CSS-105: Fundamentals of Programming (C++)

Lecture 10: Recursion

Madina Sultanova

madina.sultanova@sdu.edu.kz

Computing Factorial

factorial(0) = 1;

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

n! = n * (n-1)!

ComputeFactorial

Run

animation

Computing Factorial

factorial(0) = 1;

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

factorial(4)

animation

Computing Factorial

factorial(0) = 1;

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

factorial(4) = 4 * factorial(3)

animation

Computing Factorial

factorial(0) = 1;

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

factorial(4) = 4 * factorial(3)

= 4 * 3 * factorial(2)

animation

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))

animation

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)))

animation

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)))

animation

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)

animation

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

animation

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

animation

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

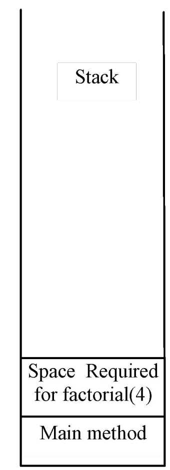
animation

Trace Recursive factorial

Executes factorial(4)

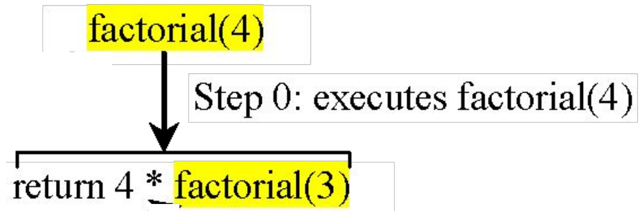
factorial(4)

0)

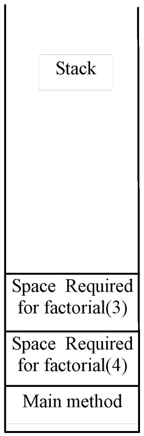


animation

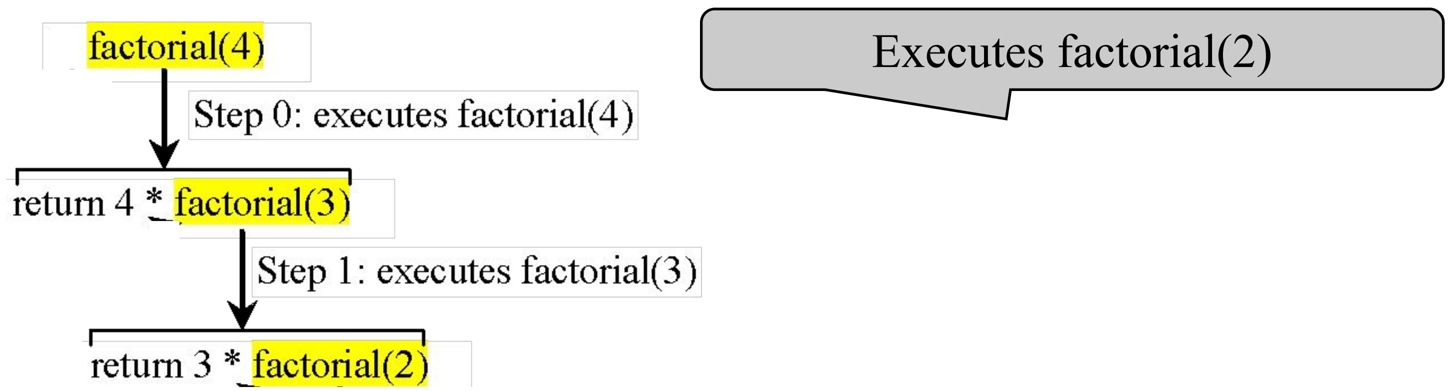
Trace Recursive factorial

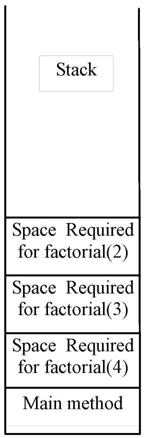


Executes factorial(3)



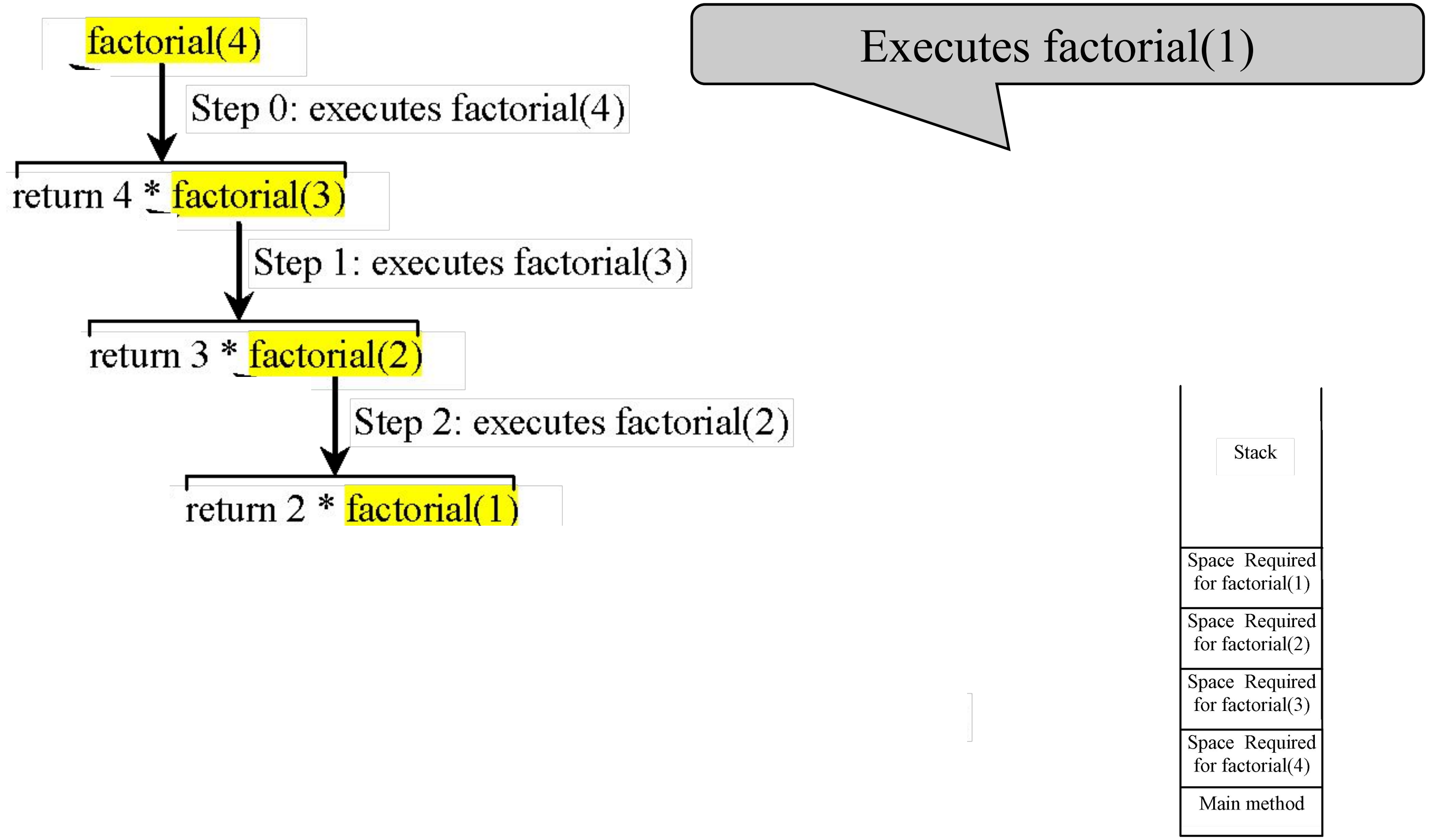
Trace Recursive factorial





animation

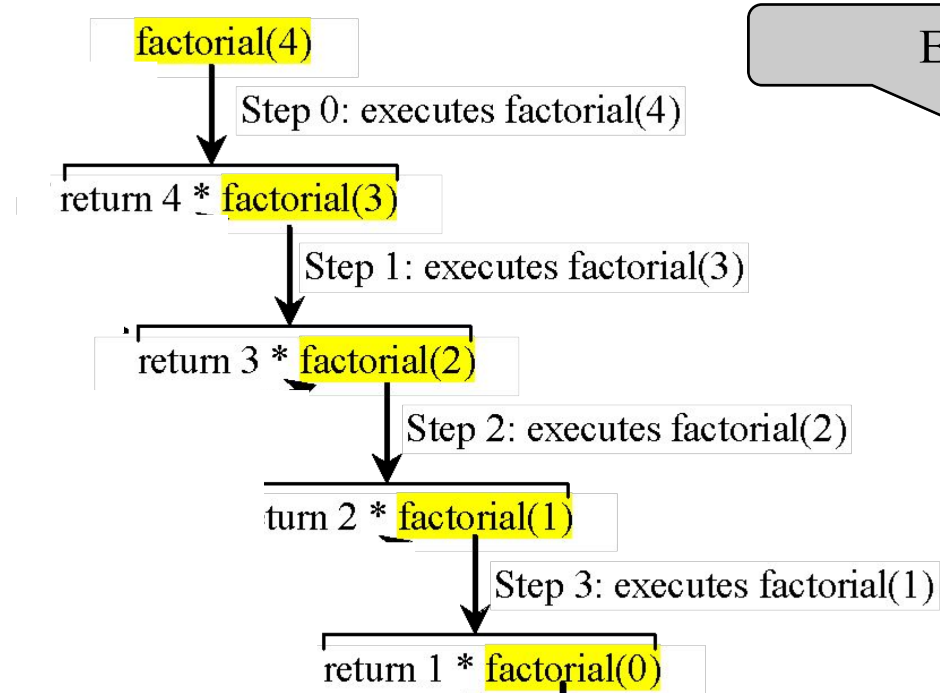
Trace Recursive factorial

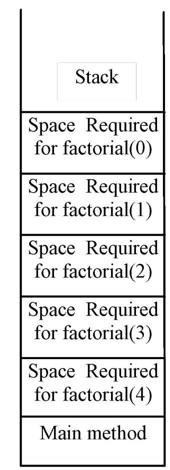


animation

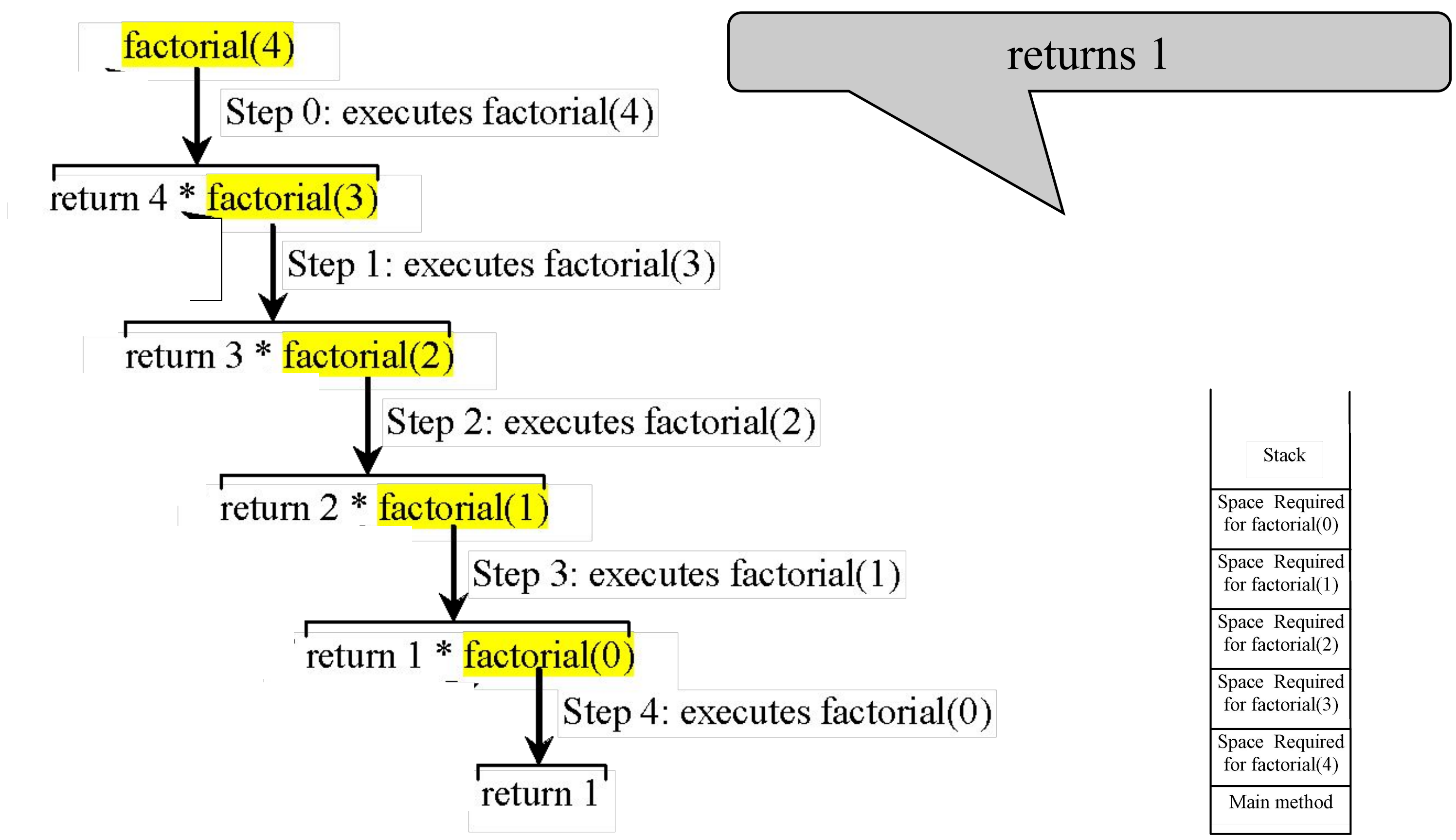
Trace Recursive factorial

Executes factorial(0)



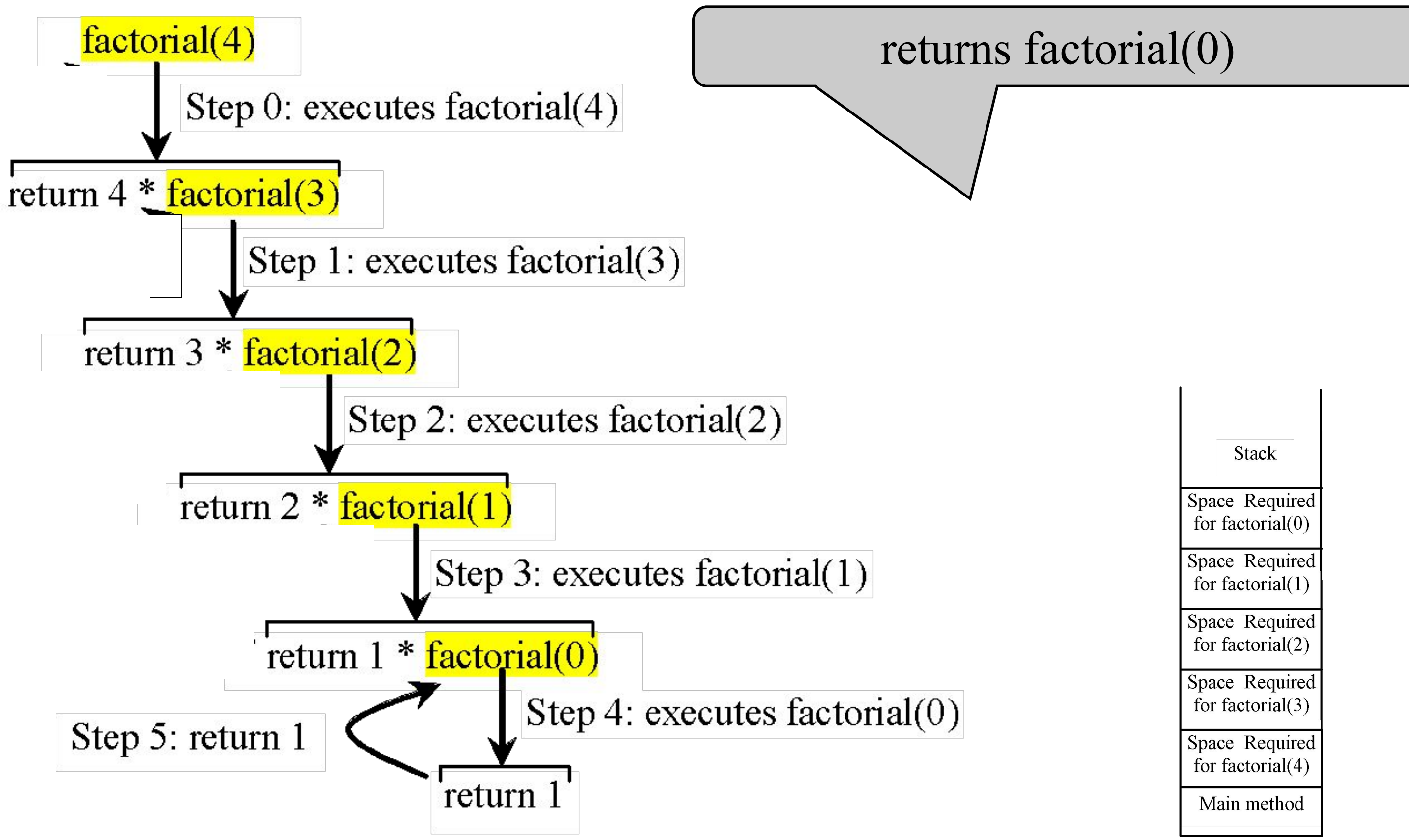


Trace Recursive factorial



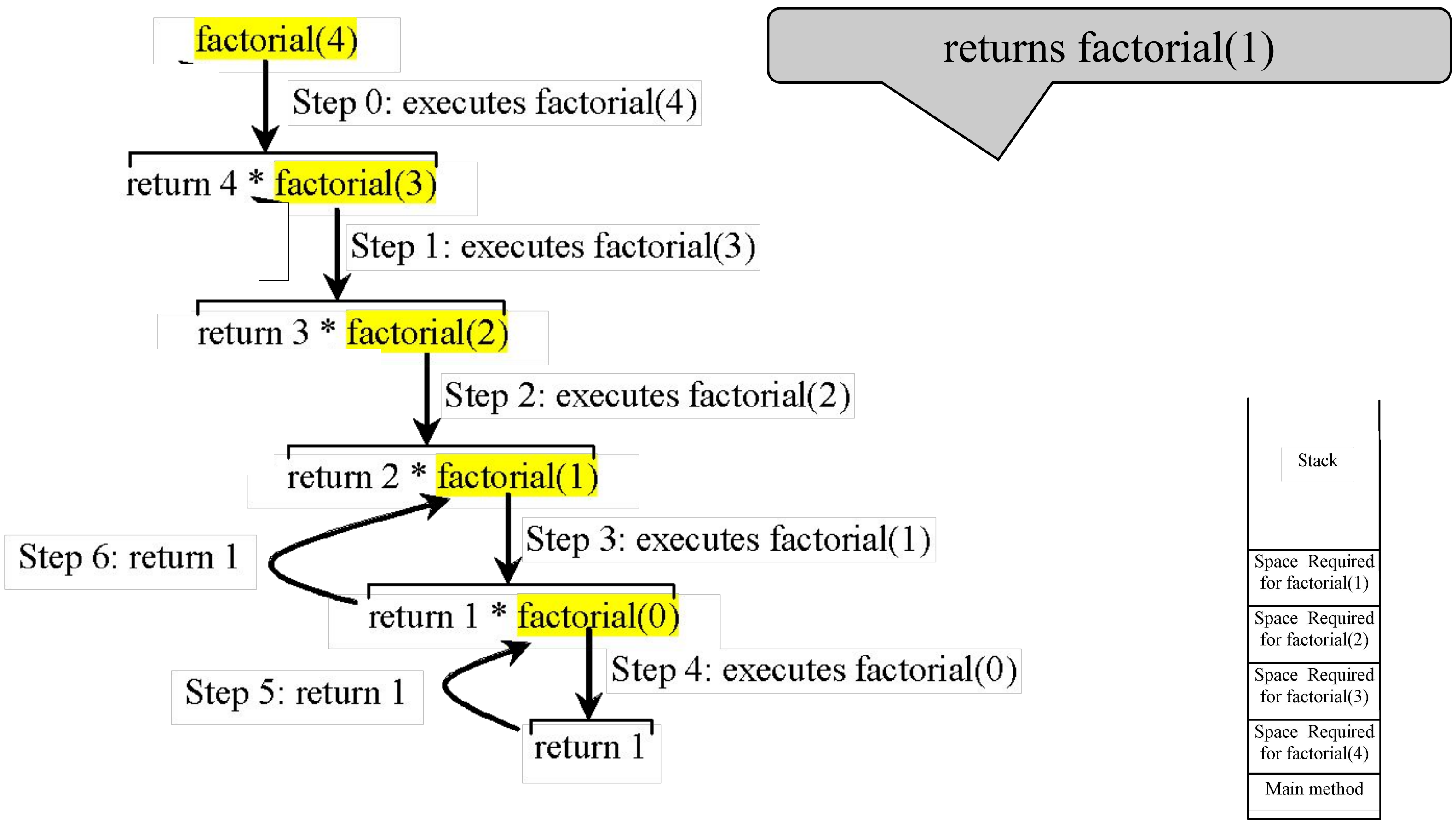
animation

Trace Recursive factorial



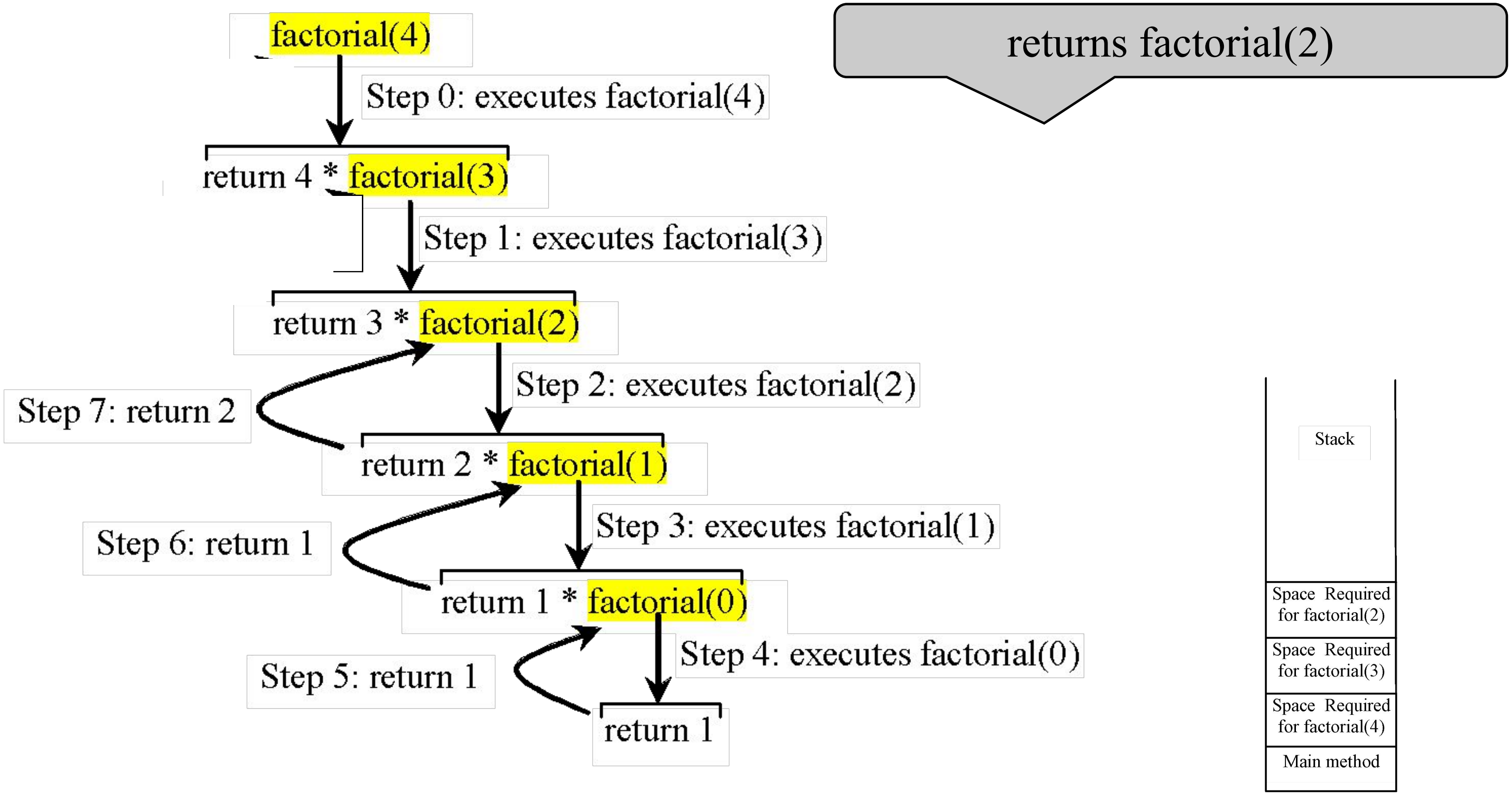
animation

Trace Recursive factorial



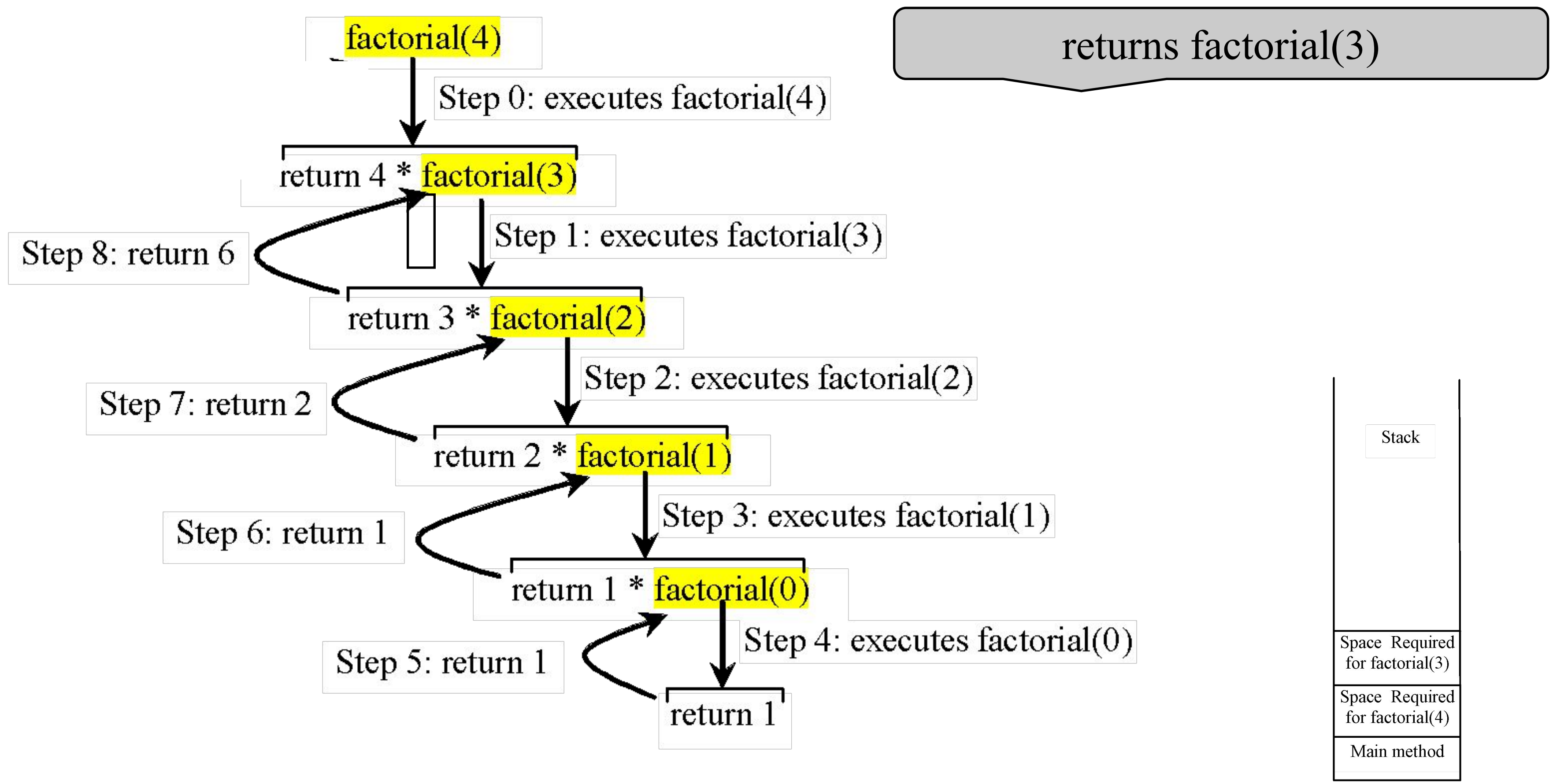
animation

Trace Recursive factorial



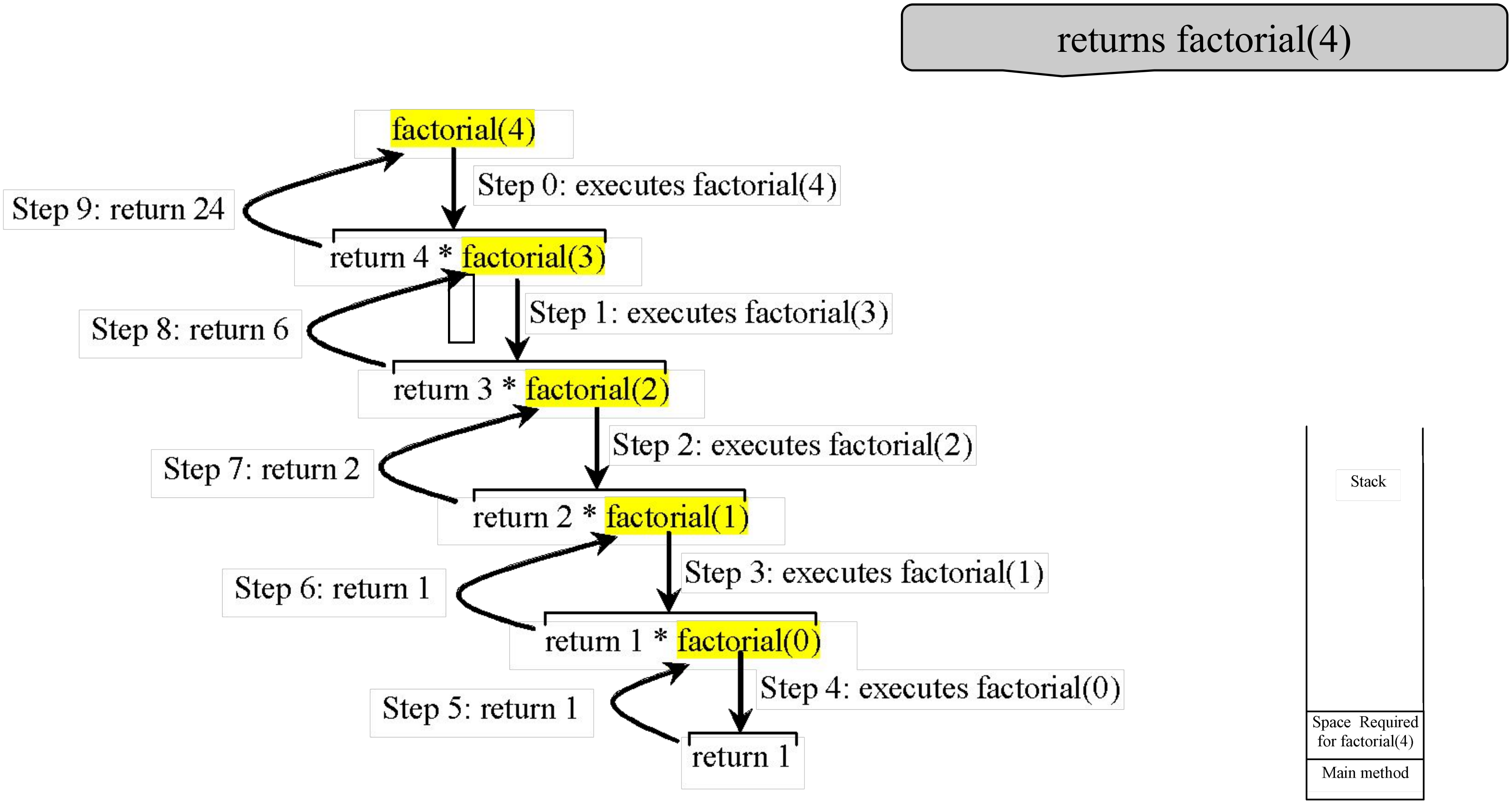
animation

Trace Recursive factorial

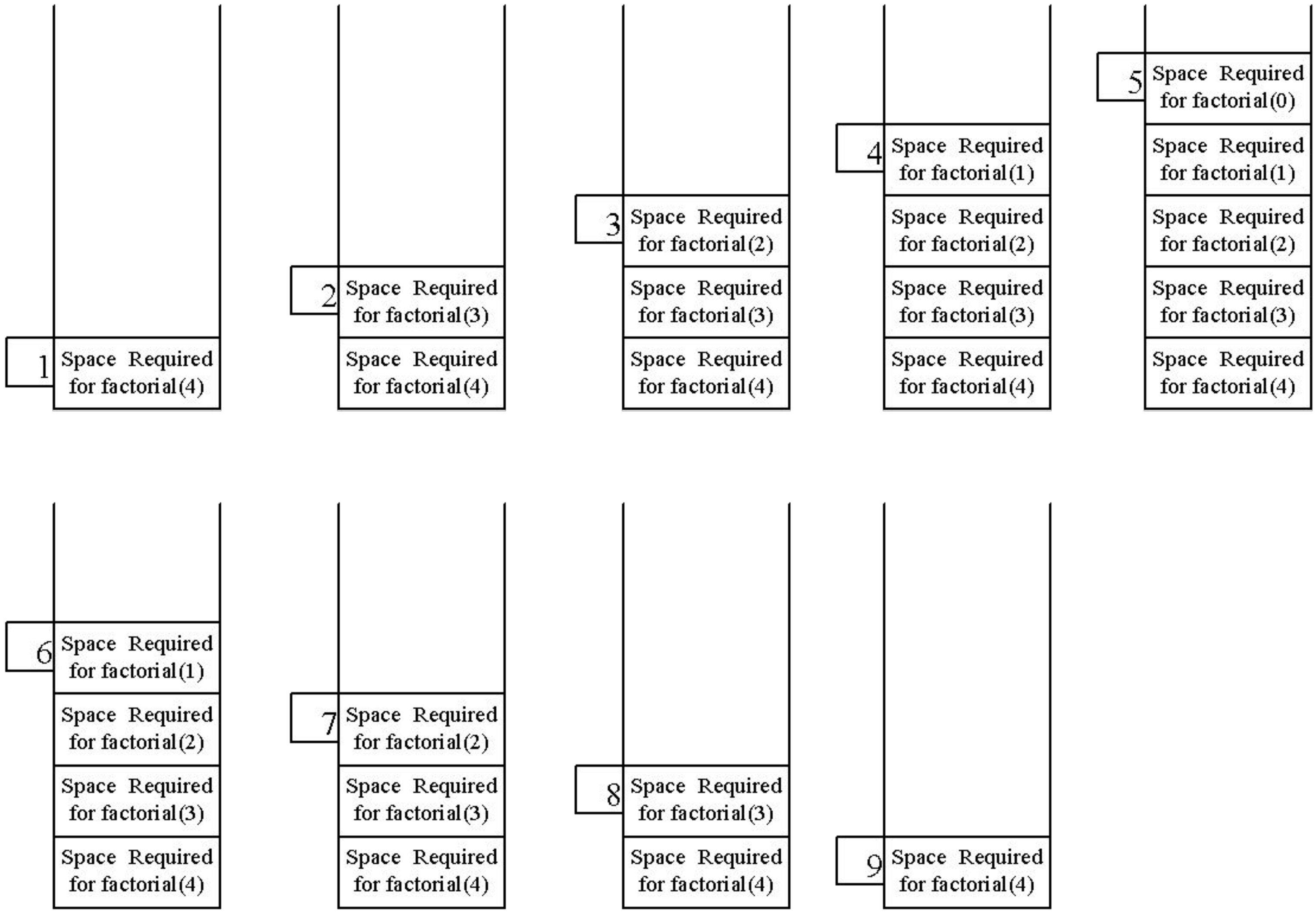


animation

Trace Recursive factorial



factorial(4) Stack Trace



Other Examples

$f(0) = 0;$

$f(n) = n + f(n-1);$

Fibonacci Numbers

```
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

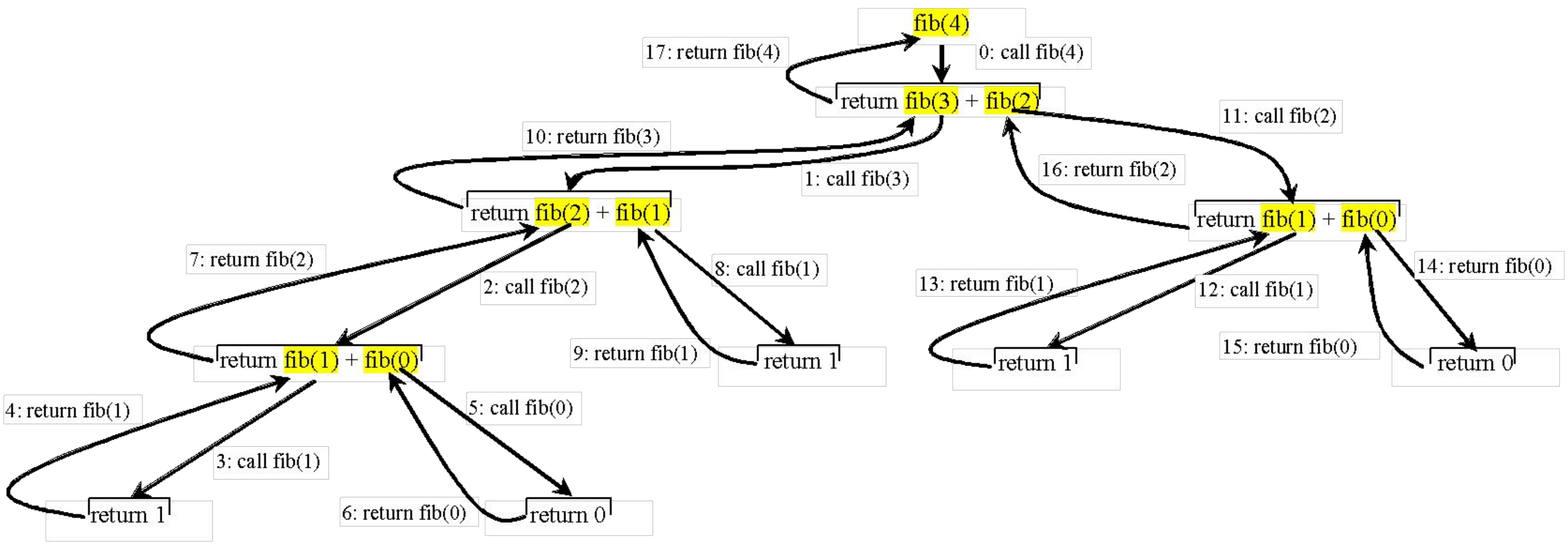
Fibonacci Numbers

```
#include <bits/stdc++.h>
using namespace std;

int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    int n = 9;
    cout << n << "th Fibonacci Number: " << fib(n);
    return 0;
}
```

Fibonnaci Numbers, cont.



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

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:

nPrintln("Welcome", 5);

```
void nPrintln(String message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

```
void nPrint(string message, int times){
    if(times == 0){
        return;
    }
    cout << message << endl;
    nPrint(message, times-1);
}
```

Recursive 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.

Examples

Input − Arr[] = { 5,7,2,3,1,4 }; length=6

Output − Sorted array: 1 2 3 4 5 7

Explanation−

```
First Pass :-
5 7 2 3 1 4 → swap → 1 2 7 3 5 4
1 2 7 3 5 4 → no swap
1 2 7 3 5 4 → swap → 1 2 3 7 5 4
1 2 3 7 5 4 → swap → 1 2 3 4 5 7
1 2 3 4 5 7 → no swap
```

```cpp
int findminpos(int arr[], int st_p, int e_p){
    if(st_p == e_p){
        return st_p;
    }

    int minp = findminpos(arr, st_p+1, e_p);

    return (arr[st_p] > arr[minp]) ? minp : st_p;
}

void selectionSort(int arr[], int start, int end){
    if(start == end){
        return;
    }

    int minpos = findminpos(arr, start, end-1);
    if(minpos != start){
        swap(arr[start], arr[minpos]);
    }

    selectionSort(arr, start+1, end);
}

int main() {

    int arr[] = {3, 1, 5, 2, 7, 0};
    int n = sizeof(arr)/ sizeof(arr[0]);

    selectionSort(arr, 0, n);

    for(int i =0; i < n; i++){
        cout << arr[i] << " ";
    }

    return 0;
}
```

Recursive Binary Search

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.

```cpp
int binarySearch(int arr[], int start, int end, int key){

  if(start > end){
    return -1;
  }

  int mid = (start + end) / 2;

  if(arr[mid] == key){
    return mid;
  } if(arr[mid] > key){
    return binarySearch(arr, start, mid-1, key);
  } else {
    return binarySearch(arr, mid+1, end, key);
  }
}

int main() {

  int arr[] = {1, 3, 5, 6, 7, 8, 9};
  int n = sizeof(arr)/ sizeof(arr[0]);

  for(int i =0; i < n; i++){
    cout << arr[i] << " ";
  }

  cout << "\n" << "Enter key :";
  int k;
  cin >> k;

  int ind = binarySearch(arr, 0, n-1, k);

  if(ind != -1){
    cout << "Key is found at position index: " << ind << endl;
  } else {
    cout << "Key is not found"<< endl;
  }

  return 0;
}
```