Recursion & Searching (7.10 & 18)

Concepts

- Recursion: Recursion is the process a method goes through when one of the steps of the method involves invoking the method itself.

- Recursive method: a method that invokes itself.

Concepts

- In computer science, a method exhibits recursive behavior when it can be defined by two properties:
  - A simple base case (or cases). Used to stop the recursion.
  - A set of rules that reduce all other cases toward the base case. Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Characteristics of Recursion

- In general, to solve a problem using recursion, you break it into sub-problems.
- If a sub-problem resembles the original problem, you can apply the same approach to solve the sub-problem recursively.
- This sub-problem is almost the same as the original problem in nature with a smaller size.

Computing Factorial
Problem Definition

factorial(0) = 1;

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

n! = n * (n-1)!

Computing Factorial

```
 1  import java.util.Scanner;
 2
 3  public class ComputeFactorial {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8      System.out.print("Enter a non-negative integer: ");
 9      int n = input.nextInt();
10
11      // Display factorial
12      System.out.println("Factorial of " + n + " is " + factorial(n));
13    }
14
15    /** Return the factorial for a specified number */
16    public static long factorial(int n) {
17      if (n == 0) // Base case
18        return 1;
19      else
20        return n * factorial(n - 1); // Recursive call
21    }
22  }
```

Computing Factorial

factorial(0) = 1;

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

factorial(3)

Computing Factorial

factorial(0) = 1;

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

factorial(3) = 3 * factorial(2)

Computing Factorial

factorial(0) = 1;

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

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

Computing Factorial

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

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

Computing Factorial

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

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

Computing Factorial

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

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

Computing Factorial

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

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

Computing Factorial

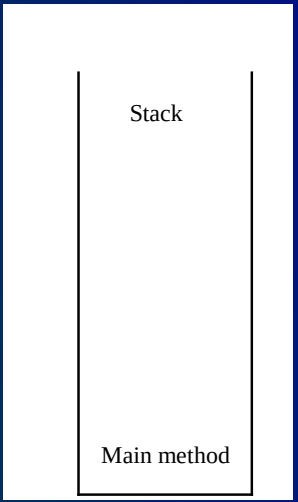
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

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

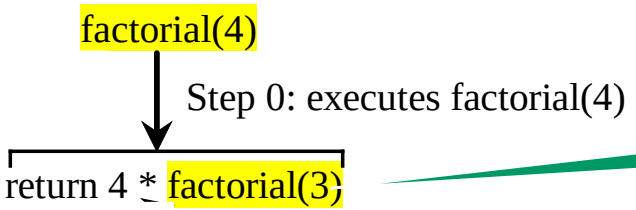
Trace Recursive factorial

Executes factorial(4)

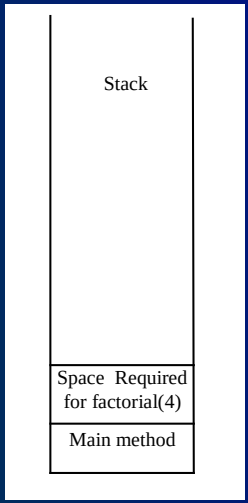
factorial(4)



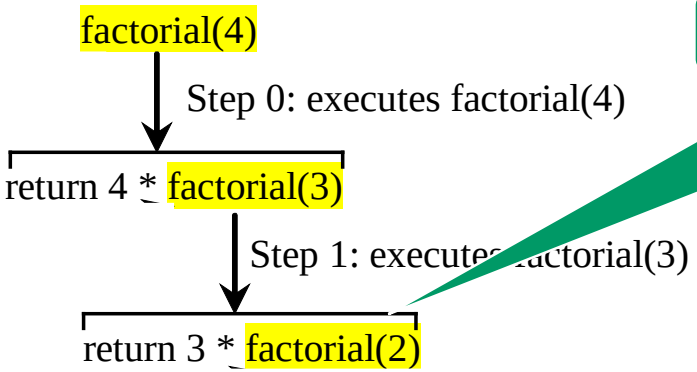
Trace Recursive factorial



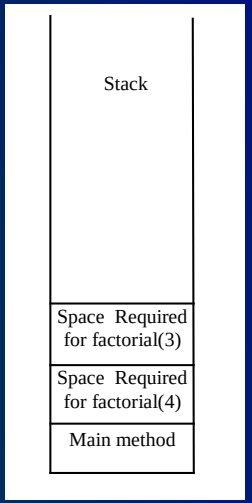
Executes factorial(3)



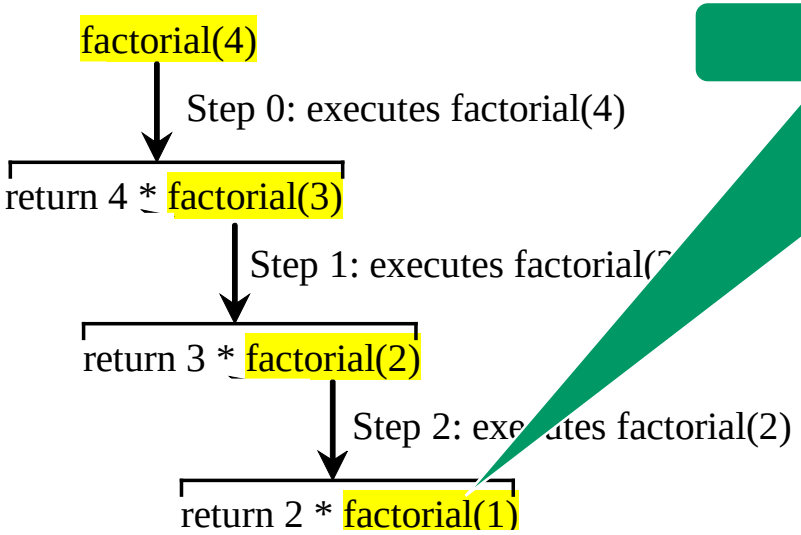
Trace Recursive factorial



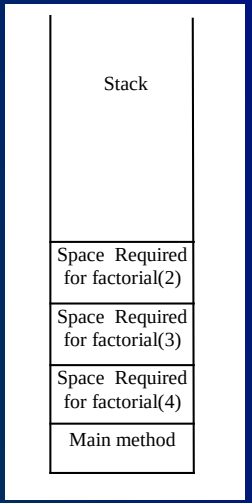
Executes factorial(2)



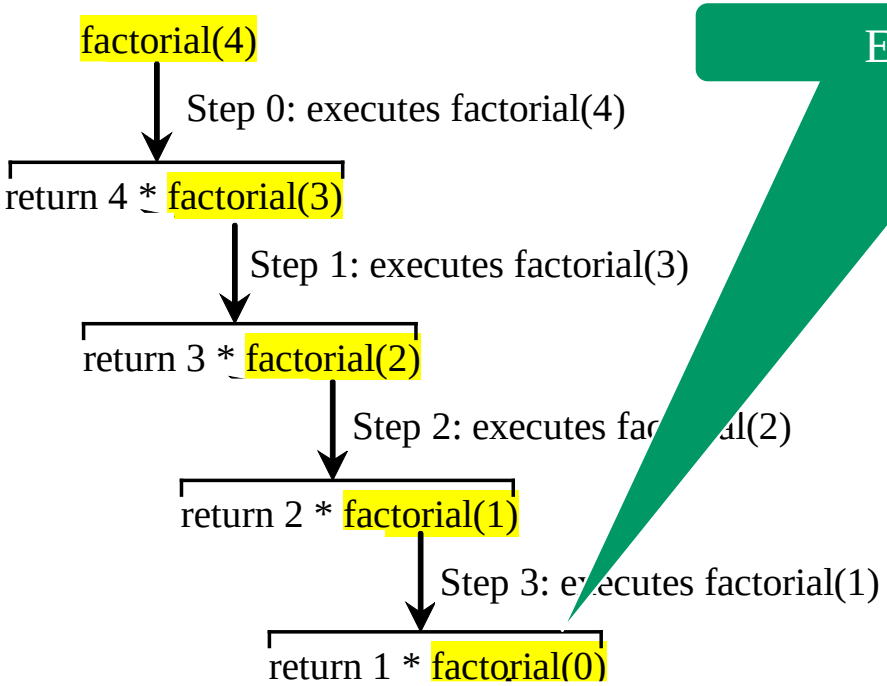
Trace Recursive factorial

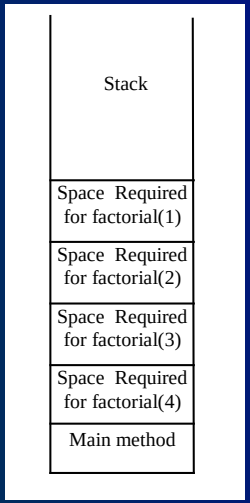


Executes factorial(1)

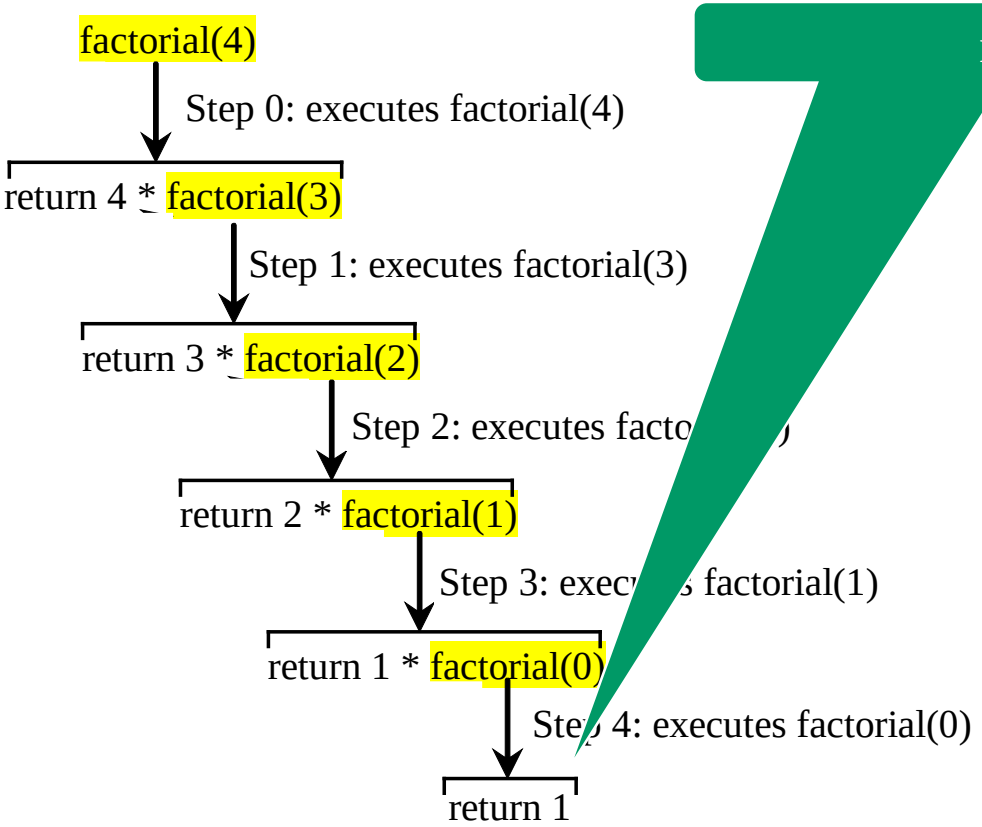


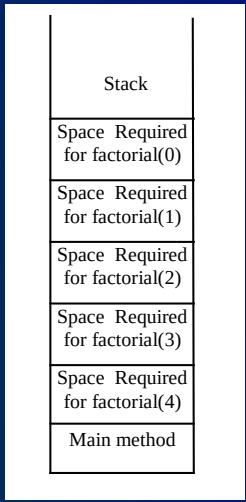
Trace Recursive factorial



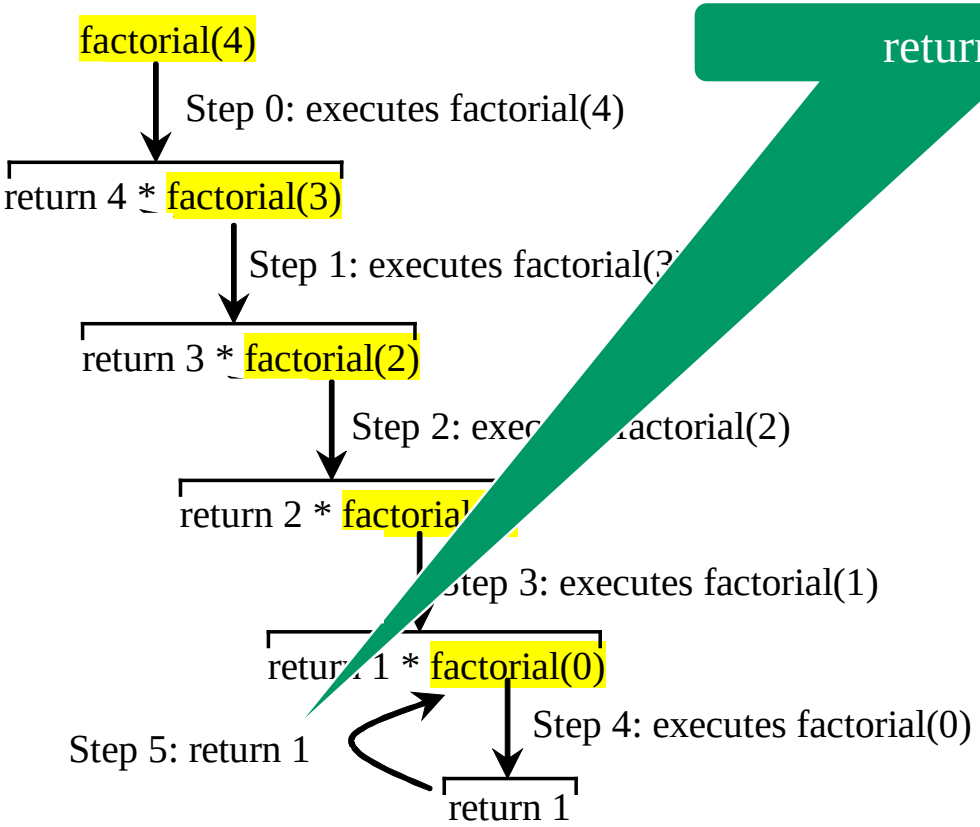


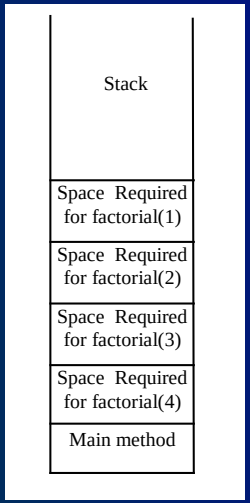
Trace Recursive factorial



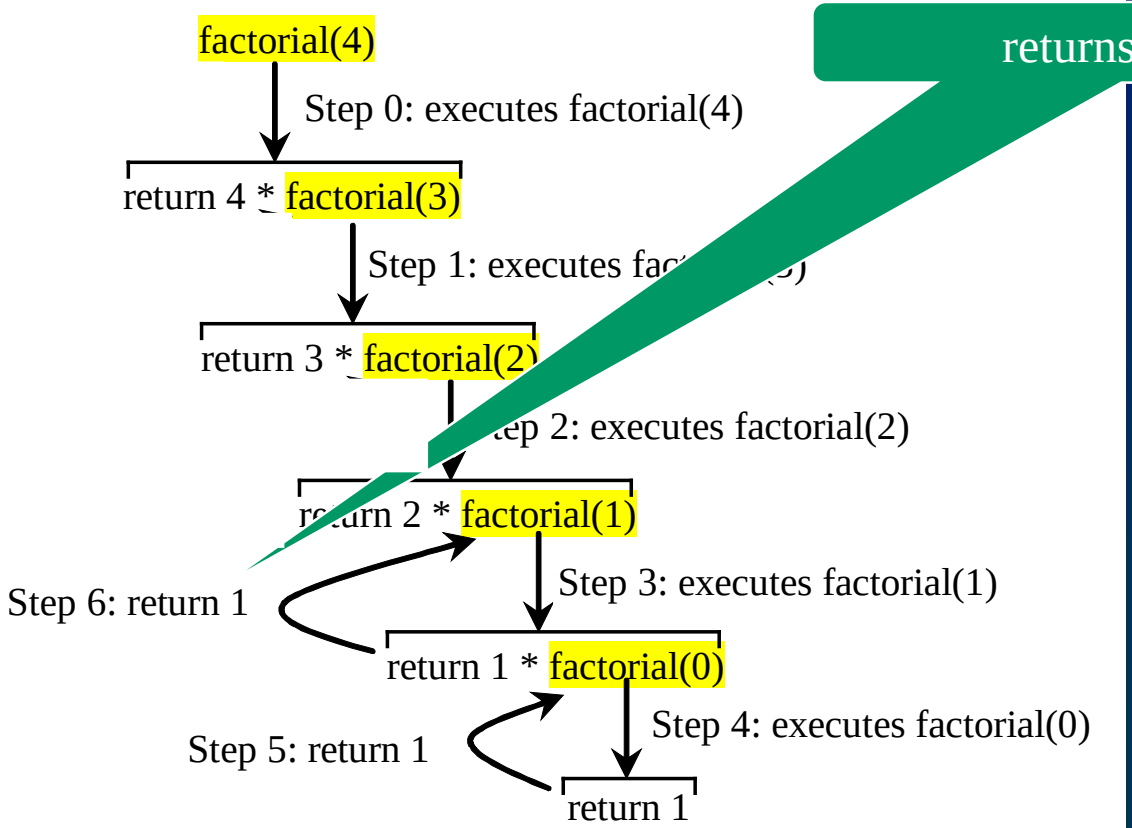


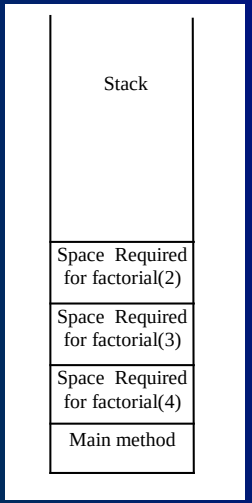
Trace Recursive factorial



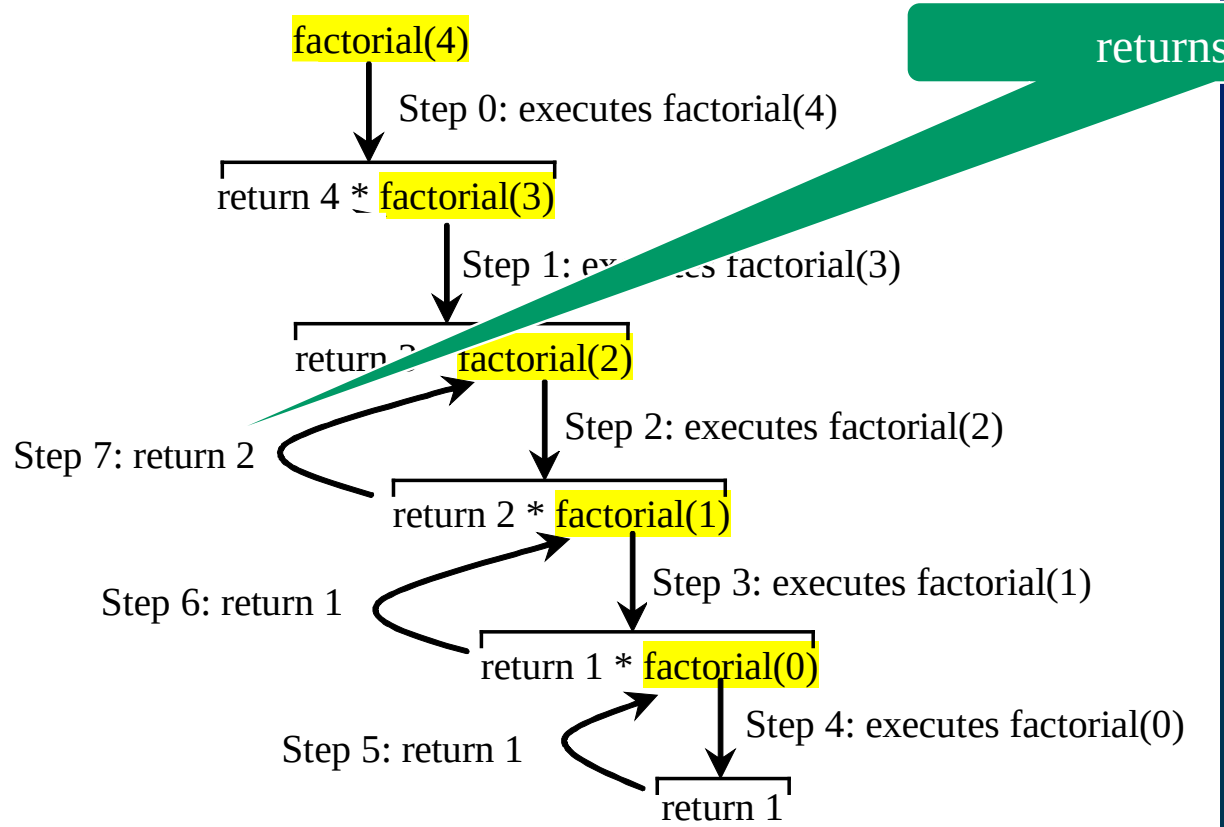


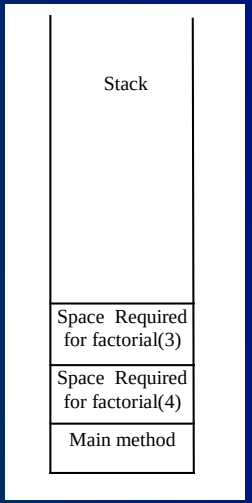
Trace Recursive factorial



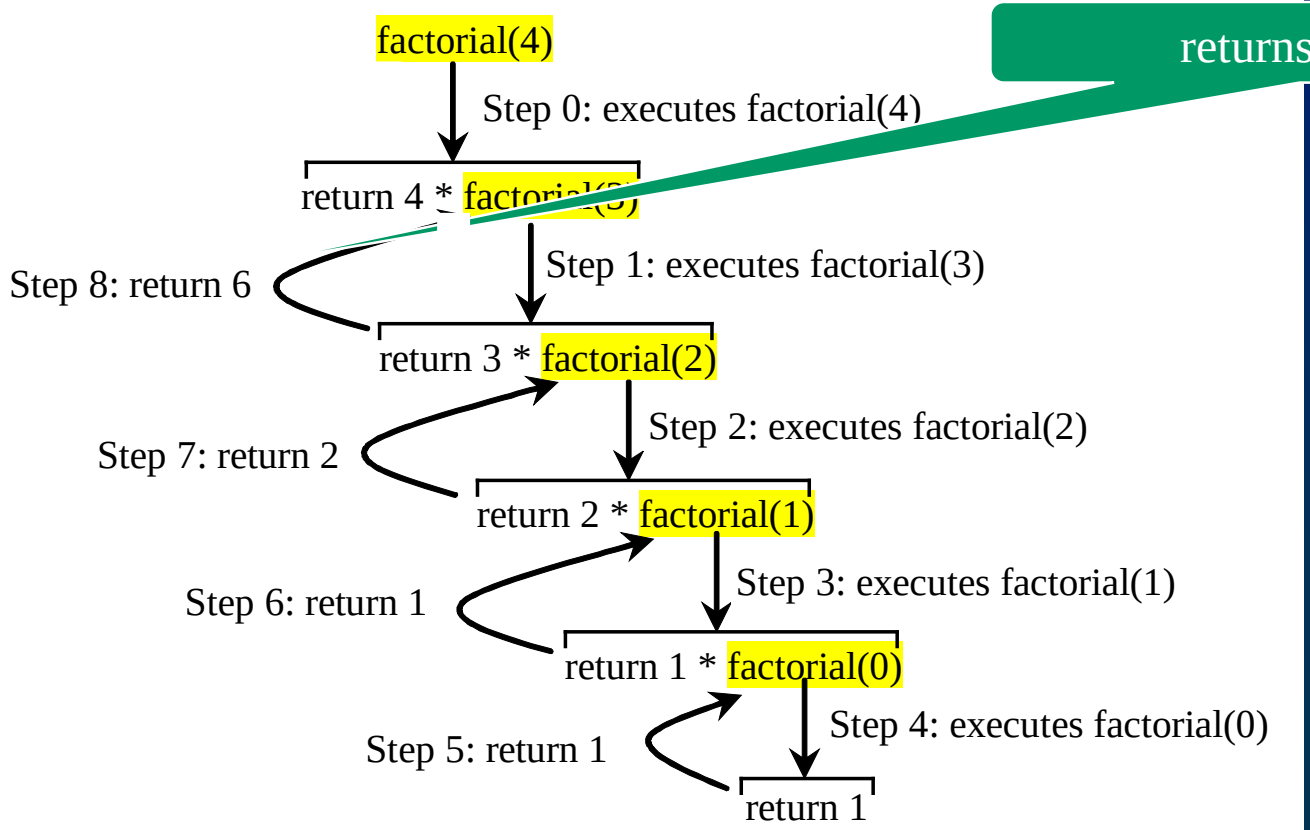


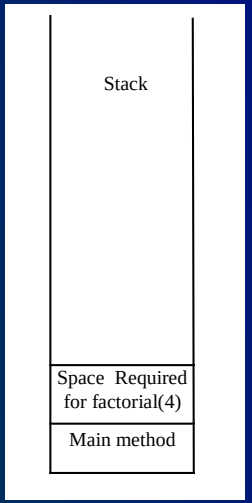
Trace Recursive factorial



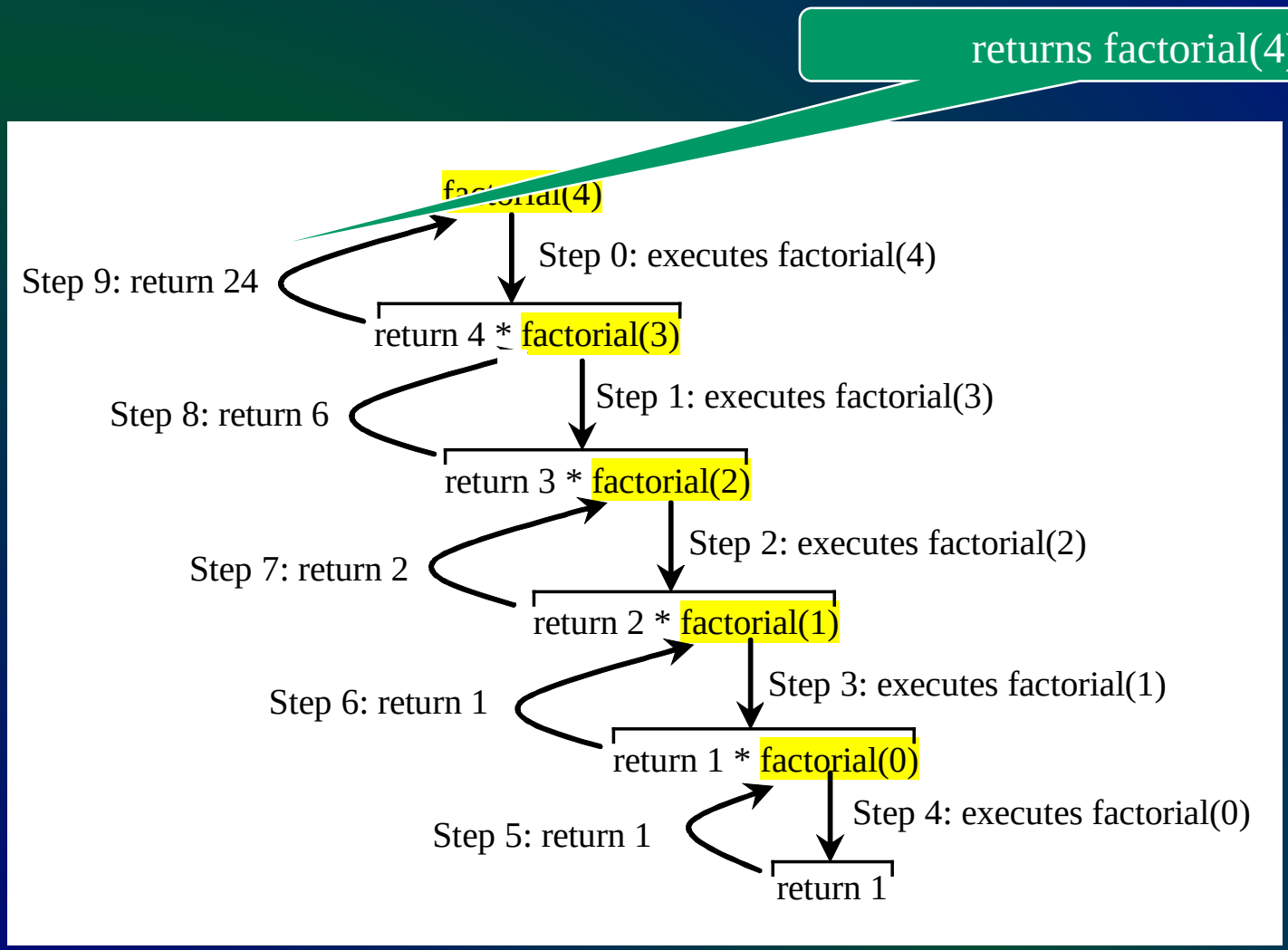


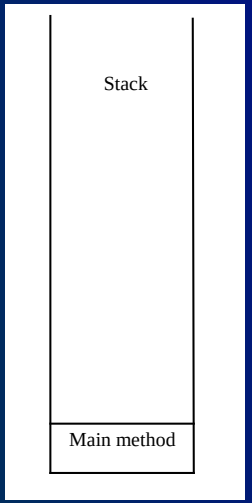
Trace Recursive factorial





Trace Recursive factorial





Fibonacci Numbers

Problem Definition

```
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;

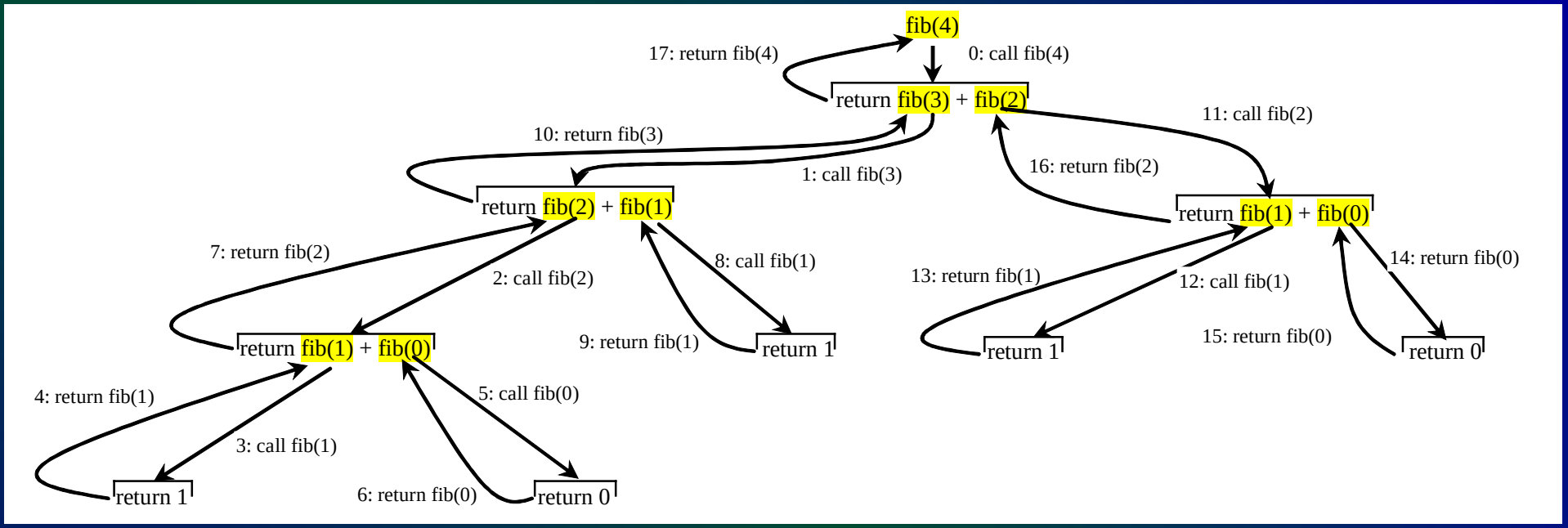
fib(index) = fib(index -1) + fib(index -2); index >=2

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

Fibonacci Numbers

```
 1  import java.util.Scanner;
 2
 3  public class ComputeFibonacci {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8      System.out.print("Enter an index for a Fibonacci number: ");
 9      int index = input.nextInt();
10
11      // Find and display the Fibonacci number
12      System.out.println("The Fibonacci number at index "
13        + index + " is " + fib(index));
14    }
15
16    /** The method for finding the Fibonacci number */
17    public static long fib(long index) {
18      if (index == 0) // Base case
19        return 0;
20      else if (index == 1) // Base case
21        return 1;
22      else  // Reduction and recursive calls
23        return fib(index - 1) + fib(index - 2);
24    }
25  }
```

Fibonnaci Numbers, cont.



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:

```
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
}
```

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));
}
```

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);
}
```

Binary Search

- A binary search algorithm finds the position of a target value within a sorted array.
- The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array.
  - If the target value is equal to the middle element's value, then the position is returned and the search is finished.
  - If the target value is less than the middle element's value, then the search continues on the lower half of the array;
  - if the target value is greater than the middle element's value, then the search continues on the upper half of the array.
- This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (and its associated element position is returned), or until the entire array has been searched (and "not found" is returned).

Binary Search without Recursion

```
 1  public class BinarySearch {
 2    /** Use binary search to find the key in the list */
 3    public static int binarySearch(int[] list, int key) {
 4      int low = 0;
 5      int high = list.length - 1;
 6
 7      while (high >= low) {
 8        int mid = (low + high) / 2;
 9        if (key < list[mid])
10          high = mid - 1;
11        else if (key == list[mid])
12          return mid;
13        else
14          low = mid + 1;
15      }
16
17      return -low - 1; // Now high < low
18    }
19  }
```

Recursive Binary Search
Problem Definition

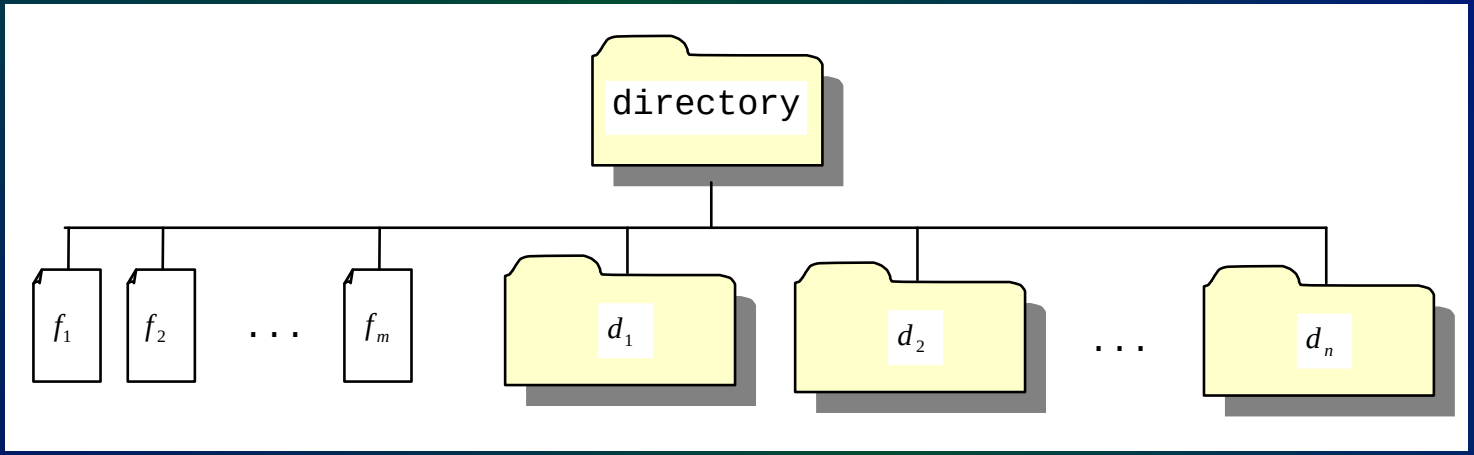
1. Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
2. Case 2: If the key is equal to the middle element, the search ends with a match.
3. Case 3: If the key is greater than the middle element, recursively search the key in the second half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

Recursive Implementation

```
 1  public class RecursiveBinarySearch {
 2    public static int recursiveBinarySearch(int[] list, int key) {
 3      int low = 0;
 4      int high = list.length - 1;
 5      return recursiveBinarySearch(list, key, low, high);
 6    }
 7
 8    private static int recursiveBinarySearch(int[] list, int key,
 9        int low, int high) {
10      if (low > high)  // The list has been exhausted without a match
11        return -low - 1;
12
13      int mid = (low + high) / 2;
14      if (key < list[mid])
15        return recursiveBinarySearch(list, key, low, mid - 1);
16      else if (key == list[mid])
17        return mid;
18      else
19        return recursiveBinarySearch(list, key, mid + 1, high);
20    }
21  }
```

Directory Size

The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to 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 , , ..., , as shown below.



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

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

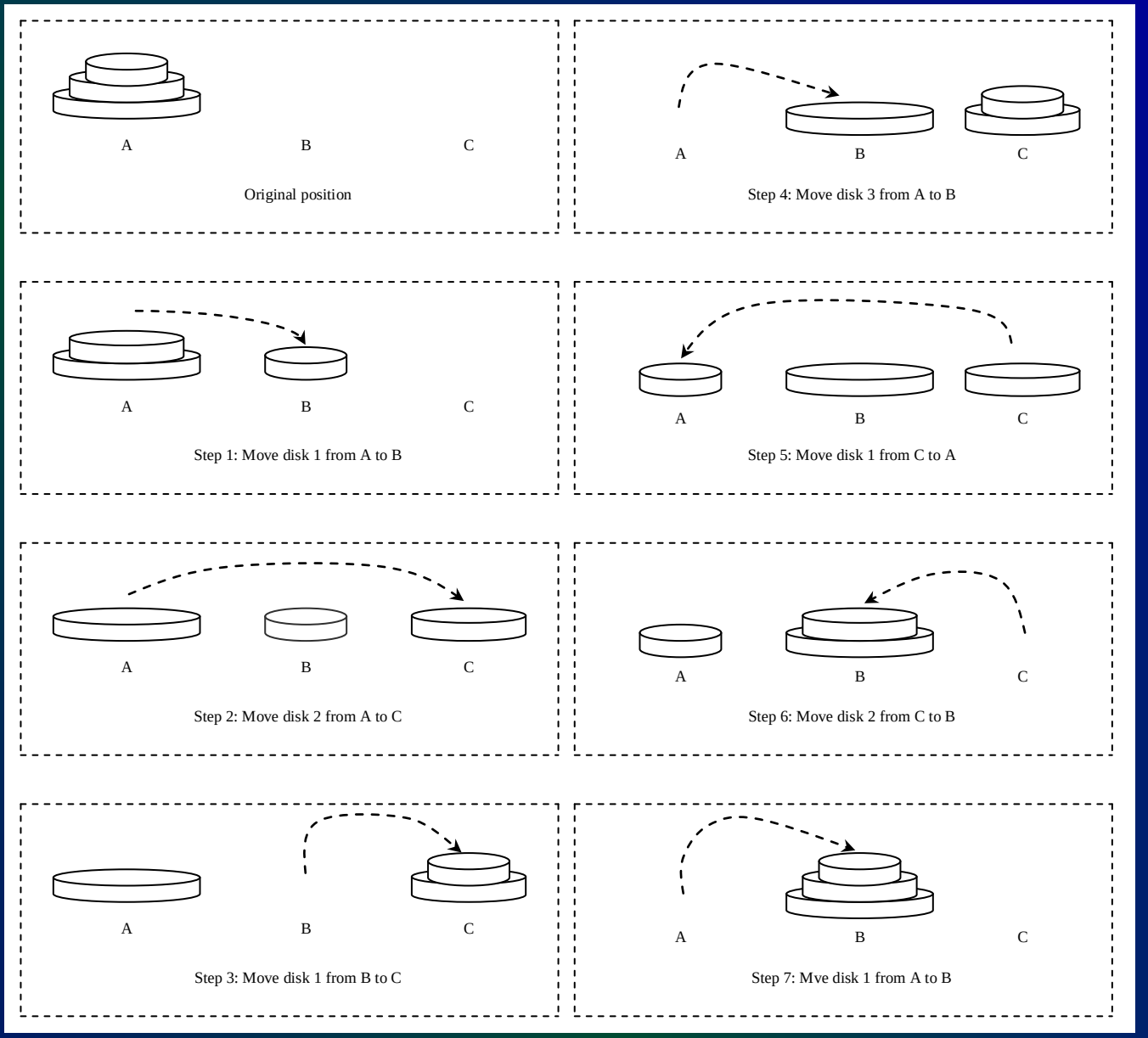
Directory Size

```
 1  import java.io.File;
 2  import java.util.Scanner;
 3
 4  public class DirectorySize {
 5    public static void main(String[] args) {
 6      // Prompt the user to enter a directory or a file
 7      System.out.print("Enter a directory or a file: ");
 8      Scanner input = new Scanner(System.in);
 9      String directory = input.nextLine();
10
11      // Display the size
12      System.out.println(getSize(new File(directory)) + " bytes");
13    }
14
15    public static long getSize(File file) {
16      long size = 0; // Store the total size of all files
17
18      if (file.isDirectory()) {
19        File[] files = file.listFiles(); // All files and subdirectories
20        for (int i = 0; files != null && i < files.length; i++) {
21          size += getSize(files[i]); // Recursive call
22        }
23      }
24      else { // Base case
25        size += file.length();
26      }
27
28      return size;
29    }
30  }
```

Towers of Hanoi

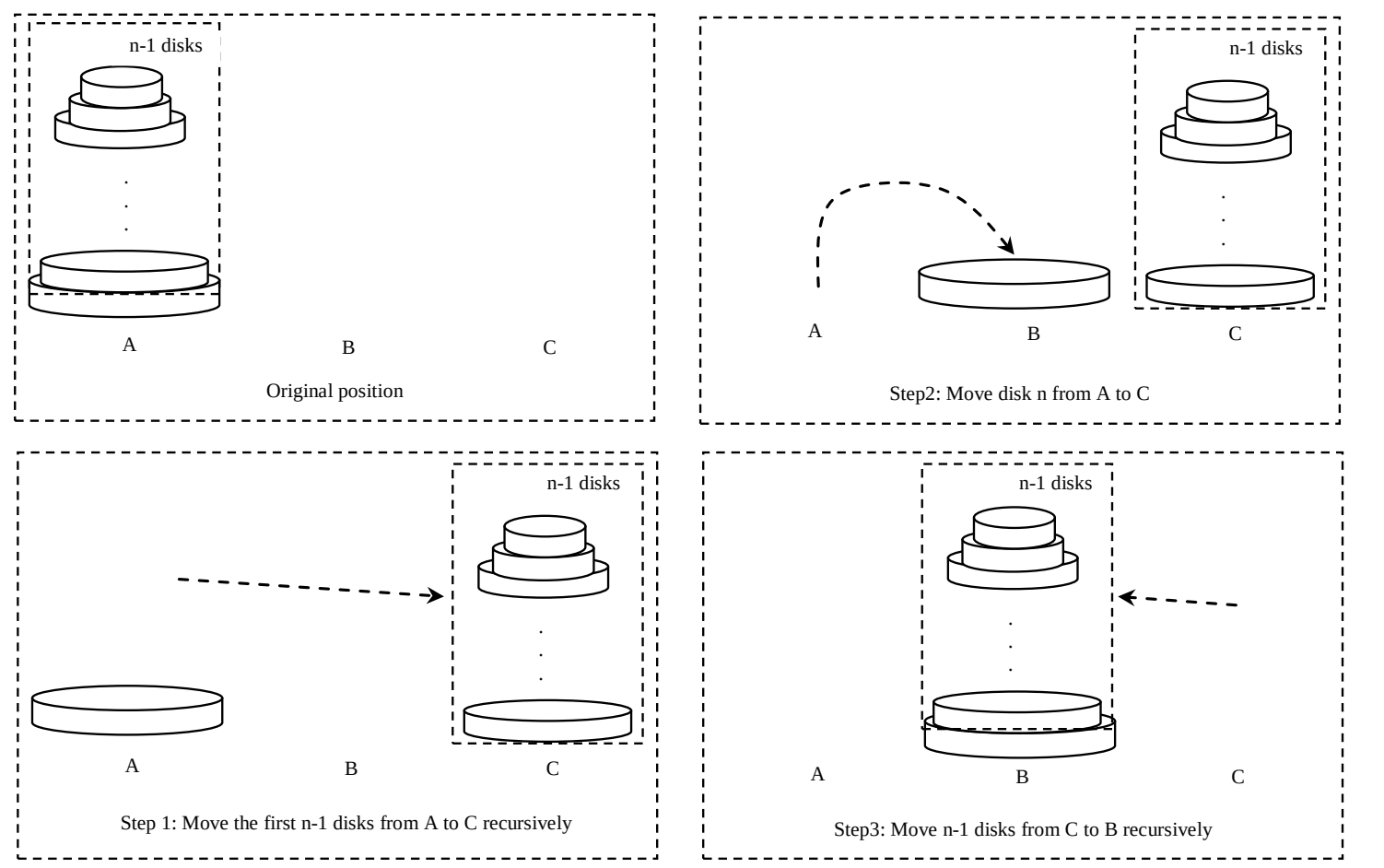
- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Towers of Hanoi, cont.



Solution to Towers of Hanoi

The Towers of Hanoi problem can be decomposed into three subproblems.



Solution to Towers of Hanoi

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

Solution to Towers of Hanoi

```
 1  import java.util.Scanner;
 2
 3  public class TowersOfHanoi {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8      System.out.print("Enter number of disks: ");
 9      int n = input.nextInt();
10
11      // Find the solution recursively
12      System.out.println("The moves are:");
13      moveDisks(n, 'A', 'B', 'C');
14    }
15
16    /** The method for finding the solution to move n disks
17        from fromTower to toTower with auxTower */
18    public static void moveDisks(int n, char fromTower,
19        char toTower, char auxTower) {
20      if (n == 1) // Stopping condition
21        System.out.println("Move disk " + n + " from " +
22          fromTower + " to " + toTower);
23      else {
24        moveDisks(n - 1, fromTower, auxTower, toTower);
25        System.out.println("Move disk " + n + " from " +
26          fromTower + " to " + toTower);
27        moveDisks(n - 1, auxTower, toTower, fromTower);
28      }
29    }
30  }
```

GCD Exercise

In mathematics, the greatest common divisor (gcd) of two or more integers, when at least one of them is not zero, is the largest positive integer that divides both numbers without a remainder.

For Example:

gcd(2, 3) = 1

gcd(2, 10) = 2

gcd(25, 35) = 5

GCD Solution

gcd(m, n)

Approach 1: Brute-force, start from 1 to min(n, m), to check if a number is common divisor for both m and n, if so, it is the greatest common divisor.

Approach 2: `Euclid's` algorithm

Approach 3: Recursive method

Approach 1: Brute Force

```
/// Return the greatest common divisor of positive numbers.
public static int GreatestCommonDivisor (int a, int b)
{
   int n = Math.Min (a, b);
   int gcd = 1, i = 1;

   while (i <= n) {
      if (a % i == 0 && b % i == 0) {
         gcd = i;
      }
      i++;
   }
   return gcd;
}
```

Approach 2: Euclid's algorithm

```
// Get absolute value of m and n;
t1 = Math.abs(m); t2 = Math.abs(n);
// r is the remainder of t1 divided by t2;
r = t1 % t2;
while (r != 0) {
  t1 = t2;
  t2 = r;
  r = t1 % t2;
}

// When r is 0, t2 is the greatest common
// divisor between t1 and t2
return t2;
```

Approach 3: `Recursive Method`

(Programming Challenge)

gcd(m, n) = n if m % n = 0; // base case

gcd(m, n) = gcd(n, m % n); otherwise;

Recursion vs. Iteration

- Any problem that can be solved recursively can be solved non recursively with iterations.
- Recursion has some negative aspects:
  - It uses up too much time and too much memory.
  - 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.
- Why, then, should you use it? In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain.

Programming Challenge

(*Sum series*) Write a recursive method to compute the following series:

$$m(i) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

Write a test program that displays `m(i)` for `i` = `1`, `2`, . . . , `10`.

Programming Challenge

(*Print the characters in a string reversely*) Write a recursive method that displays a string reversely on the console using the following header:

```
public static void reverseDisplay(String value)
```

For example, `reverseDisplay("abcd")` displays `dcba`. Write a test program that prompts the user to enter a string and displays its reversal.

Programming Challenge

(*Occurrences of a specified character in a string*) Write a recursive method that finds the number of occurrences of a specified letter in a string using the following method header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string and a character, and displays the number of occurrences for the character in the string.