Chapter 18
Recursion

Why Use Recursion

- Recursion allows solving complex problem with simple solutions.
- Reduces coding.
- Leads to efficient programs.

Example: Computing Factorial

n! = n * n-1 * n-2 * n-3 … * 1

n! = n * (n-1)!

factorial(0) = 1; //special cases

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

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

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)))

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)

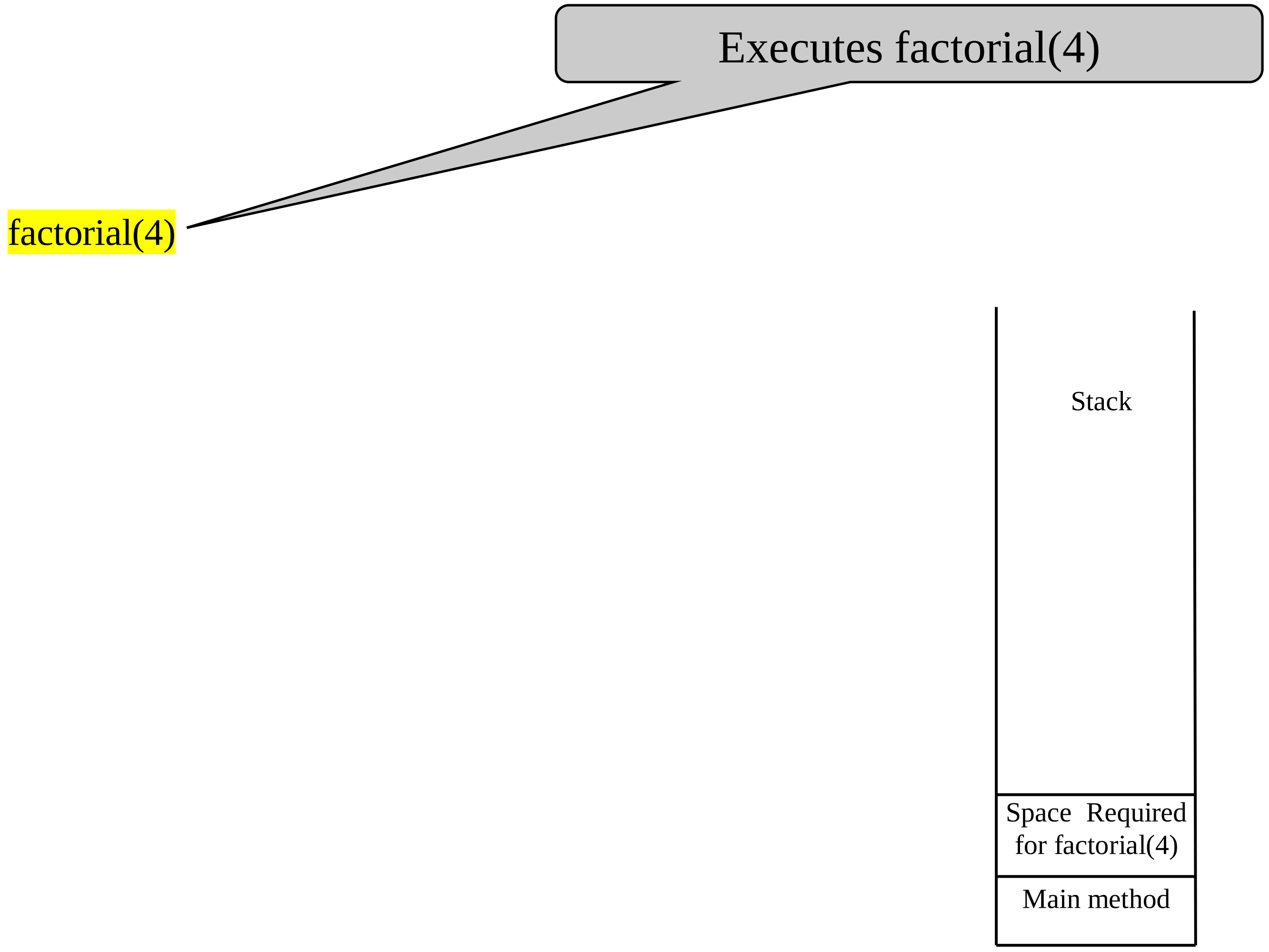
= 4 * 3 * 2

= 4 * 6

= 24

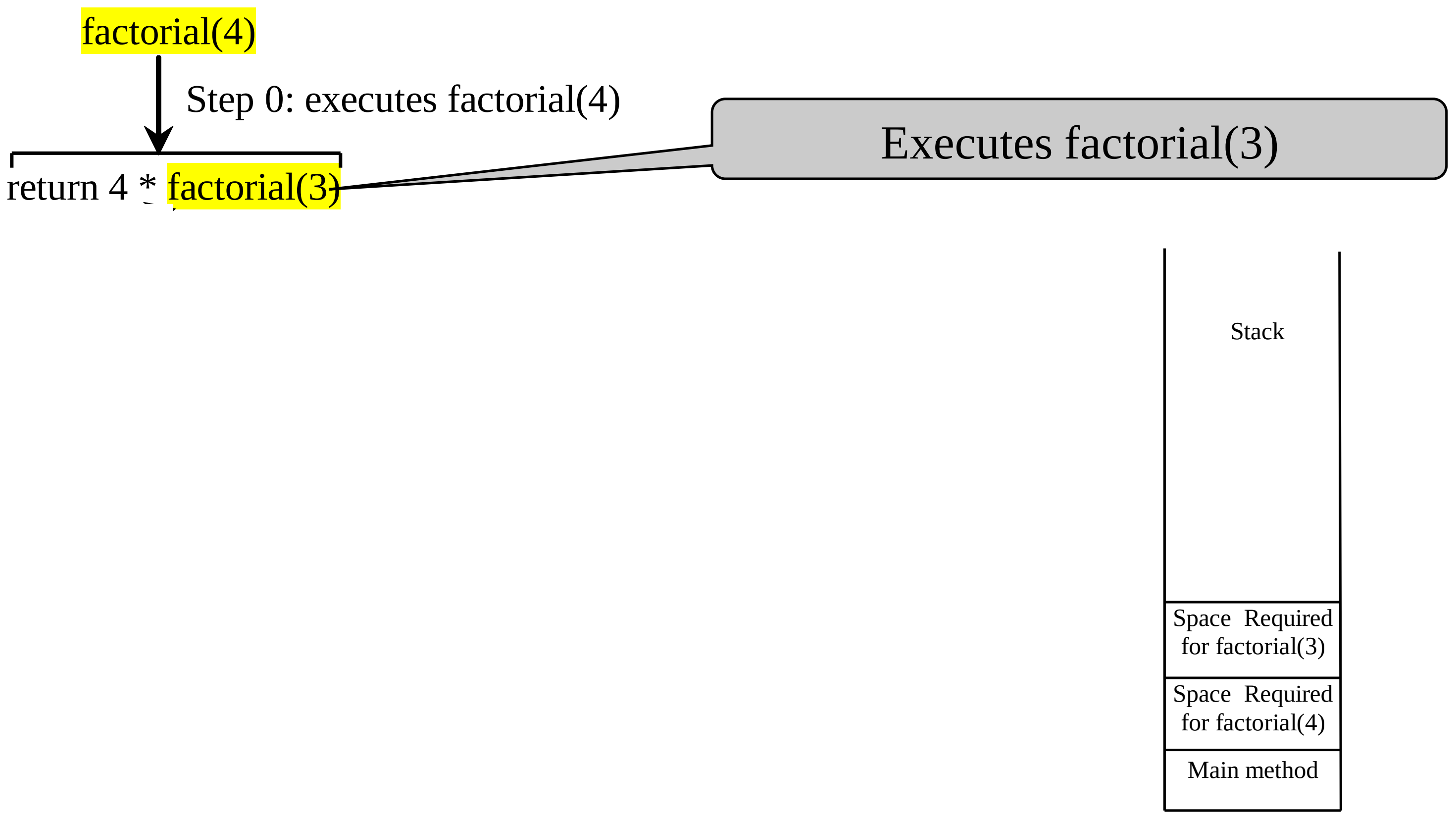
animation

Trace Recursive factorial



animation

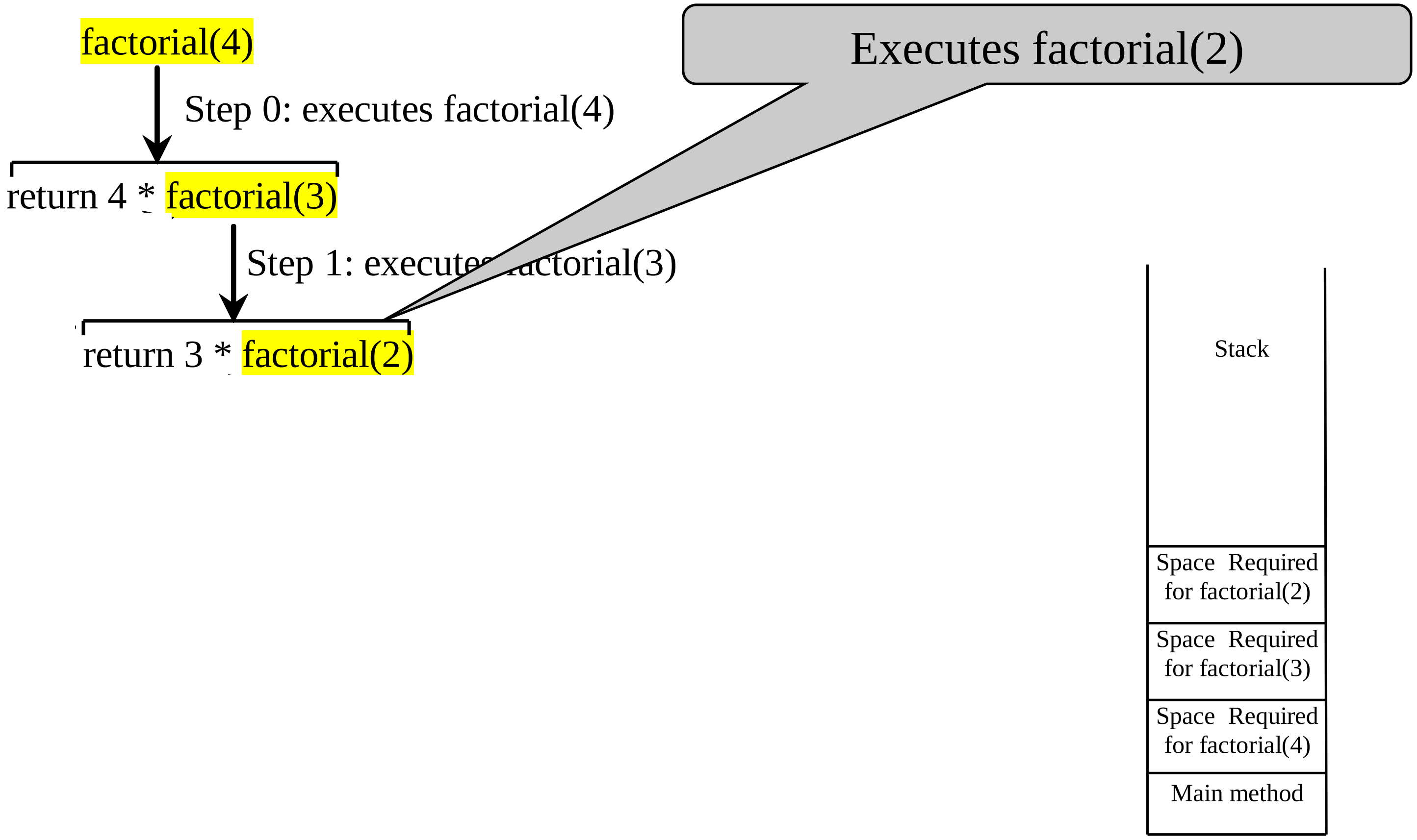
Trace Recursive factorial



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

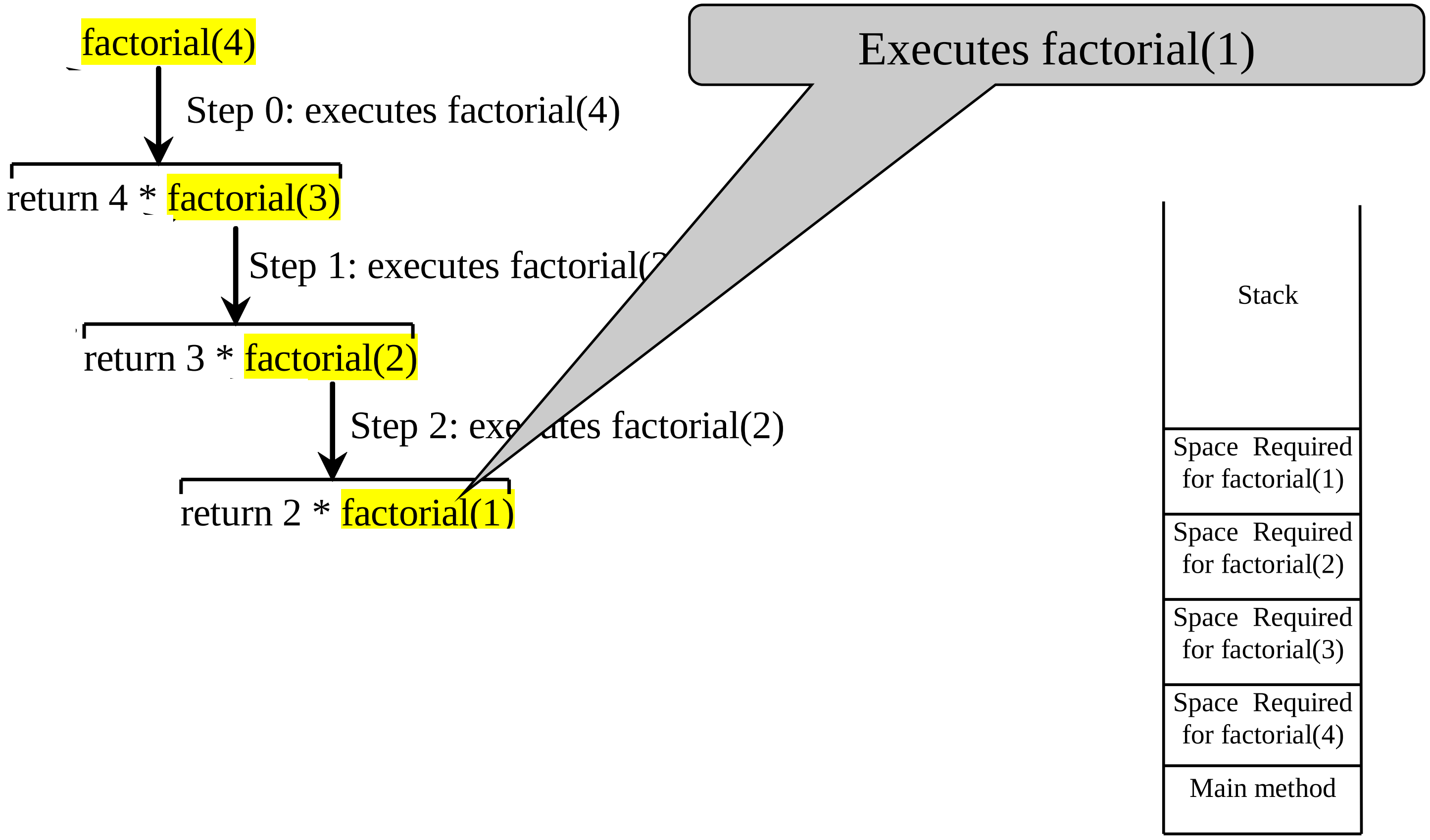
animation

Trace Recursive factorial


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

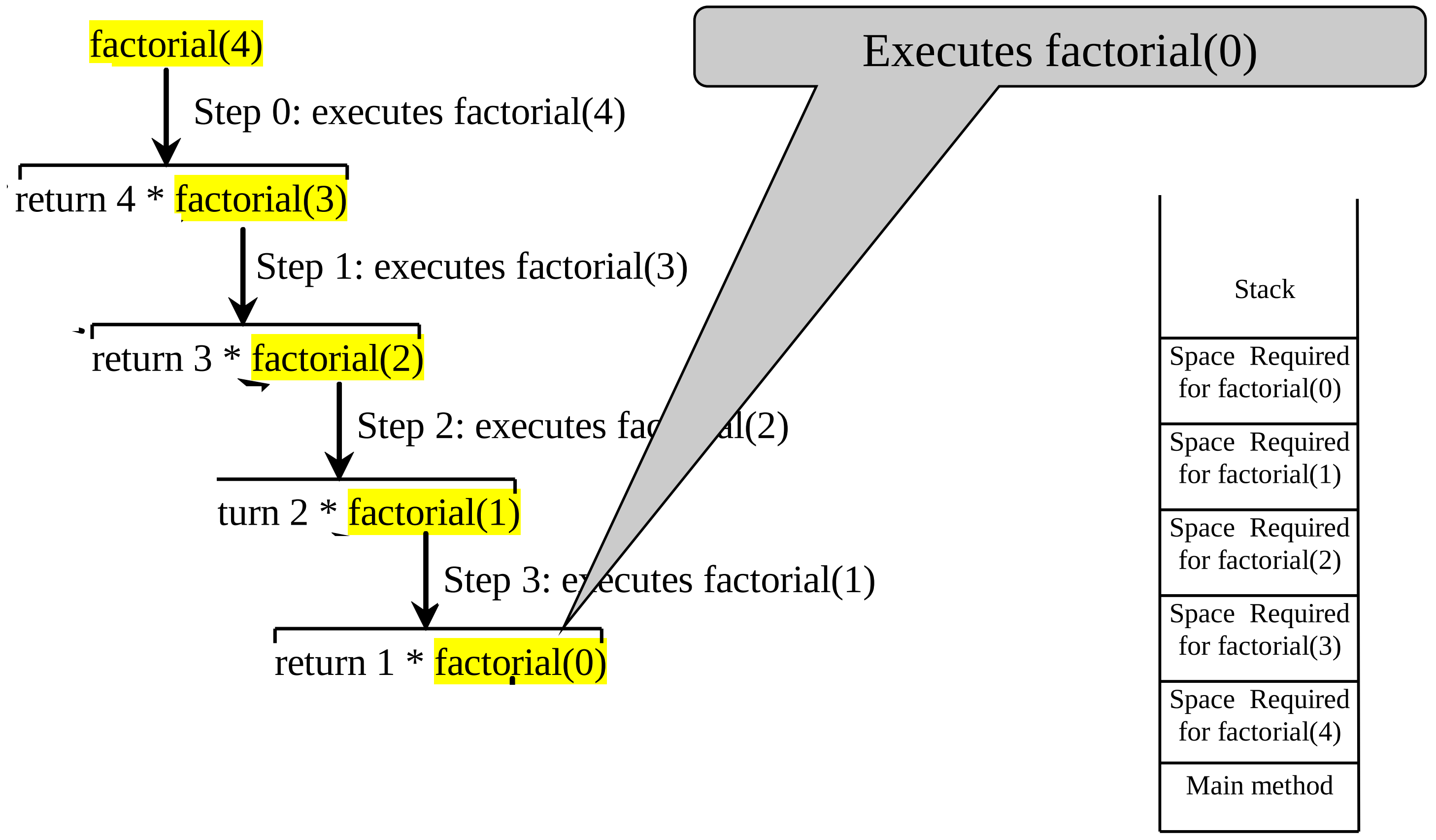
animation

Trace Recursive factorial


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)
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

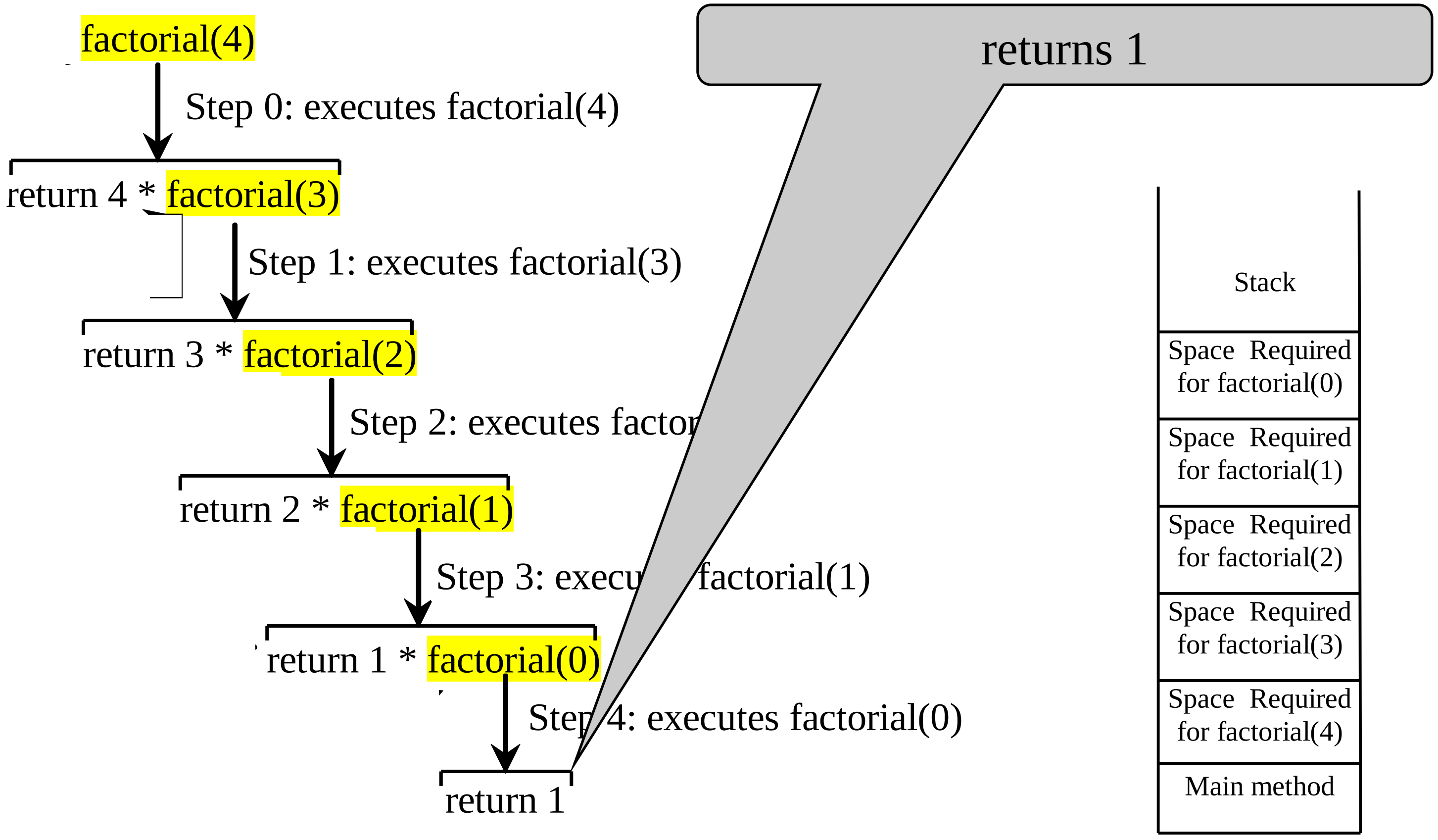
animation

Trace Recursive factorial


Executes factorial(0)
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)
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

animation

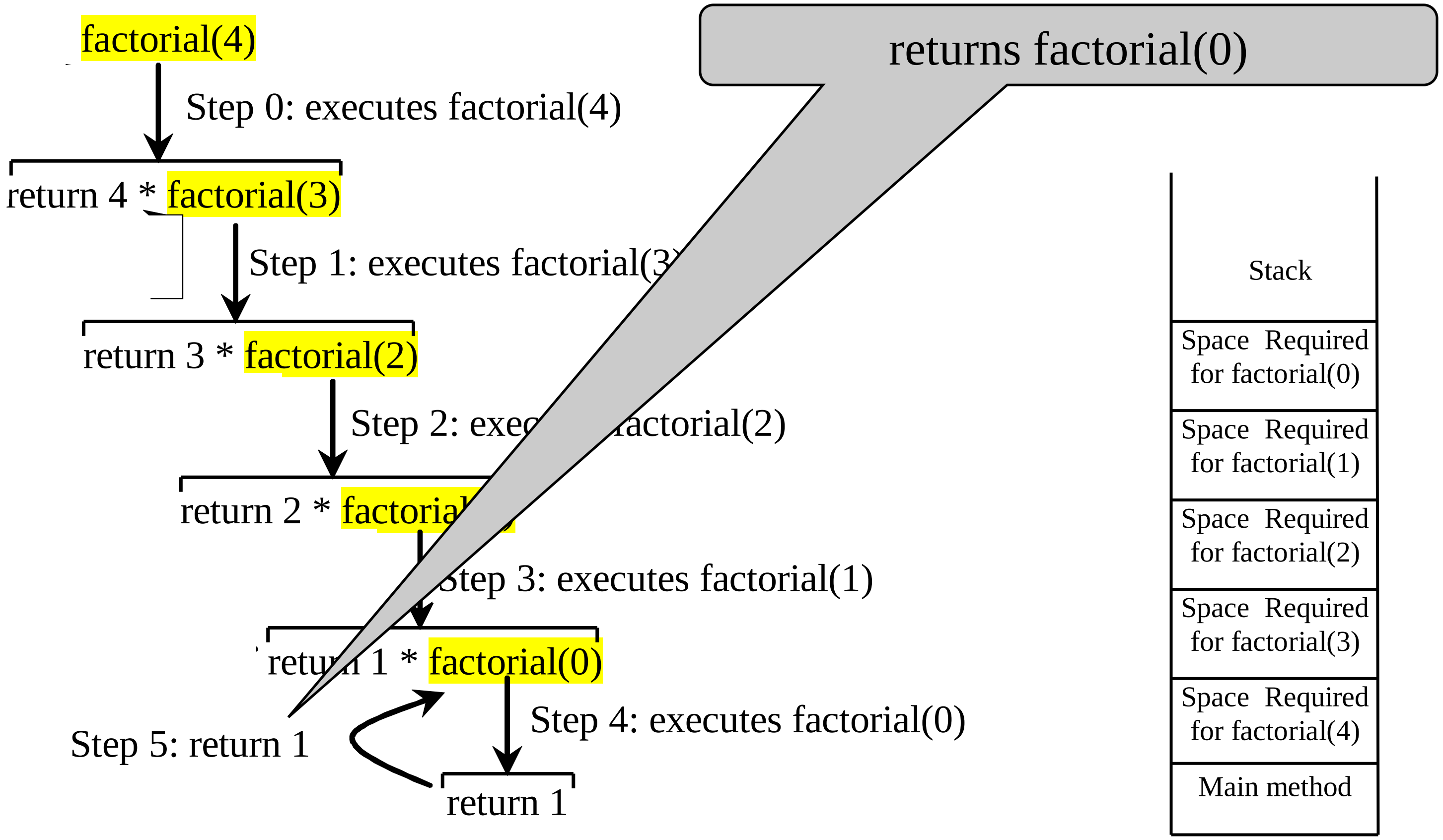
Trace Recursive factorial



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

animation

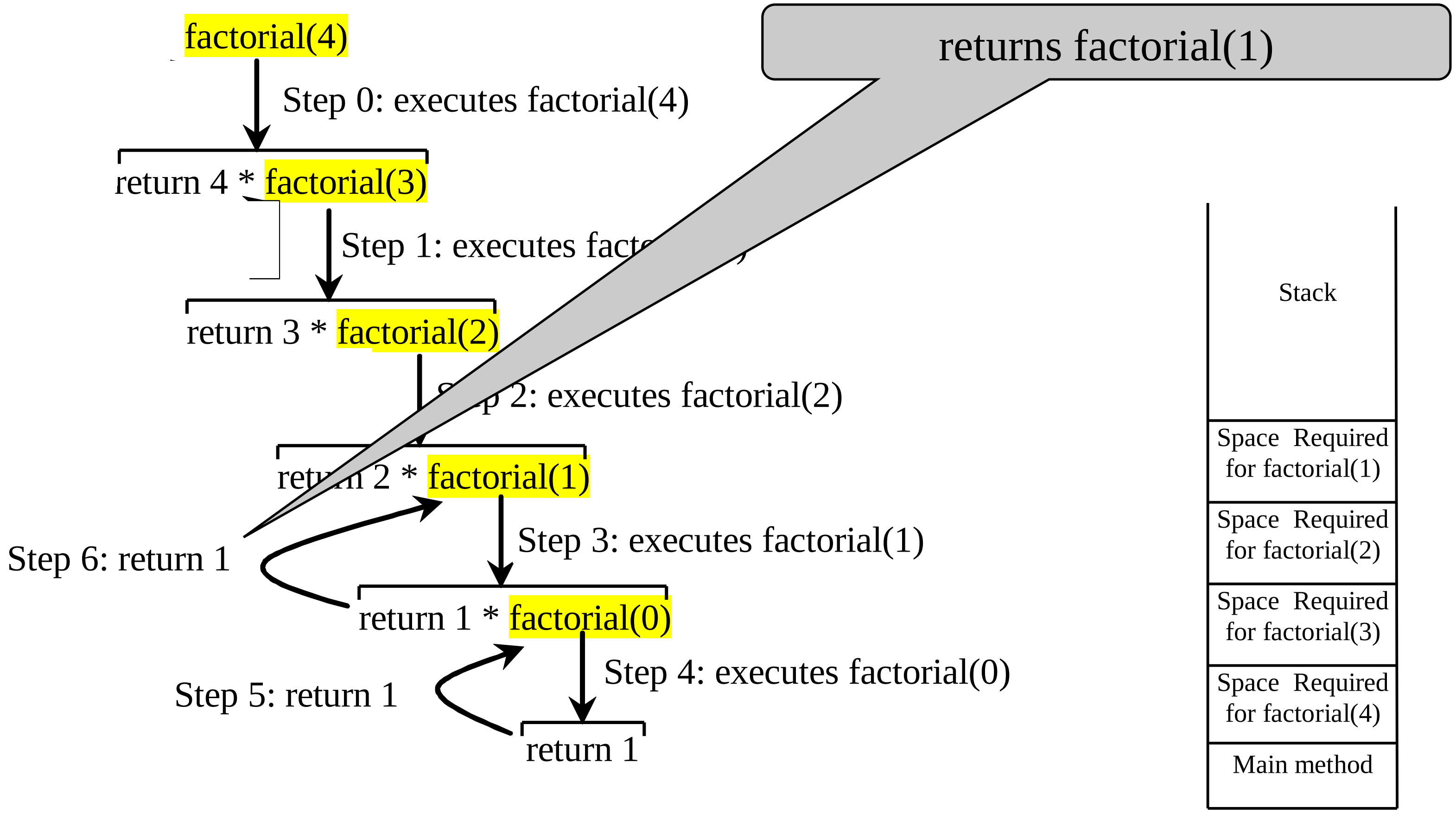
Trace Recursive factorial



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

animation

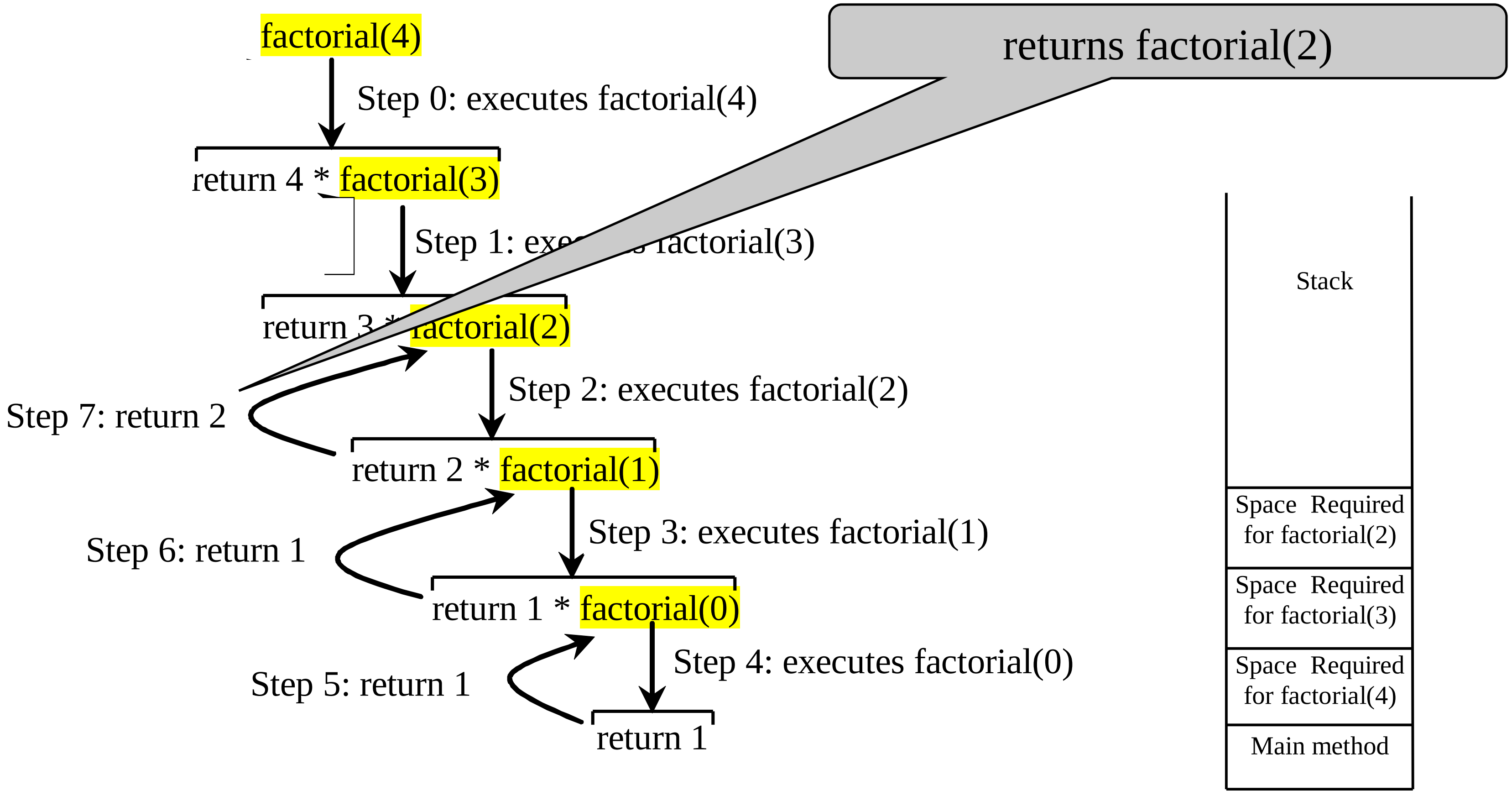
Trace Recursive factorial



returns factorial(1)
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
Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

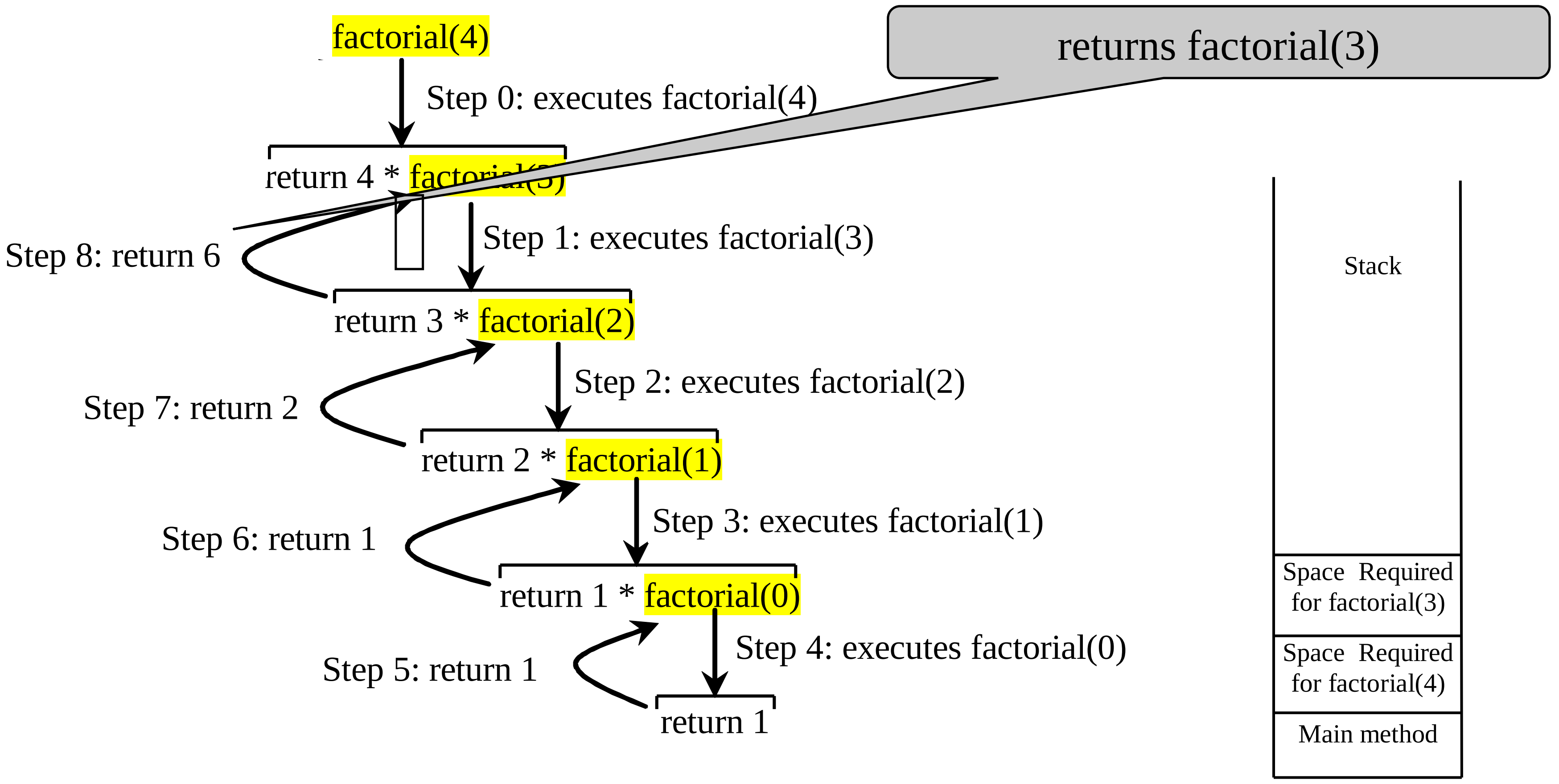
animation

Trace Recursive factorial


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

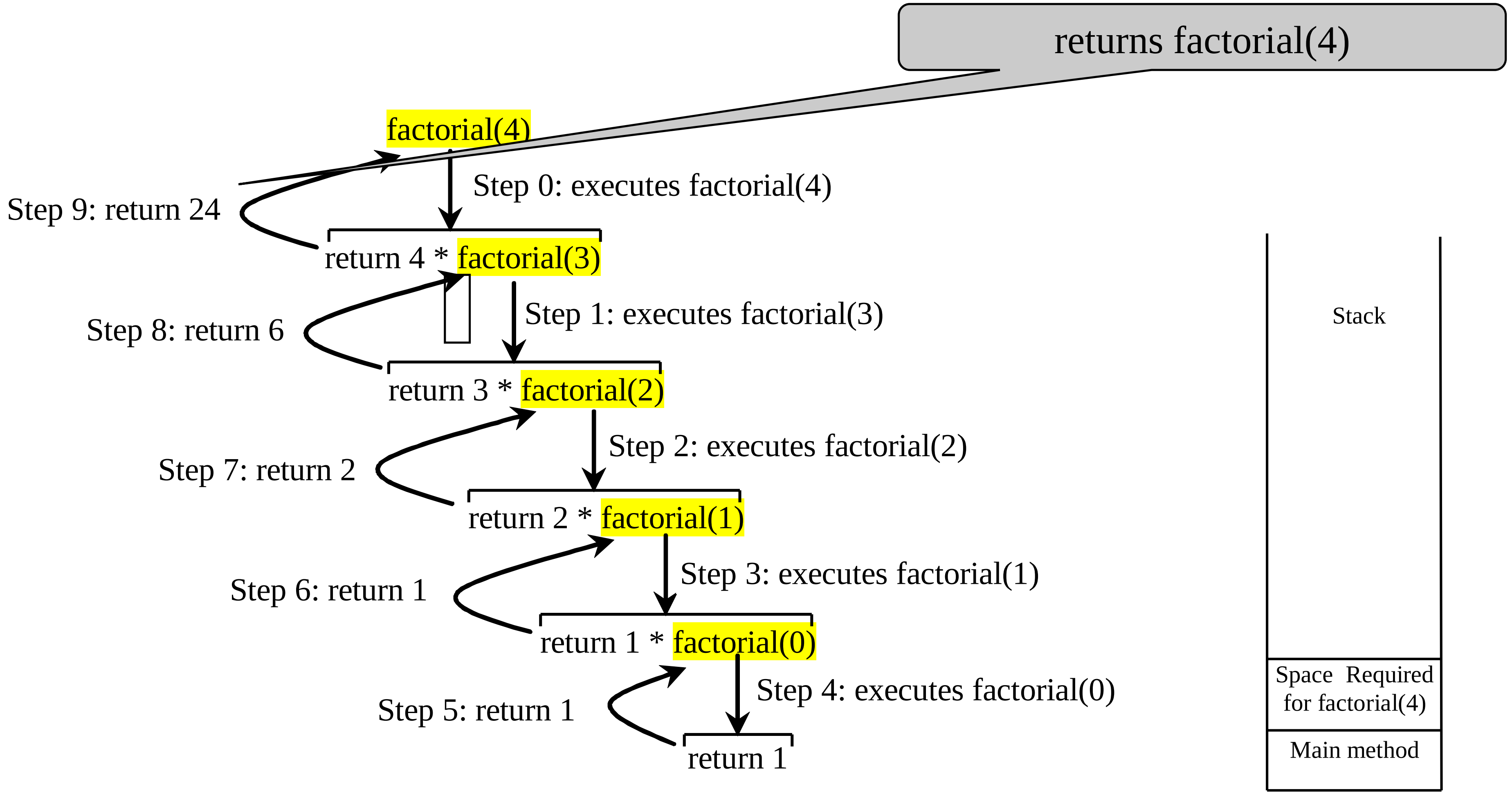
animation

Trace Recursive factorial

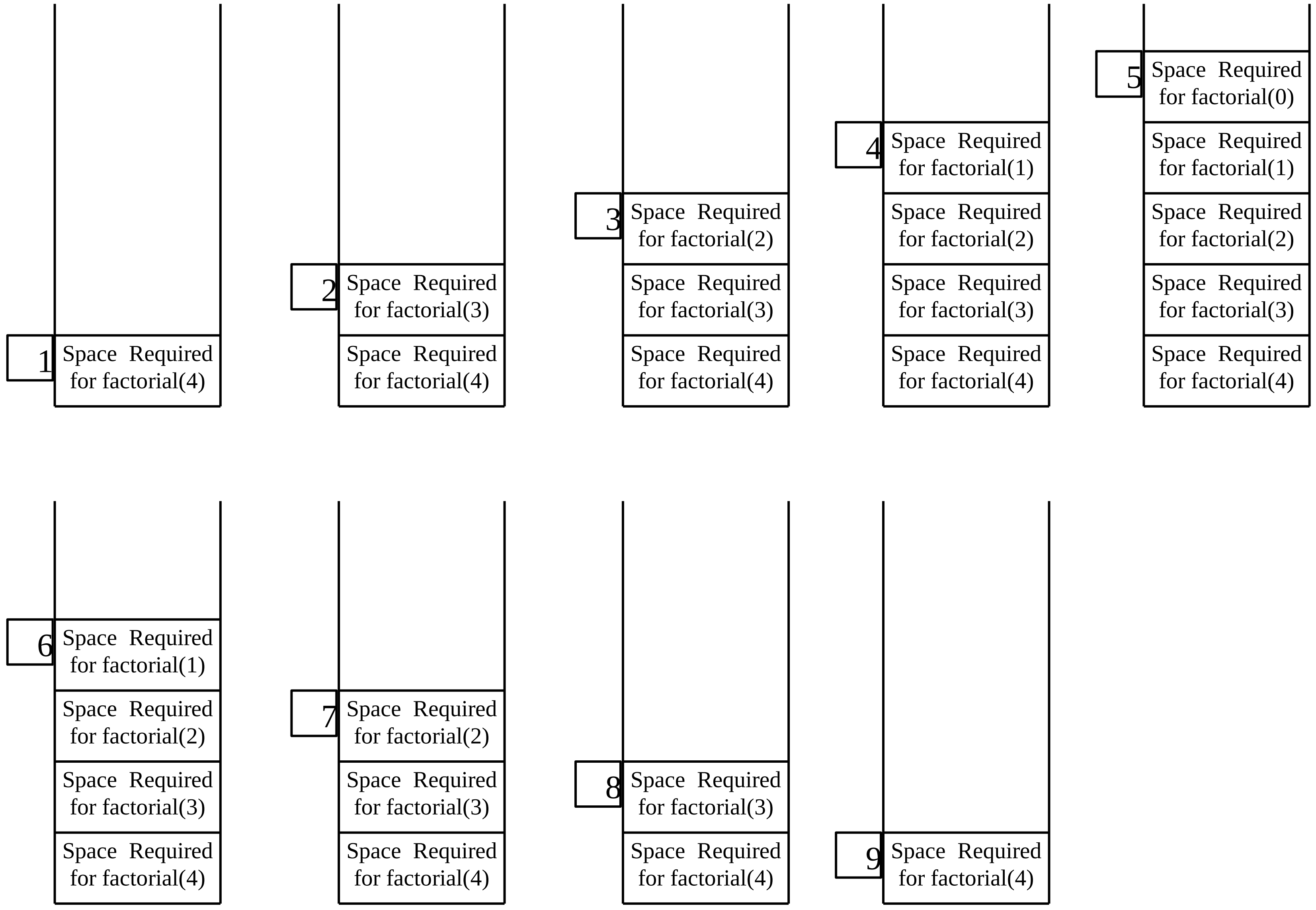


animation

Trace Recursive factorial



factorial(4) Stack Trace


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)

Example: Sum Function

sum(n) = n + n-1 + n-2 + n-3 + … + 0

sum(n) = n + sum(n-1);

sum(0) = 0; //special case

```
Sum(5)  = 5 + sum(4)
        = 5 + (4 + sum(3))
        = 5 + (4 + (3 +  sum(2)))
        = 5 + (4 + (3 + (2 + sum(1))))
        = 5 + (4 + (3 + (2 + (1 + sum(0)))))
        = 5 +  4 +  3 +  2 +  1 + 0
        = 15
```

Example: Sum Function

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

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
```

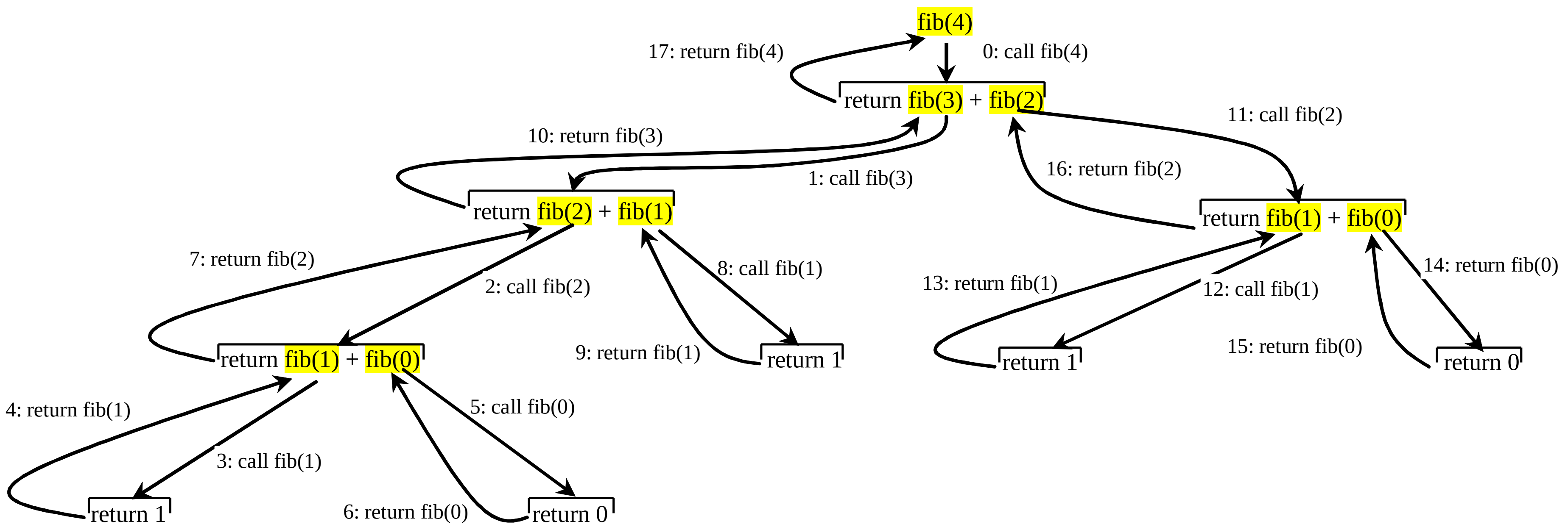
fib(0) = 0;

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
```

Fibonnaci Numbers



Fibonnaci Numbers

```
/** Return the fibonacci number */
public static long fib(long index)
{
  if (index == 0) // Base case
     return 0;
  else if (index == 1)
     return 1;
  else
     return fib(index-1) + fib(index-2); // Recursive call
}
```

Characteristics of Recursion

All recursive methods have the following characteristics:

- Base case: One or more base cases (the simplest case) are used to stop/terminate the recursive process.
- Reduction process: 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, 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.

Problem Solving Using Recursion

Consider the problem of printing a message for n times.

You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for n-1 times.

The base case is n==0.

You can solve this problem using recursion as follows:

```
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
}
```

HW: Re-write it such that counting is in ascending order.

Think Recursively

Many of the problems solved using loops can be solved using recursion if you *think recursively*.

For example, the palindrome problem can be solved recursively.

See non-recursive and recursive solutions next slides.

Non-Recursive Palindrome

```
// Demonstrates the use of nested while loops.
import java.util.Scanner;
public class PalindromeTester
{
 public static void main (String[] args)
 { String str, another = "y";
   int left, right;
   Scanner scan = new Scanner (System.in);
   while (another.equalsIgnoreCase("y")) // allows y or Y
   {
     System.out.println ("Enter a potential palindrome string:");
     str = scan.nextLine();
     left = 0;
     right = str.length() - 1;
     while (str.charAt(left) == str.charAt(right) && left < right)
     {
       left = left + 1;
       right = right - 1;
     }
     System.out.println();
     if (left < right)
        System.out.println ("That string is NOT a palindrome.");
     else
        System.out.println ("That string IS a palindrome.");
    System.out.println();
    System.out.print ("Test another palindrome (y/n)? ");
    another = scan.nextLine();
   }
 }
}
```

Recursive Palindrome

```
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));
} //This solution creates too many substrings!
```

Note: each recursive call passes a new string object since function **substring()** create a new string object. This is inefficient use of memory space.

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 (overloaded version of the method):

```
public static void main (String[] args){
  boolean result = isPalindrome("racecar");
  System.out.println ("result = " + result);
}
//-----------------------------------------------------------
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);
}
```

Other Examples…

Some problems are difficult to solve without using recursion.

Binary search: Searching an ordered list for a target value, which may be in the list or may not.

Selection sort: Find the smallest number in the list and swaps it with the first number. Ignore the first number and sort the remaining smaller list recursively as in step 1.

Finding 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.

Towers of Hanoi: Moving the disks from tower A to tower C in the same order.

- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Fractals: A shape or figure that repeats itself a number of time creating a symmetric shape.

Recursive Binary Search

1. Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
2. Case 2: If the key is equal to the middle element, the search ends with a match.
3. Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.

Recursive Implementation

```
// Use binary search to find the key in the list
public static int recursiveBinarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  return recursiveBinarySearch(list, key, low, high);
}

// Use binary search to find the key in the list between
// list[low] list[high]
public static int recursiveBinarySearch(int[] list, int key,
  int low, int high) {
  if (low > high)  // The list has been exhausted without a match
    return -low - 1;

  int mid = (low + high) / 2; // Find mid element

  if (key < list[mid]) // Check for key match
    return recursiveBinarySearch(list, key, low, mid - 1);
  else if (key == list[mid])
    return mid;
  else
    return recursiveBinarySearch(list, key, mid + 1, high);
}
```

Recursive Selection Sort

1. Find the smallest number in the list and swaps it with the first number.
2. Ignore the first number and sort the remaining smaller list recursively as in step 1.

Recursive Selection Sort

```
private static void sort(double[] list, int low, int high)
{
  if (low < high)
  {
  // Find the smallest number and its index in list(low .. high)
  int indexOfMin = low;
  double min = list[low];
  for (int i = low + 1; i <= high; i++) {
     if (list[i] < min)
     {
        min = list[i];
        indexOfMin = i;
     }
  }

  // Swap the smallest in list(low .. high) with list(low)
  list[indexOfMin] = list[low];
  list[low] = min;

  // Sort the remaining list(low+1 .. high)
  sort(list, low + 1, high);
    }
  }
```

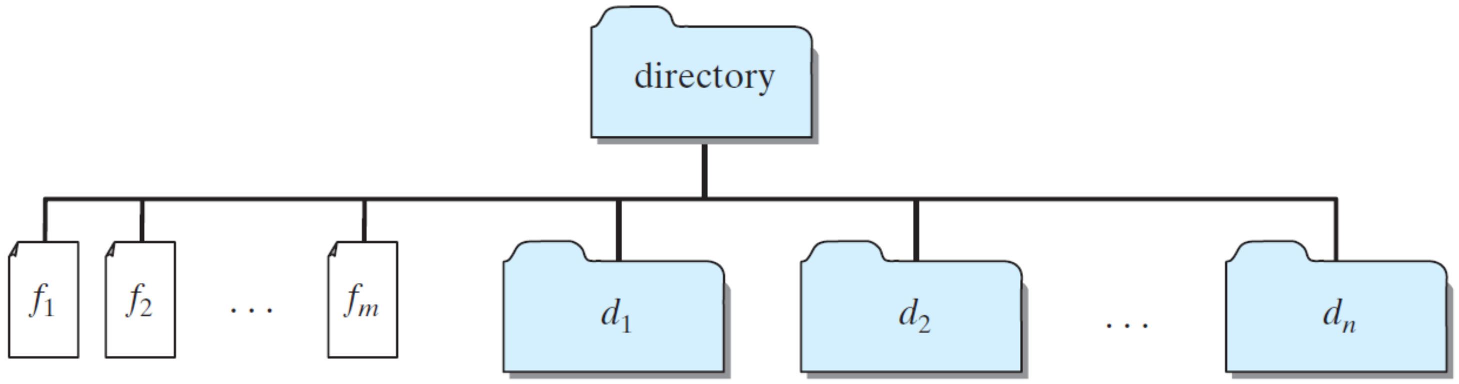
Directory Size

Some problems are difficult to solve without using recursion.

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

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



Directory Size

```
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 subdirs
    for (int i = 0; files != null && i < files.length; i++)
    {
      size = size + getSize(files[i]); //Recursive call
    }
  }
  else  //Base case
    size = size + file.length();

  return size;
}
```

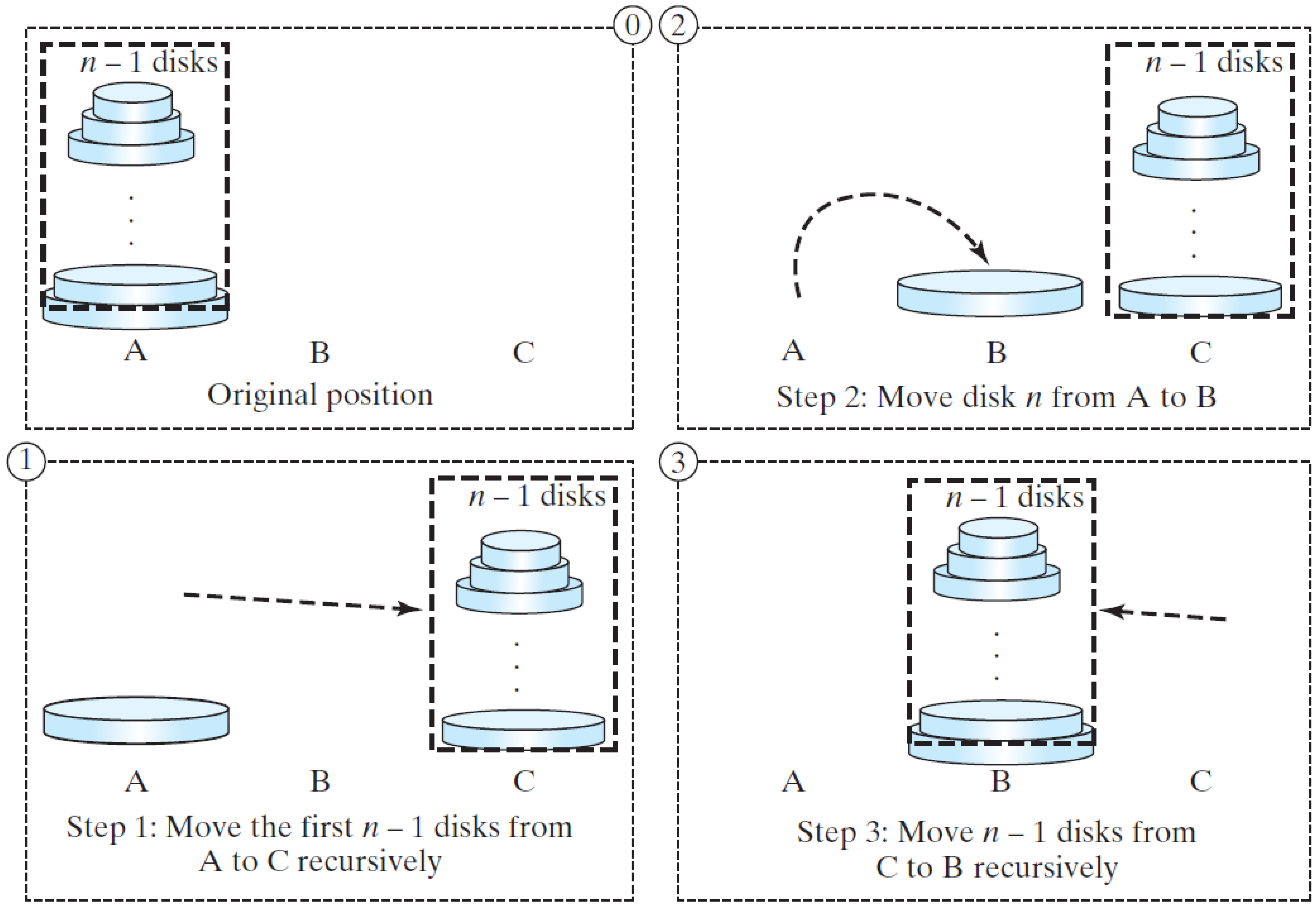
Tower of Hanoi

- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

<u>**The solution:**</u> The problem can be decomposed into three subproblems.

- Move the first <u>n - 1</u> disks from A to C with the assistance of tower B.
- Move disk <u>n</u> from A to B.
- Move <u>n - 1</u> disks from C to B with the assistance of tower A.

Tower of Hanoi


0
$n-1$ disks
A
B
C
Original position
2
$n-1$ disks
A
B
C
Step 2: Move disk n from A to B
1
$n-1$ disks
A
B
C
Step 1: Move the first $n-1$ disks from A to C recursively
3
$n-1$ disks
A
B
C
Step 3: Move $n-1$ disks from C to B recursively

Tower of Hanoi

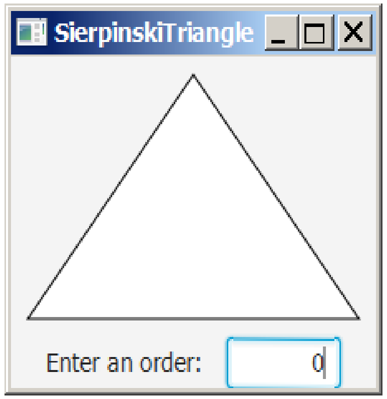
```
public static void moveDisks(int n, char fromTower,
                              char toTower, char auxTower)
{
  if (n == 1) // base case
    System.out.println("Move disk " + n + " from " +
                        fromTower + " to " + toTower);
  else //Recursive call
  {
    moveDisks(n - 1, fromTower, auxTower, toTower);
    System.out.println("Move disk " + n + " from " +
                        fromTower + " to " + toTower);
    moveDisks(n - 1, auxTower, toTower, fromTower);
  }
}
```

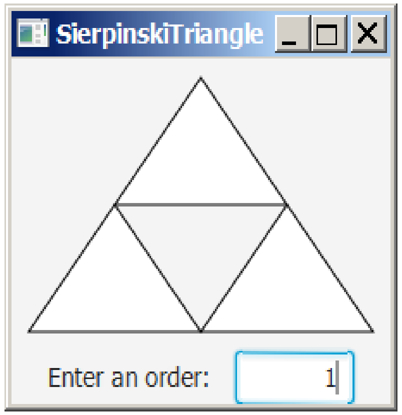
Note:

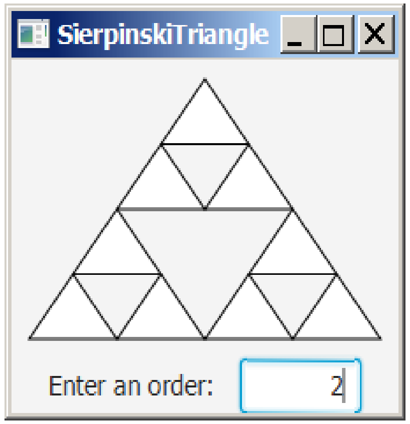
The goal is to move from tower A (**`fromTower`**) to tower B (**`toTower`**) with the help of tower C (**`auxTower`**).

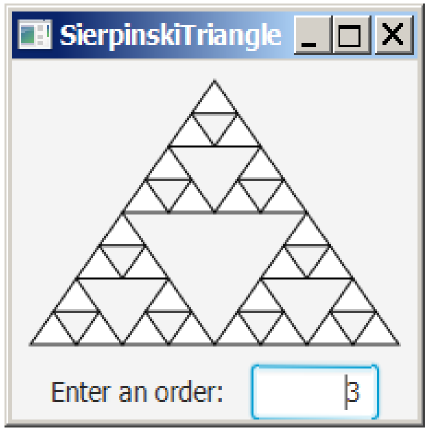
Fractals - Sierpinski Triangle

1. It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0, as shown in Figure (a).
2. Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1, as shown in Figure (b).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2, as shown in Figure (c).
4. You can repeat the same process recursively to create a Sierpinski triangle of order 3, 4, ..., and so on, as shown in Figure (d).

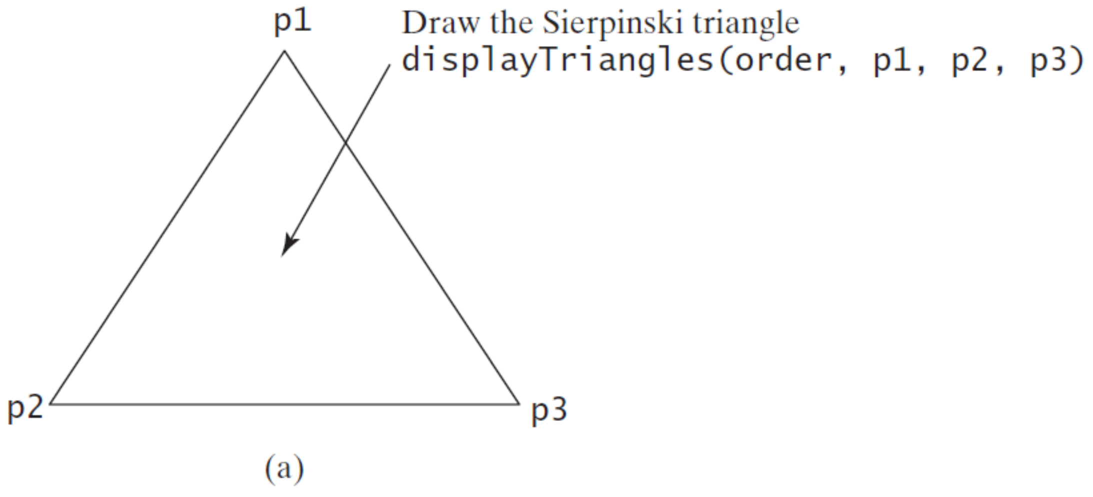




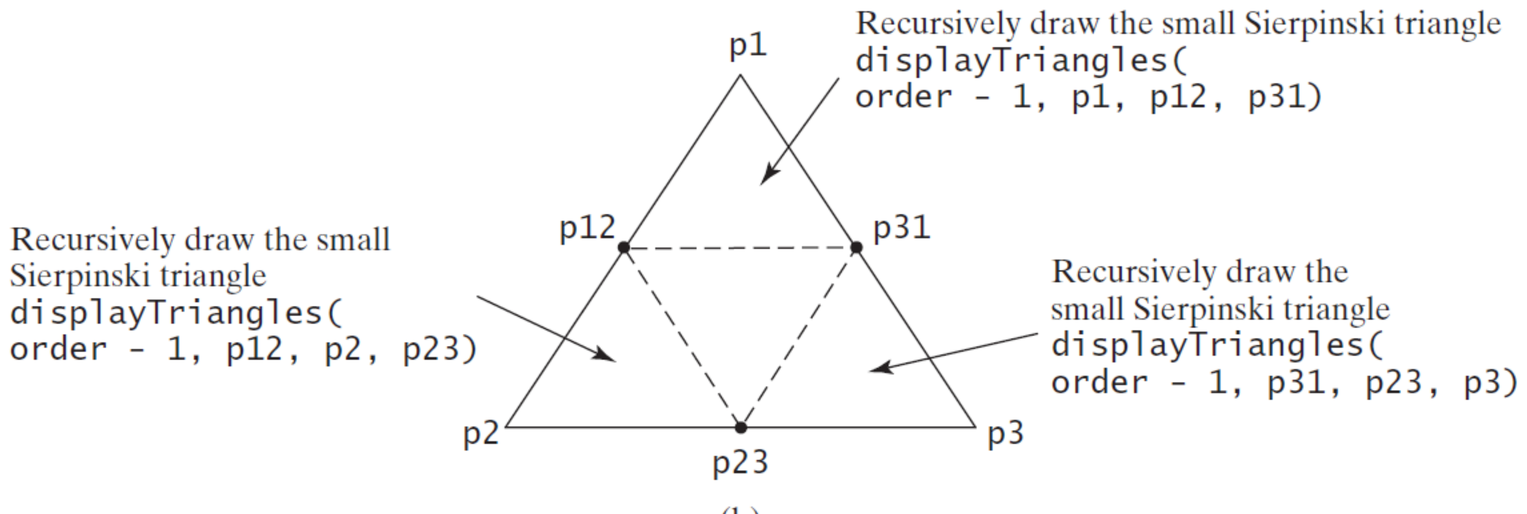




Sierpinski Triangle Solution


p1
Draw the Sierpinski triangle
displayTriangles(order, p1, p2, p3)
p2
p3

(a)



(b)

Other Examples to Think About…

Sum of digits in a positive integer: Use the division (/) and remainder (%) operators to extract digits one at a time. The base case would be when (**`n == 0`**), meaning that all digits have been extracted and added to sum. Recursive call is to return (n % 10 + sumDigits(n / 10));

Print the content of an array: Traverse the array one element at a time. The base case would be when the array length reaches 0 elements.

Print a string in reverse order: Traverse the string one character at a time. The base case would be when the end of the string is reached (that is, a counter variable is equal to the length of the string).

Return the reverse input string: Parse the input string one character at a time and add it to reverse string backward. The base case would be when the input string is empty (something like: **`inputString.equals("")==true`**).

Find the Greatest Common Divisor (GCD) of 2 numbers: Use the remainder (%) operator. The base case is when the remainder is zero, you return the second number.

Print the binary representation of an number: Use the division (/) and remainder (%) operators to reduce the number and get the binary digit, respectively. The base case is when the number is either 0 or 1.

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 memory space for all of the method's local variables and parameters (i.e., each call creates a new activation record on the runtime stack). This can consume considerable memory space and requires extra time to manage the additional space.

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. Otherwise, called *non-tail recursive*

End of Chapter 18