Recursion

CSE 114: Introduction to Object-Oriented Programming

Paul Fodor

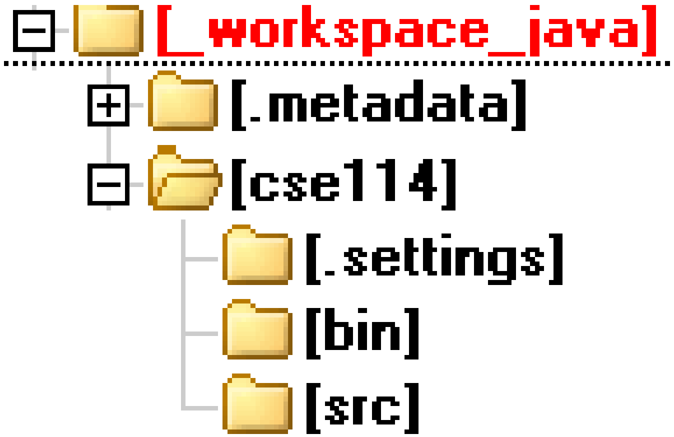
Stony Brook University

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

Contents

Motivation: Recursive Data Structures

- Suppose you want to find the size of a folder OR all the files under a folder that contains a particular word.
  - A folder contains subfolders, which also contain subfolders, and so on.

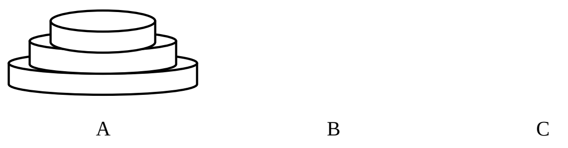


- The solution is to use recursion by looking at the files in the subfolders recursively.

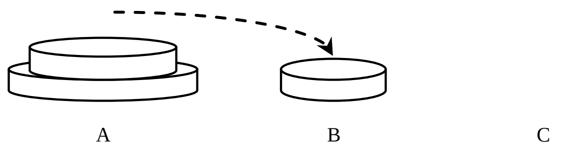
```
import java.io.File;
import java.util.Scanner;
public class DirectorySize {
  public static void main(String[] args) {
    System.out.print("Enter a folder: ");
    Scanner input = new Scanner(System.in);
    String directory = input.nextLine();
    System.out.println(getSize(new File(directory)) + " 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;
  }
}
```

Motivation: Recursive Algorithms

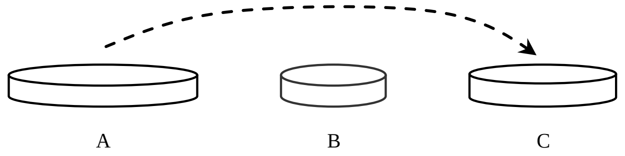
- Towers of Hanoi:
  - There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
  - No disk can be on top of a smaller disk at any time.
  - All the disks are initially placed on tower A.
  - Only one disk can be moved at a time, and it must be the top disk on the tower.



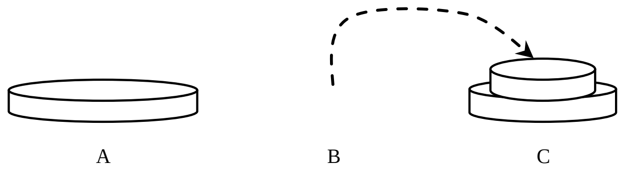
Original position



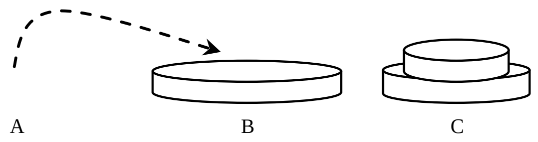
Step 1: Move disk 1 from A to B



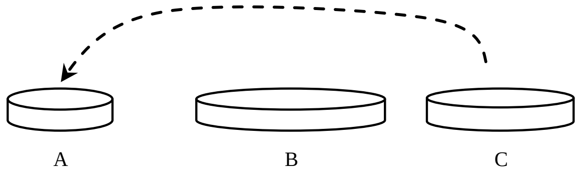
Step 2: Move disk 2 from A to C



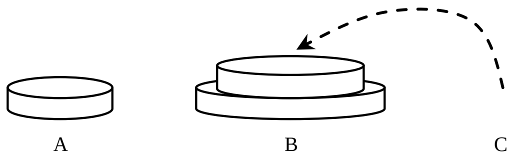
Step 3: Move disk 1 from B to C



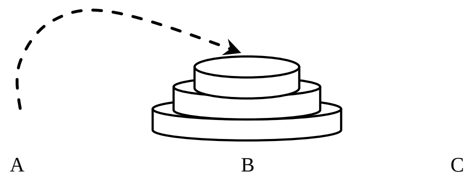
Step 4: Move disk 3 from A to B



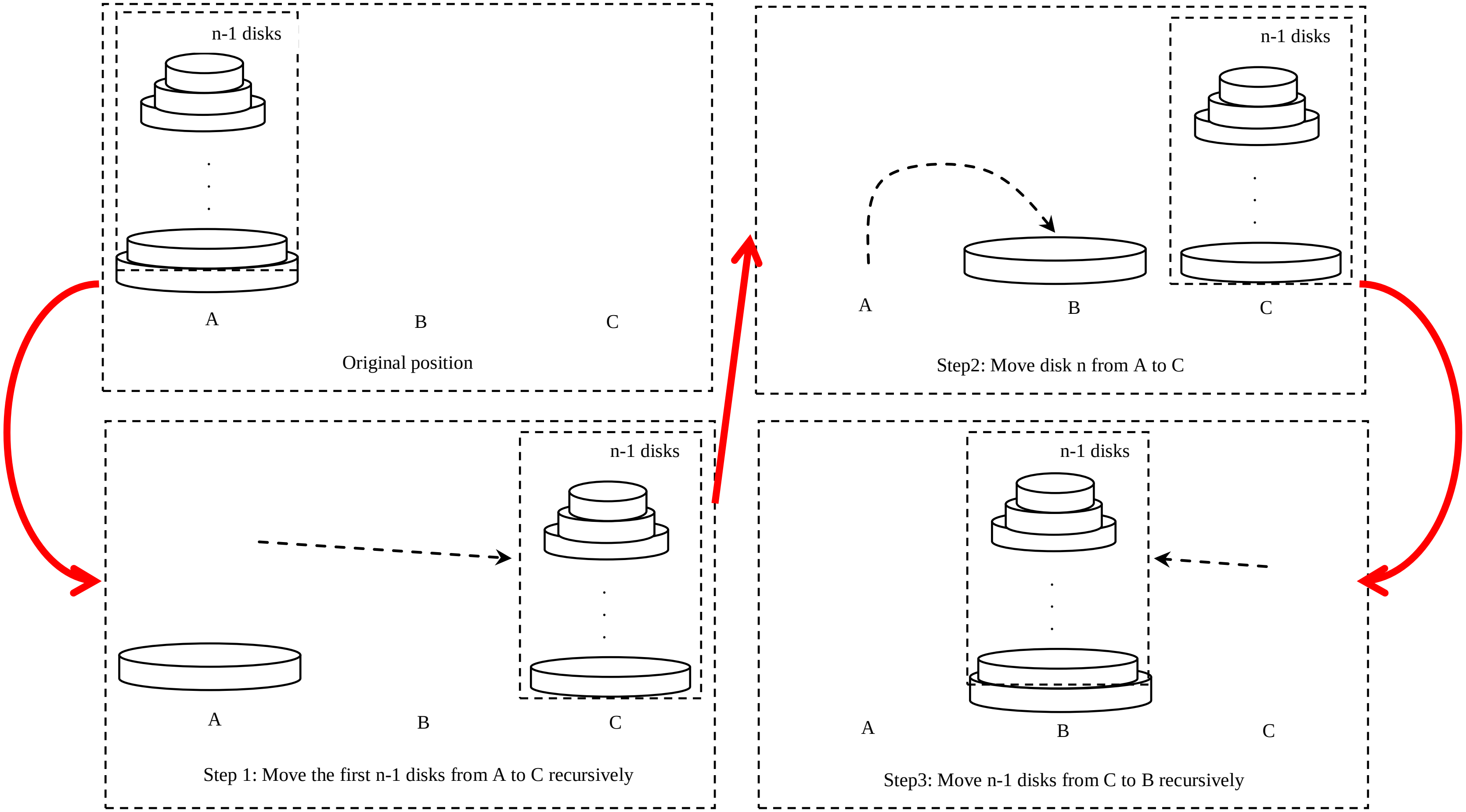
Step 5: Move disk 1 from C to A



Step 6: Move disk 2 from C to B



Step 7: Mve disk 1 from A to B



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 (called base case)
      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. Example:
  - 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 references to other node(s))

Computing Factorial

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

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

So:

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

$0! = 1$

Therefore, the recursive function is:

factorial(0) = 1;

factorial(n) = factorial(n-1) * n, 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(4) =

factorial(0) = 1;

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

Computing Factorial

factorial(4) = 4 * factorial(3)

=

factorial(0) = 1;

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

Computing Factorial

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

=

factorial(0) = 1;

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

Computing Factorial

factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

= 4 * (3 * (2 * factorial(1)))

=

factorial(0) = 1;

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

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))))
=

Computing Factorial

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)))
=

Computing Factorial

$$
\begin{aligned}
\text{factorial}(4) &= 4 * \text{factorial}(3) \\
&= 4 * (3 * \text{factorial}(2)) \\
&= 4 * (3 * (2 * \text{factorial}(1))) \\
&= 4 * (3 * (2 * (1 * \text{factorial}(0)))) \\
&= 4 * (3 * (2 * (1 * 1))) \\
&= 4 * (3 * (2 * 1)) \\
&=
\end{aligned}
$$

Computing Factorial

$$\begin{aligned}
\text{factorial}(4) &= 4 * \text{factorial}(3) \\
&= 4 * (3 * \text{factorial}(2)) \\
&= 4 * (3 * (2 * \text{factorial}(1))) \\
&= 4 * (3 * (2 * (1 * \text{factorial}(0)))) \\
&= 4 * (3 * (2 * (1 * 1))) \\
&= 4 * (3 * (2 * 1)) \\
&= 4 * (3 * 2) \\
&=
\end{aligned}$$

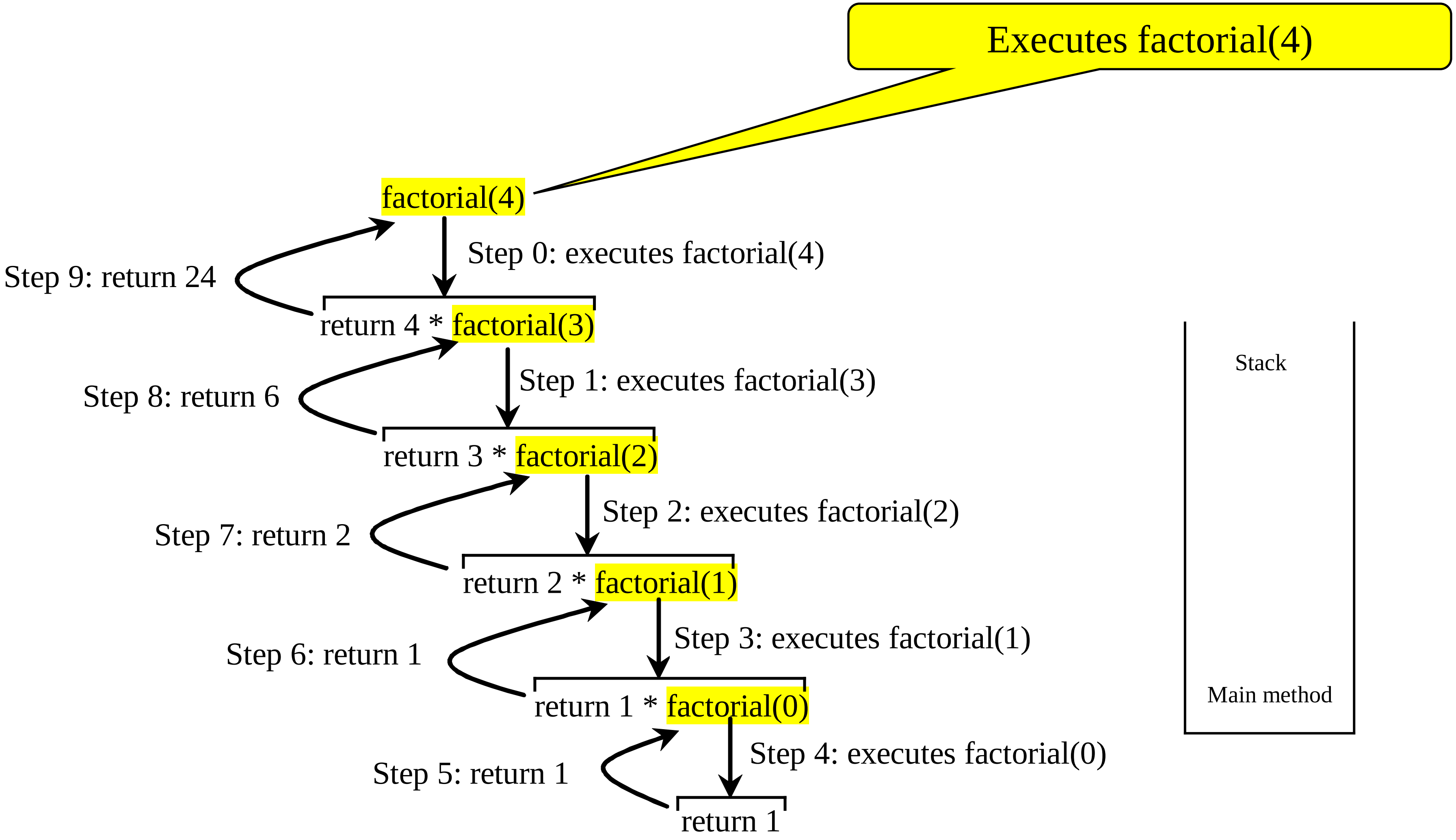
Computing Factorial

$$
\begin{aligned}
\text{factorial}(4) &= 4 * \text{factorial}(3) \\
&= 4 * (3 * \text{factorial}(2)) \\
&= 4 * (3 * (2 * \text{factorial}(1))) \\
&= 4 * (3 * (2 * (1 * \text{factorial}(0)))) \\
&= 4 * (3 * (2 * (1 * 1))) \\
&= 4 * (3 * (2 * 1)) \\
&= 4 * (3 * 2) \\
&= 4 * 6 \\
&=
\end{aligned}
$$

Computing Factorial

$$
\begin{aligned}
\text{factorial}(4) &= 4 * \text{factorial}(3) \\
&= 4 * (3 * \text{factorial}(2)) \\
&= 4 * (3 * (2 * \text{factorial}(1))) \\
&= 4 * (3 * (2 * (1 * \text{factorial}(0)))) \\
&= 4 * (3 * (2 * (1 * 1))) \\
&= 4 * (3 * (2 * 1)) \\
&= 4 * (3 * 2) \\
&= 4 * 6 \\
&= 24
\end{aligned}
$$

Trace Recursive factorial



Executes factorial(4)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

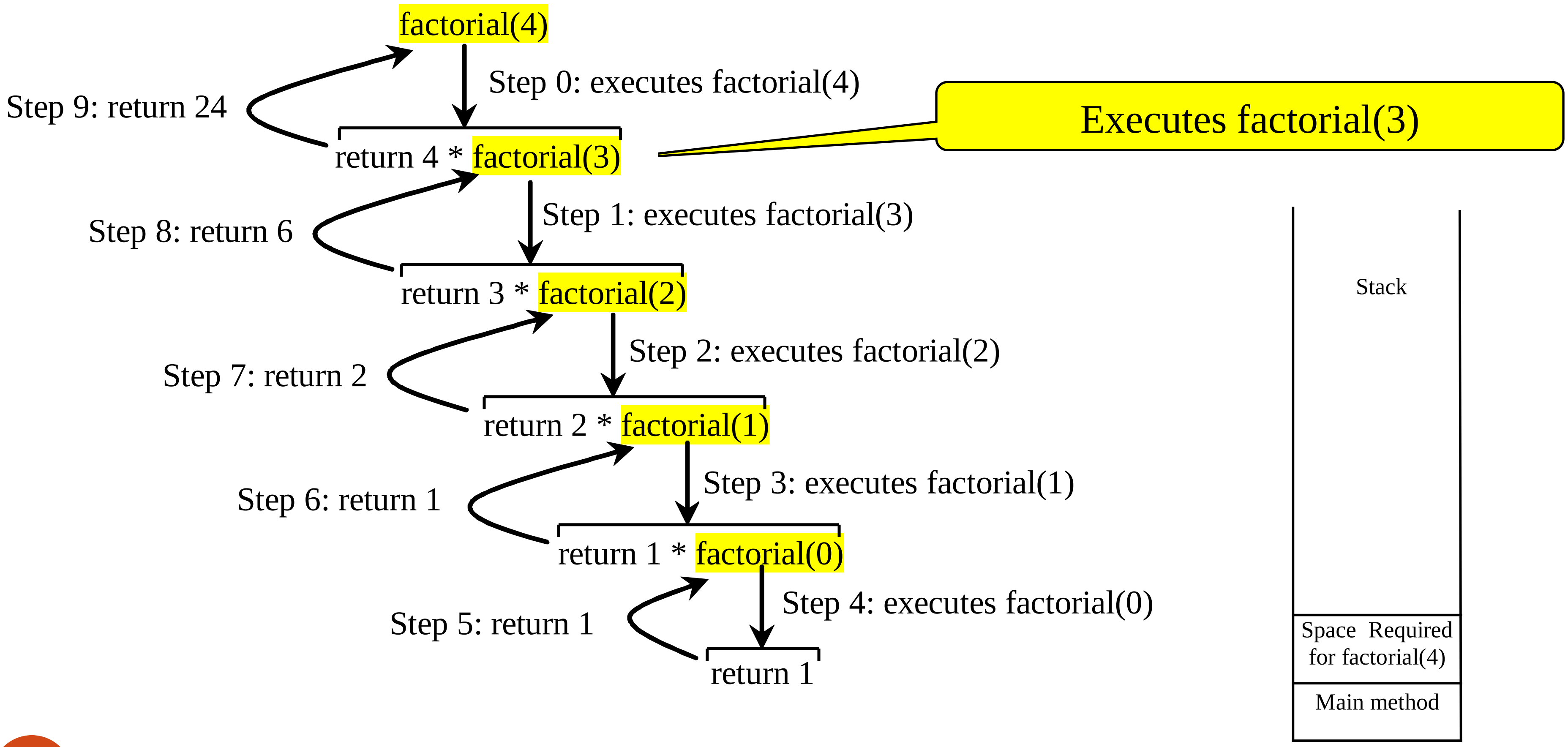
Step 5: return 1

return 1

Stack

Main method

Trace Recursive factorial



factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Executes factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

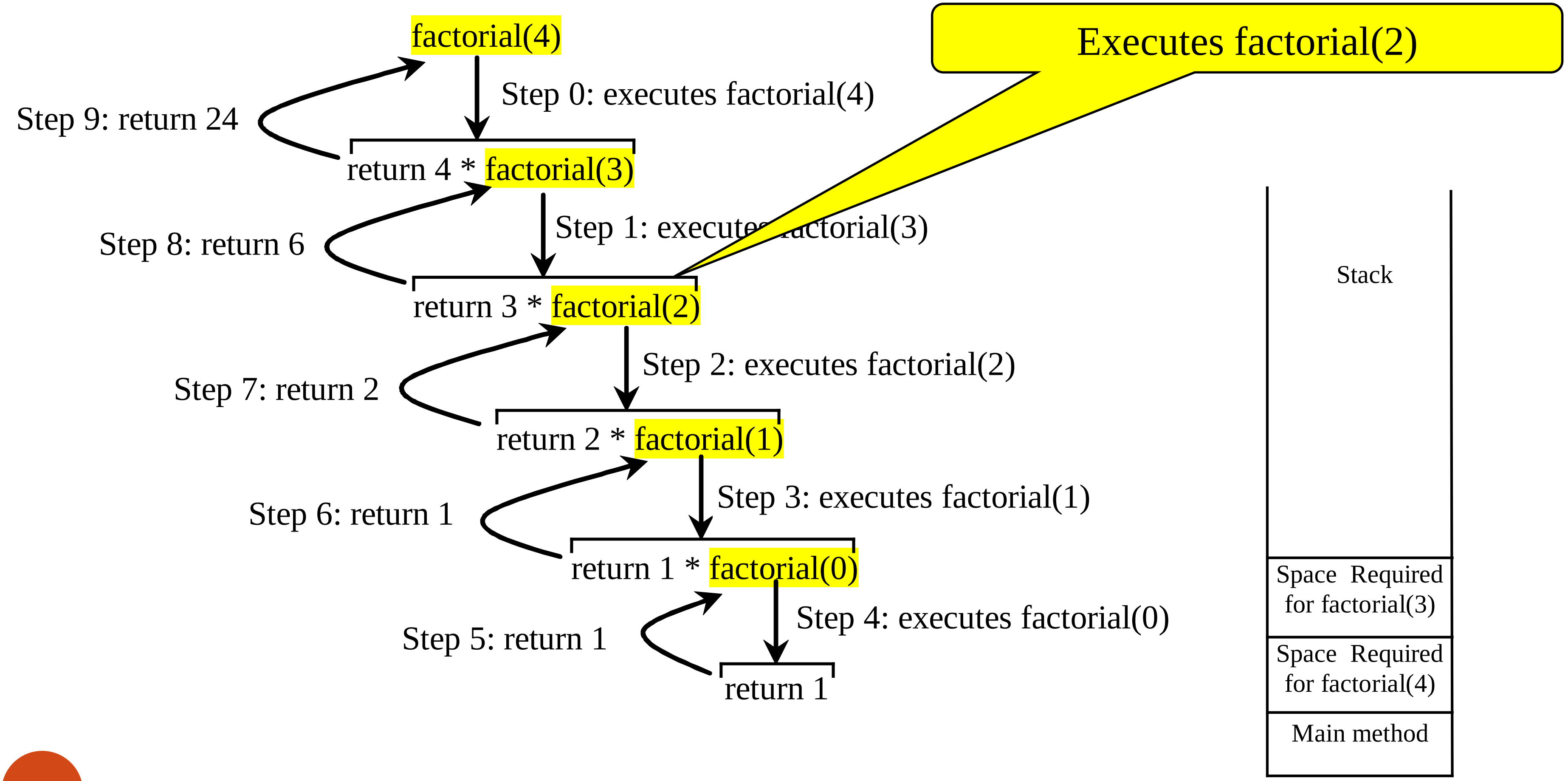
return 1

Stack

Space Required for factorial(4)

Main method

Trace Recursive factorial



Executes factorial(2)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

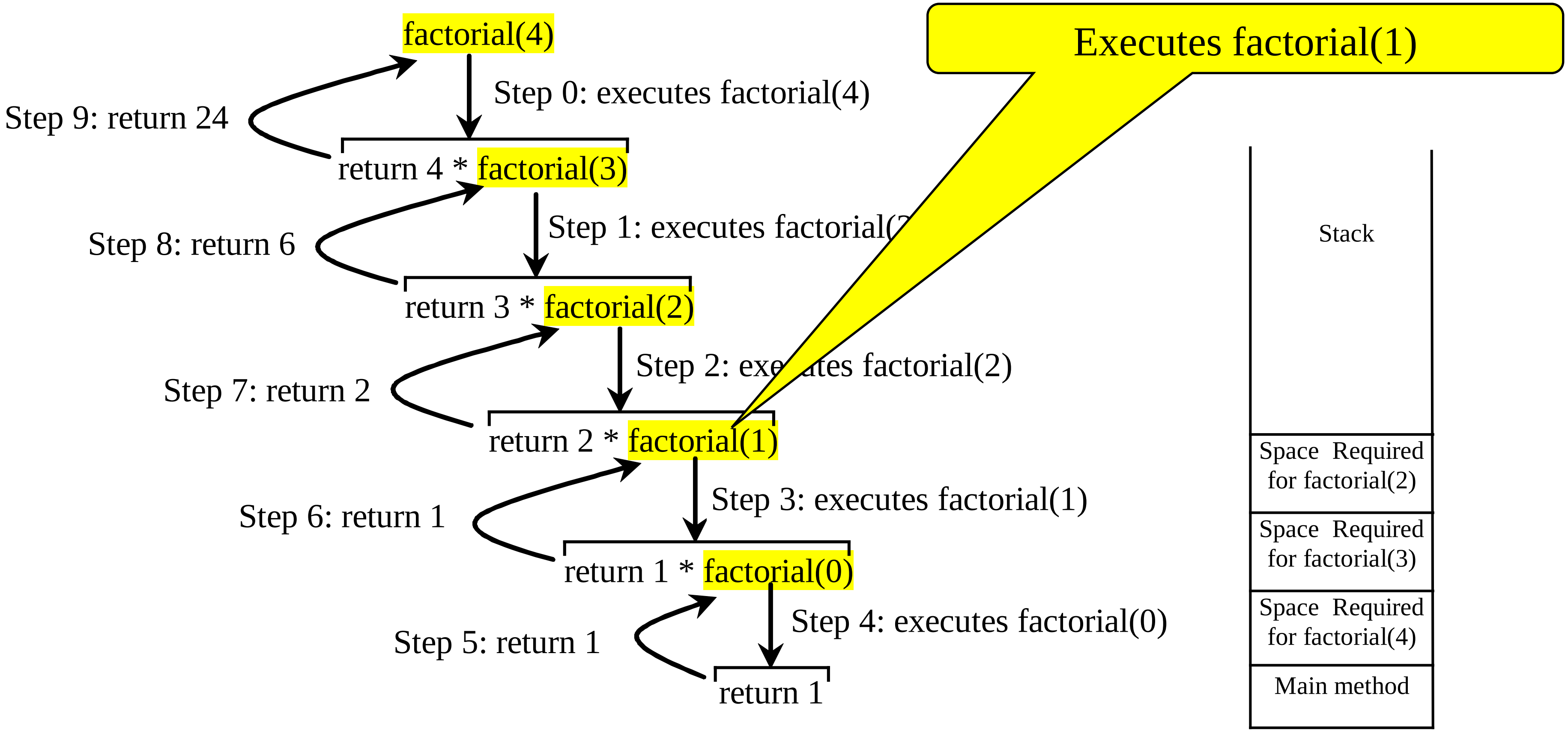
return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Trace Recursive factorial



factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(

Step 8: return 6

return 3 * factorial(2)

Step 2: exe tes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Executes factorial(1)

Stack

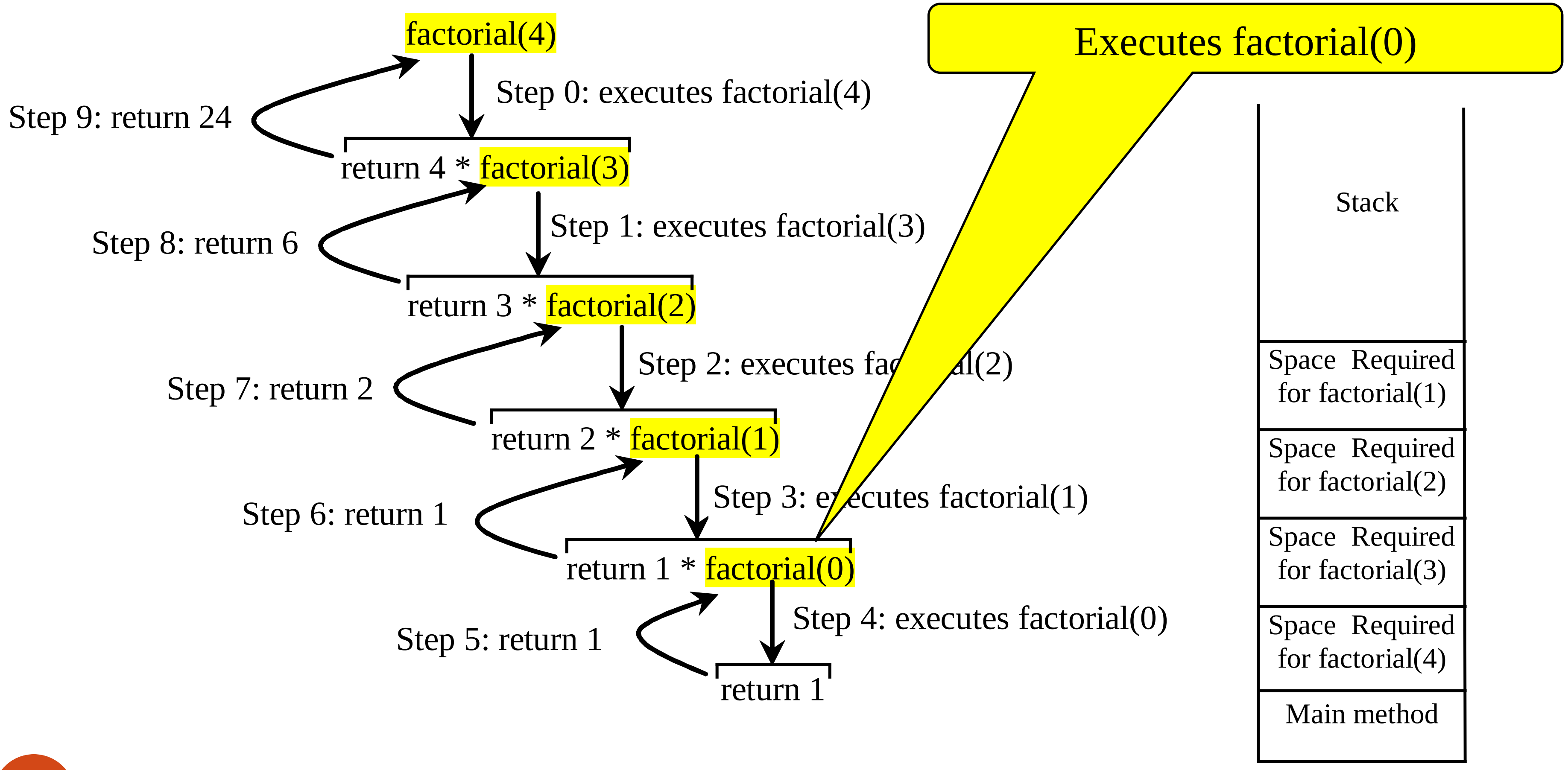
Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method

Trace Recursive factorial



Executes factorial(0)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes fac al(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Stack

Space Required for factorial(1)

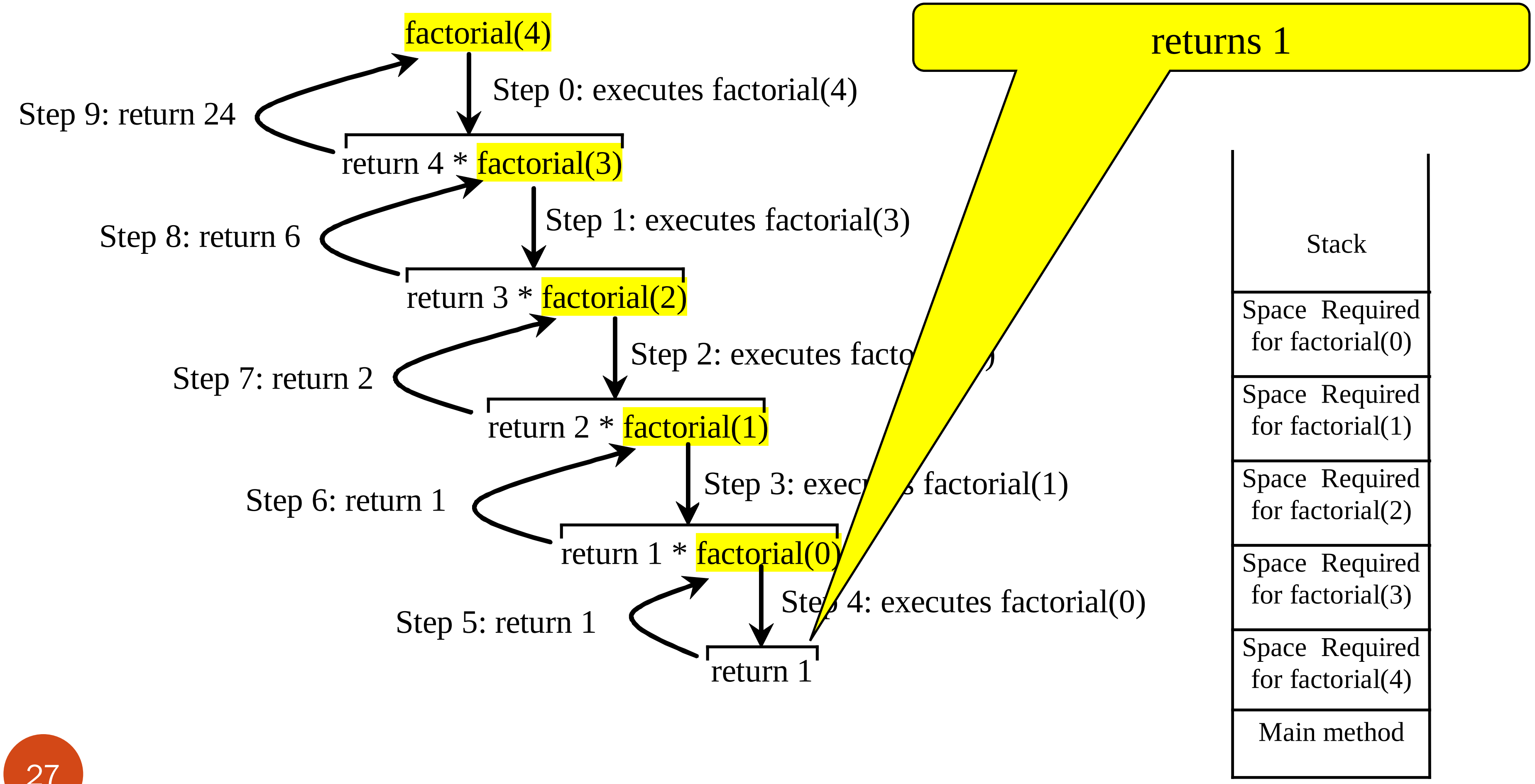
Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method

Trace Recursive factorial



returns 1

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes facto)

Step 7: return 2

return 2 * factorial(1)

Step 3: exec s factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Stack

Space Required for factorial(0)

Space Required for factorial(1)

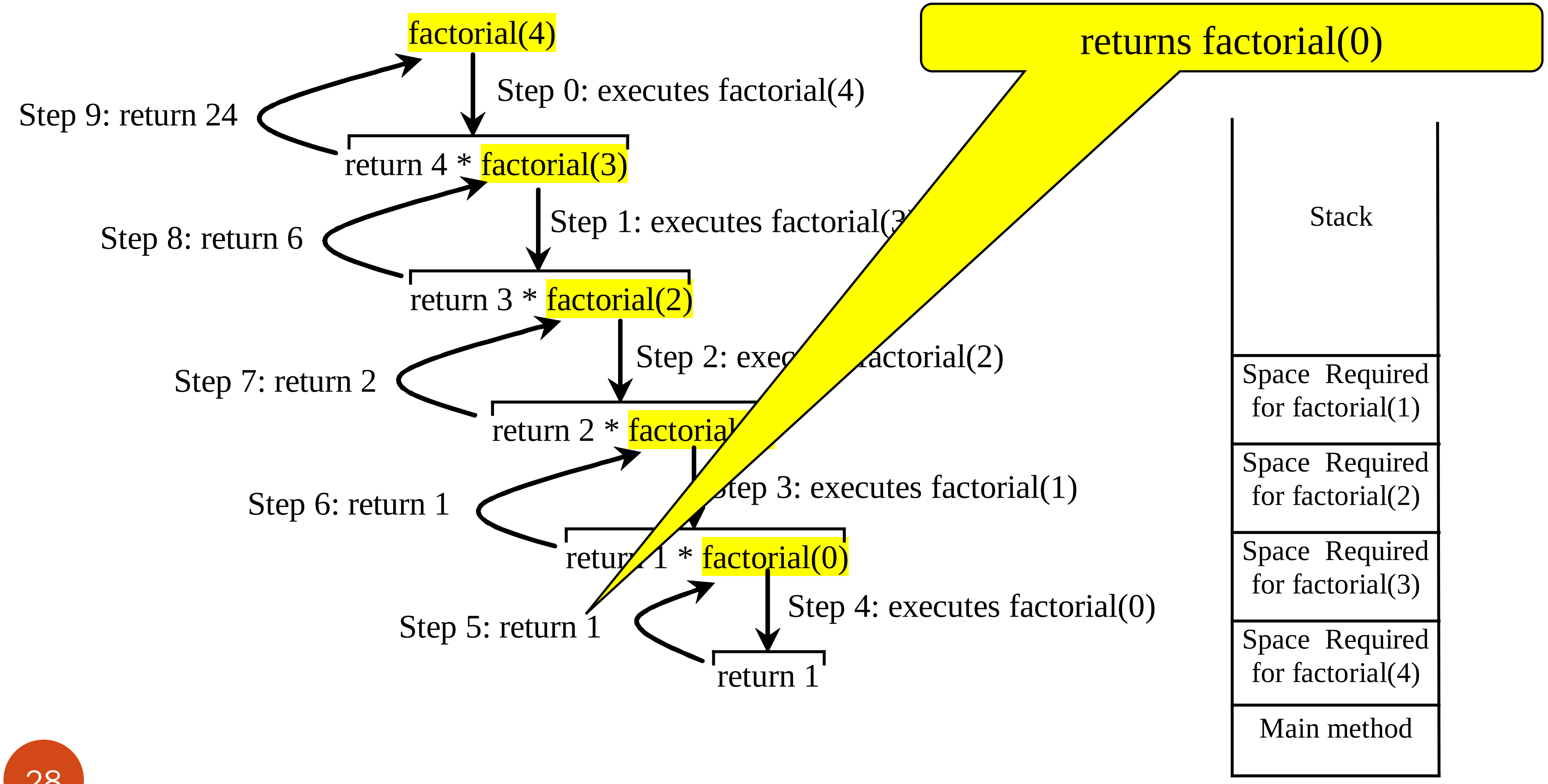
Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method

Trace Recursive factorial



returns factorial(0)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3

Step 8: return 6

return 3 * factorial(2)

Step 2: exec actorial(2)

Step 7: return 2

return 2 * factorial

tep 3: executes factorial(1)

Step 6: return 1

retur 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Stack

Space Required for factorial(1)

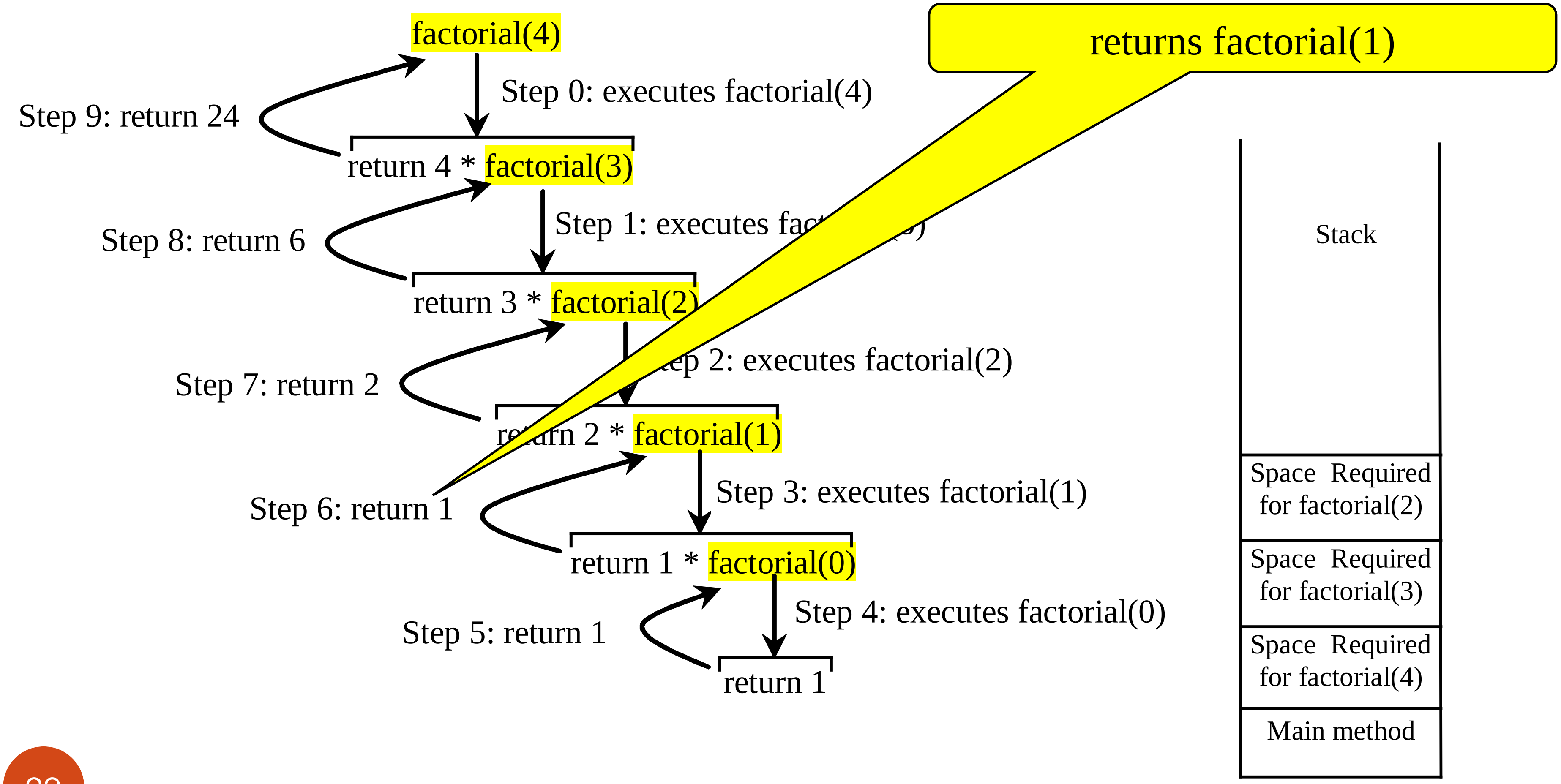
Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method

Trace Recursive factorial



returns factorial(1)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes fac 3)

Step 8: return 6

return 3 * factorial(2)

tep 2: executes factorial(2)

Step 7: return 2

re urn 2 * factorial(1)

Step 6: return 1

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

return 1

Stack

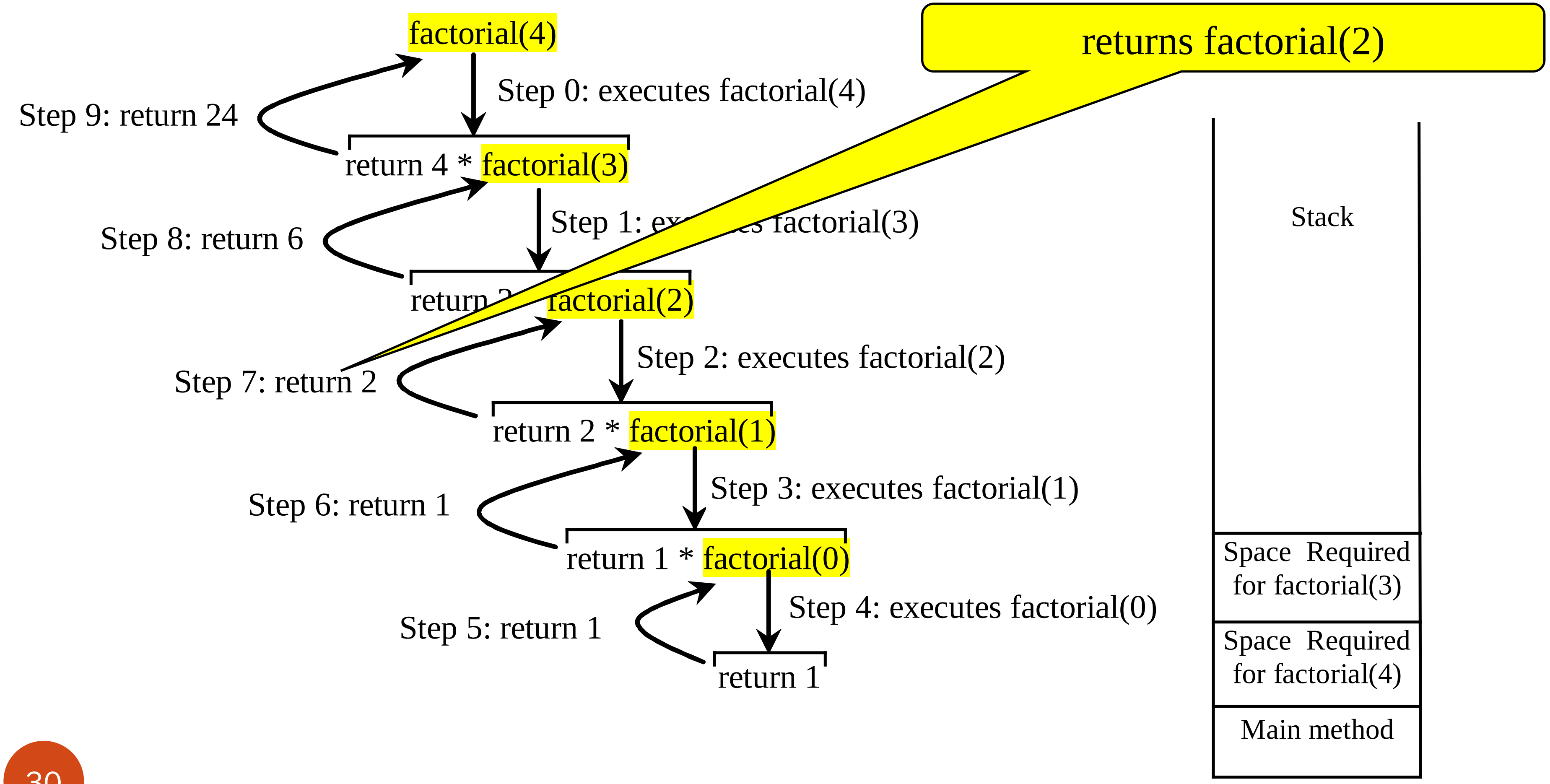
Space Required for factorial(2)

Space Required for factorial(3)

Space Required for factorial(4)

Main method

Trace Recursive factorial



returns factorial(2)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: ex es factorial(3)

Step 8: return 6

return factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

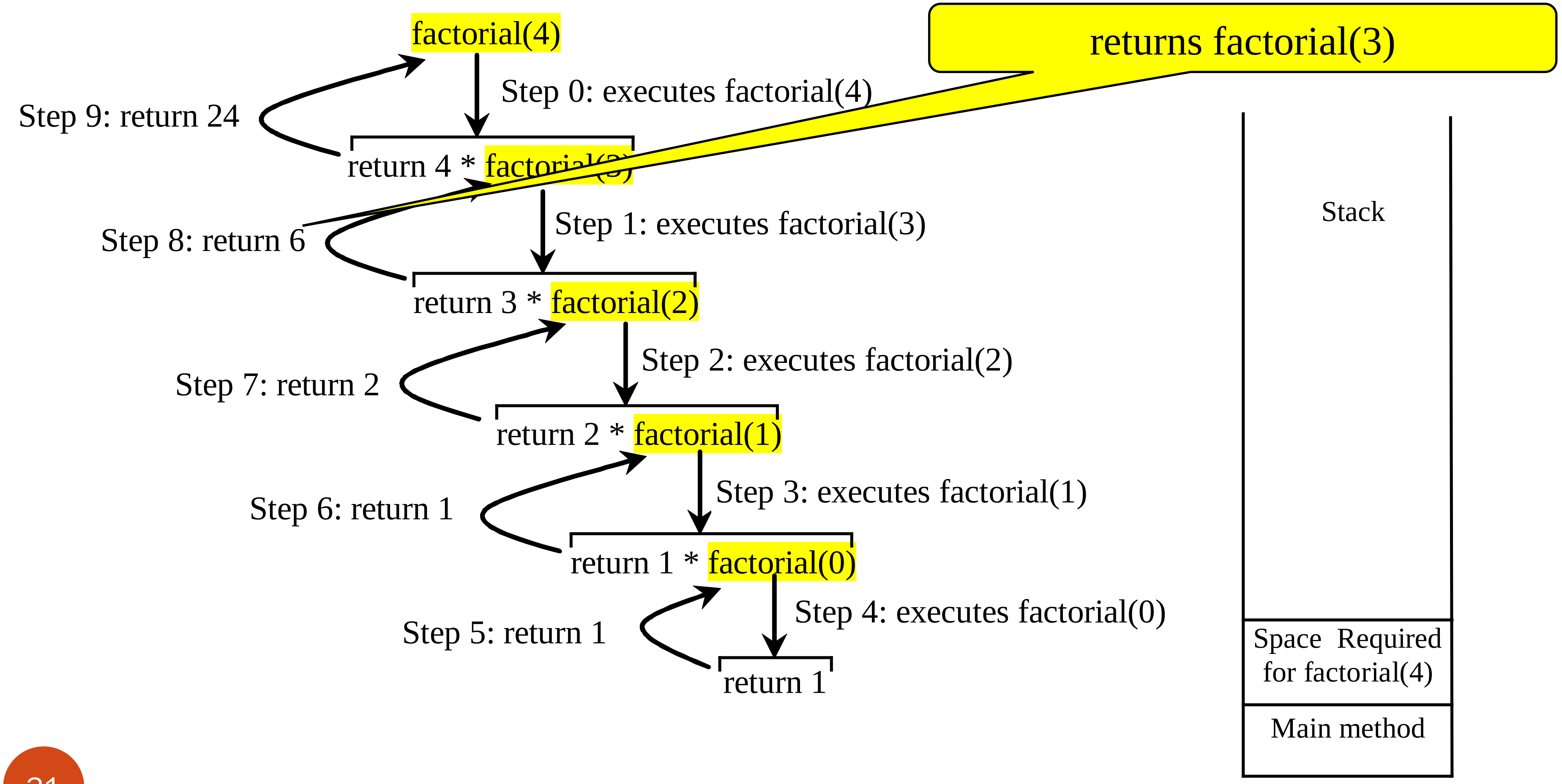
return 1 * factorial(0)

Step 4: executes factorial(0)

Step 5: return 1

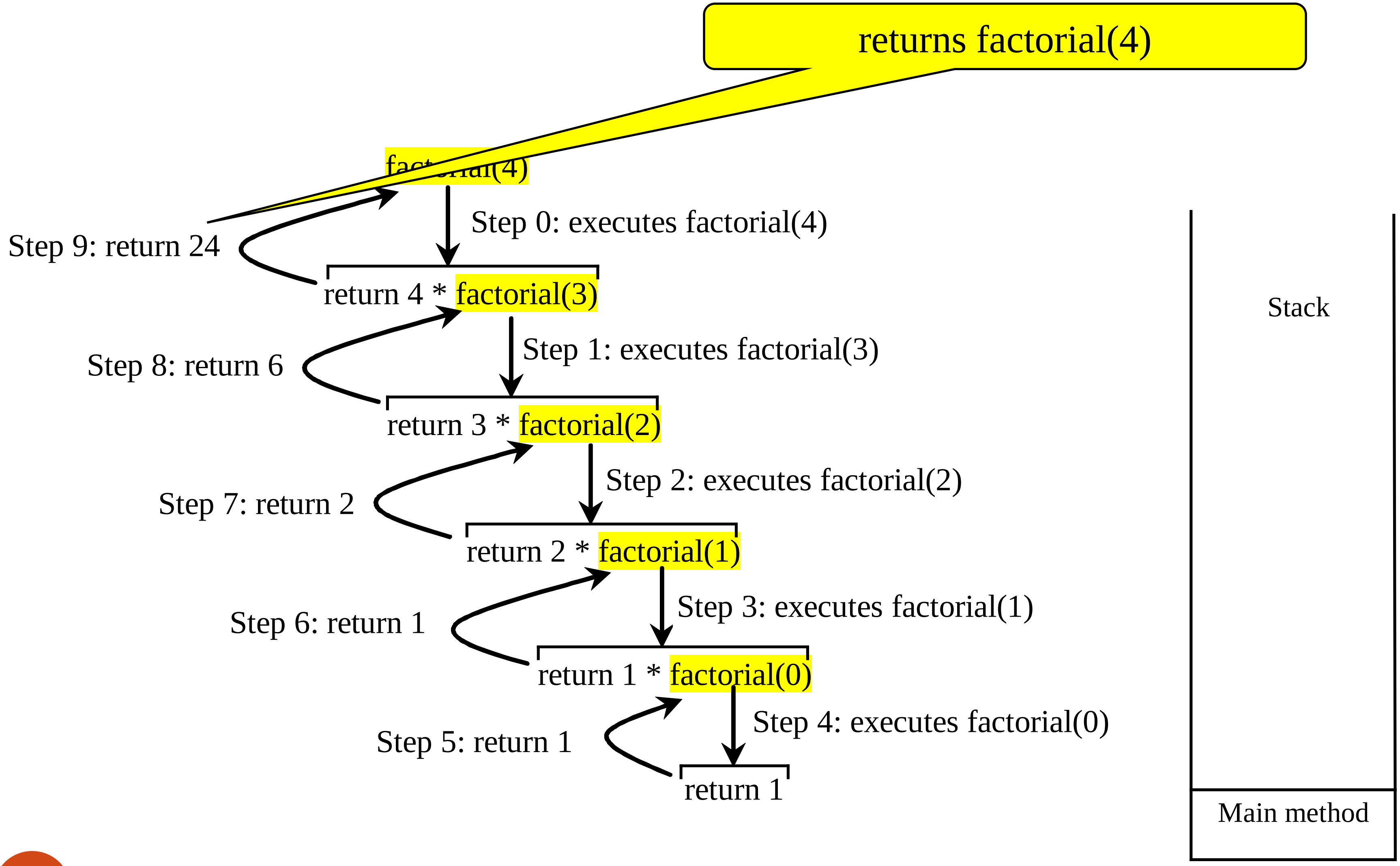
return 1

Trace Recursive factorial



returns factorial(3)

Trace Recursive factorial



returns factorial(4)

factorial(4)

Step 0: executes factorial(4)

Step 9: return 24

return 4 * factorial(3)

Step 1: executes factorial(3)

Step 8: return 6

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 7: return 2

return 2 * factorial(1)

Step 3: executes factorial(1)

Step 6: return 1

return 1 * factorial(0)

Step 4: executes factorial(0)

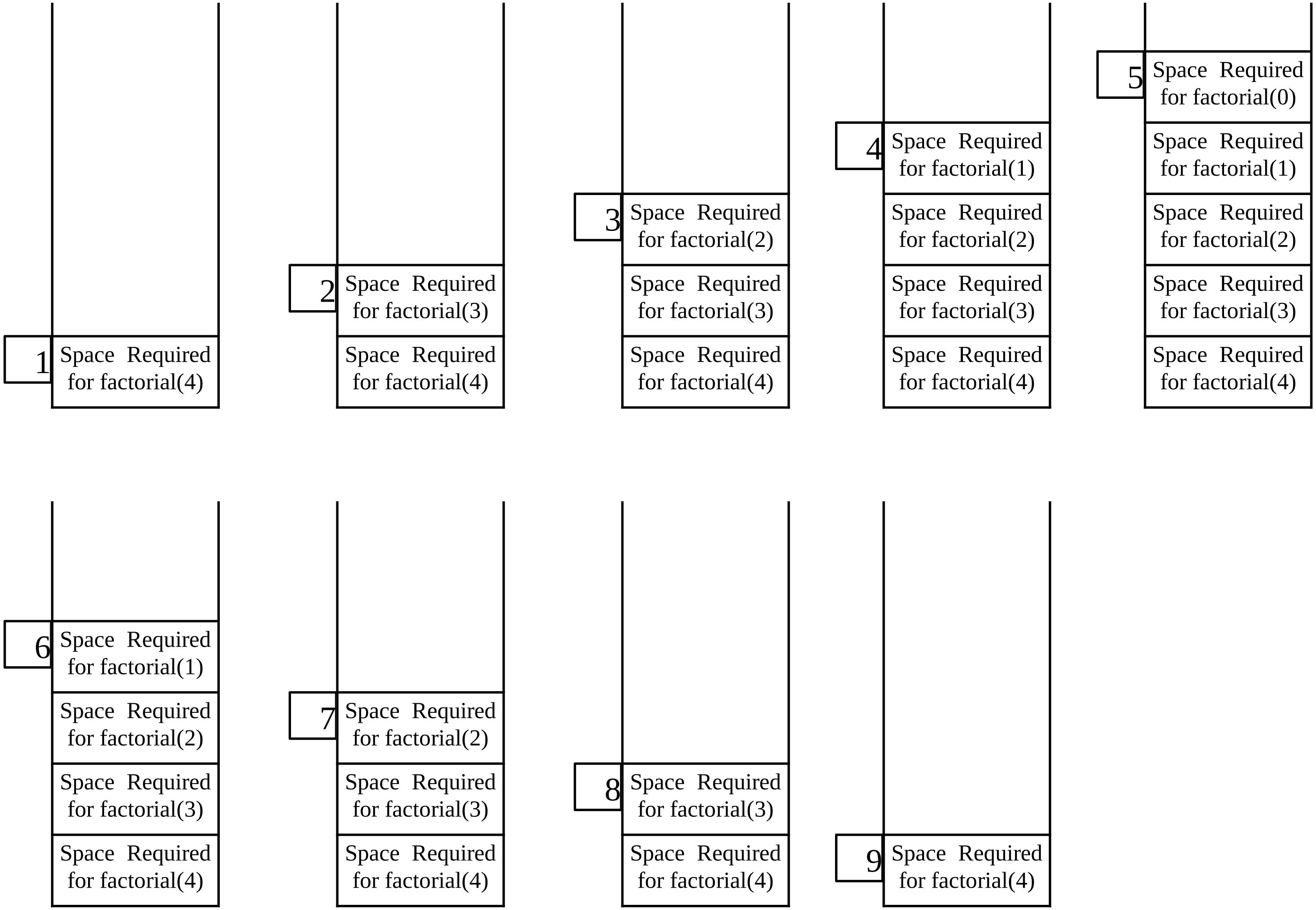
Step 5: return 1

return 1

Stack

Main method

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)

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 …
```

fib(0) = 0; // Two Base Cases

fib(1) = 1;

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

fib(4) = fib(3) + fib(2) = (fib(2) + fib(1)) + fib(2)

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

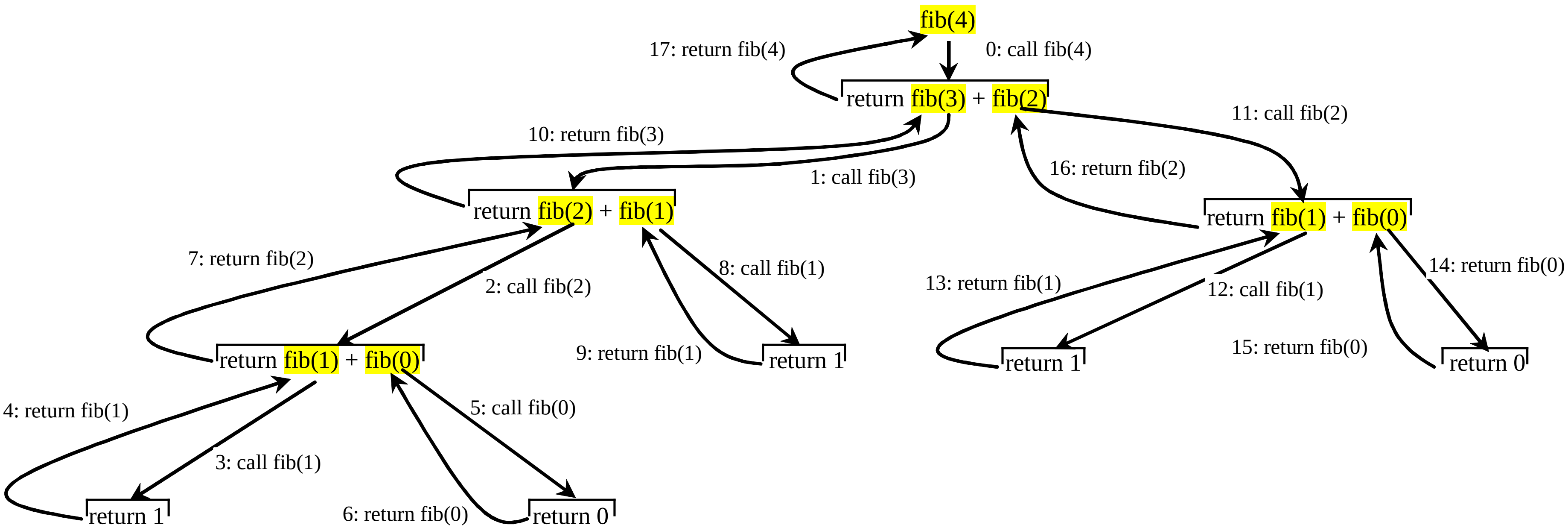
= ((1 + 0) +fib(1)) + fib(2) = (1 + fib(1)) + fib(2)

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

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

```
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(f[index] != 0)
      return f[index];
    if (index == 0)       return 0;
    if (index == 1) {      f[1]=1;      return f[1];  }
    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

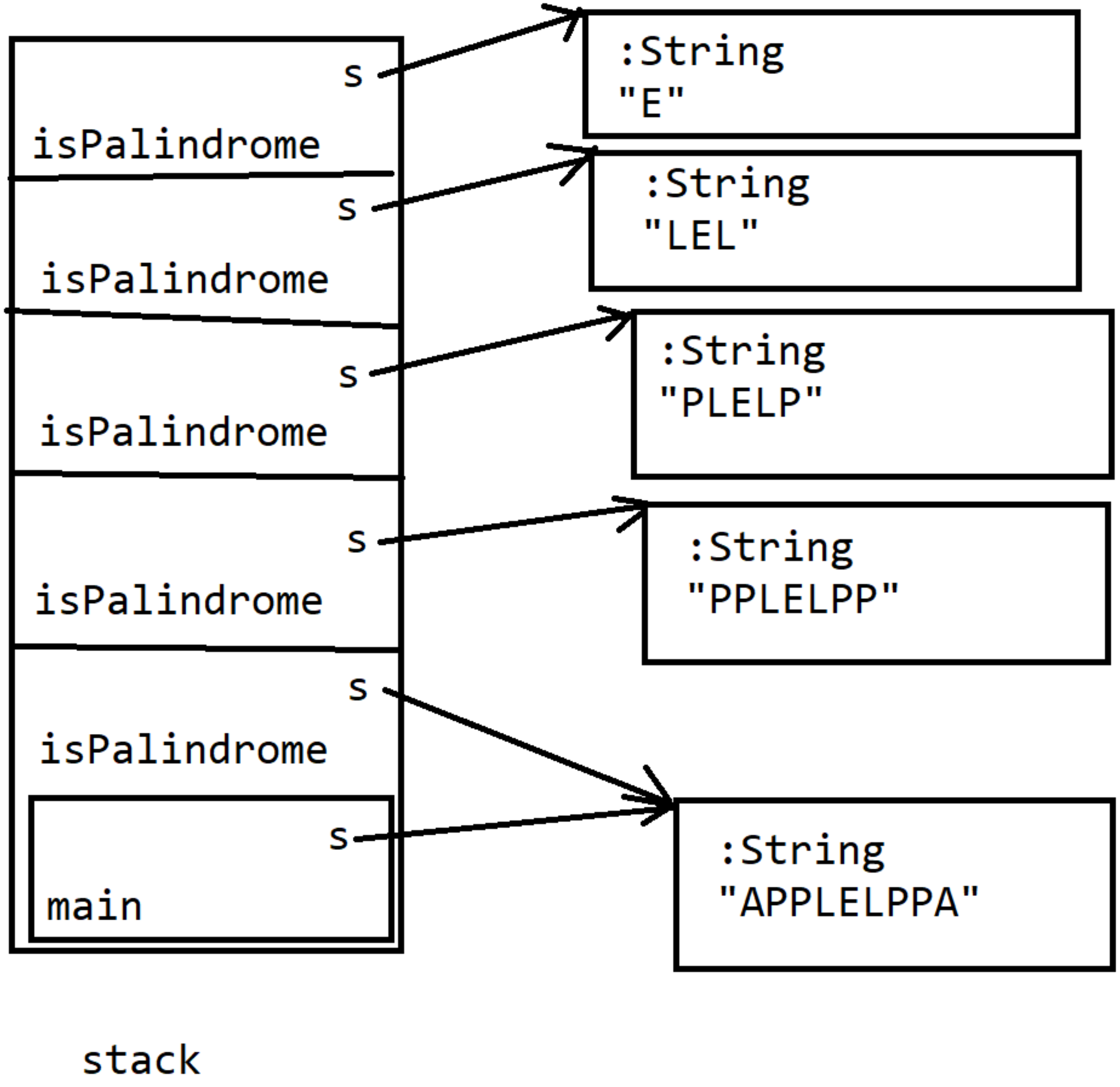
- Print a message for n times
  - break the problem into two subproblems:
    - print the message one time and
    - print the message for n-1 times
      - This new problem is the same as the original problem with a smaller size.
  - The base case for the problem is n==0.

```
public static void nPrintln(String message,int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times not >= 1
}
```

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));
}
```



s

isPalindrome

s

isPalindrome

s

isPalindrome

s

isPalindrome

s

isPalindrome

s

main

stack

:String
"E"

:String
"LEL"

:String
"PLELP"

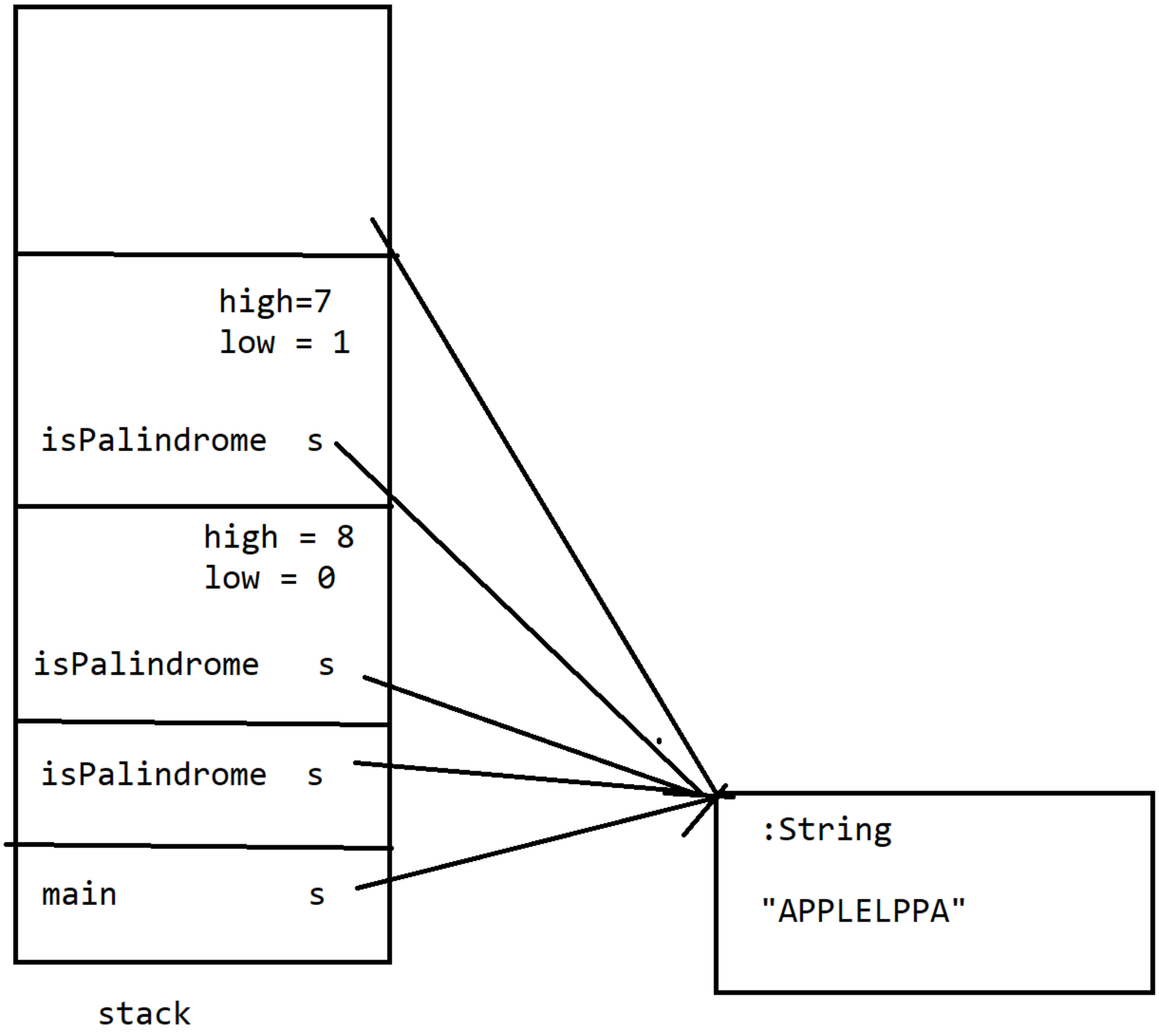
:String
"PPLELPP"

:String
"APPLELPPA"

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;
  if (s.charAt(low) != s.charAt(high))  // Base case
    return false;
  return isPalindrome(s, low + 1, high - 1);
}
```


high=7
low = 1
isPalindrome s
high = 8
low = 0
isPalindrome s
isPalindrome s
main s
stack
:String
"APPLELPPA"

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 (that is, go back to Step 1 for the remaining list)**.

- When the remaining list is empty, then we finished sorting it (this is the base case)

```
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);
    } // Base case is empty
   }
  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.
  - If the high is smaller than low index, the search fails to find the element and a negative index is returned (another Base case).

```
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 binarySearch(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) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        return recursiveBinarySearch(list, key, low, mid - 1);
      else if (key == list[mid])
        return mid; // Base case
      else
        return recursiveBinarySearch(list, key, mid + 1, high);
    } else // The list has been exhausted without a match
      return -low - 1; // Base case
  }
  // same main method
```

Greatest Common Divisor (GCD)

gcd(2, 3) = 1

gcd(2, 10) = 2

gcd(25, 35) = 5

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: Brute-force 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);
==>
public static void mr(int i, int n){
   if(i<n){
      System.out.println(i);
      mr(i+1,n);
   }
}

Call:  mr(0, n);
```

From Iteration to Recursion

```
Mechanical transformation:
  for(int i=0; condition; i++)
    body;
==>
public static void mr(int i, other_vars_in_cond_or_body){
  if(condition){
    body;
    mr(i+1, other_vars_in_cond_or_body);
  }
}

Call: mr(0,other_vars_in_cond_or_body);
```

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 m(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 m(int n){
  mr(1,n);
}
public static void mr(int i, int n){
  if(i<=n){
    mr2(1, i, n);
    System.out.println(i);
    mr(i+1,n);
  }
}
public static void mr2(int j, int i, int n){
  if(j<=n){
    System.out.print(i+j + " ");
    mr2(j+1,i,n);
  }
}
public static void main(String[] args) {
  m(10);
}
```

Conclusion

Conclusion for this course:

- That is all!
  - I hope that this course encourages you to exercise and use programming in your career.
- Thank you for taking my course!