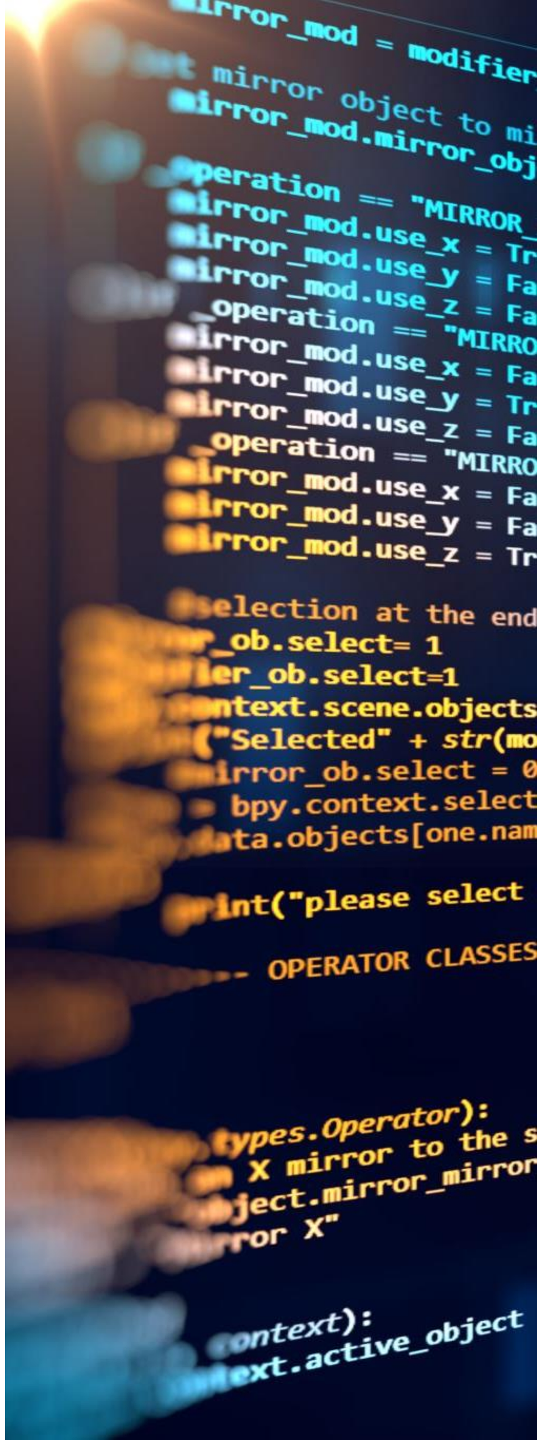
Please use the following QR code to check in and record your attendance.

CS 1027

Fundamentals of Computer Science II

Recursion in Java (cont.)

Ahmed Ibrahim



Recursive Definitions

Recursive Definition

- Defining something in terms of a smaller or simpler version of itself.
- A recursive definition consists of two parts:
  - The *base case*: this defines the *simplest* case or starting point
  - The *recursive part* is the general case that describes all the other cases in terms of smaller versions of itself.
  - Example:

```java
// Recursive method to calculate factorial
public static int factorial(int n) {
  // Base case: if n is 1, return 1 (factorial of 1 is 1)
  if (n == 1) {return 1;}
  // Recursive part: n * factorial of (n - 1)
  else {return n * factorial(n - 1);}
}
```

base

recursive

Why is a base case needed in any recursive algorithm?

Recursion vs. Iteration

- What is iteration? Repetition, as in a loop
- What is recursion? Defining something in terms of a smaller or simpler version of itself (why smaller/simpler?)
- Recursion is a very powerful problem-solving technique.
- Many complex problems would be very difficult to solve without the use of recursion.

Example of Recursive Problem

- Consider the problem of computing the sum of all the numbers between 1 and n:

$$1 + 2 + 3 + 4 + \ldots + n\text{-}1 + n$$

- Here is a simple iterative algorithm for this problem:

```
Algorithm sum (n)
total = 0
for i = 1 to n do
  total = total + i
return total
```

Example of Recursive Problem

- Consider the problem of computing the sum of all the numbers between 1 and n:

1 + 2 + 3 + 4 + … + n-1 + n

- Recursive definition:

`sum of 1 to 1 => 1` (*base case*)

`sum of 1 to n => n + ` **`the sum of 1 to n-1`**`, for n > 1`

$$\sum_{k=1}^{n} k = n + \sum_{k=1}^{n-1} k$$ (recursive case)

Recursive Algorithm

- Recursive definition:

  `sum of 1 to 1 => 1` (*base case*)

  `sum of 1 to n => n +` **`the sum of 1 to n-1`**`, for n > 1`

- Recursive algorithm for this problem:

```
Algorithm sum(n)
In: Positive value n
Out: 1 + … + n
if n = 1 then return 1 // base case
else return n + sum(n-1) // recursive case
```

How Recursion Works

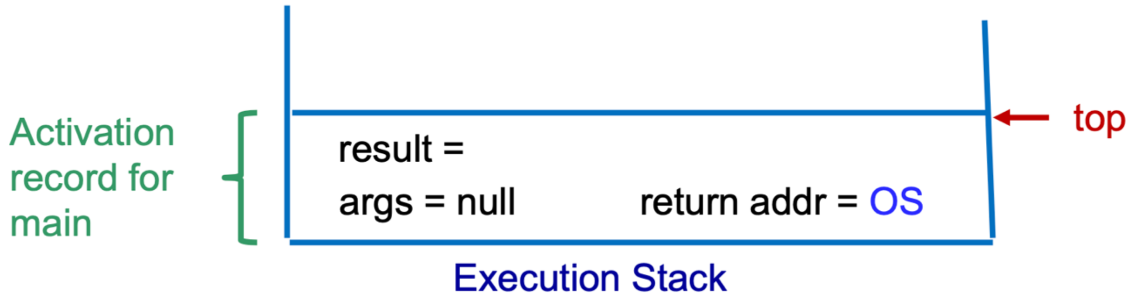
- Consider the following program

```
public static void main (String[] args) {int result = sum(4);} // addr 1
public static int sum (int n) {if (n == 1) return 1;
                               else return n + sum(n-1);} // addr2
```

- An **activation record** is created for the method main when the program is executed. This activation record stores:
  - The return address **addr1**
  - The variable **result**
  - The parameter **args**

How Recursion Works

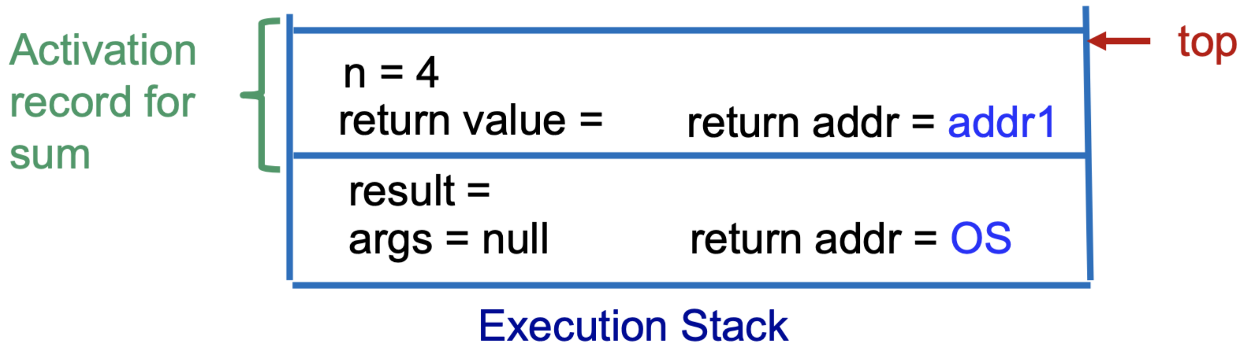
```
public static void main (String[] args)
{int result = sum(4); // addr 1
}
public static int sum (int n) {
if (n == 1) return 1;
else return n + sum(n-1); // addr2
}
```



- At this point, the execution stack looks like the following figure. We assume no parameter is passed to the main function, so args is null.
- The result variable has no value assigned to it yet, so we left its value blank. OS denotes the address of the virtual machine's instruction where the main method was invoked.

How Recursion Works

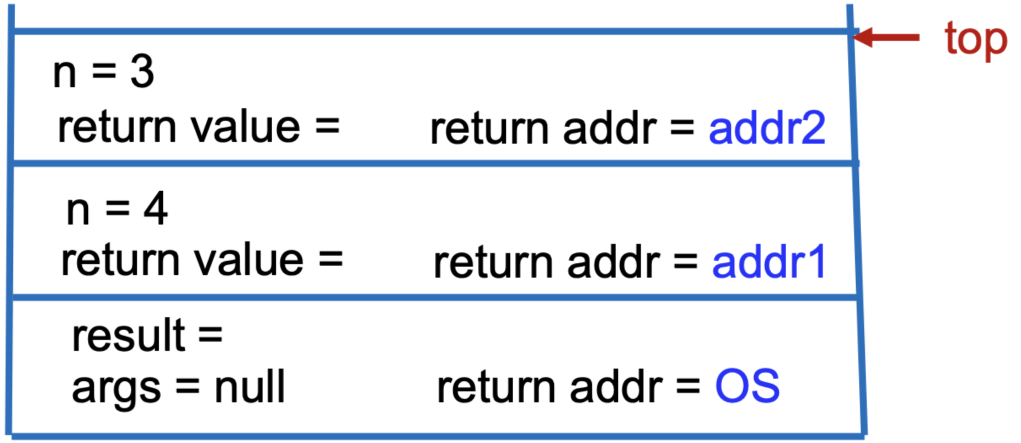
```
public static void main (String[] args)
{int result = sum(4); // addr 1
}
public static int sum (int n) {
if (n == 1) return 1;
else return n + sum(n-1); // addr2
}
```



- Once the activation record for the main function has been created and the values of the parameters and return address have been stored, the execution of the method main starts.
- The first and only statement of the main function invokes method **sum**(4).
- This creates another activation record pushed into the execution stack, as shown above.

How Recursion Works (cont.)

```
public static void main (String[] args)
{int result = sum(4); // addr 1
}
public static int sum (int n) {
if (n == 1) return 1;
else return n + sum(n-1); // addr2
}
```

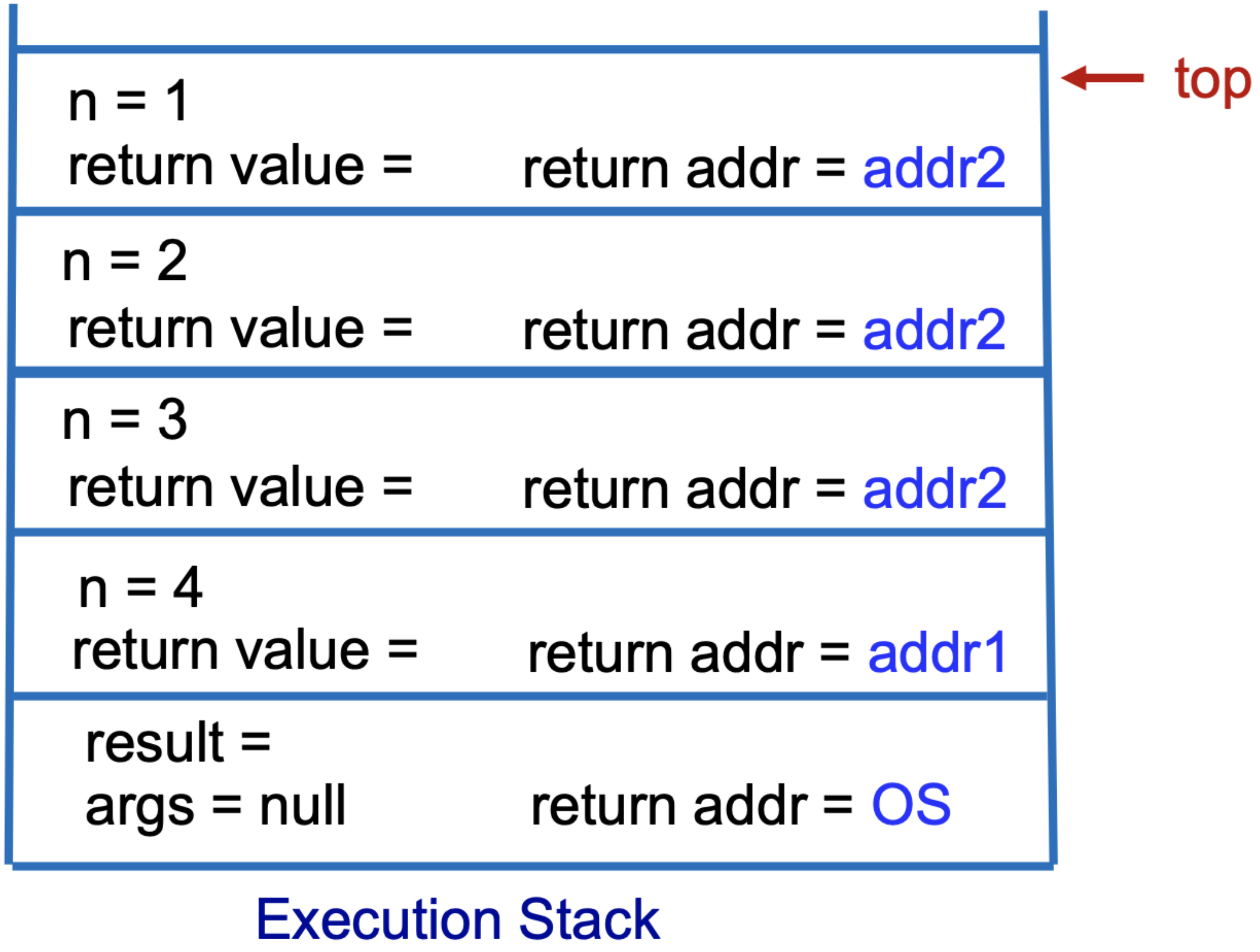


Execution Stack

- Once the activation record has been created, the execution of the method sum starts.
- Since n > 1, method sum (n-1) is invoked.
- A new activation record is created and pushed into the stack

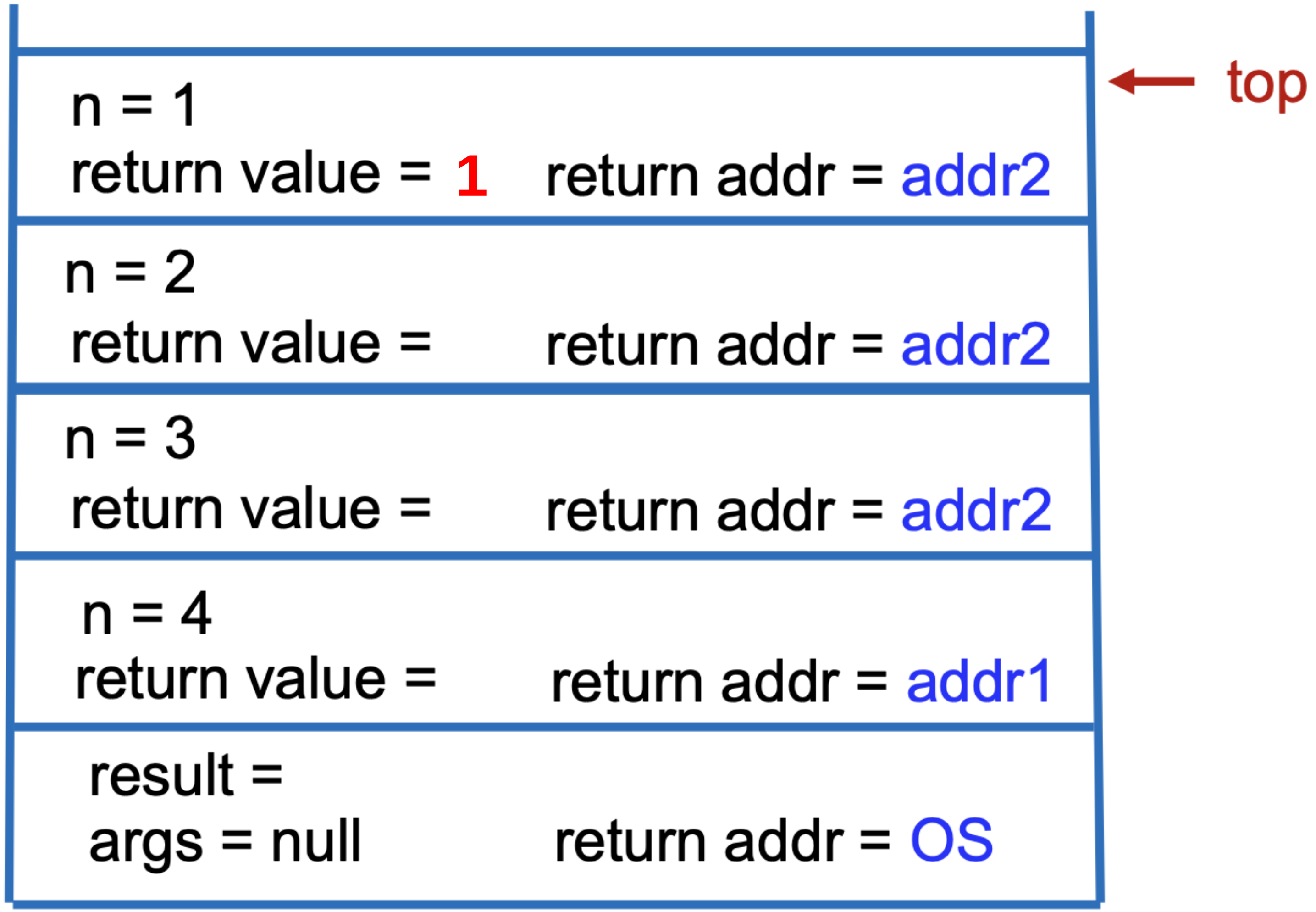
How Recursion Works (cont.)

- Then, two more invocations to the method sum with parameters 2 and 1 are made.
- After the last invocation, the execution stack looks like the figure given.



How Recursion Works (cont.)

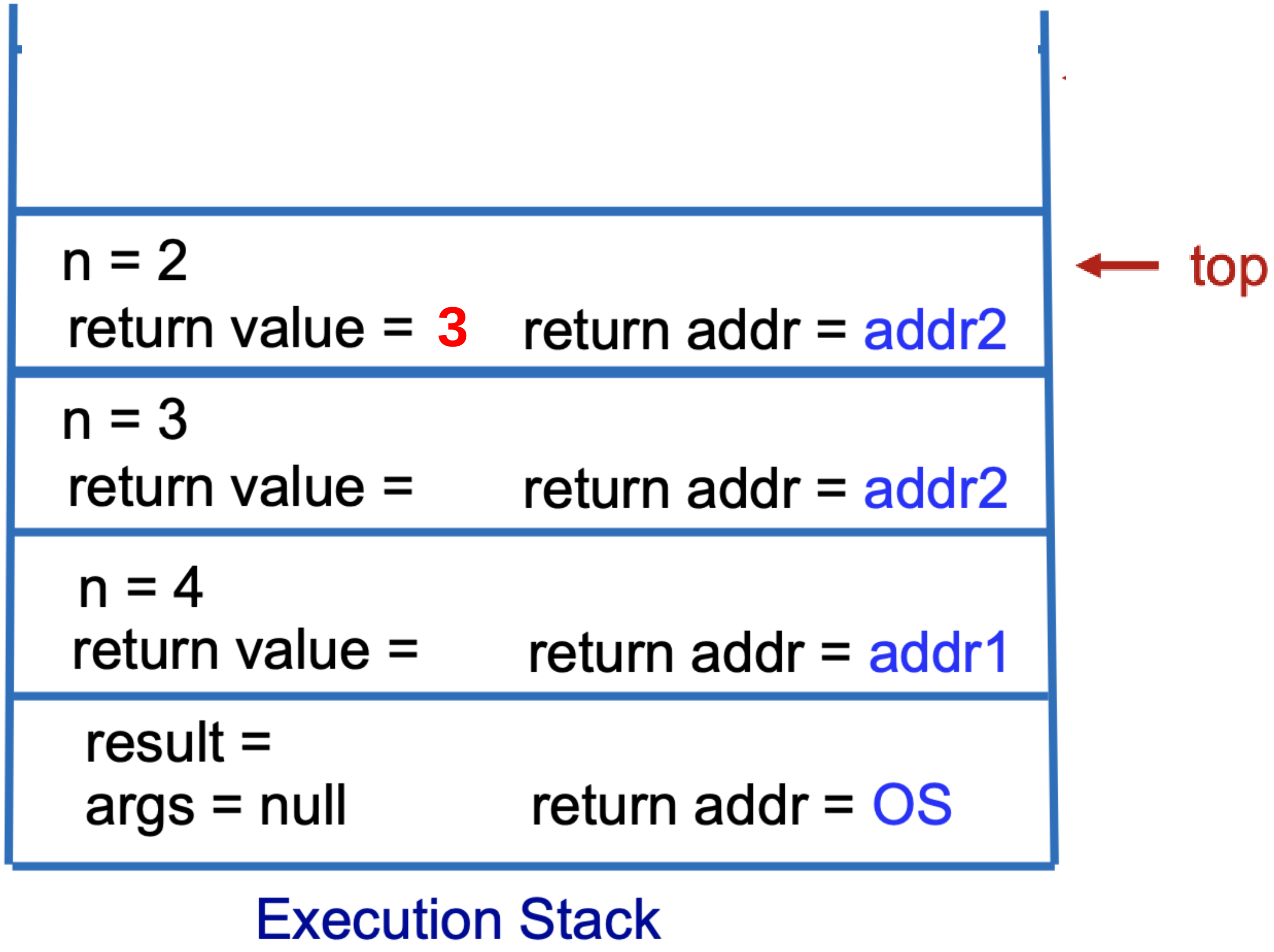
- Since the value of n is 1 in the last invocation of the method sum, the statement return 1 (base case) is executed.
- The value 1 is stored in the **return value**.



Execution Stack

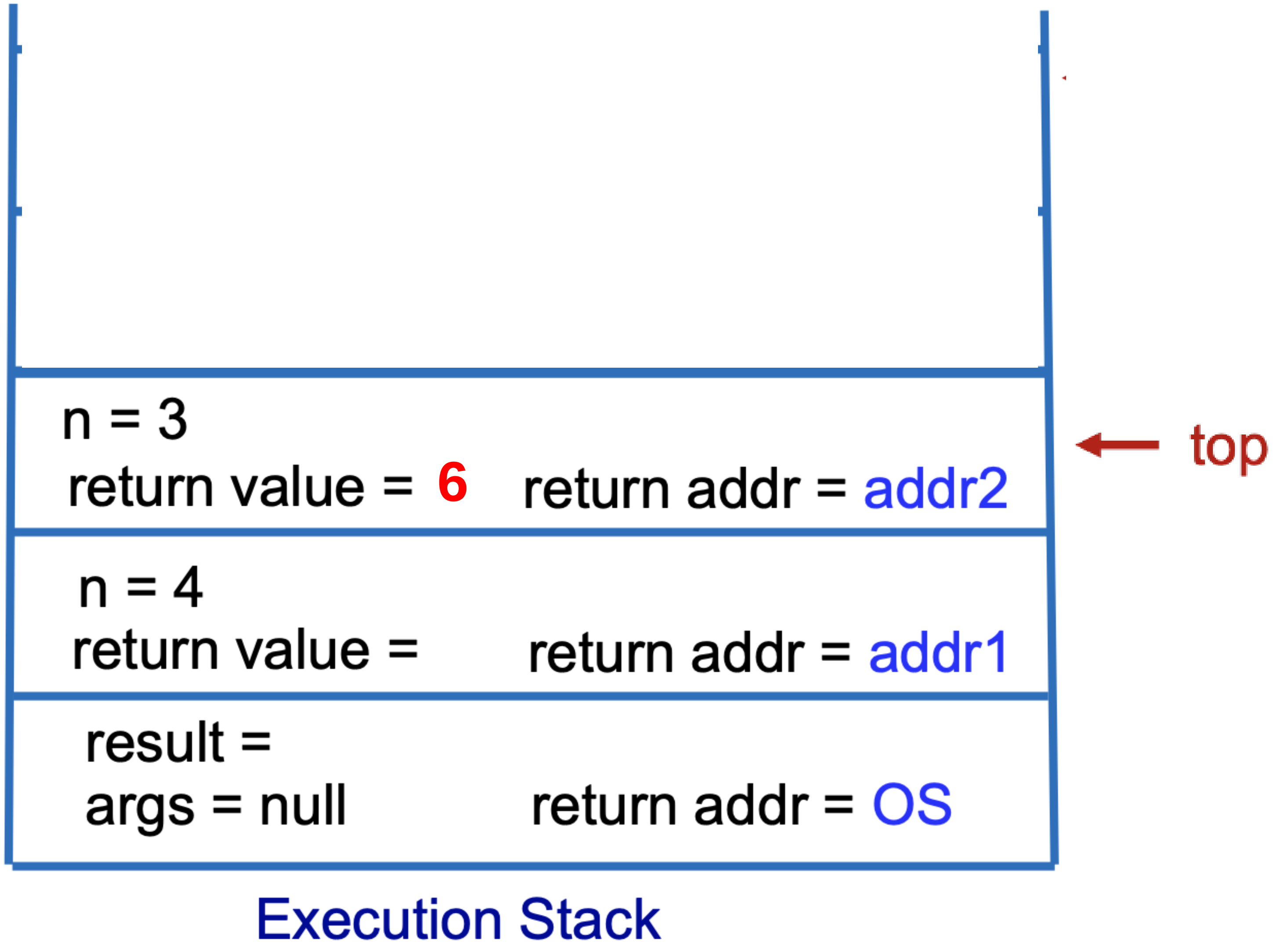
How Recursion Works (cont.)

- The method sum ends, and hence, an activation record is popped off the execution stack.
- The return address addr2 is recovered, and execution continues at the statement in that address: This call just finished, and it returned the value 1.
- Hence, n + sum(n-1) = 2 + 1 = 3 will be returned.



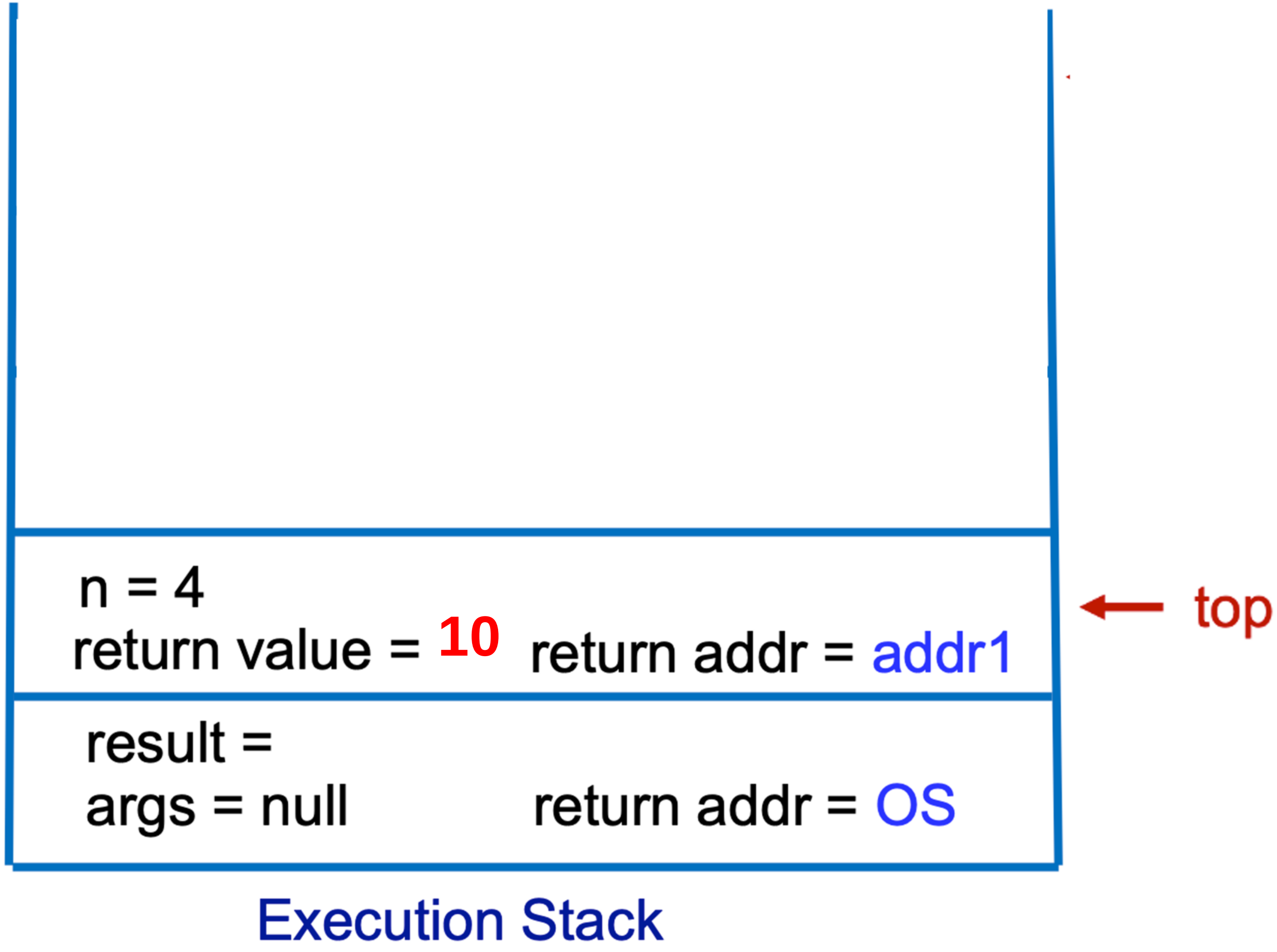
How Recursion Works (cont.)

- The next call returns the value 3, and an activation record is popped off the execution stack. The return address addr2 is recovered, and execution continues with the statement in that address.
- The value n + sum(n-1) = 3 + 3 = 6 will be returned.



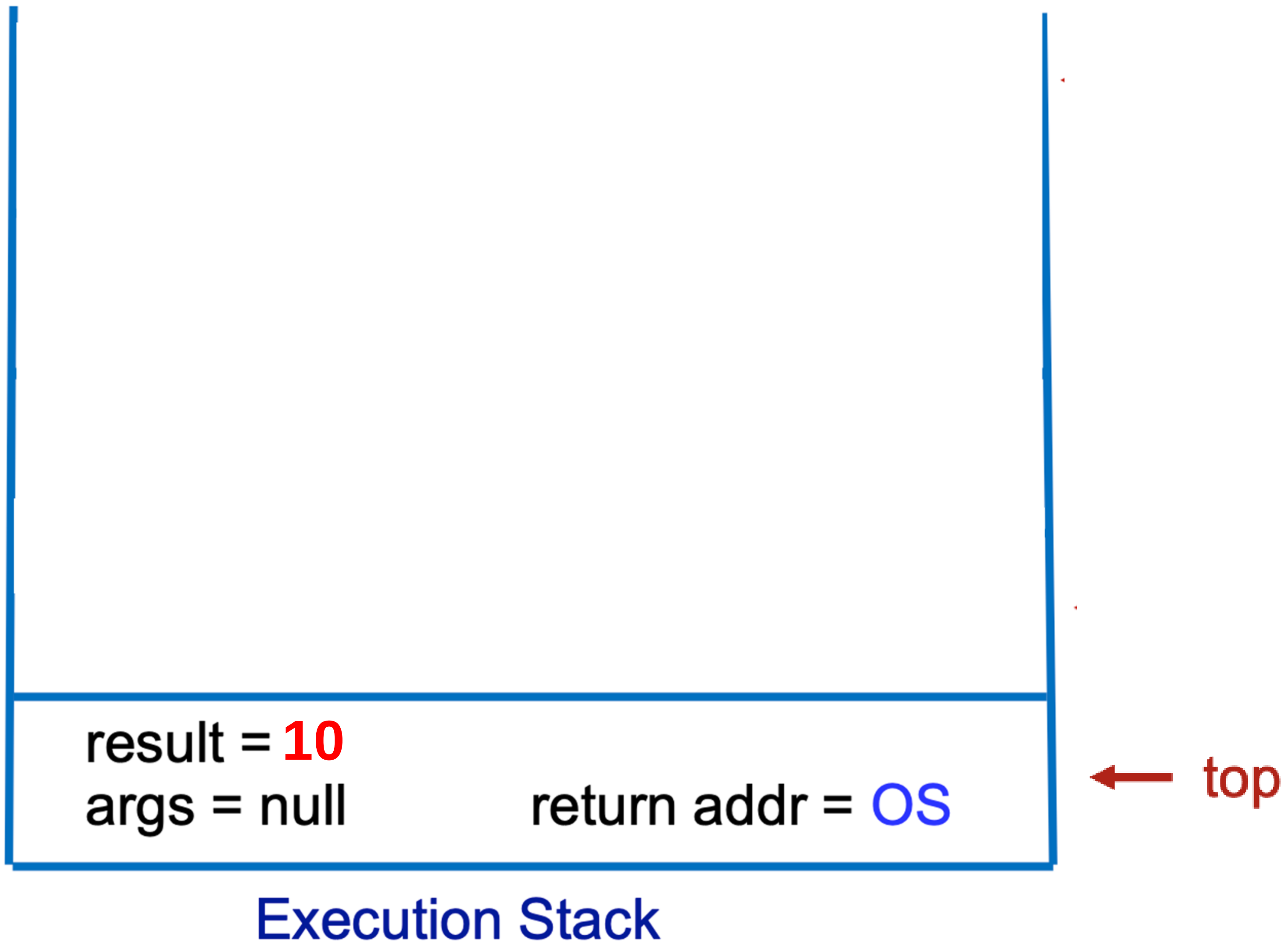
How Recursion Works (cont.)

```
public static void main
(String[] args) {
int result = sum(4); // addr 1
}
```



How Recursion Works (cont.)

```
public static void main
(String[] args) {
int result = sum(4); // addr 1
}
```



Recursion vs. Iteration

- Every recursive algorithm can also be written as an **iterative** algorithm. However, the algorithm could be much more complex and require the use of an auxiliary stack or other data structures to simulate the execution stack.
- Thus, just because we can use recursion to solve a problem does not mean we should!
- Would you use iteration or recursion to compute the sum of 1 to n? Why?

Recursion vs. Iteration

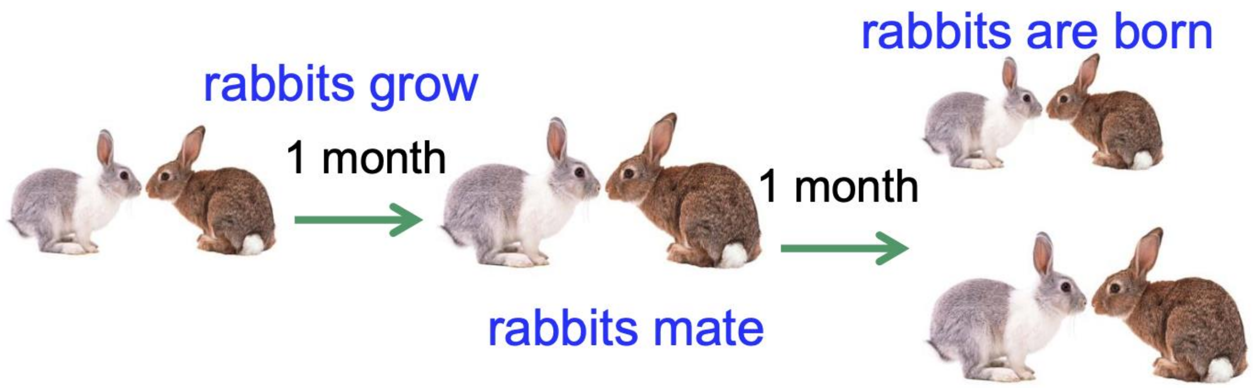
- Recursion often uses **more memory** and can lead to stack overflow errors if the recursive depth is too large. Sometimes, an iterative (loop-based) solution is **faster** and more m**emory-efficient**.

Problems Solved with Recursion

Fibonacci Sequence

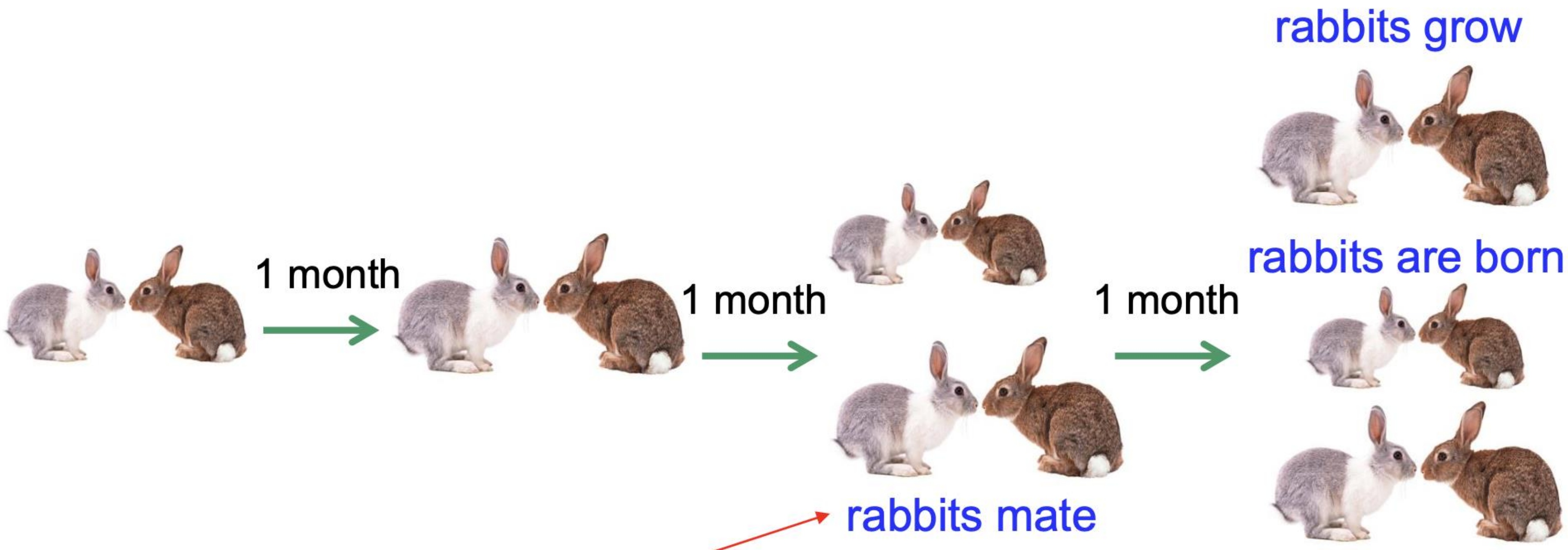
Multiplying Rabbits Problem

- Consider the following problem:
  - You have one pair of baby rabbits (1 male and 1 female).
  - After 1 month the rabbits can mate, and 2 baby rabbits are born one month after mating.
- How many rabbits will there be after n months?



Number of pairs of rabbits	1	1	2

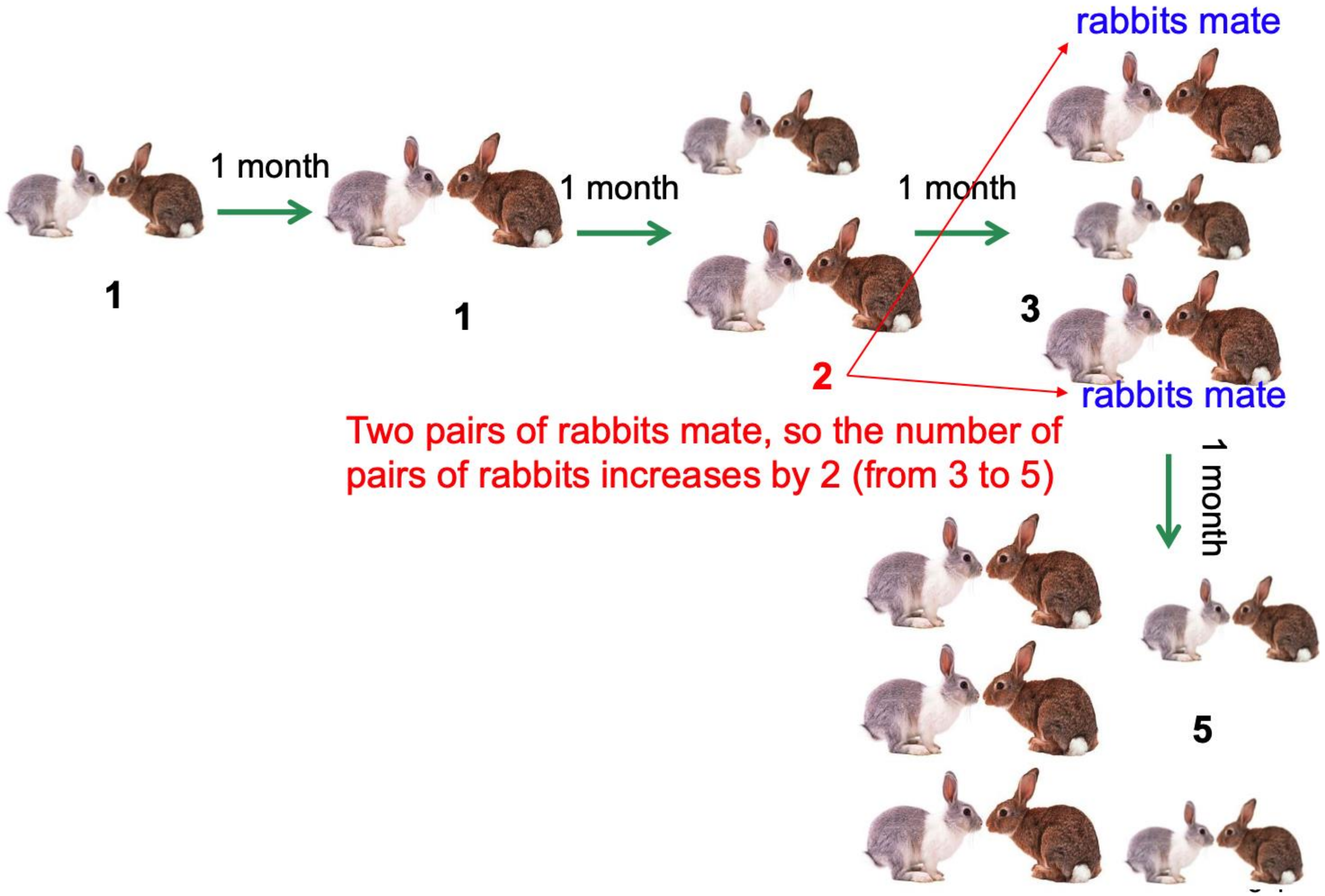
Multiplying Rabbits Problem



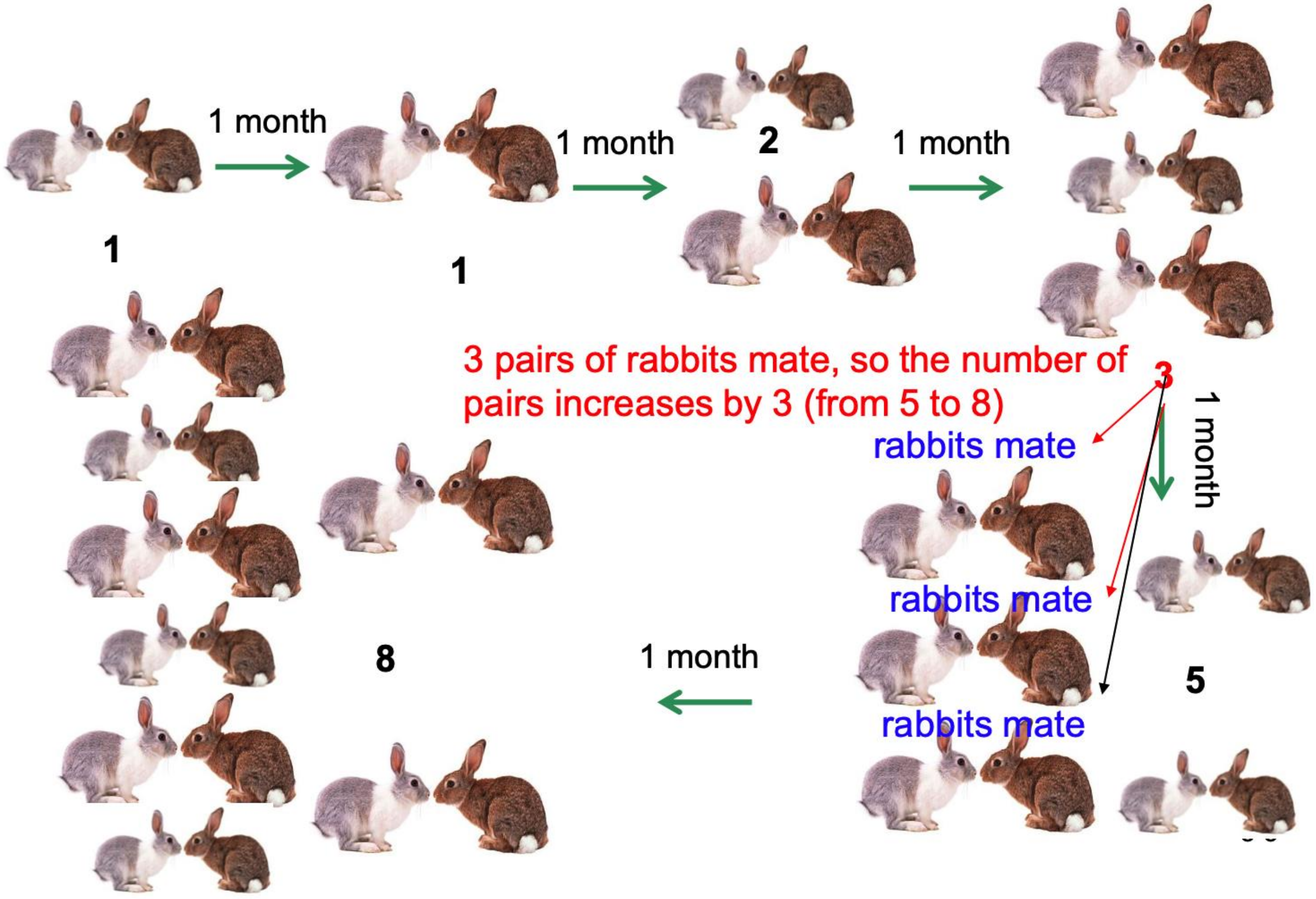
Number of pairs of rabbits: 1 1 2 3

(One pair of rabbits' mate, so, the number of pairs of rabbits increase by 1 (from 2 to 3))

Multiplying Rabbits Problem



Multiplying Rabbits Problem



Multiplying Rabbits Problem

- The number of pairs of rabbits increases every month like this:

  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- This sequence is called the *Fibonacci Sequence*
- The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones. It is often represented as:

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

$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

Fibonacci was an Italian mathematician born around **1170 CE** in Pisa, Italy.

Recursive Nature of *Fibonacci Sequence*

- The sequence is defined recursively, making it simple to implement in programming:

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

$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2$$

```
// precondition (assumption) : n > = 1
public static int rfib (int n) {
  if ((n == 1) || (n == 2))
    return 1;
    else return rfib(n – 1) + rfib(n – 2);
}
```

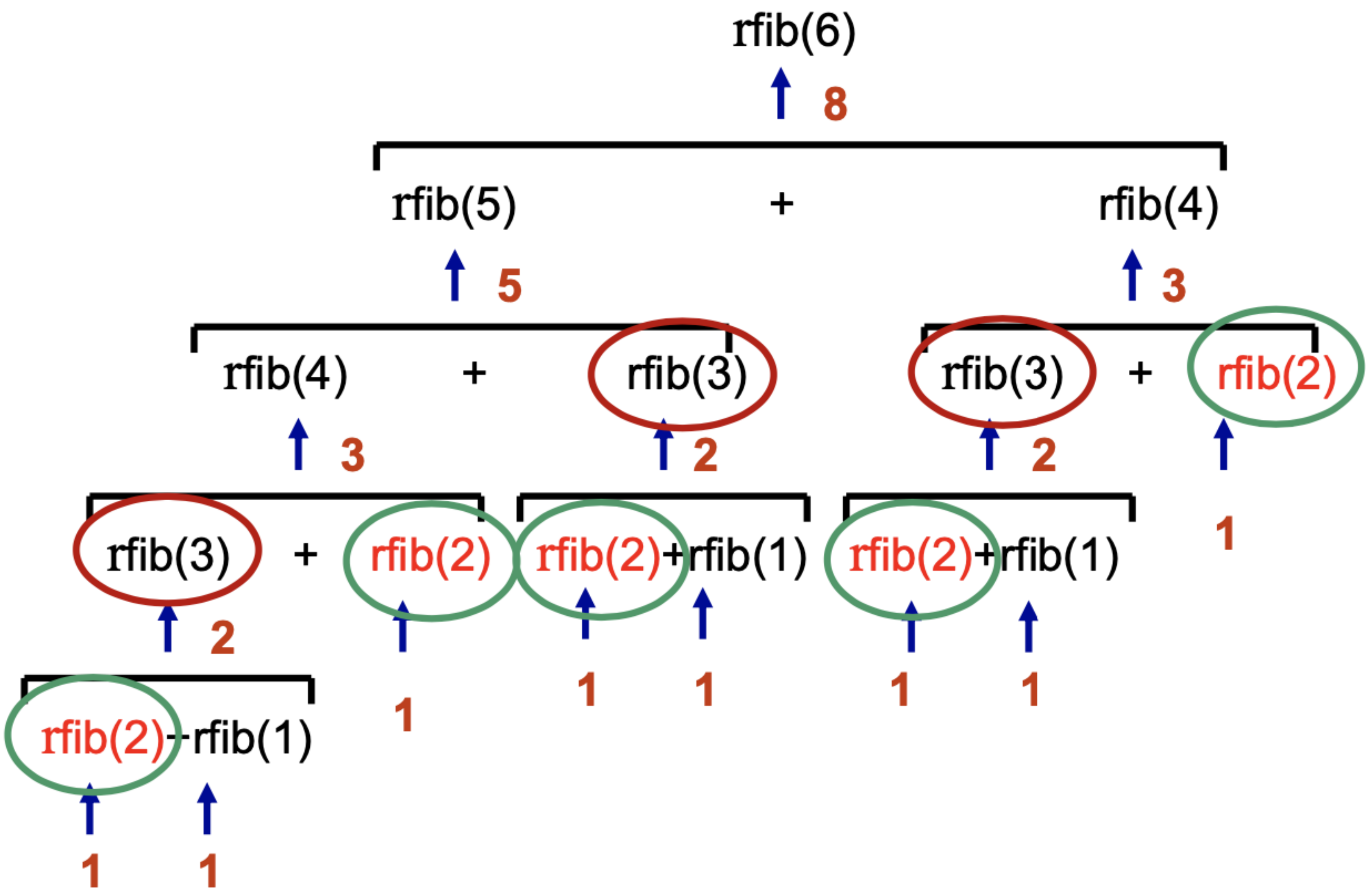
When to Use *Recursion*

- We must ensure that a simple iterative algorithm is impossible when designing a recursive algorithm. A recursive algorithm is **generally slower** than an iterative one due to the need to manage activation records.
- Designing a recursive algorithm requires ensuring that the exact same recursive call is not repeated, as otherwise, the resulting algorithm could be very slow.
- For example:

$$F(5) = F(4) + F(3)$$

$$F(4) = F(3) + F(2)$$

- This is the case with the previous algorithm for computing the *Fibonacci* numbers. The next page shows duplicated calls that make this algorithm very slow.


rfib(6)
8
rfib(5)
+
rfib(4)
5
3
rfib(4)
+
rfib(3)
rfib(3)
+
rfib(2)
3
2
2
rfib(3)
+
rfib(2)
rfib(2)+rfib(1)
rfib(2)+rfib(1)
1
2
rfib(2)+rfib(1)
1
1
1
1
1
1
1

Iterative vs. Recursive Algorithm for Fibonacci Number

```java
public static int ifib(int n) {
if (n == 1 || n == 2) return 1;

int fib1 = 1, fib2 = 1;
int fib3 = 0; // Initialize fib3
for (int i = 3; i <= n; i++) {
        fib3 = fib1 + fib2;
        fib1 = fib2;
        fib2 = fib3;
  }
     return fib3;
}
```

Iterative Algorithm for Fibonacci Number

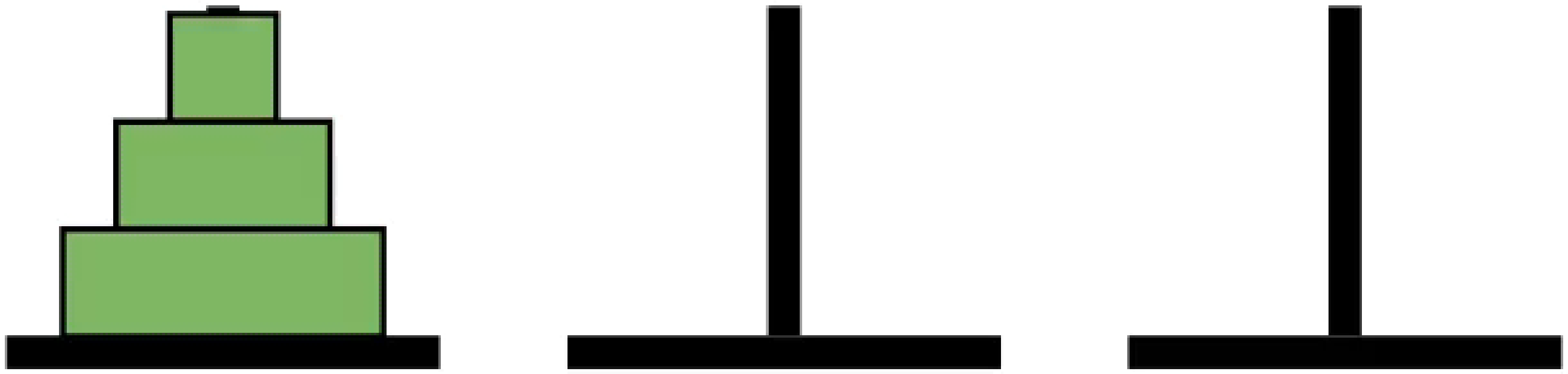
```java
// precondition (assumption) : n > = 1
public static int rfib (int n) {
  if ((n == 1) || (n == 2))
    return 1;
    else return rfib(n – 1) + rfib(n – 2);
}
```

Recursive Algorithm for Fibonacci Number

- This iterative version is more efficient than the recursive version as it does not perform repeated computations.

The Towers of Hanoi

The Towers of Hanoi



- We need to move n disks from peg A to peg C while satisfying the movements' restrictions.

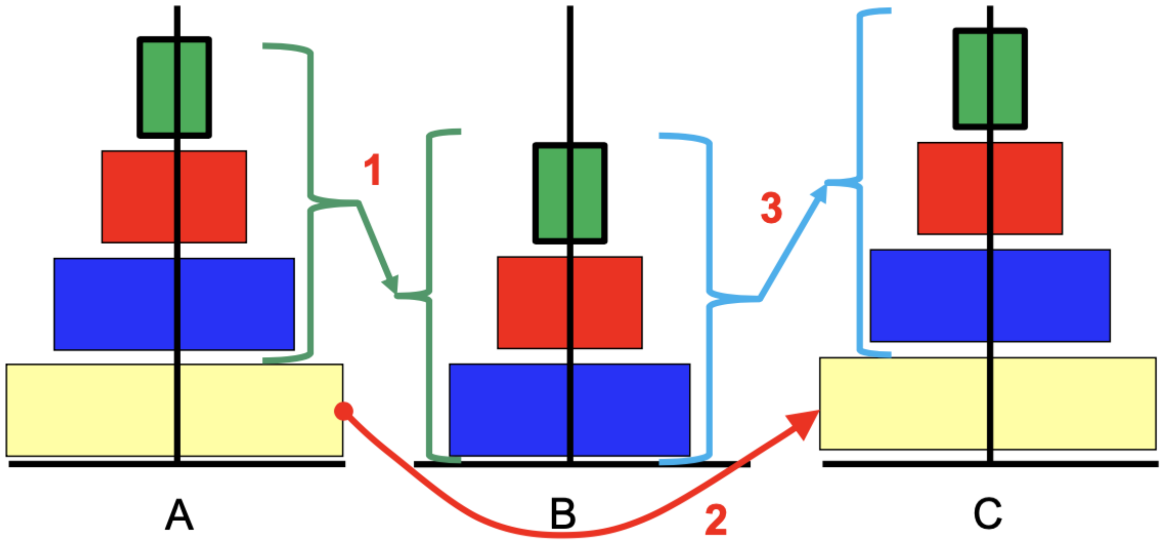
The Towers of Hanoi

- Given 3 pegs and a set of n => 4 disks of different diameters initially, in peg A, the goal is to move all of the disks from peg A to peg C following these rules:
  - Only one disk can be moved at a time; that disk must be at the top of a pile
  - A disk cannot be placed on top of a smaller disk
  - All disks must be on some peg

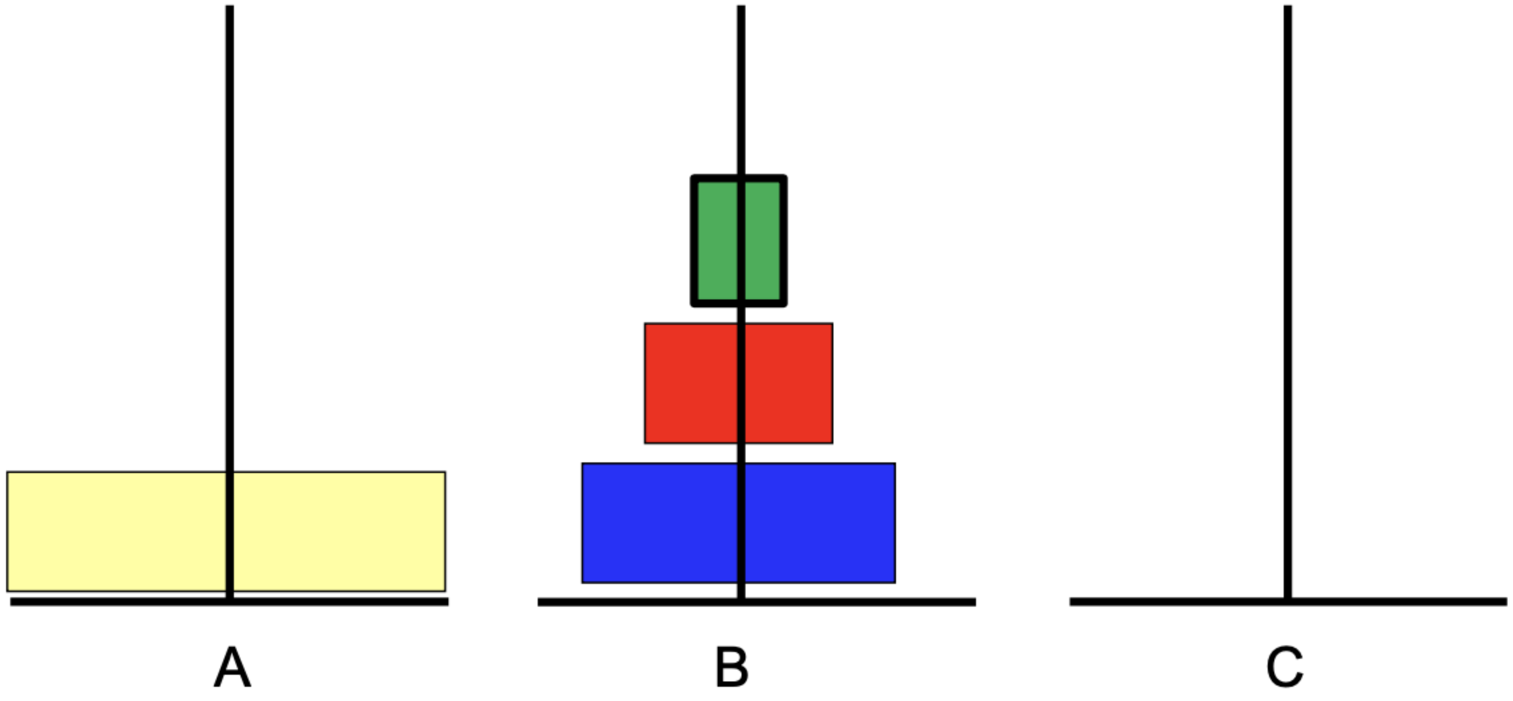
Algorithm

Recursive case (n > 1):

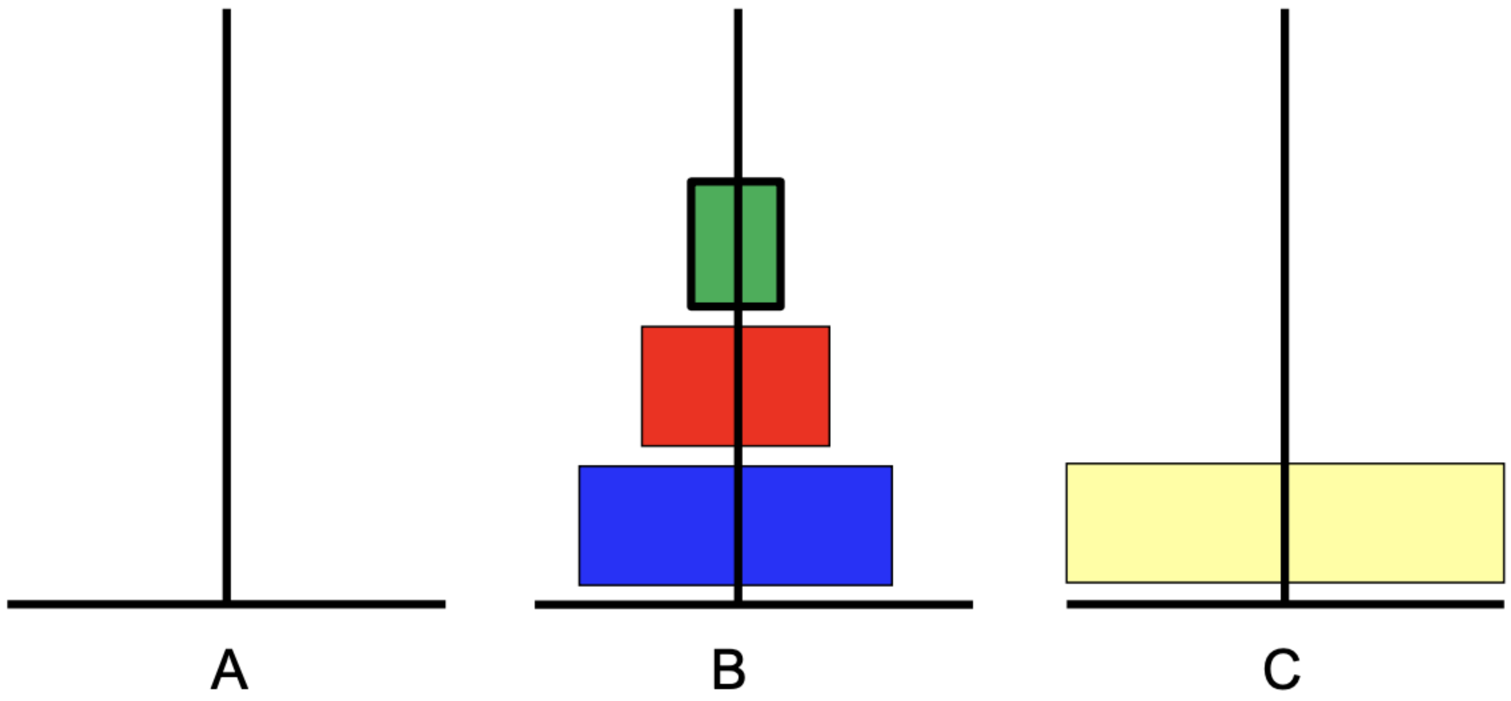
- Move n-1 disks from the initial peg to the intermediate peg (*Solving the smaller subproblem*)
- Move 1 disk from the initial peg to the final peg (*handling the largest disk*)
- Move n-1 disks from the intermediate peg to the final peg (*Solving the smaller subproblem*)



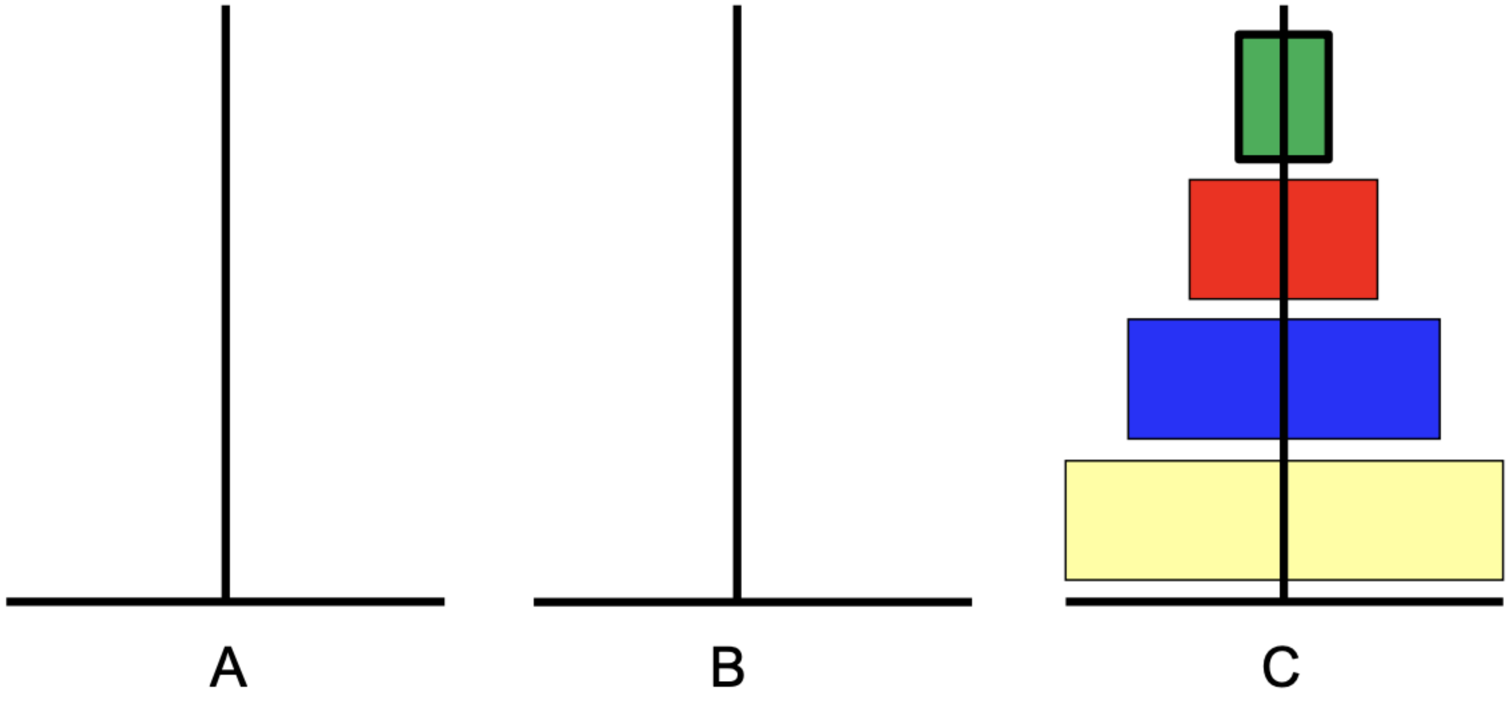
Step 1: Move n-1 disks from A to B



Step 2: Move 1 disk from A to C



Step 3: Move n-1 disks from B to C



Problem solved! ...But how do we move n-1 disks?

Why Does the Algorithm Say "Move n–1"

This is a *shorthand* for the recursive process. At each step, the algorithm focuses on:

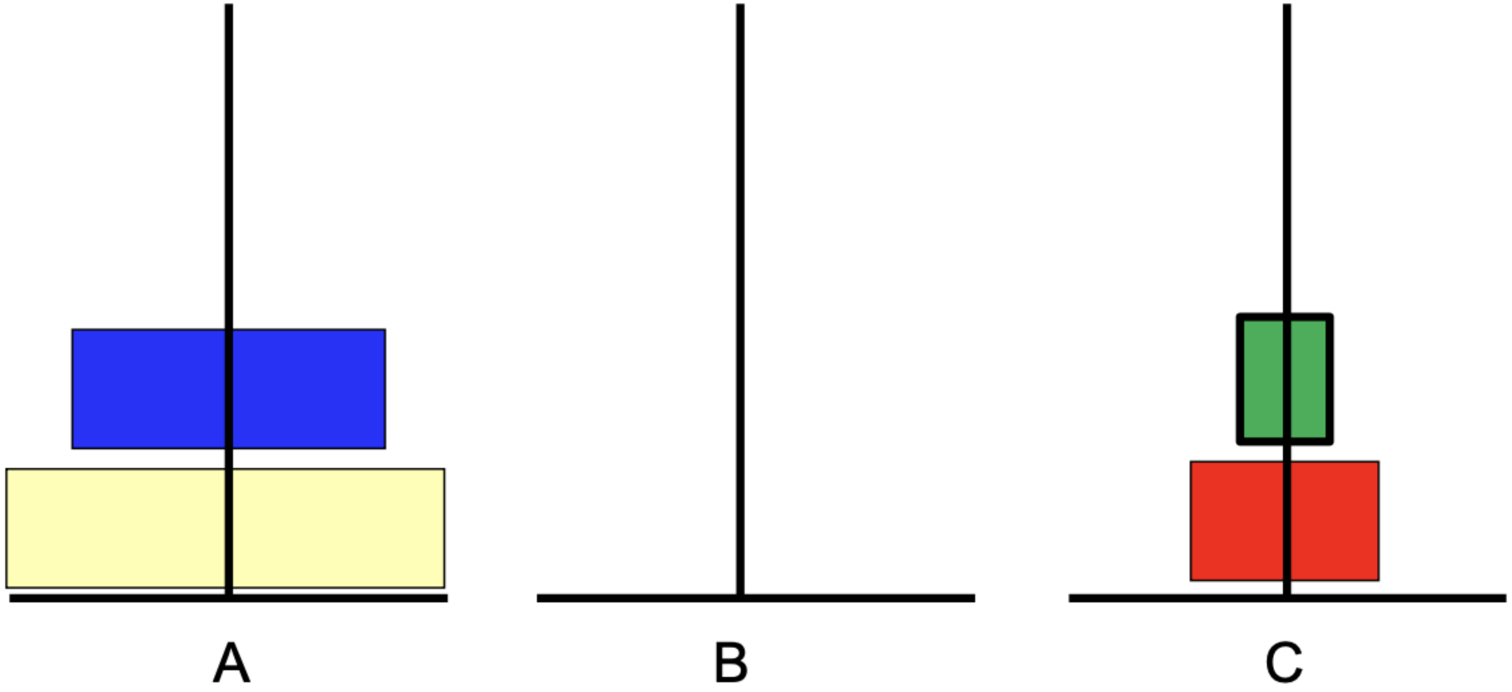
- Solving the smaller subproblem (moving n−1 disks) using the same algorithm.
- Handling the largest disk separately.
- Solving another smaller subproblem (moving n−1 disks again).

The n=3 Disks Example

- **Initial Setup**: Peg A has three disks (1, 2, 3, with 3 being the largest). Pegs B and C are empty.
- **Step 1**: Move n−1=2 disks (1 and 2) from A to B. This involves:
  - Moving disk 1 to C.
  - Moving disk 2 to B.
  - Moving disk 1 from C to B. **(All done one disk at a time.)**
- **Step 2**: Move the largest disk (3) from A to C. **(One disk moved.)**
- **Step 3**: Move the n−1=2 disks (1 and 2) from B to C. This involves:
  - Moving disk 1 to A.
  - Moving disk 2 to C.
  - Moving disk 1 from A to C. **(All done one disk at a time.)**

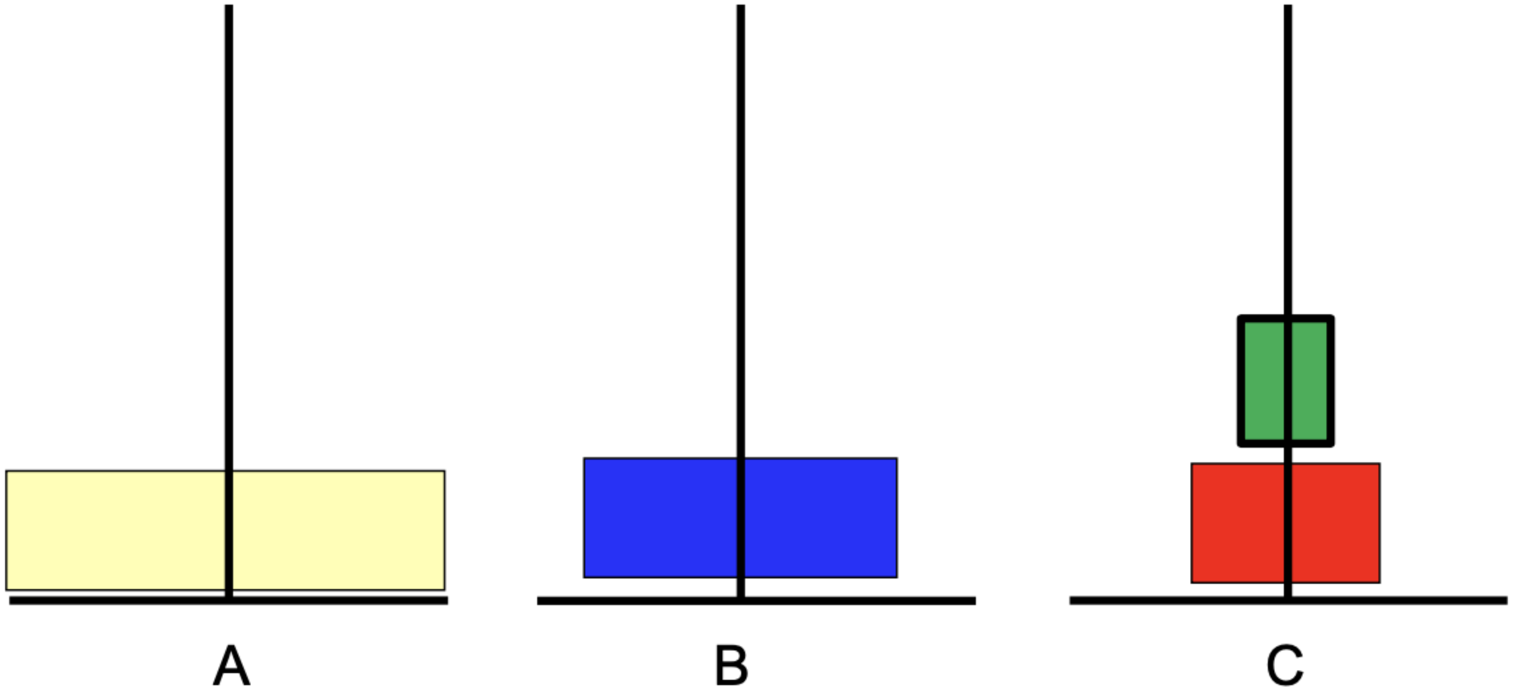
Move n-1 Disks from A to B

- How do we move n-1 disks from A to B? We use the same algorithm!
- Step 1: Move n-2 disks from A to C



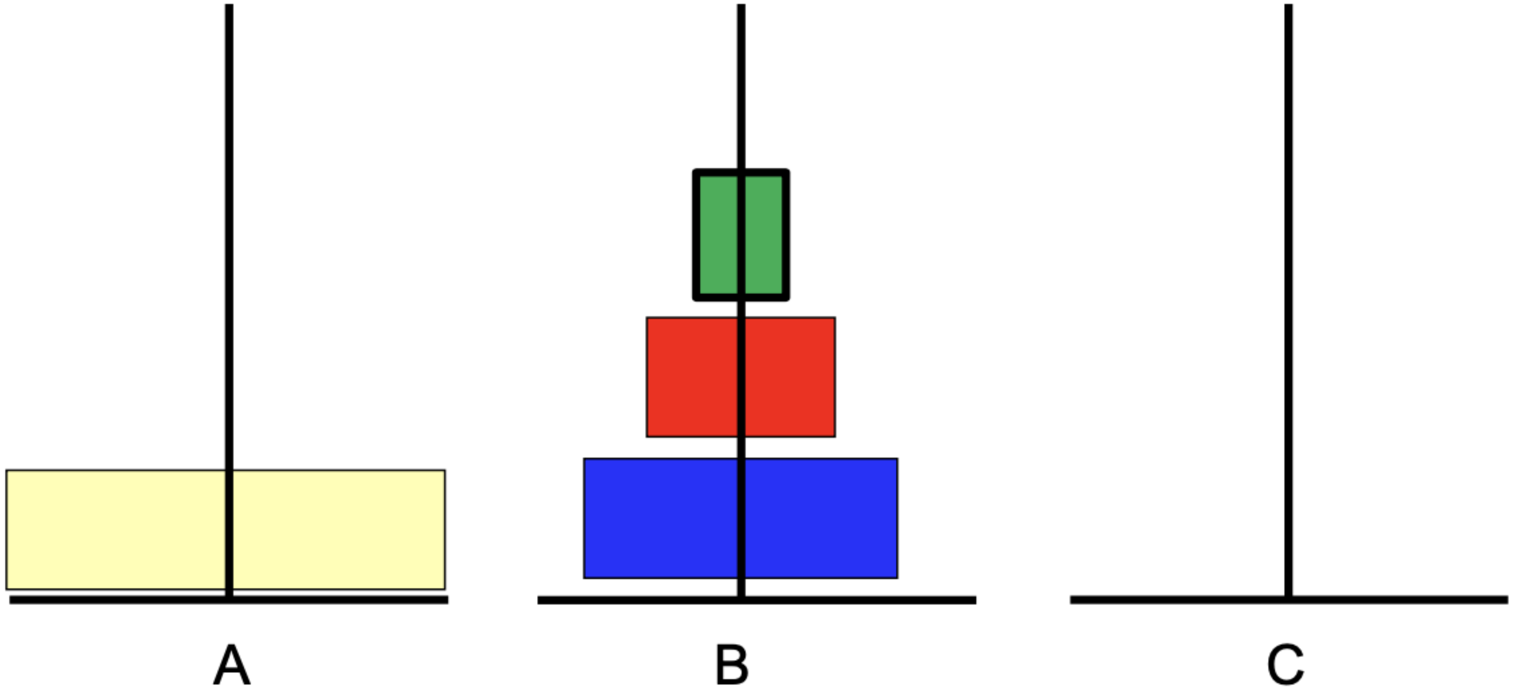
Move n-1 Disks from A to B

- How do we move n-1 disks from A to B? We use the same algorithm!
- Step 2: Move 1 disk from A to B



Move n-1 Disks from A to B

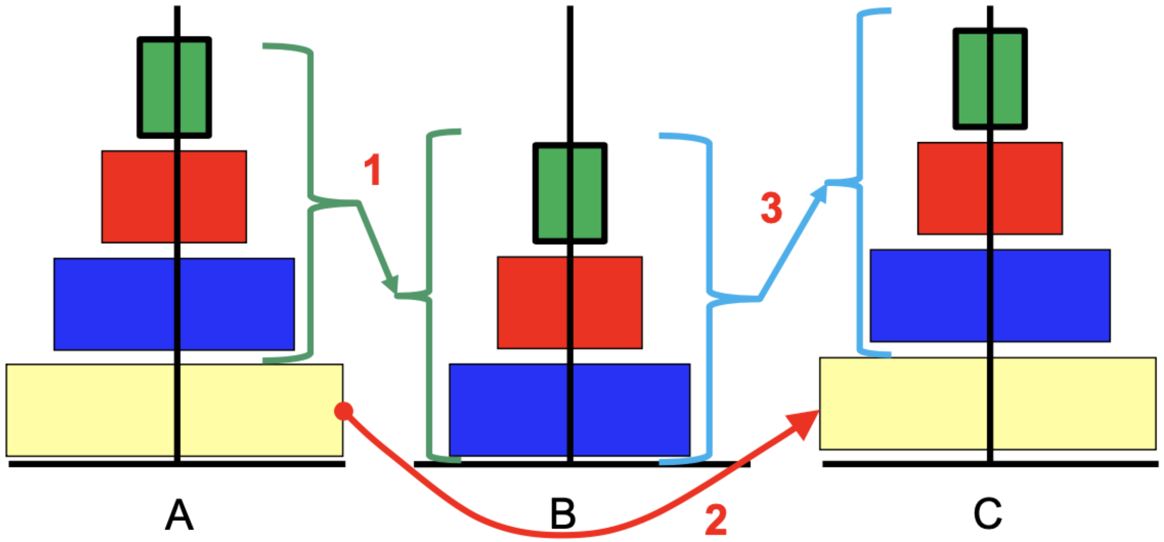
- How do we move n-1 disks from A to B? We use the same algorithm!
- Step 3: Move n-2 disks from C to B



Recall:Algorithm

Recursive case (n > 1):

- Move n-1 disks from the initial peg to the intermediate peg (*Solving the smaller subproblem*)
- Move 1 disk from the initial peg to the final peg (*handling the largest disk*)
- Move n-1 disks from the intermediate peg to the final peg (*Solving the smaller subproblem*)



Algorithm

Recursive case (n > 1):

- Move n-1 disks from the initial peg to the intermediate peg
- Move 1 disk from the initial peg to the final peg
- Move n-1 disks from the intermediate peg to the final peg

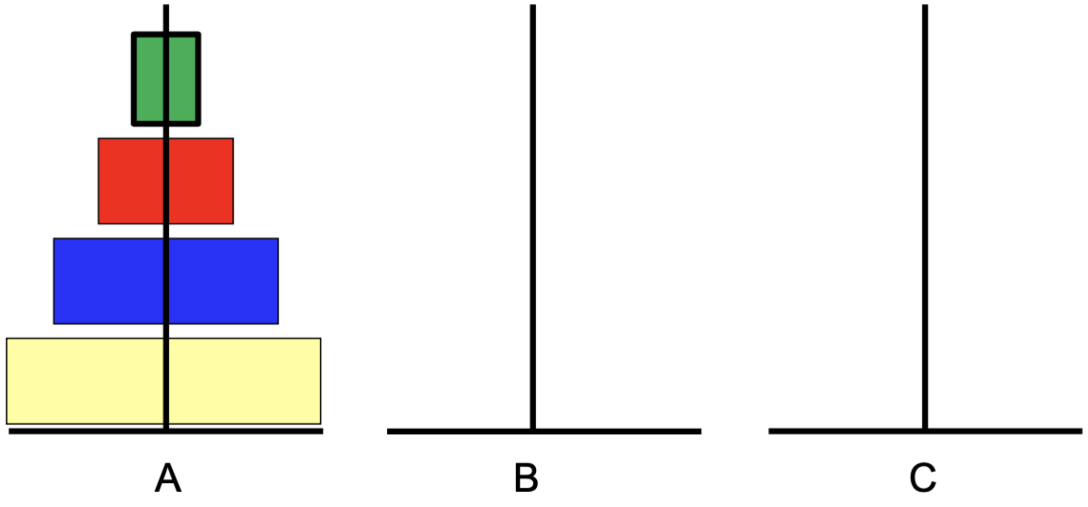
Base case (n = 1):

- Move one disk from the initial peg to the destination peg

The Towers of Hanoi Algorithm

```
Algorithm hanoi(A,C,B,n)
In: initial peg A, destination peg C, other
peg B, number of disks
Out: Sequence of moves to put all disks in
destination peg.

if n = 1, then move one disk from A to C
 else { hanoi (A,B,C,n-1)
 Move one disk from A to C
 hanoi (B,C,A,n-1)
}
```



The Algorithm in Java

```
public void hanoi (int A, int C, int B, int n) {
if (n == 1)
    System.out.println(“Move one disk from ” + A + “ to ” C);
else { hanoi(A,B,C,n-1)
    System.out.println (“Move one disk from ” + A + “ to ” C);
    hanoi(B,C,A,n-1)
}
```

- This is an example of a problem that would be difficult to solve without recursion

Inverting an Array

Inverting an Array

- We will consider another simple problem: inverting an array storing n values.

In:

12	6	2	8	9	15	21	3
0	1	2	3	4	5	6	7

⇩

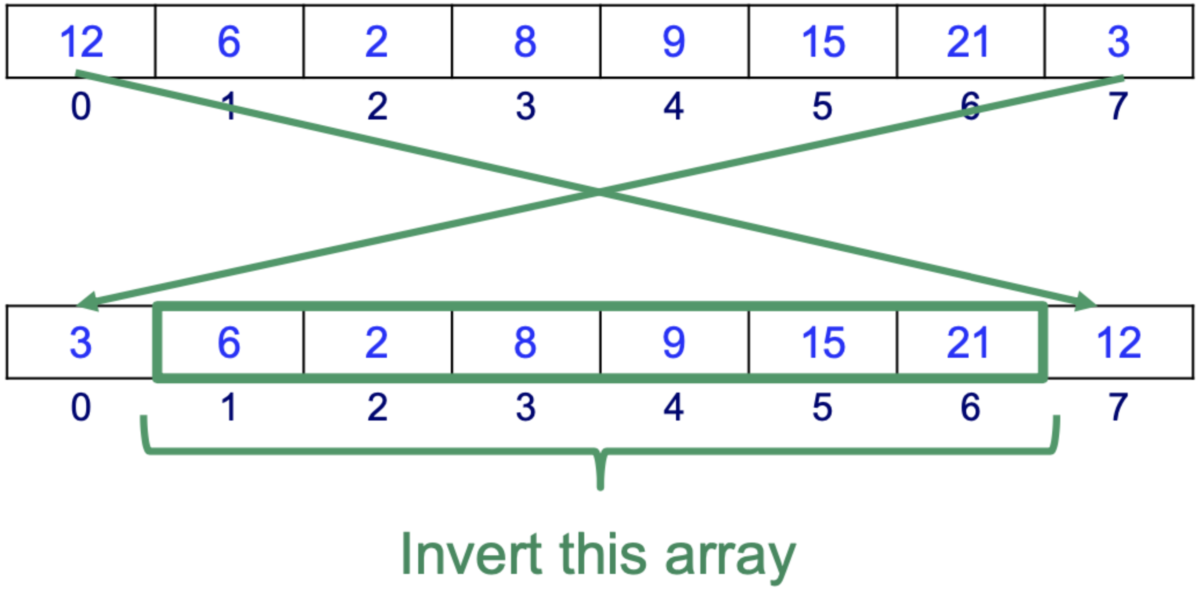
Out:

3	21	15	9	8	2	6	12
0	1	2	3	4	5	6	7

- To accomplish this, we can proceed as follows:
  - The value in position n-1 is swapped with the value in position 0, the value in position n-2 is swapped with the value in position 1, and so on.

Inverting an Array

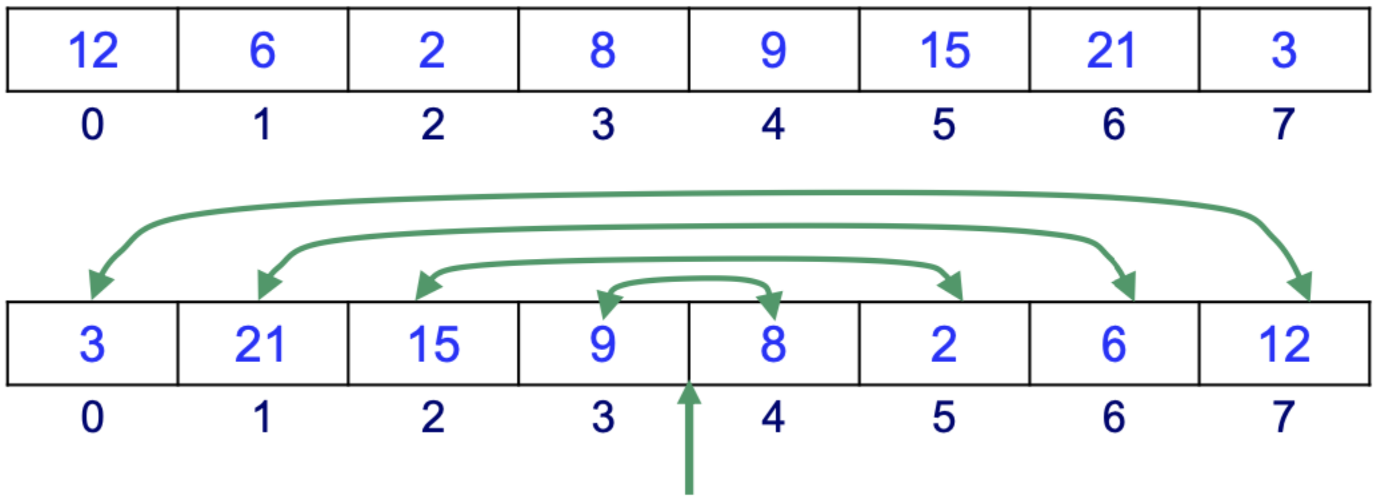
- *Recursive case* (n > 1):
  - Swap values in the first and last position
  - Invert sub-array from second to second-to-last positions



Inverting an Array

Base case (n = 1 or n = 0): Nothing to do

Even Case: the array size is even

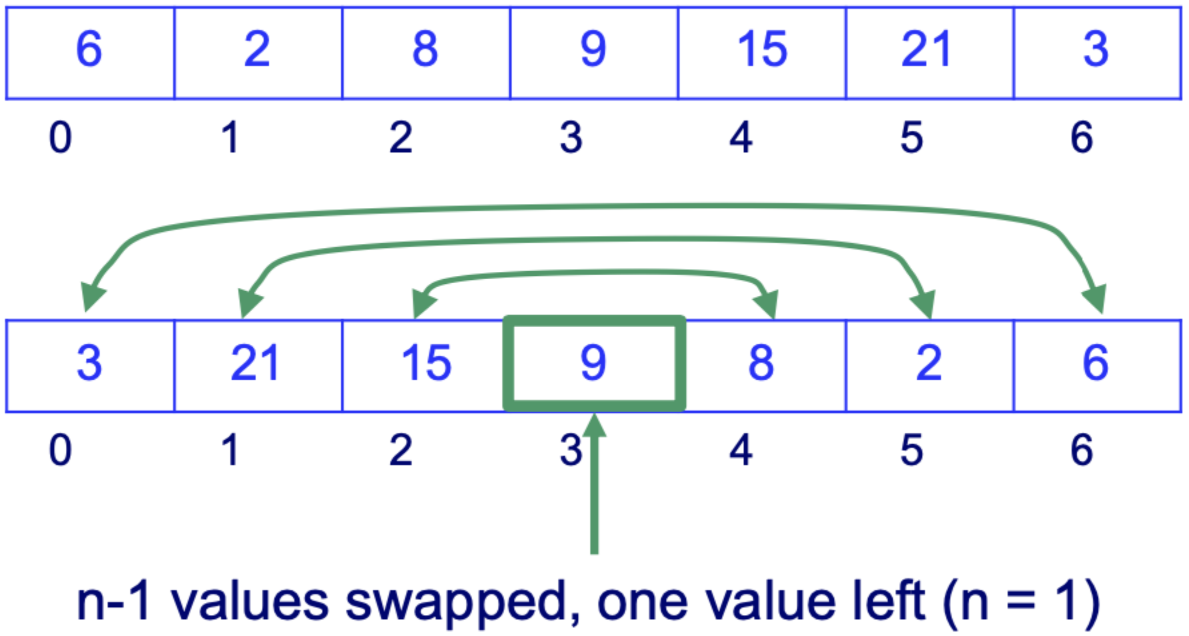


All values swapped, no values left to swap (n = 0)

Inverting an Array

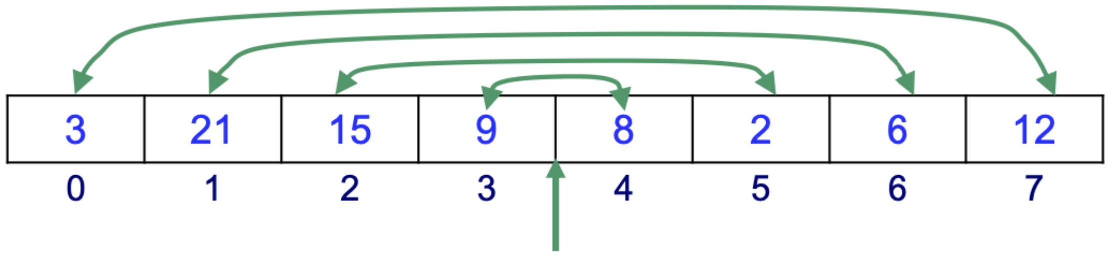
Base case (n = 1 or n = 0): Nothing to do

Odd Case: the array size is odd



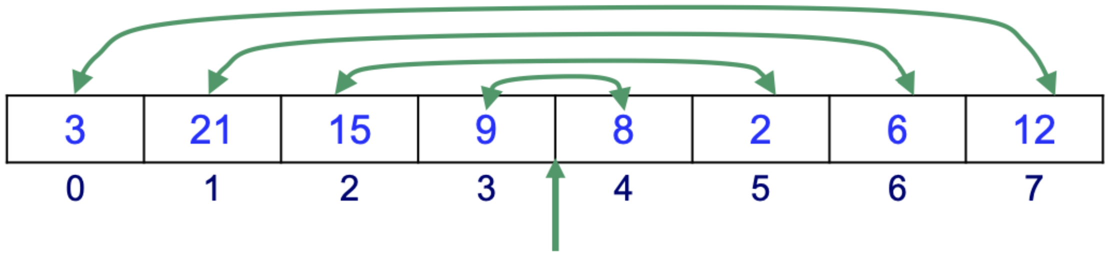
Inverting an Array Algorithm

```
Algorithm invert (arr,first,last)
Input: Array arr[first,…,last]: first is the index of the first value
and last is the index of the last value in the array
Output: Nothing, but the array is inverted
 if first < last then {
     // Swap first and last values
     tmp = arr[first]
     arr[first] = arr[last]
     arr[last] = tmp
     invert(arr,first+1,last-1)
 }
```



Inverting an Array In Java

```
public void invert (int[] arr, int first, int last) {
  if (first < last) {
  // Swap first and last values
  int tmp = arr[first];
  arr[first] = arr[last];
  arr[last] = tmp;
  invert(arr,first+1,last-1);
    }
}
```



Recursive Figures

Drawing Recursive Figures

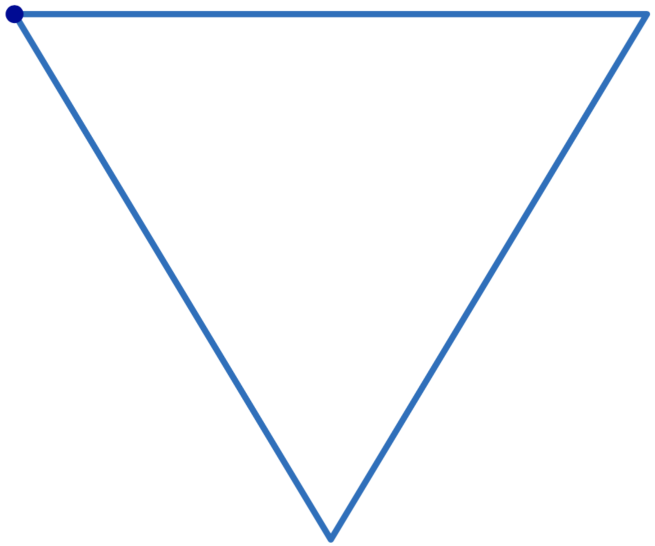
- We can draw complex figures called fractals using recursion.
- What is fractals? Fractals are complex geometrical figures that exhibit self-similarity, meaning they look similar at different scales. These patterns are often created by repeating a simple process over and over in an ongoing feedback loop.

Fractal Tree 4K /Source:Youtube/ Woltvint

Drawing Recursive Figures

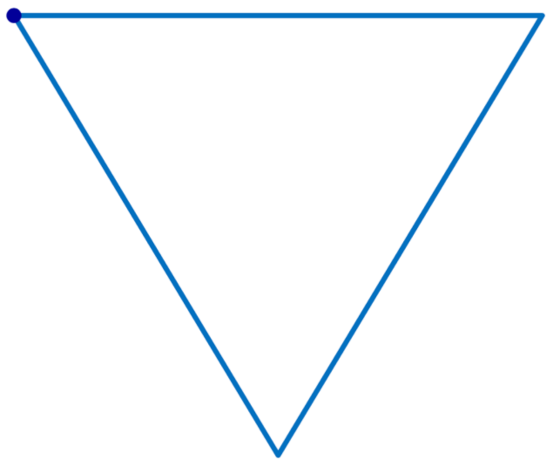
- Here is an example of a fractal called the Sierpinski Triangle:
  1. Select a point (x, y) and a length L

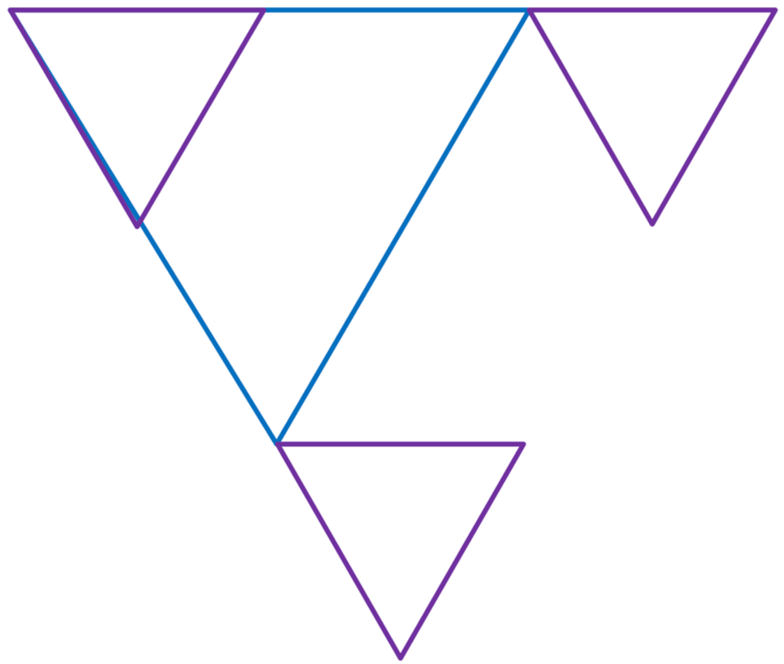
  2. Draw a triangle of length L with its left vertex at (x, y)



Drawing Recursive Figures

2. Draw a triangle of length L with its left vertex at (x, y)

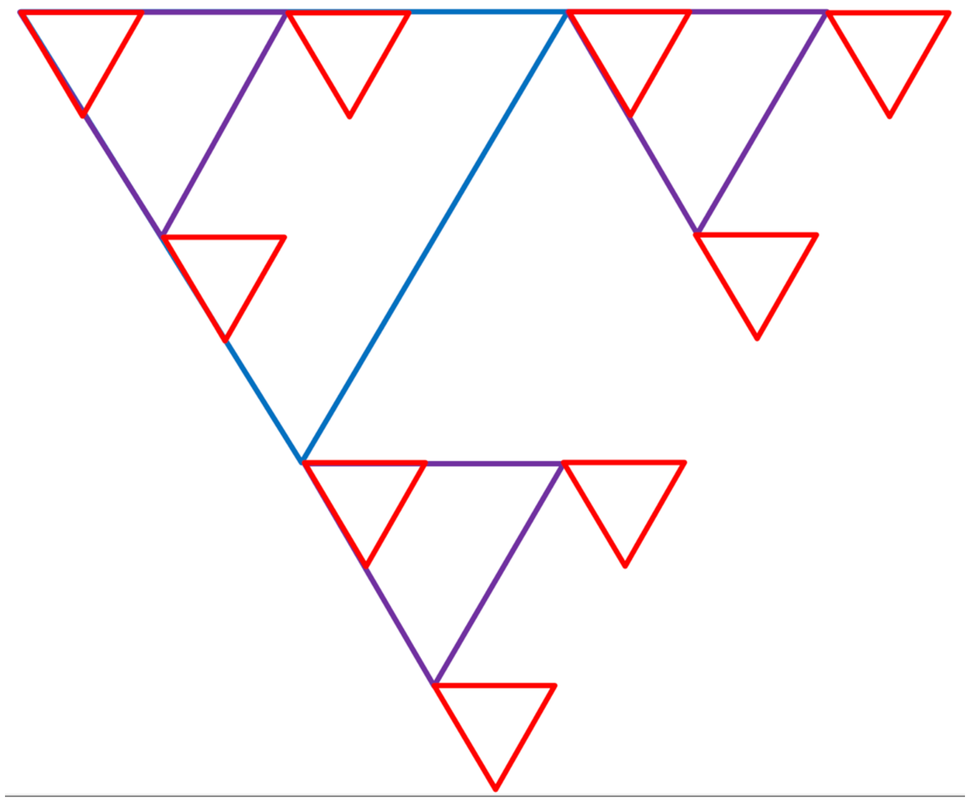




3. At each vertex of the triangle draw triangles of length L/2

Drawing Recursive Figures

4. Draw triangles of length L/4 at each vertex of the 3 triangles



Triangle Fractal

```
Algorithm triangleFractal (level, x, y, size)
Input: level (number of recursive calls), coordinates (x,y) of upper left vertex, size of triangle
Out: Draw the Sierpinski triangle

if level > 0 then {
    Calculate the coordinates (x1,y1), (x2,y2) of the other vertices of the triangle of the given size and left vertex at (x,y)
    Draw a triangle with the given vertices
        triangleFractal(level - 1, x, y, size / 2)
        triangleFractal(level - 1, x1, y1, size / 2)
        triangleFractal(level - 1, x2, y2, size / 2)
}
```

Triangle Fractal

```
private void triangleFractal(int level, int x, int y, int size) {
    if (level > 0) {
        // Compute the coordinates of the vertices
        int x1 = x + size;
        int y1 = y;
        int x2 = x + size/2;
        int height = (int)(size * 0.866); // height = size * cosine(60)
        int y2 = y + height;
        drawTriangle(x,y,x1,y1,x2,y2);
        triangleFractal(level - 1, x, y, size/2);
        triangleFractal(level - 1, x1, y1, size/2);
        triangleFractal(level - 1, x2, y2, size/2);
    }
}
```

Applications of Fractals

- Fractals are not just beautiful images; they have important applications:
  - **Medicine** – Healthy blood cells grow in a fractal pattern, so cancerous cells can be detected when they grow in an abnormal pattern.
  - **Image compression** – Fractals allow compact representation of very complex shapes.
  - **Electronics** – High-performance antennas have fractal shapes.
  - **Computer Simulation** – Fractals mimic natural landscapes.

Fractals

- Use the provided *Fractal.java* class to generate some fractals.
- Study and modify the code to use different figures to create other fractals.


Thank you