Chapter 4: Recursion

Data Structures

Base case and recursive step | The call stack | Linear recursion | Fast exponentiation | Fibonacci and binary search | Towers of Hanoi | Backtracking | Recurrence relations | Tail recursion | Recursion on linked lists

Sercan KÜLCÜ · Giresun University · Computer Engineering

INTRODUCTION

Learning Outcomes

Recursive design

Define a problem by a base case and a recursive step, and write it in Java.

The call stack

Trace a recursive method frame by frame to find its output and return value.

Kinds of recursion

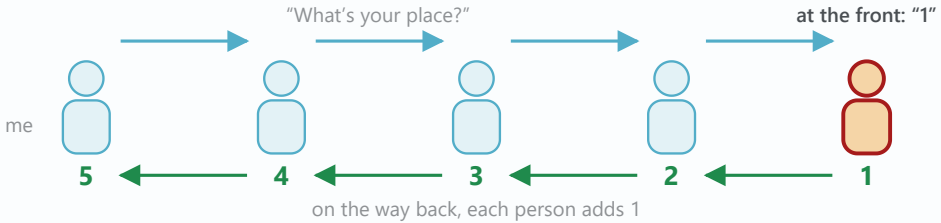
Tell linear, binary and multiple recursion and backtracking apart, with examples.

Cost

Write a recurrence, use the call tree to find time and space cost, and convert to a loop when needed.

Recursion in Everyday Life

You are in a long line and want to know your place, but you can't step out and count from the front.



Same question, shorter line

Everyone asks the person ahead the same question, each time about a line one person shorter.

Base case

The person at the front answers without asking anyone: "1". The questions stop here.

On the way back

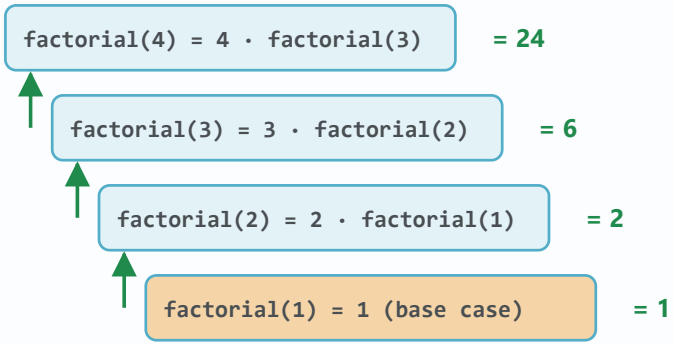
As the answer travels back, each person adds their share: 1, 2, 3, 4, 5.

A recursive method works the same way: it calls itself on a **smaller** input and answers the smallest input **directly**.

Recursion

$$n! = 1 \ (n \le 1) \cdot n! = n \cdot (n - 1)! \ (n > 1)$$

- **Recursion**: a method calling itself on a smaller input.
- **Base case**: solved directly, without recursion. Without one the calls never end → `StackOverflowError`.
- **Recursive step**: shrinks the problem toward the base case and builds the answer from the smaller problem's answer.
- Each call pushes a new **frame** on the call stack, with its own n.



↓ calls go down · ↑ values come back

Thinking Recursively

1. **Base case:** what is the smallest input, and what is its answer? Recursion stops here.
2. **Reduce:** how does the problem shrink to a smaller one of the same kind? (n − 1, n / 2, i + 1 …)
3. **Combine:** how do you build the big answer from the small one?

Assume the smaller problem is solved correctly and think about **one step** only. If the base case is right and every step is right, the whole chain is right — the logic of induction.

```
void countdown(int n) {
  if (n == 0) {     // base case
    System.out.println("Done");
    return;
  }
  System.out.println(n);
  countdown(n - 1); // recursive step
}
countdown(3);       // in main
```

countdown(n): print n down to 1, then “Done”.

Every recursive call must **move the input toward the base case**; in countdown, n drops by 1 per call and reaches 0.

Code Trace: `countdown(3)`



main calls `countdown(3)`. A new frame is about to be pushed.

Step 1 / 10

```
1 void countdown(int n) {
2   if (n == 0) {      // base case
3     System.out.println("Done");
4     return;
5   }
6   System.out.println(n);
7   countdown(n - 1); // recursive step
8 }
▶ countdown(3);       // in main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `countdown(3)`



n = 3 ≠ 0: not the base case. Print **3**, then call `countdown(2)`; this frame waits.

Step 2 / 10

```
1 void countdown(int n) {
▶   if (n == 0) {      // base case
3     System.out.println("Done");
4     return;
5   }
▶   System.out.println(n);
▶   countdown(n - 1); // recursive step
8 }
9 countdown(3);        // in main
```

VARIABLES

`n = 3`

OUTPUT

```
3
```

CALL STACK

- `countdown(3)`
- `main()`

Code Trace: `countdown(3)`



→ call · ← return

n = 2 ≠ 0: not the base case. Print **2**, then call `countdown(1)`; this frame waits.

Step 3 / 10

```
1 void countdown(int n) {
▶   if (n == 0) {      // base case
3     System.out.println("Done");
4     return;
5   }
▶   System.out.println(n);
▶   countdown(n - 1); // recursive step
8 }
9 countdown(3);        // in main
```

VARIABLES

`n = 2`

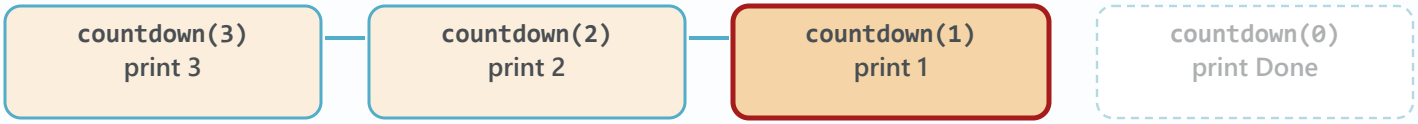
OUTPUT

```
3
2
```

CALL STACK

- `countdown(2)`
- `countdown(3)`
- `main()`

Code Trace: `countdown(3)`



n = 1 ≠ 0: not the base case. Print **1**, then call `countdown(0)`; this frame waits.

Step 4 / 10

```
1 void countdown(int n) {
▶   if (n == 0) {      // base case
3     System.out.println("Done");
4     return;
5   }
▶   System.out.println(n);
▶   countdown(n - 1); // recursive step
8 }
9 countdown(3);        // in main
```

VARIABLES

`n = 1`

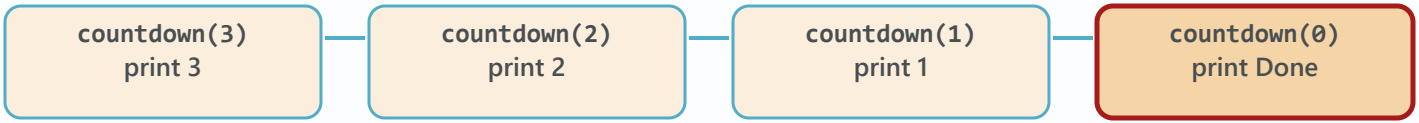
OUTPUT

```
3
2
1
```

CALL STACK

- `countdown(1)`
- `countdown(2)`
- `countdown(3)`
- `main()`

Code Trace: `countdown(3)`



n = 0: **base case**. Print “Done”. The stack is at its deepest: main + 4 frames.

Step 5 / 10

```
1 void countdown(int n) {
▶   if (n == 0) {      // base case
▶     System.out.println("Done");
4     return;
5   }
6   System.out.println(n);
7   countdown(n - 1); // recursive step
8 }
9 countdown(3);       // in main
```

VARIABLES

`n = 0`

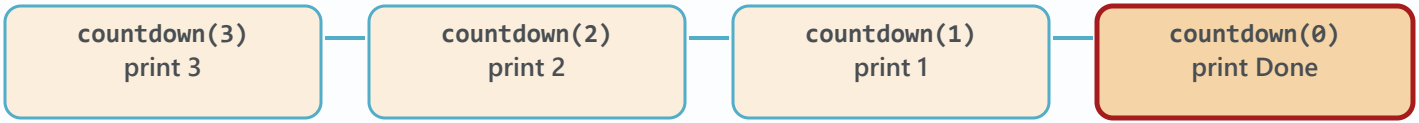
OUTPUT

```
3
2
1
Done
```

CALL STACK

- `countdown(0)`
- `countdown(1)`
- `countdown(2)`
- `countdown(3)`
- `main()`

Code Trace: `countdown(3)`



→ call · ← return

`return` : countdown(0) ends and its frame is popped.

Step 6 / 10

```
1 void countdown(int n) {
2   if (n == 0) {      // base case
3     System.out.println("Done");
▶     return;
5   }
6   System.out.println(n);
7   countdown(n - 1); // recursive step
8 }
9 countdown(3);       // in main
```

VARIABLES

`n = 0`

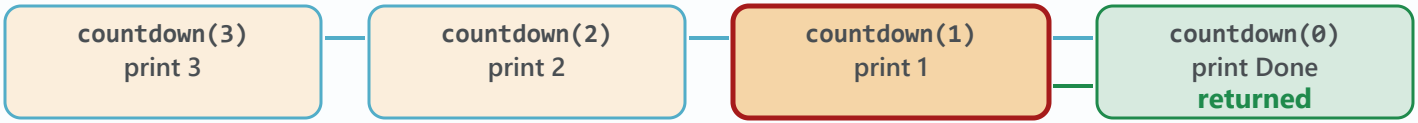
OUTPUT

```
3
2
1
Done
```

CALL STACK

- `countdown(0)`
- `countdown(1)`
- `countdown(2)`
- `countdown(3)`
- `main()`

Code Trace: `countdown(3)`



→ call · ← return

countdown(1) returns from the call on line 7. Nothing is left to do, so it ends too.

Step 7 / 10

```
1 void countdown(int n) {
2   if (n == 0) {      // base case
3     System.out.println("Done");
4     return;
5   }
6   System.out.println(n);
7   countdown(n - 1); // recursive step
▶ }
9 countdown(3);        // in main
```

VARIABLES

`n = 1`

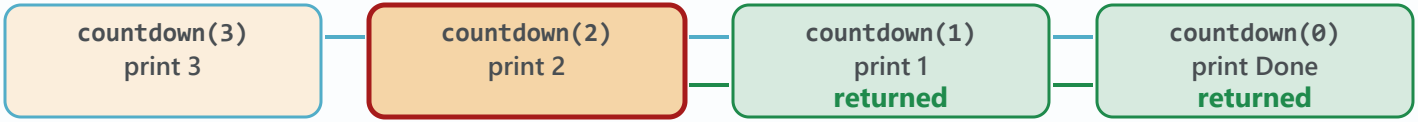
OUTPUT

```
3
2
1
Done
```

CALL STACK

- **countdown(1)**
- countdown(2)
- countdown(3)
- main()

Code Trace: `countdown(3)`



→ call · ← return

countdown(2) returns from the call on line 7. Nothing is left to do, so it ends too.

Step 8 / 10

```
1 void countdown(int n) {
2   if (n == 0) {      // base case
3     System.out.println("Done");
4     return;
5   }
6   System.out.println(n);
7   countdown(n - 1); // recursive step
▶ }
9 countdown(3);        // in main
```

VARIABLES

`n = 2`

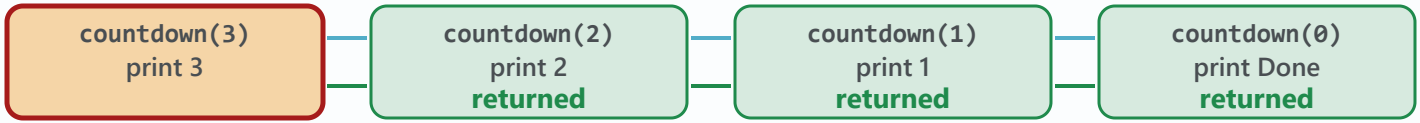
OUTPUT

```
3
2
1
Done
```

CALL STACK

- `countdown(2)`
- `countdown(3)`
- `main()`

Code Trace: `countdown(3)`



countdown(3) returns from the call on line 7. Nothing is left to do, so it ends too.

Step 9 / 10

```
1 void countdown(int n) {
2   if (n == 0) {      // base case
3     System.out.println("Done");
4     return;
5   }
6   System.out.println(n);
7   countdown(n - 1); // recursive step
▶ }
9 countdown(3);        // in main
```

VARIABLES

`n = 3`

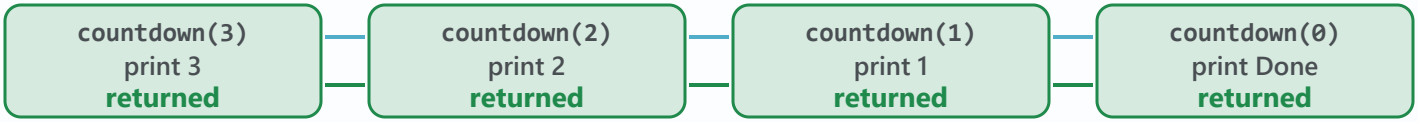
OUTPUT

```
3
2
1
Done
```

CALL STACK

- `countdown(3)`
- `main()`

Code Trace: `countdown(3)`



→ call · ← return

All frames have returned. 4 calls (n = 3, 2, 1, 0); output: 3 2 1 Done.

Step 10 / 10

```
1 void countdown(int n) {
2   if (n == 0) {     // base case
3     System.out.println("Done");
4     return;
5   }
6   System.out.println(n);
7   countdown(n - 1); // recursive step
8 }
▶ countdown(3);       // in main
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
3
2
1
Done
```

Work Before or After the Call?

```java
void countdown2(int n) {
  if (n == 0) return;
  System.out.print(n + " ");
  countdown2(n - 1);   // call after
}
```

`countdown2(3)` prints `3 2 1`
Printing happens as the calls **go down**.

```java
void countUp(int n) {
  if (n == 0) return;
  countUp(n - 1);      // call first
  System.out.print(n + " ");
}
```

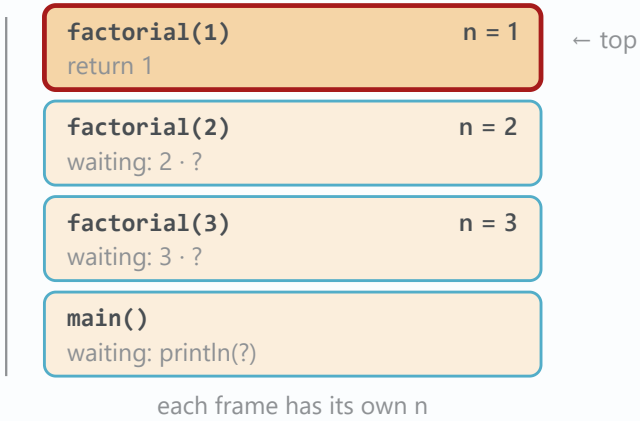
`countUp(3)` prints `1 2 3`
Printing happens as the calls **return**.

- `countUp(3)` waits for `countUp(2)` to finish before printing 3; that call in turn waits for `countUp(1)` before printing 2.
- The first call to finish is the last one started (n = 0), so work done after the call happens in **reverse order**: 1, 2, 3.

Work left after the call is kept in the frame while it waits on the stack, and runs as the stack unwinds.

The Call Stack and Frames

- Every method call gets a **stack frame** (or just frame) in memory: its parameters, local variables, and the place to resume on return.
- Frames live on the **call stack**: a call pushes one on top, and returning pops it. Last started, first finished: LIFO.
- Only the top frame runs; the ones below wait for the result of the call above them.
- In recursion, **several frames** of the same method are on the stack at once, each with its own n.



The memory cost of a recursive method is the number of frames on the stack at once — the **deepest chain of calls**.

Code Trace: `factorial(4)` and the call stack

→ call · ← returned value

main calls `factorial(4)`.

Step 1 / 13

```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
▶ System.out.println(factorial(4));
```

OUTPUT

CALL STACK

`main()`

Code Trace: `factorial(4)` and the call stack

factorial(4)
n = 4

→ call · ← returned value

New frame: **n = 4**. Is 4 ≤ 1? No.

Step 2 / 13

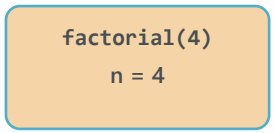
```
1 int factorial(int n) {
▶   if (n <= 1)
3     return 1;                // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

- factorial(4)
- main()

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

4 · factorial(3): the product needs **factorial(3)** first, so this frame **waits**.

Step 3 / 13

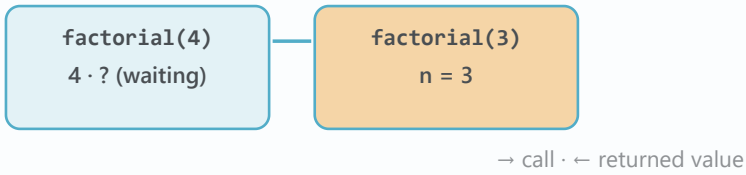
```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                // base case
▶   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



New frame: **n = 3**. Is 3 ≤ 1? No.

Step 4 / 13

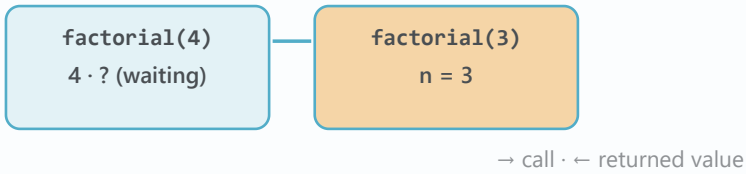
```
1 int factorial(int n) {
▶   if (n <= 1)
3     return 1;                // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



3 · factorial(2): the product needs **factorial(2)** first, so this frame **waits**.

Step 5 / 13

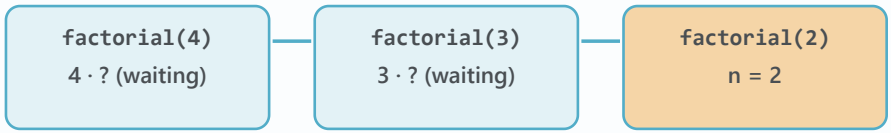
```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                // base case
▶   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

New frame: **n = 2**. Is 2 ≤ 1? No.

Step 6 / 13

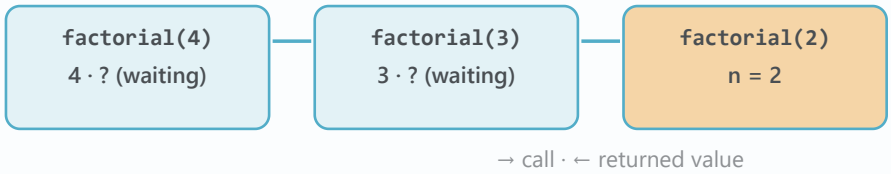
```
1 int factorial(int n) {
▶   if (n <= 1)
3     return 1;                 // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



2 · factorial(1): the product needs **factorial(1)** first, so this frame **waits**.

Step 7 / 13

```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                // base case
▶   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

New frame: **n = 1**. Is 1 ≤ 1? **Yes**.

Step 8 / 13

```
1 int factorial(int n) {
▶   if (n <= 1)
3     return 1;                 // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

Base case: return 1. No more calls; values start coming back.

Step 9 / 13

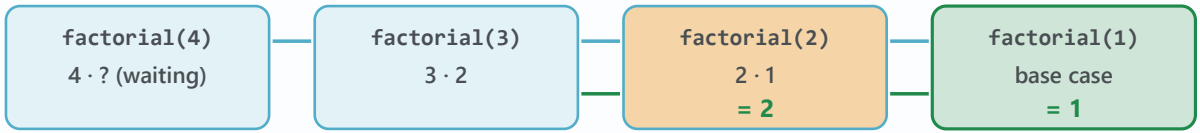
```
1 int factorial(int n) {
2   if (n <= 1)
▶     return 1;                // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

factorial(1) returned 1 → **2 · 1 = 2**, return 2.

Step 10 / 13

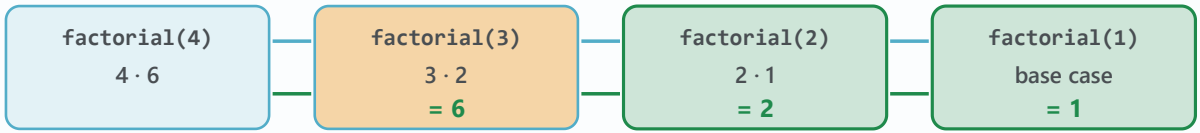
```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                 // base case
▶   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

factorial(2) returned 2 → **3 · 2 = 6**, return 6.

Step 11 / 13

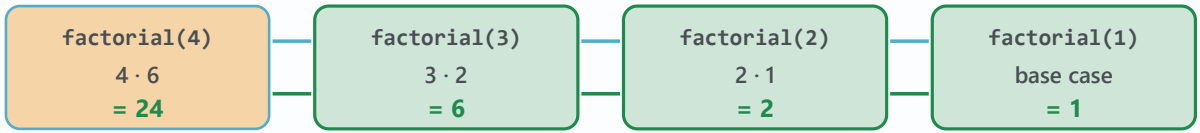
```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                // base case
▶   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

factorial(3) returned 6 → **4 · 6 = 24**, return 24.

Step 12 / 13

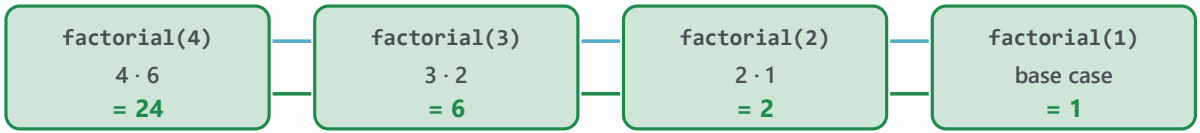
```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                // base case
▶   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

OUTPUT

CALL STACK

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

Code Trace: `factorial(4)` and the call stack



→ call · ← returned value

Prints **24**. 4 calls, stack depth at most 4 + main → time and memory **O(n)**.

Step 13 / 13

```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;                // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
▶ System.out.println(factorial(4));
```

OUTPUT

```
24
```

CALL STACK

```
main()
```

Stack Overflow: `StackOverflowError`

```
long sumTo(int n) {        // 1 + 2 + … + n
  if (n == 0) return 0;
  return n + sumTo(n - 1);
}
// in main:
System.out.println(sumTo(10_000));   // 50005000
System.out.println(sumTo(1_000_000)); // error
```

```
Exception in thread "main"
  java.lang.StackOverflowError
    at Example.sumTo(Example.java:3)
    at Example.sumTo(Example.java:3)
    …
```

- The call stack is a **limited** region of memory. Its size depends on the JVM and the operating system and can be changed with the `-Xss` option.
- Every frame takes space. With default settings, depth is usually limited to somewhere between a few thousand and a few tens of thousands of calls.
- `sumTo(10_000)` fits; `sumTo(1_000_000)` fills the stack before reaching the base case.
- A loop, or the formula n(n + 1)/2, computes the same sum without trouble.

Forgetting the Base Case

No base case

```
int f(int n) {
  return n * f(n - 1);
}
```

n = 3, 2, 1, 0, −1, −2 … the calls never stop.

Base case unreachable

```
if (n == 1) return 1;
return n * f(n - 1);
```

`f(0)` → −1 → −2 …: n never becomes 1. `n <= 1` catches this input too.

Problem doesn't shrink

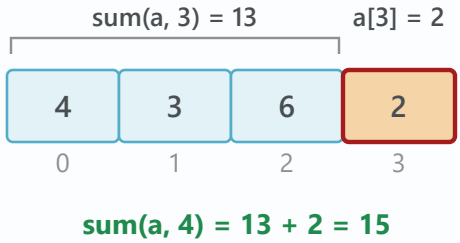
```
if (n <= 1) return 1;
return n * f(n);
```

The same input is passed again: there is a base case, but the calls never get closer to it.

All three end the same way: the stack fills up and a `StackOverflowError` is thrown. Every possible input, including 0 and negative values, must reach a base case in a finite number of steps.

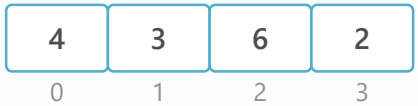
Linear Recursion

- Each call makes **at most one** recursive call, so the calls form a **chain** with no branches.
- A problem of size n reduces to a **single** problem of size n − 1 (or n / 2).
- Examples: factorial, countdown, summing an array, reversing an array, exponentiation, the length of a linked list.
- Array sum: sum of the first n items = sum of the first n − 1 items + a[n − 1]. Base case: n = 0 (empty prefix) → 0.




```
int sum(int[] a, int n) {   // sum of the first n items
  if (n == 0)
    return 0;               // base case
  return sum(a, n - 1) + a[n - 1];
}
```

Code Trace: `sum(a, 4)`



`sum(a, 4) = ?`

returned: —

main calls `sum(a, 4)`: the sum of the array's first 4 items.

Step 1 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
2   if (n == 0)
3     return 0;               // base case
4   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
▶ System.out.println(sum(a, 4));
```

VARIABLES

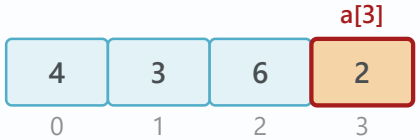
—

CALL STACK

`main()`

OUTPUT

Code Trace: `sum(a, 4)`



`sum(a, 4) = sum(a, 3) + a[3]`

returned: —

n = 4 ≠ 0 → `sum(a, 3)` must be computed first; a[3] = 2 waits to be added.

Step 2 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
▶   if (n == 0)
3     return 0;               // base case
▶   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

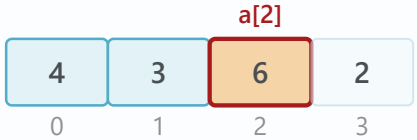
n = 4

OUTPUT

CALL STACK

- sum(a,4)
- main()

Code Trace: `sum(a, 4)`



`sum(a, 3) = sum(a, 2) + a[2]`

returned: —

n = 3 ≠ 0 → `sum(a, 2)` must be computed first; a[2] = 6 waits to be added.

Step 3 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
▶   if (n == 0)
3     return 0;               // base case
▶   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

n = 3

OUTPUT

CALL STACK

- sum(a,3)
- sum(a,4)
- main()

Code Trace: `sum(a, 4)`



`sum(a, 2) = sum(a, 1) + a[1]`

returned: —

n = 2 ≠ 0 → `sum(a, 1)` must be computed first; a[1] = 3 waits to be added.

Step 4 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
  if (n == 0)
3   return 0;                 // base case
  return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

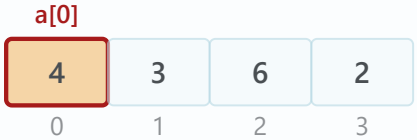
n = 2

OUTPUT

CALL STACK

- sum(a,2)
- sum(a,3)
- sum(a,4)
- main()

Code Trace: `sum(a, 4)`



`sum(a, 1) = sum(a, 0) + a[0]`

returned: —

n = 1 ≠ 0 → `sum(a, 0)` must be computed first; a[0] = 4 waits to be added.

Step 5 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
  if (n == 0)
3   return 0;                 // base case
  return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

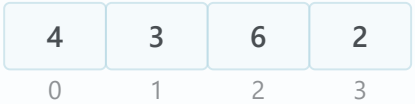
n = 1

OUTPUT

CALL STACK

- sum(a,1)
- sum(a,2)
- sum(a,3)
- sum(a,4)
- main()

Code Trace: `sum(a, 4)`



`sum(a, 0) = 0 (empty prefix)`

returned: 0

n = 0: **base case** → return 0. The stack is at its deepest: main + 5 frames.

Step 6 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
▶   if (n == 0)
▶     return 0;               // base case
4   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

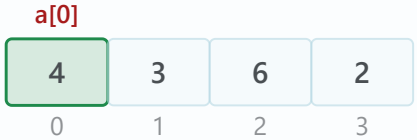
`n = 0`

OUTPUT

CALL STACK

- `sum(a,0)`
- `sum(a,1)`
- `sum(a,2)`
- `sum(a,3)`
- `sum(a,4)`
- `main()`

Code Trace: `sum(a, 4)`



`sum(a, 1) = 0 + 4 = 4`

returned: 0 → 4

sum(a, 0) returned 0 → 0 + a[0] = 0 + 4; return **4**.

Step 7 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
2   if (n == 0)
3     return 0;               // base case
▶   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

`n = 1`

`returned = 0`

OUTPUT

CALL STACK

- `sum(a,1)`
- `sum(a,2)`
- `sum(a,3)`
- `sum(a,4)`
- `main()`

Code Trace: `sum(a, 4)`



`sum(a, 2) = 4 + 3 = 7`

returned: 0 → 4 → 7

sum(a, 1) returned 4 → 4 + a[1] = 4 + 3; return **7**.

Step 8 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
2   if (n == 0)
3     return 0;               // base case
▶   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

`n = 2`

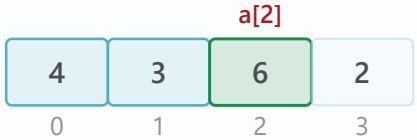
`returned 4 =`

OUTPUT

CALL STACK

- **`sum(a,2)`**
- `sum(a,3)`
- `sum(a,4)`
- `main()`

Code Trace: `sum(a, 4)`



`sum(a, 3) = 7 + 6 = 13`

returned: 0 → 4 → 7 → 13

sum(a, 2) returned 7 → 7 + a[2] = 7 + 6; return **13**.

Step 9 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
2   if (n == 0)
3     return 0;               // base case
  ▶ return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

`n = 3`

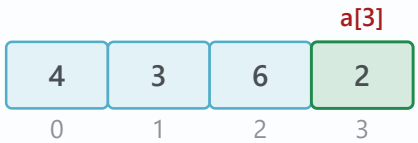
`returned =` **7**

CALL STACK

- **`sum(a,3)`**
- `sum(a,4)`
- `main()`

OUTPUT

Code Trace: `sum(a, 4)`



`sum(a, 4) = 13 + 2 = 15`

returned: 0 → 4 → 7 → 13 → 15

sum(a, 3) returned 13 → 13 + a[3] = 13 + 2; return **15**.

Step 10 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
2   if (n == 0)
3     return 0;               // base case
▶   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
8 System.out.println(sum(a, 4));
```

VARIABLES

`n = 4`

`returned = 3`

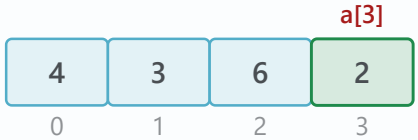
CALL STACK

`sum(a,4)`

`main()`

OUTPUT

Code Trace: `sum(a, 4)`



`sum(a, 4) = 13 + 2 = 15`

returned: 0 → 4 → 7 → 13 → 15

Result **15**. 5 calls (n = 4, 3, 2, 1, 0): time O(n); at most 5 frames (+ main): memory O(n).

Step 11 / 11

```
1 int sum(int[] a, int n) {   // sum of the first n items
2   if (n == 0)
3     return 0;               // base case
4   return sum(a, n - 1) + a[n - 1];
5 }
6 // in main:
7 int[] a = {4, 3, 6, 2};
▶ System.out.println(sum(a, 4));
```

VARIABLES

—

CALL STACK

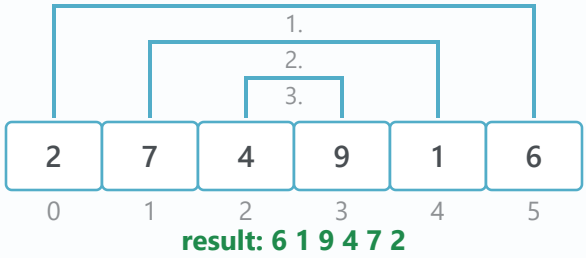
`main()`

OUTPUT

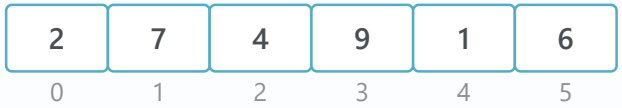
```
15
```

Reversing an Array

- Idea: swap the two **ends**, then reverse the part in between **the same way**.
- The part is given by the indices of `a[i..j]`; the first call is `reverse(a, 0, n - 1)`.
- Base case: `i >= j`, so 0 or 1 items remain and there is nothing to do.
- Each call shortens the part by 2 items: for n items, ⌊n/2⌋ swaps and ⌊n/2⌋ + 1 calls → **O(n)**.
- The array changes in place; the method returns nothing (`void`).



Code Trace: `reverse(a, 0, 5)`



`a = {2, 7, 4, 9, 1, 6}`

main calls `reverse(a, 0, 5)`.

Step 1 / 6

```
1 void reverse(int[] a, int i, int j) {
2   if (i >= j)
3     return;                    // base: 0 or 1 items
4   int t = a[i]; a[i] = a[j]; a[j] = t;  // the ends
5   reverse(a, i + 1, j - 1);    // the inner part
6 }
7 // in main:
8 int[] a = {2, 7, 4, 9, 1, 6};
▶ reverse(a, 0, a.length - 1);
```

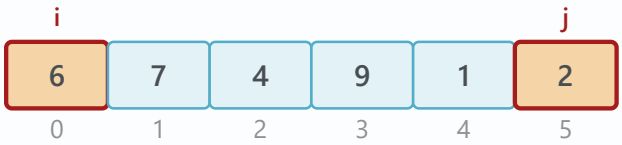
VARIABLES

—

CALL STACK

`main()`

Code Trace: `reverse(a, 0, 5)`



`a[0] ↔ a[5]` swapped

i = 0 < j = 5 → swap a[0] and a[5]: the array is **6 7 4 9 1 2**. Then call `reverse(a, 1, 4)`.

Step 2 / 6

```
1 void reverse(int[] a, int i, int j) {
▶   if (i >= j)
3     return;                    // base: 0 or 1 items
▶   int t = a[i]; a[i] = a[j]; a[j] = t;  // the ends
▶   reverse(a, i + 1, j - 1);    // the inner part
6 }
7 // in main:
8 int[] a = {2, 7, 4, 9, 1, 6};
9 reverse(a, 0, a.length - 1);
```

VARIABLES

`i = 0`

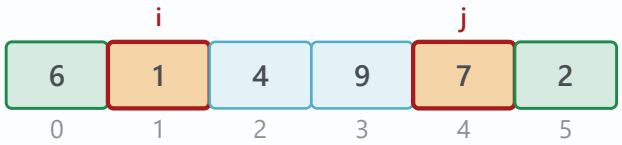
`j = 5`

CALL STACK

`reverse(0,5)`

`main()`

Code Trace: `reverse(a, 0, 5)`



`a[1] ↔ a[4]` swapped

i = 1 < j = 4 → swap a[1] and a[4]: the array is **6 1 4 9 7 2**. Then call `reverse(a, 2, 3)`.

Step 3 / 6

```
1 void reverse(int[] a, int i, int j) {
▶   if (i >= j)
3     return;                    // base: 0 or 1 items
▶   int t = a[i]; a[i] = a[j]; a[j] = t;  // the ends
▶   reverse(a, i + 1, j - 1);    // the inner part
6 }
7 // in main:
8 int[] a = {2, 7, 4, 9, 1, 6};
9 reverse(a, 0, a.length - 1);
```

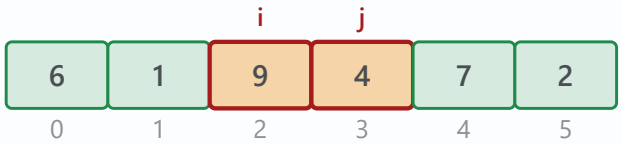
VARIABLES

- `i = 1`
- `j = 4`

CALL STACK

- `reverse(1,4)`
- `reverse(0,5)`
- `main()`

Code Trace: `reverse(a, 0, 5)`



a[2] ↔ a[3] swapped

i = 2 < j = 3 → swap a[2] and a[3]: the array is **6 1 9 4 7 2**. Then call `reverse(a, 3, 2)`.

Step 4 / 6

```
1 void reverse(int[] a, int i, int j) {
▶   if (i >= j)
3     return;                    // base: 0 or 1 items
▶   int t = a[i]; a[i] = a[j]; a[j] = t;  // the ends
▶   reverse(a, i + 1, j - 1);    // the inner part
6 }
7 // in main:
8 int[] a = {2, 7, 4, 9, 1, 6};
9 reverse(a, 0, a.length - 1);
```

VARIABLES

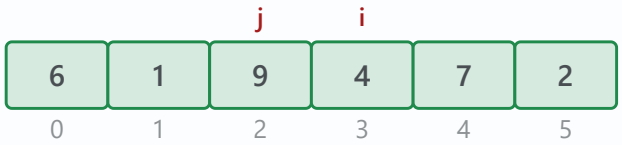
i = 2

j = 3

CALL STACK

- reverse(2,3)
- reverse(1,4)
- reverse(0,5)
- main()

Code Trace: `reverse(a, 0, 5)`



`i = 3 ≥ j = 2 → base case`

i = 3 ≥ j = 2: **base case**; nothing is left in the middle. The stack holds main + 4 frames.

Step 5 / 6

```
1 void reverse(int[] a, int i, int j) {
▶   if (i >= j)
▶     return;                    // base: 0 or 1 items
4   int t = a[i]; a[i] = a[j]; a[j] = t;  // the ends
5   reverse(a, i + 1, j - 1);     // the inner part
6 }
7 // in main:
8 int[] a = {2, 7, 4, 9, 1, 6};
9 reverse(a, 0, a.length - 1);
```

VARIABLES

`i = 3`

`j = 2`

CALL STACK

- `reverse(3,2)`
- `reverse(2,3)`
- `reverse(1,4)`
- `reverse(0,5)`
- `main()`

Code Trace: `reverse(a, 0, 5)`



The frames return one by one with no work left to do. Result: **6 1 9 4 7 2**. 3 swaps, 4 calls.

Step 6 / 6

```
1 void reverse(int[] a, int i, int j) {
2   if (i >= j)
3     return;                    // base: 0 or 1 items
4   int t = a[i]; a[i] = a[j]; a[j] = t;  // the ends
5   reverse(a, i + 1, j - 1);    // the inner part
6 }
7 // in main:
8 int[] a = {2, 7, 4, 9, 1, 6};
▶ reverse(a, 0, a.length - 1);
```

VARIABLES

—

CALL STACK

`main()`

Exponentiation: Slow and Fast

Slow: $x^n = x \cdot x^{n-1}$, $x^0 = 1$

```
long power(long x, int n) {
  if (n == 0) return 1;
  return x * power(x, n - 1);
}
```

Fast: halve the exponent at every step

$x^n = (x^{n/2})^2$ (n even)

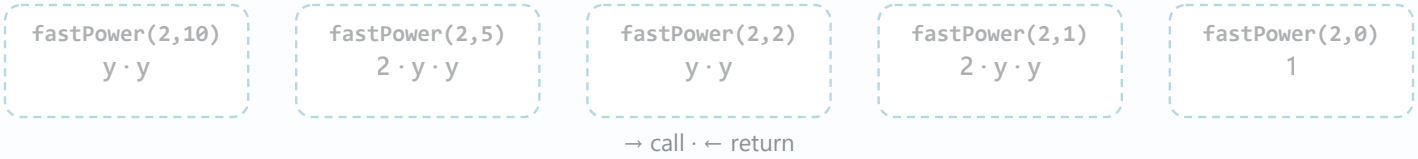
$x^n = x \cdot (x^{(n-1)/2})^2$ (n odd)

$2^{10} = (2^5)^2$, $2^5 = 2 \cdot (2^2)^2$,
$2^2 = (2^1)^2$, $2^1 = 2 \cdot (2^0)^2$

n	power: calls	fastPower: calls
10	11	5
100	101	8
1,000	1,001	11
10^6	1,000,001	21

The slow version makes n + 1 calls: **O(n)**. In the fast version n halves with every call: $\lfloor \log_2 n \rfloor + 2$ calls, **O(log n)**.

Code Trace: `fastPower(2, 10)`



main calls `fastPower(2, 10)` : 2^{10}.

Step 1 / 11

```
1 long fastPower(long x, int n) {
2   if (n == 0) return 1;            // base case
3   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
▶ System.out.println(fastPower(2, 10));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `fastPower(2, 10)`

fastPower(2,10) y · y | fastPower(2,5) 2 · y · y | fastPower(2,2) y · y | fastPower(2,1) 2 · y · y | fastPower(2,0) 1

→ call · ← return

n = 10 ≠ 0 → first compute y = `fastPower(2, 5)` (10 / 2 = 5).

Step 2 / 11

```
1 long fastPower(long x, int n) {
▶   if (n == 0) return 1;            // base case
▶   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

x = 2

n = 10

CALL STACK

fastPower(2,10)

main()

OUTPUT

Code Trace: `fastPower(2, 10)`



n = 5 ≠ 0 → first compute y = `fastPower(2, 2)` (5 / 2 = 2).

Step 3 / 11

```
1 long fastPower(long x, int n) {
▶   if (n == 0) return 1;            // base case
▶   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

`x = 2`

`n = 5`

OUTPUT

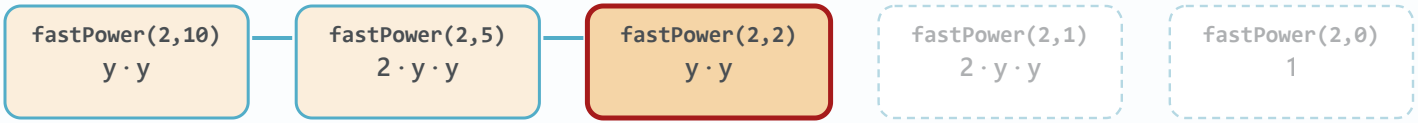
CALL STACK

`fastPower(2,5)`

`fastPower(2,10)`

`main()`

Code Trace: `fastPower(2, 10)`



n = 2 ≠ 0 → first compute y = `fastPower(2, 1)` (2 / 2 = 1).

Step 4 / 11

```
1 long fastPower(long x, int n) {
▶   if (n == 0) return 1;            // base case
▶   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

`x = 2`

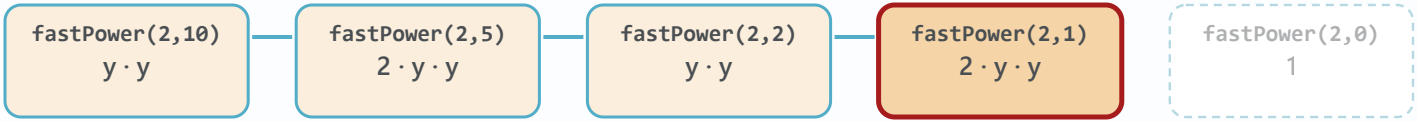
`n = 2`

OUTPUT

CALL STACK

- `fastPower(2,2)`
- `fastPower(2,5)`
- `fastPower(2,10)`
- `main()`

Code Trace: `fastPower(2, 10)`



n = 1 ≠ 0 → first compute y = `fastPower(2, 0)` (1 / 2 = 0).

Step 5 / 11

```
1 long fastPower(long x, int n) {
▶   if (n == 0) return 1;            // base case
▶   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

`x = 2`

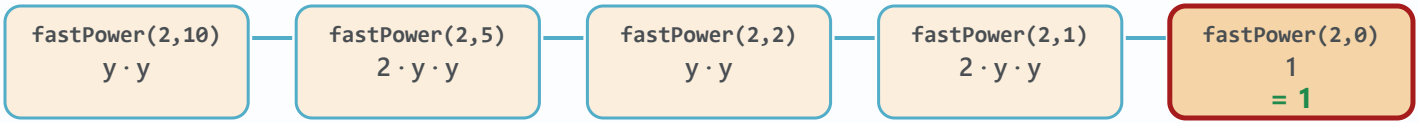
`n = 1`

OUTPUT

CALL STACK

- `fastPower(2,1)`
- `fastPower(2,2)`
- `fastPower(2,5)`
- `fastPower(2,10)`
- `main()`

Code Trace: `fastPower(2, 10)`



→ call · ← return

n = 0: **base case** → return 1 ($2^0 = 1$).

Step 6 / 11

```
1 long fastPower(long x, int n) {
▶   if (n == 0) return 1;            // base case
3   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

`x = 2`

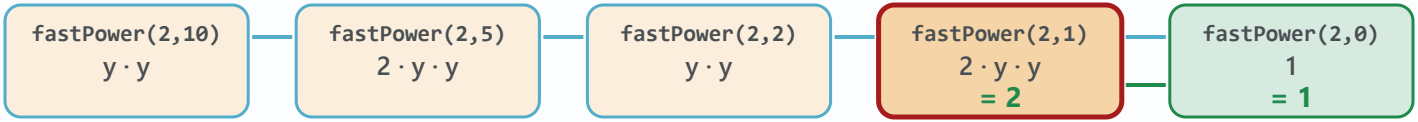
`n = 0`

OUTPUT

CALL STACK

- `fastPower(2,0)`
- `fastPower(2,1)`
- `fastPower(2,2)`
- `fastPower(2,5)`
- `fastPower(2,10)`
- `main()`

Code Trace: `fastPower(2, 10)`



y = 1 is back. n = 1 is odd → return 2 · y · y = 2 · 1 · 1 = **2**.

Step 7 / 11

```
1 long fastPower(long x, int n) {
2   if (n == 0) return 1;            // base case
3   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
▶   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

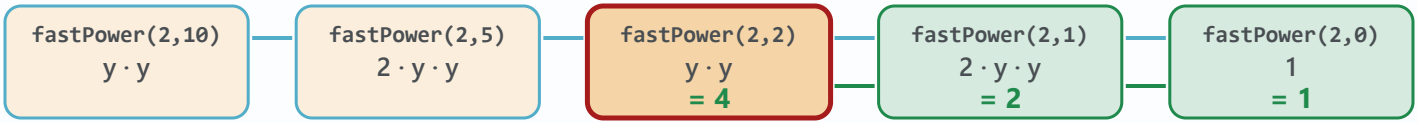
- x = 2
- n = 1
- y = 1

OUTPUT

CALL STACK

- fastPower(2,1)
- fastPower(2,2)
- fastPower(2,5)
- fastPower(2,10)
- main()

Code Trace: `fastPower(2, 10)`



y = 2 is back. n = 2 is even → return y · y = 2 · 2 = **4**.

Step 8 / 11

```
1 long fastPower(long x, int n) {
2   if (n == 0) return 1;            // base case
3   long y = fastPower(x, n / 2);    // x^(n/2)
▶   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

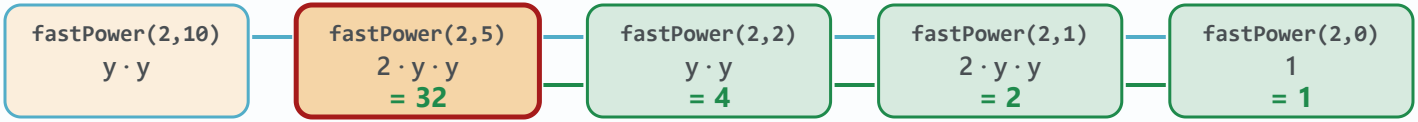
- x = 2
- n = 2
- y = 2

OUTPUT

CALL STACK

- fastPower(2,2)
- fastPower(2,5)
- fastPower(2,10)
- main()

Code Trace: `fastPower(2, 10)`



y = 4 is back. n = 5 is odd → return 2 · y · y = 2 · 4 · 4 = **32**.

Step 9 / 11

```
1 long fastPower(long x, int n) {
2   if (n == 0) return 1;            // base case
3   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
▶   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

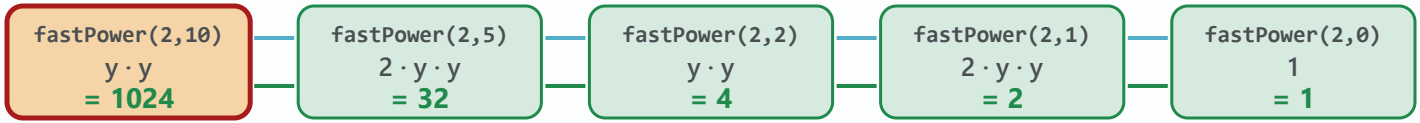
- x = 2
- n = 5
- y = 4

CALL STACK

- fastPower(2,5)
- fastPower(2,10)
- main()

OUTPUT

Code Trace: `fastPower(2, 10)`



y = 32 is back. n = 10 is even → return y · y = 32 · 32 = **1024**.

Step 10 / 11

```
1 long fastPower(long x, int n) {
2   if (n == 0) return 1;            // base case
3   long y = fastPower(x, n / 2);    // x^(n/2)
▶   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
8 System.out.println(fastPower(2, 10));
```

VARIABLES

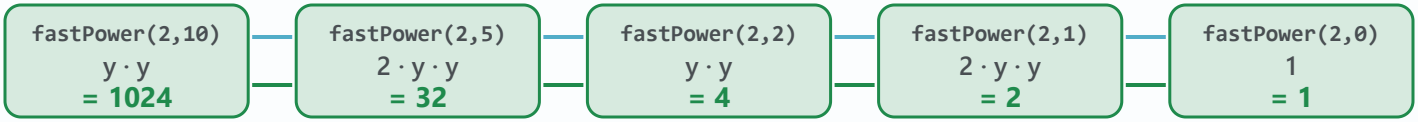
- x = 2
- n = 10
- y = 32

OUTPUT

CALL STACK

- fastPower(2,10)
- main()

Code Trace: `fastPower(2, 10)`



Result **1024**. Only 5 calls; the slow `power(2, 10)` would make 11.

Step 11 / 11

```
1 long fastPower(long x, int n) {
2   if (n == 0) return 1;            // base case
3   long y = fastPower(x, n / 2);    // x^(n/2)
4   if (n % 2 == 0) return y * y;    // n even
5   return x * y * y;                // n odd
6 }
7 // in main:
▶ System.out.println(fastPower(2, 10));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
1024
```

BINARY RECURSION

Fibonacci Numbers



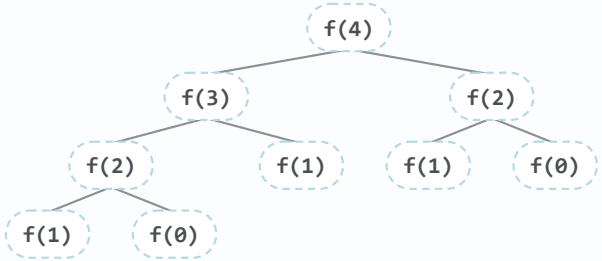

```
int fib(int n) {
  if (n <= 1) return n;     // base case
  return fib(n - 1) + fib(n - 2);
}
```

$F(0) = 0, F(1) = 1$

$F(n) = F(n - 1) + F(n - 2) \quad (n \geq 2)$

- **Binary recursion:** a call makes **two** recursive calls.
- The calls no longer form a chain but a **call tree**: each node is a call, and its children are the calls it makes.
- The definition turns straight into code, with two base cases: n = 0 and n = 1.
- The tree computes the same values more than once, and that is what sets the cost.

Code Trace: the `fib(4)` call tree



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

main calls `fib(4)`.

Step 1 / 15

```
1 int fib(int n) {
2   if (n <= 1) return n;     // base case
3   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
▶ System.out.println(fib(4));
```

VARIABLES

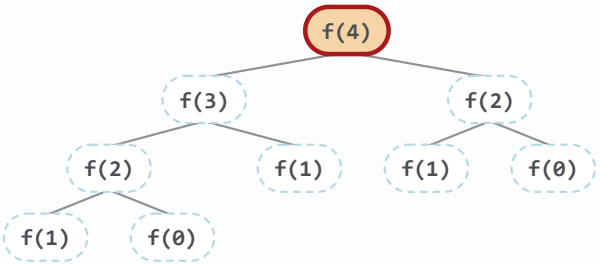
—

CALL STACK

`main()`

OUTPUT

Code Trace: the `fib(4)` call tree



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

`fib(4)` : n > 1 → call `fib(3)` first; `fib(2)` waits its turn.

Step 2 / 15

```
1 int fib(int n) {
▶   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

`n = 4`

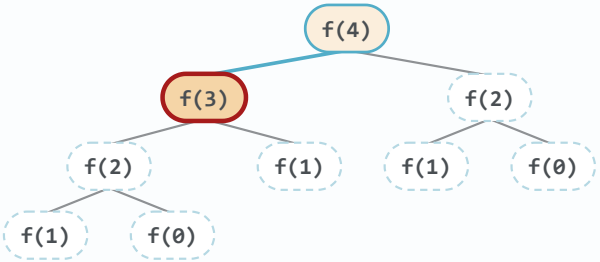
OUTPUT

CALL STACK

`fib(4)`

`main()`

Code Trace: the `fib(4)` call tree



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

`fib(3)` : n > 1 → call `fib(2)` first; `fib(1)` waits its turn.

Step 3 / 15

```
1 int fib(int n) {
▶   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

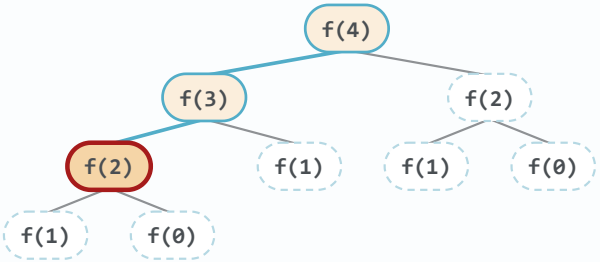
`n = 3`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

`fib(2)` : n > 1 → call `fib(1)` first; `fib(0)` waits its turn.

Step 4 / 15

```
1 int fib(int n) {
▶   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

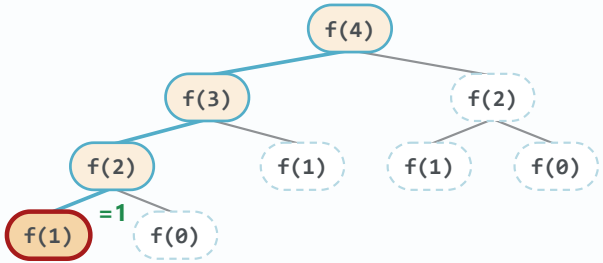
`n = 2`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

`fib(1)` : **base case** → return 1.

Step 5 / 15

```
1  int fib(int n) {
▶    if (n <= 1) return n;     // base case
3    return fib(n - 1) + fib(n - 2);
4  }
5  // in main:
6  System.out.println(fib(4));
```

VARIABLES

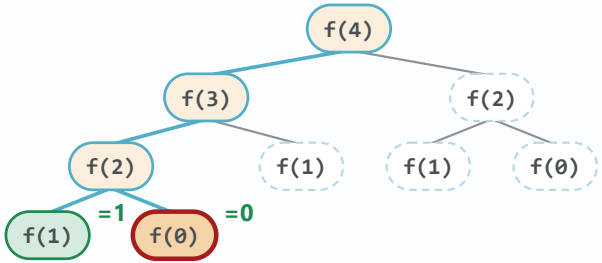
`n = 1`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

Second term: `fib(0)` : **base case** → return 0.

Step 6 / 15

```
1 int fib(int n) {
▶   if (n <= 1) return n;     // base case
3   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

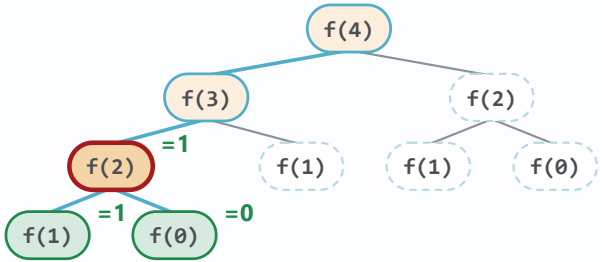
`n = 0`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

`fib(2)` = fib(1) + fib(0) = 1 + 0; return **1**.

Step 7 / 15

```
1 int fib(int n) {
2   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

`n = 2`

`fib(1) = 1`

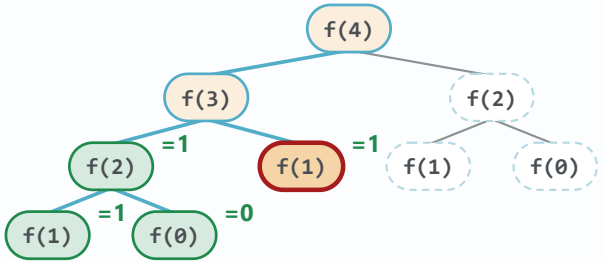
`fib(0) = 0`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

Second term: `fib(1)` : **base case** → return 1.

Step 8 / 15

```
1 int fib(int n) {
▶   if (n <= 1) return n;     // base case
3   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

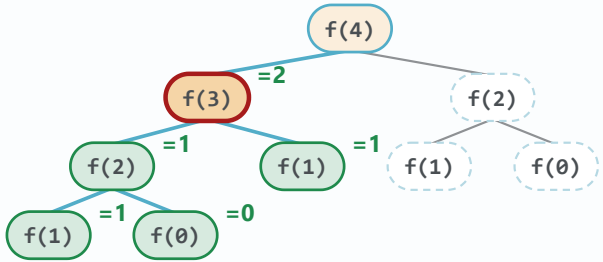
`n = 1`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

`fib(3)` = fib(2) + fib(1) = 1 + 1; return **2**.

Step 9 / 15

```
1 int fib(int n) {
2   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

`n = 3`

`fib(2) = 1`

`fib(1) = 1`

CALL STACK

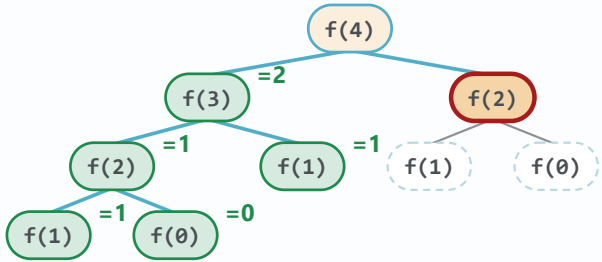
`fib(3)`

`fib(4)`

`main()`

OUTPUT

Code Trace: the `fib(4)` call tree



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

`fib(4)` calls `fib(2)` for its second term: fib(2) is computed **a second time**. First fib(1).

Step 10 / 15

```
1 int fib(int n) {
▶   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

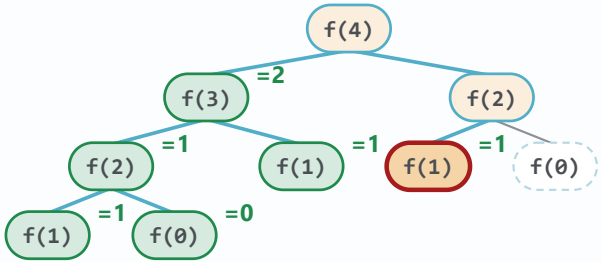
`n = 2`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

`fib(1)` : **base case** → return 1.

Step 11 / 15

```
1  int fib(int n) {
▶    if (n <= 1) return n;     // base case
3    return fib(n - 1) + fib(n - 2);
4  }
5  // in main:
6  System.out.println(fib(4));
```

VARIABLES

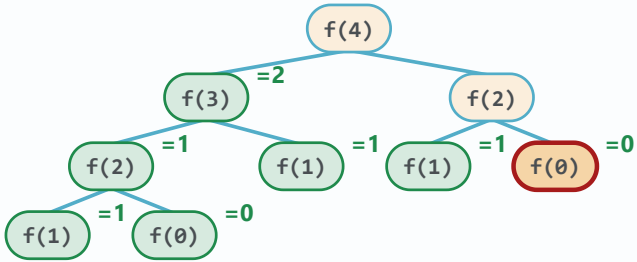
`n = 1`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

Second term: `fib(0)` : **base case** → return 0.

Step 12 / 15

```
1 int fib(int n) {
▶   if (n <= 1) return n;     // base case
3   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

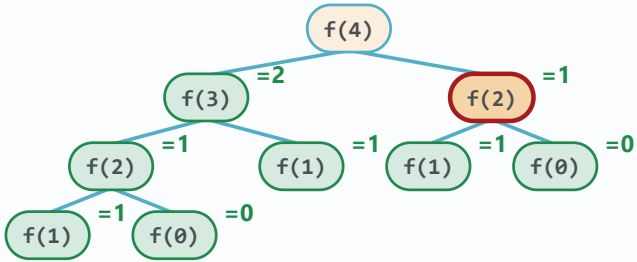
`n = 0`

OUTPUT

CALL STACK

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

Code Trace: the `fib(4)` call tree



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

`fib(2)` = fib(1) + fib(0) = 1 + 0; return **1**.

Step 13 / 15

```
1 int fib(int n) {
2   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

`n = 2`

`fib(1) = 1`

`fib(0) = 0`

CALL STACK

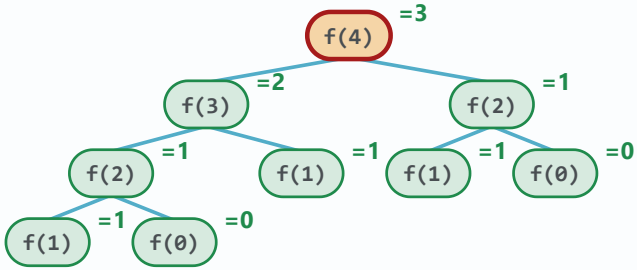
`fib(2)`

`fib(4)`

`main()`

OUTPUT

Code Trace: the `fib(4)` call tree



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

`fib(4)` = fib(3) + fib(2) = 2 + 1; return **3**.

Step 14 / 15

```
1 int fib(int n) {
2   if (n <= 1) return n;     // base case
▶   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
6 System.out.println(fib(4));
```

VARIABLES

`n = 4`

`fib(3) = 2`

`fib(2) = 1`

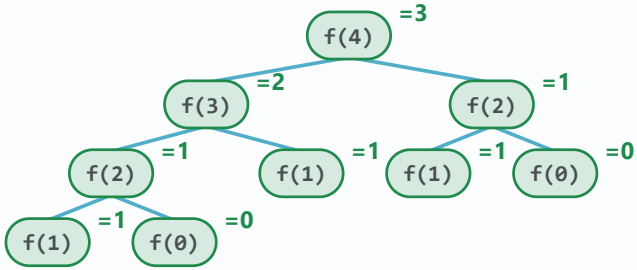
OUTPUT

CALL STACK

`fib(4)`

`main()`

Code Trace: the `fib(4)` call tree



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

`fib(4)` = **3**. 9 calls: fib(2) was computed twice, fib(1) three times, fib(0) twice.

Step 15 / 15

```
1 int fib(int n) {
2   if (n <= 1) return n;     // base case
3   return fib(n - 1) + fib(n - 2);
4 }
5 // in main:
▶ System.out.println(fib(4));
```

VARIABLES

—

OUTPUT

```
3
```

CALL STACK

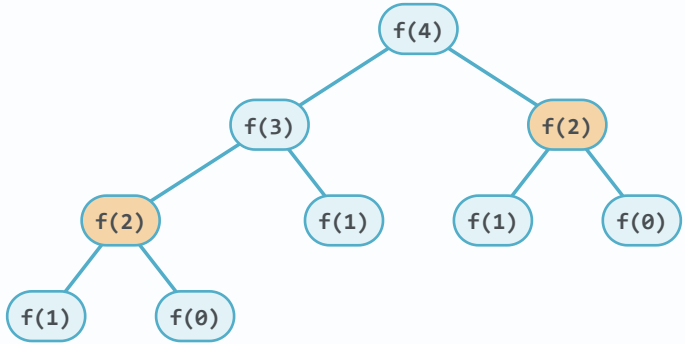
`main()`

BINARY RECURSION

The Cost of Recursion

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

- **factorial**: T(n) = T(n − 1) + c → n calls → **O(n)** time, **O(n)** stack memory.
- **fib**: every call makes **two** calls, so the same subproblems are solved over and over.
- fib(4) → 9 calls; the orange **f(2)** is computed twice. The number of calls grows **exponentially** (≈ 1.6^n).
- The fix: memoization or a loop → O(n).



Avoiding Repeated Work

n	fib(n)	calls to fib
5	5	15
10	55	177
20	6,765	21,891
30	832,040	2,692,537
40	102,334,155	331,160,281

Number of calls = 2 · F(n + 1) − 1: each time n grows by 1, it grows about 1.6 times.

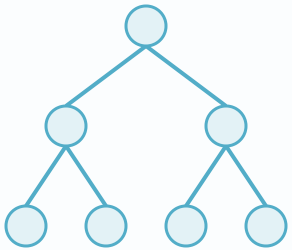
```
long[] memo = new long[91];     // 0: not computed
long fibMemo(int n) {
  if (n <= 1) return n;
  if (memo[n] != 0) return memo[n];
  memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
  return memo[n];
}
```

```
long fibLoop(int n) {
  long a = 0, b = 1;              // F(0), F(1)
  for (int i = 0; i < n; i++) {
    long t = a + b;  a = b;  b = t;
  }
  return a;
}
```

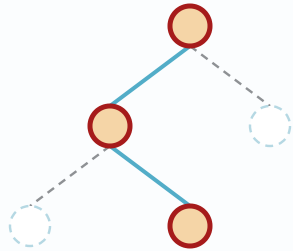
If computed values are stored in an array, each one is computed **once**: `fibMemo(40)` makes 79 calls. The loop version runs in O(n) time and O(1) memory.

Recursive Binary Search

- To find x in a sorted array, look at the middle item; if x is smaller, run **the same search** on the left half, otherwise on the right half.
- Two base cases: the range is empty (`lo > hi`) → −1; the middle item is x → its index.
- The code contains two recursive calls, but each call makes **only one** of them. Unlike Fibonacci there is no tree; the calls form a chain.
- The range halves with every call: T(n) = T(n/2) + c → **O(log n)** time and O(log n) stack memory.
- At most 5 calls for n = 10, at most 21 for n = 10^6.



fib: two calls → tree



binarySearch: one is taken → chain

BINARY RECURSION · STEP THROUGH THE CODE

Code Trace: `binarySearch(a, 23, 0, 9)`



main searches a sorted 10-item array for 23: `binarySearch(a, 23, 0, 9)`.

Step 1 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
2   if (lo > hi) return -1;        // base: empty range
3   int mid = (lo + hi) / 2;
4   if (a[mid] == x) return mid;   // base: found
5   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left
  half
7   return binarySearch(a, x, mid + 1, hi);    // right
  half
8 }
▶ System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `binarySearch(a, 23, 0, 9)`



lo = 0 ≤ hi = 9: the range is not empty. mid = (0 + 9) / 2 = **4**, a[4] = 16.

Step 2 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
▶   if (lo > hi) return -1;        // base: empty range
▶   int mid = (lo + hi) / 2;
4   if (a[mid] == x) return mid;  // base: found
5   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left
    half
7   return binarySearch(a, x, mid + 1, hi);   // right
    half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

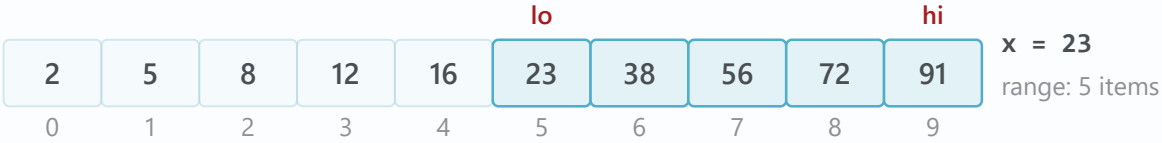
- `x = 23`
- `lo = 0`
- `hi = 9`
- `mid = 4`

OUTPUT

CALL STACK

- `binarySearch(0,9)`
- `main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



23 > 16 → **right half**: `binarySearch(a, 23, 5, 9)`. The left half is never examined.

Step 3 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
2   if (lo > hi) return -1;        // base: empty range
3   int mid = (lo + hi) / 2;
▶   if (a[mid] == x) return mid;  // base: found
▶   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left half
▶   return binarySearch(a, x, mid + 1, hi);   // right half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

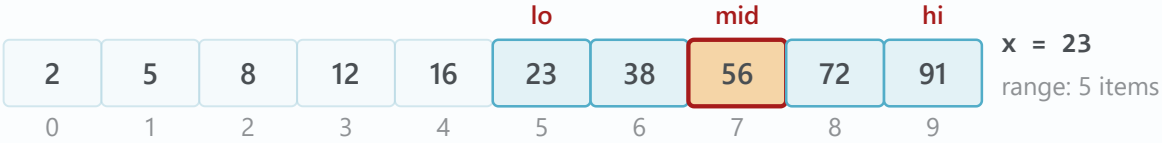
- `x = 23`
- `lo = 0`
- `hi = 9`
- `mid = 4`

OUTPUT

CALL STACK

- `binarySearch(0,9)`
- `main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



lo = 5 ≤ hi = 9: the range is not empty. mid = (5 + 9) / 2 = **7**, a[7] = 56.

Step 4 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
▶   if (lo > hi) return -1;        // base: empty range
▶   int mid = (lo + hi) / 2;
4   if (a[mid] == x) return mid;  // base: found
5   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left half
7   return binarySearch(a, x, mid + 1, hi);   // right half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

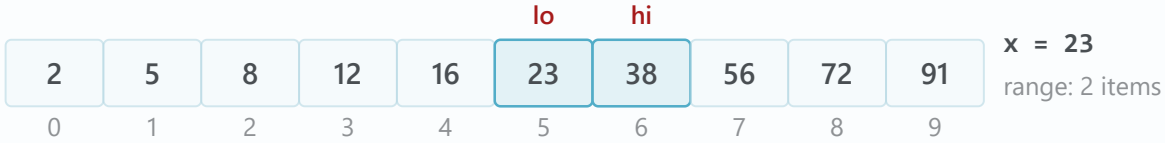
- `x = 23`
- `lo = 5`
- `hi = 9`
- `mid = 7`

OUTPUT

CALL STACK

- `binarySearch(5,9)`
- `binarySearch(0,9)`
- `main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



23 < 56 → **left half**: `binarySearch(a, 23, 5, 6)`. The right half is never examined.

Step 5 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
2   if (lo > hi) return -1;        // base: empty range
3   int mid = (lo + hi) / 2;
▶   if (a[mid] == x) return mid;  // base: found
▶   if (x < a[mid])
▶     return binarySearch(a, x, lo, mid - 1); // left half
7   return binarySearch(a, x, mid + 1, hi);   // right half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

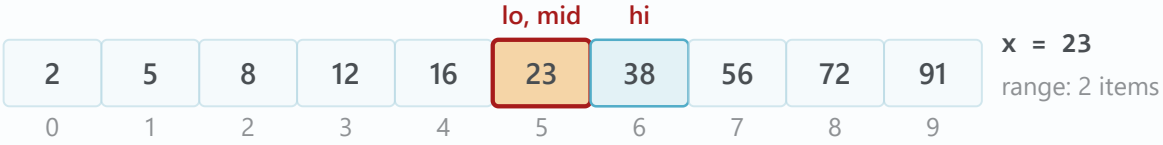
- `x = 23`
- `lo = 5`
- `hi = 9`
- `mid = 7`

OUTPUT

CALL STACK

- `binarySearch(5,9)`
- `binarySearch(0,9)`
- `main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



lo = 5 ≤ hi = 6: the range is not empty. mid = (5 + 6) / 2 = **5**, a[5] = 23.

Step 6 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
▶   if (lo > hi) return -1;        // base: empty range
▶   int mid = (lo + hi) / 2;
4   if (a[mid] == x) return mid;  // base: found
5   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left half
7   return binarySearch(a, x, mid + 1, hi);   // right half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

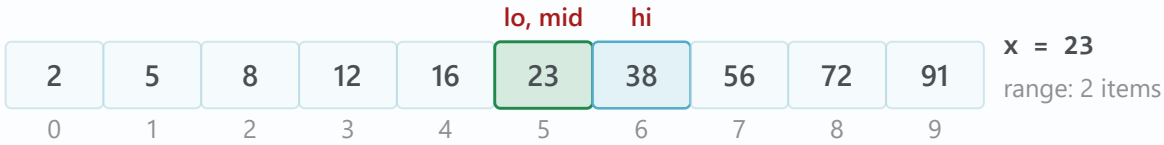
- `x = 23`
- `lo = 5`
- `hi = 6`
- `mid = 5`

OUTPUT

CALL STACK

- `binarySearch(5,6)`
- `binarySearch(5,9)`
- `binarySearch(0,9)`
- `main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



a[5] = 23 = x: **base case** (found) → return 5.

Step 7 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
2   if (lo > hi) return -1;        // base: empty range
3   int mid = (lo + hi) / 2;
▶   if (a[mid] == x) return mid;  // base: found
5   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left
    half
7   return binarySearch(a, x, mid + 1, hi);   // right
    half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

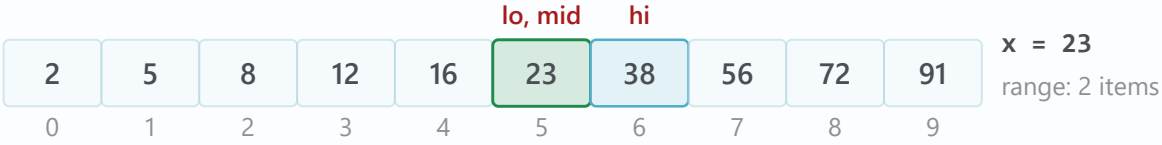
- `x = 23`
- `lo = 5`
- `hi = 6`
- `mid = 5`

OUTPUT

CALL STACK

- `binarySearch(5,6)`
- `binarySearch(5,9)`
- `binarySearch(0,9)`
- `main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



`binarySearch(a, 23, 5, 9)` passes the 5 it received straight on with `return`.

Step 8 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
2   if (lo > hi) return -1;        // base: empty range
3   int mid = (lo + hi) / 2;
4   if (a[mid] == x) return mid;   // base: found
5   if (x < a[mid])
▶     return binarySearch(a, x, lo, mid - 1); // left half
7   return binarySearch(a, x, mid + 1, hi);   // right half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

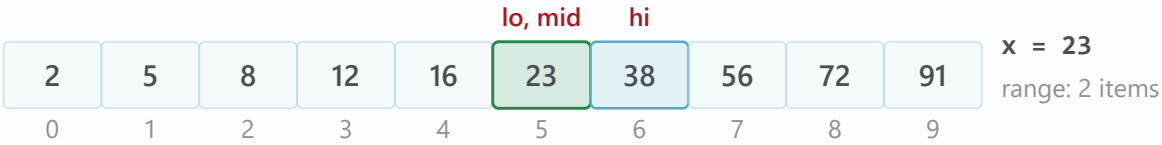
`returned = 5`

OUTPUT

CALL STACK

- `binarySearch(5,9)`
- `binarySearch(0,9)`
- `main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



`binarySearch(a, 23, 0, 9)` passes the 5 on too. No new computation happens on the way back.

Step 9 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
2   if (lo > hi) return -1;        // base: empty range
3   int mid = (lo + hi) / 2;
4   if (a[mid] == x) return mid;  // base: found
5   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left half
▶   return binarySearch(a, x, mid + 1, hi);   // right half
8 }
9 System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

`returned = 5`

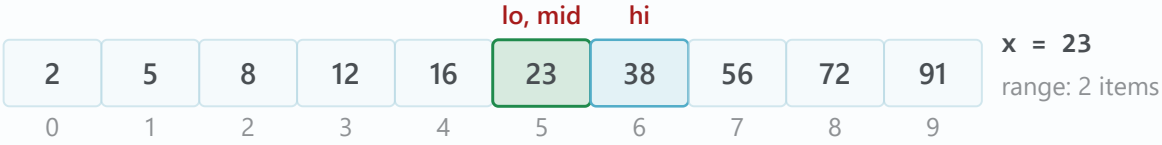
OUTPUT

CALL STACK

`binarySearch(0,9)`

`main()`

Code Trace: `binarySearch(a, 23, 0, 9)`



Result **5**. 3 calls; the range shrank from 10 to 5 to 2 items.

Step 10 / 10

```
1 int binarySearch(int[] a, int x, int lo, int hi) {
2   if (lo > hi) return -1;        // base: empty range
3   int mid = (lo + hi) / 2;
4   if (a[mid] == x) return mid;   // base: found
5   if (x < a[mid])
6     return binarySearch(a, x, lo, mid - 1); // left half
7   return binarySearch(a, x, mid + 1, hi);   // right half
8 }
▶ System.out.println(binarySearch(a, 23, 0, 9));
```

VARIABLES

—

CALL STACK

`main()`

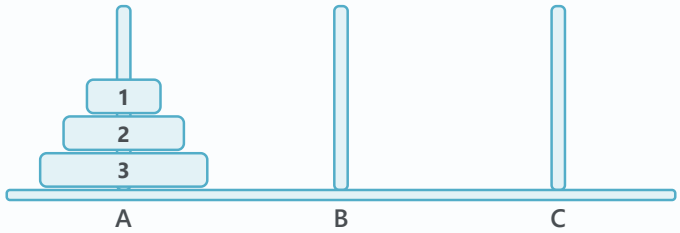
OUTPUT

```
5
```

Towers of Hanoi

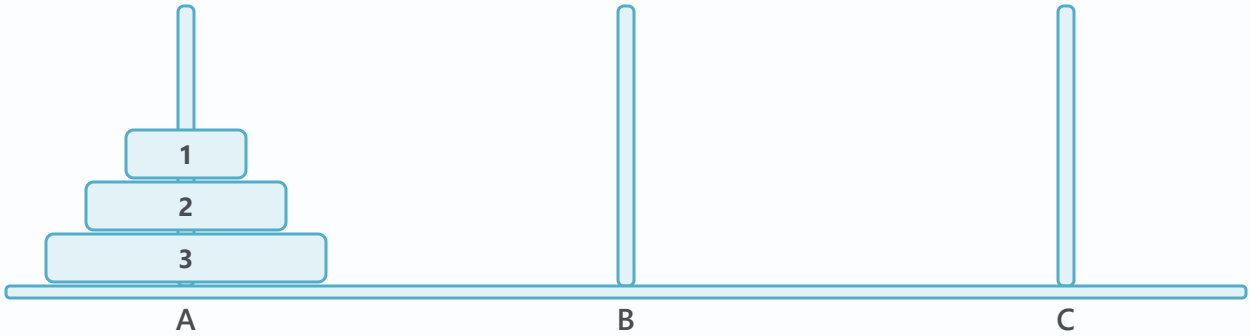
A classic problem in computer science and mathematics: move a stack of disks from the source peg to the target peg while obeying the rules.

- Each move carries **one disk** — the top disk of a peg.
- A disk may only go onto a **larger disk** or an empty peg.
- Goal: using the spare peg, move all disks from the source (A) to the target (C).



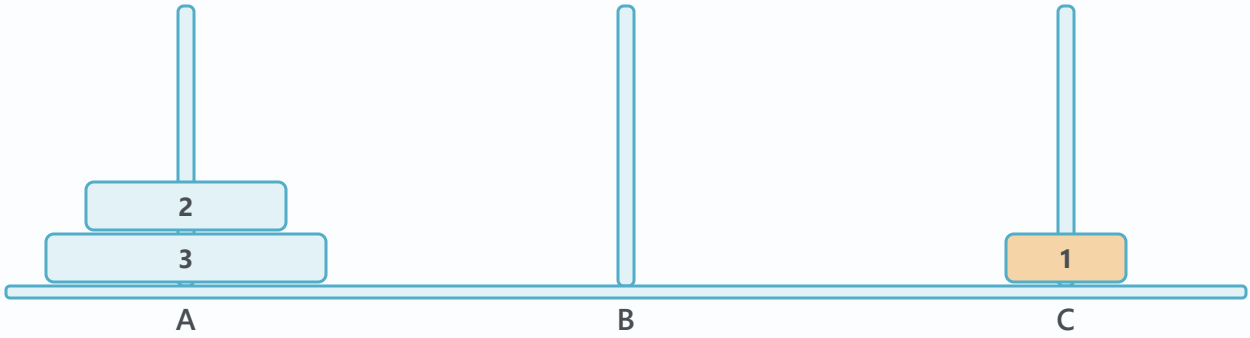
The recursive idea: move n−1 disks from A to B → move the largest disk from A to C → move the n−1 disks from B to C. Number of moves: T(n) = 2T(n−1) + 1 = $\mathbf{2^n - 1}$.

Towers of Hanoi: 7 Moves with 3 Disks



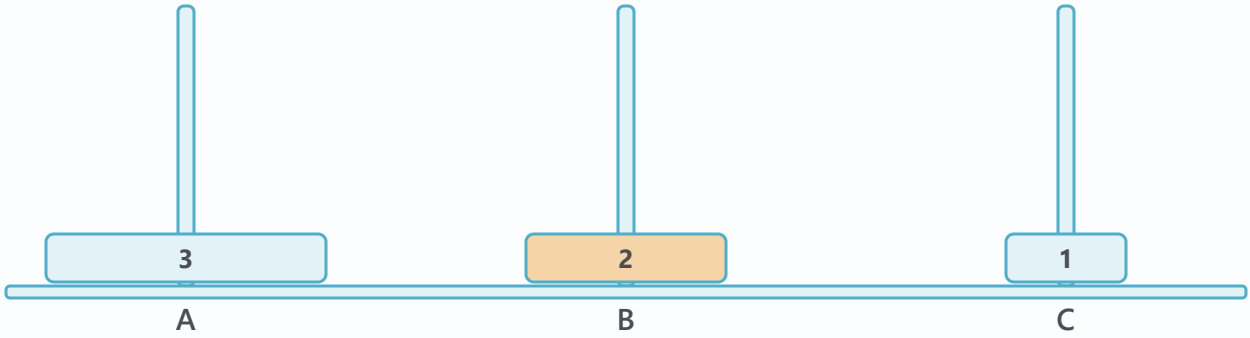
Start: 3 disks on peg A (largest at the bottom). Goal: move them all to C.

Towers of Hanoi: 7 Moves with 3 Disks



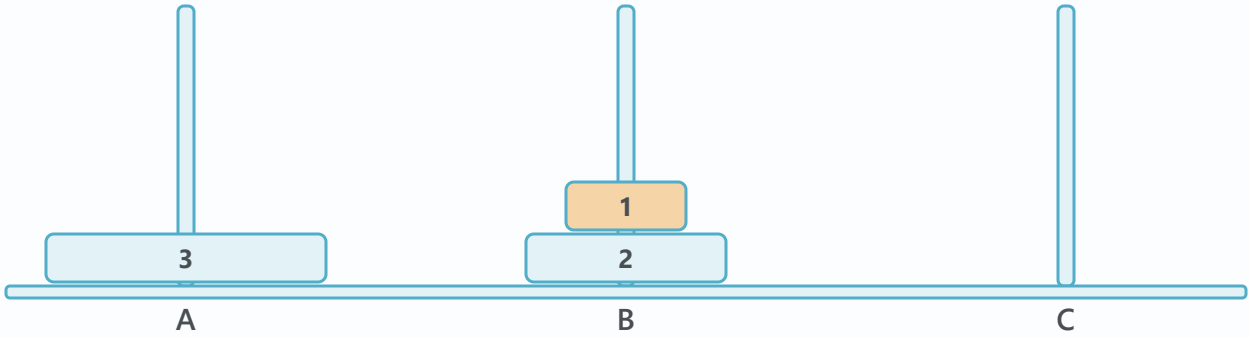
Move 1: disk 1, **A** → **C**.

Towers of Hanoi: 7 Moves with 3 Disks



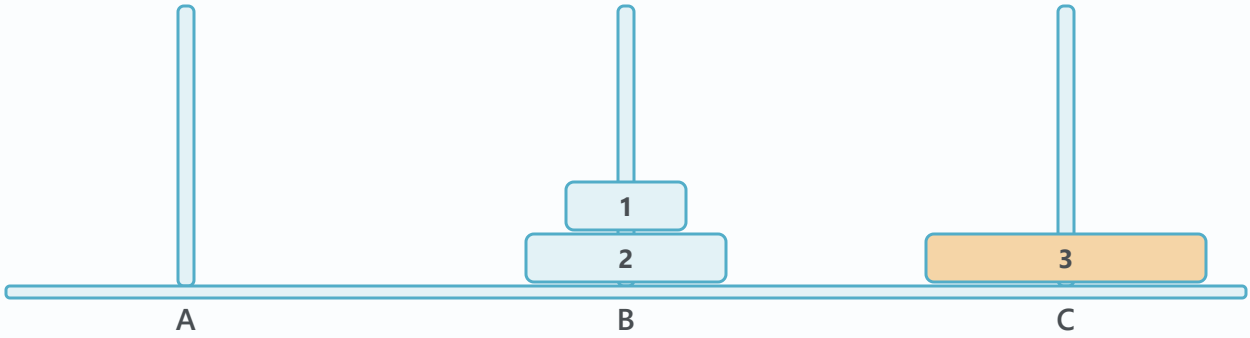
Move 2: disk 2, **A** → **B**.

Towers of Hanoi: 7 Moves with 3 Disks



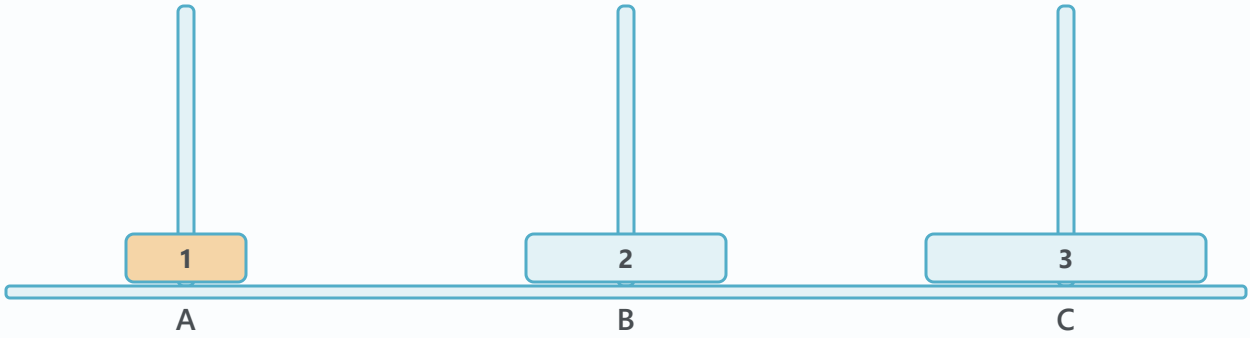
Move 3: disk 1, **C** → **B**.

Towers of Hanoi: 7 Moves with 3 Disks



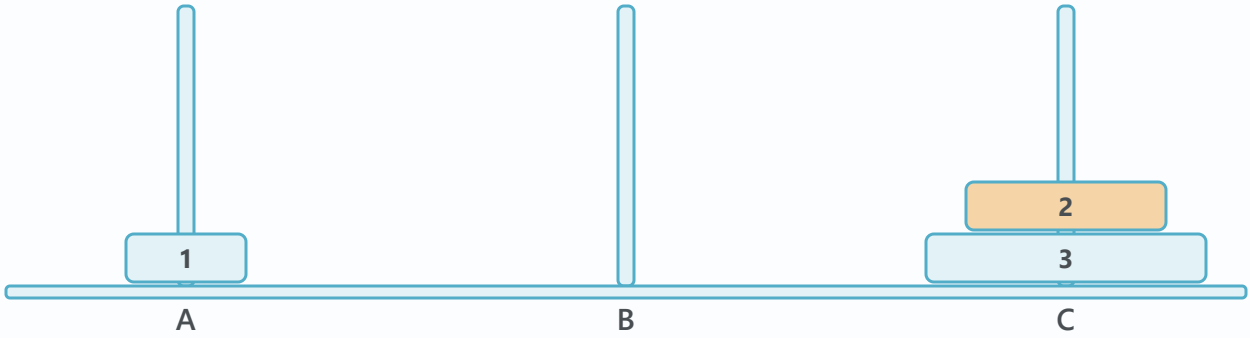
Move 4: disk 3, **A** → **C**.

Towers of Hanoi: 7 Moves with 3 Disks



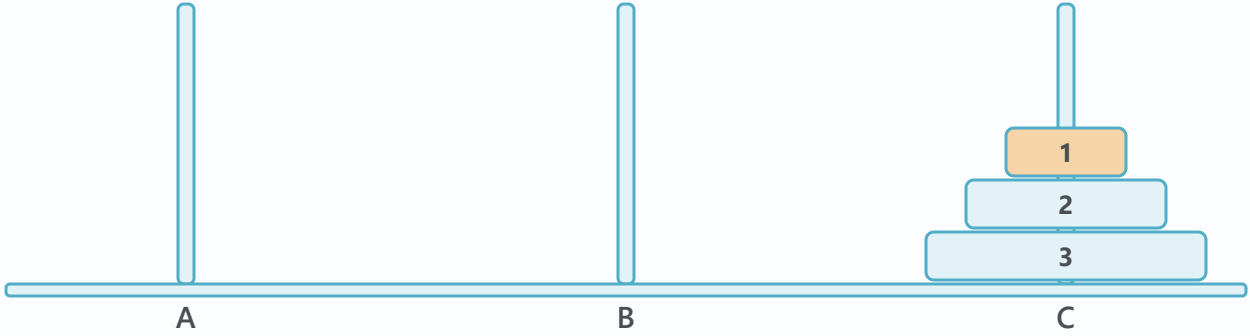
Move 5: disk 1, **B** → **A**.

Towers of Hanoi: 7 Moves with 3 Disks



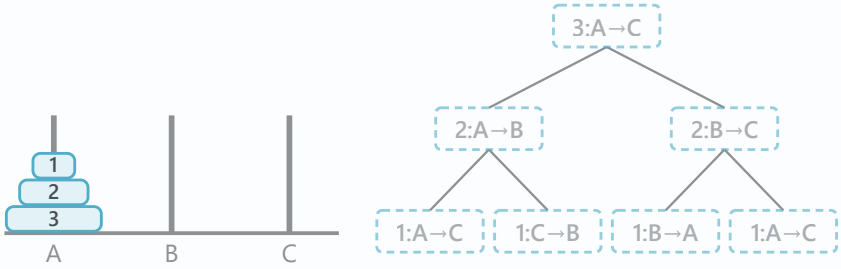
Move 6: disk 2, **B** → **C**.

Towers of Hanoi: 7 Moves with 3 Disks



Move 7: disk 1, **A** → **C**. Done: $2^3 - 1 =$ **7 moves**.

Code Trace: Recursive `hanoi(3, A, B, C)`



main calls `hanoi(3, 'A', 'B', 'C')`. Each box in the tree is a call: **n:from→to**.

Step 1 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
▶  hanoi(3, 'A', 'B', 'C');
```

VARIABLES

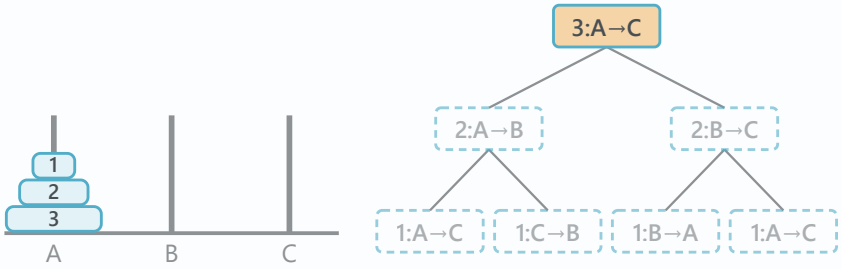
—

CALL STACK

`main()`

OUTPUT

Code Trace: Recursive `hanoi(3, A, B, C)`



hanoi(3,A,B,C): a new frame is pushed. n = 3 ≠ 1 → first the top 2 disks must move from peg A to peg B.

Step 2 / 19

```
1  void hanoi(int n, char from, char via, char to) {
▶    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

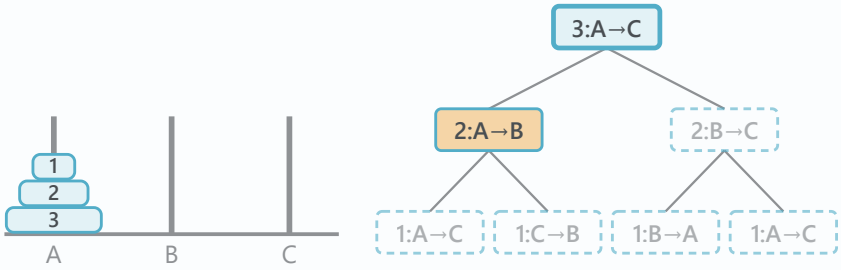
CALL STACK

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

`main()`

OUTPUT

Code Trace: Recursive `hanoi(3, A, B, C)`



Line 6: hanoi(2,A,C,B): a new frame is pushed. n = 2 ≠ 1 → first the top disk must move from peg A to peg C.

Step 3 / 19

```
1  void hanoi(int n, char from, char via, char to) {
▶    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

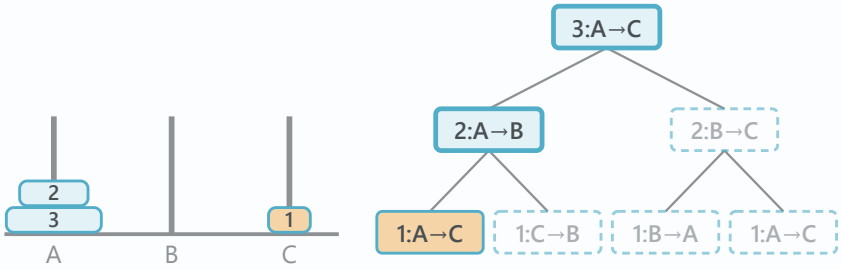
`n = 2` `from = A` `via = C` `to = B`

CALL STACK

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

OUTPUT

Code Trace: Recursive `hanoi(3, A, B, C)`



Line 6: hanoi(1,A,B,C): a new frame is pushed. **n == 1** → base case: disk 1, **A** → **C**.

Step 4 / 19

```
1  void hanoi(int n, char from, char via, char to) {
▶    if (n == 1) {
▶      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

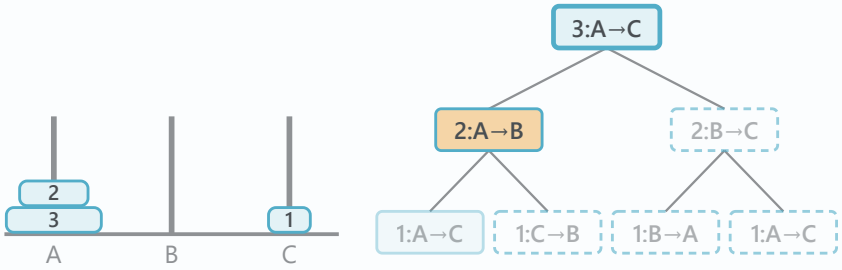
CALL STACK

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

OUTPUT

```
1: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



return: the frame is popped and control goes back to the caller.

Step 5 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
▶      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

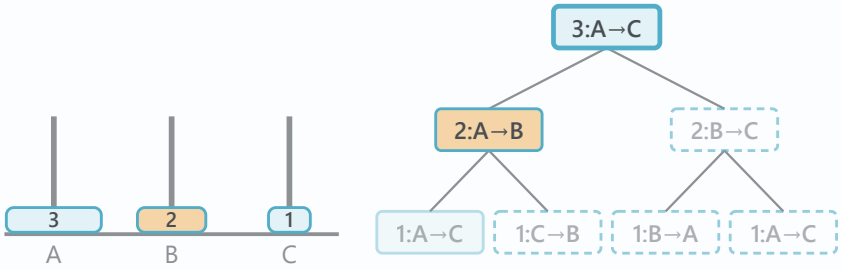
CALL STACK

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

OUTPUT

```
1: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



The left subcall is done; only disk 2 is left on peg A → **Disk 2: A → B**.

Step 6 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
▶    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

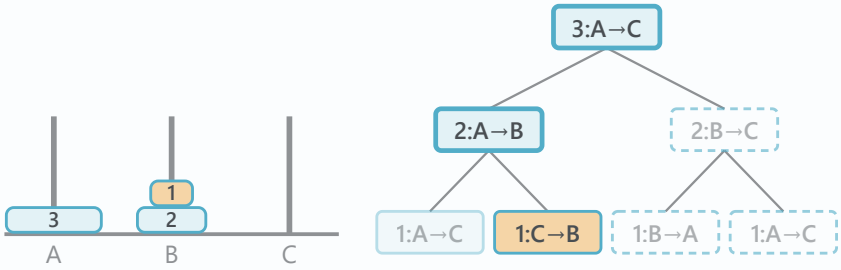
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
```

Code Trace: Recursive `hanoi(3, A, B, C)`



Line 8: hanoi(1,C,A,B): a new frame is pushed. **n == 1** → base case: disk 1, **C** → **B**.

Step 7 / 19

```
1  void hanoi(int n, char from, char via, char to) {
▶    if (n == 1) {
▶      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

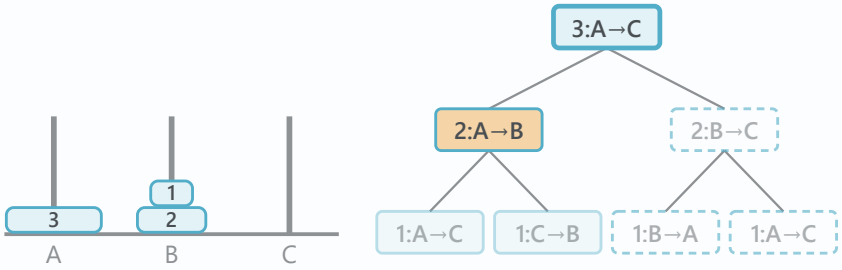
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
```

Code Trace: Recursive `hanoi(3, A, B, C)`



return: the frame is popped and control goes back to the caller.

Step 8 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
▶      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

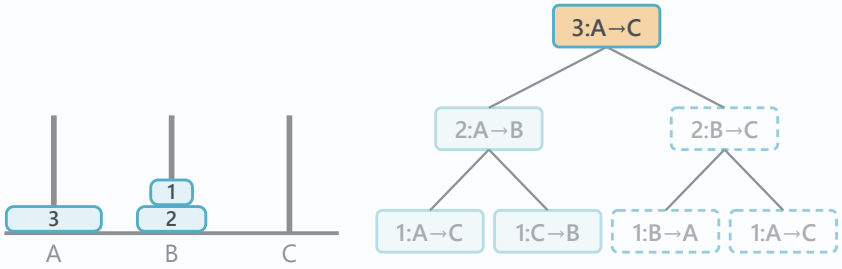
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
```

Code Trace: Recursive `hanoi(3, A, B, C)`



hanoi(2,A,C,B) ends; its frame is popped.

Step 9 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
▶ }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

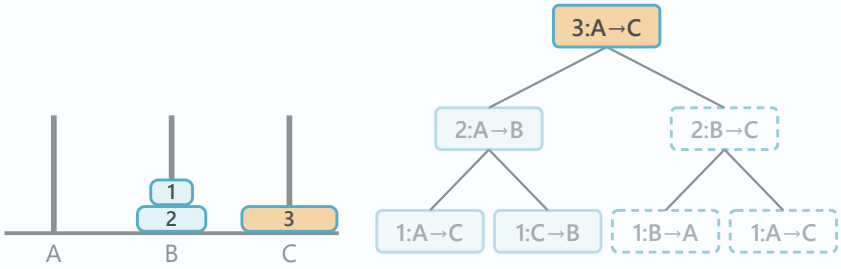
CALL STACK

- **hanoi(2,A,C,B)**
- hanoi(3,A,B,C)
- main()

OUTPUT

```
1: A→C
2: A→B
1: C→B
```

Code Trace: Recursive `hanoi(3, A, B, C)`



The left subcall is done; only disk 3 is left on peg A → **Disk 3: A → C**.

Step 10 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
▶    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

CALL STACK

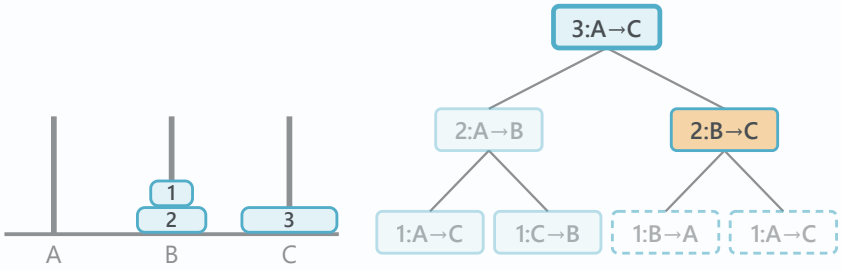
`hanoi(3,A,B,C)`

`main()`

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



Line 8: hanoi(2,B,A,C): a new frame is pushed. n = 2 ≠ 1 → first the top disk must move from peg B to peg A.

Step 11 / 19

```
1  void hanoi(int n, char from, char via, char to) {
▶    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

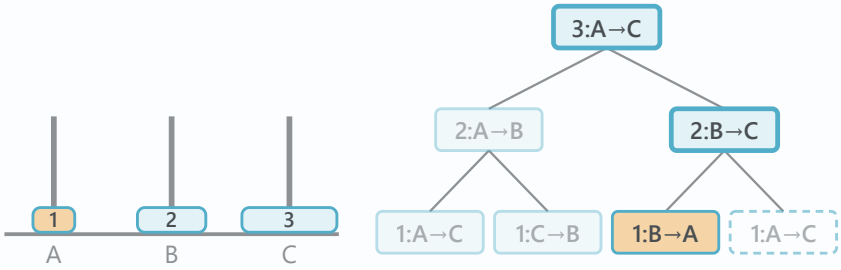
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



Line 6: hanoi(1,B,C,A): a new frame is pushed. **n == 1** → base case: disk 1, **B** → **A**.

Step 12 / 19

```
1  void hanoi(int n, char from, char via, char to) {
▶    if (n == 1) {
▶      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

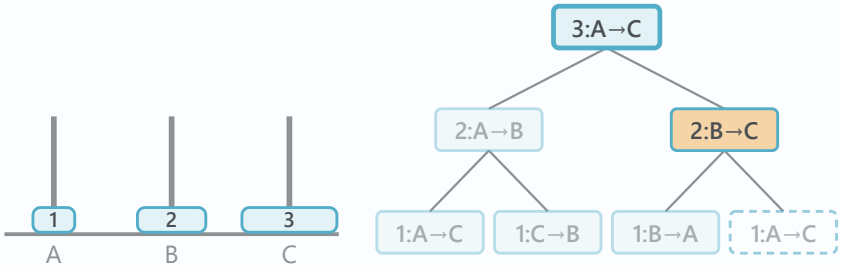
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
```

Code Trace: Recursive `hanoi(3, A, B, C)`



return: the frame is popped and control goes back to the caller.

Step 13 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
▶      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

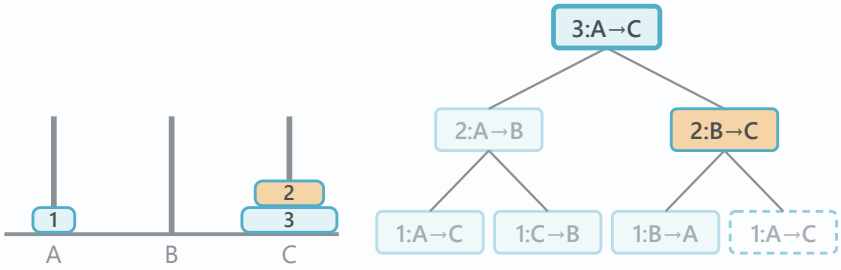
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
```

Code Trace: Recursive `hanoi(3, A, B, C)`



The left subcall is done; only disk 2 is left on peg B → **Disk 2: B → C**.

Step 14 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
▶    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

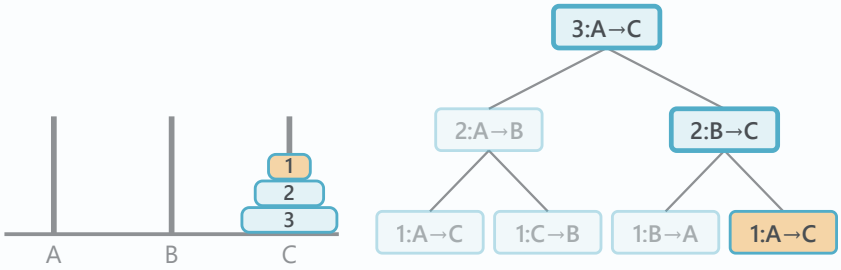
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
2: B→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



Line 8: hanoi(1,A,B,C): a new frame is pushed. **n == 1** → base case: disk 1, **A** → **C**.

Step 15 / 19

```
1  void hanoi(int n, char from, char via, char to) {
▶    if (n == 1) {
▶      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

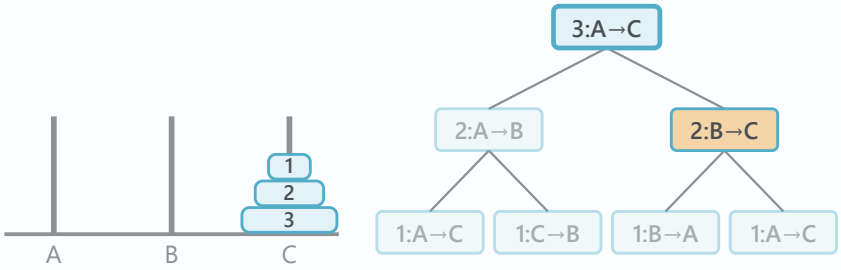
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
2: B→C
1: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



return: the frame is popped and control goes back to the caller.

Step 16 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
▶      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

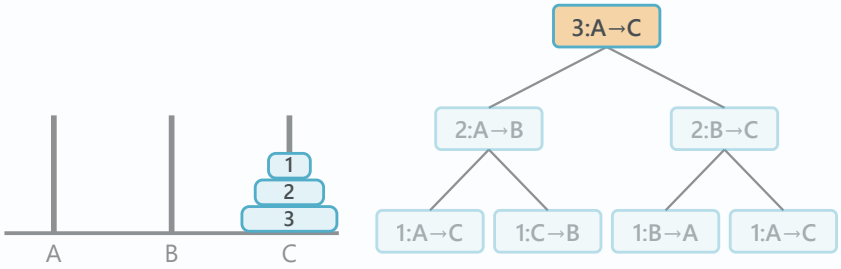
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
2: B→C
1: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



hanoi(2,B,A,C) ends; its frame is popped.

Step 17 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
▶ }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

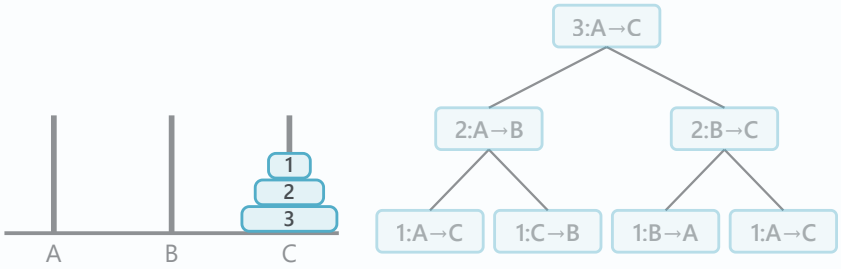
CALL STACK

- **hanoi(2,B,A,C)**
- hanoi(3,A,B,C)
- main()

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
2: B→C
1: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



hanoi(3,A,B,C) ends; its frame is popped. 7 calls in the tree = 7 moves = $2^3 - 1$.

Step 18 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
▶ }
10 // in main:
11 hanoi(3, 'A', 'B', 'C');
```

VARIABLES

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

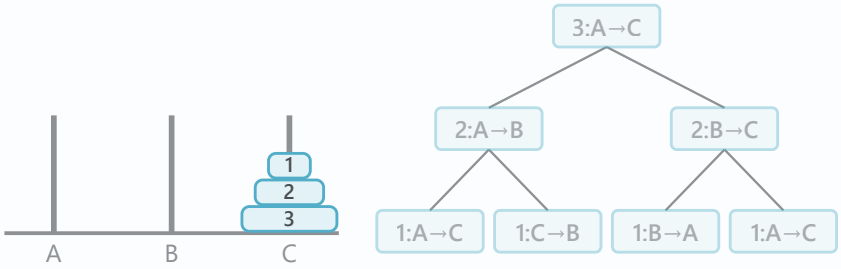
CALL STACK

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

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
2: B→C
1: A→C
```

Code Trace: Recursive `hanoi(3, A, B, C)`



Back in main: all disks are on C. The call stack is at most n = 3 deep (+ main) → memory **O(n)**, time **O(2ⁿ)**.

Step 19 / 19

```
1  void hanoi(int n, char from, char via, char to) {
2    if (n == 1) {
3      System.out.println("1: " + from + "→" + to);
4      return;
5    }
6    hanoi(n - 1, from, to, via);
7    System.out.println(n + ": " + from + "→" + to);
8    hanoi(n - 1, via, from, to);
9  }
10 // in main:
▶  hanoi(3, 'A', 'B', 'C');
```

VARIABLES

—

CALL STACK

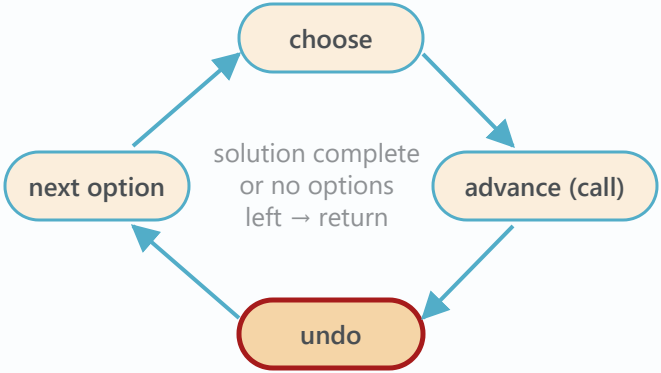
`main()`

OUTPUT

```
1: A→C
2: A→B
1: C→B
3: A→C
1: B→A
2: B→C
1: A→C
```

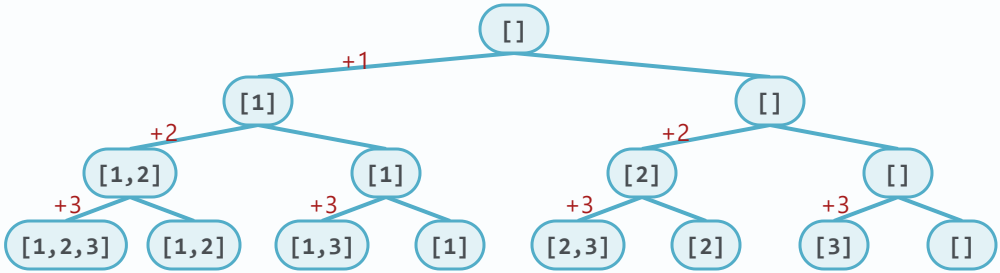
Backtracking

- The solution is built **step by step**; at each step one of the possible choices is tried.
- Making a choice moves on to the next step: a **recursive call**.
- When the call returns, the choice is **undone** and the next option is tried, so every branch starts from the same state.
- If a choice cannot lead to a solution, that branch is abandoned without going deeper.
- Examples: listing subsets or permutations, finding a path through a maze, sudoku, n-queens.



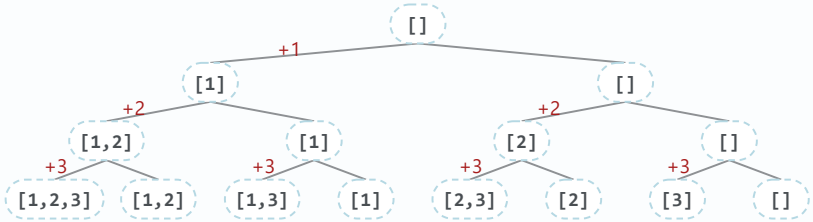
As in Hanoi, each call makes several recursive calls; the call tree is the tree of all choices tried (the **decision tree**).

Listing Subsets



- Two decisions per item: **take** it or **leave** it. The left branch takes the item (+1, +2, +3); the right branch doesn't.
- Level i decides a[i]; at a leaf (i = n) all decisions are made → s is a subset.
- n items → 2^n leaves = 2^n subsets; $2^{n+1} - 1$ calls (n = 3: 8 subsets, 15 calls).
- Output order: `[1, 2, 3] [1, 2] [1, 3] [1] [2, 3] [2] [3] []`

Code Trace: `subsets` of {1, 2, 3}



main calls `subsets(a, 0, s)` with a = {1, 2, 3} and an empty s.

Step 1 / 24

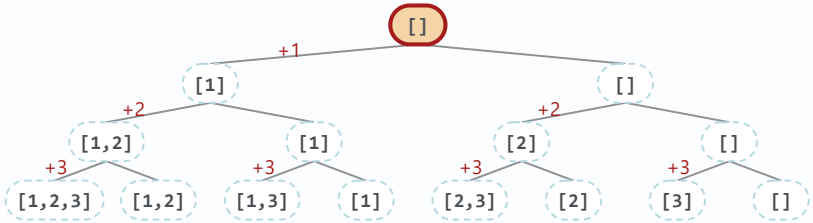
```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {              // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                      // take a[i]
6    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);           // undo
8    subsets(a, i + 1, s);             // try without a[i]
9  }
10 // in main:
▶  subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

OUTPUT

CALL STACK

`main()`

Code Trace: `subsets` of {1, 2, 3}



i = 0: **take** a[0] = 1 (s = [1]) and go down the left branch.

Step 2 / 24

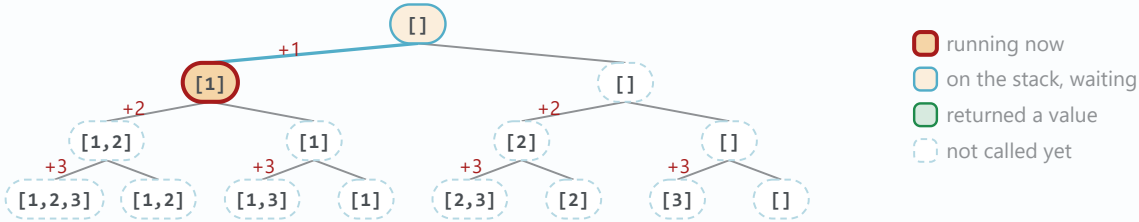
```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                // base case
3      System.out.print(s + " "); return;
4    }
▶    s.add(a[i]);                        // take a[i]
▶    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);             // undo
8    subsets(a, i + 1, s);               // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

OUTPUT

CALL STACK

- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



i = 1: **take** a[1] = 2 (s = [1, 2]) and go down the left branch.

Step 3 / 24

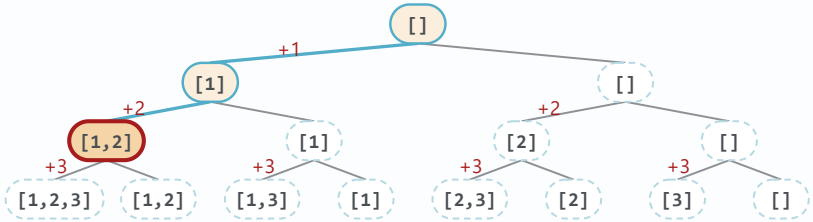
```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                // base case
3      System.out.print(s + " "); return;
4    }
▶    s.add(a[i]);                        // take a[i]
▶    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);             // undo
8    subsets(a, i + 1, s);               // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

OUTPUT

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 2: **take** a[2] = 3 (s = [1, 2, 3]) and go down the left branch.

Step 4 / 24

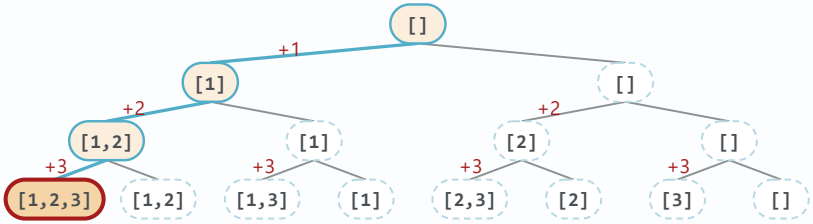
```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
▶    s.add(a[i]);                          // take a[i]
▶    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);               // undo
8    subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

OUTPUT

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 3 = a.length: **base case**. All decisions are made → print `[1, 2, 3]`.

Step 5 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶    if (i == a.length) {                  // base case
▶      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                          // take a[i]
6    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);               // undo
8    subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

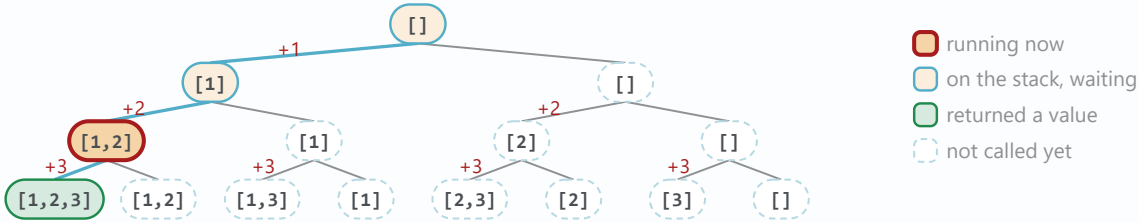
OUTPUT

```
[1, 2, 3]
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



Back from the left branch. **Undo** 3 (s = [1, 2]); with the same i, go down the right branch to the subsets without 3.

Step 6 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                          // take a[i]
6    subsets(a, i + 1, s);
▶    s.remove(s.size() - 1);               // undo
▶    subsets(a, i + 1, s);                // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

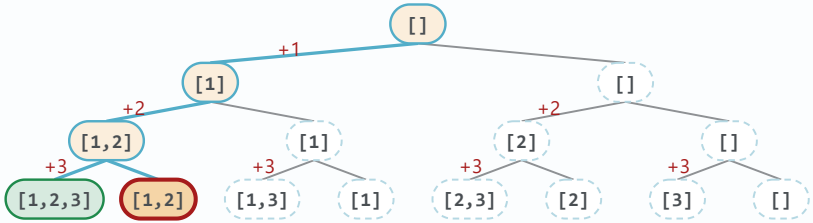
OUTPUT

```
[1, 2, 3]
```

CALL STACK

- **subsets(2)**
- subsets(1)
- subsets(0)
- main()

Code Trace: `subsets` of {1, 2, 3}



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

i = 3 = a.length: **base case**. All decisions are made → print `[1, 2]`.

Step 7 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶    if (i == a.length) {                  // base case
▶      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                          // take a[i]
6    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);               // undo
8    subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

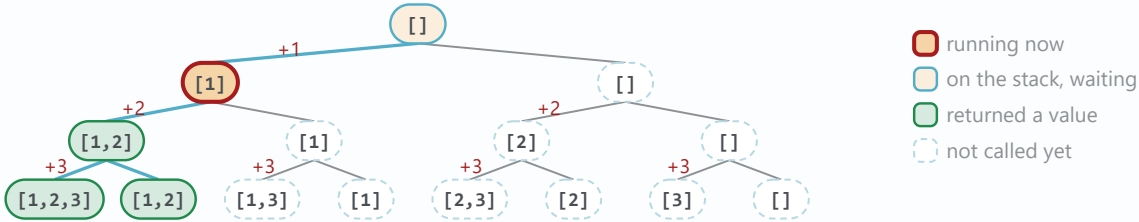
OUTPUT

```
[1, 2, 3] [1, 2]
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



Back from the left branch. **Undo** 2 (s = [1]); with the same i, go down the right branch to the subsets without 2.

Step 8 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                        // take a[i]
6    subsets(a, i + 1, s);
▶    s.remove(s.size() - 1);             // undo
▶    subsets(a, i + 1, s);               // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

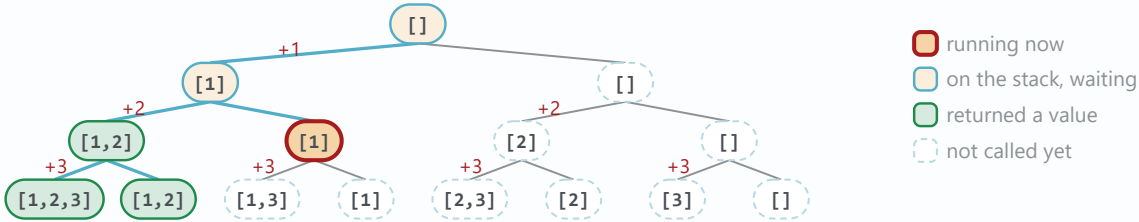
OUTPUT

```
[1, 2, 3] [1, 2]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 2: **take** a[2] = 3 (s = [1, 3]) and go down the left branch.

Step 9 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
▶    s.add(a[i]);                          // take a[i]
▶    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);               // undo
8    subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

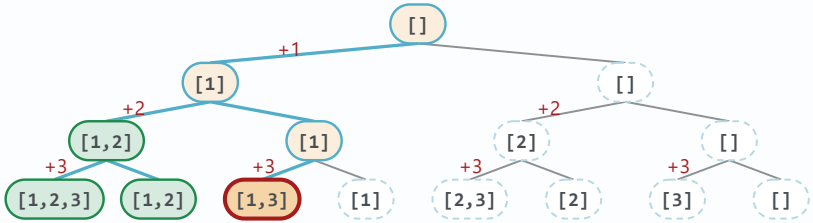
OUTPUT

```
[1, 2, 3] [1, 2]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 3 = a.length: **base case**. All decisions are made → print `[1, 3]`.

Step 10 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶     if (i == a.length) {                  // base case
▶       System.out.print(s + " "); return;
4     }
5     s.add(a[i]);                          // take a[i]
6     subsets(a, i + 1, s);
7     s.remove(s.size() - 1);               // undo
8     subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

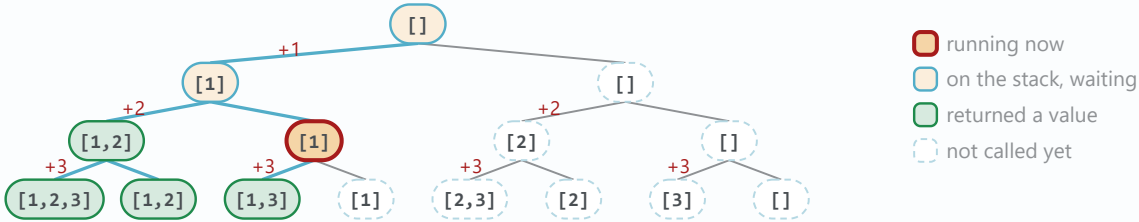
OUTPUT

```
[1, 2, 3] [1, 2] [1, 3]
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



Back from the left branch. **Undo** 3 (s = [1]); with the same i, go down the right branch to the subsets without 3.

Step 11 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                          // take a[i]
6    subsets(a, i + 1, s);
▶    s.remove(s.size() - 1);               // undo
▶    subsets(a, i + 1, s);                // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

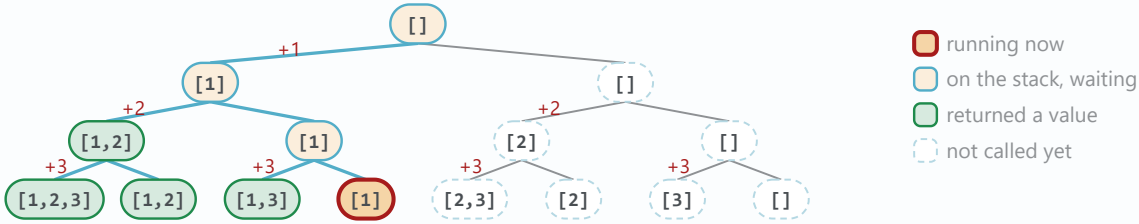
OUTPUT

```
[1, 2, 3] [1, 2] [1, 3]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 3 = a.length: **base case**. All decisions are made → print `[1]`.

Step 12 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶     if (i == a.length) {                  // base case
▶       System.out.print(s + " "); return;
4     }
5     s.add(a[i]);                          // take a[i]
6     subsets(a, i + 1, s);
7     s.remove(s.size() - 1);               // undo
8     subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

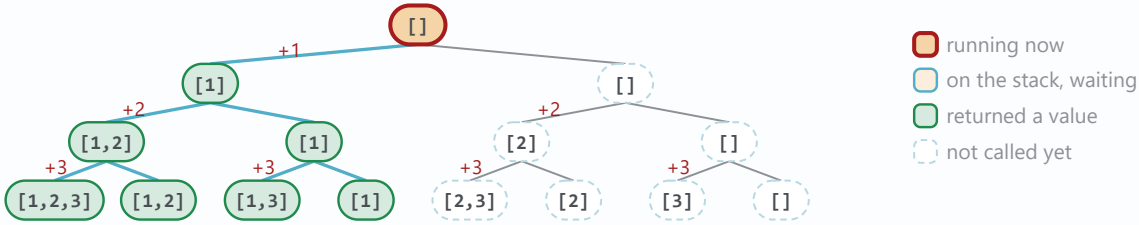
OUTPUT

```
[1, 2, 3] [1,
2] [1, 3] [1]
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



Back from the left branch. **Undo** 1 (s = []); with the same i, go down the right branch to the subsets without 1.

Step 13 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                        // take a[i]
6    subsets(a, i + 1, s);
▶    s.remove(s.size() - 1);             // undo
▶    subsets(a, i + 1, s);               // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

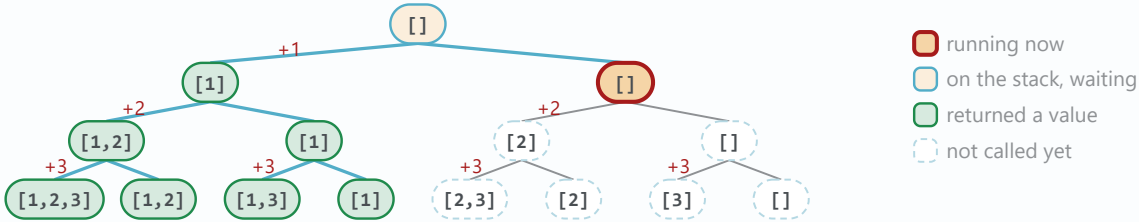
OUTPUT

```
[1, 2, 3] [1,
2] [1, 3] [1]
```

CALL STACK

- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



i = 1: **take** a[1] = 2 (s = [2]) and go down the left branch.

Step 14 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
▶    s.add(a[i]);                          // take a[i]
▶    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);               // undo
8    subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

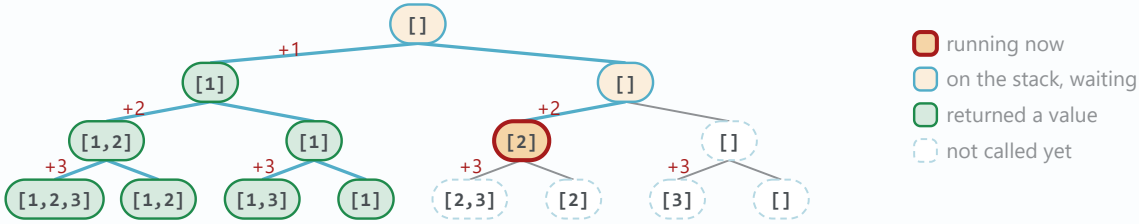
OUTPUT

```
[1, 2, 3] [1,
2] [1, 3] [1]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 2: **take** a[2] = 3 (s = [2, 3]) and go down the left branch.

Step 15 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
▶    s.add(a[i]);                          // take a[i]
▶    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);               // undo
8    subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

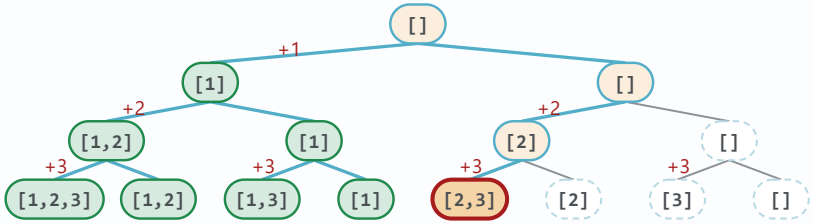
OUTPUT

```
[1, 2, 3] [1,
2] [1, 3] [1]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 3 = a.length: **base case**. All decisions are made → print `[2, 3]`.

Step 16 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶     if (i == a.length) {                  // base case
▶       System.out.print(s + " "); return;
4     }
5     s.add(a[i]);                          // take a[i]
6     subsets(a, i + 1, s);
7     s.remove(s.size() - 1);               // undo
8     subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

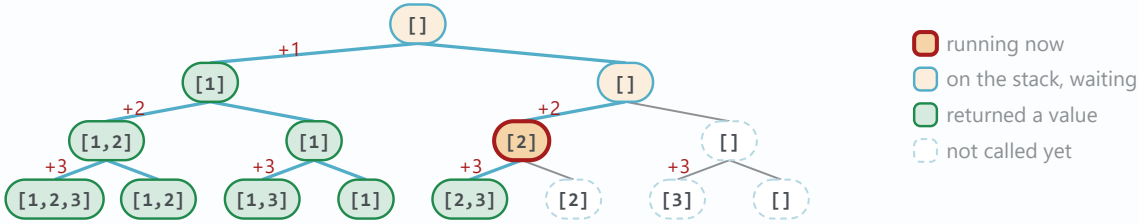
OUTPUT

```
[1, 2, 3] [1, 2] [1, 3] [1] [2, 3]
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



Back from the left branch. **Undo** 3 (s = [2]); with the same i, go down the right branch to the subsets without 3.

Step 17 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                          // take a[i]
6    subsets(a, i + 1, s);
▶    s.remove(s.size() - 1);               // undo
▶    subsets(a, i + 1, s);                // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

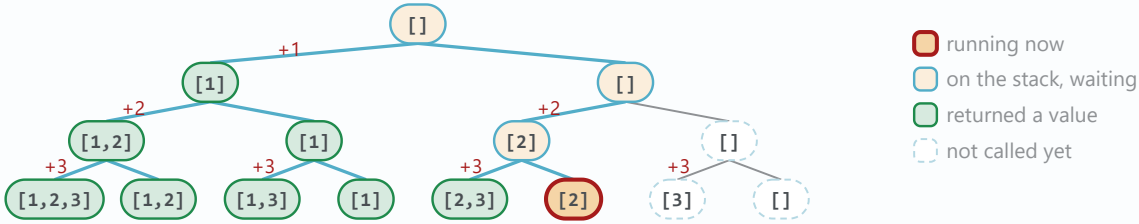
OUTPUT

```
[1, 2, 3] [1, 2] [1, 3] [1] [2, 3]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 3 = a.length: **base case**. All decisions are made → print `[2]`.

Step 18 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶     if (i == a.length) {                  // base case
▶       System.out.print(s + " "); return;
4     }
5     s.add(a[i]);                          // take a[i]
6     subsets(a, i + 1, s);
7     s.remove(s.size() - 1);               // undo
8     subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

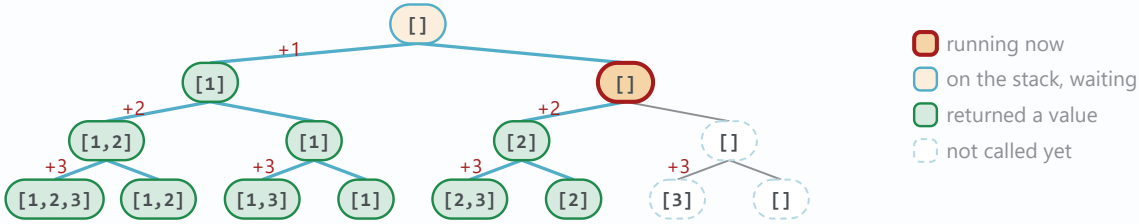
OUTPUT

```
[1, 2, 3] [1, 2] [1, 3] [1] [2, 3] [2]
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



Back from the left branch. **Undo** 2 (s = []); with the same i, go down the right branch to the subsets without 2.

Step 19 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                          // take a[i]
6    subsets(a, i + 1, s);
▶    s.remove(s.size() - 1);               // undo
▶    subsets(a, i + 1, s);                // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

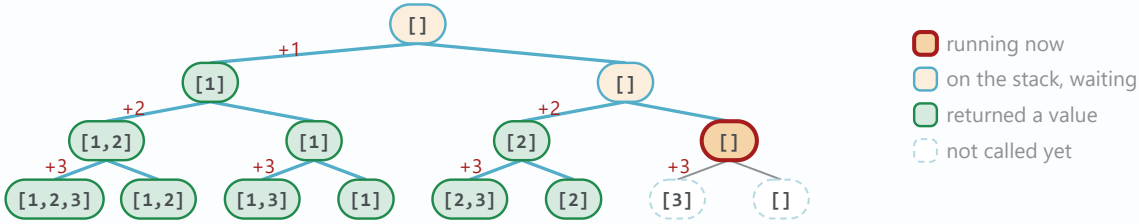
OUTPUT

```
[1, 2, 3] [1,
2] [1, 3] [1]
[2, 3] [2]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 2: **take** a[2] = 3 (s = [3]) and go down the left branch.

Step 20 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
▶    s.add(a[i]);                          // take a[i]
▶    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);               // undo
8    subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

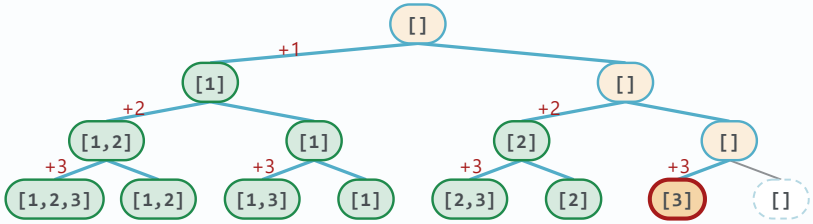
OUTPUT

```
[1, 2, 3] [1, 2] [1, 3] [1] [2, 3] [2]
```

CALL STACK

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

Code Trace: `subsets` of {1, 2, 3}



i = 3 = a.length: **base case**. All decisions are made → print `[3]`.

Step 21 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶     if (i == a.length) {                  // base case
▶       System.out.print(s + " "); return;
4     }
5     s.add(a[i]);                          // take a[i]
6     subsets(a, i + 1, s);
7     s.remove(s.size() - 1);               // undo
8     subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

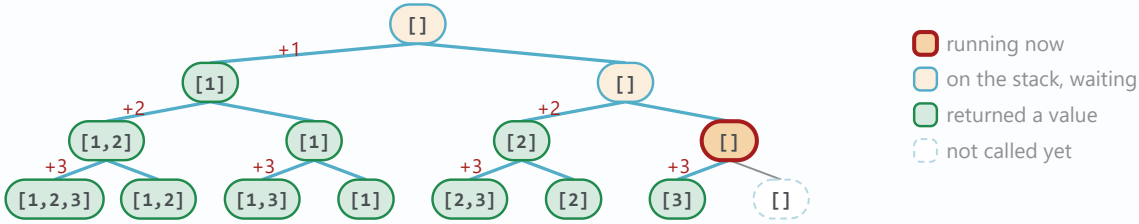
OUTPUT

```
[1, 2, 3] [1,
2] [1, 3] [1]
[2, 3] [2] [3]
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



Back from the left branch. **Undo** 3 (s = []); with the same i, go down the right branch to the subsets without 3.

Step 22 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                  // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                          // take a[i]
6    subsets(a, i + 1, s);
▶    s.remove(s.size() - 1);               // undo
▶    subsets(a, i + 1, s);                // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

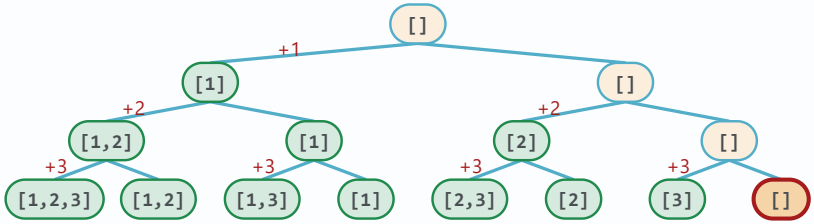
OUTPUT

```
[1, 2, 3] [1,
2] [1, 3] [1]
[2, 3] [2] [3]
```

CALL STACK

- **subsets(2)**
- subsets(1)
- subsets(0)
- main()

Code Trace: `subsets` of {1, 2, 3}



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

i = 3 = a.length: **base case**. All decisions are made → print `[]`.

Step 23 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
▶     if (i == a.length) {                  // base case
▶       System.out.print(s + " "); return;
4     }
5     s.add(a[i]);                          // take a[i]
6     subsets(a, i + 1, s);
7     s.remove(s.size() - 1);               // undo
8     subsets(a, i + 1, s);                 // try without a[i]
9  }
10 // in main:
11 subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

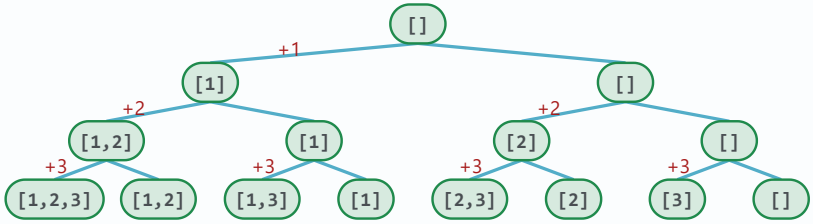
OUTPUT

```
[1, 2, 3] [1, 2] [1, 3] [1] [2, 3] [2] [3] []
```

CALL STACK

- `subsets(3)`
- `subsets(2)`
- `subsets(1)`
- `subsets(0)`
- `main()`

Code Trace: `subsets` of {1, 2, 3}



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

Done: **8 subsets**, 15 calls. The stack never held more than main + 4 frames.

Step 24 / 24

```
1  void subsets(int[] a, int i, List<Integer> s) {
2    if (i == a.length) {                // base case
3      System.out.print(s + " "); return;
4    }
5    s.add(a[i]);                        // take a[i]
6    subsets(a, i + 1, s);
7    s.remove(s.size() - 1);             // undo
8    subsets(a, i + 1, s);               // try without a[i]
9  }
10 // in main:
▶  subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

OUTPUT

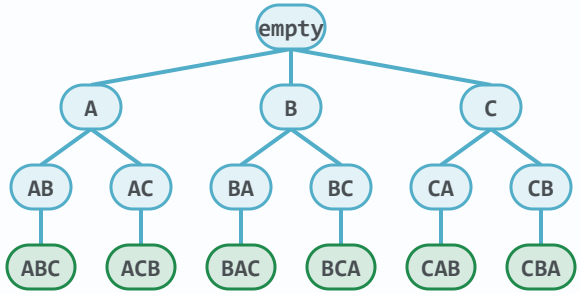
```
[1, 2, 3] [1, 2] [1, 3] [1] [2, 3] [2] [3] []
```

CALL STACK

`main()`

Listing Permutations

```
void perm(char[] c, boolean[] used, StringBuilder s) {
  if (s.length() == c.length) {  // base case
    System.out.print(s + " "); return;
  }
  for (int k = 0; k < c.length; k++) {
    if (used[k]) continue;       // letter already used
    used[k] = true; s.append(c[k]);  // choose
    perm(c, used, s);
    used[k] = false;             // undo
    s.deleteCharAt(s.length() - 1);
  }
}
```



- Output: `ABC ACB BAC BCA CAB CBA`
- Each level chooses one of the remaining letters: $3 \cdot 2 \cdot 1 = 3! = 6$ leaves.
- Number of calls: $1 + 3 + 6 + 6 = 16$.

A call makes as many recursive calls as there are letters left: **multiple recursion**. `used[]` and `s` are shared by all calls, which is why each choice is undone after it has been tried.

Recurrence Relations

- **T(n)**: the total work a method does on an input of size n. The cost of a recursive method is written in terms of its own cost on smaller inputs: a **recurrence relation**.
- Three questions to write one: What does the base case cost (usually a constant c)? How many recursive calls are made, and how much smaller are their inputs? How much work is done outside the calls?

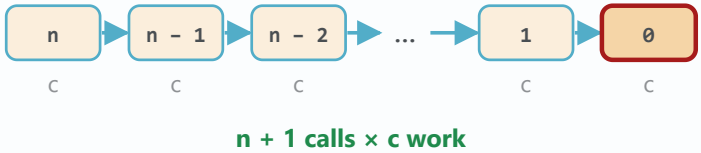
Method	Calls	Recurrence	Solution
factorial, sum, countdown	one call, $n - 1$	$T(n) = T(n - 1) + c$	$O(n)$
reverse	one call, $n - 2$	$T(n) = T(n - 2) + c$	$O(n)$
binarySearch, fastPower	one call, $n / 2$	$T(n) = T(n/2) + c$	$O(\log n)$
hanoi	two calls, $n - 1$	$T(n) = 2T(n - 1) + c$	$O(2^n)$
fib	two calls, $n - 1$ and $n - 2$	$T(n) = T(n - 1) + T(n - 2) + c$	exponential

To solve a recurrence, draw the **call tree**: if every node does constant work, the total work $\approx c \cdot$ (number of calls).

RECURRENCE RELATIONS

T(n) = T(n − 1) + c

```
T(n) = T(n − 1) + c
     = T(n − 2) + 2c
     = T(n − 3) + 3c
     …
     = T(0) + n · c
     = c + n · c   →  O(n)
```



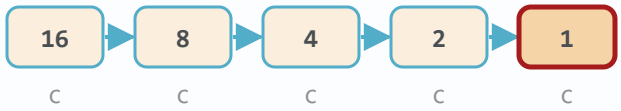
n + 1 calls × c work

- Solve it by **unrolling**: each step adds another c and lowers n. After n steps we reach the base case.
- The call tree is a chain with no branches: n + 1 calls. The whole chain is on the stack at once → memory is **O(n)** too.
- Examples: factorial, sum, countdown, slow exponentiation, the length of a linked list. For `reverse`, T(n) = T(n − 2) + c: half as many calls, still O(n).

RECURRENCE RELATIONS

$T(n) = T(n/2) + c$

```
T(n) = T(n/2) + c
     = T(n/4) + 2c
     = T(n/8) + 3c
     …
     = T(n/2^k) + k · c
```



$\log_2 16 + 1 = 5$ calls

We stop when $n/2^k = 1$, that is, $k = \log_2 n$.
$T(n) = T(1) + c \cdot \log_2 n \rightarrow$ **O(log n)**

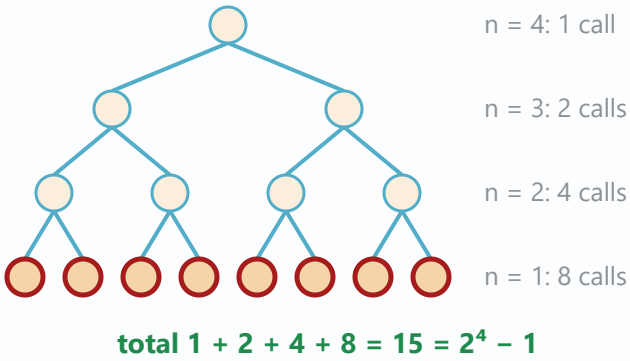
n	16	1,024	1,000,000
halving chain (calls)	5	11	20

Examples: binary search, fast exponentiation. A thousand times more input adds only about 10 calls, and since the chain is log n deep, memory is O(log n).

RECURRENCE RELATIONS

$T(n) = 2T(n - 1) + c$

- Each call makes **two** calls of size $n - 1$, so the number of calls doubles at every level of the call tree.
- Level k has 2^k calls, for levels $0 \dots n - 1$: $1 + 2 + 4 + \dots + 2^{n-1} = \mathbf{2^n - 1}$ calls $\rightarrow$ $\mathbf{O(2^n)}$.
- Hanoi: exactly $2^n - 1$ moves. That is 1,048,575 for $n = 20$ and about $1.8 \cdot 10^{19}$ for $n = 64$.
- fib: since $T(n - 2) \le T(n - 1)$, $T(n) \le 2T(n - 1) + c \rightarrow$ upper bound $O(2^n)$; the true growth is about 1.6^n.

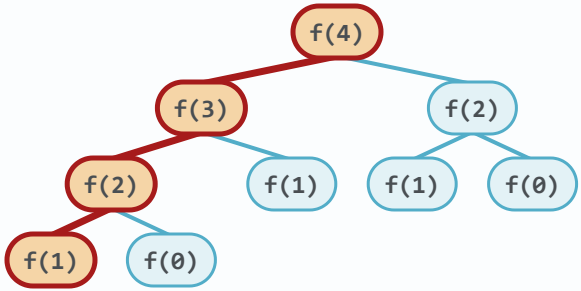


total $1 + 2 + 4 + 8 = 15 = 2^4 - 1$

The tree grows exponentially, but its **depth is n**: at most n frames are on the stack at once.

Space Cost: The Deepest Chain of Calls

- A call **finishes** before its sibling starts, so the stack only ever holds **one path** from the root to some node.
- Memory = deepest chain of calls × frame size. What matters is the **depth**, not the total number of calls.
- `fib(4)` : 9 calls, but never more than 4 frames at once (+ main).



Call	Total calls	Deepest chain	Time	Space
factorial(4)	4	4	O(n)	O(n)
fib(40)	331,160,281	40	exponential	O(n)
hanoi(3, …)	7	3	$O(2^n)$	O(n)
subsets, n = 3	15	4	$O(n \cdot 2^n)$	O(n)

Tail Recursion

- A method is **tail recursive** if the recursive call is its **last action** — nothing is left to do after the call returns.
- The partial result travels down in an **accumulator** parameter (acc); the multiplication happens **before** the call. First call: `factTail(n, 1)`.

Not tail recursive

```
int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
  // multiply after the call returns
}
```

Tail recursive

```
int factTail(int n, int acc) {
  if (n <= 1) return acc;
  return factTail(n - 1, n * acc);
  // last action: the call (multiply first)
}
```

Tail recursive: countdown, reverse, binarySearch, factTail. Not tail recursive: factorial, sum, fib, countUp (work remains after the call).

Code Trace: `factTail(4, 1)`



main calls `factTail(4, 1)`; acc starts at 1.

Step 1 / 9

```
1 int factTail(int n, int acc) {
2   if (n <= 1) return acc;   // base case
3   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
▶ System.out.println(factTail(4, 1));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `factTail(4, 1)`

`factTail(4, 1)` call: (3, 4)

`factTail(3, 4)` call: (2, 12)

`factTail(2, 12)` call: (1, 24)

`factTail(1, 24)` return acc

→ call · ← return

n = 4 > 1 → multiply **first**: 4 · 1 = 4. Then call `factTail(3, 4)`.

Step 2 / 9

```
1 int factTail(int n, int acc) {
▶   if (n <= 1) return acc;   // base case
▶   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
6 System.out.println(factTail(4, 1));
```

VARIABLES

`n = 4` `acc = 1`

OUTPUT

CALL STACK

`factTail(4,1)`

`main()`

Code Trace: `factTail(4, 1)`



→ call · ← return

n = 3 > 1 → multiply **first**: 3 · 4 = 12. Then call `factTail(2, 12)`.

Step 3 / 9

```
1 int factTail(int n, int acc) {
▶   if (n <= 1) return acc;   // base case
▶   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
6 System.out.println(factTail(4, 1));
```

VARIABLES

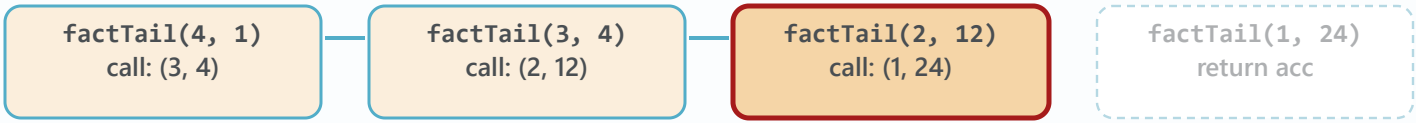
`n = 3` `acc = 4`

OUTPUT

CALL STACK

- `factTail(3,4)`
- `factTail(4,1)`
- `main()`

Code Trace: `factTail(4, 1)`



→ call · ← return

n = 2 > 1 → multiply **first**: 2 · 12 = 24. Then call `factTail(1, 24)`.

Step 4 / 9

```
1 int factTail(int n, int acc) {
▶   if (n <= 1) return acc;   // base case
▶   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
6 System.out.println(factTail(4, 1));
```

VARIABLES

`n = 2`

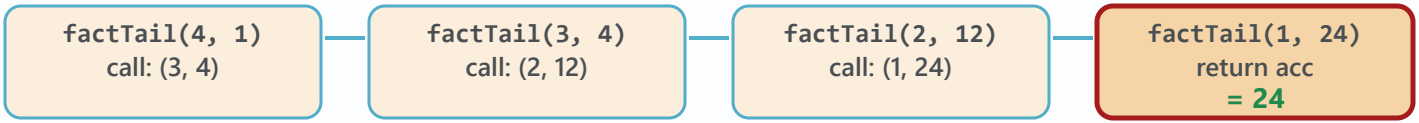
`acc = 12`

OUTPUT

CALL STACK

- `factTail(2,12)`
- `factTail(3,4)`
- `factTail(4,1)`
- `main()`

Code Trace: `factTail(4, 1)`



→ call · ← return

n = 1: **base case** → return acc = **24**. The result is ready right here.

Step 5 / 9

```
1 int factTail(int n, int acc) {
▶   if (n <= 1) return acc;   // base case
3   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
6 System.out.println(factTail(4, 1));
```

VARIABLES

`n = 1`

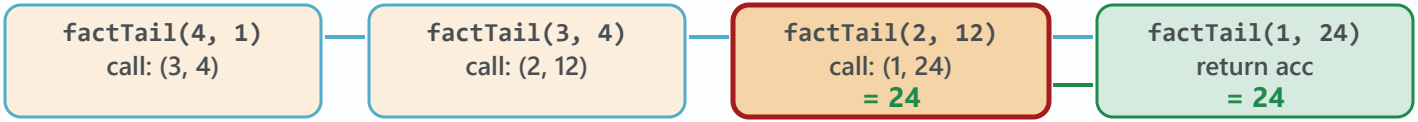
`acc = 24`

OUTPUT

CALL STACK

- `factTail(1,24)`
- `factTail(2,12)`
- `factTail(3,4)`
- `factTail(4,1)`
- `main()`

Code Trace: `factTail(4, 1)`



factTail(2, 12) passes 24 on **unchanged**: there is nothing to do on the way back.

Step 6 / 9

```
1 int factTail(int n, int acc) {
2   if (n <= 1) return acc;   // base case
▶   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
6 System.out.println(factTail(4, 1));
```

VARIABLES

`n = 2`

`acc = 12`

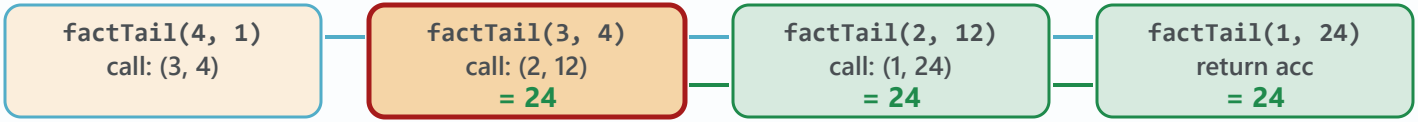
`returned = 24`

OUTPUT

CALL STACK

- `factTail(2,12)`
- `factTail(3,4)`
- `factTail(4,1)`
- `main()`

Code Trace: `factTail(4, 1)`



factTail(3, 4) passes 24 on **unchanged**: there is nothing to do on the way back.

Step 7 / 9

```
1 int factTail(int n, int acc) {
2   if (n <= 1) return acc;   // base case
▶   return factTail(n - 1, n * acc);  // last action
4 }
5 // in main:
6 System.out.println(factTail(4, 1));
```

VARIABLES

`n = 3` `acc = 4`

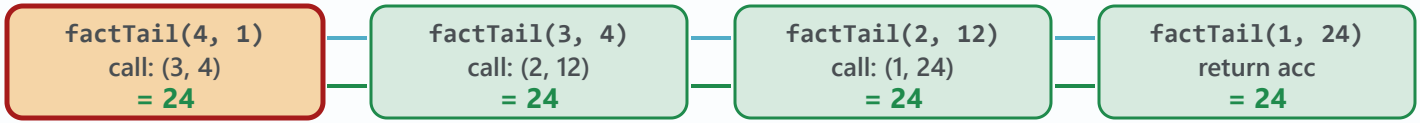
`returned = 24`

OUTPUT

CALL STACK

- `factTail(3,4)`
- `factTail(4,1)`
- `main()`

Code Trace: `factTail(4, 1)`



factTail(4, 1) passes 24 on **unchanged**: there is nothing to do on the way back.

Step 8 / 9

```
1 int factTail(int n, int acc) {
2   if (n <= 1) return acc;   // base case
▶   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
6 System.out.println(factTail(4, 1));
```

VARIABLES

`n = 4` `acc = 1`

`returned = 24`

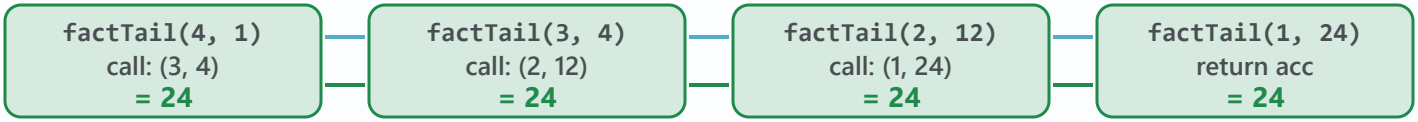
CALL STACK

`factTail(4,1)`

`main()`

OUTPUT

Code Trace: `factTail(4, 1)`



Result **24**. Java still pushed 4 frames, even though none of them was needed on the way back.

Step 9 / 9

```
1 int factTail(int n, int acc) {
2   if (n <= 1) return acc;   // base case
3   return factTail(n - 1, n * acc);  // last
  action
4 }
5 // in main:
▶ System.out.println(factTail(4, 1));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
24
```

Converting Tail Recursion to a Loop

```
int factLoop(int n) {
  int acc = 1;
  while (n > 1) {    // while not the base case
    acc = n * acc;   // the call's new
    n = n - 1;       //   parameters
  }
  return acc;        // base: return acc
}
```

- A tail call means updating the parameters and **jumping back to the start** of the method: the parameters become variables and the base case becomes the loop condition.
- The conversion is mechanical; iterative binary search is just `binarySearch` converted this way.
- The loop runs in constant memory, **O(1)**, with no frames pushed.

Java does not optimize tail calls: the compiler and the JVM push a new frame for every call to `factTail` too, so very deep recursion still throws `StackOverflowError`. Some languages, such as Scheme, guarantee this optimization; the Scala and Kotlin compilers can turn self-recursive tail calls into loops.

Recursion or Loop?

Criterion	Recursion	Loop
Readability	short, close to the definition	state variables managed by hand
Memory	O(depth) call stack	usually O(1)
Speed	every call has overhead	usually faster
Depth limit	risk of `StackOverflowError`	none

Recursion is the natural choice

Nested structures such as trees, backtracking (subsets, permutations), divide-in-half methods, Hanoi. The depth is small (log n, or a small n).

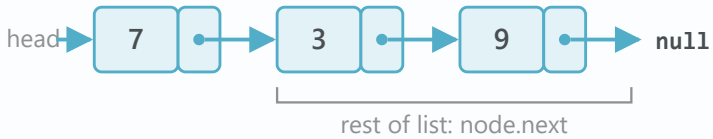
A loop is better

Simple counting and summing; linear recursion whose depth reaches n (long lists, large n); binary recursion with repeated subproblems (fib).

Any recursive method can be turned into a loop by doing the call stack's job with an **explicit stack**.

Recursion on Linked Lists

- A linked list is either **empty** (`null`) or a node followed by a **shorter list**.
- The definition is recursive: process the node, then pass the rest of the list (`node.next`) to the same method.
- Base case: `node == null` .




```
class Node { int data; Node next; }

int length(Node node) {
  if (node == null) return 0;      // empty list
  return 1 + length(node.next);    // 1 + the rest
}
```

```
void print(Node node) {
  if (node == null) return;
  System.out.print(node.data + " ");
  print(node.next);
}
```

Code Trace: `length(head)`



main calls `length(head)`; node points to the first node (7).

Step 1 / 9

```
1 int length(Node node) {
2   if (node == null) return 0;      // empty list
3   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
▶ System.out.println(length(head));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `length(head)`



node ≠ null → 1 + `length(node.next)` : a call for the rest of the list (starting at 3); this frame waits.

Step 2 / 9

```
1 int length(Node node) {
▶   if (node == null) return 0;      // empty list
▶   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
6 System.out.println(length(head));
```

VARIABLES

`node = Node(7)`

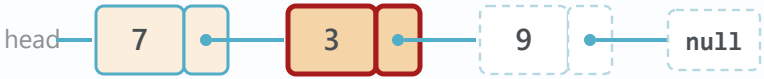
OUTPUT

CALL STACK

`length(7)`

`main()`

Code Trace: `length(head)`



node ≠ null → 1 + `length(node.next)` : a call for the rest of the list (starting at 9); this frame waits.

Step 3 / 9

```
1 int length(Node node) {
▶   if (node == null) return 0;      // empty list
▶   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
6 System.out.println(length(head));
```

VARIABLES

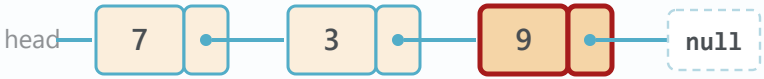
node = Node(3)

OUTPUT

CALL STACK

- length(3)
- length(7)
- main()

Code Trace: `length(head)`



node ≠ null → 1 + `length(node.next)` : node.next = null, so the call is for the empty list.

Step 4 / 9

```
1 int length(Node node) {
▶   if (node == null) return 0;      // empty list
▶   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
6 System.out.println(length(head));
```

VARIABLES

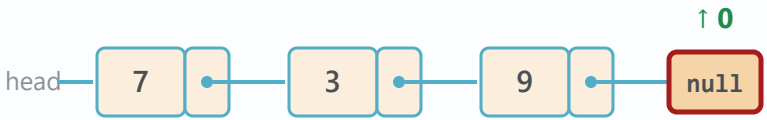
node = Node(9)

OUTPUT

CALL STACK

- length(9)
- length(3)
- length(7)
- main()

Code Trace: `length(head)`



node = null: **base case**; the empty list has length 0.

Step 5 / 9

```
1 int length(Node node) {
▶   if (node == null) return 0;      // empty list
3   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
6 System.out.println(length(head));
```

VARIABLES

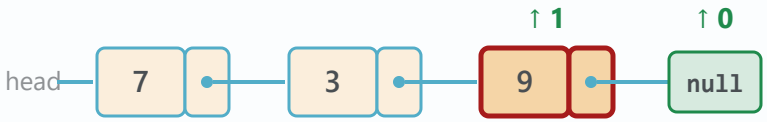
`node = null`

OUTPUT

CALL STACK

- `length(null)`
- `length(9)`
- `length(3)`
- `length(7)`
- `main()`

Code Trace: `length(head)`



The rest of the list gave 0 → return 1 + 0 = **1**.

Step 6 / 9

```
1 int length(Node node) {
2   if (node == null) return 0;      // empty list
▶   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
6 System.out.println(length(head));
```

VARIABLES

`node = Node(9)`

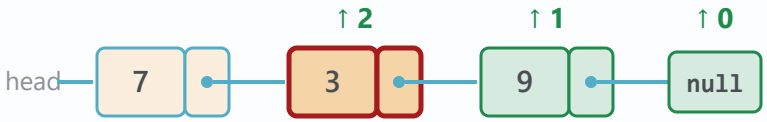
`returned = 0`

OUTPUT

CALL STACK

- `length(9)`
- `length(3)`
- `length(7)`
- `main()`

Code Trace: `length(head)`



The rest of the list gave 1 → return 1 + 1 = **2**.

Step 7 / 9

```
1 int length(Node node) {
2   if (node == null) return 0;      // empty list
▶   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
6 System.out.println(length(head));
```

VARIABLES

`node = Node(3)`

`returned = 1`

OUTPUT

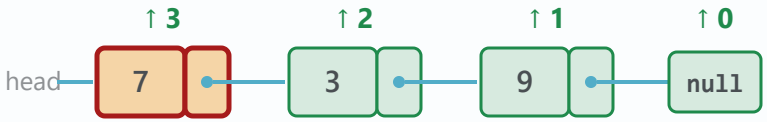
CALL STACK

`length(3)`

`length(7)`

`main()`

Code Trace: `length(head)`



The rest of the list gave 2 → return 1 + 2 = **3**.

Step 8 / 9

```
1 int length(Node node) {
2   if (node == null) return 0;      // empty list
▶   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
6 System.out.println(length(head));
```

VARIABLES

`node = Node(7)`

`returned = 2`

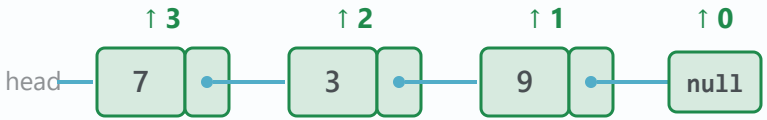
OUTPUT

CALL STACK

`length(7)`

`main()`

Code Trace: `length(head)`



Result **3**. 4 calls (3 nodes + null): time O(n), stack O(n).

Step 9 / 9

```
1 int length(Node node) {
2   if (node == null) return 0;      // empty list
3   return 1 + length(node.next);    // 1 + the
  rest
4 }
5 // in main (head → 7 → 3 → 9 → null):
▶ System.out.println(length(head));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
3
```

Printing a List in Reverse




```
void print(Node node) {
  if (node == null) return;
  System.out.print(node.data + " ");
  print(node.next);
}
```

Output: `7 3 9` (print before the call)

```
void printReverse(Node node) {
  if (node == null) return;
  printReverse(node.next);  // the rest first
  System.out.print(node.data + " ");
}
```

Output: `9 3 7` (print after the call)

- A singly linked list has no backward links; the **call stack** provides the way back, printing the nodes from last to first as it unwinds.
- The price is O(n) stack memory. For very long lists, a loop with an explicit stack is safer.

SUMMARY

Common Mistakes

Missing or unreachable base case

If 0 or a negative input skips the base case, the calls never end: `StackOverflowError`. `n <= 1` is safer than `n == 1`.

Problem doesn't shrink

If `f(n)` calls `f(n)` or `f(n + 1)` again, it never gets closer to the base case.

Losing the return value

Forget the `return` before the left-half call in `binarySearch` and the result is thrown away; control falls through to the right-half call. Wrong answer, and no compiler warning.

Forgetting to undo

Without `s.remove(...)` in backtracking, one branch's choices leak into the next, and the subsets come out wrong.

Repeated work

In binary recursion like `fib`, the same subproblem is solved an exponential number of times. Memoization or a loop gives O(n).

Deep linear recursion

On a list of a million nodes, `length` overflows the stack. Java doesn't optimize tail calls → use a loop.

LAB · JAVA CODE, STEP BY STEP

Lab: The Call Stack

factorial(4)

Input (n, 0–10) 4

CODE · **step 1 / 13**

```
1 int factorial(int n) {
2   if (n <= 1)
3     return 1;               // base case
4   return n * factorial(n - 1);
5 }
6
7 // in main:
8 System.out.println(factorial(4));
```

main waiting: factorial(4)

depth = 0 calls = 0

OUTPUT —

main calls factorial(4); a new frame will be pushed onto the call stack.

SUMMARY

Summary

Example	Kind of recursion	Time	Space (stack)
factorial, sum, reverse	linear	O(n)	O(n)
fastPower, binarySearch	linear, halving	O(log n)	O(log n)
fib	binary	exponential ($\approx 1.6^n$)	O(n)
hanoi	binary	$O(2^n)$	O(n)
subsets	backtracking	$O(n \cdot 2^n)$	O(n)
perm	multiple, backtracking	$O(n \cdot n!)$	O(n)
factTail	tail recursion	O(n)	O(n) (in Java)
length, printReverse	linear (linked list)	O(n)	O(n)

Recursion = a base case + a recursive step that moves toward it. Time comes from the number of calls, space from the deepest chain of calls.

SUMMARY

Interactive Simulators

Towers of Hanoi Simulator

Solve the puzzle by hand with 1–6 disks, step forward and back through the shortest solution, or let it solve itself. Your move count is compared with the minimum $(2^n - 1)$: 63 for n = 6.

Each peg behaves like a stack: only the top disk can be taken, and it can only go onto a larger disk.

SUMMARY

Sample Problems

1. For `int f(int n) { if (n <= 0) return 0; return n % 10 + f(n / 10); }`, what is `f(2026)`, and how many calls are made?
2. What does `countUp(4)` print? At most how many frames (including main) are on the stack?
3. How many calls does `fastPower(3, 13)` make, and what is the result?
4. How many calls does `fib(6)` make?
5. For {1, 2, 3, 4}, how many subsets does `subsets` print, and how many calls does it make?
6. If T(n) = T(n/2) + c and T(1) = c, how many calls are in the chain for n = 64?

▶ **Show answers**

SUMMARY

Exercises

1. While `factorial(5)` runs, what is the largest number of frames (including main) on the call stack?
2. How many moves does Hanoi take with 4 disks?
3. What are the base case and the recursive step of a recursive `largest(a, n)` that returns the largest item of an array?
4. `isPalindrome(s, i, j)` compares the ends and moves inward. How many calls does it make for "kayak"?
5. `int g(int n) { if (n <= 1) return 1; return g(n - 1) + g(n - 1); }` : what are g(5) and the number of calls? How can it be made O(n)?
6. In the loop version of `factTail`, how do n and acc change on each iteration?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Sedgewick and Wayne, *Computer Science: An Interdisciplinary Approach*, 2016.
- Roberts, *Thinking Recursively with Java*, 2006.
- Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 4th ed., 2022.

The End

Chapter 4: Recursion

Next deck: Chapter 5: Linked Lists · Study questions · Simulator: Towers of Hanoi Simulator · Glossary