Chapter 18 Recursion

Computing Factorial

Mathematic notation:

$n! = n * (n-1)!, n > 0$

$0! = 1$

Function:

factorial(0) = 1;

factorial(n) = n*factorial(n-1); n > 0

ComputeFactorial

animation

Computing Factorial

factorial(0) = 1;

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

factorial(4)

animation

BIRZEIT UNIVERSITY

Computing Factorial

factorial(0) = 1;

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

factorial(4) = 4 * factorial(3)

BIRZEIT UNIVERSITY

Computing Factorial

factorial(0) = 1;

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

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

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

BIRZEIT UNIVERSITY

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)))

animation

BIRZEIT UNIVERSITY

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

BIRZEIT UNIVERSITY

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

BIRZEIT UNIVERSITY

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

BIRZEIT UNIVERSITY

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

BIRZEIT UNIVERSITY

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)

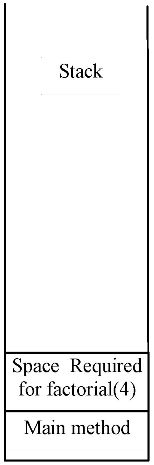
= 24

animation

Trace Recursive factorial

Executes factorial(4)

factorial(4)



animation

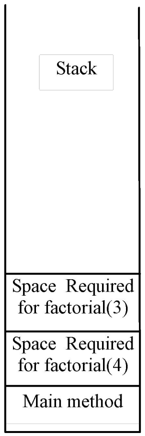
Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

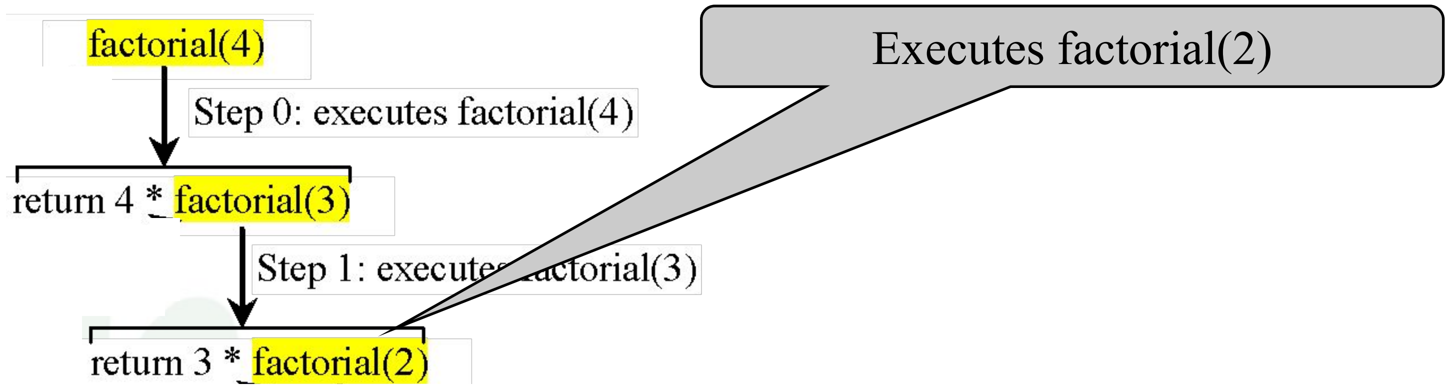
return 4 * factorial(3)

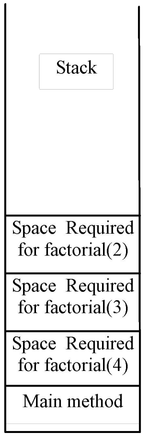
Executes factorial(3)



BIRZEIT UNIVERSITY

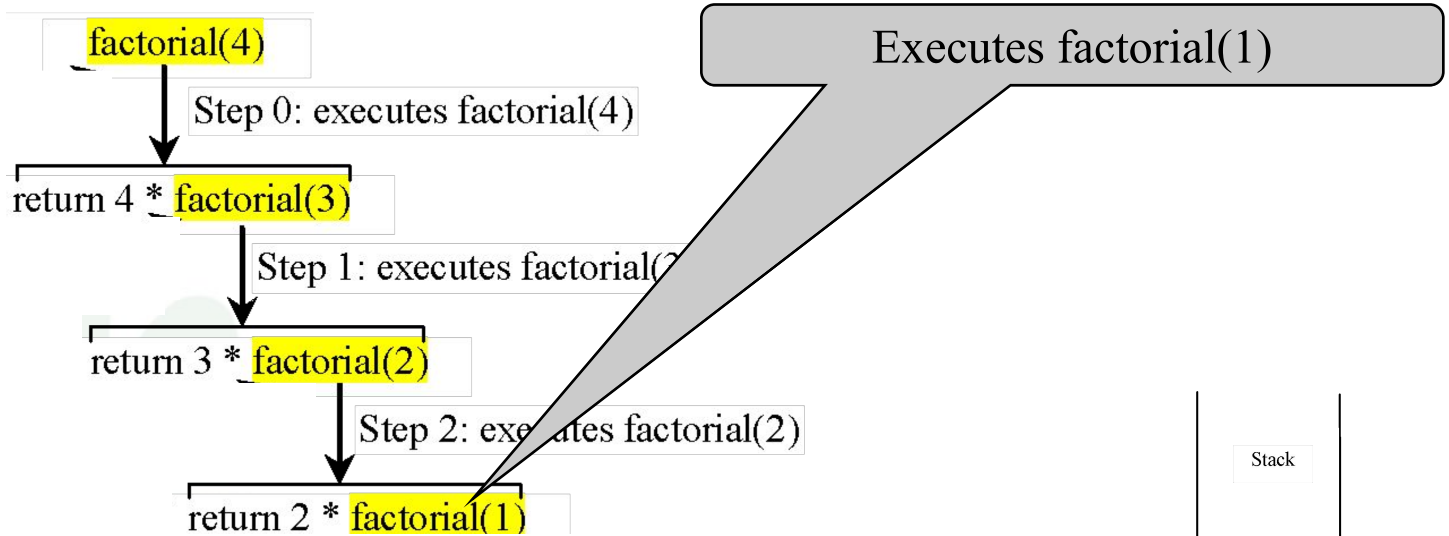
Trace Recursive factorial

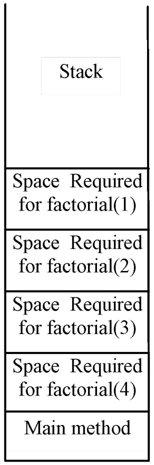




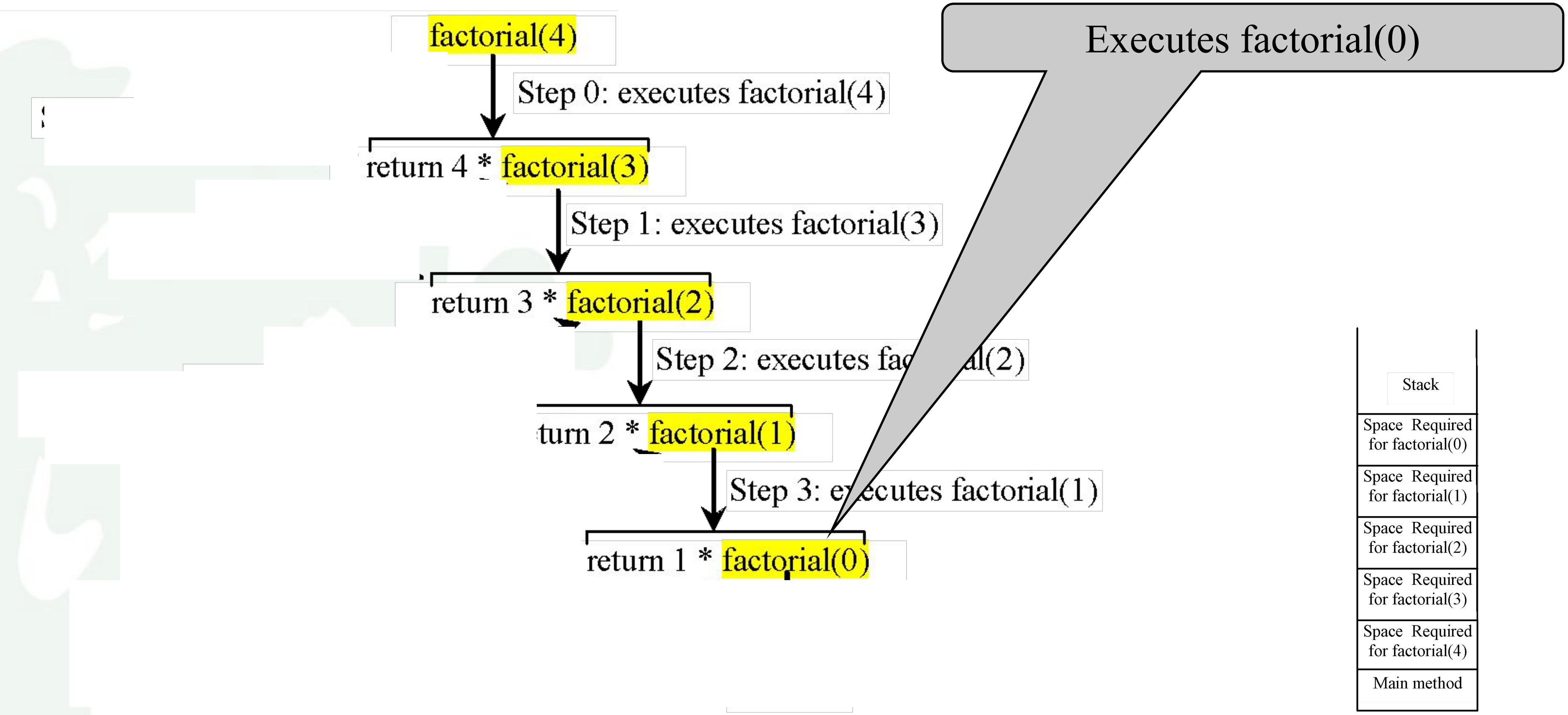
animation

Trace Recursive factorial



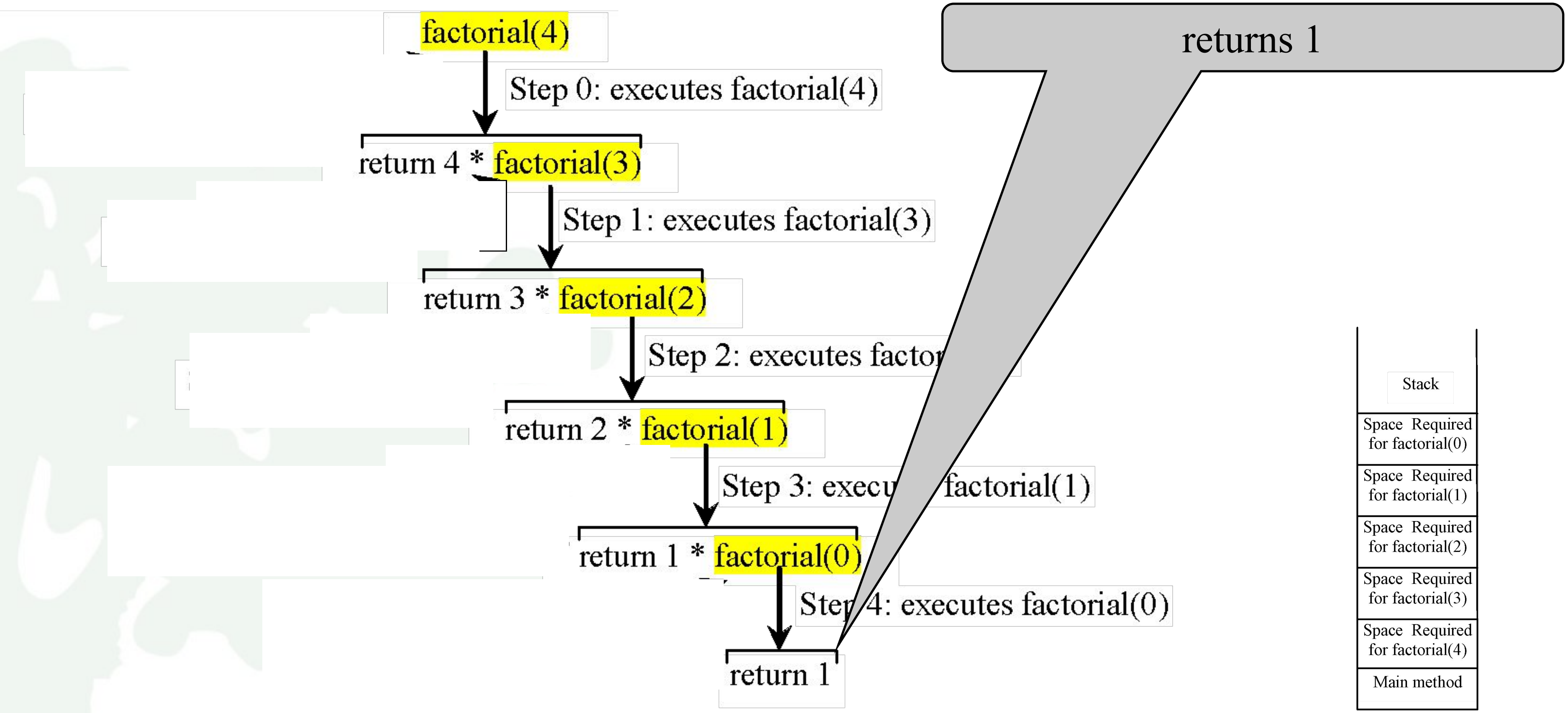


Trace Recursive factorial



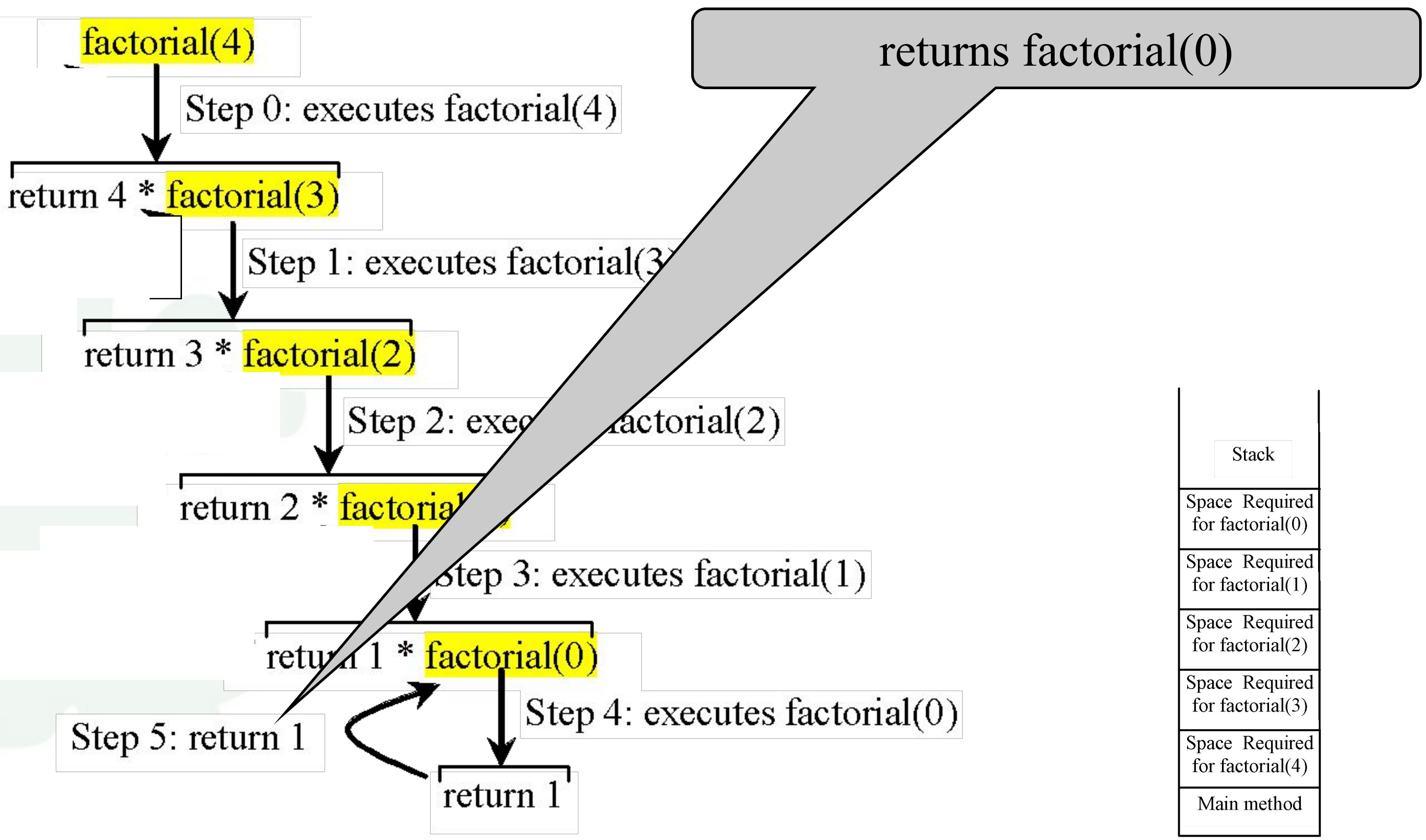
animation

Trace Recursive factorial



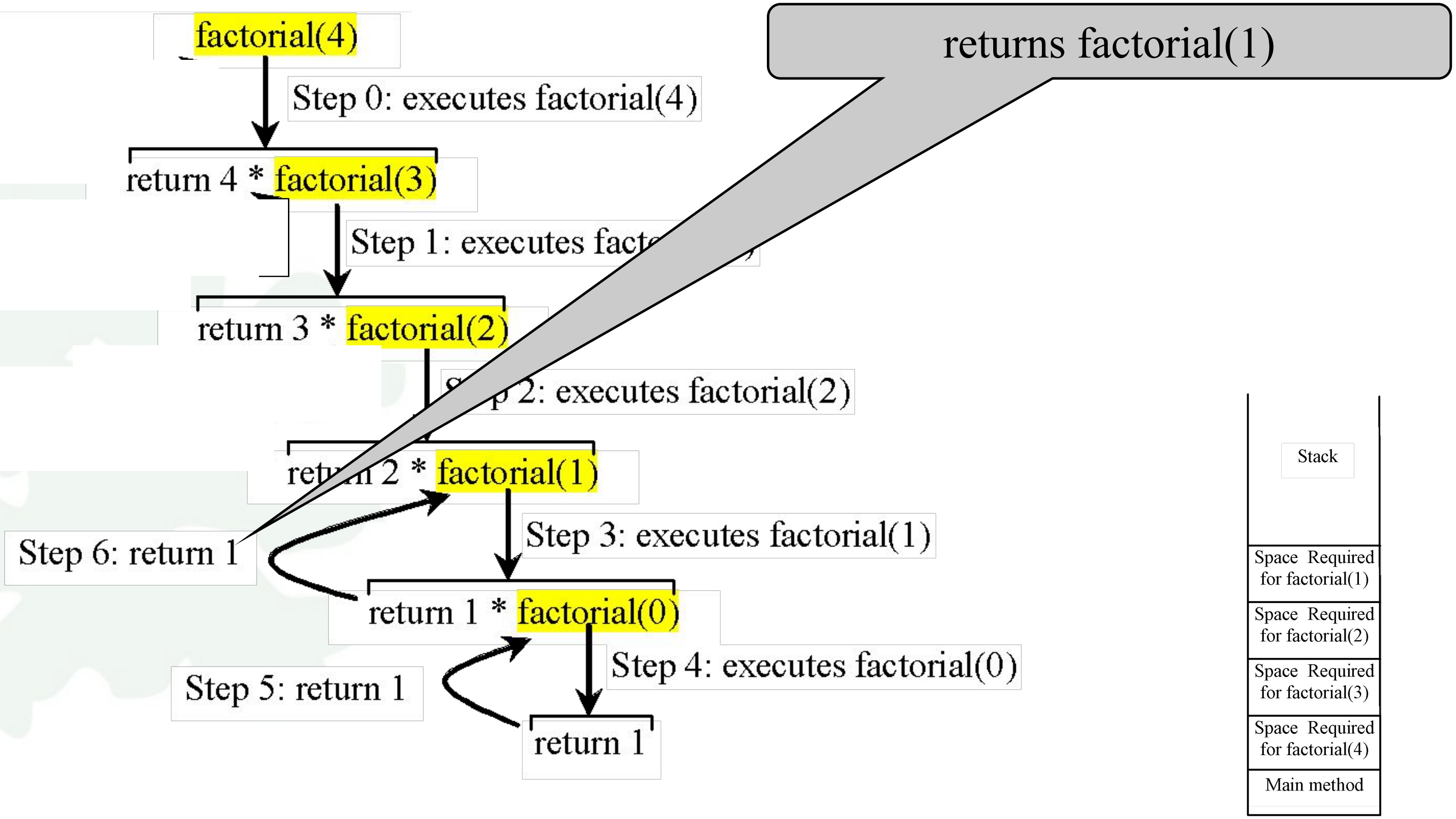
animation

Trace Recursive factorial

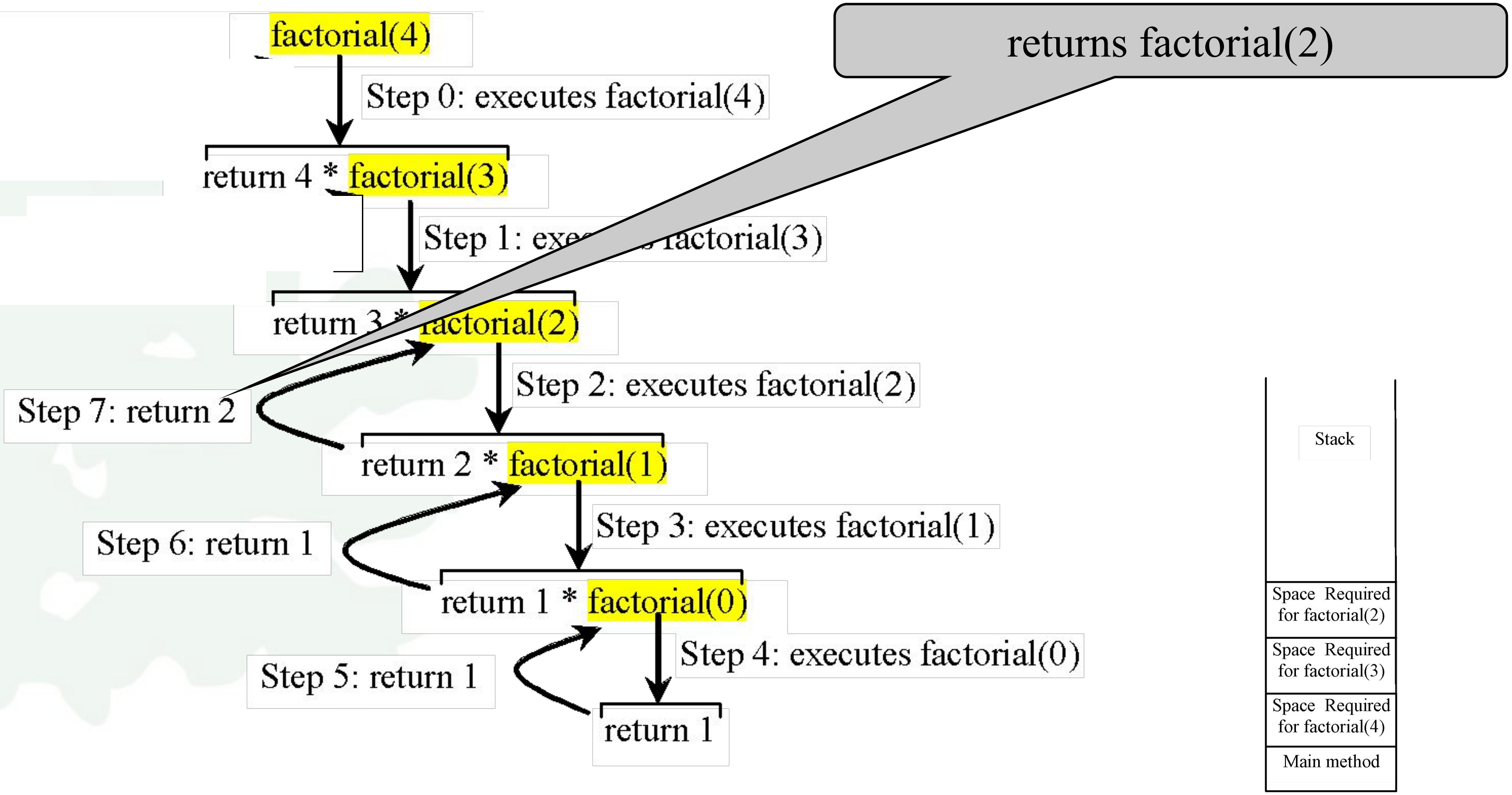


animation

Trace Recursive factorial

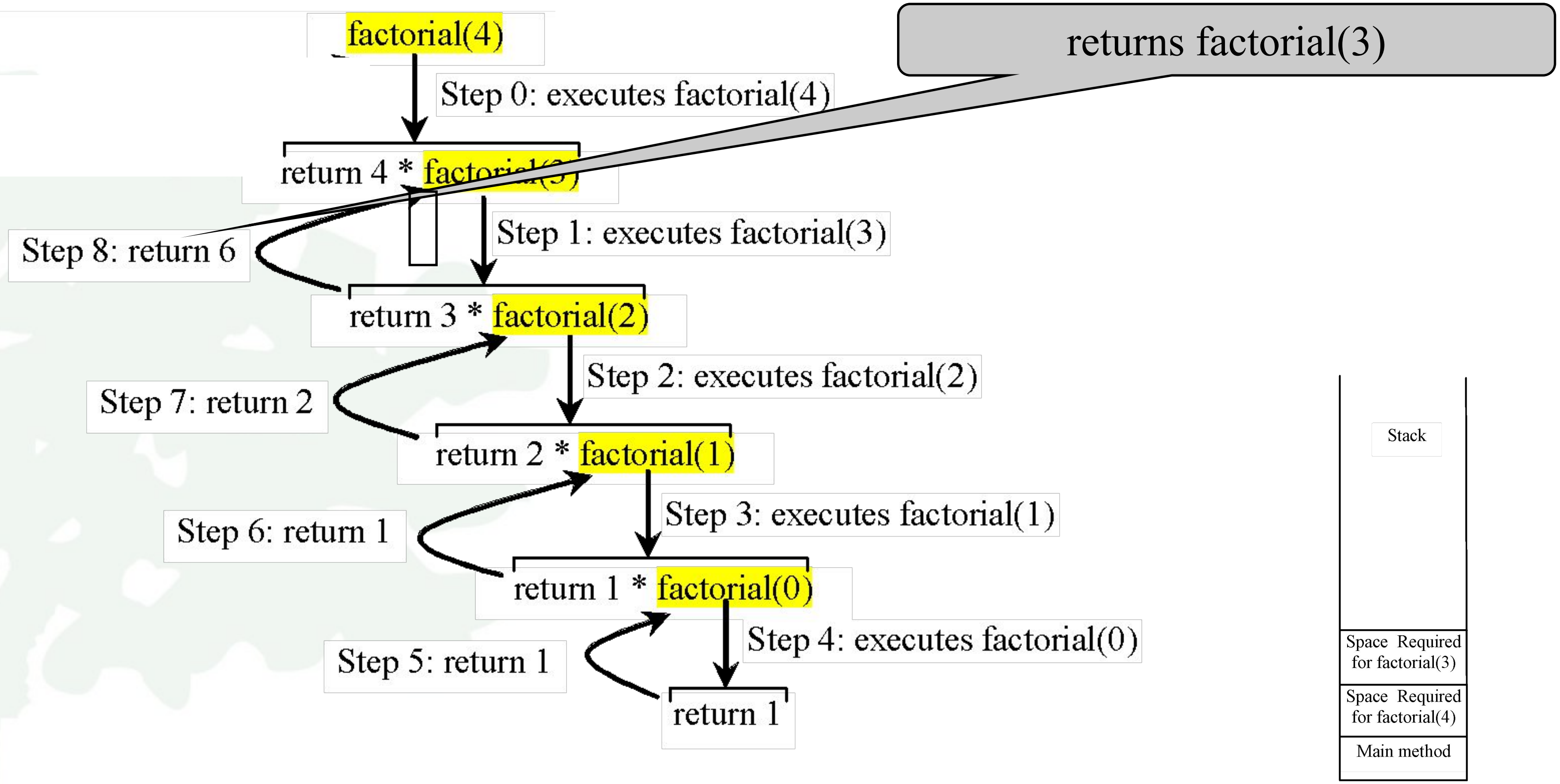


Trace Recursive factorial



animation

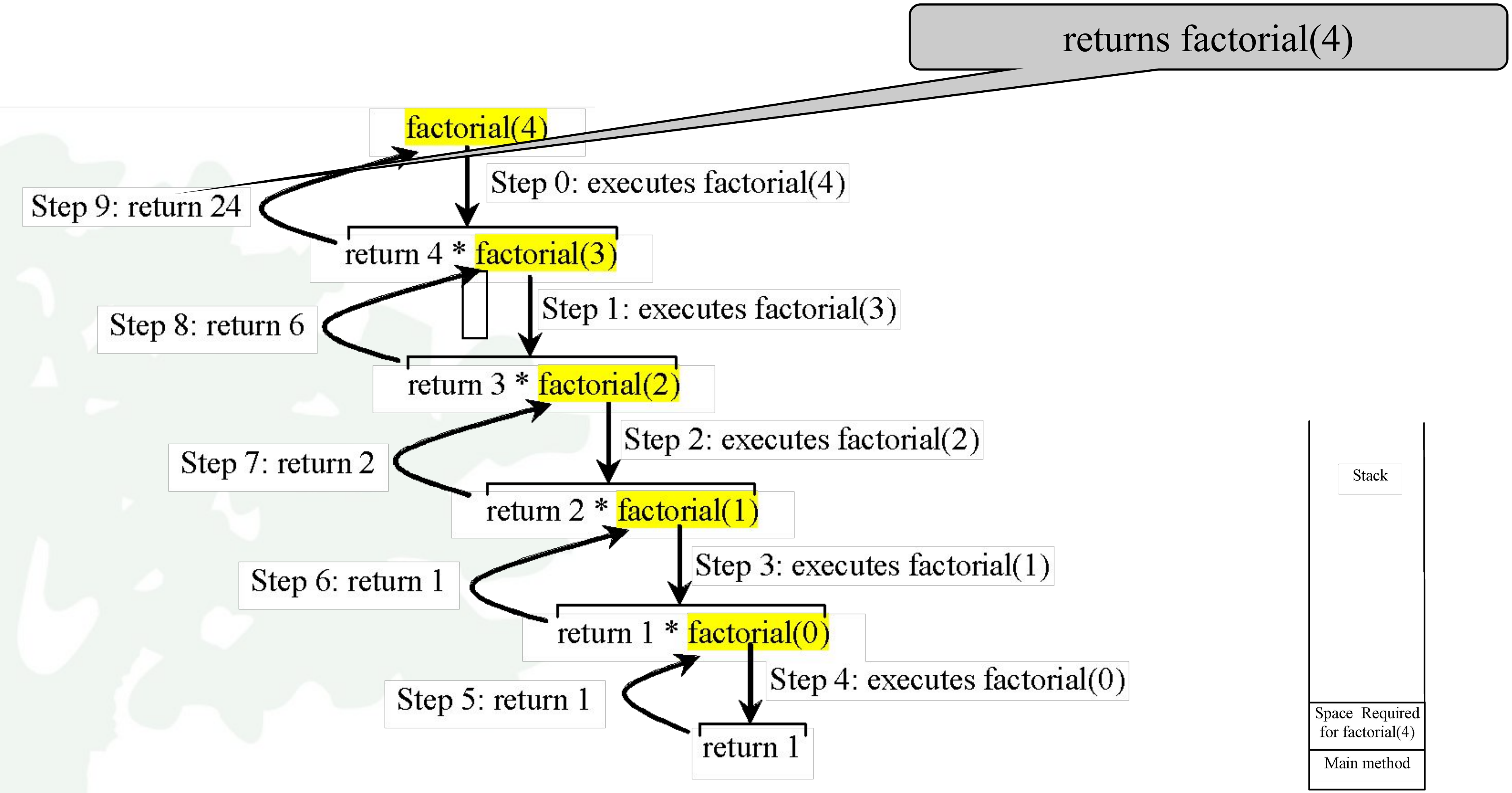
Trace Recursive factorial



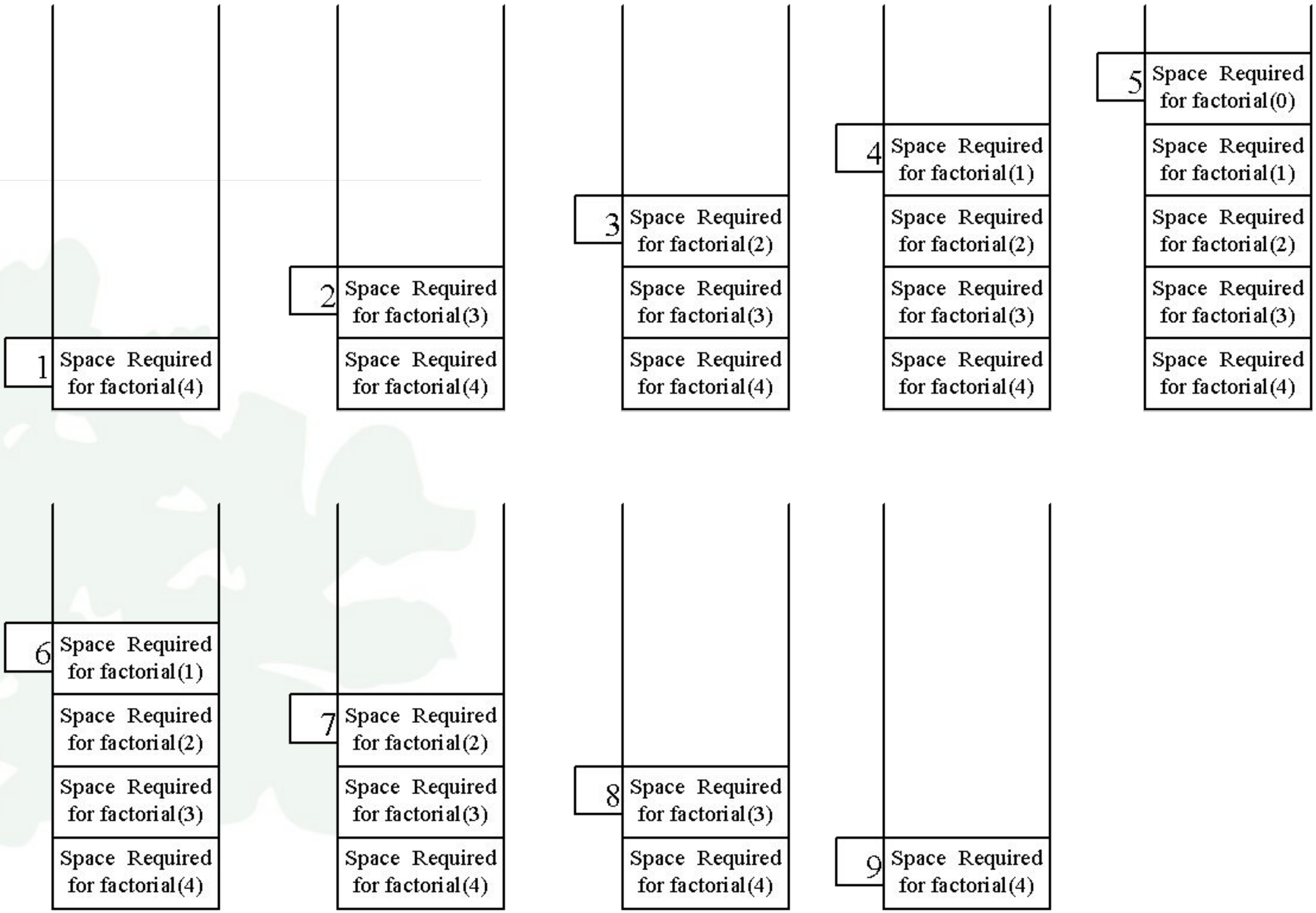
animation

BIRZEIT UNIVERSITY

Trace Recursive factorial



factorial(4) Stack Trace



Other Examples

$f(0) = 0;$

$f(n) = n + f(n-1);$

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
```

fib(0) = 0;

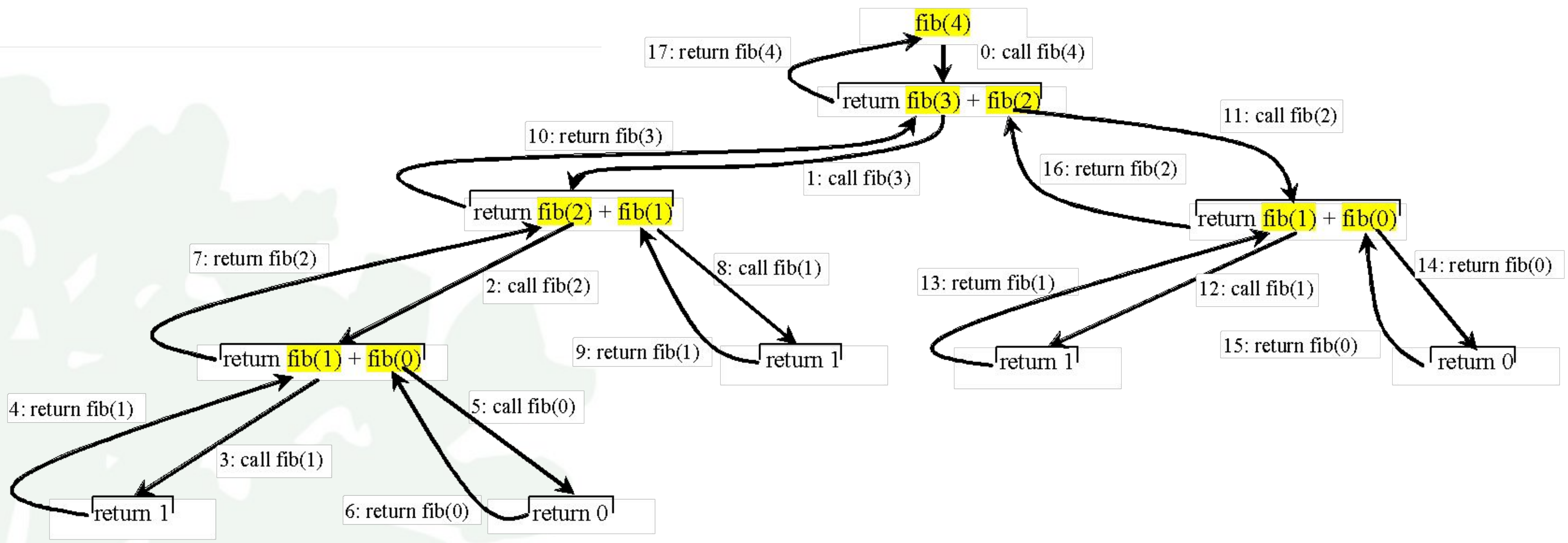
fib(1) = 1;

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

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

ComputeFibonacci

Fibonnaci Numbers, cont.



Problem Solving Using Recursion

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 subproblems recursively. A subproblem is almost the same as the original problem in nature with a smaller size.

Characteristics of Recursion

All recursive methods have the following characteristics:

- The method is implemented using a conditional 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.

Problem Solving Using Recursion

nPrintln("Welcome", n);

1. one is to print the message one time and the other is to print the message for n-1 times.
2. The second problem is the same as the original problem with a smaller size.
3. The base case for the problem is n==0. You can solve this problem using recursion as follows:

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

Think Recursively

Many of the problems presented in the early chapters can be solved using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```
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
    return isPalindrome(s.substring(1, s.length() - 1));
}
```

RecursivePalindromeUsingSubstring

Recursive Helper Methods

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

Recursive Helper Methods

The preceding recursive isPalindrome method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a 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);
}
```

RecursivePalindrome

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.

Advantages of Using Recursion

Recursion is good for solving the problems that are inherently recursive.