

Fundamentals of Programming

CCS1063/CSE1062

Lecture 8 –Recursion

Professor Noel Fernando



Recursion



What is recursion?

- Recursion is the process of defining something in terms of itself.
- A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.
- Recursion is the technique of making a function call itself.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.

How does recursion work?

How does recursion work?

sum
`void recurse()`

`{`

`... ..`

`recurse();`

`... ..`

`}`

`int main()`

`{`

`... ..`

`recurse();`

`... ..`

`}`

recursive
call

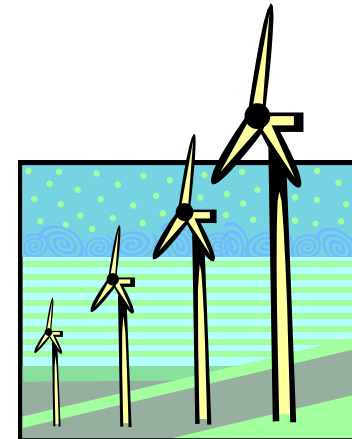
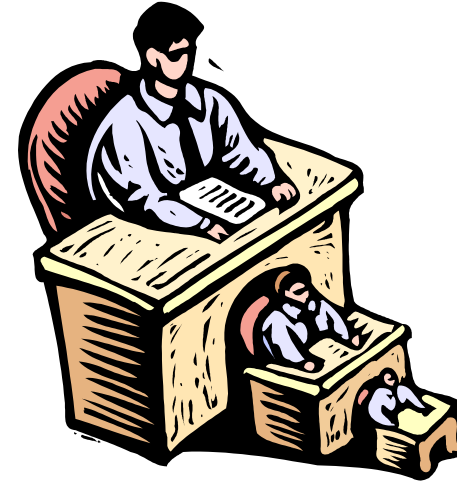
*void sum(int x) {
int y = 0;
if (x <= 10) {
y = x + sum(x+1);
0 +*

*{
return y;
}*

*main() {
sum(x)*

Basic Recursions

- Break a problem into smaller identical problems
 - Each recursive call solves an identical but smaller problem.
- Stop the break-down process at a special case whose solution is obvious, (termed **a base case**)
 - Each recursive call tests the base case to eventually stop.
 - Otherwise, we fall into an infinite recursion.



Getting downstairs



- Need to know two things:
 - Getting down one stair
 - Recognizing the bottom

- Most code will look like:

```
if (simplest case) {  
    compute and return solution  
} else {  
    divide into similar subproblem(s)  
    solve each subproblem recursively  
    assemble the overall solution  
}
```

Recursive Definitions

- Recursion
 - Process of solving a problem by reducing it to smaller versions of itself
- Recursive definition
 - Definition in which a problem is expressed in terms of a smaller version of itself
 - Has one or more base cases

Factorial Program in C

- Factorial of n is the *product of all positive descending integers*. Factorial of n is denoted by $n!$. For example:
- $5! = 5*4*3*2*1 = 120$
- $3! = 3*2*1 = 6$
- In general, $n!$ (" n^{th} factorial") means the product of all the whole numbers from 1 to n ; that is, $n! = 1 \times 2 \times 3 \times \dots \times n$.

Write a C program to find the Factorial for n values using a loop (iterative)

```
int Sum(0, 1, 2, 3, 4, 10xn) {  
    int s = 0;  
    if (n <= 10) {  
        0+1+2+3+...+10  
        y = n + Sum(n+1);  
    }  
    return y;  
}
```

Handwritten note: 0+1+2+3+...+10

```
main() {  
    int n;  
    sum(n);  
}
```

Factorial Program using loop (iterative)

```
#include<stdio.h>

int main()
{
    int i,fact=1,number;
    printf("Enter a number: ");
    scanf("%d",&number);
    for(i=1;i<=number;i++){
        fact=fact*i;
    }
    printf("Factorial of %d is: %d",number,fact);
    return 0;
}
```

Output:

Enter a number: 5
Factorial of 5 is: 120

Write a C program to calculate factorial n
(Recursive)

Example 1: The Factorial of n

- Definition:

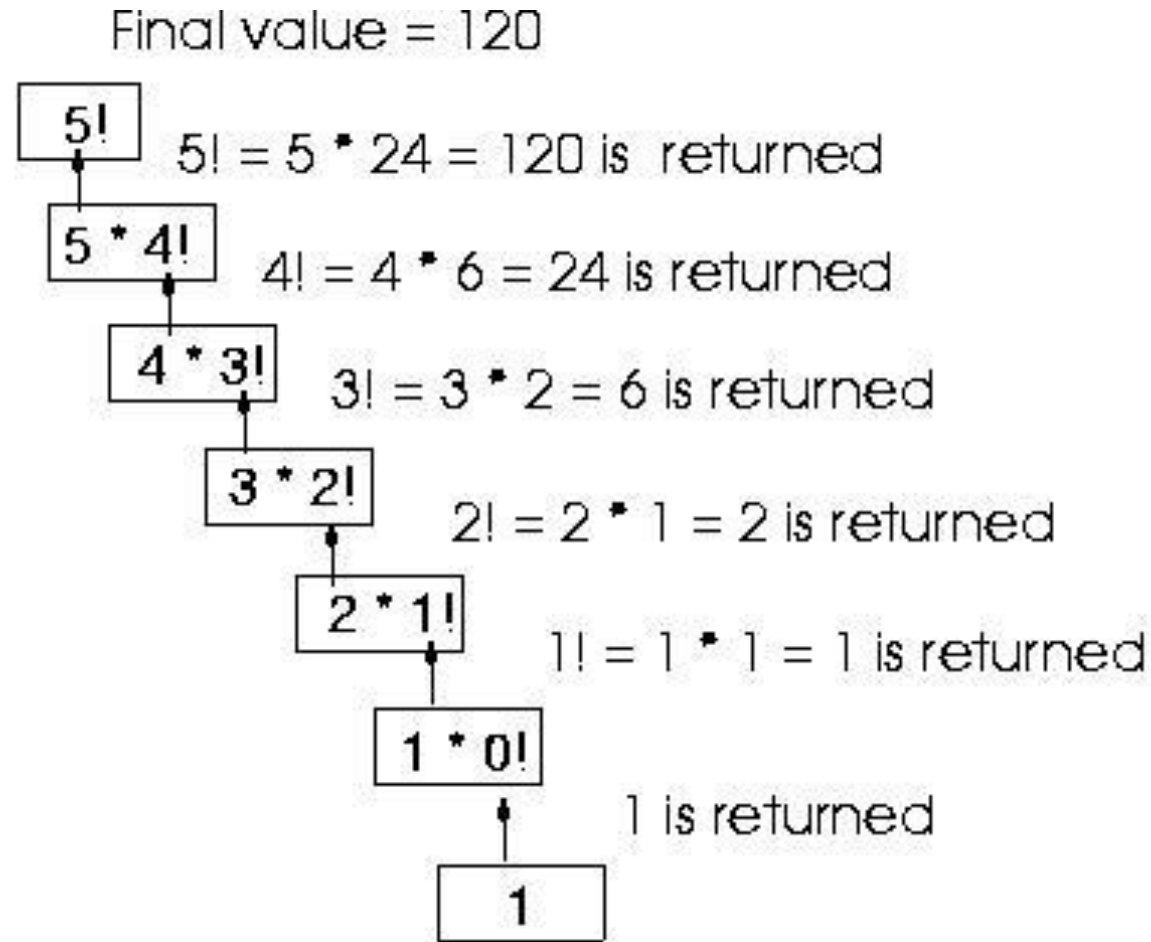
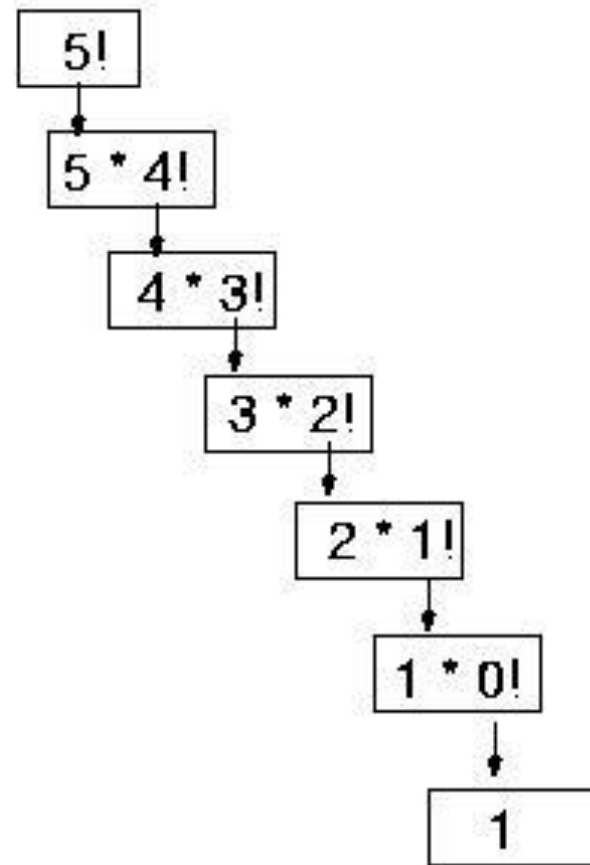
$\text{factorial}(n) = n * (n-1) * (n-2) * \dots * 1$ for any integer $n > 0$

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

- Final recursive definition:

$\text{factorial}(n)$	$= 1$	if $n = 1$
	$= n * \text{factorial}(n-1)$	if $n > 1$

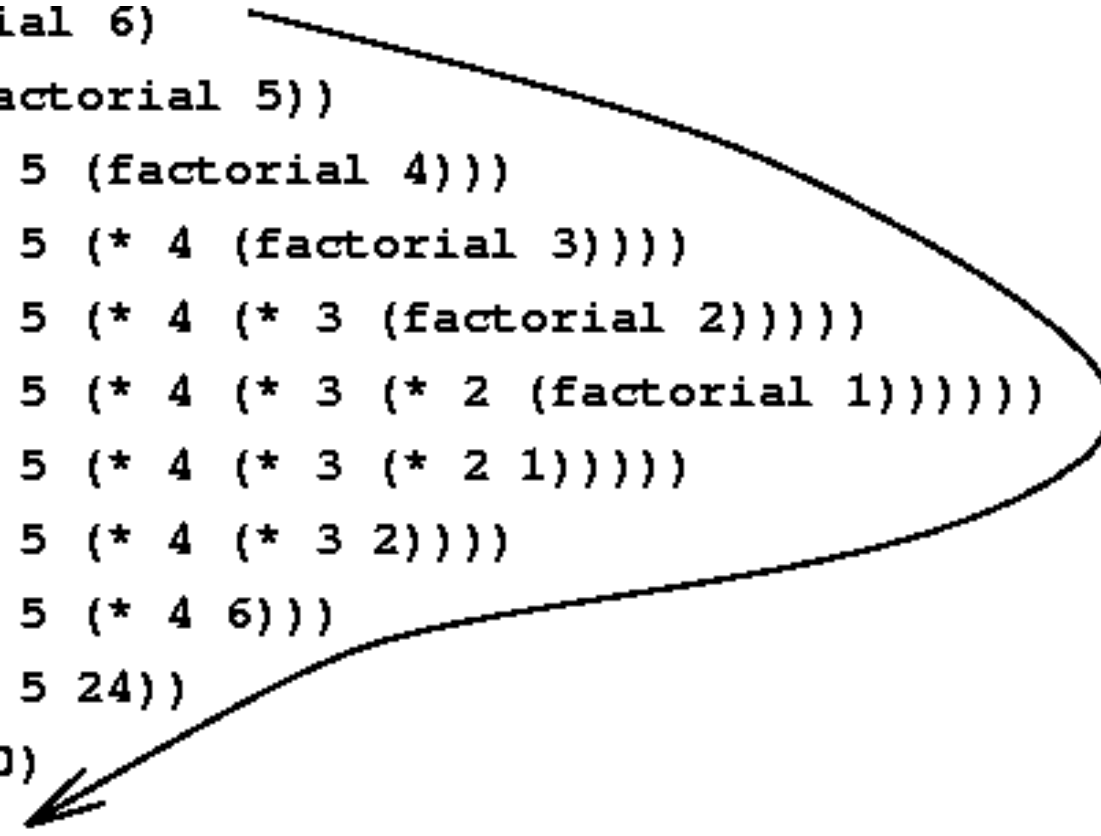
Example 1: The Factorial of n (Recursive approach)



Example 1: The Factorial of n (Recursive approach)

If we called it as
`factorial(6)`?

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```



Recursive Factorial Function

- Factorial $n = 1 \times 2 \times 3 \dots \times n$

```
int factorial(int n) {  
    if (n<=1)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

Write a C program to calculate factorial n (recursive)

```
#include<stdio.h>
```

```
long factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return(n * factorial(n-1));
}
```

```
void main()
{
    int number;
    long fact;
    printf("Enter a number: ");
    scanf("%d", &number);
    fact = factorial(number);
    printf("Factorial of %d is %ld\n", number, fact);
    return 0;
}
```


Infinite Recursion

- If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates.
- This is known as infinite recursion, and it is generally not a good idea.
- As the base condition is never met, the recursion carries on infinitely.

Example : Infinite Recursion

```
// C program to demonstrate Infinite Recursion
```

```
#include <bits/stdc++.h> is a precompiled header  
implementation file.
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
// Recursive function
```

```
void Geek(int N) {
```

```
// Base condition
```

```
    // This condition is never met here
```

```
    if (N == 0)
```

```
        return;
```

```
    // Print the current value of N
```

```
    cout << N << " ";
```

```
    // Call itself recursively
```

```
    Geek(N); }
```

```
// Driver code
```

```
int main()
```

```
{
```

```
    // Initial value of N
```

```
    int N = 5;
```

```
    // Call the recursive function
```

```
    Geek(N);
```

```
    return 0;
```

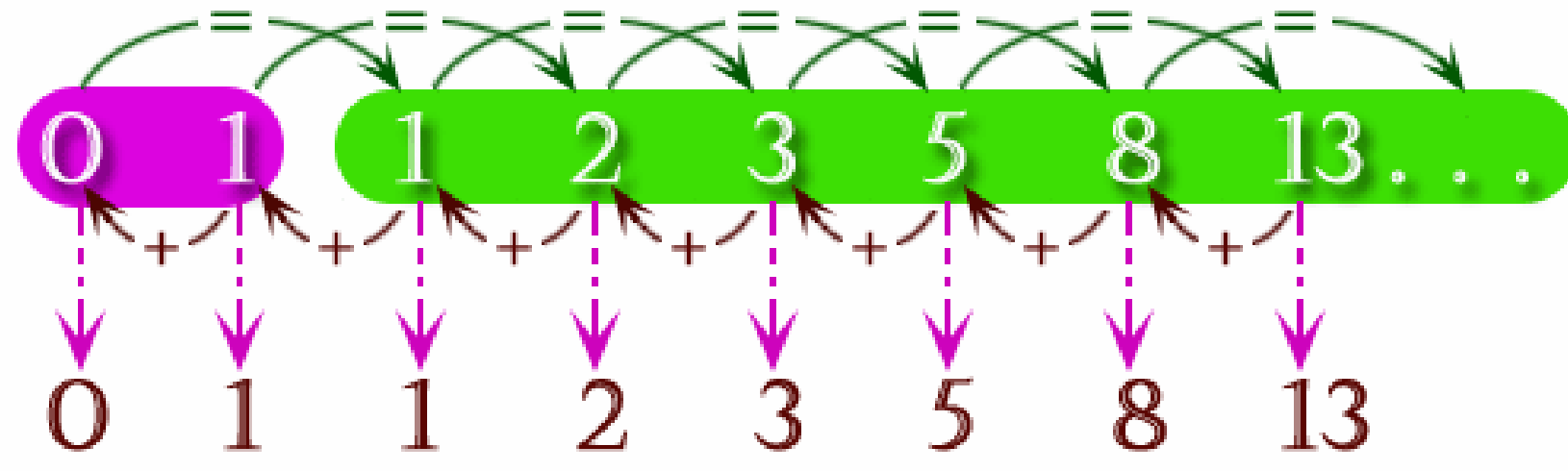
```
}
```

Fibonacci numbers

- Write a C program to get the Fibonacci series between 0 to 50.
- Note : The Fibonacci Sequence is the series of numbers :0, 1, 1, 2, 3, 5, 8, 13, 21,
Every next number is found by adding up the two numbers before it.
Expected Output : 0 1 1 2 3 5 8 13 21 34

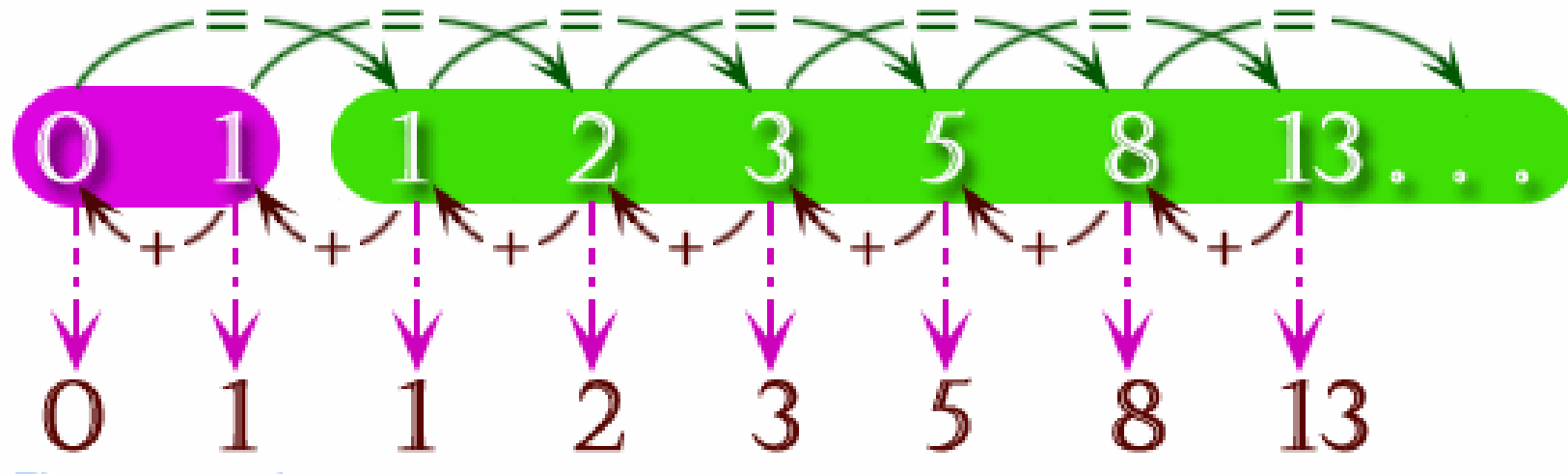
Answer

Fibonacci Sequence :



Answer

Fibonacci Sequence :

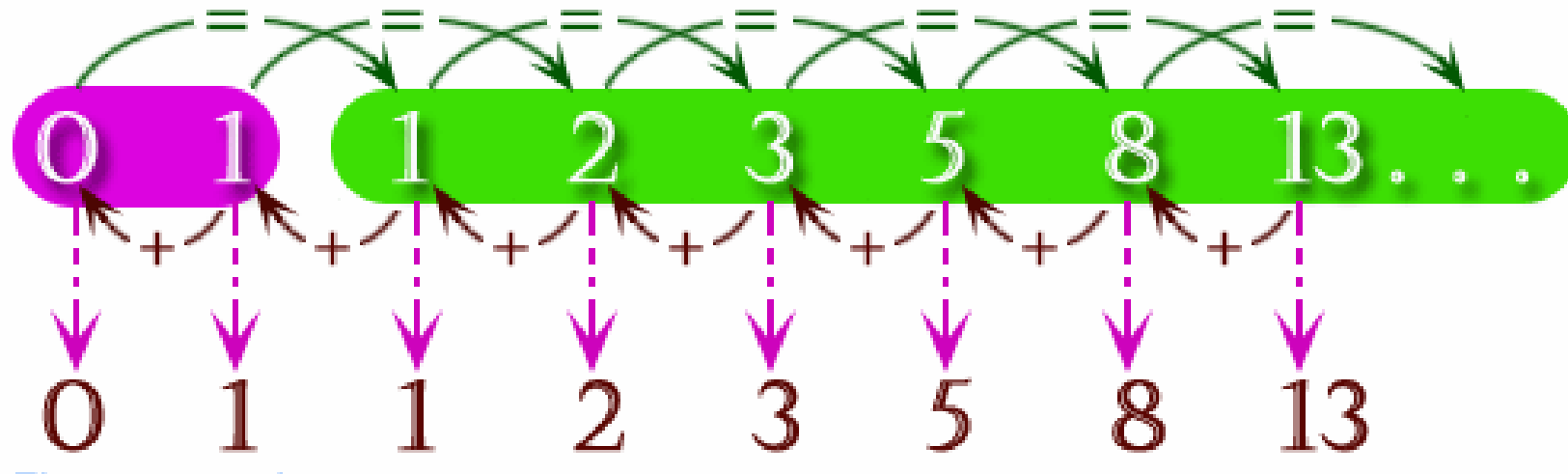


$x, y = 0, 1$

```
while y < 50:  
    print(y)  
    x, y = y, x + y
```

Answer

Fibonacci Sequence :



$x, y = 0, 1$

```
while y < 50:  
    print(y)  
    x, y = y, x + y
```

Fibonacci numbers

- Another well known recursive sequence is the Fibonacci numbers.
The first few elements of this sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21...

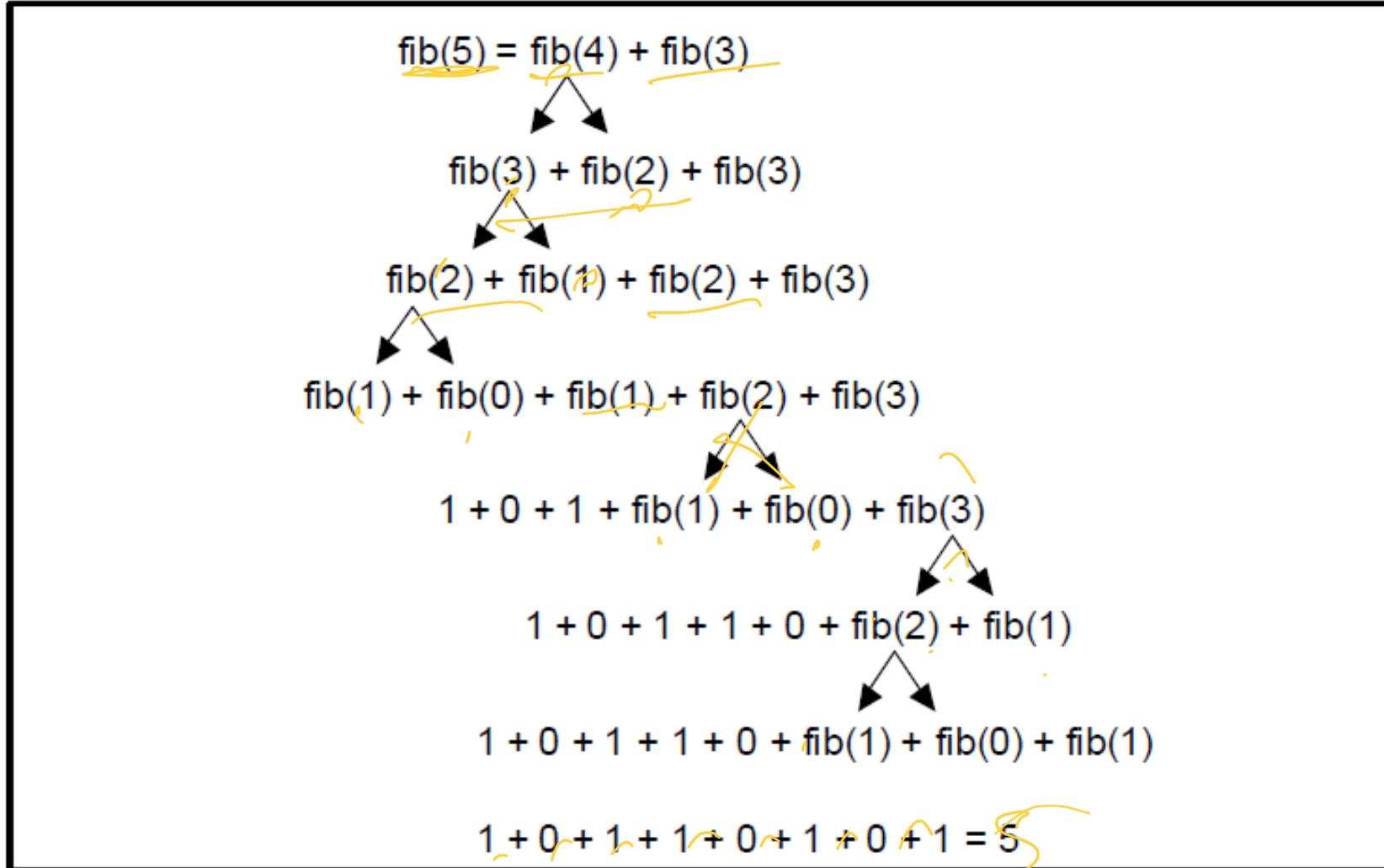
$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n \geq 2 \end{cases}$$

Fibonacci Sequence Problem

- A Fibonacci sequence starts with the integers 0 and 1.
- Successive elements in this sequence are obtained by summing the preceding two elements in the sequence.
- For example,
- third number in the sequence is $0 + 1 = 1$, fourth number is $1 + 1 = 2$,
- Fifth number is $1 + 2 = 3$ and so on.

Fibonacci number sequence is : 0 1 1 2 3 5 8 13 21

We will now use the definition to compute $\text{fib}(5)$:



Fibonacci- Pseudo code

- **function** fib is:

input: integer n such that $n \geq 0$

1. if n is 0, **return** 0
 2. if n is 1, **return** 1
 3. otherwise, **return** [fib($n-1$) + fib($n-2$)]
- end** fib

Fibonacci- Pseudo code

- **function** fib is:

input: integer n such that $n \geq 0$

1. if n is 0, **return** 0
 2. if n is 1, **return** 1
 3. otherwise, **return** [fib(n-1) + fib(n-2)]
- end** fib

C code to find Fibonacci numbers for n

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

```
int fibonacciPrint(int n) {  
    for(int i=1; i<=n; i++)  
        printf("%d, ", fibonacci(i));  
} // fibonacciPrint
```

Fibonacci- Java code

(this is only for comparison)

```
import java.*;
```

```
public class fibanach {
```

```
    public static void main (String[] args) {
```

```
        int thefib;
```

```
        System.out.println("This program computes the fibonacci value " +  
        "of a number.");
```

Fibonacci- Java code

(this is only for comparison)

```
System.out.print("Enter a number: ");  
int thenum=Integer.parseInt(args[0]);  
thefib = fibi(thenum);  
    System.out.println(thefib + " = " +thefib + ".");  
    }
```

Fibonacci- Java code

(this is only for comparison)

```
static int fibi(int n) {  
    if (n == 0) {  
        return 0 ;  
    }  
    else  
    if (n==1) {  
        return 1;  
    }  
}
```

Fibonacci- Java code

(this is only for comparison)

// Recursive Case:

```
    else {  
        return fibi(n-1)+fibi(n-2);  
    }  
}  
}
```


What is a "Common Factor" ?

Let us say you have worked out the factors of two numbers:

Example: Factors of 12 and 30

Factors of 12 are 1, 2, 3, 4, 6 and 12

Factors of 30 are 1, 2, 3, 5, 6, 10, 15 and 30

Then the common factors are those that are found in both lists:

- Notice that 1, 2, 3 and 6 appear in both lists?
- So, the common factors of 12 and 30 are: 1, 2, 3 and 6

Greatest Common Factor

The highest number that divides exactly into two or more numbers.

It is the "greatest" thing for simplifying fractions!

Let's start with an Example ...

Greatest Common Factor
of 12 and 16

- Find all the **Factors** of each number,
 - Circle the **Common** factors,
- Choose the **Greatest** of those

The "Greatest Common Factor" is the largest of the common factors (of two or more numbers)



Exercise

- Write a recursive program to determine the Greatest Common Factor

Recursive function for gcd

```
def gcd(a,b):  
    t = b  
    b = a % b  
  
    if b == 0:  
        return t  
    else:  
        return gcd(t,b)
```

print(gcd(12,6)) ?

Print gcd(30,12) ?

Exercise 2

Write a recursive C function that returns the sum of the first n integers.
(Hint: The function will similar to the factorial function!)

Example 1: The Sum of the First N Positive Integers

- Definition:

$$\text{sum}(n) = n + (n-1) + (n-2) + \dots + 1 \text{ for any integer } n > 0$$

- Recursive relation;

$$\begin{aligned}\text{sum}(n) &= n + [(n-1) + (n-2) + \dots + 1] \\ &= n + \text{sum}(n-1)\end{aligned}$$

Looks so nice, but how about $n == 1$?

$\text{sum}(1) = 1 + \text{sum}(0)$, but the argument to $\text{sum}()$ must be positive

- Final recursive definition:

$$\begin{aligned}\text{sum}(n) &= 1 && \text{if } n = 1 \text{ (Base case)} \\ &= n + \text{sum}(n-1) && \text{if } n > 1 \text{ (Recursive call)}\end{aligned}$$

Example 1: The Sum of the First N

- findSum(n):
 IF $n \leq 1$ THEN
 RETURN n
 ELSE
 RETURN $n + \text{findSum}(n-1)$
END FUNCTION

Sum of Natural Numbers Using Recursion

```
include <stdio.h>
int addNumbers(int n);
int main() {
int num; printf("Enter a positive
integer: ");
scanf("%d", &num);
printf("Sum = %d",
addNumbers(num));
return 0; }
```

```
int addNumbers(int n) {
    if (n != 0)
return n + addNumbers(n - 1);
else
    return n; }
```


Exercise -1

- Think of a recursive version of the function $f(n) = 3 * n$, (i.e. the multiples of 3)

Exercise -1

Mathematically, we can write it like this:

$$f(1)=3;$$

$$f(n+1)=f(n)+3$$

A python function can be written like this :

```
def mult3(n):  
    if n == 1:  
        return 3  
    else:  
        return mult3(n-1) + 3
```

```
for i in range(1,10):  
    print(mult3(i))
```

Exercise -1

```
>>>
```

```
3
```

```
6
```

```
9
```

```
12
```

```
15
```

```
18
```

```
21
```

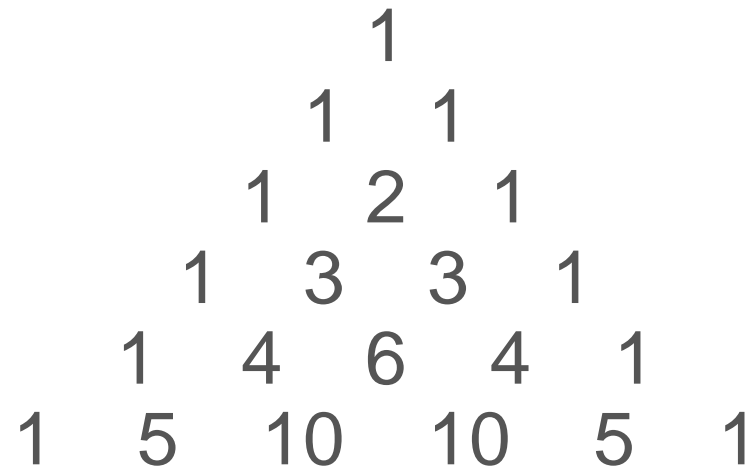
```
24
```

```
27
```

```
>>>
```

Exercise 2

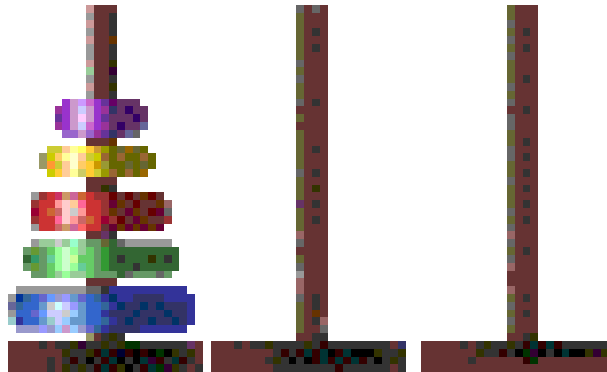
Write a function which implements the Pascal's triangle:



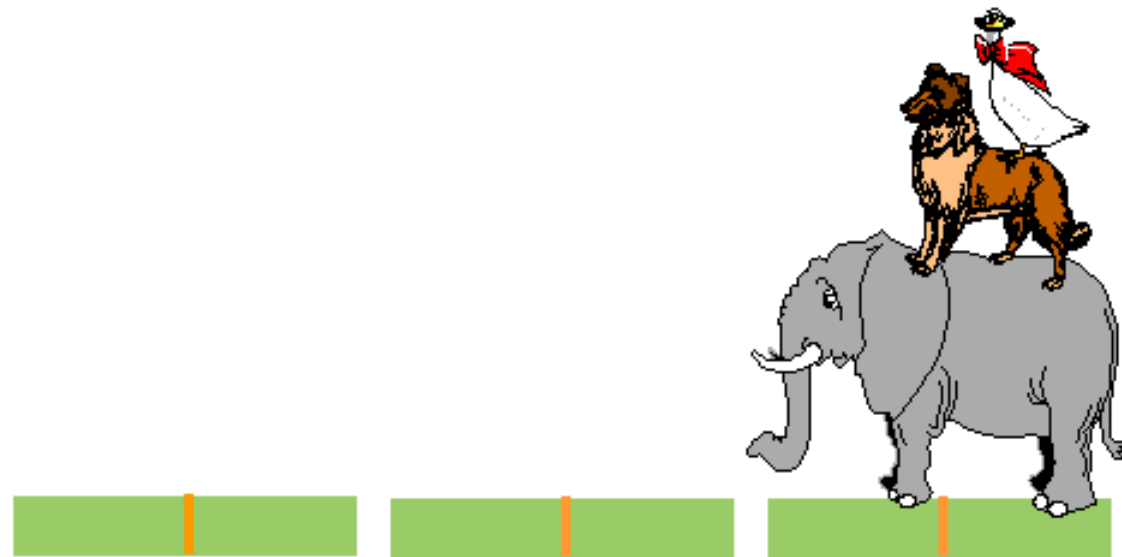
The Towers of Hanoi

A Stack-based Application

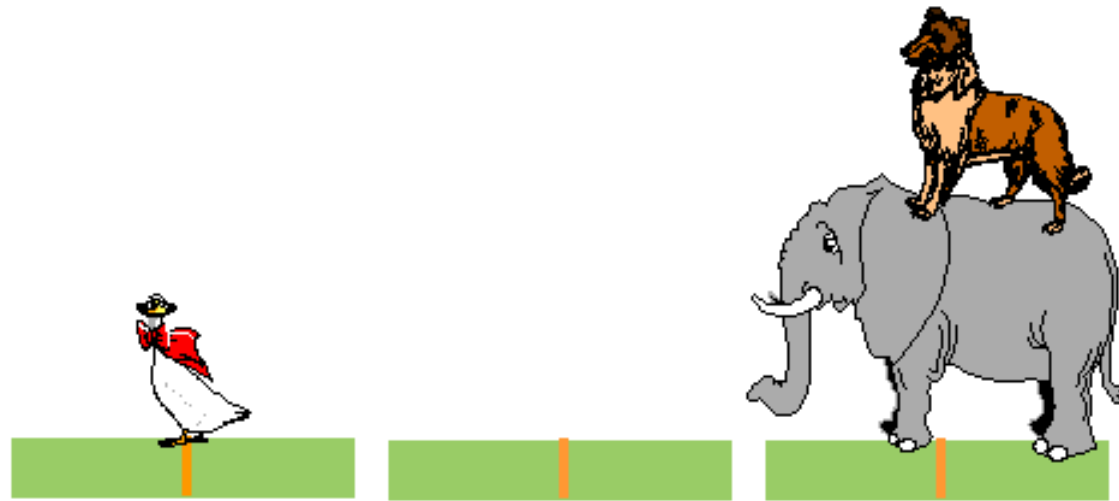
- GIVEN: three poles
- a set of discs on the first pole, discs of different sizes, the smallest discs at the top
- GOAL: move all the discs from the left pole to the right one.
- CONDITIONS: only one disc may be moved at a time.
- A disc can be placed either on an empty pole or on top of a larger disc.



Towers of Hanoi



Towers of Hanoi



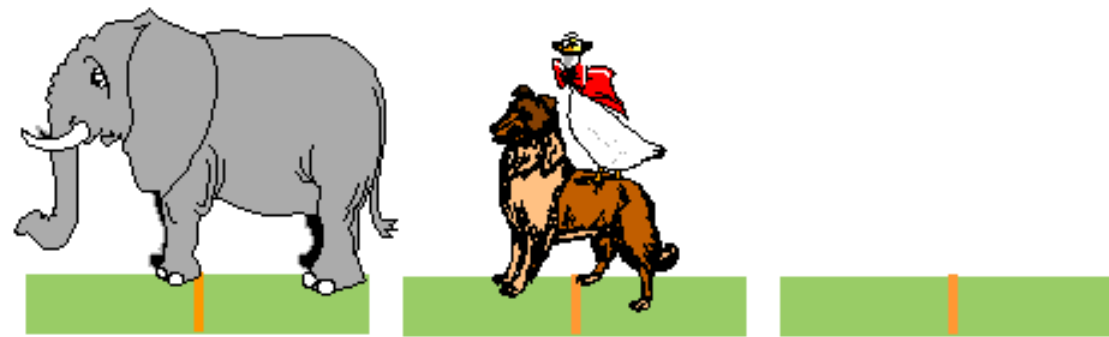
Towers of Hanoi



Towers of Hanoi



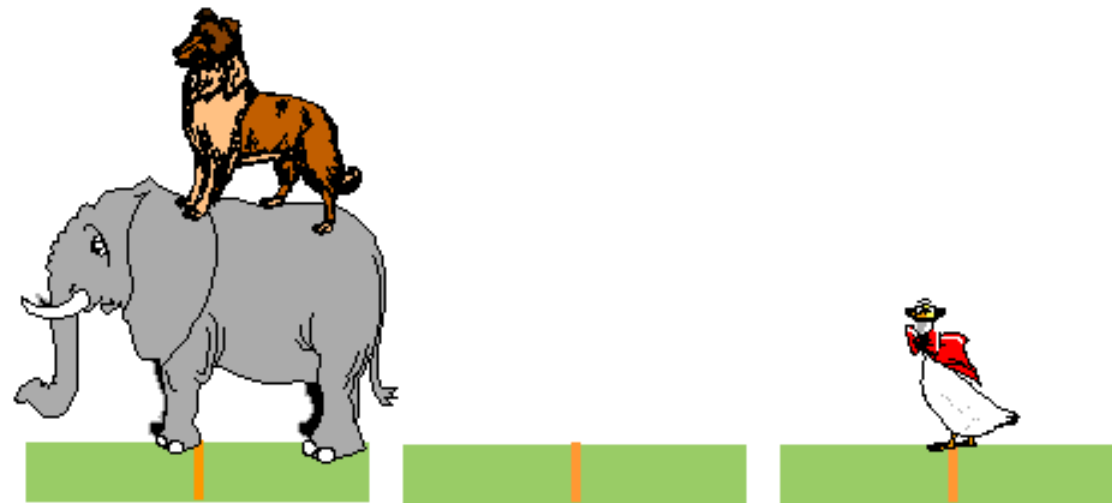
Towers of Hanoi



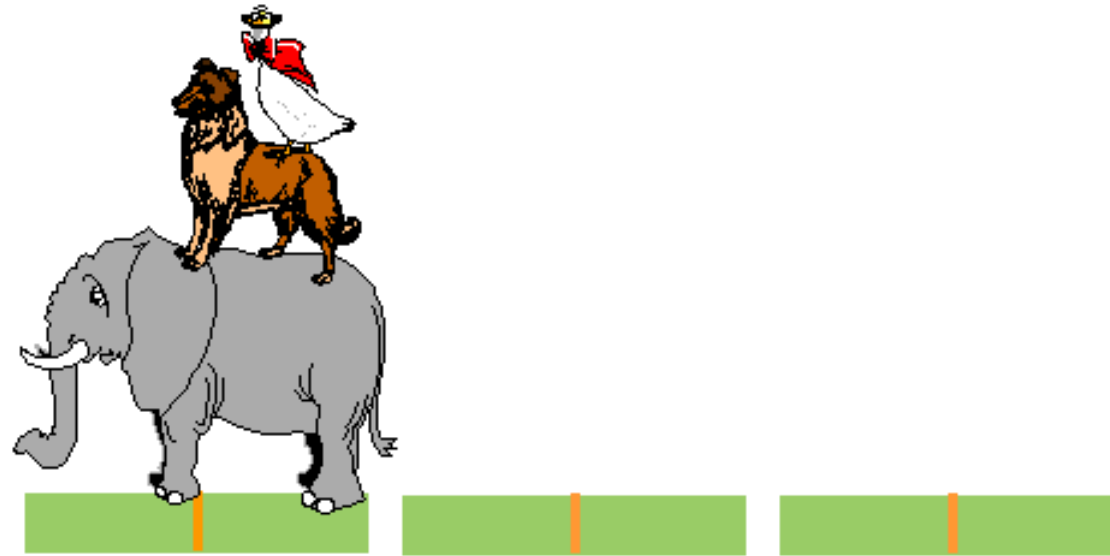
Towers of Hanoi



Towers of Hanoi



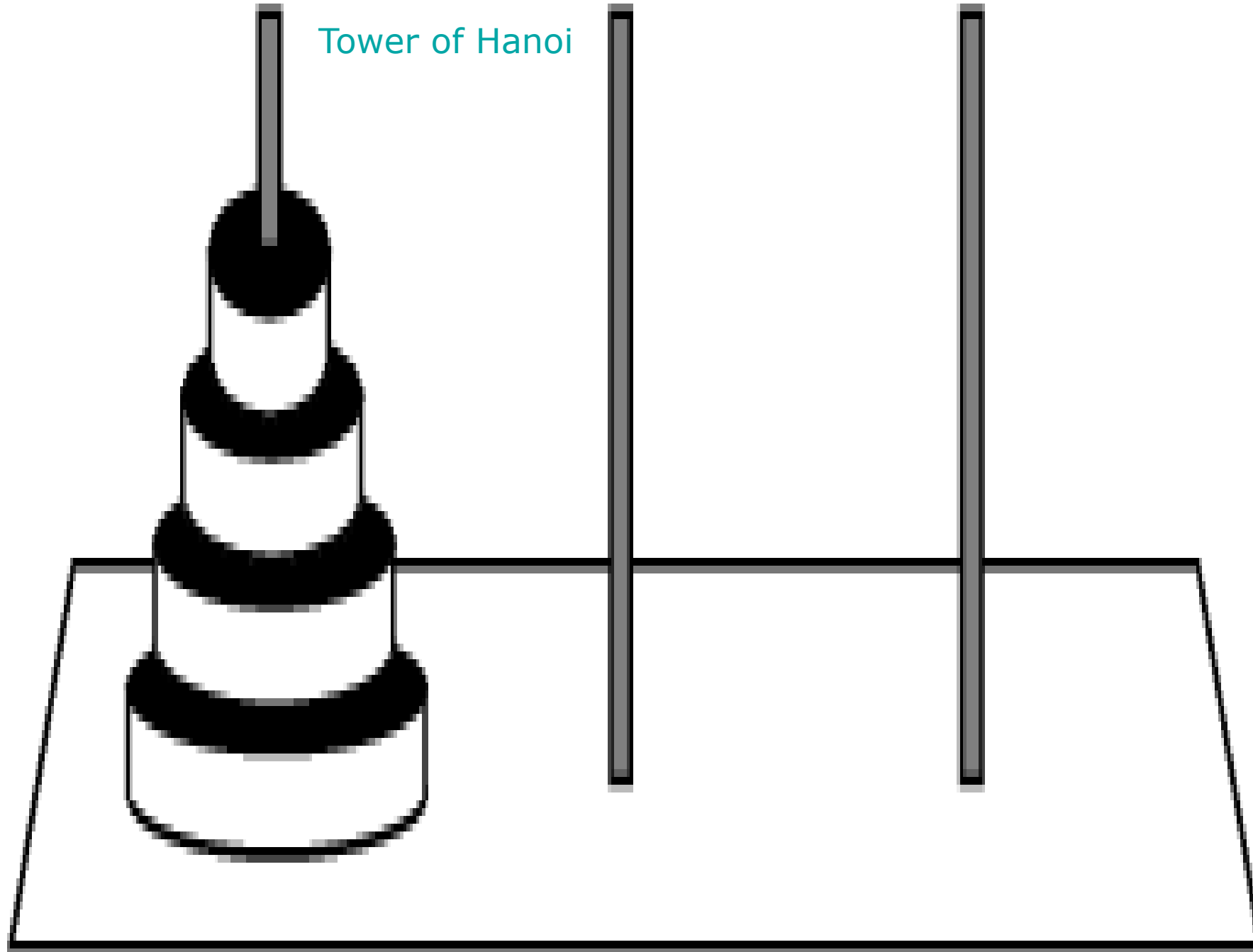
Towers of Hanoi



Towers of Hanoi – Recursive Solution

```
void hanoi (int discs,  
            Stack fromPole,  
            Stack toPole,  
            Stack aux) {  
    Disc d;  
    if( discs >= 1) {  
        hanoi(discs-1, fromPole, aux, toPole);  
        d = fromPole.pop();  
        toPole.push(d);  
        hanoi(discs-1,aux, toPole, fromPole);  
    }
```

Tower of Hanoi

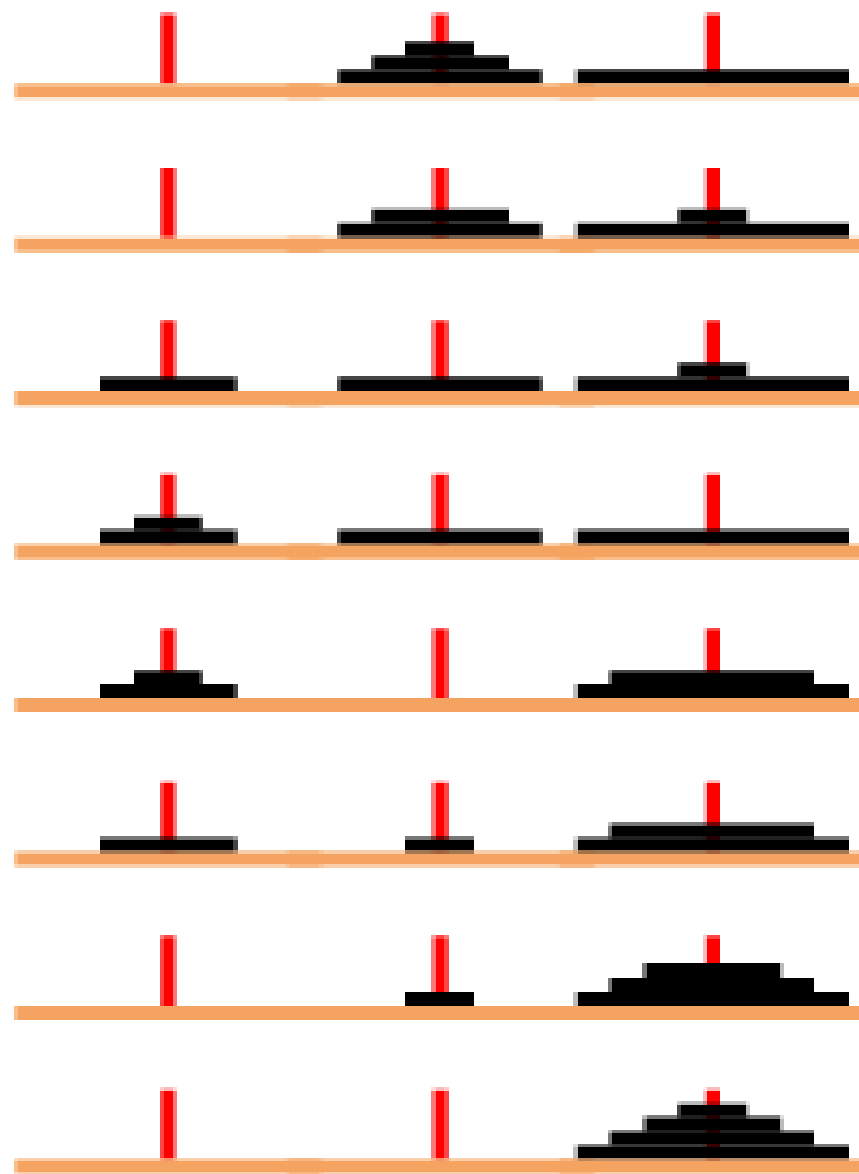
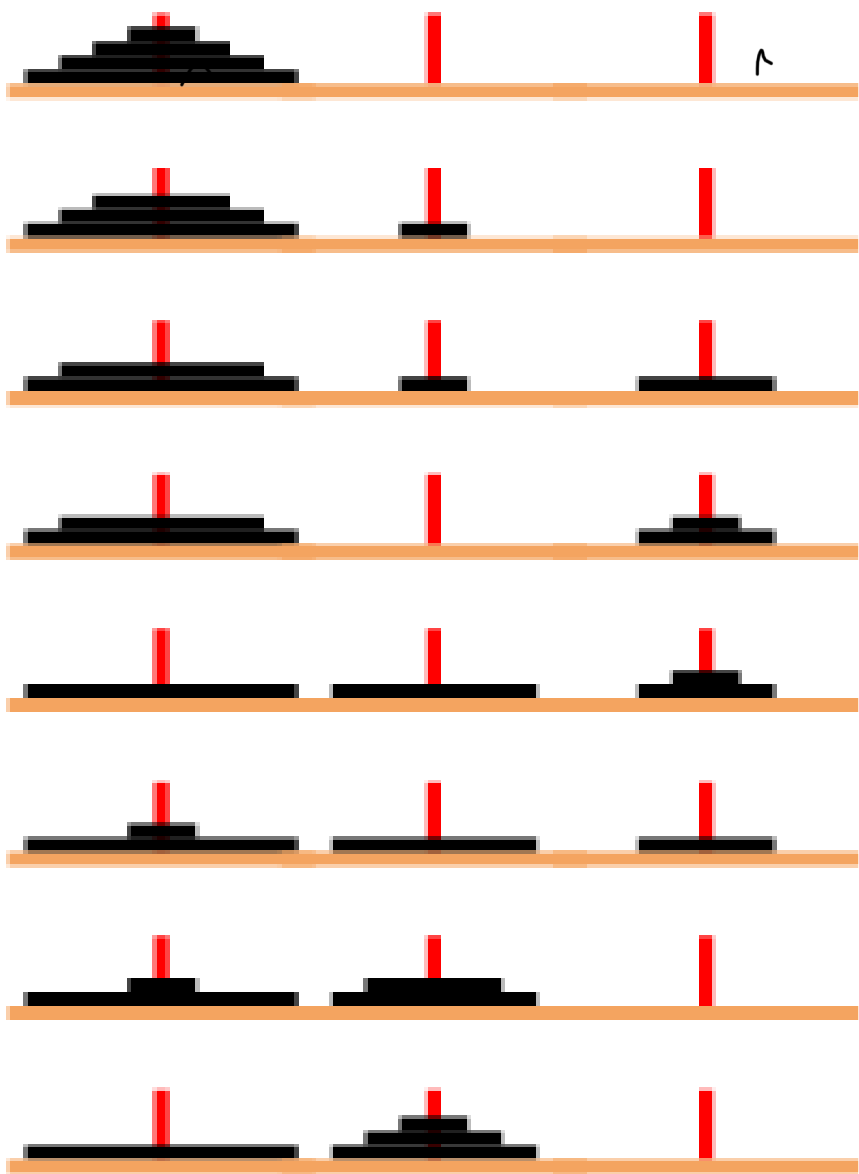


The towers of Hanoi Problem

- Given a stack of disks arranged from largest on the bottom to smallest on top placed on a rod, together with two empty rods
- the towers of Hanoi puzzle asks for the minimum number of moves required to move the stack from one rod to another, where moves are allowed only if they place smaller disks on top of larger disks.

The towers of Hanoi Problem

- <http://www.youtube.com/watch?v=aGlt2G-DC8c>
- [Animation + the towers of Hanoi](#)
- <http://www.cut-the-knot.org/recurrence/hanoi.shtml>



The towers of Hanoi Problem –recursive

Definition

- **Recursive solution**
- There are three pegs Src (Source), Aux (Auxiliary) and Dst (Destination).
- To better understand and appreciate the following solution you should try solving the puzzle for small number of disks, say, 2,3, and, perhaps, 4.
- However one solves the problem, sooner or later the bottom disk will have to be moved from Src to Dst. At this point in time all the remaining disks will have to be stacked in decreasing size order on Aux. After moving the bottom disk from Src to Dst these disks will have to be moved from Aux to Dst.
- Therefore, for a given number N of disks, the problem appears to be solved if we know how to accomplish the following tasks

The Towers of Hanoi Problem –recursive Definition

- Therefore, for a given number N of disks, the problem appears to be solved if we know how to accomplish the following tasks:
 1. Move the top $N - 1$ disks from Src to Aux (using Dst as an intermediary peg)
 2. Move the bottom disk from Src to Dst
 3. Move $N - 1$ disks from Aux to Dst (using Src as an intermediary peg)

The towers of Hanoi Problem –recursive Definition

Solve(N, Src, Aux, Dst)

if N is 0 exit

else

Solve(N - 1, Src, Dst, Aux)

Move from Src to Dst

Solve(N - 1, Aux, Src, Dst)