Chapter 3.5: Recursion

Object Oriented Programming with Java

Base case · Call stack · Factorial and sum · Fibonacci, Lucas, Catalan, Bell · Palindromes and GCD · Towers of Hanoi · Efficiency and loops · Tail recursion · Memoization

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_03_05_Recursion.html

INTRODUCTION

Learning Outcomes

Concept

Define recursion, the base case and the recursive step; reduce a problem to a smaller copy of itself.

How it works

Trace step by step how frames pile up on the call stack and return; draw a call tree.

Writing

Write recursive Java methods for factorial, sum, series, digit sum, palindromes, GCD and the Towers of Hanoi.

Errors

Recognize and prevent infinite recursion, `StackOverflowError` and missing `return` statements.

Efficiency

Compare recursive and iterative solutions; convert recursion to a loop when needed.

Advanced techniques

Explain tail recursion and memoization; eliminate repeated computation.

What Is Recursion?

- **Recursion**: a method solving a task by breaking it into smaller pieces and **calling itself**.
- Each call solves a **smaller instance of the same** problem; the smallest instance is solved directly.
- The process is self-similar: a structure contains a smaller **structure of the same kind**.
- For some problems (trees, series, divide and conquer) it gives **clearer and cleaner** solutions than loops.

Java has no keyword for recursion: a method simply calls **itself by name** in its body.

Matryoshka dolls

Each doll holds a smaller one; the smallest does not open (the base case).

Which row am I in?

Everyone asks the person in front, who asks the next… The first row says "1"; on the way back each person adds 1.

Facing mirrors

An image within an image: recursion without a base case never ends.

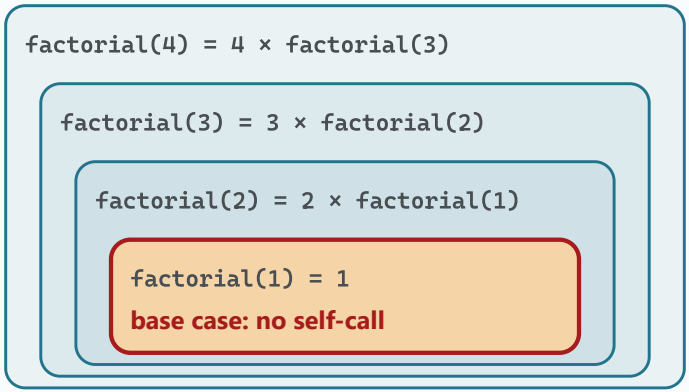
Base Case and Recursive Step

1. Base case

The smallest problem, whose answer is known **directly**. The method does not call itself here, so the recursion stops. E.g. `1! = 1`.

2. Recursive step

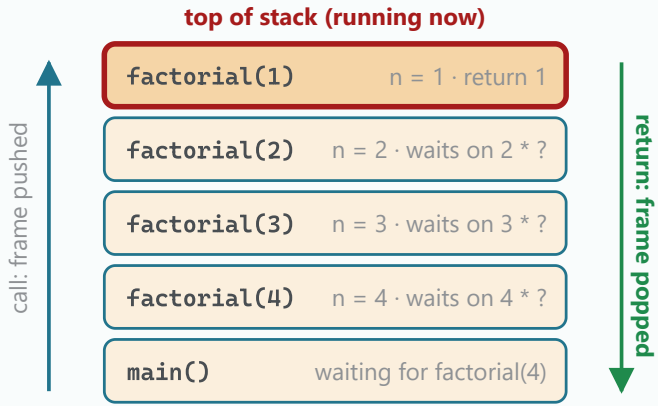
The method makes the problem **one step smaller**, calls itself on it and builds its answer from the result. E.g. `4! = 4 × 3!`.



It mirrors the inductive definition in mathematics:
$n! = 1$ (if $n \le 1$), $n! = n \times (n-1)!$ (if $n > 1$)

Call Stack

- Each method call opens a **stack frame** in memory, holding the parameters, local variables and return address.
- Frames are stacked on the **call stack**; only the **top** frame runs, the others wait.
- In recursion, **several frames** of the same method are alive at once, each with its own `n`.
- When the method executes `return`, its frame is removed and the value goes to the frame below (last in, first out: LIFO).
- The stack is limited in size; very deep recursion ends with `StackOverflowError`.



Computing a Factorial

```
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;   // base case
    } else {
        return n * factorial(n - 1); // recursion
    }
}
```

Mathematics

$n! = 1 \times 2 \times \ldots \times n$

$n! = n \times (n - 1)!,\ 0! = 1! = 1$

Lines 2–3: base case

`0! = 1! = 1`; recursion stops here.

Line 5: recursive step

`n!` is `(n - 1)!` times `n`.

Progress

Each call decreases `n` by 1 → for n ≥ 0 the base case is always reached.

Overflow

`int` cannot hold 13!: `factorial(13)` = 1932053504 (wrong!). `long` works up to 20!.

Calls: `factorial(5)` → **120** · `factorial(0)` → **1**

Expanding factorial(4)

```
factorial(4)
  = 4 * factorial(3)
  = 4 * (3 * factorial(2))
  = 4 * (3 * (2 * factorial(1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
```

Descent (calls)

In the first four lines each call opens another; the multiplications are **deferred**, because the right-hand value is not known yet.

Bottom (base case)

`factorial(1)` returns 1 directly. Since the code stops at `n == 1`, `factorial(0)` is never called.

Ascent (returns)

The pending multiplications complete **from the inside out**: 2 × 1, 3 × 2, 4 × 6.

`factorial(4)`: Step-by-Step Code Trace



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

Step 1 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
5          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
▶      System.out.println(factorial(4));
11 }
```

CALL STACK

`main()`

VARIABLES

`n = –`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace

`factorial(4)` `n = 4` | `factorial(3)` | `factorial(2)` | `factorial(1)`

New frame: `n = 4`. The condition `n == 0 || n == 1` is false → not the base case.

Step 2 / 12

```
1  static int factorial(int n) {
▶      if (n == 0 || n == 1) {
3          return 1;
4      } else {
5          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

- `factorial(4)`
- `main()`

VARIABLES

`n = 4`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace

`factorial(4)` `4 * ?` | `factorial(3)` | `factorial(2)` | `factorial(1)`

`4 * factorial(3)` must be computed. The multiplication waits; first `factorial(3)` is called.

Step 3 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
▶          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

- `factorial(4)`
- `main()`

VARIABLES

`n = 4`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



New frame: `n = 3`. The condition `n == 0 || n == 1` is false → not the base case.

Step 4 / 12

```
1  static int factorial(int n) {
▶      if (n == 0 || n == 1) {
3          return 1;
4      } else {
5          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

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

VARIABLES

`n = 3`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



`3 * factorial(2)` must be computed. The multiplication waits; first `factorial(2)` is called.

Step 5 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
▶          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

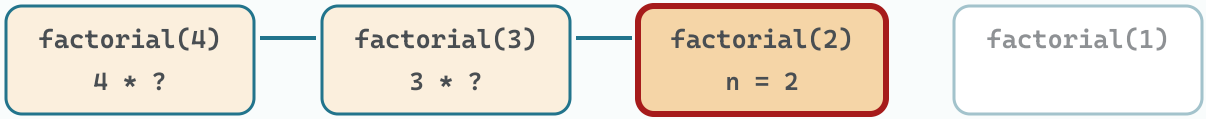
- **factorial(3)**
- factorial(4)
- main()

VARIABLES

n = **3**

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



New frame: `n = 2`. The condition `n == 0 || n == 1` is false → not the base case.

Step 6 / 12

```
1  static int factorial(int n) {
▶      if (n == 0 || n == 1) {
3          return 1;
4      } else {
5          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

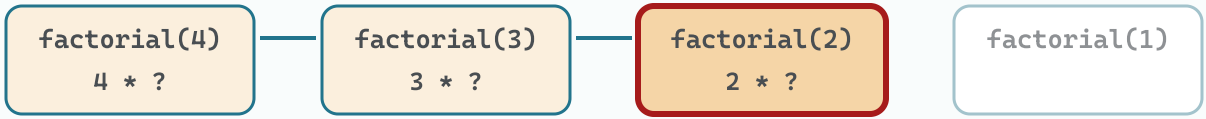
- `factorial(2)`
- `factorial(3)`
- `factorial(4)`
- `main()`

VARIABLES

`n = 2`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



`2 * factorial(1)` must be computed. The multiplication waits; first `factorial(1)` is called.

Step 7 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
▶          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

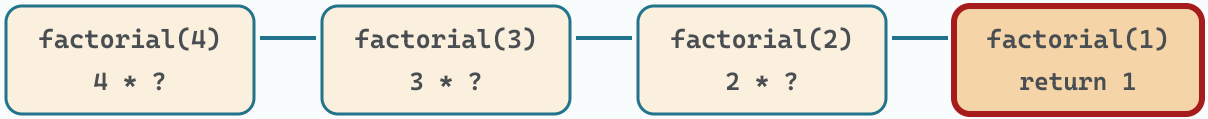
- **factorial(2)**
- factorial(3)
- factorial(4)
- main()

VARIABLES

n = **2**

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



`n = 1` : **base case**. No new call is made; **1** is returned. The frames now start to return.

Step 8 / 12

```
1  static int factorial(int n) {
▶      if (n == 0 || n == 1) {
▶          return 1;
4      } else {
5          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

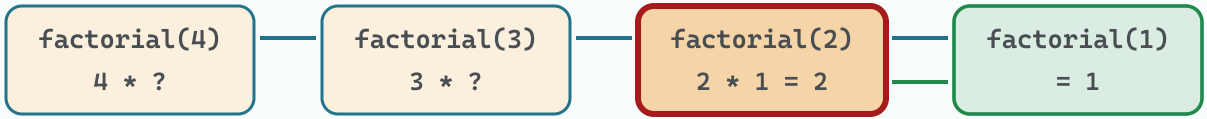
- **factorial(1)**
- factorial(2)
- factorial(3)
- factorial(4)
- main()

VARIABLES

`n = 1` `ret = 1`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



`factorial(1)` returns 1 and is popped: `2 * 1 = 2` is returned.

Step 9 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
▶          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

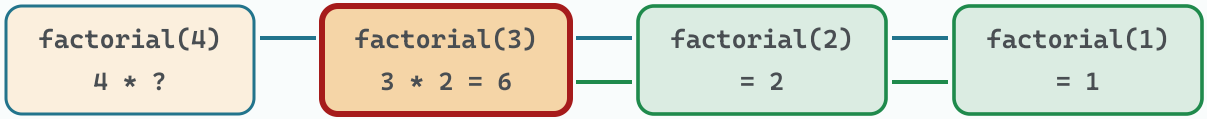
- **factorial(2)**
- factorial(3)
- factorial(4)
- main()

VARIABLES

`n = 2` `ret = 2`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



`factorial(2)` returns 2 and is popped: `3 * 2 = 6` is returned.

Step 10 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
▶          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

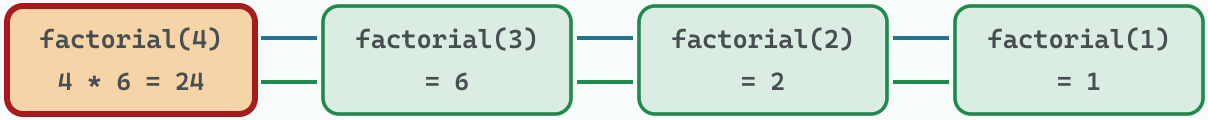
- **factorial(3)**
- factorial(4)
- main()

VARIABLES

`n = 3` `ret = 6`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



`factorial(3)` returns 6 and is popped: `4 * 6 = 24` is returned.

Step 11 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
▶          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
10     System.out.println(factorial(4));
11 }
```

CALL STACK

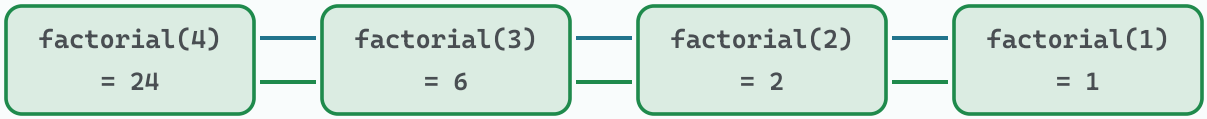
- `factorial(4)`
- `main()`

VARIABLES

`n = 4` `ret = 24`

OUTPUT

`factorial(4)`: Step-by-Step Code Trace



24 returns to `main` and is printed. 4 calls were made; at the deepest point the stack held 5 frames.

Step 12 / 12

```
1  static int factorial(int n) {
2      if (n == 0 || n == 1) {
3          return 1;
4      } else {
5          return n * factorial(n - 1);
6      }
7  }
8
9  public static void main(String[] args) {
▶      System.out.println(factorial(4));
11 }
```

CALL STACK

`main()`

VARIABLES

`n = –`

OUTPUT

```
24
```

Computing a Sum

```java
int sum(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + sum(n - 1);
    }
}
```

Sum of the numbers from 1 to n: `sum(n) = n + sum(n − 1)`, `sum(1) = 1`.

```
sum(5) = 5 + sum(4)
       = 5 + (4 + (3 + (2 + sum(1))))
       = 5 + (4 + (3 + (2 + 1)))
       = 15
```

Same pattern as factorial; only **×** becomes **+**, and the base case differs. `sum(100)` = **5050**.

The only base case is `n == 1`. `sum(0)` or a negative argument never makes n equal 1: 0, −1, −2, … infinite recursion → `StackOverflowError`.

A safe version uses the base case `if (n <= 0) return 0;`, which also catches every invalid input.

SERIES

Fibonacci Series

- Starts with 0, then 1: **F(0) = 0, F(1) = 1**.
- General formula: **F(n) = F(n − 1) + F(n − 2)**: each term is the sum of the previous two.
- The method calls itself **twice**: the calls form a **tree** (tree recursion).

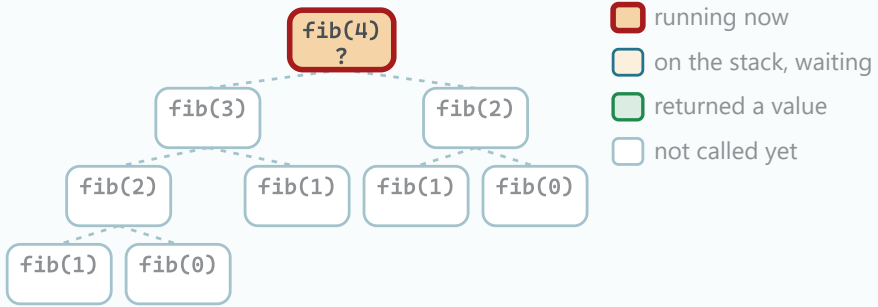
```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

n	0	1	2	3	4	5	6	7	8	9	10	11	12
F(n)	0	1	1	2	3	5	8	13	21	34	55	89	144

Known from the rabbit-breeding problem (Leonardo Fibonacci, 1202). The ratio of consecutive terms approaches the golden ratio (≈ 1.618): 144 / 89 ≈ 1.618.

SERIES · STEP-BY-STEP CODE TRACE

`fibonacci(4)`: Call Tree and Stack



`fibonacci(4)`: n > 1. The **left** term of the sum comes first: `fibonacci(3)` is called.

Step 1 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
3           return n;
4       } else {
▶           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

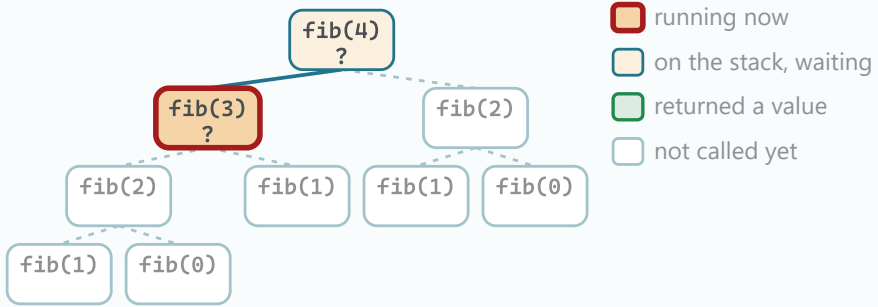
CALL STACK

`fibonacci(4)`

VARIABLES

`n = 4`

`fibonacci(4)`: Call Tree and Stack



`fibonacci(3)`: n > 1. The **left** term of the sum comes first: `fibonacci(2)` is called.

Step 2 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
3           return n;
4       } else {
▶           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

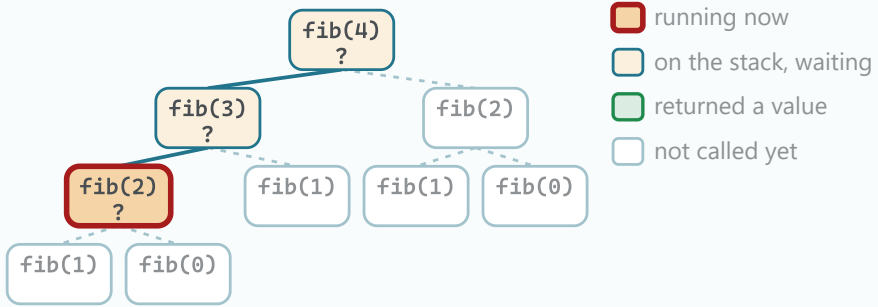
CALL STACK

- **fibonacci(3)**
- fibonacci(4)

VARIABLES

n = **3**

`fibonacci(4)`: Call Tree and Stack



`fibonacci(2)`: n > 1. The **left** term of the sum comes first: `fibonacci(1)` is called.

Step 3 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
3           return n;
4       } else {
▶           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

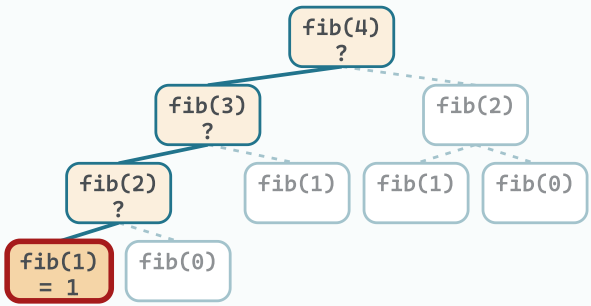
CALL STACK

- **fibonacci(2)**
- fibonacci(3)
- fibonacci(4)

VARIABLES

n = **2**

`fibonacci(4)`: Call Tree and Stack



`fibonacci(1)` : `n <= 1` → base case, **1** is returned.

Step 4 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
▶           return n;
4       } else {
5           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

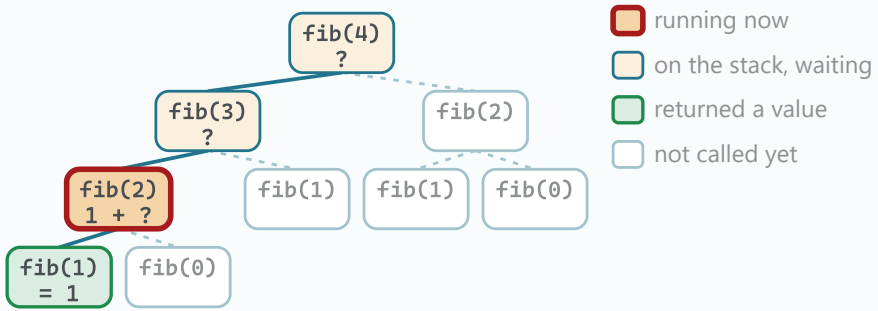
CALL STACK

- **fibonacci(1)**
- fibonacci(2)
- fibonacci(3)
- fibonacci(4)

VARIABLES

- `n = 1`
- `ret = 1`

`fibonacci(4)`: Call Tree and Stack



The left term gave 1. Now the **right** term: `fibonacci(0)` is called.

Step 5 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

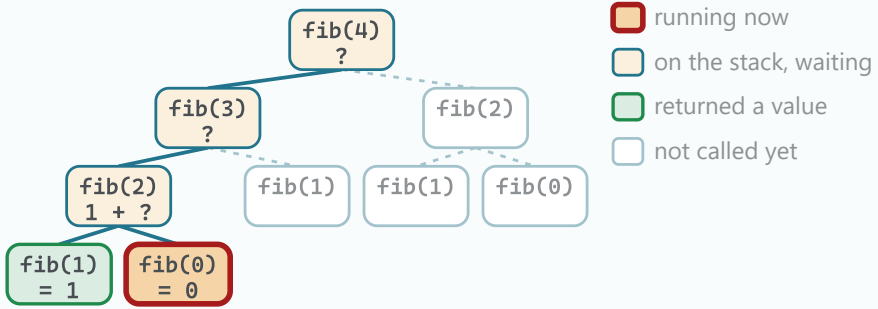
CALL STACK

- **fibonacci(2)**
- fibonacci(3)
- fibonacci(4)

VARIABLES

- n = **2**
- f(n−1) = **1**

`fibonacci(4)`: Call Tree and Stack



`fibonacci(0)` : `n <= 1` → base case, **0** is returned.

Step 6 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
▶           return n;
4       } else {
5           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

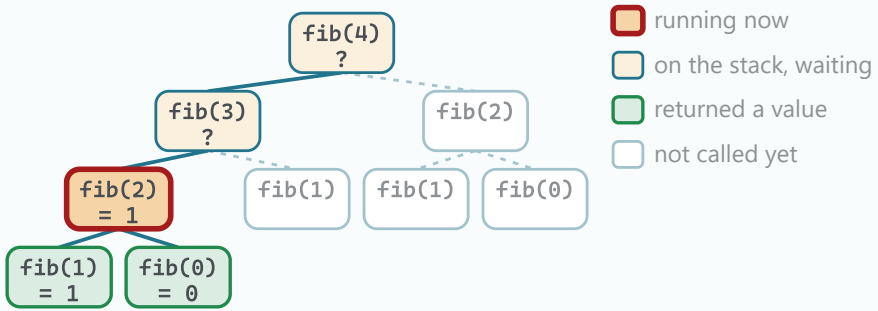
CALL STACK

- `fibonacci(0)`
- `fibonacci(2)`
- `fibonacci(3)`
- `fibonacci(4)`

VARIABLES

- `n = 0`
- `ret = 0`

`fibonacci(4)`: Call Tree and Stack



`1 + 0 = 1` is returned.

Step 7 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

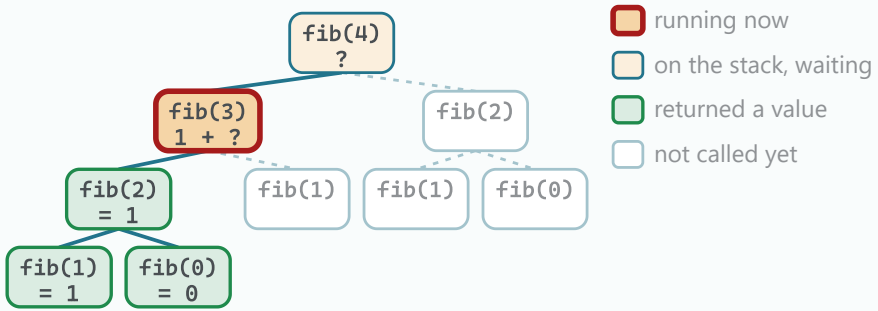
CALL STACK

- **fibonacci(2)**
- fibonacci(3)
- fibonacci(4)

VARIABLES

- `n = 2`
- `f(n-1) = 1`
- `f(n-2) = 0`
- `ret = 1`

`fibonacci(4)`: Call Tree and Stack



The left term gave 1. Now the **right** term: `fibonacci(1)` is called.

Step 8 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

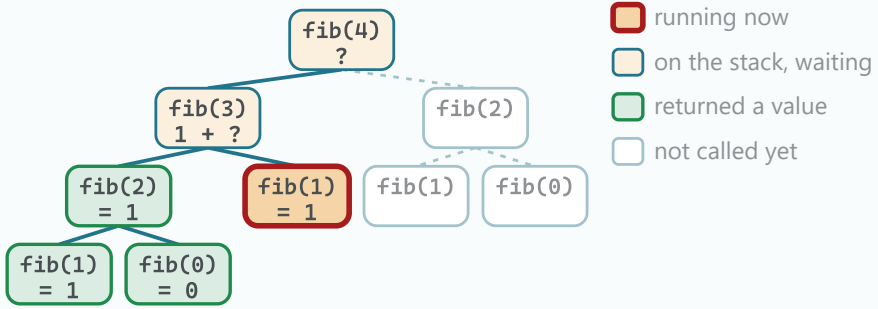
CALL STACK

- **fibonacci(3)**
- fibonacci(4)

VARIABLES

- n = **3**
- f(n−1) = **1**

`fibonacci(4)`: Call Tree and Stack



`fibonacci(1)` : `n <= 1` → base case, **1** is returned.

Step 9 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
▶           return n;
4       } else {
5           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

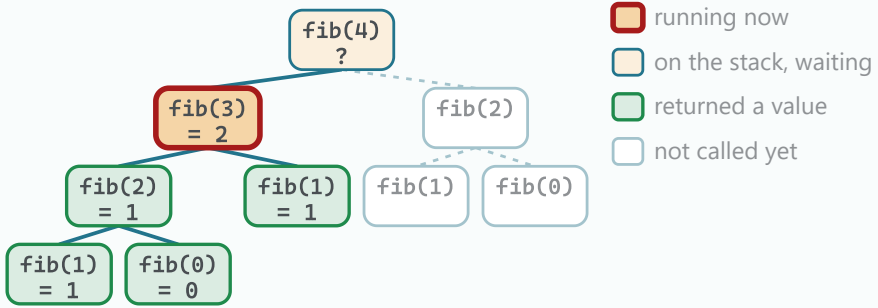
CALL STACK

- **fibonacci(1)**
- fibonacci(3)
- fibonacci(4)

VARIABLES

- `n = 1`
- `ret = 1`

`fibonacci(4)`: Call Tree and Stack



`1 + 1 = 2` is returned.

Step 10 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

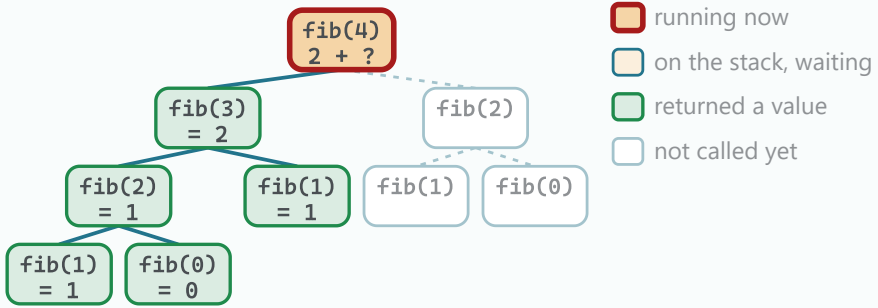
CALL STACK

- **fibonacci(3)**
- fibonacci(4)

VARIABLES

- `n = 3`
- `f(n-1) = 1`
- `f(n-2) = 1`
- `ret = 2`

`fibonacci(4)`: Call Tree and Stack



The left term gave 2. Now the **right** term: `fibonacci(2)` is called.

Step 11 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

CALL STACK

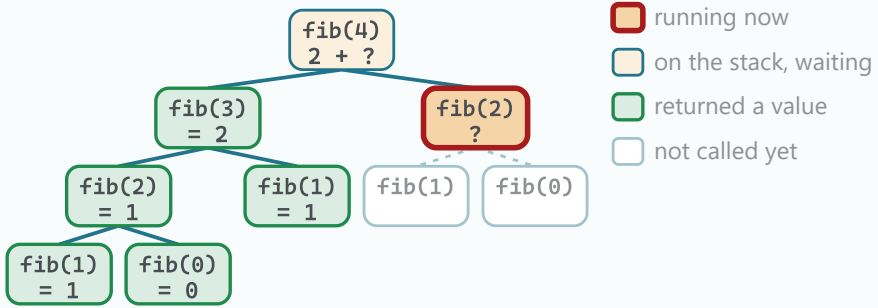
`fibonacci(4)`

VARIABLES

`n = 4`

`f(n-1) = 2`

`fibonacci(4)`: Call Tree and Stack



`fibonacci(2)`: n > 1. The **left** term of the sum comes first: `fibonacci(1)` is called.

Step 12 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
3           return n;
4       } else {
▶           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

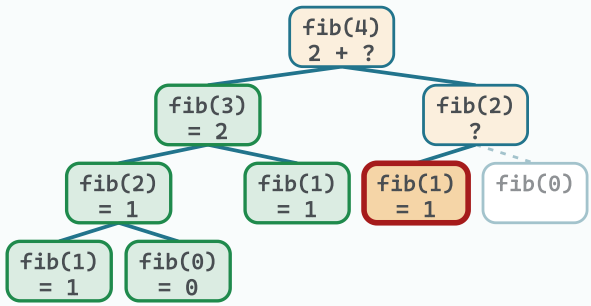
CALL STACK

- **`fibonacci(2)`**
- `fibonacci(4)`

VARIABLES

`n = 2`

`fibonacci(4)`: Call Tree and Stack



running now
on the stack, waiting
returned a value
not called yet

`fibonacci(1)` : `n <= 1` → base case, **1** is returned.

Step 13 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
▶           return n;
4       } else {
5           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

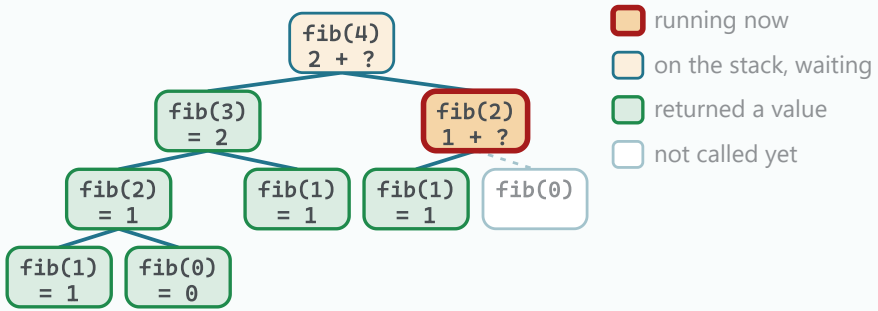
CALL STACK

- **`fibonacci(1)`**
- `fibonacci(2)`
- `fibonacci(4)`

VARIABLES

- `n = 1`
- `ret = 1`

`fibonacci(4)`: Call Tree and Stack



The left term gave 1. Now the **right** term: `fibonacci(0)` is called.

Step 14 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

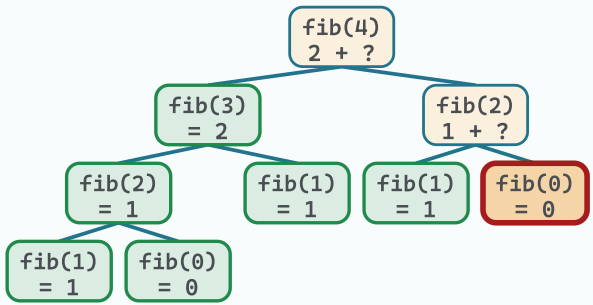
CALL STACK

- **fibonacci(2)**
- fibonacci(4)

VARIABLES

- `n = 2`
- `f(n-1) = 1`

`fibonacci(4)`: Call Tree and Stack



`fibonacci(0)` : `n <= 1` → base case, **0** is returned.

Step 15 / 17

```
1 static int fibonacci(int n) {
▶       if (n <= 1) {
▶           return n;
4       } else {
5           return fibonacci(n - 1) + fibonacci(n - 2);
6       }
7   }
```

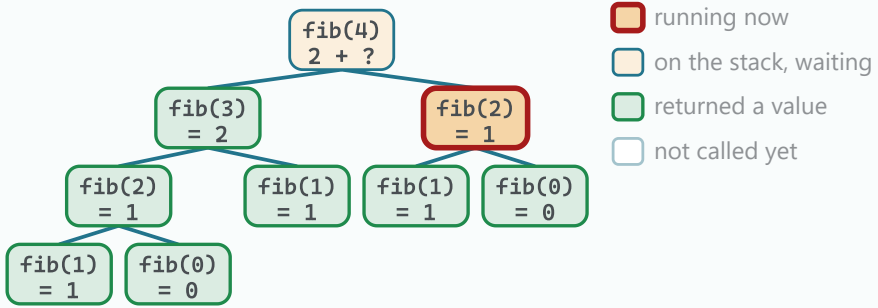
CALL STACK

- **`fibonacci(0)`**
- `fibonacci(2)`
- `fibonacci(4)`

VARIABLES

- `n = 0`
- `ret = 0`

`fibonacci(4)`: Call Tree and Stack



`1 + 0 = 1` is returned.

Step 16 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

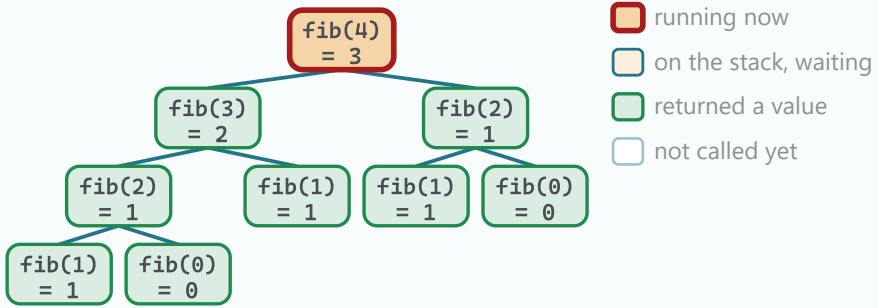
CALL STACK

- **fibonacci(2)**
- fibonacci(4)

VARIABLES

- `n = 2`
- `f(n-1) = 1`
- `f(n-2) = 0`
- `ret = 1`

`fibonacci(4)`: Call Tree and Stack



`2 + 1 = 3` is returned. In total **9 calls**; `fibonacci(2)` was computed twice.

Step 17 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
▶         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

CALL STACK

`fibonacci(4)`

VARIABLES

`n = 4`

`f(n-1) = 2`

`f(n-2) = 1`

`ret = 3`

SERIES

Lucas Series

- Starts with 2, then 1: **L(0) = 2, L(1) = 1**.
- Same general formula as Fibonacci: **L(n) = L(n − 1) + L(n − 2)**.
- Only the starting values differ → **two separate base cases** are needed.

```
int lucas(int n) {
    if (n == 0) {
        return 2;
    } else if (n == 1) {
        return 1;
    } else {
        return lucas(n - 1) + lucas(n - 2);
    }
}
```

n	0	1	2	3	4	5
L(n)	2	1	3	4	7	11
n	6	7	8	9	10	
L(n)	18	29	47	76	123	

The two series are linked: **L(n) = F(n − 1) + F(n + 1)**. E.g. L(5) = F(4) + F(6) = 3 + 8 = 11.

SERIES

Catalan Series

- Starts with 1: **C(0) = 1, C(1) = 1**.
- Closed form: **C(n) = (2n)! / ((n + 1)! · n!)**.
- Recursive formula used by the code: **C(n) = Σ C(i) · C(n − 1 − i)**, i = 0 … n − 1.
- The recursion runs **inside a loop**: each C(n) is built from all smaller C values.

```
long catalan(int n) {
    if (n <= 1) {
        return 1;
    }
    long result = 0;
    for (int i = 0; i < n; i++) {
        result += catalan(i) * catalan(n - i - 1);
    }
    return result;
}
```

Values:

`1` `1` `2` `5` `14` `42` `132` `429` `1430`

What does it count? The valid arrangements of n pairs of parentheses. n = 3 → C(3) = 5:

`((()))` `(()())` `(())()` `()(())` `()()()`

Also the ways to parenthesize a product of n + 1 numbers, and the binary trees with n nodes.

SERIES

Bell Series

- Starts with 1: **B(0) = 1, B(1) = 1**.
- General formula: **B(n) = Σ C(n − 1, k) · B(k)**, k = 0 … n − 1; here C(n, k) is the binomial coefficient.

```
int bell(int n) {
    if (n == 0) { return 1; }
    int count = 0;
    for (int k = 0; k < n; k++) {
        count += choose(n - 1, k) * bell(k);
    }
    return count;
}

int choose(int n, int k) {       // C(n, k)
    return factorial(n) / (factorial(k) * factorial(n - k));
}
```

Values:

`1` `1` `2` `5` `15` `52` `203` `877`

What does it count? The ways to partition a set of n elements into groups. {a, b, c} → B(3) = 5:

`abc` `a|bc` `b|ac` `c|ab` `a|b|c`

`choose` uses `factorial`, which returns `int`; 13! overflows, so the code is correct only for **n ≤ 13**.

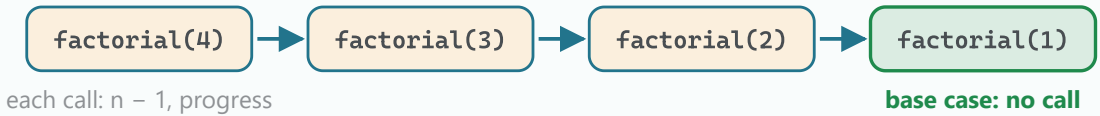
Two Basic Rules

Rule 1: Base case

At least one base case is needed to avoid infinite recursion. For example, when n is 0 or 1 the method must finish without calling itself.

Rule 2: Make progress

Every recursive call must **move toward** the base case: the parameter shrinks (or grows), the string gets shorter, the range narrows.



- **Infinite recursion** (no base case, or it is never reached) and **insufficient progress** (the parameter does not change) are the two most common errors.
- Each recursive call should work on a **smaller part** of the problem.
- The base case is always tested **before** the recursive call.
- Consider every possible input: do a negative number, 0 and an empty string reach the base case?

Infinite Recursion

```
public class Sum {
    static int sum(int n) {
        if (n == 1) {
            return 1;
        } else {
            return n + sum(n - 1);    // line 6
        }
    }
    public static void main(String[] args) {
        System.out.println(sum(0));
    }
}
```

```
Exception in thread "main" java.lang.StackOverflowError
        at Sum.sum(Sum.java:6)
        at Sum.sum(Sum.java:6)
        at Sum.sum(Sum.java:6)
        ...
```

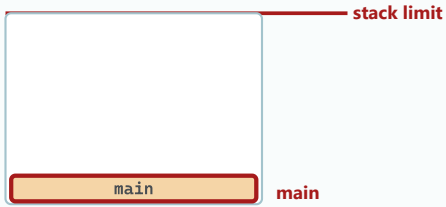
`sum(0)` : n = 0, −1, −2, … never reaches 1. Each call opens a new frame; after some ten thousand calls the stack is full and the JVM stops the program.

The compiler does **not catch** this error; it appears at run time. In the stack trace, **the same line repeating** is the sign of infinite recursion.

Fix: widen the base case (`n <= 1`), validate the input, or switch to a loop when the depth is very large.

`sum(0)`: How Does the Stack Fill Up?

```
1  public class Sum {
2      static int sum(int n) {
3          if (n == 1) {
4              return 1;
5          } else {
6              return n + sum(n - 1);    // line 6
7          }
8      }
9      public static void main(String[] args) {
▶          System.out.println(sum(0));
11     }
12 }
```



`main` calls `sum(0)`. The only base case is `n == 1`.

Step 1 / 6

VARIABLES

`n = –`

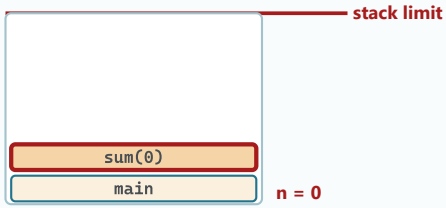
CALL STACK

`main()`

OUTPUT

`sum(0)`: How Does the Stack Fill Up?

```
1  public class Sum {
2      static int sum(int n) {
▶          if (n == 1) {
4              return 1;
5          } else {
▶              return n + sum(n - 1);   // line 6
7          }
8      }
9      public static void main(String[] args) {
10         System.out.println(sum(0));
11     }
12 }
```



`n = 0` : `0 == 1` is false → `0 + sum(-1)` must be computed: a new call.

Step 2 / 6

VARIABLES

`n = 0`

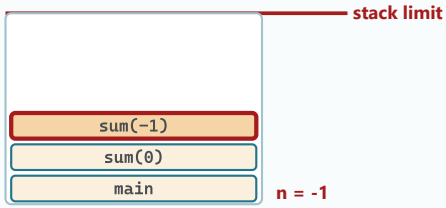
OUTPUT

CALL STACK

- **`sum(0)`**
- `main()`

`sum(0)`: How Does the Stack Fill Up?

```
1  public class Sum {
2      static int sum(int n) {
▶          if (n == 1) {
4              return 1;
5          } else {
▶              return n + sum(n - 1);   // line 6
7          }
8      }
9      public static void main(String[] args) {
10         System.out.println(sum(0));
11     }
12 }
```



`n = −1`: still not 1. With each call n moves **away**: the base case will never be reached.

Step 3 / 6

VARIABLES

`n = −1`

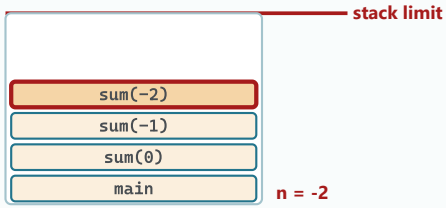
OUTPUT

CALL STACK

- `sum(−1)`
- `sum(0)`
- `main()`

`sum(0)`: How Does the Stack Fill Up?

```
1  public class Sum {
2      static int sum(int n) {
▶          if (n == 1) {
4              return 1;
5          } else {
▶              return n + sum(n - 1);   // line 6
7          }
8      }
9      public static void main(String[] args) {
10         System.out.println(sum(0));
11     }
12 }
```



Each call pushes a new frame; none can return, since each waits for the next one's result.

Step 4 / 6

VARIABLES

`n = −2`

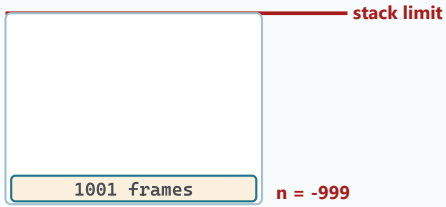
OUTPUT

CALL STACK

- `sum(−2)`
- `sum(−1)`
- `sum(0)`
- `main()`

`sum(0)`: How Does the Stack Fill Up?

```
1  public class Sum {
2      static int sum(int n) {
▶          if (n == 1) {
4              return 1;
5          } else {
▶              return n + sum(n - 1);   // line 6
7          }
8      }
9      public static void main(String[] args) {
10         System.out.println(sum(0));
11     }
12 }
```



After a thousand calls the stack holds 1001 frames; still no error.

Step 5 / 6

VARIABLES

`n = -999`

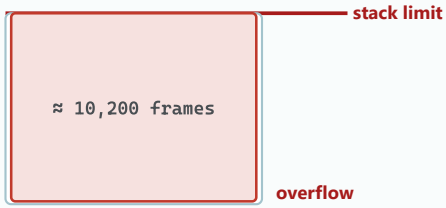
OUTPUT

CALL STACK

- `sum(-999)`
- …
- `sum(0)`
- `main()`

`sum(0)`: How Does the Stack Fill Up?

```
1  public class Sum {
2      static int sum(int n) {
3          if (n == 1) {
4              return 1;
5          } else {
▶              return n + sum(n - 1);   // line 6
7          }
8      }
9      public static void main(String[] args) {
10         System.out.println(sum(0));
11     }
12 }
```



The memory reserved for the stack is full: the JVM throws `StackOverflowError`. On this machine the measured depth is ≈ 10,200 calls; it depends on JVM settings.

Step 6 / 6

VARIABLES

—

OUTPUT

```
Exception in thread "main"
java.lang.StackOverflowError
        at Sum.sum(Sum.java:6)
        ...
```

CALL STACK

- `sum(-10200)`
- …
- `sum(0)`
- `main()`

Thinking Recursively

1. **Reduce:** express the problem in terms of a smaller copy of itself.
2. **Base case:** find the smallest input whose answer is known directly.
3. **Trust:** the smaller copy is **assumed** to be solved correctly (the leap of faith); do not unfold the call in your head.
4. **Combine:** build the big answer from the small one.

Example: the number of digits in a number.

- Digits of 48729 = 1 + digits of 4872.
- A one-digit number (n < 10) → 1.

```
int countDigits(int n) {
    if (n < 10) {                       // base case
        return 1;
    }
    return 1 + countDigits(n / 10);     // reduce + combine
}
```

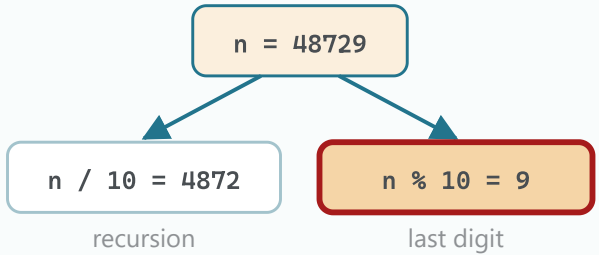
`countDigits(48729)` → **5** · `countDigits(7)` → **1**

EXAMPLES

Digit Sum

```
int digitSum(int n) {
    if (n < 0) {
        return digitSum(-n);
    } else if (n < 10) {
        return n;
    } else {
        return digitSum(n / 10) + (n % 10);
    }
}
```

- A negative number is first made positive (lines 2–3).
- A one-digit number is its own digit sum: base case (lines 4–5).
- Otherwise: last digit + sum of the remaining digits.
- For `Integer.MIN_VALUE`, `-n` is still negative: infinite recursion (compute with `long`).



`n / 10` (integer division) **drops the last digit**; `n % 10` **yields the last digit**. Each call shortens the number by one digit → progress is guaranteed.

EXAMPLES

Expanding digitSum(-48729)

```
digitSum(-48729)
  = digitSum(48729)
  = digitSum(4872) + 9
  = (digitSum(487) + 2) + 9
  = ((digitSum(48) + 7) + 2) + 9
  = (((digitSum(4) + 8) + 7) + 2) + 9
  = (((4 + 8) + 7) + 2) + 9
  = ((12 + 7) + 2) + 9
  = (19 + 2) + 9
  = 21 + 9
  = 30
```

The first call fixes the sign; after that each step leaves its last digit (9, 2, 7, 8) outside as a **pending addition**.

Base case `digitSum(4)` → 4. On the way back the additions run from the inside out: 4 + 8 + 7 + 2 + 9 = **30**.

6 calls in total (1 for the sign + 5 digits).

`digitSum(-357)`: Step-by-Step Code Trace



`main` calls `digitSum(-357)` (`r` for short in the figure).

Step 1 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
3          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
7          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
▶      System.out.println(digitSum(-357));
13 }
```

CALL STACK

`main()`

VARIABLES

`n = –`

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace

`r(-357)` `n < 0` | `r(357)` | `r(35)` | `r(3)`

`n = -357`: negative → first branch. The sign is fixed and `r(357)` is called; this frame will just pass the result back.

Step 2 / 9

```
1  static int digitSum(int n) {
▶      if (n < 0) {
▶          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
7          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
12     System.out.println(digitSum(-357));
13 }
```

CALL STACK

- `r(-357)`
- `main()`

VARIABLES

`n = -357`

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace



`n = 357` : not below 10. `357 / 10 = 35` is the next argument; the last digit **7** stays as a pending addition.

Step 3 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
3          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
▶          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
12     System.out.println(digitSum(-357));
13 }
```

CALL STACK

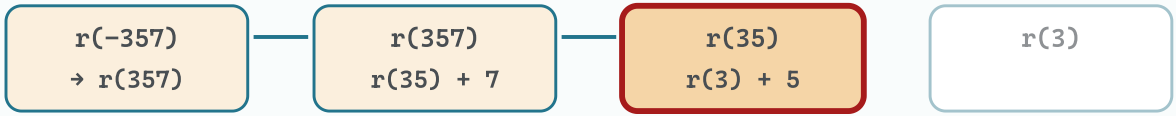
- r(357)
- r(-357)
- main()

VARIABLES

`n = 357`

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace



`n = 35` : `r(3)` is called; **5** waits.

Step 4 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
3          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
▶          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
12     System.out.println(digitSum(-357));
13 }
```

CALL STACK

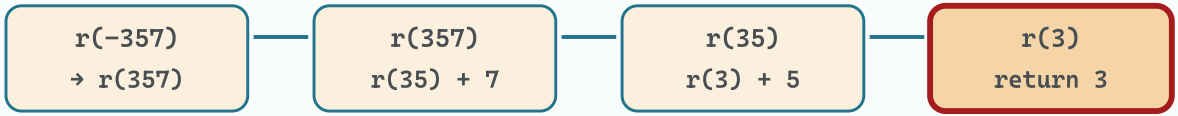
- **r(35)**
- r(357)
- r(-357)
- main()

VARIABLES

`n = 35`

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace



`n = 3 < 10` : **base case**, 3 is returned. The frames now start to return.

Step 5 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
3          return digitSum(-n);
▶      } else if (n < 10) {
▶          return n;
6      } else {
7          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
12     System.out.println(digitSum(-357));
13 }
```

CALL STACK

- r(3)
- r(35)
- r(357)
- r(-357)
- main()

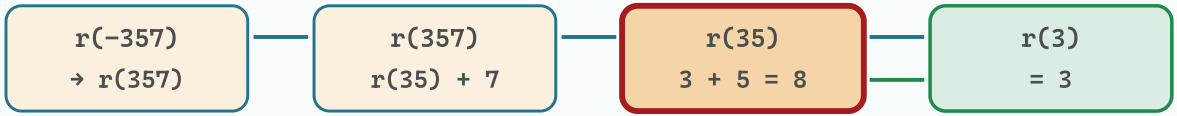
VARIABLES

`n = 3`

`ret = 3`

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace



`r(35)` got 3: `3 + 5 = 8` is returned.

Step 6 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
3          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
▶          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
12     System.out.println(digitSum(-357));
13 }
```

CALL STACK

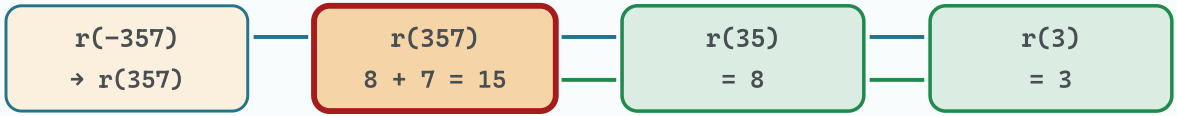
- r(35)
- r(357)
- r(-357)
- main()

VARIABLES

- n = 35
- ret = 8

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace



`r(357)` got 8: `8 + 7 = 15` is returned.

Step 7 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
3          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
▶          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
12     System.out.println(digitSum(-357));
13 }
```

CALL STACK

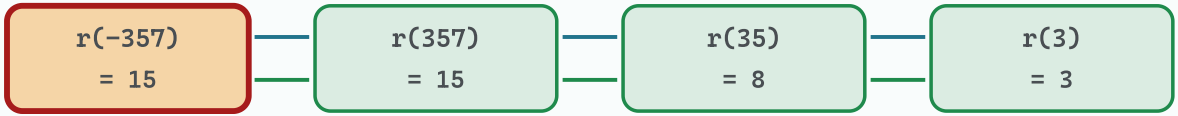
- `r(357)`
- `r(-357)`
- `main()`

VARIABLES

- `n = 357`
- `ret = 15`

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace



`r(-357)` returns the 15 unchanged.

Step 8 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
▶          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
7          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
12     System.out.println(digitSum(-357));
13 }
```

CALL STACK

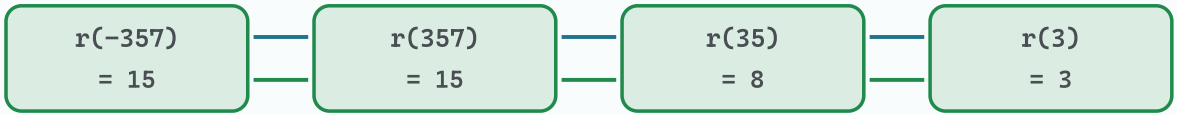
- `r(-357)`
- `main()`

VARIABLES

- `n = -357`
- `ret = 15`

OUTPUT

`digitSum(-357)`: Step-by-Step Code Trace



15 is printed: 3 + 5 + 7 = 15. Four calls were made (1 for the sign + 3 digits).

Step 9 / 9

```
1  static int digitSum(int n) {
2      if (n < 0) {
3          return digitSum(-n);
4      } else if (n < 10) {
5          return n;
6      } else {
7          return digitSum(n / 10) + (n % 10);
8      }
9  }
10
11 public static void main(String[] args) {
▶      System.out.println(digitSum(-357));
13 }
```

CALL STACK

`main()`

VARIABLES

`n = -`

OUTPUT

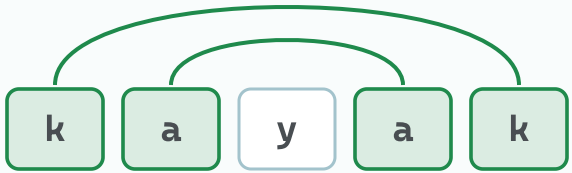
```
15
```

EXAMPLES

Checking a Palindrome

```
static boolean isPalindrome(String s) {
    if (s.length() <= 1) {
        return true;
    } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
        return isPalindrome(s.substring(1, s.length() - 1));
    }
    return false;
}
```

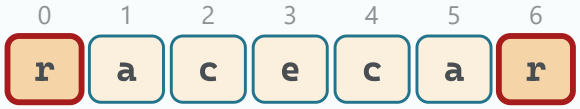
`isPalindrome("kayak")` → **true** · `isPalindrome("abba")` → **true** · `isPalindrome("java")` → **false**



outside in: k = k, a = a; y remains

- **Palindrome**: a string that reads the same backward (kayak, never odd or even).
- `s.charAt(i)`: the character at index i (from 0).
- `s.substring(1, s.length() - 1)`: the string without its first and last letters.
- Two base cases: length ≤ 1 → **true**; ends differ → **false**.
- Case-sensitive: `"Kayak"` → false.

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "racecar" (length 7)`

s = "racecar": length 7 > 1. The first letter `'r'` is compared with the last letter `'r'` → **equal**.

Step 1 / 10

```
1 static boolean isPalindrome(String s) {
▶     if (s.length() <= 1) {
3         return true;
▶     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
5         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

CALL STACK

`isPalindrome("racecar")`

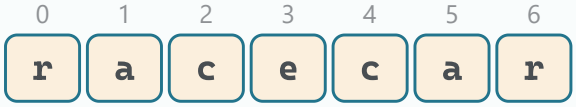
VARIABLES

`s = "racecar"`

`s.charAt(0) = 'r'`

`last = 'r'`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "racecar" (length 7)`

The ends match; the inner part `s.substring(1, 6)` = `"aceca"` is checked by a new call.

Step 2 / 10

```
1 static boolean isPalindrome(String s) {
2     if (s.length() <= 1) {
3         return true;
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
▶         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

CALL STACK

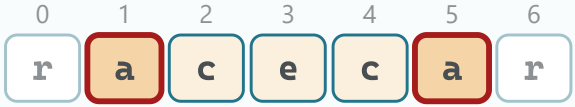
`isPalindrome("racecar")`

VARIABLES

`s = "racecar"`

`substring = "aceca"`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "aceca" (length 5)`

s = "aceca": length 5 > 1. The first letter `'a'` is compared with the last letter `'a'` → **equal**.

Step 3 / 10

```
1 static boolean isPalindrome(String s) {
▶     if (s.length() <= 1) {
3         return true;
▶     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
5         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

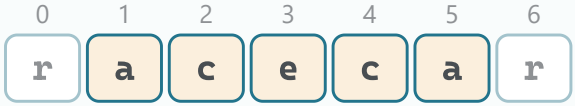
CALL STACK

- **isPalindrome("aceca")**
- isPalindrome("racecar")

VARIABLES

- `s = "aceca"`
- `s.charAt(0) = 'a'`
- `last = 'a'`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "aceca" (length 5)`

The ends match; the inner part `s.substring(1, 4)` = `"cec"` is checked by a new call.

Step 4 / 10

```
1 static boolean isPalindrome(String s) {
2     if (s.length() <= 1) {
3         return true;
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
▶         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

CALL STACK

- `isPalindrome("aceca")`
- `isPalindrome("racecar")`

VARIABLES

- `s = "aceca"`
- `substring = "cec"`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "cec" (length 3)`

s = "cec": length 3 > 1. The first letter `'c'` is compared with the last letter `'c'` → **equal**.

Step 5 / 10

```
1 static boolean isPalindrome(String s) {
▶     if (s.length() <= 1) {
3         return true;
▶     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
5         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

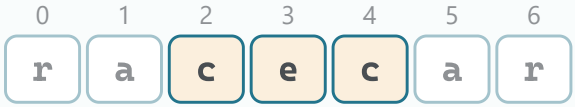
CALL STACK

- **isPalindrome("cec")**
- isPalindrome("aceca")
- isPalindrome("racecar")

VARIABLES

- `s = "cec"`
- `s.charAt(0) = 'c'`
- `last = 'c'`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "cec" (length 3)`

The ends match; the inner part `s.substring(1, 2)` = `"e"` is checked by a new call.

Step 6 / 10

```
1 static boolean isPalindrome(String s) {
2     if (s.length() <= 1) {
3         return true;
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
▶         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

CALL STACK

- **isPalindrome("cec")**
- isPalindrome("aceca")
- isPalindrome("racecar")

VARIABLES

`s = "cec"` `substring = "e"`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "e" (length 1)`

Length 1 ≤ 1 → **base case**: a one-letter string is a palindrome, **true** is returned.

Step 7 / 10

```
1 static boolean isPalindrome(String s) {
▶     if (s.length() <= 1) {
▶         return true;
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
5         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

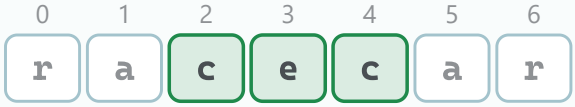
CALL STACK

- **isPalindrome("e")**
- isPalindrome("cec")
- isPalindrome("aceca")
- isPalindrome("racecar")

VARIABLES

`s = "e"` `s.length() = 1` `ret = true`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "cec" (length 3)`

The `isPalindrome("cec")` frame gets true and returns **true**.

Step 8 / 10

```
1 static boolean isPalindrome(String s) {
2     if (s.length() <= 1) {
3         return true;
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
▶         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

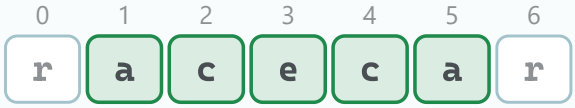
CALL STACK

- **isPalindrome("cec")**
- isPalindrome("aceca")
- isPalindrome("racecar")

VARIABLES

`s = "cec"` `ret = true`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "aceca" (length 5)`

The `isPalindrome("aceca")` frame gets true and returns **true**.

Step 9 / 10

```
1 static boolean isPalindrome(String s) {
2     if (s.length() <= 1) {
3         return true;
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
▶         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

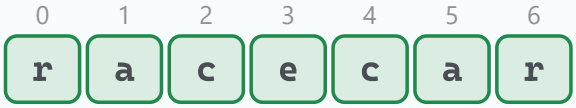
CALL STACK

- `isPalindrome("aceca")`
- `isPalindrome("racecar")`

VARIABLES

`s = "aceca"` `ret = true`

`isPalindrome("racecar")`: Step-by-Step Code Trace



`s = "racecar" (length 7)`

The `isPalindrome("racecar")` frame gets true and returns **true**. Result: "racecar" is a **palindrome**.

Step 10 / 10

```
1 static boolean isPalindrome(String s) {
2     if (s.length() <= 1) {
3         return true;
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
▶         return isPalindrome(s.substring(1, s.length() -
  1));
6     }
7     return false;
8 }
```

CALL STACK

`isPalindrome("racecar")`

VARIABLES

`s = "racecar"` `ret = true`

EXAMPLES

Greatest Common Divisor

```java
static int gcd(int x, int y) {
    if (y == 0) {
        return x;
    }
    return gcd(y, x % y);
}
```

- **Euclid's algorithm**: gcd(x, y) = gcd(y, x mod y).
- Base case: if y = 0, then gcd = x.
- The remainder shrinks at every step → it must reach 0.
- `gcd(93, 444)`: its first step is `gcd(444, 93)`; the order does not matter.

```
gcd(444, 93)
  = gcd(93, 72)      // 444 % 93 = 72
  = gcd(72, 21)      // 93 % 72 = 21
  = gcd(21, 9)       // 72 % 21 = 9
  = gcd(9, 3)        // 21 % 9 = 3
  = gcd(3, 0)        // 9 % 3 = 0
  = 3
```

gcd: greatest common divisor. `gcd(48, 18)` → 6.

`gcd(444, 93)`: Step-by-Step Code Trace

```
1 static int gcd(int x, int y) {
▶       if (y == 0) {
3           return x;
4       }
▶       return gcd(y, x % y);
6   }
```

call	x	y	x % y
`gcd(444, 93)`	444	93	72

`y = 93 ≠ 0`. Remainder `444 % 93 = 72`; new call `gcd(93, 72)`.

Step 1 / 7

CALL STACK

`gcd(444, 93)`

VARIABLES

`x = 444` `y = 93` `x % y = 72`

`gcd(444, 93)`: Step-by-Step Code Trace

```
1 static int gcd(int x, int y) {
▶       if (y == 0) {
3           return x;
4       }
▶       return gcd(y, x % y);
6   }
```

call	x	y	x % y
`gcd(444, 93)`	444	93	72
`gcd(93, 72)`	**93**	**72**	**21**

`y = 72 ≠ 0`. Remainder `93 % 72 = 21`; new call `gcd(72, 21)`.

Step 2 / 7

CALL STACK

- **`gcd(93, 72)`**
- `gcd(444, 93)`

VARIABLES

`x = 93` `y = 72` `x % y = 21`

`gcd(444, 93)`: Step-by-Step Code Trace

```
1 static int gcd(int x, int y) {
▶       if (y == 0) {
3           return x;
4       }
▶       return gcd(y, x % y);
6   }
```

call	x	y	x % y
`gcd(444, 93)`	444	93	72
`gcd(93, 72)`	93	72	21
`gcd(72, 21)`	**72**	**21**	**9**

`y = 21 ≠ 0`. Remainder `72 % 21 = 9`; new call `gcd(21, 9)`.

Step 3 / 7

CALL STACK

- **`gcd(72, 21)`**
- `gcd(93, 72)`
- `gcd(444, 93)`

VARIABLES

`x = 72` `y = 21` `x % y = 9`

`gcd(444, 93)`: Step-by-Step Code Trace

```
1 static int gcd(int x, int y) {
▶       if (y == 0) {
3           return x;
4       }
▶       return gcd(y, x % y);
6   }
```

call	x	y	x % y
`gcd(444, 93)`	444	93	72
`gcd(93, 72)`	93	72	21
`gcd(72, 21)`	72	21	9
`gcd(21, 9)`	**21**	**9**	**3**

`y = 9 ≠ 0`. Remainder `21 % 9 = 3`; new call `gcd(9, 3)`.

Step 4 / 7

CALL STACK

- **`gcd(21, 9)`**
- `gcd(72, 21)`
- `gcd(93, 72)`
- `gcd(444, 93)`

VARIABLES

`x = 21` `y = 9` `x % y = 3`

`gcd(444, 93)`: Step-by-Step Code Trace

```
1 static int gcd(int x, int y) {
▶       if (y == 0) {
3           return x;
4       }
▶       return gcd(y, x % y);
6   }
```

call	x	y	x % y
`gcd(444, 93)`	444	93	72
`gcd(93, 72)`	93	72	21
`gcd(72, 21)`	72	21	9
`gcd(21, 9)`	21	9	3
`gcd(9, 3)`	**9**	**3**	**0**

`y = 3 ≠ 0`. Remainder `9 % 3 = 0`; new call `gcd(3, 0)`.

Step 5 / 7

CALL STACK

- **`gcd(9, 3)`**
- `gcd(21, 9)`
- `gcd(72, 21)`
- `gcd(93, 72)`
- `gcd(444, 93)`

VARIABLES

`x = 9` `y = 3` `x % y = 0`

`gcd(444, 93)`: Step-by-Step Code Trace

```
1 static int gcd(int x, int y) {
▶       if (y == 0) {
▶           return x;
4       }
5       return gcd(y, x % y);
6   }
```

call	x	y	x % y
`gcd(444, 93)`	444	93	72
`gcd(93, 72)`	93	72	21
`gcd(72, 21)`	72	21	9
`gcd(21, 9)`	21	9	3
`gcd(9, 3)`	9	3	0
`gcd(3, 0)`	**3**	**0**	—

`y == 0` → **base case**: `x = 3` is returned. gcd(444, 93) = **3**.

Step 6 / 7

CALL STACK

- **`gcd(3, 0)`**
- `gcd(9, 3)`
- `gcd(21, 9)`
- `gcd(72, 21)`
- `gcd(93, 72)`
- `gcd(444, 93)`

VARIABLES

`x = 3` `y = 0` `ret = 3`

`gcd(444, 93)`: Step-by-Step Code Trace

```
1 static int gcd(int x, int y) {
2     if (y == 0) {
3         return x;
4     }
▶     return gcd(y, x % y);
6 }
```

call	x	y	x % y
`gcd(444, 93)`	444	93	72
`gcd(93, 72)`	93	72	21
`gcd(72, 21)`	72	21	9
`gcd(21, 9)`	21	9	3
`gcd(9, 3)`	9	3	0
`gcd(3, 0)`	3	0	—

After `return gcd(…)` there is nothing left to do: the value 3 is passed back **unchanged** through every frame (5 returns). `main` receives **3** as the result.

Step 7 / 7

CALL STACK

`gcd(444, 93)`

VARIABLES

`ret = 3`

EXAMPLES

Towers of Hanoi

Three pegs (A, B, C) and n disks on A, largest at the bottom. Move them all to C:

- Move **only one** disk at a time, always the top one.
- A larger disk is **never** placed on a smaller one.

Recursive idea: move the top n − 1 disks to the spare peg, the largest disk to the target, then the n − 1 disks from the spare onto the target.

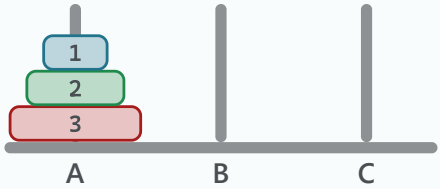
Number of moves $\mathbf{2^n - 1}$: 3 disks → 7, 10 disks → 1023, 64 disks → about 1.8×10^{19}.

```
public class TowersOfHanoi {
    static void solve(int n, char from,
                      char to, char via) {
        if (n == 1) {   // one disk: move it
            System.out.println(n + ": " + from + "->" + to);
        } else {
            solve(n - 1, from, via, to);
            System.out.println(n + ": " + from + "->" + to);
            solve(n - 1, via, to, from);
        }
    }
    public static void main(String[] args) {
        solve(3, 'A', 'C', 'B');
    }
}
```

Hard to write with loops, this problem takes only a few lines with recursion.

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
4          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
7          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



Start: 3 disks on A. `solve(3, 'A', 'C', 'B')`: move three disks to C, using B as the spare.

Step 1 / 8

OUTPUT

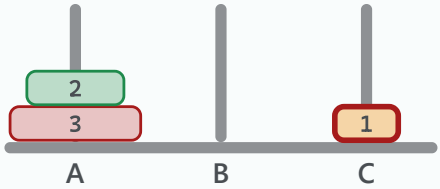
CALL STACK

`solve(3, A, C, B)`

`main()`

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
▶          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
7          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



The deepest call `solve(1, A, C, B)`: base case, disk 1 goes straight from A to C.

OUTPUT

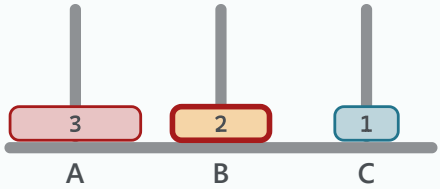
```
1: A->C
```

CALL STACK

- **`solve(1, A, C, B)`**
- `solve(2, A, B, C)`
- `solve(3, A, C, B)`
- `main()`

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
4          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
▶          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



`solve(2, A, B, C)` : disk 1 is out of the way; disk 2 goes from A to B.

Step 3 / 8

OUTPUT

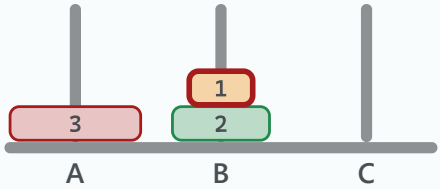
```
1: A->C
2: A->B
```

CALL STACK

- **`solve(2, A, B, C)`**
- `solve(3, A, C, B)`
- `main()`

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
▶          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
7          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



Disk 1 goes on top of disk 2: `solve(1, C, B, A)`. The top 2 disks are now on B.

Step 4 / 8

OUTPUT

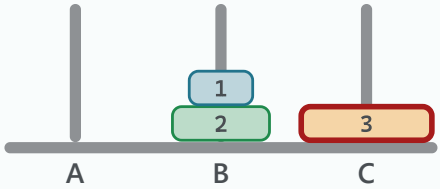
```
1: A->C
2: A->B
1: C->B
```

CALL STACK

- `solve(1, C, B, A)`
- `solve(2, A, B, C)`
- `solve(3, A, C, B)`
- `main()`

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
4          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
▶          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



`solve(3, …)` reaches its middle line: the largest disk (3) goes from A to C.

Step 5 / 8

OUTPUT

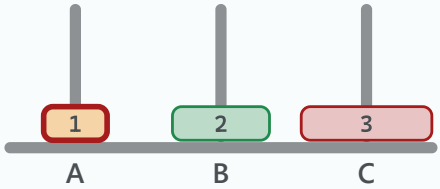
```
1: A->C
2: A->B
1: C->B
3: A->C
```

CALL STACK

- `solve(3, A, C, B)`
- `main()`

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
▶          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
7          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



Now the 2 disks on B go to C: `solve(2, B, C, A)` → first, disk 1 goes from B to A.

Step 6 / 8

OUTPUT

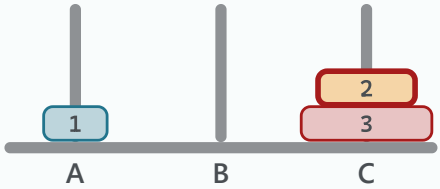
```
1: A->C
2: A->B
1: C->B
3: A->C
1: B->A
```

CALL STACK

- **`solve(1, B, A, C)`**
- `solve(2, B, C, A)`
- `solve(3, A, C, B)`
- `main()`

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
4          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
▶          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



Disk 2 goes from B to C, on top of disk 3.

Step 7 / 8

OUTPUT

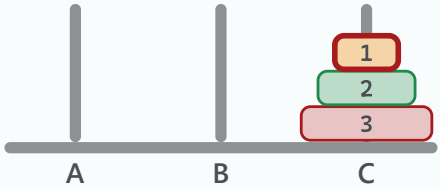
```
1: A->C
2: A->B
1: C->B
3: A->C
1: B->A
2: B->C
```

CALL STACK

- `solve(2, B, C, A)`
- `solve(3, A, C, B)`
- `main()`

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

```
1  static void solve(int n, char from,
2                    char to, char via) {
3      if (n == 1) {
▶          System.out.println(n + ": " + from + "->" + to);
5      } else {
6          solve(n - 1, from, via, to);
7          System.out.println(n + ": " + from + "->" + to);
8          solve(n - 1, via, to, from);
9      }
10 }
```



Last move: disk 1 from A to C. $2^3 - 1 = 7$ **7 moves**; the puzzle is solved.

Step 8 / 8

OUTPUT

```
1: A->C
2: A->B
1: C->B
3: A->C
1: B->A
2: B->C
1: A->C
```

CALL STACK

- `solve(1, A, C, B)`
- `solve(2, B, C, A)`
- `solve(3, A, C, B)`
- `main()`

Efficiency

- Recursive methods can sometimes be **slow**: every call has a cost (opening a frame, copying parameters).
- As recursive calls are made, the system keeps the pending **partial results** on the call stack.
- If the stack overflows, the program ends with `StackOverflowError`.
- When this is a problem, the fix is an **iterative** solution built on loops.
- Iterative solutions keep no partial results: one frame and a few variables suffice.

Memory

`factorial(n)`, recursive: n frames, O(n) memory. With a loop: O(1).

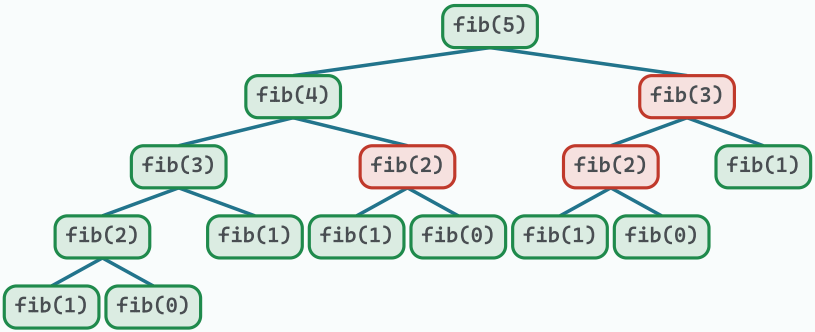
Depth limit

With the default stack, a depth of a few tens of thousands of calls gives `StackOverflowError`.

Repetition

Tree recursion (Fibonacci) may compute the same values many times.

Repeated Computation



`fibonacci(5)`: 15 calls. Red nodes are values computed **again**, although they are already known.

n	calls	F(n)
10	177	55
20	21,891	6,765
30	2,692,537	832,040
40	331,160,281	102,334,155

When n grows by 1, the number of calls grows ≈ 1.6×: **exponential** growth.
`fibonacci(40)` takes half a second;
`fibonacci(50)` takes minutes.

EFFICIENCY

Factorial with a Loop

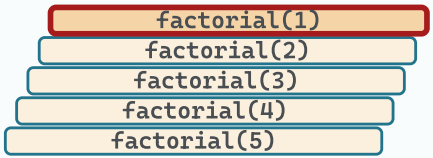
RECURSIVE

```
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

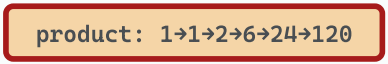
ITERATIVE (LOOP)

```
int factorial(int n) {
    int product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

recursive: 5 frames



loop: one frame



Same result (`factorial(5)` → 120). The loop version uses one frame and constant memory; the variable `product` **accumulates the product**. `n` ≥ 0 is assumed: for negative n the recursive version throws `StackOverflowError`, the loop version returns 1; 13! overflows `int`.

EFFICIENCY

Digit Sum and Palindrome with Loops

DIGIT SUM

```
int digitSum(int n) {
    int sum = 0;
    int rest = Math.abs(n);
    while (rest > 0) {
        sum += rest % 10;
        rest /= 10;
    }
    return sum;
}
```

`digitSum(-48729)` → **30**: same as the recursive version. A digit sum cannot be negative; the sign is simply dropped with `Math.abs` (for `Integer.MIN_VALUE`, `Math.abs` stays negative; `long rest = Math.abs((long) n)` fixes it).

PALINDROME

```
boolean isPalindrome(String s) {
    s = s.toLowerCase();
    int len = s.length();
    for (int i = 0; i < len / 2; i++) {
        if (s.charAt(i) != s.charAt(len - 1 - i)) {
            return false;
        }
    }
    return true;
}
```

i moves from the start, `len - 1 - i` from the end. `toLowerCase()` makes `"Kayak"` return true. Prefer `toLowerCase(Locale.ROOT)`: in the default `tr` locale, `I` → `ı`.

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
4      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
▶      System.out.println(isPalindrome("Kepek"));
14 }
```

isPalindrome("Kepek")

`main` calls `isPalindrome("Kepek")`.

Step 1 / 11

VARIABLES

`s = –`

CALL STACK

`main()`

OUTPUT

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
▶       s = s.toLowerCase();
3       int len = s.length();
4       for (int i = 0; i < len / 2; i++) {
5           if (s.charAt(i) != s.charAt(len - 1 - i)) {
6               return false;
7           }
8       }
9       return true;
10 }
11
12 public static void main(String[] args) {
13      System.out.println(isPalindrome("Kepek"));
14 }
```



Parameter `s = "Kepek"` : the first letter is uppercase.

Step 2 / 11

VARIABLES

`s = "Kepek"`

OUTPUT

CALL STACK

`isPalindrome("Kepek")`

`main()`

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
▶      s = s.toLowerCase();
3      int len = s.length();
4      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



`toLowerCase()` returns a new string: `s = "kepek"`. K and k are now the same.

Step 3 / 11

VARIABLES

`s = "kepek"`

CALL STACK

`isPalindrome("Kepek")`

`main()`

OUTPUT

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
▶      int len = s.length();
4      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



`len = 5`; the loop runs while `i < 5 / 2 = 2`.

Step 4 / 11

VARIABLES

`s = "kepek"` `len = 5`

OUTPUT

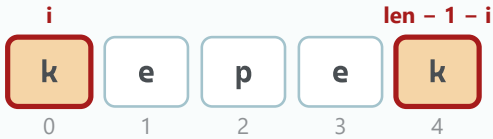
CALL STACK

`isPalindrome("Kepek")`

`main()`

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
▶      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



`i = 0` : index 0 from the start and index `5 − 1 − 0 = 4` from the end are compared.

Step 5 / 11

VARIABLES

`s = "kepek"` `len = 5` `i = 0`

CALL STACK

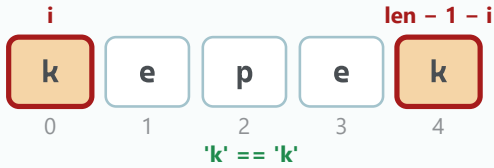
`isPalindrome("Kepek")`

`main()`

OUTPUT

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
4      for (int i = 0; i < len / 2; i++) {
▶          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



`'k' == 'k'` : equal, so `return false` is not executed.

Step 6 / 11

VARIABLES

`s = "kepek"` `len = 5` `i = 0`

OUTPUT

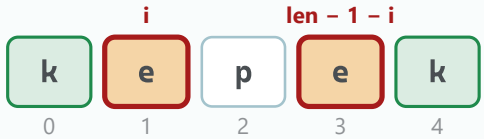
CALL STACK

`isPalindrome("Kepek")`

`main()`

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
▶      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



`i = 1` : indices 1 and 3.

Step 7 / 11

VARIABLES

`s = "kepek"` `len = 5` `i = 1`

CALL STACK

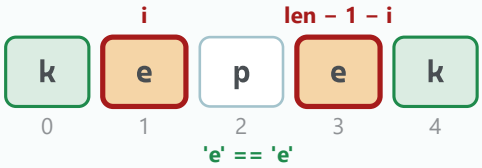
`isPalindrome("Kepek")`

`main()`

OUTPUT

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
4      for (int i = 0; i < len / 2; i++) {
▶          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



`'e' == 'e'` : equal.

Step 8 / 11

VARIABLES

`s = "kepek"` `len = 5` `i = 1`

CALL STACK

`isPalindrome("Kepek")`

`main()`

OUTPUT

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
▶      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



`i = 2` : `2 < 2` is false, the loop ends. The middle `'p'` matches itself; no need to check it.

Step 9 / 11

VARIABLES

`s = "kepek"` `len = 5` `i = 2`

CALL STACK

`isPalindrome("Kepek")`

`main()`

OUTPUT

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
4      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
▶      return true;
10 }
11
12 public static void main(String[] args) {
13     System.out.println(isPalindrome("Kepek"));
14 }
```



No mismatched pair found: `true` is returned.

Step 10 / 11

VARIABLES

`s = "kepek"` `len = 5` `ret = true`

CALL STACK

`isPalindrome("Kepek")`

`main()`

OUTPUT

Palindrome with a Loop: Step by Step

```
1  static boolean isPalindrome(String s) {
2      s = s.toLowerCase();
3      int len = s.length();
4      for (int i = 0; i < len / 2; i++) {
5          if (s.charAt(i) != s.charAt(len - 1 - i)) {
6              return false;
7          }
8      }
9      return true;
10 }
11
12 public static void main(String[] args) {
▶      System.out.println(isPalindrome("Kepek"));
14 }
```

`isPalindrome("Kepek")`

`true` is printed. Only one frame was used, and no new string was created at each step as in the recursive version.

Step 11 / 11

VARIABLES

`s = –`

CALL STACK

`main()`

OUTPUT

```
true
```

EFFICIENCY

Recursion or Loop?

	Recursion	Loop (iterative)
Stopping	Base case	Loop condition
Progress	Call with a smaller argument	Update a counter / variable
Memory	One frame per call: O(depth)	Usually O(1)
Speed	Call overhead; repetition in tree recursion	Usually faster
Error	No base case → `StackOverflowError`	Broken condition: infinite loop (program hangs)
Readability	Trees, divide and conquer, Hanoi, series: very natural	Natural for simple counting and accumulating

Any recursive solution can be written with a loop (keeping its own stack if needed), and vice versa. Choose by clarity, depth and performance.

Tail Recursion

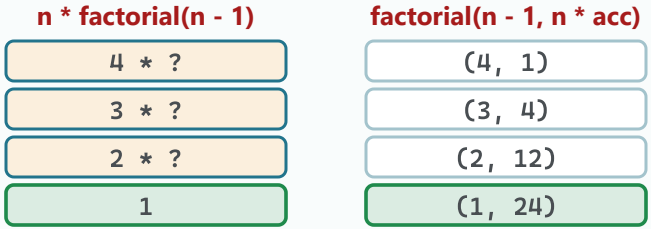
- The method's **last operation** is the recursive call itself; nothing is left to do on return.
- The result is carried in an extra **accumulator** parameter.
- This form lets the compiler apply **tail call optimization** (TCO): the call is treated like a loop and frames do not pile up → memory efficiency.
- **Java (the JVM) does not do this**: tail-recursive Java code still uses O(n) stack. Languages such as Scheme, Scala (`@tailrec`), Kotlin (`tailrec`) do it.

NOT TAIL-RECURSIVE

```
return n * factorial(n - 1);
// multiply AFTER the call returns
```

TAIL-RECURSIVE

```
return factorial(n - 1, n * acc);
// multiply first; the call is last
```



Tail-Recursive Examples

```
int digitSum(int num, int sum) {
    if (num == 0) {
        return sum;
    } else {
        return digitSum(num / 10,
                        sum + num % 10);
    }
}
```

Digit sum: the last digit is added to `sum` before the call. `digitSum(48729, 0)` → **30** (for n ≥ 0).

```
int factorial(int n, int acc) {
    if (n == 0) {
        return acc;
    } else {
        return factorial(n - 1, n * acc);
    }
}
```

Factorial: the product goes into `acc` before the call. `factorial(5, 1)` → **120**.

The accumulator starts at the identity element of the operation: for addition **0**, for multiplication **1**. A wrapper method spares the caller from passing it: `int factorial(int n) { return factorial(n, 1); }`

`factorial(4, 1)`: Tracing Tail Recursion

`factorial(4, 1)` last step: call | `factorial(3, 4)` | `factorial(2, 12)` | `factorial(1, 24)` | `factorial(0, 24)`

The multiplication happens **before the call**: `4 * 1 = 4` becomes the new acc; call `factorial(3, 4)`.

Step 1 / 9

```
1 static int factorial(int n, int acc) {
▶     if (n == 0) {
3         return acc;
4     } else {
▶         return factorial(n - 1, n * acc);
6     }
7 }
```

CALL STACK

`factorial(4, 1)`

VARIABLES

`n = 4` `acc = 1` `n * acc = 4`

`factorial(4, 1)`: Tracing Tail Recursion

`factorial(4, 1)` last step: call → `factorial(3, 4)` last step: call → `factorial(2, 12)` → `factorial(1, 24)` → `factorial(0, 24)`

The multiplication happens **before the call**: `3 * 4 = 12` becomes the new acc; call `factorial(2, 12)`.

Step 2 / 9

```
1 static int factorial(int n, int acc) {
▶       if (n == 0) {
3           return acc;
4       } else {
▶           return factorial(n - 1, n * acc);
6       }
7   }
```

CALL STACK

- **`factorial(3, 4)`**
- `factorial(4, 1)`

VARIABLES

`n = 3` `acc = 4` `n * acc = 12`

`factorial(4, 1)`: Tracing Tail Recursion

`factorial(4, 1)` last step: call — `factorial(3, 4)` last step: call — **`factorial(2, 12)` last step: call** — `factorial(1, 24)` — `factorial(0, 24)`

The multiplication happens **before the call**: `2 * 12 = 24` becomes the new acc; call `factorial(1, 24)`.

Step 3 / 9

```
1 static int factorial(int n, int acc) {
▶       if (n == 0) {
3           return acc;
4       } else {
▶           return factorial(n - 1, n * acc);
6       }
7   }
```

CALL STACK

- **`factorial(2, 12)`**
- `factorial(3, 4)`
- `factorial(4, 1)`

VARIABLES

`n = 2` `acc = 12` `n * acc = 24`

`factorial(4, 1)`: Tracing Tail Recursion

`factorial(4, 1)` last step: call — `factorial(3, 4)` last step: call — `factorial(2, 12)` last step: call — **`factorial(1, 24)` last step: call** — `factorial(0, 24)`

The multiplication happens **before the call**: `1 * 24 = 24` becomes the new acc; call `factorial(0, 24)`.

Step 4 / 9

```
1 static int factorial(int n, int acc) {
▶       if (n == 0) {
3           return acc;
4       } else {
▶           return factorial(n - 1, n * acc);
6       }
7   }
```

CALL STACK

- **`factorial(1, 24)`**
- `factorial(2, 12)`
- `factorial(3, 4)`
- `factorial(4, 1)`

VARIABLES

`n = 1` `acc = 24` `n * acc = 24`

`factorial(4, 1)`: Tracing Tail Recursion

`factorial(4, 1)` last step: call → `factorial(3, 4)` last step: call → `factorial(2, 12)` last step: call → `factorial(1, 24)` last step: call → `factorial(0, 24)` return 24

`n == 0`: base case. The result is already in `acc`: **24** is returned.

Step 5 / 9

```
1 static int factorial(int n, int acc) {
▶     if (n == 0) {
▶         return acc;
4     } else {
5         return factorial(n - 1, n * acc);
6     }
7 }
```

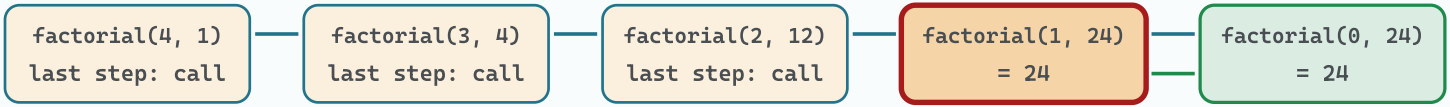
CALL STACK

- **factorial(0, 24)**
- factorial(1, 24)
- factorial(2, 12)
- factorial(3, 4)
- factorial(4, 1)

VARIABLES

`n = 0` `acc = 24` `ret = 24`

`factorial(4, 1)`: Tracing Tail Recursion



Nothing is computed on the way back; 24 is passed up as is.

Step 6 / 9

```
1 static int factorial(int n, int acc) {
2     if (n == 0) {
3         return acc;
4     } else {
▶         return factorial(n - 1, n * acc);
6     }
7 }
```

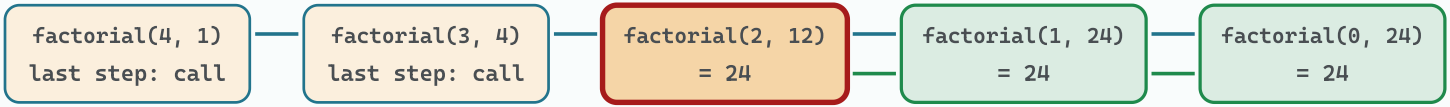
CALL STACK

- **`factorial(1, 24)`**
- `factorial(2, 12)`
- `factorial(3, 4)`
- `factorial(4, 1)`

VARIABLES

`n = 1` `ret = 24`

`factorial(4, 1)`: Tracing Tail Recursion



Nothing is computed on the way back; 24 is passed up as is.

Step 7 / 9

```
1 static int factorial(int n, int acc) {
2     if (n == 0) {
3         return acc;
4     } else {
▶         return factorial(n - 1, n * acc);
6     }
7 }
```

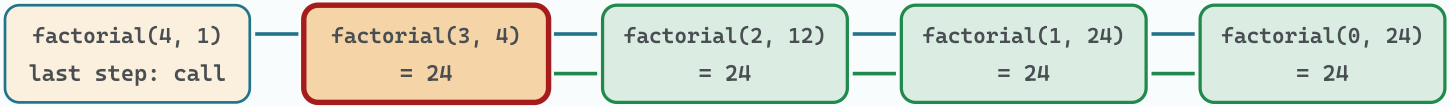
CALL STACK

- **factorial(2, 12)**
- factorial(3, 4)
- factorial(4, 1)

VARIABLES

`n = 2` `ret = 24`

`factorial(4, 1)`: Tracing Tail Recursion



Nothing is computed on the way back; 24 is passed up as is.

Step 8 / 9

```
1 static int factorial(int n, int acc) {
2     if (n == 0) {
3         return acc;
4     } else {
▶         return factorial(n - 1, n * acc);
6     }
7 }
```

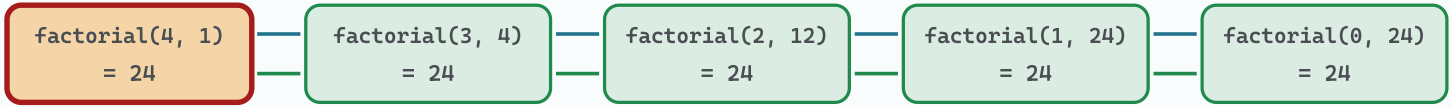
CALL STACK

- **factorial(3, 4)**
- factorial(4, 1)

VARIABLES

`n = 3` `ret = 24`

`factorial(4, 1)`: Tracing Tail Recursion



The result is 24. Java does not optimize the tail call: 5 frames still went on the stack. With TCO, one frame would be enough.

Step 9 / 9

```
1 static int factorial(int n, int acc) {
2     if (n == 0) {
3         return acc;
4     } else {
▶         return factorial(n - 1, n * acc);
6     }
7 }
```

CALL STACK

`factorial(4, 1)`

VARIABLES

`n = 4` `ret = 24`

Memoization

- The results of repeated computations are **stored and reused** (the method carries state).
- Results go into a cache (`Map`); when the same argument comes again, it is returned **without recomputation**.
- For each n the work is done **once**: the exponential number of calls drops to linear.
- `Map<Integer, Long>` : key n, value F(n).

`fibonacci(90)` → 2880067194370816120 (instantly).

```
import java.util.HashMap;
import java.util.Map;

static Map<Integer, Long> memo = new HashMap<>();

static long fibonacci(int n) {
    if (memo.containsKey(n)) {
        return memo.get(n);    // computed before
    }
    if (n == 0 || n == 1) {
        return n;
    }
    long result = fibonacci(n - 1) + fibonacci(n - 2);
    memo.put(n, result);        // store the result
    return result;
}
```

Memoized `fibonacci(4)`: Step by Step



`fibonacci(4)` is called. The memo is empty: 4 is not there and it is not a base case → first `fibonacci(3)` is called.

Step 1 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
6      if (n == 0 || n == 1) {
7          return n;
8      }
▶      long result = fibonacci(n - 1) + fibonacci(n - 2);
10     memo.put(n, result);
11     return result;
12 }
```

CALL STACK

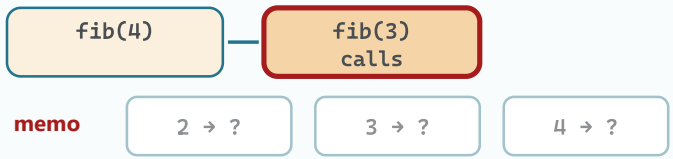
- `fibonacci(4)`
- `main()`

VARIABLES

`n = 4`

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`fibonacci(3)` : not in the memo → `fibonacci(2)` is called.

Step 2 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
6      if (n == 0 || n == 1) {
7          return n;
8      }
▶      long result = fibonacci(n - 1) + fibonacci(n - 2);
10     memo.put(n, result);
11     return result;
12 }
```

CALL STACK

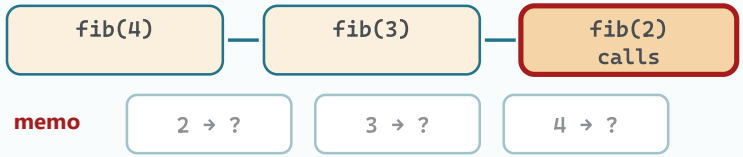
- **fibonacci(3)**
- fibonacci(4)
- main()

VARIABLES

n = 3

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`fibonacci(2)` : not in the memo → `fibonacci(1)` is called.

Step 3 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
6      if (n == 0 || n == 1) {
7          return n;
8      }
▶      long result = fibonacci(n - 1) + fibonacci(n - 2);
10     memo.put(n, result);
11     return result;
12 }
```

CALL STACK

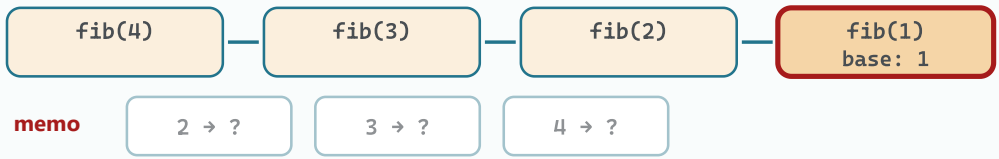
- **fibonacci(2)**
- fibonacci(3)
- fibonacci(4)
- main()

VARIABLES

n = 2

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`n = 1` : base case, 1 is returned.

Step 4 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
▶      if (n == 0 || n == 1) {
▶          return n;
8      }
9      long result = fibonacci(n - 1) + fibonacci(n - 2);
10     memo.put(n, result);
11     return result;
12 }
```

CALL STACK

- **fibonacci(1)**
- fibonacci(2)
- fibonacci(3)
- fibonacci(4)
- main()

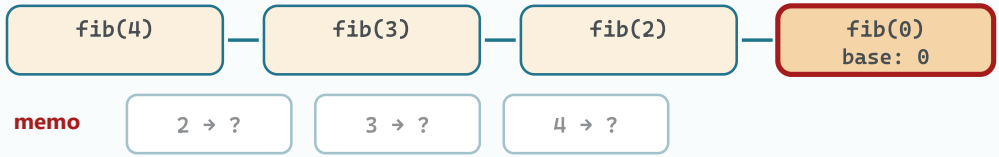
VARIABLES

`n = 1`

`ret = 1`

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`n = 0` : base case, 0 is returned.

Step 5 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
▶      if (n == 0 || n == 1) {
▶          return n;
8      }
9      long result = fibonacci(n - 1) + fibonacci(n - 2);
10     memo.put(n, result);
11     return result;
12 }
```

CALL STACK

- **fibonacci(0)**
- fibonacci(2)
- fibonacci(3)
- fibonacci(4)
- main()

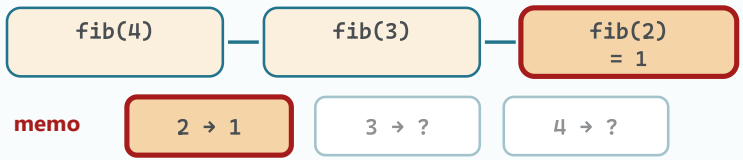
VARIABLES

`n = 0`

`ret = 0`

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`fibonacci(2) = 1 + 0 = 1`. The result goes into the memo: `{2=1}`.

Step 6 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
6      if (n == 0 || n == 1) {
7          return n;
8      }
9      long result = fibonacci(n - 1) + fibonacci(n - 2);
▶      memo.put(n, result);
▶      return result;
12 }
```

CALL STACK

- **fibonacci(2)**
- fibonacci(3)
- fibonacci(4)
- main()

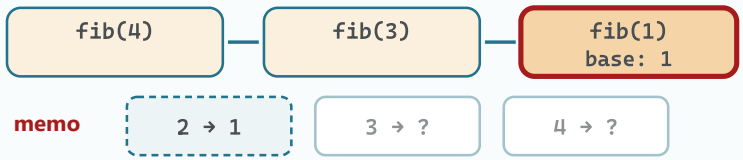
VARIABLES

`n = 2`

`ret = 1`

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`fibonacci(3)` 's second call, `fibonacci(1)` : base case, 1.

Step 7 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
▶      if (n == 0 || n == 1) {
▶          return n;
8      }
9      long result = fibonacci(n - 1) + fibonacci(n - 2);
10     memo.put(n, result);
11     return result;
12 }
```

CALL STACK

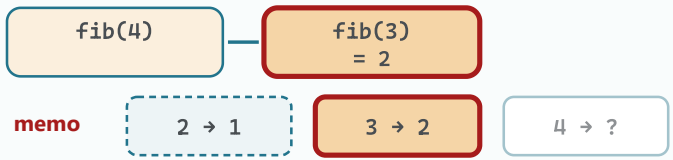
- **fibonacci(1)**
- fibonacci(3)
- fibonacci(4)
- main()

VARIABLES

- n = **1**
- ret = **1**

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`fibonacci(3) = 1 + 1 = 2`, stored in the memo.

Step 8 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
6      if (n == 0 || n == 1) {
7          return n;
8      }
9      long result = fibonacci(n - 1) + fibonacci(n - 2);
▶      memo.put(n, result);
▶      return result;
12 }
```

CALL STACK

- **fibonacci(3)**
- fibonacci(4)
- main()

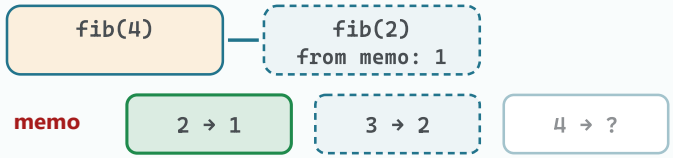
VARIABLES

`n = 3`

`ret = 2`

OUTPUT

Memoized `fibonacci(4)`: Step by Step



The second `fibonacci(2)` call: it is **in the memo**, so 1 is returned without computing (without memoization fib(2) would make 2 more calls here: fib(1) and fib(0)).

Step 9 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
▶      if (memo.containsKey(n)) {
▶          return memo.get(n);
5      }
6      if (n == 0 || n == 1) {
7          return n;
8      }
9      long result = fibonacci(n - 1) + fibonacci(n - 2);
10     memo.put(n, result);
11     return result;
12 }
```

CALL STACK

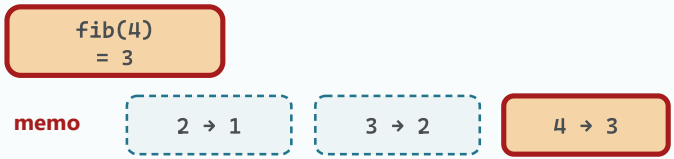
- **fibonacci(2)**
- fibonacci(4)
- main()

VARIABLES

- `n = 2`
- `ret = 1`

OUTPUT

Memoized `fibonacci(4)`: Step by Step



`fibonacci(4) = 2 + 1 = 3`, stored in the memo. 7 calls in total (without memoization: 9).

Step 10 / 10

```
1  static Map<Integer, Long> memo = new HashMap<>();
2  static long fibonacci(int n) {
3      if (memo.containsKey(n)) {
4          return memo.get(n);
5      }
6      if (n == 0 || n == 1) {
7          return n;
8      }
9      long result = fibonacci(n - 1) + fibonacci(n - 2);
▶      memo.put(n, result);
▶      return result;
12 }
```

CALL STACK

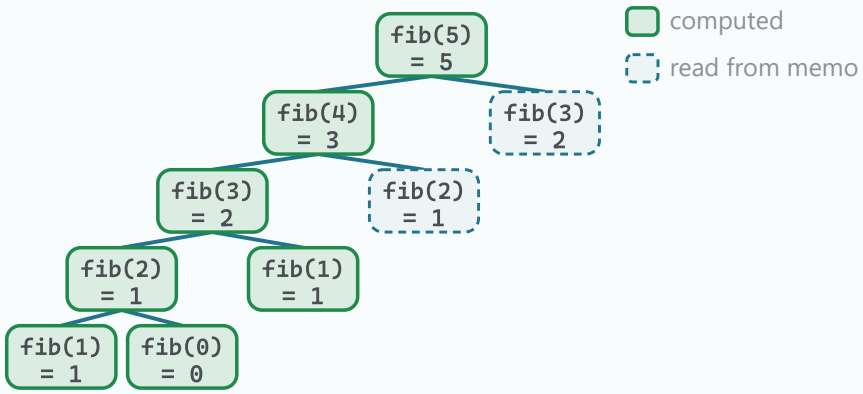
- `fibonacci(4)`
- `main()`

VARIABLES

- `n = 4`
- `ret = 3`

OUTPUT

Memoized Fibonacci: Call Tree



Memoized `fibonacci(5)`: the left branch is computed once; on the right, `fib(2)` and `fib(3)` come from the memo. 9 calls instead of 15.

n	calls without memo	calls with memo	time without memo (measured)	time with memo
20	21,891	39	≈ 0.5 ms	< 0.1 ms
40	331,160,281	79	≈ 0.5 s	< 1 ms

With memoization the number of calls is 2n − 1. Times vary by machine; the ratio does not.

Memoized Factorial

```
static Map<Integer, Long> memo = new HashMap<>();

static long factorial(int n) {
    if (memo.containsKey(n)) {
        return memo.get(n);
    }
    if (n == 0 || n == 1) {
        return 1;
    }
    long result = n * factorial(n - 1);
    memo.put(n, result);
    return result;
}
```

- Same pattern: check the memo; if absent, compute, store, return.
- `long` is exact up to 20!: `factorial(20)` = 2432902008176640000.
- A single `factorial(n)` call has no repetition; the memo does **not speed it up**.
- The gain appears when the program asks for factorials **many times**: after `factorial(10)`, `factorial(12)` needs only 2 new multiplications.
- The cost is memory: memoization trades memory for speed.

Summary

Concept	Meaning
Recursion	A method calling itself on a smaller input
Base case	The smallest case, solved directly; stops the recursion
Progress	Each call moves toward the base case (n − 1, n / 10, substring, x % y)
Call stack	One frame per call, popped in LIFO order. Too deep → `StackOverflowError`
Tree recursion	Several calls (Fibonacci, Hanoi); the call count can grow exponentially
Iterative solution	Loop + accumulator variable; constant memory, usually faster
Tail recursion	The last step is the call; the result is in the accumulator. Java has no TCO
Memoization	Storing computed results in a `Map` and reusing them

SUMMARY

Common Mistakes

Missing / unreachable base case

`sum(0)`, for example: `Exception in thread "main" java.lang.StackOverflowError`.

No progress

`return n * factorial(n);` does not shrink the parameter; again `StackOverflowError`.

Missing `return`

`if … else if` as the last branch of a method: `error: missing return statement`. Fix: make the last branch `else`.

Ignoring the call's result

`factorial(n - 1);` on its own line loses the value; it should be `return n * factorial(n - 1);`.

Overflow

`int` factorial goes wrong at 13!, the Bell coefficient after 13, with no error message. Fix: `long`.

Needless tree recursion

Without memoization, `fibonacci(50)` takes minutes. Fix: a loop or memoization.

SUMMARY

Worked Problems (1)

```
static void write(int n) {
    if (n == 0) return;
    System.out.print(n + " ");
    write(n - 1);
}
static void write2(int n) {
    if (n == 0) return;
    write2(n - 1);
    System.out.print(n + " ");
}
static int mystery(int a, int b) {
    if (b == 0) return 0;
    return a + mystery(a, b - 1);
}
```

1. `write(3)` : what does it print?
2. `write2(3)` : what does it print? Where does the difference come from?
3. `mystery(4, 3)` : what is its value? What does the method compute?
4. `mystery(4, -1)` : what happens?

▶ **Show answers**

SUMMARY

Worked Problems (2)

```
static String reverse(String s) {
    if (s.isEmpty()) return "";
    return reverse(s.substring(1)) + s.charAt(0);
}
```

1. `reverse("java")` : what does it return?
2. `reverse` : is it tail-recursive? What about `gcd` ?
3. `fibonacci(5)` : how many times is it called (including the first call)?
4. `gcd(48, 18)` : write out the calls.

▶ **Show answers**

SUMMARY

Exercises

1. `long power(int base, int exp)` : compute base to the power exp recursively. `power(2, 10)` = ?
2. `void binary(int n)` : print the binary representation of n. `binary(13)` → ?
3. `int arraySum(int[] a, int i)` : the sum of the array from index i to the end. `{3, 1, 4, 1, 5}` → ?
4. Write Fibonacci with a loop and with tail recursion (`fibT(n, a, b)`).
5. `lucas(5)` : how many calls does it make? How many with memoization?
6. How many moves do the Towers of Hanoi need for 4 disks? What is the first move?

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: Defining Methods; Returning a Value from a Method.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (15.12 Method Invocation Expressions).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 18: Recursion).
- A. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly (Chapter 8: Recursive Methods).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson.
- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley (2.3 Recursion).

The End

Chapter 3.5: Recursion

Previous: Chapter 3.4: Strings · Next: Chapter 4.1: Objects and Classes