UBC OKANAGAN

COSC 121
Computer Programming II

Recursion

Part 1/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Previous Lecture

- Binary I/O
  - `DataInputStream` / `DataOutputStream`
  - `ObjectInputStream` / `ObjectOutputStream`
    - `Serializable, transient.`
  - `BufferedInputStream / BufferedOutputStream`

Outline

Today:

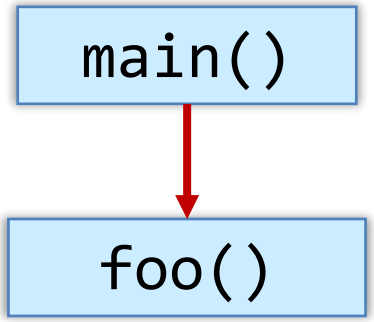
- Introduction to Recursion
- Think Reclusively
- Examples: Simple Recursive Methods

Next lecture:

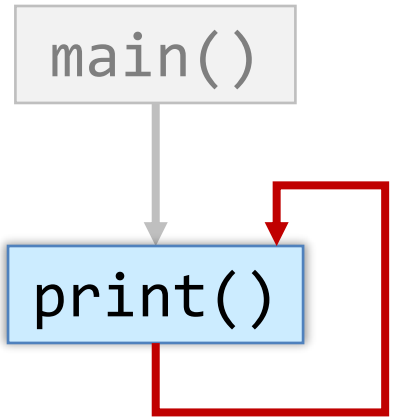
- How Recursion Really Works
- Recursive Helper Methods
- More example problems
  - Ones that have a nice recursive solution
  - Ones that really need recursion
- Tail-Recursive Method
- Recursion vs. Iteration

Introduction to Recursion

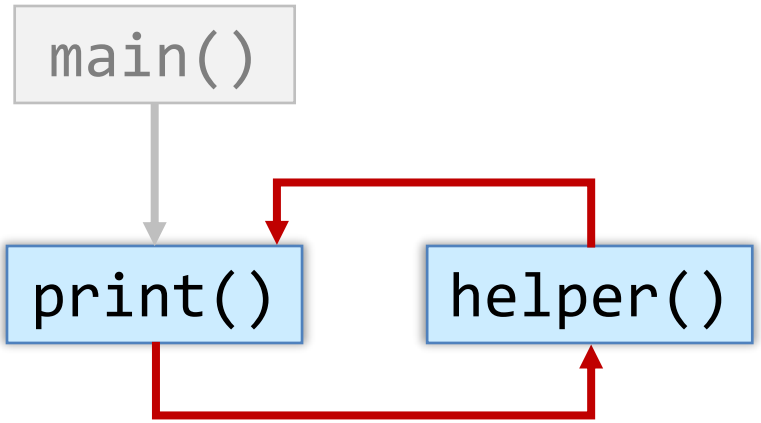
Previously, you have seen methods that call other methods



A ***recursive method*** is a method that ***calls itself…***



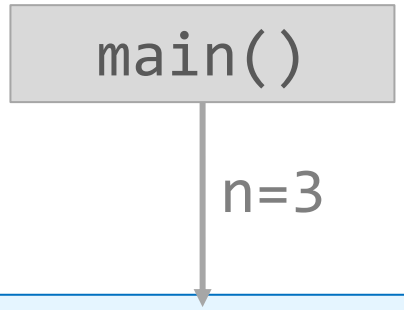
directly



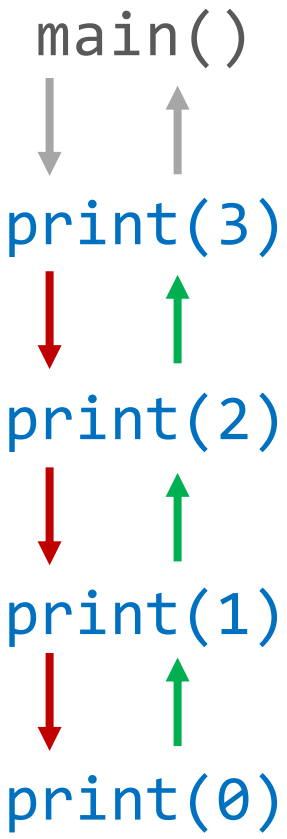
or indirectly

Example 1

This recursive method prints the numbers from n to 1




```
void print(int n){
   if(n > 0) {
      System.out.print(n+" ");
      print(n-1);
   }
   return;
}
```



Output

3 2 1

Example 1, *cont'd*

Can you think of non-recursive way of writing the **`print`** method?

```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

```
void print(int n){
   for(int i = n; i > 0; i--)
      System.out.println(i);
}
```

Output

3 2 1

Observations:

- Recursive code can be rewritten using loops.
- Both include a condition to end the recursive call or loop

Stopping Condition

Similar to loops, recursive methods need a ***"stopping condition"*** that determines when to stop the 'repetition'.

```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

```
void print(int n){
   for(int i = n; i > 0; i--)
      System.out.println(i);
}
```

So, why recursion?

Recursion can be used to solve problems that are **difficult to program using simple loops**.

- That is, recursion provides an easier, more intuitive ways to solve these problems compared to loops.

For example:

- Traversing through file system.
- Traversing through a tree of search results.

Computer science algorithms courses have many examples similar to above two.

Today

- we are going to practice on basic recursive methods that can be easily solved by either recursion or loops.

Next lecture

- problems that are much easier to solve with recursion.

Remember

Recursion is a method that **calls itself**

Recursive code can be **re-coded with loops**, but sometimes it is easier to use recursion instead of loops.

Recursive methods includes:

- a **stopping condition** (just like the condition that we use to control loops)
- A **recursive call** that involves a **sub-problem** resembling the original problem
  - *more about this next*

Next, let's take a step back and try to "think recursively"!!

Think Recursively!!

What is a Recursive Problem?

*Whenever you are able to break a problem to sub-problems (smaller) that **resembles the original problem**.*

- This means you can apply the same approach to solve the subproblem recursively.

Example: for print(3), instead of "thinking" of *printing **THREE*** numbers, let's just take one step and print 3 then *print the **remaining TWO*** numbers (a smaller problem that resembles the original one).

```
void print(int n){
   if(n>0) {
      System.out.println(n);
      print(n-1);
   }
}
```

Many of the problems presented in the early chapters can be solved using recursion if you think recursively.

Think Recursively

Drinking coffee problem

- With recursion :

```
1) Drink one sip
2) Drink a smaller cup
   (until there is no coffee)
```

- With loops:

```
While (cup is not empty){
      Drink one sip
}
```

Let's apply this to the `print()` method…

Thinking recursively about print(5) :

- **(print 1 to 5)** can be broken down to

  1) **(print 5)**
  2) THEN **(print 1 to 4)**, which is a simpler problem that resembles (print 1 to 5).

- Remember that we need a stopping condition:
  - (Until n = 1)

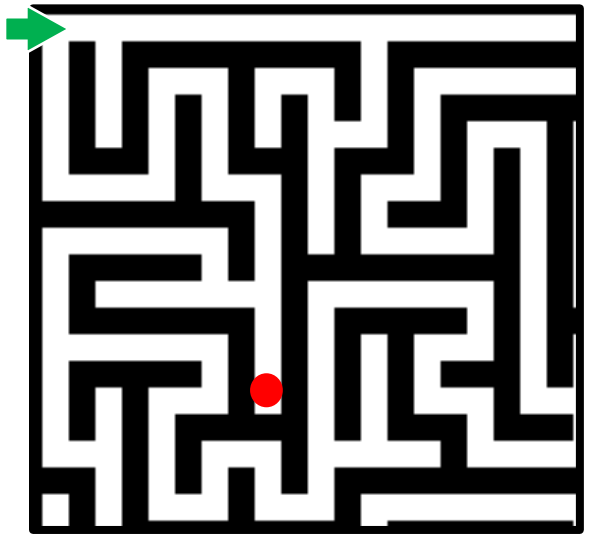
Other problems to Think Recursively

Searching a maze for an item

- With recursion :

  1) move one step in each direction from where you are standing.
  2) if target is not there, search a smaller maze

  Stopping condition: target found



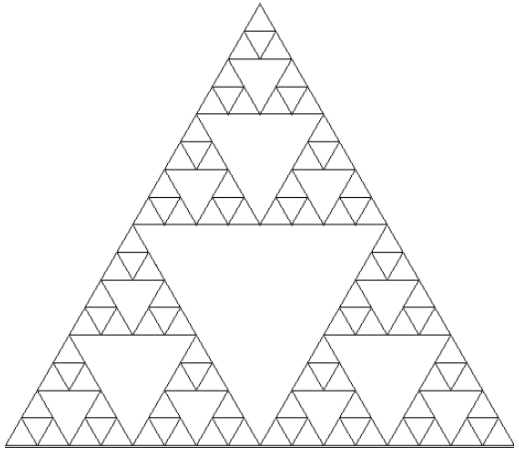
- With loops:

  ???

Think Recursively

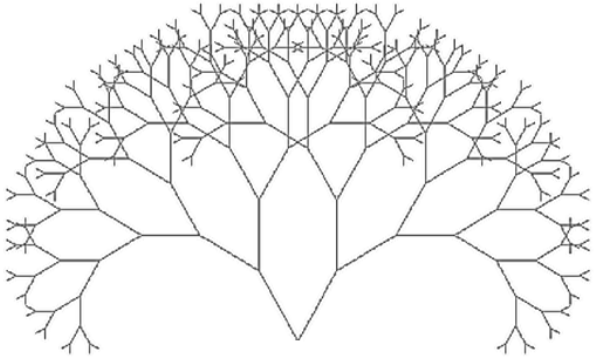
Fractals

- A fractal is a geometrical figure that can be divided into parts, each of which is a reduced-size copy of the whole.
- Example: let's say we want to draw a fractal tree

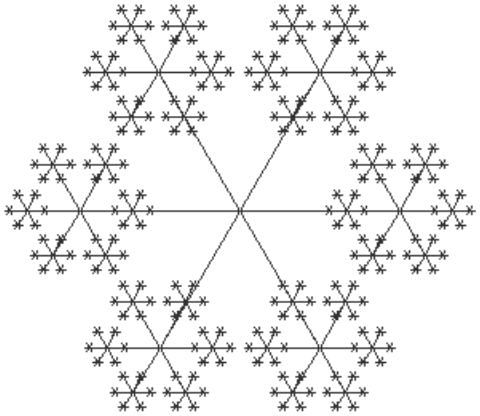


Sierpinski triangle

named after a famous Polish mathematician



Fractal tree

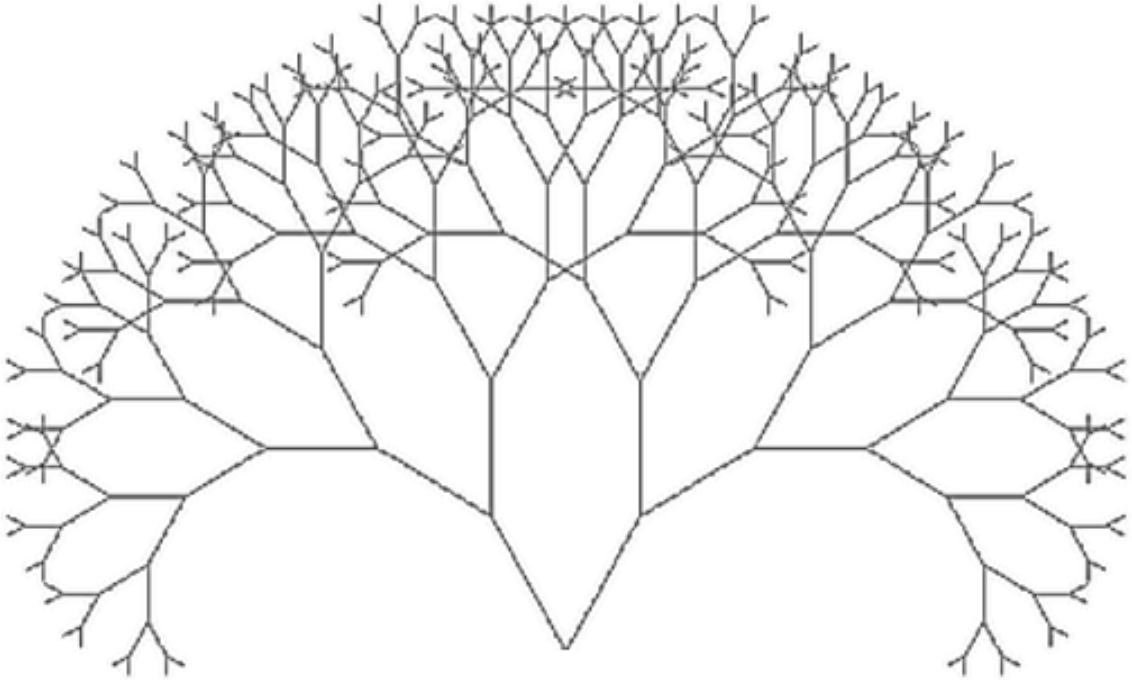


Fractal snowflake

Think Recursively

Drawing a tree problem

Can you identify the subproblem in the following draw-tree-problem?



Problem:

draw TREE = draw  + draw a smaller TREE at each end.

Think Recursively

Sierpinski Triangle

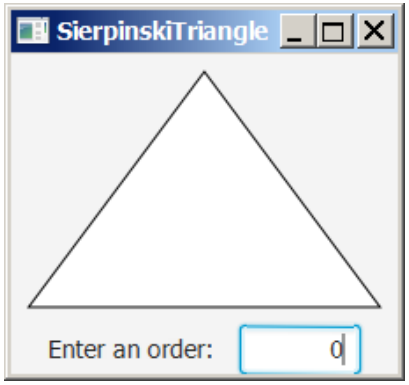
1. Start with an equilateral triangle, which is considered to be the fractal of order (or level) 0.

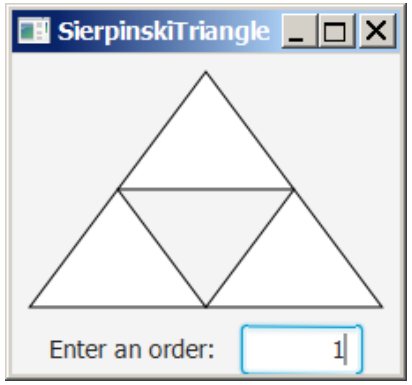
Repeat recursively

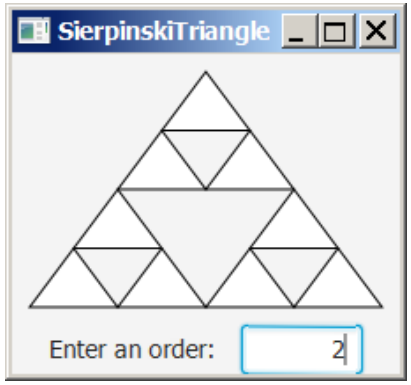
2. Connect the midpoints of the sides of the triangle of order 0 to create a Sierpinski triangle of order 1.
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski of order 2.

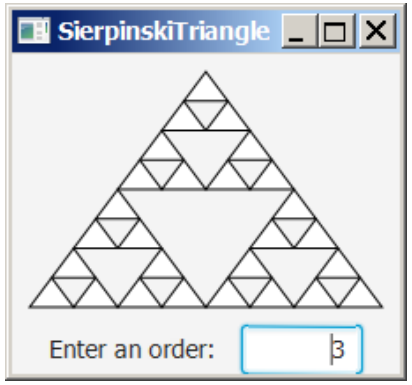
Stopping condition

4. You can repeat the same process recursively till you reach the desired order.



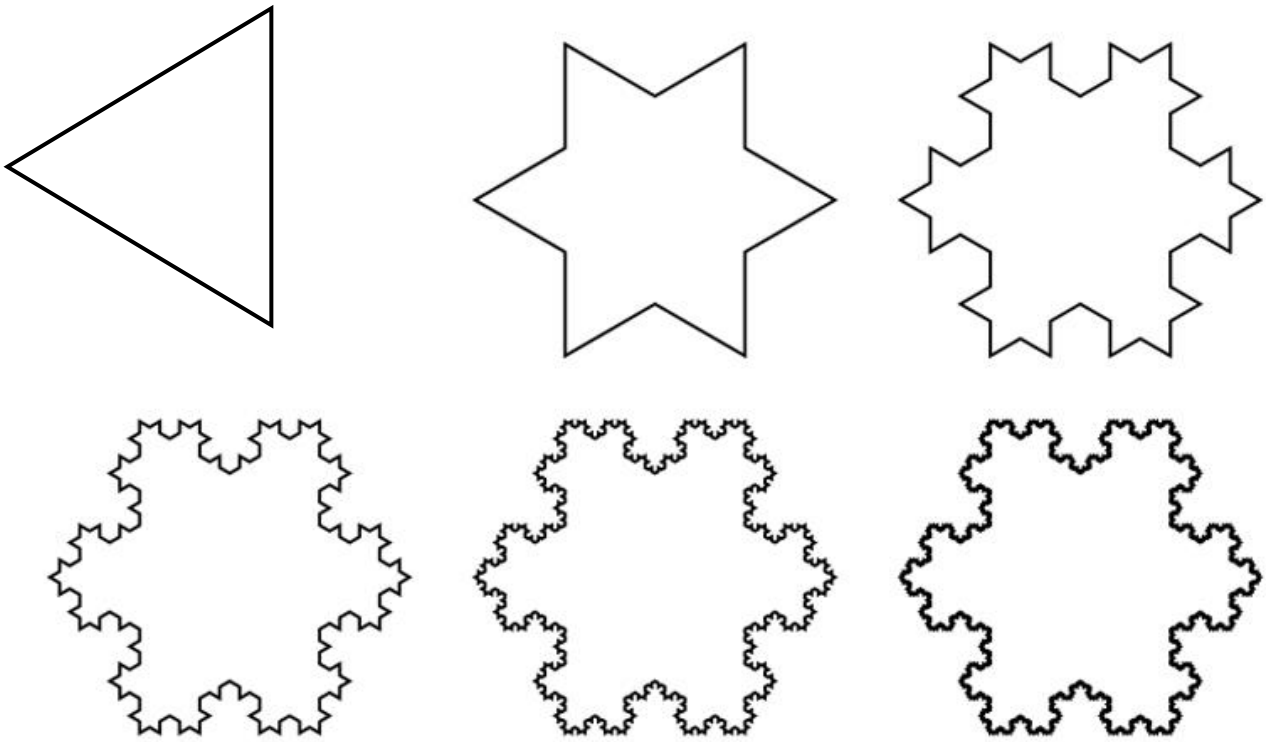






Think Recursively

Fractal Art



Can you give a recursive description of the problem?

i.e. what to repeat? And when to stop?

Back again to coding…

Example 2: Factorial

Factorial Definition:

- n! = n . (n - 1)! , when n > 0
- 0! = 1

Write a 'recursive' program to computer n!

How?

Thinking recursively: computing n! includes computing (n-1)! which resembles n! except that it is smaller

Steps:

- **1. Identify the recursive call** (i.e. **the simpler form**)
  - **(n - 1)!** is a simpler form of n!
    - **n!** = n . **(n - 1)!**
- **2. Identify stopping condition(s)** by identifying **base case(s)**
  - The base case (simplest form) is **0! = 1**.
- **3.** write the **recursive method**

Example 2: Factorial, *cont'd*

The code:

```
public class Factorial {
    public static void main(String[] args) {
        int f = factorial(4);
        System.out.println(f);
    }

    // Recursive method
    public static int factorial(int n){
        if(n == 0)                          //stopping condition
            return 1;
        else
            return n * factorial(n-1);  //recursive call
    }
}
```

Output: 24

Example 2: Factorial: *How did that happen?*

factorial(4) = 4 * factorial(3)

factorial(3) = 3 * factorial(2)

= 4 * (3 * factorial(2))

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

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

factorial(0) is the base case

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

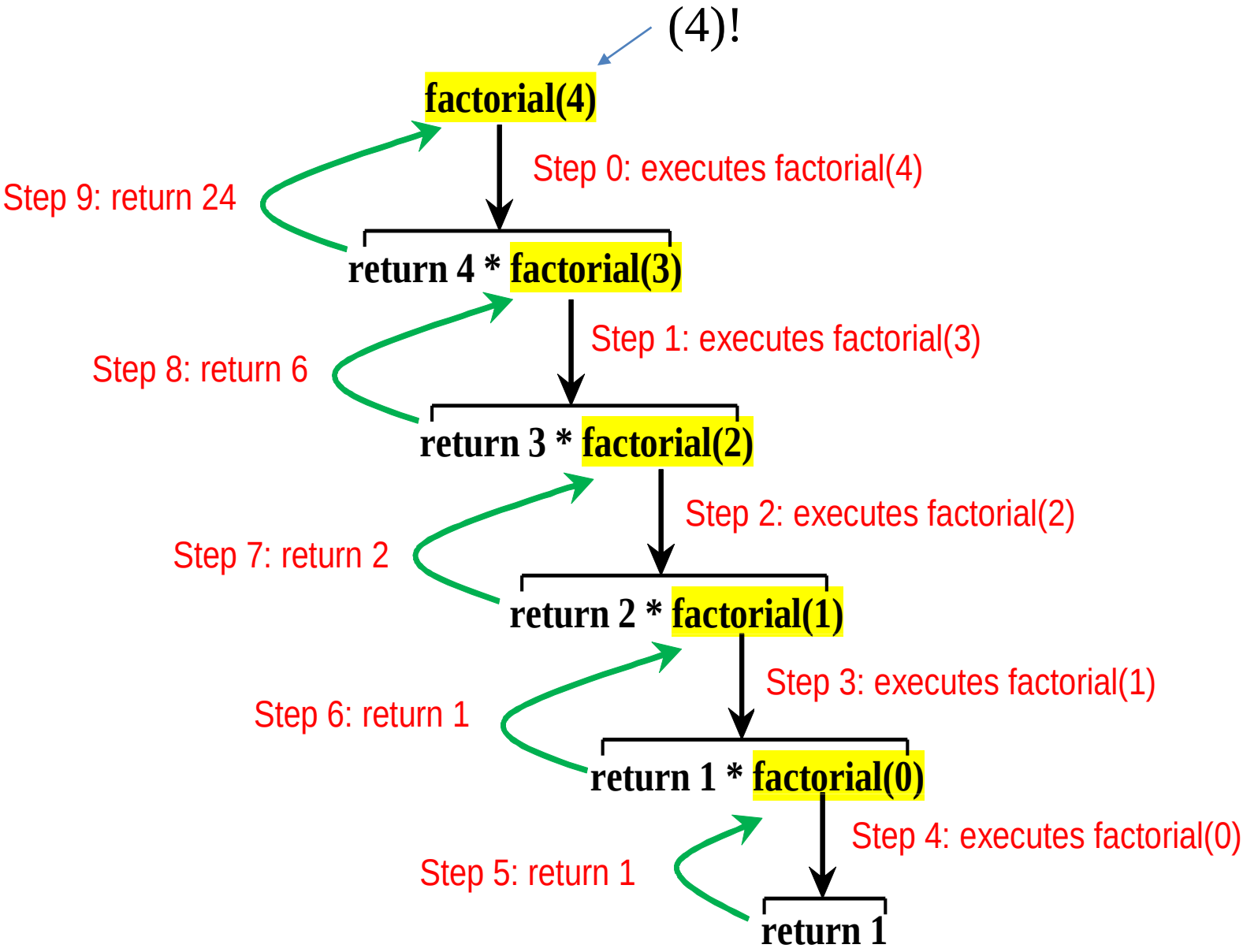
= 4 * (3 * (2 * **1**))

= 4 * (3 * **2**)

= 4 * **6**

= **24**

Example 2: Factorial, *Memory Visualization*



Example 2: Factorial using loops

Recursion might not be your best option to implement the factorial function. Loops are ***simpler*** and ***more efficient***.

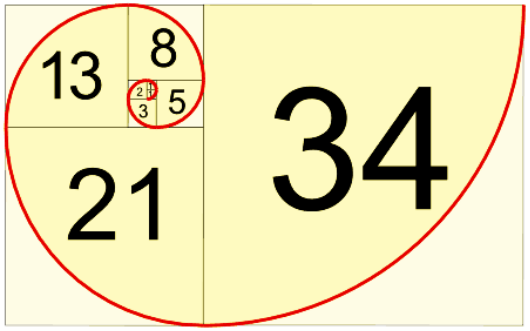
```
public class FactorialLoop {
    public static void main(String[] args) {
        int f = factorial(4);
        System.out.println(f);
    }

    // NON-recursive method
        public static int factorial(int n){
            int result = 1;
            for (int i = n; i >= 1; i--)
                result *= i;
            return result;
        }
}
```

You will see later some problems that are difficult to solve without using recursion.

Example 3: 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...



The Fibonacci series begins with **0** and **1**, and each subsequent number is **the sum of the preceding two**.

Stopping conditions (Base cases)

- fib(0) = 0;
- fib(1) = 1;

Recursive call

- fib(index) = fib(index -1) + fib(index -2); index >=2

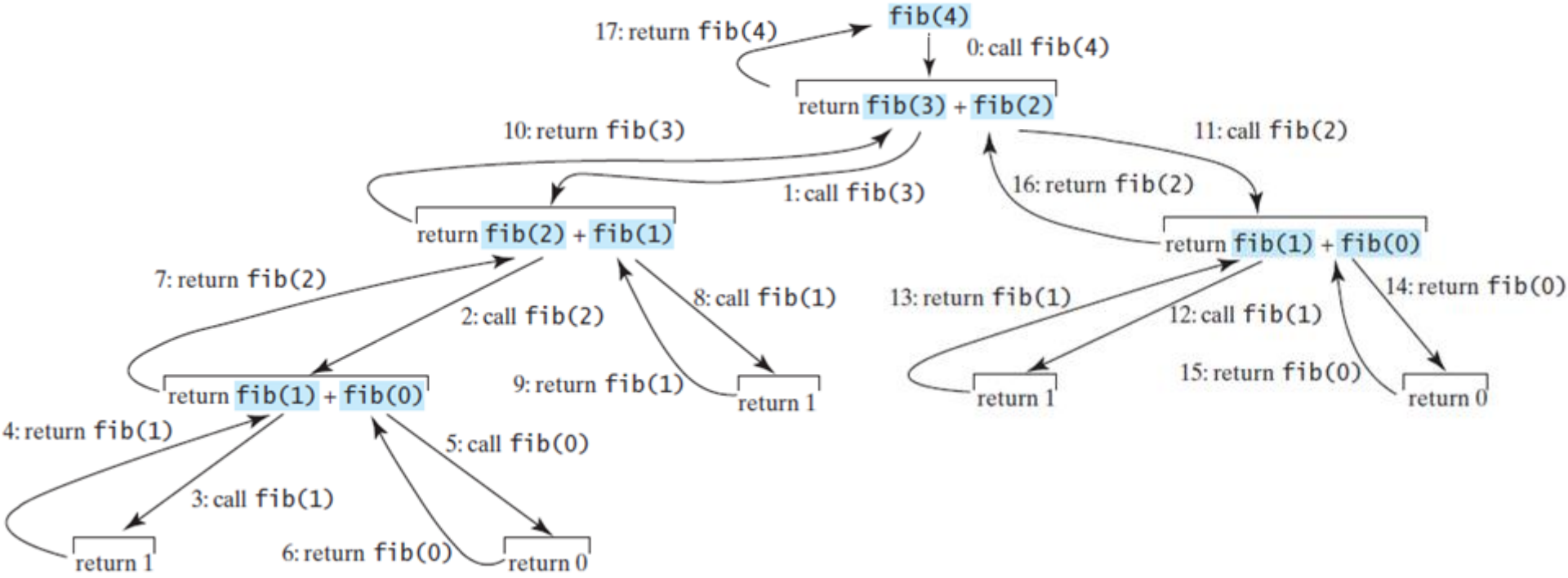
Example 3: Fibonacci Numbers, *cont'd*

```
public class ComputeFibonacci {
    public static void main(String[] args) {
        System.out.println("Fibonacci number at 4 is " + fib(4));
    }

    //recursive Fibonacci method
    public static long fib(long index) {
        if (index == 0)          //Stopping condition 1 (Base case 1)
            return 0;
        else if (index == 1)     //Stopping condition 2 (Base case 2)
            return 1;
        else                     // Reduction and recursive calls
            return fib(index - 1) + fib(index - 2);
    }
}
```

Output: 3

Example 4: Fibonacci Numbers, *cont'd*

How did that happen?



Example 3 Fibonacci Numbers using loops

Recursion is not efficient, since it requires more time and memory to run the recursive call. BUT the recursive implementation of the **fib** method is very simple and straightforward. **Lets compare it to loops-based implementation**

```
public class ComputeFibonacciLoop {
    public static void main(String[] args) {
        System.out.println("Fibonacci number at 4 is " + fib(4));
    }

    // NON-recursive Fibonacci method
    public static long fib(long index) {
        if (index == 0) return 0;
        if (index == 1) return 1;

        int f0 = 0, f1 = 1, currentFib = 1;
        for (int i = 2; i <= index; i++) {
            currentFib = f0 + f1;
            f0 = f1;
            f1 = currentFib;
        }
        return currentFib;
    }
}
```

Summary

What is recursive method?

- A method that **calls itself**

Recursive methods must have:

1. A ***recursive call*** (that calls the same method again)
   - Every recursive call reduces the original problem, bringing it closer to a base case until it becomes that case.
2. One or more ***base cases*** used to stop recursion.
   - A **stopping condition** tells the method to stop calling itself when the problem becomes in its **simplest form** (i.e. the "**base case**").
   - The method uses conditionals (e.g. if-else, switch) to run different cases

When to use recursion?

- When solving problems that have recursive structures.
  - Coding these problems recursively is simpler and more intuitive.
- Any problem that can be solved recursively can be solved nonrecursively (with iterations) but recursion may provides simpler way to solve the problem.

More practice…

Practice 1

Identify base cases and recursive calls.

What does the program do?

What is the output?

```
public class Practice1 {
    public static void main(String[] args) {
        System.out.println(m1(5));
    }

    public static int m1(int n) {
        if (n == 1)
            return 1;
        else
            return n + m1(n - 1);
    }
}
```

Practice 2

Write a ‘recursive’ program that prints the numbers from 5 to 1

- We need to identify:
  - **base case(s)**
  - **smaller problem that resembles the original problem**

```
public class printNumbers {
    public static void main(String[] args) {
        print(5);
    }

    public static void print(int n) {
        if (n > 0) {
            System.out.println(n);
            print(n - 1); // recursive call. n-1 is simpler than n
        } //stopping condition is n == 0
    }
}
```

Output

```
5
4
3
2
1
```

Practice 3

Write code to print a message for n times (using recursion)

- The problem can be broken to (**base case) + (smaller problem)**
  - Based case (stopping condition): n == 0
  - Recursive call: nPrintln(“abc”, n-1)

```
public class printMsg {
    public static void main(String[] args) {
        nPrintln("abc", 5);
    }

    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
    }
}
```

Q: Will this work if we don’t pass message again?

Practice 4

Write a *recursive* method isPalindrome that checks whether a word is palindrome or not (i.e., can be read from both directions)

- Problem can be divided into (**base cases) + (smaller problem)**
  - If length <=1, then **stop**.
  - Check the first and last characters of the string, if they are not equal **stop**.
  - Ignore the two end characters and check whether the rest of the substring is a palindrome (***recursive call***)

```
public static void main(String[] args) {
    System.out.println(isPalindrome("x"));       //true
    System.out.println(isPalindrome("racecar")); //true
    System.out.println(isPalindrome("ubc"));     //false
}
public static boolean isPalindrome(String s){
    if(s.length()<=1)                            //1)base case 1
        return true;
    else if(s.charAt(0)!=s.charAt(s.length()-1)) //2)base case 2
        return false;
    else                                         //recursive call
        return isPalindrome(s.substring(1,s.length()-1));
}
```