Lecture 09: Recursion

Agenda

1. What is Recursion?
2. Characteristics and components of Recursion
3. Examples of Recursion
4. Case Study: Tower of Hanoi
5. Tail Recursion
6. Class Activity (group based)

Objectives

1. To explain what recursion is

2. To design and write functions that use recursion.

3. To learn about recursive algorithms

4. To understand and implement tail recursion

What is Recursion????

Recursion

"Recursion are methods for solving problems that depends on solutions to smaller instances of the same problem" - Wikipedia

"Recursions provides elegant solutions to problems that are difficult to program using simple loops" - Liang

Example: Computing Factorials

$$\boldsymbol{n}! = \boldsymbol{n} * (\boldsymbol{n}-\mathbf{1})!$$

$$\mathbf{0}! = \mathbf{1}$$

$$factorial(0) = 1;$$

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

Computing Factorials: Recursive

```
public static long ComputeFactorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * ComputeFactorial(n - 1);
    }
}
```

Demo: Recursion

From Plans-> VOP-9->VOP-9 (Lecture)-> Resources and Activities-> Recursion.zip-> Factorial

- IterativeFactorial.cs
- RecursiveFactorial.cs
- Program.cs

What is $\boldsymbol{factorial(4)}$?: Computing Factorial

factorial(0) = 1;

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

Step 0: factorial(4)

Computing Factorial

factorial(0) = 1;

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

Step 1: factorial(4) = 4 * factorial(3)

Computing Factorial

factorial(0) = 1;

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

Step 2: factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

Computing Factorial

factorial(0) = 1;

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

Step 3: factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

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

Computing Factorial

factorial(0) = 1;

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

Step 4: factorial(4) = 4 * factorial(3)

= 4 * (3 * factorial(2))

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

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

Computing Factorial

factorial(0) = 1;

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

Step 5: 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

factorial(0) = 1;

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

Step 6: 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))

Computing Factorial

factorial(0) = 1;

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

Step 7: 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)

Computing Factorial

factorial(0) = 1;

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

Step 8: 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)

Computing Factorial

factorial(0) = 1;

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

Step 9: 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)

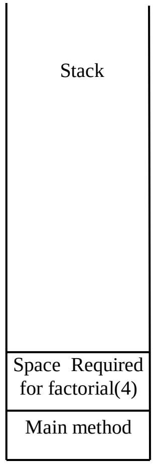
= 4 * (6)

= 24

Trace Recursive Factorial

Executes factorial(4)

factorial(4)



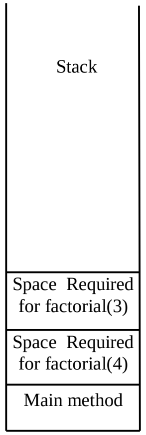
Trace Recursive Factorial

factorial(4)

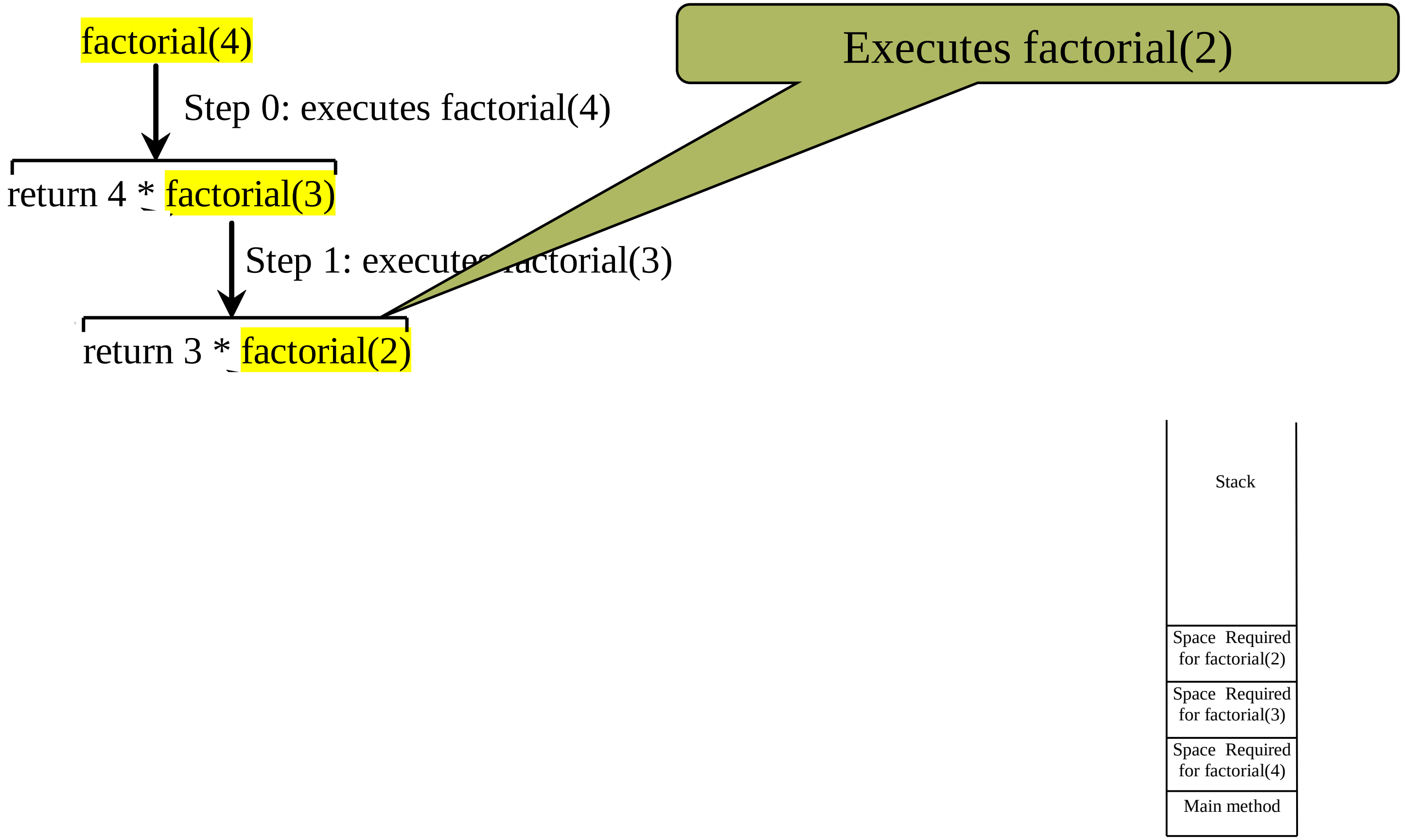
Step 0: executes factorial(4)

return 4 * factorial(3)

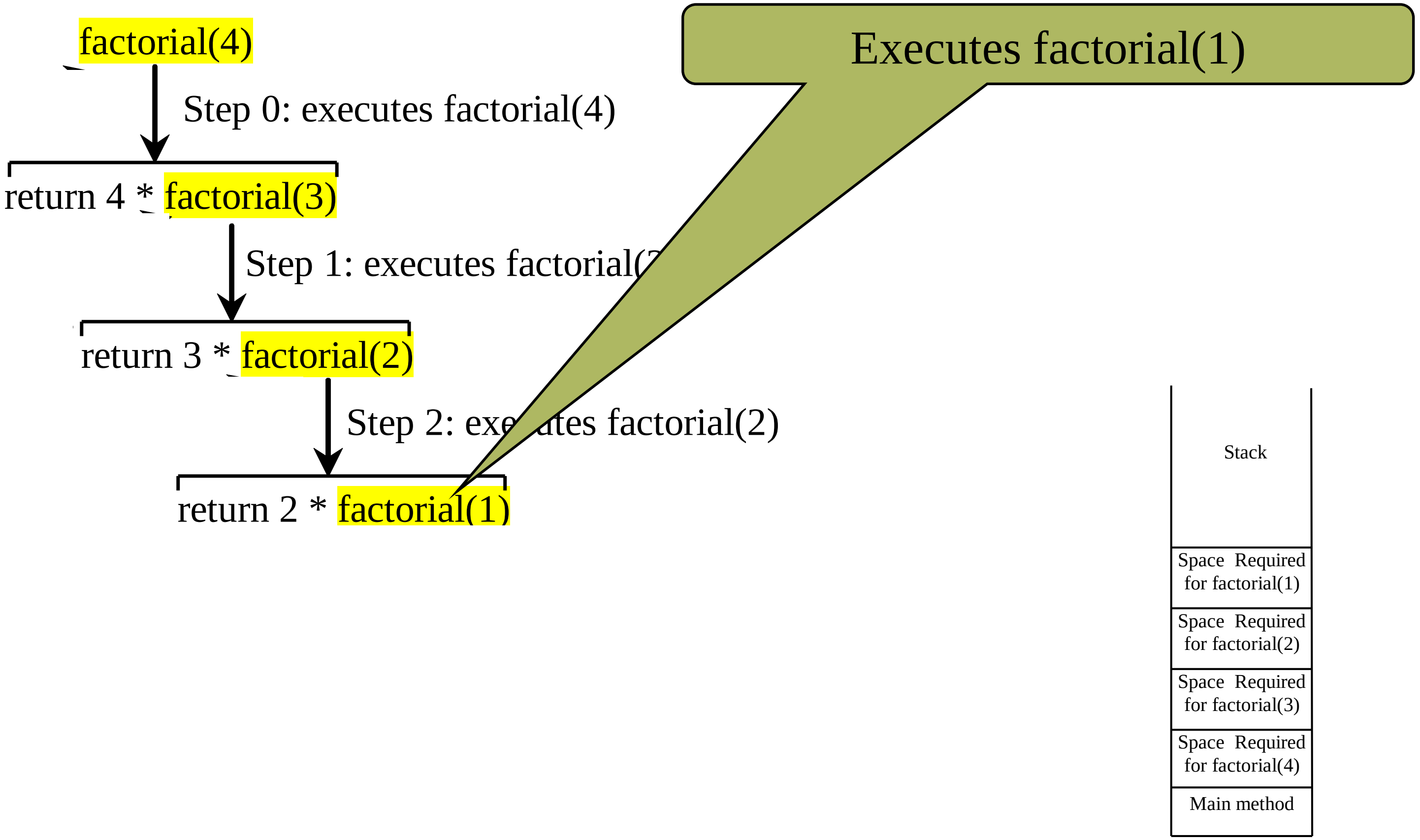
Executes factorial(3)



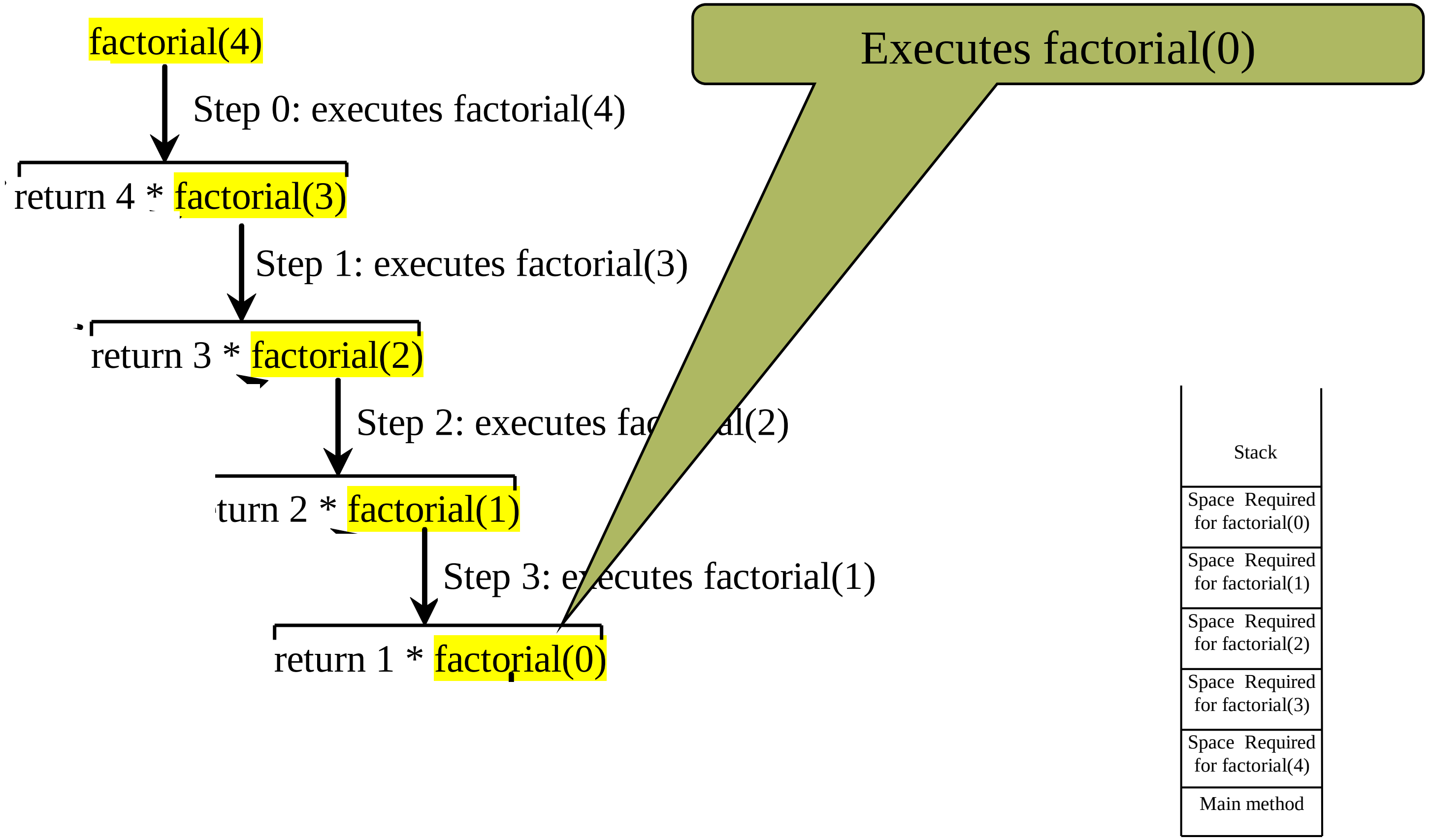
Trace Recursive Factorial



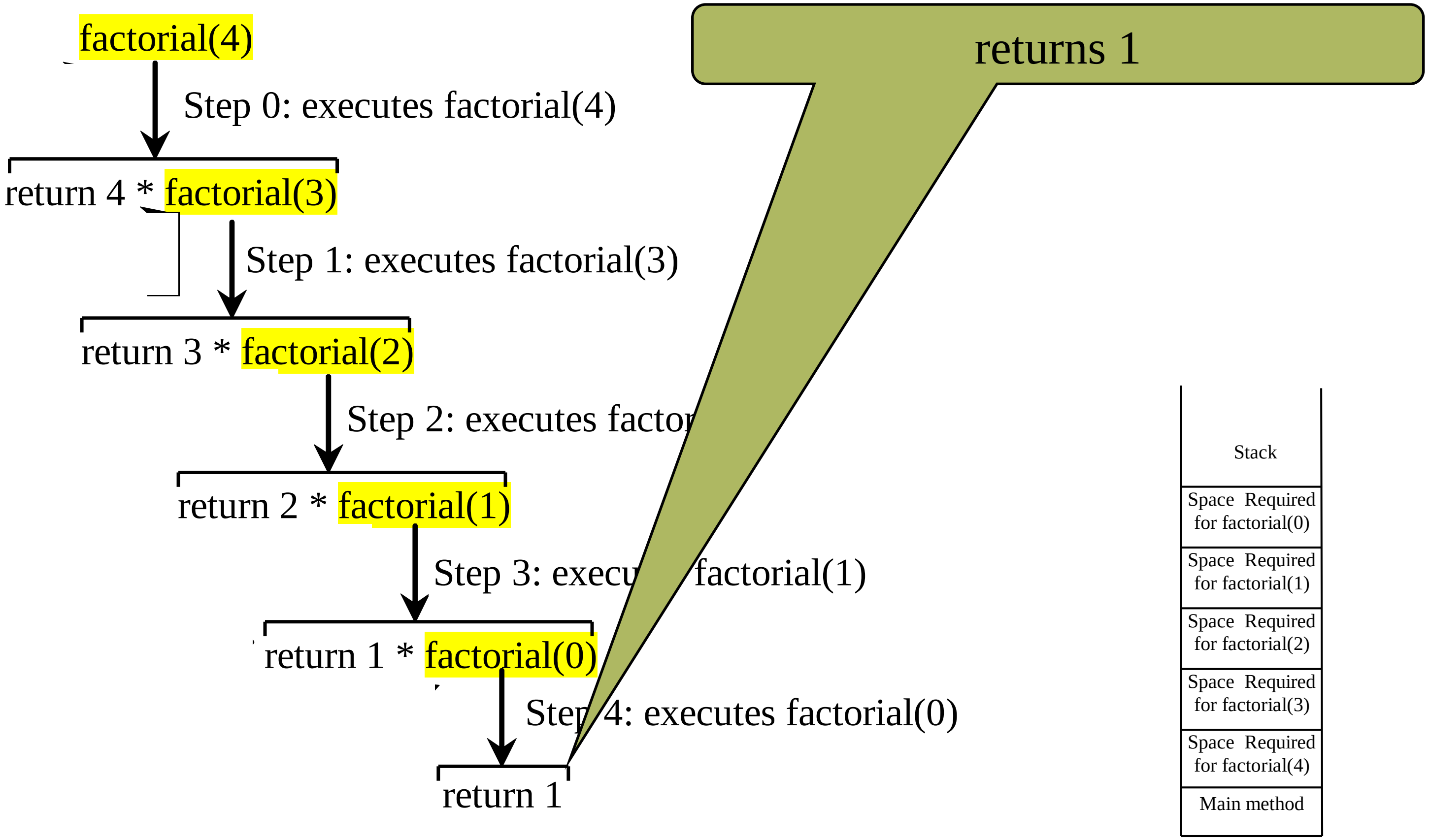
Trace Recursive Factorial



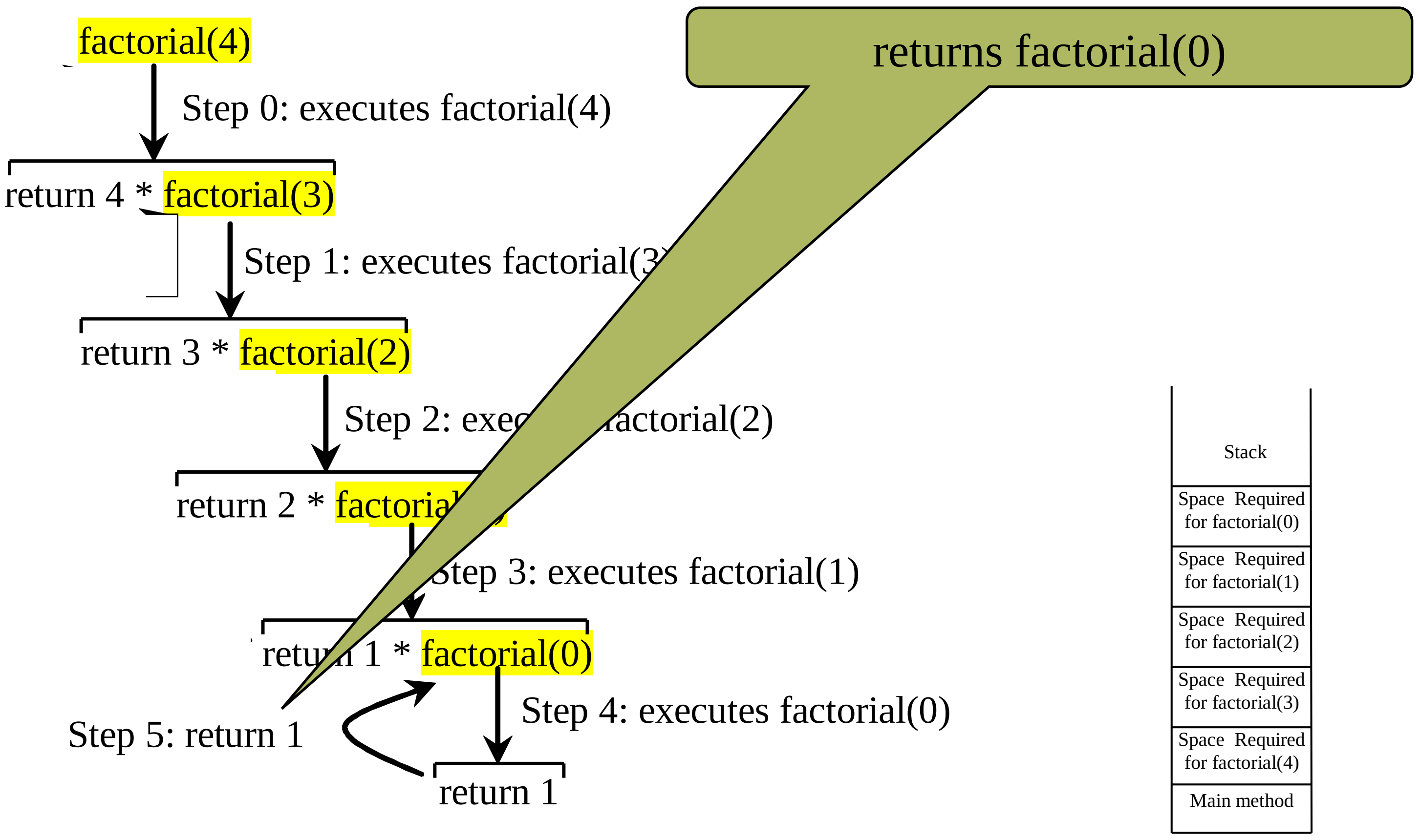
Trace Recursive Factorial



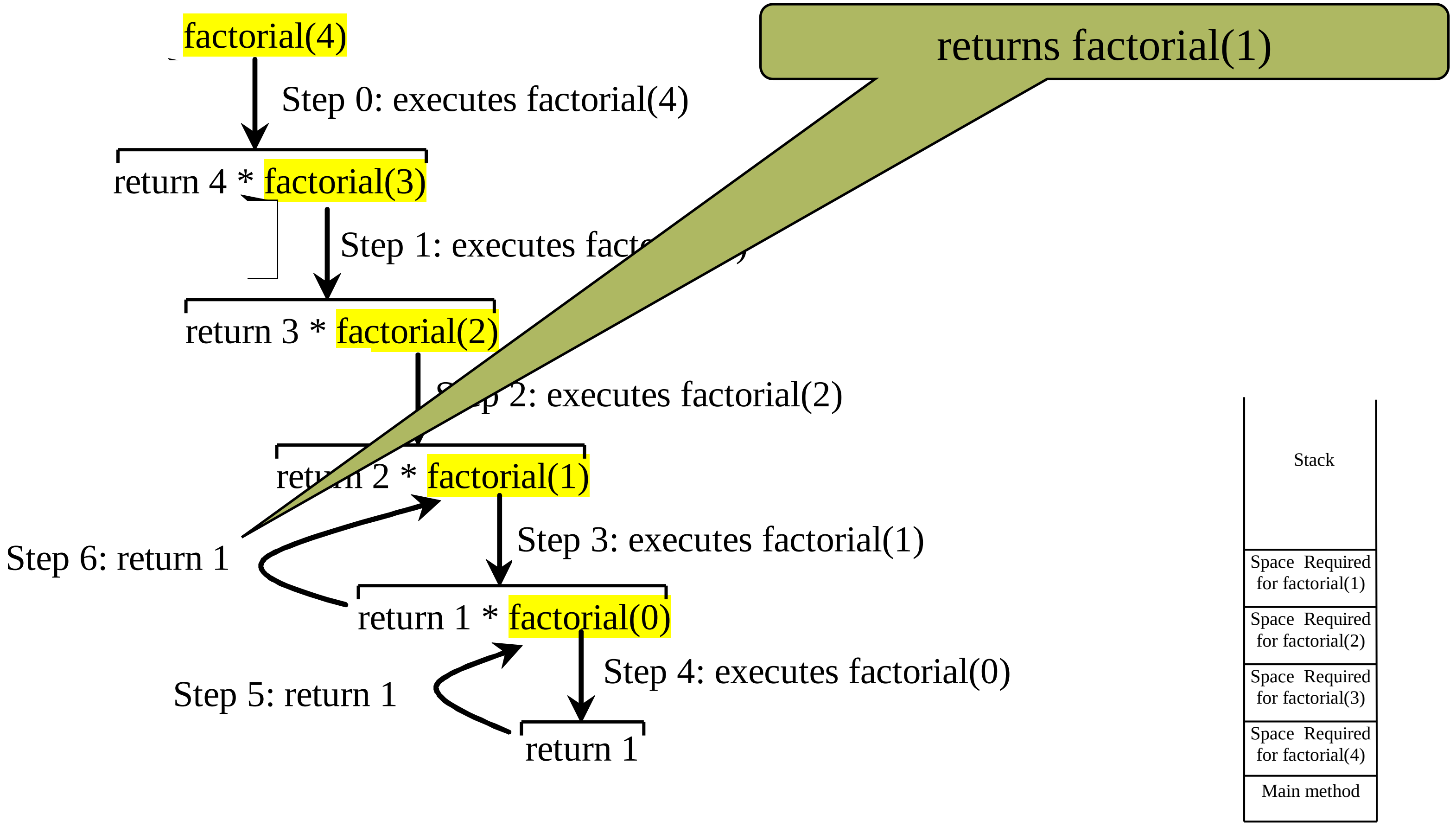
Trace Recursive Factorial



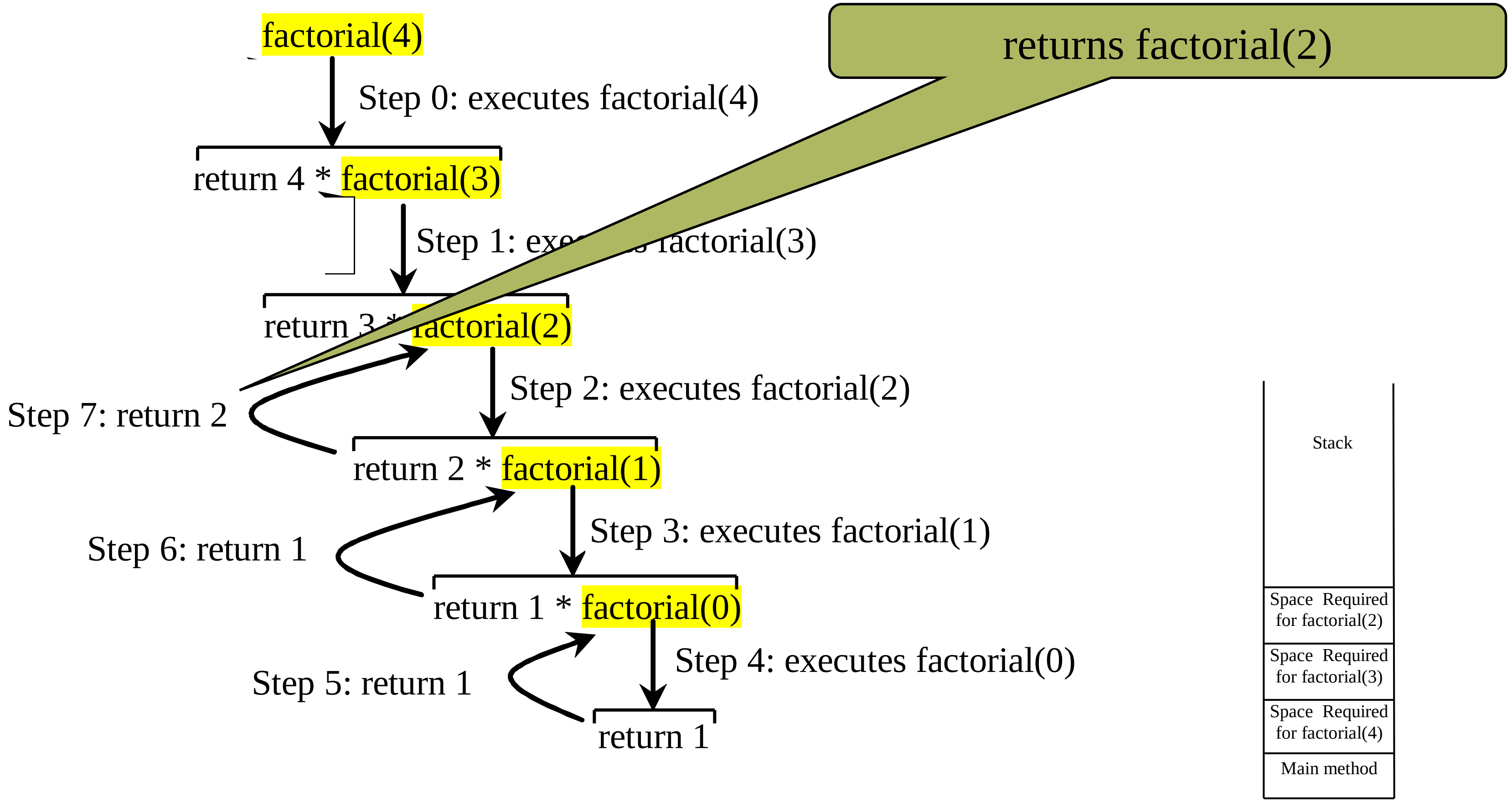
Trace Recursive Factorial



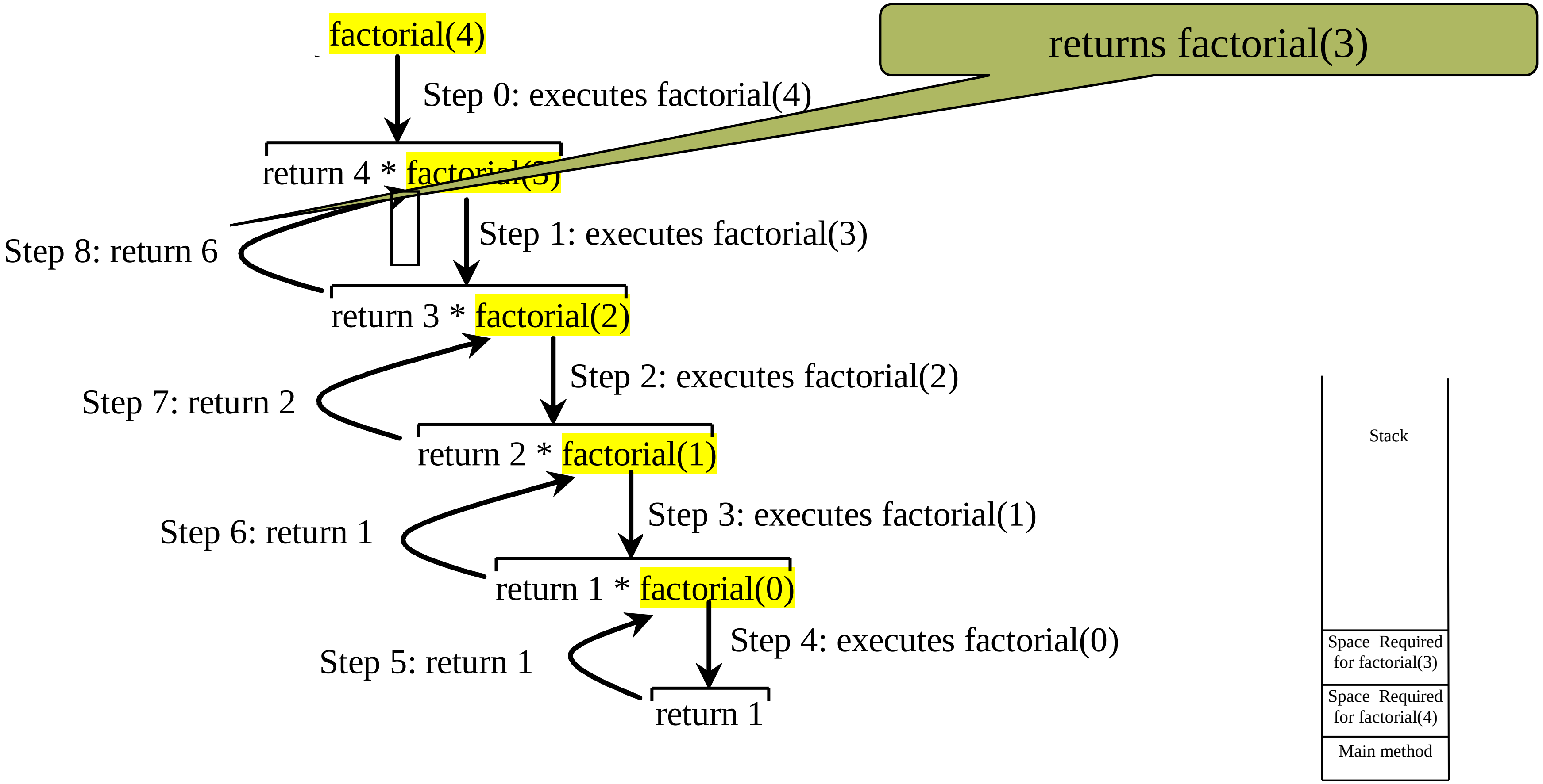
Trace Recursive Factorial



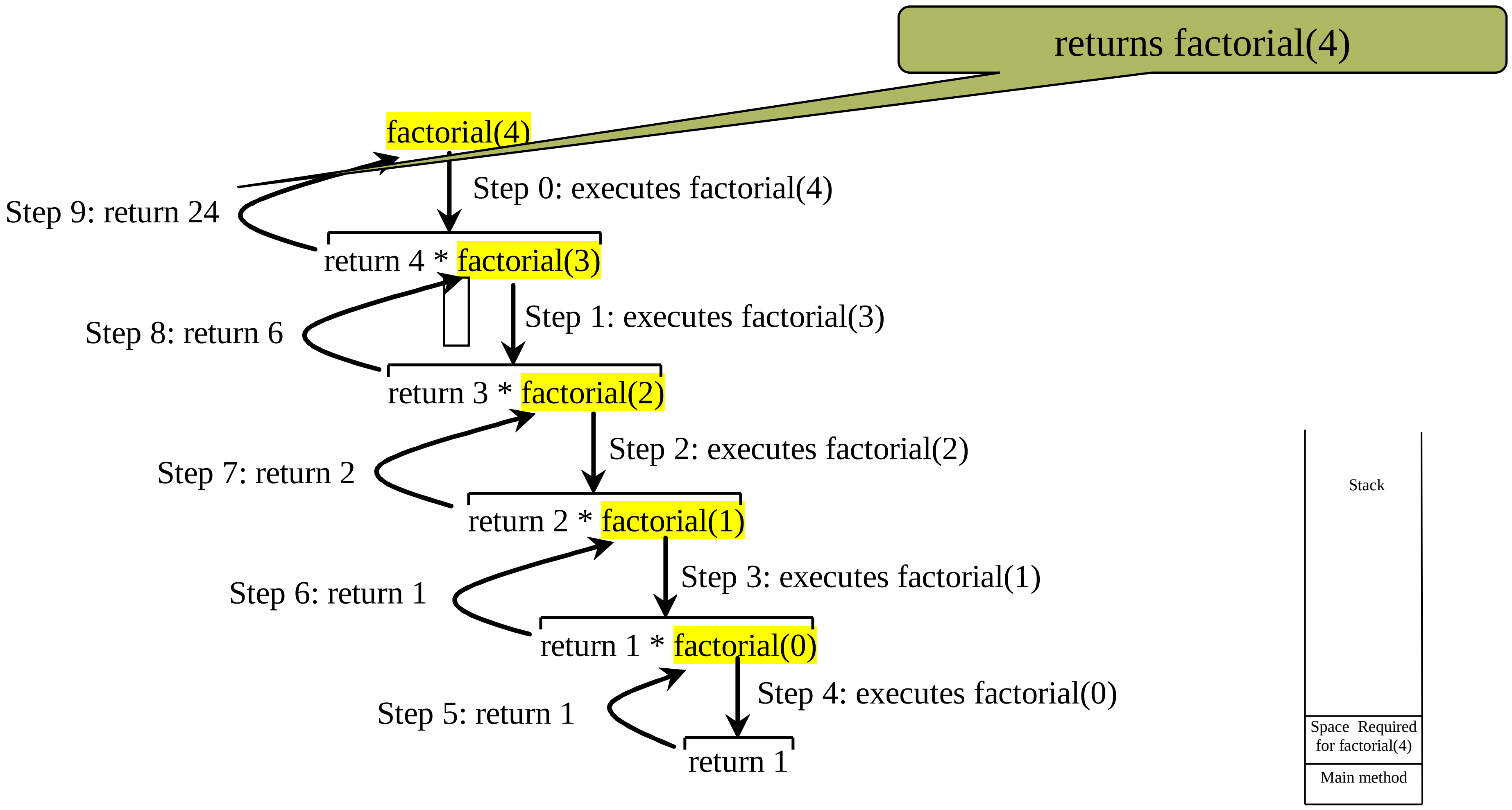
Trace Recursive Factorial



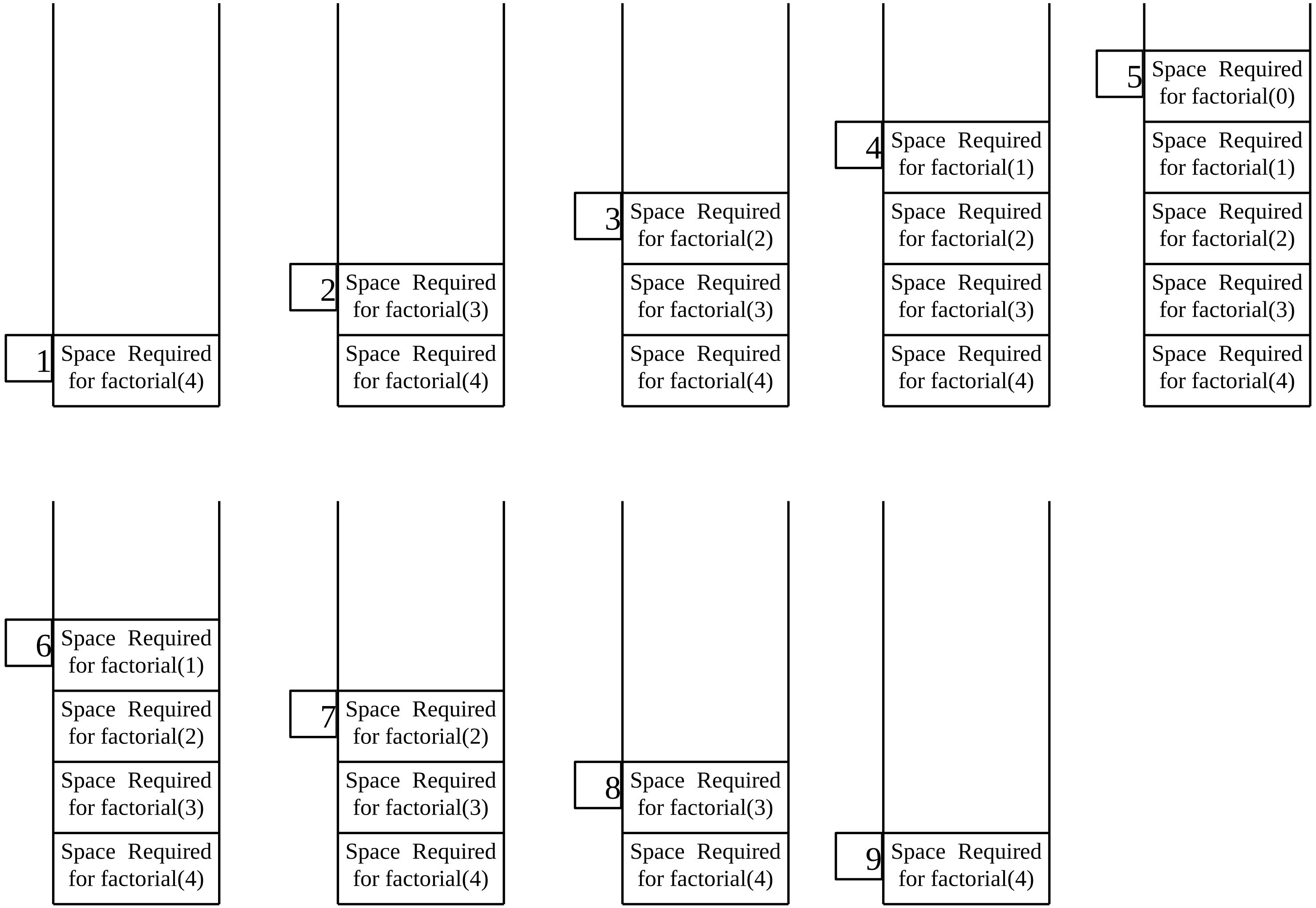
Trace Recursive Factorial



Trace Recursive Factorial



factorial(4) Stack Trace



Characteristics of Recursion

Characteristics of Recursion

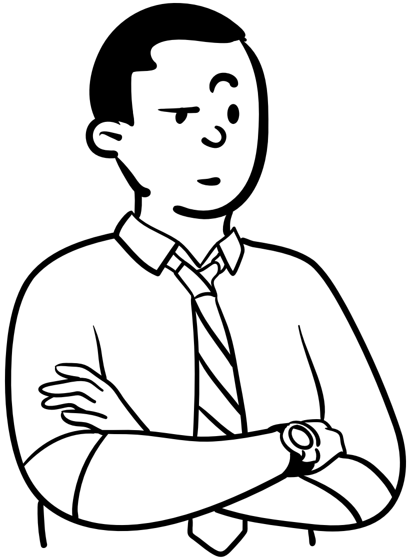
- All recursive methods have the following characteristics:

1. One or more **base cases** (bottom of recursion) are used to stop recursion.
2. Every **recursive call** reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Example:

Does it hold recursion characteristics????

```
public static void Main(string[] args)
{
    recursiveDemo(10);
}
2 references
public static void recursiveDemo(int i)
{
    if (i != 0)
    {
        i = i + 1;
        recursiveDemo(i);
    }
}
```



What will be the output of the following C# program?

Mini Exercise

```
class Recursion
{
    2 references
    public int Function(int n)
    {
        int result;
        result = Function(n - 1);
        return result;
    }
}

0 references
class Output
{
    0 references
    public static void Main(string[] args)
    {
        Recursion obj = new Recursion();
        Console.WriteLine(obj.Function(12));
    }
}
```

Do you see any problem?

Other Examples

https://abetterscientist.wordpress.com/2018/10/07/recursion-in-real-life/

Thinking Recursively: Other Examples

- With these characteristics, we can solve many problems using recursion.
- For instance, we can:
    - Printing a letter for a number of times
    - Calculating the Fibonacci series
    - Checking if a word is a Palindrome.

Example 1: Printing a message n times

```
public static void NPrintlnIterative(string message, int times) {
    for (int i = 0; i < times; i++) {
        Console.WriteLine(message);
    }
}
```

Printing a message n times: Recursive

NPrintln("Welcome", 5);

```
public static void NPrintln(string message, int times) {
  if (times >= 1) {
    Console.WriteLine(message);
    NPrintln(message, times - 1);
  }
}
```

Can you identify the base case?

Example 2: Fibonacci Sequence

Fibonacci Sequence

"Fibonacci sequence is such that each number in the sequence is the sum of the two preceding ones, starting from 0 and 1." – Wikipedia

```
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
```

$$\mathrm{Fib}(0) = 0;$$

$$\mathrm{Fib}(1) = 1;$$

$$\mathrm{Fib}(\mathrm{index}) = \mathrm{Fib}(\mathrm{index} - 1) + \mathrm{Fib}(\mathrm{index} - 2);\ \mathrm{index} >= 2$$

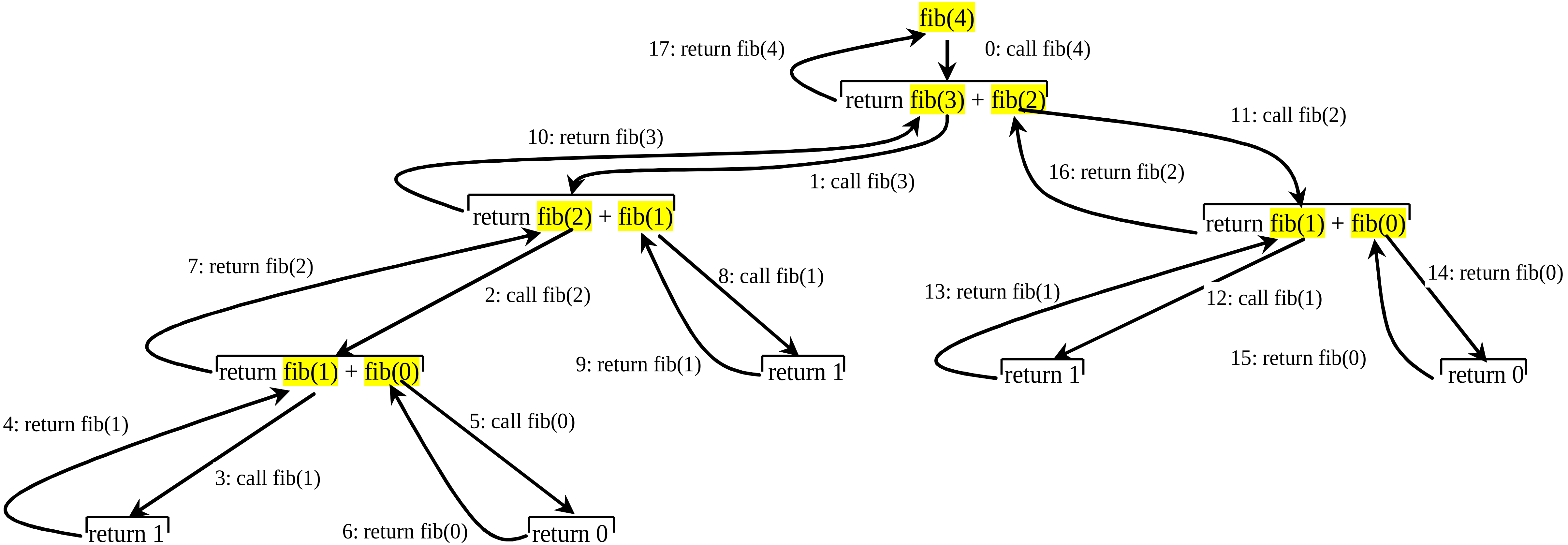
$$\begin{aligned}
\mathrm{Fib}(3) &= \mathrm{Fib}(2) + \mathrm{Fib}(1) \\
&= \left(\mathrm{Fib}(1) + \mathrm{Fib}(0)\right) + \mathrm{Fib}(1) \\
&= (1 + 0) + \mathrm{Fib}(1) \\
&= 1 + \mathrm{Fib}(1) \\
&= 1 + 1 \\
&= 2
\end{aligned}$$

Computing Fibonacci: Recursive

```
public static long ComputeFibonacci(long index) {
    if (index == 0)
        return 0;
    else if (index == 1)
        return 1;
    else
        return ComputeFibonacci(index - 1) + ComputeFibonacci(index - 2);
}
```

Fibonacci Numbers

Do you see any problem?



Demo: Recursion

From Plans-> VOP-9->VOP-9 (Lecture)-> Resources and Activities-> Recursion.zip-> Fibonacci

- IterativeFibonacci.cs
- RecursiveFibonacci.cs
- Program.cs

Example 3: Palindrome Problem

Palindrome Problem: Recursive

“A palindrome is a word, number, phrase, or other sequence of characters which reads the same backward as forward, such as mom, dad, madam, racecar.” - Wikipedia

```
public static bool IsPalindrome(string s)
{
    if (s.Length <= 1)
        return true;
    else if (s[0] != s[(s.Length - 1)])
    {
        return false;
    }
    else
    {
        return IsPalindrome(s.Substring(1, s.Length - 2));
    }
}
```

creates a new string

Recursive Helper Methods

- The IsPalindrome method is not efficient, because it creates a new string for every recursive call.
- To avoid creating new strings, use a helper method
- Recurvise helper methods usually takes more parameters than their primary methods

```
public static bool IsPalindrome(string s)
{
 return IsPalindrome(s, 0, s.Length - 1);
}
```

Recursive helper method

```
public static bool IsPalindrome(string s, int low, int high)
{
    if (high <= low) // Base case
        return true;
    else if (s[low] != s[high]) // Base case
        return false;
    else
        return IsPalindrome(s, low + 1, high - 1);
}
```

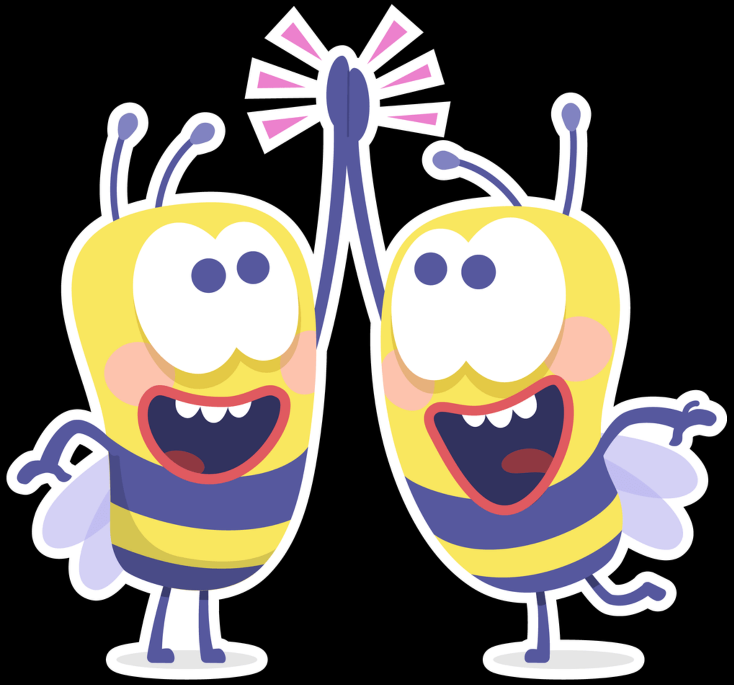
What is special about **IsPalindrome()** method?

Demo: Recursion

From Plans-> VOP-9->VOP-9 (Lecture)-> Resources and Activities-> Recursion.zip-> IsPalindrome

- IsIterativePalindrome.cs
- IsRecursivePalindrome.cs
- Program.cs

Break (10 min)



Recursion: Selection Sort

Recursion for Sorting

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

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

Selection sort example

Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	**-4**	18	12	**22**	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	**2**	12	22	27	30	36	50	7	68	91	56	**18**	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	**7**	22	27	30	36	50	**12**	68	91	56	18	85	42	98	25

Selection Sort example

```
public static void Sort(int[] intArray) {
    Sort(intArray, 0, intArray.Length);
}

private static void Sort(int[] intArray, int low, int high) {
    if (low < high) {
        int indexOfMin = low;
        int min = intArray[low];
        for (int i = low + 1; i < high; i++) {
            if (intArray[i] < min) {
                min = intArray[i];
                indexOfMin = i;
            }
        }
        // SWAP
        intArray[indexOfMin] = intArray[low];
        intArray[low] = min;
        Sort(intArray, low + 1, high);
    }
}
```

Can you spot a **primary** method and a **recursive** helper method?

SDU

Demo: Recursion

From Plans-> VOP-9->VOP-9 (Lecture)-> Resources and Activities-> Recursion.zip-> SelectionSort

- IterativeSelectionSort.cs
- RecursiveSelectionSort.cs
- Program.cs

Recursion: Binary Search

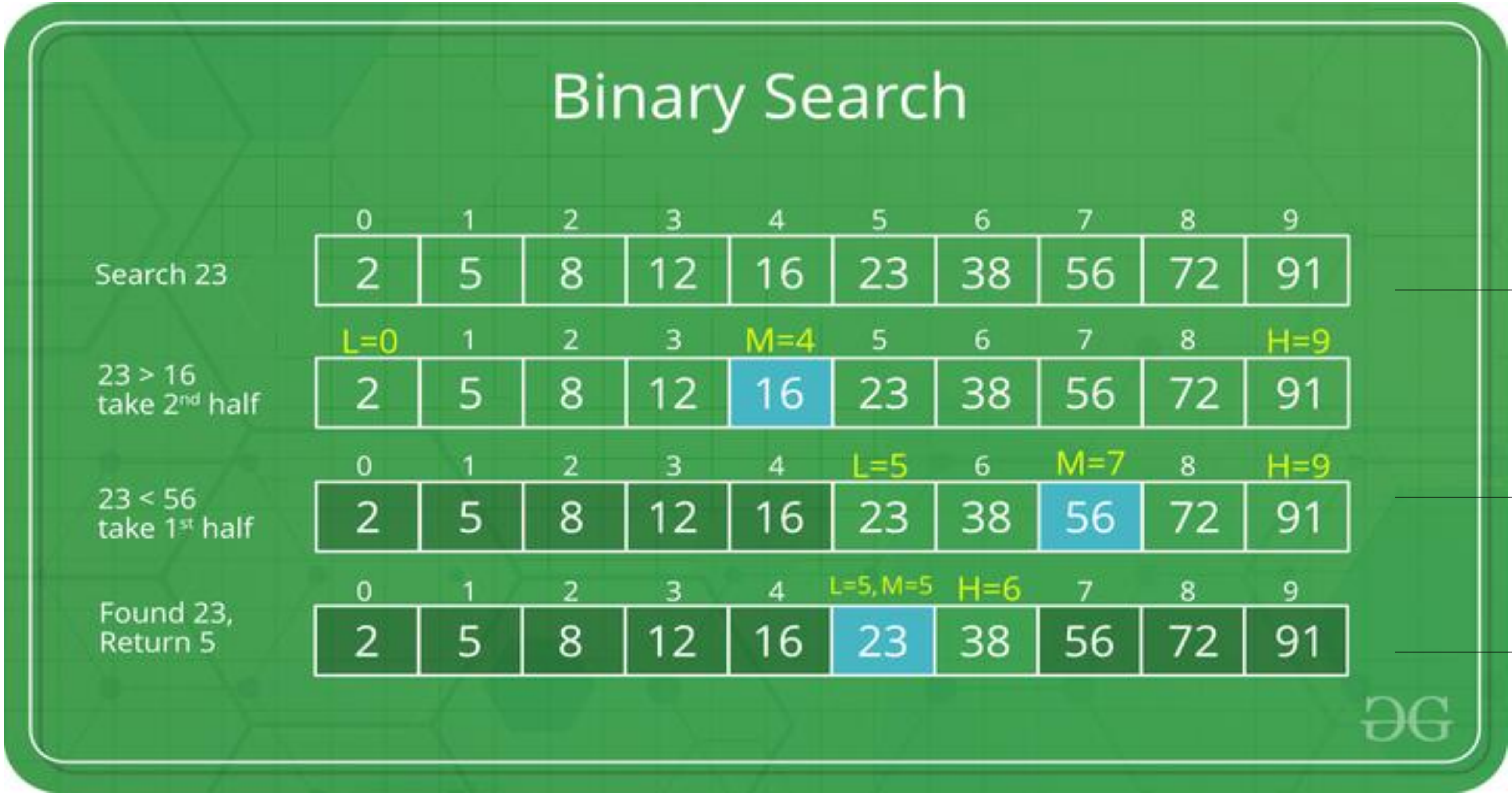
- The elements in the array must be in **increasing** order.
- The binary search first compares the key with the element in the middle of the array.
- Example:

2	5	8	12	16	23	38	56	72	91

Recursion for Binary Search

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

Illustration of Binary Search Algorithm



$M = \frac{0+9}{2}$

M= 4.5

M=4

$M = \frac{5+9}{2}$

M= 7

$M = \frac{5+6}{2}$

M= 5.5

M=5

https://www.geeksforgeeks.org/binary-search/

Binary Search Algorithm

```
public static int Find(int[] list, int key) {
    int low = 0;
    int high = list.Length - 1;
    return Find(list, key, low, high);
}

public static int Find(int[] list, int key, int low, int high) {
    if (low > high) {
        return -low - 1;                    // Search is exhausted
    }

    int mid = (low + high) / 2;             // Compute middle of the array
    if (key < list[mid]) {
        return Find(list, key, low, mid - 1);    // Find key in first half of the array
    } else if (key == list[mid]) {
        return mid;
    } else {
        return Find(list, key, mid + 1, high);   // Find key in 2nd half of the array
    }
}
```


https://www.geeksforgeeks.org/binary-search/

Demo: Recursion

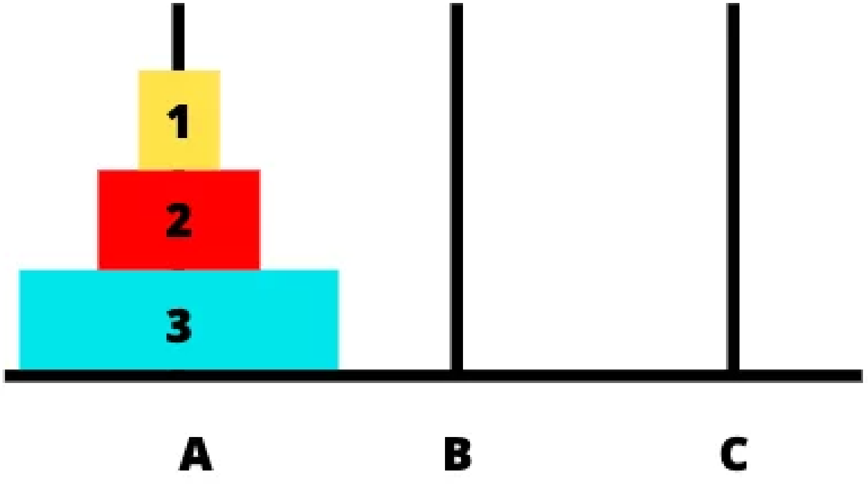
From Plans-> VOP-9->VOP-9 (Lecture)-> Resources and Activities-> Recursion.zip-> BinarySearch

- IterativeBinarySearch.cs
- RecursiveBinarySearch.cs
- Program.cs

Case Study

Tower of Hanoi

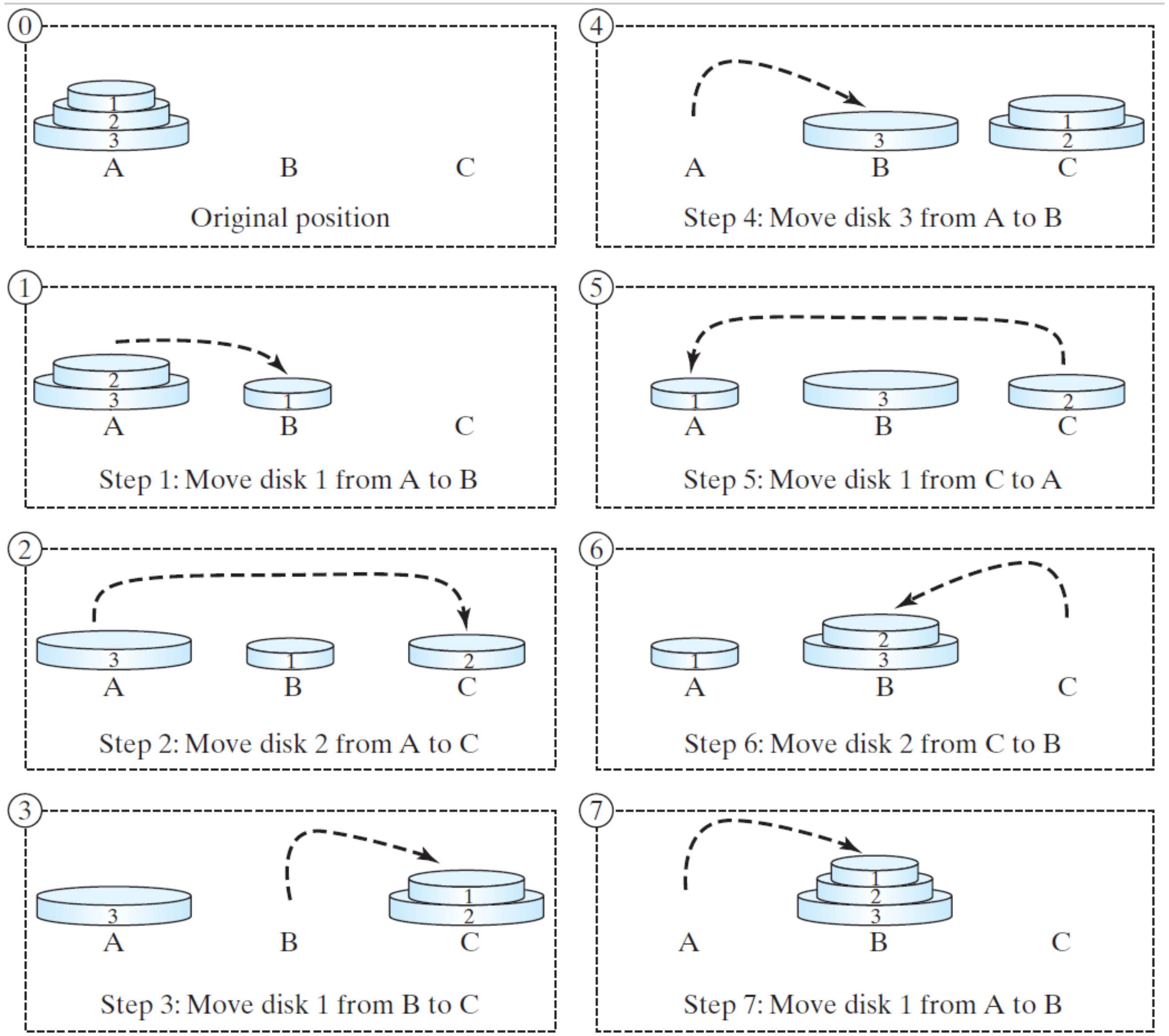
Goal: Move all the disks from Tower A to Tower B



- There are *n* disks labeled 1, 2, 3, . . ., *n*, and three towers labeled A, B, and C.
- No bigger 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.

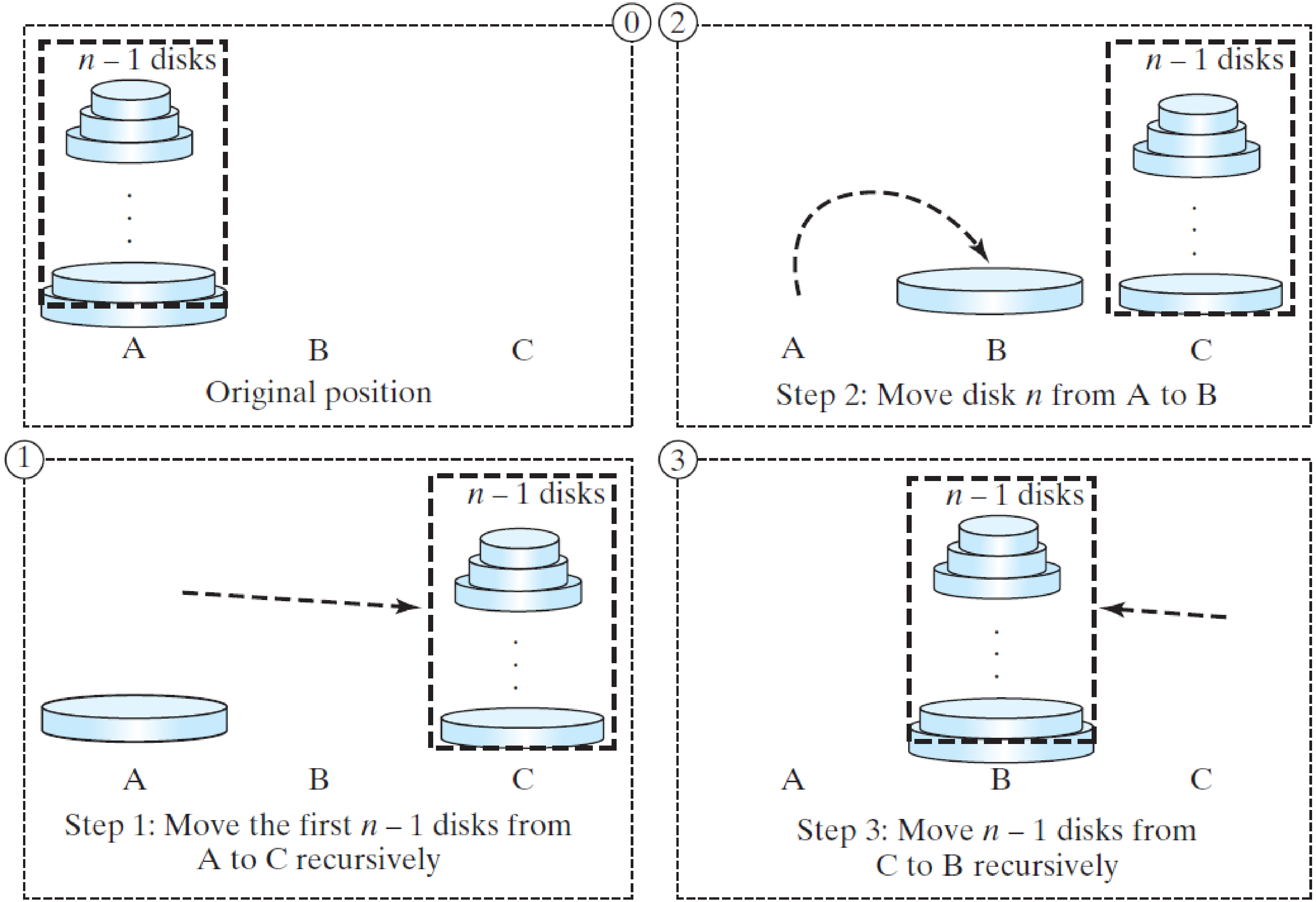
Tower of Hanoi, cont.

Animation



Solution to Tower of Hanoi

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



SDU

Solution to Tower of Hanoi:

- 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:

```
public class TowerOfHanoi
{
    public static void Main()
    {
        Console.Write("Enter number of disks: ");
        int n = int.Parse(Console.ReadLine());

        Console.WriteLine("The moves are:");
        MoveDisks(n, 'A', 'B', 'C');
    }

    public static void MoveDisks(int n, char fromTower,
        char toTower, char auxTower)
    {
        if (n == 1)
            Console.WriteLine("Move disk " + n + " from " +
              fromTower + " to " + toTower);
        else
        {
            MoveDisks(n - 1, fromTower, auxTower, toTower);
            Console.WriteLine("Move disk " + n + " from " +
              fromTower + " to " + toTower);
            MoveDisks(n - 1, auxTower, toTower, fromTower);
        }
    }
}
```

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.

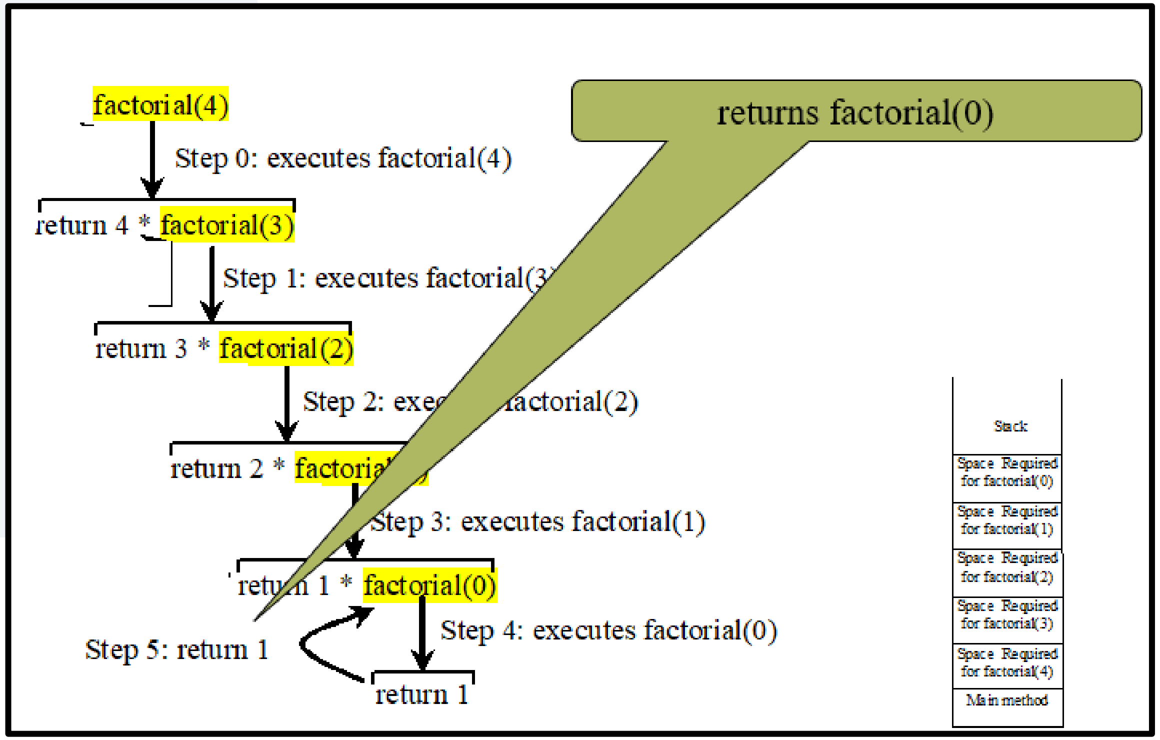
Note:

- C# Common Language Runtime (CLR) supports Tail Call Optimization (TCO), but the C# compiler doesn't reliably perform it in all cases.
- Therefore, C# support for tail recursion is spotty.

Non-tail Recursive: Factorial

```
public class ComputeFactorial
{
    public static void Main()
    {
        Console.Write("Enter a non-negative integer: ");
        string input = Console.ReadLine();
        int n = int.Parse(input);
        Console.WriteLine("Factorial of " + n + " is " + Factorial(n));
    }

    public static long Factorial(int n)
    {
        if (n == 0)
            return 1;
        else
            return n * Factorial(n - 1);
    }
}
```



Tail Recursive: Factorial

```
public class ComputeFactorialTailRecursion
{
    public static void Main()
    {
        Console.Write("Enter a non-negative integer: ");
        int n = int.Parse(Console.ReadLine());
        Console.WriteLine("Factorial of " + n + " is " + Factorial(n));
    }

    public static long Factorial(int n)
    {
        return Factorial(n, 1);
    }

    private static long Factorial(int n, int result)
    {
        if (n == 0)
            return result;
        else
            return Factorial(n - 1, n * result);
    }
}
```

How Tail Recursion works

Assume you call **Factorial(5)**

1.First Call: Factorial(5, 1)

Instead of pushing a new frame, the current frame is **updated** with new values (n = 4, result = 5).

2.Second Call: Factorial(4, 5)

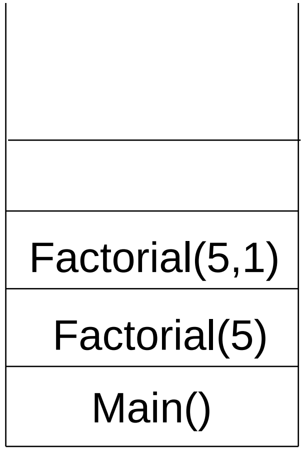
The same stack frame is **reused**, avoiding extra memory consumption.

3.Third Call: Factorial(3, 20)

Again, **no additional stack usage**.

4.Final Call: Factorial(0, 120)

The function directly **returns 120** without needing to unwind any stack.



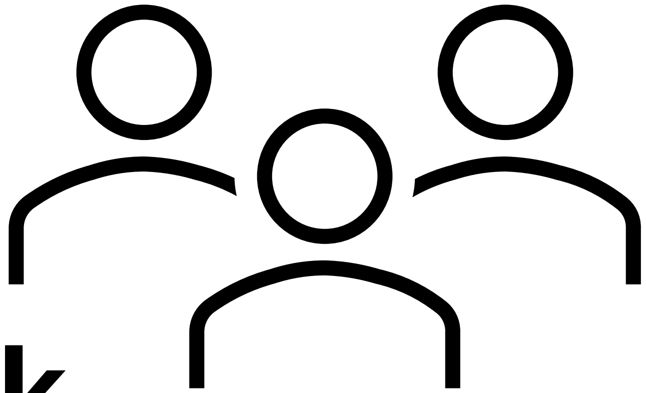
Method call Stack

Take Aways???

Take Aways

- Recursion is an alternative form of program control.
- Recursion is good for solving the problems that are inherently recursive.
- Recursion bears substantial overhead
- The decision to use recursion or iteration should be based on the nature of the problem and your understanding of the problem.

Class Activity + Homework



(group based)

Class Activity + Homework

From Plans-> VOP-9->VOP-9 (Lecture)-> Resources and Activities-> Recursion.zip-> Exercises

Please read the instructions given in “ReadMe.md”

- Exercise1.cs
- Exercise2.cs
- Exercise3.cs
- Homework.cs
- Program.cs

You can find the solution to the exercises here:

From Plans-> VOP-9->VOP-9 (Lecture)-> Resources and Activities-> Recursion.zip-> ExercisesSolution

MCQs Quiz

Go to Plans -> VOP-9 -> VOP-9 (Lecture) -> Lecture-9 Test

Good Luck ☺