Chapter 18

Recursion

Objectives

- To solve problems using recursion
- To use an overloaded helper method to derive a recursive method
- To implement a selection sort using recursion
- To implement a binary search using recursion
- To get the directory size using recursion

Computing Factorial

factorial(0) = 1;

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

n! = n * (n-1)!

animation

Computing Factorial

factorial(4)

factorial(0) = 1;

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

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);

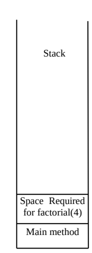
factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2
= 4 * 6
= 24

animation

Trace Recursive factorial

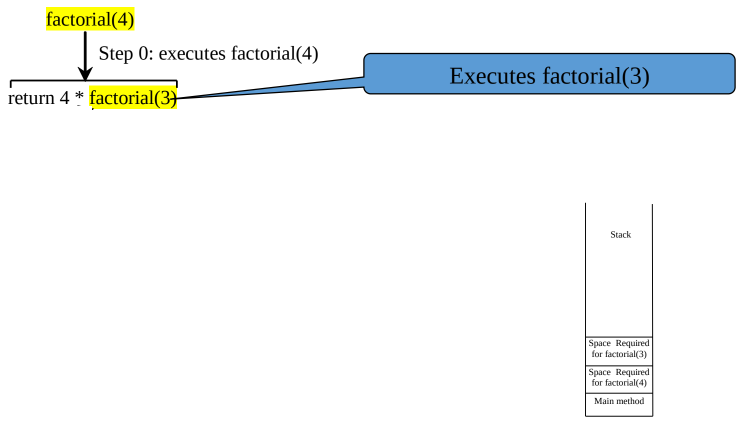
Executes factorial(4)

factorial(4)



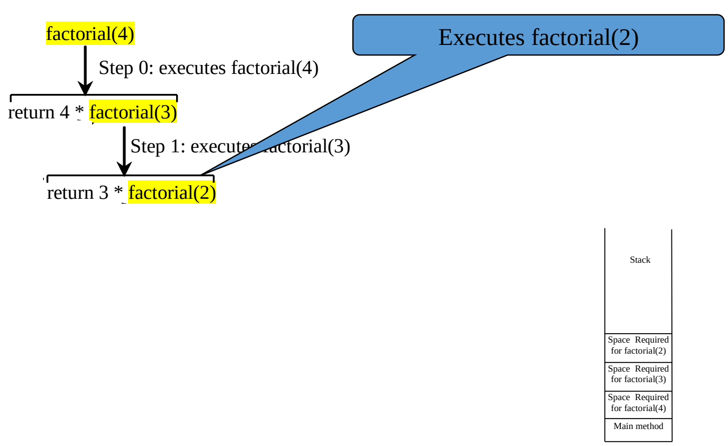
animation

Trace Recursive factorial



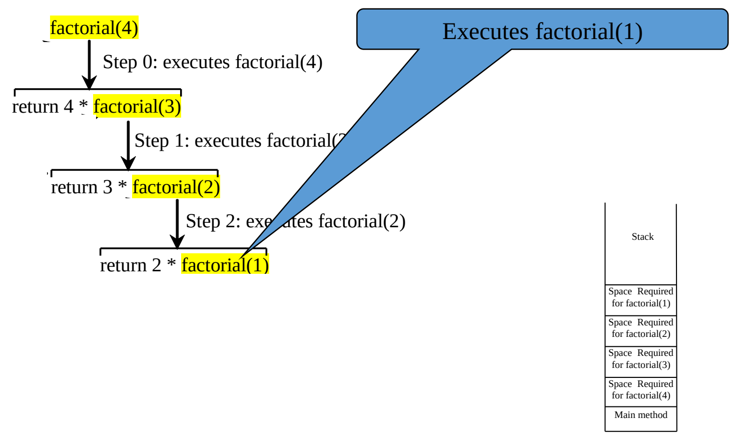
animation

Trace Recursive factorial



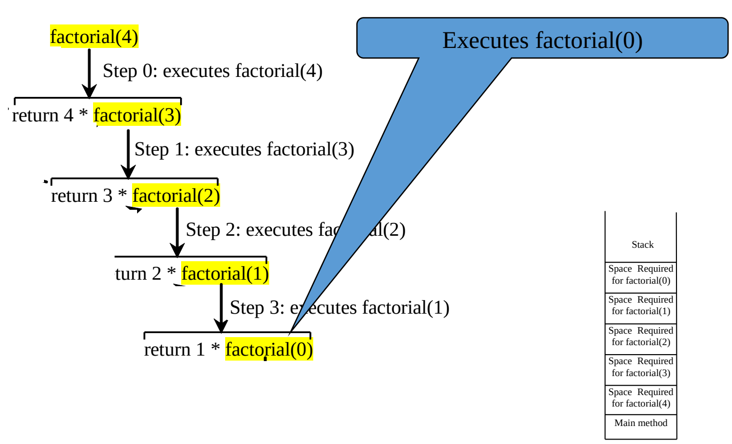
animation

Trace Recursive factorial



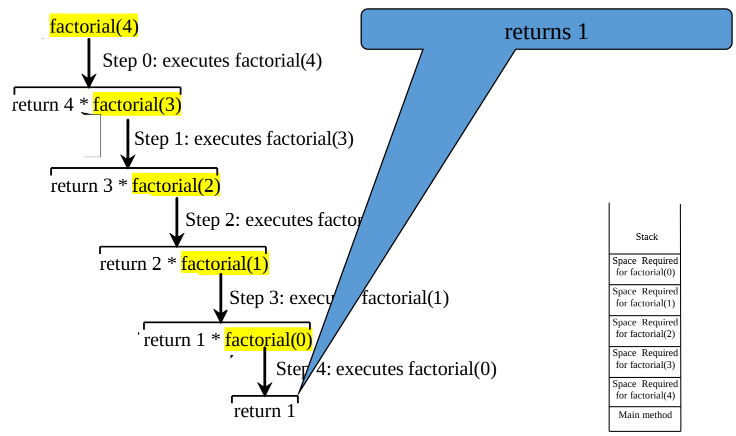
animation

Trace Recursive factorial



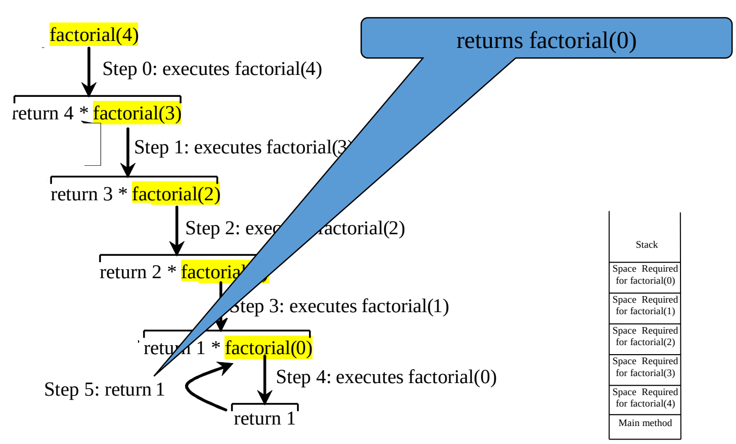
animation

Trace Recursive factorial



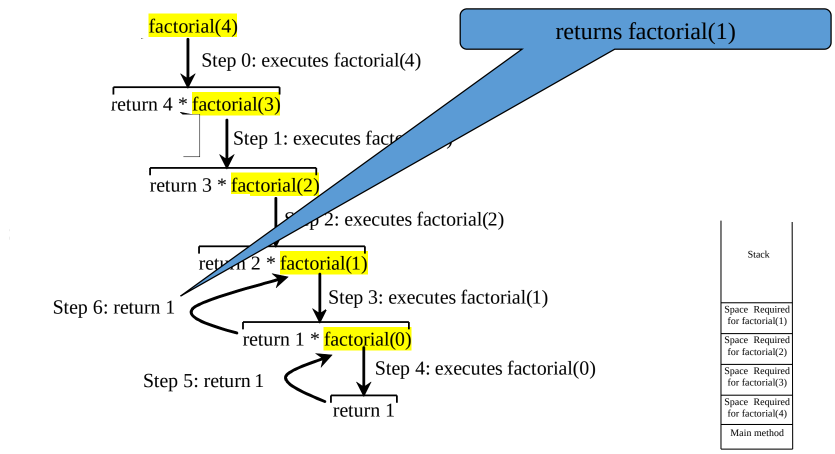
animation

Trace Recursive factorial



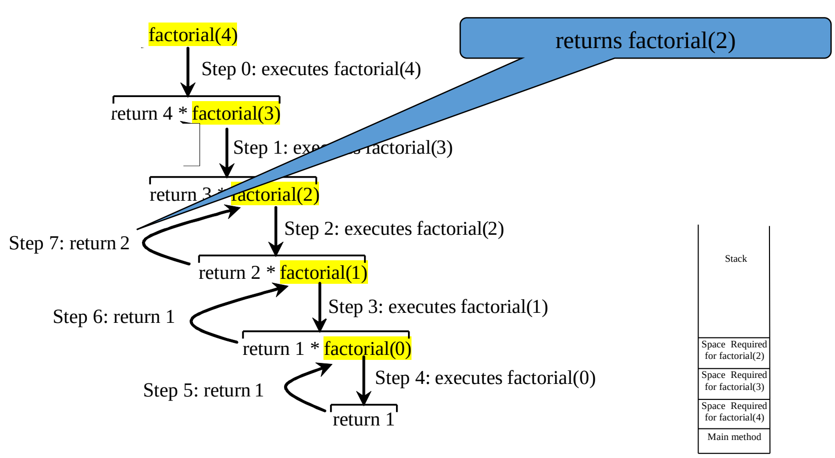
animation

Trace Recursive factorial



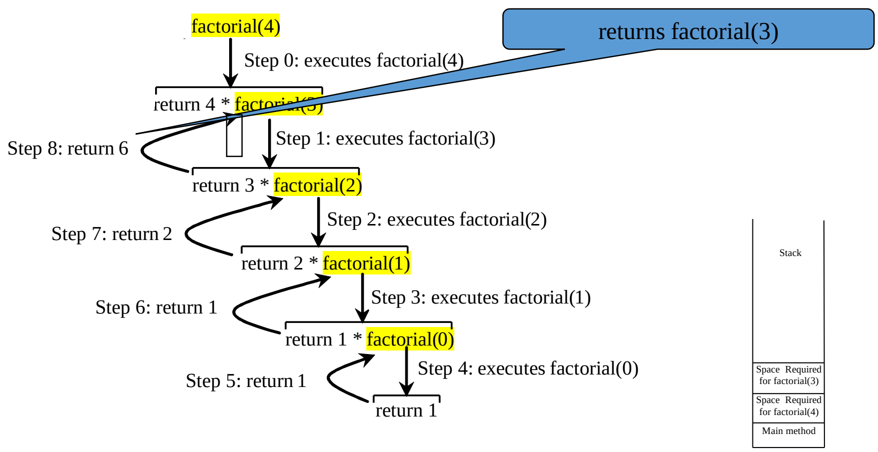
animation

Trace Recursive factorial



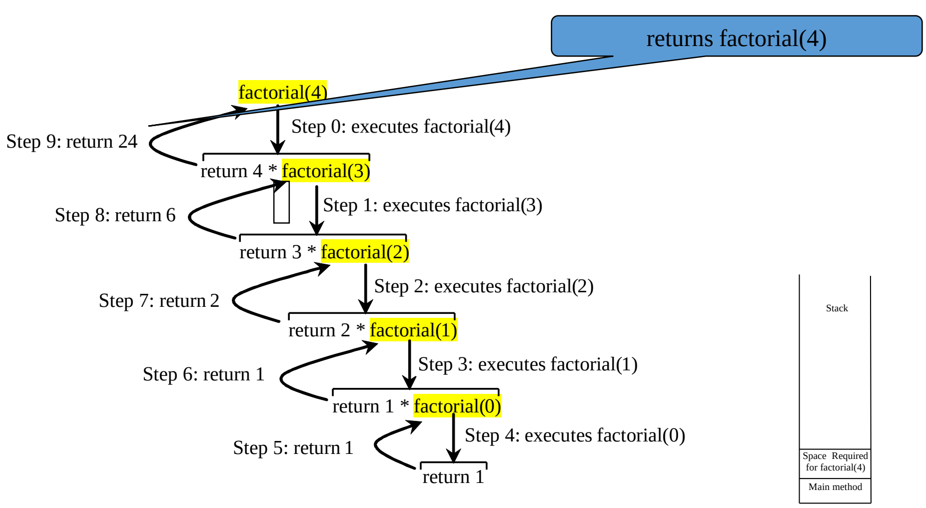
animation

Trace Recursive factorial

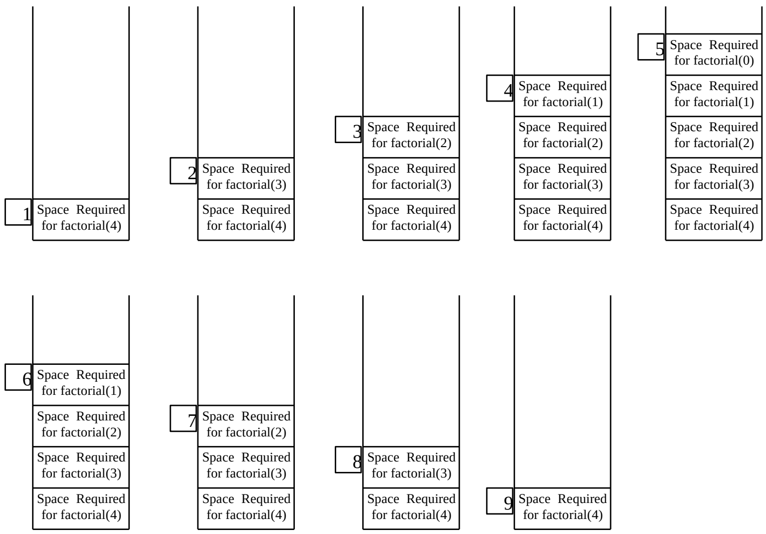


animation

Trace Recursive factorial



factorial(4) Stack Trace



Computing Factorial: Implementation

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                       base case
      return 1;
    else
      return n * factorial(n - 1); // Recursive call      recursion
  }
}
```

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.

Characteristics of Recursion

All recursive methods have the following characteristics:

- 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.

Problem Solving Using Recursion

Let us consider a simple 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 second problem is the same as the original problem with a smaller size. The base case for the problem is n==0. You can solve this problem using recursion as follows:

nPrintln("Welcome", 5);

```
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
}
```

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.*

Recursive Binary Search (with Recursive Helper Method)

- Binary search was introduced in Section 7.10.2.
- For binary search to work, the elements in the array must be in increasing order.
- The binary search first compares the key with the element in the middle of the array. Consider the following three cases:
  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.
- Case 1 and Case 3 reduce the search to a smaller list.
- Case 2 is a base case when there is a match.
- Another base case is that the search is exhausted without a match.

LISTING 18.6 Recursive Binary Search Method

```
public class RecursiveBinarySearch {
  public static int recursiveBinarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;
    return recursiveBinarySearch(list, key, low, high);
  }

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

- The first method finds a key in the whole list.
- The second method finds a key in the list with index from **low** to **high**.
- The first **binarySearch** method passes the initial array with **low = 0** and **high = list.length - 1** to the second **binarySearch** method.
- The second method is invoked recursively to find the key in an ever-shrinking subarray.

Chapter Summary

- A *recursive method* is one that invokes itself. For a recursive method to terminate, there must be one or more *base cases*.
- *Recursion* is an alternative form of program control. It is essentially repetition without a loop control.
- Sometimes the original method needs to be modified to receive additional parameters in order to be invoked recursively. A *recursive helper method* can be defined for this purpose.
- Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the memory