



# CSE 247

## Data Structures

# Recursion

- **Recursion** in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem.
- Recursion is a method of defining functions in which the function being defined is applied within its own definition (or express in term of themselves).

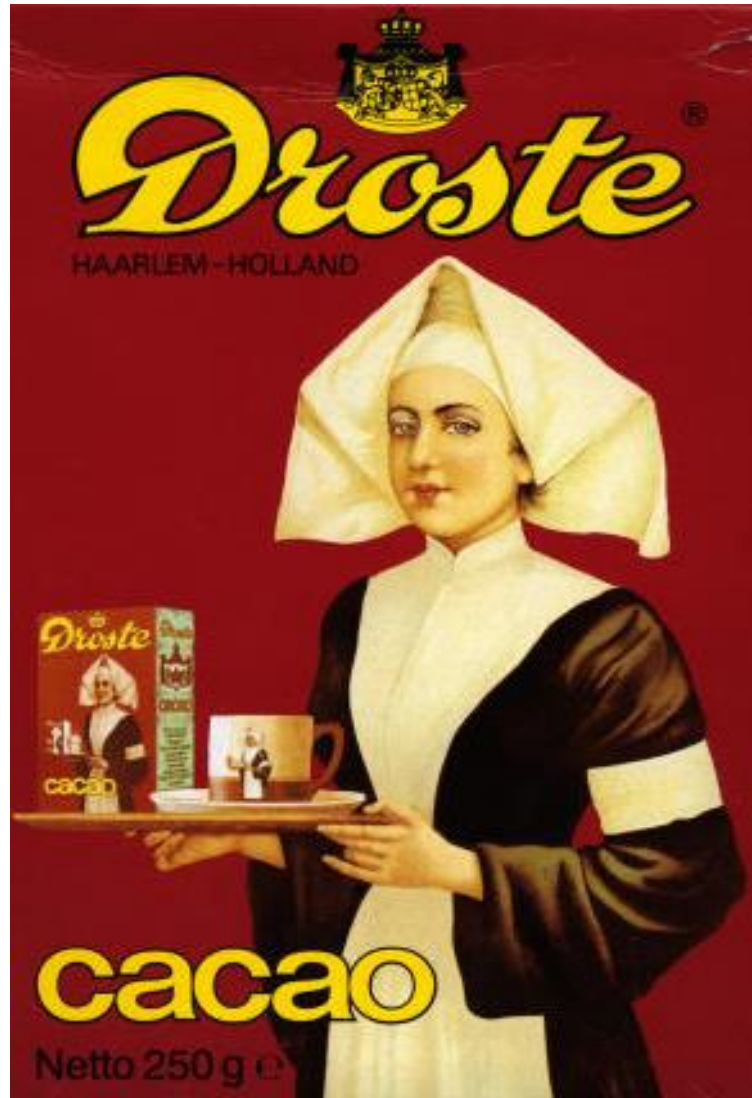
For example:  $n! = n \times (n-1)!$

# Defined/expressed in terms of themselves



<http://id.mind.net/~zona/mmts/geometrySection/fractals/tree/treeFractal.html>

Defined/expressed in terms of  
themselves



Defined/expressed in terms of themselves



# Why recursion?

- Recursion can solve some kinds of problems better than iteration(loops).
- Recursion leads to elegant, simplistic, short code (when used well).
- Many programming languages use recursion exclusively (no loops). Such as functional languages includes Scheme, ML, ...
- A different way of thinking problems.

# Why recursion

- One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

# Why use Recursive Methods?

- In computer science, some problems are more easily solved by using recursive methods.

For example:

- Traversing through a directory or file system.
- Traversing through a tree of search results.
- Some sorting algorithms recursively sort data

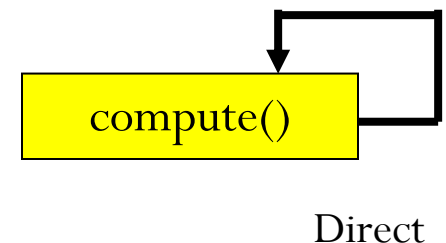
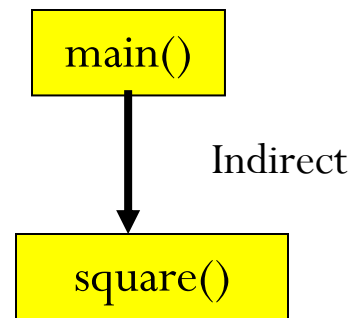


# How to think to code recursive solution

- When writing a recursive code:
  - The base case is easy .....”when to stop”
  - In recursive call:
    - How can we break the problem down into two:
      - A piece I can handle right now
      - The answer from a smaller piece of the problem
    - Assuming your self-call does the right thing on a smaller pieces of the problem.
    - How to combine parts to get the overall answer.
- Practice will make it easier
- To see ideas, do exercises.

# Recursion

- Recursion is one technique for representing data whose exact size the programmer does not know: the programmer can specify this data with a self-referential definition such as inductive definition.
- Types of recursion
  - Indirect: function call other function
  - Direct: function call itself



# Recursion

- Tail recursive method: recursive method in which the last statement executed is the recursive call
- Infinite recursion: case where every recursive call results in another recursive call

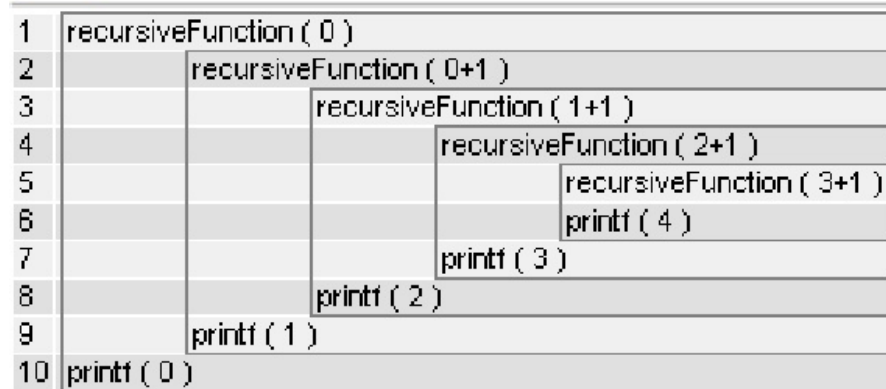
# First two rules of recursion

- **Base case**: You must always have some base case which can be solved without recursion
- **Making Progress**: For cases that are to be solved recursively, *the recursive call* must always be a case that *makes progress toward the base case*.

# Recursion

LIFO behavior in function calls.

Figure shows, the function called at last return first from that function.




# World's Simplest Recursion Program

```
public class Recursion
{
    public static void main (String args[])
    {
        count(0);
        System.out.println();
    }

    public static void count (int index)
    {
        System.out.print(index);
        if (index < 2)
            count(index+1);
    }
}
```

**This program simply counts from 0-2:  
and prints: 012**

**This is where the recursion occurs.  
You can see that the `count()` method  
calls itself.**



# Visualizing Recursion

- To understand how recursion works, it helps to visualize what's going on.
- To help visualize, we will use a common concept called the *Stack*.
- A stack basically operates like a container of trays in a cafeteria. It has only two operations:
  - **Push**: you can push something onto the stack.
  - **Pop**: you can pop something off the top of the stack.

# Stacks and Recursion

- Each time a method is called, you *push* the method on the stack.
- Each time the method returns or exits, you *pop* the method off the stack.
- If a method calls itself *recursively*, you just **push another copy of the method onto the stack**.
- We therefore have a simple way to visualize how recursion really works.



# Factorials

- Computing factorials are a classic problem for examining recursion.

- A factorial is defined as follows:

$$n! = n * (n-1) * (n-2) \dots * 1;$$

- For example:

$$1! = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3 * 2 * 1 = 24$$

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

If you study this table closely, you will start to see a pattern.

# Seeing the Pattern

- Seeing the pattern in the factorial example is difficult at first.
- But, once you see the pattern, you can apply this pattern to create a recursive solution to the problem.
- Divide a problem up into:
  - What we know (call this the base case)
  - Making progress towards the base
    - Each step resembles original problem
    - The method launches a new copy of itself (recursion step) to make the progress.

# Factorials

- Computing factorials are a classic problem for examining recursion.

- A factorial is defined as follows:

$$n! = n * (n-1) * (n-2) \dots * 1;$$

- For example:

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 \text{ (Base Case)}$$

If you study this table closely, you will start to see a pattern.

The pattern is as follows:

You can compute the factorial of any number (n) by taking n and multiplying it by the factorial of (n-1).

For example:

$$5! = 5 * 4!$$

(which translates to  $5! = 5 * 24 = 120$ )

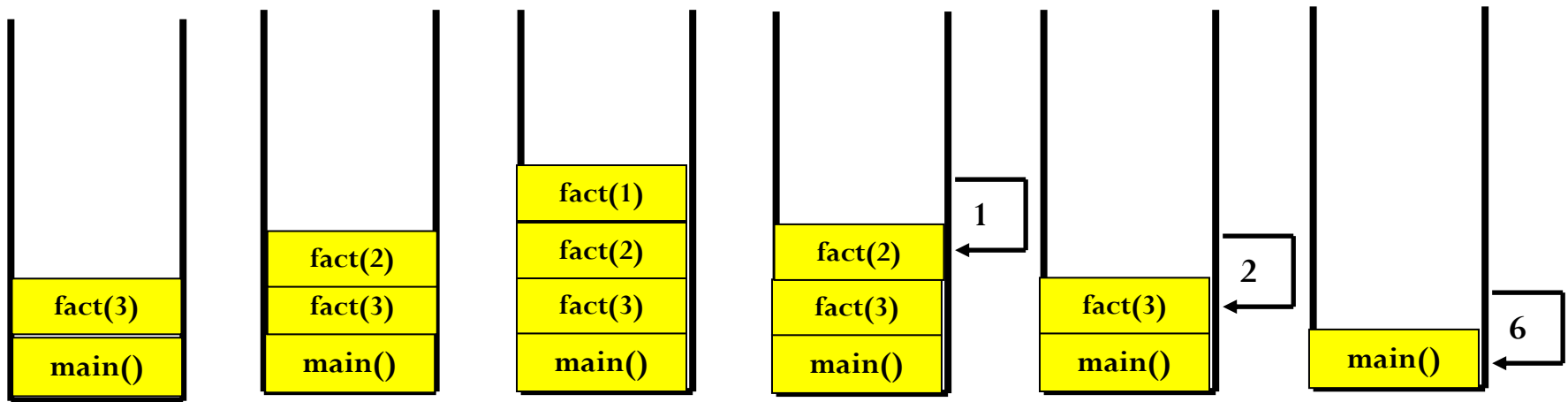
# Recursive Solution

```
public class FindFactorialRecursive
{
    public static void main (String args[])
    {
        for (int i = 1; i < 10; i++)
            System.out.println ( i + "! = " + findFactorial(i));
    }

    public static int findFactorial (int number)
    {
        if ( (number == 1) || (number == 0) )
            return 1; ← Base Case.
        else
            return (number * findFactorial (number-1)); ← Making
    }
}
```

progress

# Finding the factorial of 3



Time 2:  
Push: fact(3)

Time 3:  
Push: fact(2)

Time 4:  
Push: fact(1)

Time 5:  
Pop: fact(1)  
returns 1.

Time 6:  
Pop: fact(2)  
returns 2.

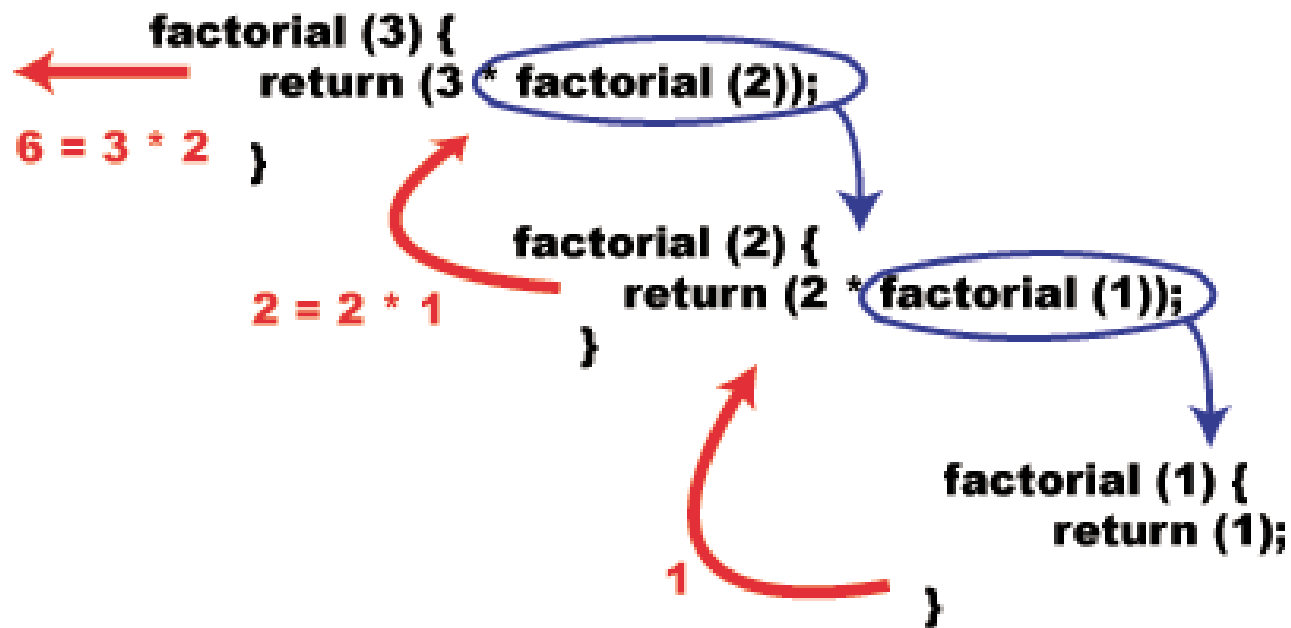
Time 7:  
Pop: fact(3)  
returns 6.

`findFactorial(3){`  
`if (number <= 1) return 1;`  
`else return (3 * factorial (2));`  
`}`

`findFactorial(2){`  
`if (number <= 1) return 1;`  
`else return (2 * factorial (1));`  
`}`

`findFactorial(1){`  
`if (number <= 1) return 1;`  
`else return (1 * factorial (0));`  
`}`

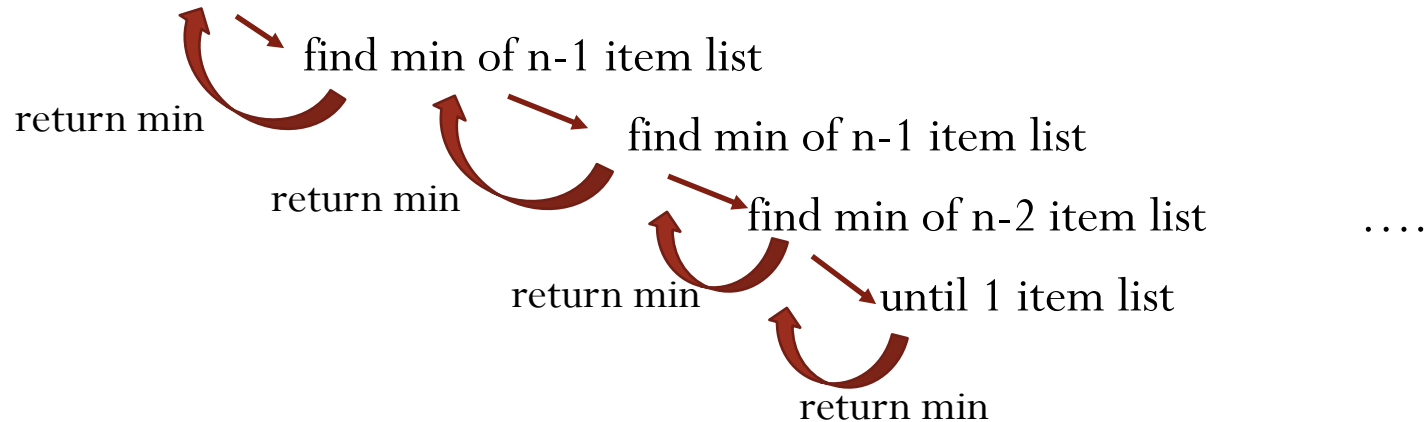
# Factorial recursive calls



# Find min using recursion

Think recursively, divide the problem into smaller sub-problem of similar kind.

To find min of n item list



```
public static int findMinRec(int A[], int n) {  
    // if size = 0 means whole array has been traversed  
    if(n == 1) return A[0];  
    return Math.min(A[n-1], findMinRec(A, n-1));  
}
```

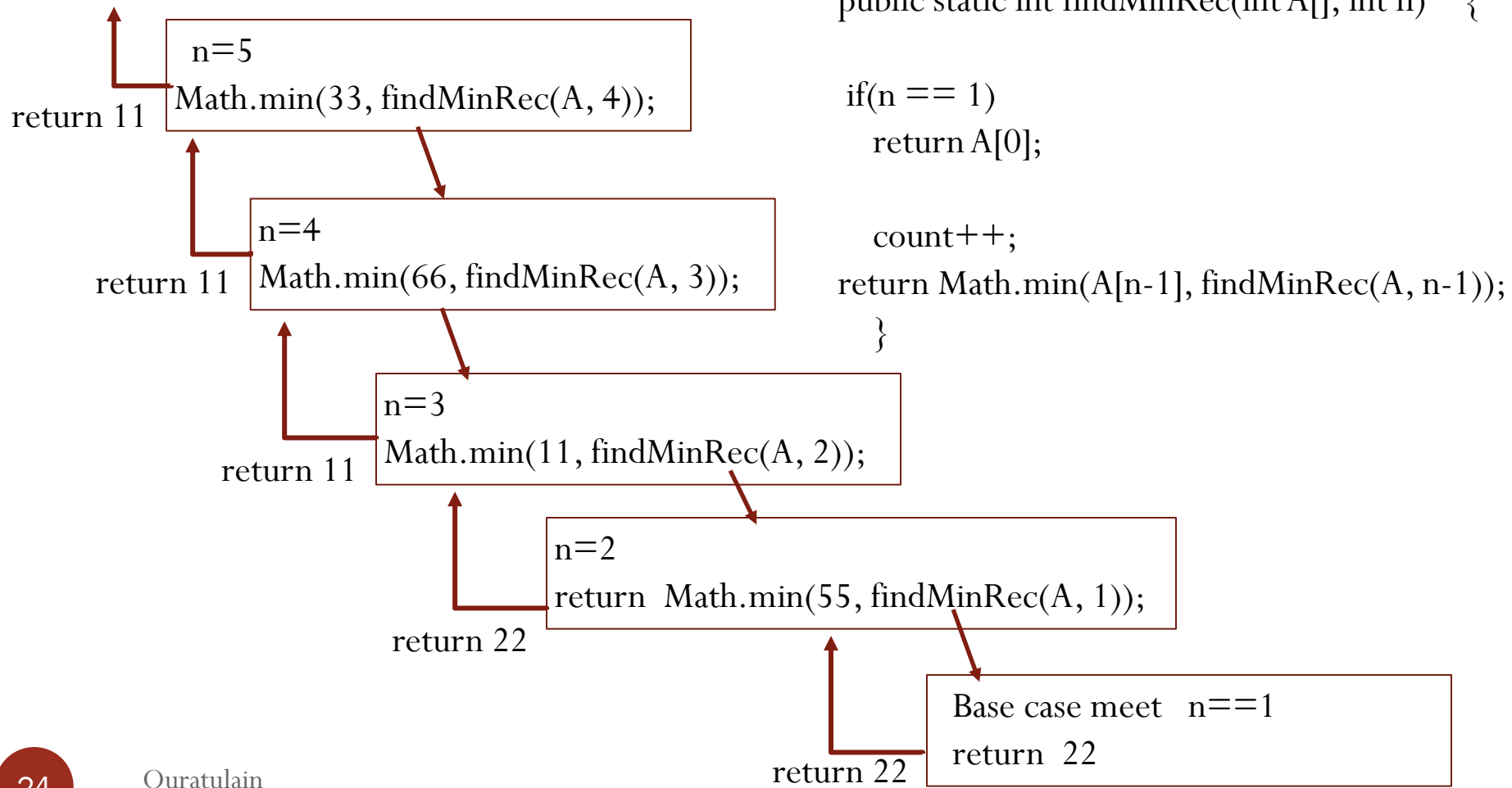
# Recursive calls trace for array of size 4. find min?

- $A = \{22, 55, 11, 66, 33\}$ ; so  $n=5$ , find min?

return 11

n=5

Math.min(33, findMinRec(A, 4));



return 11

n=4

Math.min(66, findMinRec(A, 3));

return 11

n=3

Math.min(11, findMinRec(A, 2));

return 22

n=2

return Math.min(55, findMinRec(A, 1));

return 22

Base case meet  $n==1$

return 22

```
public static int findMinRec(int A[], int n) {  
    if(n == 1)  
        return A[0];  
  
    count++;  
    return Math.min(A[n-1], findMinRec(A, n-1));  
}
```



# Example Using Recursion: The Fibonacci Series

- Fibonacci series
  - Each number in the series is sum of two previous numbers
    - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21...

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

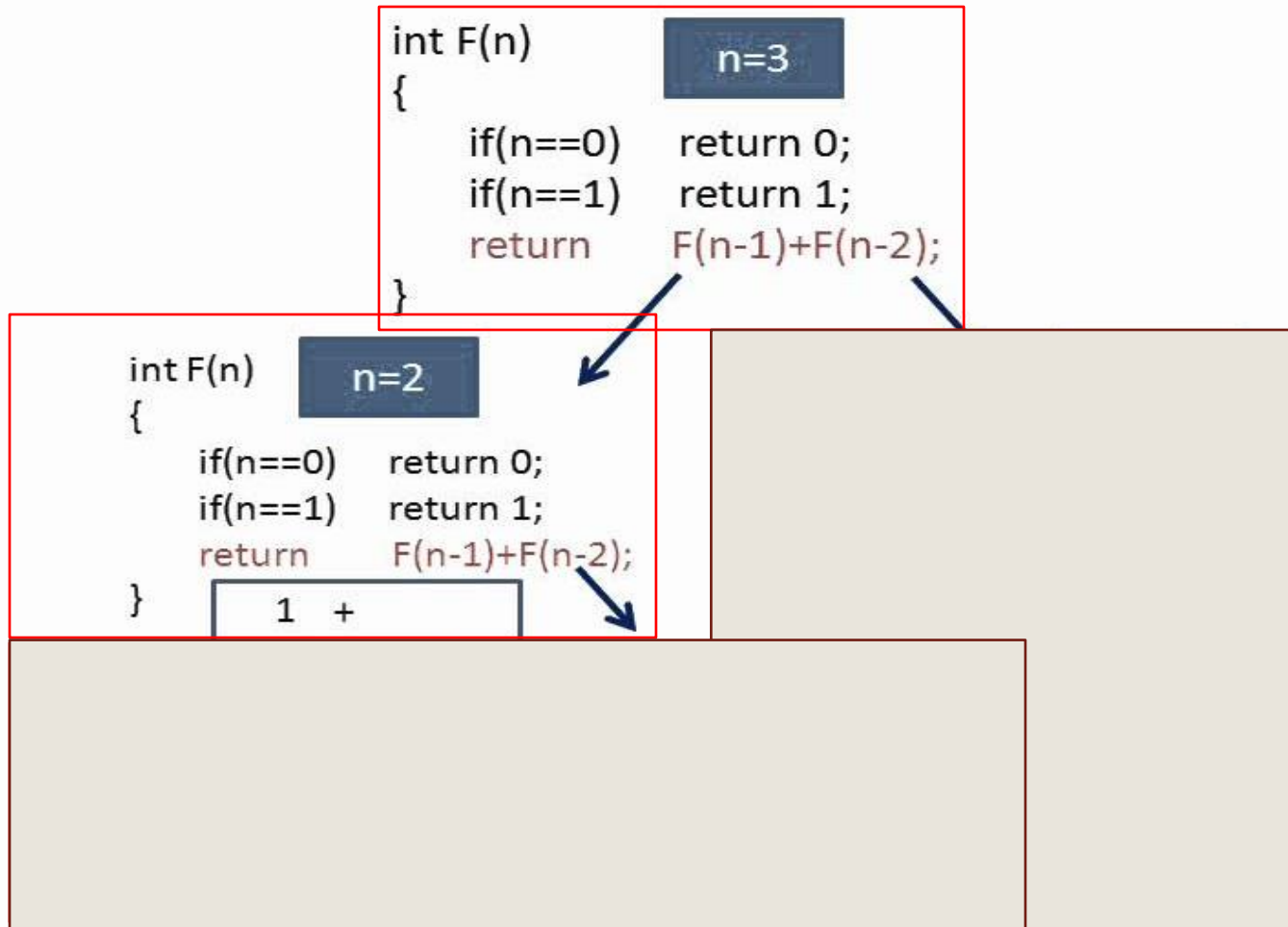
# Fibonacci

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n \geq 2 \end{cases}$$

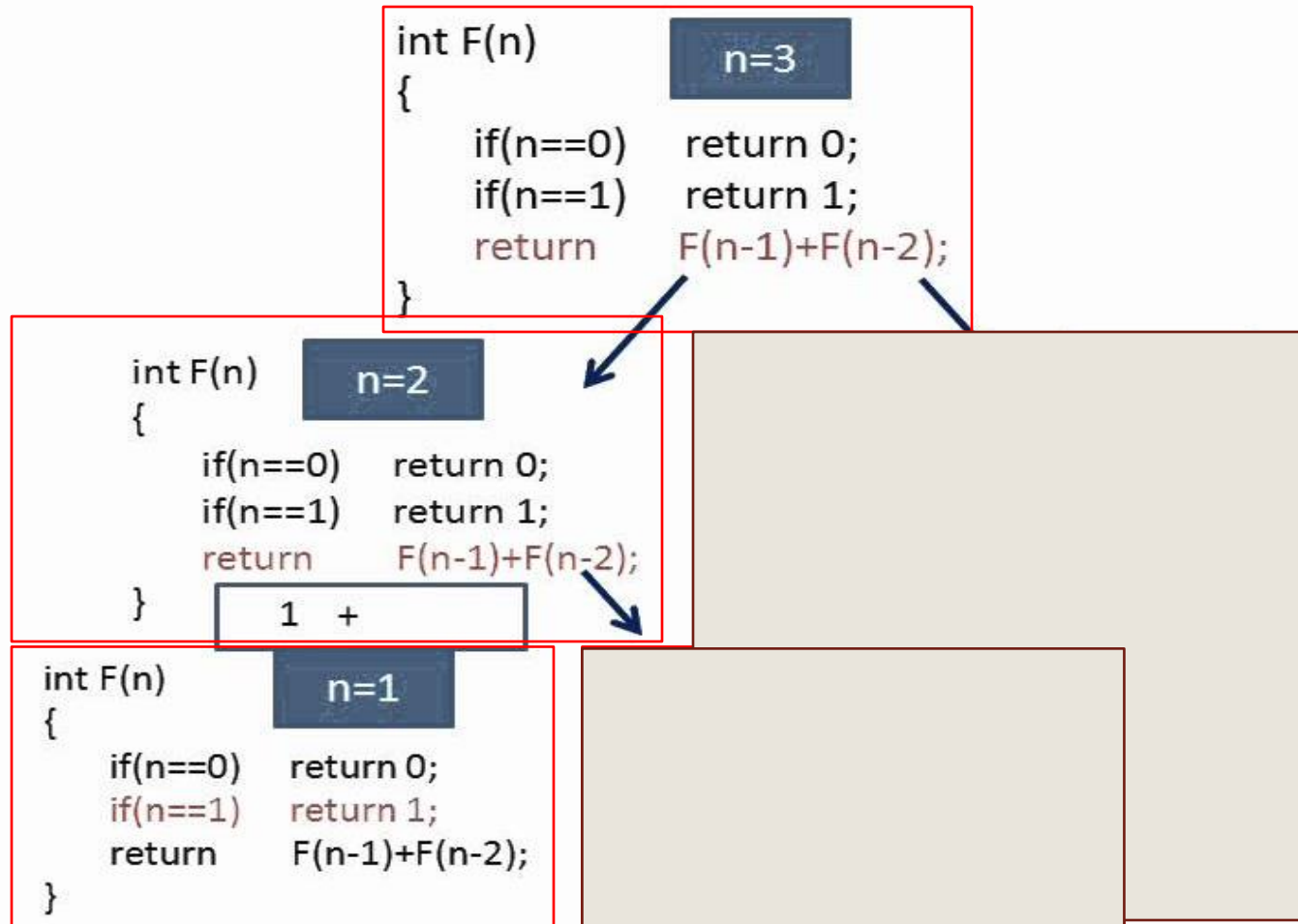
# Fibonacci Recursive calls

```
int F(n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    return F(n-1)+F(n-2);
}
```

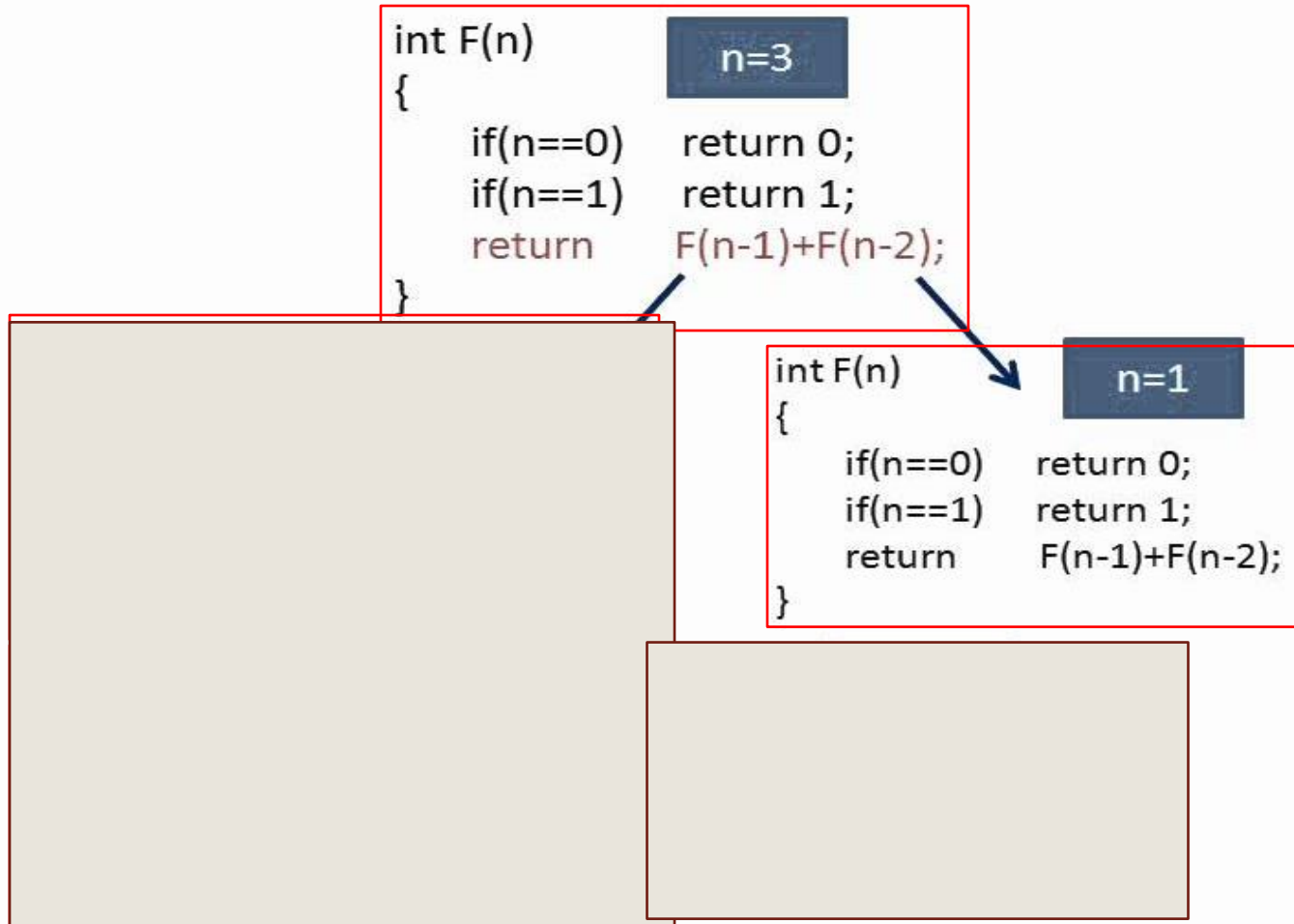
# Fibonacci Recursive calls



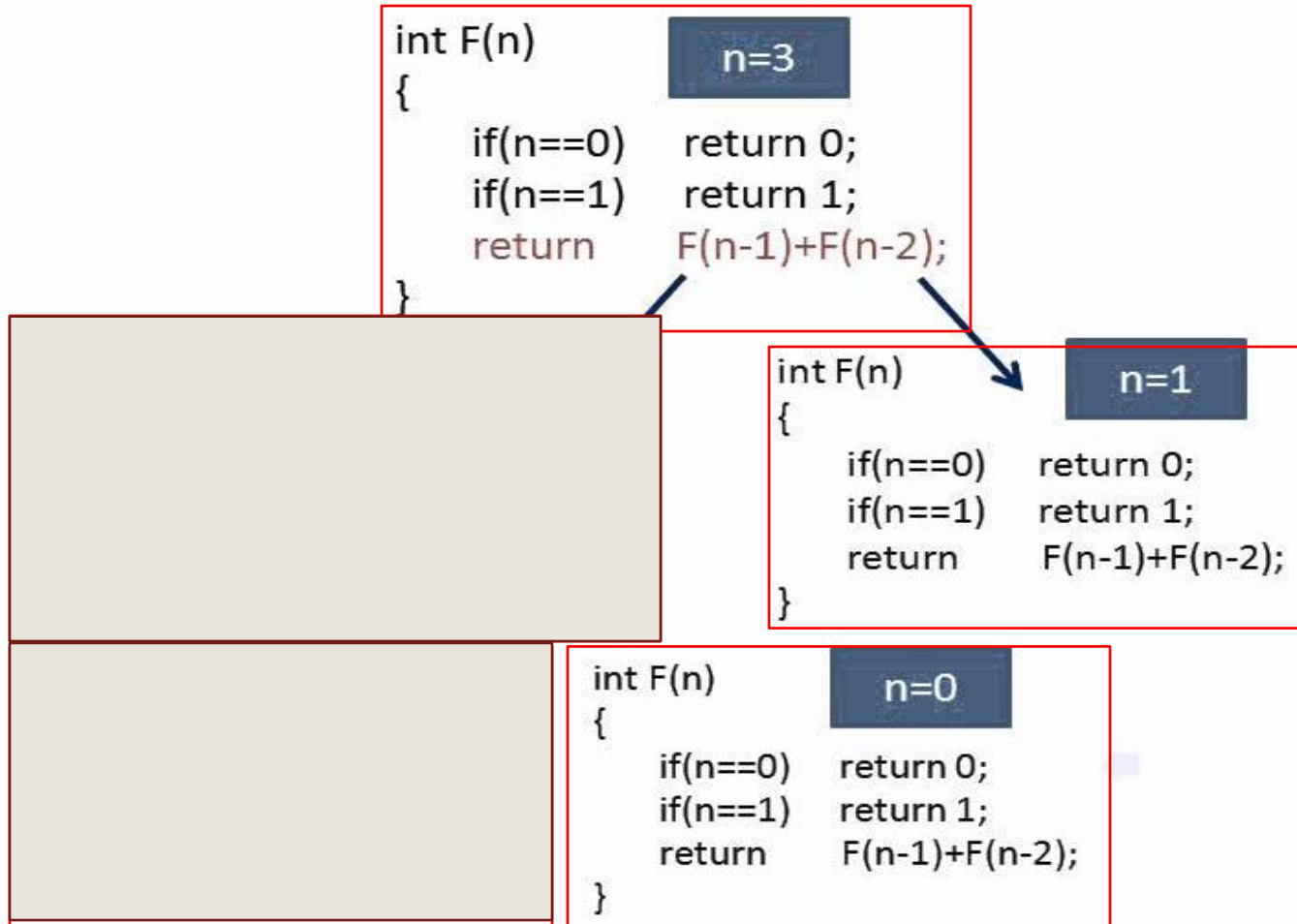
# Fibonacci Recursive calls



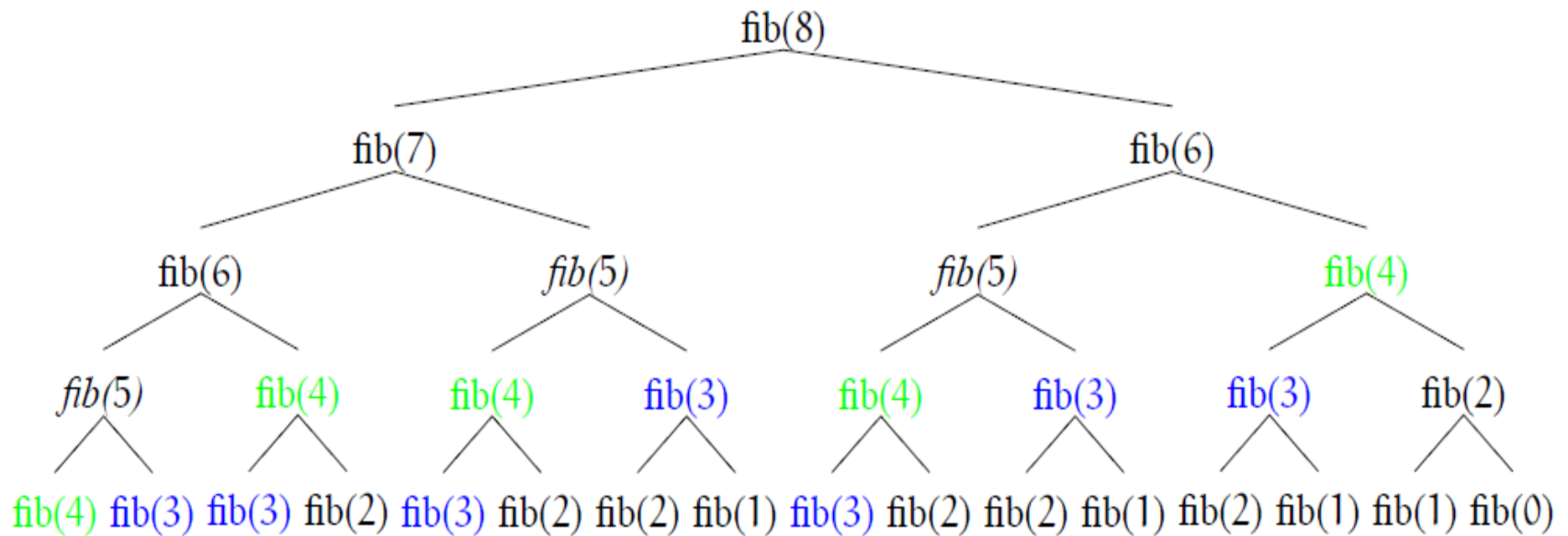
# Fibonacci Recursive calls



# Fibonacci Recursive calls



# Fibonacci Recursive calls



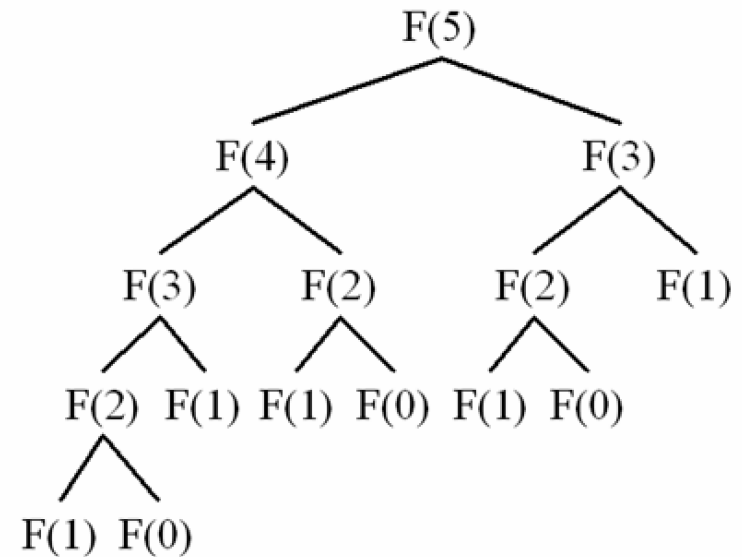
recursive calls to compute  $F(n) = O(2^n)$



# Fibonacci – avoid to recomputed same results

- save away intermediate results in a table.

```
int[] known = new int[1000];  
-----  
int F(int n) {  
    if (known[n] != 0)  
        return known[n];  
    else if (n == 0 || n == 1)  
        return n;  
    else {  
        known[n] = F(n-1) + F(n-2);  
        return known[n];  
    }  
}
```



This algorithm uses only  $2n$  recursive calls to compute  $F(n)$ , rather than  $2^n$

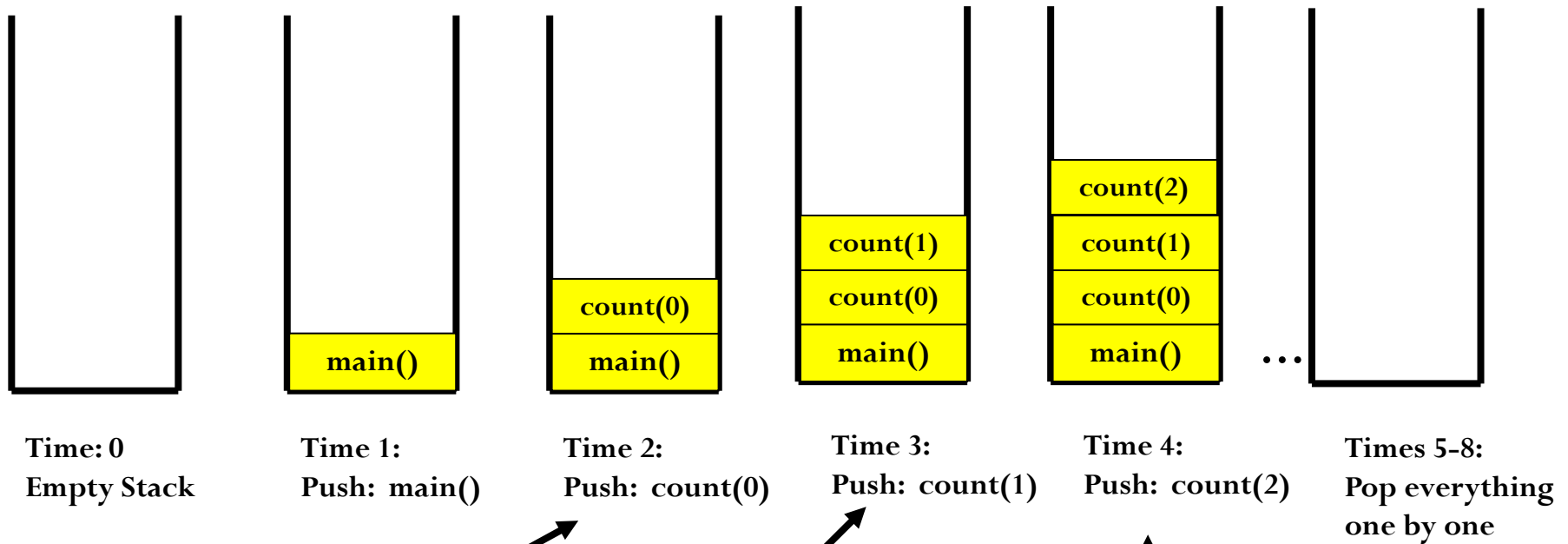
# Back to the Simple Recursion Program

- Here's the code again.
- Now, that we understand stacks, we can visualize the recursion.

```
public class Recursion
{
    public static void main (String args[])
    {
        count(0);
        System.out.println();
    }

    public static void count (int index)
    {
        System.out.print(index);
        if (index < 2)
            count(index+1);
    }
}
```

# Stacks and Recursion in Action



```
public static void count (int index){  
    print (index); → 0  
    if (index < 2)  
        count(index+1);  
}
```

```
public static void count (int index){  
    print (index); → 1  
    if (index < 2)  
        count(index+1);  
}
```

```
public static void count (int index){  
    count(2):  
    print (index); → 2  
    if (index < 2)  
        count(index+1);  
}
```

**This condition now fails!**

**Hence, recursion stops, and we proceed to pop all methods off the stack one by one.**

# Recursion, Variation 1

What will the following program do?

```
public class RecursionVar1
{
    public static void main (String args[])
    {
        count(3);
        System.out.println();
    }

    public static void count (int index)
    {
        System.out.print(index);
        if (index < 2)
            count(index+1);
    }
}
```

**Base case never meet...  
run infinitely**

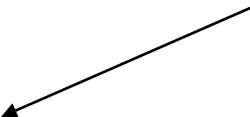
# Recursion, Variation 2

What will the following program do?

```
public class RecursionVar2
{
    public static void main (String args[])
    {
        count(0);
        System.out.println();
    }

    public static void count (int index)
    {
        if (index < 2)
            count(index+1);
        System.out.print(index);
    }
}
```

**Note that the print statement  
has been moved to the end  
of the method.**



# Recursion, Variation 3

What will the following program do?

```
public class RecusionVar3
{
    public static void main (String args[])
    {
        count(3);
        System.out.println();
    }

    public static void count (int index)
    {
        if (index > 2)
            count(index+1);
        System.out.print(index);
    }
}
```

# Problem: Not working towards base case

- In variation #3, we do not work towards our base case. This causes infinite recursion and will cause our program to crash.
- Java throws a `StackOverflowError` exception.
- In some other languages this is a fatal error.

# Tower of Hanoi, History

- The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883.
- He was inspired by a legend of a Hindu temple where the puzzle was presented to young priests.
- The priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it.
- Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints.
  - They could only move one disk at a time, and
  - they could never place a larger disk on top of a smaller one.
- The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.
- Although the legend is interesting, you need not worry about the world ending any time soon.



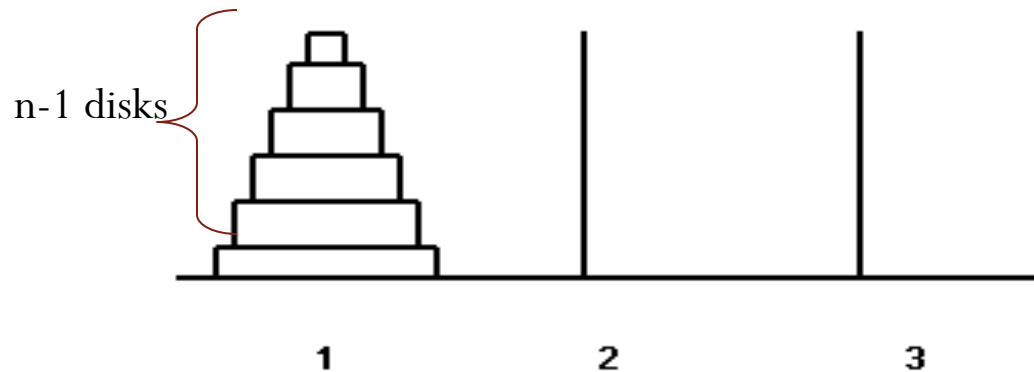
# Tower of Hanoi

- Moves required to correctly move a tower of 64 disks is  
 $2^{64} - 1 = 18,446,744,073,709,551,616$
- At a rate of one move per second, that is  
 $584,942,417,355584,942,417,355$  years!

Clearly there is more to this puzzle than meets the eye.

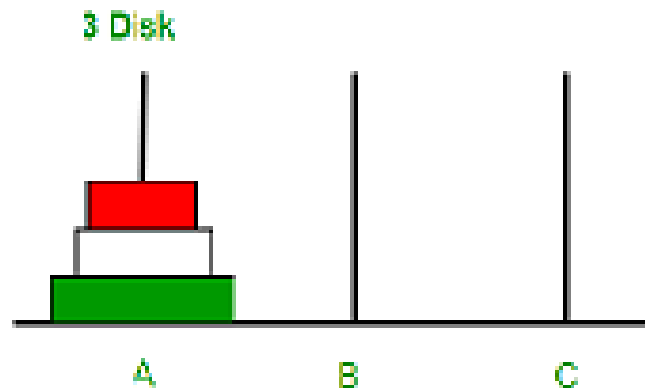
# Example

- Move the top  $n-1$  disks from 1 to 2 using 3 as auxiliary
- Move the disk from 1 to 3
- Move the  $n-1$  disks from 2 to 3 using 1 as auxiliary



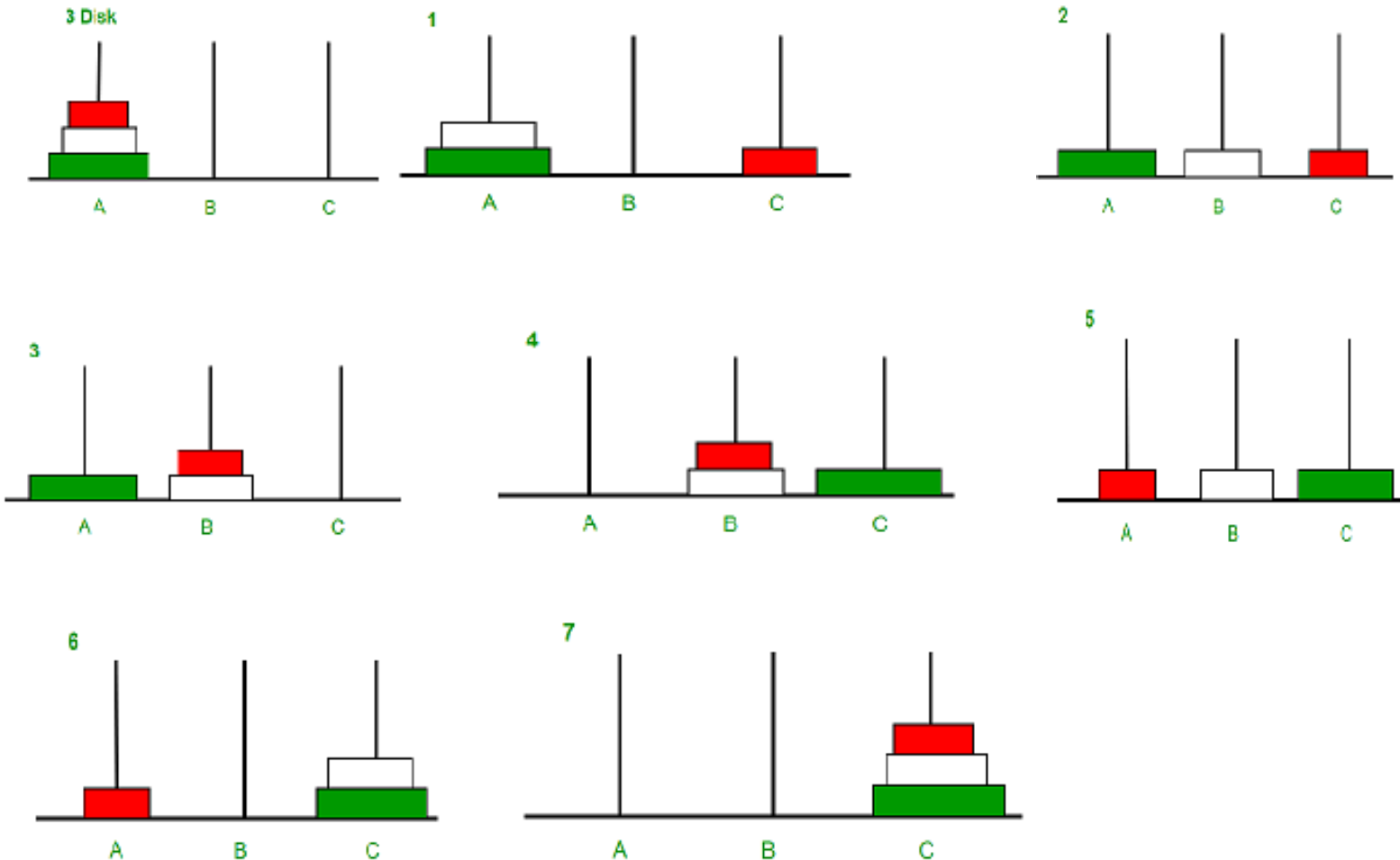
# Recursive solution

1. If  $n=1$ , move the single disk from A to C and stop
2. Move the top  $n-1$  disks from A to B using C as auxiliary
3. Move the disk from A to C
4. Move the  $n-1$  disks from B to C using A as auxiliary



# Tower with 3 disk

Move all disks from rod A to rod C



# Pseudo-code

- Call tower (n, 'A', 'C', 'B');

## Function Tower (int n, char from, char to, char aux)

$$\{ \quad \quad \quad A \quad \quad C \quad \quad B$$

**If ( $n==1$ )**

 $\{$ 

```
print(from + "-" to) return;
```

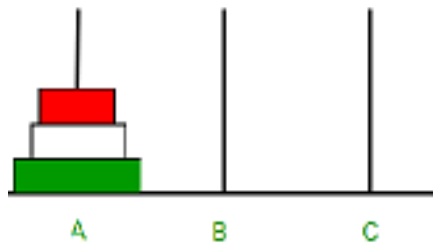
$\}$	A	B	C
------	---	---	---

Tower(n-1, from, aux, to);

```
Print (from+"-"+to) ;
```

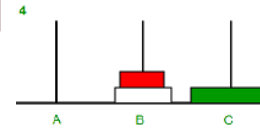
Tower (n-1,aux,to,from);

}



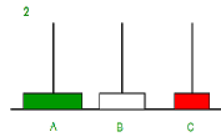
```
Tower(N=3,from=A,to=C,aux=B){
  If(n==1){print (from+"-"+to); return;}
  else{
    Tower(2,from,aux,to);
    print (from+"-"+to)
    Tower(2,aux,to,from);
  }}

```



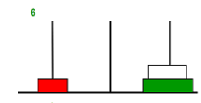
```
Tower(N=2,from=A,to=B,aux=C){
  If(n==1)print (from+"-"+to); return;}
  else{
    Tower(1,from,aux,to);
    print (from+"-"+to)
    Tower(1,aux,to,from);
  }}

```

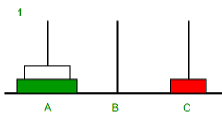


```
Tower(N=2,from=B,to=C,aux=A){
  If(n==1)print (from+"-"+to); return;}
  else{
    Tower(1,from,aux,to);
    print (from+"-"+to)
    Tower(1,aux,to,from);
  }}

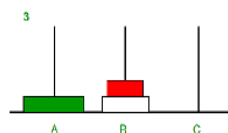
```



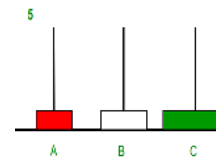
```
N=1,fromA,to=C,aux=B
If(n==1){
  print (from+"-"+to);
  return;
}
else{
}
```



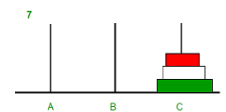
```
N=1,from=C,to=B,aux=A
If(n==1){
  print (from+"-"+to);
  return;
}
else{
}
```



```
N=1,from=B,to=A,aux=C
If(n==1){
  print (from+"-"+to);
  return;
}
else{
}
```



```
N=1,from=A,to=C,aux=B
If(n==1){
  print (from+"-"+to);
  return;
}
else{
}
```



# Tower of Hanoi Big Oh

- Let's figure out values of  $T$  for the first few numbers.

$$T(1) = 1$$

$$T(2) = 2T(1) + 1 = 3$$

$$T(3) = 2T(2) + 1 = 7$$

$$T(4) = 2T(3) + 1 = 15$$

$$T(5) = 2T(4) + 1 = 31$$

....

Recurrence relation:

$$T(n) = 2T(n-1) + 1$$

$$T(n) = 2[2T(n-2) + 1] + 1$$

$$= 2^2 T(n-2) + 3$$

$$= 2[2[2T(n-3) + 1] + 1] + 1$$

$$= 2^3 T(n-3) + 7 \quad \dots\dots$$

For  $n$  disc

$$T(n) = 2^k T(n-k) + 2^k - 1$$

as  $k \rightarrow n$  so,

$$T(n) = 2^k T(n-k) + 2^k - 1$$

$$T(n) = 2^n T(n-n) + 2^n - 1$$

$$= 2^n (0) + 2^n - 1$$

$$= 2^n - 1$$

$$= O(2^n)$$

We can verify this easily by plugging it into our recurrence:

$$T(1) = 2^1 - 1 = 1$$

$$T(2) = 2^2 - 1 = 3$$

$$T(3) = 2^3 - 1 = 7$$

# Expressive power

- Most programming languages in use today allow the direct specification of recursive functions and procedures. When such a function is called, the program's runtime environment keeps track of the various instances of the function (often using a call stack, although other methods may be used).
- Every recursive function can be transformed into an iterative function by replacing recursive calls with iterative control constructs and simulating the call stack with a stack explicitly managed by the program



# Performance issues

- In languages (such as C and Java) that favor iterative looping constructs, there is usually significant time and space cost associated with recursive programs, due to the overhead required to manage the stack and the relative slowness of function calls.
- Efficiency is not determined solely by execution time and memory usage. Efficient use of a programmer's time also is an important consideration.

# Recursion pros and cons

- All recursive solutions can be implemented **without recursion**.
- Recursion is "expensive". The expense of recursion lies in the fact that we have **multiple activation frames** and the fact that there is **overhead involved with calling a method**.
- If both of the above statements are true, why would we ever use recursion.
  - In many cases, the extra "expense" of recursion is balanced by a **simpler more clear algorithm which leads to an implementation that is easier to code**.
- Ultimately, **the recursion is eliminated when the compiler creates assembly language (it does this by implementing the stack)**.

# Recursion vs. Iterations

Recursion	Iteration
Mechanism defining in terms of itself.	Block of statement executed repeatedly using loop.
Base case condition is required to stop the execution.	Loop condition contains statement for stopping the execution.
At some places, use of recursion generates extra overhead in term of time and space. Therefore, better to skip when easy solution available in iterative manner.	All problems can be solved with iteration. But for some problems logic become lengthy and complicated in iterative solution not simple as in recursion.
Recursion is expensive in terms of speed and memory.	Iteration does not create any overhead. All the programming languages support iteration.
Some of the problems describe easily in simple code using recursion.	Some problems become complicated/lengthy to code in the iterative solution.
Compiler manage the use of stack so no implementation less burden on programmer.	Programmers have to develop own stack and have to manage it.

# Quiz

- Write code to print Arraylist in reverse order in  $O(n)$ .

Understand recursive and iterative solutions and discuss when you prefer one solution over other.

# Quiz

- Compute powers using recursive implementation where base and power provided as input.

# Quiz

1. Write a recursive function that takes as a parameter a nonnegative integer and generates the following pattern of stars. If the nonnegative integer is 4, the pattern generated is as follows:

```
*****  
***  
**  
*  
*  
**  
***  
*****
```

# Quiz

- Write a recursive function, vowels, that returns the number of vowels in a string. Also, write a program to test your function.

# Quiz

- Find max value in array using recursion. Understand big oh and draw memory images to visualize the working of recursion.



# Quiz

1. How many recursive call are made in tower of Hanoi function, when  $n = 3$ .
2. How many recursive call are made in tower of Hanoi function, when  $n = 20$ .
3. At most how many frames are created in stack during recursive calls of tower of Hanoi method when  $n=4$ .

# Reference

- <https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/>