Chapter 12: Recursion

Computer Fundamentals and Introduction to Programming

Base case · Calls · Stack · Efficiency · Memoization

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/computer/slides/Chapter_12_Recursion.html

INTRODUCTION

Learning Outcomes

Structure

Identify the base case and the recursive case.

Tracing

Follow the call stack step by step.

Comparison

Weigh recursion against loops.

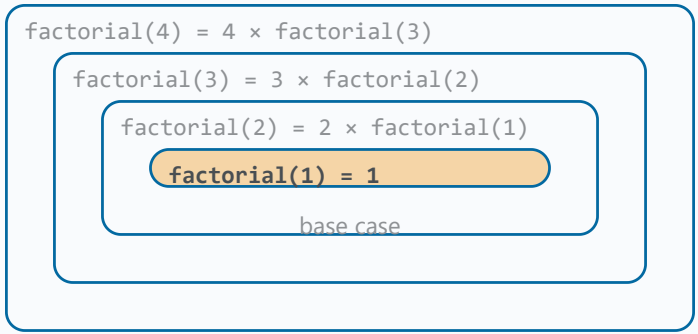
Improvement

Cut repeated computation with memoization.

BASICS

What Is Recursion?

- A method reduces a problem to a **smaller instance** of itself and calls itself.
- Each call gets its own **stack frame** with its own parameters and local variables.
- Each call must move toward the **base case**; otherwise the calls never end.



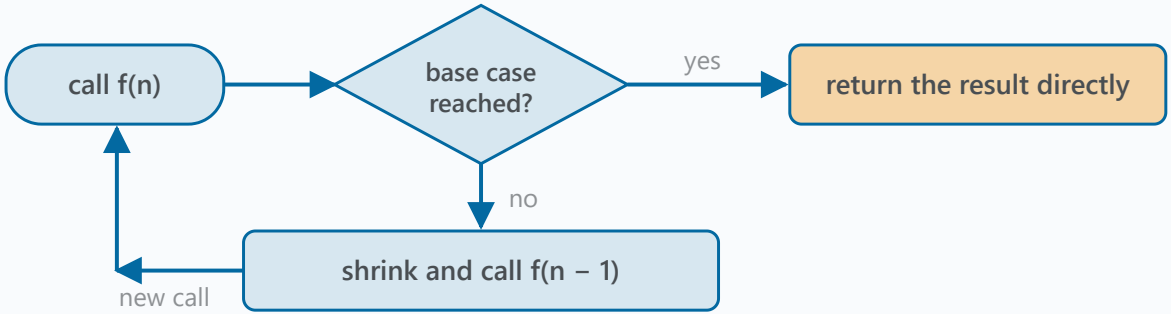
Two Required Parts

Base case

The result is known directly and no new call is made (e.g. $n \leq 1 \rightarrow 1$).

Recursive case

A call that turns the problem into a smaller instance (e.g. $n \times f(n - 1)$).



EXAMPLES

Recursive Factorial

```
static long factorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Definition

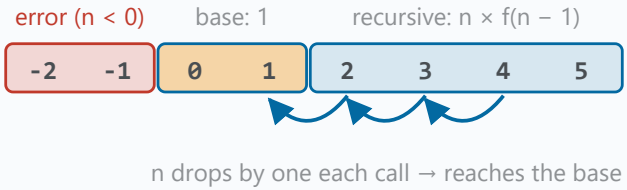
n! = n × (n − 1)!, 0! = 1! = 1

Base case

n ≤ 1 → 1; n drops by one on each call.

Negative n

Factorial is undefined → an exception is thrown.



n drops by one each call → reaches the base

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

main calls `factorial(4)` and waits for the result.

Step 1 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
5     return n * factorial(n - 1);
6 }
▶ System.out.println(factorial(4));
```

VARIABLES

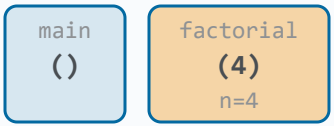
—

CALL STACK

`main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

New frame: **n = 4**. **4 < 0** → **false**, **4 <= 1** → **false**.

Step 2 / 12

```
1 static long factorial(int n) {
▶     if (n < 0)
3         throw new IllegalArgumentException();
▶     if (n <= 1) return 1;
5     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

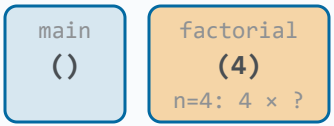
`n = 4`

CALL STACK

`factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

`4 * factorial(3)` : the product needs the subresult first → this frame **waits** and makes a new call.

Step 3 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
▶     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

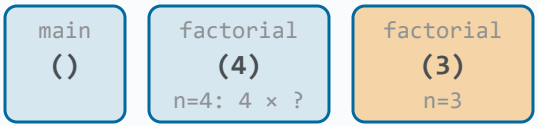
`n = 4`

CALL STACK

`factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

New frame: **n = 3**. **3 < 0** → **false**, **3 <= 1** → **false**.

Step 4 / 12

```
1 static long factorial(int n) {
▶     if (n < 0)
3         throw new IllegalArgumentException();
▶     if (n <= 1) return 1;
5     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

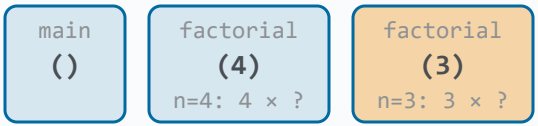
`n = 3`

CALL STACK

`factorial(3)` `factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

`3 * factorial(2)` : the product needs the subresult first → this frame **waits** and makes a new call.

Step 5 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
▶     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

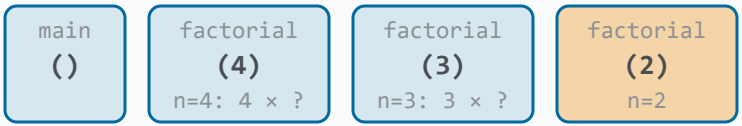
`n = 3`

CALL STACK

`factorial(3)` `factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

New frame: **n = 2**. **2 < 0** → **false**, **2 <= 1** → **false**.

Step 6 / 12

```
1 static long factorial(int n) {
▶     if (n < 0)
3         throw new IllegalArgumentException();
▶     if (n <= 1) return 1;
5     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

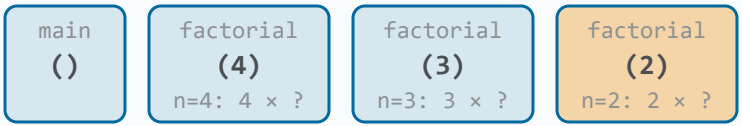
`n = 2`

CALL STACK

`factorial(2)` `factorial(3)` `factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

`2 * factorial(1)` : the product needs the subresult first → this frame **waits** and makes a new call.

Step 7 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
▶     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

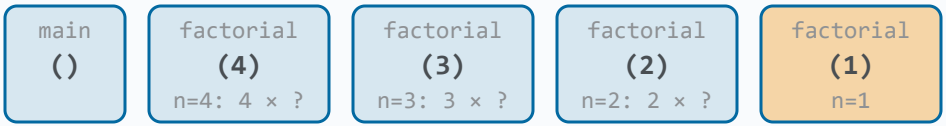
`n = 2`

CALL STACK

`factorial(2)` `factorial(3)` `factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

New frame: **n = 1**. **1 < 0** → **false**, **1 <= 1** → **true**: **base case**, return 1.

Step 8 / 12

```
1 static long factorial(int n) {
▶     if (n < 0)
3         throw new IllegalArgumentException();
▶     if (n <= 1) return 1;
5     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

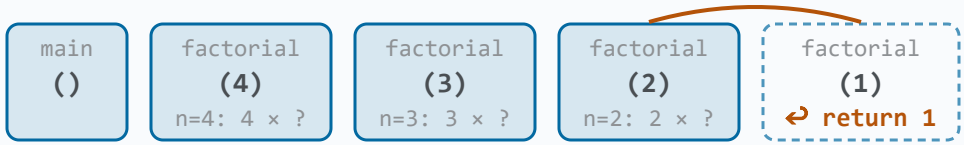
`n = 1`

CALL STACK

`factorial(1)` `factorial(2)` `factorial(3)` `factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

`factorial(1)` returned **1**; its frame is gone. Now **2 × 1 = 2**.

Step 9 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
▶     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

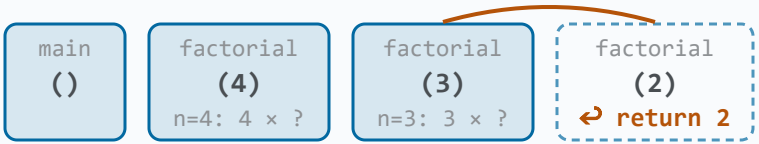
`n = 2`

CALL STACK

`factorial(2)` `factorial(3)` `factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

`factorial(2)` returned **2**; its frame is gone. Now **3 × 2 = 6**.

Step 10 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
▶     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

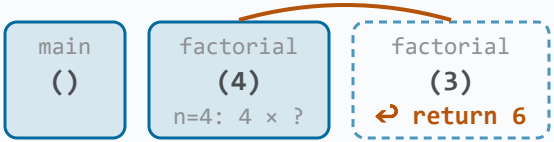
`n = 3`

CALL STACK

`factorial(3)` `factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

`factorial(3)` returned **6**; its frame is gone. Now **4 × 6 = 24**.

Step 11 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
▶     return n * factorial(n - 1);
6 }
7 System.out.println(factorial(4));
```

VARIABLES

`n = 4`

CALL STACK

`factorial(4)` `main()`

OUTPUT

Code Trace: `factorial(4)`



main (stack bottom) → stack grows this way → top

The top frame closes too; main prints **24**. At most 5 frames were open → O(n) extra memory.

Step 12 / 12

```
1 static long factorial(int n) {
2     if (n < 0)
3         throw new IllegalArgumentException();
4     if (n <= 1) return 1;
5     return n * factorial(n - 1);
6 }
▶ System.out.println(factorial(4));
```

VARIABLES

—

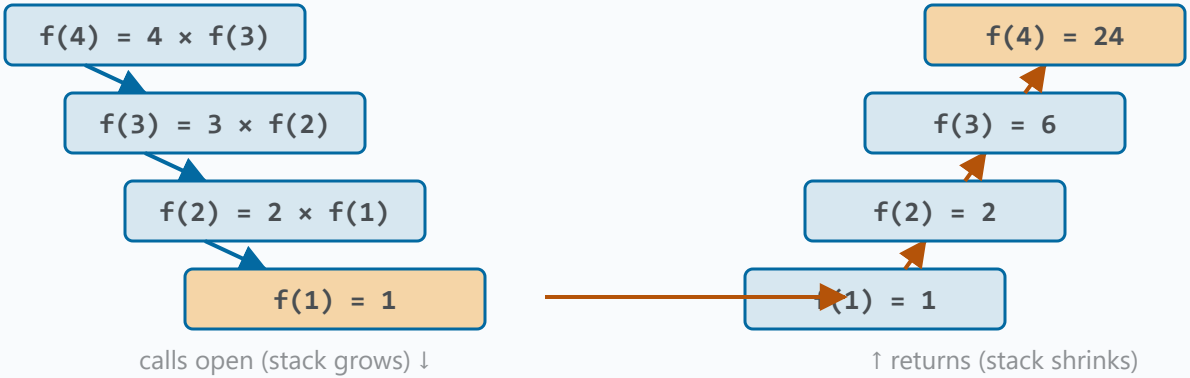
CALL STACK

`main()`

OUTPUT

```
24
```

factorial(4): Unwinding and Returning



- Calls pile up on the stack until the base case; no multiplication has happened yet.
- Once f(1) = 1 returns, the pending products finish **in reverse order**: 2, 6, 24.

EXAMPLES · STEP THROUGH THE CODE

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

Definition: sum(n) = n + sum(n − 1), sum(0) = 0. main calls `sum(3)`.

Step 1 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
3     return n + sum(n - 1);
4 }
▶ System.out.println(sum(3));
```

VARIABLES

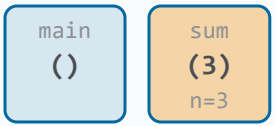
—

CALL STACK

`main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

3 <= 0 → false.

Step 2 / 12

```
1 static int sum(int n) {
▶     if (n <= 0) return 0;
3     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

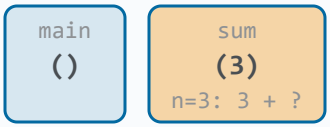
`n = 3`

CALL STACK

`sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

`3 + sum(2)` : the addition waits; new call.

Step 3 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
▶     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

`n = 3`

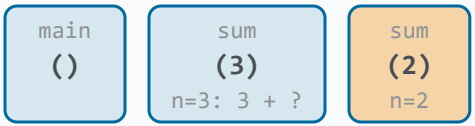
CALL STACK

`sum(3)` `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

2 <= 0 → false.

Step 4 / 12

```
1 static int sum(int n) {
▶     if (n <= 0) return 0;
3     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

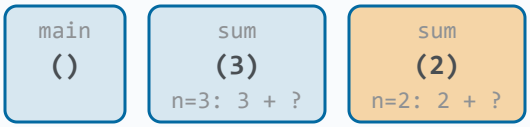
`n = 2`

CALL STACK

`sum(2)` `sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

`2 + sum(1)` : the addition waits; new call.

Step 5 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
▶     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

`n = 2`

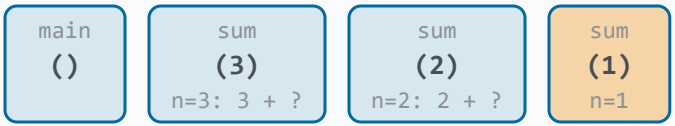
CALL STACK

`sum(2)` `sum(3)` `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

1 <= 0 → false.

Step 6 / 12

```
1 static int sum(int n) {
▶     if (n <= 0) return 0;
3     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

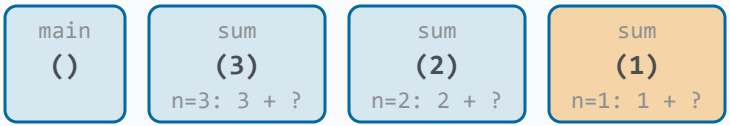
`n = 1`

CALL STACK

`sum(1)` `sum(2)` `sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

`1 + sum(0)` : the addition waits; new call.

Step 7 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
▶     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

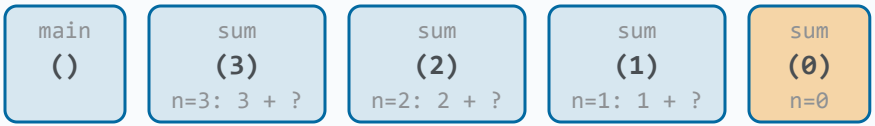
`n = 1`

CALL STACK

`sum(1)` `sum(2)` `sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

0 <= 0 → **true**: base case, **return 0**.

Step 8 / 12

```
1 static int sum(int n) {
▶     if (n <= 0) return 0;
3     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

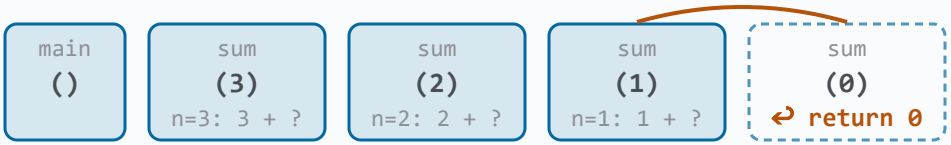
`n = 0`

CALL STACK

`sum(0)` `sum(1)` `sum(2)` `sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

`sum(0)` = 0 returned → **1 + 0 = 1**.

Step 9 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
▶     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

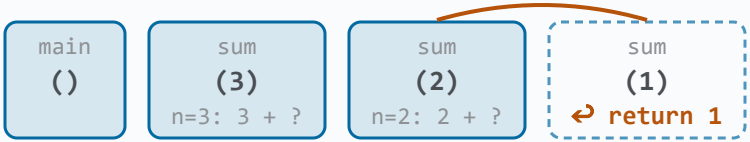
`n = 1`

CALL STACK

`sum(1)` `sum(2)` `sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

`sum(1)` = 1 returned → **2 + 1 = 3**.

Step 10 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
▶     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

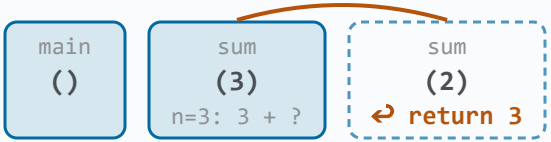
`n = 2`

CALL STACK

`sum(2)` `sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

`sum(2)` = 3 returned → **3 + 3 = 6**.

Step 11 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
▶     return n + sum(n - 1);
4 }
5 System.out.println(sum(3));
```

VARIABLES

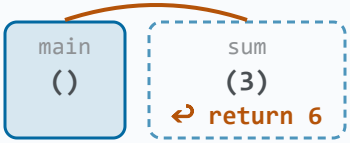
`n = 3`

CALL STACK

`sum(3)` `main()`

OUTPUT

Sum from 1 to n: `sum(3)`



main (stack bottom) → stack grows this way → top

6 = 3 + 2 + 1 + 0 is printed.

Step 12 / 12

```
1 static int sum(int n) {
2     if (n <= 0) return 0;
3     return n + sum(n - 1);
4 }
▶ System.out.println(sum(3));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
6
```

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

main calls `digitSum(472)`.

Step 1 / 12

```
1 static int digitSum(int n) {
2     long m = Math.abs((long) n);
3     return digitSumPos(m);
4 }
5 static int digitSumPos(long n) {
6     if (n < 10) return (int) n;
7     return (int) (n % 10)
8         + digitSumPos(n / 10);
9 }
▶ System.out.println(digitSum(472));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

Negatives become their absolute value: **m = 472** (converting to long avoids overflow at the smallest int).

Step 2 / 12

```
1  static int digitSum(int n) {
▶      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
6      if (n < 10) return (int) n;
7      return (int) (n % 10)
8          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 472` `m = 472`

CALL STACK

`digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

A helper method does the real work.

Step 3 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
▶      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
6      if (n < 10) return (int) n;
7      return (int) (n % 10)
8          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 472` `m = 472`

CALL STACK

`digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

472 < 10 → false: last digit 2, rest 47.

Step 4 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
▶      if (n < 10) return (int) n;
7      return (int) (n % 10)
8          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 472`

CALL STACK

`digitSumPos(472)` `digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

`472 % 10 = 2` is kept; a new call with `472 / 10 = 47`.

Step 5 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
6      if (n < 10) return (int) n;
▶      return (int) (n % 10)
▶          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

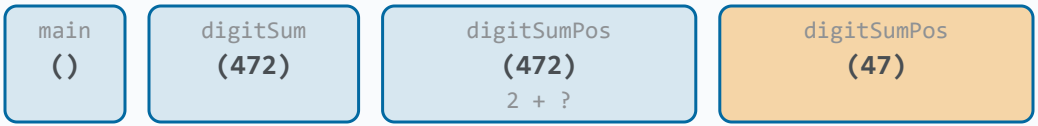
`n = 472`

CALL STACK

`digitSumPos(472)` `digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

47 < 10 → **false**: last digit 7, rest 4.

Step 6 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
▶      if (n < 10) return (int) n;
7      return (int) (n % 10)
8          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 47`

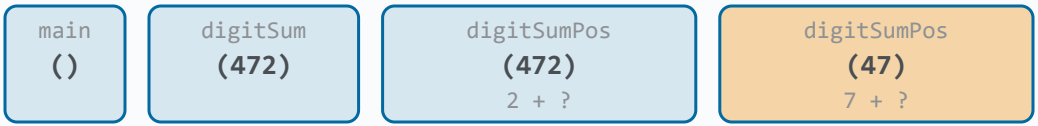
CALL STACK

`digitSumPos(47)`

`digitSumPos(472)` `digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

`47 % 10 = 7` is kept; a new call with `47 / 10 = 4` .

Step 7 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
6      if (n < 10) return (int) n;
▶      return (int) (n % 10)
▶          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 47`

CALL STACK

`digitSumPos(47)`

`digitSumPos(472)` `digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

4 < 10 → **true**: a single digit, **return 4**.

Step 8 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
▶      if (n < 10) return (int) n;
7      return (int) (n % 10)
8          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 4`

CALL STACK

`digitSumPos(4)` `digitSumPos(47)` `digitSumPos(472)` `digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

The subcall returned 4 → **7 + 4 = 11**.

Step 9 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
6      if (n < 10) return (int) n;
▶      return (int) (n % 10)
▶          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 47`

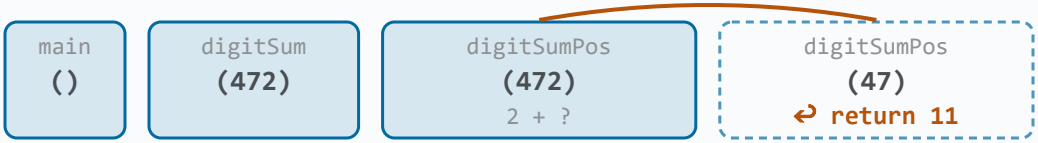
CALL STACK

`digitSumPos(47)`

`digitSumPos(472)` `digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

The subcall returned 11 → **2 + 11 = 13**.

Step 10 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
3      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
6      if (n < 10) return (int) n;
▶      return (int) (n % 10)
▶          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 472`

CALL STACK

`digitSumPos(472)` `digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

The helper returned **13**; it is passed straight back.

Step 11 / 12

```
1  static int digitSum(int n) {
2      long m = Math.abs((long) n);
▶      return digitSumPos(m);
4  }
5  static int digitSumPos(long n) {
6      if (n < 10) return (int) n;
7      return (int) (n % 10)
8          + digitSumPos(n / 10);
9  }
10 System.out.println(digitSum(472));
```

VARIABLES

`n = 472` `m = 472`

CALL STACK

`digitSum(472)` `main()`

OUTPUT

Sum of Digits: `digitSum(472)`



main (stack bottom) → stack grows this way → top

4 + 7 + 2 = **13** is printed.

Step 12 / 12

```
1 static int digitSum(int n) {
2     long m = Math.abs((long) n);
3     return digitSumPos(m);
4 }
5 static int digitSumPos(long n) {
6     if (n < 10) return (int) n;
7     return (int) (n % 10)
8         + digitSumPos(n / 10);
9 }
▶ System.out.println(digitSum(472));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
13
```

Palindrome: `isPal("kayak", 0, 4)`



main calls `isPal("kayak", 0, 4)`.

Step 1 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
6     return isPal(s, left + 1, right - 1);
7 }
▶ System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

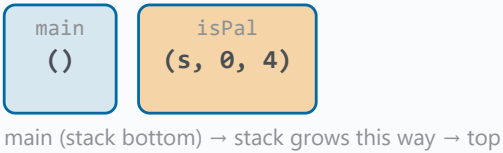
—

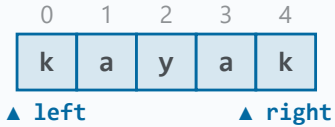
CALL STACK

`main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`





0 >= 4 → false.

Step 2 / 11

```
1 static boolean isPal(String s,
2                      int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
6     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

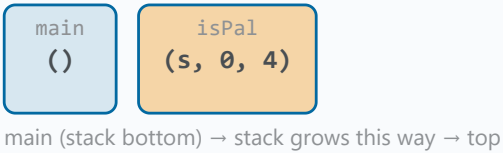
`s = "kayak"` `left = 0` `right = 4`

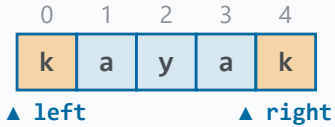
CALL STACK

`isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`





`'k' != 'k'` → **false**: the ends match; continue inward.

Step 3 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
▶     if (s.charAt(left) != s.charAt(right))
5         return false;
6     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

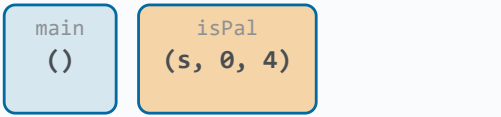
`s = "kayak"` `left = 0` `right = 4`

CALL STACK

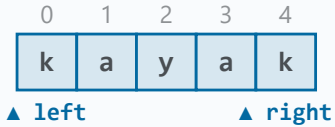
`isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`



main (stack bottom) → stack grows this way → top



Drop both ends: call `isPal(s, 1, 3)`.

Step 4 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
▶     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

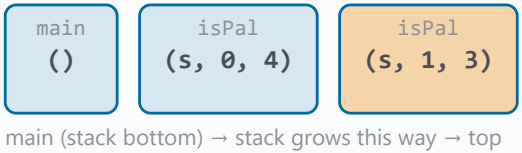
`s = "kayak"` `left = 0` `right = 4`

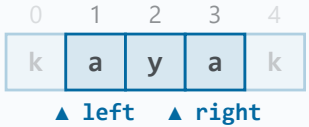
CALL STACK

`isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`





1 >= 3 → false.

Step 5 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
6     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

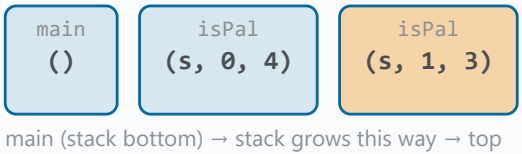
`s = "kayak"` `left = 1` `right = 3`

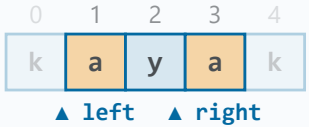
CALL STACK

`isPal(s, 1, 3)` `isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`





`'a' != 'a'` → **false**: the ends match; continue inward.

Step 6 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
▶     if (s.charAt(left) != s.charAt(right))
5         return false;
6     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

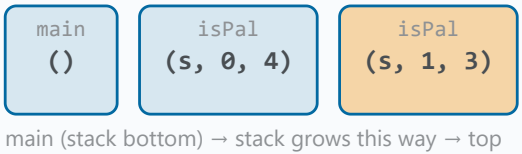
`s = "kayak"` `left = 1` `right = 3`

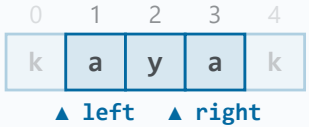
CALL STACK

`isPal(s, 1, 3)` `isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`





Drop both ends: call `isPal(s, 2, 2)`.

Step 7 / 11

```
1 static boolean isPal(String s,
2                      int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
▶     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

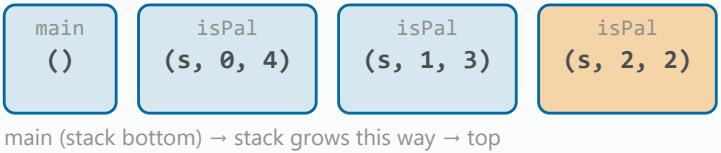
`s = "kayak"` `left = 1` `right = 3`

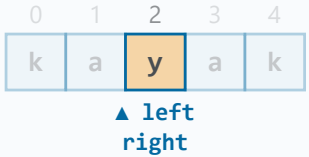
CALL STACK

`isPal(s, 1, 3)` `isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`





2 >= 2 → **true**: one character left in the middle → **return true**.

Step 8 / 11

```
1 static boolean isPal(String s,
2                      int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
6     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

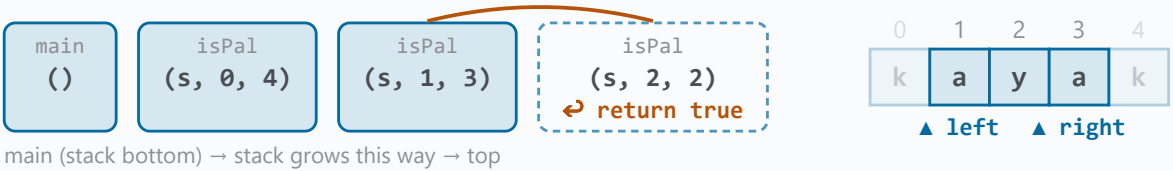
`s = "kayak"` `left = 2` `right = 2`

CALL STACK

`isPal(s, 2, 2)` `isPal(s, 1, 3)` `isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`



The subcall returned **true**; this frame returns the same value.

Step 9 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
▶     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

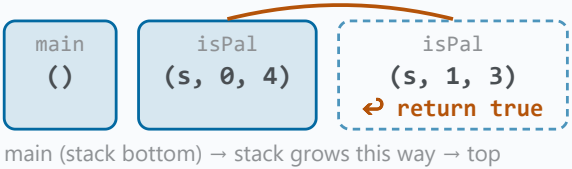
`s = "kayak"` `left = 1` `right = 3`

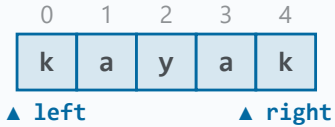
CALL STACK

`isPal(s, 1, 3)` `isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`





The subcall returned **true**; this frame returns the same value.

Step 10 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
▶     return isPal(s, left + 1, right - 1);
7 }
8 System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

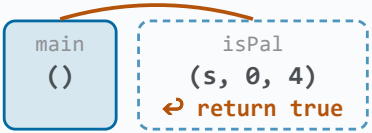
`s = "kayak"` `left = 0` `right = 4`

CALL STACK

`isPal(s, 0, 4)` `main()`

OUTPUT

Palindrome: `isPal("kayak", 0, 4)`



main (stack bottom) → stack grows this way → top

0	1	2	3	4
k	a	y	a	k

The result reaches main: **true**. "kayak" reads the same backward.

Step 11 / 11

```
1 static boolean isPal(String s,
2                       int left, int right) {
3     if (left >= right) return true;
4     if (s.charAt(left) != s.charAt(right))
5         return false;
6     return isPal(s, left + 1, right - 1);
7 }
▶ System.out.println(isPal("kayak", 0, 4));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
true
```

EXAMPLES · STEP THROUGH THE CODE

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

main calls `gcd(48, 18)`.

Step 1 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
3     return gcd(b, a % b);
4 }
▶ System.out.println(gcd(48, 18));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

18 == 0 → false.

Step 2 / 12

```
1 static int gcd(int a, int b) {
▶     if (b == 0) return Math.abs(a);
3     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

a = 48 b = 18

CALL STACK

gcd(48, 18) main()

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

`48 % 18 = 12` → call `gcd(18, 12)` (the numbers shrink).

Step 3 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
▶     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

`a = 48` `b = 18`

CALL STACK

`gcd(48, 18)` `main()`

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

12 == 0 → false.

Step 4 / 12

```
1 static int gcd(int a, int b) {
▶     if (b == 0) return Math.abs(a);
3     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

`a = 18` `b = 12`

CALL STACK

`gcd(18, 12)` `gcd(48, 18)` `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

`18 % 12 = 6` → call `gcd(12, 6)` (the numbers shrink).

Step 5 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
      return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

`a = 18` `b = 12`

CALL STACK

`gcd(18, 12)` `gcd(48, 18)` `main()`

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

6 == 0 → false.

Step 6 / 12

```
1 static int gcd(int a, int b) {
▶     if (b == 0) return Math.abs(a);
3     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

`a = 12` `b = 6`

CALL STACK

`gcd(12, 6)` `gcd(18, 12)` `gcd(48, 18)` `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

`12 % 6 = 0` → call `gcd(6, 0)` (the numbers shrink).

Step 7 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
▶     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

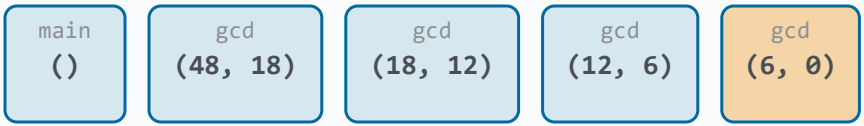
`a = 12` `b = 6`

CALL STACK

`gcd(12, 6)` `gcd(18, 12)` `gcd(48, 18)` `main()`

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

0 == 0 → **true**: base case, **return 6**.

Step 8 / 12

```
1 static int gcd(int a, int b) {
▶     if (b == 0) return Math.abs(a);
3     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

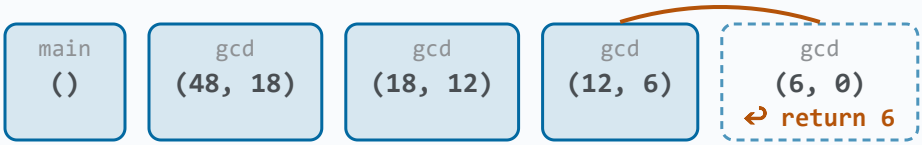
`a = 6` `b = 0`

CALL STACK

`gcd(6, 0)` `gcd(12, 6)` `gcd(18, 12)` `gcd(48, 18)` `main()`

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

The subcall returned **6**; it is passed straight back.

Step 9 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
▶     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

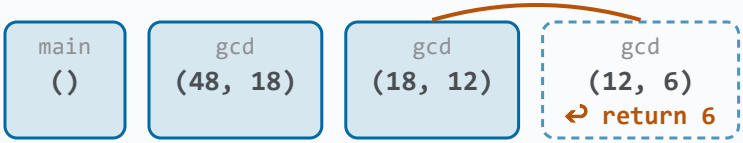
a = 12 b = 6

CALL STACK

gcd(12, 6) gcd(18, 12) gcd(48, 18) main()

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

The subcall returned **6**; it is passed straight back.

Step 10 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
▶     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

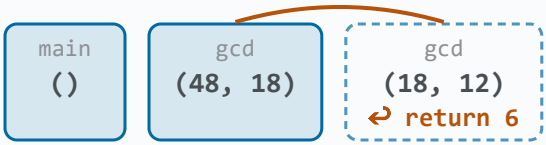
`a = 18` `b = 12`

CALL STACK

`gcd(18, 12)` `gcd(48, 18)` `main()`

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

The subcall returned **6**; it is passed straight back.

Step 11 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
▶     return gcd(b, a % b);
4 }
5 System.out.println(gcd(48, 18));
```

VARIABLES

a = 48 b = 18

CALL STACK

gcd(48, 18) main()

OUTPUT

GCD by Euclid: `gcd(48, 18)`



main (stack bottom) → stack grows this way → top

GCD(48, 18) = **6**: the remainders shrank to 0; the last nonzero remainder is the GCD.

Step 12 / 12

```
1 static int gcd(int a, int b) {
2     if (b == 0) return Math.abs(a);
3     return gcd(b, a % b);
4 }
▶ System.out.println(gcd(48, 18));
```

VARIABLES

—

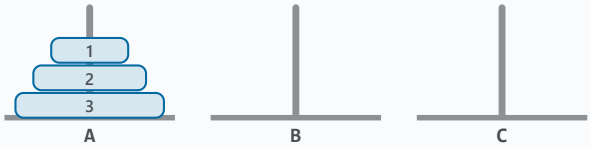
CALL STACK

`main()`

OUTPUT

```
6
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 0

Move 3 disks from A to C, with B as the helper. Rule: a larger disk never goes on a smaller one.

Step 1 / 12

```
1  static void hanoi(int n, char from,
2                    char to, char via) {
3    if (n == 0) return;
4    hanoi(n - 1, from, via, to);
5    System.out.print(from + "->" + to + " ");
6    hanoi(n - 1, via, to, from);
7  }
▶ hanoi(3, 'A', 'C', 'B');
```

VARIABLES

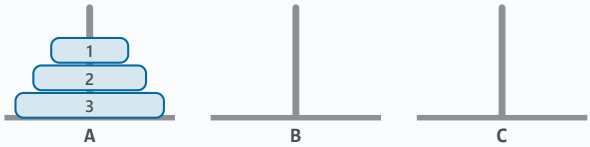
—

CALL STACK

`main()`

OUTPUT

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 0

`hanoi(3, A, C, B)` : first the top 2 disks must move A → B (to the helper).

Step 2 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
▶   hanoi(n - 1, from, via, to);
5   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

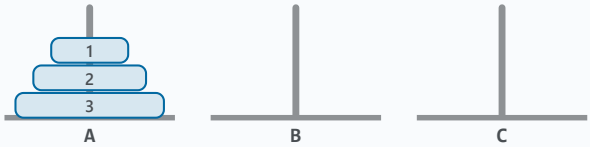
`n = 3` `from = A` `to = C` `via = B`

CALL STACK

`hanoi(3,A,C,B)` `main()`

OUTPUT

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 0

`hanoi(2, A, B, C)` : first the top disk must move A → C (to the helper).

Step 3 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
▶   hanoi(n - 1, from, via, to);
5   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

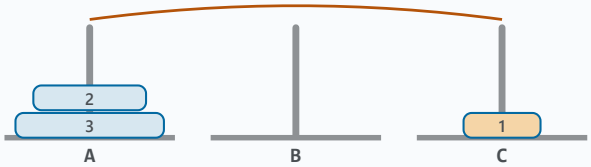
`n = 2` `from = A` `to = B` `via = C`

CALL STACK

`hanoi(2,A,B,C)` `hanoi(3,A,C,B)` `main()`

OUTPUT

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 1

Move 1: disk **1** A → C. (the hanoi(0) calls return at once)

Step 4 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
▶   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 1` `from = A` `to = C` `via = B`

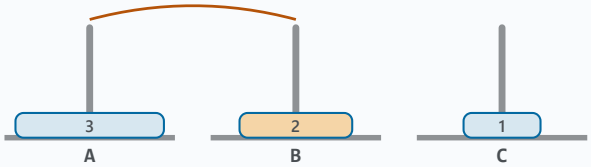
CALL STACK

`hanoi(1,A,C,B)` `hanoi(2,A,B,C)` `hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 2

Move 2: disk **2** A → B.

Step 5 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
▶   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 2` `from = A` `to = B` `via = C`

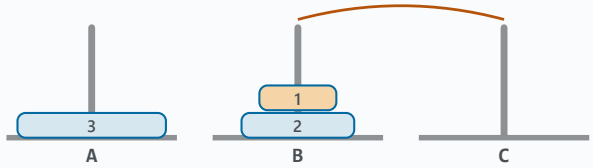
CALL STACK

`hanoi(2,A,B,C)` `hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C A->B
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 3

Move 3: disk **1** C → B.

Step 6 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
▶   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 1` `from = C` `to = B` `via = A`

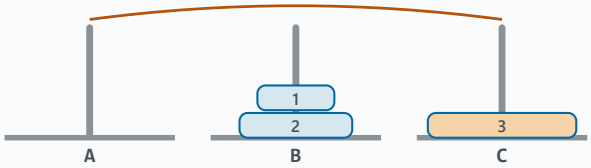
CALL STACK

`hanoi(1,C,B,A)` `hanoi(2,A,B,C)` `hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C A->B C->B
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 4

Move 4: disk **3** A → C.

Step 7 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
▶   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 3` `from = A` `to = C` `via = B`

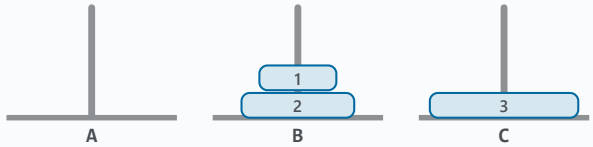
CALL STACK

`hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C A->B C->B A->C
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 4

`hanoi(2, B, C, A)` : first the top disk must move B → A (to the helper).

Step 8 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
▶   hanoi(n - 1, from, via, to);
5   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 2` `from = B` `to = C` `via = A`

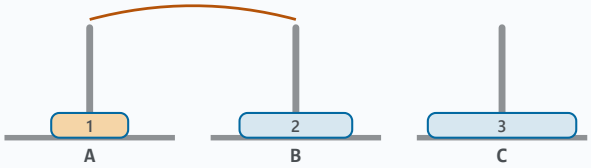
CALL STACK

`hanoi(2,B,C,A)` `hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C A->B C->B A->C
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 5

Move 5: disk **1** B → A.

Step 9 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
▶   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 1` `from = B` `to = A` `via = C`

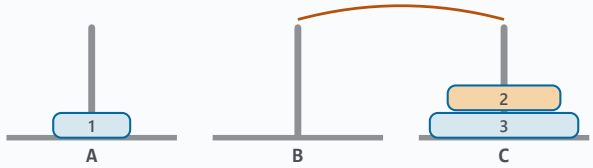
CALL STACK

`hanoi(1,B,A,C)` `hanoi(2,B,C,A)` `hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C A->B C->B A->C B->A
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 6

Move 6: disk **2** B → C.

Step 10 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
▶   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 2` `from = B` `to = C` `via = A`

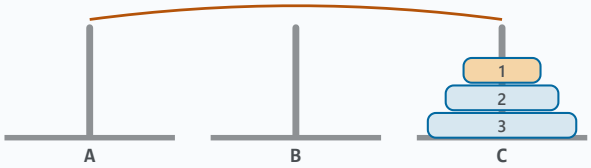
CALL STACK

`hanoi(2,B,C,A)` `hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C A->B C->B A->C B->A B->C
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 7

Move 7: disk **1** A → C.

Step 11 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
▶   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
8 hanoi(3, 'A', 'C', 'B');
```

VARIABLES

`n = 1` `from = A` `to = C` `via = B`

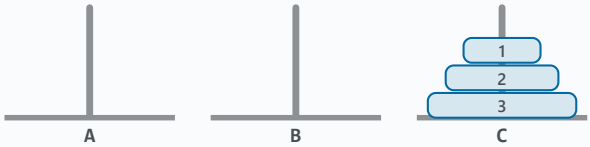
CALL STACK

`hanoi(1,A,C,B)` `hanoi(2,B,C,A)` `hanoi(3,A,C,B)` `main()`

OUTPUT

```
A->C A->B C->B A->C B->A B->C A->C
```

Towers of Hanoi: `hanoi(3, 'A', 'C', 'B')`



moves: 7

Done: **7 = 2^3 − 1** moves. n disks need 2^n − 1 moves (exponential).

Step 12 / 12

```
1 static void hanoi(int n, char from,
2                   char to, char via) {
3   if (n == 0) return;
4   hanoi(n - 1, from, via, to);
5   System.out.print(from + "->" + to + " ");
6   hanoi(n - 1, via, to, from);
7 }
▶ hanoi(3, 'A', 'C', 'B');
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
A->C A->B C->B A->C B->A B->C A->C
```

EXAMPLES · STEP THROUGH THE CODE

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

$2^4 = 2 \times 2^3$, so: power(base, n) = base × power(base, n − 1), power(base, 0) = 1.

Step 1 / 11

```
1 static long power(long base, int n) {
2   if (n == 0) return 1;
3   return base * power(base, n - 1);
4 }
▶ System.out.println(power(2, 4));
```

VARIABLES

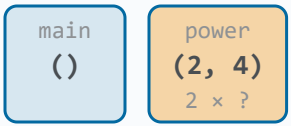
—

CALL STACK

`main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

4 == 0 → **false**: new call for 2 × power(2, 3).

Step 2 / 11

```
1 static long power(long base, int n) {
    if (n == 0) return 1;
    return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

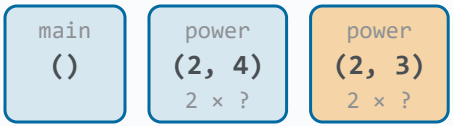
`base = 2` `n = 4`

CALL STACK

`power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

3 == 0 → **false**: new call for 2 × power(2, 2).

Step 3 / 11

```
1 static long power(long base, int n) {
▶   if (n == 0) return 1;
▶   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

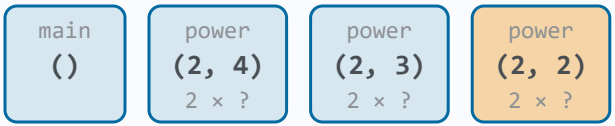
`base = 2` `n = 3`

CALL STACK

`power(2, 3)` `power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

2 == 0 → **false**: new call for 2 × power(2, 1).

Step 4 / 11

```
1 static long power(long base, int n) {
▶   if (n == 0) return 1;
▶   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

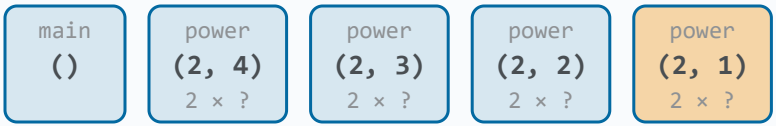
`base = 2` `n = 2`

CALL STACK

`power(2, 2)` `power(2, 3)` `power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

1 == 0 → **false**: new call for 2 × power(2, 0).

Step 5 / 11

```
1 static long power(long base, int n) {
▶   if (n == 0) return 1;
▶   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

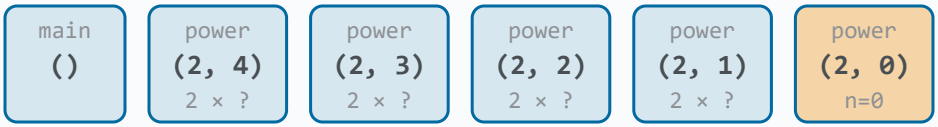
`base = 2` `n = 1`

CALL STACK

`power(2, 1)` `power(2, 2)` `power(2, 3)` `power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

0 == 0 → **true**: base case, return 1.

Step 6 / 11

```
1 static long power(long base, int n) {
▶   if (n == 0) return 1;
3   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

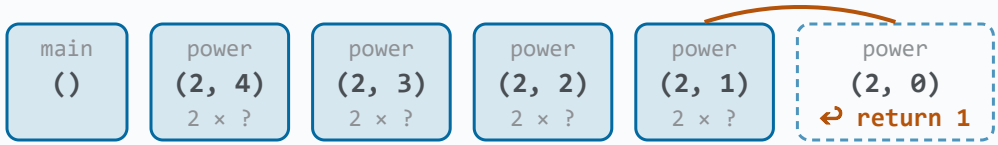
`base = 2` `n = 0`

CALL STACK

`power(2, 0)` `power(2, 1)` `power(2, 2)` `power(2, 3)` `power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

power(2, 0) = 1 returned → **2 × 1 = 2**.

Step 7 / 11

```
1 static long power(long base, int n) {
2   if (n == 0) return 1;
▶   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

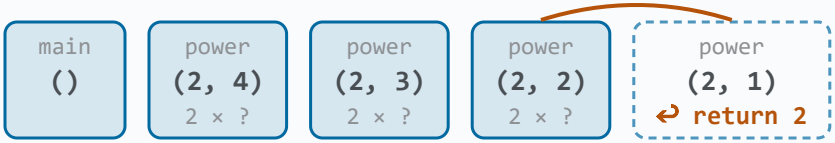
`base = 2` `n = 1`

CALL STACK

`power(2, 1)` `power(2, 2)` `power(2, 3)` `power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

power(2, 1) = 2 returned → **2 × 2 = 4**.

Step 8 / 11

```
1 static long power(long base, int n) {
2   if (n == 0) return 1;
▶   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

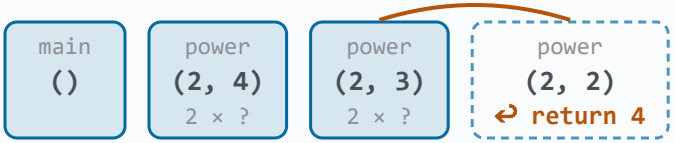
`base = 2` `n = 2`

CALL STACK

`power(2, 2)` `power(2, 3)` `power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

power(2, 2) = 4 returned → **2 × 4 = 8**.

Step 9 / 11

```
1 static long power(long base, int n) {
2   if (n == 0) return 1;
▶   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

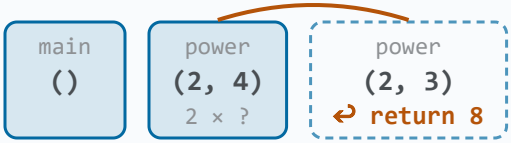
`base = 2` `n = 3`

CALL STACK

`power(2, 3)` `power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

power(2, 3) = 8 returned → **2 × 8 = 16**.

Step 10 / 11

```
1 static long power(long base, int n) {
2   if (n == 0) return 1;
▶   return base * power(base, n - 1);
4 }
5 System.out.println(power(2, 4));
```

VARIABLES

`base = 2` `n = 4`

CALL STACK

`power(2, 4)` `main()`

OUTPUT

Recursive Power: `power(2, 4)`



main (stack bottom) → stack grows this way → top

16: n + 1 = 5 calls → O(n).

Step 11 / 11

```
1 static long power(long base, int n) {
2   if (n == 0) return 1;
3   return base * power(base, n - 1);
4 }
▶ System.out.println(power(2, 4));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
16
```

EXAMPLES · STEP THROUGH THE CODE

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

Idea: $2^{10} = (2^5)^2$, $2^5 = (2^2)^2 \times 2$. The exponent is **halved** on each call.

Step 1 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
4   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
▶ System.out.println(fastPow(2, 10));
```

VARIABLES

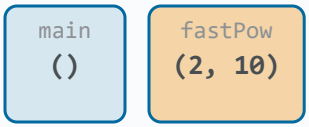
—

CALL STACK

`main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

exp = 10 ≠ 0 → first `fastPow(2, 5)` (10 / 2 = 5).

Step 2 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
▶   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

`base = 2` `exp = 10`

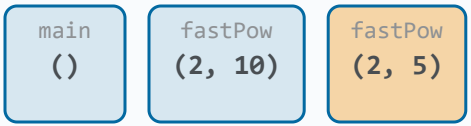
CALL STACK

`fastPow(2, 10)` `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

exp = 5 ≠ 0 → first `fastPow(2, 2)` (5 / 2 = 2).

Step 3 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
▶   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

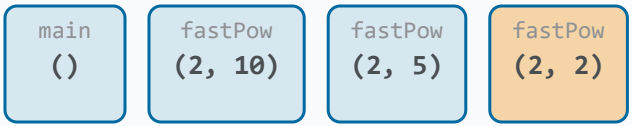
`base = 2` `exp = 5`

CALL STACK

`fastPow(2, 5)` `fastPow(2, 10)` `main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

exp = 2 ≠ 0 → first `fastPow(2, 1)` (2 / 2 = 1).

Step 4 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
▶   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

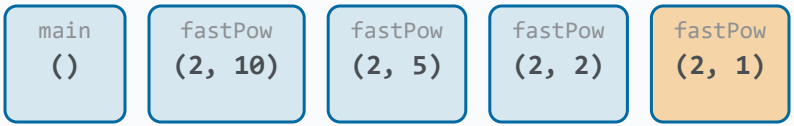
`base = 2` `exp = 2`

CALL STACK

`fastPow(2, 2)` `fastPow(2, 5)` `fastPow(2, 10)` `main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

exp = 1 ≠ 0 → first `fastPow(2, 0)` (1 / 2 = 0).

Step 5 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
▶   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

`base = 2` `exp = 1`

CALL STACK

`fastPow(2, 1)` `fastPow(2, 2)` `fastPow(2, 5)` `fastPow(2, 10)` `main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

exp == 0: return 1.

Step 6 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
▶   if (exp == 0) return 1;
4   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

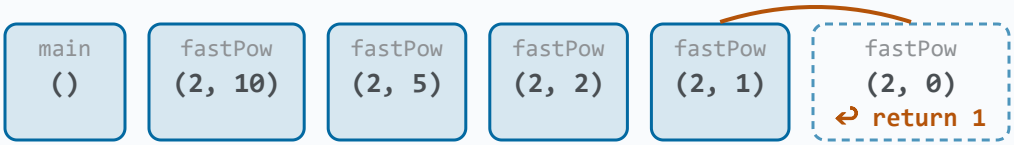
`base = 2` `exp = 0`

CALL STACK

`fastPow(2, 0)` `fastPow(2, 1)` `fastPow(2, 2)` `fastPow(2, 5)` `fastPow(2, 10)` `main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

half = 1; 1 is odd → **1 × 1 × 2 = 2**.

Step 7 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
4   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
▶   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

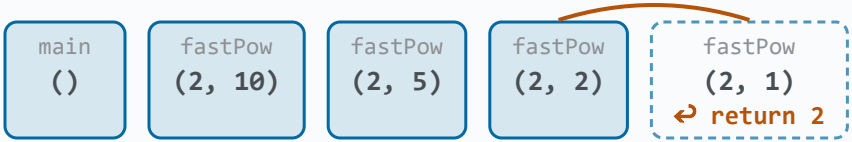
`base = 2` `exp = 1` `half = 1`

CALL STACK

`fastPow(2, 1)` `fastPow(2, 2)` `fastPow(2, 5)` `fastPow(2, 10)` `main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

half = 2; 2 is even → **2 × 2 = 4**.

Step 8 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
4   long half = fastPow(base, exp / 2);
▶   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

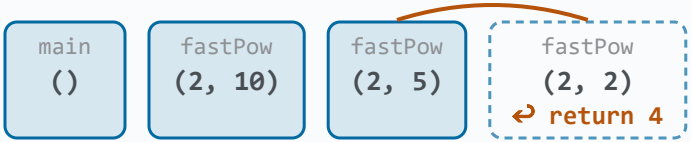
`base = 2` `exp = 2` `half = 2`

CALL STACK

`fastPow(2, 2)` `fastPow(2, 5)` `fastPow(2, 10)` `main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

half = 4; 5 is odd → **4 × 4 × 2 = 32**.

Step 9 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
4   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
▶   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

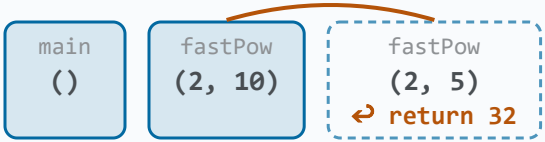
`base = 2` `exp = 5` `half = 4`

CALL STACK

`fastPow(2, 5)` `fastPow(2, 10)` `main()`

OUTPUT

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

half = 32; 10 is even → **32 × 32 = 1024**.

Step 10 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
4   long half = fastPow(base, exp / 2);
▶   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
8 System.out.println(fastPow(2, 10));
```

VARIABLES

`base = 2` `exp = 10` `half = 32`

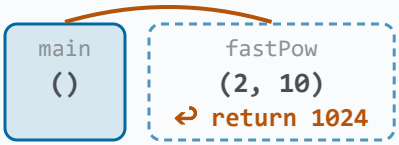
CALL STACK

`fastPow(2, 10)` `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Fast Power: `fastPow(2, 10)`



main (stack bottom) → stack grows this way → top

1024 in only 5 calls (the plain method needs 11) → **O(log n)**.

Step 11 / 11

```
1 static long fastPow(long base, int exp) {
2   if (exp < 0) throw new
  IllegalArgumentException("exp < 0");
3   if (exp == 0) return 1;
4   long half = fastPow(base, exp / 2);
5   if (exp % 2 == 0) return half * half;
6   return half * half * base;
7 }
▶ System.out.println(fastPow(2, 10));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
1024
```

EXAMPLES · STEP THROUGH THE CODE

Recursive Binary Search: `search(a, 38, 0, 7)`



Recursion instead of a loop: each call recurses on half the range.

Step 1 / 8

```
1 static int search(int[] a, int x,
2                   int lo, int hi) {
3   if (lo > hi) return -1;
4   int mid = lo + (hi - lo) / 2;
5   if (a[mid] == x) return mid;
6   if (a[mid] < x) return search(a, x, mid + 1, hi);
7   return search(a, x, lo, mid - 1);
8 }
9 // a = {2, 5, 8, 12, 16, 23, 38, 56}
▶ System.out.println(search(a, 38, 0, 7));
```

VARIABLES

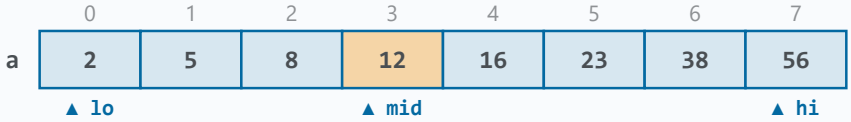
x = 38

CALL STACK

main()

OUTPUT

Recursive Binary Search: `search(a, 38, 0, 7)`



0 <= 7; **mid = 3**, a[mid] = 12.

Step 2 / 8

```
1  static int search(int[] a, int x,
2                    int lo, int hi) {
▶    if (lo > hi) return -1;
▶    int mid = lo + (hi - lo) / 2;
5    if (a[mid] == x) return mid;
6    if (a[mid] < x) return search(a, x, mid + 1, hi);
7    return search(a, x, lo, mid - 1);
8  }
9  // a = {2, 5, 8, 12, 16, 23, 38, 56}
10 System.out.println(search(a, 38, 0, 7));
```

VARIABLES

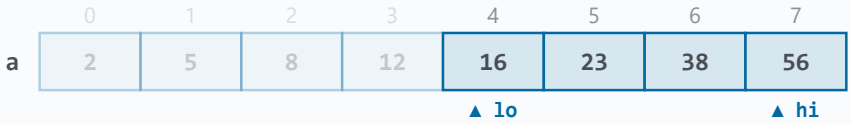
`x = 38` `lo = 0` `hi = 7` `mid = 3`

CALL STACK

`search(a, 38, 0, 7)` `main()`

OUTPUT

Recursive Binary Search: `search(a, 38, 0, 7)`



12 < 38 → right half: `search(a, 38, 4, 7)`.

Step 3 / 8

```
1  static int search(int[] a, int x,
2                    int lo, int hi) {
3    if (lo > hi) return -1;
4    int mid = lo + (hi - lo) / 2;
5    if (a[mid] == x) return mid;
▶    if (a[mid] < x) return search(a, x, mid + 1, hi);
7    return search(a, x, lo, mid - 1);
8  }
9  // a = {2, 5, 8, 12, 16, 23, 38, 56}
10 System.out.println(search(a, 38, 0, 7));
```

VARIABLES

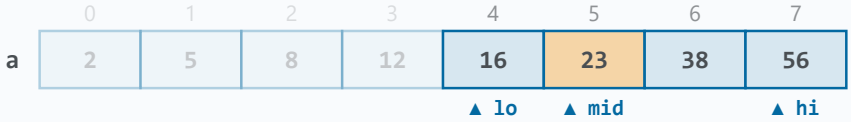
`x = 38` `lo = 0` `hi = 7` `mid = 3`

CALL STACK

`search(a, 38, 0, 7)` `main()`

OUTPUT

Recursive Binary Search: `search(a, 38, 0, 7)`



4 <= 7; **mid = 5**, a[mid] = 23.

Step 4 / 8

```
1  static int search(int[] a, int x,
2                    int lo, int hi) {
▶    if (lo > hi) return -1;
▶    int mid = lo + (hi - lo) / 2;
5    if (a[mid] == x) return mid;
6    if (a[mid] < x) return search(a, x, mid + 1, hi);
7    return search(a, x, lo, mid - 1);
8  }
9  // a = {2, 5, 8, 12, 16, 23, 38, 56}
10 System.out.println(search(a, 38, 0, 7));
```

VARIABLES

`x = 38` `lo = 4` `hi = 7` `mid = 5`

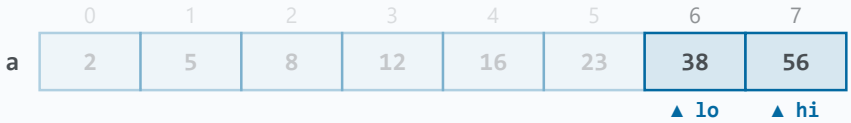
CALL STACK

`search(a, 38, 4, 7)`

`search(a, 38, 0, 7)` `main()`

OUTPUT

Recursive Binary Search: `search(a, 38, 0, 7)`



23 < 38 → right half: `search(a, 38, 6, 7)`.

Step 5 / 8

```
1  static int search(int[] a, int x,
2                    int lo, int hi) {
3    if (lo > hi) return -1;
4    int mid = lo + (hi - lo) / 2;
5    if (a[mid] == x) return mid;
▶    if (a[mid] < x) return search(a, x, mid + 1, hi);
7    return search(a, x, lo, mid - 1);
8  }
9  // a = {2, 5, 8, 12, 16, 23, 38, 56}
10 System.out.println(search(a, 38, 0, 7));
```

VARIABLES

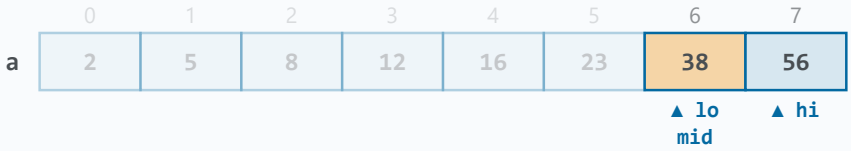
`x = 38` `lo = 4` `hi = 7` `mid = 5`

CALL STACK

`search(a, 38, 4, 7)`
`search(a, 38, 0, 7)` `main()`

OUTPUT

Recursive Binary Search: `search(a, 38, 0, 7)`



6 <= 7; **mid = 6**, a[mid] = 38.

Step 6 / 8

```
1  static int search(int[] a, int x,
2                    int lo, int hi) {
▶    if (lo > hi) return -1;
▶    int mid = lo + (hi - lo) / 2;
5    if (a[mid] == x) return mid;
6    if (a[mid] < x) return search(a, x, mid + 1, hi);
7    return search(a, x, lo, mid - 1);
8  }
9  // a = {2, 5, 8, 12, 16, 23, 38, 56}
10 System.out.println(search(a, 38, 0, 7));
```

VARIABLES

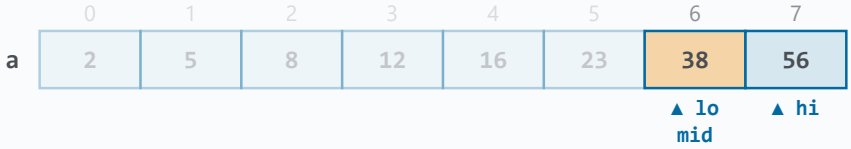
`x = 38` `lo = 6` `hi = 7` `mid = 6`

CALL STACK

`search(a, 38, 6, 7)`
`search(a, 38, 4, 7)`
`search(a, 38, 0, 7)` `main()`

OUTPUT

Recursive Binary Search: `search(a, 38, 0, 7)`



38 == 38 → return 6 (base case).

Step 7 / 8

```
1  static int search(int[] a, int x,
2                    int lo, int hi) {
3    if (lo > hi) return -1;
4    int mid = lo + (hi - lo) / 2;
▶    if (a[mid] == x) return mid;
6    if (a[mid] < x) return search(a, x, mid + 1, hi);
7    return search(a, x, lo, mid - 1);
8  }
9  // a = {2, 5, 8, 12, 16, 23, 38, 56}
10 System.out.println(search(a, 38, 0, 7));
```

VARIABLES

`x = 38` `lo = 6` `hi = 7` `mid = 6`

CALL STACK

`search(a, 38, 6, 7)`
`search(a, 38, 4, 7)`
`search(a, 38, 0, 7)` `main()`

OUTPUT

EXAMPLES · STEP THROUGH THE CODE

Recursive Binary Search: `search(a, 38, 0, 7)`



The result passes back unchanged through the frames: **6**.

Step 8 / 8

```
1 static int search(int[] a, int x,
2                   int lo, int hi) {
3   if (lo > hi) return -1;
4   int mid = lo + (hi - lo) / 2;
5   if (a[mid] == x) return mid;
6   if (a[mid] < x) return search(a, x, mid + 1, hi);
7   return search(a, x, lo, mid - 1);
8 }
9 // a = {2, 5, 8, 12, 16, 23, 38, 56}
▶ System.out.println(search(a, 38, 0, 7));
```

VARIABLES

x = 38

CALL STACK

main()

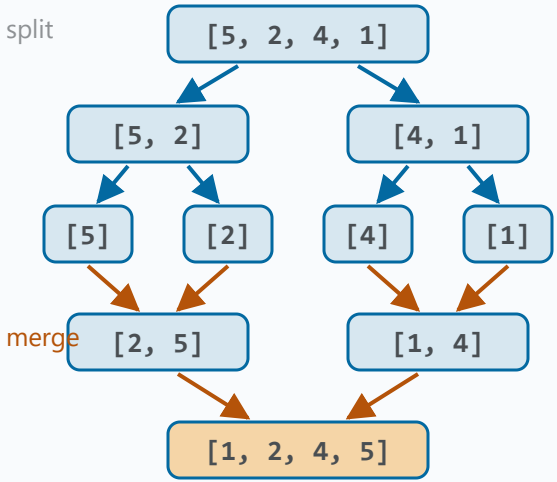
OUTPUT

```
6
```

Divide and Conquer: Merge Sort

```java
static void mergeSort(int[] a, int lo, int hi) {
    if (lo >= hi) return;
    int mid = (lo + hi) / 2;
    mergeSort(a, lo, mid);
    mergeSort(a, mid + 1, hi);
    int[] t = new int[hi - lo + 1];
    int i = lo, j = mid + 1, k = 0;
    while (i <= mid && j <= hi)
        t[k++] = a[i] <= a[j] ? a[i++] : a[j++];
    while (i <= mid) t[k++] = a[i++];
    while (j <= hi) t[k++] = a[j++];
    System.arraycopy(t, 0, a, lo, t.length);
}
```

Each level merges n elements in total, and there are $\log_2 n$ levels: O(n log n) time, O(n) extra memory.



gray: split · colored: merge

EXAMPLES · STEP THROUGH THE CODE

Reversing a String Recursively



main (stack bottom) → stack grows this way → top

reverse(s) = reverse(everything but the first character) + the first character.

Step 1 / 7

```
1 static String reverse(String s) {
2   if (s.length() <= 1) return s;
3   return reverse(s.substring(1)) + s.charAt(0);
4 }
▶ System.out.println(reverse("abc"));
```

VARIABLES

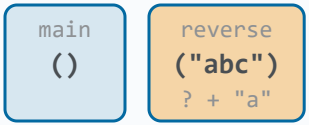
—

CALL STACK

`main()`

OUTPUT

Reversing a String Recursively



main (stack bottom) → stack grows this way → top

`reverse("bc")` is called; 'a' waits to be appended at the end.

Step 2 / 7

```
1 static String reverse(String s) {
2   if (s.length() <= 1) return s;
▶   return reverse(s.substring(1)) + s.charAt(0);
4 }
5 System.out.println(reverse("abc"));
```

VARIABLES

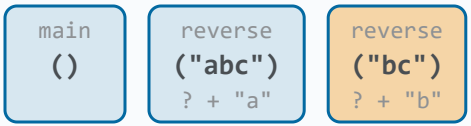
`s = "abc"`

CALL STACK

`reverse("abc")` `main()`

OUTPUT

Reversing a String Recursively



main (stack bottom) → stack grows this way → top

`reverse("c")` is called; 'b' waits to be appended at the end.

Step 3 / 7

```
1 static String reverse(String s) {
2   if (s.length() <= 1) return s;
▶   return reverse(s.substring(1)) + s.charAt(0);
4 }
5 System.out.println(reverse("abc"));
```

VARIABLES

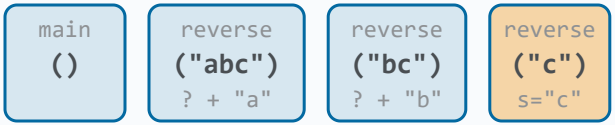
`s = "bc"`

CALL STACK

`reverse("bc")` `reverse("abc")` `main()`

OUTPUT

Reversing a String Recursively



main (stack bottom) → stack grows this way → top

Length 1 ≤ 1 → base case, "c" is returned.

Step 4 / 7

```
1 static String reverse(String s) {
▶   if (s.length() <= 1) return s;
3   return reverse(s.substring(1)) + s.charAt(0);
4 }
5 System.out.println(reverse("abc"));
```

VARIABLES

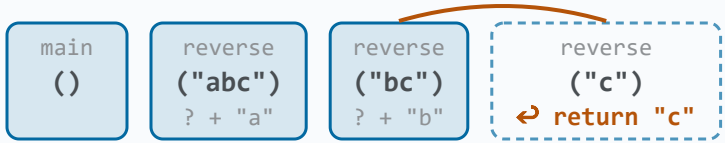
`s = "c"`

CALL STACK

`reverse("c")` `reverse("bc")` `reverse("abc")` `main()`

OUTPUT

Reversing a String Recursively



main (stack bottom) → stack grows this way → top

"c" + 'b' = **"cb"**.

Step 5 / 7

```
1 static String reverse(String s) {
2   if (s.length() <= 1) return s;
▶   return reverse(s.substring(1)) + s.charAt(0);
4 }
5 System.out.println(reverse("abc"));
```

VARIABLES

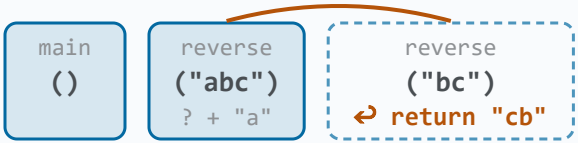
`s = "bc"`

CALL STACK

`reverse("bc")` `reverse("abc")` `main()`

OUTPUT

Reversing a String Recursively



main (stack bottom) → stack grows this way → top

"cb" + 'a' = **"cba"**.

Step 6 / 7

```
1 static String reverse(String s) {
2   if (s.length() <= 1) return s;
▶   return reverse(s.substring(1)) + s.charAt(0);
4 }
5 System.out.println(reverse("abc"));
```

VARIABLES

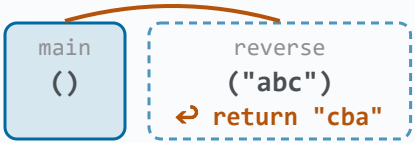
`s = "abc"`

CALL STACK

`reverse("abc")` `main()`

OUTPUT

Reversing a String Recursively



main (stack bottom) → stack grows this way → top

cba is printed. Each call creates a new string with `substring`.

Step 7 / 7

```
1 static String reverse(String s) {
2   if (s.length() <= 1) return s;
3   return reverse(s.substring(1)) + s.charAt(0);
4 }
▶ System.out.println(reverse("abc"));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
cba
```

EXAMPLES · STEP THROUGH THE CODE

Code Trace: Generating Subsets



main (stack bottom) → stack grows this way → top

main calls `subsets(a, 0, chosen)` with an empty list. Each element has two choices: include / exclude.

Step 1 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
▶ subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: Generating Subsets



main (stack bottom) → stack grows this way → top

i = 0: the array isn't finished (0 ≠ 2). First the **include 1** branch.


```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
▶      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 0` `chosen = []`

CALL STACK

`subsets(0)` `main()`

OUTPUT

Code Trace: Generating Subsets



`chosen.add(1)` : the list is now [1].


```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
▶      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 0` `chosen = [1]`

CALL STACK

`subsets(0)` `main()`

OUTPUT

Code Trace: Generating Subsets



New frame **i = 1**: decide on 2; include it first.

Step 4 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
▶      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 1` `chosen = [1]`

CALL STACK

`subsets(1)` `subsets(0)` `main()`

OUTPUT

Code Trace: Generating Subsets



`chosen.add(2)` : the list is [1, 2].

Step 5 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
▶      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 1` `chosen = [1, 2]`

CALL STACK

`subsets(1)` `subsets(0)` `main()`

OUTPUT

Code Trace: Generating Subsets



i = 2 = a.length: base case. Print the subset: **[1, 2]**.

Step 6 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
▶      if (i == a.length) {
▶          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 2` `chosen = [1, 2]`

CALL STACK

`subsets(2)` `subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
```

Code Trace: Generating Subsets



On the way back, `chosen.remove(...)` takes 2 out (**backtracking**) → [1].

Step 7 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
▶      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 1` `chosen = [1]`

CALL STACK

`subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
```

Code Trace: Generating Subsets



The **exclude 2** branch reaches a leaf: **[1]**.

Step 8 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
▶      if (i == a.length) {
▶          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 2` `chosen = [1]`

CALL STACK

`subsets(2)` `subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
[1]
```

Code Trace: Generating Subsets



main (stack bottom) → stack grows this way → top

subsets(i=1) is done; `remove` takes 1 out → [].


```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
▶      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 0` `chosen = []`

CALL STACK

`subsets(0)` `main()`

OUTPUT

```
[1, 2]
[1]
```

Code Trace: Generating Subsets



The **exclude 1** branch: a new frame with i = 1.

Step 10 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
▶      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 1` `chosen = []`

CALL STACK

`subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
[1]
```

Code Trace: Generating Subsets



`chosen.add(2)` : the list is [2].

Step 11 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
▶      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 1` `chosen = [2]`

CALL STACK

`subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
[1]
```

Code Trace: Generating Subsets



Leaf: **[2]** is printed.

Step 12 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
▶      if (i == a.length) {
▶          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 2` `chosen = [2]`

CALL STACK

`subsets(2)` `subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
[1]
[2]
```

Code Trace: Generating Subsets



`remove` takes 2 out → [].

Step 13 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
▶      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 1` `chosen = []`

CALL STACK

`subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
[1]
[2]
```

Code Trace: Generating Subsets



Last leaf: the empty set **[]**.

Step 14 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
▶      if (i == a.length) {
▶          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
12 subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

`i = 2` `chosen = []`

CALL STACK

`subsets(2)` `subsets(1)` `subsets(0)` `main()`

OUTPUT

```
[1, 2]
[1]
[2]
[]
```

Code Trace: Generating Subsets



main (stack bottom) → stack grows this way → top

All frames have closed. 2 elements → 2^2 = **4 subsets**; output order: [1, 2], [1], [2], [].

Step 15 / 15

```
1  static void subsets(int[] a, int i,
2                      List<Integer> chosen) {
3      if (i == a.length) {
4          System.out.println(chosen);
5          return;
6      }
7      chosen.add(a[i]);
8      subsets(a, i + 1, chosen);
9      chosen.remove(chosen.size() - 1);
10     subsets(a, i + 1, chosen);
11 }
▶ subsets(new int[]{1, 2}, 0, new ArrayList<>());
```

VARIABLES

—

CALL STACK

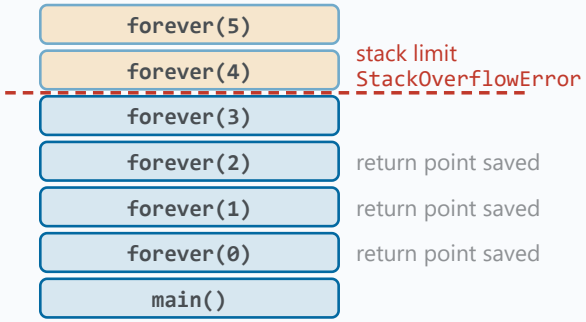
`main()`

OUTPUT

```
[1, 2]
[1]
[2]
[]
```

The Call Stack

- On a new call, the current method's **return point** and local variables are kept in its frame.
- After the base case, frames close **in reverse order** (last opened, first closed).
- With no base case (infinite recursion) or very deep recursion, the stack fills up and overflows: `StackOverflowError`.



Code Trace: `StackOverflowError`



main (stack bottom) → stack grows this way → top

main calls `forever(0)`.

Step 1 / 6

```
1 static void forever(int n) {
2     forever(n + 1);
3 }
▶ forever(0);
```

VARIABLES

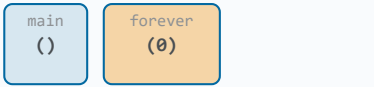
—

CALL STACK

`main()`

OUTPUT

Code Trace: `StackOverflowError`



First frame. With **no base case**, `forever(1)` is called at once.

Step 2 / 6

```
1 static void forever(int n) {
▶     forever(n + 1);
3 }
4 forever(0);
```

VARIABLES

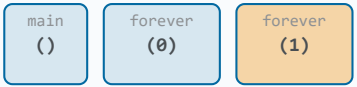
`n = 0`

CALL STACK

`forever(0)` `main()`

OUTPUT

Code Trace: `StackOverflowError`



The second frame; the previous one is still on the stack.

Step 3 / 6

```
1 static void forever(int n) {
▶     forever(n + 1);
3 }
4 forever(0);
```

VARIABLES

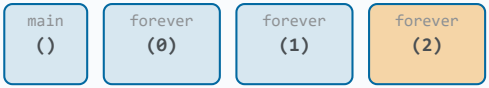
`n = 1`

CALL STACK

`forever(1)` `forever(0)` `main()`

OUTPUT

Code Trace: `StackOverflowError`



main (stack bottom) → stack grows this way → top

The third frame. No call returns, so no frame is removed.

Step 4 / 6

```
1 static void forever(int n) {
▶     forever(n + 1);
3 }
4 forever(0);
```

VARIABLES

`n = 2`

CALL STACK

`forever(2)` `forever(1)` `forever(0)` `main()`

OUTPUT

Code Trace: `StackOverflowError`



main (stack bottom) → stack grows this way → top

This goes on, but stack memory is **limited** (typically tens of thousands of calls).

Step 5 / 6

```
1 static void forever(int n) {
▶     forever(n + 1);
3 }
4 forever(0);
```

VARIABLES

`n = …`

CALL STACK

`…` `forever(2)` `forever(1)` `forever(0)` `main()`

OUTPUT

Code Trace: `StackOverflowError`



The stack is full: the JVM throws `StackOverflowError`, printed to **stderr**.

Step 6 / 6

```
1 static void forever(int n) {
▶     forever(n + 1);
3 }
4 forever(0);
```

VARIABLES

`n = …`

CALL STACK

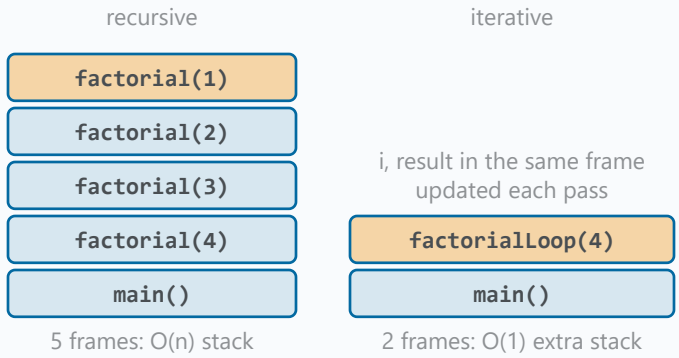
`forever(k)` `…` `forever(2)` `forever(1)` `forever(0)` `main()`

OUTPUT

```
Exception in thread "main"
java.lang.StackOverflowError
    at Main.forever(Main.java:2)
    at Main.forever(Main.java:2)
    ...
```

Recursion or Loop?

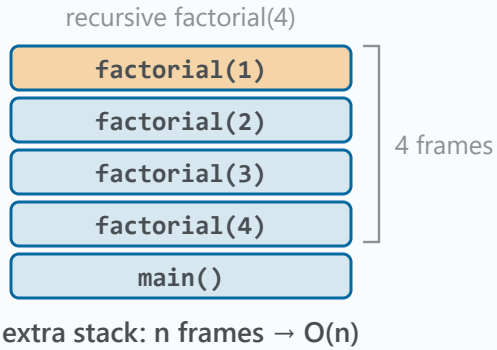
- For self-similar problems (trees, divide and conquer), recursion stays close to the definition and reads clearly.
- For simple linear repetition, a loop runs in one frame and uses less stack memory.
- Base the choice not just on short code but on correctness, readability and resource use.



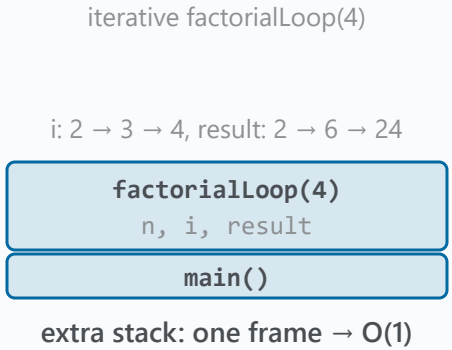
EFFICIENCY

Comparing Factorials

Property	Recursive	Iterative
Time complexity	O(n)	O(n)
Extra stack	O(n)	O(1)
Flow	Close to the definition	Explicit counter
Depth risk	Yes	No



extra stack: n frames → O(n)



extra stack: one frame → O(1)

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

main calls `factorialLoop(4)`.

Step 1 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
3     for (int i = 2; i <= n; i++)
4         result *= i;
5     return result;
6 }
▶ System.out.println(factorialLoop(4));
```

VARIABLES

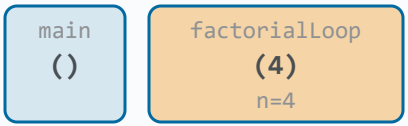
—

CALL STACK

`main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

A single frame opens; **result = 1**.

Step 2 / 11

```
1 static long factorialLoop(int n) {
▶     long result = 1;
3     for (int i = 2; i <= n; i++)
4         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

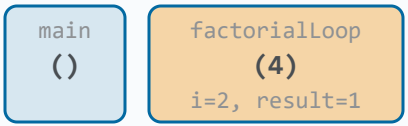
`n = 4` `result = 1`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

2 <= 4 → true

Step 3 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
▶     for (int i = 2; i <= n; i++)
4         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

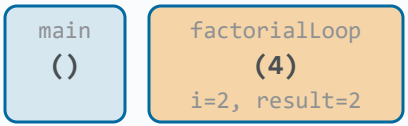
`n = 4` `result = 1` `i = 2`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

`result *= 2` → 1 × 2 = **2**. No new frame opened.

Step 4 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
3     for (int i = 2; i <= n; i++)
▶         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

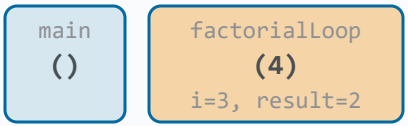
`n = 4` `result = 2` `i = 2`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

`i++` → **3 <= 4 → true**

Step 5 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
▶     for (int i = 2; i <= n; i++)
4         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

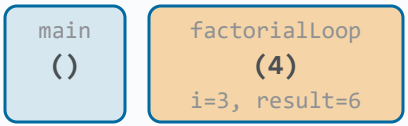
`n = 4` `result = 2` `i = 3`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

`result *= 3` → 2 × 3 = **6**. No new frame opened.

Step 6 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
3     for (int i = 2; i <= n; i++)
▶         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

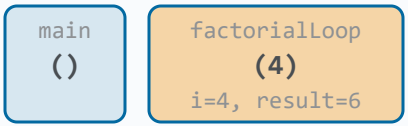
`n = 4` `result = 6` `i = 3`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

`i++` → **4 <= 4 → true**

Step 7 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
▶     for (int i = 2; i <= n; i++)
4         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

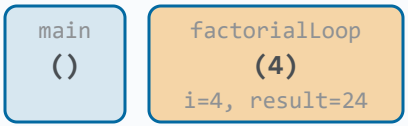
`n = 4` `result = 6` `i = 4`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

`result *= 4` → 6 × 4 = **24**. No new frame opened.


```
1 static long factorialLoop(int n) {
2     long result = 1;
3     for (int i = 2; i <= n; i++)
▶         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

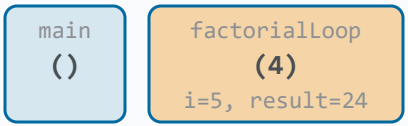
`n = 4` `result = 24` `i = 4`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

`i++` → **5 <= 4** → **false**: the loop ends.

Step 9 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
▶     for (int i = 2; i <= n; i++)
4         result *= i;
5     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

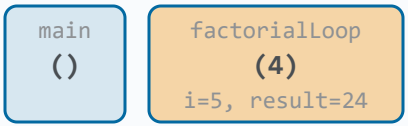
`n = 4` `result = 24` `i = 5`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

return 24.

Step 10 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
3     for (int i = 2; i <= n; i++)
4         result *= i;
▶     return result;
6 }
7 System.out.println(factorialLoop(4));
```

VARIABLES

`n = 4` `result = 24`

CALL STACK

`factorialLoop(4)` `main()`

OUTPUT

Comparison: Iterative `factorial(4)`



main (stack bottom) → stack grows this way → top

24 is printed. The stack never went beyond 2 frames → O(1) extra stack.

Step 11 / 11

```
1 static long factorialLoop(int n) {
2     long result = 1;
3     for (int i = 2; i <= n; i++)
4         result *= i;
5     return result;
6 }
▶ System.out.println(factorialLoop(4));
```

VARIABLES

—

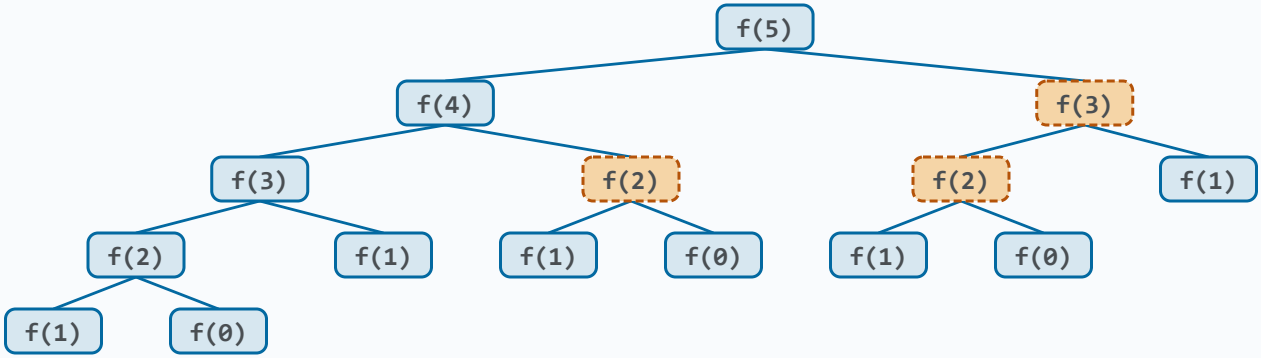
CALL STACK

`main()`

OUTPUT

```
24
```

The Fibonacci Problem: the `fib(5)` Recursion Tree



- fib(5) makes **15** calls: f(3) is computed **2** times, f(2) **3** times, f(1) 5 times (dashed orange = repeat).
- The subtrees of fib(n − 1) and fib(n − 2) overlap heavily; the number of calls grows roughly **exponentially** in n.

Code Trace: `fib(4)`

main calls `fib(4)`.

Step 1 / 15

```
1 static long fib(int n) {
2     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
▶ System.out.println(fib(4));
```

VARIABLES

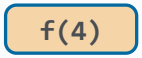
—

CALL STACK

`main()`

OUTPUT

Code Trace: `fib(4)`



`fib(4)` called: **4 <= 1 → false**.

Step 2 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

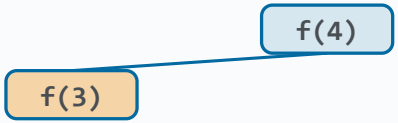
`n = 4`

CALL STACK

`fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(3)` called: **3 <= 1 → false**.

Step 3 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

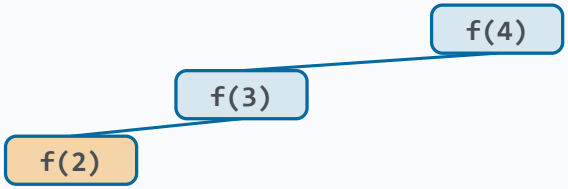
`n = 3`

CALL STACK

`fib(3)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(2)` called: **2 <= 1 → false**.

Step 4 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

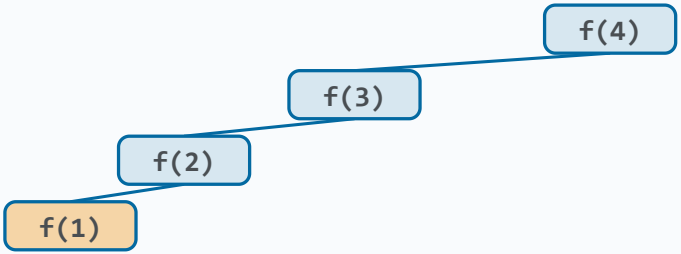
`n = 2`

CALL STACK

`fib(2)` `fib(3)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(1)` called: **1 <= 1** → **true**, **return 1**.

Step 5 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

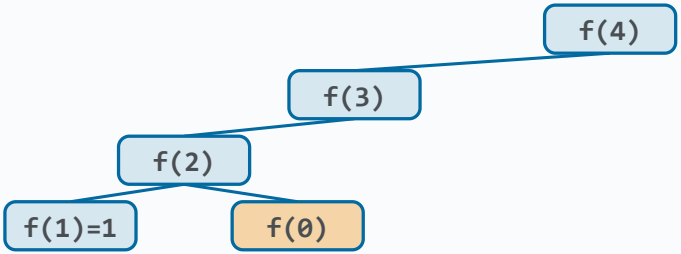
`n = 1`

CALL STACK

`fib(1)` `fib(2)` `fib(3)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(0)` called: **0 <= 1 → true**, **return 0**.

Step 6 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

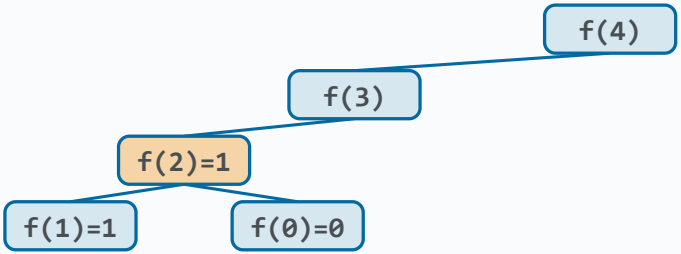
`n = 0`

CALL STACK

`fib(0)` `fib(2)` `fib(3)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



Both subresults ready: **1 + 0 = 1**, return.

Step 7 / 15

```
1 static long fib(int n) {
2     if (n <= 1) return n;
▶     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

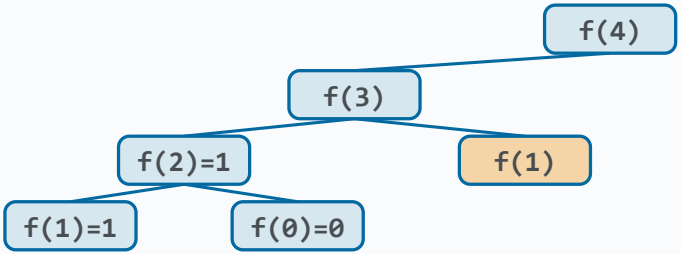
`n = 2`

CALL STACK

`fib(2)` `fib(3)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(1)` called: **1 <= 1** → **true**, **return 1**.

Step 8 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

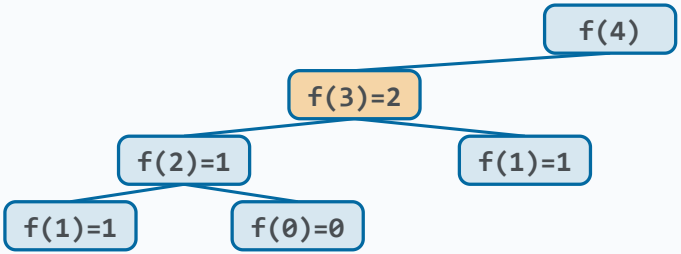
`n = 1`

CALL STACK

`fib(1)` `fib(3)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



Both subresults ready: **1 + 1 = 2**, return.

Step 9 / 15

```
1 static long fib(int n) {
2     if (n <= 1) return n;
▶     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

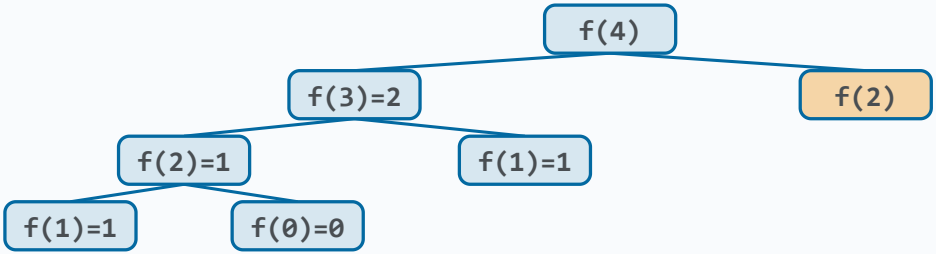
`n = 3`

CALL STACK

`fib(3)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(2)` called: **2 <= 1** → **false**. **Repeat:** f(2) is computed again!

Step 10 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

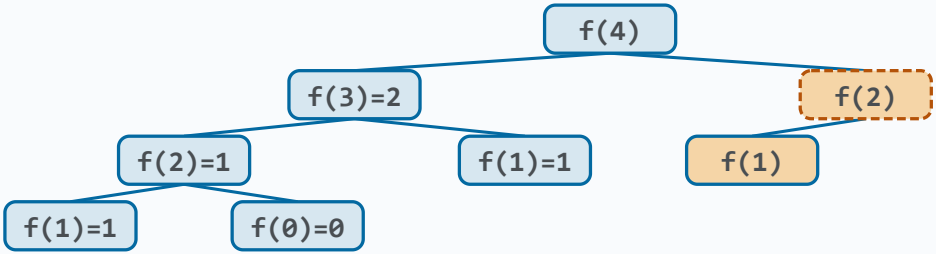
`n = 2`

CALL STACK

`fib(2)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(1)` called: **1 <= 1** → **true**, **return 1**.

Step 11 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

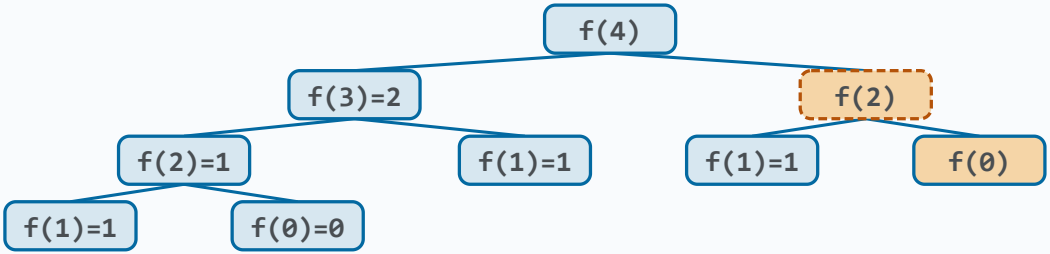
`n = 1`

CALL STACK

`fib(1)` `fib(2)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



`fib(0)` called: **0 <= 1 → true**, **return 0**.

Step 12 / 15

```
1 static long fib(int n) {
▶     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

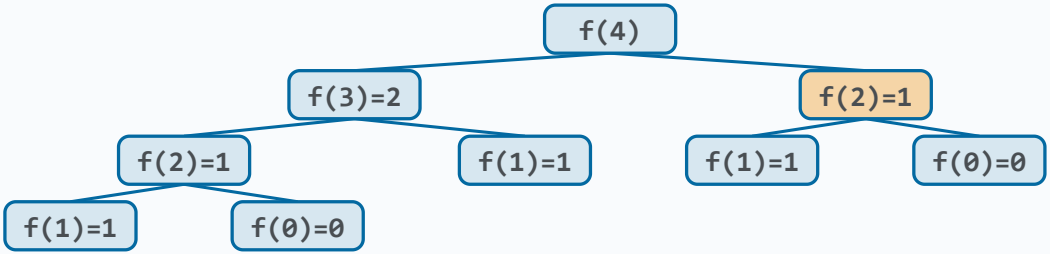
`n = 0`

CALL STACK

`fib(0)` `fib(2)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



Both subresults ready: **1 + 0 = 1**, return.

Step 13 / 15

```
1 static long fib(int n) {
2     if (n <= 1) return n;
▶     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

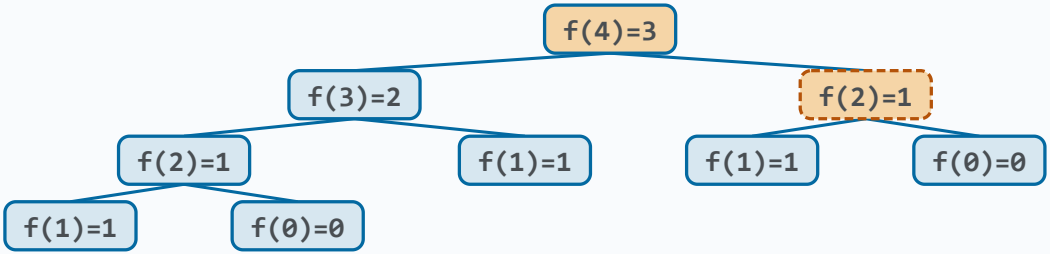
`n = 2`

CALL STACK

`fib(2)` `fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



Both subresults ready: **2 + 1 = 3**, return.

Step 14 / 15

```
1 static long fib(int n) {
2     if (n <= 1) return n;
▶     return fib(n - 1) + fib(n - 2);
4 }
5 System.out.println(fib(4));
```

VARIABLES

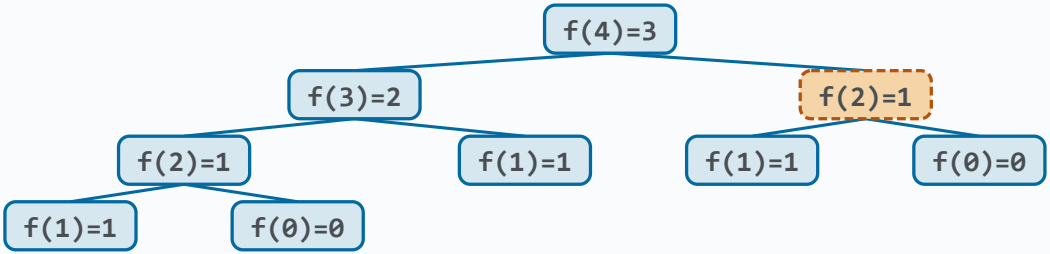
`n = 4`

CALL STACK

`fib(4)` `main()`

OUTPUT

Code Trace: `fib(4)`



3 is printed. Even for a small n, 9 calls were made.

Step 15 / 15

```
1 static long fib(int n) {
2     if (n <= 1) return n;
3     return fib(n - 1) + fib(n - 2);
4 }
▶ System.out.println(fib(4));
```

VARIABLES

—

CALL STACK

`main()`

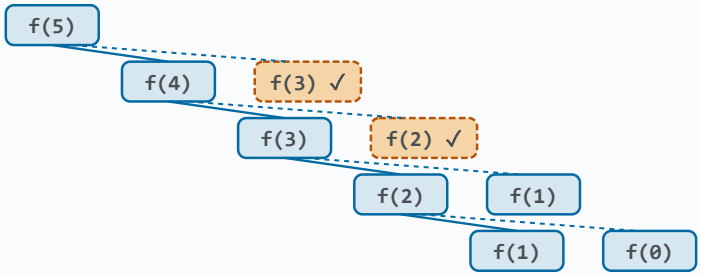
OUTPUT

```
3
```

IMPROVEMENT

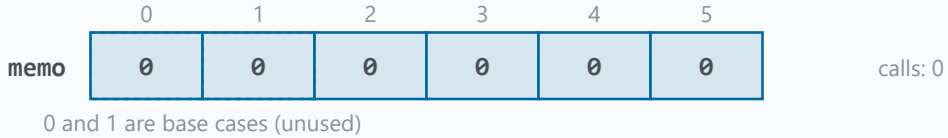
Memoization

- Results of solved subproblems are stored in an array (a cache).
- When the same input comes again, the recursion tree isn't rebuilt; the **stored value** is returned.
- 9 calls instead of 15 for fib(5); the gap grows huge for large n.
- The time savings cost O(n) extra memory.



✓ = read from memo, subtree not expanded

Code Trace: Memoized `fib(5)`



A 6-cell memo array; **0** = not computed yet.

Step 1 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
▶ long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

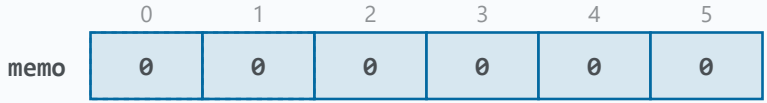
`memo = [0, 0, 0, 0, 0, 0]`

CALL STACK

`main()`

OUTPUT

Code Trace: Memoized `fib(5)`



`fib(5)` : **memo[5] == 0** → not yet known; compute it.

Step 2 / 15

```
1 static long fib(int n, long[] memo) {
▶     if (n <= 1) return n;
▶     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 5`

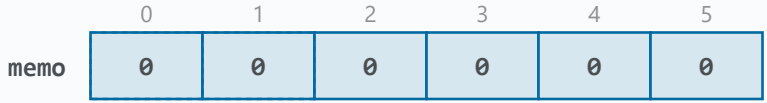
`memo = [0, 0, 0, 0, 0, 0]`

CALL STACK

`fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



calls: 2

0 and 1 are base cases (unused)

`fib(4)` : **memo[4] == 0** → not yet known; compute it.

Step 3 / 15

```
1 static long fib(int n, long[] memo) {
▶     if (n <= 1) return n;
▶     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 4`

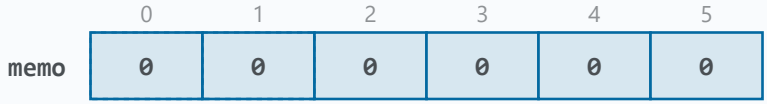
`memo = [0, 0, 0, 0, 0, 0]`

CALL STACK

`fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



calls: 3

0 and 1 are base cases (unused)

`fib(3)` : **memo[3] == 0** → not yet known; compute it.

Step 4 / 15

```
1 static long fib(int n, long[] memo) {
▶     if (n <= 1) return n;
▶     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 3`

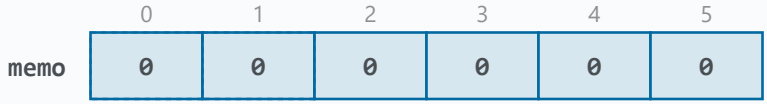
`memo = [0, 0, 0, 0, 0, 0]`

CALL STACK

`fib(3)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



calls: 4

0 and 1 are base cases (unused)

`fib(2)` : **memo[2] == 0** → not yet known; compute it.

Step 5 / 15

```
1 static long fib(int n, long[] memo) {
▶     if (n <= 1) return n;
▶     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 2`

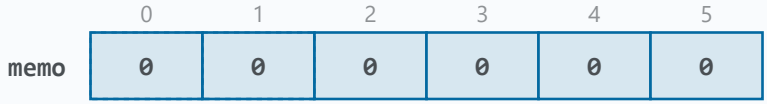
`memo = [0, 0, 0, 0, 0, 0]`

CALL STACK

`fib(2)` `fib(3)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



calls: 5

0 and 1 are base cases (unused)

`fib(1)` : **1 <= 1** → **true**, return 1.

Step 6 / 15

```
1 static long fib(int n, long[] memo) {
▶     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 1`

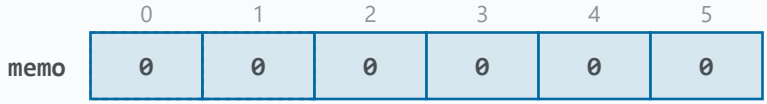
`memo = [0, 0, 0, 0, 0, 0]`

CALL STACK

`fib(1)` `fib(2)` `fib(3)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



calls: 6

0 and 1 are base cases (unused)

`fib(0)` : **0 <= 1** → **true**, return 0.

Step 7 / 15

```
1 static long fib(int n, long[] memo) {
▶     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 0`

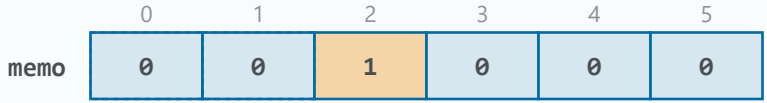
`memo = [0, 0, 0, 0, 0, 0]`

CALL STACK

`fib(0)` `fib(2)` `fib(3)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



memo[2] = 1 + 0 = 1 is stored and returned.

Step 8 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
▶     memo[n] = fib(n - 1, memo)
▶             + fib(n - 2, memo);
▶     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 2`

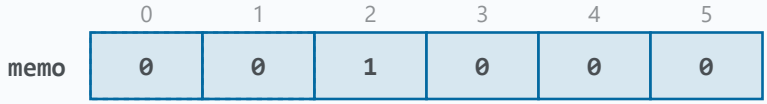
`memo = [0, 0, 1, 0, 0, 0]`

CALL STACK

`fib(2)` `fib(3)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



calls: 7

0 and 1 are base cases (unused)

`fib(1)` : **1 <= 1** → **true**, return 1.

Step 9 / 15

```
1 static long fib(int n, long[] memo) {
▶     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 1`

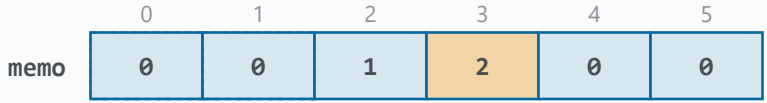
`memo = [0, 0, 1, 0, 0, 0]`

CALL STACK

`fib(1)` `fib(3)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



memo[3] = 1 + 1 = 2 is stored and returned.

Step 10 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
▶     memo[n] = fib(n - 1, memo)
▶             + fib(n - 2, memo);
▶     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 3`

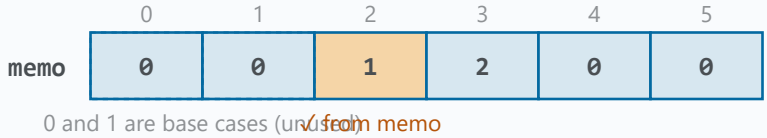
`memo = [0, 0, 1, 2, 0, 0]`

CALL STACK

`fib(3)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



`fib(2)` : **memo[2] = 1 != 0 → return 1** without computing.

Step 11 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
▶     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 2`

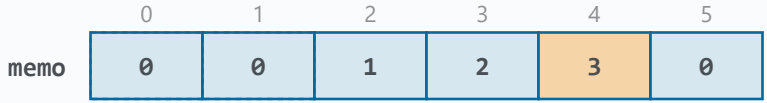
`memo = [0, 0, 1, 2, 0, 0]`

CALL STACK

`fib(2)` `fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



memo[4] = 2 + 1 = 3 is stored and returned.

Step 12 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
▶     memo[n] = fib(n - 1, memo)
▶             + fib(n - 2, memo);
▶     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 4`

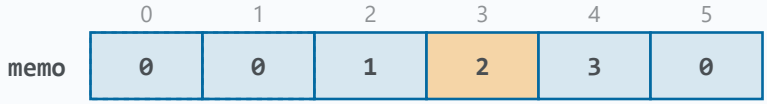
`memo = [0, 0, 1, 2, 3, 0]`

CALL STACK

`fib(4)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



`fib(3)` : **memo[3] = 2 != 0 → return 2** without computing.

Step 13 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
▶     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 3`

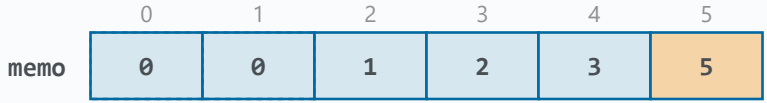
`memo = [0, 0, 1, 2, 3, 0]`

CALL STACK

`fib(3)` `fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



memo[5] = 3 + 2 = 5 is stored and returned.

Step 14 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
▶     memo[n] = fib(n - 1, memo)
▶             + fib(n - 2, memo);
▶     return memo[n];
7 }
8 long[] memo = new long[6];
9 System.out.println(fib(5, memo));
```

VARIABLES

`n = 5`

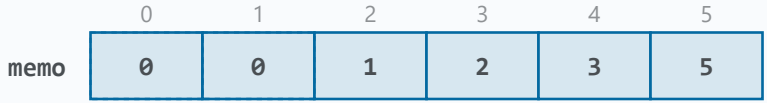
`memo = [0, 0, 1, 2, 3, 5]`

CALL STACK

`fib(5)` `main()`

OUTPUT

Code Trace: Memoized `fib(5)`



5 is printed. 9 calls in all (15 without the memo) → O(n) time.

Step 15 / 15

```
1 static long fib(int n, long[] memo) {
2     if (n <= 1) return n;
3     if (memo[n] != 0) return memo[n];
4     memo[n] = fib(n - 1, memo)
5               + fib(n - 2, memo);
6     return memo[n];
7 }
8 long[] memo = new long[6];
▶ System.out.println(fib(5, memo));
```

VARIABLES

`memo = [0, 0, 1, 2, 3, 5]`

CALL STACK

`main()`

OUTPUT

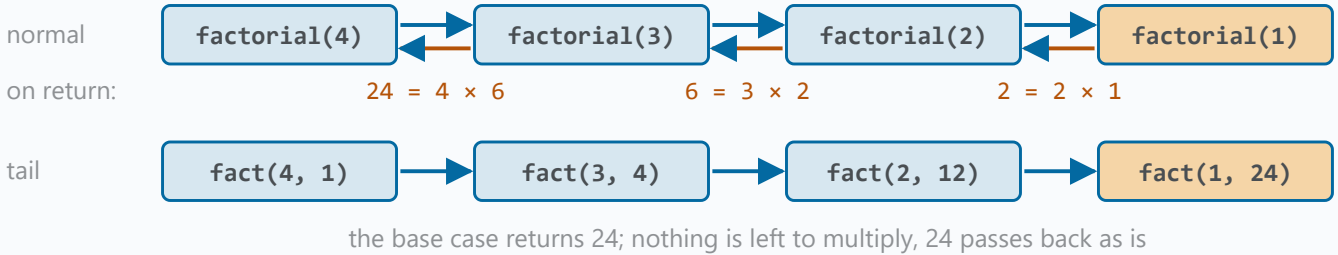
```
5
```

IMPROVEMENT

Tail Recursion

- The recursive call is the method's **last operation**; nothing is left to do on return.
- The partial result is carried in a parameter (`acc`).
- Some languages optimize these calls without growing the stack.
- The Java language/JVM does **not guarantee** tail-call optimization: frames still pile up.

```
static long fact(int n, long acc) {
    if (n <= 1) return acc;
    return fact(n - 1, n * acc);
}
// fact(4, 1) → fact(3, 4)
//   → fact(2, 12) → fact(1, 24) → 24
```



the base case returns 24; nothing is left to multiply, 24 passes back as is

IMPROVEMENT · STEP THROUGH THE CODE

Code Trace: `fact(4, 1)`



main calls fact(4, 1); acc starts at 1

main (stack bottom) → stack grows this way → top

main calls `fact(4, 1)` : the second parameter, **acc**, starts at 1.

Step 1 / 6

```
1 static long fact(int n, long acc) {
2     if (n <= 1) return acc;
3     return fact(n - 1, n * acc);
4 }
▶ System.out.println(fact(4, 1)); // 24
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `fact(4, 1)`



main (stack bottom) → stack grows this way → top

The frame opens. **4 <= 1 → false**. The last operation is the call `fact(3, 4 * 1)`, i.e. **fact(3, 4)**; no multiplication waits for the return.

Step 2 / 6

```
1 static long fact(int n, long acc) {
▶     if (n <= 1) return acc;
▶     return fact(n - 1, n * acc);
4 }
5 System.out.println(fact(4, 1)); // 24
```

VARIABLES

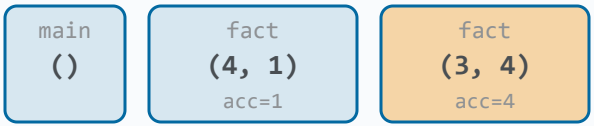
`n = 4` `acc = 1`

CALL STACK

`fact(4, 1)` `main()`

OUTPUT

Code Trace: `fact(4, 1)`



main (stack bottom) → stack grows this way → top

The multiplication happened **before** the call: next comes **fact(2, 12)**. The old frame has nothing left to do, yet Java does **not remove** it.

Step 3 / 6

```
1 static long fact(int n, long acc) {
▶     if (n <= 1) return acc;
▶     return fact(n - 1, n * acc);
4 }
5 System.out.println(fact(4, 1)); // 24
```

VARIABLES

`n = 3` `acc = 4`

CALL STACK

`fact(3, 4)` `fact(4, 1)` `main()`

OUTPUT

Code Trace: `fact(4, 1)`



main (stack bottom) → stack grows this way → top

Next call: **fact(1, 24)**. The stack still grows by **one frame** per call.

Step 4 / 6

```
1 static long fact(int n, long acc) {
▶     if (n <= 1) return acc;
▶     return fact(n - 1, n * acc);
4 }
5 System.out.println(fact(4, 1)); // 24
```

VARIABLES

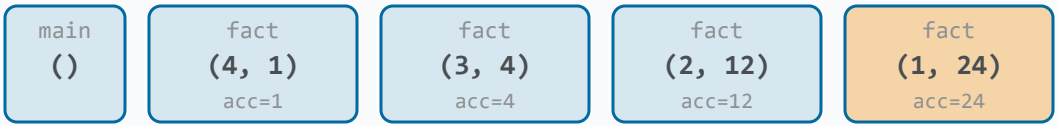
`n = 2` `acc = 12`

CALL STACK

`fact(2, 12)` `fact(3, 4)` `fact(4, 1)` `main()`

OUTPUT

Code Trace: `fact(4, 1)`



main (stack bottom) → stack grows this way → top

1 <= 1 → true: base case, `return acc` → **24**. Tail recursion or not, **5 frames** (main + 4) piled up: Java has no tail-call optimization.

Step 5 / 6

```
1 static long fact(int n, long acc) {
▶     if (n <= 1) return acc;
3     return fact(n - 1, n * acc);
4 }
5 System.out.println(fact(4, 1)); // 24
```

VARIABLES

`n = 1` `acc = 24`

CALL STACK

`fact(1, 24)` `fact(2, 12)` `fact(3, 4)` `fact(4, 1)` `main()`

OUTPUT

IMPROVEMENT · STEP THROUGH THE CODE

Code Trace: `fact(4, 1)`



main (stack bottom) → stack grows this way → top

24 passes back through the frames to main; no multiplication, just carrying the result. Memory is still **O(n)**.

Step 6 / 6

```
1 static long fact(int n, long acc) {
2     if (n <= 1) return acc;
3     return fact(n - 1, n * acc);
4 }
▶ System.out.println(fact(4, 1)); // 24
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
24
```

VERIFICATION

Overflow and Edge Cases

long overflow

`factorial(20)` → `2432902008176640000` (correct).
`factorial(21)` → `-4249290049419214848`
No exception is thrown; the value silently wraps. A `long` holds at most 9223372036854775807.

Negative exponent

`power(2, -1)`: the base case `n == 0` is never reached; n goes −1, −2, … → **StackOverflowError**. Guard: throw `IllegalArgumentException` if `n < 0`.

Fibonacci: recursive ≈ O(2ⁿ), iterative O(n)

n	fib(n) recursive: number of calls	fib(n) iterative: number of passes
10	177	10
20	21891	20
30	2692537	30

The numbers come from a counter incremented on each call; this is not a timing.

VERIFICATION

Debugging

- Show that every input reaches the base case.
- Pin down the measure by which the problem shrinks on each call.
- For a small input, tabulate the order of calls and returns (right: `sum(2)`).
- Assess the recursion depth and any repeated subproblems.

Depth	Event	Value
0	`sum(2)` called	
1	`sum(1)` called	
2	`sum(0)` called	
2	`sum(0)` returned	0
1	`sum(1)` returned	1
0	`sum(2)` returned	3

SUMMARY

Chapter Summary

Structure

A base case + a recursive case that shrinks the problem.

Call stack

Each call opens a frame; after the base case the frames close in reverse order.

Cost

Linear recursion uses O(n) stack; a loop usually does the same job in O(1).

Memoization

Storing repeated subproblem results takes fib from exponential to O(n).

SUMMARY

Review Questions

1. What happens without a base case?
2. Why does the call stack unwind in reverse order?
3. How much extra stack does recursive factorial use?
4. Why is naive Fibonacci slow?
5. Why doesn't tail recursion make Java stack-safe?

▶ **Answers**

Exercises

1. Write the sum of an array recursively: `sum(a, i)` = a[i] + sum(a, i + 1).
2. Count the digits of a number recursively (`digits(472)` = 3).
3. Write a `toBinary(n)` method that converts a decimal number to binary recursively (13 → "1101").
4. How many moves does Hanoi need for n = 4? Check it with the program.
5. Write the base cases for `choose(n, k) = choose(n − 1, k − 1) + choose(n − 1, k)` ; speed it up with memoization.
6. Count recursively how many times a character occurs in a string.

References

- Roberts, Thinking Recursively with Java, Wiley, 2006.
- Sedgewick and Wayne, Computer Science: An Interdisciplinary Approach, Addison-Wesley, 2016: **2.3 Recursion**.
- Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms, 4th ed., MIT Press, 2022.
- Liang, Introduction to Java Programming and Data Structures, 13th ed., Pearson, 2024.
- Oracle, Java SE 21 API: **StackOverflowError**.
- Downey and Mayfield, Think Java, 2nd ed., O'Reilly, 2019 (**free online edition**).

The End

Chapter 12: Recursion

Previous: Chapter 11: Strings