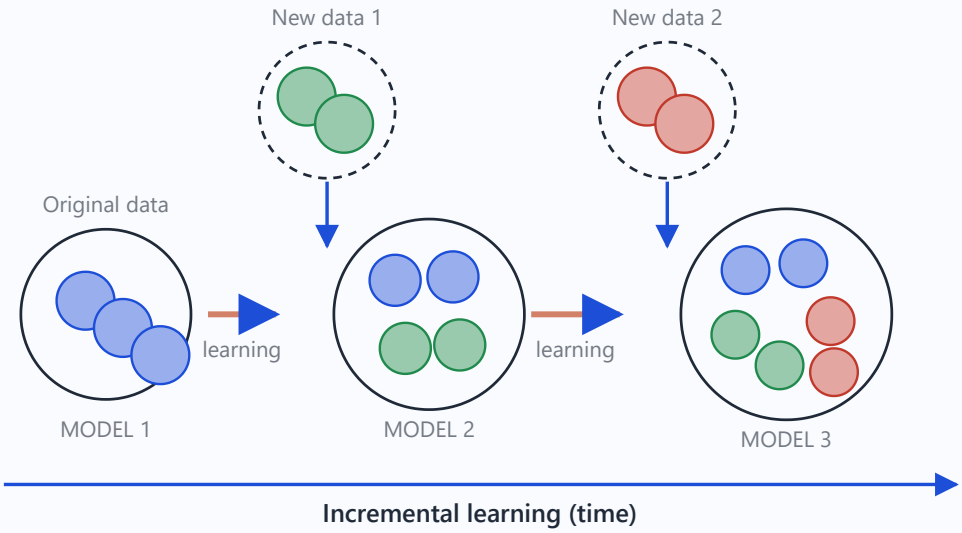


Figure 8.24: Incremental learning. The model is updated with each new data set; MODEL 3 carries the knowledge of three data sets.

The similarity to stream algorithms is clear. The model is a summary of the past data, just like the sum in the moving average or the two heaps in the online median. The difference is that the summary is more complex: the weights of a classifier, the branches of a decision tree, or the parameters of a neural network.

8.7.2 Running mean and online gradient descent

The simplest example of an incremental update is the **running mean**. The mean of all items in the stream can be updated without storing the past:

$$\mu_n = \mu_{n-1} + (x_n - \mu_{n-1}) / n$$

The formula corrects by $1/n$ of the difference between the new item and the old mean. For the stream 4, 8, 6 we get $\mu_1 = 4$, $\mu_2 = 4 + (8 - 4)/2 = 6$, $\mu_3 = 6 + (6 - 6)/3 = 6$. This equals $(4 + 8 + 6)/3 = 6$. Only n and μ are stored; memory is $O(1)$. Because the formula does not keep a separate sum, it also causes no trouble with large numbers on long streams.

The same idea carries over to model training as **online gradient descent**. The model is defined by a weight vector w. When each example (x_t, y_t) arrives, the weight takes a small step in the direction opposite to the gradient of the loss:

$$w \leftarrow w - \eta \cdot \nabla \ell(w; x_t, y_t)$$

Here ℓ is the loss function and η is the learning rate. Each example is used once and discarded.

Example 8.10

Question. Let the model be $\hat{y} = w \cdot x$ and the loss $\ell = (w \cdot x - y)^2 / 2$. The derivative of this loss with respect to w is $(w \cdot x - y) \cdot x$. Initially $w = 0$ and the learning rate is $\eta = 0.1$. The examples (1, 2), (2, 4), (3, 6) arrive in order. What is w after each example?

Solution. On the first example the prediction is 0, the error is $0 - 2 = -2$, and the gradient is $-2 \cdot 1 = -2$. $w = 0 - 0.1 \cdot (-2) = 0.2$. On the second example the prediction is $0.2 \cdot 2 = 0.4$, the error is -3.6, and the gradient is $-3.6 \cdot 2 = -7.2$. $w = 0.2 + 0.72 = 0.92$. On the third example the prediction is 2.76, the error is -3.24, and the gradient is -9.72. $w = 0.92 + 0.972 = 1.892$. The true relationship in the data is $y = 2x$, and w has approached 2 in three examples. If a fourth example, (1, 2), arrived again, w would be 1.9028.

8.7.3 Difficulties

Incremental learning gives a cheap and up-to-date model, but it has problems of its own.

- **Catastrophic forgetting**: while adapting to new data, the model loses old knowledge. A neural network learning a new task may suddenly lose its performance on the old task.
- **Concept drift**: the distribution of the data changes over time. Last year's shopping habits may not represent this year well. In this case old data becomes misleading.
- **Order effects**: if the same data arrives in a different order, the model takes a different shape. In offline training the data can be shuffled; in a stream the order comes with the data.
- **Noise**: a single outlier can corrupt the model. A small learning rate and a sliding window reduce this effect.

Forgetting and concept drift pull in opposite directions. When there is concept drift, some forgetting of old data is desirable. When there is none, forgetting is a loss. The window size in a moving average or the learning rate are the knobs that tune this balance.

8.8 Problems and results

Table 8.24 gives the problems of the chapter and their results together. The numbers in the first five rows are competitive ratios. The rows below give the time per item.

Problem	Online algorithm	Result	Note
Ski rental	Rent for $B - 1$ days, then buy	$2 - 1/B$	best among deterministic; randomized $e/(e - 1)$
Paging	LRU, FIFO, FWF	k	best among deterministic
Paging	LFU, LIFO	unbounded	not competitive

Problem	Online algorithm	Result	Note
Paging (randomized)	Marking	$2H_k$	lower bound H_k
List update	Move-to-front	2	transpose and frequency count are not competitive
Moving average	Queue and sum	$O(1)$	memory $O(k)$
Online median	Two heaps	$O(\log n)$	median read $O(1)$
Reservoir sampling	Algorithm R	$O(1)$	each item with probability k/n

Table 8.24: Online problems, algorithms, and results.

The proofs for the three competitive problems in the table follow the same skeleton. For the upper bound, the algorithm's cost is bounded by a multiple of OPT. In ski rental this is done with a case analysis, in paging with phases, and in list update with a potential function. For the lower bound, the adversary knows the deterministic algorithm's next decision and generates the request that hurts it most. In all three problems randomization lowers the ratio, because it takes this knowledge away from the adversary.

8.9 Common mistakes

- **Thinking "online = internet".** An online algorithm is a model in which the input is revealed over time. It has nothing to do with a network connection.
- **Confusing the ratio with average-case behavior.** The competitive ratio refers to the worst input. "It usually works well" is not a competitive ratio.
- **Confusing FIFO with LRU.** In FIFO a hit does not change the order; in LRU a hit makes the page the newest. This is the only difference between the two policies.
- **Forgetting compulsory faults.** The first k distinct pages fault in every policy. Even OPT cannot make zero faults.
- **Thinking OPT evicts the "least used page".** OPT evicts the page whose next use is farthest away; it does not look at the number of uses.
- **Thinking a larger cache is always better.** In FIFO, faults can increase as the number of slots grows: Belady's anomaly. This does not happen in LRU and OPT.
- **Wrong range in the reservoir.** j must be chosen from the range $0 \dots i$, including i. Writing `nextInt(i)` breaks the probabilities.
- **Forgetting the balance in the median.** Putting the item in the right heap is not enough. When the size difference exceeds 1, the top must be moved to the other heap.
- **Counting hash equality as a match.** In a rolling hash, equal values can arise from collisions. The characters must be compared separately.

8.10 Summary

- **Model:** the input arrives piece by piece as a request sequence. The decision is made the moment the request arrives and cannot be undone. An offline algorithm, by contrast, sees the whole input from the start.
- **Competitive ratio:** if $ALG(\sigma) \le c \cdot OPT(\sigma) + \alpha$ holds for every σ, the algorithm is c-competitive. The analysis measures the worst case. Lower bounds are built with the adversary model.
- **Ski rental:** $OPT = min(d, B)$. The break-even strategy rents for $B - 1$ days and buys on day B; its ratio is $2 - 1/B$. No deterministic strategy is better; a randomized strategy comes down to $e/(e - 1) \approx 1.58$.
- **Paging:** FIFO evicts the page loaded earliest, LRU the least recently used, and OPT the one whose next use is farthest away. On the example sequence: 10, 9, and 7 faults. LRU and FIFO are k-competitive, and nothing better exists among deterministic algorithms. Randomized marking comes down to $2H_k$.
- **Belady's anomaly:** in FIFO, faults can increase as the number of slots grows (9 → 10). LRU and OPT are stack algorithms; they show no anomaly.
- **Implementation:** LRU with a linked list costs $O(k)$ per request, and with `LinkedHashMap` $O(1)$. FIFO is $O(1)$ with a queue and a set.
- **List update:** the cost is the position. Move-to-front satisfies $MTF \le 2 \cdot OPT - m$; the proof takes the number of inversions as the potential. Transpose and frequency count are not competitive.
- **Data streams:** single pass and little memory. The moving average runs in $O(1)$ with a sliding window, the online median in $O(\log n)$ with two heaps, and reservoir sampling in $O(1)$ per item with $O(k)$ memory. In the reservoir every item is present with probability k/n. The rolling hash updates the window in $O(1)$.
- **Incremental learning:** the model is updated with new data. The running mean and online gradient descent are the basic examples. Forgetting and concept drift are the main difficulties.

8.11 Key terms

online algorithm offline algorithm competitive analysis competitive ratio adversary

ski rental problem randomized algorithm paging cache page fault eviction policy

Belady's anomaly stack algorithm marking algorithm list update problem move-to-front (MTF)

transpose amortized analysis inversion data stream running median reservoir sampling

rolling hash incremental learning

8.12 Test Yourself

1. In ski rental, $B = 8$. What is the worst-case ratio of the strategy that buys on day 6? How different is it from that of the break-even strategy?

   Answer

   The adversary ends the season on day 6. ALG = $5 + 8 = 13$, OPT = $\min(6, 8) = 6$; ratio $13/6 \approx 2.17$. The break-even strategy's ratio is $2 - 1/8 = 15/8 = 1.875$. The early-buying strategy is worse.

2. The sequence 1 2 1 3 1 4 1 2 is processed with $k = 2$ slots. How many page faults do FIFO and LRU make?

   Answer

   FIFO makes 6 faults: when 3 arrives, 1, the earliest loaded, is evicted and is requested again right away. When 1 arrives, 2 is evicted; when 4 arrives, 3; when the final 2 arrives, 1. LRU makes 5 faults. Because 1 is the newest on every hit, it is never evicted; when 3, 4, and the final 2 arrive, 2, 3, and 4 are evicted respectively. On this sequence OPT also makes 5 faults.

3. The sequence 1 2 3 1 4 2 is processed with $k = 3$ under the OPT policy. How many faults occur? Which pages can be evicted when 4 arrives?

   Answer

   There are 4 faults: three compulsory faults and the fault on 4. When 4 arrives the cache is {1, 2, 3}. 1 and 3 are never requested again, and 2 is requested in the last request. Either 1 or 3 can be evicted; both give the same result. The 2 in the last request is a hit. On the same sequence LRU had made 5 faults.

4. On the list A B C, what is the total cost of the requests C C B A under move-to-front? What does the list that never moves pay?

   Answer

   C at position 3: 3, list C A B. C at position 1: 1. B at position 3: 3, list B C A. A at position 3: 3, list A B C. Total $3 + 1 + 3 + 3 = 10$. The non-moving list pays $3 + 3 + 2 + 1 = 9$. On this short sequence move-to-front is slightly more expensive; being 2-competitive does not mean being best on every sequence.

5. 3, 5, 9, 1 are given in order to `new MovingAverage(2)`. What values does `next` return?

> **Answer**
>
> 3.0; *(3 + 5)/2 = 4.0*; 3 leaves and *(5 + 9)/2 = 7.0*; 5 leaves and *(9 + 1)/2 = 5.0*.

6. 10, 2, 7, 4 are added in order to the online median structure. What is the median after each insertion? When 7 is added, is there a transfer between the heaps?

> **Answer**
>
> The medians are 10, 6, 7, 5.5. When 2 is added it goes into `left`, `left` holds two items, and the top, 10, moves to `right`: median *(2 + 10)/2 = 6*. When 7 is added, because *7 > 2*, it goes into `right`. `right` holds two items and `left` one. The top of `right`, 7, moves to `left`; the median is 7. 4 goes into `left`; `left` holds three items and its top, 7, moves to `right`. The sizes become 2 and 2: *(4 + 7)/2 = 5.5*.

7. In reservoir sampling with $k = 1$, with what probability does the 0-based item 4 enter the reservoir? If the stream ends with 8 items, with what probability is this item selected at the end?

> **Answer**
>
> The probability of entering is $1/5$. The probabilities of staying in the next three steps are $5/6$, $6/7$, and $7/8$. The product is $1/5 \cdot 5/6 \cdot 6/7 \cdot 7/8 = 1/8 = k/n$.

8. In a cache of $k = 6$ pages, what are the competitive ratios of deterministic LRU and the randomized marking algorithm? Which gives the better guarantee?

> **Answer**
>
> LRU's ratio is 6. $H_6 = 1 + 1/2 + \dots + 1/6 = 49/20 = 2.45$; the marking algorithm's ratio is $2H_6 = 4.9$. In this case marking is below the deterministic bound. No randomized algorithm can do better than 2.45.

9. In the call `search("ABA", "BA")` with $B = 101$, what is `hp`? What value does `h` take at `t = 2`, and what is printed?

> **Answer**
>
> `hp` $= 66 \cdot 101 + 65 = 6731$. At `t = 1` the window is "AB" and `h` = 6631. At `t = 2`, A leaves: $(6631 - 65 \cdot 101) \cdot 101 + 65 = 66 \cdot 101 + 65 = 6731$. The hashes are equal and the characters match too; "match: 1" is printed.

8.13 Further reading

- A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*: the standard book in the field. It treats paging, list update, adversary models, and randomized algorithms with detailed proofs.
- T. H. Cormen et al., *Introduction to Algorithms*, 4th edition, chapter 27: an introduction to online algorithms; it covers the elevator waiting problem (a variant of ski rental), list update, and cache problems.
- S. Albers, "Online algorithms: a survey" (2003): gives the main results of competitive analysis in a short, readable summary.
- J. S. Vitter, "Random Sampling with a Reservoir" (1985): introduces Algorithm R and the skip-ahead versions that speed it up.

8.14 References

1. A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.
2. D. D. Sleator and R. E. Tarjan, “Amortized Efficiency of List Update and Paging Rules”, *Communications of the ACM* 28(2), 1985.
3. L. A. Belady, “A Study of Replacement Algorithms for a Virtual-Storage Computer”, *IBM Systems Journal* 5(2), 1966.
4. L. A. Belady, R. A. Nelson, and G. S. Shedler, “An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine”, *Communications of the ACM* 12(6), 1969.
5. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, “Competitive Paging Algorithms”, *Journal of Algorithms* 12(4), 1991.
6. A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, “Competitive Randomized Algorithms for Nonuniform Problems”, *Algorithmica* 11(6), 1994.
7. J. S. Vitter, “Random Sampling with a Reservoir”, *ACM Transactions on Mathematical Software* 11(1), 1985.
8. S. Albers, “Online Algorithms: A Survey”, *Mathematical Programming* 97(1–2), 2003.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th edition, MIT Press, 2022 (chapter 27: Online Algorithms).

Parallel Algorithms

Even a laptop today has several processor cores. A sequential algorithm uses only one of them while the others sit idle. A parallel algorithm splits the work into pieces, solves the pieces at the same time on different processors, and combines the results. These notes first explain why parallelism is needed and what costs it brings. They then present the classification of parallel computers, the PRAM model, and the performance measures: speedup, efficiency, cost, Amdahl's law, work, and depth. The basic techniques are run step by step on small examples: tree summation, prefix sums, pipelining, and the pipelined tree. In the last part the same ideas are built in Java. We write merge sort with Fork/Join, word count with MapReduce, and matrix multiplication with row tasks. The race condition caused by shared state is shown by measurement.

LEARNING OUTCOMES

Concepts

Explain the divide, assign, solve, and combine structure of a parallel algorithm, and the gains and difficulties it brings (communication, synchronization, load balancing).

Models and measures

Recognize Flynn's classes, the memory architectures, and the PRAM model; compute speedup, efficiency, cost, work, and depth; apply Amdahl's law and Brent's theorem.

Basic techniques

Trace PRAM tree summation, parallel prefix sums, the pipeline, and the pipelined tree step by step; find the number of rounds and steps.

Parallelism in Java

Read parallel merge sort with Fork/Join, MapReduce, and data-parallel examples; recognize and prevent race conditions.

9.1 The idea of a parallel algorithm

Suppose a single teacher grades the 1000 exam papers of a class. At about 36 seconds per paper the job takes 10 hours. If the same papers are shared among 10 teachers, each reads 100 papers and the job finishes in about 1 hour. But this calculation is incomplete. Handing out the papers takes time. Collecting the grades into a single list after everyone finishes also takes time. If one teacher is late, the other nine wait for that teacher. The whole subject of parallel algorithms is hidden in this small example: the work is split, the pieces are done at the same time, and the results are combined. The cost of splitting and combining takes back part of the gain.

9.1.1 Why parallelism?

Some jobs take very long on a single processor. Analyzing a data set with billions of rows, running a weather model days ahead, and training a neural network on thousands of images are of this kind. There are two ways to shorten such jobs: make the single processor faster, or split the work among several processors.

For many years the first way was enough. Processor clock speeds doubled every few years, and the same program ran faster without any change. Since the mid-2000s this growth has almost stopped. Raising the clock speed increases heating and power consumption to unacceptable levels. Manufacturers began placing several cores on a single chip instead. Today performance growth comes largely from the number of cores. A program benefits from this growth only if it can split its work among cores.

Definition 9.1 (Parallel algorithm)

A **parallel algorithm** is an algorithm that solves a problem using several processors (or cores) working at the same time. Each processor does a part of the work; when necessary the processors exchange data and coordinate among themselves.

The phrase "at the same time" in the definition is important. A single-core computer also appears to run several programs "at the same time". In reality the operating system switches rapidly between programs, and only one runs at any instant. In a parallel algorithm, several processors really do work in the same time slice. In these notes the words processor, core, and thread are used, depending on context, with the same meaning: "a computing unit that can work independently".

9.1.2 Divide, assign, solve, combine

The skeleton of a parallel algorithm usually consists of four steps. The problem is first **divided** into pieces that are as independent of each other as possible. The pieces are **assigned** to different processors. The processors **solve** their pieces at the same time. Finally the partial results are **combined** to obtain the result of the original problem. Figure 9.1 shows this flow for four processors.

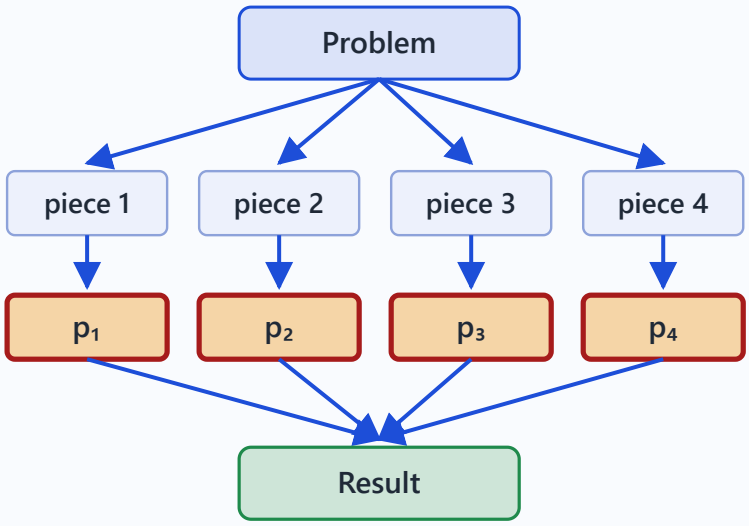


Figure 9.1: The general structure of a parallel algorithm. The problem is divided into four pieces, the pieces are solved at the same time on four processors (p_1 … p_4), and the results are combined.

This structure resembles divide and conquer. In divide and conquer, too, the problem is split into subproblems and the results are combined. The difference lies in how the subproblems are solved. In sequential divide and conquer the subproblems are solved one after another. In a parallel algorithm the independent subproblems are solved at the same time. For this reason divide-and-conquer algorithms are among the algorithms best suited to parallelization.

A group of people carrying out a joint task is a good analogy. When a house is being moved, one person packs the books, one the kitchen, and one the clothes. When everyone finishes, the boxes are loaded onto the same truck. Two conditions are needed for the work to speed up. First, the pieces must really be independent: the person packing the kitchen should not wait for the one packing the books. Second, the pieces must be of roughly equal size: if the books take two hours and the clothes ten minutes, the person packing clothes sits idle most of the time.

Parallel computing is the foundation of **high-performance computing** (HPC). Areas such as big data analysis, scientific simulation, and machine learning rely heavily on parallel algorithms today.

9.1.3 Gains and difficulties

Parallelism has three basic gains:

- **Speedup.** The same job finishes in less time. A weather forecast model is ready this evening instead of tomorrow morning.
- **Scalability.** Large problems that do not fit in the memory or disk of a single machine can be solved by distributing them across many machines.
- **Efficient use of hardware.** All the cores of the computer and the graphics processor (GPU) do work; idle hardware decreases.

These gains also have a price:

- **Splitting and combining require extra effort.** Dividing the problem into pieces and combining the partial results are extra operations that a sequential algorithm does not have.
- **Communication and synchronization add overhead.** Processors must exchange data and wait for each other. No useful work is done during this time.
- **Load balancing is needed.** The work must be distributed evenly among processors. The piece that takes longest determines the time of the whole job.
- **Debugging is hard.** Parallel programs suffer from bugs such as **race conditions** and **deadlock**. These bugs may not show up on every run.
- **Some problems are inherently sequential.** If a step needs the result of the previous step, those steps cannot be done at the same time.

The last item draws the theoretical limit of parallelism. Consider a recurrence of the form $x_k = f(x_{k-1})$, where f is an arbitrary function. To compute x_5, x_4 must be known, and for x_4, x_3. This is a chain of **data dependencies**. However many processors there are, the links of the chain are computed one by one. Such a computation cannot be parallelized.

Not every chain is this rigid, though. The recurrence $y_i = y_{i-1} + x_i$ has the same structure at first glance. But because addition is associative, the parentheses can be rearranged and the computation can be done in parallel in a tree shape. This idea is treated in detail in the section on prefix sums. Whether a chain can be parallelized depends on the algebraic properties of the operation in the chain.

> **Common mistake**
>
> Assuming every loop can be parallelized. In the loop `for (i = 1; i < n; i++) x[i] = x[i - 1] * x[i - 1] + 1;`, every iteration reads the value written by the previous one. If the iterations of the loop are distributed to different threads, a thread reads a value not yet computed and the result is wrong. Before parallelizing, check whether there are data dependencies between iterations.

9.1.4 Application areas

The problems where parallelism pays off share one feature: most of the work can be split into small pieces independent of one another. Four typical examples follow:

- **Matrix multiplication.** It is a basic operation of scientific computing. In the product $C = A \cdot B$, each row of the result matrix, even each cell, can be computed independently of the others. At the end of these notes a Java solution is given that works with one task per row.
- **Image processing.** In operations such as edge detection, blurring, and object recognition, the same computation is done for each pixel or each block of pixels. This is the clearest example of data parallelism. Graphics processors contain thousands of small cores for exactly this kind of work.

- **Computational fluid dynamics.** In designing aircraft, ships, or cars, the flow of air and water is simulated. The flow field is divided into a grid and each region is given to a processor. At each time step the processors share boundary values only with neighboring regions.
- **Genetic algorithms.** The fitness values of the individuals in a population are independent of one another and can be computed at the same time. In problems such as scheduling and route finding, this evaluation is the most expensive part of the job.

9.2 Models of parallel computation

To design and analyze a parallel algorithm, we must first decide what kind of machine it will run on. For sequential algorithms this question is rarely asked. Every computer works more or less the same way: a processor executes instructions in order and accesses memory. Parallel machines, however, differ greatly from one another. What varies is whether the processors run the same instruction or different ones, whether they share memory, and how they communicate. This section describes two viewpoints that classify parallel machines, and the PRAM model used in theoretical analysis.

9.2.1 Flynn's classification

In 1966 Michael Flynn classified computers by two questions. The first question is how many instruction streams are executed at the same time. The second is how many data streams are processed at the same time. Because the answer to each question can be "single" or "multiple", four classes arise. Table 9.1 summarizes these classes. The class names are formed from initials: S stands for single, M for multiple, I for instruction, and D for data.

Class	Expansion	Meaning	Example
SISD	Single Instruction, Single Data	one instruction stream, one data stream	classic single-core processor
SIMD	Single Instruction, Multiple Data	the same instruction on many data items at once	vector instructions (AVX), GPU
MISD	Multiple Instruction, Single Data	different instructions on the same data	rare; fault-tolerant systems
MIMD	Multiple Instruction, Multiple Data	each processor runs its own program on its own data	multicore CPU, clusters

Table 9.1: Flynn's classification.

SISD is the classic machine assumed by sequential algorithms. In every clock step a single instruction is executed on a single data item.

A **SIMD** machine has a single control unit, but this unit applies the same instruction to many data items at the same time. The vector instructions of modern processors are of this kind. For example, an addition instruction in the AVX instruction set adds eight 32-bit floating-point numbers to eight other

numbers in a single step. Graphics processors apply the same idea on a much larger scale. SIMD is very effective for jobs where the same operation is applied to every element of an array. In branching code, where different elements follow different paths, its efficiency drops.

MISD is rare in practice. The same data is processed with different instructions. In fault-tolerant systems such as aircraft control systems, the same input is computed independently on several separate units and the results are compared. This arrangement is given as an example of the MISD class.

MIMD is the most general class. Each processor runs its own program on its own data, independently of the others. Multicore personal computers and clusters, in which many computers are connected by a network, are MIMD machines.

Flynn's classification also helps to tell the two forms of parallelism apart. In **data parallelism** the same operation is applied to different parts of the data at the same time; this is the SIMD style. In **task parallelism**, different tasks are executed at the same time; this is the MIMD style. A web server preparing different pages for different clients at the same time is task parallelism. Raising the brightness of all the pixels of an image by the same ratio is data parallelism.

9.2.2 Memory architectures

MIMD machines are divided into two by the way they organize memory. Figure 9.2 shows the two layouts side by side.

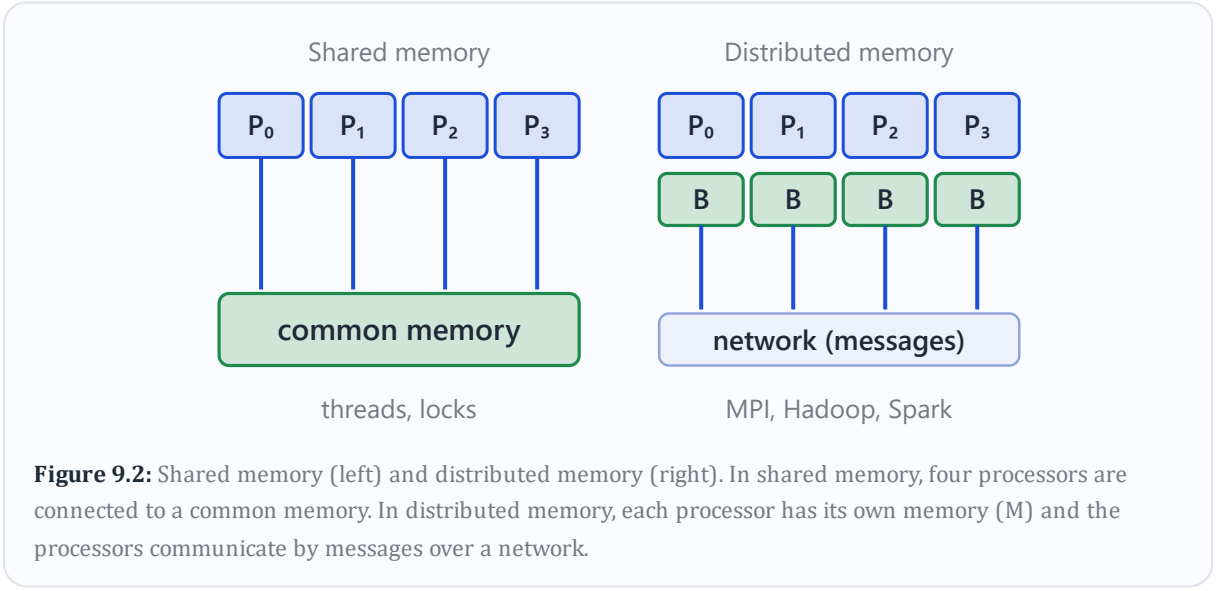


Figure 9.2: Shared memory (left) and distributed memory (right). In shared memory, four processors are connected to a common memory. In distributed memory, each processor has its own memory (M) and the processors communicate by messages over a network.

In the **shared memory** layout all processors see the same address space. A value one processor writes to memory can be read directly by another processor. Communication is therefore cheap: instead of moving data, it is enough to use a common variable. The price is controlling concurrent access. If two processors try to change the same variable at the same time, the result can be

corrupted; accesses must therefore be ordered with a synchronization mechanism. Java threads and the `ForkJoinPool` class, and OpenMP for C and Fortran, are tools for this layout. The cores of a single computer work with shared memory.

In the **distributed memory** layout each node has its own memory. A node cannot directly access another node's memory. If data is needed, its owner sends it over the network as a **message**. This layout scales very well: thousands of nodes can solve the same problem. But communication is expensive. Sending a message over the network can take hundreds of times longer than reading from local memory. Programs written for this layout try to minimize the number of messages. The MPI (Message Passing Interface) library, widely used in scientific computing, and Hadoop MapReduce are distributed memory tools.

In these notes the Java examples run on shared memory. The pipeline and pipelined tree examples, however, assume a distributed layout in which processors exchange messages only with their neighbors. MapReduce too was originally designed for distributed memory; the Java example here shows the same idea on a single machine.

9.2.3 The PRAM model

When analyzing sequential algorithms, a simplified model is used instead of a real computer: the RAM (Random Access Machine) model. In this model every basic operation and every memory access takes one unit of time. Details such as caches, virtual memory, and pipelining are ignored. The model does not reflect reality exactly, but it is accurate enough for comparing algorithms. A similar model is needed for parallel algorithms.

Definition 9.2 (PRAM model)

The **PRAM** (Parallel Random Access Machine) is the parallel counterpart of the RAM model. It consists of p processors, a shared memory, and a shared clock. The processors step in lockstep. In each step every processor reads from memory, does a local computation, and writes to memory. A step takes $O(1)$ time. The cost of communication and synchronization is ignored.

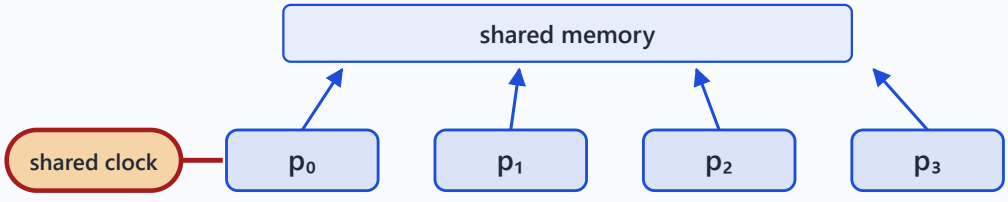


Figure 9.3: The PRAM model. Four processors (p_0 … p_3) are connected to a shared memory and step in lockstep with a shared clock.

The model's strongest assumption is that communication is free. On a real machine it takes time for two processors to share a value; in the PRAM, a value written by one processor can be read by all processors in the next step. The processors working with a shared clock also makes synchronization free. When a round ends, it is certain that all processors have finished that round. These assumptions are not realistic, but they serve a purpose. By setting aside communication and hardware details, the model measures an algorithm's **inherent parallelism**. If an algorithm cannot run in parallel even on a PRAM, it cannot run in parallel on a real machine. An algorithm that runs fast on a PRAM slows down on a real machine by an amount that depends on the communication cost.

There is one more problem with shared memory: what happens if several processors access the same memory cell in the same step? The PRAM variants are named by the answer given to this question. Table 9.2 shows three common variants.

Variant	Read	Write
EREW	exclusive	exclusive
CREW	concurrent	exclusive
CRCW	concurrent	concurrent

Table 9.2: PRAM variants. E = Exclusive, C = Concurrent, R = read, W = write.

Exclusive access means that only a single processor can access the same cell in one step. In **concurrent** access, several processors can access the same cell in the same step. EREW is the most restrictive variant and CRCW the most powerful. Concurrent reading is harmless: everyone reads the same value. Concurrent writing, however, requires a rule. If three processors try to write 5, 7, and 9 to the same cell, which value remains in the cell? The common rules are these:

- **Common rule:** A write is valid only if all processors write the same value.
- **Arbitrary rule:** One of the processors is chosen at random and its value is written.
- **Priority rule:** The value of the processor with the smallest number (highest priority) is written.

If an algorithm works on EREW, it also works on CREW and CRCW, because the stronger model allows every operation of the weaker one. The converse is not always true. So when stating which model an algorithm works on, the weakest possible model is chosen. The tree summation and prefix sum algorithms examined in these notes manage with EREW.

Example 9.1

In a PRAM algorithm, each of n processors wants to read a single `threshold` value in shared memory and compare it with its own element. In which model is this done in one step?

All processors read the same cell in the same step; this is concurrent reading. There is no writing. CREW suffices, and the operation finishes in one step. On EREW, however, only one processor can read the same cell in one step. The value must first be replicated by copying. In

the first step one processor copies it, giving 2 copies; in the second step 2 processors copy it, giving 4 copies. The number of copies doubles every step. n copies are reached in $\lceil \log_2 n \rceil$ steps. This example shows that the choice of model can change the time.

9.3 Performance measures

Whether a parallel algorithm is good cannot be stated with a single number. An algorithm may be fast but use too many processors to be so. It may be fast with many processors, but its gain may stall as the number of processors grows. This section introduces the measures used to compare parallel algorithms.

9.3.1 Speedup, efficiency, cost

All the measures rest on two times. T_1 is the time of the *best sequential algorithm* that solves the problem on a single processor. T_p is the time of the parallel algorithm on p processors. Table 9.3 gives the three measures derived from these two times.

Measure	Definition
T_1	time of the best sequential algorithm
T_p	parallel time on p processors
Speedup	$S = T_1 / T_p$ (ideal: $S = p$)
Efficiency	$E = S / p$ ($0 < E \le 1$)
Cost	$C = p \cdot T_p$

Table 9.3: Parallel performance measures.

Definition 9.3 (Speedup, efficiency, cost)

Speedup is the ratio of the sequential time to the parallel time: $S = T_1 / T_p$. **Efficiency** is the speedup per processor: $E = S / p$. **Cost** is the total time spent by all processors: $C = p \cdot T_p$. A parallel algorithm whose cost is of the same order as the best sequential time, that is, with $C = \Theta(T_1)$, is called **cost-optimal**.

Speedup tells how many times adding processors shortens the job. With p processors, a speedup of p is expected at best; this is called **linear speedup**. Efficiency shows how much of the processors' time goes to useful work. If $E = 1$, no processor has wasted any time. If $E = 0.5$, half of the processor time has gone to waiting, communication, or unnecessary work.

Cost measures how many "processor-seconds" in total the parallel algorithm consumes. The sequential algorithm spends T_1 processor-seconds. The parallel algorithm cannot spend less than that; if it did, running the steps of the parallel algorithm in order on a single processor would give a

sequential algorithm shorter than T_1. That contradicts T_1 being the best sequential time. So always $C \geq T_1$, hence $S \leq p$ and $E \leq 1$. These bounds hold in the model; in reality cache effects or early finding in search can give $S > p$ (**superlinear speedup**). When the data is split into pieces, each piece may fit in the cache; a parallel search may also find the target much earlier than in the sequential search order. A cost-optimal algorithm distributes the work done by the sequential algorithm among the processors, increasing it by at most a constant factor.

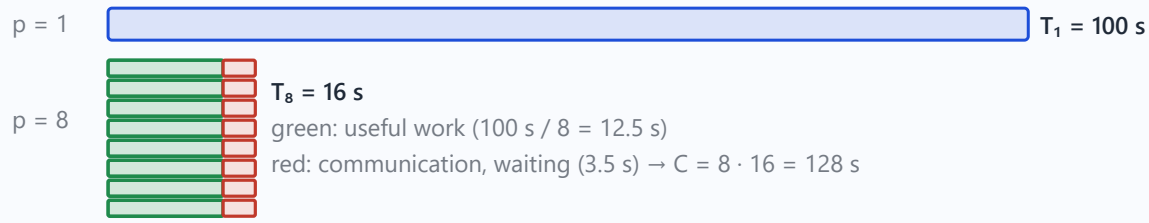


Figure 9.4: A job that takes 100 s on one processor takes 16 s on eight processors. Of each processor's time, 12.5 s is useful work (green) and 3.5 s is communication and waiting (red). Cost $8 \cdot 16 = 128$ s.

Example 9.2

A job takes $T_1 = 100$ s with the best sequential algorithm. The parallel algorithm takes $T_8 = 16$ s on $p = 8$ processors. The three measures are computed as follows:

1. Speedup: $S = 100 / 16 = 6.25$. The job got 6.25 times shorter; the ideal value would have been 8.
2. Efficiency: $E = 6.25 / 8 \approx 0.78$. About 78% of the processor time went to useful work.
3. Cost: $C = 8 \cdot 16 = 128$ s. The sequential algorithm spent 100 s; 28 processor-seconds went to overhead.

Figure 9.4 shows the source of this overhead. When the 100 s job is split eight ways, each processor gets 12.5 s. Since each processor works for 16 s, it spent 3.5 s on communication, waiting, or combining. The total loss of the eight processors is $8 \cdot 3.5 = 28$ s; the difference between cost and sequential time is exactly this much.

There are four common reasons why speedup falls below p. The first is **communication**: processors do no computing while they exchange data. The second is **waiting**: a processor is idle while it waits for another processor's result. The third is **load imbalance**: processors that finish early wait for the slowest one. The fourth is **sequential parts**: the part of the program that cannot be parallelized always takes the same time regardless of the number of processors. The effect of the last cause is computed with Amdahl's law.

Common mistake

Computing speedup relative to the parallel algorithm's own time on a single processor. A parallel algorithm often does more work than the sequential algorithm. For example, Hillis–Steele prefix sums run on a single processor are slower than a sequential scan. If speedup is computed by dividing by this slow time, it looks better than it is. T_1 is always the time of the best sequential algorithm.

9.3.2 Amdahl's law and Gustafson's law

Almost every real program has a part that cannot be parallelized. Reading input from disk, loading settings, and merging partial results into a single output are jobs of this kind. In 1967 Gene Amdahl showed with a simple calculation the limit that this sequential part places on speedup.

Let the program's time on a single processor be T_1. A fraction f of this time can be parallelized and a fraction $1 - f$ is sequential. On p processors the sequential part still takes $(1 - f)\,T_1$. The parallel part, at best, divides exactly by p and takes $f\,T_1 / p$. The total parallel time is

$$T_p = (1 - f)\,T_1 + f\,T_1 / p$$

Since speedup is T_1 / T_p, T_1 cancels and the following result is obtained.

Theorem 9.1 (Amdahl's law)

If a fraction f of a program can be parallelized and a fraction $1 - f$ is sequential, the speedup attainable on p processors is at most

$$S(p) = 1 / ((1 - f) + f / p)$$

As $p \to \infty$, $S(p) \to 1 / (1 - f)$.

The real message of the law is in the limit. However many processors are added, the term f / p approaches zero but the term $1 - f$ stays where it is. The sequential part limits the speedup. If 10% of the program is sequential ($f = 0.9$), speedup cannot exceed 10 even with infinitely many processors. If 5% is sequential, the limit is 20.

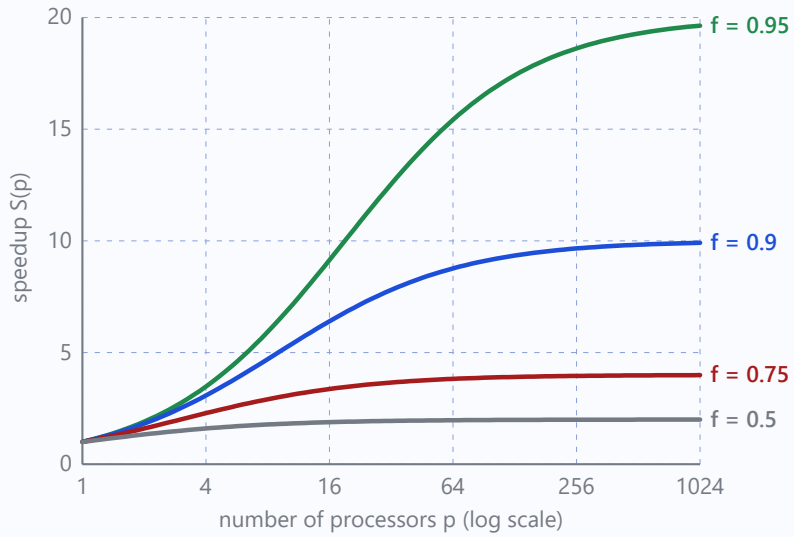


Figure 9.5: Speedup versus number of processors according to Amdahl's law (horizontal axis logarithmic). The curves for $f = 0.95$, 0.9, 0.75, and 0.5 saturate at 20, 10, 4, and 2 respectively.

Example 9.3

90% of a program can be parallelized: $f = 0.9$. What is the speedup on eight processors?

1. Sequential part: $1 - f = 0.1$.
2. Share of the parallel part: $f / p = 0.9 / 8 = 0.1125$.
3. Speedup: $S = 1 / (0.1 + 0.1125) = 1 / 0.2125 \approx 4.71$.
4. Upper limit: $1 / 0.1 = 10$.

Eight processors give a speedup of about 4.7, not 8. Efficiency is $4.71 / 8 \approx 0.59$. Even though only 10% of the program is sequential, on eight processors the sequential part takes about half of the time: $0.1 / 0.2125 \approx 0.47$. As the parallel part speeds up, the share of the sequential part in the total time grows.

The curves in Figure 9.5 show this effect. The $f = 0.9$ curve reaches about 3.1 at 4 processors, 6.4 at 16, and 8.8 at 64. At 1024 processors it is only 9.9. Going from 64 to 1024 multiplies the number of processors by 16 but increases speedup by only 13%. Before moving a program to many processors, it is therefore more profitable to shrink its sequential part. Reducing the sequential part from 10% to 1% raises the speedup on 1000 processors from about 10 to 91.

Amdahl's law rests on an assumption: the problem size is fixed and only the number of processors grows. In practice this is often not the case. A researcher who finds more processors solves a larger problem instead of solving the same problem faster. The weather forecast is made on a finer grid, and the simulation is run with more particles. In this case the sequential part mostly stays the same, but the parallel part grows with the number of processors.

In 1988 John Gustafson put this viewpoint into a formula. Let the time of the parallel run be 1 unit. A fraction $1 - f$ of it is sequential and a fraction f is parallel. If the same job were done on a single processor, the sequential part would still take $1 - f$. The parallel part would have to do alone the work of p processors and would take $f \cdot p$. The scaled speedup is

$$S = (1 - f) + f \cdot p$$

This is called **Gustafson's law**. For $f = 0.9$ and $p = 8$ we get $S = 0.1 + 7.2 = 7.3$. With the same values Amdahl's law gave 4.71. The two laws do not contradict each other; they answer different questions. Amdahl's law answers "how many times faster do I solve the same problem?", and Gustafson's law answers "how many times larger a problem do I solve in the same time?" According to Gustafson's law, parallelism pays off more on large problems, because the share of the sequential part in the total work shrinks.

Common mistake

Forgetting the sequential part in an Amdahl calculation. For $f = 0.9$ and $p = 8$ one cannot say "the parallel part speeds up 8 times, so the program speeds up about 8 times". In the correct calculation the sequential 10% stays as it is and the speedup is 4.71. If 10% of the program is sequential, speedup is bounded by 10 even at best.

9.3.3 Work, depth, and Brent's theorem

Speedup and efficiency are computed for a particular number of processors. To describe an algorithm's parallelism independently of the number of processors, two quantities are used: work and depth. Both rest on viewing the algorithm's operations as a **dependency graph**. Each node of the graph is a basic operation. An edge $u \to v$ shows that operation v uses the result of u, that is, it cannot start before u finishes. There can be no cycles, because an operation cannot wait for its own result.

Definition 9.4 (Work and depth)

Work W is the number of all operations the algorithm performs; it equals the number of nodes in the dependency graph and gives the time on a single processor: $W = T_1$. **Depth** (or **span**) D is the length of the longest dependency chain in the graph; it gives the time with unlimited processors: $D = T_\infty$. The ratio W / D is called the algorithm's **parallelism**.

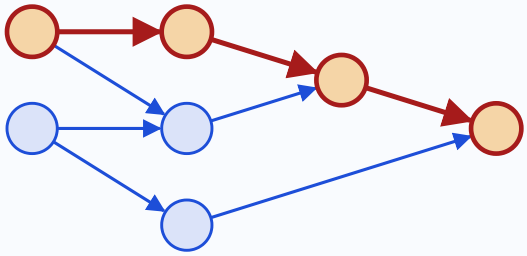


Figure 9.6: A dependency graph with seven operations. The longest chain has four nodes (red path). *W = 7*, *D = 4*, parallelism *W / D = 1.75*.

The graph in Figure 9.6 has seven operations; a single processor finishes them in seven steps. Even with unlimited processors, the four operations on the red path wait for one another and cannot finish before four steps. Depth is a limit that cannot be beaten by adding processors. The parallelism is $7 / 4 = 1.75$: on average 1.75 operations can be done at the same time in each step. More than two processors bring almost no benefit to this graph.

Parallelism is interpreted as **the maximum number of processors that can be used profitably**. If the number of processors exceeds the parallelism, some processors sit idle in most steps. The two basic lower bounds are as follows:

- $T_p \geq W / p$: p processors do at most p operations in a step; W operations need at least W / p steps.
- $T_p \geq D$: the operations in the longest chain wait for one another; the number of processors does not change this.

There is also an upper bound, widely used in the analysis of parallel algorithms.

Theorem 9.2 (Brent's theorem)

A computation with work W and depth D can be executed on p processors in $T_p \leq W / p + D$ steps.

Proof. Divide the operations into levels. Level i contains the operations whose longest chain of predecessors has length $i - 1$. Operations on the same level do not wait for one another; if there were an edge between them, one would be deeper than the other. The number of levels is D. If level i has W_i operations, p processors finish this level in $\lceil W_i / p \rceil \leq W_i / p + 1$ steps. The levels are executed in order. The sum over all levels is $\Sigma\,(W_i / p + 1) = W / p + D$. ■

Reading the lower and upper bounds together gives $\max(W / p, D) \leq T_p \leq W / p + D$. The upper bound is at most twice the lower bound. So once W and D are known, the time for any number of processors can be estimated to within a factor of two. If p is much smaller than the parallelism, the term W / p

dominates and the speedup is almost linear. If p is larger than the parallelism, the term D dominates and new processors do not shorten the time.

A parallel algorithm whose work is of the same order as the time of the best sequential algorithm for the same problem is called **work-efficient**. An algorithm that is not work-efficient can be even slower than the sequential algorithm when processors are few. The section on prefix sums gives a concrete example of this.

Example 9.4

$n = 1024$ numbers are summed with a balanced binary tree. In the first round the numbers are added in pairs, in the second round these sums are added in pairs, and so on. What are the work, the depth, and the time on 32 processors?

1. Each addition reduces two values to one. To reduce 1024 values to one, $W = n - 1 = 1023$ additions are needed.
2. In each round the number of values halves: 1024, 512, …, 1. The number of rounds is $D = \log_2 1024 = 10$.
3. Parallelism: $1023 / 10 \approx 102$. Extra processors help up to about a hundred processors.
4. Brent: $T_{32} \le 1023 / 32 + 10 \approx 31.97 + 10 \approx 42$ steps.
5. Lower bound: $T_{32} \ge \max(1023 / 32, 10) \approx 32$ steps.

With 32 processors the time lies between 32 and 42 steps. The sequential algorithm takes 1023 steps; the speedup lies between 24 and 32.

9.4 Summation on a PRAM

The simplest and most instructive example of a parallel algorithm is the sum of n numbers. The problem is very simple, but the parallel solution contains an idea that will recur throughout the rest of these notes: a chain built with an associative operation can be turned into a balanced tree.

9.4.1 The problem and associative operations

The input is n numbers; the output is their sum. The sequential algorithm starts a sum variable at zero and adds each number to it. It does $n - 1$ additions and takes $O(n)$. The dependency graph of this sequential loop is a chain: each addition waits for the result of the previous one. The chain has length $n - 1$; depth equals work, and there is no parallelism.

What breaks the chain is that addition is **associative**: $(a + b) + c = a + (b + c)$. With an associative operation the parentheses can be placed as desired. The sum of eight numbers is computed in the sequential loop as

$$((((((2 + 3) + 5) + 1) + 7) + 6) + 8) + 4$$

The same sum can also be written as follows:

$$((2 + 3) + (5 + 1)) + ((7 + 6) + (8 + 4))$$

The second form is a balanced binary tree. The four innermost additions are independent of one another and can be done at the same time. The two additions that use their results are also independent of each other. The outermost addition is done last. The additions at each level of the tree are done in parallel in the same round. Figure 9.7 shows this tree.

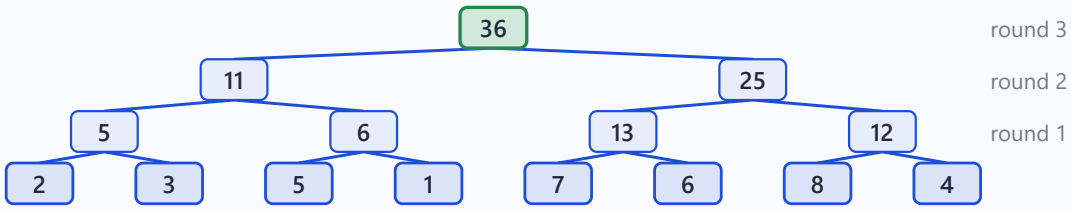


Figure 9.7: A balanced summation tree. The sequence 2 3 5 1 7 6 8 4 is summed in three rounds: 5 6 13 12 → 11 25 → 36.

The same idea works for every associative operation: maximum, minimum, product, logical AND, logical OR. This general operation, which reduces an array to a single value with an associative operation, is called a **reduction**. If `Math.max` is used in the summation tree instead of addition, the largest element is found in the same rounds.

Common mistake

Reducing a non-associative operation with a tree. Subtraction is not associative: $(8 - 4) - 2 = 2$ but $8 - (4 - 2) = 6$. Because the tree order uses a different parenthesization from the sequential order, the result changes. Although addition of real numbers is associative, floating-point addition on a computer is not exactly associative because of rounding. A parallel sum can differ from the sequential sum in the last digits.

9.4.2 Tree summation step by step

The example sequence is 2, 3, 5, 1, 7, 6, 8, 4; its sum is 36. Eight processors p_0 … p_7 each hold one number. The algorithm proceeds in rounds:

1. **Start.** The eight numbers are distributed to the eight processors, one each.
2. **Round 1.** Four processors add neighboring pairs at the same time: $2 + 3 = 5$, $5 + 1 = 6$, $7 + 6 = 13$, $8 + 4 = 12$. The results stay in the processor on the left of each pair (p_0, p_2, p_4, p_6).

3. **Round 2.** Two processors combine partial sums at distance 2 from each other: *5 + 6 = 11* at p_0 and *13 + 12 = 25* at p_4.
4. **Round 3.** p_0 finds the result *11 + 25 = 36*.

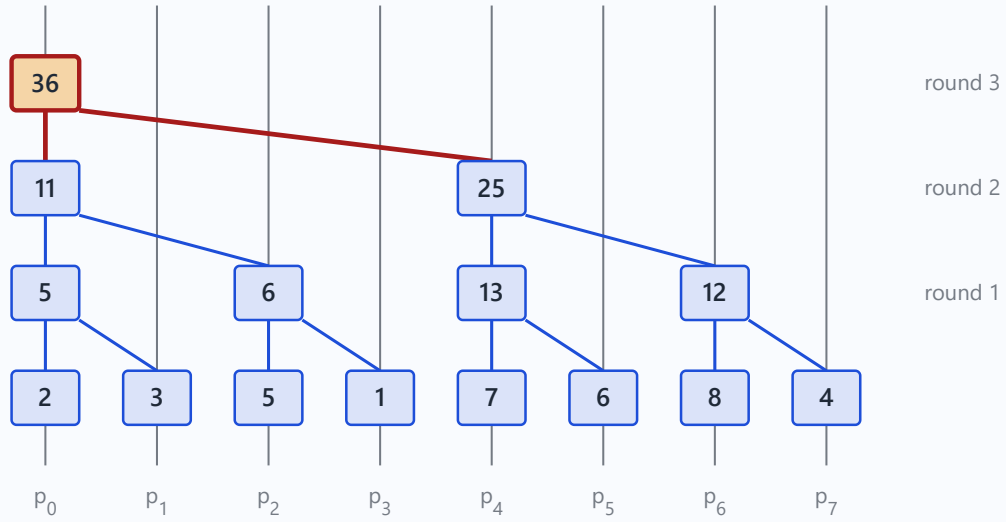


Figure 9.8: The final state of PRAM summation. The bottom row holds the initial values of the eight processors, and the rows above hold the partial sums of rounds 1, 2, and 3. The result, 36, is found at p_0 in three rounds.

Eight numbers were summed in $log_2\ 8 = 3$ rounds. The sequential algorithm would have taken 7 steps. The number of additions, however, did not change: *4 + 2 + 1 = 7*. The parallel algorithm did no extra work; it only changed the order. For this reason tree summation is work-efficient.

One more thing about the rounds is worth noting. In round 1 four processors work, in round 2 two, and in round 3 one. The other processors sit idle. In no round do all eight processors work; even for the first round four processors are enough. This observation will matter in the cost analysis.

9.4.3 Code trace

Tree summation is written in Java as follows. The code is a sequential program, but the iterations of the inner loop are independent within the same round and are executed in parallel on a PRAM. Each iteration of the outer loop is one round.

```
int[] a = {2, 3, 5, 1, 7, 6, 8, 4};
int n = a.length;
for (int step = 1; step < n; step *= 2) {
  // all i in the same round run in parallel
  for (int i = 0; i + step < n; i += 2 * step)
    a[i] = a[i] + a[i + step];
}  // end of round: synchronization
// the total is in a[0]
```

The parts of the code work as follows:

- `step` is the distance between the two cells added in a round. It starts at 1 and doubles each round: 1, 2, 4. The loop ends when `step` is at least `n`.
- In the inner loop `i` runs over the multiples of `2 * step`. These are the cells that work in the round: 0, 2, 4, 6 in round 1; 0, 4 in round 2; only 0 in round 3.
- `a[i] = a[i] + a[i + step]` adds the partial sum on the right to the one on the left. After each round, `a[i]` holds the sum of the block of length `2 * step` starting at `i`.
- The condition `i + step < n` prevents a cell without a partner from reading outside the array. If n is not a power of two, some cells are left unpaired for a round and pass to the next round without waiting.
- The comment at the end of the round is a reminder of synchronization. The next round must start only after all the writes of the previous round have finished.

The writes in the same round go to separate cells. The cells read also differ from round to round: in round 1, `a[1]`, `a[3]`, `a[5]`, `a[7]` are read and are not written. No cell is accessed by two processors in the same round. An EREW PRAM is therefore enough. Table 9.4 is a trace of the code; the values were verified by running the program.

Round	`step`	Parallel additions	`a` after the round
—	—	initial	[2, 3, 5, 1, 7, 6, 8, 4]
1	1	`a[0]` = 2 + 3 = 5; `a[2]` = 5 + 1 = 6; `a[4]` = 7 + 6 = 13; `a[6]` = 8 + 4 = 12	[5, 3, 6, 1, 13, 6, 12, 4]
2	2	`a[0]` = 5 + 6 = 11; `a[4]` = 13 + 12 = 25	[11, 3, 6, 1, 25, 6, 12, 4]
3	4	`a[0]` = 11 + 25 = 36	[36, 3, 6, 1, 25, 6, 12, 4]
—	8	`step` = 8, loop ends	total `a[0]` = 36

Table 9.4: Trace of tree summation for `a = {2, 3, 5, 1, 7, 6, 8, 4}`.

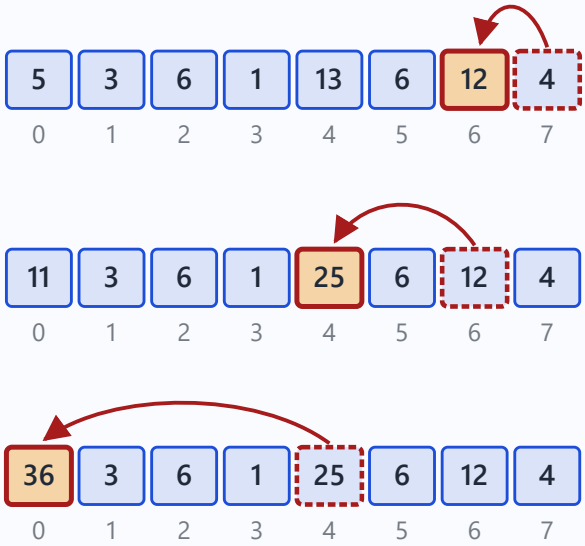


Figure 9.9: The state of the array at the end of the three rounds (the drawings are rounds 1, 2, 3 in order). The cells written in a round are highlighted. In the final state the total is in `a[0]`; intermediate sums remain in the other cells.

Intermediate sums such as `a[4] = 25` and `a[2] = 6` remain in the final array. The algorithm destroys the array; only `a[0]` is meaningful. If the original array is needed, the algorithm should be run on a copy.

Example 9.5

How does the code work when n is not a power of two? For `a = {3, 1, 4, 1, 5}` ($n = 5$) the trace is as follows:

1. `step` = 1: for `i` = 0, `a[0]` = 3 + 1 = 4; for `i` = 2, `a[2]` = 4 + 1 = 5. At `i` = 4, $4 + 1 < 5$ fails; `a[4]` = 5 is left unpaired. The array is [4, 1, 5, 1, 5].
2. `step` = 2: for `i` = 0, `a[0]` = 4 + 5 = 9. At `i` = 4, $4 + 2 < 5$ fails. The array is [9, 1, 5, 1, 5].
3. `step` = 4: for `i` = 0, $0 + 4 < 5$ holds; `a[0]` = 9 + 5 = 14.
4. `step` = 8: the loop ends. `a[0]` = 14 = 3 + 1 + 4 + 1 + 5.

The number of rounds is $\lceil \log_2 5 \rceil = 3$ and the number of additions is $n - 1 = 4$. The unpaired `a[4]` joins the addition in the last round after waiting two rounds.

9.4.4 Analysis and the cost-optimal version

The time of tree summation is the number of rounds: $\lceil \log_2 n \rceil$. Its work is $n - 1$ additions, the same as the sequential algorithm. The number of processors must equal the number of additions in the first round, that is, $n / 2$. For convenience in the analysis, n processors are often written; the order of

magnitude does not change. The cost, however, is not good: $(n / 2) \cdot \log_2 n = \Theta(n \log n)$. Since the best sequential time is $\Theta(n)$, the algorithm is not cost-optimal.

The reason for the high cost was visible in the rounds. Half of the processors are idle after round 1, and three quarters after round 2. Idle processors also count toward the cost, because cost multiplies the number of processors by the whole time.

The solution is to reduce the number of processors and give each processor more work. Let $p = n / \log_2 n$ processors be used. The algorithm consists of two phases:

1. **Local sum.** The array is divided into p blocks; each block has $\log_2 n$ numbers. Each processor sums its own block sequentially. This phase takes $O(\log n)$.
2. **Combining with a tree.** The $p = n / \log_2 n$ partial sums are combined by tree summation. This phase takes $\lceil \log_2 p \rceil \le \log_2 n$ rounds, that is, $O(\log n)$.

The total time is still $O(\log n)$. The cost, however, becomes $(n / \log n) \cdot O(\log n) = \Theta(n)$. The algorithm is now cost-optimal. Table 9.5 compares the two versions.

Measure	n processors ($n / 2$ suffice)	$p = n / \log_2 n$ processors
Time T_p	$\log_2 n$	$O(\log n)$
Work W	$n - 1$	$n - 1$
Cost $p \cdot T_p$	$\Theta(n \log n)$	$\Theta(n)$
Cost-optimal?	no	yes

Table 9.5: Two versions of PRAM tree summation.

This technique is a concrete application of Brent's theorem. In tree summation $W = n - 1$ and $D = \log_2 n$. For $p = n / \log_2 n$, Brent's upper bound gives $W / p + D \approx \log_2 n + \log_2 n = 2 \log_2 n$. The time only doubles, while the number of processors falls by a factor of $\log_2 n$. For example, for $n = 1024$ the number of processors drops from 1024 to about 102.

For the algorithm to work correctly, a **barrier** is needed at the end of each round. A barrier is a coordination point that no processor can pass until all processors have reached it. Because the PRAM works with a shared clock, it provides the barrier for free. On a real machine each barrier has a cost; for this reason a small number of rounds matters on a real machine too.

Common mistake

Forgetting the synchronization between rounds. In a tree summation written with real threads, if the thread doing round 2 reads `a[2]` before all the writes of round 1 have finished, it reads the old value (5). `a[0]` is then computed as 5 + 5 = 10 instead of 5 + 6, and the result is wrong. Because the bug depends on timing, it may not show up on every run.

9.5 Parallel prefix sums

Tree summation produces only a single result: the sum of the array. In many problems, however, a separate sum is needed for every position. The balance of a bank account after each transaction, the cumulative values of a histogram, and the position at which each item is placed in counting sort are of this kind. This section describes two parallel algorithms that compute all these values at the same time.

9.5.1 The problem

Definition 9.5 (Prefix sum)

The **inclusive prefix sum** (or **scan**) of the sequence $x_0, x_1, \ldots, x_{n-1}$ is the sequence of the values $y_i = x_0 + x_1 + \ldots + x_i$ for every i. In the **exclusive prefix sum**, position i does not include x_i: $y_0 = 0$ and $y_i = x_0 + \ldots + x_{i-1}$.

The inclusive prefix sum of the sequence 2 3 5 1 7 6 8 4 is 2 5 10 11 18 24 32 36. The exclusive prefix sum is the same shifted one position to the right: 0 2 5 10 11 18 24 32. One is easily obtained from the other. To go from inclusive to exclusive, subtract each position's own element; in the reverse direction, add it.

The sequential algorithm is a single loop: $y_i = y_{i-1} + x_i$. It does $n - 1$ additions and takes $O(n)$. But it is entirely a chain: each step waits for the result of the previous one. At first sight a parallel solution looks impossible, because y_7 depends on y_6, and y_6 on y_5.

Tree summation showed the way to break this chain. In tree summation too, every intermediate sum is the sum of a block. The difference is this: tree summation stops only at the last cell, that is, at the sum of the whole array. In prefix sums, every cell has to learn its own prefix. The intermediate sums in the tree are parts of this information; what remains is to distribute them correctly.

9.5.2 The Hillis–Steele algorithm

The algorithm published by W. Daniel Hillis and Guy Steele in 1986 rests on a very simple rule. For $d = 1, 2, 4, \ldots$, in each round every position $i \geq d$ adds to its own value the value d positions to its left:

> $x[i] \leftarrow x[i] + x[i - d]$ (for every $i \geq d$, in parallel in the same round)

All reads in a round must see the values from before that round's writes. On a PRAM all processors first read and then write; this condition is satisfied automatically. In a sequential simulation, a copy of the old array is kept in each round and the reads are done from the copy.

Why the algorithm is correct can be seen with an invariant. When round d is finished, every $x[i]$ holds the sum of the window ending at i that contains at most $2d$ elements. At the start (before round $d = 1$) every cell holds only itself, that is, a window of 1 element. In round d, $x[i - d]$ is added to $x[i]$. Before the round, $x[i]$ holds the range $i - d + 1 \dots i$ and $x[i - d]$ holds the range $i - 2d + 1 \dots i - d$. The two ranges are adjacent and their sum gives the range $i - 2d + 1 \dots i$. When the window reaches the start of the array, it is cut off. After $\lceil \log_2 n \rceil$ rounds the window length exceeds n and every cell holds its entire prefix.

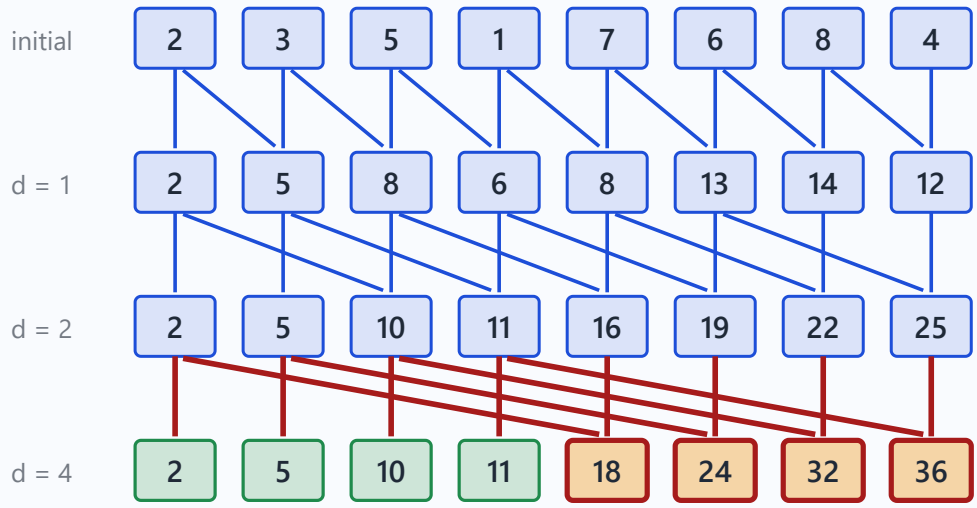


Figure 9.10: Hillis–Steele prefix sums. The input is at the top, and below are the results of the rounds $d = 1, 2, 4$. Each line shows the cell on the left that a cell reads in that round. The last row is the inclusive prefix sum: 2 5 10 11 18 24 32 36.

Table 9.6 gives the result of each round for the example sequence. The values were verified with a Java program, using a simulation with copies.

Round	Operation	Additions	x after the round	Window
initial	—	—	2 3 5 1 7 6 8 4	1
$d = 1$	$i \geq 1$: $x[i] \mathrel{+}= x[i - 1]$	7	2 5 8 6 8 13 14 12	2
$d = 2$	$i \geq 2$: $x[i] \mathrel{+}= x[i - 2]$	6	2 5 10 11 16 19 22 25	4
$d = 4$	$i \geq 4$: $x[i] \mathrel{+}= x[i - 4]$	4	2 5 10 11 18 24 32 36	8

Table 9.6: Hillis–Steele rounds for 2 3 5 1 7 6 8 4.

Let us check one row of the table explicitly. At the end of round $d = 2$, $x[5] = 19$. This value is the sum of the 4-element window ending at 5: $x_2 + x_3 + x_4 + x_5 = 5 + 1 + 7 + 6 = 19$. The computation went as follows: before the round, $x[5] = 13$ (6 + 7) and $x[3] = 6$ (1 + 5); $13 + 6 = 19$.

The algorithm finished in $\log_2 8 = 3$ rounds. The depth is $D = \lceil \log_2 n \rceil$. But the number of additions is $7 + 6 + 4 = 17$; the sequential algorithm does 7 additions. In general, round d does $n - d$ additions. If n is a power of two, the total work is

$$W = \Sigma_{d = 1, 2, 4, \ldots, n/2} (n - d) = n \log_2 n - (n - 1)$$

For $n = 8$ this is $24 - 7 = 17$. The work is $\Theta(n \log n)$, more than the $\Theta(n)$ work of the sequential algorithm. Hillis–Steele is **not work-efficient**. If processors are plentiful this does not matter, because the extra operations are done for free in the same round. If processors are few, the extra work adds directly to the time. For example, on a single processor Hillis–Steele takes $n \log n$, while the sequential scan takes n.

Common mistake

Forgetting the copy at the start of the round in a sequential simulation. If the computation is done on the same array with i in increasing order, $x[i - d]$ may already have been updated in this round. On the sequence 2 3 5 1, if the round $d = 1$ is done without a copy, $x[1] = 5$, then $x[2] = 5 + 5 = 10$, then $x[3] = 1 + 10 = 11$. The full prefix sum comes out in a single round; the next round adds to these values again and the result is corrupted: after round $d = 2$ one gets 2 5 12 16. The correct answer is 2 5 10 11. In real parallel code this error corresponds to forgetting the barrier between rounds.

9.5.3 The Blelloch algorithm

The algorithm popularized by Guy Blelloch in 1990 computes prefix sums with $O(n)$ work and $O(\log n)$ depth. It uses the array like an implicit binary tree and works in two phases. The **up-sweep** phase is the same as tree summation: partial sums are built from the leaves toward the root. The **down-sweep** phase descends from the root toward the leaves and carries to each node "the sum of all the elements to its left".

```
// up-sweep; x = 2 3 5 1 7 6 8 4
for (int d = 1; d < n; d *= 2)
  for (int i = 2*d - 1; i < n; i += 2*d)
    x[i] += x[i - d];
x[n - 1] = 0;
// down-sweep (inner loops in parallel)
for (int d = n / 2; d >= 1; d /= 2)
  for (int i = 2*d - 1; i < n; i += 2*d) {
    int t = x[i - d];
    x[i - d] = x[i];
    x[i] += t;
  }
// x: exclusive prefix sums
```

The parts of the code work as follows:

- **Up-sweep.** For *d = 1, 2, 4*, `x[i] += x[i - d]` is done in the cells *i = 2d − 1, 4d − 1, ...*. Its only difference from tree summation is that the sums are written to the right end of the block, its last cell, rather than to its left. At the end of the phase `x[n - 1]` holds the sum of the whole array.
- **The root is zeroed.** `x[n - 1] = 0`. There are no elements to the left of the root; the sum of those to its left is 0.
- **Down-sweep.** For *d = n/2, ..., 2, 1*, three lines run on each pair *(i − d, i)*. `t` stores the left child's old value, that is, the sum of the left subtree. The left child takes the parent's value: the elements to the left of the left subtree are the same as those to the left of the parent. The right child adds the sum of the left subtree to the parent's value: to the left of the right subtree there is, in addition to what is to the left of the parent, the left subtree.
- The pairs in a round touch disjoint cells; the inner loop runs in parallel and EREW suffices.

Table 9.7 gives all the rounds of both phases for the example sequence. The values were verified by running the code.

Phase	*d*	Parallel operations	*x* after the round
input	—	—	2 3 5 1 7 6 8 4
up	1	*x[1] += x[0], x[3] += x[2], x[5] += x[4], x[7] += x[6]*	2 5 5 6 7 13 8 12
up	2	*x[3] += x[1], x[7] += x[5]*	2 5 5 11 7 13 8 25
up	4	*x[7] += x[3]*	2 5 5 11 7 13 8 36
root	—	*x[7] = 0*	2 5 5 11 7 13 8 0
down	4	pair (3, 7): *x[3] = 0, x[7] = 0 + 11*	2 5 5 0 7 13 8 11
down	2	pair (1, 3): *x[1] = 0, x[3] = 0 + 5*; pair (5, 7): *x[5] = 11, x[7] = 11 + 13*	2 0 5 5 7 11 8 24
down	1	pairs (0, 1), (2, 3), (4, 5), (6, 7)	0 2 5 10 11 18 24 32

Table 9.7: Trace of Blelloch prefix sums for 2 3 5 1 7 6 8 4.

Let us unfold the last round one pair at a time. Pair (0, 1): `t = x[0] = 2`; `x[0] = x[1] = 0`; `x[1] = 0 + 2 = 2`. Pair (2, 3): `t = 5`; `x[2] = 5`; `x[3] = 5 + 5 = 10`. Pair (4, 5): `t = 7`; `x[4] = 11`; `x[5] = 11 + 7 = 18`. Pair (6, 7): `t = 8`; `x[6] = 24`; `x[7] = 24 + 8 = 32`. The result is 0 2 5 10 11 18 24 32; this is the exclusive prefix sum. If the inclusive result is needed, each cell's own input value is added, or the array is shifted one position to the left and the total (36) is written at the end.

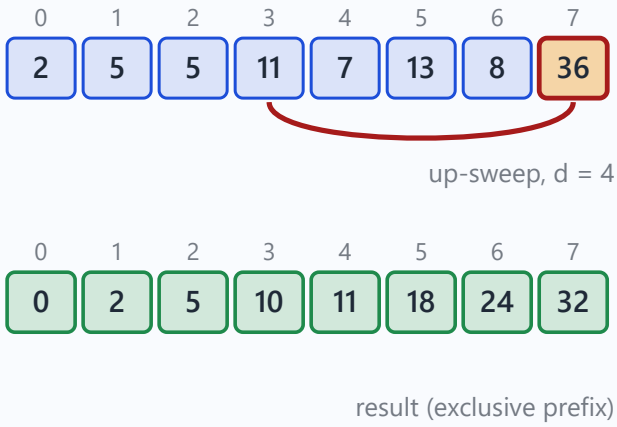


Figure 9.11: The end of the up-sweep (first drawing; root *x[7] = 36*) and the end of the down-sweep (second drawing; exclusive prefix sum 0 2 5 10 11 18 24 32) in the Blelloch algorithm.

The algorithm's cost is counted as follows. The up-sweep is the same as tree summation: $n - 1$ additions, $\log_2 n$ rounds. In the down-sweep one operation is done for each internal node: again $n - 1$ operations, $\log_2 n$ rounds. The total work is $2(n - 1)$; in the example, 14 operations. The depth is $2 \log_2 n$; in the example, 6 rounds. Because the work is $O(n)$, the Blelloch algorithm is work-efficient. It uses twice as many rounds as Hillis–Steele but its total work is smaller by a factor of $\log n$.

Example 9.6

At the end of the down-sweep *x[5] = 18* came out. Where does this value come from?

1. Cell 5 is the right child of the pair (4, 5) in the implicit tree. The parent of this pair is the left cell 5 of the pair (5, 7); its parent is in turn the right cell 7 of the pair (3, 7).
2. In the first round of the down-sweep ($d = 4$), *x[7] = 11*: the sum of the elements to the left of the right half, $2 + 3 + 5 + 1 = 11$.
3. In round $d = 2$, the left child of the pair (5, 7) takes the parent's value: *x[5] = 11*. To the left of cells 4 and 5 there are again the first four elements.
4. In round $d = 1$, the right child of the pair (4, 5) becomes $x[5] = 11 + t$. t is the left child's value left over from the up-sweep: *x[4] = 7*. The result is $11 + 7 = 18$.

18 is indeed the sum of the five elements to the left of cell 5: $2 + 3 + 5 + 1 + 7 = 18$.

9.5.4 Comparing the two algorithms

Algorithm	Work W	Depth D	Result
Sequential scan	$n - 1$ (7)	$n - 1$ (7)	inclusive
Hillis–Steele	$n \log_2 n - (n - 1)$ (17)	$\log_2 n$ (3)	inclusive

Algorithm	Work W	Depth D	Result
Blelloch	$2(n-1)$ (14)	$2 \log_2 n$ (6)	exclusive

Table 9.8: Prefix sum algorithms (values for $n = 8$ in parentheses).

Which algorithm to choose depends on the number of processors. If the number of processors is on the order of n, for example inside a block on a GPU, Hillis–Steele's few rounds are an advantage. If the number of processors is much smaller than n, Blelloch's small amount of work is decisive. By Brent's theorem, on p processors Hillis–Steele takes $O(n \log n / p + \log n)$ and Blelloch $O(n / p + \log n)$.

Prefix sum is one of the most widely used building blocks of parallel algorithms. In parallel counting sort, the position of each key in the output is found with a prefix sum. To select and compact the elements of an array that satisfy a condition (filter), write 1 for each element that satisfies it and 0 for each that does not. The exclusive prefix sum of this 0/1 array gives the position of each selected element in the new array. In Java, the method `Arrays.parallelPrefix(array, Integer::sum)` modifies an array in place into its inclusive prefix sums. For the example array it gives 2 5 10 11 18 24 32 36.

Common mistake

Looking only at depth. Hillis–Steele finishes in $O(\log n)$ rounds, and at first glance this looks better than Blelloch. But it does $O(n \log n)$ work. On a four-core machine, for an array of a million elements, Hillis–Steele does about 20 million additions, which is 5 million per core. The sequential scan does 1 million additions on a single core. With few processors, an algorithm that is not work-efficient can be slower than the sequential algorithm.

9.6 The pipeline

The algorithms so far assumed that the processors share a common memory. In this section the processors are connected as a chain and exchange messages only with their neighbors. In such a layout an important source of parallelism is the pipeline.

9.6.1 The pipeline idea

Definition 9.6 (Pipeline)

A **pipeline** is a layout in which a job is split into consecutive stages and each stage runs on a separate processor. While a job moves from one stage to the next, the earlier stage starts the next job; thus different jobs are processed overlapped in different stages.

An assembly line in a factory is a good analogy. When a car body moves from the welding section to the paint section, the welding section starts on the next body. The time for a single car to leave the line does not shrink; every body has to pass through all the stages. But the number of cars leaving the line per unit time, that is, the **throughput**, grows by a factor equal to the number of stages.

Let m jobs be done in a pipeline with k stages, each stage taking one time step. In sequential execution each job takes k steps and the jobs are done one after another: $k \cdot m$ steps. In the pipeline the first job finishes in k steps. Each following job finishes one step after the previous one. The total time is $k + m - 1$ steps.

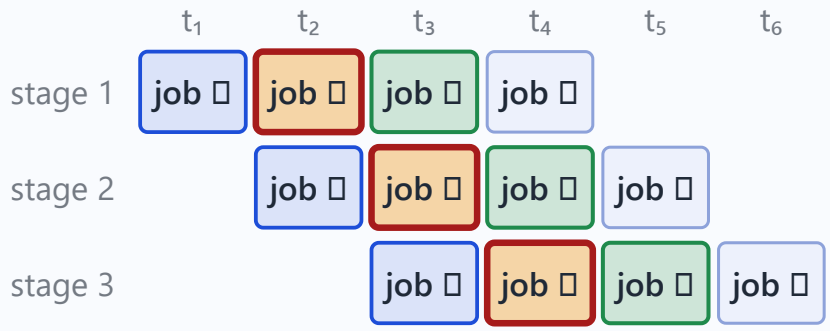


Figure 9.12: Four jobs in a three-stage pipeline. In every time step (t_1 … t_6) each job advances one stage. The jobs finish in $3 + 4 - 1 = 6$ steps instead of $3 \cdot 4 = 12$.

Figure 9.12 shows the line for three stages and four jobs. At the start, while the line is filling (t_1 and t_2), and at the end, while it is emptying (t_5 and t_6), some stages are idle. The line is full only at times t_3 and t_4; at those times all three stages work. If the number of jobs is very large relative to the number of stages, the fill and drain time becomes negligible and the time approaches m.

9.6.2 Row sums on a chain

A concrete problem shows the use of the pipeline in parallel algorithms. Five processors p_0 … p_4 form a linear chain. Each processor can send messages only to its right neighbor. Each p_j holds a column of 4 values; the rows are numbered 0 to 3. Putting the processors' columns side by side gives a table of 4 rows and 5 columns.

The goal is this: for every row r, let the value at p_j be the sum of the values of p_0 … p_j in that row. This is a separate prefix sum in each row. At the end p_4 holds the sum of every row. In one step each processor can send one value to its right neighbor; the receiver adds it to its own value. The values of p_0 are already the result, because there is no one to its left.

The **naive method** processes the rows one at a time. First row 0 is carried along the chain: $p_0 \to p_1$, then $p_1 \to p_2$, and so on. This takes 4 steps because it passes through four links. Then row 1, row 2, and row 3 are carried in turn. The total takes $4 \cdot 4 = 16$ steps. In each step only one of the four links is used.

The **pipeline method** streams the rows one behind another along the chain. p_j sends row r at time $t = r + j$. One step after row 0 sets out, row 1 sets out; row 0 is by then one link ahead. The rows advance like a wave, and most of the links work at the same time.

9.6.3 Step-by-step execution

In the example the processors' columns are as follows: p_0 = (2, 1, 7, 4), p_1 = (3, 7, 2, 9), p_2 = (3, 4, 7, 3), p_3 = (6, 1, 0, 2), p_4 = (8, 6, 3, 5). Which link carries which row at each step is given in Table 9.9.

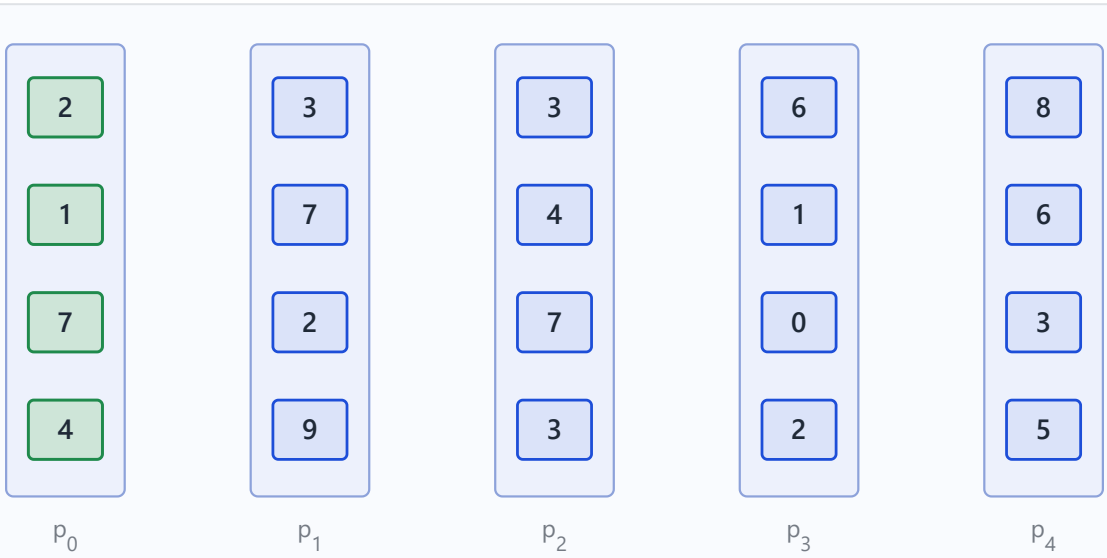


Figure 9.13: The start of the pipeline. Each of the five processors holds a column of 4 rows. The values of p_0 are already the result.

Step	$p_0 \to p_1$	$p_1 \to p_2$	$p_2 \to p_3$	$p_3 \to p_4$
1	row 0: 2 + 3 = 5	—	—	—
2	row 1: 1 + 7 = 8	row 0: 5 + 3 = 8	—	—
3	row 2: 7 + 2 = 9	row 1: 8 + 4 = 12	row 0: 8 + 6 = 14	—
4	row 3: 4 + 9 = 13	row 2: 9 + 7 = 16	row 1: 12 + 1 = 13	row 0: 14 + 8 = 22
5	—	row 3: 13 + 3 = 16	row 2: 16 + 0 = 16	row 1: 13 + 6 = 19
6	—	—	row 3: 16 + 2 = 18	row 2: 16 + 3 = 19
7	—	—	—	row 3: 18 + 5 = 23

Table 9.9: The seven steps of the pipeline. Each cell gives the row carried by that link in that step and the addition performed.

The table shows a diagonal pattern. Row 0 sets out at step 1 and reaches p_4 at step 4. Row 3 sets out at step 4 and arrives at step 7. At step 4 the line is full: all four links work. Before that the line is filling, and after it, emptying.

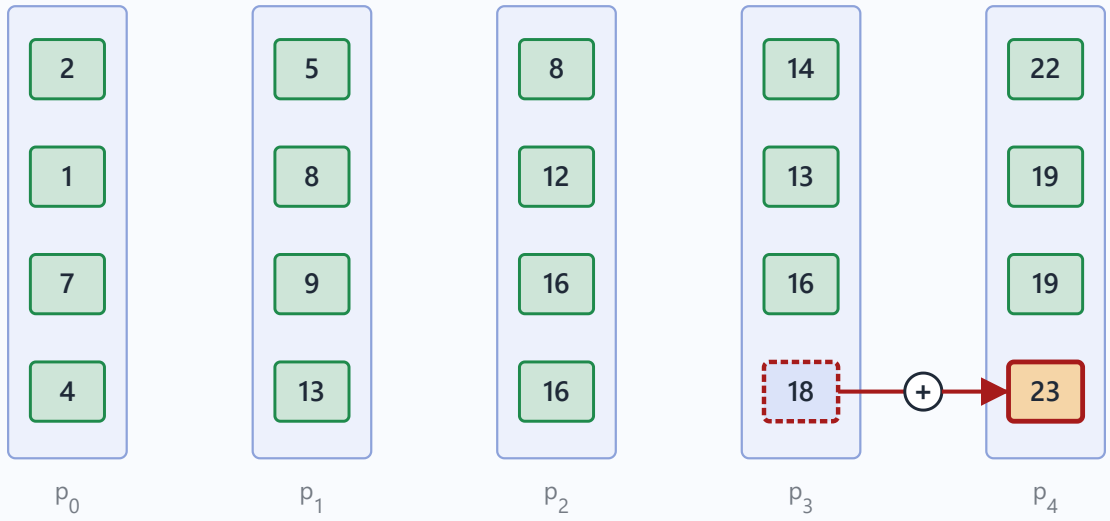


Figure 9.14: Step seven: p_3 sends row 3 (18 + 5 = 23). The line has emptied; p_4 holds the row sums: 22, 19, 19, 23.

Let us verify the result for one row. The values of row 2 from left to right are 7, 2, 7, 0, 3. Their sum is *7 + 2 + 7 + 0 + 3 = 19*. In the pipeline this row leaves p_0 at step 3 (9), reaches p_2 at step 4 (16), p_3 at step 5 (16), and p_4 at step 6 (19). The intermediate values are also correct: the 16 at p_2 is the sum of the first three values of the row.

9.6.4 Code trace and analysis

The pipeline's schedule is written as a short loop. `x[j][r]` is the value of p_j in row r. Each iteration of the outer loop is a time step; the iterations of the inner loop run in parallel in the same step.

```
for (int t = 0; t < m + p - 2; t++)
  // all j at the same time
  for (int j = 0; j < p - 1; j++) {
    int r = t - j;
    if (r >= 0 && r < m)
      x[j + 1][r] += x[j][r];
  }
```

- `t` is the time step, starting from zero. The loop runs $m + p - 2$ times; in the example $4 + 5 - 2 = 7$.
- `j` denotes the link $p_j \to p_{j+1}$; there are $p - 1$ links.
- `r = t - j` is the row this link carries at time `t`. p_j sends row r at time $t = r + j$.
- The condition `if (r >= 0 && r < m)` filters out the cases where the row has not yet set out ($r < 0$, while the line is filling) or all rows have already passed ($r \geq m$, while the line is emptying).
- `x[j + 1][r] += x[j][r]` adds the sent value to the receiver's value.

Two observations show that the schedule is correct. First, when p_j sends row r, its value is ready. p_{j-1} updated this value one step earlier, at time $t - 1 = r + j - 1$. Second, operations in the same step do not conflict. At time `t`, link j reads `x[j][r]`. Link $j - 1$ writes to `x[j][r + 1]` in the same column. The cells read and written are different.

Table 9.10 gives the additions the code performs at each step and the column of p_4. The values were verified by running the code; they are the same as in Table 9.9, only with time counted from zero.

`t`	Active `j` (`r = t - j`)	Number of additions	p_4 column
0	0 (r = 0)	1	8, 6, 3, 5
1	0 (1), 1 (0)	2	8, 6, 3, 5
2	0 (2), 1 (1), 2 (0)	3	8, 6, 3, 5
3	0 (3), 1 (2), 2 (1), 3 (0)	4	22, 6, 3, 5
4	1 (3), 2 (2), 3 (1)	3	22, 19, 3, 5
5	2 (3), 3 (2)	2	22, 19, 19, 5
6	3 (3)	1	22, 19, 19, 23

Table 9.10: Trace of the pipeline code ($p = 5$, $m = 4$).

The general formula for the number of steps is found as follows. The last row is $r = m - 1$ and the last link is $j = p - 2$. The last row passes the last link at time $t = (m - 1) + (p - 2)$. Since time is counted from zero, the number of steps is

$$m + p - 2$$

In the example it is $4 + 5 - 2 = 7$. The naive method sends each row separately across $p - 1$ links: $m(p - 1) = 16$ steps. This result agrees with the formula $k + m - 1$ from the pipeline idea: here the number of stages is $k = p - 1$ links.

Efficiency can also be computed. The useful work is the number of additions performed: each row passes each link once, $m(p - 1)$ additions in total. The resource used is the $p - 1$ links being open for $m + p - 2$ steps. The efficiency is

$$E = m(p - 1) / ((p - 1)(m + p - 2)) = m / (m + p - 2)$$

In the example, $4 / 7 \approx 0.57$, about 57%. If the number of rows is much larger than the number of processors ($m \gg p$), efficiency approaches 1. For example, for $p = 5$ and $m = 1000$ it is $1000 / 1003 \approx 0.997$. The pipeline is effective when many independent data items pass through the same stages.

Common mistake

Thinking the number of steps of a pipeline is m. The line is empty at the start and takes time to fill. The last row sets out only after $m - 1$ steps and needs $p - 1$ more steps to cross the chain. For m rows and p processors the correct number is $m + p - 2$. For $p = 6$, $m = 50$ this difference is 4 steps; for $p = 100$, $m = 10$ it is 98 steps and changes the result completely.

9.7 The pipelined tree

The pipeline idea applies not only to chains but also to trees. In this section the processors are connected as a tree and each processor holds a vector. The goal is to sum the vectors component by component at the root.

9.7.1 Reduction on a tree

Let the processors form a rooted tree. Each node can send its value to its parent; the parent adds the incoming value to its own. The leaves send their values, internal nodes add what comes from their children and pass it up, and finally the total accumulates at the root. This is the message-passing form of tree summation. If the depth of the tree is h, reducing a single value takes h stages.

In the example there are seven processors. The children of the root p_0 are p_1 and p_5. The children of p_1 are p_2 and p_4, the child of p_2 is p_3, and the child of p_5 is p_6. The longest path is $p_3 \to p_2 \to p_1 \to p_0$; the depth is $h = 3$.

Now let each processor hold not a single number but a vector with k components. In the example $k = 3$, and the components are shown in green, blue, and red. Each component is to be summed separately. A message carries a single component, and the rule is this: **a node can receive at most one message per round**; in the same round it may also send one message.

The naive method reduces the components one after another. First the green component is brought up to the root, then the blue, then the red. Each component reaches the root in $h = 3$ rounds; the naive method needs $k \cdot h = 9$ rounds in total. In the **pipelined tree** method, while one component is moving up, the next component sets out from the leaves.

9.7.2 Step-by-step execution

Each component reaches the root in three stages:

1. **Stage 1:** $p_3 \to p_2$, $p_4 \to p_1$, $p_6 \to p_5$.
2. **Stage 2:** $p_2 \to p_1$, $p_5 \to p_0$.
3. **Stage 3:** $p_1 \to p_0$.

p_1 receives messages from its two children: from p_4 in stage 1 and from p_2 in stage 2. p_0 likewise receives from p_5 in stage 2 and from p_1 in stage 3. The stages must therefore take place in separate rounds. There must also be at least two rounds between stage 1 of one component and stage 1 of the next. Otherwise p_1 would have to receive in the same round both the new component from p_4 and the old component from p_2. For this reason a new component sets out every two rounds. Table 9.11 gives the rounds.

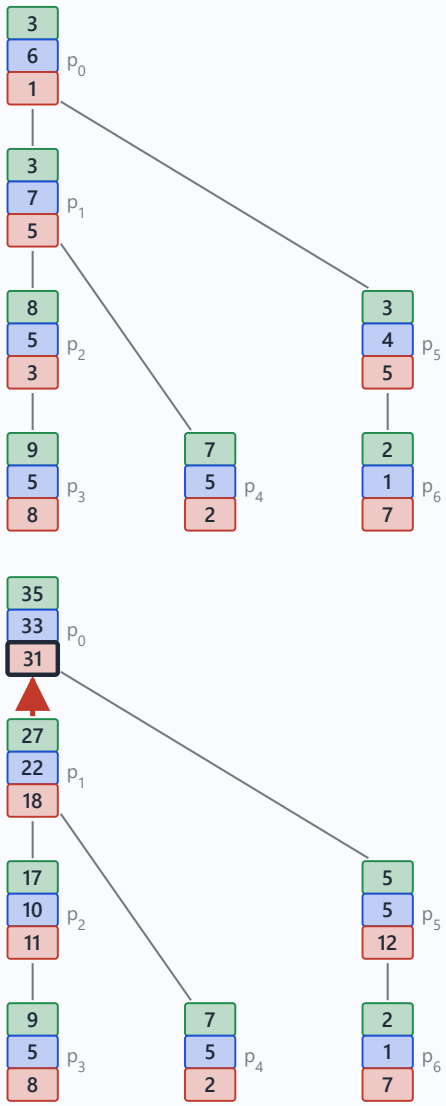


Figure 9.15: The pipelined tree. The first drawing shows the start: each of the seven processors holds a vector of three components (green, blue, red). The second drawing shows the end of round seven: (35, 33, 31) at the root. The color of an edge shows the component carried in that round, and a thick outline shows the value updated in that round.

Round	Stage 1	Stage 2	Stage 3	Additions
1	green	—	—	$p_3 \to p_2$: 9 + 8 = 17; $p_4 \to p_1$: 7 + 3 = 10; $p_6 \to p_5$: 2 + 3 = 5
2	—	green	—	$p_2 \to p_1$: 17 + 10 = 27; $p_5 \to p_0$: 5 + 3 = 8

Round	Stage 1	Stage 2	Stage 3	Additions
3	blue	—	green	green $p_1 \to p_0$: 27 + 8 = 35; blue 5 + 5 = 10, 5 + 7 = 12, 1 + 4 = 5
4	—	blue	—	$p_2 \to p_1$: 10 + 12 = 22; $p_5 \to p_0$: 5 + 6 = 11
5	red	—	blue	blue $p_1 \to p_0$: 22 + 11 = 33; red 8 + 3 = 11, 2 + 5 = 7, 7 + 5 = 12
6	—	red	—	$p_2 \to p_1$: 11 + 7 = 18; $p_5 \to p_0$: 12 + 1 = 13
7	—	—	red	$p_1 \to p_0$: 18 + 13 = 31

Table 9.11: The seven rounds of the pipelined tree. The values were verified with a coded simulation.

Round 3 is the first round in which the pipeline pays off. While the green component is completed at the root, the blue component sets out from the leaves. In that round p_0 receives only from p_1, p_1 only from p_4, p_2 only from p_3, and p_5 only from p_6. The rule is not violated. p_1 both receives (blue, from p_4) and sends (green, to p_0) in the same round.

The result is green 35, blue 33, red 31. To verify, sum all the green components: $3 + 3 + 8 + 9 + 7 + 3 + 2 = 35$. Here the order is $p_0, p_1, \ldots, p_6$.

9.7.3 Analysis

Each component reaches the root in $h = 3$ rounds. A new component starts every two rounds. The last component starts $(k - 1) \cdot 2$ rounds later and finishes in h rounds. The total number of rounds is

$$h + (k - 1) \cdot 2 = 3 + 2 \cdot 2 = 7$$

The naive method needs $k \cdot h = 9$ rounds. The gain looks small, but it grows as k grows. The pipelined tree takes $2k + 1$, that is, about $2k$ rounds; the naive method takes $3k$ rounds. For example, for $k = 10$, 21 and 30 rounds.

That seven rounds is the best result for this tree can be seen with a simple count. The root has two children, and for each of the three components it must receive a message from each child: 6 messages in total. Since the root can receive one message per round, it needs at least 6 rounds of receiving. The first message can arrive at the earliest in round 2: the value p_5 will send must first receive p_6's contribution, and the value of p_1 depends on an even deeper subtree. Six rounds starting from round 2 end at the earliest in round 7.

In general the number of rounds is about $h + (k - 1) \cdot c$, where c is the maximum number of children of a node. A node with many children spends c rounds to receive one component, and it determines the speed of the line. If the tree is a chain ($c = 1$), the formula reduces to $h + k - 1$; this is the same as the pipeline formula $k + m - 1$.

Common mistake

Thinking a component can set out every round. Then the number of rounds comes out as $h + k - 1 = 5$. But if the second component sets out in round 2, p_1 must receive in that round both the blue component from p_4 and the green component from p_2. This breaks the "one message per round" rule. Nodes with two children cut the speed of the line in half.

9.8 Kinds of parallel algorithms and Java

The algorithms in the earlier sections ran on abstract models such as the PRAM or processors exchanging messages. This section builds the same ideas in a real language, Java. Java programs run on a machine with shared memory. Threads can access common objects. Instead of managing threads directly, task-based tools are mostly used.

9.8.1 Three kinds of parallelism

Most parallel algorithms met in practice fit one of three patterns:

- **Divide and conquer.** The problem is split into independent subproblems. The subproblems are solved at the same time on different cores and the results are combined. Parallel merge sort, parallel quicksort, and blocked matrix multiplication follow this pattern.
- **MapReduce.** A large data set is processed on many nodes. The *map* step produces intermediate key–value pairs from each piece; the *reduce* step combines the values of the same key. Hadoop and Spark are common implementations of this pattern. Word counting, building an index for a search engine, and log analysis are typical uses.
- **Data parallelism.** The same computation is applied to different parts of the data at the same time. SIMD instructions and GPUs are designed for this pattern. Scientific simulation, image processing, and numerical computing use it.

The counterpart of the divide-and-conquer pattern in Java is the **Fork/Join** framework. Figure 9.16 shows the idea of the framework. A task **forks** into subtasks. The subtasks run in parallel. The main task **joins**, that is, waits for the subtasks to finish, and combines their results.

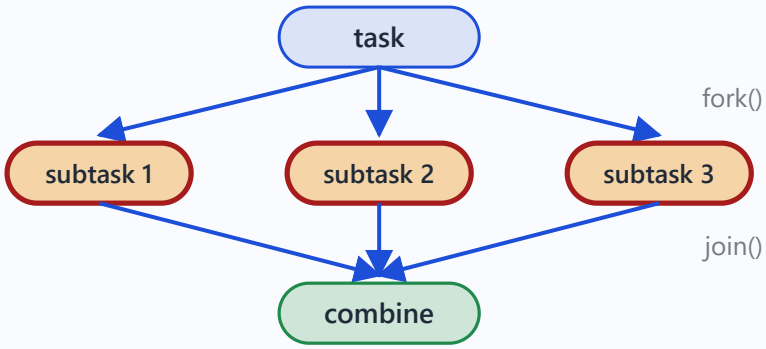


Figure 9.16: The Fork/Join structure. A task forks into three subtasks (fork), the subtasks run in parallel, and the results are combined with join.

9.8.2 Parallel merge sort with Fork/Join

Merge sort is the most familiar example of divide and conquer. The array is split in two, the two halves are sorted separately, and then the two sorted halves are merged. Sorting the two halves is independent; to run them in parallel they only need to be given to separate tasks.

```
class SortTask extends RecursiveTask<int[]> {
  private final int[] array;
  SortTask(int[] array) { this.array = array; }

  protected int[] compute() {
    if (array.length <= 1) return array;
    int mid = array.length / 2;
    var left = new SortTask(
        Arrays.copyOfRange(array, 0, mid));
    var right = new SortTask(
        Arrays.copyOfRange(array, mid, array.length));
    left.fork();              // start in parallel
    right.fork();
    return merge(left.join(), right.join());
  }
}
// usage:
int[] s = new ForkJoinPool().invoke(
    new SortTask(array));
```

The elements of the code are as follows:

- `RecursiveTask<int[]>` is a Fork/Join task that returns a result. The task's work is written in the `compute()` method. For tasks that return no result, `RecursiveAction` is used.
- Base case: an array with one element is already sorted and is returned as is.
- The array is split in two at the midpoint `mid`. `Arrays.copyOfRange` makes a copy of each half, so the two tasks do not share the same array.

- `fork()` hands the subtask to the pool and returns immediately. The task may run on another thread; the calling task continues without waiting. This operation is **asynchronous**.
- `join()` waits for the subtask to finish and returns its result.
- `merge` is the familiar sequential merge that turns two sorted arrays into one sorted array; it takes *O(n)*.
- `new ForkJoinPool().invoke(...)` runs the root task in a pool and waits for its result.

`ForkJoinPool` is a thread pool. Each thread in the pool has its own task queue. A thread puts a task it has `fork()`ed into its own queue. A thread whose queue has run empty does not wait idle; it takes a task from the other end of another thread's queue. This arrangement is called **work stealing**. Work stealing provides load balancing automatically: even when it is not known in advance how long each task will take, an idle core finds work.

Table 9.12 shows what the tasks do level by level for the example array 2 3 5 1 7 6 8 4. Splitting is done from top to bottom, merging from bottom to top.

Level	Operation	Tasks	Number of parallel tasks
0	root	[2 3 5 1 7 6 8 4]	1
1	fork	[2 3 5 1] [7 6 8 4]	2
2	fork	[2 3] [5 1] [7 6] [8 4]	4
3	base	8 single-element arrays	8
4	join + merge	[2 3] [1 5] [6 7] [4 8]	4
5	join + merge	[1 2 3 5] [4 6 7 8]	2
6	join + merge	[1 2 3 4 5 6 7 8]	1

Table 9.12: Task levels in parallel merge sort.

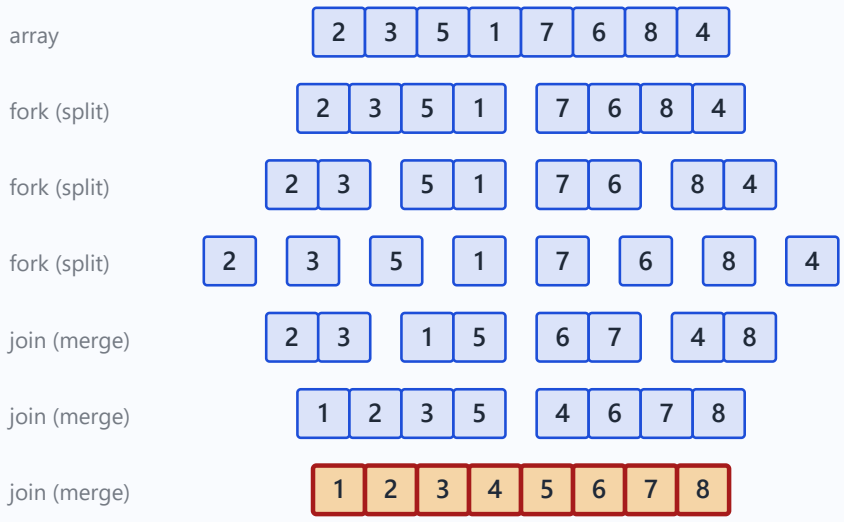


Figure 9.17: Parallel merge sort of the array 2 3 5 1 7 6 8 4. Three levels of splitting (fork), single-element base cases, and three levels of merging (join). The root task does the final merge by itself.

The program was run 11 times on a machine with 12 logical cores under JDK 21 and gave [1, 2, 3, 4, 5, 6, 7, 8] every time. Which thread did the merges, and in what order, varied from run to run. In a series of ten runs, the first merge to finish was [8, 4] five times, [2, 3] twice, [7, 6] twice, and [5, 1] once. Four separate threads carried out the four base merges. The result of a parallel program is deterministic, but the execution order is not. The guarantee for this is that the tasks do not modify shared data: each task works on its own copy and returns its result.

Common mistake

Producing tasks that are too fine-grained. The code creates a new task for every subarray, down to one-element arrays. For an array of a million elements this means about two million tasks. Creating a task, putting it in a queue, and waiting for it are much more expensive than comparing two numbers. In practice a threshold is chosen. For example, pieces below 1000 elements are sorted sequentially with `Arrays.sort`.

A small improvement can be made to the code. Instead of `left.fork(); right.fork();`, write `left.fork(); int[] y = right.compute(); int[] x = left.join();`. The calling task computes the right half itself; only the left half is given to the pool. The calling thread thus does not wait idle at `join()`, and the number of tasks handed to the pool is halved.

9.8.3 Analysis of parallel sorting

The work of parallel merge sort is the same as the sequential version: at each level a total of n elements are merged, and there are $\log_2 n$ levels. $W = \Theta(n \log n)$. The depth is found by a recurrence. Because the two halves are sorted in parallel, only the depth of one of them counts. The merge is sequential and adds $\Theta(n)$ depth:

$$D(n) = D(n / 2) + \Theta(n) = \Theta(n)$$

Unrolling the recurrence gives $n + n/2 + n/4 + \ldots \le 2n$. The parallelism is $W / D = \Theta(n \log n) / \Theta(n) = \Theta(\log n)$.

Version	Work W	Depth D	Parallelism W / D
Sequential merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	1
Parallel recursion + sequential merge	$\Theta(n \log n)$	$\Theta(n)$	$\Theta(\log n)$
Parallel recursion + parallel merge	$\Theta(n \log n)$	$\Theta(\log^3 n)$	$\Theta(n / \log^2 n)$

Table 9.13: Three versions of merge sort.

The parallelism is only $\Theta(\log n)$, which is quite little. For $n = 10^6$, $\log_2 n \approx 20$; with this code, more than about 20 cores bring little benefit. The bottleneck is the final merge done by the root task. That merge walks the whole array alone, and the other cores are idle meanwhile.

To remove the bottleneck, the merge itself is parallelized. The idea is this: the median element of the larger half is chosen, and the position of this element in the other half is found by binary search. The two halves are thus each split into two pieces, and the pieces are merged independently of each other, again in parallel. The depth of this version drops to $\Theta(\log^3 n)$ and the parallelism becomes $\Theta(n / \log^2 n)$. The details are given in the book by Cormen et al. Java also has a ready-made solution: `Arrays.parallelSort(array)` sorts large arrays in parallel with Fork/Join.

9.8.4 MapReduce

MapReduce is a programming model developed at Google in 2004 for processing large data sets on thousands of machines. The programmer writes only two functions; distributing the data, redoing work when something fails, and collecting the results are left to the framework. The model has three steps:

1. **Map.** The input is split into pieces. Each piece is processed independently and produces intermediate key–value pairs. In word counting the pair has the form "word → count".
2. **Shuffle.** Pairs with the same key are sent to the same reducer.
3. **Reduce.** The values of each key are combined; in word counting they are summed.

The example has 13 words: apple, banana, apple, orange, banana, apple, banana, banana, orange, apple, cherry, banana, apple. Four threads are used. The chunk size is $\lfloor 13 / 4 \rfloor = 3$; the first three chunks get three words each, and the last chunk gets the remaining four words. Figure 9.18 shows the flow.

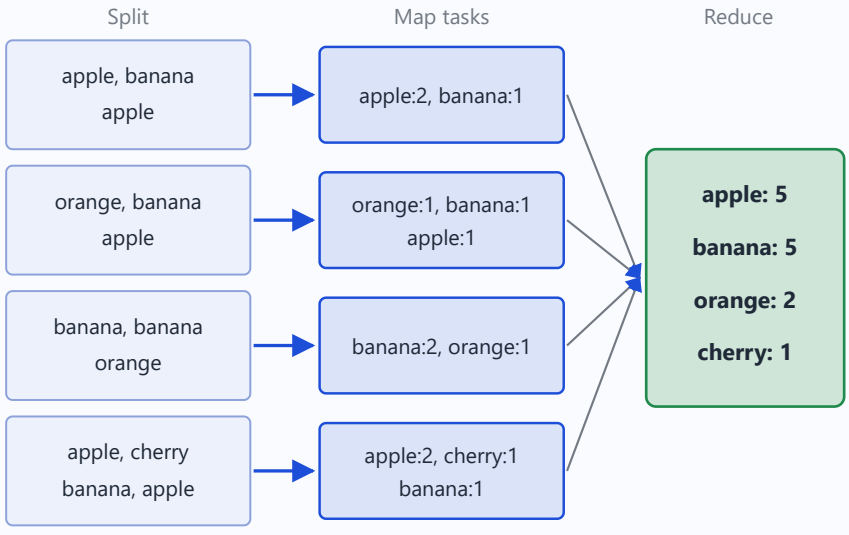


Figure 9.18: Word count with MapReduce. The 13 words are split into four chunks (left column), each map task counts its own chunk (middle column), and reduce combines the results: apple 5, banana 5, orange 2, cherry 1.

In Java the map tasks are written as Fork/Join tasks. Each task takes a list of words (`part`) and returns its own count map.

```
// Map: count the words in one part
protected Map<String, Integer> compute() {
  Map<String, Integer> result = new HashMap<>();
  for (String word : part)
    result.merge(word, 1, Integer::sum);
  return result;
}
// Main flow
for (MapTask t : mapTasks) t.fork();
List<Map<String, Integer>> partial = new ArrayList<>();
for (MapTask t : mapTasks) partial.add(t.join());
// Reduce: combine the partial results
Map<String, Integer> total = new HashMap<>();
for (Map<String, Integer> m : partial)
  m.forEach((k, v) -> total.merge(k, v, Integer::sum));
```

The parts of the code work as follows:

- `compute()` creates a new `HashMap`. This map belongs only to this task.
- `result.merge(word, 1, Integer::sum)` adds the word with value 1 if it is not in the map. If it is, it adds 1 to the old value. In a single line it does "insert if absent, increment if present".
- In the main flow, all map tasks are first `fork()`ed; the tasks start running in parallel in the pool. Then each task's result is taken with `join()` and added to the `partial` list. All tasks are forked before any is joined. Had each task been joined as soon as it was forked, the tasks would have run one after another.
- The reduce step walks over the partial maps in order. Each (key, value) pair is added to the total map with `total.merge(k, v, Integer::sum)`.

Table 9.14 gives the intermediate results and the reduce steps obtained by running the program. The order in which the maps are printed depends on the internal layout of `HashMap`. This order is determined by the hash values of the keys; with the same keys it comes out the same on every run, but it is not alphabetical.

Step	Input	Result
map 0	apple, banana, apple	banana 1, apple 2
map 1	orange, banana, apple	banana 1, orange 1, apple 1
map 2	banana, banana, orange	orange 1, banana 2
map 3	apple, cherry, banana, apple	banana 1, cherry 1, apple 2
reduce partial[0]	empty map + partial[0]	banana 1, apple 2
reduce partial[1]	+ partial[1]	orange 1, banana 2, apple 3

Step	Input	Result
reduce partial[2]	+ partial[2]	orange 2, banana 4, apple 3
reduce partial[3]	+ partial[3]	orange 2, banana 5, cherry 1, apple 5

Table 9.14: Trace of the word count program.

The most important property of this design is **the absence of shared state**. Each map task has its own `HashMap`. The tasks do not touch one another's data; therefore no lock is needed and no race condition can occur. The combining is done in one place after all the tasks have finished. The same idea was present in tree summation: each processor produces its own partial sum, and the partial results are combined afterwards.

In the example, reduce is a single task and it handles all the keys. In a real MapReduce system the keys are divided among many reducers. For the division, *hash(k) mod r* is usually used, where *r* is the number of reducers. The same key always goes to the same reducer. The reduce step thus also runs in parallel.

The chunk sizes in the example are unbalanced: the last task gets four words and the others three each. In a small example this is unimportant. But with millions of words, one task getting much more work than the others makes the other tasks wait for it. Real systems balance the load by splitting the input into many more chunks than there are tasks.

9.8.5 Data parallelism: matrix multiplication

In the product $C = A \cdot B$ of two $n \times n$ matrices, each cell is computed as $C[i][j] = \Sigma_k A[i][k] \cdot B[k][j]$. Each row of the result matrix depends on only one row of A and the whole of B. The rows are independent of one another; each row can be computed as a separate task.

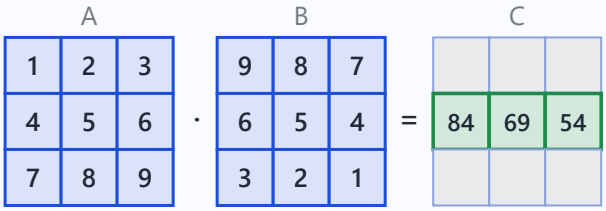
```
// each row is a separate task
class RowTask extends RecursiveAction {
  protected void compute() {
    for (int j = 0; j < n; j++) {
      C[row][j] = 0;
      for (int k = 0; k < n; k++)
        C[row][j] += A[row][k] * B[k][j];
    }
  }
}
try (var pool = new ForkJoinPool()) {
  for (int i = 0; i < n; i++)
    pool.submit(new RowTask(i));
}   // close(): waits for all tasks
```

- `RowTask` is a task that returns no result (`RecursiveAction`). It writes the values it computes directly into the matrix `C`. The field `row` holds which row the task will compute; its constructor is not shown for brevity.
- Two nested loops compute the n cells of a row. Each cell needs n multiplications and additions.
- The `try (var pool = new ForkJoinPool())` block hands one task per row to the pool (`submit`). Since Java 19, `ForkJoinPool` is a closeable resource. The `close()` called when the block ends waits for all the tasks in the pool to finish.

In the example A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] and B = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]. When the program was run it gave C = [[30, 24, 18], [84, 69, 54], [138, 114, 90]]. Table 9.15 shows every operation performed by the row 1 task.

`j`	`k`	**Operation**	`C[1][j]`
0	—	start	0
0	0	+ 4 · 9	36
0	1	+ 5 · 6	66
0	2	+ 6 · 3	84
1	0, 1, 2	4 · 8 + 5 · 5 + 6 · 2 (32, 57, 69)	69
2	0, 1, 2	4 · 7 + 5 · 4 + 6 · 1 (28, 48, 54)	54

Table 9.15: Trace of the row 1 task. Row 1 of A (4, 5, 6) is multiplied by the columns of B.



Rows 0 and 2 are computed in other tasks at the same time.

Figure 9.19: The end of the row 1 task: $C[1]$ = (84, 69, 54). The other rows are computed in other tasks.

The algorithm's work and depth are as follows. Each of the n tasks does $\Theta(n^2)$ operations; the total work is $W = \Theta(n^3)$, the same as the sequential algorithm. Because the tasks run in parallel, the depth is the time of a single task: $D = \Theta(n^2)$. The parallelism is $W / D = \Theta(n)$. More cores than rows bring no benefit to this design. If more parallelism is needed, each cell can be made a separate task, or the matrix can be split into blocks.

There is no race condition in this code, because each task writes only to its own row. A and B are read by all tasks but no task writes to them. Concurrent reading is harmless. Only one task writes to each row of C. As long as the regions written by the tasks are disjoint, a shared matrix can be used safely.

9.9 Shared state and races

In the Java examples so far, the tasks either worked on their own data or wrote to disjoint regions. This was not a coincidence but a deliberate design decision. When several threads modify the same data, the program's behavior becomes unpredictable. This section shows the source of the problem and describes the ways to solve it.

9.9.1 The race condition

Let two threads increment a shared `counter` variable. In Java `counter++` looks like a single statement. At the processor level, though, it is three steps: **read** from memory, **increment** the local copy, **write** to memory. If the steps of the two threads interleave, an increment can be lost. Table 9.16 shows such an interleaving.

Time	Thread 1	Thread 2	`counter`
1	read (0)		0
2		read (0)	0
3	write 0 + 1		1
4		write 0 + 1	1

Table 9.16: A lost update. Initially `counter` = 0; the two threads each increment once.

Both threads read 0 and wrote 1. The expected result is 2, but the counter is 1. The second write overwrote the result of the first. This bug is called a **lost update**.

> **Definition 9.7 (Race condition)**
>
> A **race condition** is a situation in which a program's result depends on the timing of its threads, that is, on the order in which the operating system runs them. At least two threads access the same data without coordination, and at least one of them writes.

To see a race condition in a real program, the following code was written. Each of two threads increments the shared `counter` a million times.

```
public class CounterRace {
  static int counter = 0;                    // shared state

  public static void main(String[] args) throws InterruptedException {
    Runnable task = () -> {
      for (int i = 0; i < 1_000_000; i++)
        counter++;                           // read → increment → write
    };
    Thread t1 = new Thread(task), t2 = new Thread(task);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println(counter);             // expected 2000000
  }
}
```

`Runnable` defines the work a thread will execute; here it is a lambda expression. `start()` starts the thread and returns immediately; from this point the two threads run at the same time. `join()` makes the main thread wait until that thread finishes. `println` therefore runs after both loops have finished.

The program was run 30 times on a Windows machine with 12 logical cores under JDK 21, starting a fresh JVM each time. **The result varied from run to run.** In none of the 30 runs did the expected 2,000,000 appear. The smallest value was 1,049,868 and the largest 1,920,460; in some runs about half of the increments were lost. These numbers will differ on another machine, on another JDK version, or on the same machine at another moment. The one thing that does not change is that the result cannot exceed 2,000,000: a lost update erases an increment but never adds an extra one.

The same experiment was also done with 1000 increments per thread. This time all 20 runs gave the correct result (2000). The reason is that 1000 increments take so little time that the first thread usually finishes before the second one starts. The two threads never overlap. But this does not mean the program is correct. If the machine is busy or the timing changes, the same program can give a wrong result. This is what makes race conditions hard to debug: the bug may not show in testing and appear only in real use, by chance. In 20 runs with 100,000 increments per thread, the results varied between 109,546 and 161,741; the expected value is 200,000.

Example 9.7

Two threads increment the shared `counter` (initially 0) twice each, without coordination. Each increment has the form "read, write": one more than the value read is written. What is the smallest possible result?

1. T1 reads: 0.
2. T2 completes its first increment: reads 0, writes 1. `counter` = 1.
3. T1 completes its first increment: writes one more than the 0 it read, that is, 1. T2's increment is erased. `counter` = 1.

4. T2 reads for its second increment: 1.
5. T1 completes its second increment: reads 1, writes 2.
6. T2 completes its second increment: one more than the 1 it read, 2, is written.

The result is 2; two of the four increments are lost. A program that tries all possible interleavings shows that with two increments each, the result can be 2, 3, or 4. By a similar scenario, as long as each thread increments at least twice, the result can drop to 2 however many increments there are.

9.9.2 Preventing races

There are three basic ways to prevent a race condition.

Locks. Only one thread at a time can enter the block `synchronized (lock) { counter++; }`. The other thread waits until the lock is released. The read, increment, and write steps thus become an indivisible whole. The locked version gave 2,000,000 in all 20 runs.

Atomic operations. The classes in the `java.util.concurrent.atomic` package offer indivisible operations using the processor's special instructions. `AtomicInteger.incrementAndGet()` performs the read, increment, and write steps in a single atomic step.

```
import java.util.concurrent.atomic.AtomicInteger;

public class CounterAtomic {
  static final AtomicInteger counter = new AtomicInteger();

  public static void main(String[] args) throws InterruptedException {
    Runnable task = () -> {
      for (int i = 0; i < 1_000_000; i++)
        counter.incrementAndGet();            // one atomic step
    };
    Thread t1 = new Thread(task), t2 = new Thread(task);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println(counter.get());        // always 2000000
  }
}
```

The atomic version printed 2,000,000 in all 10 separate runs. Concurrent data structures work on the same principle. `ConcurrentHashMap` allows many threads to write to the same map safely.

Avoiding shared state. The best solution is not to use a shared variable at all. Each task produces its own partial result; the results are combined after the tasks finish. Tree summation, MapReduce, and the row tasks work on this principle. In the counter example each thread increments its own local

counter, and after `join()` the two local counters are added. This solution is both correct and the fastest, because the threads never wait.

Locks are not free. In a locked region only one thread runs at a time; the region becomes sequential. By Amdahl's law this sequential part limits speedup. Moreover, when several locks are used, there is a risk of **deadlock**. If T1 holds lock A and waits for B, while T2 holds lock B and waits for A, both wait forever.

> **Common mistake**
>
> Thinking that making the shared counter a `volatile int` solves the race. `volatile` ensures that a value written by one thread is immediately seen by the others. But it does not make the read, increment, and write steps of `counter++` indivisible. Two threads can still read the same value and write the same value. Atomicity requires `AtomicInteger` or a lock.

9.9.3 Concurrent graph traversal

Race conditions arise not only in counters but also in more complex data structures. Consider a parallel version of depth-first traversal. For each unvisited neighbor a new traversal task is `fork()`ed, and the subtrees are traversed in parallel. In the graph in Figure 9.20, the traversal starting from 1 opens two tasks for 2 and 3; these tasks in turn open new tasks for their own neighbors.

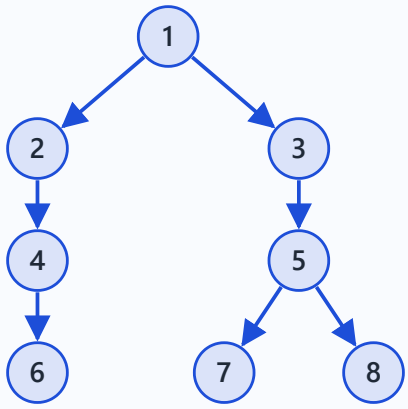


Figure 9.20: Example graph: 1→2, 1→3, 2→4, 3→5, 4→6, 5→7, 5→8.

The tasks share a common set that records which nodes have been visited. There are two separate problems with this set. First, if the set is an ordinary `HashSet`, its internal structure can be corrupted by concurrent writes. Second, even if the set is safe, the question "is it in there?" and the "add" operation are two separate steps. Two tasks can ask about the same node at the same time, both can get the answer "no", and both visit the node. The correct code is this:

```
Set<Node> visited = ConcurrentHashMap.newKeySet();
if (!visited.add(next)) continue; // atomic check + add
```

`ConcurrentHashMap.newKeySet()` returns a set that is safe under concurrent access. The `add` method returns `true` if the element was added and `false` if it was already there. The check and the insertion are a single atomic operation. If two tasks try to add the same node, only one gets `true`, and only that one visits the node.

An experiment was done to see this difference. Each of eight threads tried to "visit" the same 100,000 nodes, 0 to 99,999, in order. Each thread counted a node only if it decided it was the "first visitor". A correct set should give 100,000 visits. Each method was run 20 times, and **the results varied from run to run**:

Set	Check + add	Extra visits	Note
`HashSet`	`contains`, then `add`	44.821 – 147.433	`size()` 104,205 – 137,960; the set was corrupted
`newKeySet()`	`contains`, then `add`	53.122 – 212.552	the set is intact, but nodes were visited more than once
`newKeySet()`	`if (!add(k)) continue`	0 (in all 20 runs)	correct

Table 9.17: Eight threads, 100,000 nodes; 20 runs per method (JDK 21, 12 logical cores).

In the `HashSet` version `size()` exceeded 100,000 although the set can hold only 100,000 distinct numbers. This shows that the set's internal counter also fell into the race. The second row shows that a safe set alone is not enough: the two-step check is still open to a race. Only the atomic check and add in the third row gave the correct result on every run.

Another form of parallel traversal is used on graphs: **level-synchronous breadth-first search**. All the nodes of each level are expanded in parallel, and a barrier is placed at the end of the level. The next level starts after all the nodes of the previous level have been processed. In the ideal PRAM model, with enough processors, the depth is *O(diameter)*: the number of levels does not exceed the diameter of the part reachable from the source. In a realistic model the combining and barrier steps of each level take *O(log n)*; the depth becomes *O(diameter · log n)*. On graphs with a small diameter and many nodes, such as social networks, this method offers a lot of parallelism.

9.9.4 Parallelism tools in Java

Java offers parallelism tools at different levels. Table 9.18 summarizes which to use when.

Tool	When?
`Thread`, `Runnable`	lowest level; manual management, rarely used directly
`ExecutorService`	running independent tasks in a fixed thread pool

Tool	When?
`ForkJoinPool`, `RecursiveTask`	divide and conquer; recursive tasks with work stealing
`list.parallelStream()`	data-parallel map / filter / reduce (uses the common `ForkJoinPool`)
`Arrays.parallelSort`	sorting large arrays in parallel
`Arrays.parallelPrefix`	parallel prefix sum (scan)
`AtomicInteger`, `ConcurrentHashMap`	safe access when a shared counter or set is needed

Table 9.18: Java's parallelism tools.

The tools in the upper rows of the table give more control but need more code. The tools in the lower rows reduce the job to a single method call. For example, word counting fits into a single expression with the stream interface:

```
Map<String, Long> counts = words.parallelStream()
    .collect(Collectors.groupingByConcurrent(w -> w, Collectors.counting()));
```

`parallelStream()` splits the list into pieces and processes them on the common `ForkJoinPool`. `groupingByConcurrent` groups the words in a concurrent map and counts the elements of each group. On the 13-word example this expression gave banana 5, orange 2, apple 5, cherry 1. All the work of the MapReduce code, namely splitting, map, shuffle, and reduce, is done in a single expression. The parallelism level of the common pool is by default one less than the number of cores; on the machine where the experiment was run it was 11.

9.10 Common mistakes

- **Thinking p processors bring a speedup of p.** Because of the sequential part, communication, and waiting, speedup stays below p. If 10% of the program is sequential, by Amdahl's law speedup cannot exceed 10.
- **Computing speedup against the wrong baseline.** T_1 is not the parallel algorithm's time on a single processor but the time of the best sequential algorithm.
- **Using a shared `HashSet` or counter.** Concurrent writing leads to a race condition. Use `ConcurrentHashMap`, `AtomicInteger`, or local results combined afterwards. Even in a safe set, "check, then add" is two steps; look at the return value of `add`, which is a single step.
- **Tasks that are too fine-grained.** Calling `fork()` for every element pushes the task-creation overhead above the gain. Pieces below a threshold should be processed sequentially.
- **Forgetting synchronization between rounds.** In PRAM summation, if round $t + 1$ starts before all the writes of round t have finished, old values are read. In a sequential simulation of Hillis–Steele, the same bug appears when the copy at the start of the round is forgotten.

- **Looking only at depth.** Hillis–Steele finishes in $O(\log n)$ rounds but does $O(n \log n)$ work; with few processors it can be slower than the sequential scan. When choosing an algorithm, work and depth are evaluated together.
- **Miscounting the steps of a pipeline.** The filling time of the line is forgotten. For m rows and p processors the number of steps is not m but $m + p - 2$.
- **Reducing a non-associative operation with a tree.** If subtraction or division is reduced with a tree, the result changes. In floating-point addition too, the parallel result can differ from the sequential result in the last digits.
- **Assuming a parallel program that worked once is correct.** A race condition does not show up on every run. The counter experiment with 1000 increments per thread gave the correct result in all 20 runs; the same code with a million increments gave the correct result in none of 30 runs.

9.11 Summary

- A parallel algorithm divides the problem into pieces, assigns the pieces to processors, solves them at the same time, and combines the results. The price is communication, synchronization, and load balancing. A chain of data dependencies cannot be parallelized; but if the operation in the chain is associative, the chain can be turned into a tree.
- Flynn's classification divides machines into SISD, SIMD, MISD, and MIMD by instruction and data streams. Memory can be shared or distributed. The PRAM is a theoretical model with shared memory and a shared clock; the EREW, CREW, and CRCW variants differ in how they treat concurrent access to the same cell.
- Speedup is $S = T_1 / T_p$, efficiency $E = S / p$, and cost $C = p \cdot T_p$. In a cost-optimal algorithm $C = \Theta(T_1)$. Amdahl's law gives the bound $S \le 1 / (1 - f)$. Gustafson's law says that as the problem grows, the scaled speedup is $(1 - f) + f \cdot p$.
- Work W is the total number of operations, and depth D is the longest dependency chain. $\max(W / p, D) \le T_p \le W / p + D$ (Brent). Parallelism W / D is the maximum number of processors that can be used profitably.
- PRAM tree summation works in $\log_2 n$ rounds with $n - 1$ additions and manages with EREW. With $n / \log n$ processors it becomes cost-optimal.
- For prefix sums, Hillis–Steele spends $\log_2 n$ rounds and $n \log_2 n - (n - 1)$ work; it is not work-efficient. Blelloch finds the exclusive prefix sum with an up-sweep and a down-sweep in $2(n - 1)$ work and $2 \log_2 n$ rounds.
- In a pipeline with p processors and m rows, the number of steps is $m + p - 2$ (naive method $m(p - 1)$), and the efficiency is $m / (m + p - 2)$. In a pipelined tree the number of rounds is $h + (k - 1) \cdot c$; in the example 7 rounds (naive 9).
- In Java, divide and conquer is written with `ForkJoinPool`: `fork()` hands a subtask to the pool, `join()` waits for its result, and work stealing balances the load. Parallel merge sort with a sequential merge has depth $\Theta(n)$ and parallelism $\Theta(\log n)$.

- MapReduce consists of the map, shuffle, and reduce steps. Because each map task uses its own map, no lock is needed. In matrix multiplication with row tasks too, the tasks write to disjoint regions.
- In a race condition the result depends on timing and varies from run to run. The solutions are locks, atomic operations, concurrent data structures, and, best of all, avoiding shared state.

9.12 Key terms

parallel algorithm PRAM model speedup efficiency Amdahl's law work / span prefix sum (scan) pipelining message passing work stealing race condition cost-optimal work-efficient Brent's theorem Gustafson's law reduction barrier lost update deadlock

9.13 Test Yourself

1. $n = 32$ numbers are summed with PRAM tree summation. How many rounds does it take, how many additions are done in the first round, and how many in total? What is the cost with 16 processors; is the algorithm cost-optimal?

   Answer

   $\log_2 32 = 5$ rounds. In the first round 16 additions are done; in total $16 + 8 + 4 + 2 + 1 = 31 = n - 1$. With 16 processors the cost is $16 \cdot 5 = 80$; the sequential algorithm takes 31 steps. The ratio grows like $\log_2 n / 2$ and increases without bound as n grows; the algorithm is not cost-optimal. If the blocks are first summed sequentially with $n / \log_2 n \approx 6$ processors, the cost drops to $\Theta(n)$.

2. 95% of a program can be parallelized. What are the Amdahl speedup and efficiency on 16 processors? What is the limit as the number of processors goes to infinity? What does Gustafson's law give if the problem size grows with the number of processors?

   Answer

   $S = 1 / (0.05 + 0.95 / 16) = 1 / (0.05 + 0.059375) \approx 9.14$. Efficiency $9.14 / 16 \approx 0.57$. Limit $1 / 0.05 = 20$. Gustafson: $0.05 + 0.95 \cdot 16 = 15.25$.

3. A job takes 200 s with the best sequential algorithm. The 10-processor parallel version takes 25 s.

What are the speedup, efficiency, and cost? How many processor-seconds went to overhead?

Answer

$S = 200 / 25 = 8$, $E = 8 / 10 = 0.8$, $C = 10 \cdot 25 = 250$ s. The overhead is $250 - 200 = 50$ processor-seconds; 5 s per processor.

4. The Blelloch algorithm is applied to the array [1, 2, 3, 4]. What is the array after each round of the up-sweep, after zeroing the root, and after each round of the down-sweep? How many operations and rounds are used?

Answer

Up $d = 1$: [1, 3, 3, 7]; $d = 2$: [1, 3, 3, 10]. Root: [1, 3, 3, 0]. Down $d = 2$, pair (1, 3): $t = 3$, `x[1]` = 0, `x[3]` = 0 + 3 → [1, 0, 3, 3]. Down $d = 1$, pair (0, 1): `x[0]` = 0, `x[1]` = 0 + 1; pair (2, 3): `x[2]` = 3, `x[3]` = 3 + 3 → [0, 1, 3, 6]. This is the exclusive prefix sum. $2(n - 1) = 6$ operations, $2 \log_2 4 = 4$ rounds.

5. A chain of three processors has the columns $p_0 = (1, 2, 3, 4, 5)$, $p_1 = (1, 1, 1, 1, 1)$, $p_2 = (2, 2, 2, 2, 2)$. How many steps does the pipeline take, how many does the naive method take, and what is the efficiency? What does p_2 hold at the end, and at which step does row 4 reach p_2?

Answer

$m + p - 2 = 5 + 3 - 2 = 6$ steps; the naive method takes $m(p - 1) = 10$ steps. Efficiency $5 / 6 \approx 0.83$. p_1 becomes (2, 3, 4, 5, 6) and p_2 becomes (4, 5, 6, 7, 8). p_1 sends row 4 at time $t = 4 + 1 = 5$; this is the sixth and last step.

6. Four processors form a chain $p_3 \to p_2 \to p_1 \to p_0$ (a tree with one child per node), and each holds a vector with 5 components. The root is p_0. How many rounds does the pipelined reduction take, and how many does the naive method take?

Answer

$h = 3$, $k = 5$, and every node has at most one child ($c = 1$). A new component can set out every round, because each node receives only one message per round. The number of rounds is $h + (k - 1) \cdot 1 = 7$; the naive method takes $k \cdot h = 15$ rounds.

7. $n = 2^{20}$ numbers are summed with a tree and there are 1024 processors. What are the work, depth, and parallelism? What range does Brent's theorem give for the time; what is the minimum

speedup?

Answer

$W = 2^{20} - 1 = 1{,}048{,}575$, $D = 20$, parallelism $\approx 52{,}429$. Lower bound $W / p \approx 1024$; upper bound $W / p + D \approx 1044$ steps. The speedup is at least $1{,}048{,}575 / 1044 \approx 1004$; almost linear with 1024 processors. When the number of processors is much smaller than the parallelism, the depth term becomes negligible.

8. 1000×1000 matrices are multiplied with row tasks. What are the work, depth, and parallelism? On a machine with 2000 cores, what is the maximum possible speedup?

Answer

$W = n^3 = 10^9$ multiply-adds, $D = n^2 = 10^6$ (the time of a single row), parallelism $W / D = 1000$. Because there are only 1000 tasks, at most half of the cores work; since $T_p \geq D$, the speedup is at most 1000. For more parallelism, cell or block tasks are needed.

9. A student wants to find the sum of a list with a variable `long total = 0;` and `list.parallelStream().forEach(x -> total += x)`. Why is this wrong? Does Java compile this code? How should it be written correctly?

Answer

Java does not compile this code: only effectively final local variables can be used inside a lambda. If the variable is made a field, the code compiles, but `total += x` consists of read, add, and write steps, and lost updates occur between threads. The result varies from run to run. The correct version is a reduction that uses no shared state: `list.parallelStream().mapToLong(Long::longValue).sum()`. Each thread produces its own partial sum, and the partial sums are then combined.

9.14 Further reading

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4th edition, 2022), chapter 26: the notions of work, span, and parallelism and the greedy scheduler bound are treated here. The detailed analysis of parallel merging and parallel merge sort is also in this chapter.
- J. JáJá, *An Introduction to Parallel Algorithms* (1992): a comprehensive source for the PRAM model, cost-optimal tree summation, prefix sums, and the relationships among PRAM variants.
- A. Grama, A. Gupta, G. Karypis, V. Kumar, *Introduction to Parallel Computing* (2nd edition, 2003): covers practical topics such as pipelining, communication costs, scalability, and programming with MPI.

- D. Lea, “A Java Fork/Join Framework” (2000): describes the design of `ForkJoinPool` in Java and its work-stealing mechanism, in the words of its author.

9.15 References

1. G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”, *AFIPS Spring Joint Computer Conference*, 1967.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th edition, MIT Press, 2022 (chapter 26: Parallel Algorithms).
3. J. Dean, S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, *OSDI*, 2004.
4. A. Grama, A. Gupta, G. Karypis, V. Kumar, *Introduction to Parallel Computing*, 2nd edition, Addison-Wesley, 2003.
5. J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
6. D. Lea, “A Java Fork/Join Framework”, *ACM Java Grande Conference*, 2000.

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

activity selection 2
Selecting the largest number of non-overlapping activities by always taking the one that finishes earliest.

adaptive adversary 8
An adversary that observes the algorithm's random decisions and picks the next request accordingly; against it, most of the benefit of randomization is lost.

adversary 8
An imaginary opponent that chooses the input that forces the algorithm into its worst case.

Amdahl's law 9
The principle that the part of a computation that must stay sequential limits the maximum achievable speedup.

amplification 7
A technique that repeats a randomized algorithm independently to reduce its probability of error.

ant colony optimization 6
A method inspired by ants: artificial ants that build short tours deposit more pheromone on their edges, and pheromone evaporates over time.

approximation algorithm 6
An algorithm that produces, in polynomial time, a solution to an NP-hard problem that is provably within a guaranteed ratio of the optimum.

approximation ratio 6
The worst-case ratio between the algorithm's solution and the optimum (ρ).

approximation scheme (PTAS / FPTAS) 6
A family of algorithms that, for every $\varepsilon > 0$, produces a solution within a factor of $(1 + \varepsilon)$ of the optimum.

assignment problem 5
The problem of assigning n jobs to n people, one job per person, at the least total cost.

B

backtracking 4
A search that extends a partial solution step by step and, at a dead end, undoes the last choice and tries another option.

barrier 9
A synchronization point that no processor can pass until all processors have reached it.

Belady's anomaly 8
The phenomenon that, under FIFO, increasing the number of frames can increase the number of page faults.

best-first search 5
A search strategy that expands the node whose bound is the most promising first.

bin packing 2
The problem of packing items of given sizes into the fewest bins of fixed capacity; it is NP-hard.

bound 5
An optimistic estimate of the best value obtainable from a branch.

branch and bound 5
A method that branches over the state space, computes a bound for each branch, and prunes branches whose bound is worse than the best known solution.

branching 5
Splitting a subproblem into smaller subproblems.

C

cache 8
A small, fast memory that holds frequently used data.

canonical coin system 2
A coin system in which the greedy change-making algorithm uses the fewest coins for every amount.

Carmichael number 7
A composite number that passes the Fermat test for every base coprime to it.

chromatic number 4
The smallest number of colors needed to color a graph so that no two nodes joined by an edge share a color.

closest pair of points 1
Finding the two points that are closest to each other among a set of points in the plane.

coin change problem 2
Forming a given amount using the fewest coins.

competitive analysis 8
Comparing an online algorithm against the best offline algorithm (OPT).

competitive ratio 8
The smallest c such that ALG $\le c \cdot$ OPT (+ a constant) holds for every input.

compulsory miss (cold miss) 8
A page fault on the first distinct pages, which no policy can avoid because the cache starts empty.

concept drift 8
A change in the distribution of the data over time; in incremental learning, old data becomes misleading under it.

constraint satisfaction problem 4
The problem of assigning values to variables so that all constraints are satisfied.

convergence 6
The iterates getting steadily closer to the true value.

convex hull 1
The smallest convex polygon that contains a set of points.

D

data stream 8
Continuously arriving data that must be processed in a single pass using little memory.

dead node 5
A node that has been expanded, found infeasible, or pruned because its bound is insufficient; it is never considered again.

decision problem 4
A problem whose answer is yes or no.

depth-first branch and bound 5
A branch and bound that explores live nodes with a stack (depth first), so its memory is limited by the depth.

distributed memory 9
An arrangement in which each node has its own memory and data moves over the network as messages; it scales well but communication is expensive.

divide and conquer 1
An approach that splits a problem into smaller subproblems, solves them recursively, and combines the results.

dynamic programming (DP) 3
A design technique that solves overlapping subproblems once and reuses the stored results.

E

E-node (expanding node) 5
The node whose children, that is, subproblems, are currently being generated.

edge contraction 7
Merging the two endpoints of an edge into a single node (Karger's algorithm).

efficiency 9
Speedup per processor: speedup / p.

entropy 2
The average amount of information per symbol of a source; a lower bound on the average length of a prefix-free code.

eviction policy 8
The rule that decides which page to remove when the cache is full (FIFO, LRU, OPT).

exchange argument 2
Proving correctness by transforming an optimal solution so that it contains the greedy choice.

expected running time 7
The expected value of a randomized algorithm's running time on a given input.

expected value 7
The average of a random variable weighted by probabilities.

F

feasible 5
Satisfying all the constraints (said of a solution).

Fisher–Yates shuffle 7
An O(n) shuffle that produces every permutation with equal probability.

forward checking 4
Immediately removing conflicting values from the possible values of connected variables when a value is assigned to a variable.

fractional knapsack 2
A problem in which items can be taken fractionally, solved greedily by value-to-weight ratio.

G

genetic algorithm 6
A method that works with a population of solutions: good solutions are selected more often and combined by crossover, and mutation preserves diversity.

geometric distribution 7
The distribution of the number of trials up to the first success when each trial succeeds with probability p.

global optimum 6
The best of all solutions; local search stops at a local optimum and may fail to reach it.

graph coloring 4
Coloring nodes so that adjacent nodes receive different colors.

greedy algorithm 2
An algorithm that makes the choice that looks best at the moment at every step and never goes back.

greedy-choice property 2
The property that a locally best choice can lead to an optimal solution.

H

Hamiltonian cycle 6
A cycle that passes through every node exactly once and returns to the start.

Hamiltonian path 4
A path that passes through every node exactly once.

hit (cache hit) 8
Finding the requested page in the cache; it costs 0. If it is not found, a page fault occurs.

Huffman coding 2
An optimal prefix-free coding that gives short codes to frequent characters.

Hungarian method 5
An algorithm that solves the assignment problem in $O(n^3)$ time.

I

incremental learning 8
Updating the model with each new piece of data without reprocessing the past.

incumbent 5
The value of the best complete solution found so far in branch and bound.

indicator random variable 7
A random variable that takes the value 1 if an event occurs and 0 if it does not.

integer linear programming (ILP) 5
Linear programming in which the variables must be integers.

J

job sequencing with deadlines 2
Choosing, from jobs that have deadlines, the schedule with the greatest total profit.

K

Karatsuba multiplication 1
A method that multiplies large numbers with three subproducts in O(n^1.585) time.

#

0/1 knapsack 3
A problem in which each item is either taken whole or not at all, maximizing value without exceeding the capacity.

L

Las Vegas algorithm 7
An algorithm that always gives the correct answer but whose running time is random.

least recently used (LRU) 8
A policy that, when the cache is full, removes the page that has gone unused for the longest time; its competitive ratio is k.

least-cost (LC) search 5
A search that expands the live node with the smallest estimated cost $\hat{c} = f + \hat{g}$.

linear congruential generator (LCG) 7
The oldest and simplest pseudo-random number generator, producing the next number by $(a \cdot x + c) \bmod m$; its low bits have a short period.

list scheduling 6
Greedy scheduling that assigns each job to the machine that becomes free earliest.

list update problem 8
The problem of reducing access cost by reordering a list according to the items accessed.

live node 5
A node generated but not yet expanded in branch and bound that is still promising; live nodes wait in a queue structure.

local optimum 6
A solution none of whose neighboring solutions is better.

local search 6
A search that starts from a solution and improves it with small changes (for example 2-opt).

locality of reference 8
The tendency of programs to access memory locations they used in the recent past rather than random ones; this is what makes LRU better than FIFO.

longest common subsequence (LCS) 3
The longest sequence of characters that appears in the same relative order in two sequences, not necessarily contiguously.

longest common substring 3
The longest piece that appears contiguously in both strings.

longest increasing subsequence (LIS) 3
The longest increasing sequence that can be selected from a sequence while preserving the order of its elements.

longest processing time first (LPT) 6
A scheduling rule that sorts jobs from largest to smallest and assigns each to the least loaded machine.

lost update 9
When two threads read and write the same value and one write overwrites the other.

LP relaxation 5
Obtaining a bound by dropping the integrality condition and solving the resulting linear program.

M

makespan 6
The moment the last job in a schedule finishes (C_max).

marking algorithm 8
An online caching policy that divides the sequence into phases and evicts only unmarked pages in each phase.

Markov's inequality 7
The inequality $P(X \geq a) \leq E[X] / a$ for a non-negative X.

matching 6
A set of edges no two of which share an endpoint.

matrix chain multiplication 3
The problem of finding the parenthesization that multiplies a sequence of matrices with the fewest scalar multiplications.

matroid 2
A set structure that guarantees that the greedy algorithm finds the best solution.

maximum clique 5
The largest set of nodes in which every two nodes are adjacent.

maximum coverage 2
The problem of covering as many elements as possible by choosing at most k sets; the greedy choice achieves at least $1 - 1/e$ of the best.

median 8
The middle value of sorted data.

memoization 3
Storing and reusing results computed in top-down recursion.

message passing 9
Information that processors send to exchange data in distributed memory.

metaheuristic 6
A general search strategy for escaping local optima (simulated annealing, genetic algorithm).

min-conflicts heuristic 4
A local search that gives a conflicting variable the value that creates the fewest conflicts.

minimum cut 7
A cut that splits a graph into two parts with the fewest edges (or the smallest capacity).

minimum remaining values (MRV) 4
A heuristic for constraint problems that chooses first the variable with the fewest remaining possible values.

Monte Carlo algorithm 7
An algorithm with bounded running time whose answer may be wrong with a small probability.

move-to-front (MTF) 8
A rule that moves the accessed item to the front of the list; it is 2-competitive.

moving average 8
Updating the average of the last k values with each new value.

N

N-queens problem 4
Placing n queens on an n × n board so that none threatens another.

nearest neighbor heuristic 6
A TSP heuristic that goes to the nearest unvisited city at each step.

NP-complete 4
A problem that is both in NP and NP-hard.

NP-hard 4
A problem to which every problem in NP can be reduced in polynomial time (at least as hard as NP).

O

oblivious adversary 8
An adversary that knows the code of a randomized algorithm but cannot see its coin flips, and therefore fixes the request sequence in advance.

offline algorithm 8
An algorithm that knows the entire input in advance.

online algorithm 8
An algorithm that receives the input piece by piece and makes a decision at each step without knowing the future.

optimal 6
The best according to the objective function (said of a solution).

optimal substructure 3
The property that an optimal solution can be built from optimal solutions to its subproblems.

optimization problem 4
A problem whose answer is the best value rather than yes or no; it can often be converted into a decision problem.

overlapping subproblems 3
The same subproblem appearing again and again in the recursion.

P

page fault 8
The requested page not being in the cache.

paging 8
The problem of deciding which pages to keep in a limited cache.

parallel algorithm 9
An algorithm that splits the work among several processors running at the same time.

partial solution 4
A solution built step by step in backtracking that is not yet complete.

pipelining 9
Splitting the work into stages and processing different data in the stages at the same time.

pivot 7
The element chosen to split the array into two parts in partitioning.

polynomial time 4
Running time $O(n^k)$ for some constant k.

power iteration 6
A method that repeatedly multiplies a vector by a matrix so that it converges to the dominant eigenvalue and eigenvector.

PRAM model 9
An abstract model made of processors that share a common memory and run in synchronous steps.

prefix sum (scan) 9
At each position, the sum of the elements up to that position; computable in parallel with O(log n) depth.

prefix-free code 2
A code in which no codeword is a prefix of another codeword.

primality test 7
Testing whether a number is prime (Miller–Rabin).

pruning 4
Removing from the search a branch that is known not to lead to a solution.

pseudo-polynomial 3
Running time that is polynomial in the numeric value in the input (for example W) rather than in the input length.

pseudo-random number 7
A number that looks random but is produced by a deterministic generator.

Q

quickselect 7
Finding the k-th smallest element in expected O(n) time with a random pivot.

R

race condition 9
A situation where the result depends on the order in which threads run.

randomized algorithm 7
An algorithm that makes random choices in its steps.

reduced cost matrix 5
A cost matrix from which row and column minima have been subtracted; the total subtracted is a lower bound for the TSP.

reduction 6
Transforming instances of one problem into instances of another problem.

reservoir sampling 8
Maintaining a uniform random sample of k items from a stream of unknown length.

running median 8
Keeping the median of the values in a stream up to date with two heaps.

S

satisfiability (SAT) 4
The problem of asking whether some assignment of values makes a formula built from Boolean variables true; it is the first NP-complete problem.

scheduling 6
The problem of assigning jobs to machines or to time slots.

seed 7
The starting value of a pseudo-random number generator.

self-organizing list 8
A list that adapts itself to usage by moving accessed items toward the front with a rule such as move-to-front or transpose.

shared memory 9
An arrangement in which all processors see the same address space; communication is cheap but concurrent access must be controlled.

simulated annealing 6
A metaheuristic that accepts worsening moves with a decreasing probability.

ski rental problem 8
The problem of choosing between renting and buying without knowing the future.

sliding window 8
A method that keeps a running total in a variable and, at each step, adds the new element and removes the oldest to update the value over the last k elements.

speedup 9
Sequential running time divided by parallel running time: T_1 / T_p.

stack algorithm 8
A page replacement policy under which the set of pages in a larger memory always contains the set in a smaller one; it does not exhibit Belady's anomaly.

state-space tree 4
A search tree in which partial solutions are nodes and choices are edges.

Strassen's algorithm 1
A method that multiplies matrices with seven subproducts instead of eight, in O(n^2.81) time.

strong liar 7
For a composite number, a base that is not a witness and so makes it look prime.

subproblem 3
A smaller instance of the original problem.

subset sum 3
The problem of deciding whether some subset has a sum equal to a given value.

T

tabulation 3
Filling in the subproblems in a table from smallest to largest (bottom-up).

throughput 9
The number of jobs completed per unit of time; in a pipeline it grows by a factor equal to the number of stages.

top-down / bottom-up 3
The two forms of DP: recursion with memoization, or filling a table with a loop.

traceback (solution reconstruction) 3
Finding the solution itself by walking backward through the filled DP table.

transpose (heuristic) 8
A list reordering rule that swaps the accessed item with the one before it.

travelling salesman problem (TSP) 6
Finding the shortest tour that visits every city once and returns to the start.

triangle inequality 6
The condition d(a, c) ≤ d(a, b) + d(b, c) for every three points.

U

unbounded knapsack 3
A knapsack problem in which each item can be taken any number of times.

up-sweep / down-sweep 9
The two phases of Blelloch's prefix sum: the up-sweep builds partial sums from the leaves to the root, and the down-sweep carries to each node the sum of the nodes to its left, from the root to the leaves.

V

vertex cover 6
A set of nodes that contains at least one endpoint of every edge.

W

witness 7
A base that proves a number is composite (Miller–Rabin).

work / span 9
The total number of operations (T_1) and the longest chain of dependencies ($T\infty$).

work stealing 9
An idle thread taking work from another thread's queue.

work-efficient 9
A parallel algorithm whose work is of the same order as the running time of the best sequential algorithm for the same problem.

APPENDIX B

Prerequisites for Advanced Algorithms

ADVANCED ALGORITHMS · PREREQUISITES

Advanced Algorithms is the continuation of the Algorithms course; it assumes you know complexity analysis and sorting, searching, graph, and string algorithms. The course combines programming in Java, basic data structures, discrete mathematics, proof techniques, sums and logarithms, probability, and counting. The 20 topics on this page gather the background the course builds on into 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 reveals its answer.

Java and programming (4) **Basic data structures (3)**

Discrete mathematics and proof (2) **Analysis tools (2)**

Probability and counting (3) **Other mathematics and concepts (6)**

Java and programming

What you need to read and trace the code in the slides and to write solutions in Java.

[ ]

Arrays and two-dimensional arrays

indices from 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 · Table-Filling Trace**

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
```

↻

Recursion and the call stack

base case · recursive call · depth

Merge sort, quicksort, DFS, backtracking, and divide-and-conquer algorithms are all 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 1 · Fast Exponentiation: Trace** **Chapter 4 · General Template**

CHECK QUESTION

▶ **In the call power(2, 10), counting the first call, how many times does the power method run?**

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 begin with a sort; Dijkstra and Prim put comparable objects into a priority queue.

WHERE IT IS USED **Chapter 2 · The Activity-Selection Problem**

CHECK QUESTION

▶ **Why is it risky to write return w - o.w; inside compareTo?**

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, and HashMap for memoization. Knowing the cost of each basic operation is a prerequisite for computing the total running time of an algorithm correctly.

WHY IT MATTERS

BFS is written with a queue, Dijkstra with a priority queue, and memoization with a map.

WHERE IT IS USED **Chapter 3 · Memoization 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
```

Basic data structures

The structures that algorithms operate on, and the cost of their basic operations. These structures are covered in detail in the prerequisite Data Structures course.

→ Linked lists

node · next · O(1) insertion at the head

Like an array, it stores a sequence, but the elements are not adjacent 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 all rely on linked lists.

WHY IT MATTERS

Hash tables with chaining and the O(1) LRU cache use linked lists.

WHERE IT IS USED **Chapter 8 · LRU: O(1) Implementation**

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 + " ");
```

▲ 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 extract-min take O(log n); heap sort, Dijkstra, Prim, Huffman, and the online median all rely on this structure.

WHY IT MATTERS

Heap sort, the complexity of Dijkstra's algorithm, the construction of the Huffman tree, and the online median all rest on the heap.

WHERE IT IS USED **Chapter 2 · Huffman: Steps** **Chapter 8 · Running Median**

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   (indices 1, 2)
children of 5: 9 and 6   (indices 3, 4)
child of 3:    4         (index 5)
```

A Trees and binary search trees

root · parent · child · leaf · height

A tree with n nodes has n − 1 edges, and there is exactly one path between any two nodes. The terms root, parent, child, leaf, depth, and height are used unchanged for 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

BFS trees, spanning trees, tries, and state-space trees are all described with the same terms.

WHERE IT IS USED **Chapter 4 · State-Space Tree**

CHECK QUESTION

▶ **What is the maximum number of nodes in a binary tree of height h?**

EXAMPLE

```
        8            height 2; leaves 1, 6, 14
      /   \
     3     10        parent of 3 is 8, children 1 and 6
    / \      \
   1   6      14     inorder traversal: 1 3 6 8 10 14
```

Discrete mathematics and proof

The language used to define algorithms and to 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 a knapsack. You should know union, intersection, difference, the Cartesian product, and subsets; the domain and codomain of a function; and the reflexive, symmetric, and transitive properties of relations.

WHY IT MATTERS

The cut property, the transitive closure in Floyd-Warshall, and the feasible solutions in branch and bound are all defined with sets.

WHERE IT IS USED **Chapter 5 · Basic Concepts**

CHECK QUESTION

▶ **How many subsets does a set of n elements have, and which brute-force method costs that much?**

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.
```

∴ Proof techniques: direct proof, contradiction, counterexample

direct proof · proof by contradiction · counterexample

We prove that an algorithm works on every input with a proof, and show that it is wrong with a single counterexample. The correctness of greedy algorithms is usually proved by contradiction or by an exchange argument; the claim “greedy change-making always gives the best answer” is refuted by a counterexample.

WHY IT MATTERS

The correctness of greedy choices, the cut property of the MST, and 2-approximation ratios are all shown by proof.

WHERE IT IS USED **Chapter 2 · The Change-Making Problem**

Chapter 2 · Why Earliest Finish? **Chapter 6 · Proof of the 2-Approximation**

CHECK QUESTION

▶ **Which technique is used to prove that $\sqrt{2}$ is irrational?**

EXAMPLE

```
Claim: greedy change-making uses the fewest coins in every coin system.
Counterexample: coins {1, 3, 4}, amount 6
  greedy: 4 + 1 + 1  → 3 coins
  best:   3 + 3      → 2 coins        ⇒ the claim is false.
```

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

Loop counting, summing the levels of a recursion tree, and the expected number of comparisons of randomized quicksort all use these series.

WHERE IT IS USED **Chapter 7 · Expected Number of Comparisons**

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 for 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 the exponent n^1.585 of Karatsuba are all computed with logarithms.

WHERE IT IS USED **Chapter 1 · Karatsuba Analysis**

CHECK QUESTION

▶ **About how many comparisons does binary search make at most on a sorted array of one 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 only changes a constant factor
```

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 the TSP, and the size of the n-queens search tree are all found by counting.

WHERE IT IS USED **Chapter 6 · Comparison with Brute Force**

Chapter 4 · Brute Force vs. Pruning

CHECK QUESTION

▶ **In how many ways can 8 queens be placed on an 8 × 8 board with one queen in each row? And 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
```

P

Probability basics

sample space · independence · conditional probability

Computing the success and error probabilities of randomized algorithms requires the sample space, events, complements, independent events, and conditional probability. An important pattern: if a single trial fails with probability p, all k independent trials fail with probability p^k; this is how the error of Monte Carlo algorithms is reduced.

WHY IT MATTERS

The error analysis of the Miller–Rabin test and of Karger's algorithm uses exactly this pattern.

WHERE IT IS USED **Chapter 7 · Probability Review**

Chapter 7 · Reducing the Error Probability **Chapter 7 · Karger's Minimum Cut**

CHECK QUESTION

▶ **An algorithm finds the correct answer with probability 1/2 in a single run. What is the probability that at least one of 10 independent runs finds the correct answer?**

EXAMPLE

```
P(A ∪ B) = P(A) + P(B) − P(A ∩ B)
A and B independent  ⇔  P(A ∩ B) = P(A) · P(B)
```

```
Miller–Rabin: error per round ≤ 1/4  ⇒  after 20 rounds ≤ (1/4)^20 ≈ 9 ·
10^−13
```

E Expected value and linearity of expectation

random variable · E[X] · indicator variable

Results such as the expected O(n log n) running time of randomized quicksort are stated as expected values. Linearity of expectation, E[X + Y] = E[X] + E[Y], holds even when the variables are dependent; splitting a complicated count into a sum of 0/1 indicator variables and adding their probabilities is a standard technique.

WHY IT MATTERS

The analyses of randomized quicksort, QuickSelect, and randomized paging are done with expected values.

WHERE IT IS USED

Chapter 7 · Expected Number of Comparisons

Chapter 7 · Analysis of Quicksort **Chapter 8 · Randomized Paging**

CHECK QUESTION

▶ **The worst case of randomized quicksort is $O(n^2)$. Why do we say "expected O(n log n)"?**

EXAMPLE

```
Die: E[X] = (1 + 2 + … + 6) / 6 = 3.5
In a random permutation of n elements, let X be the number of elements that
stay in place:
  X = X1 + … + Xn,  Xi = 1 (if element i stays in place),  P(Xi = 1) = 1/n
  E[X] = n · (1/n) = 1
Repeating a trial with success probability p until the first success: E[number
of trials] = 1/p
```

Other mathematics and concepts

Matrices, modular arithmetic, bit operations, optimization, and the basics of concurrency used in particular chapters.

M Matrix operations

row–column product · identity matrix · block form

Multiplying two n × n matrices directly takes n^3 multiplications in three nested loops; Strassen reduces this number with block matrices. For adjacency matrices, the distance table of Floyd-Warshall, 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 · Matrix Multiplication Chapter 6 · Power Iteration

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 only horizontal and vertical steps, is $|x_1 - x_2| + |y_1 - y_2|$. If a distance measure satisfies the triangle inequality, d(a, c) ≤ d(a, b) + d(b, c): visiting a third point never shortens the path. 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), can be computed with integers; its sign tells whether P lies to the left of the 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 the Euclidean distance, and the 2-approximation for metric TSP on the triangle inequality.

WHERE IT IS USED **Chapter 1 · Orientation Test** **Chapter 1 · Closest Pair of Points**

Chapter 6 · Metric TSP

CHECK QUESTION

▶ **What are the Euclidean and Manhattan distances for A = (0, 0) and B = (3, 4)? On which side of the line AB is the point P = (1, 5)?**

EXAMPLE (JAVA)

```
static double euclidean(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(euclidean(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, the rolling hash of Rabin-Karp, pseudo-random number generators, and the Miller–Rabin test all work with modular arithmetic. You should 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, the rolling hash, Miller–Rabin, and pseudo-random numbers use modular arithmetic.

WHERE IT IS USED **Chapter 7 · Miller–Rabin: Code**

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, the O(1) safety test and, for small n, the enumeration of all subsets 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 4 · O(1) Safety Test

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
```

opt Decision and optimization problems

objective function · constraint · feasible solution · lower/upper bound

Greedy, dynamic programming, approximation, and branch-and-bound algorithms solve optimization problems: among the solutions that satisfy the constraints (the feasible solutions), we look for the one that maximizes or minimizes the objective function. The decision version of the same problem ("is there a solution with value at least K?") is used in the discussion of P, NP, and NP-completeness. In a maximization problem, the value of every feasible solution is a lower bound on the optimum.

WHY IT MATTERS

The knapsack problem, NP-hardness, approximation ratios, and the bounds in branch and bound are all expressed in this language.

WHERE IT IS USED **Chapter 3 · The 0/1 Knapsack Problem** **Chapter 4 · P, NP and NP-Hard** **Chapter 5 · Choosing a Bound**

CHECK QUESTION

▶ **In a maximization problem, a greedy algorithm finds a feasible solution of value 40, and the value of the fractional relaxation is 46. In what range does the optimum lie?**

EXAMPLE

```
0/1 knapsack, capacity W = 5
  items (weight, value): 1 = (2, 3), 2 = (3, 4), 3 = (4, 5)
  feasible: {1, 2} → weight 5, value 7      not feasible: {2, 3} → weight 7
  optimization: what is the largest value?             → 7
  decision: is there a feasible choice with value ≥ 8? → no
```

‖ Concurrency basics

thread · shared variable · synchronization

The chapter on parallel algorithms uses Java to show how work is divided among several threads and how the results are combined. An intuitive grasp is enough: what a thread is, that two threads updating the same variable without protection cause a race condition, and that the sequential part of a program limits the speedup.

WHY IT MATTERS

Race conditions, parallel sorting with fork/join, and Amdahl's law all build on these ideas.

WHERE IT IS USED **Chapter 9 · Race Conditions** **Chapter 9 · Fork/Join Step by Step** **Chapter 9 · Amdahl's Law**

CHECK QUESTION

▶ **If 20% of a program remains sequential, what is the largest speedup that can be obtained with an unlimited number of processors?**

EXAMPLE (JAVA)

```
final AtomicInteger counter = new AtomicInteger();
Runnable job = () -> { for (int i = 0; i < 100_000; i++)
counter.incrementAndGet(); };
Thread t1 = new Thread(job), t2 = new Thread(job);
t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(counter.get());   // always 200000; an unprotected int++
may give less
```

APPENDIX C

Important People in Advanced Algorithms

ADVANCED ALGORITHMS · PEOPLE

The 43 people behind the algorithms and ideas in the nine chapters of the course: from Karatsuba multiplication and Huffman coding to Bellman's dynamic programming, Cook's theorem on NP-completeness, the branch-and-bound method, competitive analysis, and parallel computing.

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.

LF

Leonardo Fibonacci

c. 1170 – c. 1250 · Italian · Pisa

Liber Abaci · 1202

With Liber Abaci (1202) he introduced the Hindu-Arabic numerals to Europe; the rabbit problem in the book produces 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 for memoization.

IN THE COURSE **Chapter 3 · Fibonacci: Plain Recursion** **Chapter 3 · Repeated Work**

recurrence relation overlapping subproblems

PF

Pierre de Fermat

d. 1665 · French · Toulouse

Fermat's little theorem · 1640

In a 1640 letter to Frénicle de Bessy he stated the result now known as the little theorem: if n is prime and 1 < a < n, then $a^{n-1} \equiv 1 \pmod{n}$. Euler gave the first published proof in 1736. The Fermat primality test checks this congruence for randomly chosen bases; if it fails, the number

is certainly composite. The converse does not always hold: Carmichael numbers such as 561 = 3 · 11 · 17 satisfy the congruence for every base coprime to them. Robert Carmichael published 561 in 1910. The Miller–Rabin test closes this gap with a square-root property.

IN THE COURSE **Chapter 7 · Miller–Rabin Primality Test**

Chapter 7 · Miller–Rabin Examples

Fermat test Carmichael numbers modular arithmetic

LE

Leonhard Euler

1707–1783 · Swiss · St. Petersburg, Berlin

Bridges of Königsberg · 1736

In 1736 he proved that no walk crosses each of the seven bridges of Königsberg 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. An Euler tour, which uses every edge once, is also a step of Christofides' algorithm.

IN THE COURSE **Chapter 6 · Metric TSP**

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 appears 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; Karatsuba's method applies a similar idea to large integers.

IN THE COURSE **Chapter 1 · Big-Number Multiplication**

arithmetic series loop counting

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 4 · P, NP and NP-Hard**

computability halting problem Church–Turing thesis

FY Ronald A. Fisher and Frank Yates

1890–1962 · 1902–1994 · British · Rothamsted

Fisher–Yates shuffle · 1938

In the 1938 book Statistical Tables for Biological, Agricultural and Medical Research they described how to generate a random permutation of a list by hand, with the help of a table of random numbers. Richard Durstenfeld adapted the method to computers in 1964, giving the O(n) swap-based version that Knuth popularized. Fisher is one of the founders of experimental design and of modern statistics.

IN THE COURSE **Chapter 7 · Example: Shuffling** **Chapter 7 · Fisher–Yates Trace**

Fisher–Yates shuffle uniformly distributed permutation

JN John von Neumann

1903–1957 · Hungarian-born American · Princeton IAS

Merge sort · 1945

In 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. With Stanisław Ulam he contributed to the development of the Monte Carlo method, and today's computer architecture bears his name.

IN THE COURSE **Chapter 1 · Merge Sort** **Chapter 7 · Monte Carlo estimate of π**

divide and conquer stable sorting Monte Carlo

SU

Stanisław Ulam

1909–1984 · Polish-born · USA, Los Alamos

Monte Carlo method · 1946–1949

In 1946 at Los Alamos, starting from the idea of estimating the chance of winning a game of solitaire by running random trials, he proposed the Monte Carlo method; together with von Neumann he applied it to neutron diffusion calculations on the ENIAC. The first paper to carry the method's name appeared in 1949, with Nicholas Metropolis. Estimating π with random points is the simplest example used in the course.

IN THE COURSE **Chapter 7 · Monte Carlo estimate of π**

Chapter 7 · Las Vegas and Monte Carlo

Monte Carlo random sampling

GD

George B. Dantzig

1914–2005 · American · RAND, Stanford

Simplex method · 1947

In 1947 he developed the simplex method for solving linear programs; in branch-and-bound algorithms, the bound at each node is most often computed from such an LP relaxation. In 1954, with Fulkerson and Johnson, he solved a 49-city traveling salesman problem (TSP) to optimality using cutting planes. The story that, as a student, he mistook two unsolved statistics problems for homework and solved them is also famous.

IN THE COURSE **Chapter 5 · Integer Programming** **Chapter 5 · TSP: Lower Bound**

linear programming simplex TSP

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 one of the foundations of digital circuit design.

IN THE COURSE **Chapter 2 · Codes and Savings**

entropy compression bound

DL

Derrick H. Lehmer

1905–1991 · American · UC Berkeley

Lehmer generator (1949) · the term “backtrack”

In 1949 he introduced the pseudo-random number method now known as the Lehmer generator, a multiplicative linear congruential generator $x(k+1) = a \cdot x(k) \bmod m$. In the 1950s he used the term “backtrack” for a search that extends a partial solution and steps back when it reaches a dead end; today this approach is known as backtracking. He also did pioneering work on primality tests and calculating machines in number theory.

IN THE COURSE **Chapter 7 · Pseudorandom Numbers** **Chapter 4 · Go Back and Try Again**

pseudo-random numbers backtracking

DH

David A. Huffman

1925–1999 · American · MIT, UC Santa Cruz

Huffman coding · 1952

As a graduate student at MIT, in Robert Fano's course, he solved the problem of finding the shortest prefix code as a term paper that took the place of the final exam, and published the result in 1952. The greedy method that repeatedly merges the two nodes of lowest frequency minimizes the expected number of bits per character and surpasses his professor's Shannon–Fano method. Huffman coding is a stage of formats such as DEFLATE and JPEG today.

IN THE COURSE **Chapter 2 · Huffman Coding** **Chapter 2 · Why Is Huffman Optimal?**

Huffman coding prefix code greedy choice

RB

Richard Bellman

1920–1984 · American · RAND Corporation

Dynamic programming · 1950s

In the 1950s at the RAND Corporation he introduced the name “dynamic programming” and the principle of optimality for multistage decision problems: 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 3 · What Is Dynamic Programming?**

Chapter 3 · Two Key Properties

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 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 1967 conference talk he argued that the sequential part of a program limits parallel speedup; this observation is known as Amdahl's law.

IN THE COURSE **Chapter 9 · Amdahl's Law**

linear probing speedup Amdahl's law

MR

Michael O. Rabin

b. 1931 · Israeli · Hebrew University of Jerusalem; Harvard

Miller–Rabin (1980) · Rabin-Karp (1987)

In 1959, with Dana Scott, he defined nondeterministic finite automata; this work earned him 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 one 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 among the pioneers of deliberately using randomness in algorithm design.

IN THE COURSE **Chapter 7 · Miller–Rabin Primality Test**

primality test rolling hash finite automata

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. The same year he published the Find method for the k-th smallest element (today's QuickSelect); he also founded Hoare logic for program correctness and received the Turing Award in 1980.

IN THE COURSE **Chapter 7 · Randomized Selection (Quickselect)**

quicksort partitioning QuickSelect

KA Anatoly Karatsuba

1937–2008 · Soviet · Moscow

Karatsuba multiplication · 1960/1962

In 1960, while Kolmogorov's seminar was discussing the conjecture that multiplying two n-digit numbers requires $\Omega(n^2)$ operations, he found within a week a method that uses three subproducts and runs in O(n^1.585) time; he was 23 at the time. Kolmogorov had the result published in 1962 under the names Karatsuba and Ofman. It is one of the first examples showing that divide and conquer can break an assumed lower bound.

IN THE COURSE **Chapter 1 · Big-Number Multiplication** **Chapter 1 · Karatsuba Analysis**

Karatsuba multiplication divide and conquer

LD Ailsa Land and Alison Doig

1927–2021 · b. 1929 · British · London School of Economics

Branch and bound · 1960

In their 1960 Econometrica paper, written at the London School of Economics, they proposed the branch-and-bound method for solving integer linear programs: the solution space is split into branches, and a bound is computed for each branch with an LP relaxation. Branches whose bound is worse than the best known solution are pruned. This idea is still at the core of today's integer programming solvers; Doig later worked at the University of Melbourne under the name Alison Harcourt.

IN THE COURSE **Chapter 5 · The Branch and Bound Idea**

Chapter 5 · Integer Programming

branch and bound LP relaxation

HK Michael Held and Richard M. Karp

Karp: b. 1935 · *Journal of the SIAM*, 1962

Dynamic programming for the traveling salesman · 1962

In their 1962 paper "A Dynamic Programming Approach to Sequencing Problems" they solved the traveling salesman problem with dynamic programming. A state consists of a set and a vertex: dp[S][i] is the shortest path that leaves the start, visits all cities in the set S, and ends at i. The algorithm runs in $O(n^2 \cdot 2^n)$ time and $O(n \cdot 2^n)$ memory; it is far faster than trying all n! tours one by one, but still exponential. Richard Bellman published the same method independently in the same year.

IN THE COURSE **Chapter 3 · Other Classic DP Problems**

Chapter 6 · TSP: Solution Approaches

Held–Karp traveling salesman subset DP

LM John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel

Little: 1928–2024 · American · MIT

Branch and bound for the traveling salesman · 1963

In their paper "An Algorithm for the Traveling Salesman Problem," published in Operations Research in 1963, they gave a branch-and-bound algorithm for the traveling salesman problem; the name "branch and bound" appears for the first time in this paper. The lower bound comes from a reduced cost matrix: the smallest entry is subtracted from every row and every column, and the sum of the subtracted values is a lower bound on the cost of every tour. The method does not require symmetry; it also works for asymmetric problems where d[i][j] ≠ d[j][i]. Little is also known for Little's law (1961) in queueing theory.

IN THE COURSE **Chapter 5 · Reduced Cost Matrix** **Chapter 5 · TSP: A 6-City Example**

branch and bound reduced cost matrix traveling salesman

JE Jack Edmonds

b. 1934 · American-born · Canada, Waterloo

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 a 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 2 · When Does It Work?**

Edmonds-Karp polynomial time matroid

SG Solomon W. Golomb

1932–2016 · American · University of Southern California

Backtrack Programming · 1965

“Backtrack Programming,” his 1965 Journal of the ACM paper with Leonard Baumert, is among the first works to describe backtracking systematically as a general search method; it shows with examples how pruning shrinks the search tree. His definition of polyominoes in 1953 inspired computer searches for puzzles and tiling problems. Golomb coding is also his contribution.

IN THE COURSE **Chapter 4 · State-Space Tree** **Chapter 4 · Brute Force vs. Pruning**

backtracking pruning

RG Ronald Graham

1935–2020 · American · Bell Labs, UC San Diego

List scheduling (1966) · Graham scan (1972)

In 1966 he proved that for list scheduling, which gives each job to the first machine to become free, the makespan is at most $(2 - 1/m)$ times the optimum; this is one of the first examples of an approximation ratio. In 1969 he showed that the LPT rule, which sorts jobs from longest to shortest, has ratio $4/3 - 1/(3m)$. In 1972 he published the Graham scan, which finds the convex hull of points in the plane in $O(n \log n)$ time.

IN THE COURSE **Chapter 6 · List Scheduling** **Chapter 6 · The LPT Rule**

Chapter 1 · Convex Hull

list scheduling approximation ratio convex hull

LB

László Bélády

1928–2021 · Hungarian-born · USA, IBM

OPT (MIN) · Belady's anomaly

In 1966 at IBM he defined the offline algorithm that evicts the page that will be needed furthest in the future and causes the fewest page faults (MIN, today OPT); online policies are compared with this ideal. With Nelson and Shedler he showed that under FIFO the number of page faults can increase when the number of frames increases (Belady's anomaly). He had emigrated to the United States in 1961.

IN THE COURSE **Chapter 8 · OPT Step by Step** **Chapter 8 · Belady's Anomaly**

paging eviction policy

MF

Michael J. Flynn

1934–2025 · American · Stanford

Flynn's taxonomy · 1966

In the paper “Very High-Speed Computing Systems,” published in the Proceedings of the IEEE in 1966, he divided computers into four classes by the number of instruction and data streams they process at the same time: SISD (single instruction, single data), SIMD (single instruction, multiple data), MISD (multiple instruction, single data), and MIMD (multiple instruction, multiple data). The classic single-core processor is SISD, vector units are SIMD, and multicore processors and clusters are MIMD. The taxonomy became the common language for describing parallel machines.

IN THE COURSE **Chapter 9 · Flynn's Taxonomy**

SIMD MIMD parallel architecture

AV

Andrew Viterbi

b. 1935 · Italian-born · USA, UCLA, Qualcomm

Viterbi algorithm · 1967

In 1967 he proposed a dynamic programming algorithm for decoding convolutional codes that keeps, at each step, the best path reaching each state. This method, which finds the most likely sequence of states in Markov models, is used in mobile and satellite communication, speech recognition, and bioinformatics. In 1985 he co-founded Qualcomm with Irwin Jacobs.

IN THE COURSE **Chapter 3 · Other Classic DP Problems** **Chapter 3 · DP Design Steps**

dynamic programming most likely path

VS

Volker Strassen

b. 1936 · German · Zürich, Konstanz

Strassen's algorithm · 1969

In his 1969 paper “Gaussian Elimination is not Optimal” he split matrices into four blocks and multiplied them with seven block multiplications instead of eight, obtaining matrix multiplication in O(n^2.81) time; this was the first result showing that matrices can be multiplied faster than n^3. In 1971, with Schönhage, he developed fast multiplication of very large integers, and in 1977, with Solovay, a probabilistic primality test.

IN THE COURSE **Chapter 1 · Strassen's 7 Products** **Chapter 1 · Strassen Analysis**

Strassen's algorithm matrix multiplication

SC

Stephen Cook

b. 1939 · American-Canadian · University of Toronto

Cook's theorem · 1971

In his 1971 paper “The Complexity of Theorem-Proving Procedures” he showed that every problem in NP reduces to SAT in polynomial time, that is, that SAT is NP-complete. This result stated the P = NP question formally and began the theory of NP-completeness. He received the Turing Award in 1982.

IN THE COURSE **Chapter 4 · P, NP and NP-Hard**

NP-complete SAT P = NP?

RK

Richard M. Karp

b. 1935 · American · Berkeley

21 NP-complete problems · 1972

In 1972, starting from SAT, he showed by polynomial-time reductions that 21 problems, including 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 4 · P, NP and NP-Hard** **Chapter 6 · 3-SAT → Vertex Cover**

NP-complete reduction Rabin-Karp

LL

Leonid Levin

b. 1948 · Soviet-American · Boston University

Cook–Levin theorem · 1973

In the USSR, independently of Cook, he developed the notion of “universal problems” for search problems and published it in 1973; this is why the theorem is called the Cook–Levin theorem. For these problems he also gave a nearly optimal “universal search” algorithm. He later emigrated to the United States and has worked at Boston University since 1980.

IN THE COURSE **Chapter 4 · P, NP and NP-Hard** **Chapter 6 · 3-SAT → Vertex Cover**

NP-complete reduction

RB

Richard P. Brent

b. 1946 · Australian · Australian National University

Brent's theorem · 1974

In his 1974 paper “The Parallel Evaluation of General Arithmetic Expressions,” published in the Journal of the ACM, he bounded how long a computation written for many processors takes on fewer processors. In the language of work and depth, the result is $T_p \le W/p + D$, where W is the total work, D the depth, and p the number of processors. Read together with the lower bounds $T_p \ge W/p$ and $T_p \ge D$, it shows that the time on p processors is at most twice the best possible. Brent is also known for Brent's method in numerical root finding (1973).

IN THE COURSE **Chapter 9 · Work and Depth**

Brent's theorem work depth

NC

Nicos Christofides

1942–2019 · Cypriot-British · Imperial College London

Christofides' algorithm · 1976

In 1976 he gave the algorithm that, for the TSP satisfying the triangle inequality (metric TSP), finds a tour at most 3/2 times as long as the best tour: build an MST, add a minimum perfect matching on the odd-degree vertices, and traverse the Euler tour with shortcuts. This ratio was

the best known guarantee for metric TSP until 2020. Anatoliy Serdyukov reached the same result in the USSR at almost the same time.

IN THE COURSE **Chapter 6 · Metric TSP** **Chapter 6 · MST-Based 2-Approximation**

traveling salesman problem approximation ratio

JK Joseph B. (Jay) Kadane

b. 1941 · American · Carnegie Mellon

Maximum subarray · 1977

Michael Shamos described the maximum subarray problem, which Ulf Grenander had posed in 1977 for image analysis, at a seminar at Carnegie Mellon. Kadane, a statistician attending the seminar, found the O(n) algorithm that solves it in a single scan: V(i) = max(a[i], V(i−1) + a[i]) is the sum of the best subarray ending at i, and the answer is the largest of these values. The story and the algorithm became widely known through Jon Bentley's 1984 Programming Pearls column. Kadane is known for his work in Bayesian statistics.

IN THE COURSE **Chapter 3 · Other Classic DP Problems**

Kadane's algorithm maximum subarray O(n)

GJ Michael Garey and David S. Johnson

b. 1945 · 1945–2016 · American · Bell Labs

Computers and Intractability · 1979

Their book Computers and Intractability: A Guide to the Theory of NP-Completeness (1979), written at Bell Labs, became the basic reference that lists hundreds of NP-complete problems and teaches reduction techniques. In 1974 Johnson carried out one of the first systematic analyses of approximation algorithms for problems such as bin packing and set cover. The view that proving a problem NP-complete justifies turning to approximation algorithms or heuristic search became established through this work.

IN THE COURSE **Chapter 6 · Why Approximation Algorithms?**

Chapter 4 · P, NP and NP-Hard

NP-completeness approximation algorithm

LV

Leslie Valiant

b. 1949 · Hungarian-born · United Kingdom; USA, Harvard

#P · PAC · BSP · 2010 Turing Award

In 1979 he defined the class #P for counting problems and showed that computing the permanent is #P-complete. He proposed the PAC learning model, one of the theoretical foundations of machine learning, in 1984, and the bulk synchronous parallel (BSP) bridging model for parallel computation in 1990. He worked with Vazirani on randomized reductions and received the Turing Award in 2010.

IN THE COURSE **Chapter 9 · The PRAM Model** **Chapter 7 · Randomized Algorithms**

#P parallel model randomized reduction

DS

Daniel Sleator

b. 1953 · American · Carnegie Mellon

Competitive analysis · 1985

In the 1985 paper “Amortized Efficiency of List Update and Paging Rules,” written with Tarjan, he proposed comparing an online algorithm with the best offline algorithm, which knows the future; they showed that the move-to-front (MTF) rule is 2-competitive and LRU is k-competitive. The name “competitive analysis” was coined in the 1988 paper by Karlin, Manasse, Rudolph, and Sleator. With Tarjan he also developed splay trees and link-cut trees.

IN THE COURSE **Chapter 8 · Competitive Ratio** **Chapter 8 · MTF Is 2-Competitive**

Chapter 8 · Upper Bound: LRU Is k-Competitive

competitive analysis move-to-front paging

JV

Jeffrey S. Vitter

b. 1955 · American · Brown University

Reservoir sampling · 1985

In his 1985 paper “Random Sampling with a Reservoir” he analyzed methods that keep a uniform random sample of k items from a data stream of unknown length in a single pass; he named the basic method Algorithm R and developed the faster Algorithm Z. He earned his doctorate under Knuth; he is also among the pioneers of external-memory algorithms and data compression.

IN THE COURSE **Chapter 8 · Reservoir Sampling** **Chapter 8 · The Data Stream Model**

reservoir sampling data stream

HS

W. Daniel Hillis and Guy L. Steele Jr.

b. 1956 · b. 1954 · American · Thinking Machines

Parallel prefix sum · 1986

In their 1986 paper “Data Parallel Algorithms,” published in Communications of the ACM, they described data-parallel algorithms for the Connection Machine, which has thousands of processors. In the prefix sum, for d = 1, 2, 4, …, every position i ≥ d performs x[i] ← x[i − d] + x[i] at the same time, and the computation finishes in ⌈log n⌉ rounds. Because the total work is O(n log n), the method is not work-efficient; Blelloch's method does O(n) work at O(log n) depth. Hillis designed the Connection Machine; Steele is one of the designers of the Scheme language and a co-author of the Java language specification.

IN THE COURSE **Chapter 9 · Parallel Prefix Sums** **Chapter 9 · Prefix Sums Step by Step**

prefix sum data parallelism work efficiency

JG

John L. Gustafson

b. 1955 · American · *Communications of the ACM* 31(5), 1988

Gustafson's law · 1988

In the 1988 article “Reevaluating Amdahl's Law,” written with Edwin Barsis, he questioned the assumption behind Amdahl's law. Amdahl keeps the problem size fixed, so the sequential part limits the speedup to 1/(1 − f). In practice, as the number of processors grows, larger problems are usually solved; if the problem grows with p, the scaled speedup is S = (1 − f) + f · p, which grows almost linearly with the number of processors. The same year he received the first Gordon Bell Prize.

IN THE COURSE **Chapter 9 · Amdahl's Law**

Gustafson's law scaled speedup Amdahl's law

CL

Charles E. Leiserson

b. 1953 · American · MIT

Cilk · work stealing · CLRS

Introduction to Algorithms, which he wrote with Cormen, Rivest, and Stein (first edition 1990), is one of the most widely used algorithms textbooks in the world and one of the main references of this course. The Cilk language he developed at MIT uses a work-stealing scheduler with proven efficiency, in which an idle processor takes work from another processor's queue; Java's fork/join framework is based on the same idea. He also has a large share in the spread of work/span analysis.

IN THE COURSE **Chapter 9 · Work and Depth** **Chapter 9 · Fork/Join Step by Step**

work stealing work / span

GB Guy Blelloch

present · American · Carnegie Mellon

Prefix sum (scan) · NESL

In his 1990 work “Prefix Sums and Their Applications” he treated the prefix sum (scan) as a basic building block of parallel algorithms; the up-sweep/down-sweep method that runs with O(n) work and O(log n) depth is named after him. He developed NESL, a parallel programming language based on the work–depth cost model. At Carnegie Mellon he works on parallel graph processing libraries.

IN THE COURSE **Chapter 9 · Parallel Prefix Sums** **Chapter 9 · Prefix Sums Step by Step**

prefix sum work / span

DG David Karger

b. 1967 · American · Stanford, MIT

Randomized minimum cut · 1993

As a doctoral student at Stanford he published, in 1993, a minimum cut algorithm that contracts randomly chosen edges until two vertices remain. A single run finds a minimum cut with probability at least $2/(n(n-1))$; with enough repetitions the error probability becomes as small as desired. The Karger–Stein version, developed with Stein, made the method much faster.

IN THE COURSE **Chapter 7 · Karger's Minimum Cut**

Chapter 7 · Reducing the Error Probability

minimum cut Monte Carlo algorithm

VV

Vijay V. Vazirani

present · Indian-born · USA, Georgia Tech, UC Irvine

Approximation Algorithms · 2001

His book Approximation Algorithms (2001) is the standard textbook that presents approximation algorithms through shared techniques such as LP relaxation and the primal–dual scheme. With Micali he gave one of the fastest known algorithms for maximum matching in general graphs (1980). With Valiant he showed in 1986 that unique-solution SAT is also hard under randomized reductions (the Valiant–Vazirani theorem).

IN THE COURSE **Chapter 6 · Approximation Ratio** **Chapter 6 · Approximation Techniques**

approximation algorithm LP relaxation

Years and contributions are based on the original publications and encyclopedic sources. “Present” marks living people whose birth year is not given in published sources. Because Fermat's birth year is disputed in the sources, only his year of death is given; for people whose life dates could not be found in reliable sources, only the year of the contribution and the journal in which it appeared are given.