AP Computer Science A

Java Programming Essentials [Ver.4.0]

Unit 4: Data Collections

CHAPTER 21: RECURSION

DR. ERIC CHOU
IEEE SENIOR MEMBER

AP Computer Science Curriculum

- Recursion (T4.16)
- Recursive Searching and Sorting (T4.17)

Objectives:

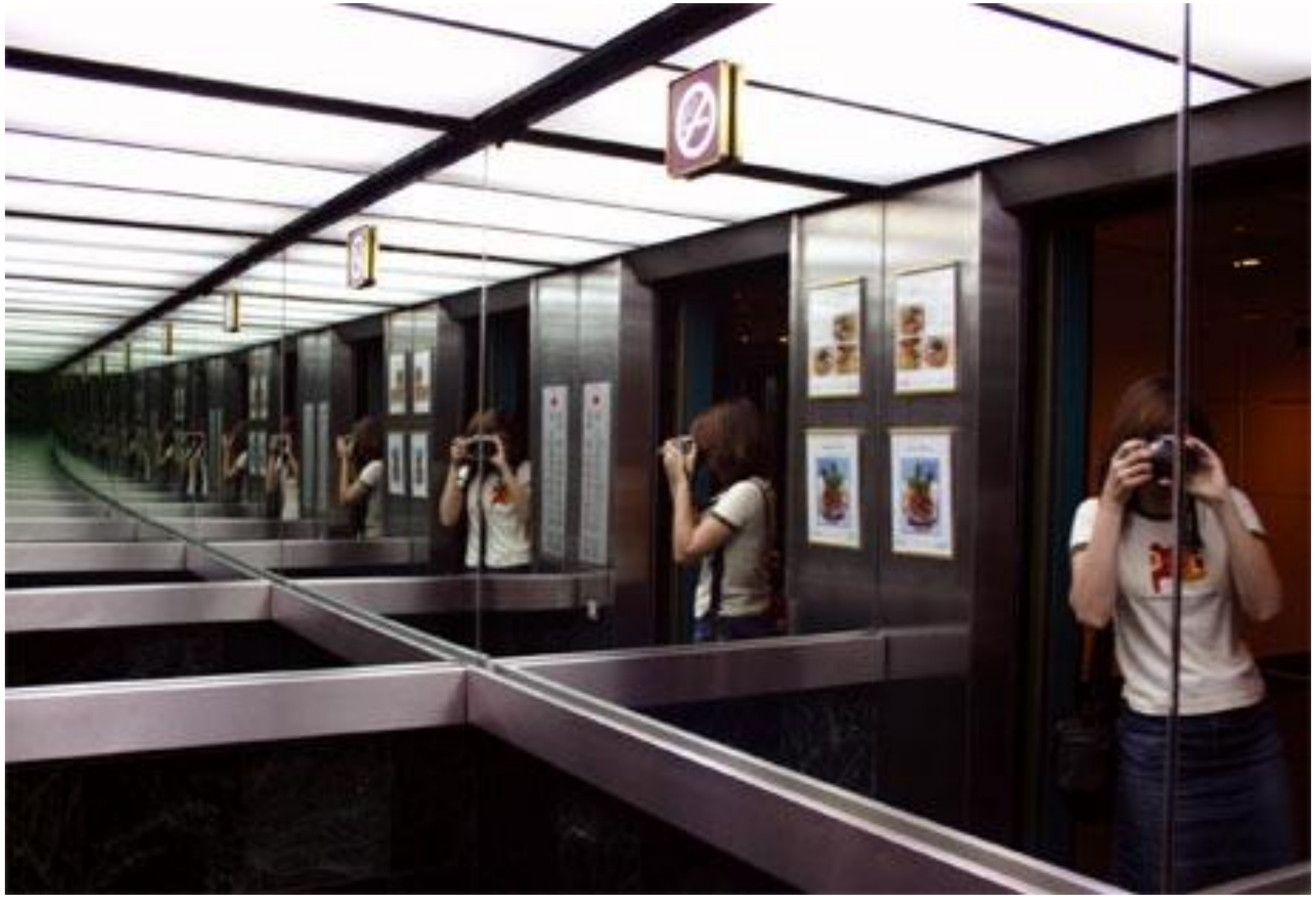
- What is recursion?
- Basic Recursion
- Tail Recursion and Recursive Call with Accumulator
- Recursive Function with Running Index.
- Fibonacci Numbers and Dynamic Programming
- Recursive Processing

Overview

Lecture 1

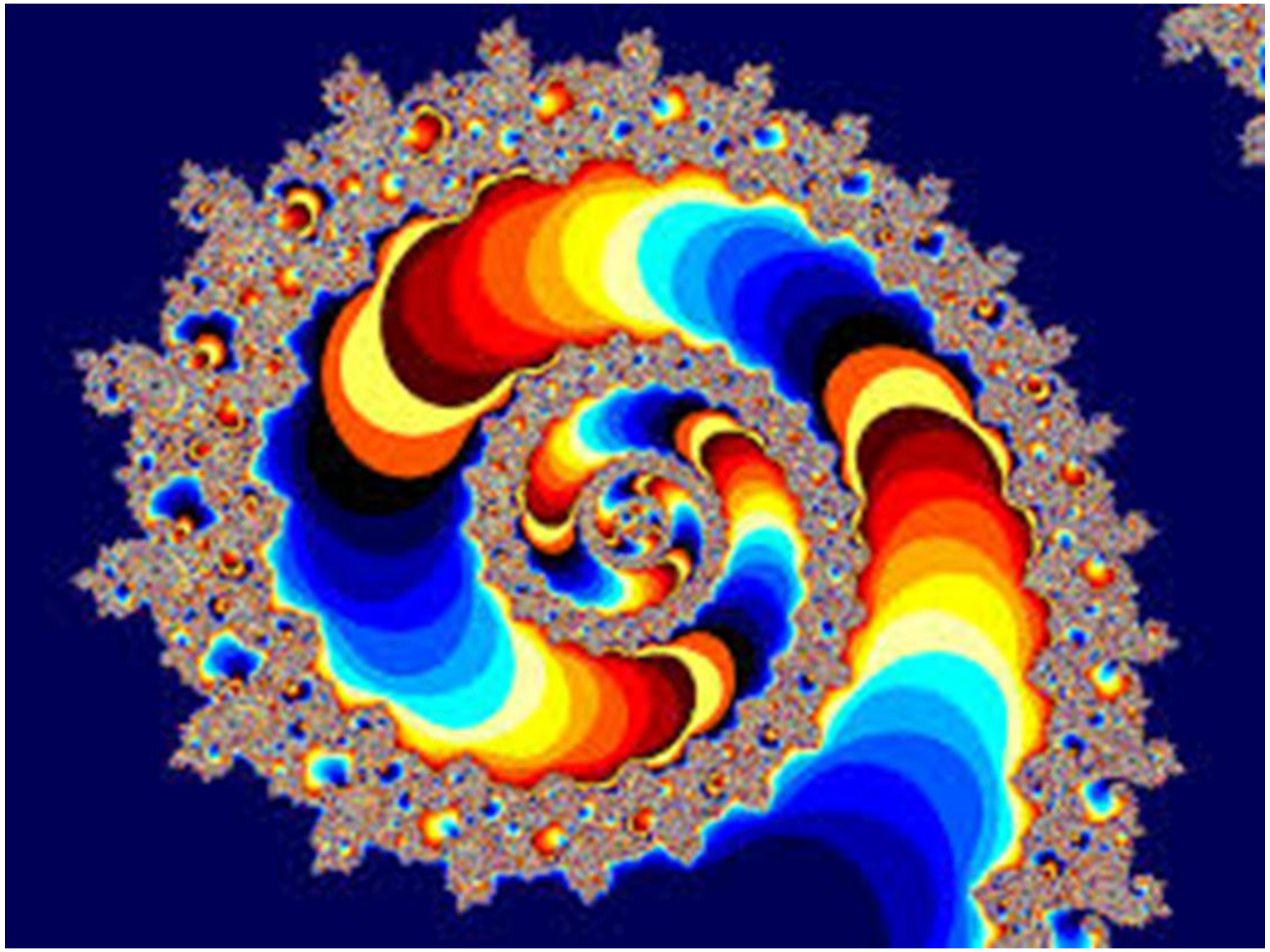
Real World Recursive Problems and Reading Recursive Patterns

Lecture 1



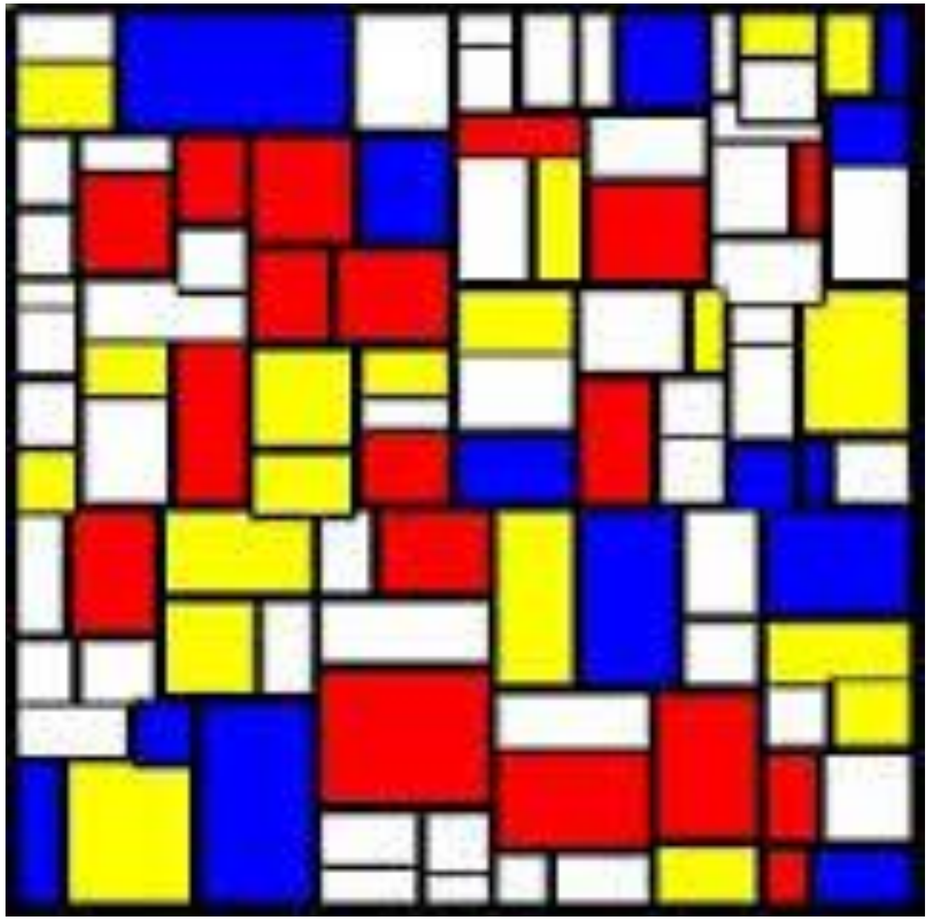
Mirror in Mirror

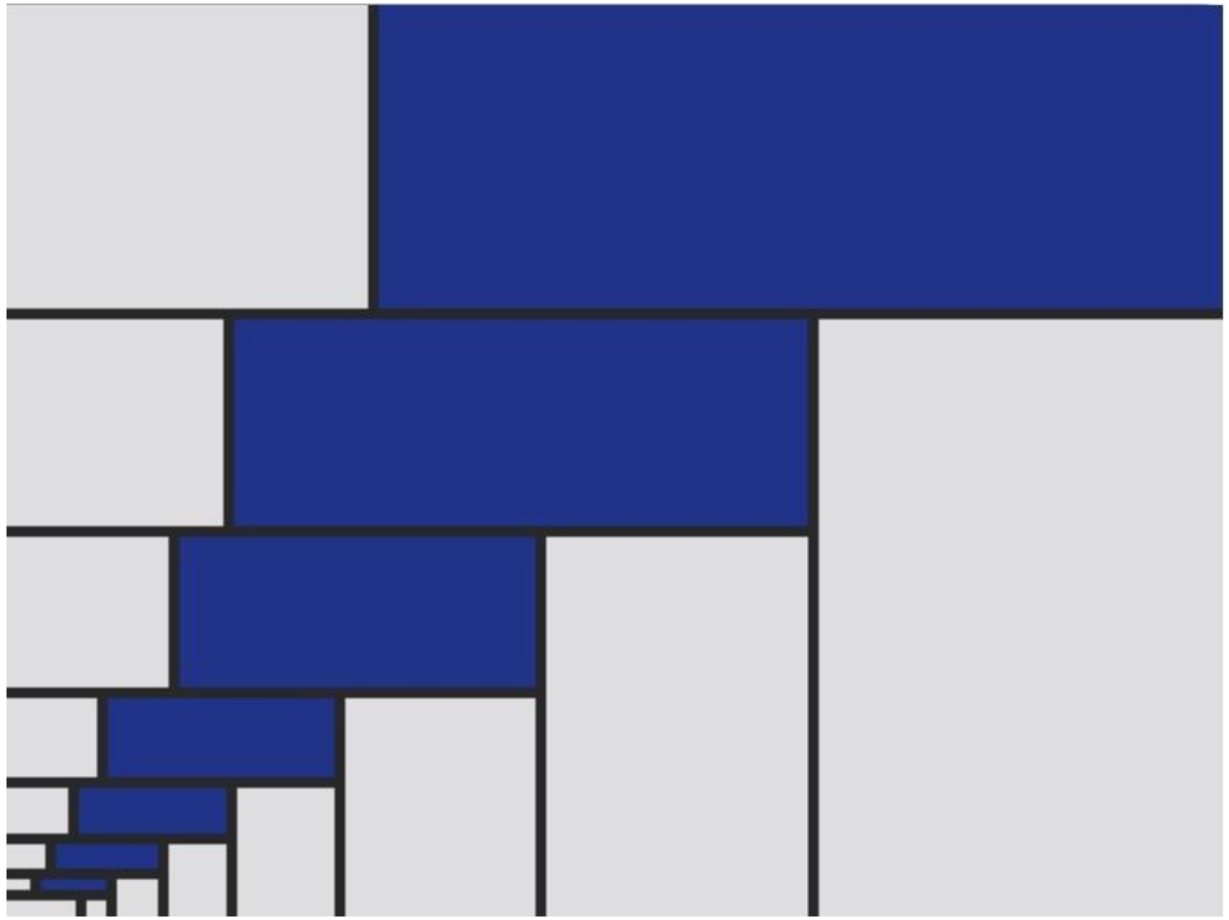
Fractals (Computer Art)



Mondrimat Arts

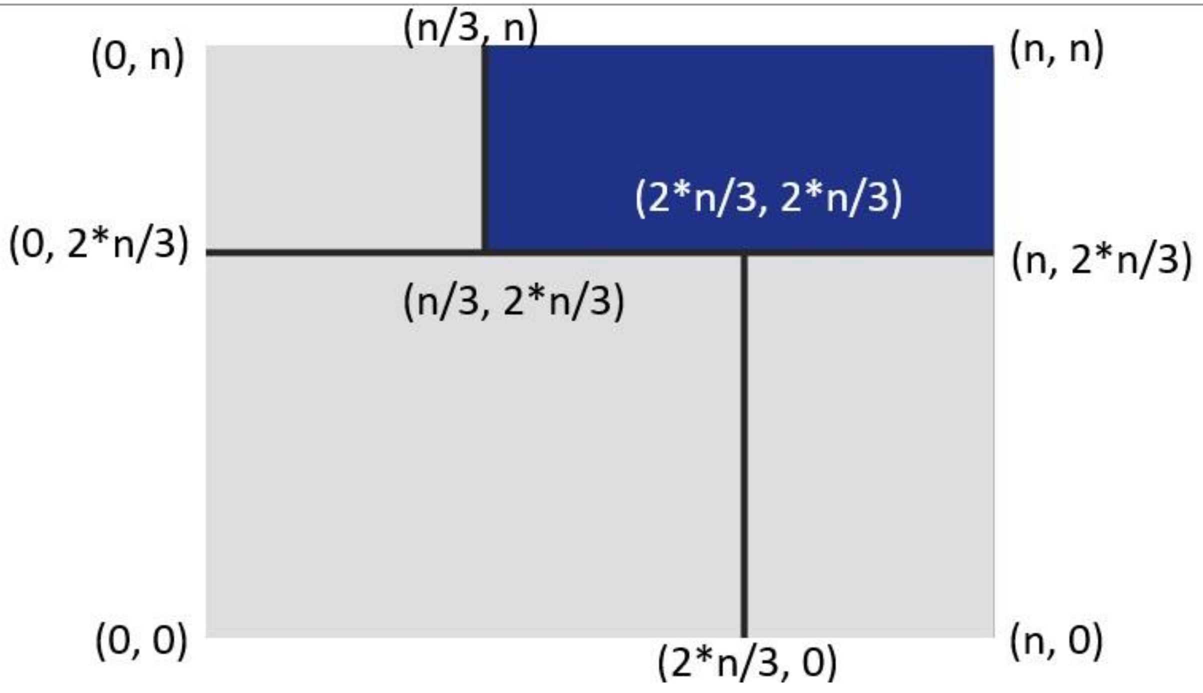
http://www.stephen.com/mondrimat/



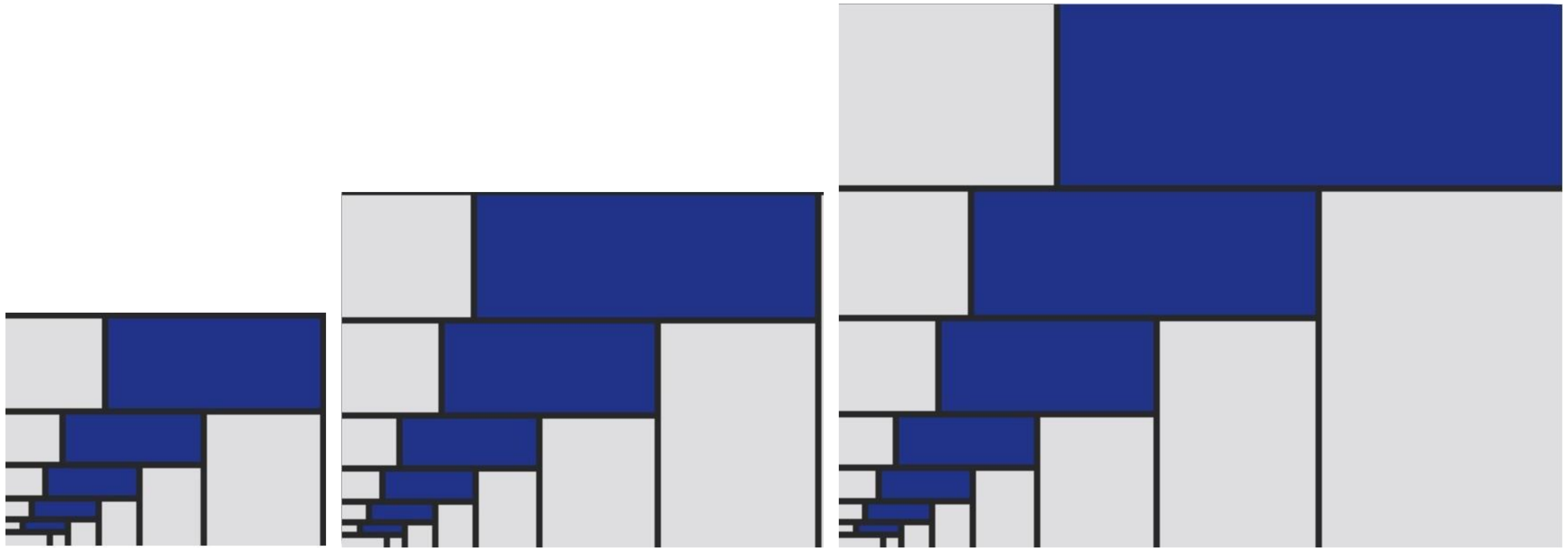


Reading the Recursive Pattern

draw(int n) { /* will draw the following pattern
for one recursion call. */ }



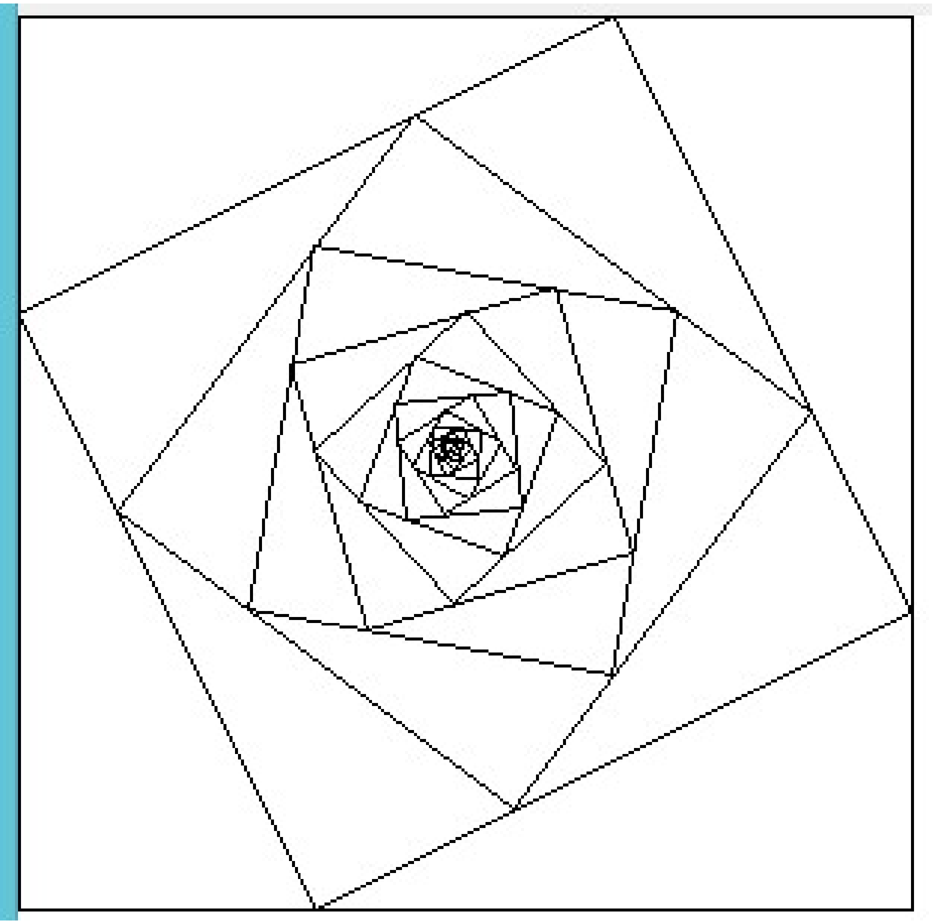
Reading the Recursive Pattern



Three Principles in Recursion

- (1) Divide and Conquer
- (2) Recursive Function
- (3) Stop Condition (Otherwise, it will never stop.)

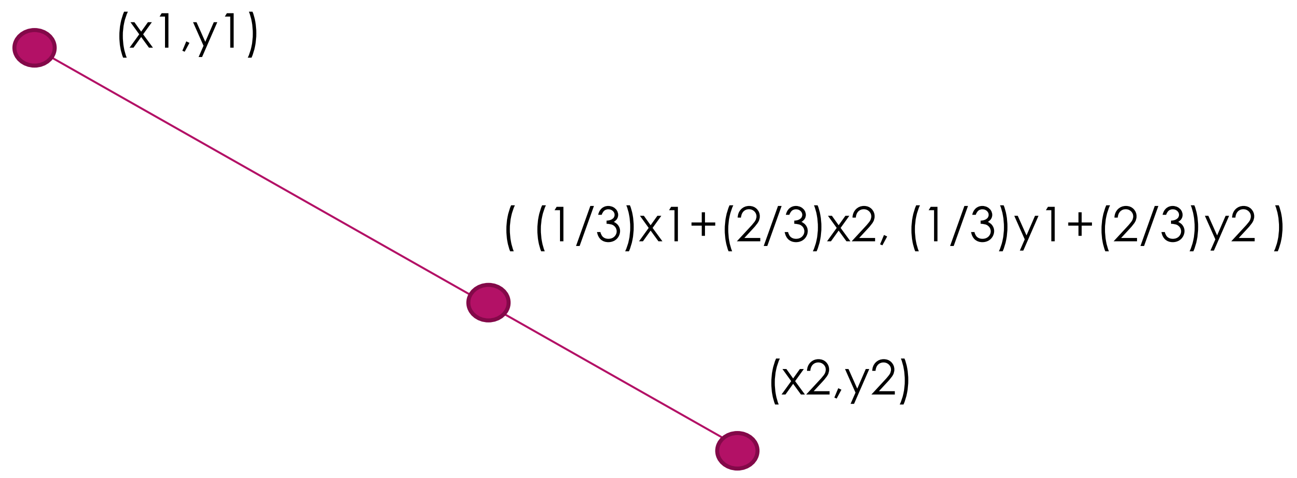
Demo Program: RecursiveRectangle.java



- This recursive rectangle project requires you to draw rectangles recursively on the one-third points of the line segments as shown on the left.
- After finishing a rectangle, you need to draw smaller rectangles based on the one-third points of the previous rectangle's line segments. (Recursion)
- Until the line segment's length is less than square root of ten, which also means the square of the line segment's length is less than 10. Then, you stop the recursion. (Stop Condition)
- Your results should look like the picture on the left side.

Demo Program: RecursiveRectangle.java

Calculation of the one-third point



Demonstration Program

RECURSIVERECTANGLE.JAVA

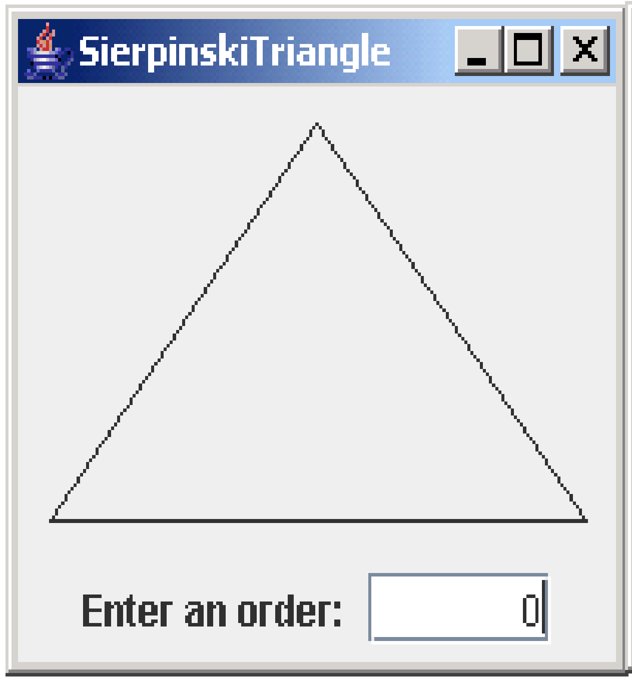
Fractals?

A fractal is a geometrical figure just like triangles, circles, and rectangles, but fractals can be divided into parts, each of which is a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, called *Sierpinski triangle*, named after a famous Polish mathematician.

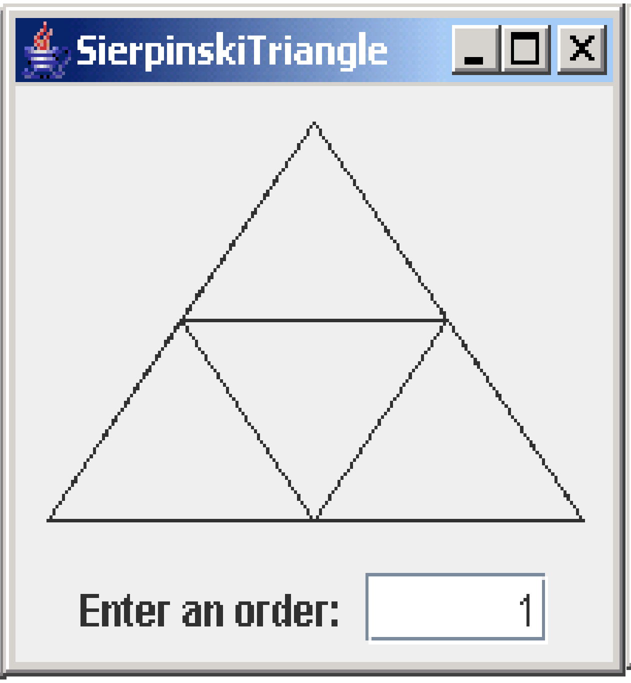
Sierpinski Triangle

- It begins with an equilateral triangle, which is considered to be the Sierpinski fractal of order (or level) 0, as shown in Figure (a).
- Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1, as shown in Figure (b).
- Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2, as shown in Figure (c).
- You can repeat the same process recursively to create a Sierpinski triangle of order 3, 4, ..., and so on, as shown in Figure (d).

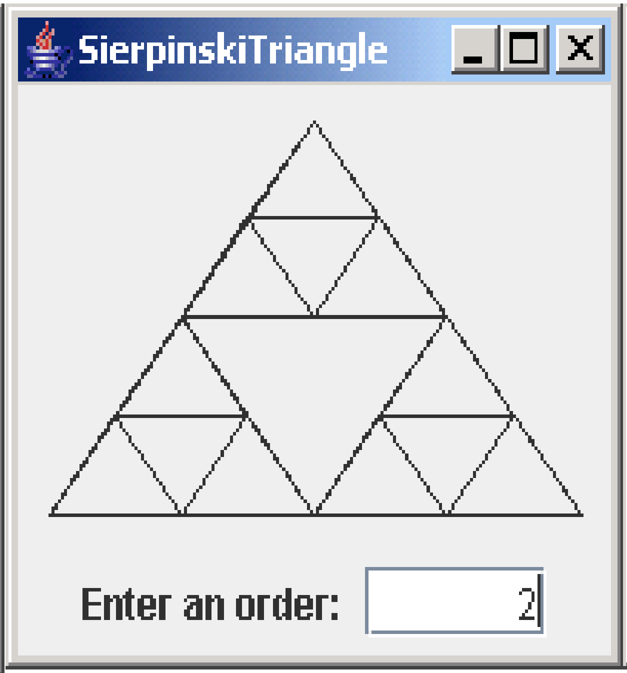
Recursion


SierpinskiTriangle

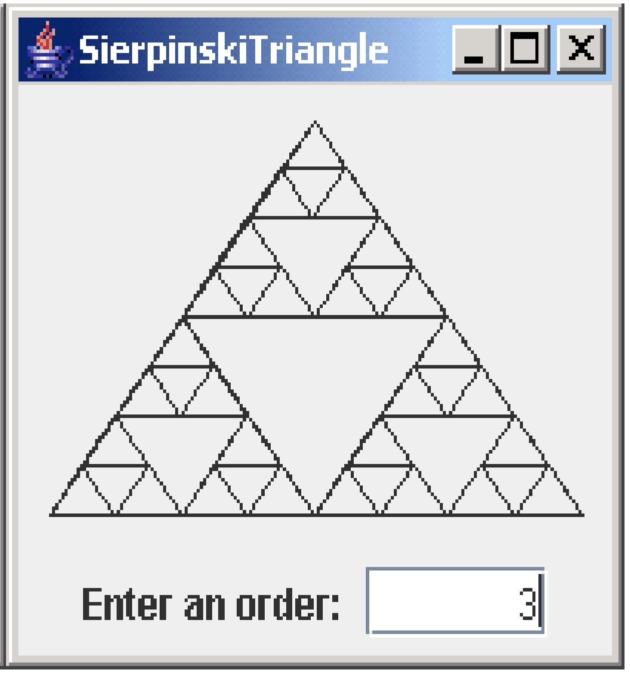
Enter an order: 0


SierpinskiTriangle

Enter an order: 1

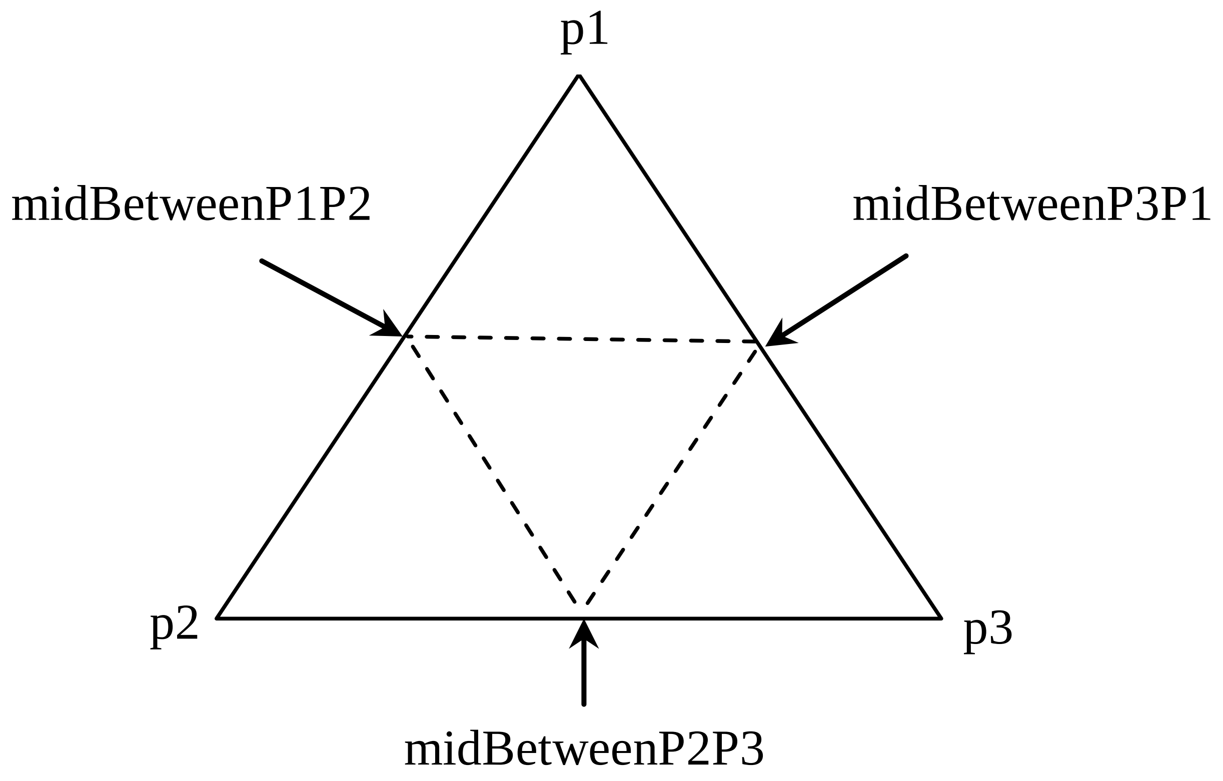

SierpinskiTriangle

Enter an order: 2


SierpinskiTriangle

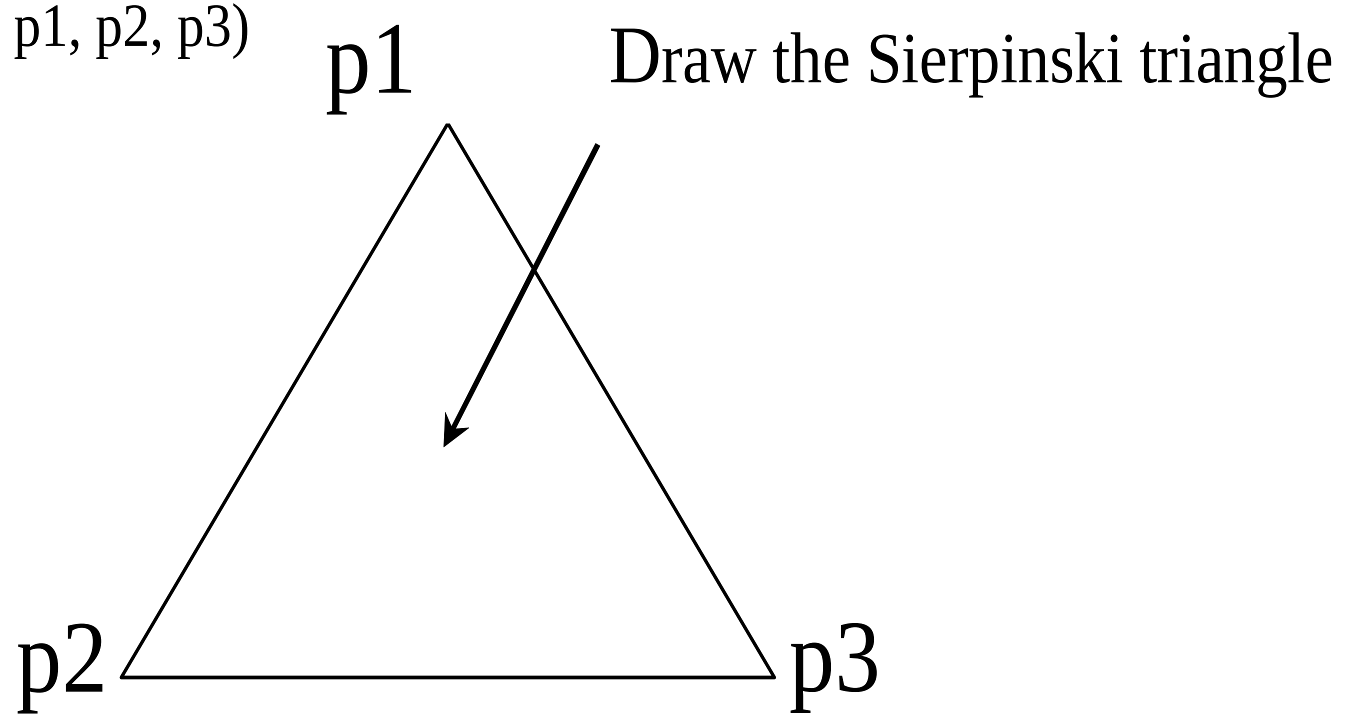
Enter an order: 3

Sierpinski Triangle Solution



Sierpinski Triangle Solution

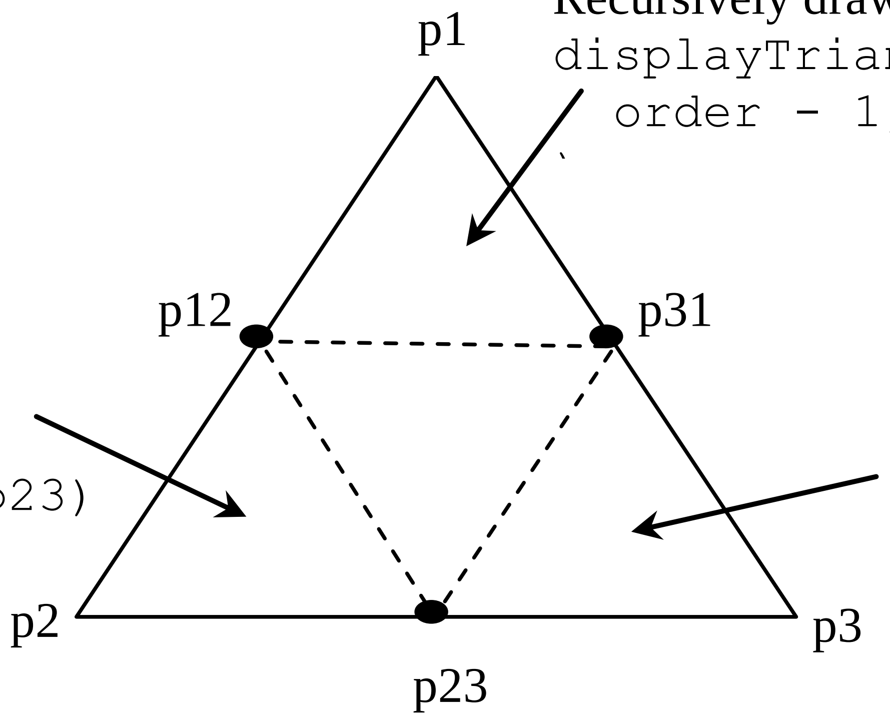
displayTriangles(g, order, p1, p2, p3)



Sierpinski Triangle Solution

Recursively draw the small Sierpinski triangle

```
displayTriangles(g,
  order - 1, p1, p12, p31)
```



Recursively draw the small
Sierpinski triangle

```
displayTriangles(g,
  order - 1, p12, p2, p23)
```

Recursively draw the
small Sierpinski triangle

```
displayTriangles(g,
  order - 1, p31, p23, p3)
```

Demonstration Program

SIERPINSKITRIANGLES.JAVA

Basic Recursion

Lecture 1

Recursion

- Base Condition (Termination Condition)
- Recursive Structure
  - Divide and Conquer: Split a big problem to smaller ones
  - Incremental Design: explore the relationship between small ones and the large one
  - Recursive Formula: describe the recursive relationship into Java code

Explicit Formula Versus Recursive Formula

Base Case: $a_0 = 0$

$a_1 = 1$

$a_2 = 2$

…

Explicit Formula: $a_n = n$

Base Case: $a_0 = 0$

$a_1 = a_0 + 1$

$a_2 = a_1 + 1$

…

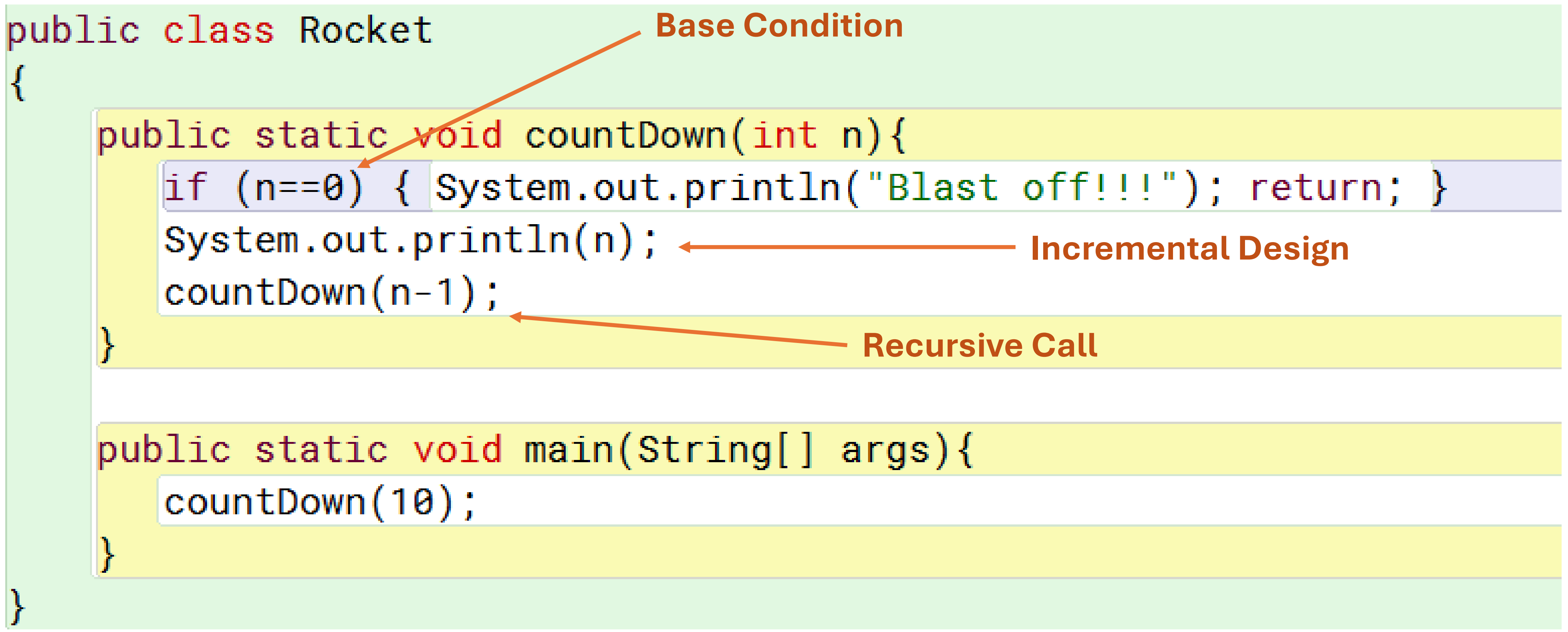
Recursive Formula: $a_n = a_{n-1} + 1$

Countdown Recursion

- Call a recursive function with a counter.
- The counter decreases for each sub-case.
- When counter reaches 0, the rocket blasts off.

Demo Program:

Rocket.java



```java
public class Rocket
{
    public static void countDown(int n){
        if (n==0) { System.out.println("Blast off!!!"); return; }
        System.out.println(n);
        countDown(n-1);
    }

    public static void main(String[] args){
        countDown(10);
    }
}
```

Demonstration Program

ROCKET.JAVA

Countdown Recursion with Return Value

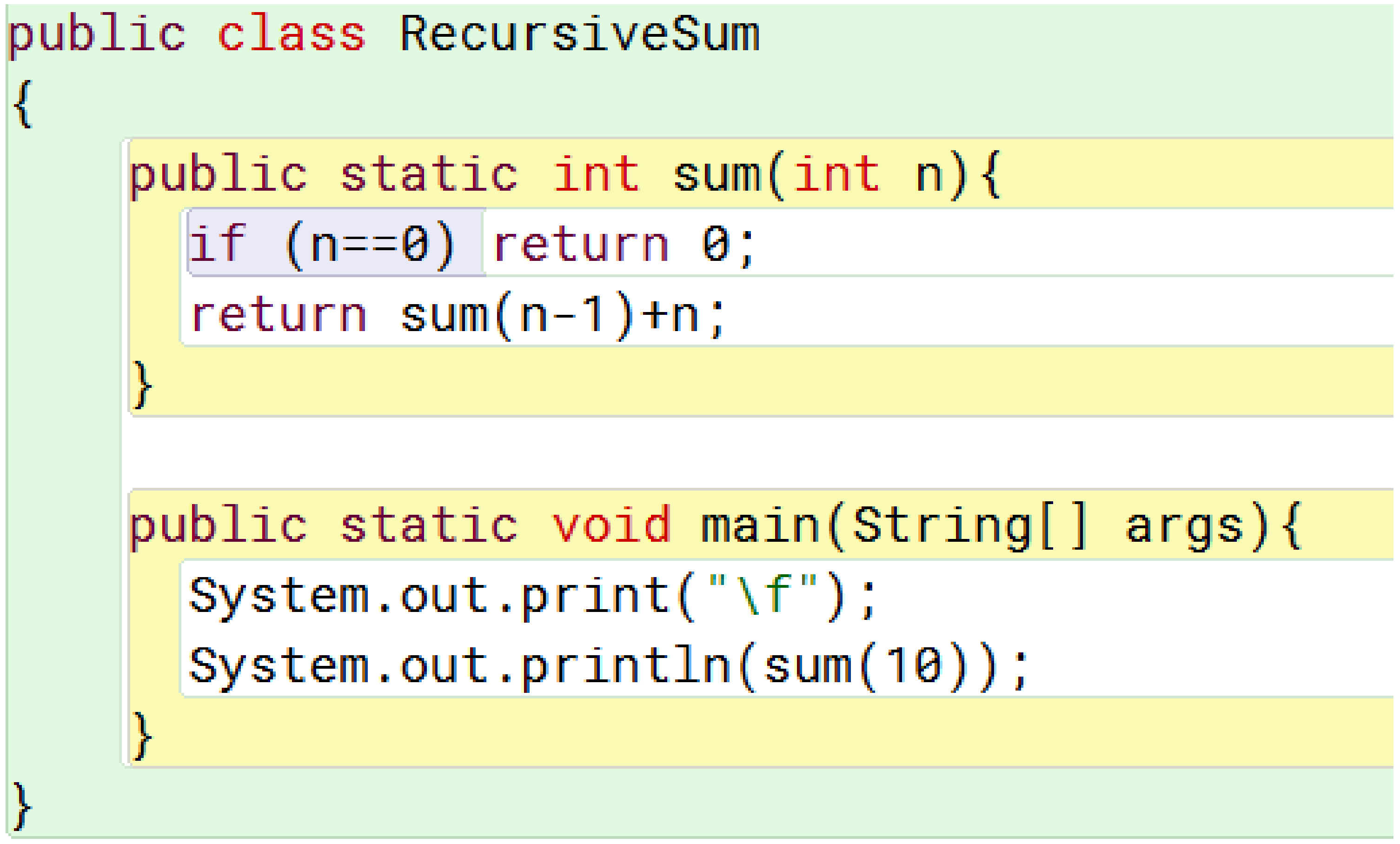
- Call a recursive function with a counter.
- The counter decreases for each sub-case.
- When counter reaches 0, the base case return a value.
- Each bigger case receive a return value and returning another value.

Recursive Sum

Lecture 1

Demo Program:

RecursiveSum.java



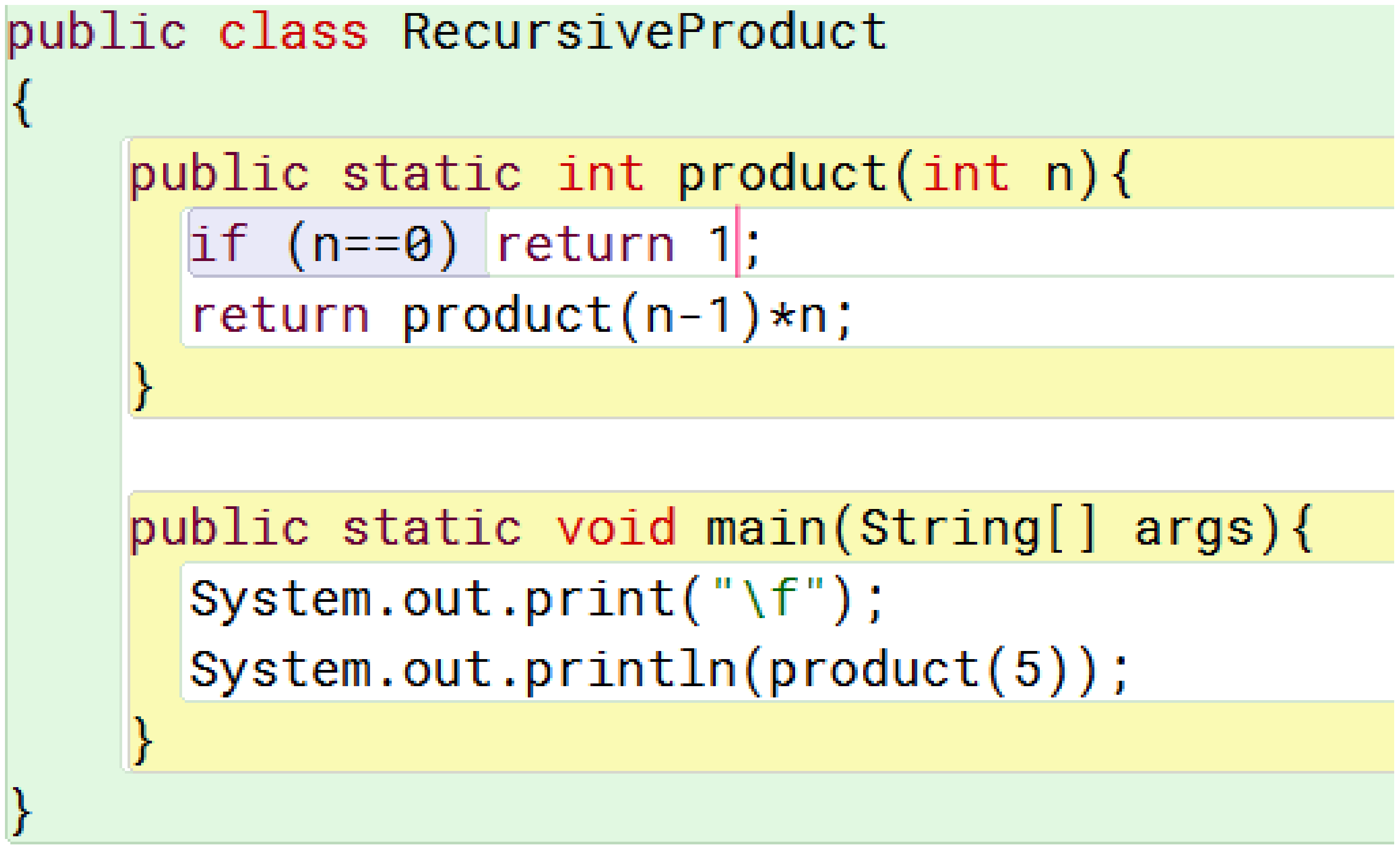
```java
public class RecursiveSum
{
    public static int sum(int n){
        if (n==0) return 0;
        return sum(n-1)+n;
    }

    public static void main(String[] args){
        System.out.print("\f");
        System.out.println(sum(10));
    }
}
```

Demonstration Program

RECURSIVESUM.JAVA

Demo Program:

RecursiveProduct.java



```
public class RecursiveProduct
{
    public static int product(int n){
      if (n==0) return 1;
      return product(n-1)*n;
    }

    public static void main(String[] args){
      System.out.print("\f");
      System.out.println(product(5));
    }
}
```

Demonstration Program

RECURSIVEPRODUCT.JAVA

Recursion with Accumulator

Lecture 1

Recursion with Accumulator

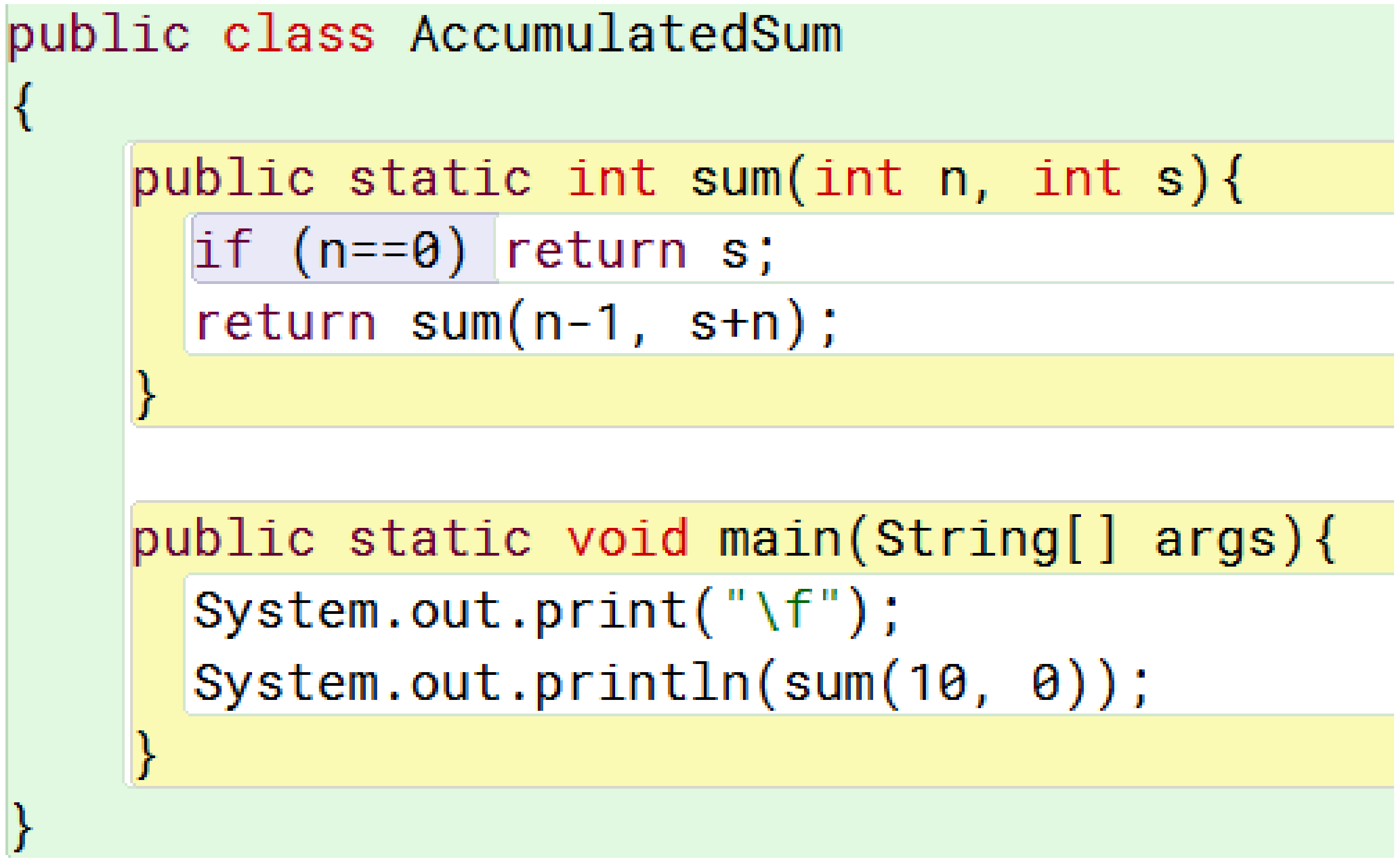
- Call a recursive function with an accumulator as a parameter.
- In the base case, the accumulator is returned.
- Each recursive call, the accumulator is updated (accumulates).

Demonstration Program

ACCUMULATEDSUM.JAVA

Demo Program:

AccumulatedSum.java



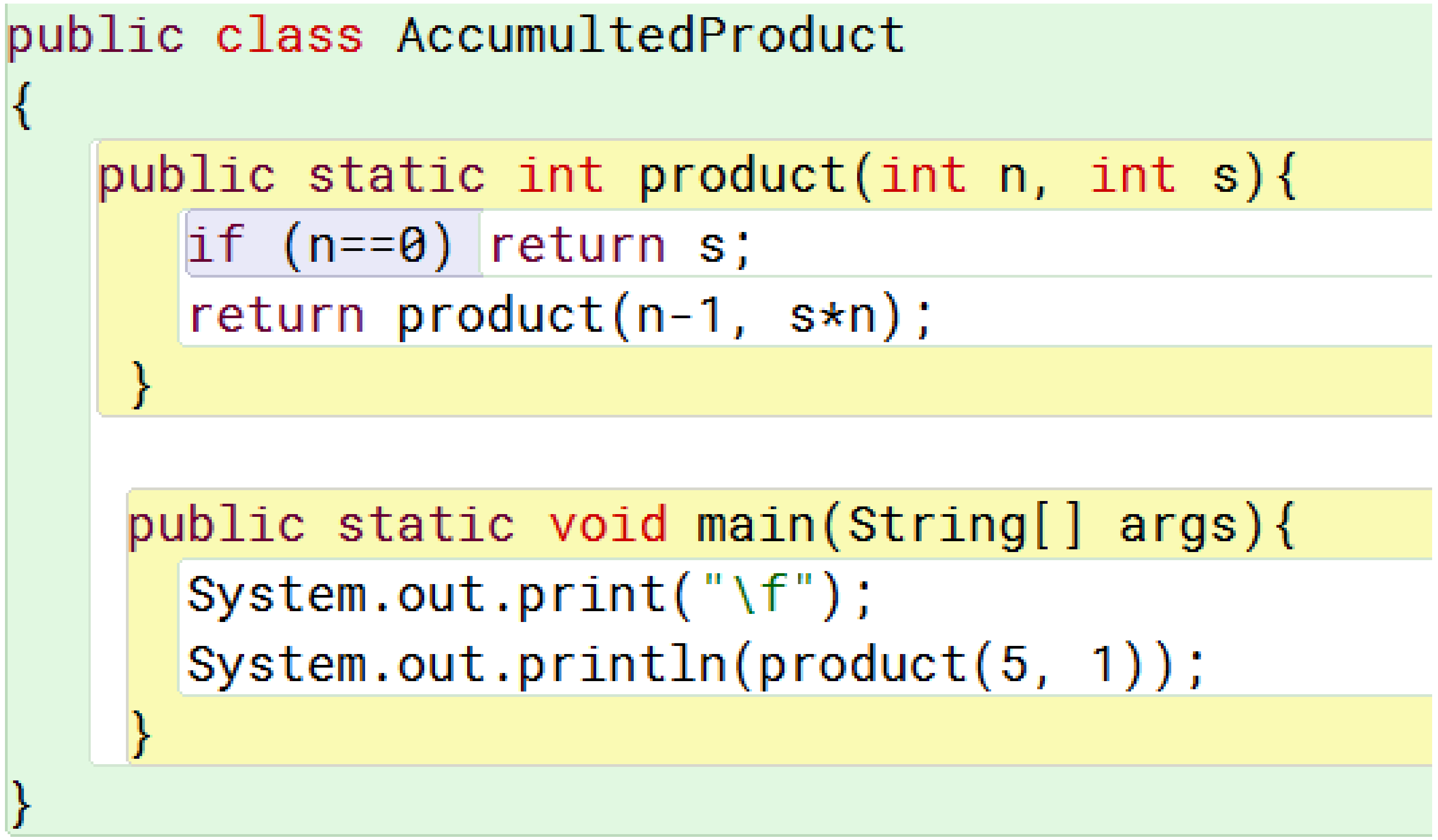
```
public class AccumulatedSum
{
    public static int sum(int n, int s){
      if (n==0) return s;
      return sum(n-1, s+n);
    }

    public static void main(String[] args){
      System.out.print("\f");
      System.out.println(sum(10, 0));
    }
}
```

Demonstration Program

ACCUMULATEDPRODUCT.JAVA

Demo Program:

AccumulatedProduct.java



```java
public class AccumultedProduct
{
    public static int product(int n, int s){
        if (n==0) return s;
        return product(n-1, s*n);
    }

    public static void main(String[] args){
        System.out.print("\f");
        System.out.println(product(5, 1));
    }
}
```

Why with accumulator?

- Tail recursion.
- More efficient than normal recursive function.
- As a recursive helper function.

Tail Recursion

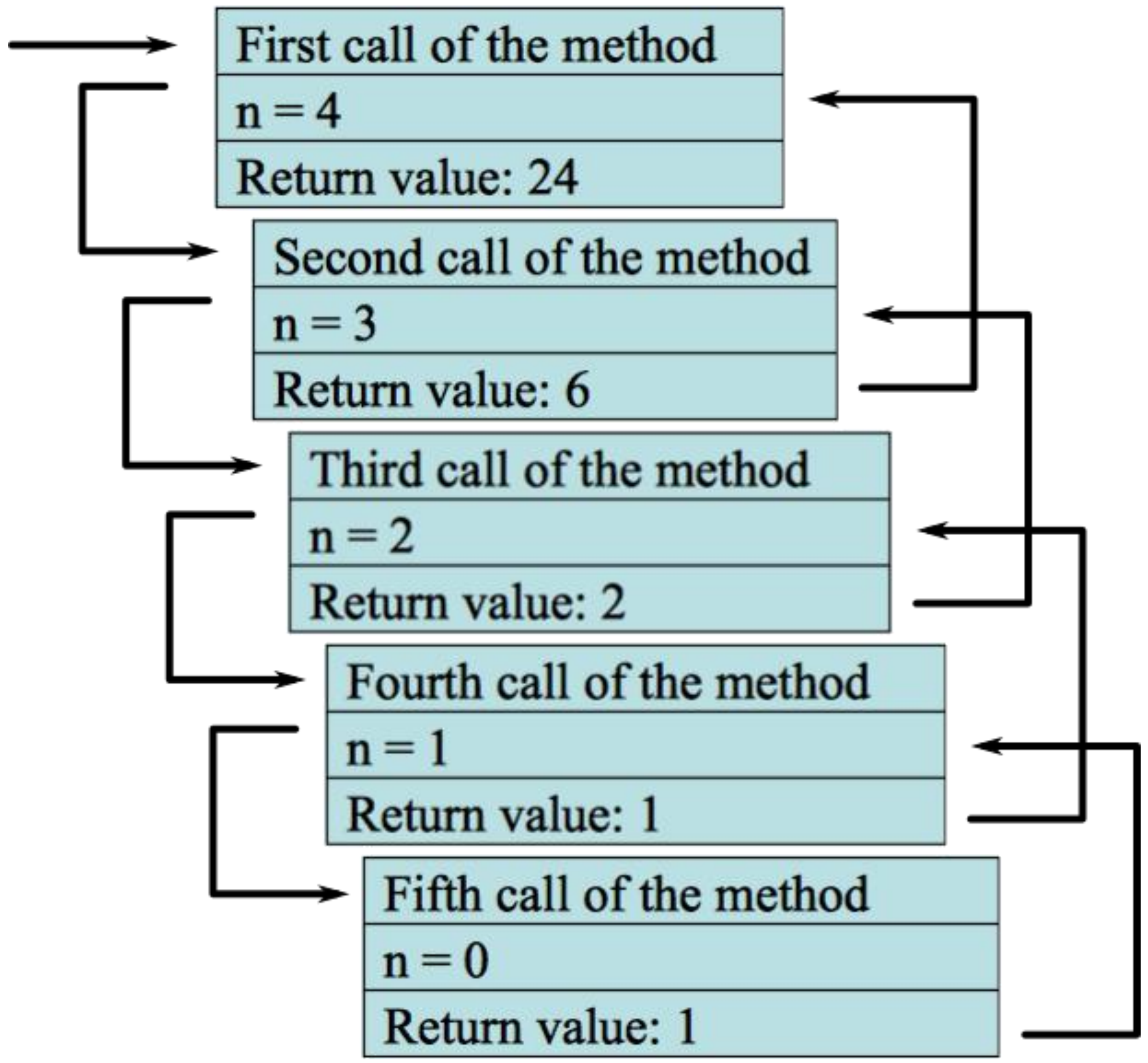
Lecture 1

Tail Recursion

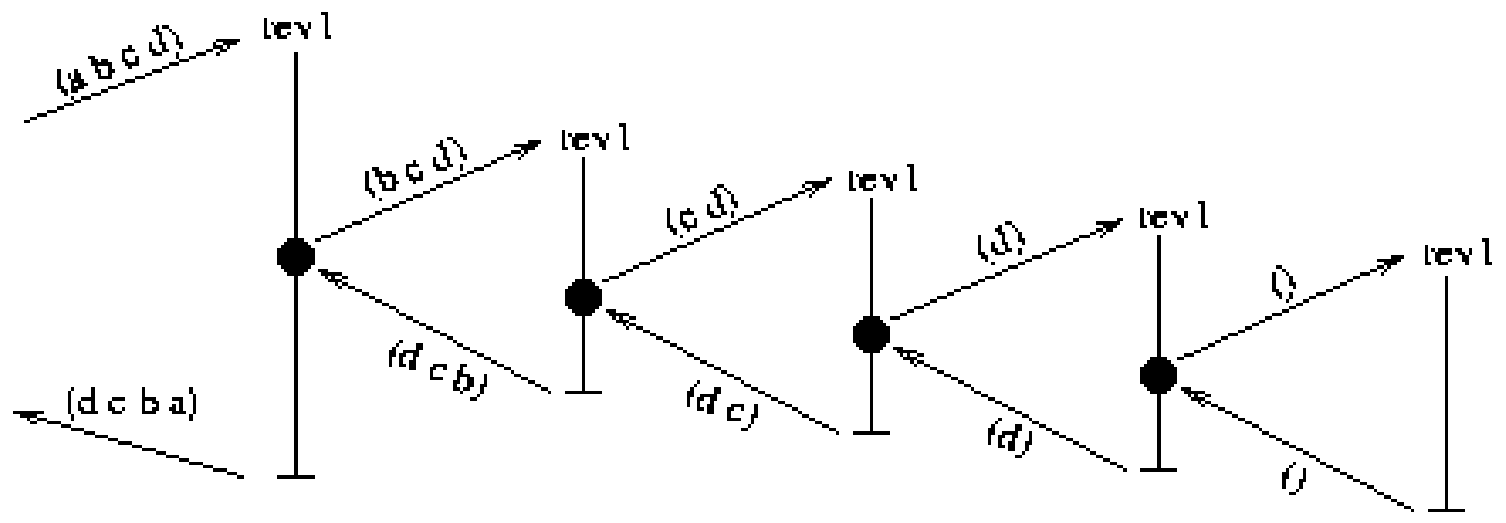
- A **tail recursion** is a **recursive function** where the **function** calls itself at the end ("**tail**") of the **function** in which no computation is done after the return of **recursive** call.
- Many compilers optimize to change a **recursive** call to a **tail recursive** or an iterative call. …
- Hence the compiler can do away with a stack.

Tail Recursion

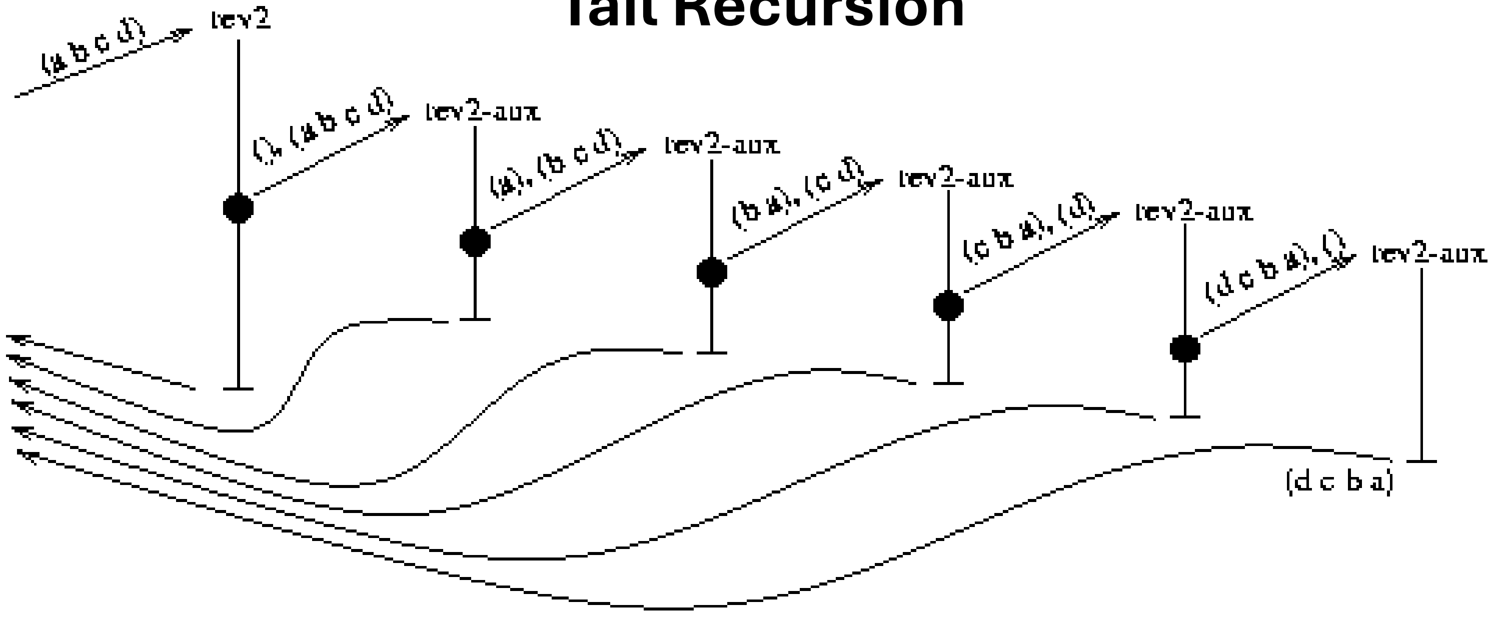
- A tail-recursive function is just a function whose very the last action is a call to itself. Tail-Call Optimisation (TCO) lets us convert regular recursive calls into tail calls to make recursions practical for large inputs, which was earlier leading to stack overflow error in normal recursion scenario.
- Java does not directly support TCO at the compiler level, but with the introduction of lambda expressions and functional interfaces in JAVA 8, we can implement this concept in a few lines of code.


First call of the method
n = 4
Return value: 24
Second call of the method
n = 3
Return value: 6
Third call of the method
n = 2
Return value: 2
Fourth call of the method
n = 1
Return value: 1
Fifth call of the method
n = 0
Return value: 1

Non-Tail Recursion



Tail Recursion



Computing Factorial

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

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

Computing Factorial

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

```
factorial(3)
```

Computing Factorial

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

```
factorial(3) = 3 * factorial(2)
```

Computing Factorial

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

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

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)))
```

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)))
```

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)
```

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
```

Computing Factorial

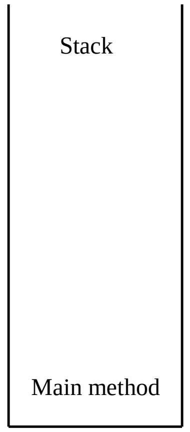
```
factorial(4) = 4 * factorial(3)
             = 4 * 3 * factorial(2)
             = 4 * 3 * (2 * factorial(1))
             = 4 * 3 * ( 2 * (1 * factorial(0)))
             = 4 * 3 * ( 2 * ( 1 * 1)))
             = 4 * 3 * ( 2 * 1)
             = 4 * 3 * 2
             = 4 * 6
             = 24
```

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

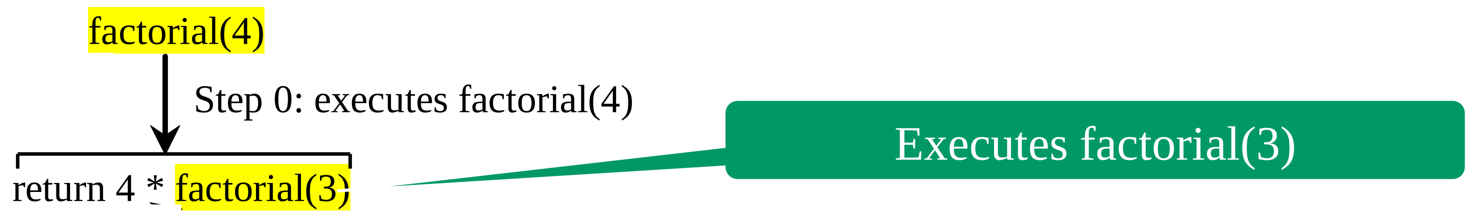
Trace Recursive factorial

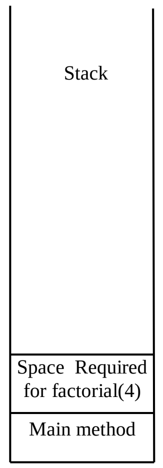
Executes factorial(4)

factorial(4)

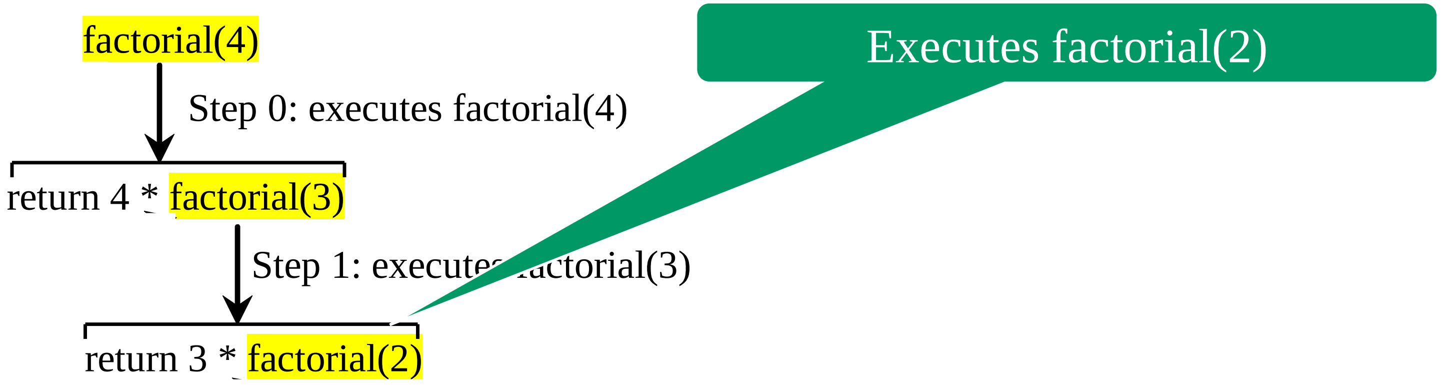


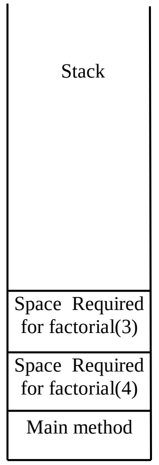
Trace Recursive factorial



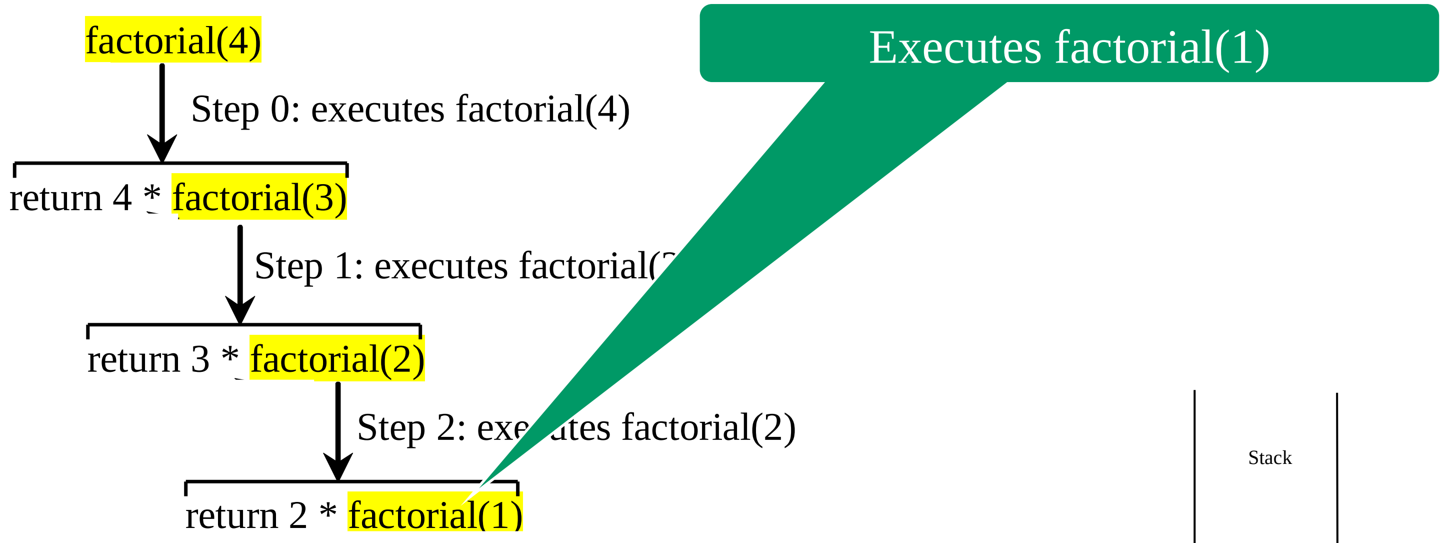


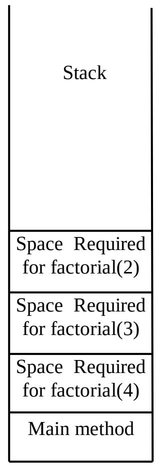
Trace Recursive factorial



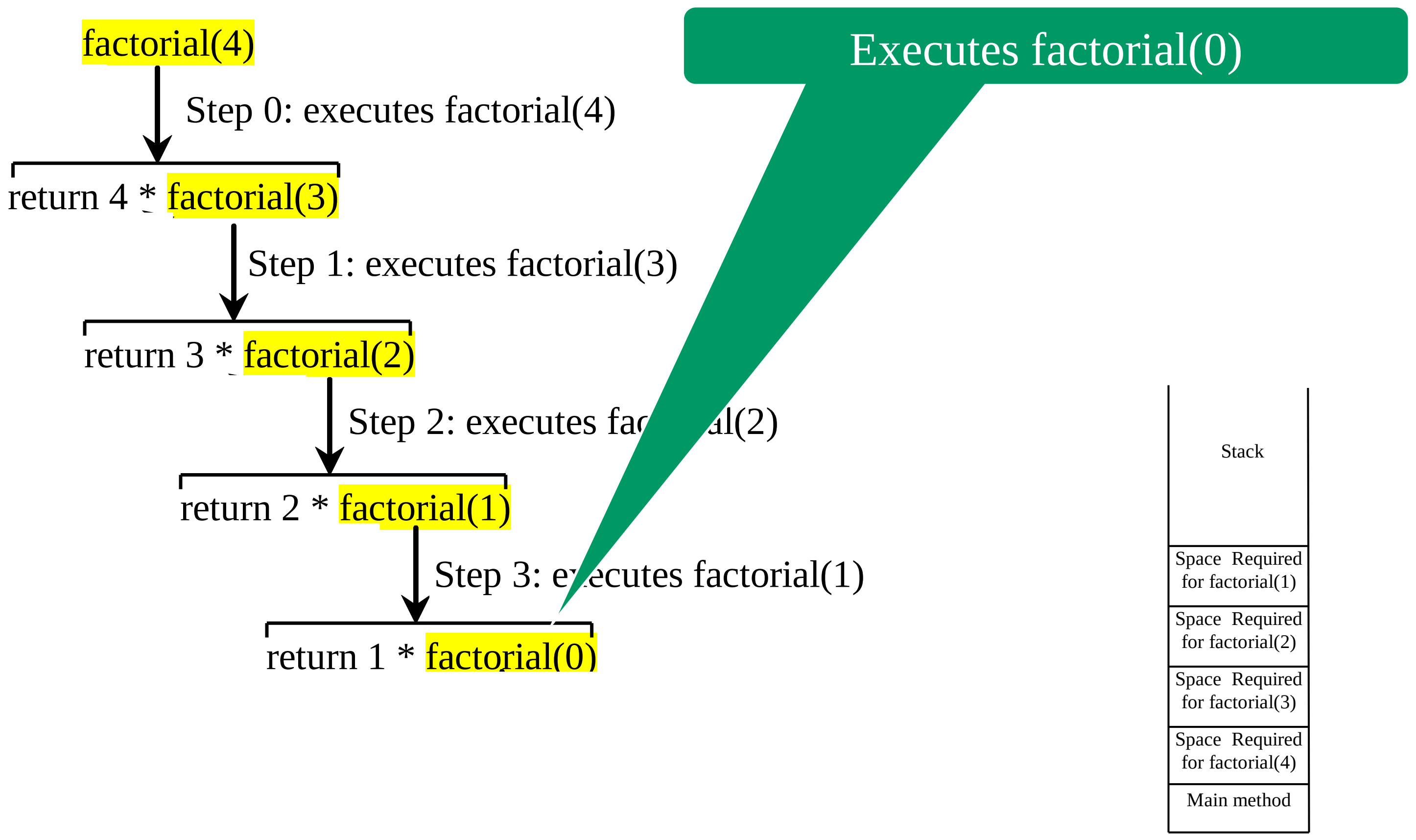

Stack
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

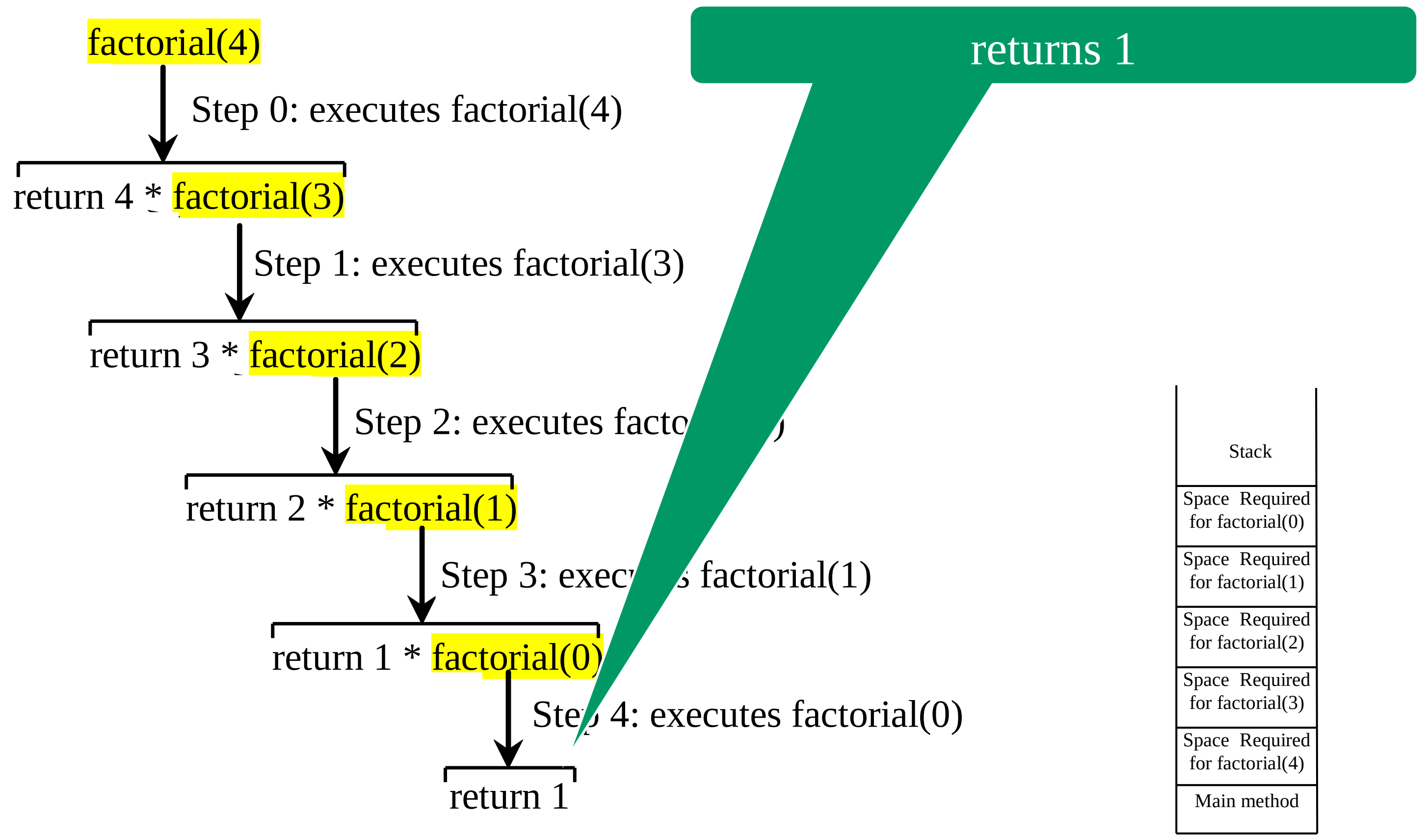



Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

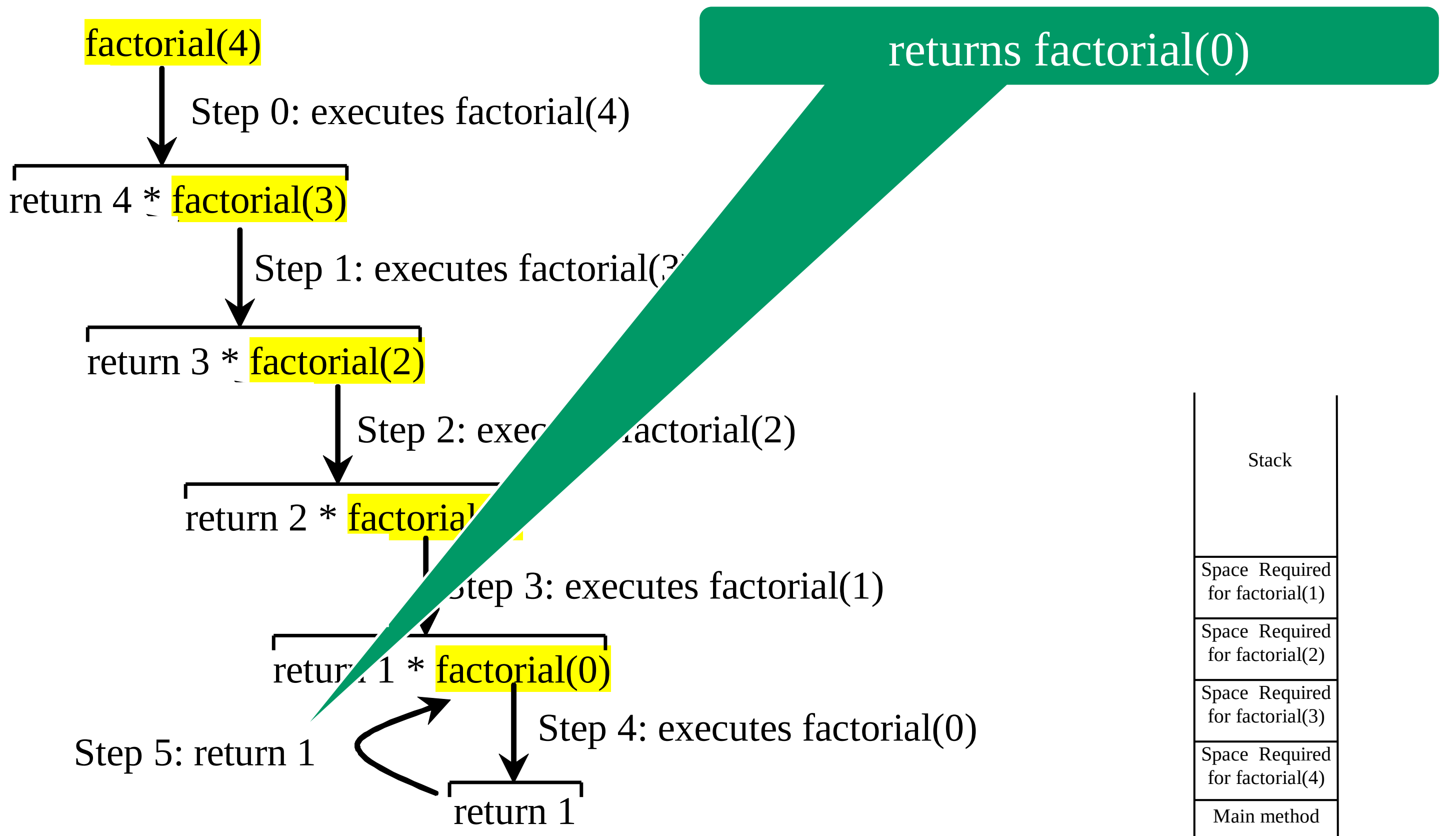
Trace Recursive factorial

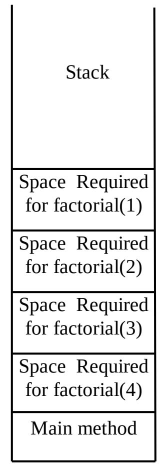

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes fac al(2)
return 2 * factorial(1)
Step 3: e ecutes factorial(1)
return 1 * factorial(0)
Executes factorial(0)
Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

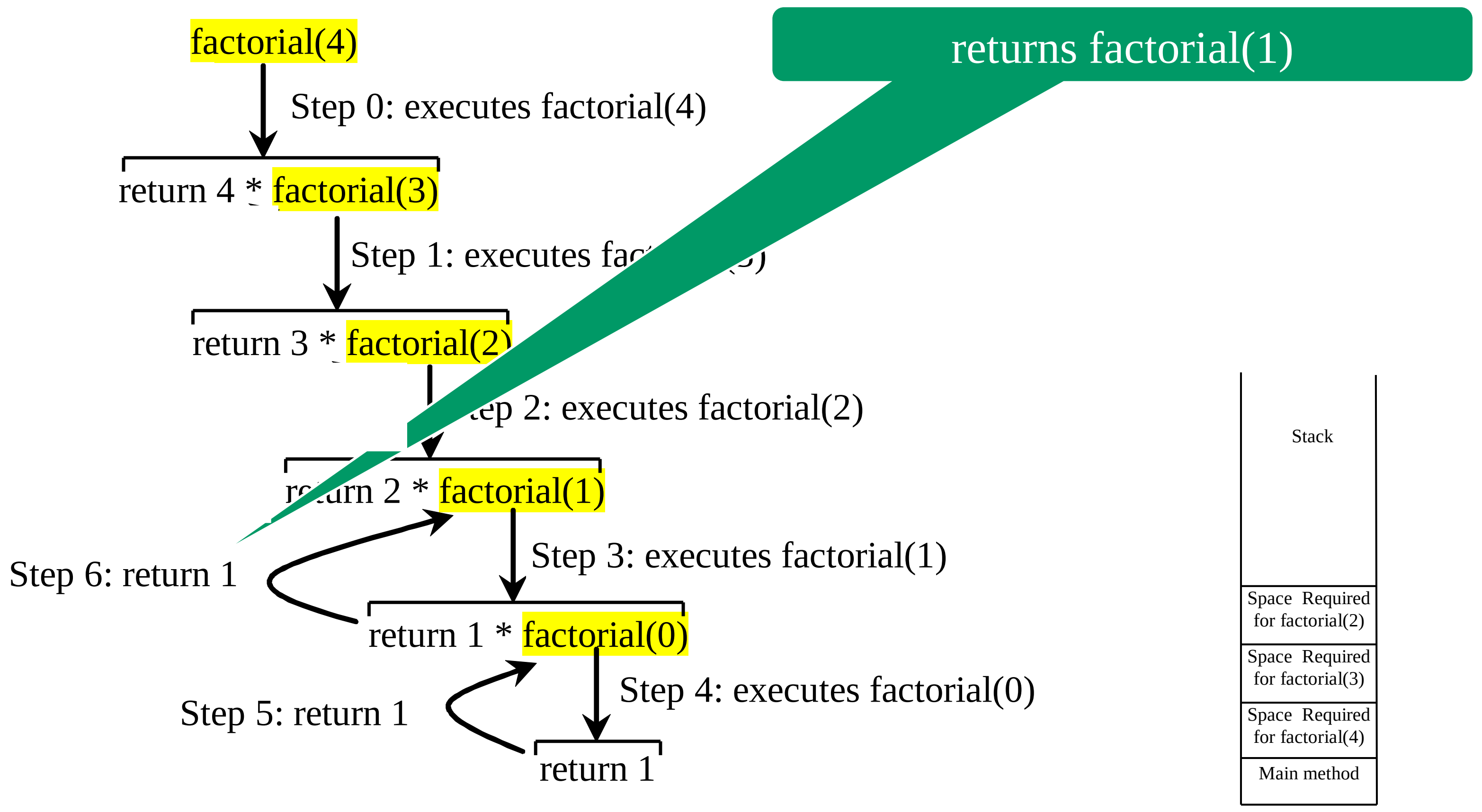

factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes facto
return 2 * factorial(1)
Step 3: exec s factorial(1)
return 1 * factorial(0)
Ste 4: executes factorial(0)
return 1
returns 1
Stack
Space Required for factorial(0)
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

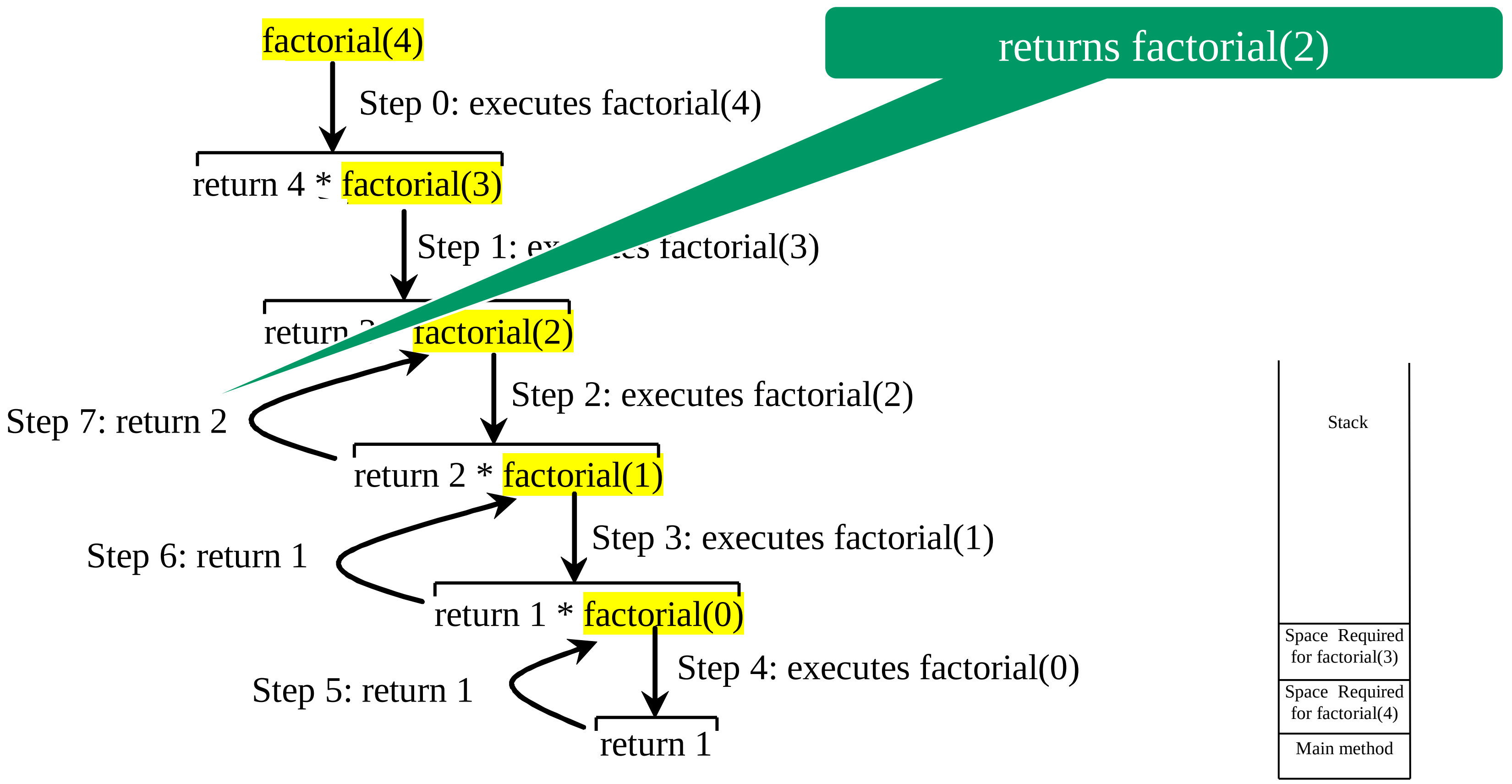

returns factorial(0)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes factorial(3)
return 3 * factorial(2)
Step 2: executes factorial(2)
return 2 * factorial
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1


Stack
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

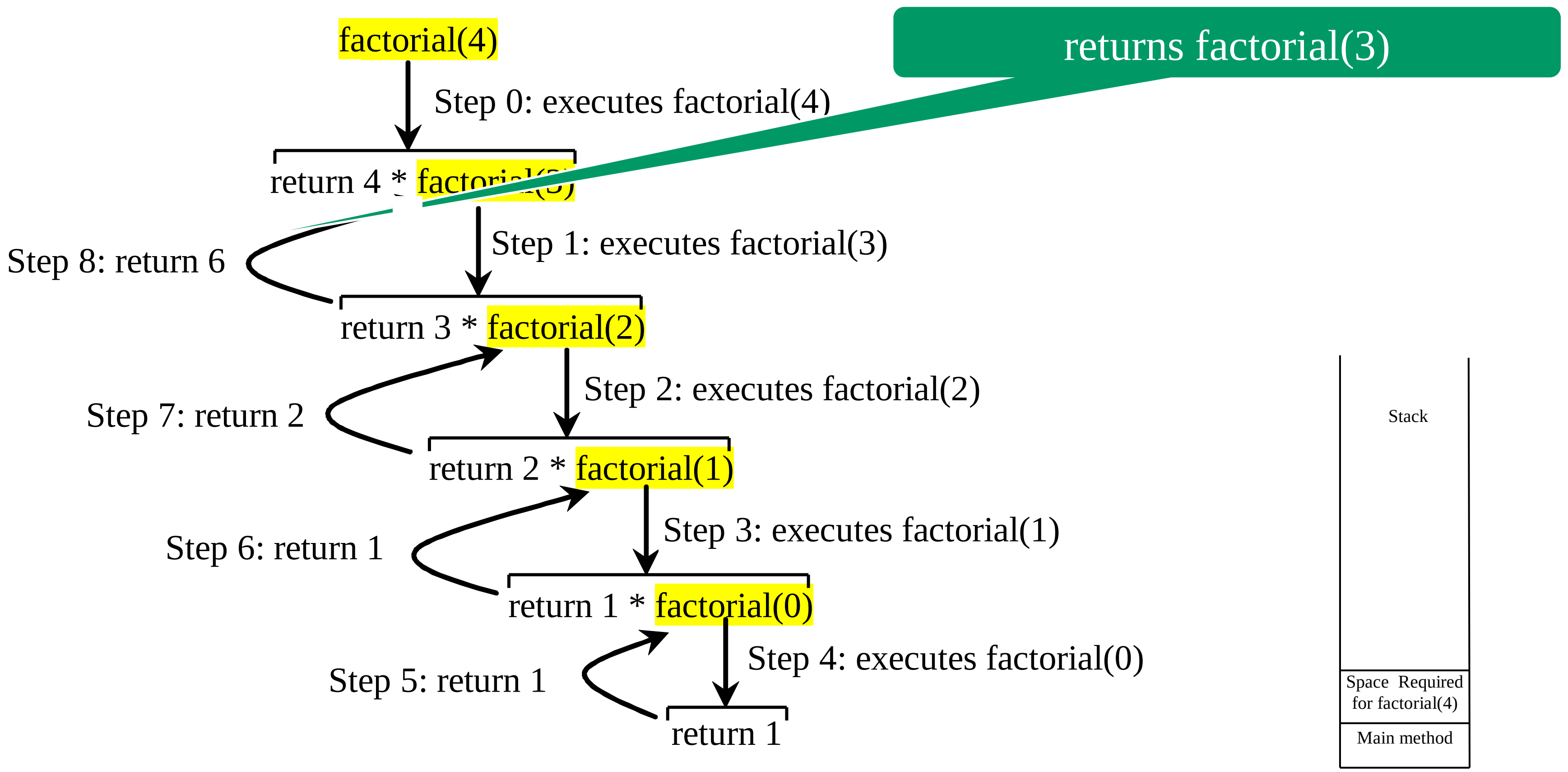

returns factorial(1)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: executes fac
return 3 * factorial(2)
ep 2: executes factorial(2)
return 2 * factorial(1)
Step 3: executes factorial(1)
Step 6: return 1
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1
Stack
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

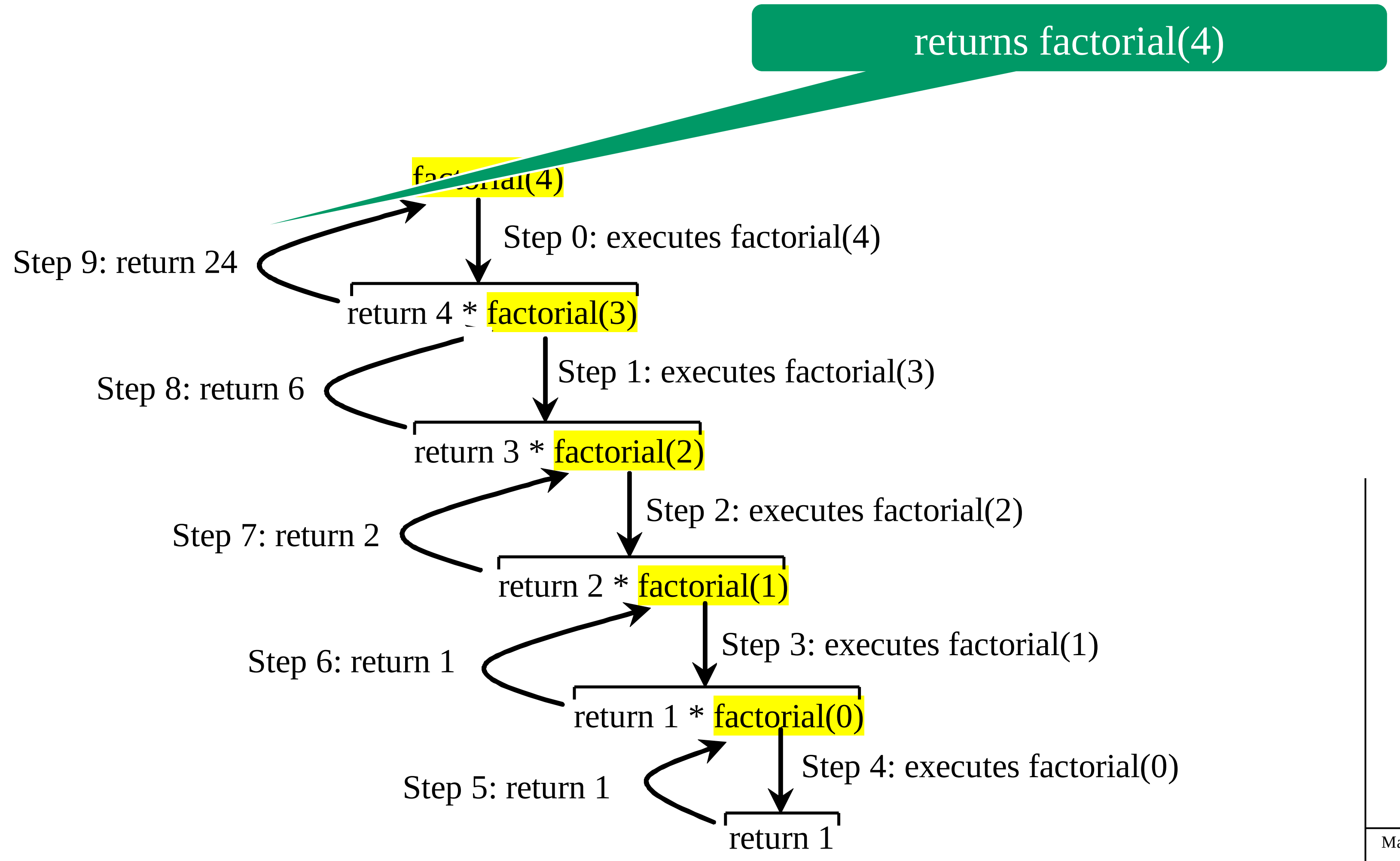


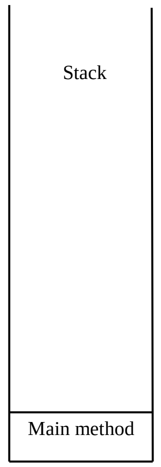
returns factorial(2)
factorial(4)
Step 0: executes factorial(4)
return 4 * factorial(3)
Step 1: ex es factorial(3)
return   factorial(2)
Step 2: executes factorial(2)
return 2 * factorial(1)
Step 3: executes factorial(1)
return 1 * factorial(0)
Step 4: executes factorial(0)
return 1
Step 5: return 1
Step 6: return 1
Step 7: return 2
Stack
Space Required for factorial(3)
Space Required for factorial(4)
Main method

Trace Recursive factorial

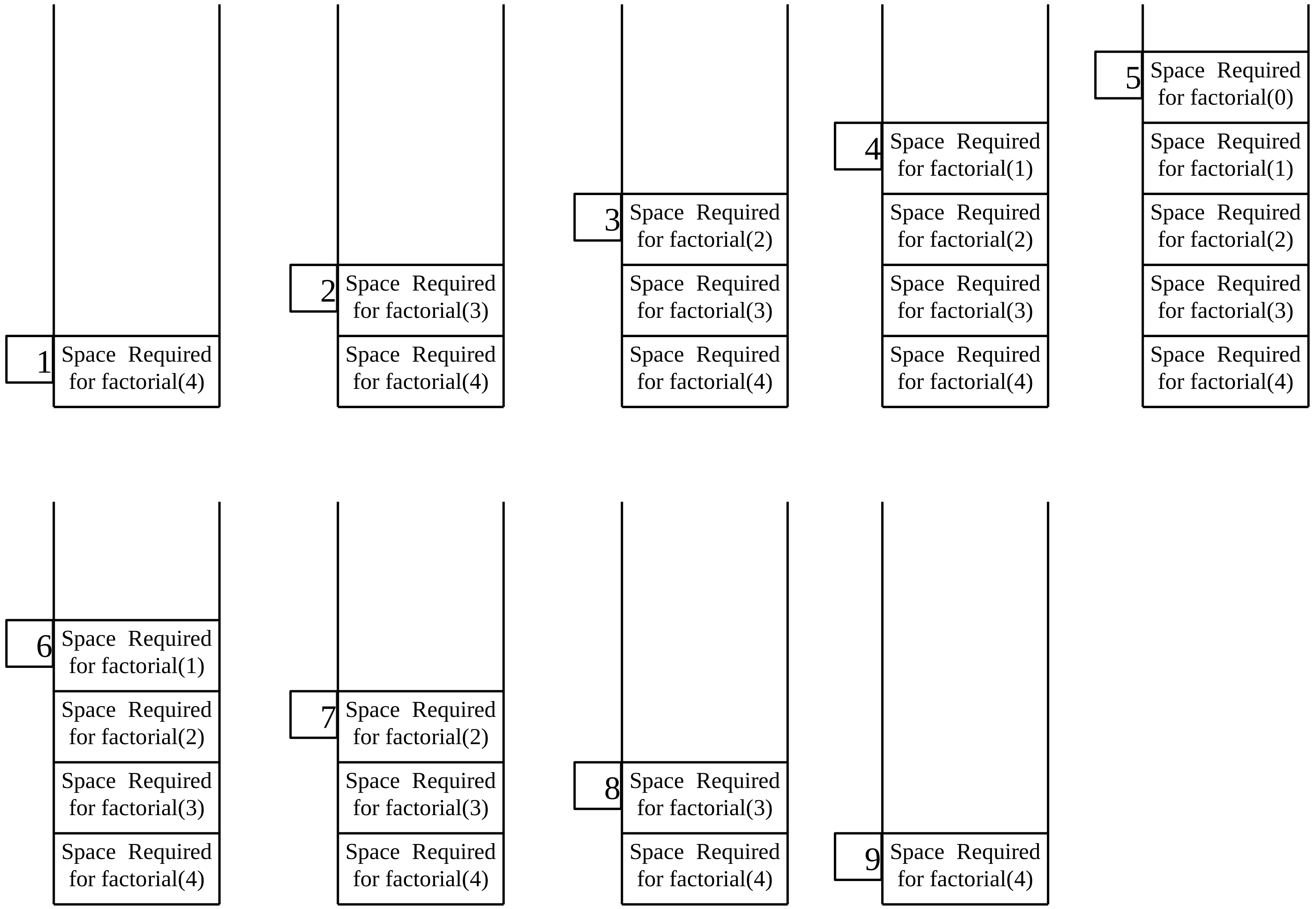


Trace Recursive factorial





factorial(4) Stack



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)

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.

Demo Program:
Non-Tail-Recursive: ComputeFactorial.java
Tail-Recursive: ComputeFactorialTailRecursion.java

Fibonacci Numbers

Lecture 1

Recursive Method

A method calling itself.

- A recursive method is defined as a method which calls itself.
- It is realized by call-stacks. **Call-stack** keep track of which call frame is currently active. Call-stack‘s push and pop operations works as the branching out new method call or returning from a branch-out method call.
- Stop condition will stop a method to continue to branch out and return certain value back to the calling frame.

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

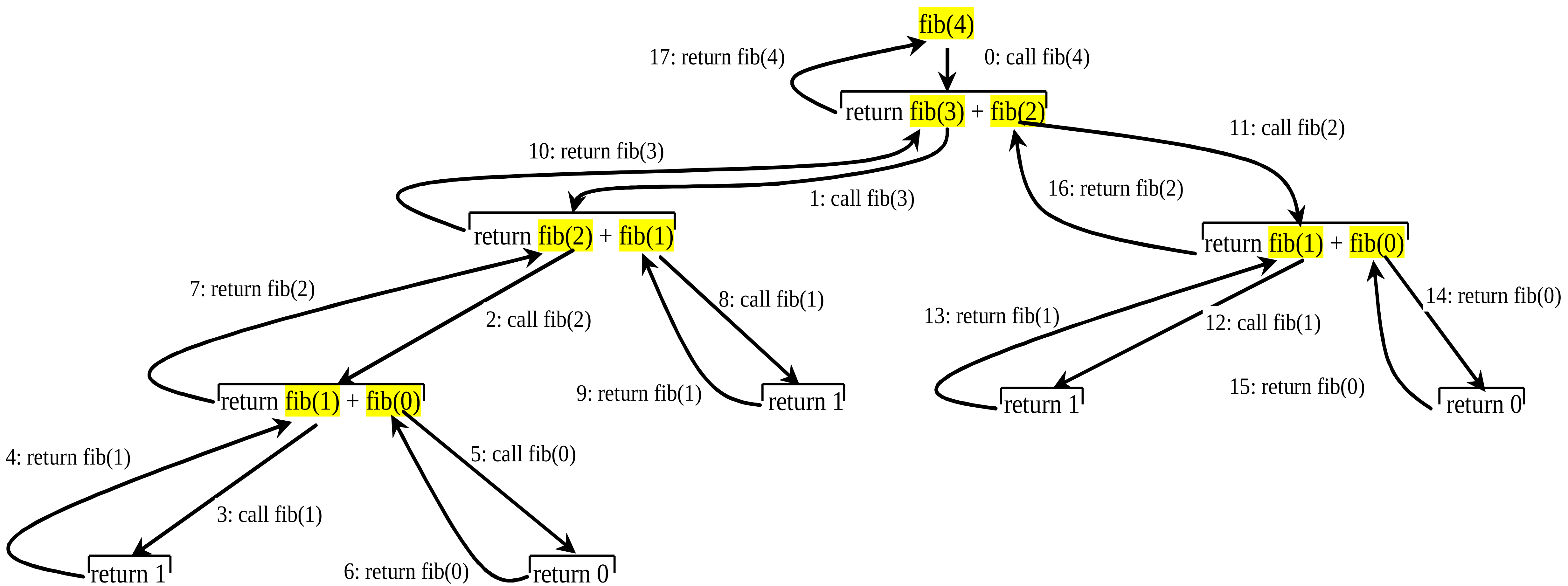
fib(0) = 0;

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

Fibonnaci Numbers, cont.



Fibonacci Method is a 2nd Order Linear Recurrence Equation

http://mathworld.wolfram.com/LinearRecurrenceEquation.html

- For recursive method which branch out two method call, it is called 2nd order recursive method.
- 2nd order recursive method may need 2 stop conditions.
- The time complexity grows in order of $\boldsymbol{O(2^n)}$

Demonstration Program

COMPUTEFIBONACCI.JAVA

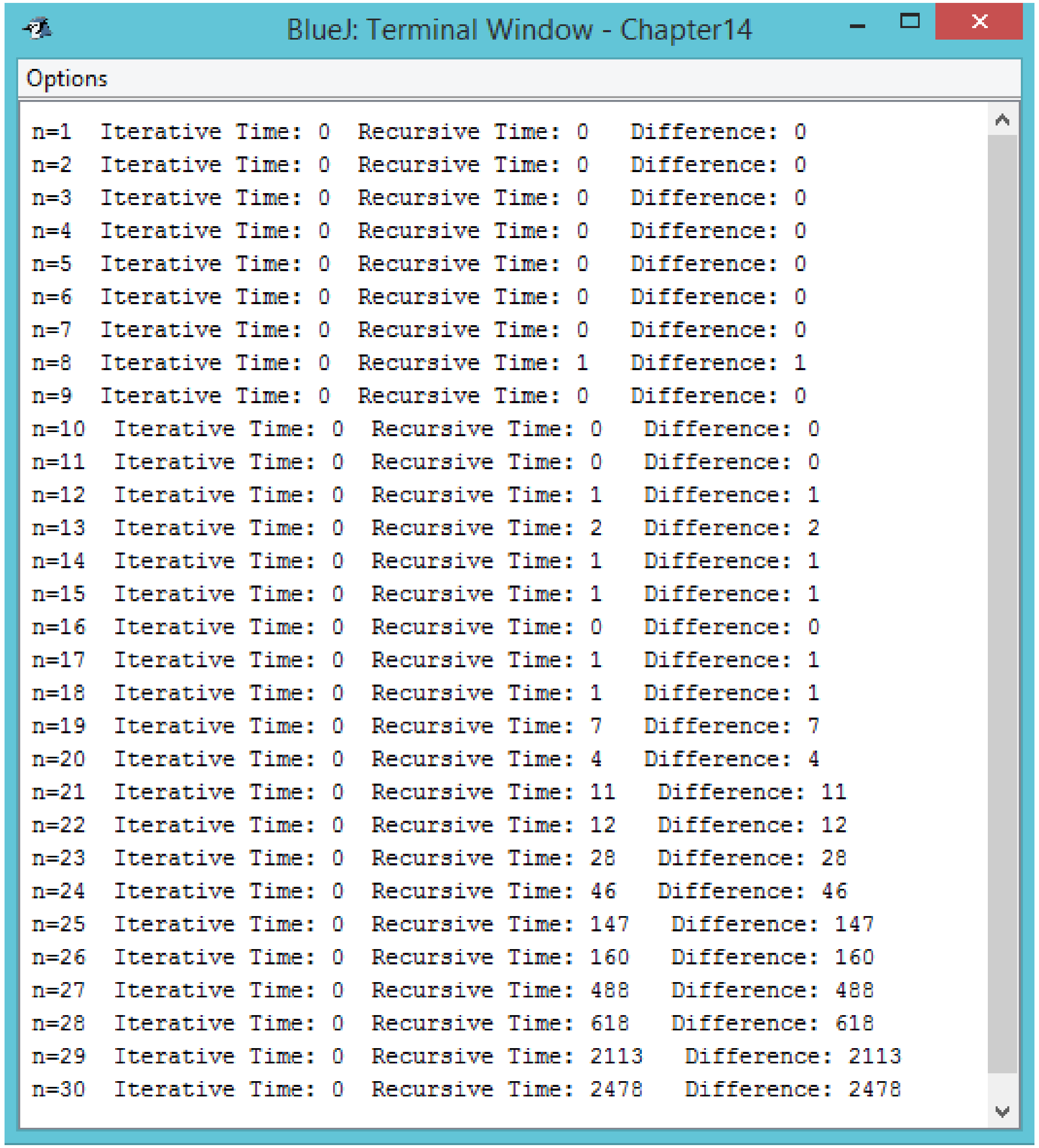
Fibonacci Number

```
int fibonacci(int n){
   int[] a = new int[n];
   a[0] = 1;
    a[1] = 1;
    for (int i = 2; i<=n; i++){
       a[i] = a[i-1] + a[i-2];
     }
    return a[n];
} // runs faster
```

```
int fibonacci(int n){
   if (n == 0 || n == 1) return 1;
   return fibonacci(n-1) +
          fibonacci(n-2);
} // shorter code
```

Demonstration Program

FIBONACCINUMBERS.JAVA



```
n=1  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=2  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=3  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=4  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=5  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=6  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=7  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=8  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=9  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=10  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=11  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=12  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=13  Iterative Time: 0  Recursive Time: 2   Difference: 2
n=14  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=15  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=16  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=17  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=18  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=19  Iterative Time: 0  Recursive Time: 7   Difference: 7
n=20  Iterative Time: 0  Recursive Time: 4   Difference: 4
n=21  Iterative Time: 0  Recursive Time: 11   Difference: 11
n=22  Iterative Time: 0  Recursive Time: 12   Difference: 12
n=23  Iterative Time: 0  Recursive Time: 28   Difference: 28
n=24  Iterative Time: 0  Recursive Time: 46   Difference: 46
n=25  Iterative Time: 0  Recursive Time: 147   Difference: 147
n=26  Iterative Time: 0  Recursive Time: 160   Difference: 160
n=27  Iterative Time: 0  Recursive Time: 488   Difference: 488
n=28  Iterative Time: 0  Recursive Time: 618   Difference: 618
n=29  Iterative Time: 0  Recursive Time: 2113   Difference: 2113
n=30  Iterative Time: 0  Recursive Time: 2478   Difference: 2478
```

- Recursion is good for solving the problems that are inherently recursive. (Especially 1st order)

Problem Solving using Recursion

Lecture 1

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:

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

Think Recursively

Many of the problems presented in the early chapters can be solved using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```
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));
}
```

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

Demonstration Program

PALINDROME.JAVA

PALINDROMETEST.JAVA

Dynamic Programming

Lecture 1

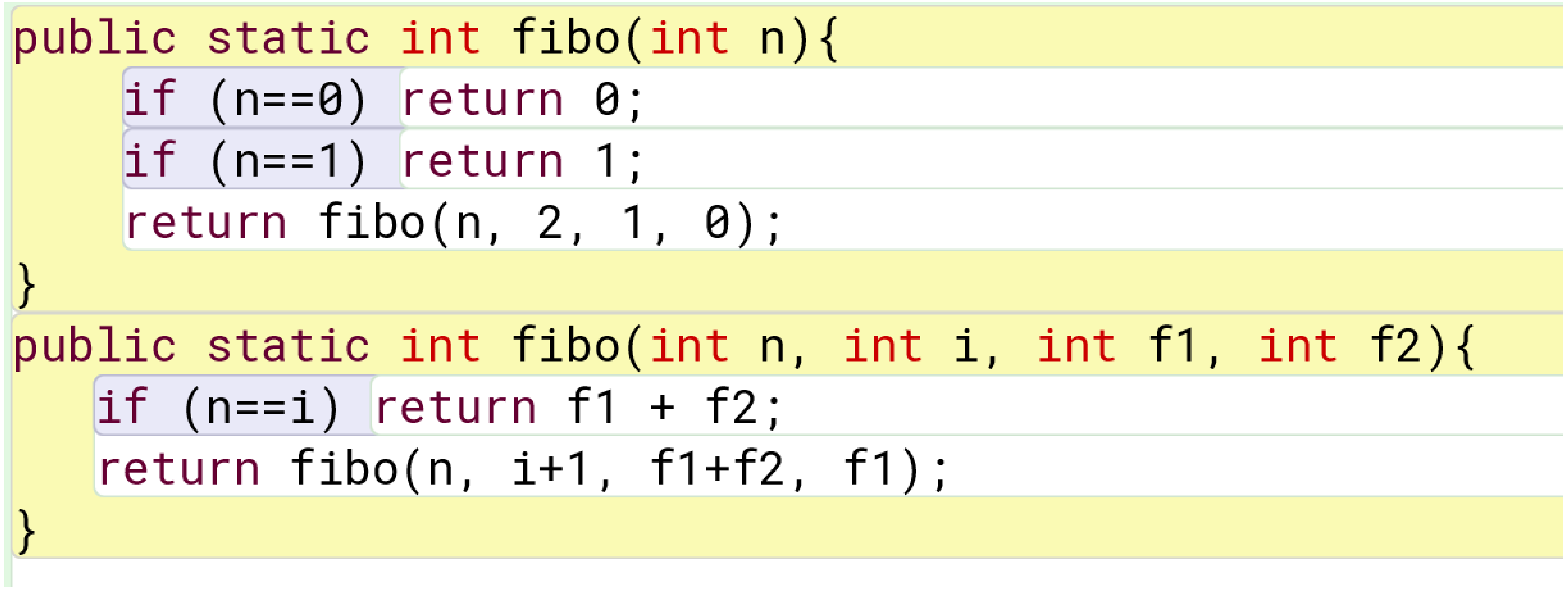
Avoiding Exponential Expansion

- If an algorithm has exponential expansion, it is actually not scalable.
- An algorithm of exponential growth can not be done with reasonable time when the problem size scale up.
- Avoiding exponential growth is the key issue for algorithm development.

Dynamic Programming

- If a recursive program can be done with a recursive result from smaller problem size, then we call this type of algorithms as Dynamic Programming.
- Dynamic Programming can be done using top-down approach or bottom-up approach

Dynamic Programming



```
public static int fibo(int n){
    if (n==0) return 0;
    if (n==1) return 1;
    return fibo(n, 2, 1, 0);
}
public static int fibo(int n, int i, int f1, int f2){
   if (n==i) return f1 + f2;
   return fibo(n, i+1, f1+f2, f1);
}
```

Recursive Processing

Lecture 1

Topics

- Problem Solving using Recursion
- Recursive Processing I: Statistics
- Recursive Processing II: Text Processing
- Recursive Processing III: Algorithms
- Recursion versus Iteration
- Tower of Hanoi (Chapter 19)

Algorithms in Recursive Processing I

1) Recursive Traversal
2) Maximum
3) Minimum
4) Sum
5) Average
6) Example Array:

```
static int[] a1 = {5, 3, 6, 7, 9, 24, 27, 1, 0 , 16};
```

(1) Recursive Traversal

```
public static void traverse(int[] a){
     traverse(a, 0);
}

public static void traverse(int[] a, int current){
     if (current == a.length) {
        System.out.println();
        return;
     }
     System.out.print("("+a[current]+") ");
     traverse(a, current+1);
}
```

(2) Maximum

```
public static int max(int[] a){
     return max(a, 0, Integer.MIN_VALUE);
 }

 public static int max(int[] a, int current, int result){
      if (current == a.length) return result;
      if (a[current] > result) result = a[current];
      return max(a, current+1, result);
 }
```

(3) Minimum

```
public static int min(int[] a){
    return min(a, 0, Integer.MAX_VALUE);
}

public static int min(int[] a, int current, int result){
     if (current == a.length) return result;
     if (a[current] < result) result = a[current];
     return min(a, current+1, result);
}
```

Sum and Average

```
public static int sum(int[] a){
   return sum(a, 0, 0);
}
public static int sum(int[] a, int current, int result){
     if (current == a.length) return result;
     result += a[current];
     return sum(a, current+1, result);
}
public static double avg(int[] a){
    return (double) sum(a)/a.length;
}
```

Demonstration Program

RECURSIVEPROCESSINGI_STATS.JAVA

Recursive Processing II

Lecture 6

Algorithms in Recursive Processing II

1)Palindrome Check (already discussed)
2)Reverse of String
3)Permutation of String

Reverse of String

```
public static String reverse(String a){
     if (a.length()==1) return a;
     return a.charAt(a.length()-1) +
                 reverse(a.substring(0, a.length()-1));
   }
```

Permutation of String

```
public static void permute(String a, ArrayList<String> alist){
      permute(a, "", alist);
   }
  public static String cut(String a, int i){
      return a.substring(0, i)+a.substring(i+1);
   }
  public static void permute(String a, String result, ArrayList<String> alist){
        if (a.length() == 1) { alist.add(a.charAt(0)+ result); return; }
        for (int i=0; i<a.length(); i++)
              permute(cut(a, i), a.charAt(i)+result, alist);
   }
```

Demonstration Program

RECURSIVEPROCESSINGII_TEXT.JAVA

Recursive Processing III

Lecture 7

Algorithms in Recursive Processing III

- (1) Linear Search
- (2) Recursive Selection Sort (Linear Traversal Version calling minIndex())
- (3) Recursive Binary Search
- (4) Recursive Selection Sort
- (5) Directory Size (Finding files and directory recursively.)

Recursive Linear Search

```
public static int search(int[] a, int key){
    return search(a, key, 0, -1);
 }

public static int search(int[] a, int key, int current, int result){
    if (current == a.length) return result;
    if (a[current] == key) return current;
    return search(a, key, current+1, result);
 }
```

Selection Sort with minIndex()

minIndex(): finding the element with minimum value

```
public static int minIndex(int[] a){
     return minIndex(a, 0, Integer.MAX_VALUE, -1);
 }

public static int minIndex(int[] a, int current, int result, int index){
      if (current == a.length) return index;
      if (a[current] < result) {
          result = a[current];
          index = current;
      }
      return minIndex(a, current+1, result, index);
 }
```

Selection Sort with minIndex()

sort(): using the concept of available list (MAX_VALUE unavailable elements)

```
public static void sort(int[] a){
     int[] c = new int[a.length]; int[] b = new int[a.length];
     for (int i=0; i<a.length; i++) c[i] = a[i];
     sort(c, b, 0);
     for (int i=0; i<a.length; i++) a[i] = b[i];
 }
public static void sort(int[] c, int[] b, int current){
     if(current == c.length) return;
     b[current] = c[minIndex(c)];
     c[minIndex(c)] = Integer.MAX_VALUE;
     sort(c, b, current+1);
 }
```

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.
- Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.

- Demo Program: RecursiveBinarySearch.java

Recursive Implementation

```
/** Use binary search to find the key in the list */
public static int recursiveBinarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  return recursiveBinarySearch(list, key, low, high);
}

/** Use binary search to find the key in the list between
    list[low] list[high] */
public static int recursiveBinarySearch(int[] list, int key,
  int low, int high) {
  if (low > high)  // The list has been exhausted without a match
    return -low - 1;

  int mid = (low + high) / 2;
  if (key < list[mid])
    return recursiveBinarySearch(list, key, low, mid - 1);
  else if (key == list[mid])
    return mid;
  else
    return recursiveBinarySearch(list, key, mid + 1, high);
}
```

Demonstration Program

RECURSIVEPROCESSINGIII_ALGORITHMS.JAVA

Recursive Selection Sort (by swap)

- Find the smallest number in the list and swaps it with the first number.
- Ignore the first number and sort the remaining smaller list recursively.

Demonstration Program

RECURSIVESELECTIONSORT.JAVA

Sort

```
public static void sort(double[] list) {
   sort(list, 0, list.length - 1); // Sort the entire list
 }

private 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);
   }
 }
```

Recursion vs. Iteration

Lecture 8

Recursion vs. Iteration

- Recursion is an alternative form of program control. It is essentially repetition without a loop.

- Recursion bears substantial overhead. Each time the program calls a method, the system must assign space for all of the method’s local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.

	Iteration	Recursion
Control	Loop Statement	Recursive Call
Local Variables	Required	Not Required
Assignments	Required	Not Required
Style	Imperative	Declarative
Size	Larger	Smaller
Nontermination	Infinite Loop	Infinite Recursion