

# Recursion

CSE 114: Introduction to Object-Oriented Programming

Paul Fodor

Stony Brook University

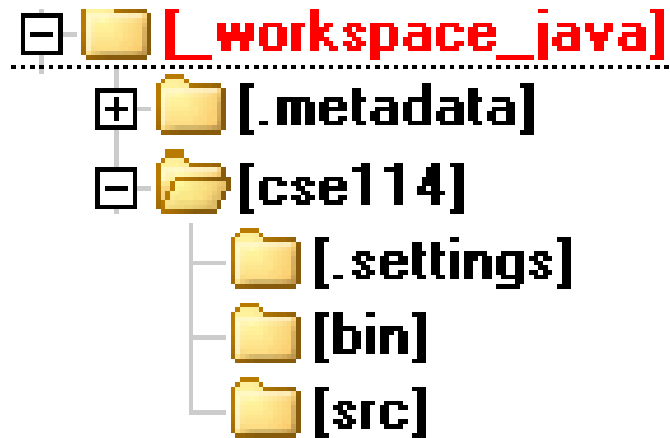
<http://www.cs.stonybrook.edu/~cse114>

# Contents

- Motivation: Recursive Data Structures
- Motivation: Recursive Algorithms
- What is recursion?
- Computing Factorial
- Stack Trace
- Fibonacci Numbers
- Characteristics of Recursion
- Problem Solving Using Recursion
- Recursive Helper Methods
- Recursive Selection Sort
- Recursive Binary Search
- Directory Size
- Towers of Hanoi
- Greatest Common Divisor (GCD)
- From Iteration to Recursion

# Motivation: Recursive Data Structures

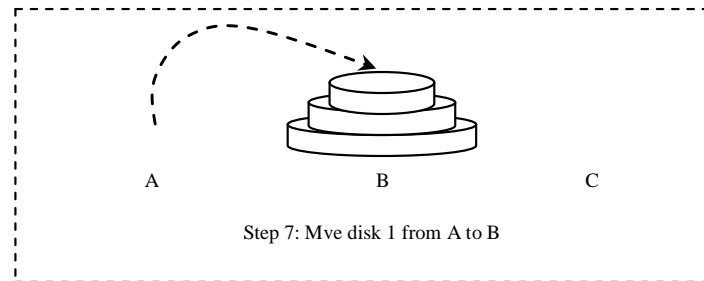
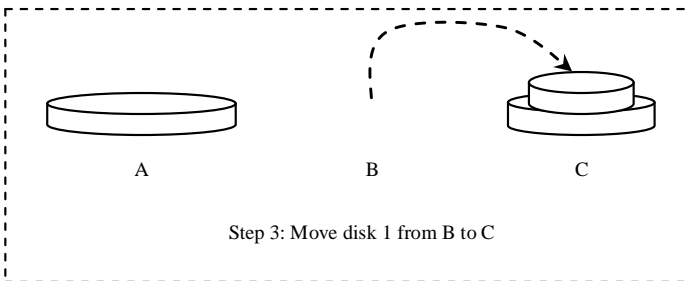
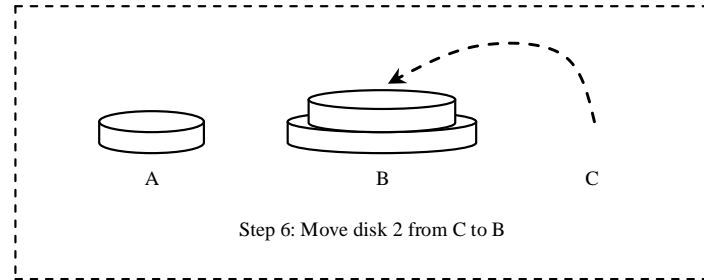
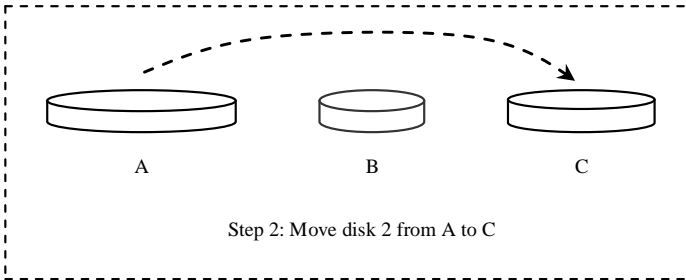
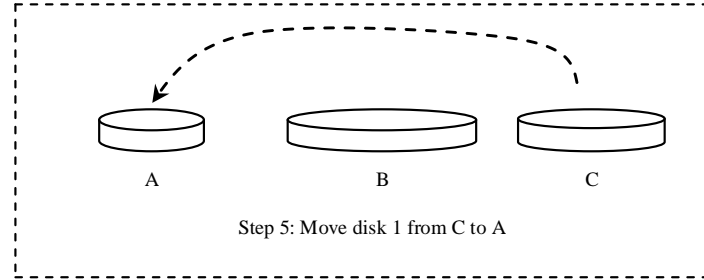
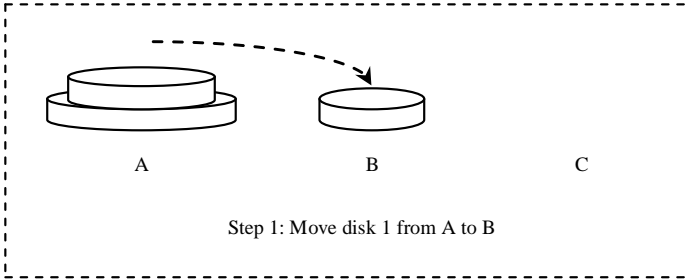
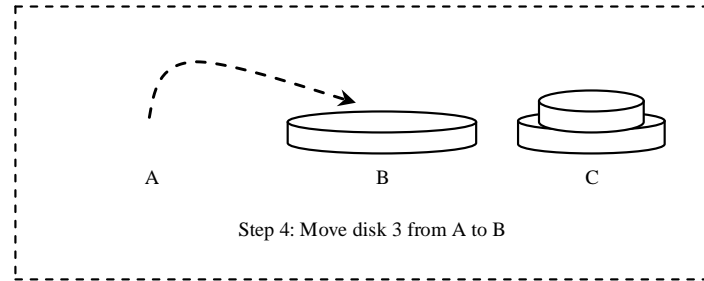
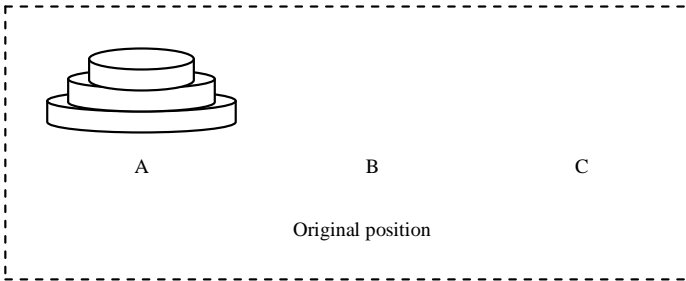
- Suppose you want to find all the files **under a directory** that contains a particular word.
  - The directory contains subdirectories that also contain subdirectories, and so on.

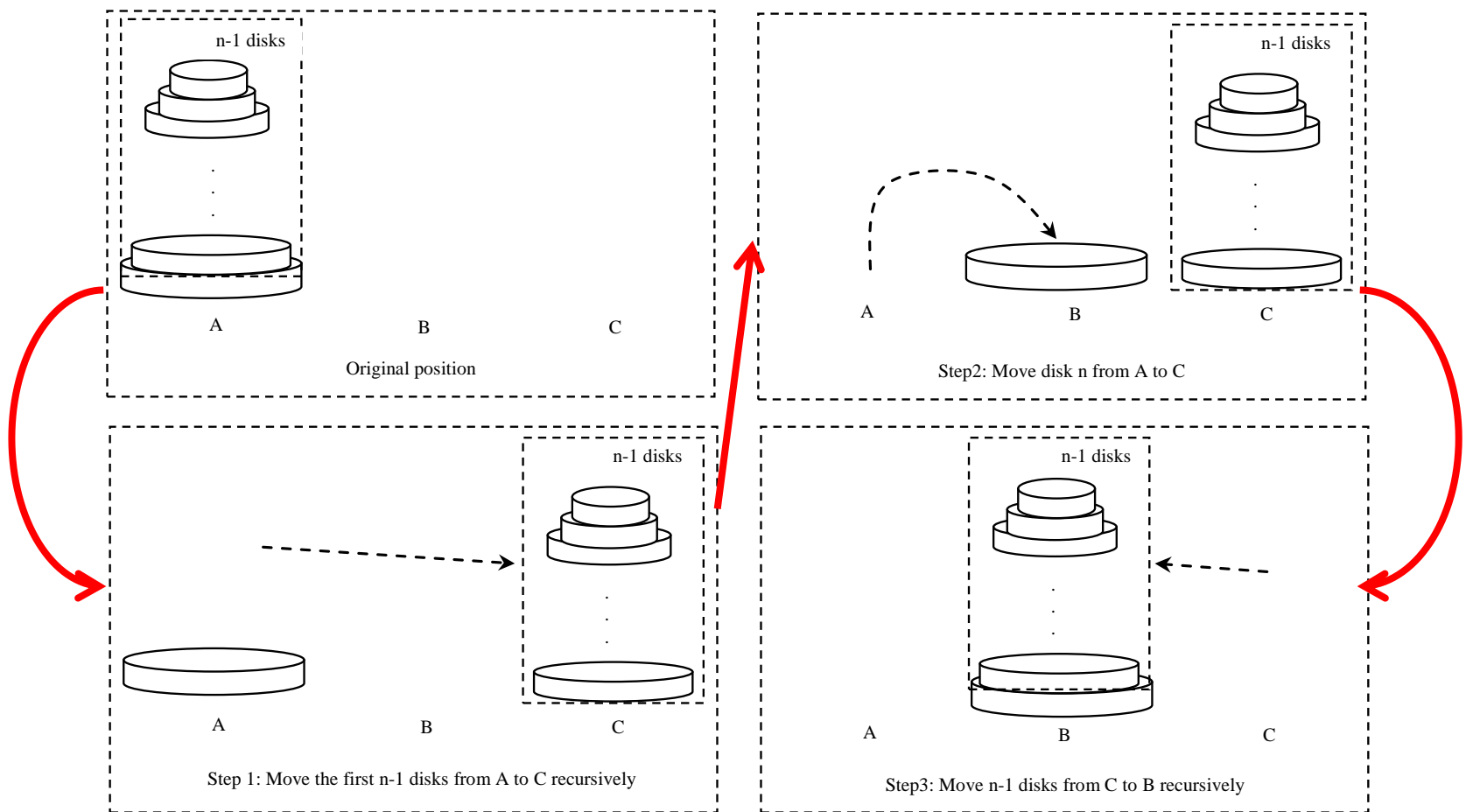


- The solution is to use recursion by searching the files in the subdirectories recursively.

# Motivation: Recursive Algorithms

- Towers of Hanoi:
  - There are  $n$  disks labeled  $1, 2, 3, \dots, n$ , 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.





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

- 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.

```

import java.util.Scanner;
public class TowersOfHanoi {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter number of disks: ");
        int n = input.nextInt(); System.out.println("The moves are:");
        moveDisks(n, 'A', 'B', 'C');
    }
    public static void moveDisks(int n, char fromTower, char toTower,
        char auxTower) {
        if (n == 1) // Stopping condition
            System.out.println("Move disk " + n + " from " +
                fromTower + " to " + toTower);
        else {
            moveDisks(n - 1, fromTower, auxTower, toTower);
            System.out.println("Move disk " + n + " from " +
                fromTower + " to " + toTower);
            moveDisks(n - 1, auxTower, toTower, fromTower);
        }
    }
}

```

# What is recursion?

- Recursion is the use of recursive methods—methods that invoke themselves.
- Recursion is highly useful when your problem has a recursive structure or formula. Examples:
  - Folders: each folder has children sub-folders (like itself and its parent folder)
  - Data structures such as linked lists, trees and graphs (you will see in CSE214 that every node has its value and other node references)



# Computing Factorial

$$n! = 1 * 2 * 3 * 4 * 5 * \dots * (n-1) * n$$

$$(n-1)! = 1 * 2 * 3 * 4 * 5 * \dots * (n-1)$$

So:

$$n! = n * (n-1)!$$

Therefore, the recursive function is:

$$\text{factorial}(0) = 1;$$

$$\text{factorial}(n) = n * \text{factorial}(n-1), \text{ for } n > 0$$

```

import java.util.Scanner;
public class ComputeFactorial {
    public static void main(String[] args) {
        // Create a Scanner
        Scanner input = new Scanner(System.in);
        System.out.print("Enter a non-negative integer: ");
        int n = input.nextInt();
        // Display factorial
        System.out.println("Factorial of "+n+" is "+factorial(n));
    }
    /** Return the factorial for a specified number */
    public static int factorial(int n) {
        if (n == 0) // Base case
            return 1;
        else
            return n * factorial(n - 1); // Recursive call
    }
}

```

# Computing Factorial

factorial(3) =

factorial(0) = 1;

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

# Computing Factorial

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$\text{factorial}(0) = 1;$$

$$\text{factorial}(n) = n * \text{factorial}(n-1);$$

# Computing Factorial

$$\begin{aligned}\text{factorial}(3) &= 3 * \text{factorial}(2) \\ &= 3 * (2 * \text{factorial}(1))\end{aligned}$$

$$\begin{aligned}\text{factorial}(0) &= 1; \\ \text{factorial}(n) &= n * \text{factorial}(n-1);\end{aligned}$$

# Computing Factorial

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$= 3 * (2 * \text{factorial}(1))$$

$$= 3 * (2 * (1 * \text{factorial}(0)))$$

$$\text{factorial}(0) = 1;$$

$$\text{factorial}(n) = n * \text{factorial}(n-1);$$

# Computing Factorial

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$= 3 * (2 * \text{factorial}(1))$$

$$= 3 * (2 * (1 * \text{factorial}(0)))$$

$$= 3 * (2 * (1 * 1))$$

$$\text{factorial}(0) = 1;$$

$$\text{factorial}(n) = n * \text{factorial}(n-1);$$

# Computing Factorial

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$= 3 * (2 * \text{factorial}(1))$$

$$= 3 * (2 * (1 * \text{factorial}(0)))$$

$$= 3 * (2 * (1 * 1))$$

$$= 3 * (2 * 1)$$

$$\text{factorial}(0) = 1;$$

$$\text{factorial}(n) = n * \text{factorial}(n-1);$$



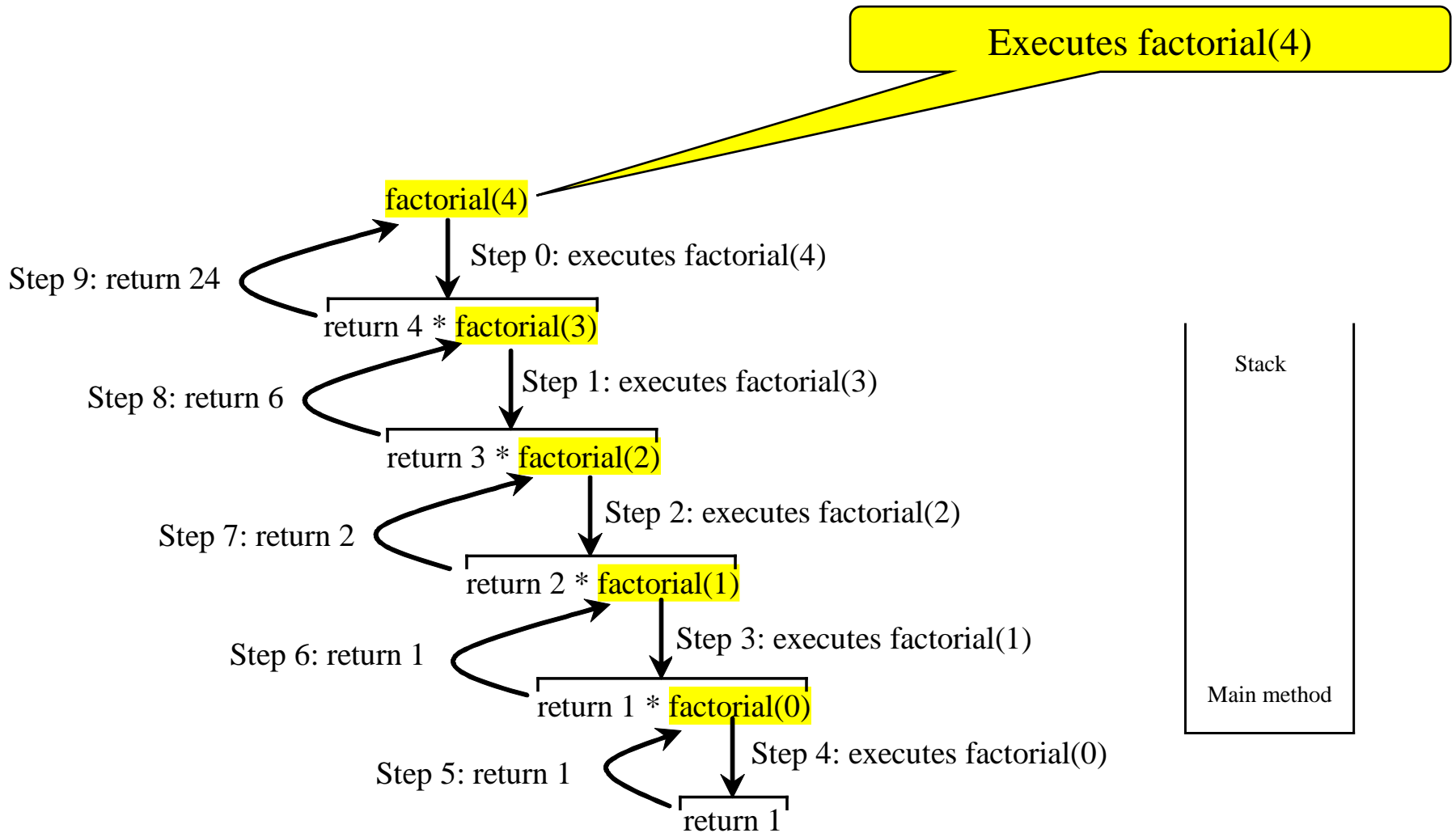
# Computing Factorial

$$\begin{aligned}\text{factorial}(3) &= 3 * \text{factorial}(2) & \text{factorial}(0) &= 1; \\ &= 3 * (2 * \text{factorial}(1)) & \text{factorial}(n) &= n * \text{factorial}(n-1); \\ &= 3 * (2 * (1 * \text{factorial}(0))) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2\end{aligned}$$

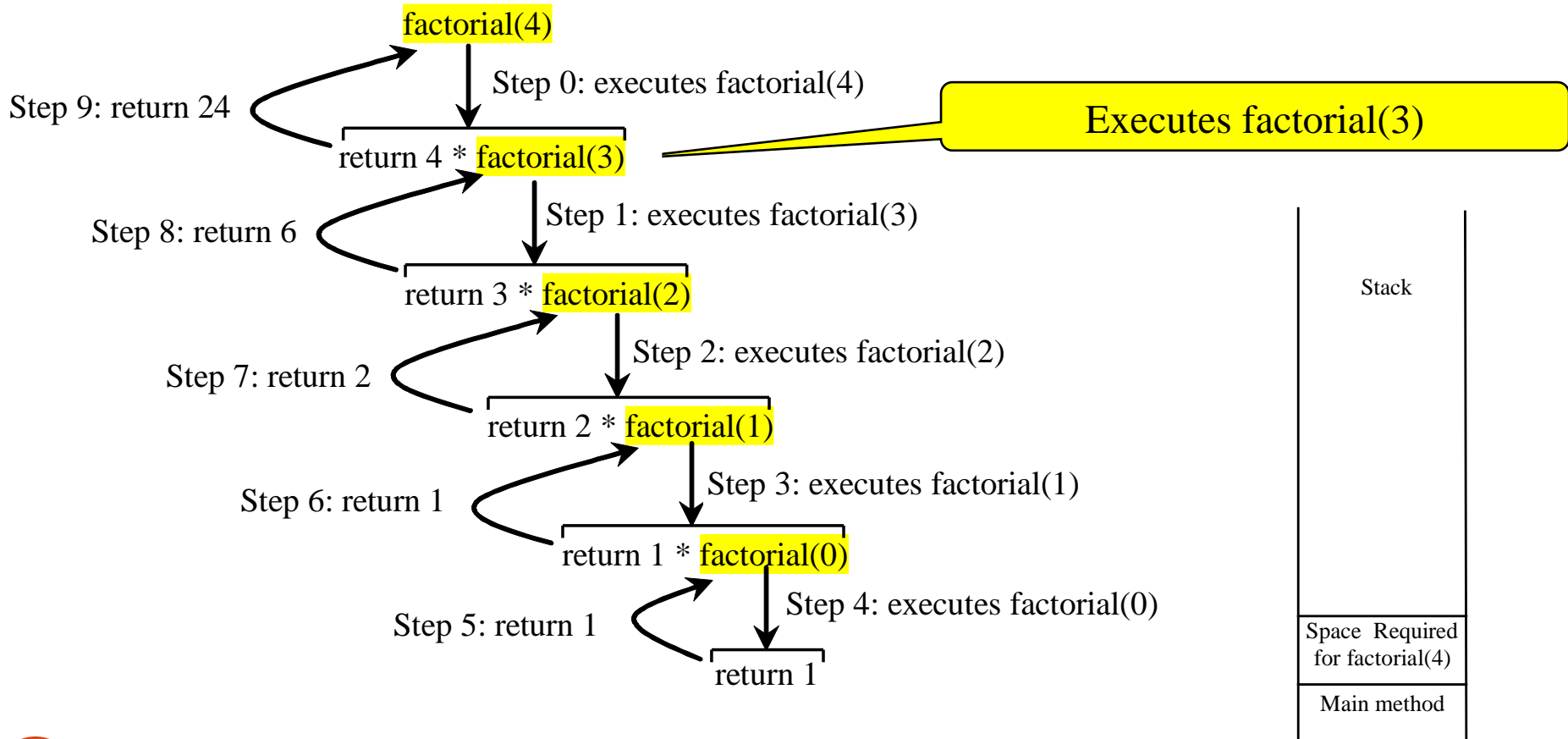
# Computing Factorial

$$\begin{aligned} \text{factorial}(3) &= 3 * \text{factorial}(2) & \text{factorial}(0) &= 1; \\ &= 3 * (2 * \text{factorial}(1)) & \text{factorial}(n) &= n * \text{factorial}(n-1); \\ &= 3 * (2 * (1 * \text{factorial}(0))) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

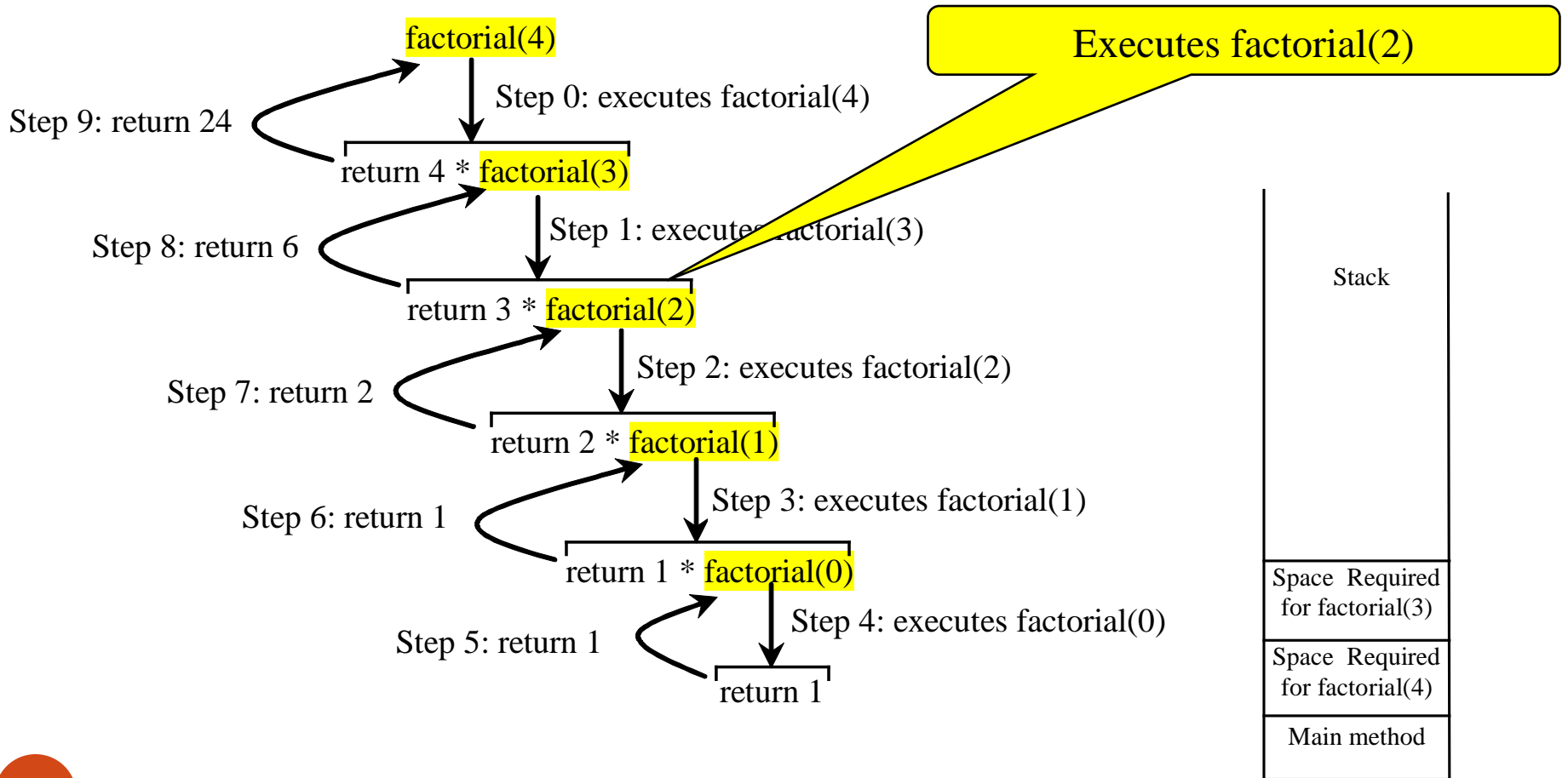
# Trace Recursive factorial



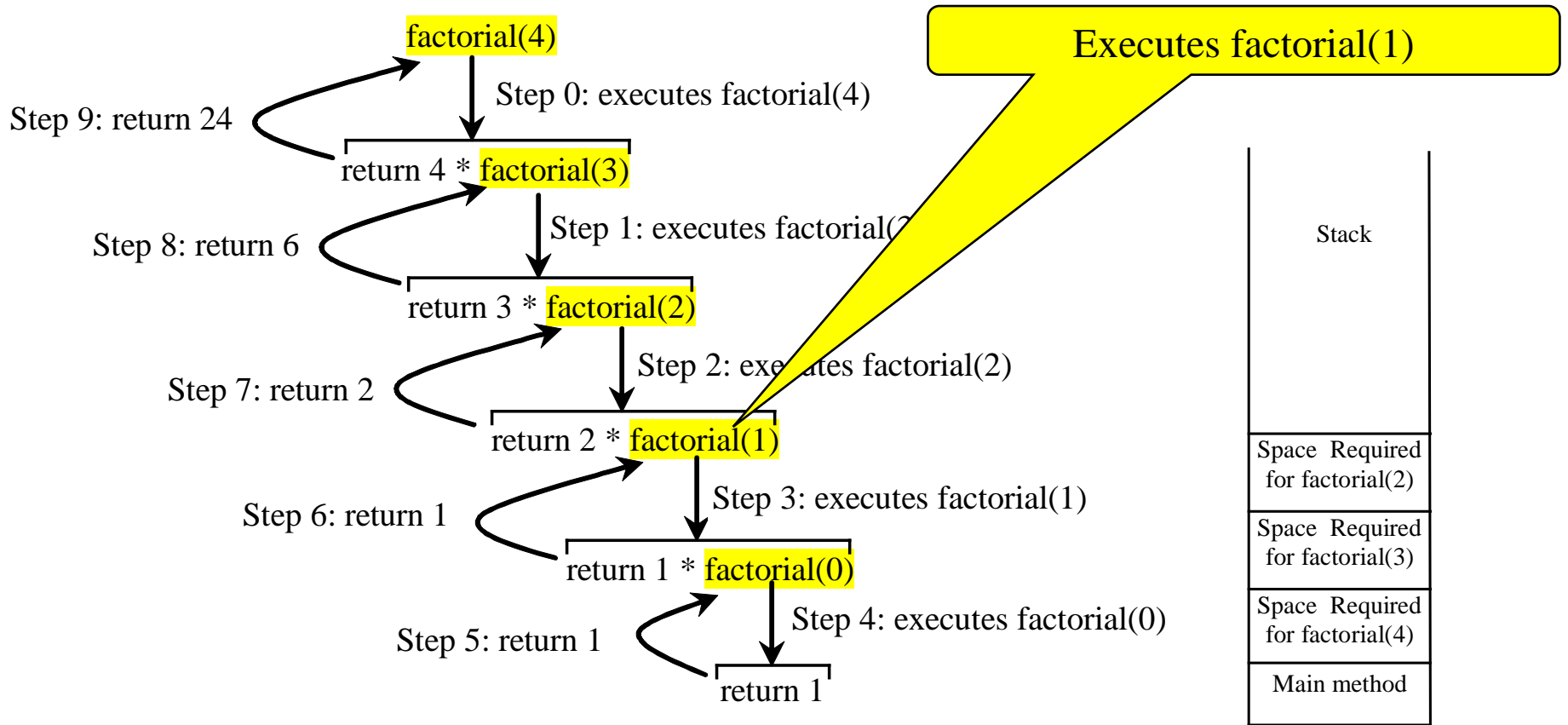
# Trace Recursive factorial



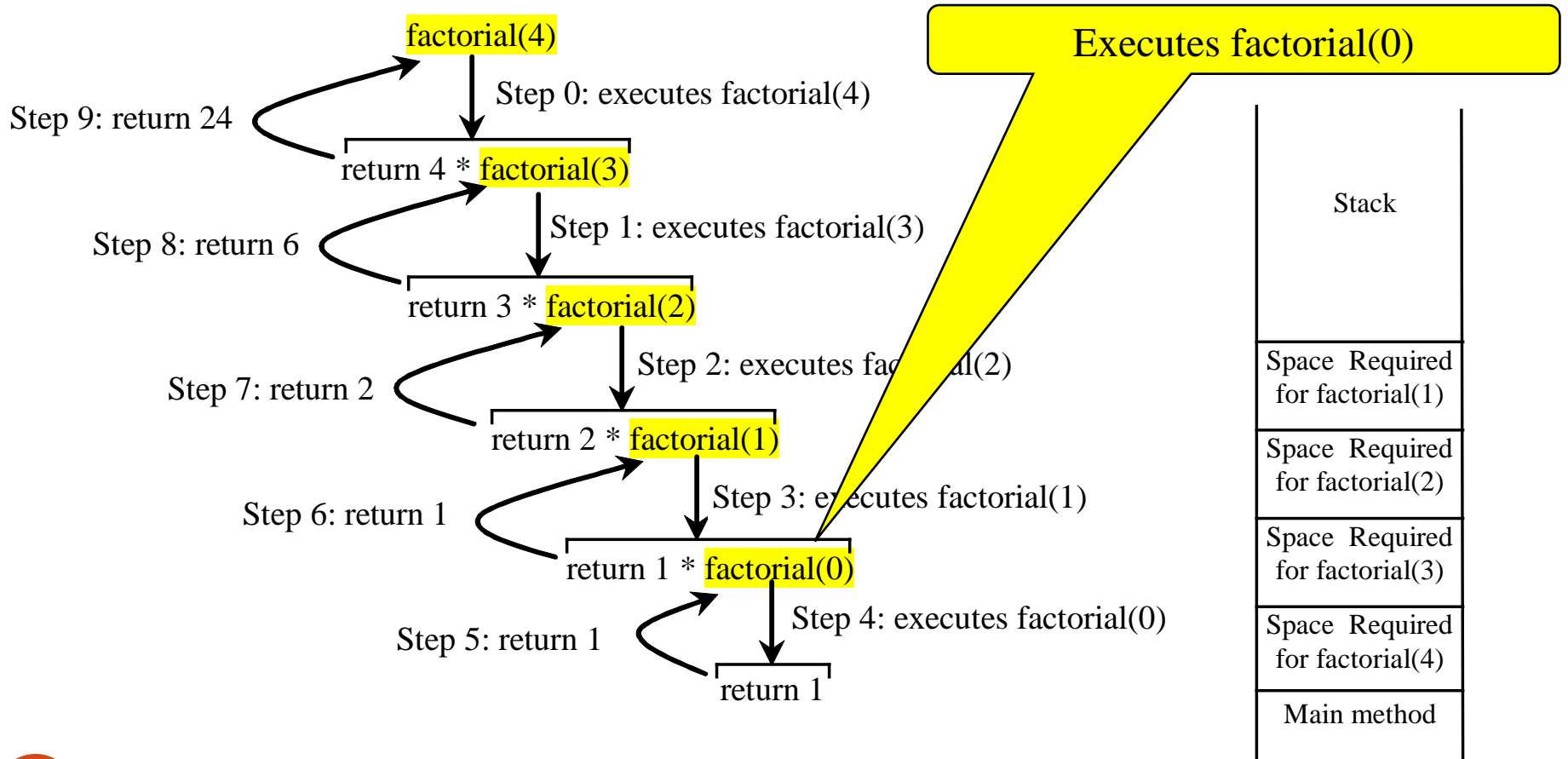
# Trace Recursive factorial



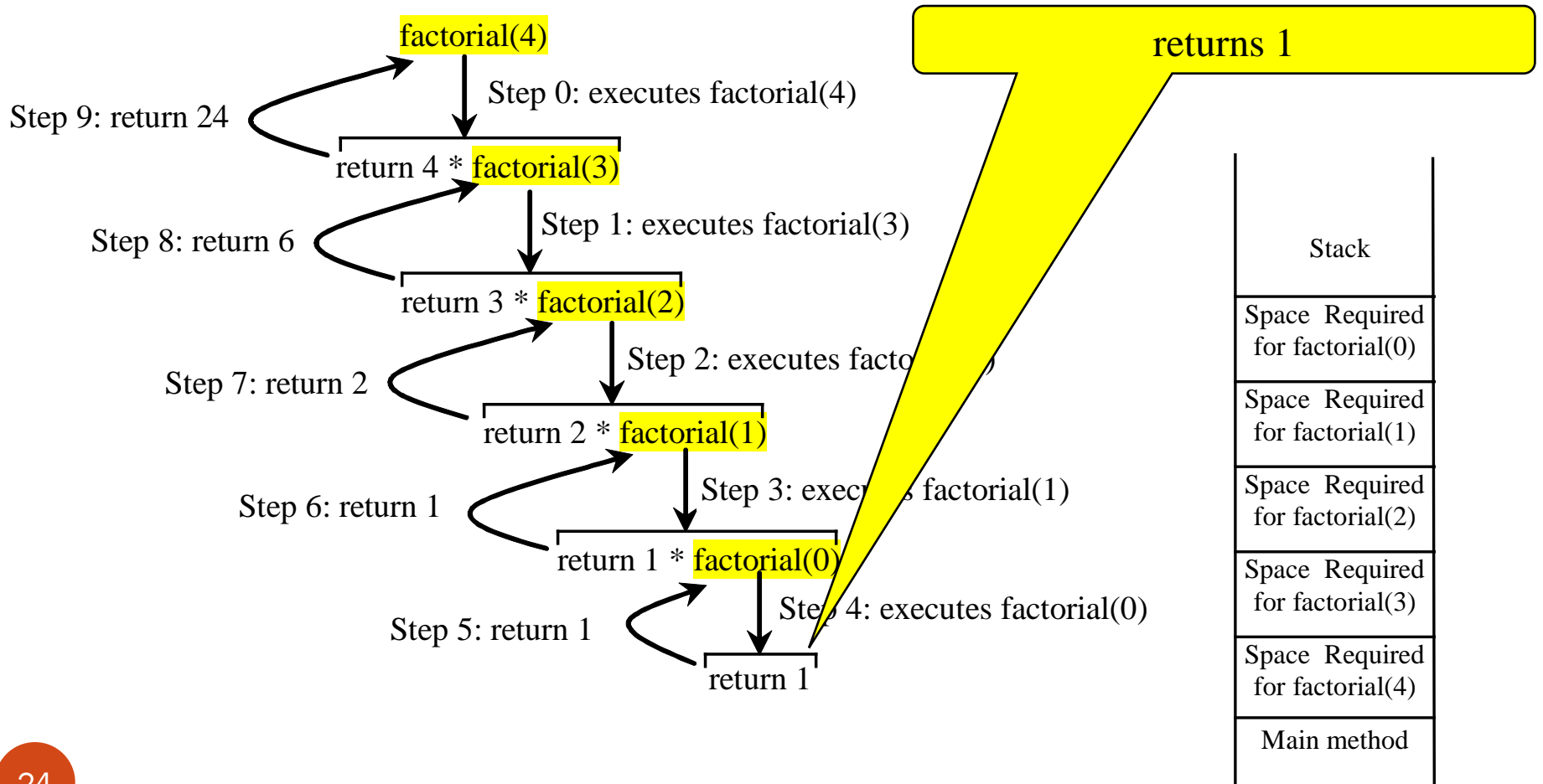
# Trace Recursive factorial



# Trace Recursive factorial

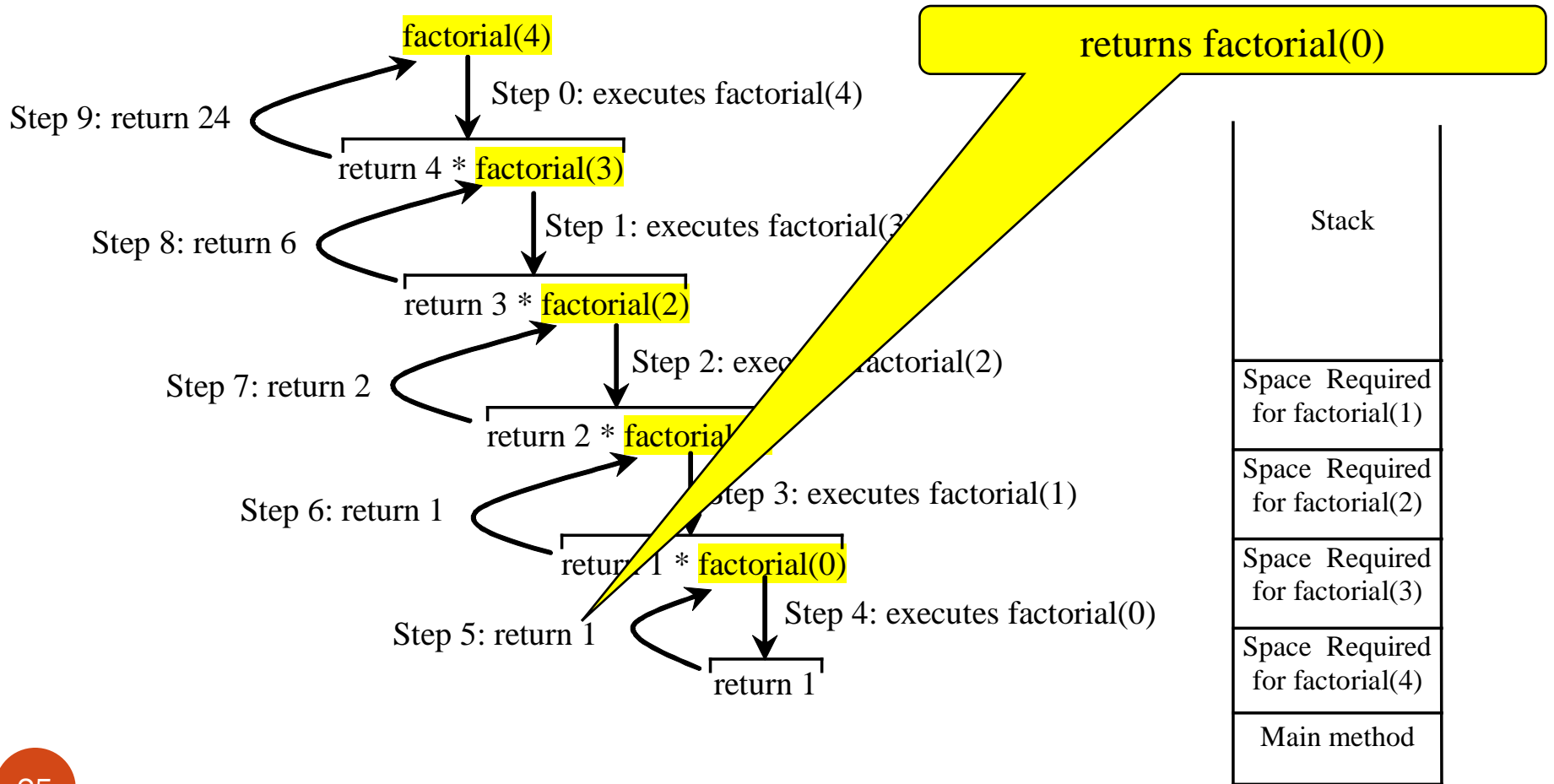


# Trace Recursive factorial

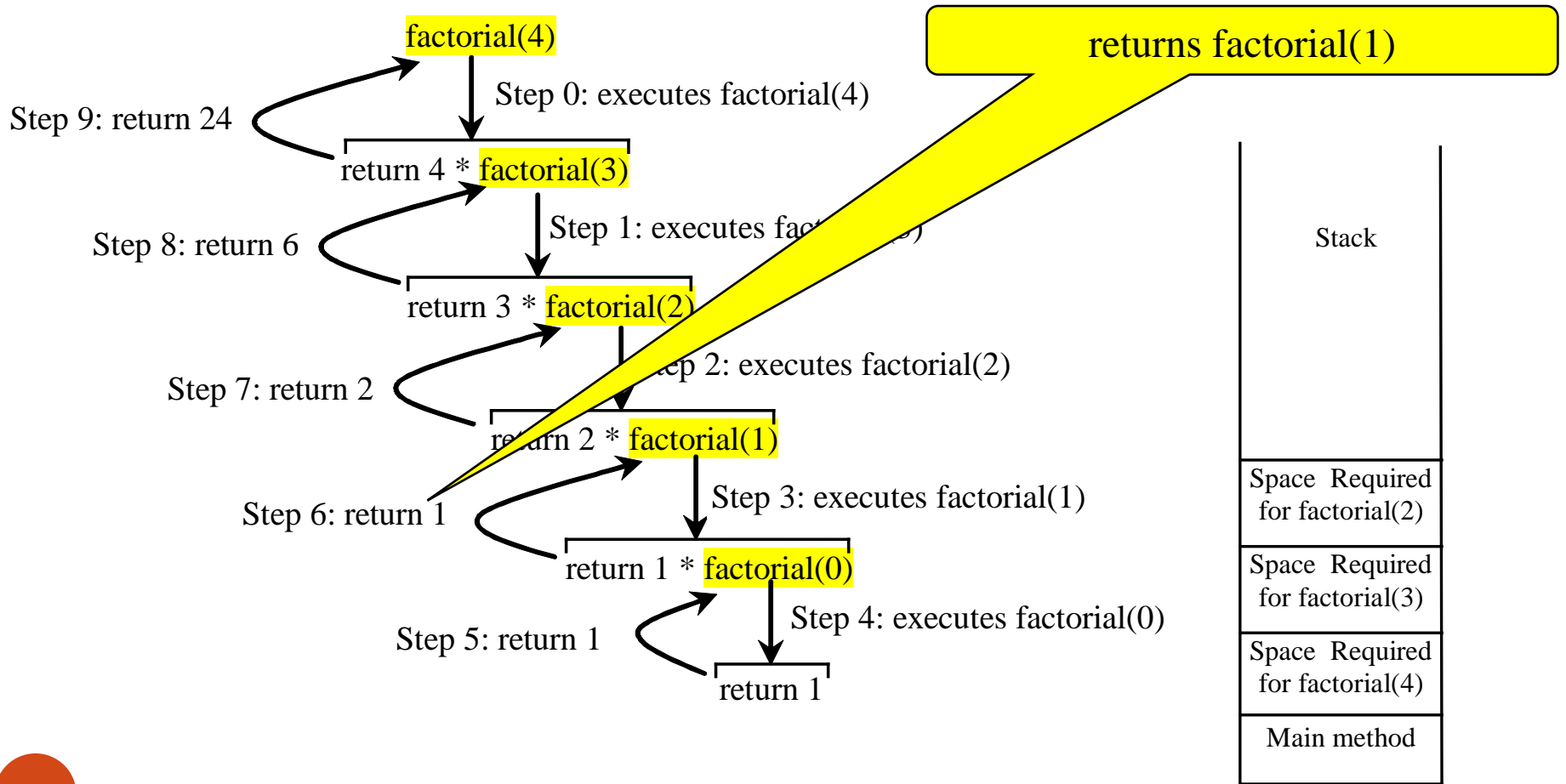




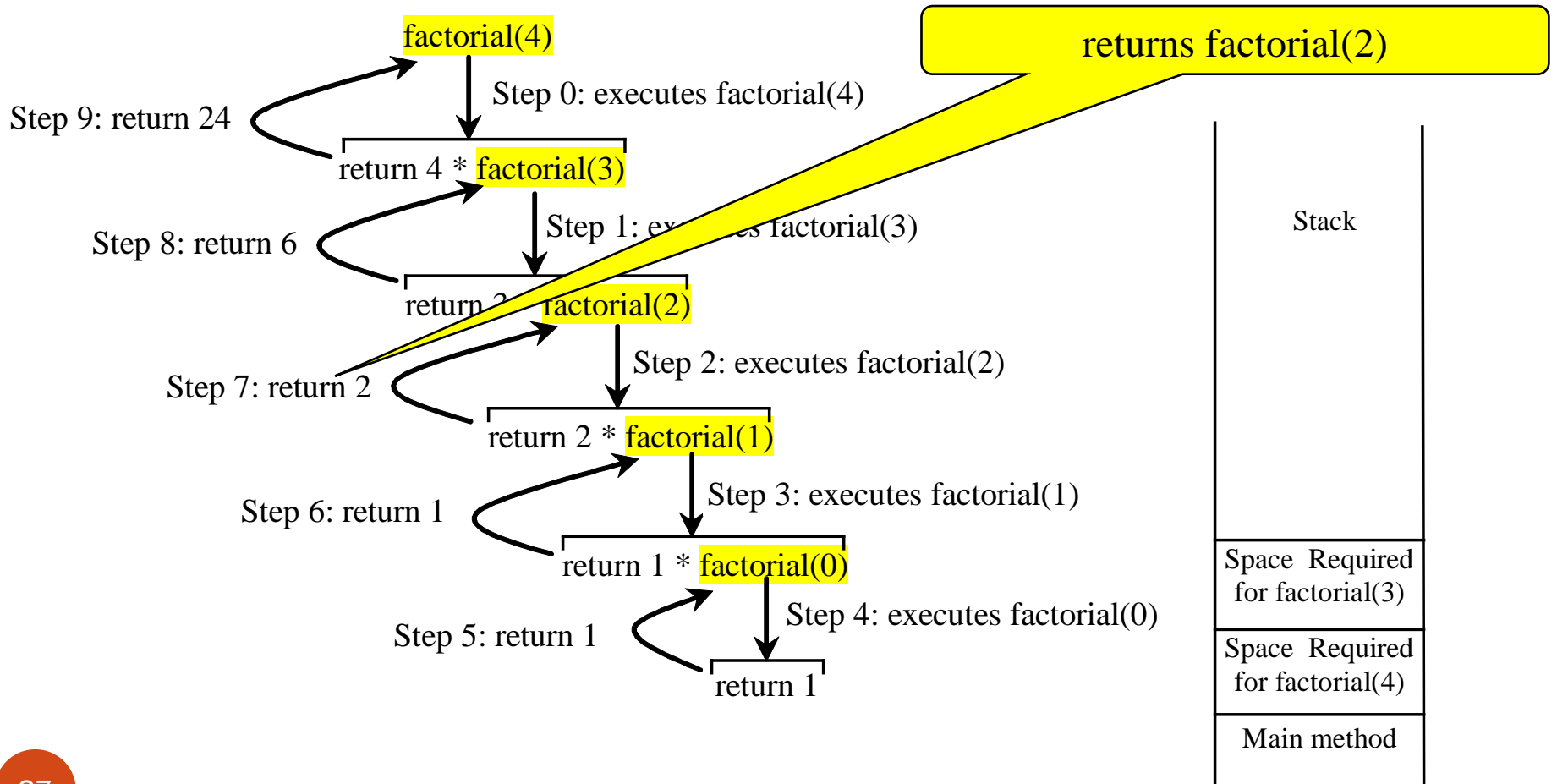
# Trace Recursive factorial



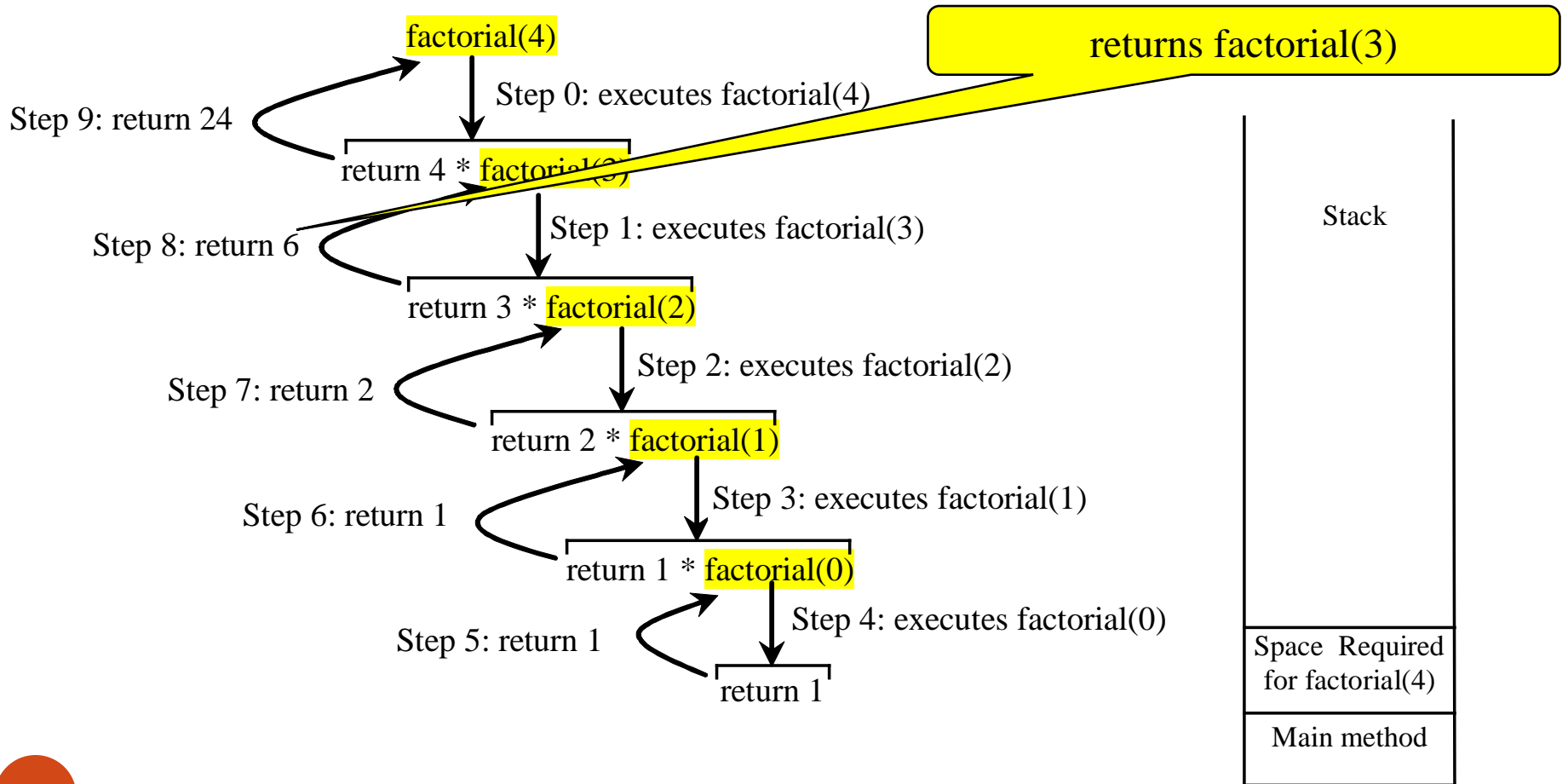
# Trace Recursive factorial



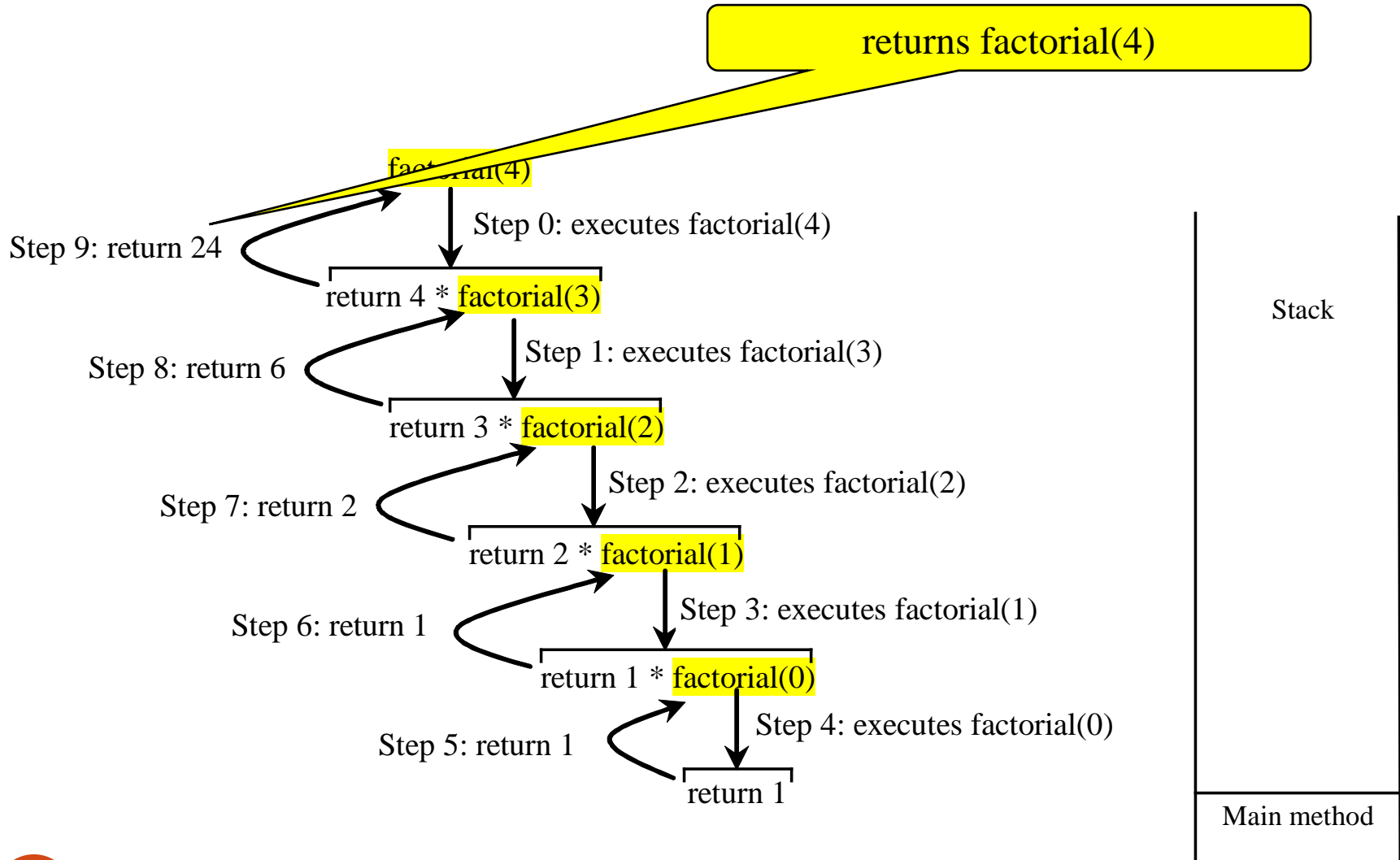
# Trace Recursive factorial



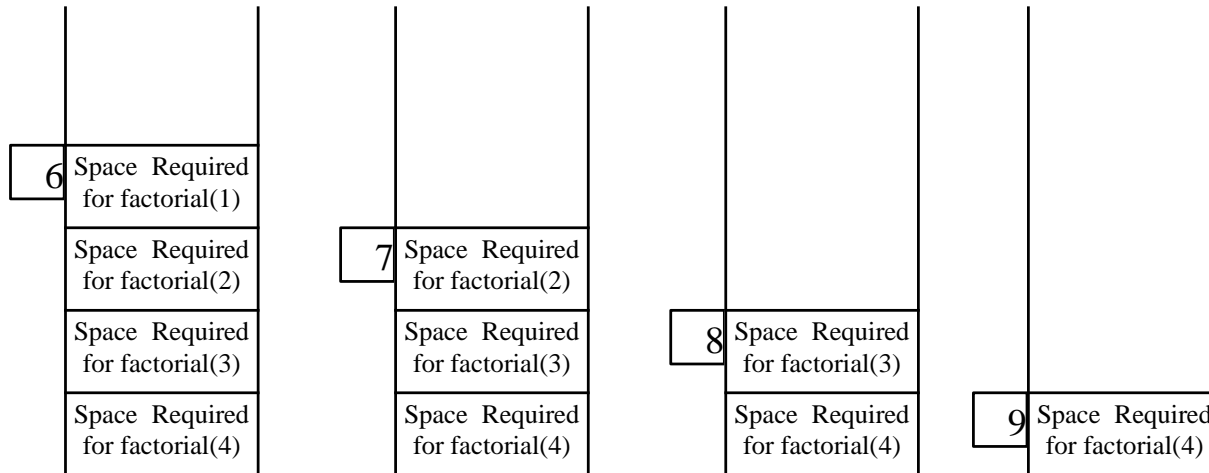
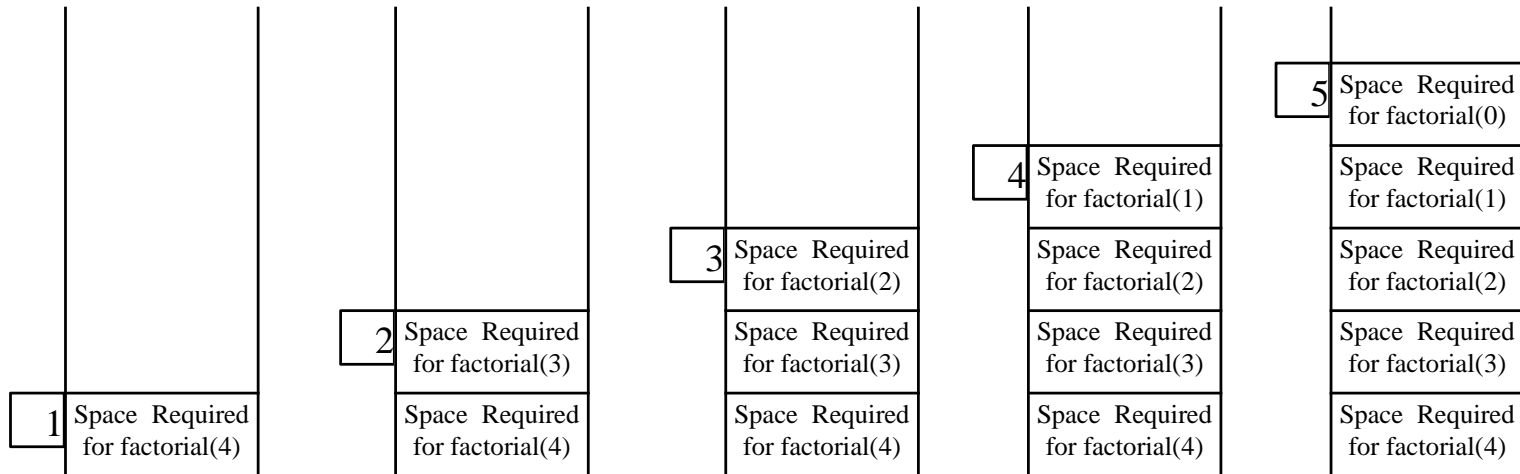
# Trace Recursive factorial



# Trace Recursive factorial



# factorial(4) Stack Trace



# Fibonacci Numbers

indices:	0	1	2	3	4	5	6	7	8	9	10	11	...
Fibonacci series:	0	1	1	2	3	5	8	13	21	34	55	89	...

$$\text{fib}(0) = 0;$$

$$\text{fib}(1) = 1;$$

$$\text{fib}(\text{index}) = \text{fib}(\text{index} - 1) + \text{fib}(\text{index} - 2); \quad \text{for integers index } \geq 2$$

$$\begin{aligned} \text{fib}(3) &= \text{fib}(2) + \text{fib}(1) = (\text{fib}(1) + \text{fib}(0)) + \text{fib}(1) \\ &= (1 + 0) + \text{fib}(1) = 1 + \text{fib}(1) = 1 + 1 = 2 \end{aligned}$$

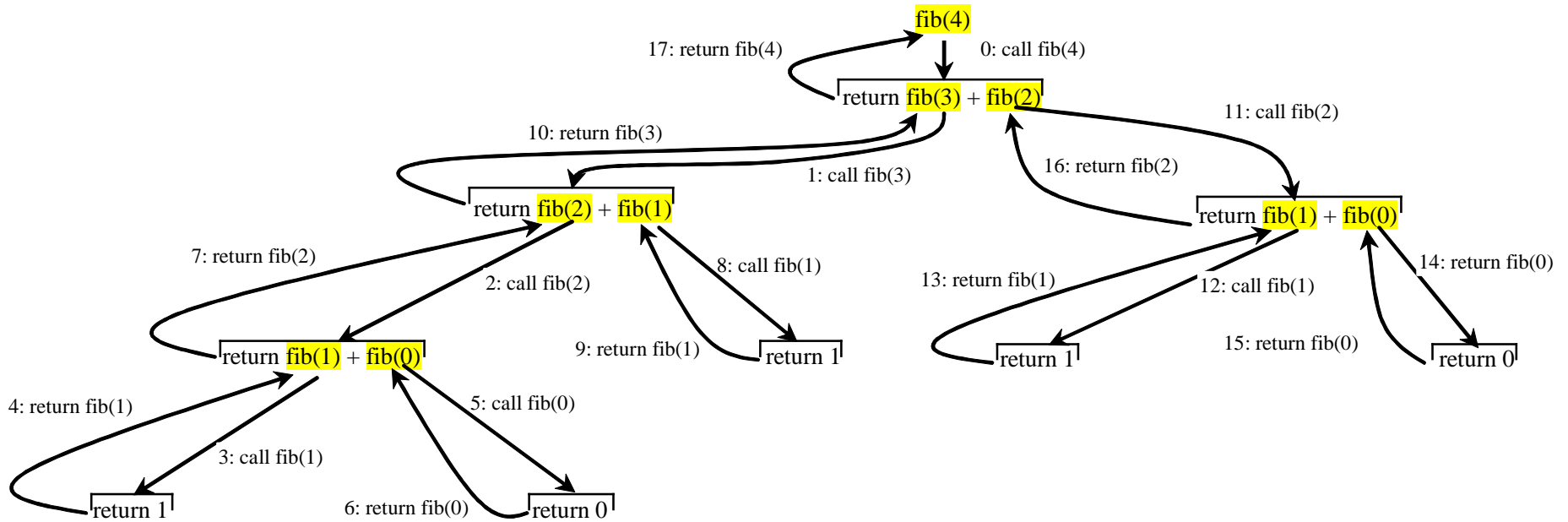
```

import java.util.Scanner;
public class ComputeFibonacci {
public static void main(String args[]) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for the Fibonacci number: ");
    int index = input.nextInt();
    // Find and display the Fibonacci number
    System.out.println("Fibonacci(" + index + ") is " + fib(index));
}
/** The method for finding the Fibonacci number */
public static int fib(int 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



```

import java.util.Scanner; // Dynamic programming
public class ComputeFibonacciTabling { // NO REPEATED COMPUTATION
public static void main(String args[]) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter an index for the Fibonacci number: ");
    int index = input.nextInt();
    f = new int[index+1];
    System.out.println("Fibonacci(" + index + ") is " + fib(index));
}
public static int[] f;
public static int fib(int index) {
    if (index == 0) return 0;
    if (index == 1) { f[1]=1; return 1; }
    if(f[index]!=0)
        return f[index];
    else // Reduction and recursive calls
        f[index] = fib(index - 1) + f[index - 2];
    return f[index];
}
}

```

# 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

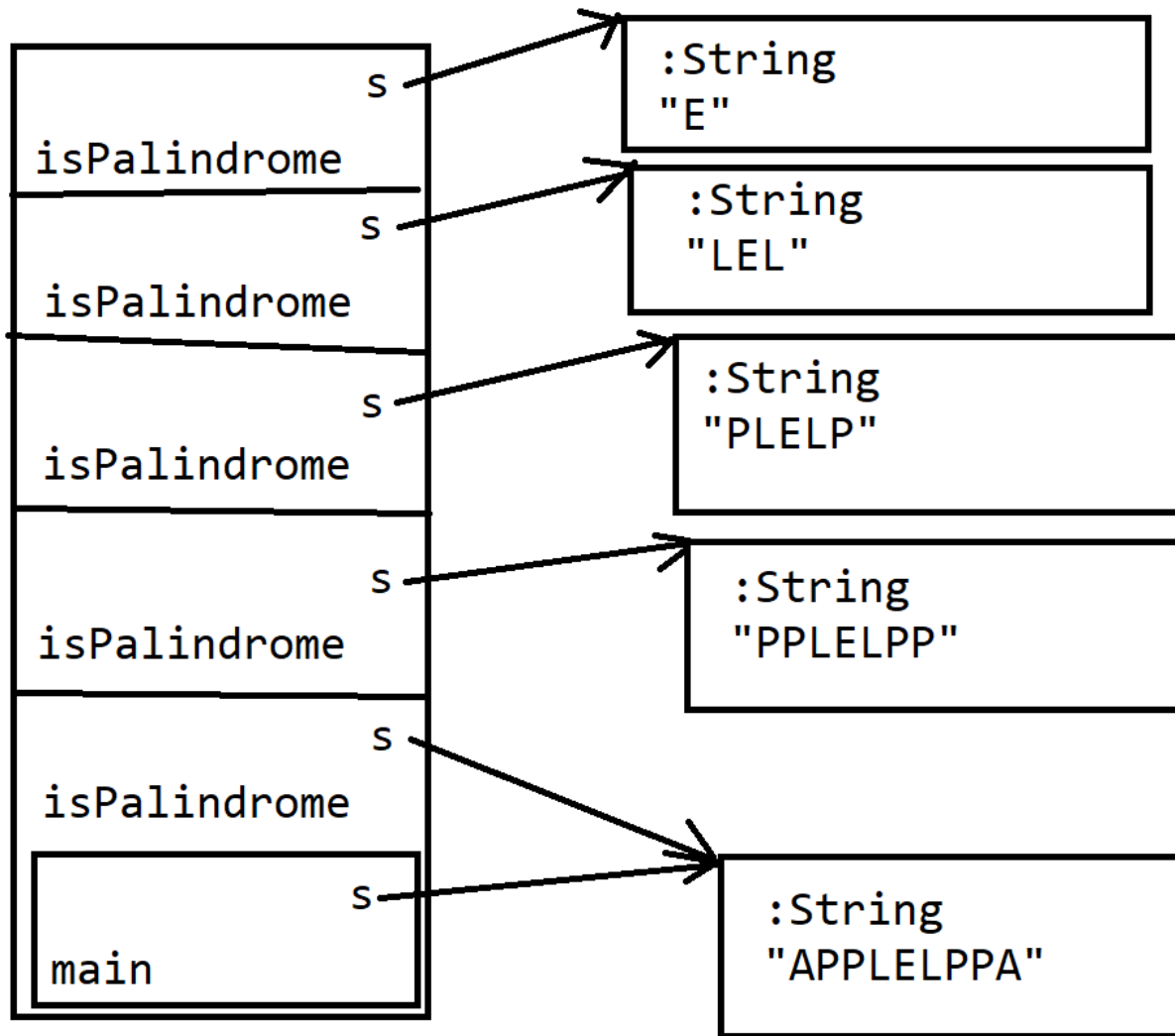
- Print a message for n times
- break the problem into two subproblems:
  - print the message one time and
  - print the message for n-1 times
    - This new problem is the same as the original problem with a smaller size.
    - The base case for the problem is n==0.

```
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

- The palindrome problem (e.g., “eye”, “racecar”):

```
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));  
}
```



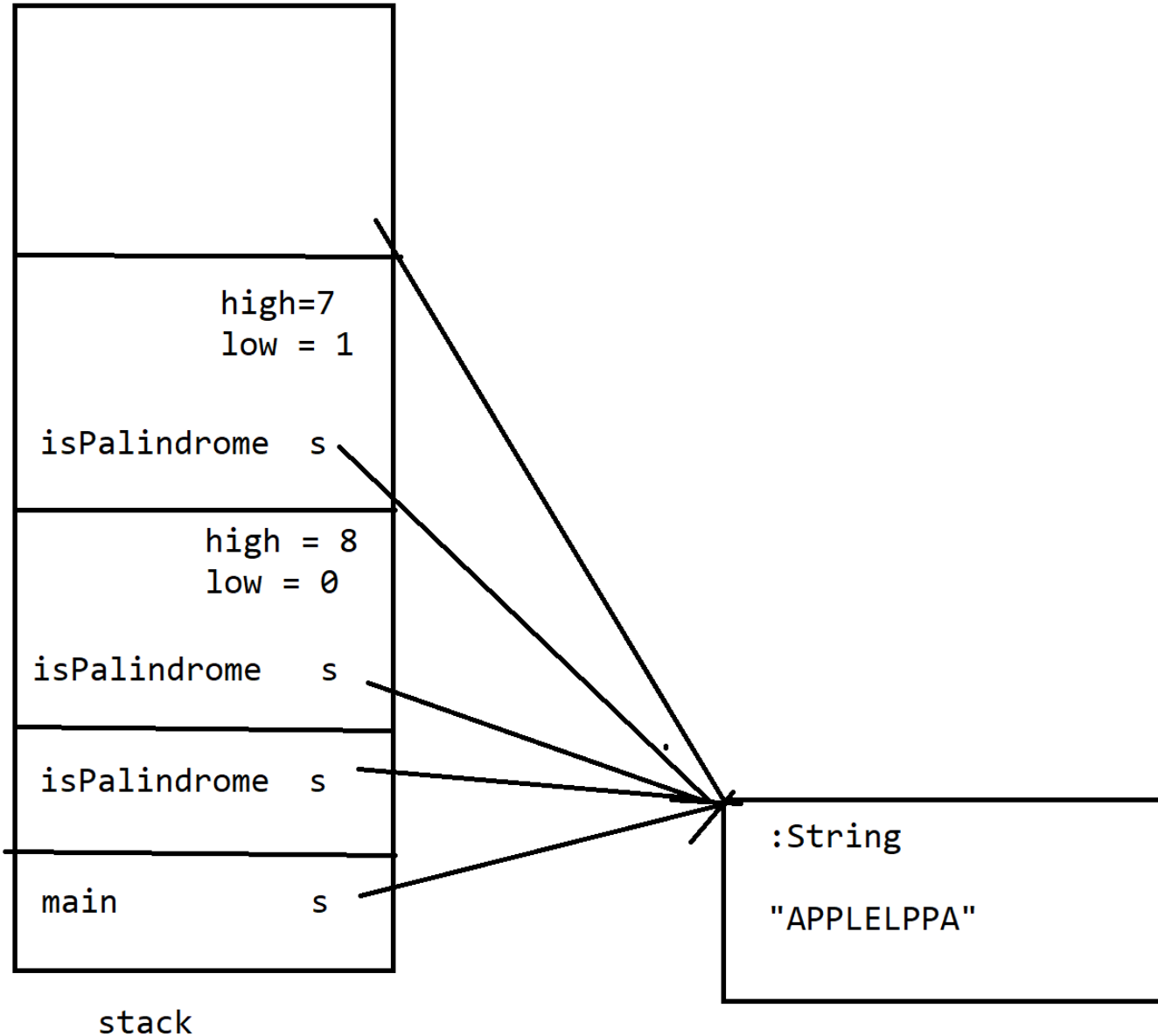
stack

# 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);
}
```





# Recursive Selection Sort

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

```

public class IterativeSelectionSort {
    public static void sort(double[] list) {
        int low = 0, high = list.length - 1;
        while (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;
            low = low + 1;
        }
    }

    public static void main(String[] args) {
        double[] list = { 2, 1, 3, 1, 2, 5, 2, -1, 0 };
        sort(list);
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
    }
}

```

```

public class RecursiveSelectionSort {
    public static void sort(double[] list) {
        sort(list, 0, list.length - 1); // Sort the entire list
    }
    public 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);
        }
    }
    public static void main(String[] args) {
        double[] list = {2, 1, 3, 1, 2, 5, 2, -1, 0};
        sort(list);
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
    }
}

```

# Recursive Binary Search

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

```

public class IterativeBinarySearch {
    public static int binarySearch(int[] list, int key) {
        int low = 0;
        int high = list.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (key < list[mid])
                high = mid - 1;
            else if (key == list[mid])
                return mid;
            else
                low = mid + 1;
        }
        // The list has been exhausted without a match
        return -low - 1;
    }
    public static void main(String[] args) {
        int[] list = { 1,2,3,4,5,6,10 };
        System.out.print(binarySearch(list,6));
    }
}

```

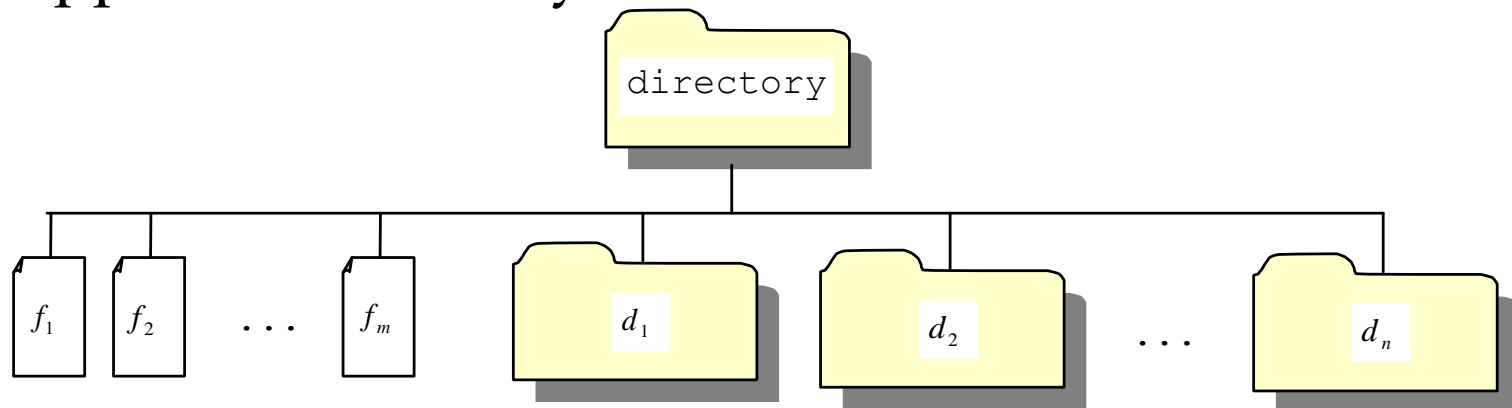
```

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);
    }
    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;
        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);
    }
}

```

# Directory Size

- Some problems are impossible to solve without recursion.
- Example: 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



```

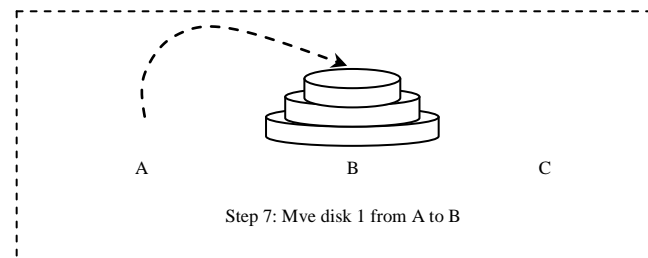
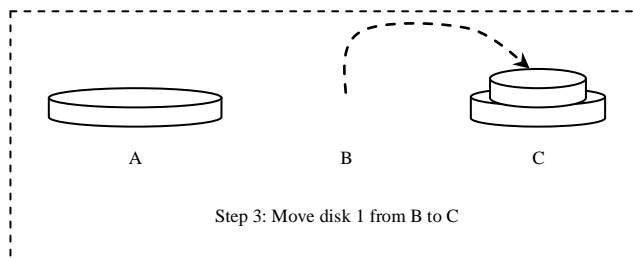
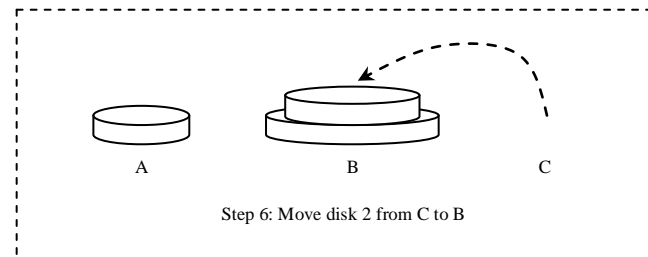
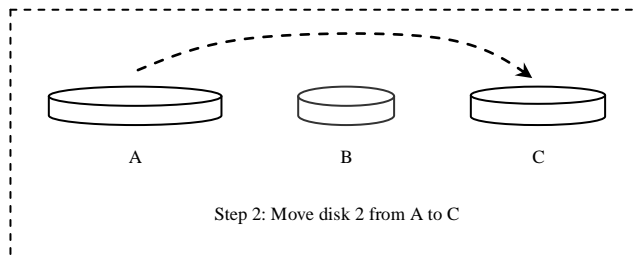
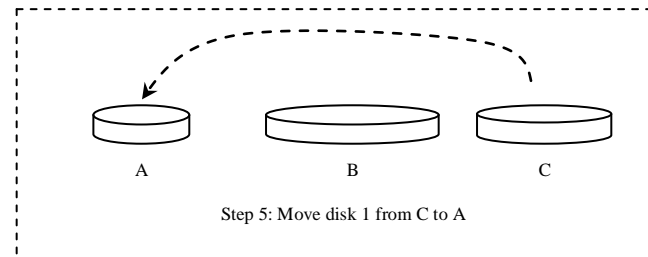
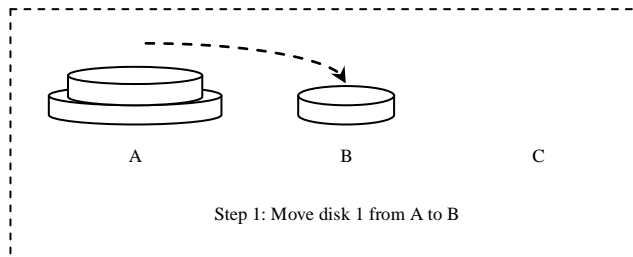
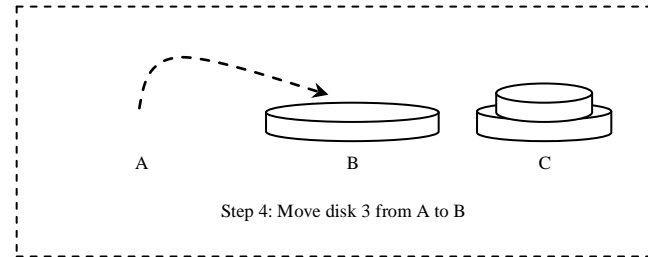
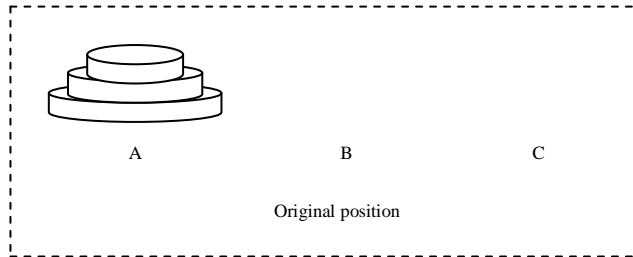
import java.io.File;
import java.util.Scanner;
public class DirectorySize {
    public static void main(String[] args) {
        System.out.print("Enter a directory or a file: ");
        Scanner input = new Scanner(System.in);
        String directory = input.nextLine();
        System.out.println(getSize(new File(directory)) + " bytes");
    }
    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; i < files.length; i++) {
                size += getSize(files[i]); // Recursive call
            }
        } else { // Base case
            size += file.length();
        }
        return size;
    }
}

```

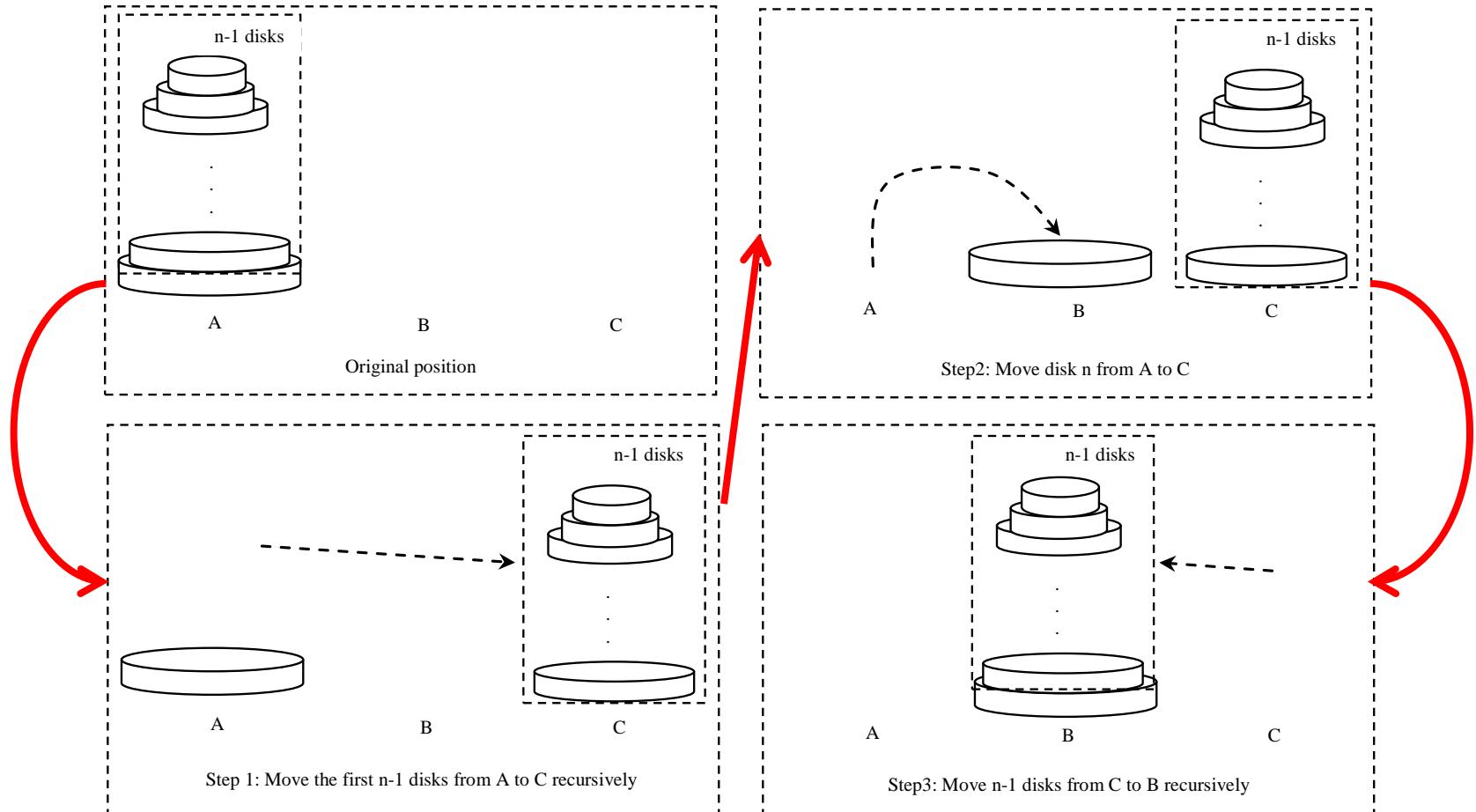


# Towers of Hanoi

- There are  $n$  disks labeled  $1, 2, 3, \dots, n$ , 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.



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.

```

import java.util.Scanner;
public class TowersOfHanoi {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter number of disks: ");
        int n = input.nextInt(); System.out.println("The moves are:");
        moveDisks(n, 'A', 'B', 'C');
    }
    public static void moveDisks(int n, char fromTower, char toTower,
        char auxTower) {
        if (n == 1) // Stopping condition
            System.out.println("Move disk " + n + " from " +
                fromTower + " to " + toTower);
        else {
            moveDisks(n - 1, fromTower, auxTower, toTower);
            System.out.println("Move disk " + n + " from " +
                fromTower + " to " + toTower);
            moveDisks(n - 1, auxTower, toTower, fromTower);
        }
    }
}

```

# Greatest Common Divisor (GCD)

$$\text{gcd}(2, 3) = 1$$

$$\text{gcd}(2, 10) = 2$$

$$\text{gcd}(25, 35) = 5$$

$$\text{gcd}(205, 5) = 5$$

**gcd(m, n):**

- Approach 1: Brute-force, start from  $\min(n, m)$  down to 1, 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: GCD

```
public static int gcd(int m, int n) {  
    int min = n;  
    if (m < n) min = m;  
    for (int i=min; i>1; i--)  
        if (m%i==0 && n%i==0)  
            return i;  
    return 1;  
}
```

## 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

`gcd(m, n) = n`                      `if m % n = 0`  
`gcd(m, n) = gcd(n, m % n);`    `otherwise`

```
public static int gcd(int m, int n) {  
    if (m % n == 0) return n;  
    else return gcd(n, m % n);  
}
```

# From Iteration to Recursion

**Example:**

```
for(int i=0; i<n; i++)  
    System.out.println(i);
```

==>

```
mr(0, n);  
public static void mr(int i, int n){  
    if(i<n){  
        System.out.println(i);  
        mr(i+1, n);  
    }  
}
```

# From Iteration to Recursion

**Mechanical transformation:**

```
for(int i=0; condition; i++)  
    body;
```

==>

```
mr(0, other_vars_in_condition);  
public static void mr(int i, other_vars_in_condition) {  
    if (condition) {  
        body;  
        mr(i+1, other_vars_in_condition);  
    }  
}
```

# From Iteration to Recursion

```
public static void m(int n) {  
    for(int i=1; i<=n; i++){  
        for(int j=1; j<=n; j++){  
            System.out.print(i+j + " ");  
        }  
        System.out.println(i);  
    }  
}  
  
public static void main(String[] args) {  
    m(10);  
}
```

```
public static void mr(int n){  
    mr(1, n);  
}  
public static void mr(int i, int n){  
    if(i<=n){  
        for(int j=1; j<=n; j++){  
            System.out.print(i+j + " ");  
        }  
        System.out.println(i);  
        mr(i+1, n);  
    }  
}
```

```

public static void mr(int n) {
    mr(1,n);
}
public static void mr(int i, int n) {
    if(i<=n) {
        mr(1, i, n);
        System.out.println(i);
        mr(i+1,n);
    }
}
public static void mr(int j, int i, int n) {
    if(j<=n) {
        System.out.print(i+j + " ");
        mr(j+1,i,n);
    }
}
public static void main(String[] args) {
    mr(10);
}

```

# Conclusion

Conclusion for this course

- That is all!
  - I hope that this course has sparked a lot of ideas and encourages you to exercise programming
- Thank you!