Recursion

WIA1002/WIB1002 : Data Structure

Recursion

- Programming technique where **a method calls itself** to fulfil its overall purpose.
- Also known as **Self-Invocation**

Characteristics of Recursion

All recursive methods have the following characteristics:

- **One or more base cases** (the simplest case) are used to stop recursion.
- **Recursive case** - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

To solve a problem using recursion, break it into **subproblems** that resemble the original problem in nature but with a smaller size. Apply the same approach to solve the subproblem recursively.

Computing Factorial

3! = 3 * 2 * 1;
5! = 5 * 4 * 3 * 2 * 1;

The factorial of a number **n** can be recursively defined as follows:

0! = 1; //Base Case

n! = n * (n - 1)!; n > 0 //Recursive Case

How do you find **n!** for a given **n**?

To find **1!** is easy, because you know that **0!** is **1**, and **1!** is **1** × **0!**. Assuming that you know **(n - 1)!**, you can obtain **n!** immediately by using **n** × **(n - 1)!**. Thus, the problem of computing **n!** is reduced to computing **(n - 1)!**. When computing **(n - 1)!**, you can apply the same idea recursively until **n** is reduced to **0**.

Computing Factorial

Let **factorial(n)** be the method for computing **n!**.

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

LISTING 18.1 ComputeFactorial.java

```java
import java.util.Scanner;

public class ComputeFactorial {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a nonnegative integer: ");
    int n = input.nextInt();

    // Display factorial
    System.out.println("Factorial of " + n + " is " + factorial(n));
  }

  /** Return the factorial for the specified number */
  public static long factorial(int n) {
    if (n == 0) // Base case
      return 1;
    else
      return n * factorial(n - 1); // Recursive call
  }
}
```

base case

recursion

```
Enter a nonnegative integer: 4 ↵Enter
Factorial of 4 is 24
```

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

1. temporarily suspended until invocation complete

2. Invocation complete when it reaches base case (n==0)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1)))

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1)))

= 4 * (3 * (2 * 1))

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1)))

= 4 * (3 * (2 * 1))

= 4 * (3 * 2)

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1)))

= 4 * (3 * (2 * 1))

= 4 * (3 * 2)

= 4 * 6

animation

Computing Factorial

factorial(0) = 1;

factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

= 4 * (3 * (2 * (1 * factorial(0))))

= 4 * (3 * (2 * (1 * 1)))

= 4 * (3 * (2 * 1))

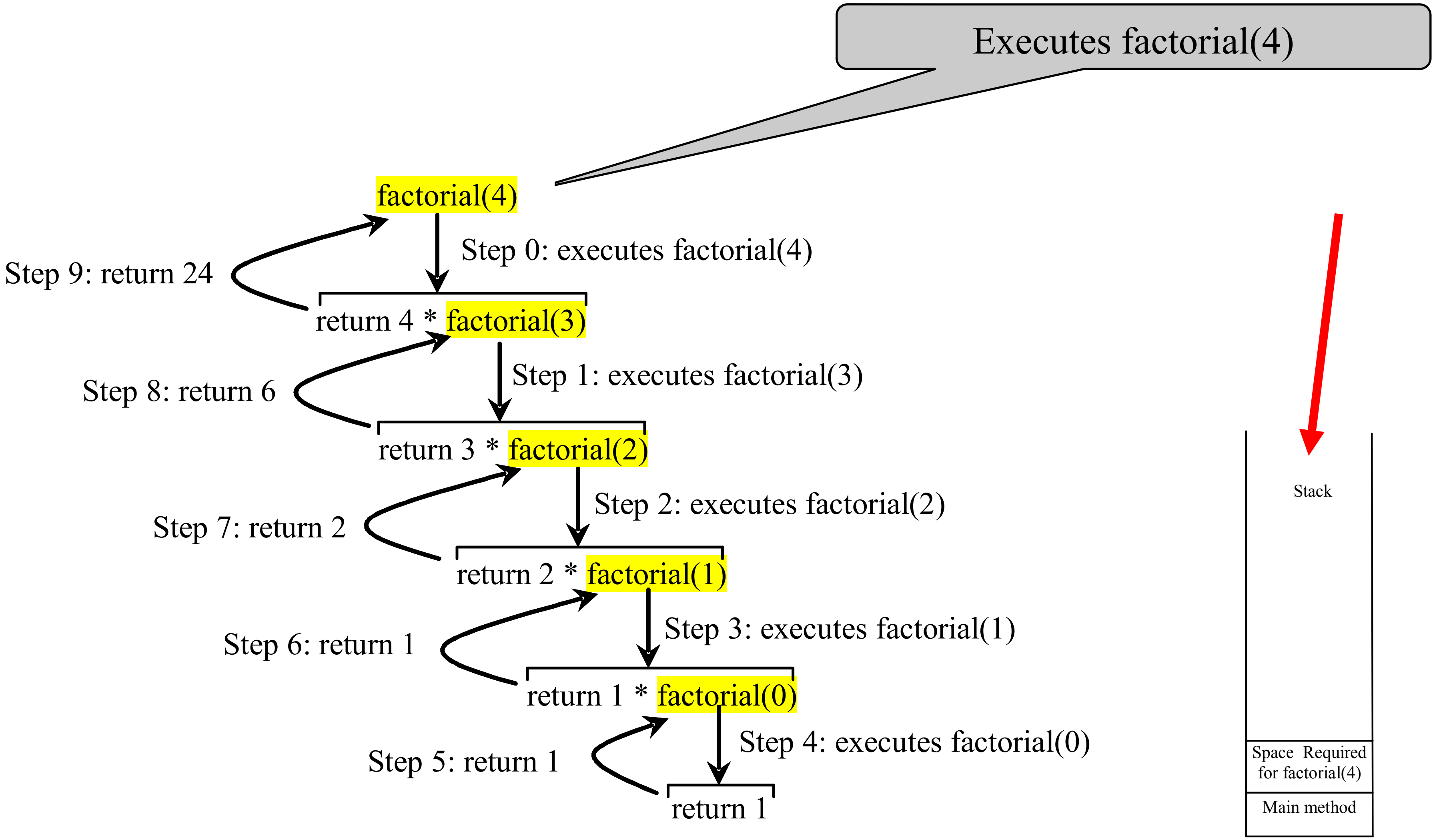
= 4 * (3 * 2)

= 4 * 6

= 24

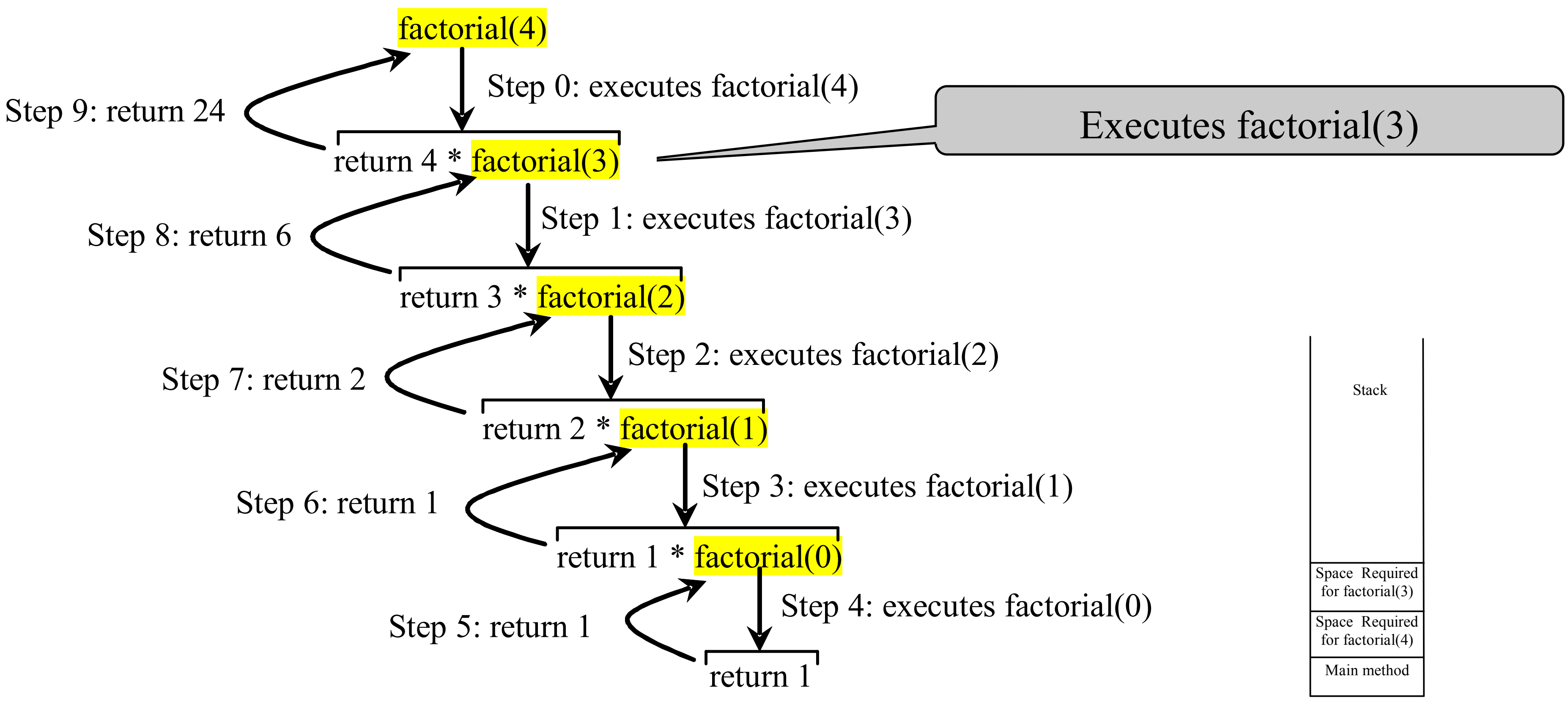
animation

Trace Recursive factorial



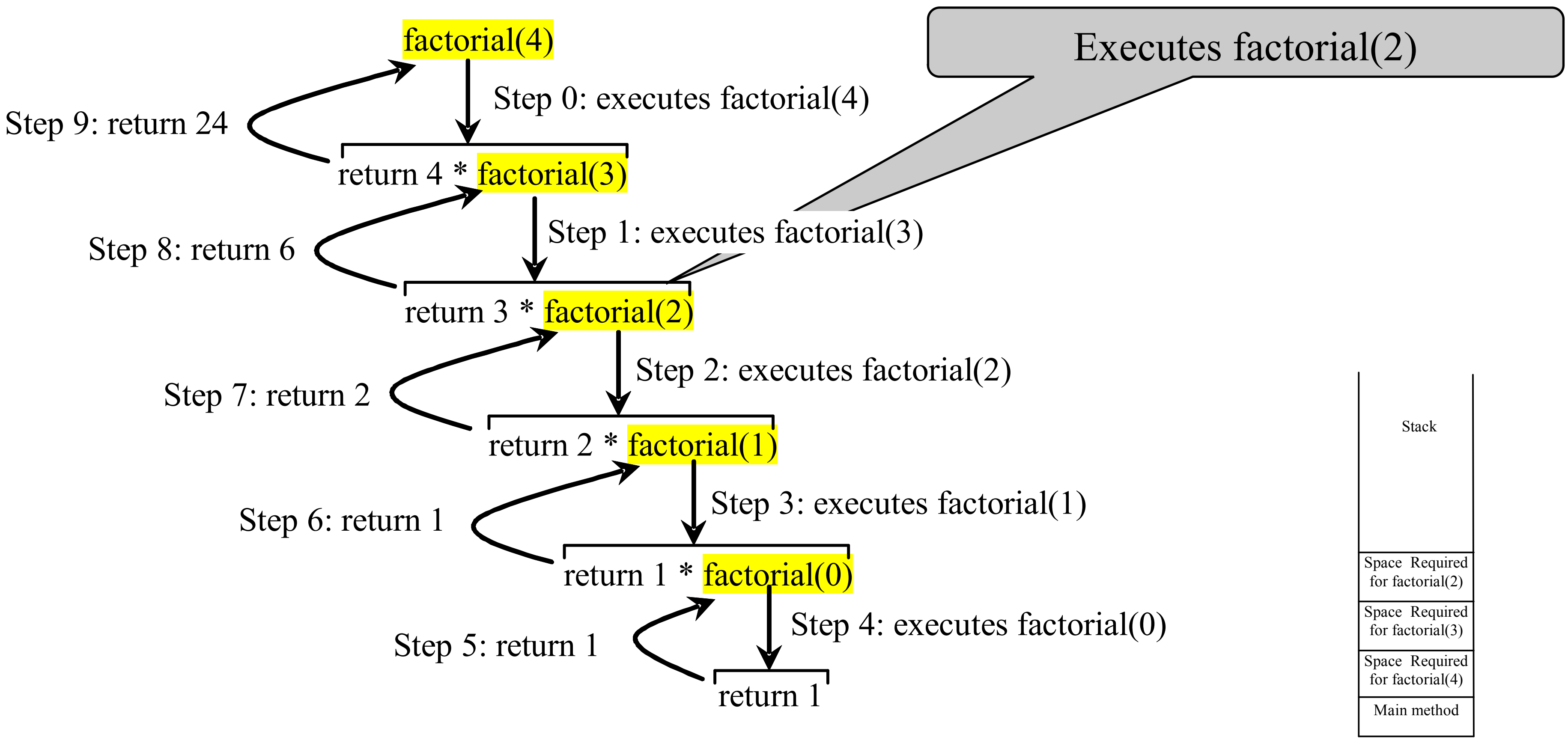
animation

Trace Recursive factorial



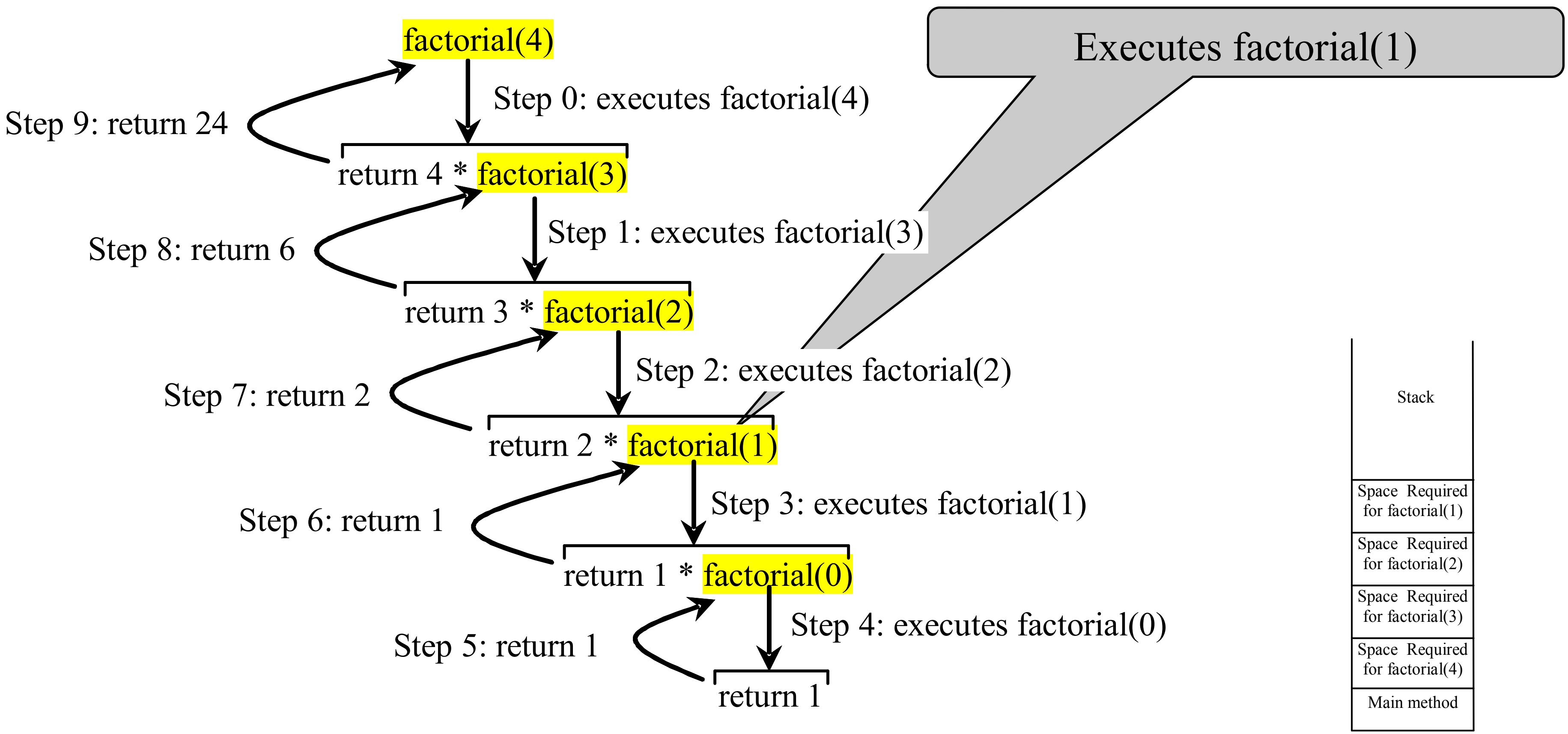
animation

Trace Recursive factorial



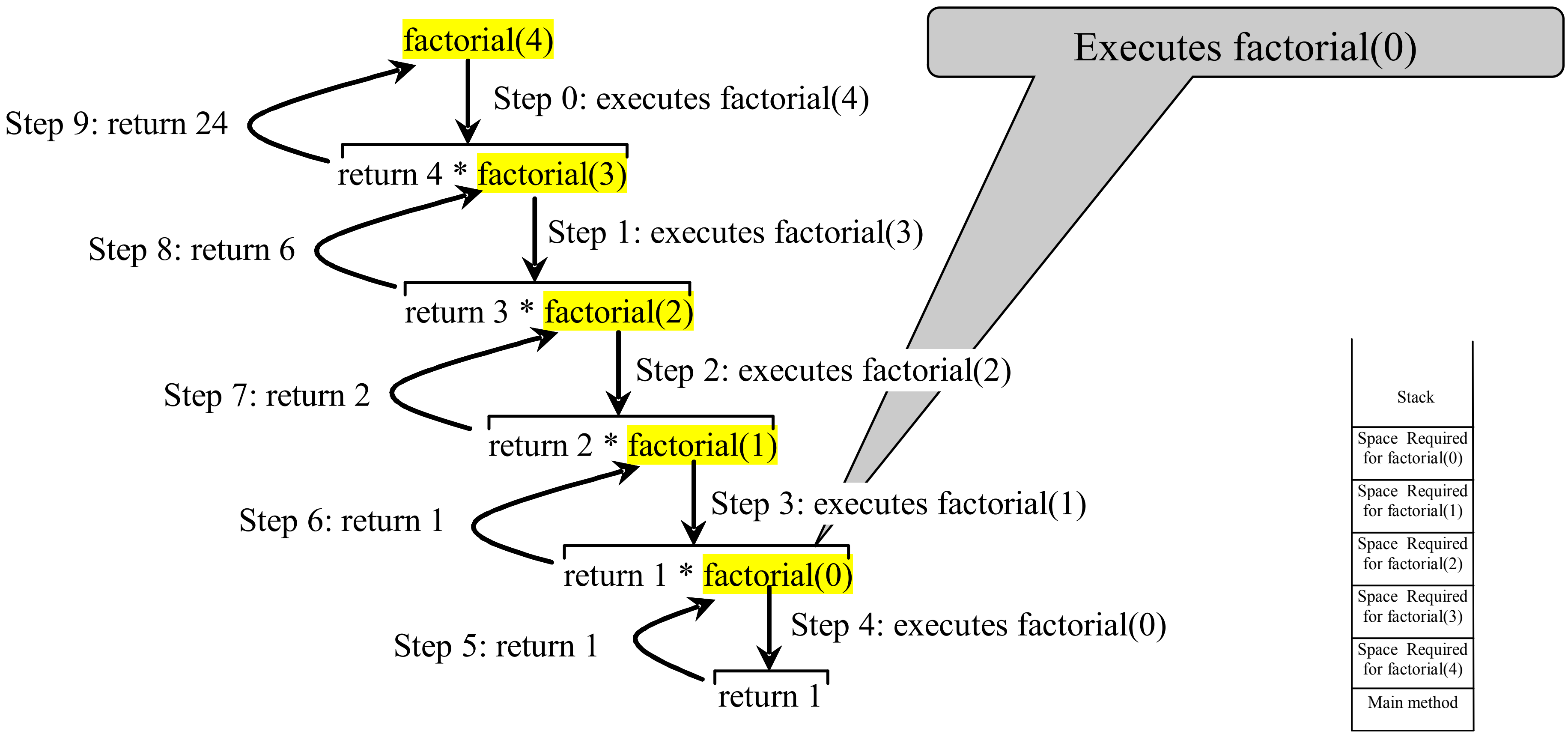
animation

Trace Recursive factorial



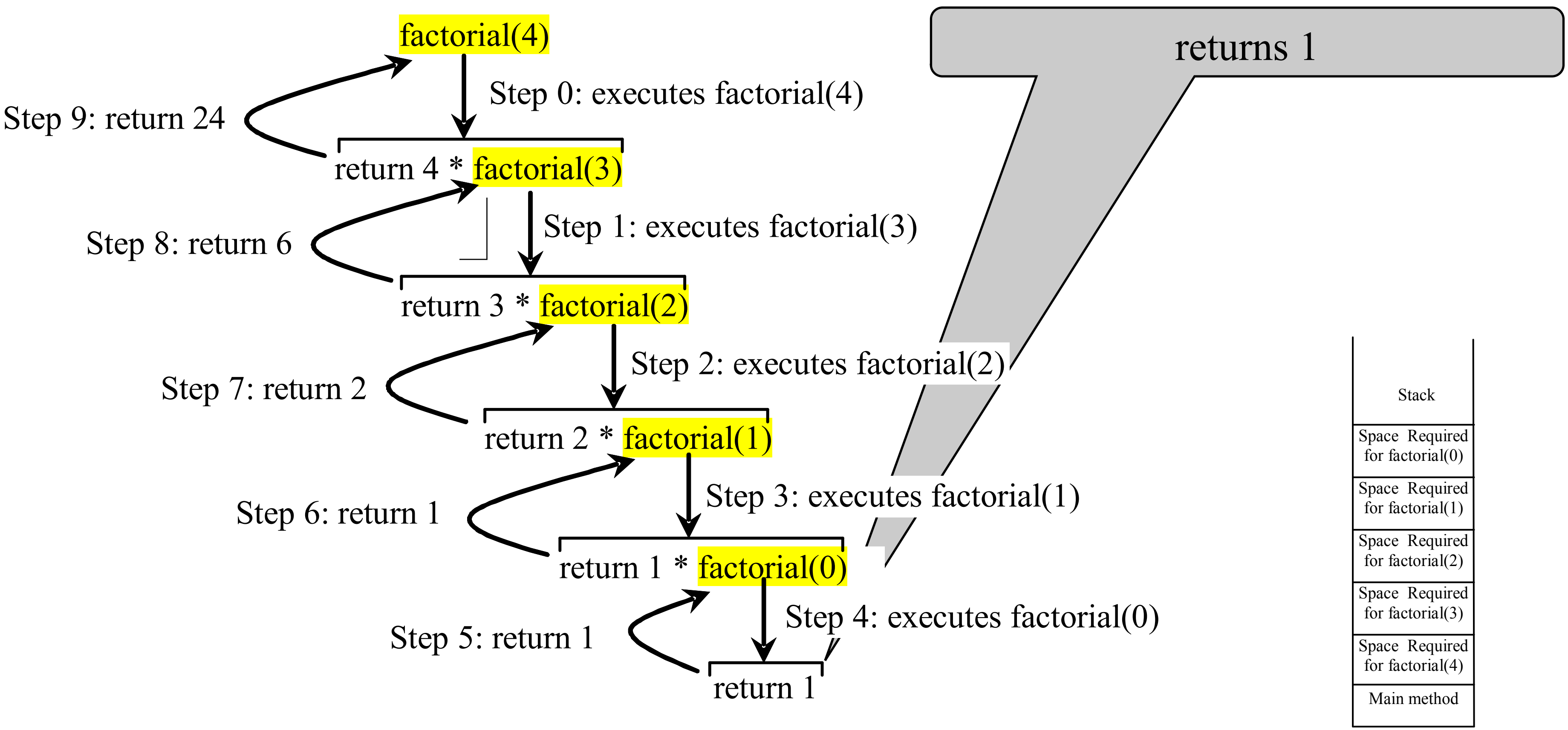
animation

Trace Recursive factorial



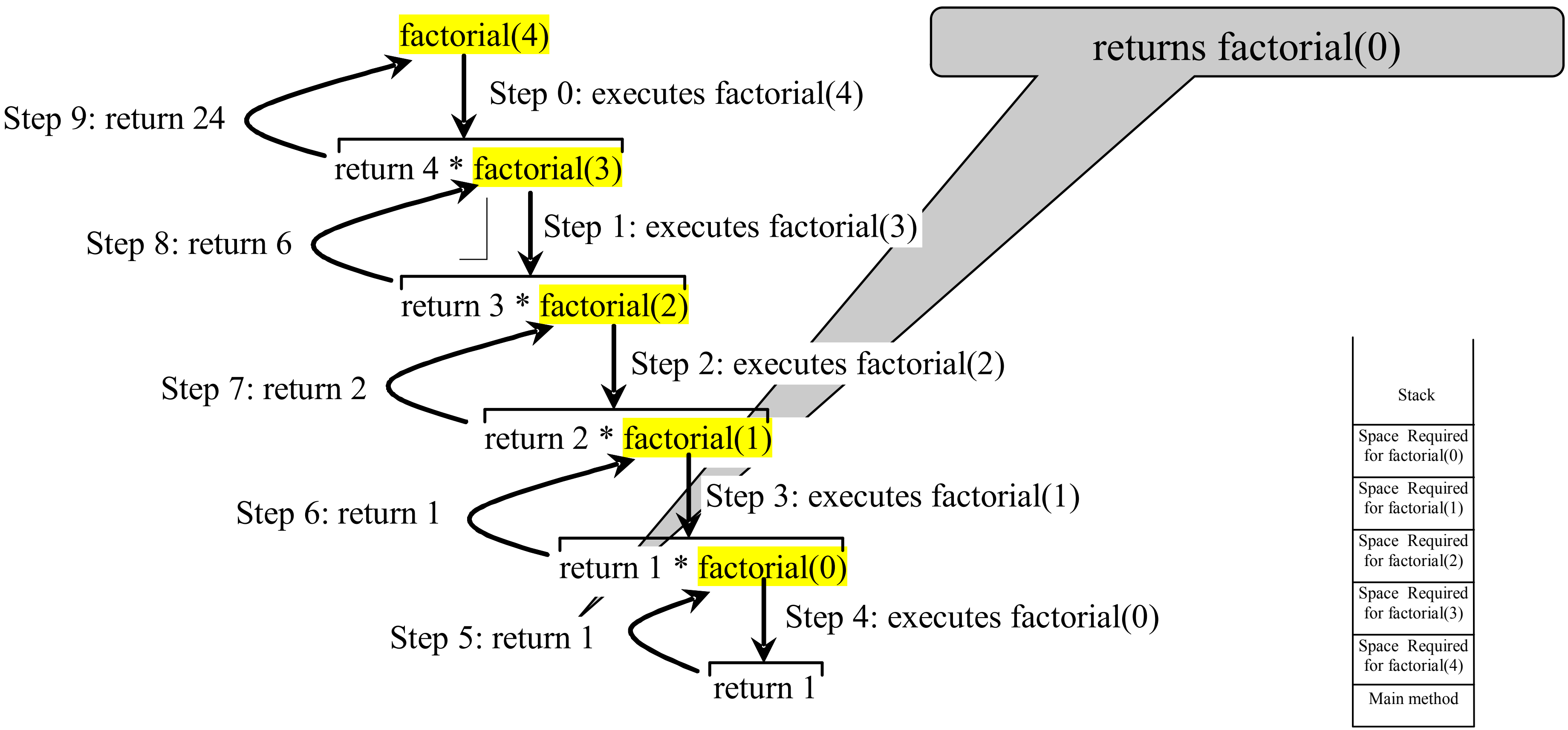
animation

Trace Recursive factorial



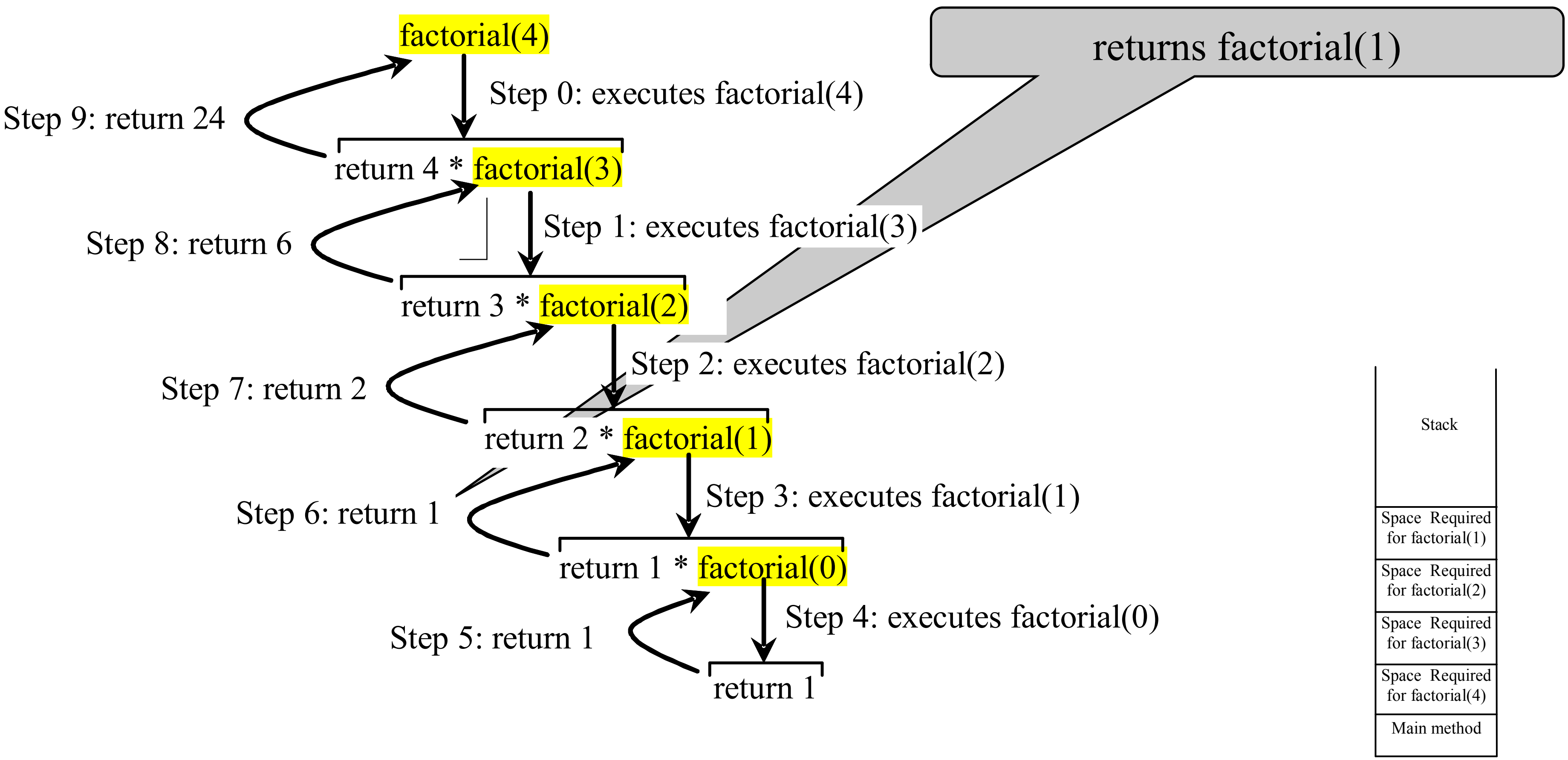
animation

Trace Recursive factorial



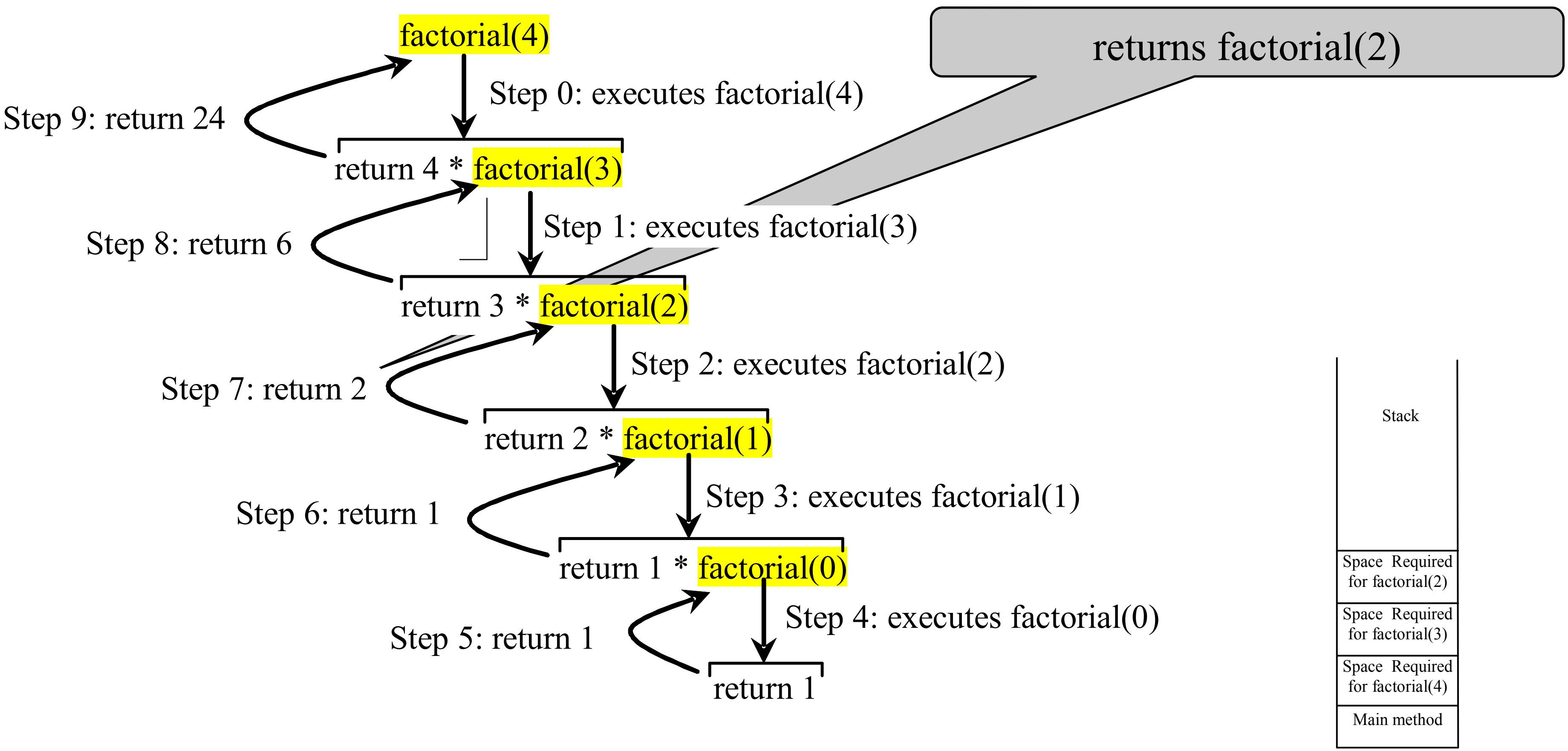
animation

Trace Recursive factorial



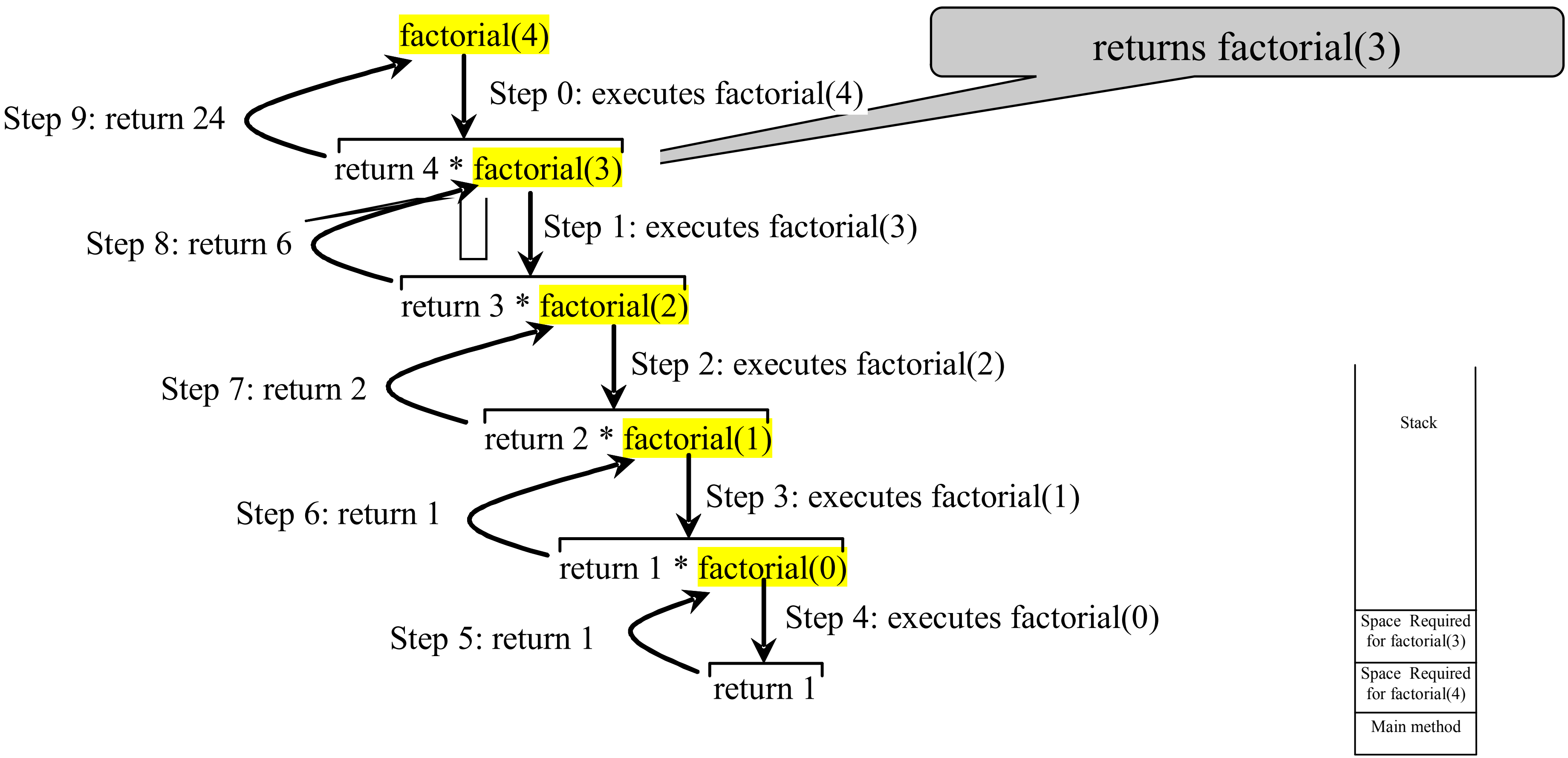
animation

Trace Recursive factorial



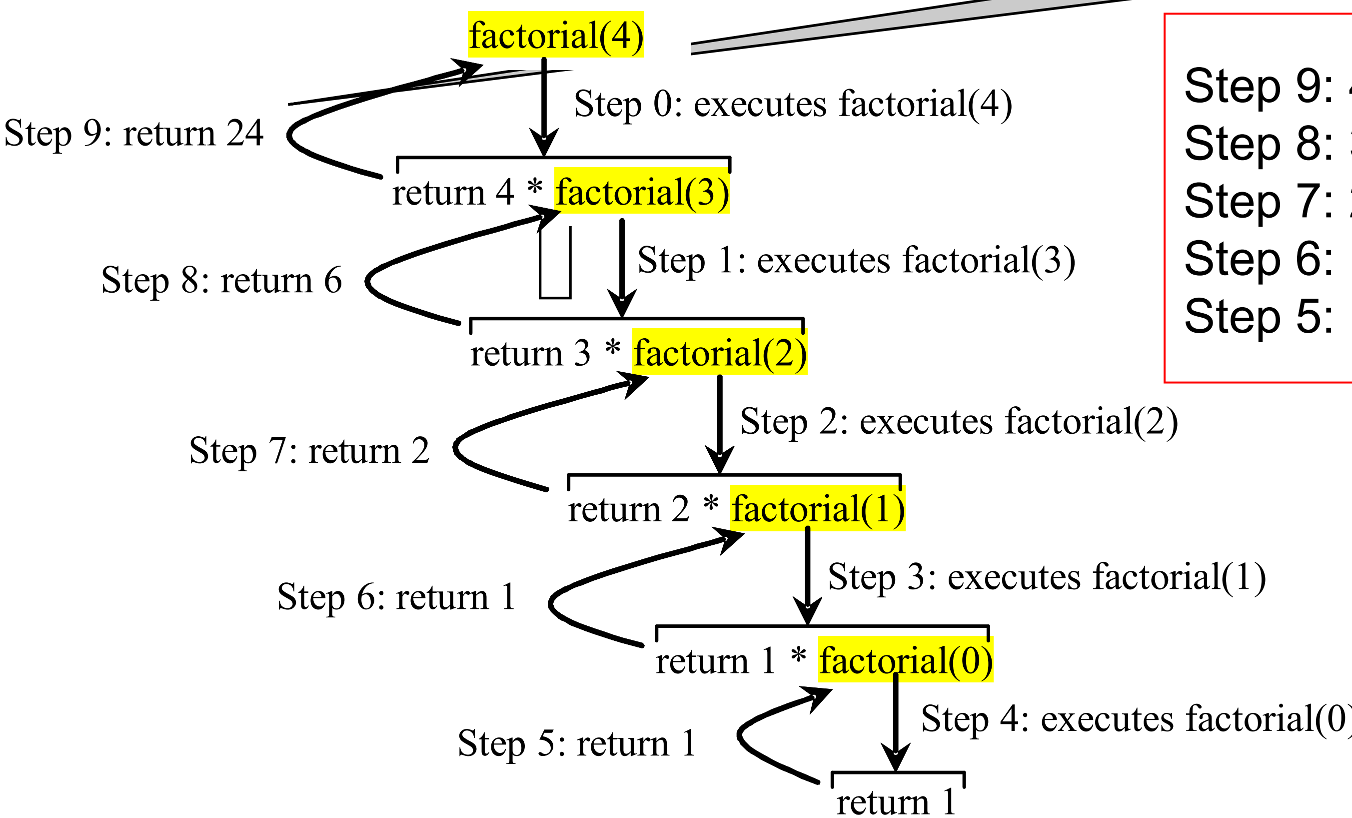
animation

Trace Recursive factorial

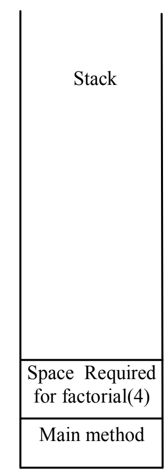


animation

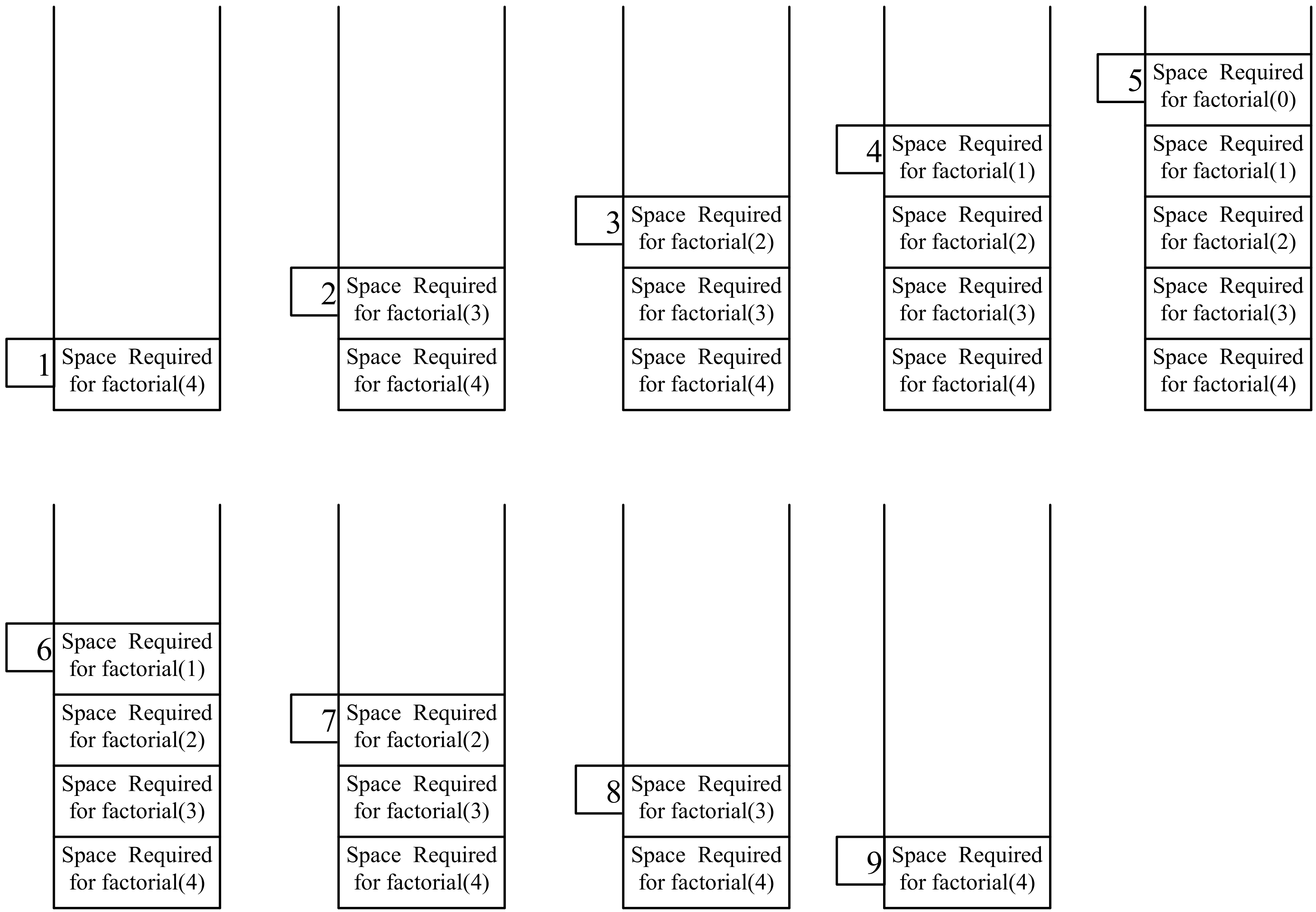
Trace Recursive factorial



Step 9: 4 * 6 = 24 Final Answer
Step 8: 3 * 2 = 6 Go Up
Step 7: 2 * 1 = 2 Go Up
Step 6: 1 * 1 = 1 Go Up
Step 5: 1 Go Up



factorial(4) Stack Trace



Fibonacci Numbers

```
Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89…

          indices: 0 1 2 3 4 5 6 7  8  9  10 11
```

The Fibonacci series begins with **0** and **1**, and each subsequent number is the sum of the preceding two. The series can be recursively defined as:

fib(0) = 0;   //Base case

fib(1) = 1;   //Base case

fib(index) = fib(index -1) + fib(index -2); index >=2   //Recursive Case

```
fib(3) = fib(2) + fib(1)
       = (fib(1) + fib(0)) + fib(1)
       = (1 + 0) +fib(1)
       = 1 + fib(1)
       = 1 + 1
       = 2
```

Fibonacci Numbers

How do you find **fib(index)** for a given **index**?

It is easy to find **fib(2)**, because you know **fib(0)** and **fib(1)**. Assuming that you know **fib(index - 2)** and **fib(index - 1)**, you can obtain **fib(index)** immediately. Thus, the problem of computing **fib(index)** is reduced to computing **fib(index - 2)** and **fib(index - 1)**. When doing so, apply the idea recursively until **index** is reduced to **0** or **1**.

The base case is **index = 0** or **index = 1**. If you call the method with **index = 0** or **index = 1**, it immediately returns the result. If you call the method with **index >= 2**, it divides the problem into two subproblems for computing **fib(index - 1)** and **fib(index - 2)** using recursive calls.

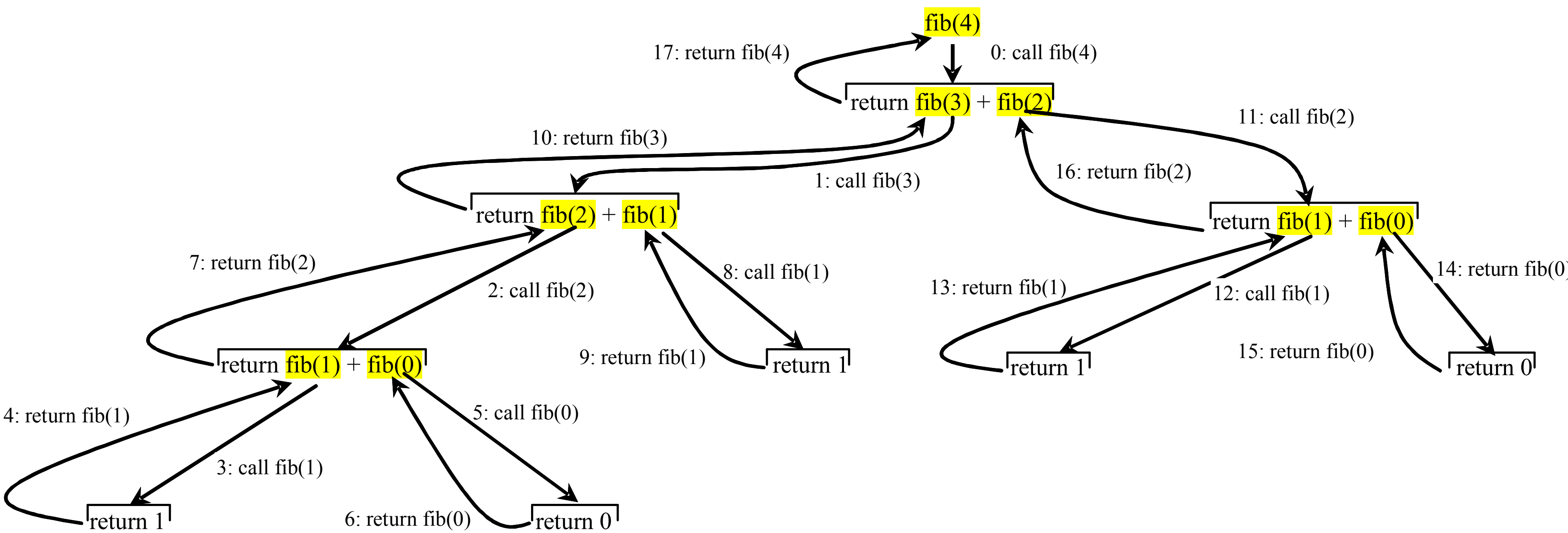
```java
import java.util.Scanner;

public class ComputeFibonacci {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for a Fibonacci number: ");
    int index = input.nextInt();

    System.out.println("The Fibonacci number at index "
      + index + " is " + fib(index));
  }

  /** The method for finding the Fibonacci number */
  public static long fib(long index) {
    if (index == 0) // Base case
      return 0;
    else if (index == 1) // Base case
      return 1;
    else  // Reduction and recursive calls
      return fib(index - 1) + fib(index - 2);
  }
}

```

```
Enter an index for a Fibonacci number: 1
The Fibonacci number at index 1 is 1
```

```
Enter an index for a Fibonacci number: 6
The Fibonacci number at index 6 is 8
```

```
Enter an index for a Fibonacci number: 7
The Fibonacci number at index 7 is 13
```

Fibonnaci Numbers, cont.



Characteristics of Recursion (Recap)

All recursive methods have the following characteristics:

- **One or more base cases** (the simplest case) are used to stop recursion.
- **Recursive case** - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Break a problem into **subproblems** that resemble the original problem in nature but with a smaller size. Apply the same approach to solve the subproblem recursively.

What happens if a recursive method never reaches a base case?

- Infinite recursion - occurs if recursion does not reduce the problem in a manner that allows it to eventually converge into the base case
- The stack will never stop growing.
- But OS limits the stack to a particular height, so that no program eats up too much memory.
- If a program's stack exceeds this size, the computer initiates an exception (**StackOverflowError)**, which typically would crash the program.

What is the output? What is the base case?

```
public static void main (String[] args)   {
  recursion(735);
  // System.out.println(result);
}

public static void recursion (int n) {
  if (n>0) {
     System.out.print(n%10);
     recursion(n/10);
  }
}
```

What is the output? What is the base case?

```java
public static void main (String[] args)   {
  recursion(735);
  // System.out.println(result);
}

public static void recursion (int n) {
  if (n>0) {
     System.out.print(n%10);
     recursion(n/10);
  }
}
```

Output : 537

Base case : n <= 0

What is the ouput?

```
public static long factorial(int n)
{
    return n * factorial(n - 1);
}
```

What is the ouput?

```
public static long factorial(int n) {
    return n * factorial(n - 1);
}
```

Output : The method runs infinitely and causes a StackOverflowError.

Recursion vs Iteration

- Recursion and loop are related concepts.
- Anything you can do with a loop, you can do with recursion, and vice versa.
- Sometimes recursion is simpler to write, and sometimes loop is, but in principle they are interchangeable.

Recursion vs Iteration

Implementing factorial using a loop:

```
public static long factorialLoop(int n){
    long result = 1;
    while (n>0) {
        result *= n;
        n--;
    }
    return result;
}
```

Recursion vs Iteration

Recursion

- Terminate when a base case is reached
- Each recursive call requires extra space on the stack frame (memory)
- If we get infinite recursion, it may result in stack overflow

Iteration

- Terminates when a condition is proven to be false
- Each iteration does not require any extra space
- An infinite loop could loop forever since there is no extra memory being created

Recursion

- is an alternative form of program control.
- repetition without a loop.
- substantial overhead –
  - the system must assign space for all of the method's local variables and parameters each time a method is called.
  - consume considerable memory and requires extra time to manage the additional space.
- However, it is good for solving the problems that are inherently recursive.

Problem Solving Using Recursion - Think Recursively

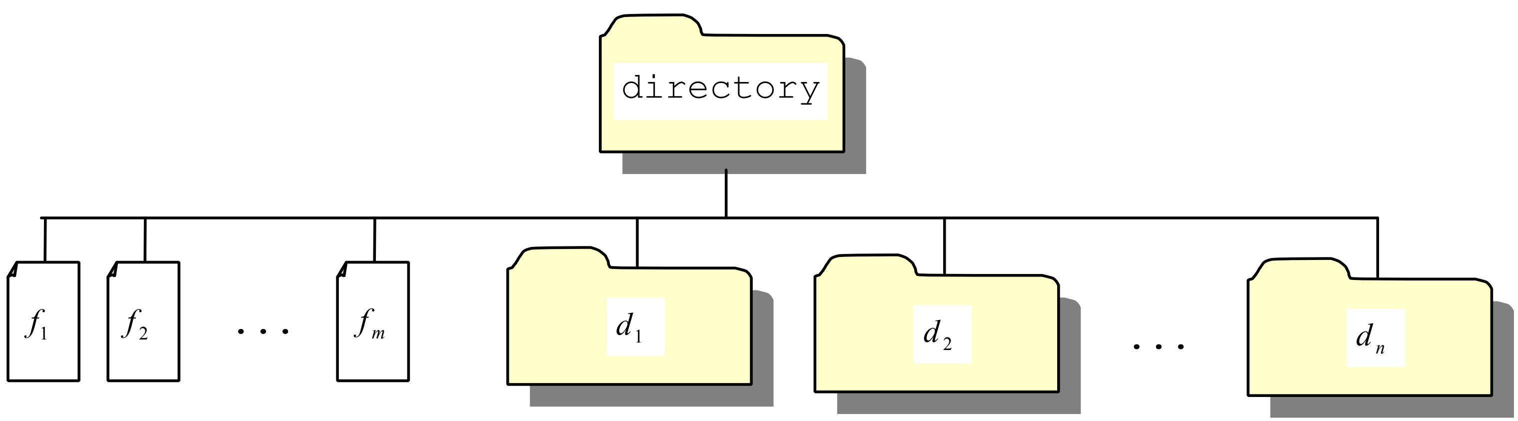
- Example:
  - a simple problem of printing a message for n times.
  - break the problem into two subproblems:
    - one problem is to print the message one time
    - the other problem is to print the message for n-1 times. The 2nd problem is the same as the original problem with a smaller size.
  - the base case for the problem is n==0.

```
public static void nPrintln(String message, int times)
{
    if (times >= 1) {
           System.out.println(message);
           nPrintln(message, times - 1);
    } // The base case is times == 0
}
```

nPrintln("Welcome", 5);

Directory Size

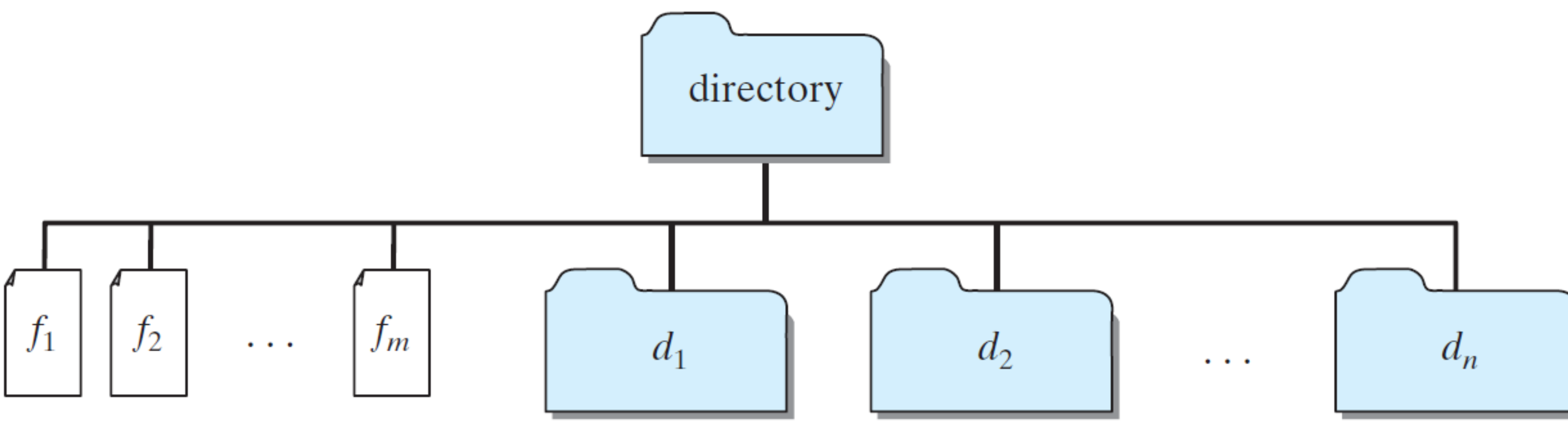
- A problem that is difficult to solve without using recursion.
- The size of a directory is the sum of the sizes of all files in the directory.
- A directory may contain subdirectories.



Directory Size

The size of the directory can be defined recursively as follows:

$$size(d) = size(f_1) + size(f_2) + \ldots + size(f_m) + size(d_1) + size(d_2) + \ldots + size(d_n)$$



LISTING 18.7 DirectorySize.java

```java
import java.io.File;
import java.util.Scanner;

public class DirectorySize {
  public static void main(String[] args) {
    // Prompt the user to enter a directory or a file
    System.out.print("Enter a directory or a file: ");
    Scanner input = new Scanner(System.in);
    String directory = input.nextLine();

    // Display the size
    System.out.println(getSize(new File(directory)) + " bytes");
  }

  public static long getSize(File file) {
    long size = 0; // Store the total size of all files

    if (file.isDirectory()) {
      File[] files = file.listFiles(); // All files and subdirectories
      for (int i = 0; files != null && i < files.length; i++) {
        size += getSize(files[i]); // Recursive call
      }
    }
    else { // Base case
      size += file.length();
    }

    return size;
  }
}
```

```
Enter a directory or a file: c:\book ↵Enter
48619631 bytes
```

```
Enter a directory or a file: c:\book\Welcome.java ↵Enter
172 bytes
```

References

Chapter 18 Recursion, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015