# Recursion

Introduction to Programming and Computational Problem Solving - 2

CSE 8B

Lecture 15

# Announcements

- Assignment 7 is due today, 11:59 PM
- Assignment 8 will be released today
  - Due Jun 1, 11:59 PM
- Reading
  - Liang
    - Chapter 18

# Recursion

- Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops
- A recursive method is one that invokes itself directly or indirectly

# Computing factorials

- Example

  4! = 4 * 3 * 2 * 1 = 24

- Remember, 0! = 1 (and 1! = 1)
- As a (non-recursive) method

```
public static long factorial(int n) {
  long nfactorial = 0 == n ? 1 : n;
  for (int i = n - 1; 1 < i; --i) {
    nfactorial *= i;
  }
  return nfactorial;
}
```

# Computing factorials

- Alternatively, think recursively

  0! = 1

  - *Base case* or *stopping condition*

  n! = n * (n – 1)!; n > 0

  - (n – 1)! is a *subproblem* of n! and is a *recursive call*
- Example

  4! = 4 * 3!

  4! = 4 * 3 * 2!

  4! = 4 * 3 * 2 * 1!

  4! = 4 * 3 * 2 * 1 * 0!

  4! = 4 * 3 * 2 * 1 * 1 = 24

# Computing factorials

0! = 1 `factorial(0) = 1`

n! = n * (n – 1)!; n > 0 `factorial(n) = n * factorial(n – 1)`

- As a recursive method

```
public static long factorial(int n) {
  if (0 == n) {
    // Base case
    return 1;
  }
  else {
    // Recursive call
    return n * factorial(n – 1);
  }
}
```

# Computing factorials

- Example

  4! = 4 * 3!

  4! = 4 * (3 * 2!)

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

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

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

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

  4! = 4 * (3 * 2)

  4! = 4 * 6

  4! = 24

0! = 1
n! = n * (n – 1)!; n > 0

# Computing factorials

- Example

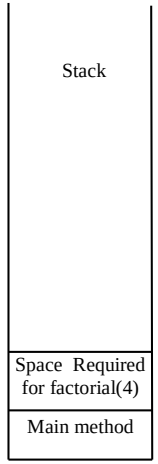
```
factorial(4) = 4 * factorial(3)
factorial(4) = 4 * (3 * factorial(2))
factorial(4) = 4 * (3 * (2 * factorial(1)))
factorial(4) = 4 * (3 * (2 * (1 * factorial(0))))
factorial(4) = 4 * (3 * (2 * (1 * 1)))
factorial(4) = 4 * (3 * (2 * 1))
factorial(4) = 4 * (3 * 2)
factorial(4) = 4 * 6
factorial(4) = 24
```

```
public static long factorial(int n) {
  if (0 == n) {
    // Base case
    return 1;
  }
  else {
    // Recursive call
    return n * factorial(n – 1);
  }
}
```

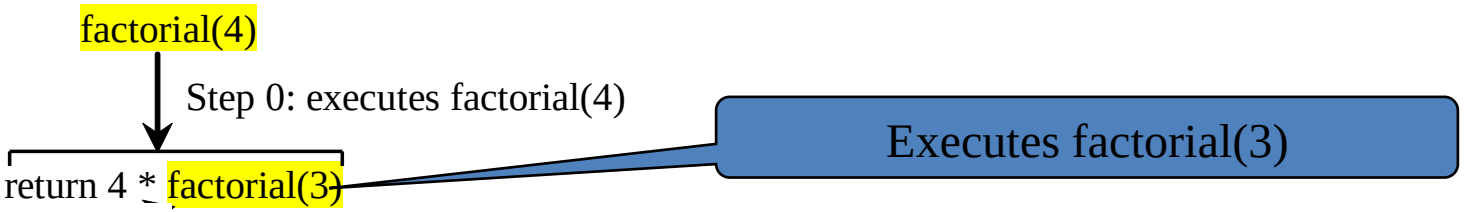
# Trace code

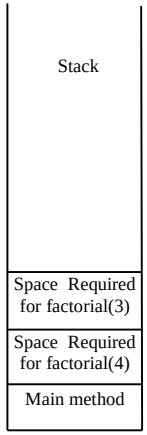
Executes factorial(4)

factorial(4)

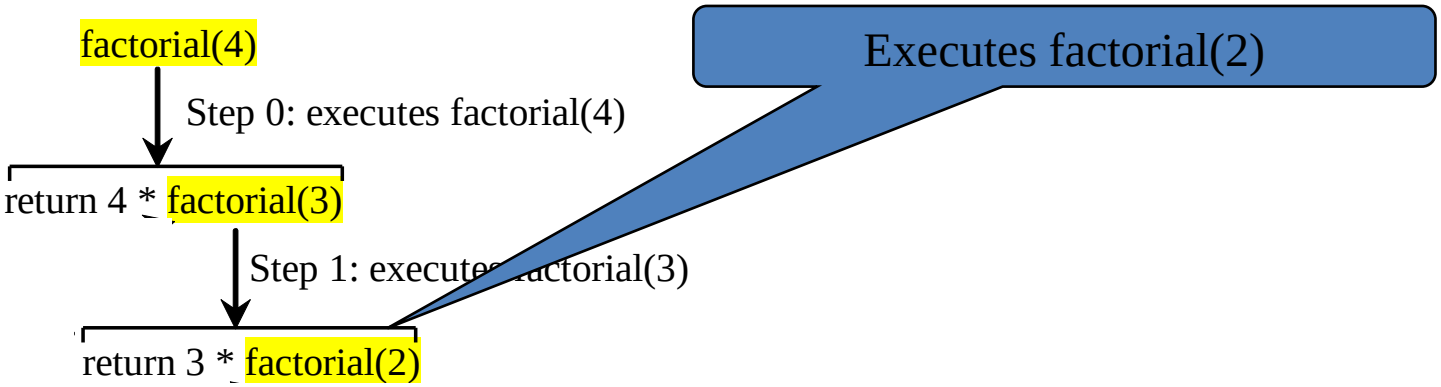


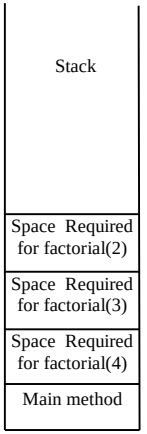
# Trace code


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Executes factorial(3)

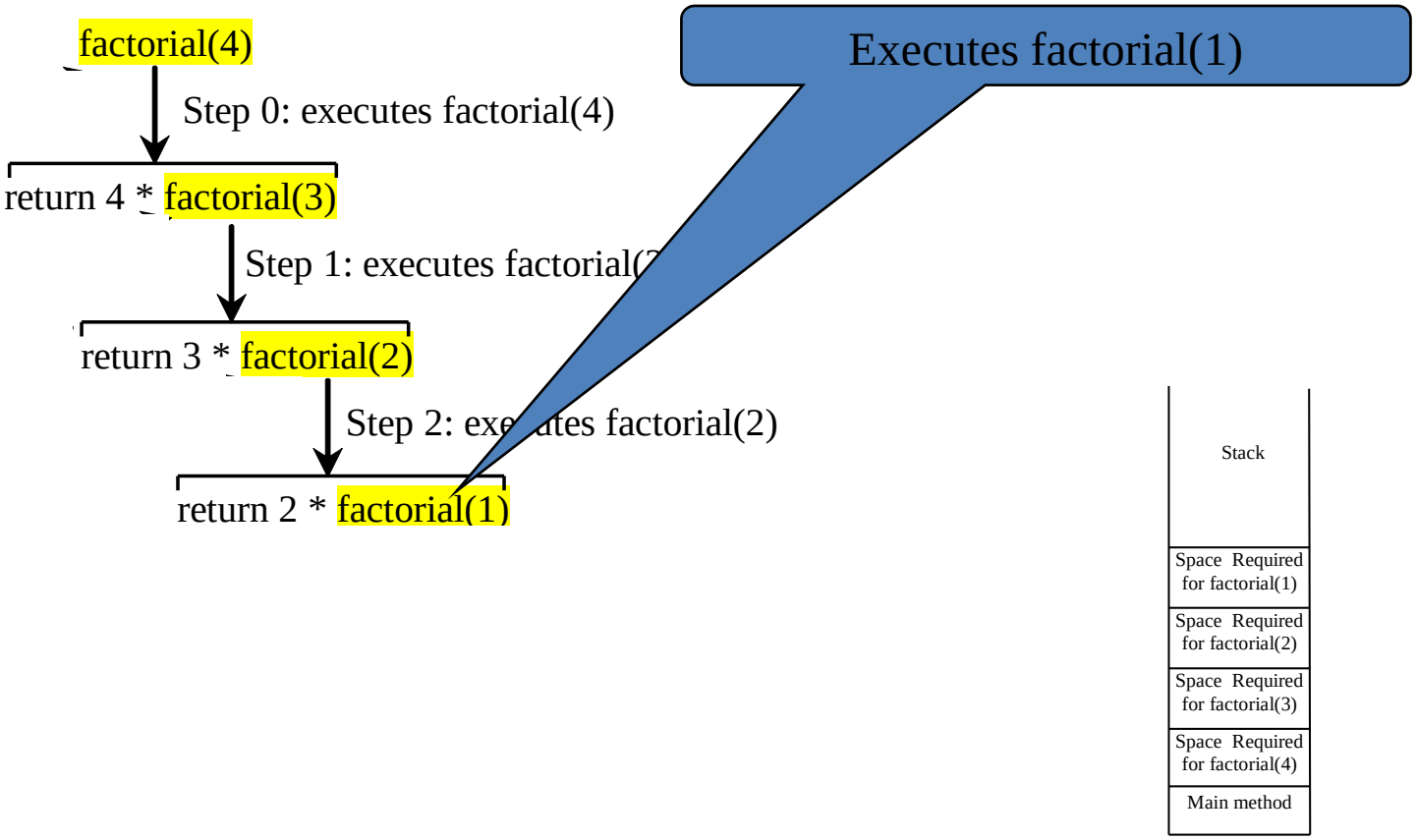

Stack
Space Required for factorial(3)
Space Required for factorial(4)
Main method

# Trace code

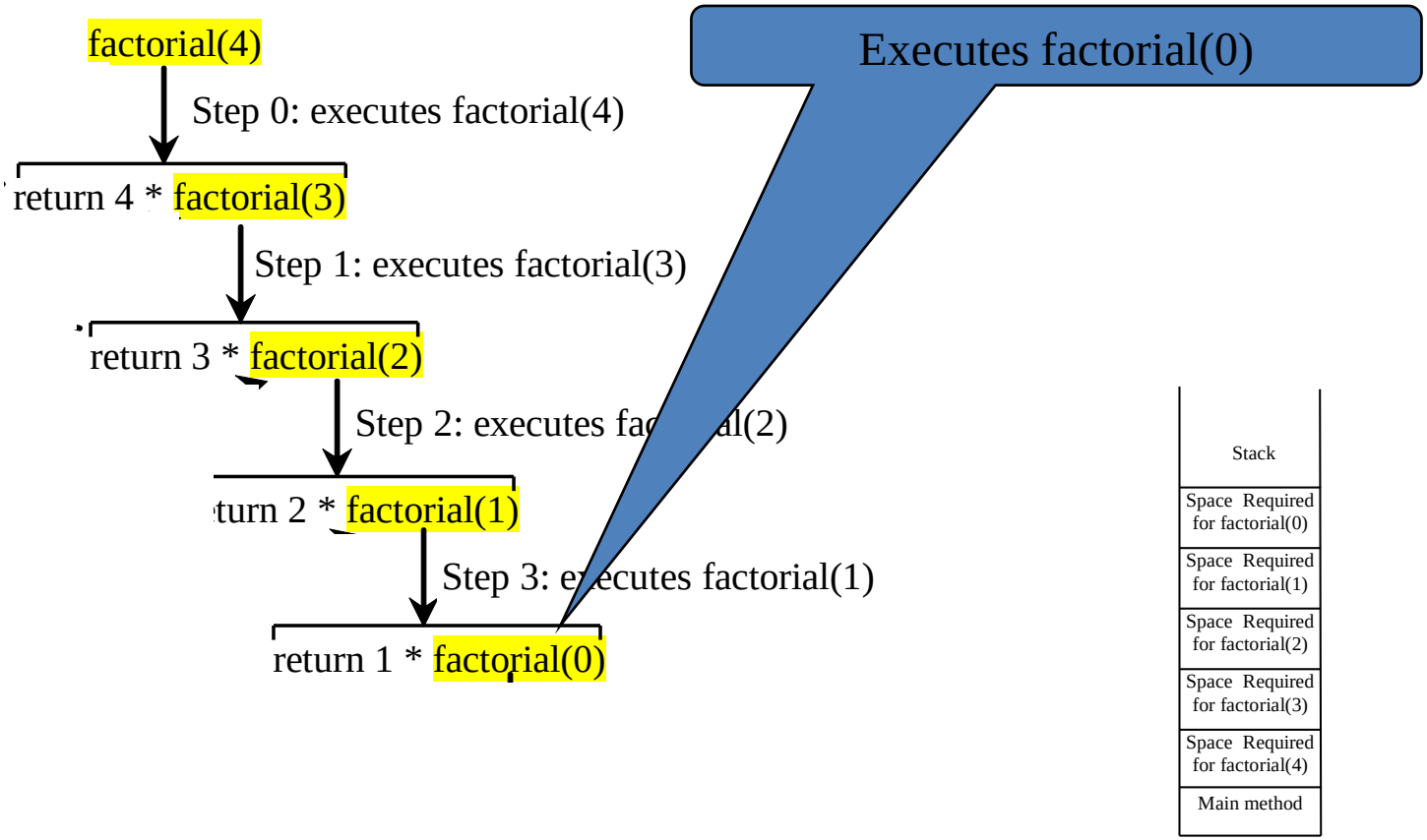

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Executes factorial(2)


Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

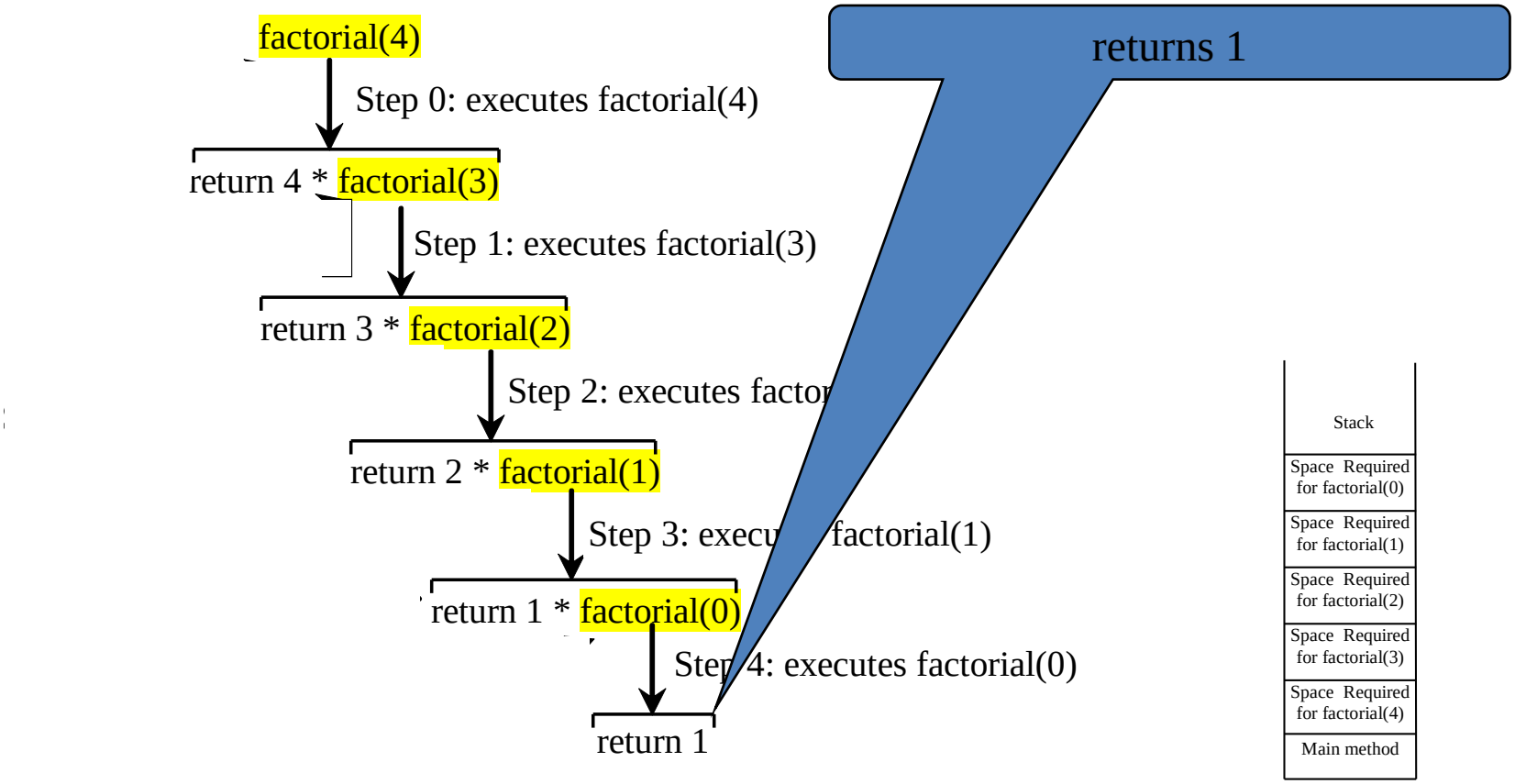
# Trace code


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(
return 3 * factorial(2)
Step 2: exe utes factorial(2)
return 2 * factorial(1)
Executes factorial(1)
Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

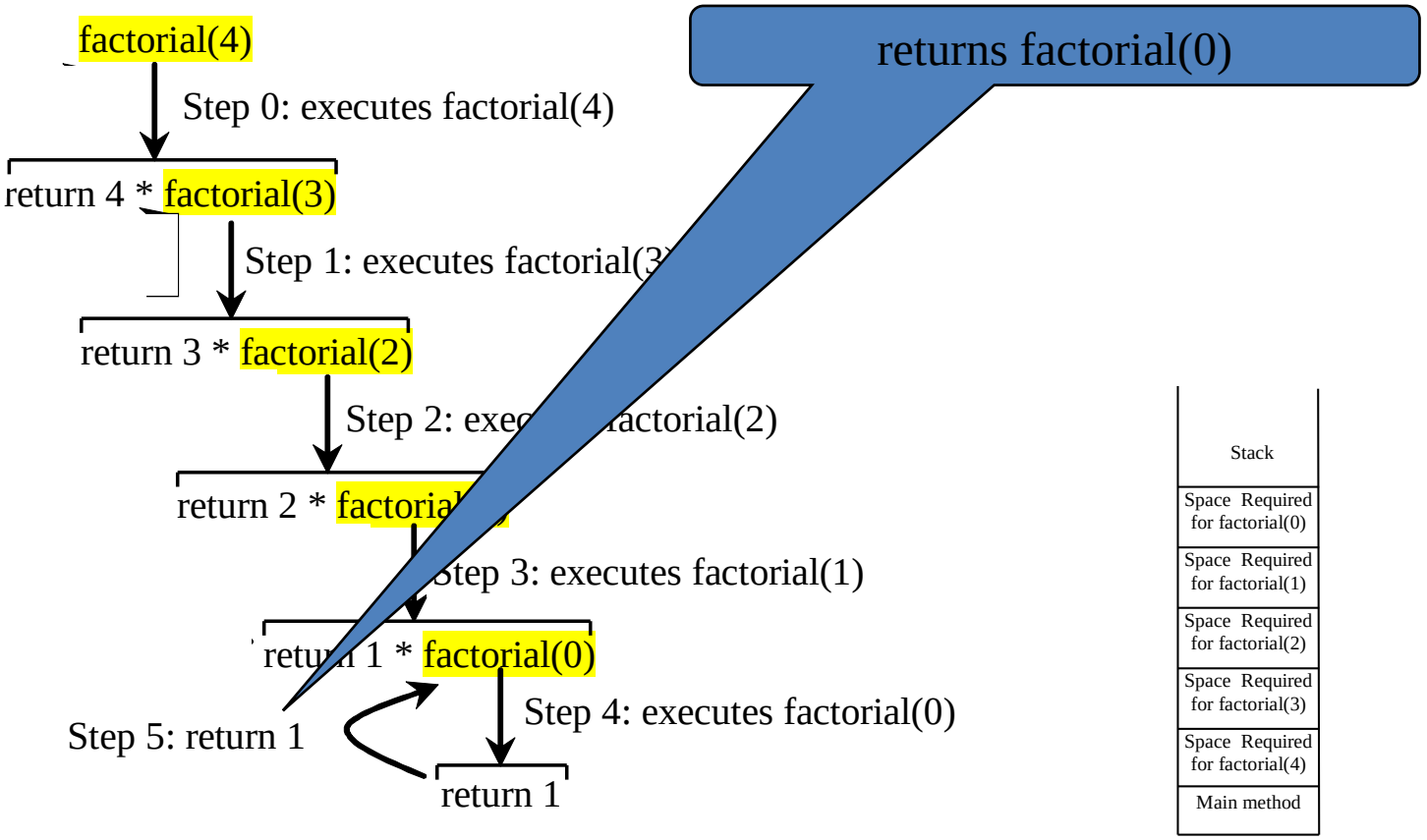
# Trace code


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes fac al(2)
turn 2 * factorial(1)
Step 3: e ecutes factorial(1)
return 1 * factorial(0)
Executes factorial(0)
Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

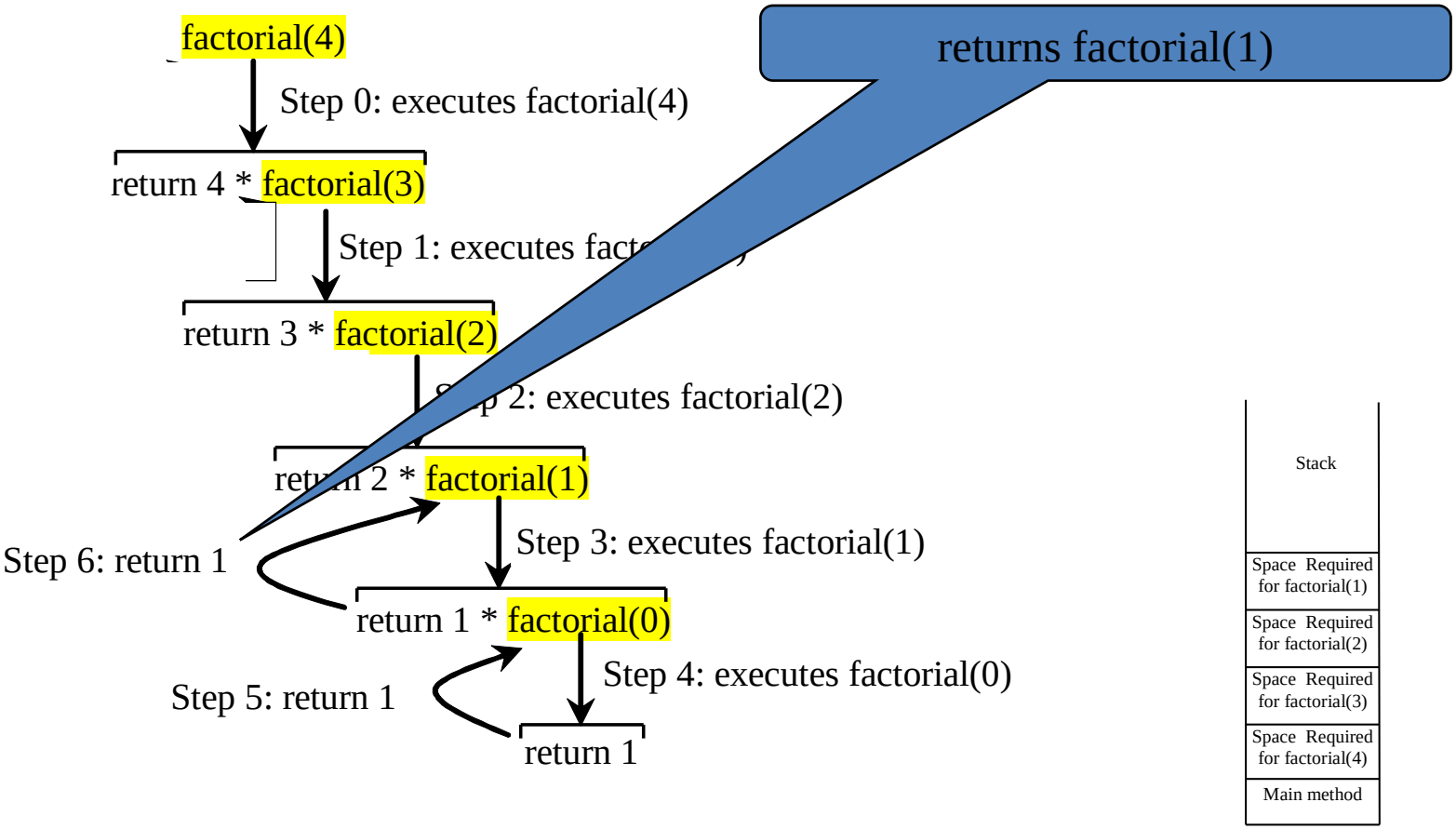
# Trace code


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factor
return 2 * factorial(1)
Step 3: execu factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
returns 1
Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

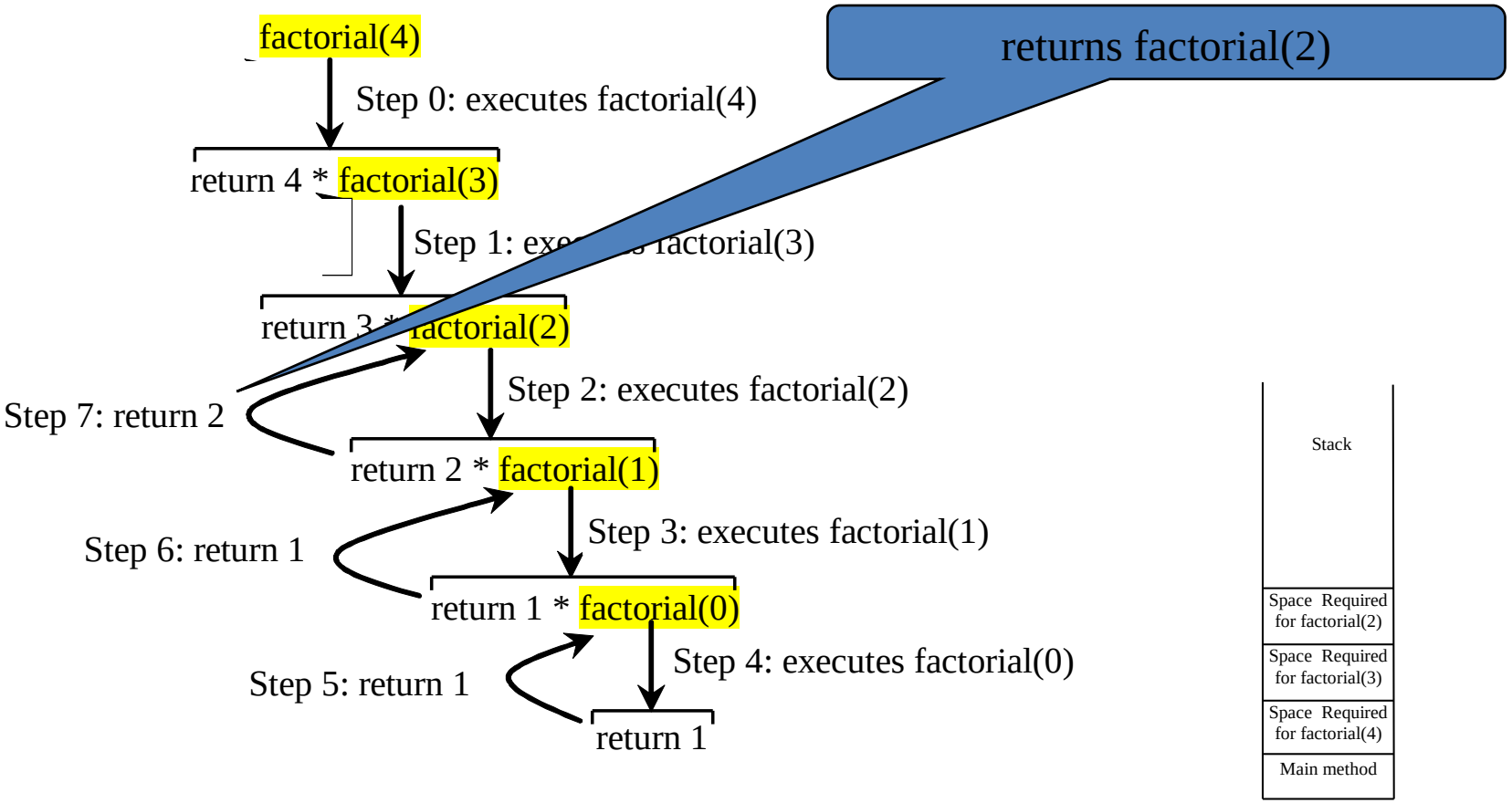
# Trace code


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
returns factorial(0)
Stack
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

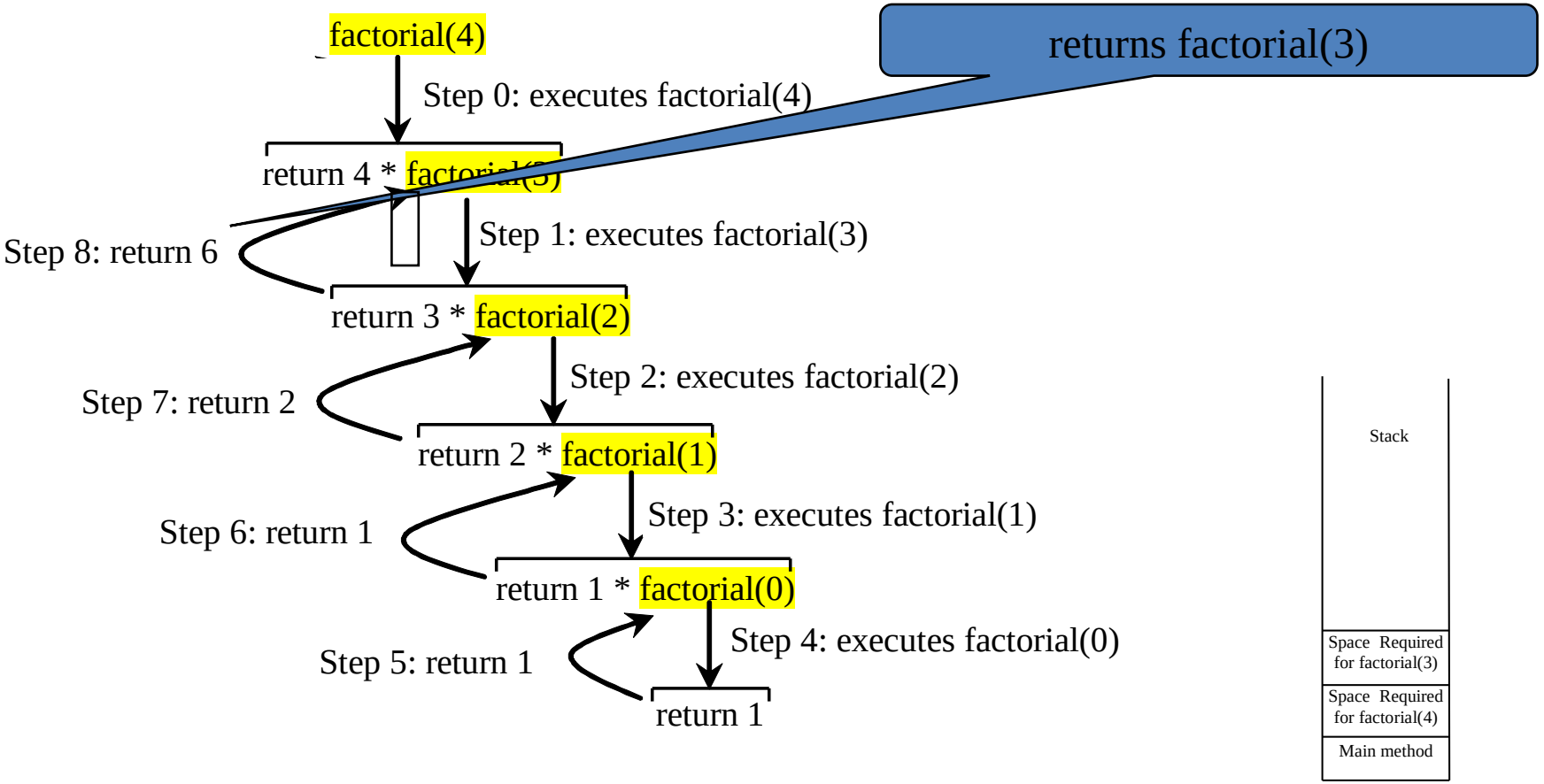
# Trace code


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes fact
return 3 * factorial(2)
2: executes factorial(2)
ret 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
Step 6: return 1
returns factorial(1)
Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

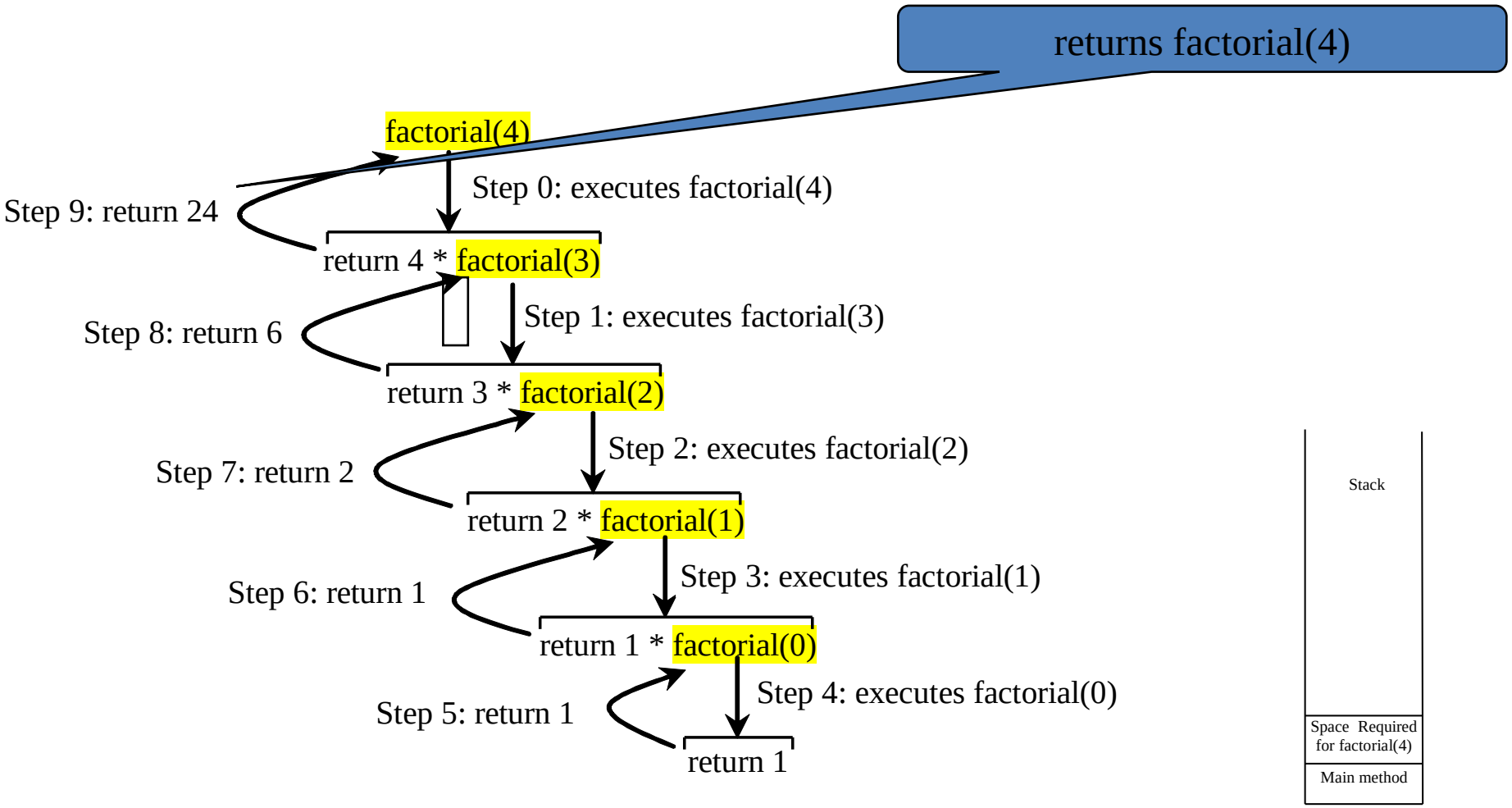
# Trace code


factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
Step 6: return 1
Step 7: return 2
returns factorial(2)
Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

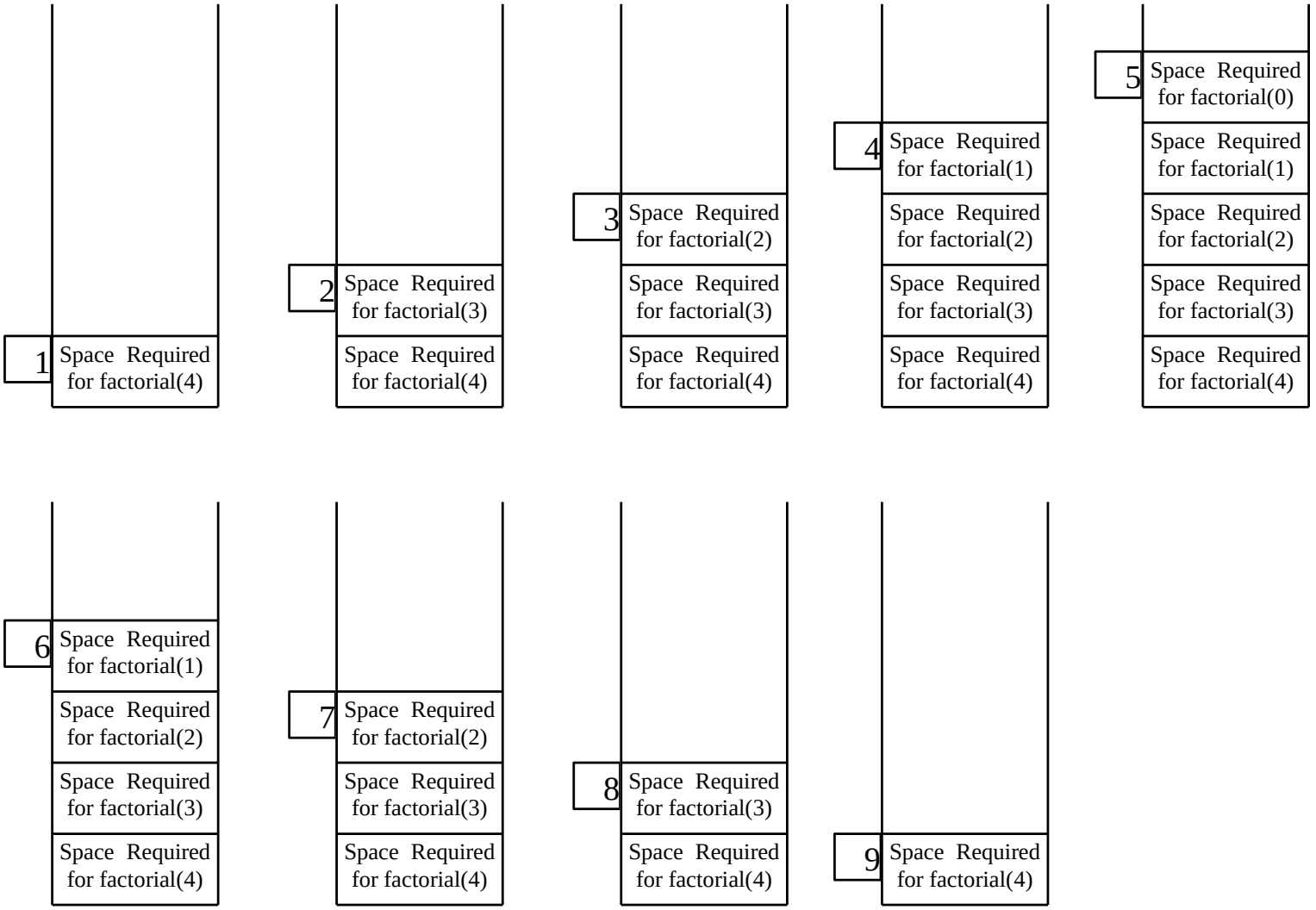
# Trace code


factorial(4)
returns factorial(3)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
Step 8: return 6
return 3 * factorial(2)
Step 2: executes factorial(2)
Step 7: return 2
return 2 * factorial(1)
Step 3: executes factorial(1)
Step 6: return 1
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1
Stack
Space Required for factorial(3)
Space Required for factorial(4)
Main method

# Trace code


returns factorial(4)
factorial(4)
Step 0: executes factorial(4)
Step 9: return 24
return 4 * factorial(3)
Step 1: executes factorial(3)
Step 8: return 6
return 3 * factorial(2)
Step 2: executes factorial(2)
Step 7: return 2
return 2 * factorial(1)
Step 3: executes factorial(1)
Step 6: return 1
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1
Stack
Space Required for factorial(4)
Main method

# Trace stack


1
Space Required for factorial(4)
2
Space Required for factorial(3)
Space Required for factorial(4)
3
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
4
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
5
Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
6
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
7
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
8
Space Required for factorial(3)
Space Required for factorial(4)
9
Space Required for factorial(4)

# Stack overflow

- Deep recursion may result in stack overflow
- If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified, *infinite recursion* can occur
  - Example

```
public static long factorial(int n) {
  // Mistakenly omit base case
  return n * factorial(n – 1);
}
```

    - Results in stack overflow

# Computing factorials

- As a recursive method

```
public static long factorial(int n) {
  if (0 == n) {
    // Base case
    return 1;
  }
  else {
    // Recursive call
    return n * factorial(n – 1);
  }
}
```

A recursive method is one that invokes itself directly or indirectly

Direct recursion

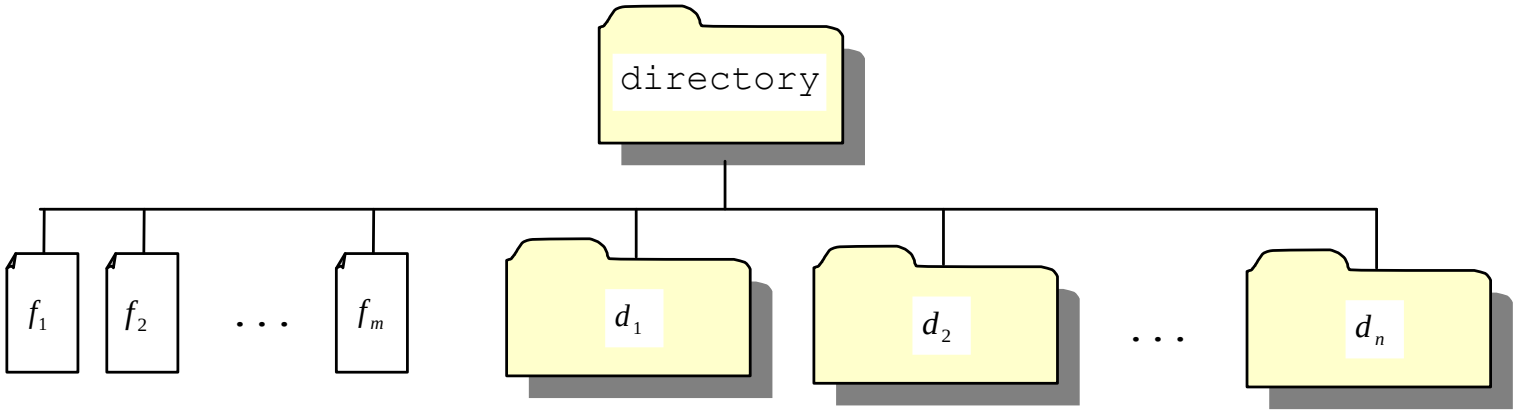
- As a non-recursive method

```
public static long factorial(int n) {
  long nfactorial = 0 == n ? 1 : n;
  for (int i = n - 1; 1 < i; --i) {
    nfactorial *= i;
  }
  return nfactorial;
}
```

Recursive algorithms can be replaced with non-recursive counterparts. However, some problems are inherently recursive, and difficult to solve without using recursion.

# Recursion

- Recursive methods are efficient for solving problems with recursive structures
  - Example problem: find the size of a directory
    - The size of a directory is the sum of the sizes of all files in the directory
    - A directory  may contain subdirectories
    - Suppose a directory contains files and subdirectories



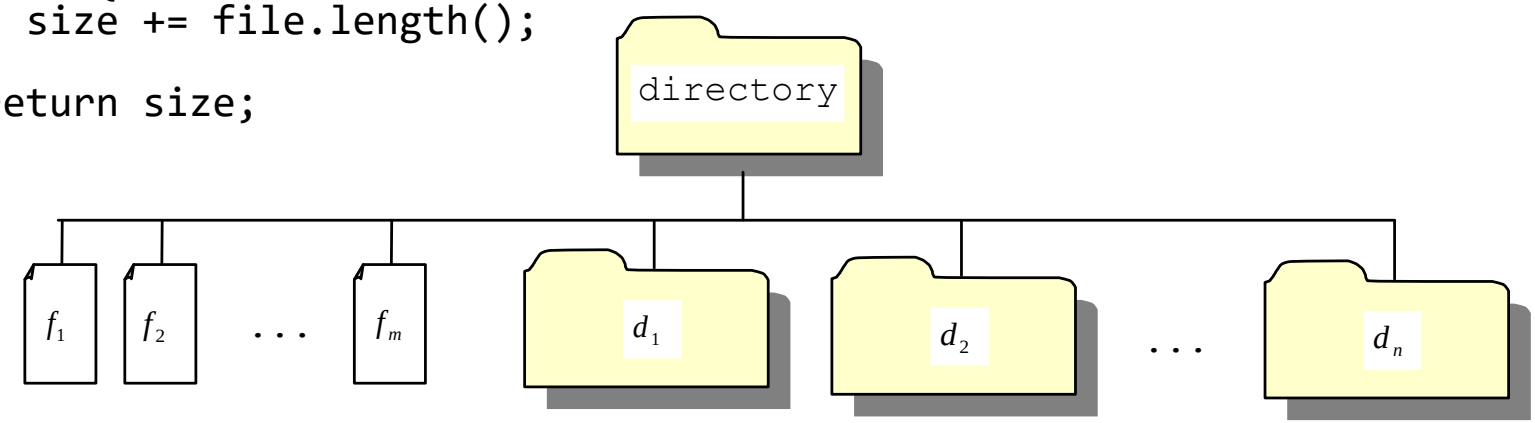
# Finding the directory size

- The size of the directory can be defined recursively as

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

- Recursive method

```
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;
}
```



# Recursion

- In some cases, recursion enables you to create an intuitive, straightforward, simple solution to a problem
  - Example
    - Fibonacci numbers

      f(0) = 0

      f(1) = 1

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

```
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 ...
```

# Fibonacci numbers

- Think recursively

```
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 ...
```

fib(0) = 0;

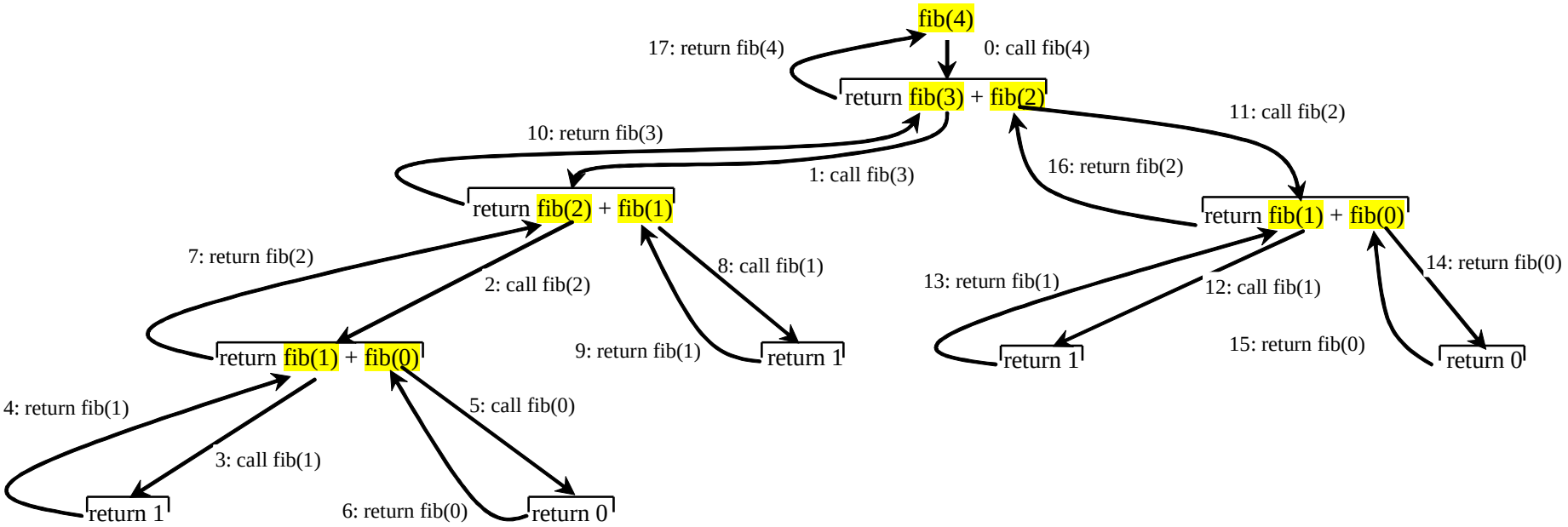
fib(1) = 1;

fib(index) = fib(index -1) + fib(index -2); index >=2

```
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);
}
```

# Fibonacci numbers

- Intuitive, straightforward, and simple, but inefficient
  - Many duplicate calls



# Characteristics of recursion

- All recursive methods have the following characteristics
  - The method is implemented using an if-else (or a switch) statement that leads to **different cases**
  - One or more **base cases** (the simplest case) are used to stop recursion
  - Every recursive call **reduces** the original problem, bringing it increasingly closer to a base case until it becomes that case
- In general, to solve a problem using recursion, you break it into subproblems
  - If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively
  - This subproblem is almost the same as the original problem in nature with a smaller size

# Recursive helper methods

- Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem
- This new method is called a recursive helper method
- The original problem can be solved by invoking the recursive helper method

# The palindrome problem

- A string is a palindrome if it reads the same from the left and from the right
  - Subproblems
    - Check whether the first character and the last character of the string are equal
    - Ignore the two end characters and check whether the rest of the substring is a palindrome
      - Same as the original problem, but smaller in size (i.e., reduction)
  - Base cases
    - The two end characters are not the same
    - The string size is 0 or 1

# The palindrome problem

- Without a recursive helper method

```
public static boolean isPalindrome(String s) {
  if (s.length() <= 1) {
    // Base case
    return true;
  }
  else if (s.charAt(0) != s.charAt(s.length() - 1)) {
    // Base case
    return false;
  }
  else {
    // Reduction and recursive call
    return isPalindrome(s.substring(1, s.length() - 1));
  }
}
```

  - Inefficiently creates a new string for every recursive call

# The palindrome problem

- With a recursive helper method

```
public static boolean isPalindrome(String s) {
  return isPalindrome(s, 0, s.length() - 1);
}

public static boolean isPalindrome(String s, int low, int high) {
  if (high <= low) // Base case
    return true;
  else if (s.charAt(low) != s.charAt(high)) // Base case
    return false;
  else
    return isPalindrome(s, low + 1, high - 1);
}
```

# Tail recursion

- A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call
- A tail-recursive method is more efficient than a nontail-recursive method
  - Compilers can optimize to reduce stack size

# Computing factorials

- Without tail recursion

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

Pending operation

- With tail recursion

```
public static long factorial(int n) {
  return factorial(n, 1); // Call auxiliary method
}

// Auxiliary tail-recursive method for factorial
private static long factorial(int n, int result) {
  if (n == 0)
    return result;
  else
    return factorial(n - 1, n * result); // Recursive call
}
```

# Recursion vs. iteration

- Recursion is an alternative form of program control
- It is essentially repetition without a loop
- Recursion bears substantial overhead
  - Each time the program calls a method, the system must assign space for all of the method's local variables and parameters
  - This can consume considerable memory and requires extra time to manage the additional space

# Recursion vs. iteration

- Recursive algorithms can be replaced with non-recursive counterparts
  - If performance is a concern, then avoid using recursion
  - However, some problems are inherently recursive, and difficult to solve without using recursion
- Use whichever approach can best develop an intuitive solution that naturally mirrors the problem
  - If an iterative solution is obvious, then use it

# Next Lecture

- Event-driven programming
- Reading
  - Liang
    - Chapter 15