RECURSION

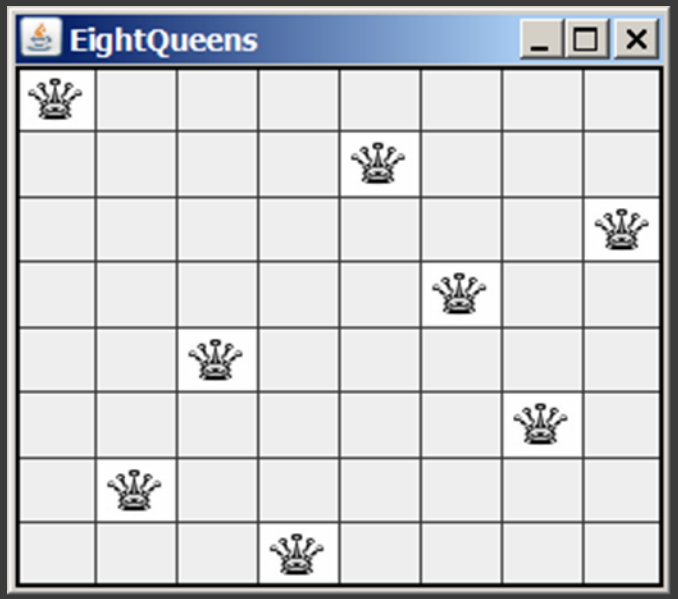
in order to understand recursion, one must first understand recursion, in order to understand recursion, one must first understand recursion,

Motivations

- Suppose you want to find all the files under a directory that contains a particular word. There are several ways to solve this problem. An intuitive solution is to use *recursion* by searching the files in the subdirectories *recursively*.

Motivations

- The Eight Queens puzzle:



Recursive Thinking

- A *recursive definition* is one which uses the word or concept being defined in the definition itself
- But in other situations, a recursive definition can be an appropriate way to express a concept
- Before applying recursion to programming, it is best to practice thinking recursively

Computing Factorials

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

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

ComputeFactorial

Computing Factorials

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

factorial(3)

Computing Factorials

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

factorial(3) = 3 * factorial(2)

Computing Factorials

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

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

Computing Factorials

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

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

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

Computing Factorials

```
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 Factorials

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 Factorials

```
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)

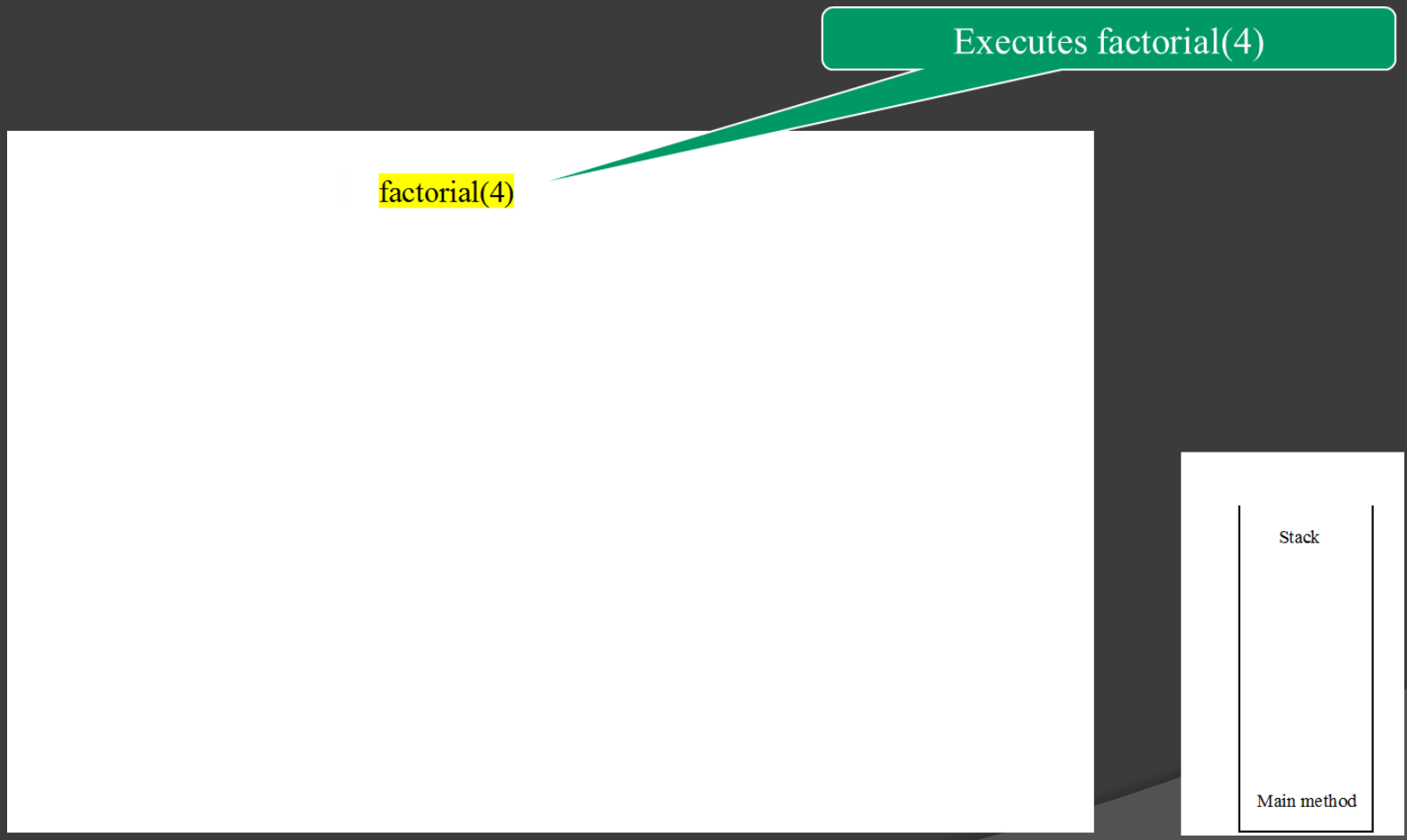
= 3 * 2

Computing Factorials

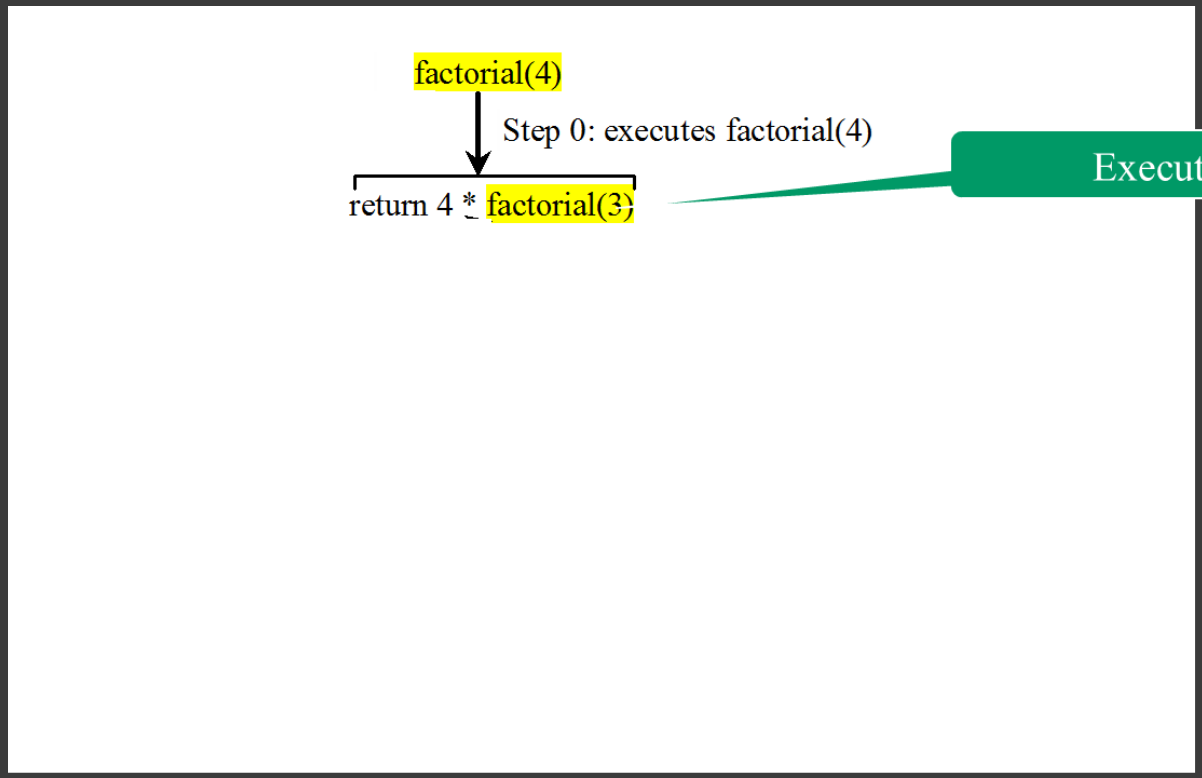
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

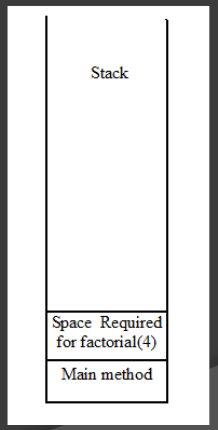
factorial(3) = 3 * factorial(2)
= 3 * (2 * factorial(1))
= 3 * (2 * (1 * factorial(0)))
= 3 * (2 * (1 * 1)))
= 3 * (2 * 1)
= 3 * 2
= 6

Trace Recursive Factorial

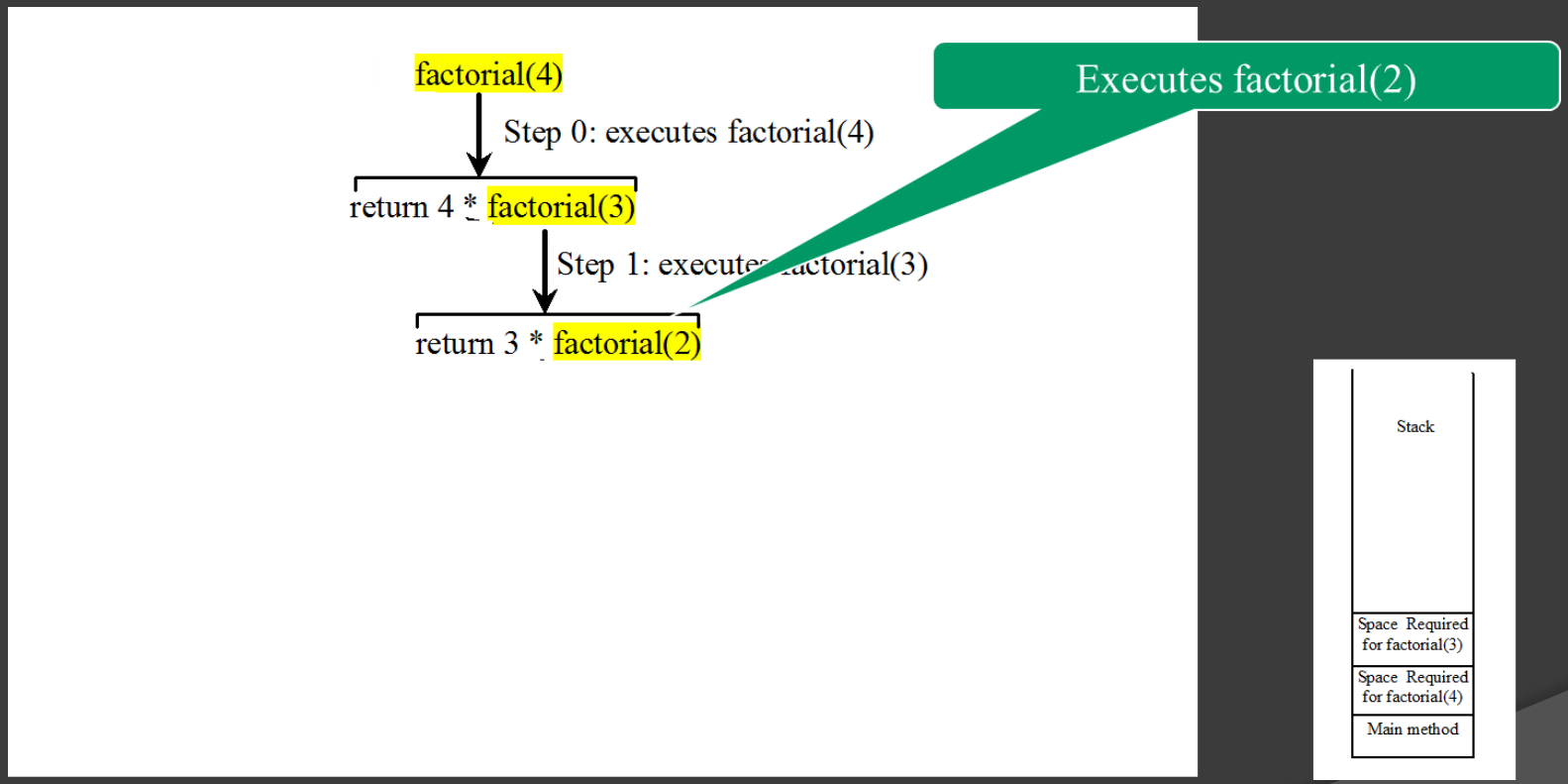


Trace Recursive Factorial

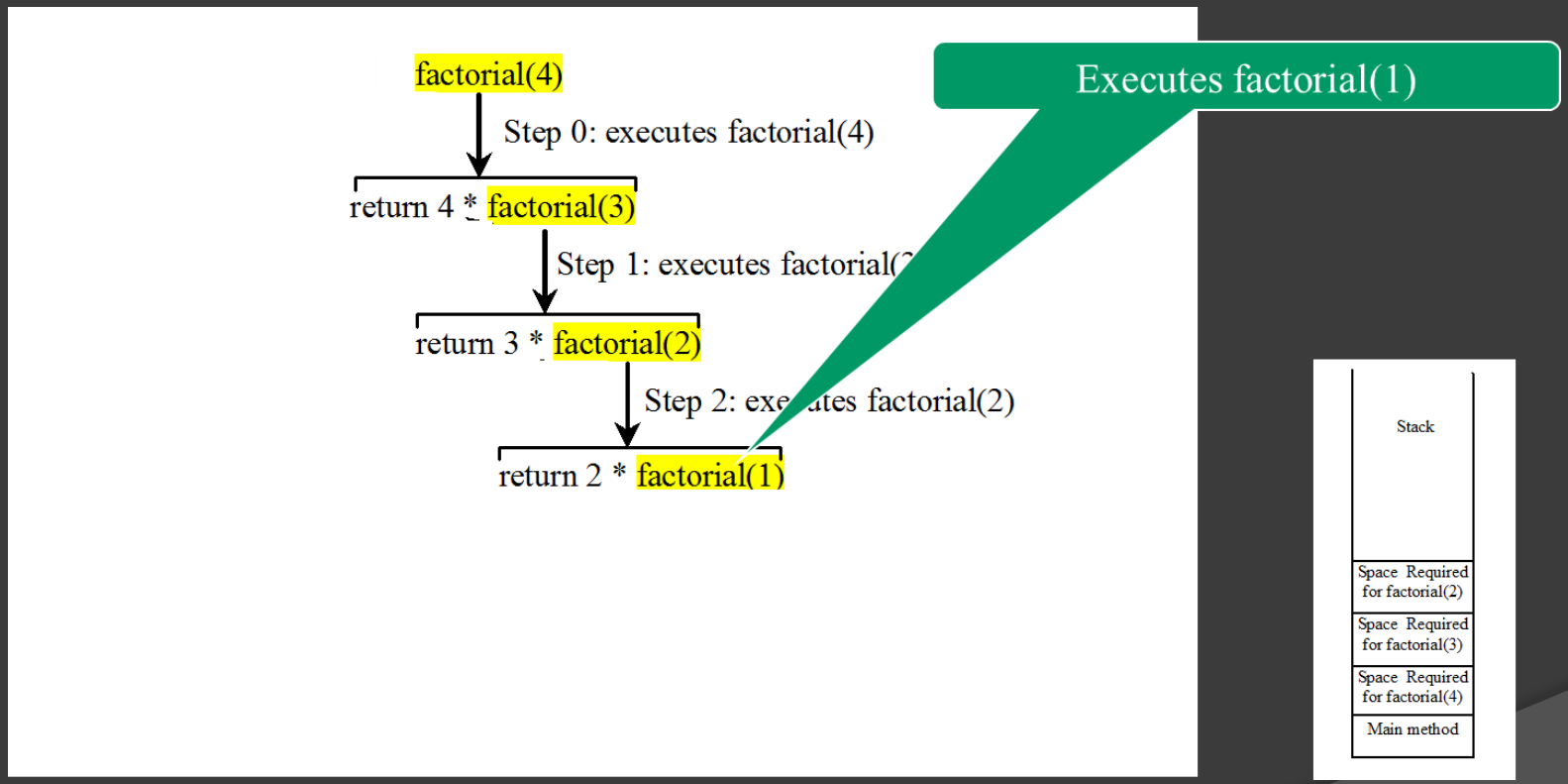




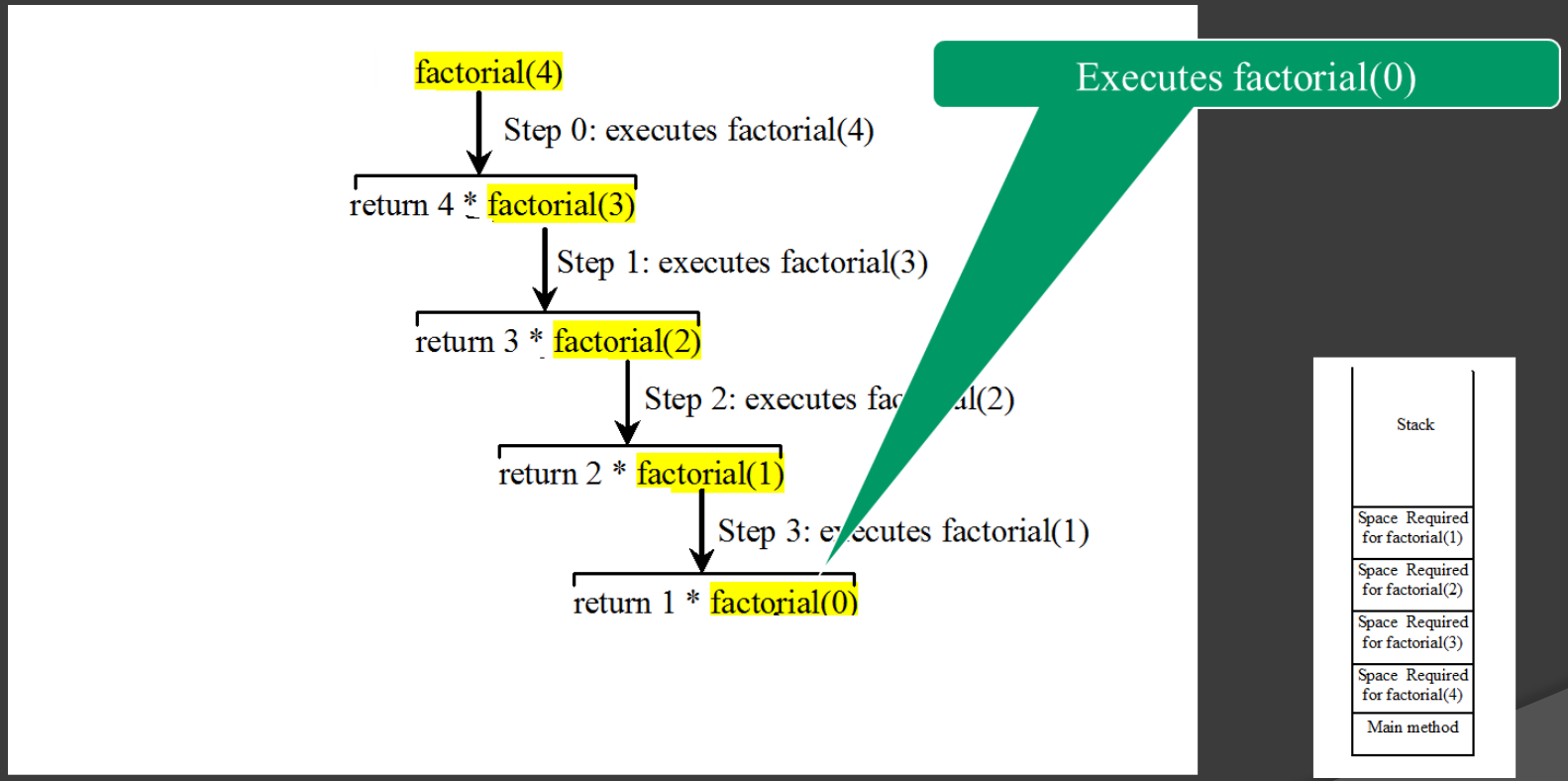
Trace Recursive Factorial



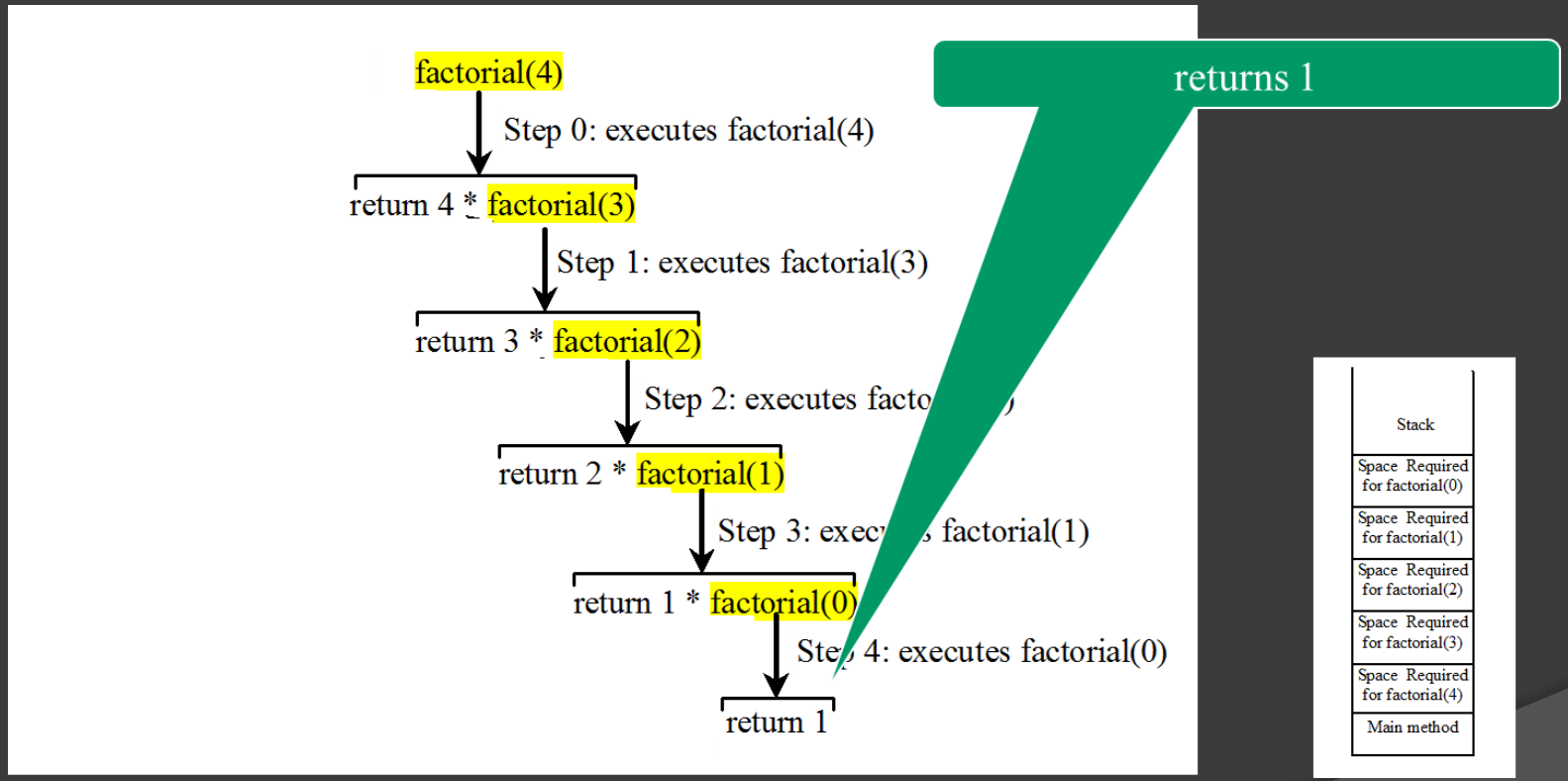
Trace Recursive Factorial



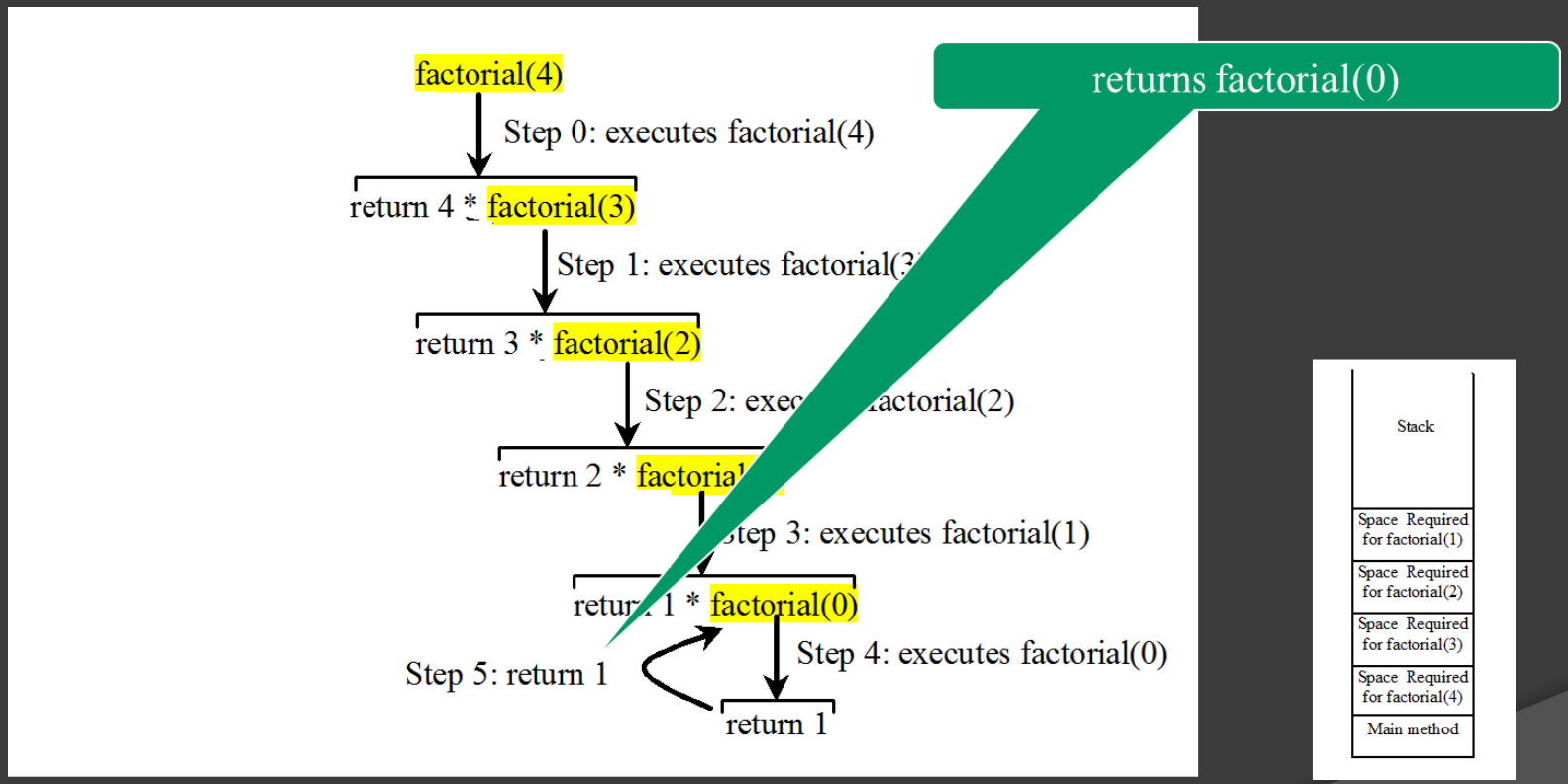
Trace Recursive Factorial



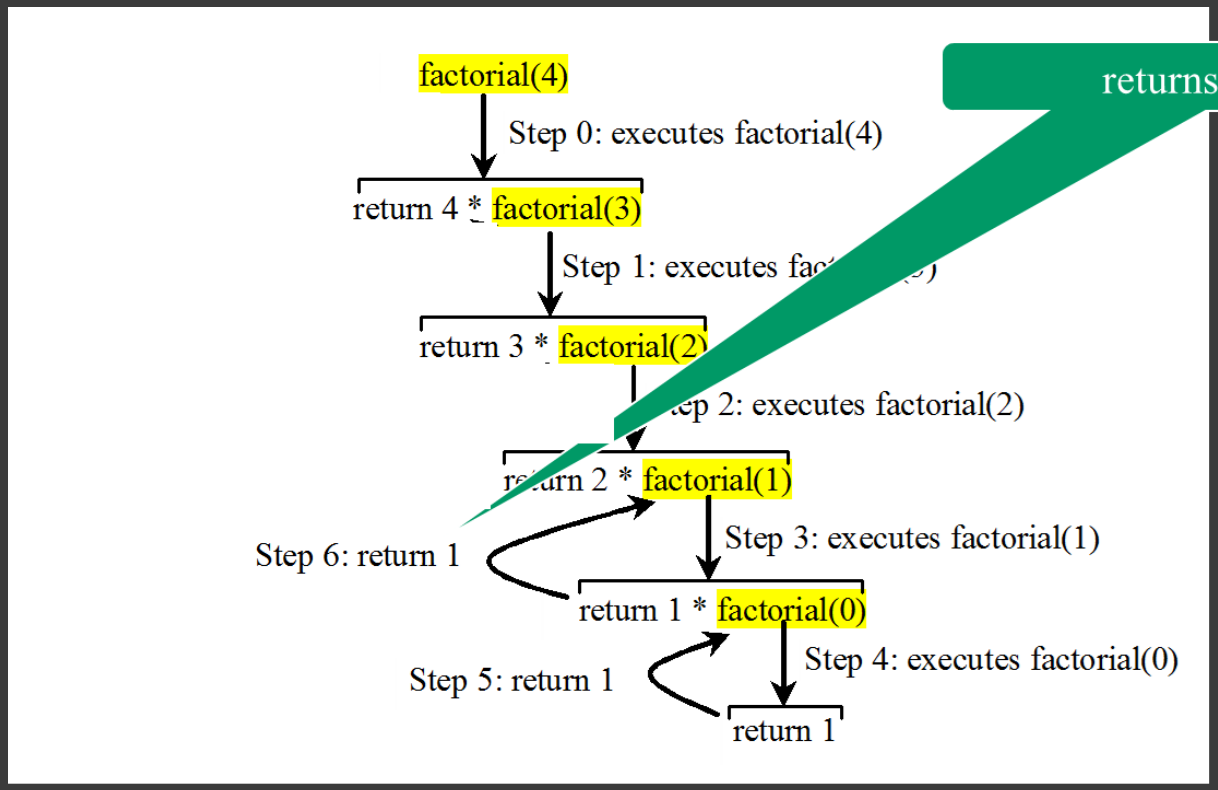
Trace Recursive Factorial

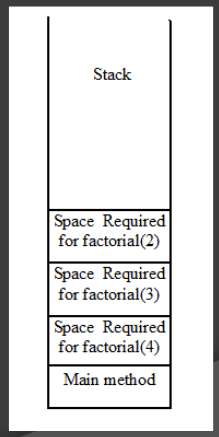


Trace Recursive Factorial

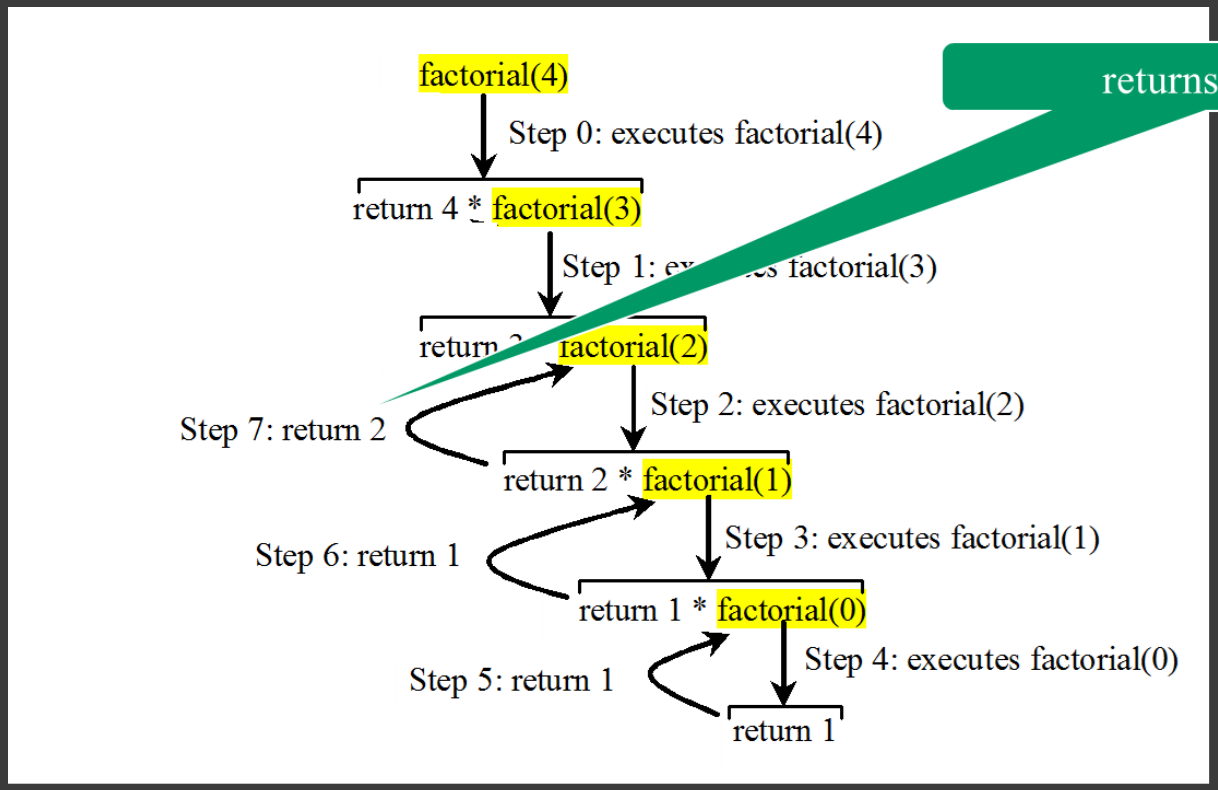


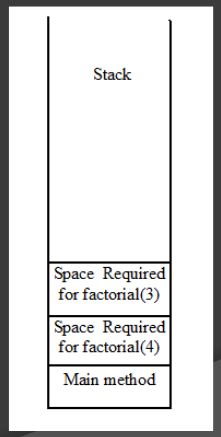
Trace Recursive Factorial



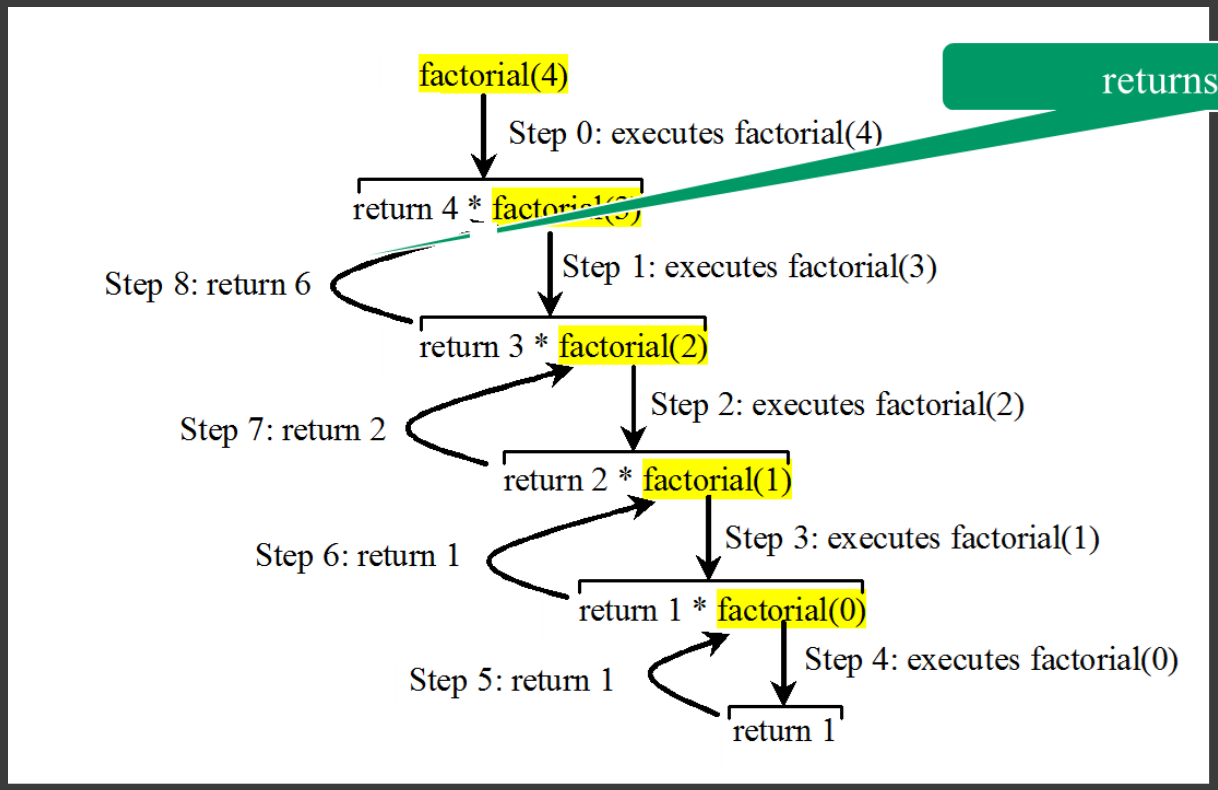


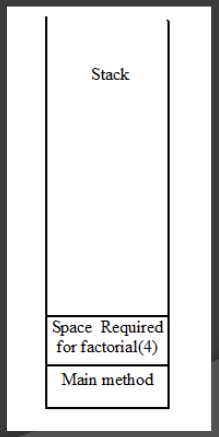
Trace Recursive Factorial



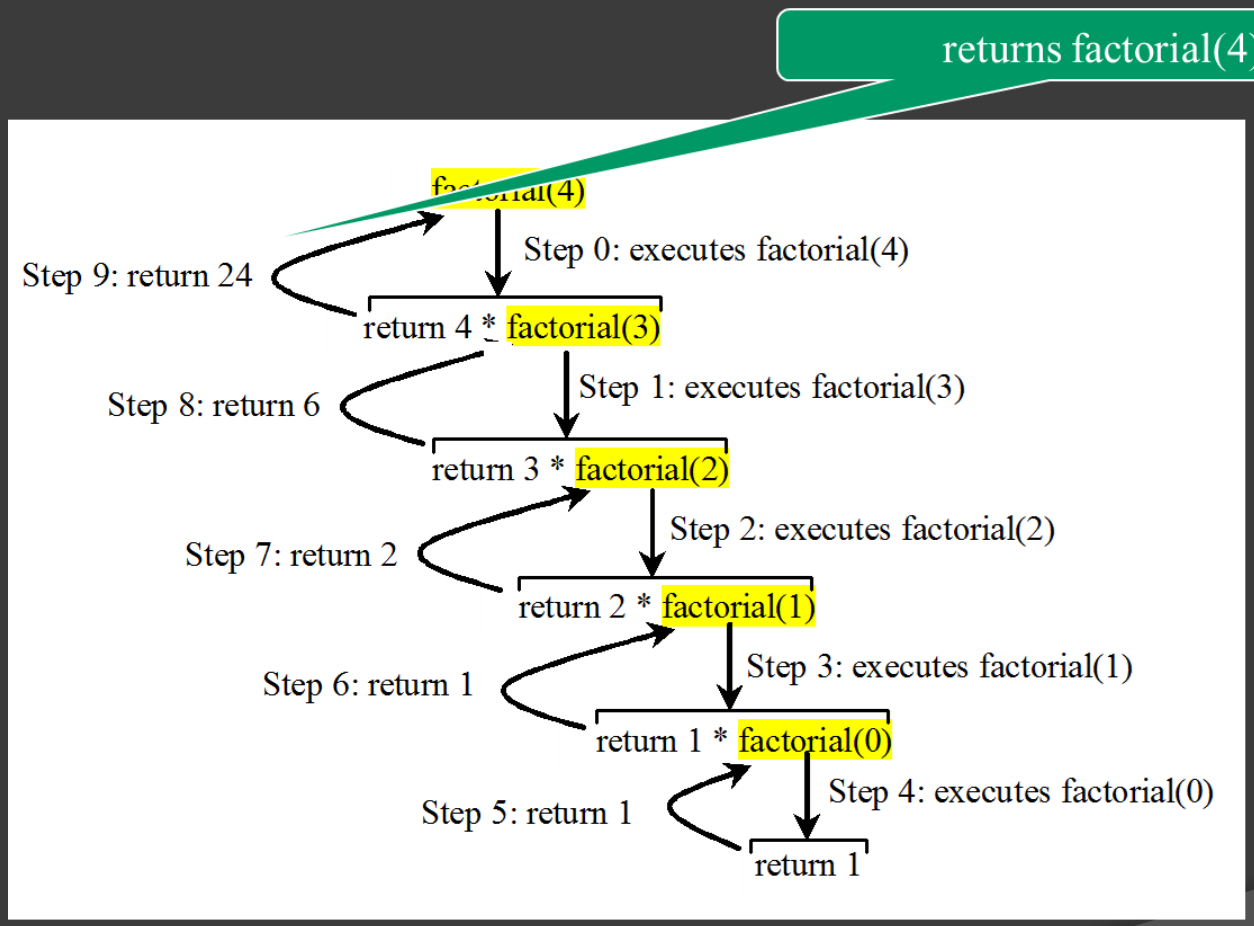


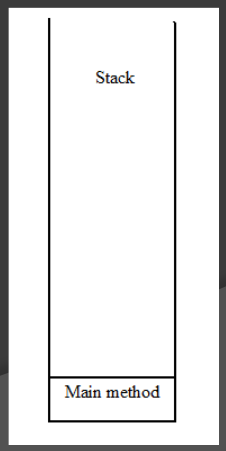
Trace Recursive Factorial



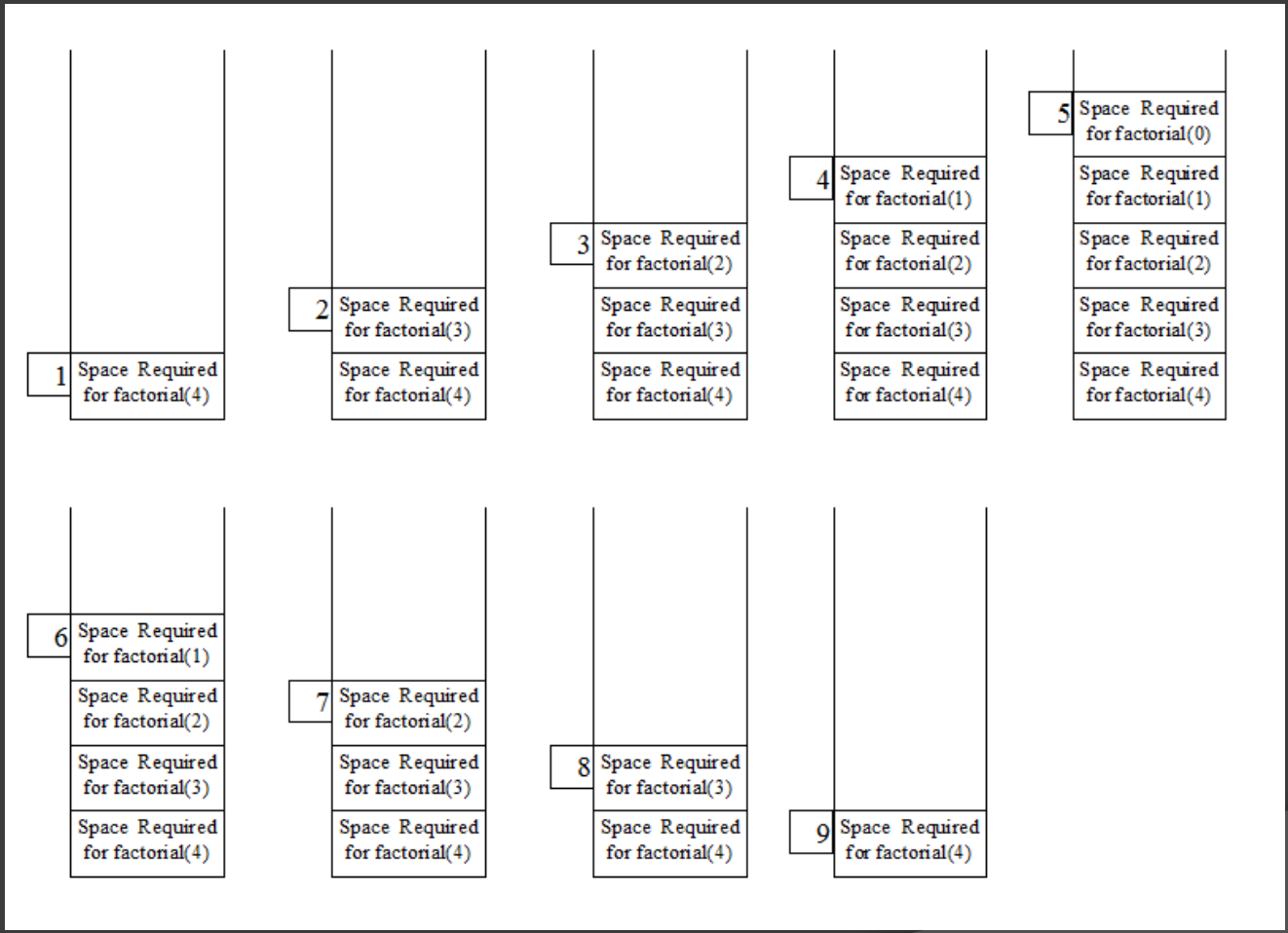


Trace Recursive Factorial





factorial(4) Stack Trace



Characteristics of Recursion

- The method is implemented using an if-else or a switch statement that leads to different cases.
- 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.

Non-Recursive Sorting

```
for (int i = 0; i < list.length; i++)
{
  select the smallest element in list[i..listSize-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i..listSize-1]
}
```

Non-Recursive Sorting

```
/** The method for sorting the numbers */
public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length; i++) {
      // Find the minimum in the list[i..list.length-1]
      double currentMin = list[i];
      int currentMinIndex = i;
      for (int j = i + 1; j < list.length; j++) {
        if (currentMin > list[j]) {
          currentMin = list[j];
          currentMinIndex = j;
        }
      }
      // Swap list[i] with list[currentMinIndex] if necessary;
      if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
      }
    }
  }
```

Non-Recursive Sorting

- Total number of elements examined is:

$T(n) = 2*(n-1) + 2*(n-2) + 2*(n-3) + \ldots + 2*(n-(n-2)) + 2*(n-(n-1))$

$= 2*((n-1) + (n-2) + (n-3) + \ldots + 2 + 1)$ (or 2*(sum of first n-1 ints)

$= 2*((n-1)*n)/2) = n^2 – n$, so the algorithm is $O(n^2)$

- The Selection Sort algorithm has deterministic complexity
- -the number of operations does not depend on specific items, it depends only on the number of items
- -all possible instances of the problem (“best case”, “worst case”, “average case”) give the same number of operations:

$T(n)=n^2–n=O(n^2)$

Non-Recursive Sorting

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[1]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest item in an unsorted array
"n log star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\varepsilon})$ for all $\varepsilon > 0$	$O(2^{\log n^{\log \log n}})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[2]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/2}}$	Best-known algorithm for integer factorization and graph isomorphism
exponential time	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

Quicksort

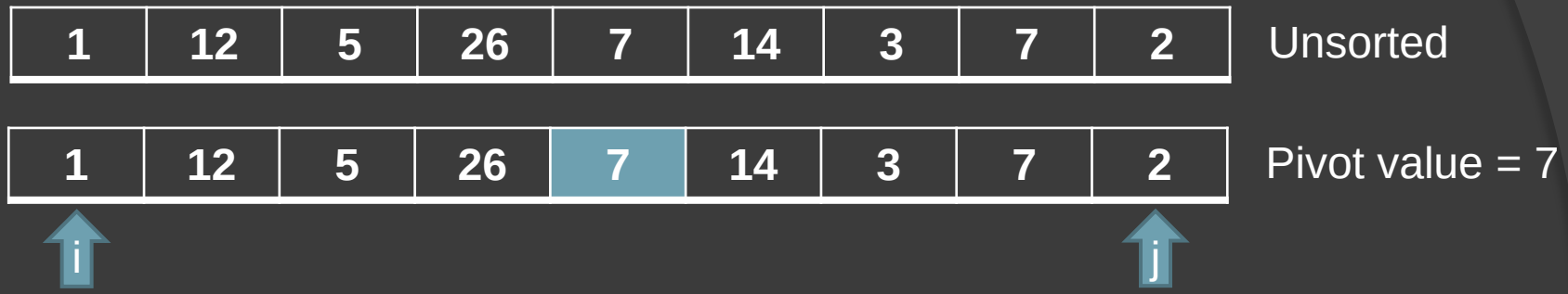
- **Choose a pivot value.** We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it isn't present in the array.
- **Partition.** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
- **Sort both parts.** Apply quicksort algorithm *recursively* to the left and the right parts.

Quicksort

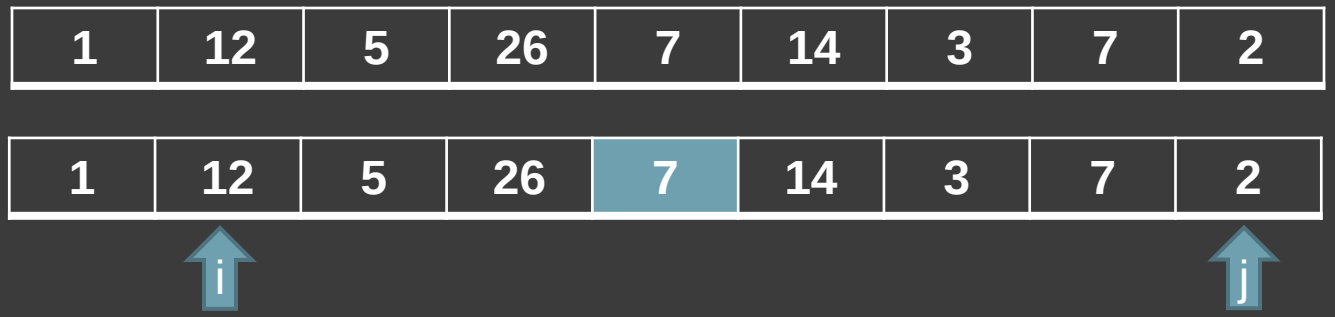
1	12	5	26	7	14	3	7	2

Unsorted

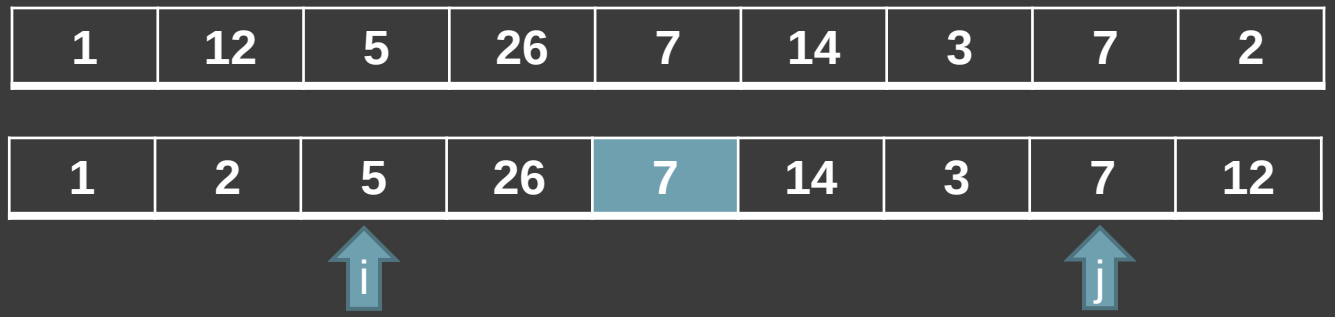
Quicksort



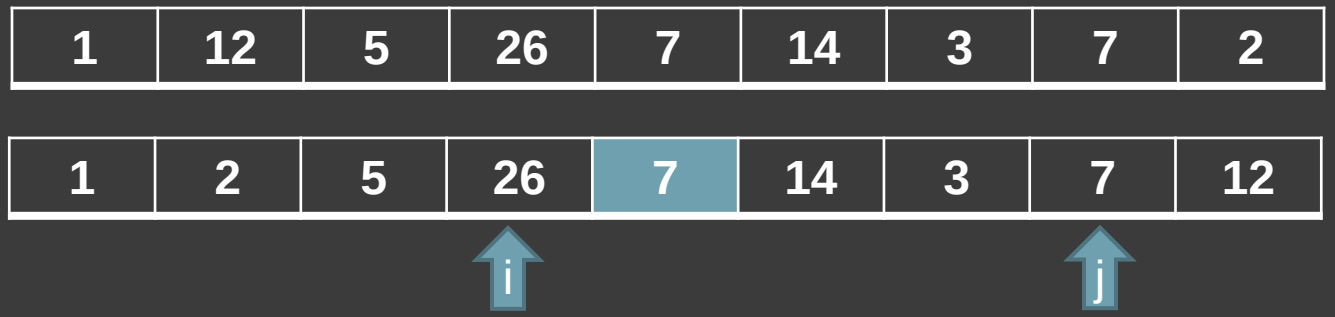
Quicksort



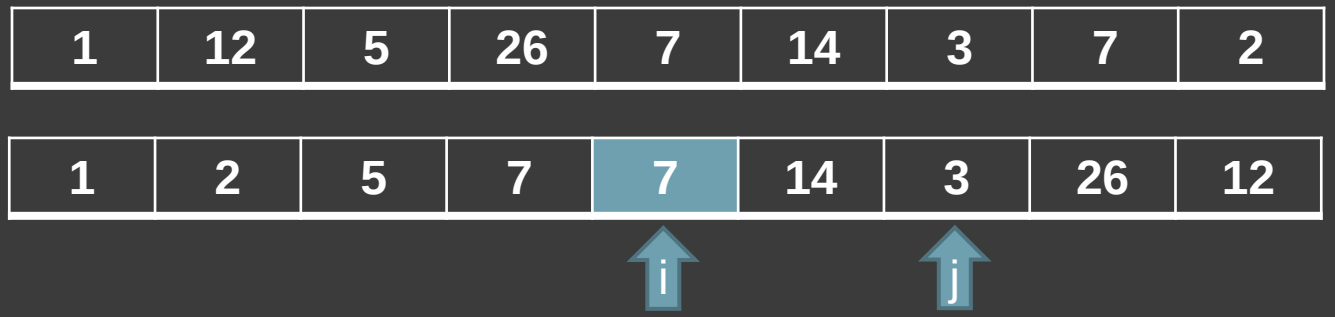
Quicksort



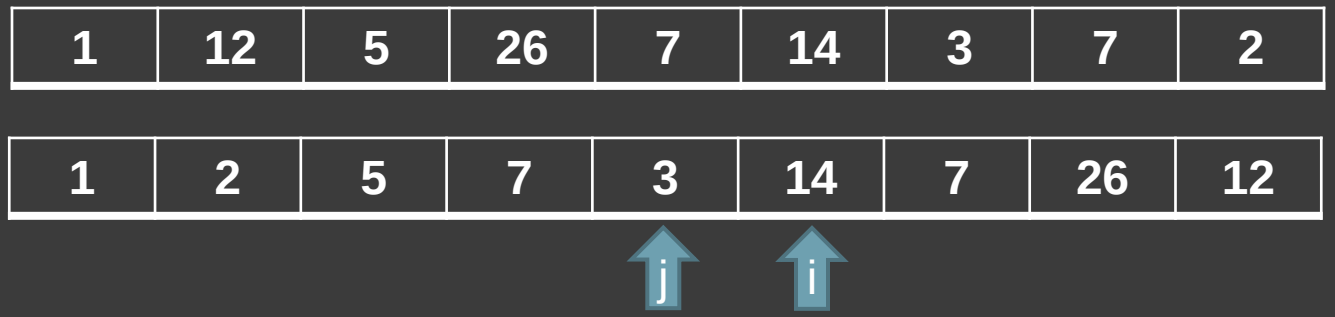
Quicksort



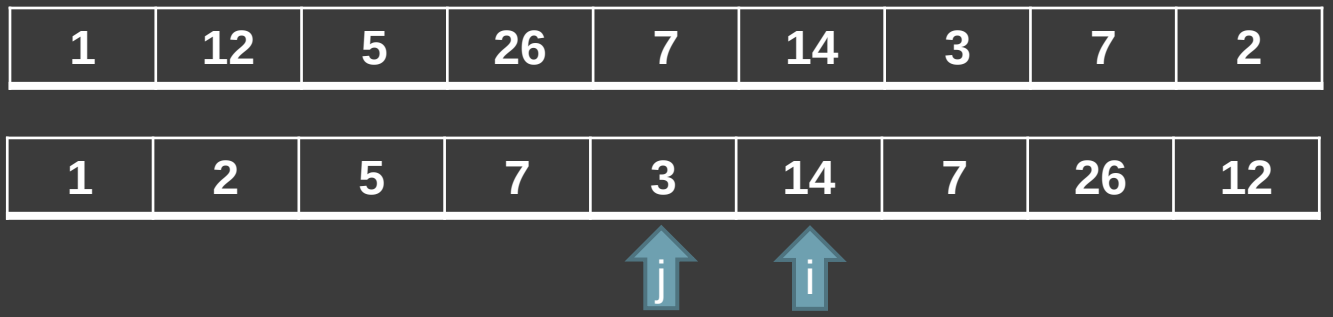
Quicksort



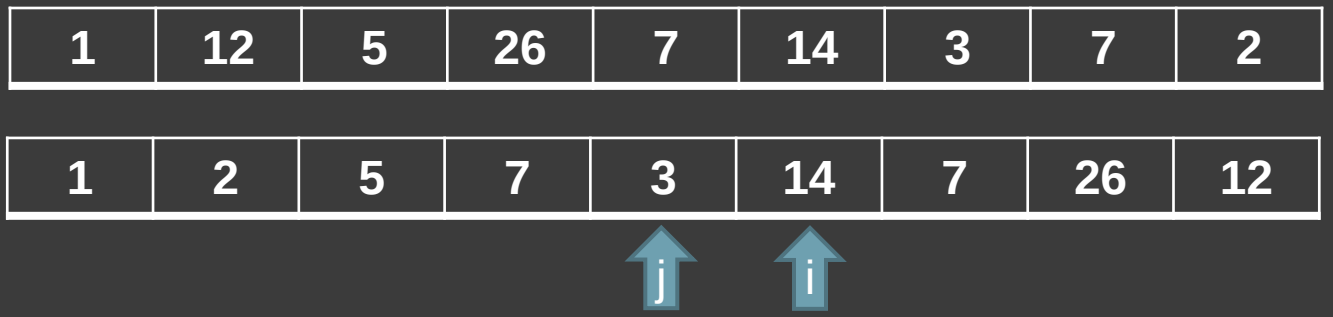
Quicksort



Quicksort


1 12 5 26 7 14 3 7 2
Unsorted
1 2 5 7 3 14 7 26 12
i > j, stop partition
j
i
1 2 5 7 3
14 7 26 12
Run Quicksort recursively

Quicksort


1 12 5 26 7 14 3 7 2
Unsorted
1 2 5 7 3 14 7 26 12
i > j, stop partition
j
i
1 2 5 7 3
14 7 26 12
Run Quicksort recursively
. . .
1 2 3 5 7 7 12 14 26
Sorted