Software Engineering 2 (C++)

CSY2006 (Week 9)

Introduction to Recursion

[Print a line n (1000) times w/o loops]

(Pr19-1, Pr19-2)

Introduction to Recursion

- A recursive function contains a call to itself:

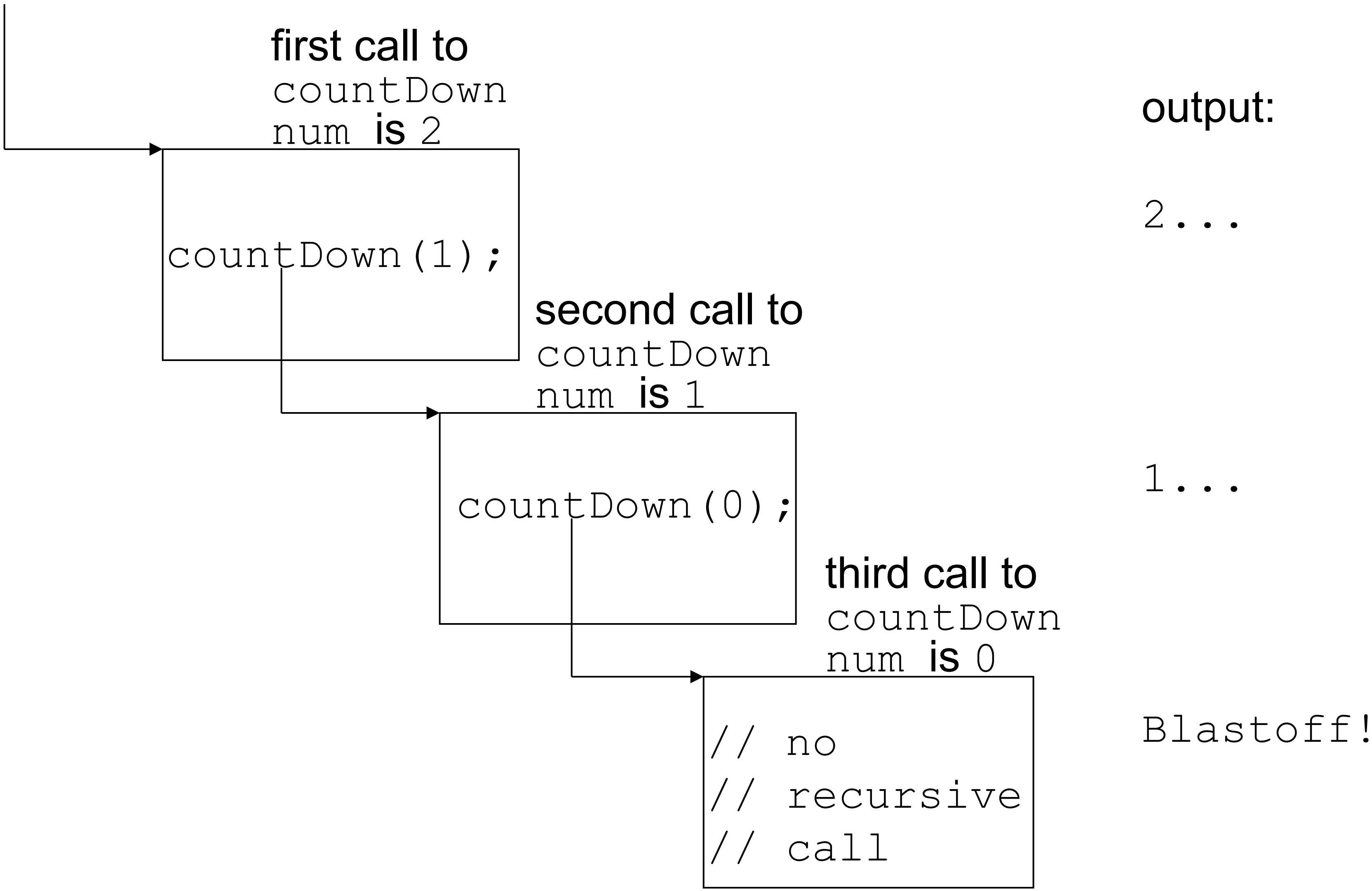
```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```

What Happens When Called?

If a program contains a line like `countDown(2);`

1. `countDown(2)` generates the output `2...`, then it calls `countDown(1)`
2. `countDown(1)` generates the output `1...`, then it calls `countDown(0)`
3. `countDown(0)` generates the output `Blastoff!`, then returns to `countDown(1)`
4. `countDown(1)` returns to `countDown(2)`
5. `countDown(2)` returns to the calling function

What Happens When Called?



Solving Problems with Recursion

Recursive Functions - Purpose

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simpler-to-solve problem is known as the base case
- Recursive calls stop when the base case is reached

Stopping the Recursion

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- In the sample program, the test is:

```
if (num == 0)
```

Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0) // test
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1); // recursive
   }                    // call
}
```

Stopping the Recursion

- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved
- In the `countDown` function, a different value is passed to the function each time it is called
- Eventually, the parameter reaches the value in the test, and the recursion stops

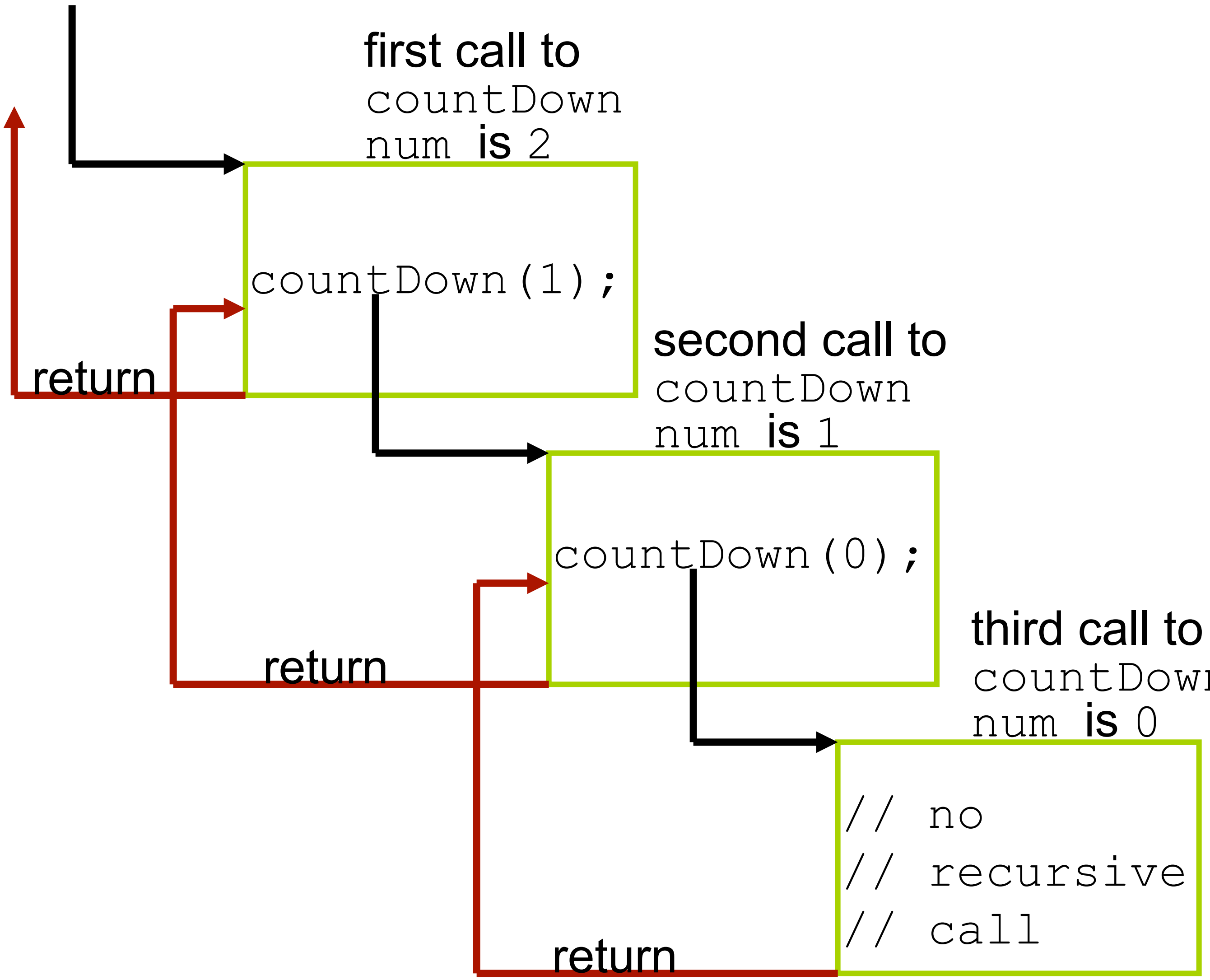
Stopping the Recursion

```
void countDown(int num)
{
   if (num == 0)
      cout << "Blastoff!";
   else
   {
      cout << num << "...\n";
      countDown(num-1);// note that the value
   }                   // passed to recursive
}                      // calls decreases by
                       // one for each call
```

What Happens When Called?

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

What Happens When Called?



output:

2...

1...

Blastoff!

Types of Recursion

- Direct
  - a function calls itself
- Indirect
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

The Recursive Factorial Function

- The factorial function:

  `n! = n*(n-1)*(n-2)*...*3*2*1` if `n > 0`

  `n! = 1` if `n = 0`

- Can compute factorial of `n` if the factorial of `(n-1)` is known:

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

- `n = 0` is the base case

The Recursive Factorial Function

```
int factorial (int num)
{
  if (num > 0)
     return num * factorial(num - 1);
 else
     return 1;
}
```

Program 19-3

```
// This program demonstrates a recursive function to
// calculate the factorial of a number.
#include <iostream>
using namespace std;

// Function prototype
int factorial(int);

int main()
{
   int number;

   // Get a number from the user.
   cout << "Enter an integer value and I will display\n";
   cout << "its factorial: ";
   cin >> number;

   // Display the factorial of the number.
   cout << "The factorial of " << number << " is ";
   cout << factorial(number) << endl;
   return 0;
}

```

Program 19-3 (Continued)

```
//*****************************************************************
// Definition of factorial. A recursive function to calculate *
// the factorial of the parameter n.                          *
//*****************************************************************

int factorial(int n)
{
   if (n == 0)
      return 1;                      // Base case
   else
      return n * factorial(n - 1);  // Recursive case
}
```

Program Output with Example Input Shown in Bold

```
Enter an integer value and I will display
its factorial: 4 [Enter]
The factorial of 4 is 24
```

animation

Computing Factorial

factorial(0) = 1;

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

factorial(3)

animation

Computing Factorial

factorial(0) = 1;

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

factorial(3) = 3 * factorial(2)

animation

Computing Factorial

factorial(0) = 1;

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

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

animation

Computing Factorial

factorial(0) = 1;

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

factorial(3) = 3 * factorial(2)

= 3 * (2 * factorial(1))

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

animation

Computing Factorial

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)))

animation

Computing Factorial

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)

animation

Computing Factorial

```
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

animation

Computing Factorial

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

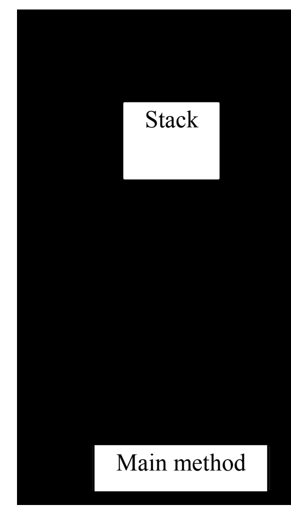
= 6

animation

Trace Recursive factorial

Executes factorial(4)

factorial(4)



animation

Trace Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Executes factorial(3)

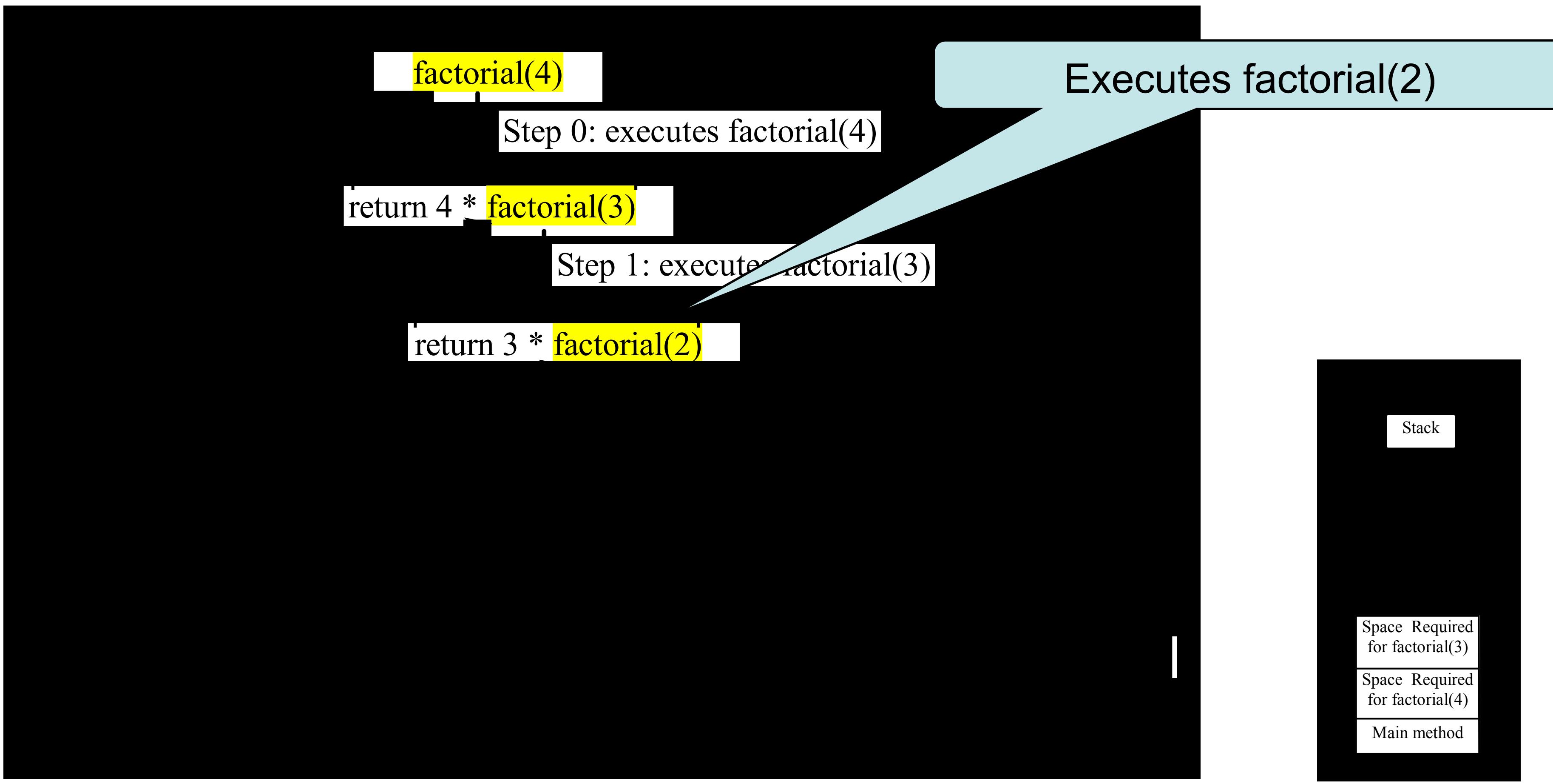
Stack

Space Required for factorial(4)

Main method

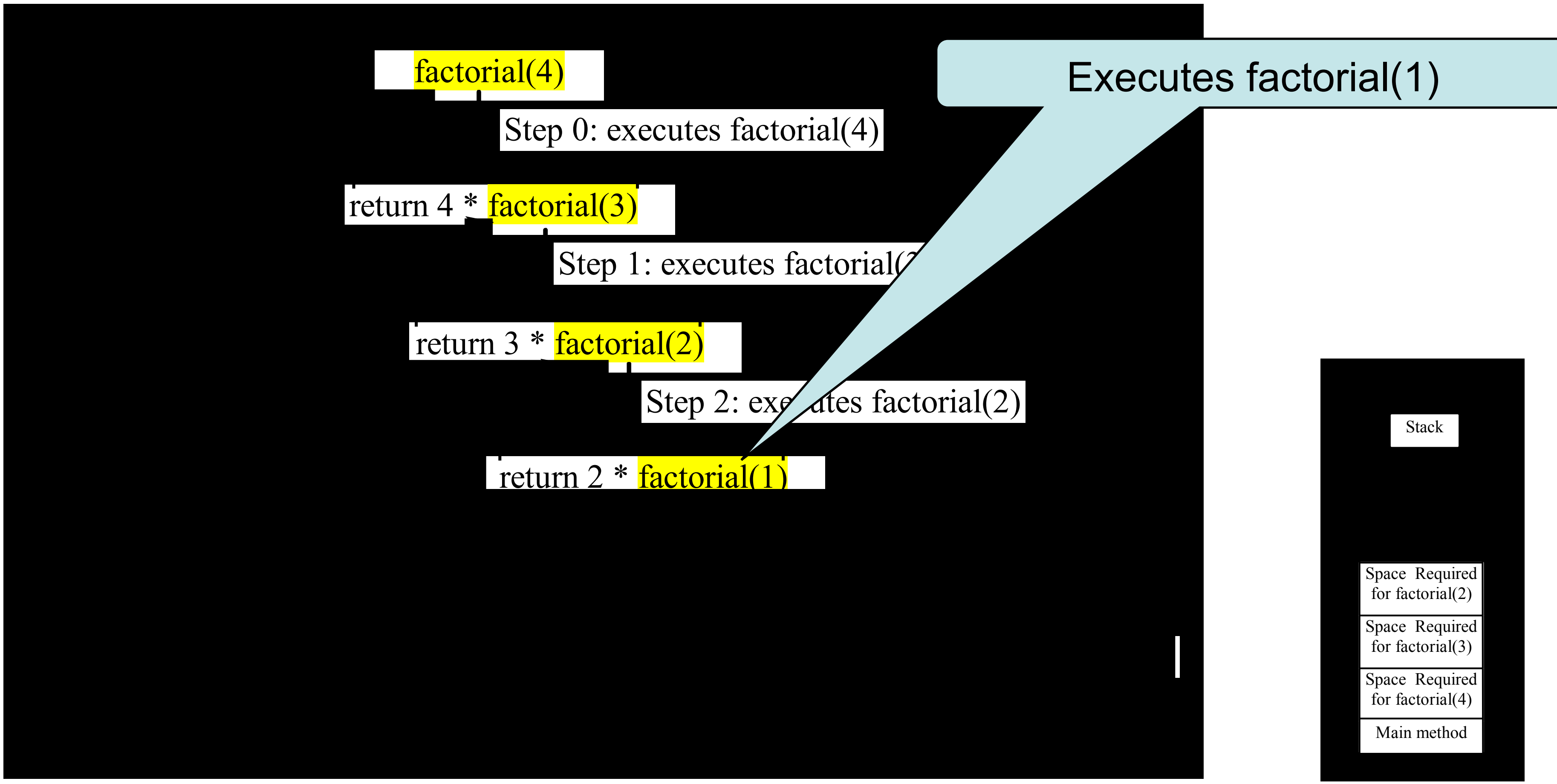
animation

Trace Recursive factorial



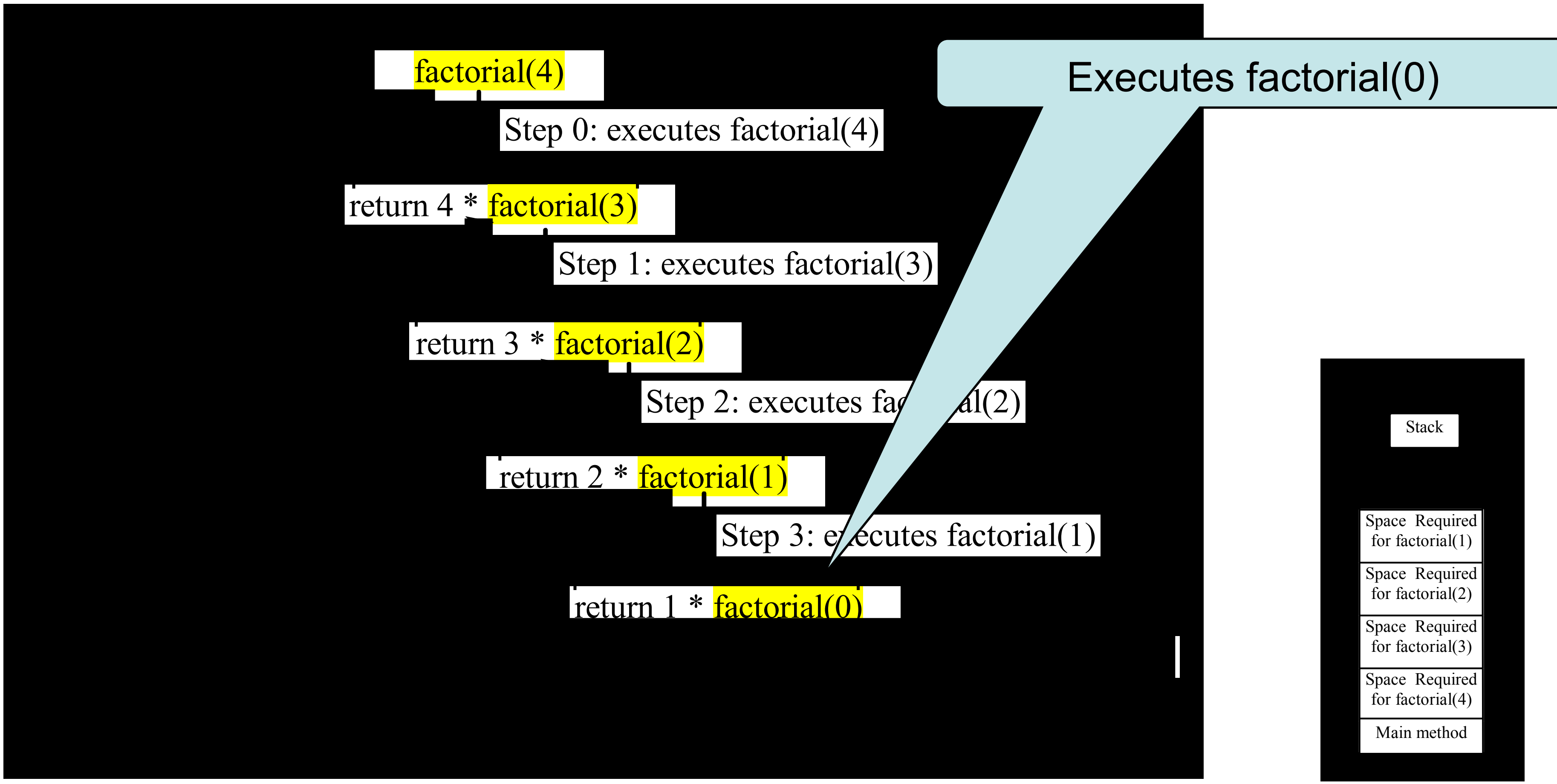
animation

Trace Recursive factorial



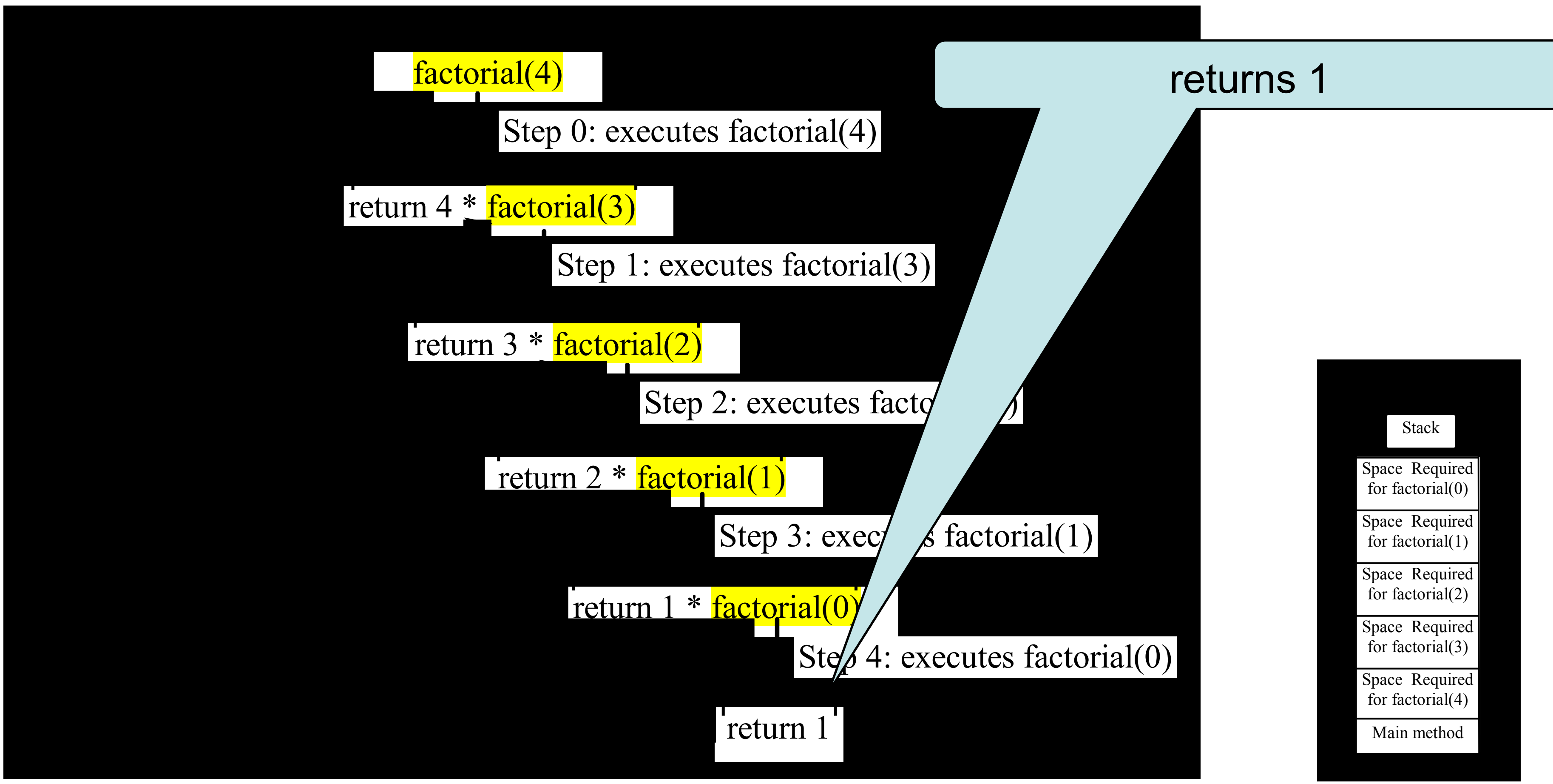
animation

Trace Recursive factorial



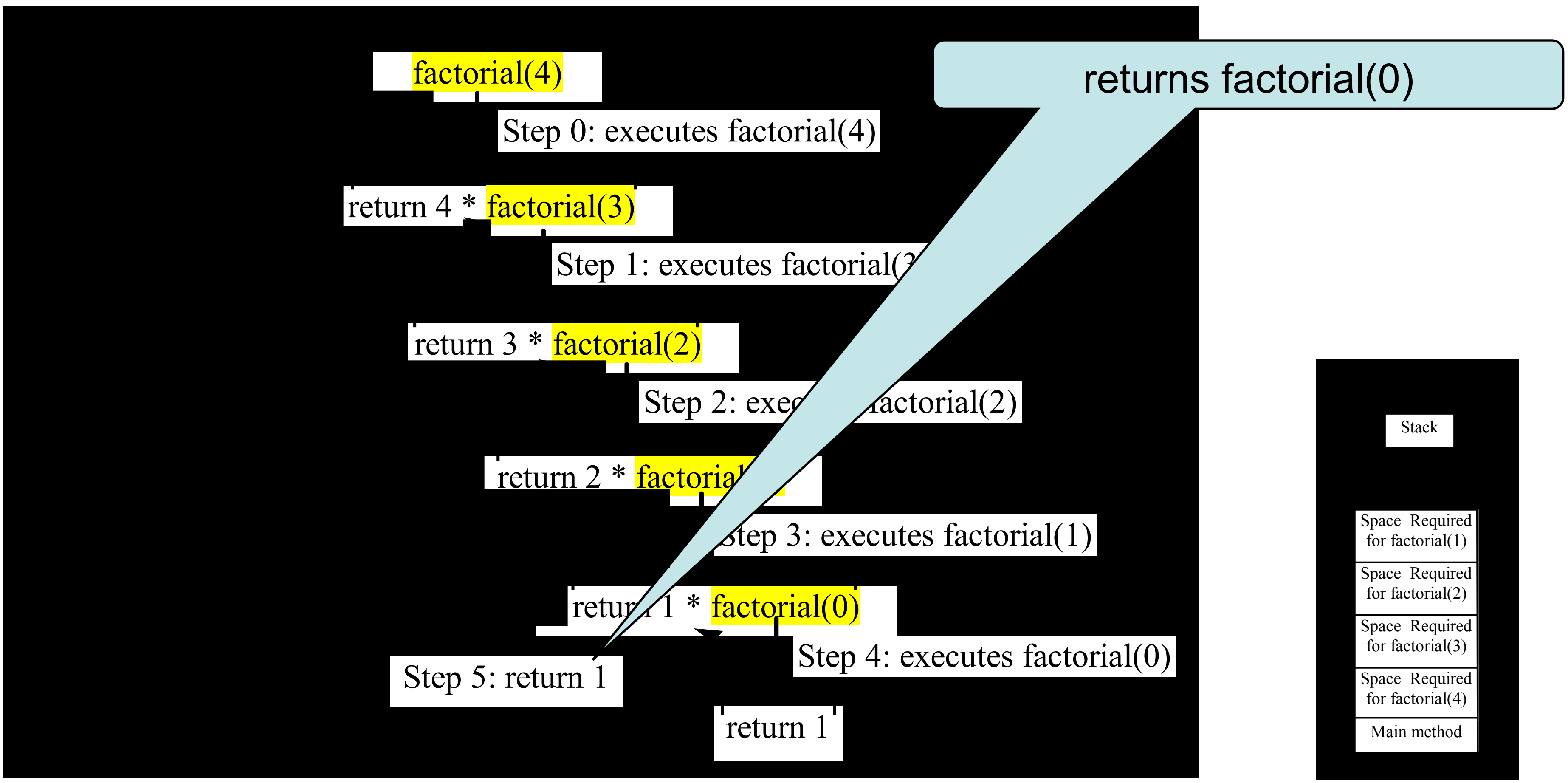
animation

Trace Recursive factorial



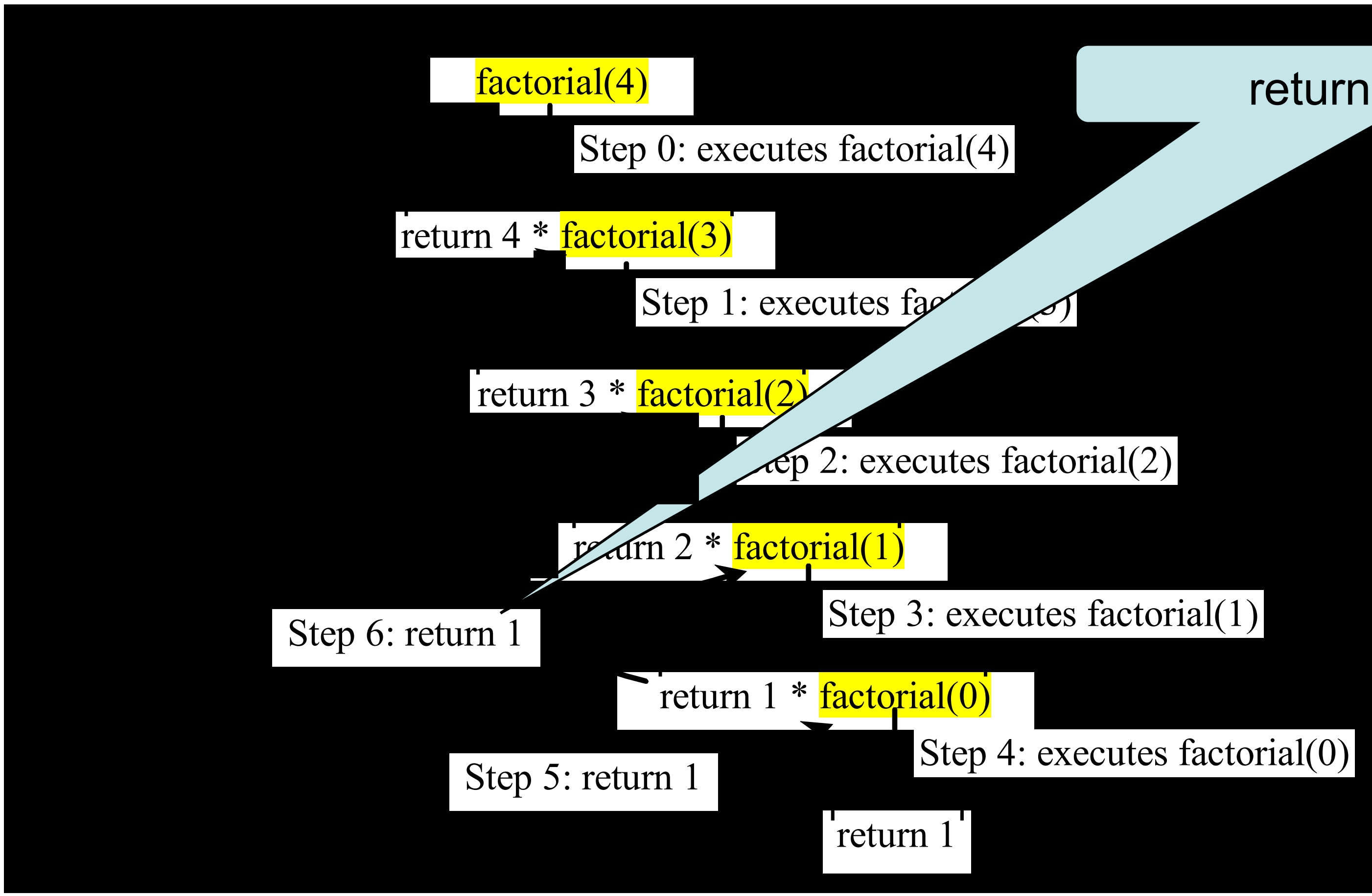
animation

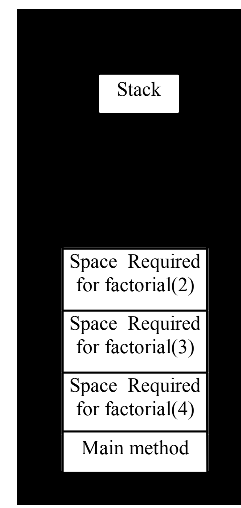
Trace Recursive factorial



Trace Recursive factorial

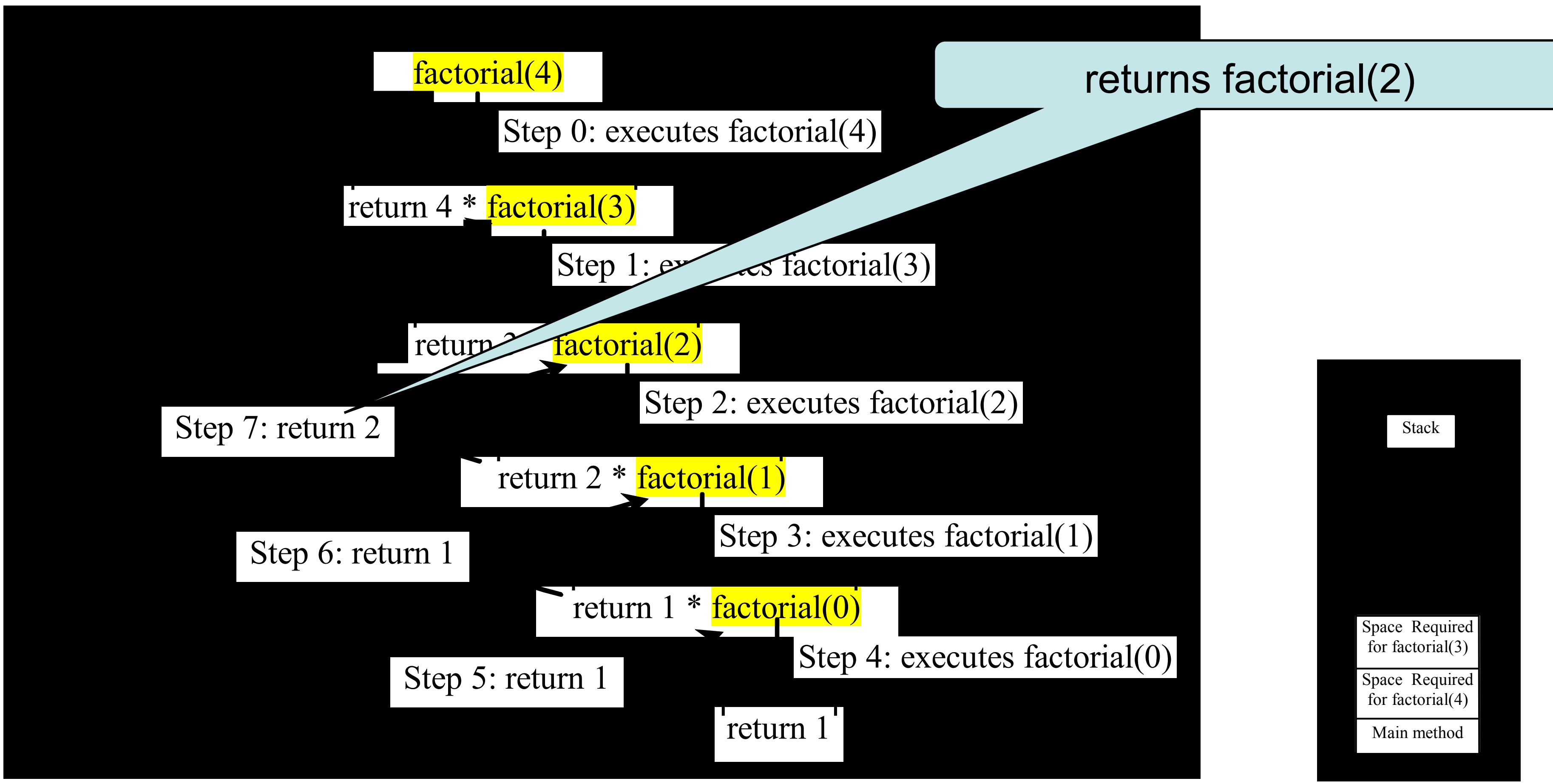
returns factorial(1)





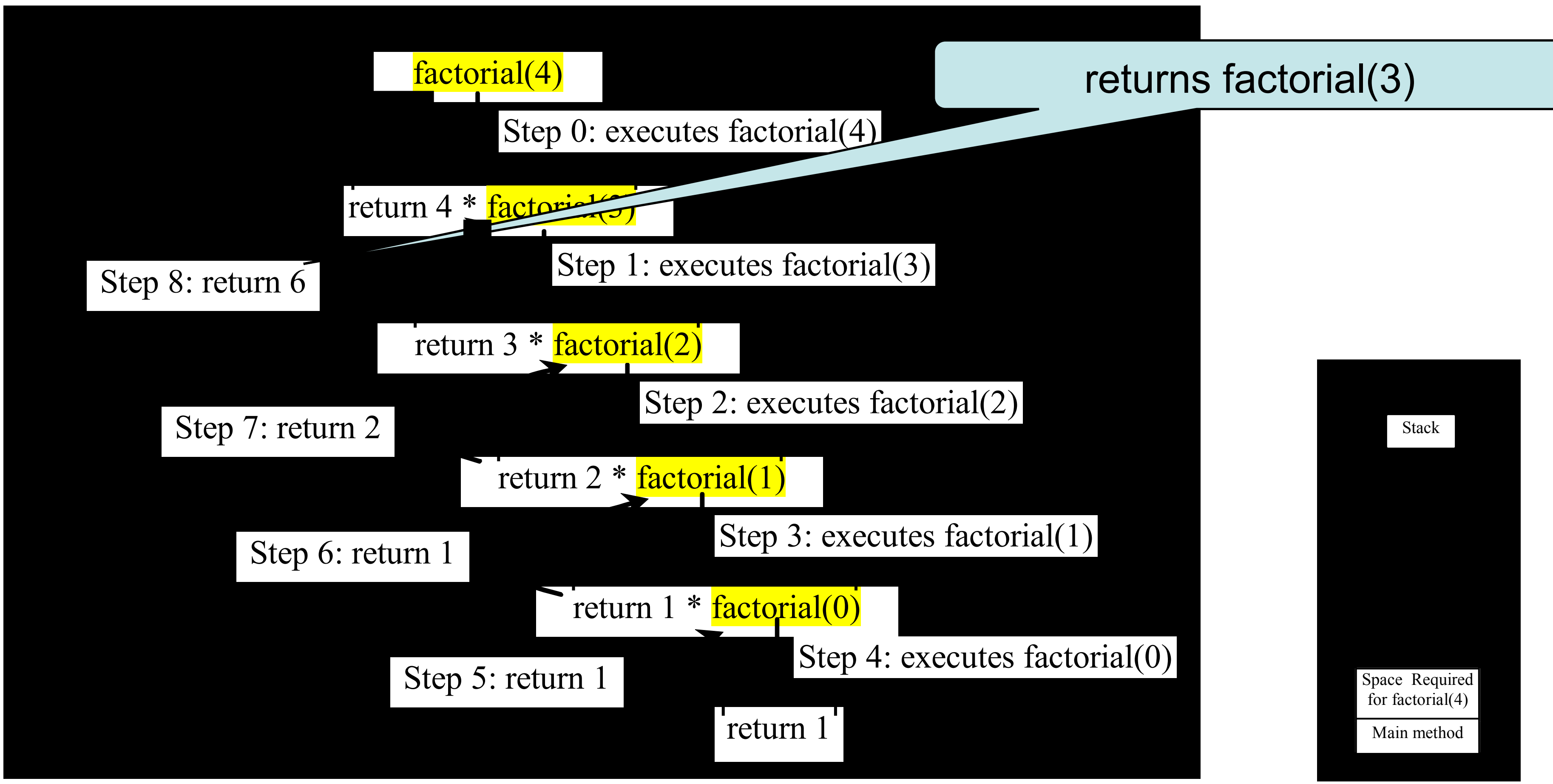
animation

Trace Recursive factorial

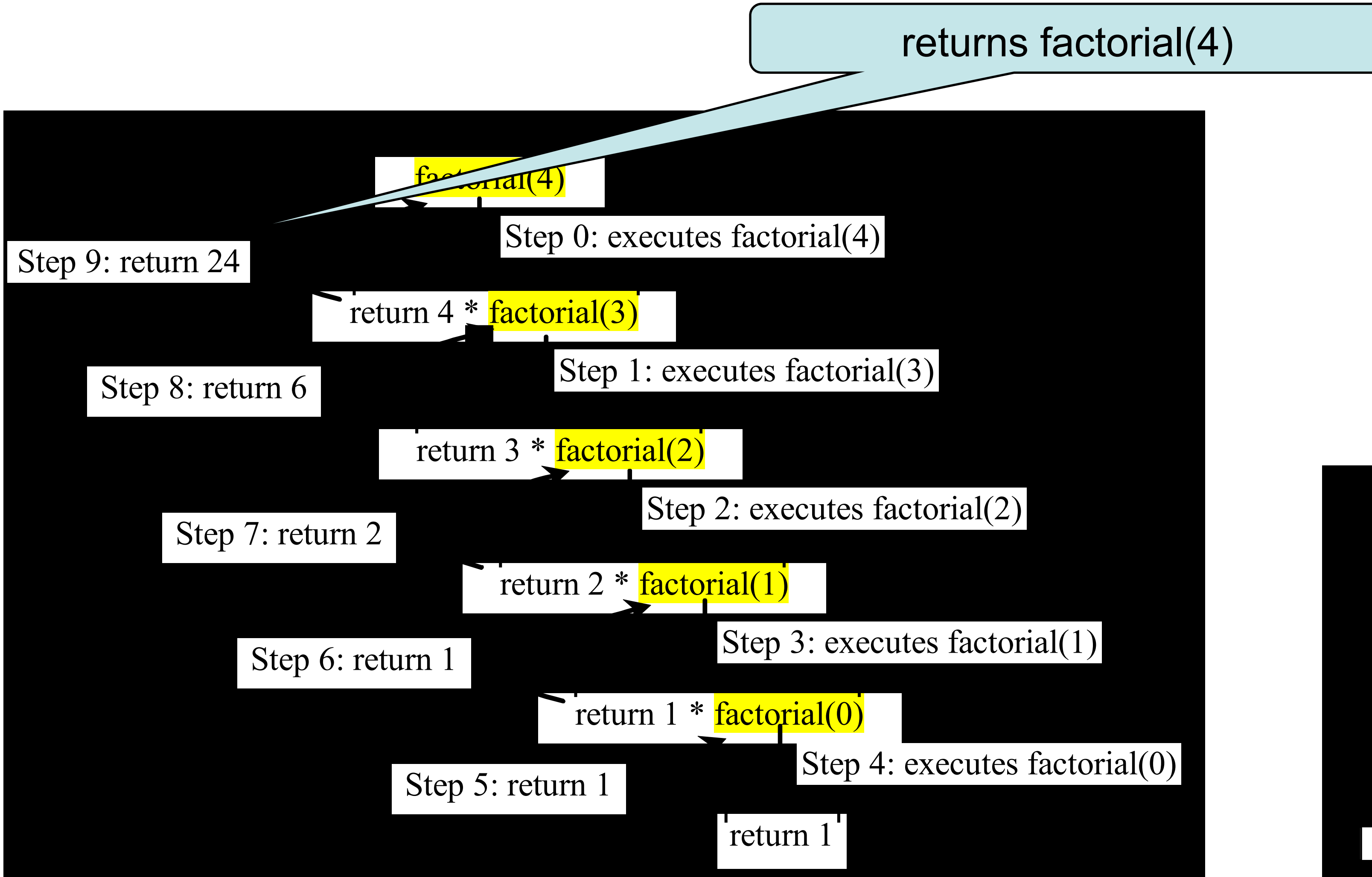


animation

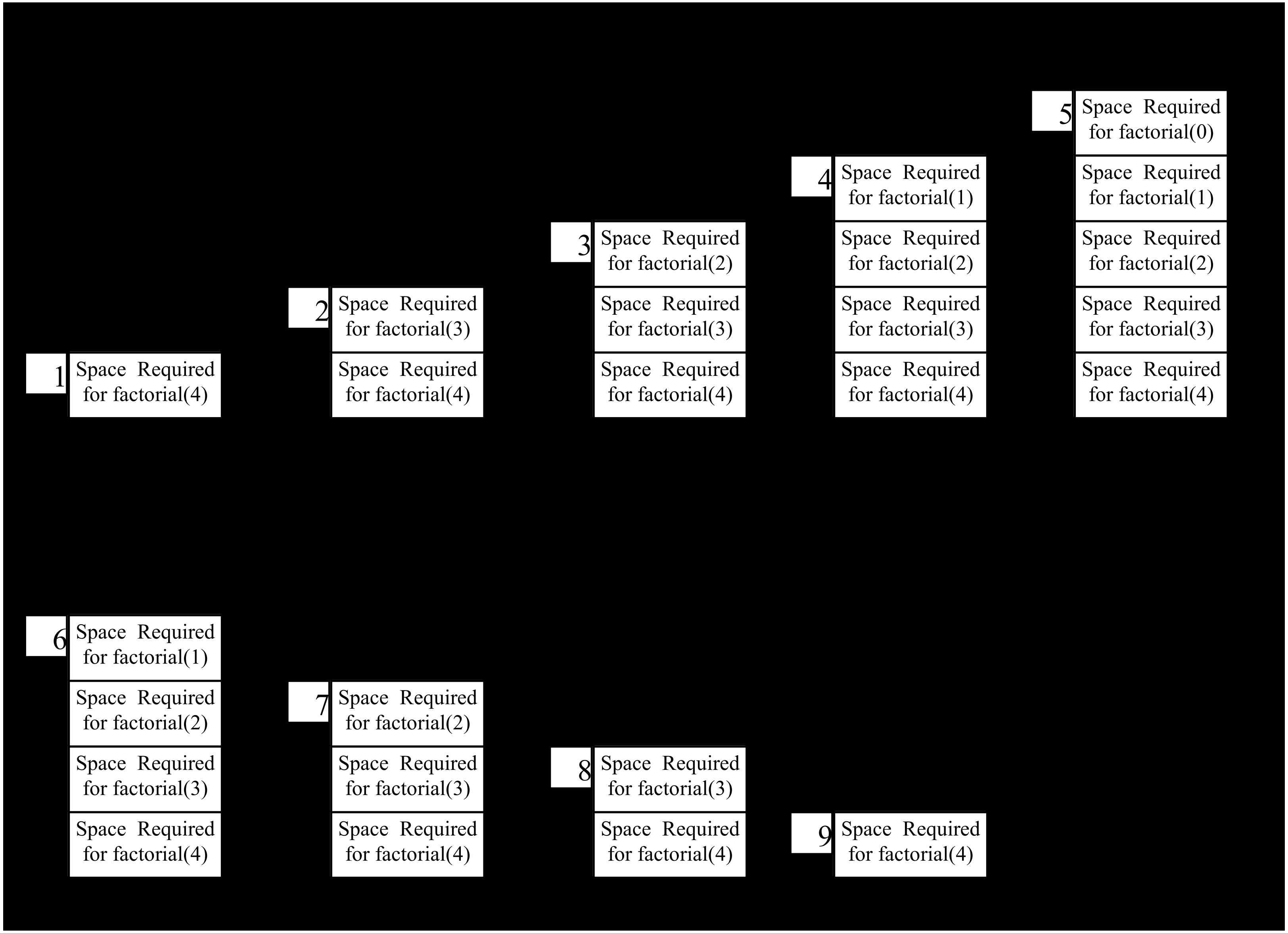
Trace Recursive factorial



Trace Recursive factorial



factorial(4) Stack Trace



The Recursive gcd Function

The Recursive gcd Function

- Greatest common divisor (gcd) is the largest factor that two integers have in common
- Computed using Euclid's algorithm:

  `gcd(x, y) = y` if `y` divides `x` evenly

  `gcd(x, y) = gcd(y, x % y)` otherwise
- `gcd(x, y) = y` is the base case

The Recursive gcd Function

```
int gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
```

Solving Recursively Defined Problems

Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`
- After the starting `0, 1`, each number is the sum of the two preceding numbers
- Recursive solution:

  `fib(n) = fib(n – 1) + fib(n – 2);`
- Base cases: `n <= 0, n == 1`

Solving Recursively Defined Problems

```
int fib(int n)
{
   if (n <= 0)
       return 0;
   else if (n == 1)
       return 1;
   else
         return fib(n – 1) + fib(n – 2);
}
```

A Recursive Binary Search Function

A Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- Base cases: desired value is found, or no more array elements to search
- Algorithm (array in ascending order):
  - If middle element of array segment is desired value, then done
  - Else, if the middle element is too large, repeat binary search in first half of array segment
  - Else, if the middle element is too small, repeat binary search on the second half of array segment

A Recursive Binary Search Function (Continued)

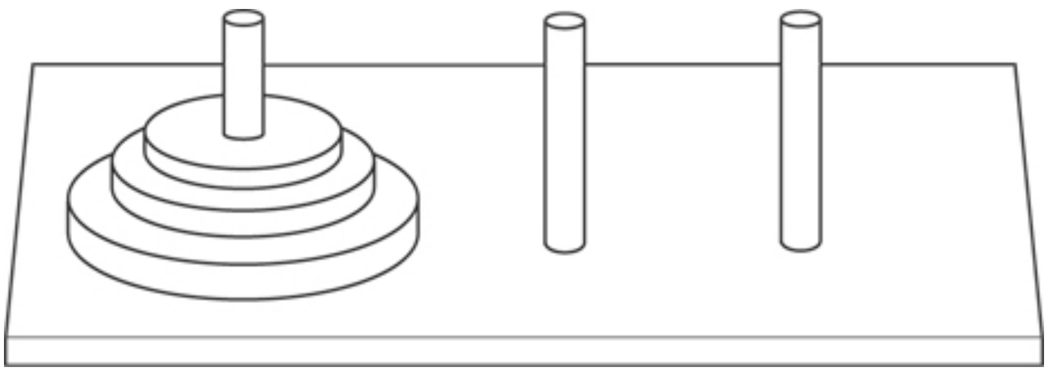
```
 int binarySearch(int array[], int first, int last, int value)
{
   int middle;   // Mid point of search

   if (first > last)
      return -1;
   middle = (first + last) / 2;
   if (array[middle] == value)
      return middle;
   if (array[middle] < value)
      return binarySearch(array, middle+1,last,value);
   else
      return binarySearch(array, first,middle-1,value);
}
```

The Towers of Hanoi

The Towers of Hanoi

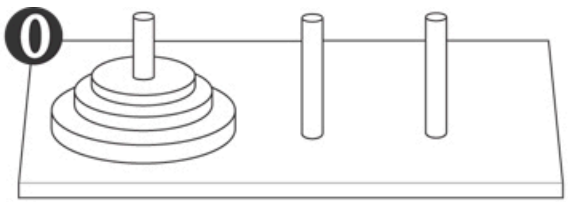
- The Towers of Hanoi is a mathematical game that is often used to demonstrate the power of recursion.
- The game uses three pegs and a set of discs, stacked on one of the pegs.



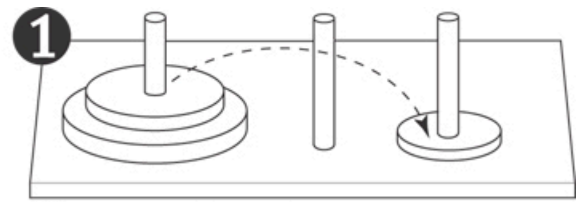
The Towers of Hanoi

- The object of the game is to move the discs from the first peg to the third peg. Here are the rules:
  - Only one disc may be moved at a time.
  - A disc cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

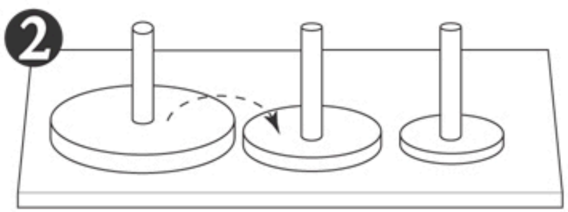
Moving Three Discs



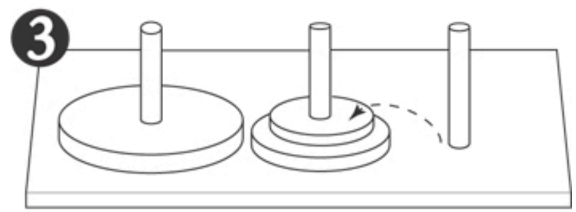
Original setup.



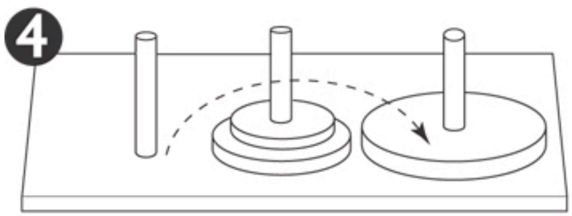
First move: Move disc 1 to peg 3.



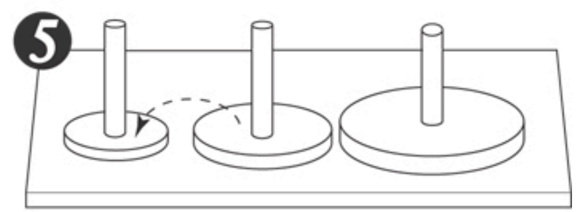
Second move: Move disc 2 to peg 2.



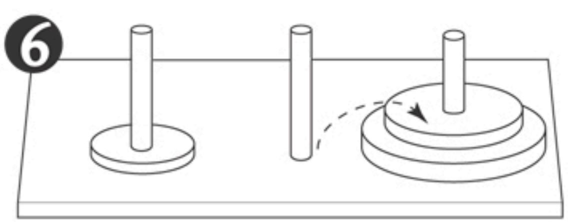
Third move: Move disc 1 to peg 2.



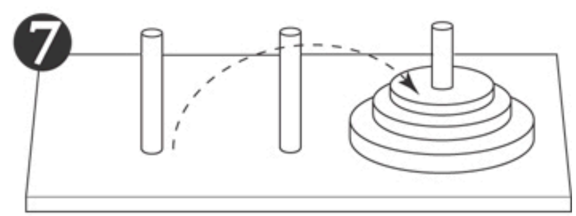
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

The Towers of Hanoi

- The following statement describes the overall solution to the problem:
  - *Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.*

The Towers of Hanoi

- Algorithm
  - *To move n discs from peg A to peg C, using peg B as a temporary peg:*
    If n > 0 Then
    Move n – 1 discs from peg A to peg B, using peg C as a temporary peg.

    Move the remaining disc from the peg A to peg C.

    Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.

    End If

Program 19-10

```
// This program displays a solution to the Towers of
// Hanoi game.
#include <iostream>
using namespace std;

// Function prototype
void moveDiscs(int, int, int, int);

int main()
{
   const int NUM_DISCS = 3;    // Number of discs to move
   const int FROM_PEG = 1;     // Initial "from" peg
   const int TO_PEG = 3;       // Initial "to" peg
   const int TEMP_PEG = 2;     // Initial "temp" peg

```

Program 19-10 *(continued)*

```
   // Play the game.
   moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG);
   cout << "All the pegs are moved!\n";
   return 0;
}

//*****************************************************
// The moveDiscs function displays a disc move in    *
// the Towers of Hanoi game.                          *
// The parameters are:                                *
//    num:     The number of discs to move.           *
//    fromPeg: The peg to move from.                  *
//    toPeg:   The peg to move to.                    *
//    tempPeg: The temporary peg.                     *
//*****************************************************

void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
   if (num > 0)
   {
      moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
      cout << "Move a disc from peg " << fromPeg
           << " to peg " << toPeg << endl;
      moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
   }
}
```

Program 19-10 (Continued)

Program Output

```
Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
Move a disc from peg 1 to peg 3
Move a disc from peg 2 to peg 1
Move a disc from peg 2 to peg 3
Move a disc from peg 1 to peg 3
All the pegs are moved!
```

Recursion vs. Iteration

Recursion vs. Iteration

- Benefits (+), disadvantages(-) for recursion:
  - \+ Models certain algorithms most accurately
  - \+ Results in shorter, simpler functions
  - – May not execute very efficiently
- Benefits (+), disadvantages(-) for iteration:
  - \+ Executes more efficiently than recursion
  - – Often is harder to code or understand

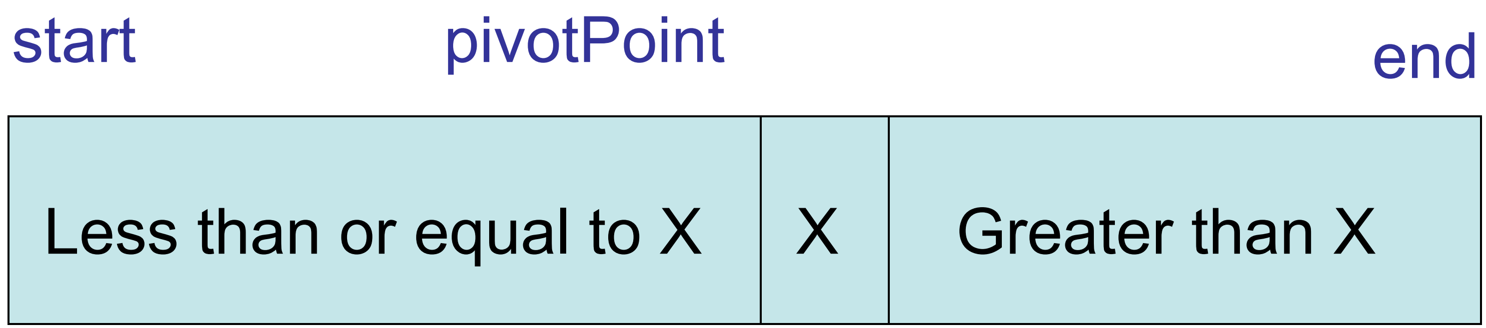
Quick Sort

Quick sort, developed by C. A. R. Hoare (1962), works as follows: The algorithm selects an element, called the *pivot*, in the array. Divide the array into two parts such that all the elements in the first part are less than or equal to the pivot and all the elements in the second part are greater than the pivot. Recursively apply the quick sort algorithm to the first part and then the second part.

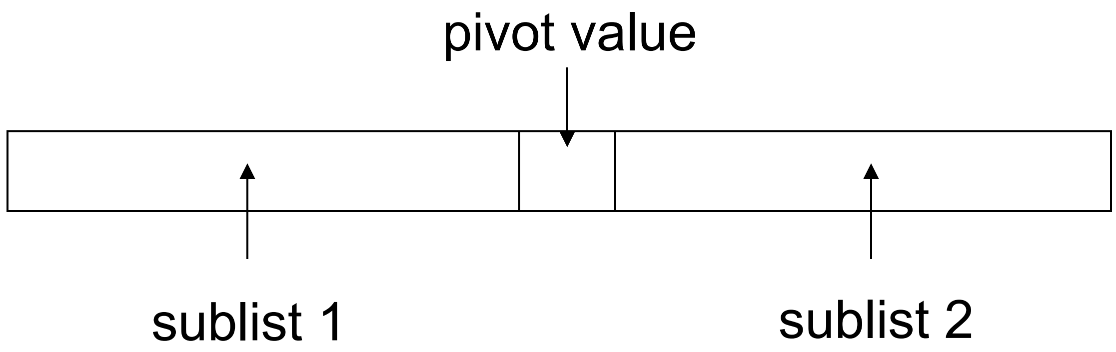
See QuickSort.cpp

How to Partition

Given an array segment *A*[start..end], we want to partition it and return the pivot point:

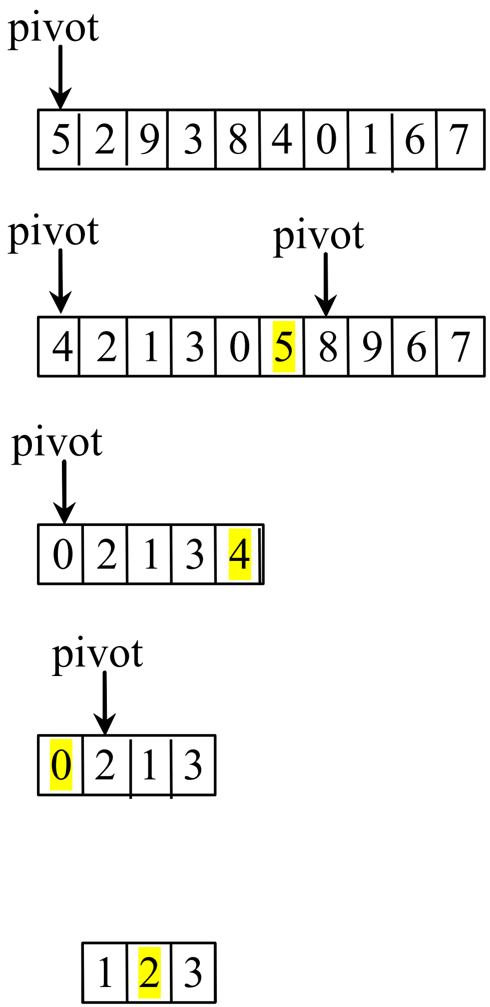


The QuickSort Algorithm



- Once pivot value is determined, values are shifted so that elements in sublist1 are $\leq$ pivot and elements in sublist2 are > pivot
- Algorithm then sorts sublist1 and sublist2
- Base case: sublist has size 1

Quick Sort



(a) The original array

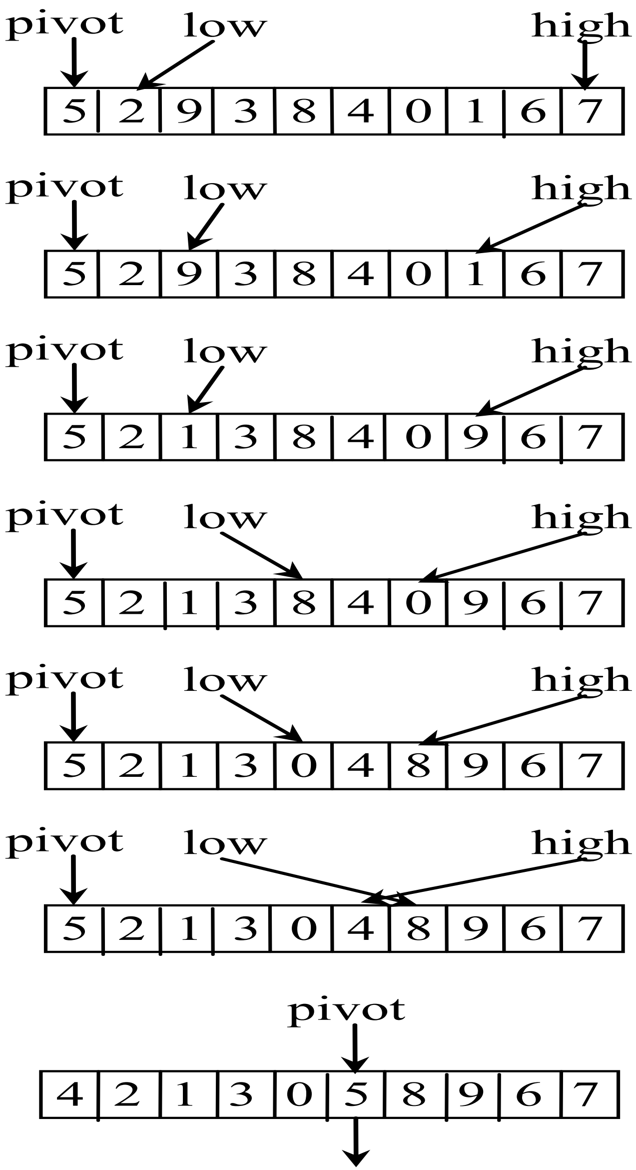
(b)The original array is partitioned

(c) The partial array (4 2 1 3 0) is partitioned

(d) The partial array (0 2 1 3) is partitioned

(e) The partial array (2 1 3) is partitioned

Partition



(a) Initialize pivot, low, and high

(b) Search forward and backward

(c) 9 is swapped with 1

(d) Continue search

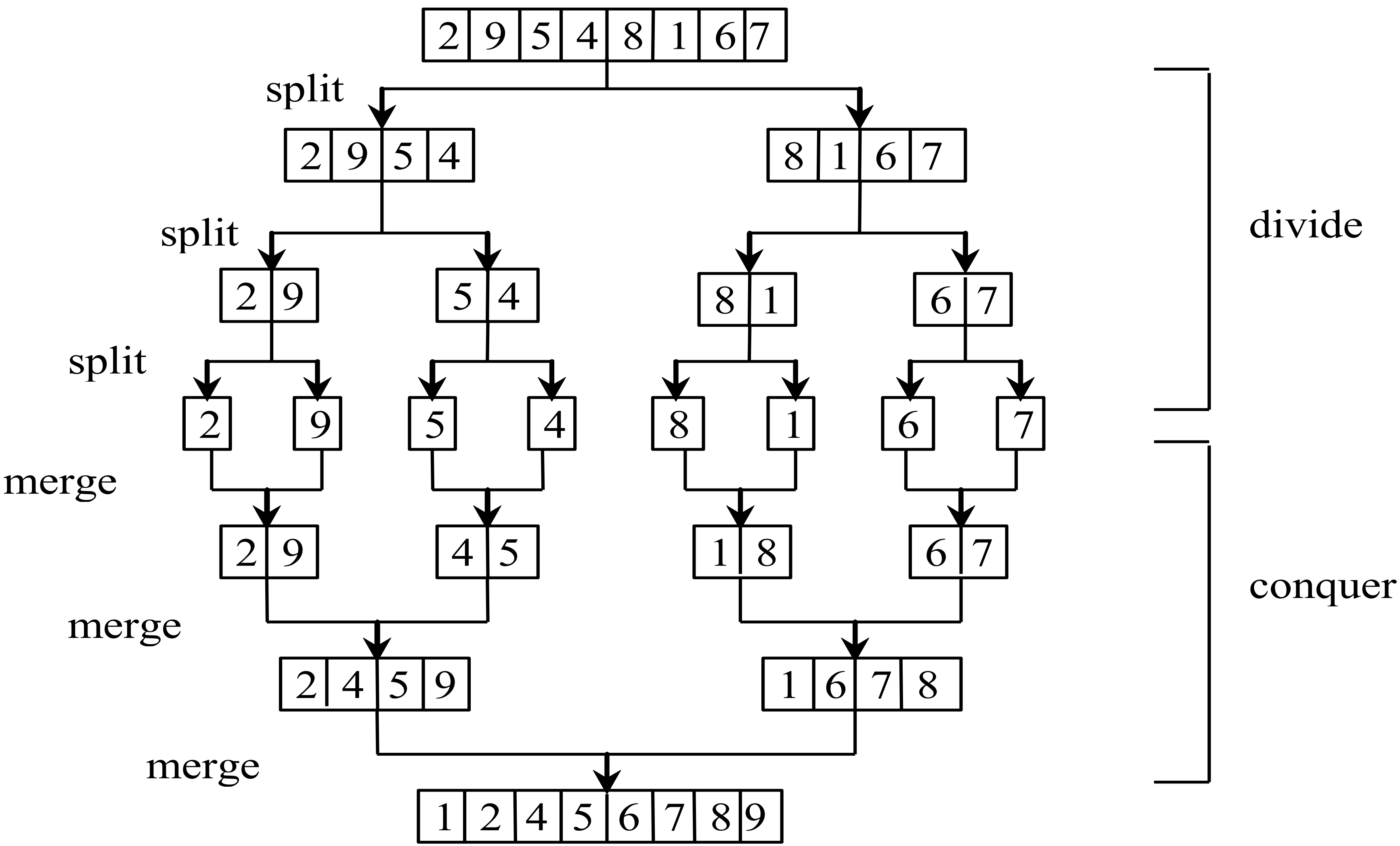
(e) 8 is swapped with 0

(f) when high < low, search is over

(g) pivot is in the right place

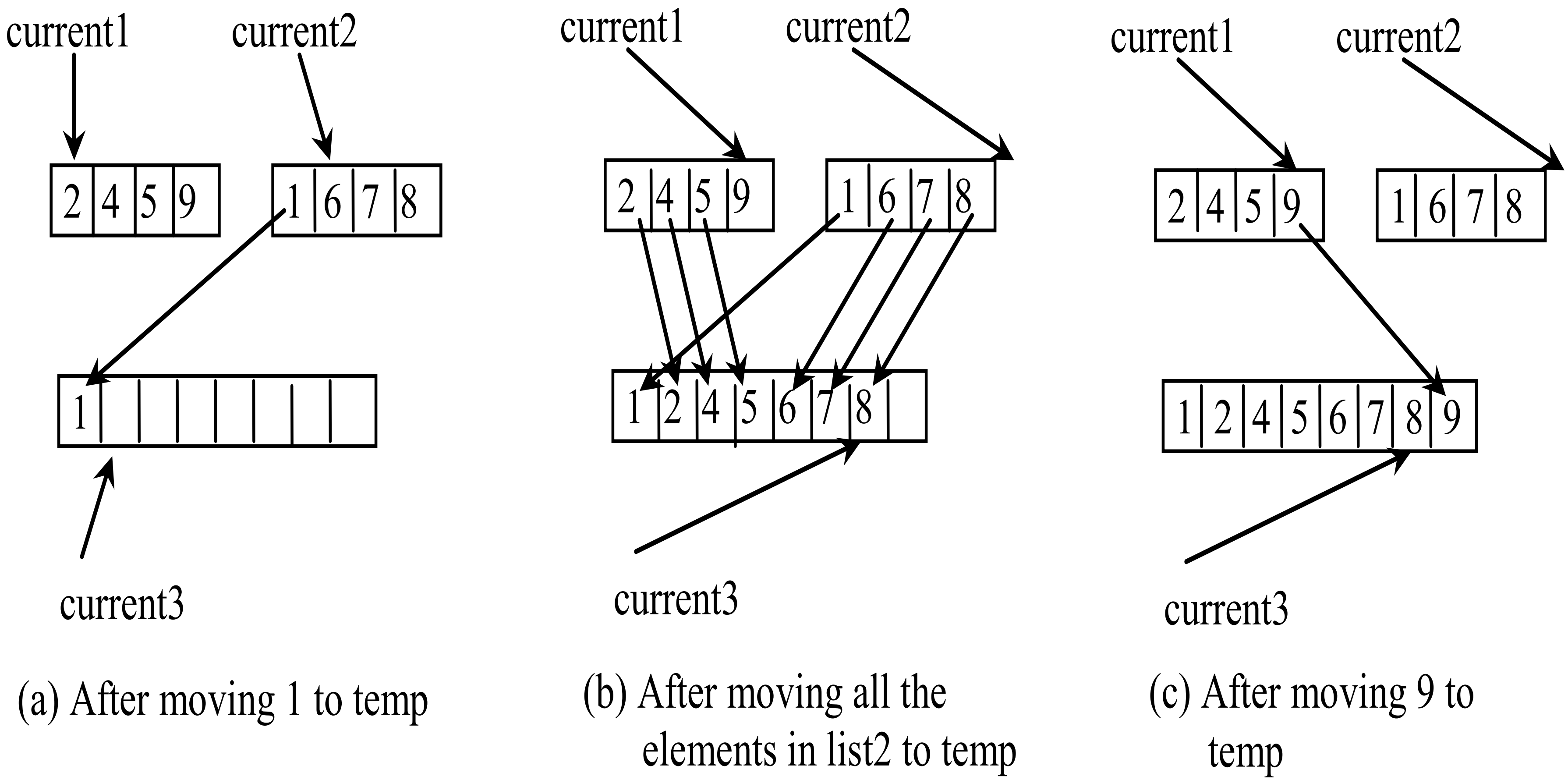
The index of the pivot is returned

Merge Sort



2 9 5 4 8 1 6 7
split
2 9 5 4
8 1 6 7
split
2 9
5 4
8 1
6 7
split
2
9
5
4
8
1
6
7
merge
2 9
4 5
1 8
6 7
merge
2 4 5 9
1 6 7 8
merge
1 2 4 5 6 7 8 9
divide
conquer

Merge Two Sorted Lists



See MergeSort.cpp

Executing Time

Suppose two algorithms perform the same task such as search (linear search vs. binary search) and sorting (selection sort vs. insertion sort). Which one is better? One possible approach to answer this question is to implement these algorithms in Java and run the programs to get execution time. But there are two problems for this approach:

- First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
- Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

Comparison of Quadratic Sorts

Comparison of growth rates

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Sort Review

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$