Chapter 1: Introduction

Algorithms

What is an algorithm? | Kinds of algorithms | Time complexity | Growth table

Primality test | Matrix multiplication | Permutations and Fibonacci | Euclid's algorithm

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/algorithms/slides/Chapter_01_Introduction.html

Learning Outcomes

Algorithms

Define an algorithm and its properties (input, output, definiteness, finiteness, effectiveness); write an everyday task as an algorithm.

Analysis

Express running time as a function T(n) of the input size n; read from a growth table how time changes as n grows.

Complexity examples

Count the operations in short code to find its complexity class (1, n, $\sqrt{n}$, n^3, n!, 2^n); see how memoization cuts exponential time.

Euclid's algorithm

Trace a gcd computation step by step; justify its correctness and its O(log min(a, b)) running time.

What Is an Algorithm?

- A **finite sequence of step-by-step instructions** for solving a problem or carrying out a task.
- It takes input, performs operations and produces **output**.
- Every step must be **precise and unambiguous**, and the algorithm must **terminate after finitely many steps**.

Input
Zero or more values supplied from outside.

Output
At least one result, related to the input.

Definiteness
Each step has exactly one meaning; nothing is left to interpretation.

Finiteness
Halts after a finite number of steps on every input.

Effectiveness
Each step is basic enough to actually be carried out.

Correctness
Produces the correct output for every valid input.

An Everyday Example: Making Tea

Input: water, tea leaves, sugar (optional), milk (optional)
Output: tea

1. Boil the water.
2. Pour the boiling water into the teapot.
3. Add the tea leaves to the pot.
4. Let the tea brew.
5. Strain the tea into a cup.
6. Add sugar and milk if desired.

Order matters

Steps 2 and 3 can be swapped, but steps 1 and 5 cannot: an algorithm is an **ordered** list of steps.

Is it precise?

"Let it brew" for how long? For a computer the step must read: **wait 10 minutes**.

Decisions

Step 6 contains a **condition**: "add sugar if wanted". Algorithms have decision points too.

Computing an Average: From Algorithm to Code

Algorithm

Input: x1, x2, x3 · Output: average

1. sum = x1 + x2 + x3
2. average = sum / 3
3. Print average.

Python

```python
def average(x1, x2, x3):
    avg = (x1 + x2 + x3) / 3
    return avg

x1 = float(input("Enter the first number: "))
x2 = float(input("Enter the second number: "))
x3 = float(input("Enter the third number: "))
avg = average(x1, x2, x3)
print("Average of the numbers:", avg)
```

Java

```java
public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  System.out.println("Enter the first number: ");
  double x1 = in.nextDouble();
  System.out.println("Enter the second number: ");
  double x2 = in.nextDouble();
  System.out.println("Enter the third number: ");
  double x3 = in.nextDouble();
  double avg = (x1 + x2 + x3) / 3;
  System.out.println("Average of the numbers: "
                     + avg);
}
```

The same algorithm can be written in any language: an algorithm is **language-independent**, and a program is one implementation of it.

Main Kinds of Algorithms

Sorting
Put items in order: insertion sort, merge sort, quicksort.

Searching
Find an item: linear and binary search, hashing.

Graphs
Shortest paths (Dijkstra), minimum spanning trees, maximum flow (Ford–Fulkerson).

Strings and compression
Text search, tries; shrinking files: Huffman, gzip.

Design techniques
Divide and conquer (merge sort, binary search, fast exponentiation), dynamic programming, greedy, backtracking, branch and bound.

Randomized and approximate
Randomized algorithms may take different paths on the same input; approximate solutions to hard problems.

A **deterministic** algorithm always takes the same steps and gives the same output on the same input (e.g. finding the maximum). A **randomized** algorithm flips coins, as when quicksort picks a random pivot.

Time Complexity

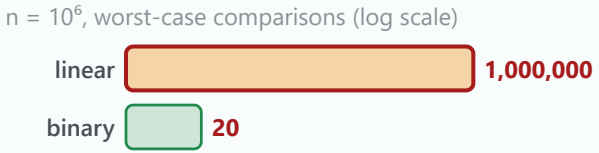
- **Algorithm analysis** measures the use of **resources** such as processor time and memory, so we can choose between algorithms that solve the same problem.
- Running time is expressed in terms of the **input size n**: **T(n)**. We count **basic operations**, not seconds: seconds depend on the machine and language, operation counts do not.

Example: summing an n-element array, counted line by line

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

Why does it matter?

Searching a sorted array of 10^6 items: linear search makes up to **1,000,000** comparisons, binary search at most **20**.



T(n) = 1 + (n + 1) + n + 1 = **2n + 3**. The constants depend on the machine; what matters is that T(n) **grows linearly with n** → Θ(n).

Growth Table: As n Grows

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

Last column: time for $n = 10^6$ on a computer doing 10^9 operations per second. A faster computer shrinks the constant; **a better algorithm changes the growth rate**. Replacing n^2 with n log n turns minutes into milliseconds at 10^6 items.

Constant and Linear Time: O(1), O(n)

Printing the first element T(n) = O(1)

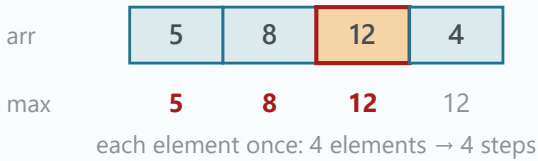
```
void printFirst(int[] arr) {
  if (arr.length > 0)
    System.out.println("First element: "
                       + arr[0]);
  else
    System.out.println("Array is empty");
}
```

Whether the array has 10 elements or 10 million, it does **one** comparison and one access: the time does not depend on n.

Finding the maximum T(n) = O(n)

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

Each element is examined **once**: n comparisons, O(1) extra memory. Example: [5, 8, 12, 4] → 12.



Primality Test: O(√n)

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

Why stop at √n?

If n = a · b, one of a and b is **at most √n**. If nothing up to √n divides n, n is prime.

How many divisions?

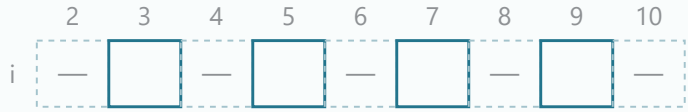
Even numbers are ruled out up front and only odd i are tried: at most ≈ √n / 2 divisions → **T(n) = O(√n)**.

n = 97: i = 3, 5, 7, 9 → **4** divisions (95 if we tried 2 to 96).
n = 1,000,000,007: **15,810** divisions.

Note: here the input size is the **value** of n. If n has b bits, $\sqrt{n} = 2^{b/2}$: the method is exponential in the number of bits.

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
▶    if (n <= 1)
3      return false;
▶    else if (n == 2)
5      return true;
▶    else if (n % 2 == 0)
7      return false;
8    int limit = (int) Math.sqrt(n) + 1;
9    for (int i = 3; i < limit; i += 2)
10     if (n % i == 0)
11       return false;
12   return true;
13 }
```



even i (gray) are never tried

n = 97: greater than 1, not equal to 2, and odd. None of the first three checks returns; multiples of 2 are ruled out up front.

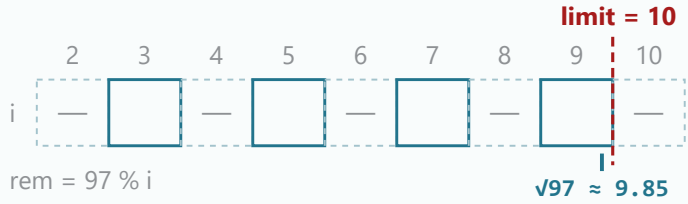
Step 1 / 8

VARIABLES

`n = 97` `limit = –` `i = –`

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
2    if (n <= 1)
3      return false;
4    else if (n == 2)
5      return true;
6    else if (n % 2 == 0)
7      return false;
▶    int limit = (int) Math.sqrt(n) + 1;
9    for (int i = 3; i < limit; i += 2)
10     if (n % i == 0)
11       return false;
12   return true;
13 }
```



even i (gray) are never tried

limit = (int) √97 + 1 = 9 + 1 = **10**. Only odd numbers below 10 are tried: 3, 5, 7, 9.

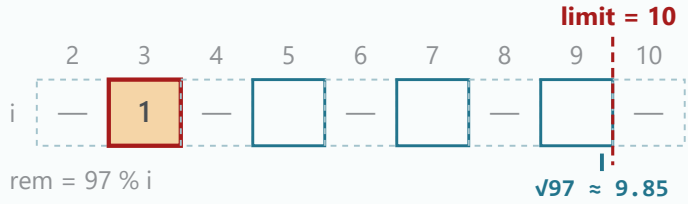
Step 2 / 8

VARIABLES

`n = 97` `limit = 10` `i = –`

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
2    if (n <= 1)
3      return false;
4    else if (n == 2)
5      return true;
6    else if (n % 2 == 0)
7      return false;
8    int limit = (int) Math.sqrt(n) + 1;
▶    for (int i = 3; i < limit; i += 2)
▶      if (n % i == 0)
11       return false;
12   return true;
13 }
```



even i (gray) are never tried

i = 3: 97 % 3 = **1** ≠ 0 → 3 does not divide 97. `i += 2`.

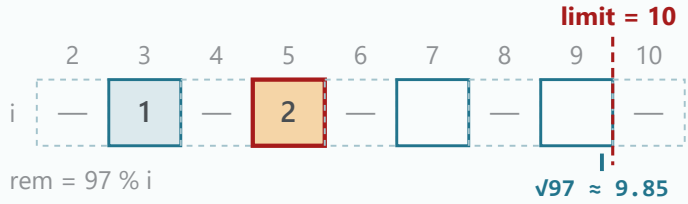
Step 3 / 8

VARIABLES

`n = 97` `limit = 10` `i = 3` `n % i = 1`

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
2    if (n <= 1)
3      return false;
4    else if (n == 2)
5      return true;
6    else if (n % 2 == 0)
7      return false;
8    int limit = (int) Math.sqrt(n) + 1;
▶    for (int i = 3; i < limit; i += 2)
▶      if (n % i == 0)
11       return false;
12   return true;
13 }
```



even i (gray) are never tried

i = 5: 97 % 5 = **2** ≠ 0 → 5 does not divide 97. `i += 2`.

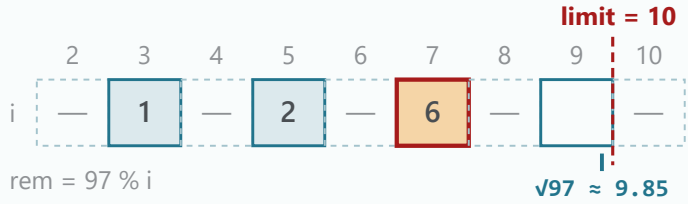
Step 4 / 8

VARIABLES

`n = 97` `limit = 10` `i = 5` `n % i = 2`

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
2    if (n <= 1)
3      return false;
4    else if (n == 2)
5      return true;
6    else if (n % 2 == 0)
7      return false;
8    int limit = (int) Math.sqrt(n) + 1;
▶    for (int i = 3; i < limit; i += 2)
▶      if (n % i == 0)
11       return false;
12   return true;
13 }
```



even i (gray) are never tried

i = 7: 97 % 7 = **6** ≠ 0 → 7 does not divide 97. `i += 2`.

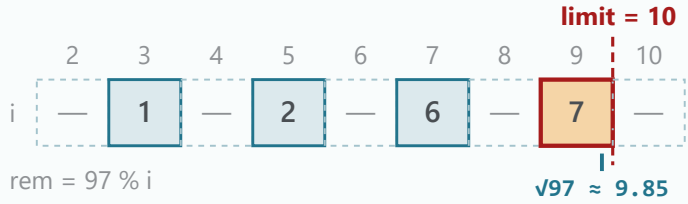
Step 5 / 8

VARIABLES

`n = 97` `limit = 10` `i = 7` `n % i = 6`

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
2    if (n <= 1)
3      return false;
4    else if (n == 2)
5      return true;
6    else if (n % 2 == 0)
7      return false;
8    int limit = (int) Math.sqrt(n) + 1;
▶    for (int i = 3; i < limit; i += 2)
▶      if (n % i == 0)
11       return false;
12   return true;
13 }
```



even i (gray) are never tried

i = 9: 97 % 9 = **7** ≠ 0 → 9 does not divide 97. `i += 2`. Next candidate: i = 11.

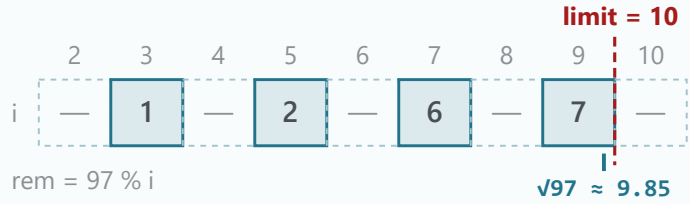
Step 6 / 8

VARIABLES

`n = 97` `limit = 10` `i = 9` `n % i = 7`

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
2    if (n <= 1)
3      return false;
4    else if (n == 2)
5      return true;
6    else if (n % 2 == 0)
7      return false;
8    int limit = (int) Math.sqrt(n) + 1;
▶    for (int i = 3; i < limit; i += 2)
10     if (n % i == 0)
11       return false;
12   return true;
13 }
```



even i (gray) are never tried

i = 11 < 10 fails and the loop ends. No odd number up to √97 divides 97.

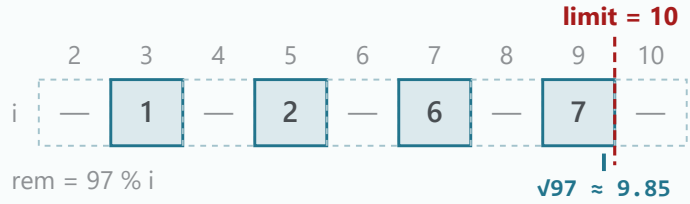
Step 7 / 8

VARIABLES

`n = 97` `limit = 10` `i = 11` `n % i = –`

isPrime(97): Step by Step

```
1  boolean isPrime(int n) {
2    if (n <= 1)
3      return false;
4    else if (n == 2)
5      return true;
6    else if (n % 2 == 0)
7      return false;
8    int limit = (int) Math.sqrt(n) + 1;
9    for (int i = 3; i < limit; i += 2)
10     if (n % i == 0)
11       return false;
▶    return true;
13 }
```



even i (gray) are never tried

`return true` : 97 is prime. Only **4 divisions** were needed; trying every number from 2 to 96 would take 95.

Step 8 / 8

VARIABLES

`n = 97` `limit = 10` `i = 11` `result = true`

Cubic Time: Matrix Multiplication, $\Theta(n^3)$

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

Input: $A = [a_{ij}]$, $B = [b_{ij}]$ ($n \times n$). **Output:** $C = [c_{ij}] = A \cdot B$, $i, j = 1, 2, \ldots, n$.

Definition

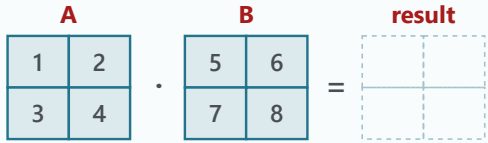
$$c_{ij} = \sum_{k=1..n} a_{ik} \cdot b_{kj}$$

1	**2**		**5**	**6**	=	**19**	**22**
3	**4**	·	**7**	**8**		**43**	**50**

$c_{11} = 1 \cdot 5 + 2 \cdot 7 = 19$. Each c_{ij} takes n multiplications and there are n^2 entries → **n^3 multiplications**: $T(n) = \Theta(n^3)$.

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
▶    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
▶    int n = A.length, m = B[0].length, p = B.length;
▶    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
8        int sum = 0;
9        for (int k = 0; k < p; k++)
10         sum += A[i][k] * B[k][j];
11       result[i][j] = sum;
12     }
13   return result;
14 }
```



A and B are 2 × 2 with compatible sizes: n = m = p = 2. `result` starts as a 2 × 2 zero matrix.

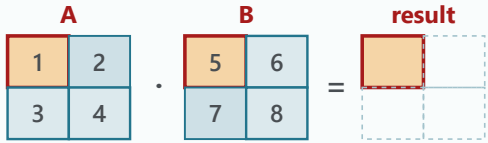
Step 1 / 10

VARIABLES

n = 2 | m = 2 | p = 2 | i = – | j = – | k = – | sum = – | mults = 0

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
▶        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
11       result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 1·5 = 5

i = 0, j = 0: `sum = 0`. k = 0: add A[0][0] · B[0][0] = 1 · 5 = 5.

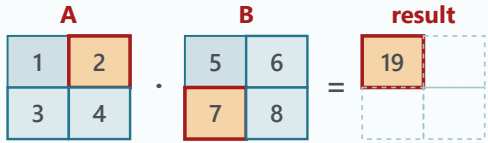
Step 2 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 0` `j = 0` `k = 0` `sum = 5` `mults = 1`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
8        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
▶        result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 5 + 2·7 = 19

k = 1: A[0][1] · B[1][0] = 2 · 7 → sum = 5 + 14 = **19**. Inner loop ends: `result[0][0] = 19`.

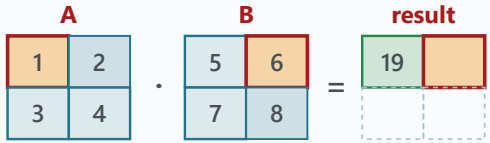
Step 3 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 0` `j = 0` `k = 1` `sum = 19` `mults = 2`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
▶        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
11       result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 1·6 = 6

i = 0, j = 1: `sum = 0`. k = 0: add A[0][0] · B[0][1] = 1 · 6 = 6.

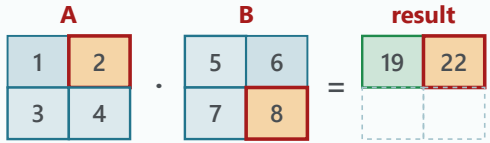
Step 4 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 0` `j = 1` `k = 0` `sum = 6` `mults = 3`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
8        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
▶        result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 6 + 2·8 = 22

k = 1: A[0][1] · B[1][1] = 2 · 8 → sum = 6 + 16 = **22**. Inner loop ends: `result[0][1] = 22`.

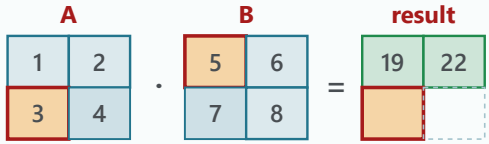
Step 5 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 0` `j = 1` `k = 1` `sum = 22` `mults = 4`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
▶        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
11       result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 3·5 = 15

i = 1, j = 0: `sum = 0`. k = 0: add A[1][0] · B[0][0] = 3 · 5 = 15.

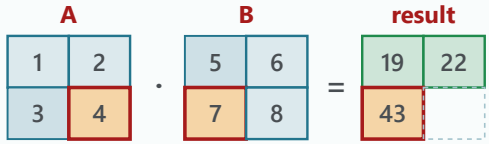
Step 6 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 1` `j = 0` `k = 0` `sum = 15` `mults = 5`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
8        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
▶        result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 15 + 4·7 = 43

k = 1: A[1][1] · B[1][0] = 4 · 7 → sum = 15 + 28 = **43**. Inner loop ends: `result[1][0] = 43`.

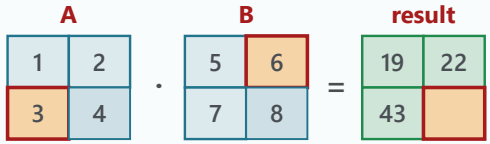
Step 7 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 1` `j = 0` `k = 1` `sum = 43` `mults = 6`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
▶        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
11       result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 3·6 = 18

i = 1, j = 1: `sum = 0`. k = 0: add A[1][0] · B[0][1] = 3 · 6 = 18.

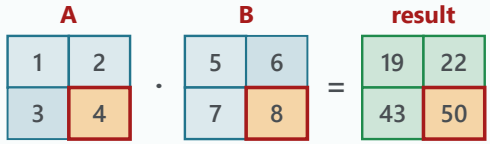
Step 8 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 1` `j = 1` `k = 0` `sum = 18` `mults = 7`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
8        int sum = 0;
▶        for (int k = 0; k < p; k++)
▶          sum += A[i][k] * B[k][j];
▶        result[i][j] = sum;
12     }
13   return result;
14 }
```



sum = 18 + 4·8 = 50

k = 1: A[1][1] · B[1][1] = 4 · 8 → sum = 18 + 32 = **50**. Inner loop ends: `result[1][1] = 50`.

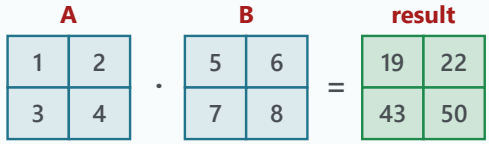
Step 9 / 10

VARIABLES

`n = 2` `m = 2` `p = 2` `i = 1` `j = 1` `k = 1` `sum = 50` `mults = 8`

multiply: 2 × 2 Matrices Step by Step

```
1  int[][] multiply(int[][] A, int[][] B) {
2    if (B.length == 0 || (A.length > 0 && A[0].length !=
   B.length))
3      throw new IllegalArgumentException("size");
4    int n = A.length, m = B[0].length, p = B.length;
5    int[][] result = new int[n][m];
6    for (int i = 0; i < n; i++)
7      for (int j = 0; j < m; j++) {
8        int sum = 0;
9        for (int k = 0; k < p; k++)
10         sum += A[i][k] * B[k][j];
11       result[i][j] = sum;
12     }
▶    return result;
14 }
```



Returns result = [[19, 22], [43, 50]]. **8 = 2^3 multiplications**: n multiplications for each of the n^2 entries → n^3.

Step 10 / 10

VARIABLES

n = **2** m = **2** p = **2** i = **2** j = **–** k = **–** sum = **–**

mults = **8**

Factorial Time: Generating All Permutations

```java
void permute(int[] arr, int k) {
  if (k == arr.length) {           // one permutation done
    System.out.println(Arrays.toString(arr));
    return;
  }
  for (int i = k; i < arr.length; i++) {
    swap(arr, k, i);               // put element i at k
    permute(arr, k + 1);           // arrange the rest
    swap(arr, k, i);               // undo the swap
  }
}
```

[1, 2, 3] → 123, 132, 213, 231, 321, 312. There are n choices for the first position, n−1 for the second, …: **n · (n−1) · … · 1 = n!** permutations.

n	n!
3	6
5	120
10	3,628,800
15	$1.3 \cdot 10^{12}$
20	$2.4 \cdot 10^{18}$

Printing each permutation takes n steps: **Θ(n · n!)** in total, the "O(n!) class" for short. Even n = 20 is out of reach in practice.

Exponential Time: Fibonacci Numbers

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

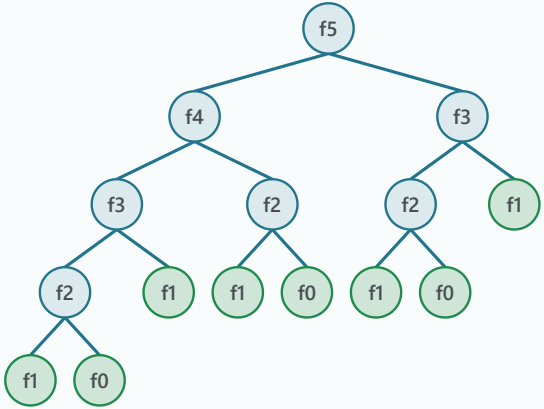
static final int N = 92;            // largest n for long
long[] memo = new long[N + 1];   // 0: not set
long fibMemo(int n) {       // memoization
  if (n <= 1) return n;
  if (memo[n] == 0)
    memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
  return memo[n];
}
```

F(0) = 0, F(1) = 1, F(n) = F(n−1) + F(n−2).

fib: each call spawns two calls → **O(2^n)** (exactly Θ(1.618^n)); fib(30) makes 2,692,537 calls.

fibMemo: each F(k) is computed once, 2n − 1 calls → **O(n)**; 59 calls for n = 30.

F(92) is the largest value that fits in a `long`; beyond that use `BigInteger`.



Naive recursion: **15 calls**. f3 is computed twice, f2 three times, f1 five times; the tree roughly doubles at each level → exponential growth.

Exponential Time: Fibonacci Numbers

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

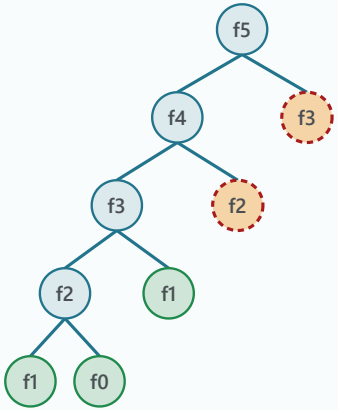
static final int N = 92;            // largest n for long
long[] memo = new long[N + 1];   // 0: not set
long fibMemo(int n) {       // memoization
  if (n <= 1) return n;
  if (memo[n] == 0)
    memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
  return memo[n];
}
```

F(0) = 0, F(1) = 1, F(n) = F(n−1) + F(n−2).

fib: each call spawns two calls → **O(2^n)** (exactly Θ(1.618^n)); fib(30) makes 2,692,537 calls.

fibMemo: each F(k) is computed once, 2n − 1 calls → **O(n)**; 59 calls for n = 30.

F(92) is the largest value that fits in a `long`; beyond that use `BigInteger`.



Memoized: **9 calls**. The dashed orange nodes (f2, f3) come straight from the memo array; their subtrees are never expanded → O(n).

Euclid's Algorithm (gcd)

- Finds the **greatest common divisor** (gcd).
- One of the oldest known algorithms (Euclid, c. 300 BC) and the simplest way to compute a gcd.
- Key idea: **gcd(a, b) = gcd(b, a mod b)**, gcd(a, 0) = a.

1. **Step 1:** divide the larger number by the smaller and take the remainder.
2. **Step 2:** repeat with the pair (divisor, remainder) until the remainder is zero.
3. **Step 3:** when the remainder is zero, the last **divisor** is the gcd.

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

Why is it correct?

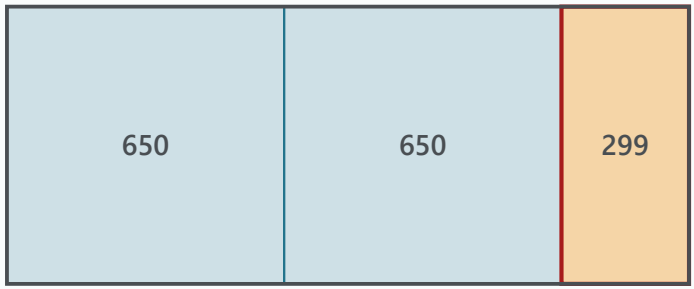
If a = q·b + r, every common divisor of a and b also divides r = a − q·b, and every common divisor of b and r divides a. The common divisors are the same.

gcd(1599, 650): Step by Step

```
1 int gcd(int a, int b) {
▶   if (b == 0)
3     return a;
▶   return gcd(b, a % b);
5 }
```

CALL STACK

`gcd(1599, 650)`



1599 × 650 rectangle

1599 = 2 · 650 + **299**. Cut 2 squares from the 1599×650 rectangle; the leftover strip is 650×299 → `gcd(650, 299)`.

Step 1 / 7

VARIABLES

`a = 1599` `b = 650` `a%b = 299`

gcd(1599, 650): Step by Step

```
1 int gcd(int a, int b) {
▶   if (b == 0)
3     return a;
▶   return gcd(b, a % b);
5 }
```

CALL STACK

- `gcd(650, 299)`
- `gcd(1599, 650)`



650 × 299 rectangle

650 = 2 · 299 + **52**. Cut 2 squares from the 650×299 rectangle; the leftover strip is 299×52 → `gcd(299, 52)`.

Step 2 / 7

VARIABLES

`a = 650` `b = 299` `a%b = 52`

gcd(1599, 650): Step by Step

```
1 int gcd(int a, int b) {
▶   if (b == 0)
3     return a;
▶   return gcd(b, a % b);
5 }
```

CALL STACK

- `gcd(299, 52)`
- `gcd(650, 299)`
- `gcd(1599, 650)`



299 × 52 rectangle

299 = 5 · 52 + **39**. Cut 5 squares from the 299×52 rectangle; the leftover strip is 52×39 → `gcd(52, 39)`.

Step 3 / 7

VARIABLES

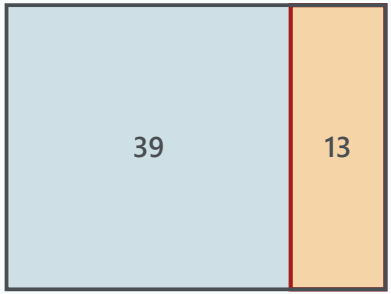
`a = 299` `b = 52` `a%b = 39`

gcd(1599, 650): Step by Step

```
1 int gcd(int a, int b) {
▶   if (b == 0)
3     return a;
▶   return gcd(b, a % b);
5 }
```

CALL STACK

- **gcd(52, 39)**
- gcd(299, 52)
- gcd(650, 299)
- gcd(1599, 650)



52 × 39 rectangle

52 = 1 · 39 + **13**. Cut 1 square from the 52×39 rectangle; the leftover strip is 39×13 → `gcd(39, 13)`.

Step 4 / 7

VARIABLES

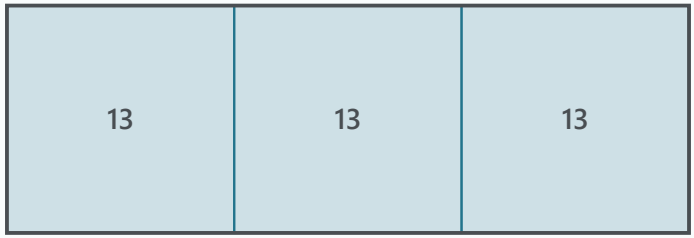
`a = 52` `b = 39` `a%b = 13`

gcd(1599, 650): Step by Step

```
1 int gcd(int a, int b) {
▶   if (b == 0)
3     return a;
▶   return gcd(b, a % b);
5 }
```

CALL STACK

- `gcd(39, 13)`
- `gcd(52, 39)`
- `gcd(299, 52)`
- `gcd(650, 299)`
- `gcd(1599, 650)`



39 × 13 rectangle

39 = 3 · 13 + **0**. Cut 3 squares from the 39×13 rectangle; the leftover strip is 13×0 → `gcd(13, 0)`.

Step 5 / 7

VARIABLES

`a = 39` `b = 13` `a%b = 0`

gcd(1599, 650): Step by Step

```
1 int gcd(int a, int b) {
▶   if (b == 0)
▶     return a;
4   return gcd(b, a % b);
5 }
```

CALL STACK

- **gcd(13, 0)**
- gcd(39, 13)
- gcd(52, 39)
- gcd(299, 52)
- gcd(650, 299)
- gcd(1599, 650)

gcd(13, 0) = **13**

b == 0 → **return 13**. 13 is the largest number that divides both 1599 and 650.

Step 6 / 7

VARIABLES

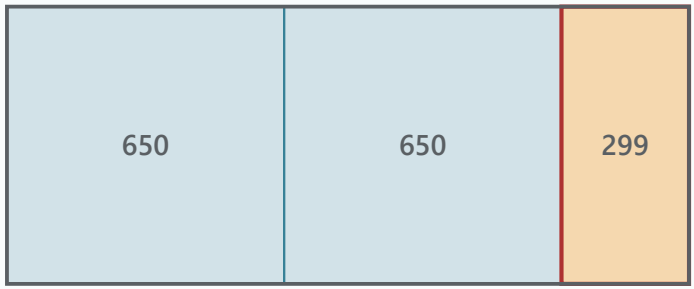
a = **13** b = **0**

gcd(1599, 650): Step by Step

```
1 int gcd(int a, int b) {
2   if (b == 0)
3     return a;
  ▶ return gcd(b, a % b);
5 }
```

CALL STACK

`gcd(1599, 650)`



1599 × 650 rectangle

The value 13 is passed back unchanged through every call: **gcd(1599, 650) = 13**. Five divisions suffice, and 13 × 13 squares tile the 1599 × 650 rectangle exactly.

Step 7 / 7

VARIABLES

`a = 1599` `b = 650` `result = 13`

Euclid's Algorithm: Why Is It Fast?

- It is efficient: **every two steps** the larger number drops to **less than half** (a mod b < a/2).
- So the number of steps is at most $\approx 2 \log_2 \min(a, b)$: **O(log min(a, b))**.
- The **worst case** is consecutive Fibonacci numbers: every quotient is q = 1, so the numbers shrink as slowly as possible (Lamé's theorem).
- For comparison: trying every divisor from 1 to min(a, b) takes O(min(a, b)).

Worst case: gcd(55, 34)

```
55 = 1 · 34 + 21
34 = 1 · 21 + 13
21 = 1 · 13 + 8
13 = 1 · 8 + 5
8 = 1 · 5 + 3
5 = 1 · 3 + 2
3 = 1 · 2 + 1
2 = 2 · 1 + 0
```

8 divisions, gcd = 1. gcd(1599, 650) needs only 5.

Summary

Topic	Key points
Algorithms	input → precise, finite, effective steps → output; language-independent
Analysis	T(n): number of basic operations as a function of input size; time and memory
Growth	$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$; a better algorithm changes the growth rate
Examples	first element $O(1)$, maximum $O(n)$, primality $O(\sqrt{n})$, matrix multiplication $\Theta(n^3)$, permutations $O(n!)$, naive Fibonacci $O(2^n)$
Memoization	storing repeated subproblems takes Fibonacci from $O(2^n)$ down to $O(n)$
Euclid	gcd(a, b) = gcd(b, a mod b); $O(\log \min(a, b))$; worst case: consecutive Fibonacci numbers

SUMMARY

Common Mistakes

Measuring time in seconds

Seconds depend on the machine; analysis uses operation counts as a function of n and their growth rate.

Forgetting memoization

Naive recursive Fibonacci is $O(2^n)$; storing repeated subproblems brings it down to O(n).

Getting the √n bound wrong

With `i < (int) Math.sqrt(n)`, √n itself is never tried: squares such as 9, 25, 49 come out prime.

Confusing the input size

In primality testing n is the value of the number; for a b-bit number $\sqrt{n} = 2^{b/2}$, i.e. exponential in the bit length.

Thinking matrix multiplication is n^2

The result has n^2 entries, but each needs n multiplications: n^3 in total.

Not undoing the swap

Forget the second `swap` and the array gets scrambled: for [1, 2, 3] some permutations repeat and others never appear.

SUMMARY

Sample Problems

1. For `isPrime(221)`, which values of i does the loop try, how many divisions are made, and what is the result?
2. Give the Θ class of $T(n) = 5n^2 + 100n \log_2 n + 7$. Which term dominates at $n = 1000$?
3. How many multiplications does `multiply` perform on two 3×3 matrices? For $n = 100$?
4. How many lines does `permute([1, 2, 3, 4], 0)` print? What are the first two?
5. Find gcd(252, 105) with Euclid's algorithm.

▶ **Show answers**

SUMMARY

Exercises

1. Write an algorithm that counts how many times a given number occurs in an array; give its time and space complexity.
2. Nested loops: how many times does `for i in 0..n-1: for j in i..n-1: op` run? What is its class?
3. Modify `findMax` to return both the minimum and the maximum of the array. How many comparisons does it make?
4. Write Euclid's algorithm with a loop (no recursion).
5. At 10^9 operations per second, find the largest n solvable in 1 second for $f(n) = n^2$ and $f(n) = 2^n$.
6. How many times does the loop `while (n > 1) n = n / 2;` run?

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapters 1–3).
- Sedgewick and Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 1.4: Analysis of Algorithms).
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed., 1997 (properties of algorithms; Euclid).
- J. Kleinberg and É. Tardos, *Algorithm Design*, Pearson, 2005 (Chapter 2: Basics of Algorithm Analysis).

The End

Chapter 1: Introduction

Next: Chapter 2: Algorithm Complexity