Chapter 1.3: Algorithms

Object Oriented Programming with Java

Problem-solving steps · Pseudocode · Implementation · Sample algorithms · Dry runs · Searching · Sorting

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_01_03_Algorithms.html

Learning Outcomes

Problem solving

Define a problem by its input, output and test cases; write the solution first as an algorithm, then as code.

Pseudocode and tracing

Express an algorithm in pseudocode and trace it by hand, step by step, with a variable table.

Translating to Java

Turn input, output, decision and loop steps into their Java equivalents.

Basic algorithms

Write and test algorithms for printing in order, factorial, Fibonacci, palindromes, primes and powers.

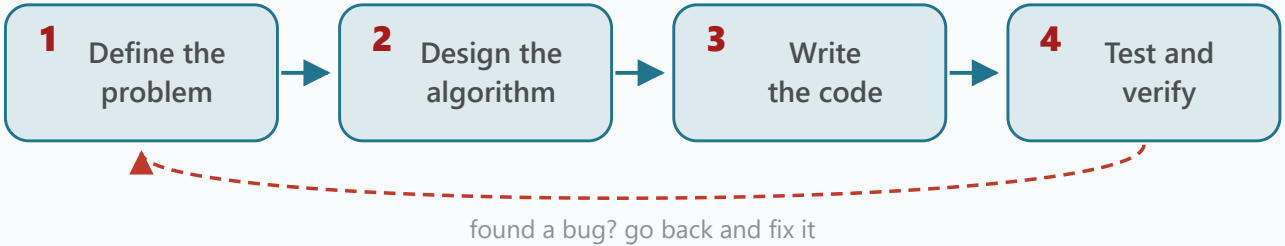
Searching

Compare random, linear and binary search; explain why binary search needs sorted data.

Sorting

Apply shuffling, selection sort and bubble sort to an array step by step.

Algorithm First, Then Code



- **Step 1:** A clear problem definition is the key to success: what is given, what is wanted?
- **Step 2:** The algorithm gives step-by-step directions for solving the problem. Choosing the right one makes the program efficient.
- **Step 3:** With the algorithm settled, we code it in the chosen language (Java, for us).
- **Step 4:** Test cases verify the program and expose bugs; go back to earlier steps when needed.

PROBLEM SOLVING

What Is an Algorithm?

Algorithm: a finite, ordered sequence of clear steps that solves a problem.

- The word comes from the Latin spelling of the name of the 9th-century mathematician **al-Khwarizmi**.
- A computer does **not understand** algorithms; it runs one once we translate it into a programming language.
- Several algorithms may solve the same problem; we compare them by speed and memory.

Analogy: a recipe is an algorithm. The ingredients are the *input*, the steps the *algorithm*, the dish the *output*. A vague step like "add a little salt" is not good enough for a computer.

Input
Zero or more initial values.

Output
At least one result.

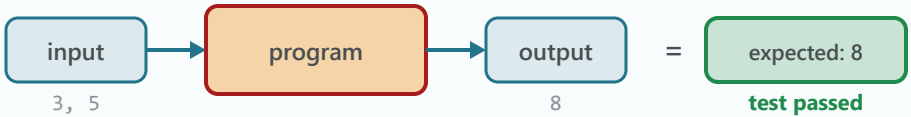
Definiteness
Every step is clear and unambiguous.

Finiteness
It ends after finitely many steps.

Effectiveness
Every step can actually be carried out.

Defining the Problem

- Describe the program's **input** and **output** in plain language.
- Write **test cases** to check the program works: for each input, write down the *expected output* in advance.



Problem: add two numbers

Test	Input	Expected output
1	3, 5	8
2	−2, 7	5
3 (edge)	0, 0	0

Problem: count the words in a text

Test	Input	Expected output
1	"Hello, world!"	2
2	"Just a test."	3
3 (edge)	"" (empty text)	0

The third rows are edge cases: extreme inputs like zero, empty text or negative numbers are where bugs most often hide.

Designing and Testing the Algorithm

Lay out, in order, the steps that solve the problem logically.

Problem: add two numbers.

Algorithm:

1. Get the first number.
2. Get the second number.
3. Add the two numbers.
4. Print the sum.

Once the algorithm exists, the next step is **testing**:

- Pick small but meaningful test cases.
- Run the algorithm on paper step by step, recording each result in a **variable table** (a *dry run*).

Dry run of test 1 (input 3, 5): it prints 8, as expected

Step	num1	num2	sum	Screen
1. Get the first number	3			
2. Get the second number	3	5		
3. Add the two numbers	3	5	8	
4. Print the sum	3	5	8	8

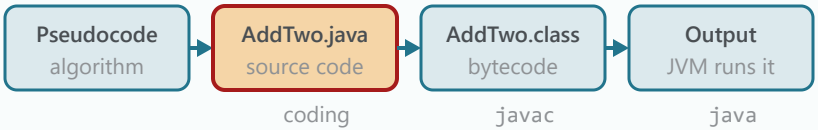
Expressing an Algorithm: Pseudocode

An algorithm can be written in plain language, in **pseudocode**, as a flowchart or directly in a programming language. Pseudocode looks like code but has no compiler rules; it is written for people, and it helps catch logic errors before coding and makes teamwork easier.

Pseudocode	Meaning	Java
`Start` … `End`	Start and end of the algorithm	the `{ … }` body of `main`
`Read x`	Get a value from the user	`x = scanner.nextInt();`
`Print "…", x`	Print to the screen	`System.out.println("…" + x);`
`x = 5`	Assign a value to a variable	`x = 5;`
`If x = y then`	Test for equality	`if (x == y)` (two equals signs!)
`Else`	If the condition is false	`else`
`For i = 1 to n`	Repeat	`for (int i = 1; i <= n; i++)`
`Break`	Leave the loop at once	`break;`
`%`, `and`, `or`	Remainder, logical and/or	`%`, `&&`, `||`

Implementation

- After the algorithm comes **coding**.
- A programming language expresses the algorithm in a form the computer can compile and run.
- To keep a program manageable and readable, split the code into **logical parts**: input, computation, output; in larger programs, **methods**.
- Each pseudocode line usually maps to one or a few lines of Java.

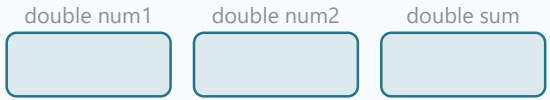


Algorithm step	Java
Get the first number	`num1 = scanner.nextDouble();`
Get the second number	`num2 = scanner.nextDouble();`
Add the two numbers	`sum = num1 + num2;`
Print the sum	`System.out.println(sum);`

The point is to see how each algorithm step maps to a line of code. Choose short, descriptive names for your own variables.

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter the first number: ");
3 double num1 = scanner.nextDouble();
4 System.out.print("Enter the second number: ");
5 double num2 = scanner.nextDouble();
6 double sum = num1 + num2;
7 System.out.println(num1 + " + " + num2
8         + " = " + sum);
9 scanner.close();
```



A `Scanner` is created to read the keyboard; no numbers yet.

Step 1 / 8

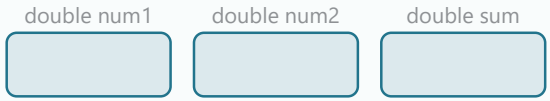
VARIABLES

`scanner = Scanner`

OUTPUT

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
▶ System.out.print("Enter the first number: ");
3 double num1 = scanner.nextDouble();
4 System.out.print("Enter the second number: ");
5 double num2 = scanner.nextDouble();
6 double sum = num1 + num2;
7 System.out.println(num1 + " + " + num2
8         + " = " + sum);
9 scanner.close();
```



`print` writes the prompt without a newline; the cursor waits right after it.

Step 2 / 8

VARIABLES

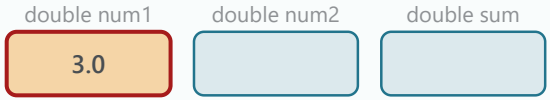
`scanner = Scanner`

OUTPUT

```
Enter the first number:
```

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter the first number: ");
▶ double num1 = scanner.nextDouble();
4 System.out.print("Enter the second number: ");
5 double num2 = scanner.nextDouble();
6 double sum = num1 + num2;
7 System.out.println(num1 + " + " + num2
8         + " = " + sum);
9 scanner.close();
```



The user types **3** and presses Enter; `nextDouble()` reads it as 3.0 into `num1`.

Step 3 / 8

VARIABLES

`scanner = Scanner` `num1 = 3.0`

OUTPUT

```
Enter the first number: 3
```

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter the first number: ");
3 double num1 = scanner.nextDouble();
▶ System.out.print("Enter the second number: ");
5 double num2 = scanner.nextDouble();
6 double sum = num1 + num2;
7 System.out.println(num1 + " + " + num2
8         + " = " + sum);
9 scanner.close();
```

double num1	double num2	double sum
3.0		

The second prompt is printed.

Step 4 / 8

VARIABLES

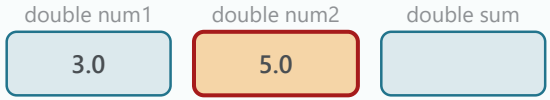
`scanner = Scanner` `num1 = 3.0`

OUTPUT

```
Enter the first number: 3
Enter the second number:
```

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter the first number: ");
3 double num1 = scanner.nextDouble();
4 System.out.print("Enter the second number: ");
▶ double num2 = scanner.nextDouble();
6 double sum = num1 + num2;
7 System.out.println(num1 + " + " + num2
8         + " = " + sum);
9 scanner.close();
```



The user enters **5**; `num2` = 5.0.

Step 5 / 8

VARIABLES

`scanner` = **Scanner** | `num1` = **3.0** | `num2` = **5.0**

OUTPUT

```
Enter the first number: 3
Enter the second number: 5
```

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter the first number: ");
3 double num1 = scanner.nextDouble();
4 System.out.print("Enter the second number: ");
5 double num2 = scanner.nextDouble();
▶ double sum = num1 + num2;
7 System.out.println(num1 + " + " + num2
8         + " = " + sum);
9 scanner.close();
```

double num1	double num2	double sum
3.0	5.0	8.0

The right side is evaluated first: 3.0 + 5.0 = 8.0; the result goes into `sum`.

Step 6 / 8

VARIABLES

`scanner = Scanner` `num1 = 3.0` `num2 = 5.0` `sum = 8.0`

OUTPUT

```
Enter the first number: 3
Enter the second number: 5
```

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter the first number: ");
3 double num1 = scanner.nextDouble();
4 System.out.print("Enter the second number: ");
5 double num2 = scanner.nextDouble();
6 double sum = num1 + num2;
▶ System.out.println(num1 + " + " + num2
▶         + " = " + sum);
9 scanner.close();
```

double num1	double num2	double sum
3.0	5.0	8.0

`+` joins strings; the numbers are converted to text to form one line.

Step 7 / 8

VARIABLES

`scanner` = **Scanner** · `num1` = **3.0** · `num2` = **5.0** · `sum` = **8.0**

OUTPUT

```
Enter the first number: 3
Enter the second number: 5
3.0 + 5.0 = 8.0
```

Tracing the Addition Program

```
1 Scanner scanner = new Scanner(System.in);
2 System.out.print("Enter the first number: ");
3 double num1 = scanner.nextDouble();
4 System.out.print("Enter the second number: ");
5 double num2 = scanner.nextDouble();
6 double sum = num1 + num2;
7 System.out.println(num1 + " + " + num2
8         + " = " + sum);
▶ scanner.close();
```

double num1	double num2	double sum
3.0	5.0	8.0

The scanner is closed and the program ends. The output matches the expected 8.

Step 8 / 8

VARIABLES

`scanner = closed` `num1 = 3.0` `num2 = 5.0` `sum = 8.0`

OUTPUT

```
Enter the first number: 3
Enter the second number: 5
3.0 + 5.0 = 8.0
```

SAMPLE ALGORITHMS

Sample Algorithms

For each algorithm: problem definition → pseudocode → dry run with a variable table → Java code and tests.

Printing two integers in order

Decision: `If … Else`

Printing three integers in order

Nested decisions, `and`

Factorial

A loop and an accumulator

Fibonacci sequence

Shifting variables in a loop

Palindrome check

Clean, reverse, compare

Prime check

A flag and breaking out of a loop

Powers

Repeated multiplication

Searching and sorting

In the second half: random, linear and binary search; shuffle, selection sort, bubble sort.

Printing Two Integers in Ascending Order

```
Start
    Print "Enter the first integer: "
    Read firstNum
    Print "Enter the second integer: "
    Read secondNum
    If firstNum < secondNum then
        Print firstNum, " ", secondNum
    Else
        Print secondNum, " ", firstNum
    End If
End
```

Problem definition

Input: two integers. Output: the same numbers, smaller first.

Input	Expected
4, 11	4 11
13, 11	11 13
7, 7	7 7

If the numbers are equal the condition is false and the `Else` branch runs; since both are the same, the output is still right.

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```



The user is asked for the first number.

Step 1 / 13

VARIABLES

`firstNum = –` `secondNum = –`

OUTPUT

```
Enter the first integer:
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```



The user enters **4**; it goes into the `firstNum` box.

Step 2 / 13

VARIABLES

`firstNum = 4` `secondNum = –`

OUTPUT

```
Enter the first integer: 4
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
▶      Print "Enter the second integer: "
▶      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```

firstNum	secondNum
4	11

The second number is requested; the user enters **11**.

Step 3 / 13

VARIABLES

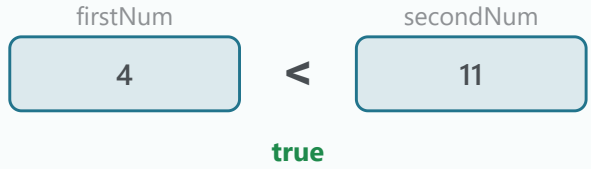
`firstNum = 4` `secondNum = 11`

OUTPUT

```
Enter the first integer: 4
Enter the second integer: 11
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
▶      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```



Condition: 4 < 11 → **true**.

Step 4 / 13

VARIABLES

`firstNum = 4` `secondNum = 11`

OUTPUT

```
Enter the first integer: 4
Enter the second integer: 11
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
▶          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```

firstNum		secondNum
4	<	11

true

The condition is true, so the `then` branch runs: first, then second.

Step 5 / 13

VARIABLES

`firstNum = 4` `secondNum = 11`

OUTPUT

```
Enter the first integer: 4
Enter the second integer: 11
4 11
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
▶  End
```

firstNum	secondNum
4	11

The algorithm ends. Output **4 11**, as expected.

Step 6 / 13

VARIABLES

`firstNum = 4` `secondNum = 11`

OUTPUT

```
Enter the first integer: 4
Enter the second integer: 11
4 11
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```



Second test: the algorithm runs again with 13 and 11.

Step 7 / 13

VARIABLES

`firstNum = –` `secondNum = –`

OUTPUT

```
Enter the first integer:
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
▶      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```



The user enters **13**; it goes into the `firstNum` box.

Step 8 / 13

VARIABLES

`firstNum = 13` `secondNum = –`

OUTPUT

```
Enter the first integer: 13
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
▶      Print "Enter the second integer: "
▶      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```

firstNum	secondNum
13	11

The second number is requested; the user enters **11**.

Step 9 / 13

VARIABLES

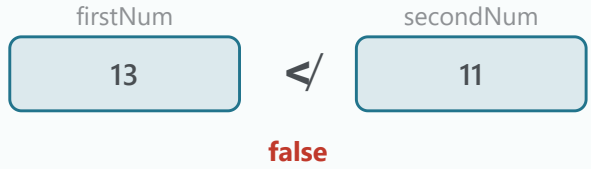
`firstNum = 13` `secondNum = 11`

OUTPUT

```
Enter the first integer: 13
Enter the second integer: 11
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
▶      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```



false

Condition: 13 < 11 → **false**.

Step 10 / 13

VARIABLES

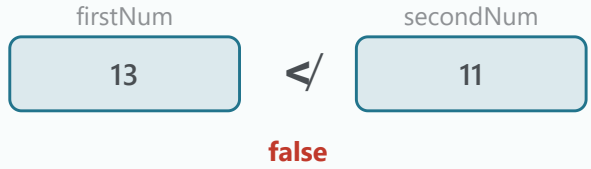
`firstNum = 13` `secondNum = 11`

OUTPUT

```
Enter the first integer: 13
Enter the second integer: 11
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
▶      Else
9          Print secondNum, " ", firstNum
10     End If
11 End
```



false

False: we move to the `Else` branch.

Step 11 / 13

VARIABLES

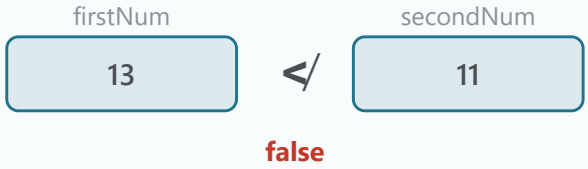
`firstNum = 13` `secondNum = 11`

OUTPUT

```
Enter the first integer: 13
Enter the second integer: 11
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
▶          Print secondNum, " ", firstNum
10     End If
11 End
```



The second (smaller) prints first, then the first.

Step 12 / 13

VARIABLES

`firstNum = 13` `secondNum = 11`

OUTPUT

```
Enter the first integer: 13
Enter the second integer: 11
11 13
```

Two Integers: Dry Run with (4, 11) and (13, 11)

```
1  Start
2      Print "Enter the first integer: "
3      Read firstNum
4      Print "Enter the second integer: "
5      Read secondNum
6      If firstNum < secondNum then
7          Print firstNum, " ", secondNum
8      Else
9          Print secondNum, " ", firstNum
10     End If
▶  End
```

firstNum	secondNum
13	11

The algorithm ends. Output **11 13**, as expected.

Step 13 / 13

VARIABLES

`firstNum = 13` `secondNum = 11`

OUTPUT

```
Enter the first integer: 13
Enter the second integer: 11
11 13
```

Printing Three Integers in Ascending Order

```
Start
    // Get the first integer
    Print "Enter the first integer: "
    Read firstNum
    // Get the second integer
    Print "Enter the second integer: "
    Read secondNum
    // Get the third integer
    Print "Enter the third integer: "
    Read thirdNum
    // Print in ascending order
    If firstNum ≤ secondNum and
         firstNum ≤ thirdNum then
        Print firstNum
        If secondNum ≤ thirdNum then
            Print secondNum, " ", thirdNum
        Else
            Print thirdNum, " ", secondNum
        End If
    Else If secondNum ≤ firstNum and
         secondNum ≤ thirdNum then
        Print secondNum
        If firstNum ≤ thirdNum then
            Print firstNum, " ", thirdNum
        Else
            Print thirdNum, " ", firstNum
        End If
    Else
        Print thirdNum
        If firstNum ≤ secondNum then
            Print firstNum, " ", secondNum
        Else
            Print secondNum, " ", firstNum
        End If
    End If
End
```

Idea: find and print **the smallest** first, then order the other two with the two-integer algorithm. The same small algorithm appears three times: a natural candidate for a method.

Three Integers: Java Code and Tests

```java
import java.util.Scanner;

public class SortThree {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter three integers: ");
        int first = scanner.nextInt();
        int second = scanner.nextInt();
        int third = scanner.nextInt();
        if (first <= second && first <= third) {
            System.out.print(first + " ");
            if (second <= third) System.out.println(second + " " + third);
            else                 System.out.println(third + " " + second);
        } else if (second <= first && second <= third) {
            System.out.print(second + " ");
            if (first <= third) System.out.println(first + " " + third);
            else                System.out.println(third + " " + first);
        } else {
            System.out.print(third + " ");
            if (first <= second) System.out.println(first + " " + second);
            else                System.out.println(second + " " + first);
        }
        scanner.close();
    }
}
```

Tests: all 3! = 6 orders (1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2, 3 2 1) → `1 2 3`; equal values 5 5 2 → `2 5 5`.

Three Integers: Tracing the Java Code (7, 3, 5)

```
1  if (first <= second && first <= third) {
2      System.out.print(first + " ");
3      if (second <= third)
4          System.out.println(second + " " + third);
5      else
6          System.out.println(third + " " + second);
7  } else if (second <= first && second <= third) {
8      System.out.print(second + " ");
9      if (first <= third)
10         System.out.println(first + " " + third);
11     else
12         System.out.println(third + " " + first);
13 } else {
14     System.out.print(third + " ");
15     if (first <= second)
16         System.out.println(first + " " + second);
17     else
18         System.out.println(second + " " + first);
19 }
```



7 ≤ 3 ? false

Input 7, 3, 5. First condition: 7 ≤ 3 is **false**; the right side of `&&` is not even evaluated.

Step 1 / 5

VARIABLES

`first` = **7** `second` = **3** `third` = **5**

OUTPUT

Three Integers: Tracing the Java Code (7, 3, 5)

```
1  if (first <= second && first <= third) {
2      System.out.print(first + " ");
3      if (second <= third)
4          System.out.println(second + " " + third);
5      else
6          System.out.println(third + " " + second);
▶  } else if (second <= first && second <= third) {
8      System.out.print(second + " ");
9      if (first <= third)
10         System.out.println(first + " " + third);
11     else
12         System.out.println(third + " " + first);
13 } else {
14     System.out.print(third + " ");
15     if (first <= second)
16         System.out.println(first + " " + second);
17     else
18         System.out.println(second + " " + first);
19 }
```

first	second	third
7	3	5

3 ≤ 7 and 3 ≤ 5 : true

Second condition: 3 ≤ 7 is **true** and 3 ≤ 5 is **true** → the smallest is `second`.

Step 2 / 5

VARIABLES

`first = 7` `second = 3` `third = 5`

OUTPUT

Three Integers: Tracing the Java Code (7, 3, 5)

```
1  if (first <= second && first <= third) {
2      System.out.print(first + " ");
3      if (second <= third)
4          System.out.println(second + " " + third);
5      else
6          System.out.println(third + " " + second);
7  } else if (second <= first && second <= third) {
▶      System.out.print(second + " ");
9      if (first <= third)
10         System.out.println(first + " " + third);
11     else
12         System.out.println(third + " " + first);
13 } else {
14     System.out.print(third + " ");
15     if (first <= second)
16         System.out.println(first + " " + second);
17     else
18         System.out.println(second + " " + first);
19 }
```



smallest: 3

The smallest is printed; `print` adds no newline.

Step 3 / 5

VARIABLES

`first = 7` `second = 3` `third = 5`

OUTPUT

```
3
```

Three Integers: Tracing the Java Code (7, 3, 5)

```
1  if (first <= second && first <= third) {
2      System.out.print(first + " ");
3      if (second <= third)
4          System.out.println(second + " " + third);
5      else
6          System.out.println(third + " " + second);
7  } else if (second <= first && second <= third) {
8      System.out.print(second + " ");
▶      if (first <= third)
10         System.out.println(first + " " + third);
11     else
12         System.out.println(third + " " + first);
13 } else {
14     System.out.print(third + " ");
15     if (first <= second)
16         System.out.println(first + " " + second);
17     else
18         System.out.println(second + " " + first);
19 }
```

first	second	third
7	3	5

7 ≤ 5 ? false

The other two are compared: 7 ≤ 5 is **false**.

Step 4 / 5

VARIABLES

`first = 7` `second = 3` `third = 5`

OUTPUT

```
3
```

Three Integers: Tracing the Java Code (7, 3, 5)

```
1  if (first <= second && first <= third) {
2      System.out.print(first + " ");
3      if (second <= third)
4          System.out.println(second + " " + third);
5      else
6          System.out.println(third + " " + second);
7  } else if (second <= first && second <= third) {
8      System.out.print(second + " ");
9      if (first <= third)
10         System.out.println(first + " " + third);
▶      else
▶          System.out.println(third + " " + first);
13 } else {
14     System.out.print(third + " ");
15     if (first <= second)
16         System.out.println(first + " " + second);
17     else
18         System.out.println(second + " " + first);
19 }
```



order: 3 5 7

The `else` branch prints 5, then 7. Output **3 5 7**.

Step 5 / 5

VARIABLES

`first = 7` `second = 3` `third = 5`

OUTPUT

```
3 5 7
```

SAMPLE ALGORITHMS

Computing a Factorial

```
Start
    // Enter a number
    Read num
    // Initialize the result
    factorial = 1
    // Loop to compute the factorial
    For i = 1 to num
        factorial = factorial * i
    End For
    Print "Factorial of the number: ", factorial
End
```

Definition: n! = 1 × 2 × 3 × … × n, and 0! = 1.

n	n!
0	1
1	1
3	6
5	120
10	3,628,800

`factorial` is an **accumulator**: it starts at **1**, the identity for multiplication. Starting at 0 would make the result always 0.

Factorial: Dry Run (num = 5)

```
1 Start
▶     Read num
3     factorial = 1
4     For i = 1 to num
5         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1

The user enters **5**.

Step 1 / 14

VARIABLES

num = **5** | factorial = **–** | i = **–**

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
▶     factorial = 1
4     For i = 1 to num
5         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1

The accumulator starts at 1.

Step 2 / 14

VARIABLES

`num = 5` `factorial = 1` `i = –`

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
5         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1

Loop: i = 1 (i ≤ 5, so the body runs).

Step 3 / 14

VARIABLES

num = 5 | factorial = 1 | i = 1

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
▶         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1

factorial = 1 × 1 = **1**.

Step 4 / 14

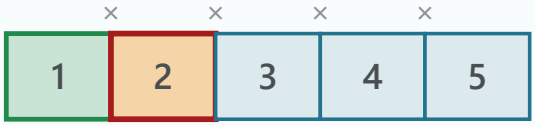
VARIABLES

`num = 5` `factorial = 1` `i = 1`

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
▶     For i = 1 to num
5         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1

Loop: i = 2 (i ≤ 5, so the body runs).

Step 5 / 14

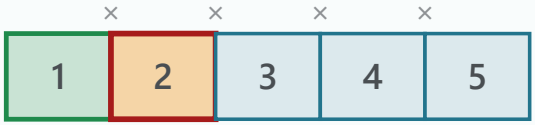
VARIABLES

`num = 5` `factorial = 1` `i = 2`

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
▶         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 = 2

factorial = 1 × 2 = **2**.

Step 6 / 14

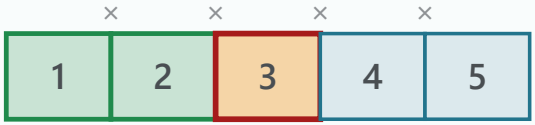
VARIABLES

`num = 5` `factorial = 2` `i = 2`

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
▶     For i = 1 to num
5         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 = 2

Loop: i = 3 (i ≤ 5, so the body runs).

Step 7 / 14

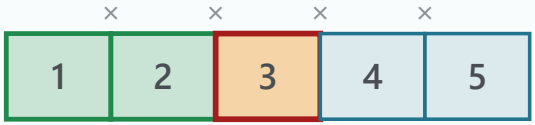
VARIABLES

num = 5 | factorial = 2 | i = 3

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
▶         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 × 3 = 6

factorial = 2 × 3 = **6**.

Step 8 / 14

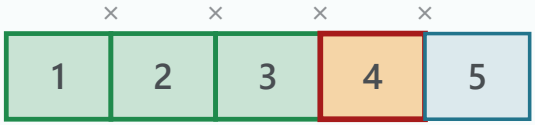
VARIABLES

`num = 5` `factorial = 6` `i = 3`

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
▶     For i = 1 to num
5         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 × 3 = 6

Loop: i = 4 (i ≤ 5, so the body runs).

Step 9 / 14

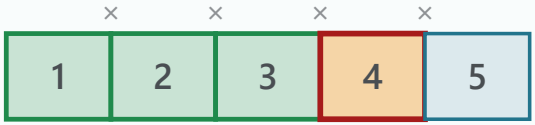
VARIABLES

num = 5 | factorial = 6 | i = 4

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
▶         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 × 3 × 4 = 24

factorial = 6 × 4 = **24**.

Step 10 / 14

VARIABLES

`num = 5` `factorial = 24` `i = 4`

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
▶     For i = 1 to num
5         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 × 3 × 4 = 24

Loop: i = 5 (i ≤ 5, so the body runs).

Step 11 / 14

VARIABLES

num = 5 | factorial = 24 | i = 5

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
▶         factorial = factorial * i
6     End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 × 3 × 4 × 5 = 120

factorial = 24 × 5 = **120**.

Step 12 / 14

VARIABLES

`num = 5` `factorial = 120` `i = 5`

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
5         factorial = factorial * i
      End For
7     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 × 3 × 4 × 5 = 120

i becomes 6; 6 ≤ 5 is false, so the loop ends.

Step 13 / 14

VARIABLES

num = 5 | factorial = 120 | i = 6

OUTPUT

Factorial: Dry Run (num = 5)

```
1 Start
2     Read num
3     factorial = 1
4     For i = 1 to num
5         factorial = factorial * i
6     End For
▶     Print "Factorial of the number: ", factorial
8 End
```



factorial = 1 × 2 × 3 × 4 × 5 = 120

The result is printed: 5! = 120.

Step 14 / 14

VARIABLES

`num = 5` `factorial = 120` `i = 6`

OUTPUT

```
Factorial of the number: 120
```

SAMPLE ALGORITHMS

The Fibonacci Sequence

```
Start
    // Enter the number of terms
    Read n
    // Initialize the first two terms
    a = 0
    b = 1
    If n ≥ 1 then Print a
    If n ≥ 2 then Print b
    // Compute the remaining terms
    For i = 2 to n - 1
        result = a + b
        a = b
        b = result
        Print result
    End For
End
```

Definition: the first two terms are 0 and 1; each later term is the sum of the previous two.

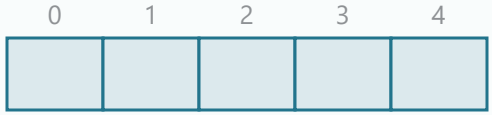
`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …`

n (number of terms)	Expected output
1	0
2	0 1
5	0 1 1 2 3
7	0 1 1 2 3 5 8

The loop runs **n − 2** times, for i = 2 … n − 1; the first two terms print before it. The lines `a = b` and `b = result` "shift" the values one step.

Fibonacci: Dry Run (n = 5)

```
1  Start
▶      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



The user enters **n = 5**: we want the first 5 terms.

Step 1 / 20

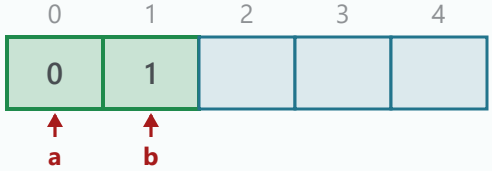
VARIABLES

`n = 5` `a = –` `b = –` `i = –` `result = –`

OUTPUT

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
▶      a = 0
▶      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



The first two terms: a = 0, b = 1.

Step 2 / 20

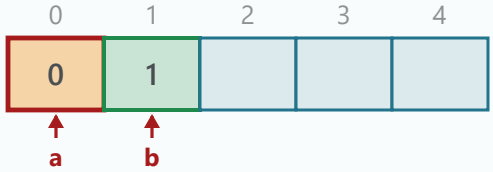
VARIABLES

`n = 5` `a = 0` `b = 1` `i = –` `result = –`

OUTPUT

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
▶      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



5 ≥ 1: print `a`.

Step 3 / 20

VARIABLES

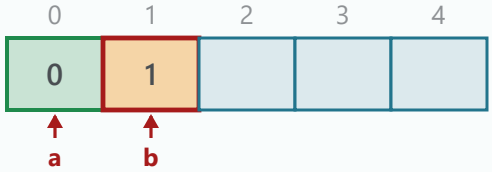
`n = 5` `a = 0` `b = 1` `i = –` `result = –`

OUTPUT

```
0
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
▶      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



5 ≥ 2: print `b`.

Step 4 / 20

VARIABLES

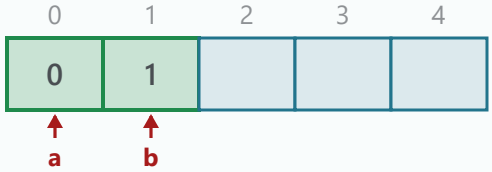
`n = 5` `a = 0` `b = 1` `i = –` `result = –`

OUTPUT

```
0 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
▶      For i = 2 to n - 1
8          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



Loop: i = 2 (i ≤ n − 1 = 4).

Step 5 / 20

VARIABLES

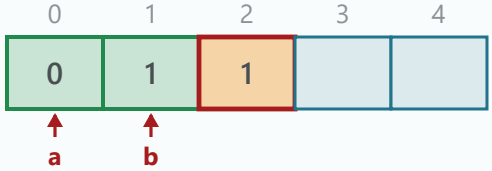
n = **5** | a = **0** | b = **1** | i = **2** | result = **–**

OUTPUT

```
0 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
▶          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



result = a + b = 0 + 1 = **1**.

Step 6 / 20

VARIABLES

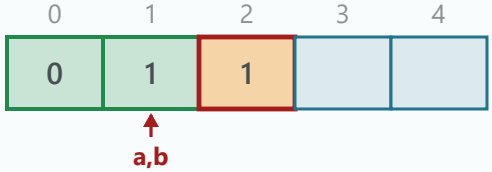
`n = 5` `a = 0` `b = 1` `i = 2` `result = 1`

OUTPUT

```
0 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
▶          a = b
10         b = result
11         Print result
12     End For
13 End
```



a = b → a = 1 (shifted one step).

Step 7 / 20

VARIABLES

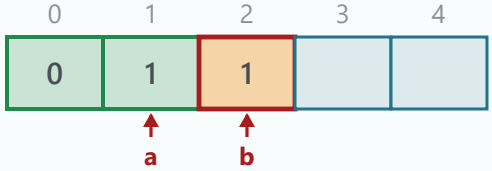
n = 5 | a = 1 | b = 1 | i = 2 | result = 1

OUTPUT

```
0 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
▶          b = result
11         Print result
12     End For
13 End
```



b = result → b = 1.

Step 8 / 20

VARIABLES

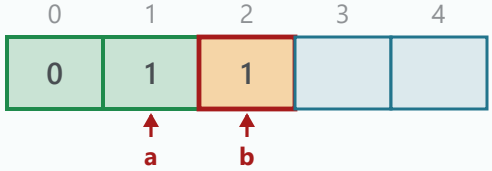
`n = 5` `a = 1` `b = 1` `i = 2` `result = 1`

OUTPUT

```
0 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
▶          Print result
12     End For
13 End
```



The new term is printed: 1.

Step 9 / 20

VARIABLES

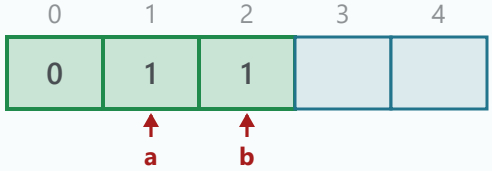
`n = 5` `a = 1` `b = 1` `i = 2` `result = 1`

OUTPUT

```
0 1 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
▶      For i = 2 to n - 1
8          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



Loop: i = 3 (i ≤ n − 1 = 4).

Step 10 / 20

VARIABLES

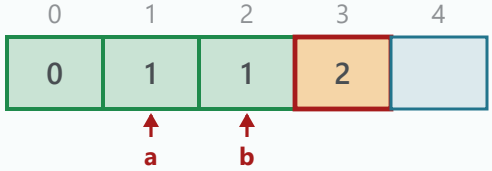
n = 5 | a = 1 | b = 1 | i = 3 | result = 1

OUTPUT

```
0 1 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
▶          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



result = a + b = 1 + 1 = **2**.

Step 11 / 20

VARIABLES

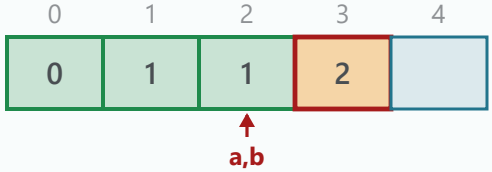
`n = 5` `a = 1` `b = 1` `i = 3` `result = 2`

OUTPUT

```
0 1 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
▶          a = b
10         b = result
11         Print result
12     End For
13 End
```



a = b → a = 1 (shifted one step).

Step 12 / 20

VARIABLES

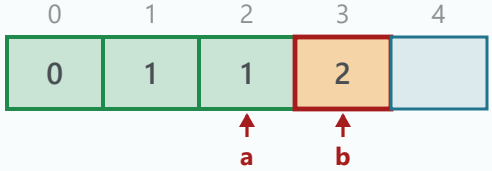
`n = 5` `a = 1` `b = 1` `i = 3` `result = 2`

OUTPUT

```
0 1 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
▶          b = result
11         Print result
12     End For
13 End
```



b = result → b = 2.

Step 13 / 20

VARIABLES

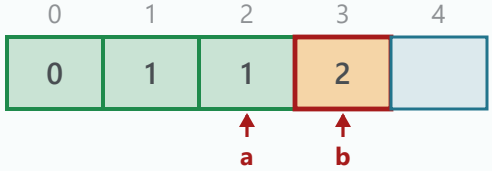
n = 5 | a = 1 | b = 2 | i = 3 | result = 2

OUTPUT

```
0 1 1
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
▶          Print result
12     End For
13 End
```



The new term is printed: 2.

Step 14 / 20

VARIABLES

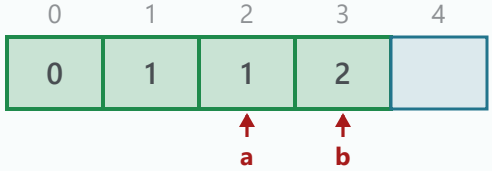
`n = 5` `a = 1` `b = 2` `i = 3` `result = 2`

OUTPUT

```
0 1 1 2
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
▶      For i = 2 to n - 1
8          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



Loop: i = 4 (i ≤ n − 1 = 4).

Step 15 / 20

VARIABLES

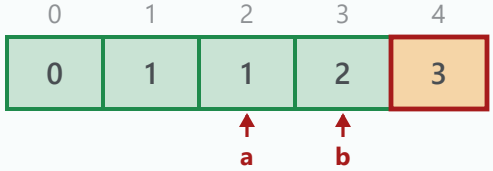
n = 5 | a = 1 | b = 2 | i = 4 | result = 2

OUTPUT

```
0 1 1 2
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
▶          result = a + b
9          a = b
10         b = result
11         Print result
12     End For
13 End
```



result = a + b = 1 + 2 = **3**.

Step 16 / 20

VARIABLES

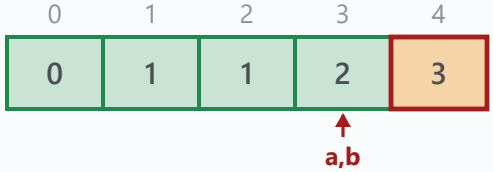
`n = 5` `a = 1` `b = 2` `i = 4` `result = 3`

OUTPUT

```
0 1 1 2
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
▶          a = b
10         b = result
11         Print result
12     End For
13 End
```



a = b → a = 2 (shifted one step).

Step 17 / 20

VARIABLES

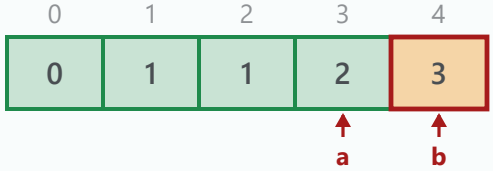
`n = 5` `a = 2` `b = 2` `i = 4` `result = 3`

OUTPUT

```
0 1 1 2
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
▶          b = result
11         Print result
12     End For
13 End
```



b = result → b = 3.

Step 18 / 20

VARIABLES

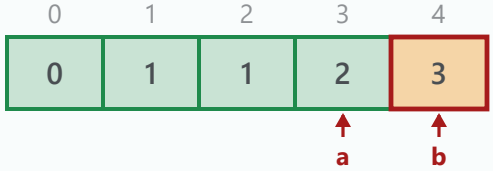
n = 5 | a = 2 | b = 3 | i = 4 | result = 3

OUTPUT

```
0 1 1 2
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
▶          Print result
12     End For
13 End
```



The new term is printed: 3.

Step 19 / 20

VARIABLES

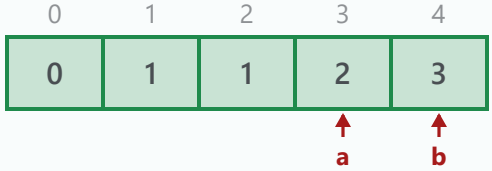
`n = 5` `a = 2` `b = 3` `i = 4` `result = 3`

OUTPUT

```
0 1 1 2 3
```

Fibonacci: Dry Run (n = 5)

```
1  Start
2      Read n
3      a = 0
4      b = 1
5      If n ≥ 1 then Print a
6      If n ≥ 2 then Print b
7      For i = 2 to n − 1
8          result = a + b
9          a = b
10         b = result
11         Print result
▶      End For
13 End
```



i becomes 5; 5 ≤ 4 is false, the loop ends. Exactly 5 terms printed: **0 1 1 2 3**.

Step 20 / 20

VARIABLES

`n = 5` `a = 2` `b = 3` `i = 5` `result = 3`

OUTPUT

```
0 1 1 2 3
```

SAMPLE ALGORITHMS

Checking for a Palindrome

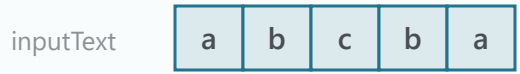
```
Start
    // Ask for a word or sentence
    Read inputText
    // Clean: lowercase, no spaces or punctuation
    cleanText = Clean(inputText)
    // Reverse the cleaned input
    reversed = Reverse(cleanText)
    // Compare cleaned and reversed input
    If cleanText = reversed then
        Print "The text is a palindrome."
    Else
        Print "The text is not a palindrome."
    End If
End
```

Palindrome: a word or sentence that reads the same backwards: *kayak, abcba, "Ey Edip Adana'da pide ye"*. Case, spaces and punctuation are ignored, so the text is **cleaned** before comparing.

Input	Expected
abcba	palindrome
Kayak	palindrome
Kitap	not

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
▶      Read inputText
3      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



The user enters **abcba**.

Step 1 / 10

VARIABLES

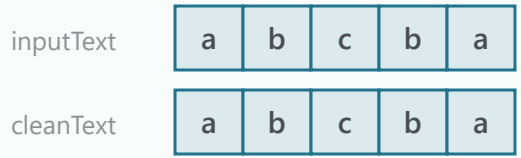
`inputText = "abcba"`

`cleanText = –` `reversed = –`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
▶      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



Cleaning: convert to lowercase (no change).

Step 2 / 10

VARIABLES

`inputText = "abcba"`

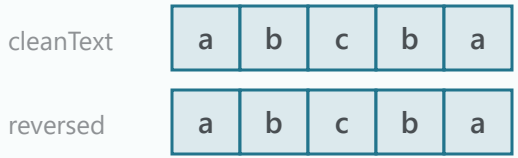
`cleanText = "abcba"`

`reversed = –`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
3      cleanText = Clean(inputText)
▶      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



The clean text read backwards: **abcba**.

Step 3 / 10

VARIABLES

`inputText = "abcba"`

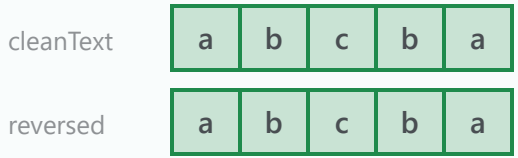
`cleanText = "abcba"`

`reversed = "abcba"`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
3      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
▶      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



Compare letter by letter: all letters match → **true**.

Step 4 / 10

VARIABLES

`inputText = "abcba"`

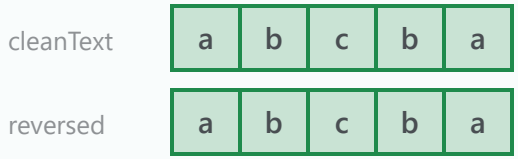
`cleanText = "abcba"`

`reversed = "abcba"`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
3      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



True: the palindrome message prints.

Step 5 / 10

VARIABLES

`inputText = "abcba"`

`cleanText = "abcba"`

`reversed = "abcba"`

OUTPUT

```
The text is a palindrome.
```

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
▶      Read inputText
3      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```

inputText

K	i	t	a	p

Second test: the user enters **Kitap**.

Step 6 / 10

VARIABLES

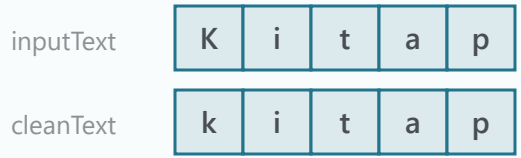
`inputText = "Kitap"`

`cleanText = –` `reversed = –`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
▶      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



Cleaning: convert to lowercase ("Kitap" → "kitap").

Step 7 / 10

VARIABLES

`inputText = "Kitap"`

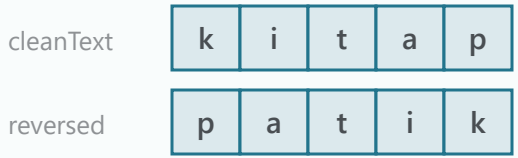
`cleanText = "kitap"`

`reversed = –`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
3      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



The clean text read backwards: **patik**.

Step 8 / 10

VARIABLES

`inputText = "Kitap"`

`cleanText = "kitap"`

`reversed = "patik"`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
3      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
▶      If cleanText = reversed then
6          Print "The text is a palindrome."
7      Else
8          Print "The text is not a palindrome."
9      End If
10 End
```



Compare letter by letter: kitap ≠ patik → **false**.

Step 9 / 10

VARIABLES

`inputText = "Kitap"`

`cleanText = "kitap"`

`reversed = "patik"`

OUTPUT

Palindrome: Dry Run (abcba and Kitap)

```
1  Start
2      Read inputText
3      cleanText = Clean(inputText)
4      reversed = Reverse(cleanText)
5      If cleanText = reversed then
6          Print "The text is a palindrome."
▶      Else
▶          Print "The text is not a palindrome."
9      End If
10 End
```



False: the `Else` branch runs.

Step 10 / 10

VARIABLES

`inputText = "Kitap"`

`cleanText = "kitap"`

`reversed = "patik"`

OUTPUT

```
The text is not a palindrome.
```

SAMPLE ALGORITHMS

Checking for a Prime

```
Start
    // Enter a number
    Read num
    // Flag: 0 and 1 are not prime
    If num < 2 then isPrime = False
    Else isPrime = True
    // Try divisors from 2 to half the number
    For divisor = 2 to num / 2
        If num % divisor = 0 then
            // If divisible, not prime
            isPrime = False
            Break
        End If
    End For
    // Print the result based on the flag
    If isPrime = True then
        Print num, " is a prime number."
    Else
        Print num, " is not a prime number."
    End If
End
```

Prime: a number greater than 1 divisible only by 1 and itself: 2, 3, 5, 7, 11, ...

- **Flag**: assume "prime" at first; lower the flag when a divisor turns up.
- `%` gives the remainder: 21 % 3 = 0 → divisible.
- Once a divisor is found there is no need to go on: **Break**.
- No number above half of n (other than n) can divide it; `num / 2` is integer division (7 / 2 = 3).
- For 0 and 1 the loop never runs, so they are checked separately.

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor: 2 3

no divisor tried yet

The user enters **7**.

Step 1 / 17

VARIABLES

`num = 7` `isPrime = –` `divisor = –`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
▶      If num < 2 then isPrime = False
▶      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor: 2 | 3

no divisor tried yet

7 ≥ 2, so the flag starts as **True**.

Step 2 / 17

VARIABLES

`num = 7` `isPrime = True` `divisor = –`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
▶      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```



7 % 2 = 1

Loop: divisor = 2 (≤ 7 / 2 = 3).

Step 3 / 17

VARIABLES

`num = 7` `isPrime = True`

`divisor = 2`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
▶          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor: 2 3

7 % 2 = 1 → not divisible

7 % 2 = 1 ≠ 0 → not divisible, go on.

Step 4 / 17

VARIABLES

`num = 7` `isPrime = True` `divisor = 2`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
▶      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```



7 % 3 = 1

Loop: divisor = 3 (≤ 7 / 2 = 3).

Step 5 / 17

VARIABLES

`num = 7` `isPrime = True` `divisor = 3`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor: 2 | 3

7 % 3 = 1 → not divisible

7 % 3 = 1 ≠ 0 → not divisible, go on.

Step 6 / 17

VARIABLES

num = 7 | isPrime = True | divisor = 3

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
▶      End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor: 2 | 3

7 % 3 = 1 → not divisible

divisor becomes 4; 4 ≤ 3 is false, the loop ends. No divisor found.

Step 7 / 17

VARIABLES

num = 7 | isPrime = True | divisor = 4

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
▶      If isPrime = True then
▶          Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor: 2 | 3

7 % 3 = 1 → not divisible

The flag is True: **7 is prime**.

Step 8 / 17

VARIABLES

`num = 7` `isPrime = True` `divisor = 4`

OUTPUT

```
7 is a prime number.
```

Prime: Dry Run (7 and 21)

```
1  Start
▶      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor:	2	3	4	5	6	7	8	9	10

no divisor tried yet

Second test: the user enters **21**.

Step 9 / 17

VARIABLES

`num = 21` `isPrime = –` `divisor = –`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
▶      If num < 2 then isPrime = False
▶      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor:

2	3	4	5	6	7	8	9	10

no divisor tried yet

21 ≥ 2, so the flag starts as **True**.

Step 10 / 17

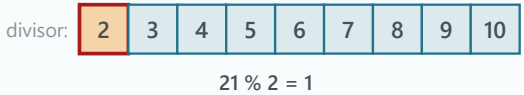
VARIABLES

`num = 21` `isPrime = True` `divisor = –`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
▶      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```



Loop: divisor = 2 (≤ 21 / 2 = 10).

Step 11 / 17

VARIABLES

`num = 21` `isPrime = True` `divisor = 2`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```

divisor:	2	3	4	5	6	7	8	9	10

21 % 2 = 1 → not divisible

21 % 2 = 1 ≠ 0 → not divisible, go on.

Step 12 / 17

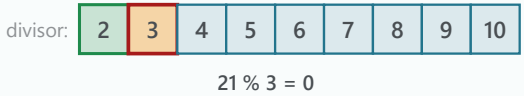
VARIABLES

`num = 21` `isPrime = True` `divisor = 2`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
▶      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```



Loop: divisor = 3 (≤ 21 / 2 = 10).

Step 13 / 17

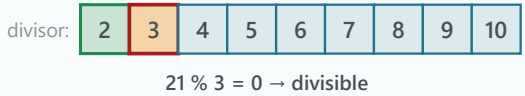
VARIABLES

`num = 21` `isPrime = True` `divisor = 3`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```



21 % 3 = 0 → divisible!

Step 14 / 17

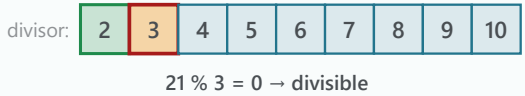
VARIABLES

`num = 21` `isPrime = True` `divisor = 3`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
▶              isPrime = False
8              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```



21 % 3 = 0 → divisible

The flag is lowered: isPrime = **False**.

Step 15 / 17

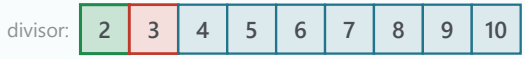
VARIABLES

`num = 21` `isPrime = False` `divisor = 3`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
▶              Break
9          End If
10     End For
11     If isPrime = True then
12         Print num, " is a prime number."
13     Else
14         Print num, " is not a prime number."
15     End If
16 End
```



21 % 3 = 0 → divisible

Break: the remaining divisors are skipped.

Step 16 / 17

VARIABLES

`num = 21` `isPrime = False` `divisor = 3`

OUTPUT

Prime: Dry Run (7 and 21)

```
1  Start
2      Read num
3      If num < 2 then isPrime = False
4      Else isPrime = True
5      For divisor = 2 to num / 2
6          If num % divisor = 0 then
7              isPrime = False
8              Break
9          End If
10     End For
▶      If isPrime = True then
12         Print num, " is a prime number."
▶      Else
▶          Print num, " is not a prime number."
15     End If
16 End
```



21 % 3 = 0 → divisible

The flag is False: the `Else` branch runs.

Step 17 / 17

VARIABLES

`num = 21` `isPrime = False` `divisor = 3`

OUTPUT

```
21 is not a prime number.
```

SAMPLE ALGORITHMS

Computing a Power

```
Start
    // Enter a number
    Read base
    // Enter an exponent
    Read exp
    // Initialize the result
    result = 1
    // Compute the result using the exponent
    For i = 1 to exp
        result = result * base
    End For
    Print "Result: ", result
End
```

Definition: $base^{exp}$ = base × base × … × base (exp times); $base^{0}$ = 1.

base	exp	Expected
3	4	81
2	10	1024
5	0	1

The same pattern as factorial: the accumulator starts at 1 and each round does one multiplication. The difference: it always multiplies by the **same** number (the base). With exponent 0 the loop never runs and the result stays 1.

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
6         result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 1

The user enters base = **3**.

Step 1 / 13

VARIABLES

`base = 3` `exp = –` `result = –` `i = –`

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
▶     Read exp
4     result = 1
5     For i = 1 to exp
6         result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 1

The user enters exp = **4**: we will compute 3^4.

Step 2 / 13

VARIABLES

`base = 3` `exp = 4` `result = –` `i = –`

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
▶     result = 1
5     For i = 1 to exp
6         result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 1

The accumulator starts at 1.

Step 3 / 13

VARIABLES

base = 3 | exp = 4 | result = 1 | i = –

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
6         result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 1

Loop: i = 1.

Step 4 / 13

VARIABLES

base = 3 | exp = 4 | result = 1 | i = 1

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
          result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 3

result = 1 × 3 = **3**.

Step 5 / 13

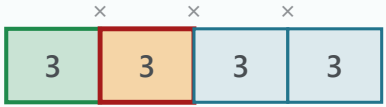
VARIABLES

`base = 3` `exp = 4` `result = 3` `i = 1`

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
      For i = 1 to exp
6         result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 3

Loop: i = 2.

Step 6 / 13

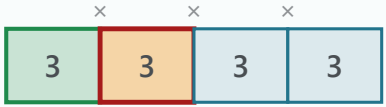
VARIABLES

base = 3 | exp = 4 | result = 3 | i = 2

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
          result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 9

result = 3 × 3 = **9**.

Step 7 / 13

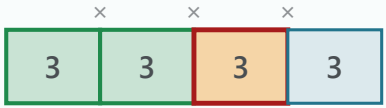
VARIABLES

`base` = **3** `exp` = **4** `result` = **9** `i` = **2**

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
6         result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 9

Loop: i = 3.

Step 8 / 13

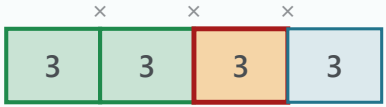
VARIABLES

base = 3 | exp = 4 | result = 9 | i = 3

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
          result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 27

result = 9 × 3 = **27**.

Step 9 / 13

VARIABLES

`base = 3` `exp = 4` `result = 27` `i = 3`

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
6         result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 27

Loop: i = 4.

Step 10 / 13

VARIABLES

base = 3 | exp = 4 | result = 27 | i = 4

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
          result = result * base
7     End For
8     Print "Result: ", result
9 End
```



result = 81

result = 27 × 3 = **81**.

Step 11 / 13

VARIABLES

`base = 3` `exp = 4` `result = 81` `i = 4`

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
6         result = result * base
      End For
8     Print "Result: ", result
9 End
```



result = 81

i = 5 > 4: the loop ends.

Step 12 / 13

VARIABLES

base = 3 | exp = 4 | result = 81 | i = 5

OUTPUT

Powers: Dry Run (3^4)

```
1 Start
2     Read base
3     Read exp
4     result = 1
5     For i = 1 to exp
6         result = result * base
7     End For
▶     Print "Result: ", result
9 End
```



result = 81

The result is printed: $3^4 = 81$.

Step 13 / 13

VARIABLES

`base = 3` `exp = 4` `result = 81` `i = 5`

OUTPUT

```
Result: 81
```

Search Algorithms

- Asks **whether** a given item is in a collection (and if so, where).
- A basic operation for processing data and retrieving information.
- It helps find specific items quickly and correctly.
- **Uses:** finding a book in a library, a page on a website, a record in a database, a name in a phone book.

Problem definition

Input:

- A collection of objects, say a "Basket".
- A particular object, say an "Apple".

Output:

- `true` if the "Apple" is in the basket.
- `false` if it is not.

In code we usually return the **index** (position) of the item, or `-1` if it is missing.

SEARCHING

Three Search Methods

Random

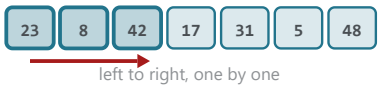
Each step picks a random item and compares it with the target.



No order needed · **no guarantee** of finding it

Linear

Compares the items one by one from start to end.



No order needed · at most **n** comparisons

Binary

Looks at the middle item and discards half the range each step.



Array **must be sorted** · at most ≈ $\log_2 n$ comparisons

Random Search

- Checks items **at random**: each step picks a random item.
- Runs until it finds the target or runs out of tries.
- Pseudocode:

1. Pick a random item.
2. Compare it with the target.
3. If it is the target, stop.
4. Otherwise repeat.

Pros

+ Simple.
+ The order of the data does not matter.

Cons

– Not always effective: it may pick the same item again and again.
– It can raise the odds of finding the item but guarantees nothing.
– If the target is absent it runs forever; it needs a limit on tries.
– In the worst case it scans the whole collection (or more).

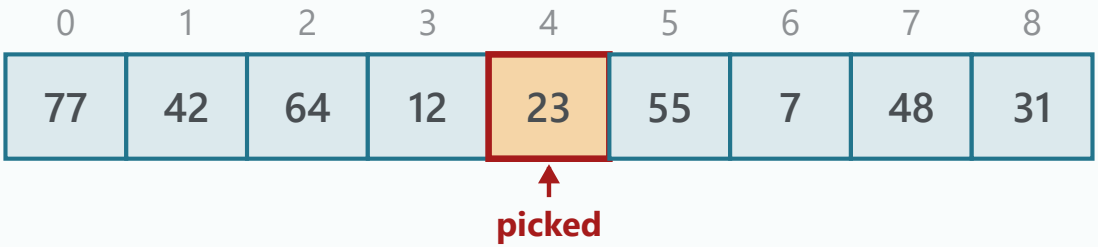
Random Search: Step by Step



target: 48

We have an unsorted array; the target is **48**.

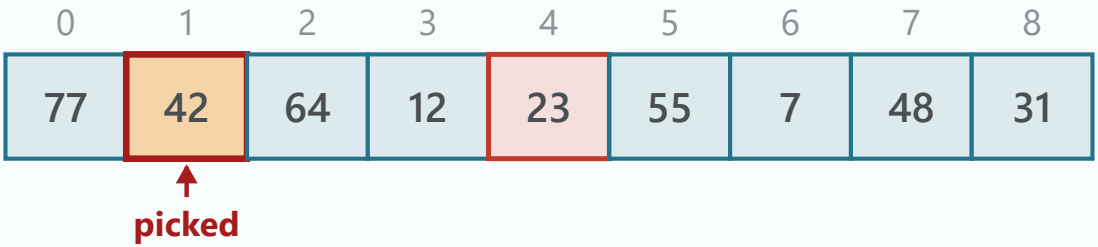
Random Search: Step by Step



comparing 23 with 48

Step 1: the random pick is 23; 23 ≠ 48, not found yet. Pick another.

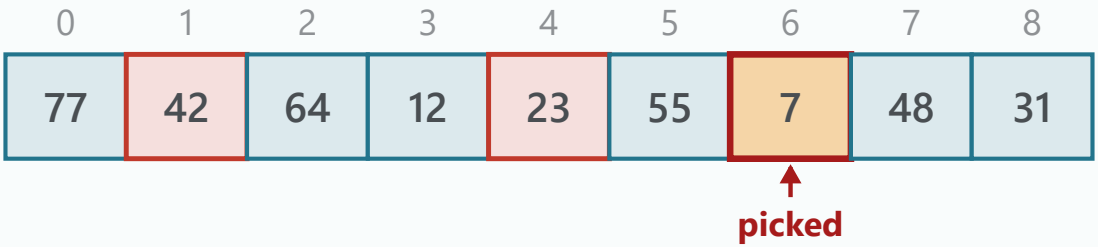
Random Search: Step by Step



comparing 42 with 48

Step 2: the random pick is 42; 42 ≠ 48, not found yet. Pick another.

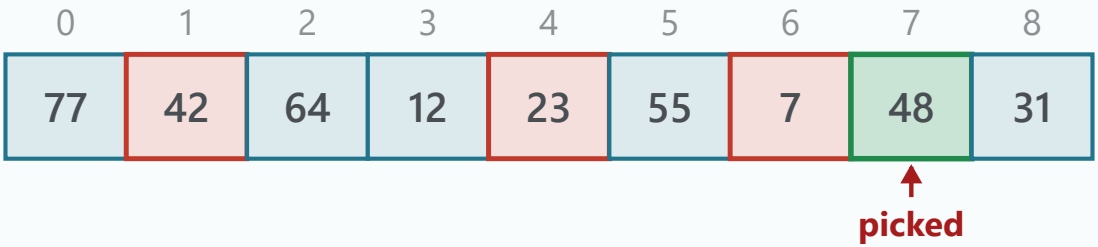
Random Search: Step by Step



comparing 7 with 48

Step 3: the random pick is 7; 7 ≠ 48, not found yet. Pick another.

Random Search: Step by Step



comparing 48 with 48

Step 4: the random pick is 48 → target **found**; the search ends (4 tries).

SEARCHING

Linear Search

- The most basic way to find an item in a collection.
- It checks the items **in order** looking for the target.
- The items need not be sorted.
- Good for small collections.

Steps:

1. Put the items in a list (array).
2. Starting from the first, compare each item with the target in turn.
3. If it is the target, stop and return its position.
4. Otherwise move to the next item.
5. Continue to the end of the list; reaching the end means the item is absent.

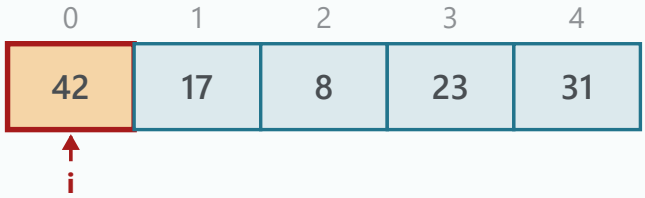
Pros

+ Simple and easy to follow.
+ Works in any order (on unsorted data).
+ The result is certain: if the item is there, it is found.

Cons

– Slow on large collections: it may have to check every item (worst case n comparisons).

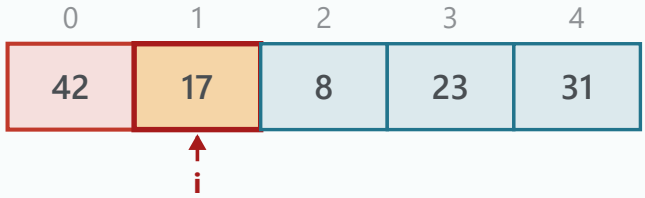
Linear Search: Step by Step



arr[0] = 42 vs 23

Step 1: the target is 23. Start with the first item (42) and compare it with the target.

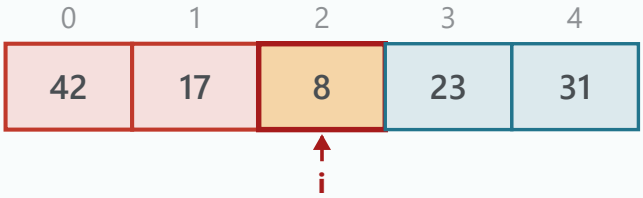
Linear Search: Step by Step



arr[1] = 17 vs 23

Step 2: the first item (42) does not match 23. Move on: 17.

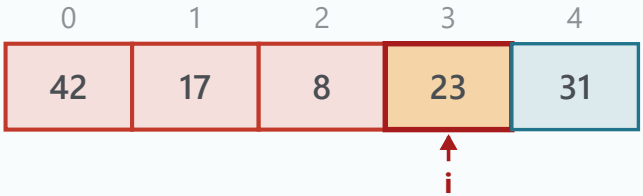
Linear Search: Step by Step



arr[2] = 8 vs 23

Step 3: the second item (17) does not match either. Next: 8.

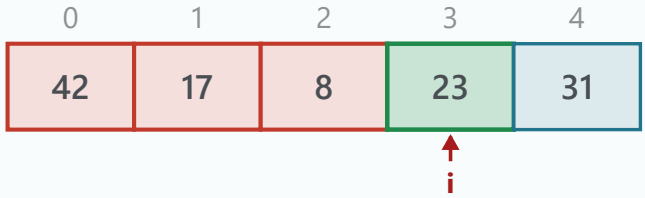
Linear Search: Step by Step



arr[3] = 23 vs 23

Step 4: the third item (8) does not match either. Next: 23.

Linear Search: Step by Step



arr[3] = 23: found

Step 5: the fourth item (23) **matches**; the search ends here (4 comparisons).

SEARCHING

Binary Search

- A fast way to find an item in a collection.
- Works on **sorted** data (ascending or descending).
- Each step discards **half** of the list.
- Ideal for searching large collections.

Steps:

1. Take the middle item and compare it with the target.
2. If it is the target, stop.
3. If the target is smaller, focus on the left half.
4. If the target is larger, focus on the right half.
5. Repeat until the target is found or the range is empty.

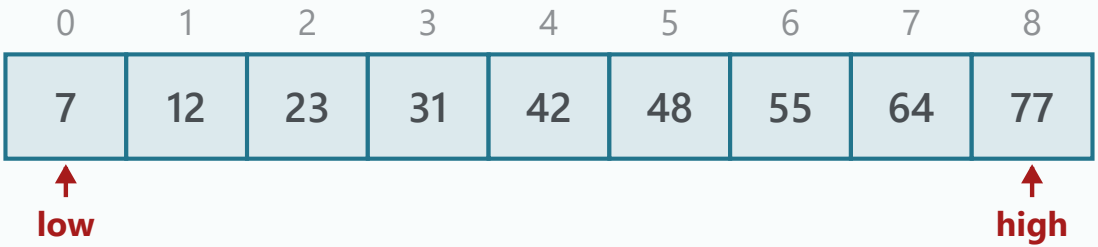
Pros

+ Fast: at most 20 comparisons for 1,000,000 items.
+ Very fast on large collections.

Cons

– The data must be sorted (and sorting has a cost).
– On an unsorted array it can give a wrong answer (without any error!).

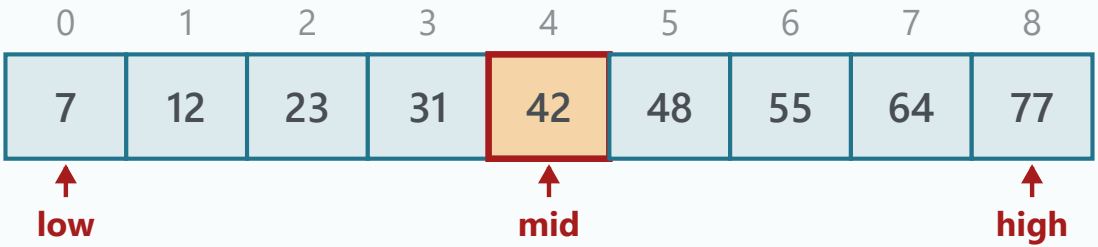
Binary Search: Step by Step



target: 48

We have a sorted array; the target is **48**. At first the range is the whole array: low = 0, high = 8.

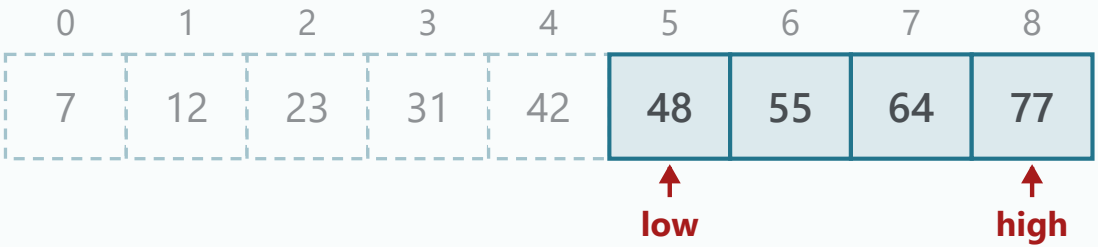
Binary Search: Step by Step



mid = (0 + 8) / 2 = 4 → 42

Step 1: take the middle item (42) and compare it with the target (48).

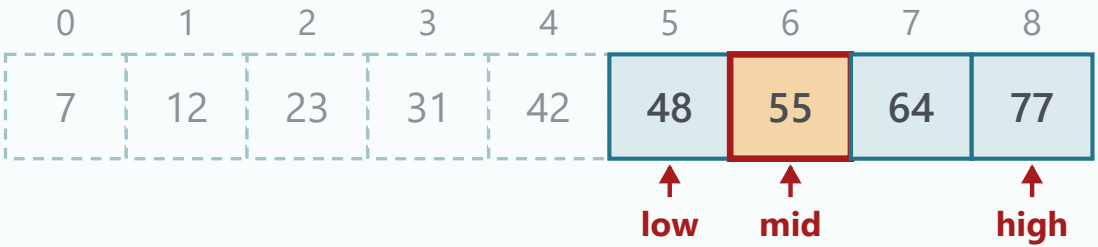
Binary Search: Step by Step



48 > 42 → right half

Step 2: 48 is greater than the middle item (42), so the target is in the right half. The left half is dropped: [48, 55, 64, 77].

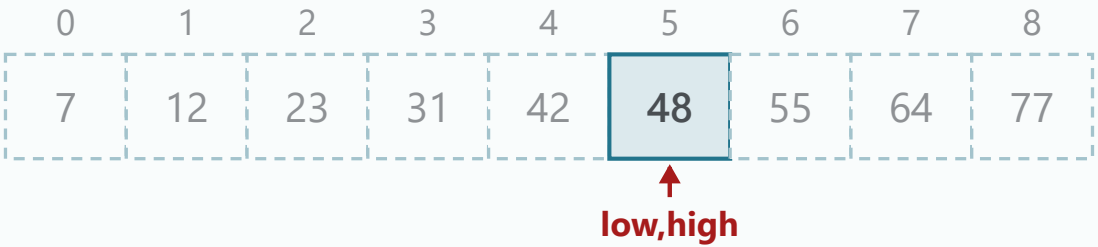
Binary Search: Step by Step



mid = (5 + 8) / 2 = 6 → 55

Step 3: take the middle of the right half. With 4 items there are two middles (55 and 64); `(low + high) / 2` picks the left one, 55.

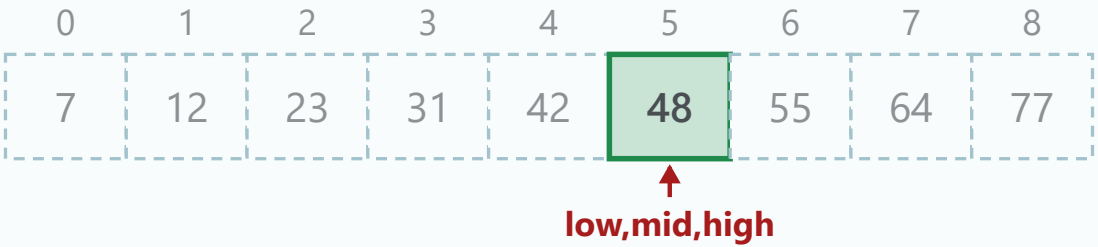
Binary Search: Step by Step



48 < 55 → left half

Step 4: 48 is less than the middle item (55); the right side is dropped. Only [48] remains.

Binary Search: Step by Step



mid = 5 → 48 = 48

Step 5: the middle item, 48, equals the target: **found**. Just **3 comparisons** in a 9-item array.

Binary vs. Linear Search

Linear search

- Checks the items one by one, in order.
- Runs until it finds the target or the list ends.
- Data may be unsorted.
- Can be slow on large collections.

Binary search

- Finds the target quickly in sorted data.
- Data must be sorted (ascending or descending).
- Each step discards half of what remains.
- Very fast on large collections.

Comparisons in the worst case

Items n	Linear (n)	Binary ($\lfloor \log_2 n \rfloor + 1$)
9	9	4
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

When n doubles, linear search does twice the work; binary search does just **one** more comparison.

SEARCHING

Linear and Binary Search: Try It

Target 48 **linear search** **binary search** **new array**

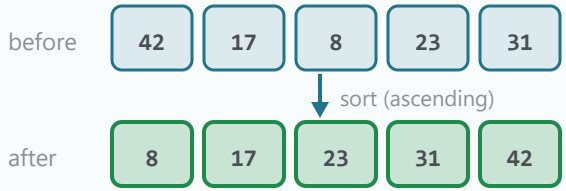
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	7	12	18	23	31	36	42	48	55	61	64	70	77	89

A sorted array is ready. Enter a number and pick a method.

The array is sorted, so both methods give correct results. Orange: item being checked; red: discarded items; green: found item. Try different targets (first, last, a missing number) and watch the comparison counts.

Sorting Algorithms

- **Arrange** the items of a collection by some rule.
- Usually in ascending or descending order.
- A basic step for data analysis and searching.
- Sorted data speeds up search: binary search works only on sorted arrays.



Random

Shuffles the items; no guarantee of order.

Selection sort

Selects the smallest and puts it first.

Bubble sort

Compares neighbors and moves the larger toward the end.

Faster methods exist (merge sort, quicksort); Java's built-in `Arrays.sort` uses refined versions of them.

Random Sorting (Shuffling)

- Putting a collection into a random order: the items are **shuffled**.
- Uses: shuffling a deck of cards, reordering exam questions, generating test data.
- Some algorithms (e.g. quicksort) shuffle their input first to avoid bad cases.

Pros

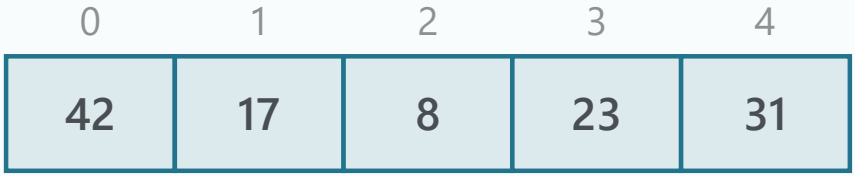
+ Gives a different arrangement every time.
+ Built into Java: `Collections.shuffle(list)`.

Cons

– Does **not** sort the data.

– "Shuffle until sorted" (*bogosort*) is hopelessly slow: only one of the 3,628,800 orderings of 10 items is sorted.

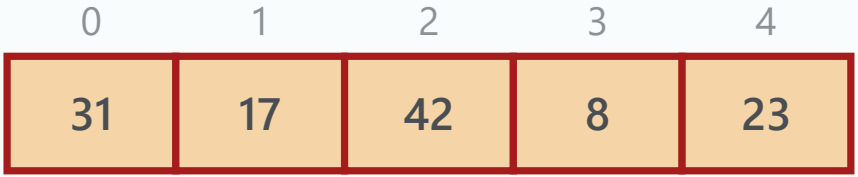
Random Sorting: Step by Step



initial array

We start with an unsorted array: [42, 17, 8, 23, 31].

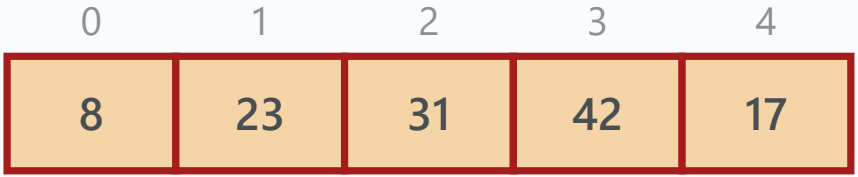
Random Sorting: Step by Step



shuffle 1

Step 1: the items are put in a random order: [31, 17, 42, 8, 23]. Each shuffle gives a different result.

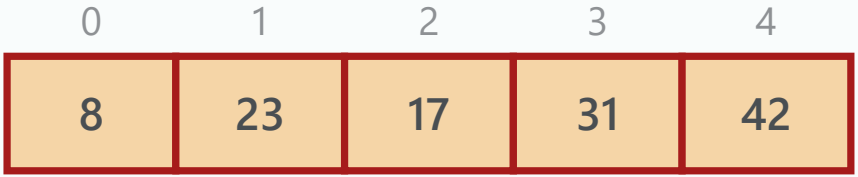
Random Sorting: Step by Step



shuffle 2

Step 2: shuffled again: [8, 23, 31, 42, 17].

Random Sorting: Step by Step



shuffle 3

Step 3: shuffled again: [8, 23, 17, 31, 42]. Closer, but still not sorted: shuffling guarantees nothing.

Selection Sort

- A simple comparison-based sorting algorithm.
- It repeatedly finds the smallest (or largest) item and swaps it into place.

Steps:

1. Find the smallest item.
2. Move it to the front (swap with the first item).
3. Find the smallest of the rest.
4. Move it to the second position.
5. Repeat to the end.

Pros

+ Simple and easy to follow.
+ Good for small collections.
+ At most n − 1 swaps.

Cons

− Slow on large collections: always n(n − 1)/2 comparisons (even if already sorted).

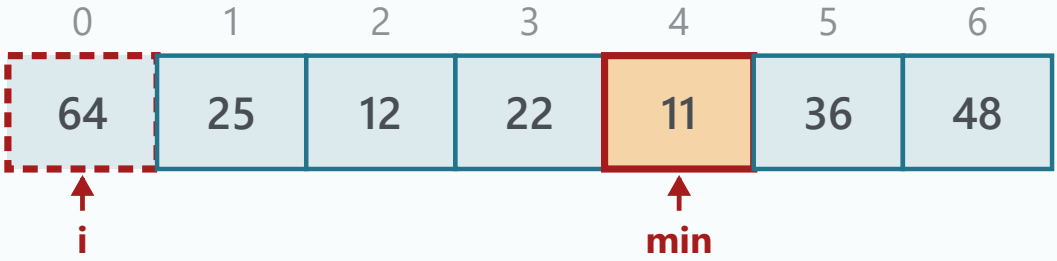
Selection Sort: Step by Step



start

We start with an unsorted array: [64, 25, 12, 22, 11, 36, 48].

Selection Sort: Step by Step



min of unsorted part: 11

Step 1: the minimum of the unsorted part (indices 0–6) is **11**; it will swap with the 64 at position 1.

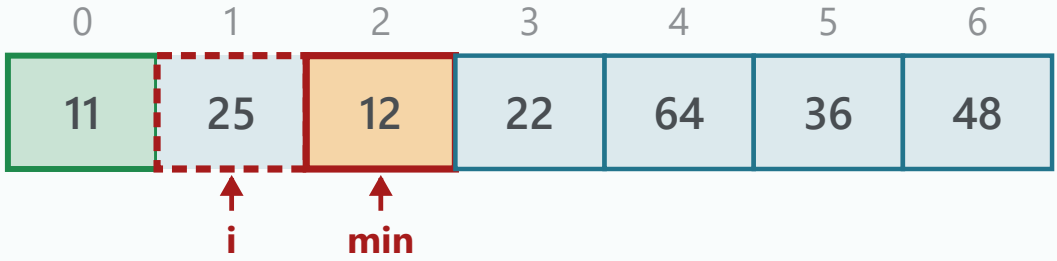
Selection Sort: Step by Step



after step 1

After step 1: [11, 25, 12, 22, 64, 36, 48]. The green part is sorted and in place.

Selection Sort: Step by Step



min of unsorted part: 12

Step 2: the minimum of the unsorted part (indices 1–6) is **12**; it will swap with the 25 at position 2.

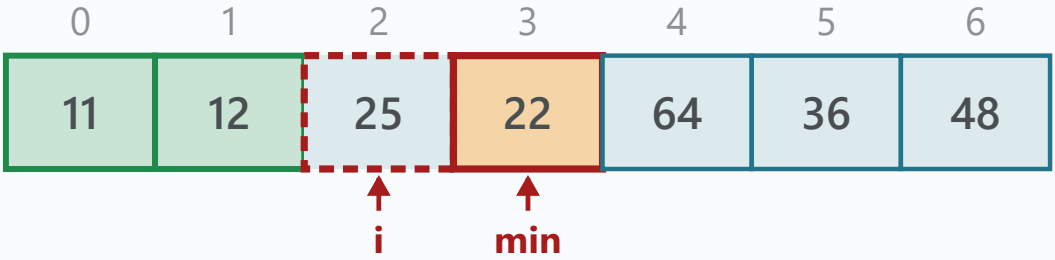
Selection Sort: Step by Step



after step 2

After step 2: [11, 12, 25, 22, 64, 36, 48]. The green part is sorted and in place.

Selection Sort: Step by Step



min of unsorted part: 22

Step 3: the minimum of the unsorted part (indices 2–6) is **22**; it will swap with the 25 at position 3.

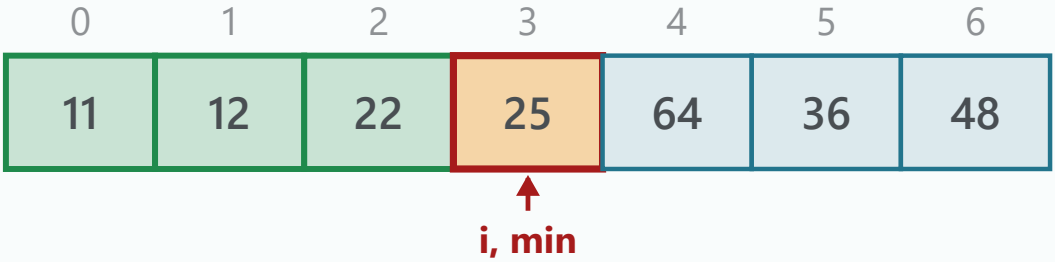
Selection Sort: Step by Step



after step 3

After step 3: [11, 12, 22, 25, 64, 36, 48]. The green part is sorted and in place.

Selection Sort: Step by Step



min of unsorted part: 25

Step 4: the minimum of the unsorted part (indices 3–6) is **25**. Already in place: no change.

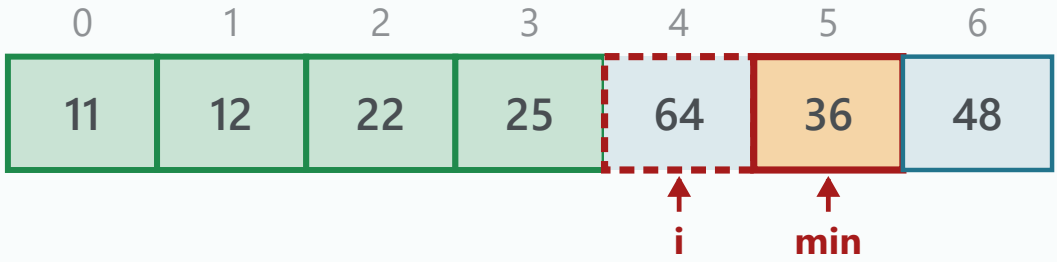
Selection Sort: Step by Step



after step 4

After step 4: [11, 12, 22, 25, 64, 36, 48]. The green part is sorted and in place.

Selection Sort: Step by Step



min of unsorted part: 36

Step 5: the minimum of the unsorted part (indices 4–6) is **36**; it will swap with the 64 at position 5.

Selection Sort: Step by Step



after step 5

After step 5: [11, 12, 22, 25, 36, 64, 48]. The green part is sorted and in place.

Selection Sort: Step by Step



min of unsorted part: 48

Step 6: the minimum of the unsorted part (indices 5–6) is **48**; it will swap with the 64 at position 6.

Selection Sort: Step by Step



after step 6

After step 6: [11, 12, 22, 25, 36, 48, 64]. The green part is sorted and in place. The last item falls into place on its own: **6 steps** suffice for 7 items.

Bubble Sort

- A simple algorithm that sorts by comparing items in sequence.
- It compares neighbors and carries the larger one toward the end, like a **bubble** rising to the surface.

Steps:

1. Start at the beginning.
2. Compare two neighbors; swap if needed so the larger moves right.
3. Repeat to the end of the collection (one **pass**).
4. The largest item is now last, so treat it as sorted.
5. Repeat for the rest; stop when a pass makes no swaps.

Pros

+ Simple and easy to follow.
+ Good for small or nearly sorted collections (with early exit).

Cons

– Very slow on large collections: in the worst case $n(n - 1)/2$ comparisons and as many swaps.

Bubble Sort: Step by Step



start

We start with an unsorted array: [64, 25, 12, 22, 11, 36, 48].

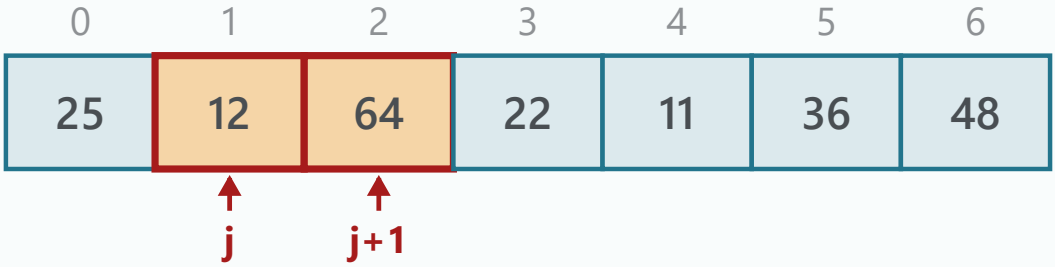
Bubble Sort: Step by Step



pass 1 · swapped

Step 1 (pass 1): compare 64 and 25; 64 > 25, so swap → [25, 64, 12, 22, 11, 36, 48].

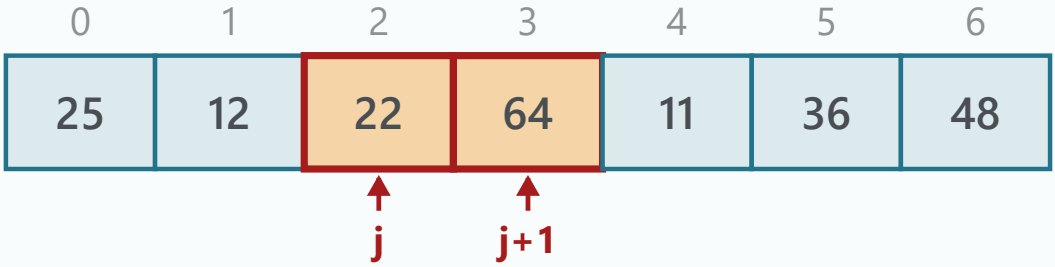
Bubble Sort: Step by Step



pass 1 · swapped

Step 2 (pass 1): compare 64 and 12; 64 > 12, so swap → [25, 12, 64, 22, 11, 36, 48].

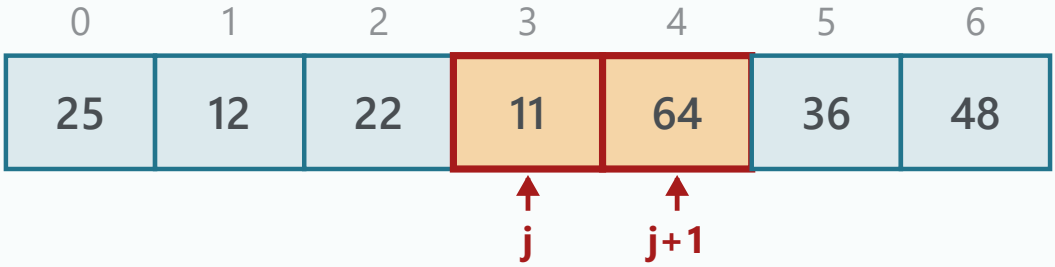
Bubble Sort: Step by Step



pass 1 · swapped

Step 3 (pass 1): compare 64 and 22; 64 > 22, so swap → [25, 12, 22, 64, 11, 36, 48].

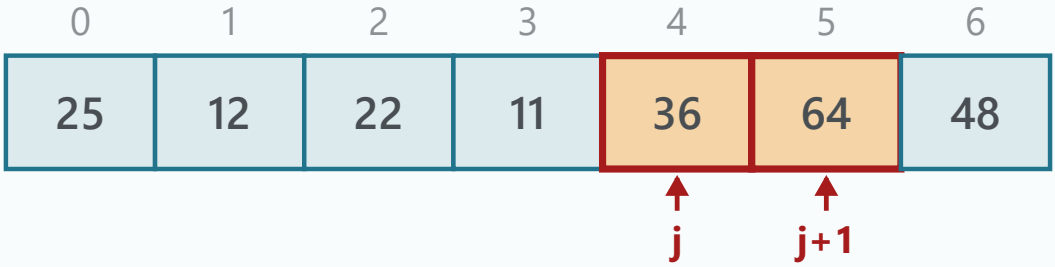
Bubble Sort: Step by Step



pass 1 · swapped

Step 4 (pass 1): compare 64 and 11; 64 > 11, so swap → [25, 12, 22, 11, 64, 36, 48].

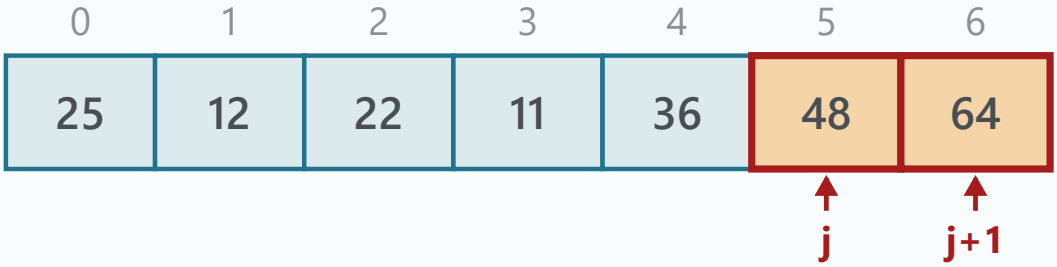
Bubble Sort: Step by Step



pass 1 · swapped

Step 5 (pass 1): compare 64 and 36; 64 > 36, so swap → [25, 12, 22, 11, 36, 64, 48].

Bubble Sort: Step by Step



pass 1 · swapped

Step 6 (pass 1): compare 64 and 48; 64 > 48, so swap → [25, 12, 22, 11, 36, 48, 64].

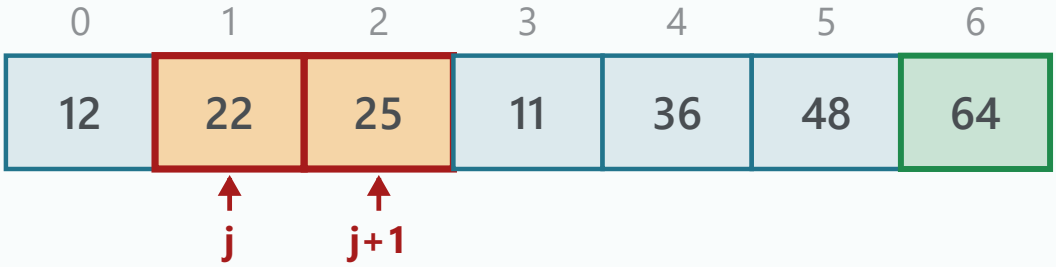
Bubble Sort: Step by Step



pass 2 · swapped

Step 7 (pass 2): compare 25 and 12; 25 > 12, so swap → [12, 25, 22, 11, 36, 48, 64].

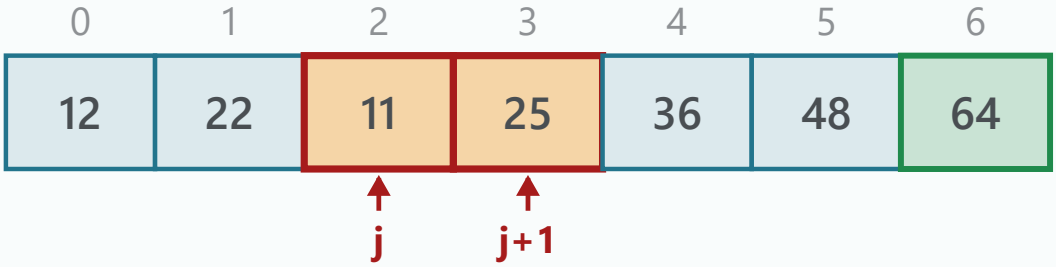
Bubble Sort: Step by Step



pass 2 · swapped

Step 8 (pass 2): compare 25 and 22; 25 > 22, so swap → [12, 22, 25, 11, 36, 48, 64].

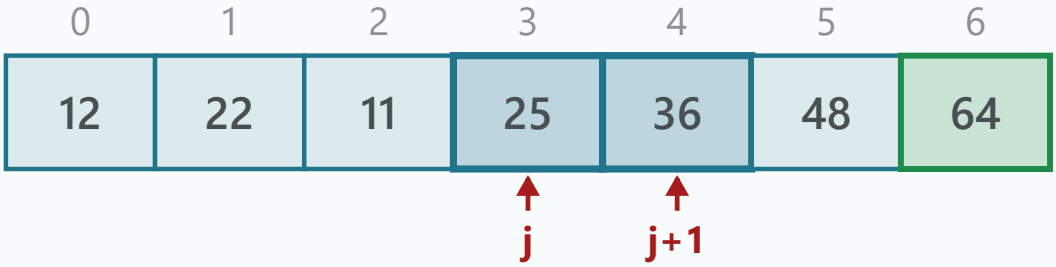
Bubble Sort: Step by Step



pass 2 · swapped

Step 9 (pass 2): compare 25 and 11; 25 > 11, so swap → [12, 22, 11, 25, 36, 48, 64].

Bubble Sort: Step by Step



pass 2 · no change

Step 10 (pass 2): compare 25 and 36; 25 < 36, no change.

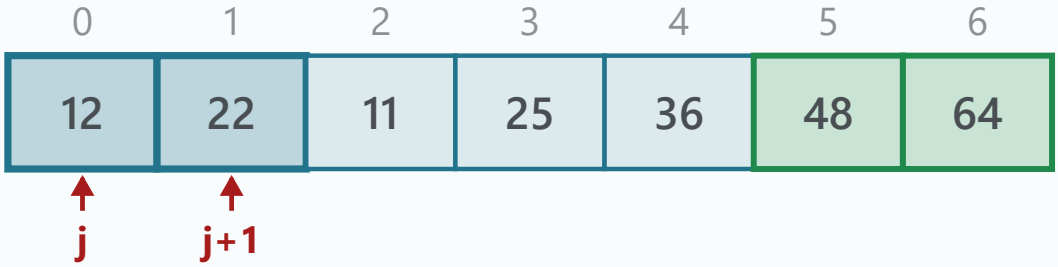
Bubble Sort: Step by Step



pass 2 · no change

Step 11 (pass 2): compare 36 and 48; 36 < 48, no change.

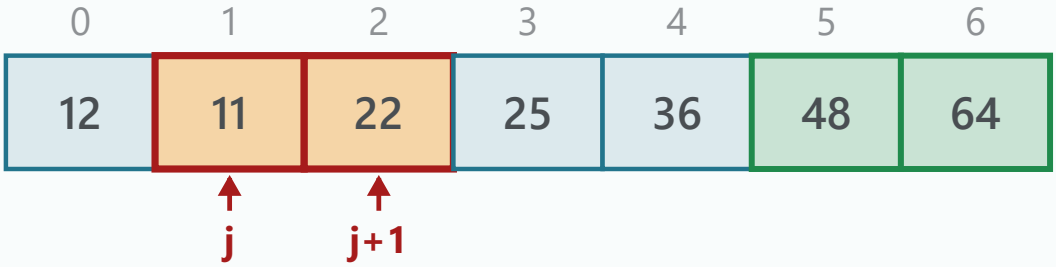
Bubble Sort: Step by Step



pass 3 · no change

Step 12 (pass 3): compare 12 and 22; 12 < 22, no change.

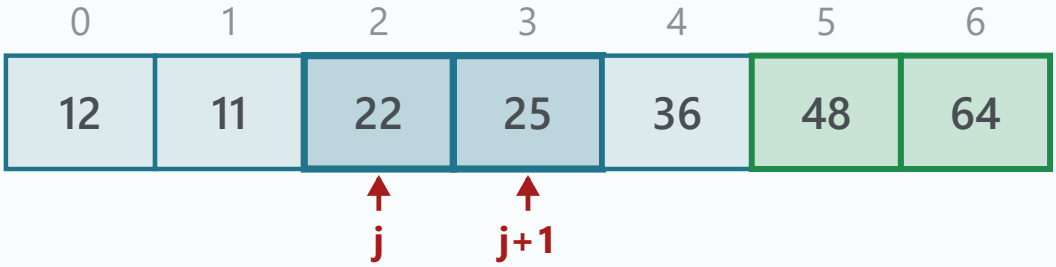
Bubble Sort: Step by Step



pass 3 · swapped

Step 13 (pass 3): compare 22 and 11; 22 > 11, so swap → [12, 11, 22, 25, 36, 48, 64].

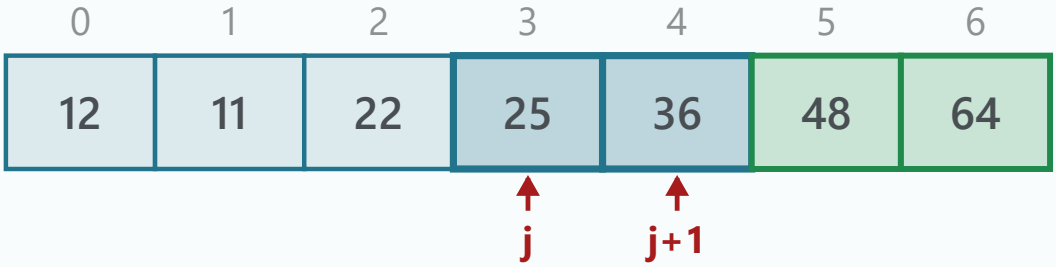
Bubble Sort: Step by Step



pass 3 · no change

Step 14 (pass 3): compare 22 and 25; 22 < 25, no change.

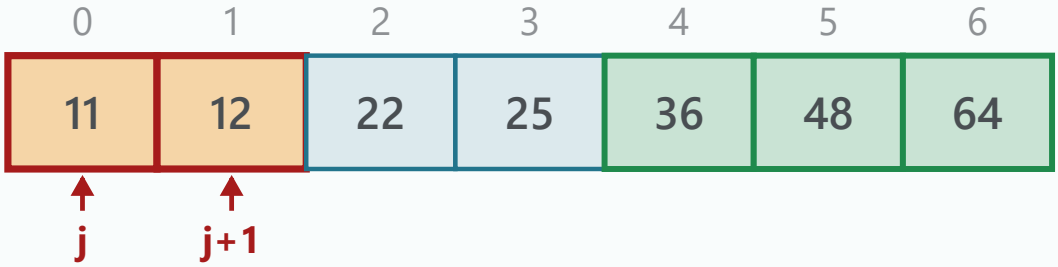
Bubble Sort: Step by Step



pass 3 · no change

Step 15 (pass 3): compare 25 and 36; 25 < 36, no change.

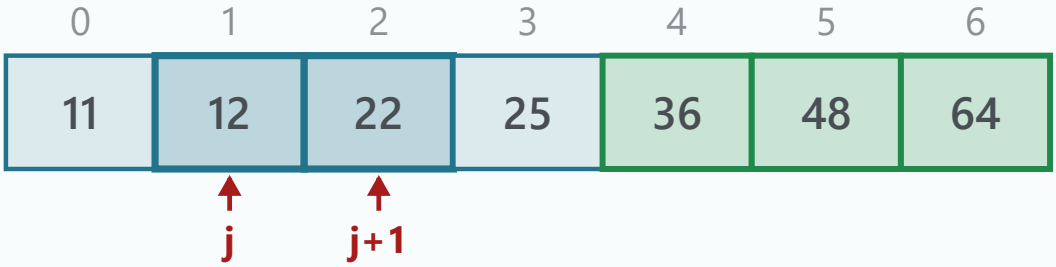
Bubble Sort: Step by Step



pass 4 · swapped

Step 16 (pass 4): compare 12 and 11; 12 > 11, so swap → [11, 12, 22, 25, 36, 48, 64].

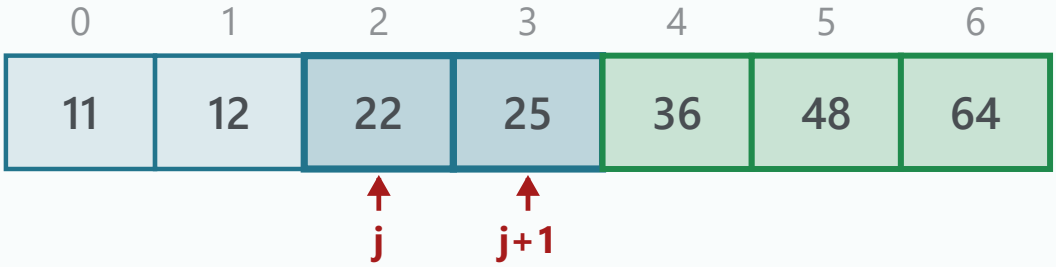
Bubble Sort: Step by Step



pass 4 · no change

Step 17 (pass 4): compare 12 and 22; 12 < 22, no change.

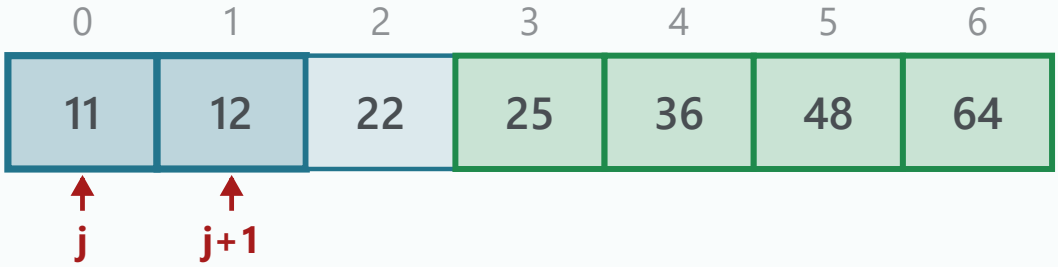
Bubble Sort: Step by Step



pass 4 · no change

Step 18 (pass 4): compare 22 and 25; 22 < 25, no change.

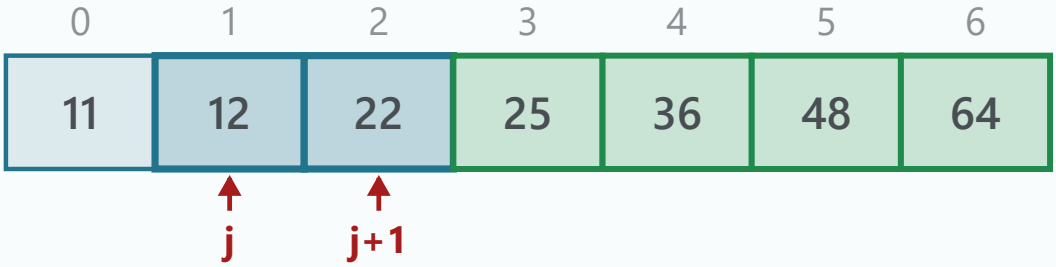
Bubble Sort: Step by Step



pass 5 · no change

Step 19 (pass 5): compare 11 and 12; 11 < 12, no change.

Bubble Sort: Step by Step



pass 5 · no change

Step 20 (pass 5): compare 12 and 22; 12 < 22, no change.

Bubble Sort: Step by Step



array sorted

Pass 5 made no swaps: the array is sorted and the algorithm stops. In total **20 comparisons**, 11 swaps.

Comparing the Sorting Methods

	Random (shuffle)	Selection	Bubble
Idea	Produce a random order	Select the smallest, put it first	Compare neighbors, move the larger to the end
Sorted result?	No guarantee	Yes	Yes
Comparisons (n items)	—	always $n(n-1)/2$	worst $n(n-1)/2$, $n-1$ if sorted
Swaps	$n-1$	at most $n-1$	worst $n(n-1)/2$
Sample array ($n = 7$)	—	21 comparisons, 5 actual swaps	20 comparisons, 11 swaps

Selection and bubble sort grow with n^2: about 500,000 comparisons for 1,000 items and about 500 billion for 1,000,000. Large data calls for smarter algorithms (e.g. merge sort, $n \cdot \log_2 n$).

SORTING

Five Sorting Algorithms Compared

Algorithm	Idea	Average	Worst	Note
Bubble sort	compare two neighbors, swap if needed	$O(n^2)$	$O(n^2)$	simple; poor on large data
Selection sort	find the smallest, put it first; repeat on the rest	$O(n^2)$	$O(n^2)$	slow on large data
Insertion sort	insert the next item into its place in the sorted part	$O(n^2)$	$O(n^2)$	good for small or nearly sorted data
Merge sort	split in two, sort each half recursively, merge	$O(n \log n)$	$O(n \log n)$	stable; good on large data
Quicksort	pick a pivot, smaller left, larger right (divide and conquer)	$O(n \log n)$	$O(n^2)$	very fast on average

An $n \cdot \log n$ algorithm makes about 20 million comparisons for 1,000,000 items; an n^2 algorithm makes about 500 billion.

SORTING

Selection and Bubble Sort: Try It

Method selection sort | one step | run to end | new array

0	1	2	3	4	5	6	7
14	51	73	58	77	20	60	12

Selection sort ready. Advance with "one step".

Each "one step" makes a single comparison. Orange: items being compared; green: items in their final place. Compare the two methods' comparison and swap counts on the same array.

Summary

Algorithm	Key idea	Structure	Example
Ordering two / three numbers	Compare, print the smaller first	If … Else, nested decisions	13, 11 → 11 13
Factorial	Multiply 1 through n	Loop + accumulator (starts at 1)	5! = 120
Fibonacci	Each term is the sum of the previous two	Loop + shifting variables	0 1 1 2 3
Palindrome	Clean, reverse, compare	String operations, decision	kayak → yes
Prime	Look for a divisor in 2 … n/2	Flag + Break	21 → no
Power	Multiply the base exp times	Loop + accumulator	$3^4 = 81$
Random / linear / binary search	Try at random / scan in order / halve	Loop, decision; binary needs a sorted array	at most 4 comparisons for 9 items (binary)
Shuffle / selection / bubble	Random order / select the smallest / swap neighbors	Nested loops, swaps	n(n − 1)/2 comparisons

Same path every time: define the problem and tests → pseudocode → dry run → Java → run the tests.

SUMMARY

Common Mistakes: Algorithms

Wrong accumulator start

Starting with `factorial = 0` makes every product 0. Start sums at 0 and products at 1.

Off-by-one loop bounds

Writing 2 … n instead of 2 … n − 1 in Fibonacci gives n + 1 terms. Check bounds with a small dry run.

Binary search on unsorted data

Searching [77, 42, 64, 12, 23, 55, 7, 48, 31] for 77 returns `-1`: no error message, wrong answer.

Forgetting edge cases

Test 0 and 1 for primes, 0! for factorial and n = 1 for Fibonacci separately.

Random search with no stopping rule

If the target is absent the loop runs forever; it needs a limit on tries.

Swapping without a temporary

`a = b; b = a;` makes both equal to b. Save `temp = a` first.

SUMMARY

Common Mistakes: Java

Mistake	What happens?
Forgot `import java.util.Scanner;`	Compile error: `error: cannot find symbol … symbol: class Scanner`
Missing `;` at line end: `int sum = 0`	Compile error: `error: ';' expected`
Single `=` in a comparison: `if (num % divisor = 0)`	Compile error: `error: unexpected type … required: variable, found: value`. Test equality with `==`.
Comparing strings with `==`	Compiles, but gives `false` for "abcba" and its reverse. Use `input.equals(rev)`.
Looping with `i <= arr.length`	Runtime error: `ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5`
Using `int` for 13!	No error, wrong result: 1932053504 (should be 6227020800). Use `long`.

SUMMARY

Sample Problems

1. Trace the factorial algorithm for **num = 4**: what are `i` and `factorial` on each round?
2. What does the Fibonacci algorithm print for **n = 7**? How many times does the loop run?
3. How many divisors does the prime algorithm try for **25**, and what does it print?
4. How many comparisons does linear search make for **31** and **50** in [42, 17, 8, 23, 31]?
5. Which items does binary search inspect for **12** and **77** in [7, 12, 23, 31, 42, 48, 55, 64, 77]?
6. In the power algorithm with base = 2, exp = 10, how many times does the loop run and what is the result?

▶ **Show answers**

SUMMARY

Sample Problems: Sorting

1. Sort [29, 10, 14, 37, 13] with selection sort, writing the array after each step.
2. Apply bubble sort to [5, 1, 4, 2, 8]. What is the array after the first pass? How many passes with early exit?
3. How many comparisons does selection sort make on 7 items? Would it change if the array were already sorted?
4. How many comparisons does bubble sort (with early exit) make on a sorted array?

▶ **Show answers**

SUMMARY

Exercises

1. Write pseudocode and a Java program that find the **largest** of three integers; test all 6 orders.
2. Sum 1 to n with a loop. For n = 100, compare with the formula n(n + 1)/2.
3. Find the digit sum of a number (hint: `% 10` and `/ 10`). What is it for 1234?
4. Change linear search to count **how many times** the target occurs.
5. Change bubble sort to sort in **descending** order.
6. Write Fibonacci in Java with `int` and run it for n = 50. After which term do the results go wrong? Why?

▶ **Answers / hints**

References

- P. Deitel and H. Deitel, *Java: How to Program (Early Objects)*, 11th ed., Pearson, 2017 (Chapter 4: algorithms and pseudocode; Chapter 19: searching and sorting).
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (free online).
- R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 1.1 binary search, 2.1 elementary sorts).
- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.
- Oracle, *The Java Tutorials*: Language Basics, docs.oracle.com/javase/tutorial.

The End

Chapter 1.3: Algorithms

Previous: Chapter 1.2: Computation and Computational Thinking · Next: Chapter 1.4: Flowcharts