

# Programming and Data Structures - I

## Lecture 11

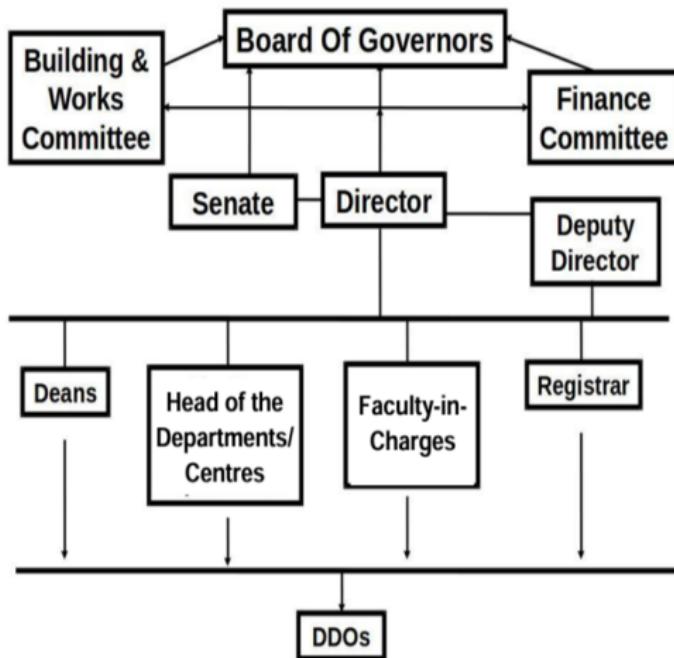
**Kripabandhu Ghosh**

CDS, IISER Kolkata

## USER-DEFINED FUNCTIONS



# IISER Kolkata: Delegation of Authority



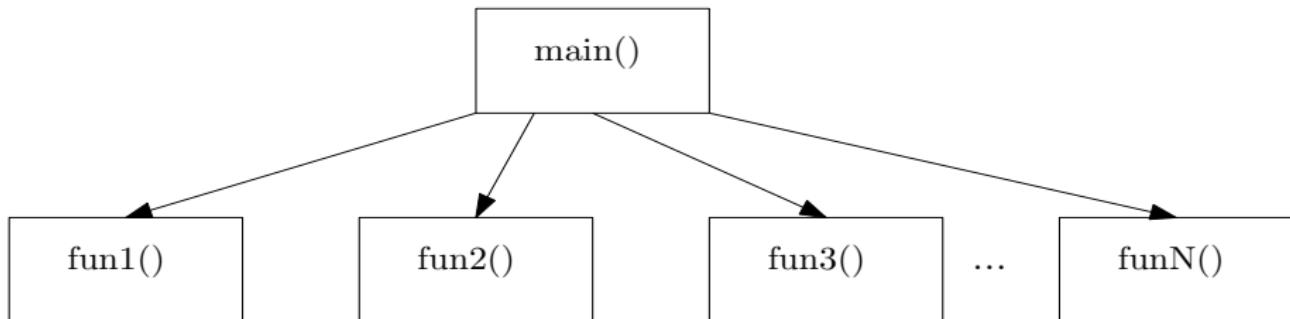
# User-defined functions

## Format

return-type function-name(argument-list)

- **return-type:** Data type returned by the function; default is integer ; can also be void (nothing returned)
  - **argument-list:** datatype arg1, datatype arg2,...
- 
- A function may nor may not have any arguments
  - A function may or may not return any value (maximum one value)

# Functions: utility



## Utilities

- Top-down modular programming
- **Decomposition** of a bigger problem into smaller ones, and **recomposition** from the smaller solutions
- Simplification of a complex logic
- Easier debugging

# Arguments and return values

## Program

```
#include <stdio.h>
void fun1() //No arguments, no return values
{
    printf("Fun1: No arguments, no return values!\n");
}
void fun2(int n) //Has argument(s) but no return values
{
    int i;
    for(i = 1 ; i<=n ; i++)
    {
        printf("Fun2: Has argument(s), but no return values! - %d\n\n", i);
    }
}
int fun3() //No argument(s) but has a returned value
{
    static int a = 0;
    a++;
    printf("Fun3: No argument(s), but has a return value!\n");
    return a;
}
```

# Arguments and return values (contd.)

## Program

```
int fun4(int n) //Has both argument(s) and a return value
{
    printf("Fun4: Has both argument(s) and a return value!\n");
    return (n + 1);
}

int main()
{
    int n = 4, i;
    fun1();
    fun2(n);
    printf("value = %d\n", fun3());
    printf("value = %d\n", fun4(n));
return 0;
}
```

# Arguments and return values (contd.)

## Output

Fun1: No arguments, no return values!

Fun2: Has argument(s), but no return values! - 1

Fun2: Has argument(s), but no return values! - 2

Fun2: Has argument(s), but no return values! - 3

Fun2: Has argument(s), but no return values! - 4

Fun3: No argument(s), but has a return value!

value = 1

Fun4: Has both argument(s) and a return value!

value = 5

# Modular programming

## Program

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

int factorial(int n) //Calculates factorial
{
    int i, fact = 1;
    for(i = 2 ; i <= n ; i++)
    {
        fact = i*fact;
    }
    return fact;
}

float nCr(int n, int r) //Calculates nCr
{
    float comb;
    comb = factorial(n)/(1.0*factorial(r)*factorial(n-r));
    return comb;
}
```

# Modular programming (contd.)

## Program

```
float binomialExpansion(int a, int b, int n) //Calculates (a + b)^n using  
Binomial Expansion  
{  
    int r;  
    float sum = 0;  
    for(r = 0 ; r <= n ; r++)  
    {  
        sum += nCr(n, r)*pow(a, n-r)*pow(b, r);  
    }  
    return sum;  
}  
  
float powerOfSum(int a, int b, int n) //Explicitly calculates (a + b)^n  
{  
    return (pow(a + b, n));  
}
```

# Modular programming (contd.)

## Program

```
int main()
{
    int a = 4, b = 6, n = 3, r;
    printf("Binomial expansion for (%d + %d)^%d = %f\n", a, b, n,
binomialExpansion(a, b, n));
    printf("Explicit calculation for (%d + %d)^%d = %f\n", a, b, n,
powerOfSum(a, b, n));
    return 0;
}
```

## Output

Binomial expansion for  $(4 + 6)^3 = 1000.000000$   
Explicit calculation for  $(4 + 6)^3 = 1000.000000$

# Stack

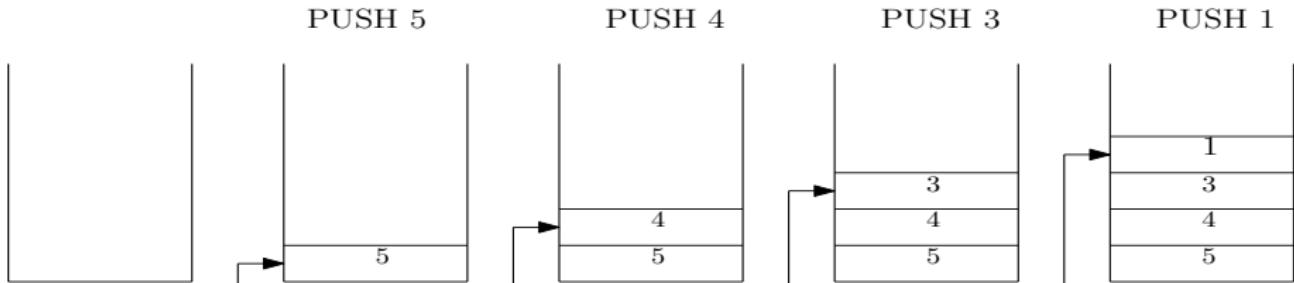
## Features

- Dynamic data structure (Abstract Data Type)
- Insertion and deletion at **one end**
- Deletion based on **Last In First Out (LIFO)** policy
- Insertion called **Push** operation
- Deletion called **Pop** operation
- Usually **Top** attribute is used to point to the last inserted element

## Applications

- Function (subroutine) call and return
- Backtracking
- Syntax analysis

# Stack example



EMPTY

TOP

TOP

TOP

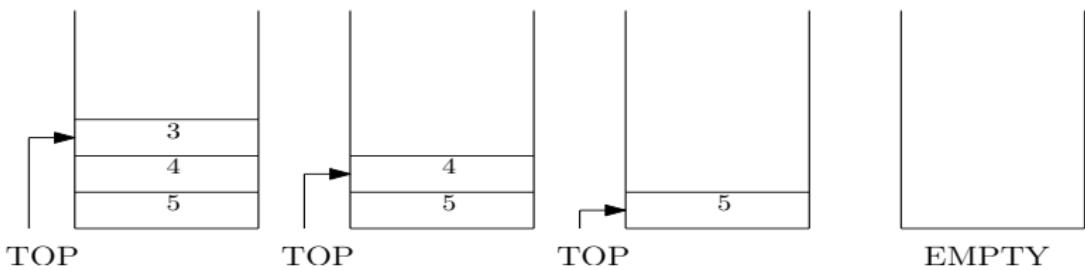
TOP

POP  
1

POP  
3

POP  
4

POP  
5



TOP

TOP

TOP

EMPTY

# Origin of Stack

- Jan Lukasiewicz's Polish notation possibly lead to the idea of stack
- Alan Turing used it for subroutines (functions) in 1946
- Konrad Zuse implemented stack in Z4 (computer) in 1945

# Stack in function call

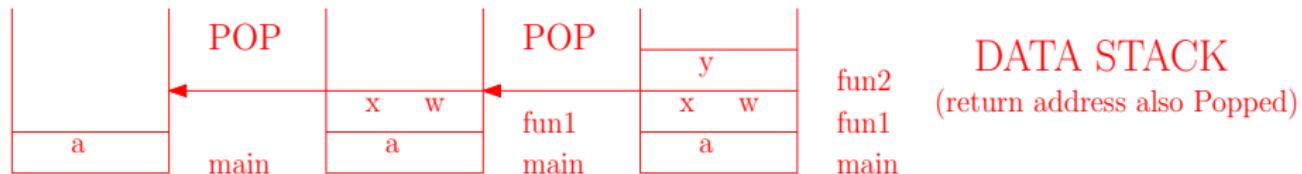
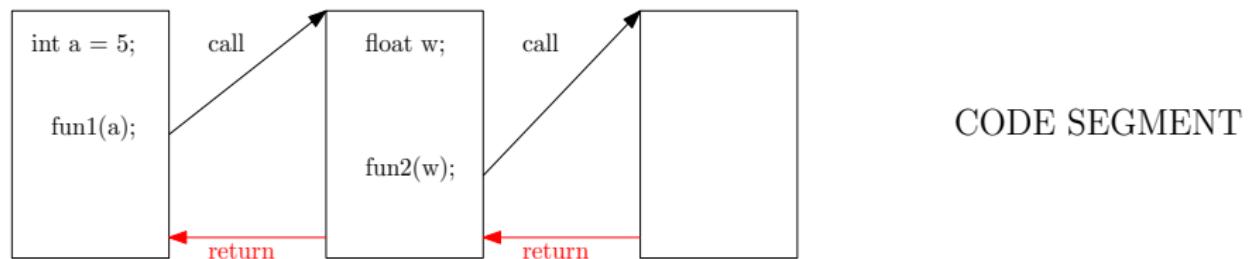
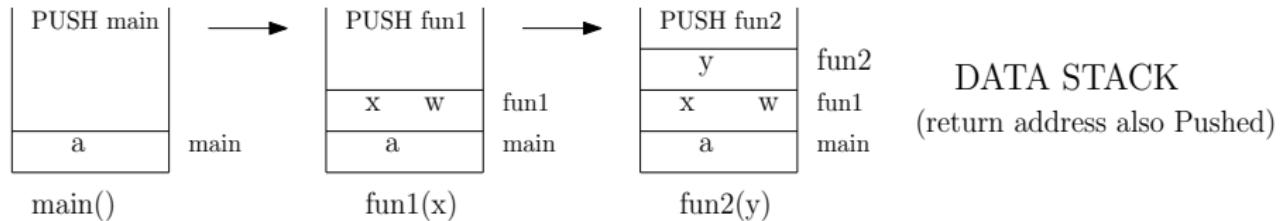
## Function elements

- **Code segment:** Machine readable code generated upon the compilation
- **Data segment:** Local variables for a function created upon the compilation

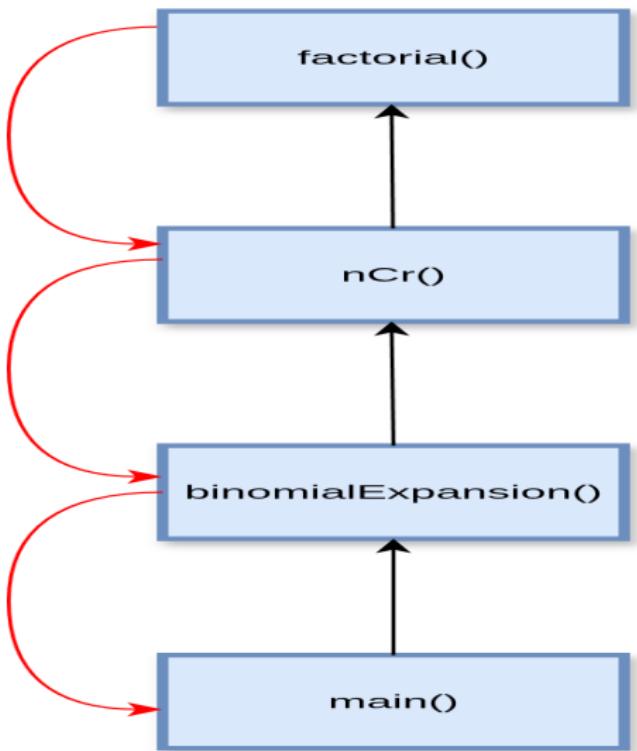
## Premise of a function call

- Local data stored in a stack (retrieved on return from the call)
- Return address of the point of return from the function call also stored (retrieved to enable return from the call)

# Stack in function call (illustration)



# Stack in function call (binomial)



# Stack trace

```
> gcc -g -o binomial_expansion binomial_expansion.c -lm  
> gdb ./binomial_expansion
```

```
(gdb) b 6  
Breakpoint 1 at 0x1174: file binomial_expansion.c, line 6.  
(gdb) r  
Starting program: /home/kripa/kripa/programs/C/binomial_expansion
```

```
Breakpoint 1, factorial (n=3) at binomial_expansion.c:6  
6          int i, fact = 1;  
(gdb) bt  
#0  factorial (n=3) at binomial_expansion.c:6  
#1  0x0000555555551bb in nCr (n=3, r=0) at binomial_expansion.c:21  
#2  0x00005555555524f in binomialExpansion (a=4, b=6, n=3) at binomial_expansion.c:33  
#3  0x000055555555365 in main () at binomial_expansion.c:47
```

Figure: Setting breakpoint and printing the stack trace

## Customized headers: mathfunc.h

```
1 // Function declarations
2 int add(int a, int b);
3 int subtract(int a, int b);
4 int multiply(int a, int b);
5 float divide(int a, int b);
```

## Customized headers: mathfunc.c

```
1 #include "mathfunc.h"
2
3 int add(int a, int b) {
4     return a + b;
5 }
6
7 int subtract(int a, int b) {
8     return a - b;
9 }
10
11 int multiply(int a, int b) {
12     return a * b;
13 }
14
15 float divide(int a, int b) {
16     if (b != 0)
17         return (float)a / b;
18     else
19         return 0; // Error: divide by zero
20 }
```

# Customized headers: main program (myprog\_withmyheader.c)

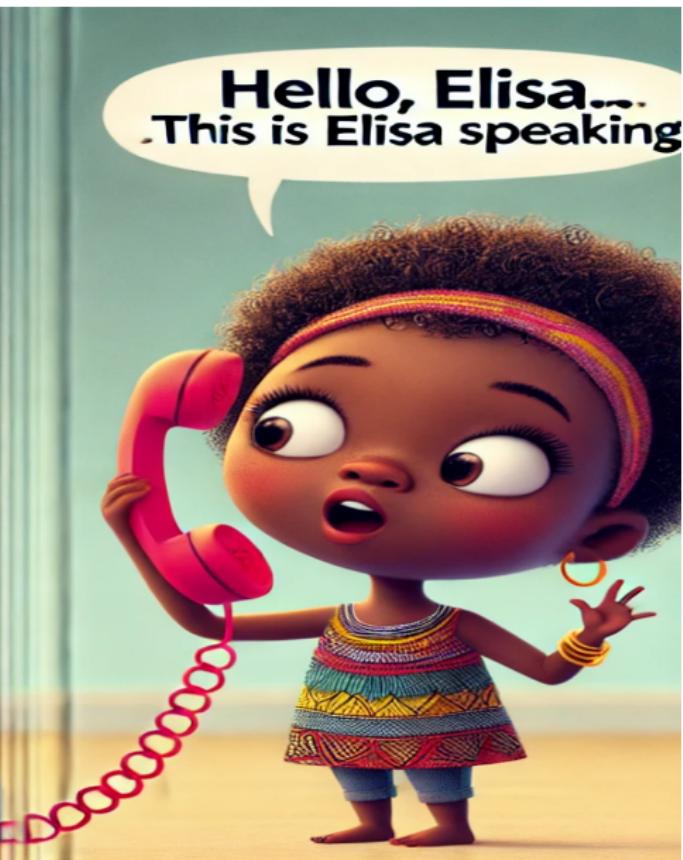
```
1 #include <stdio.h>
2 #include "mathfunc.h"      // My header
3
4 int main() {
5     printf("Add: %d\n", add(5, 3));
6     printf("Divide: %.2f\n", divide(10, 2));
7     return 0;
8 }
```

```
> gcc -o myprogout myprog_withmyheader.c mathfunc.c
```

## OUTPUT

Add: 8  
Divide: 5.00

# RECURSION



# Recursion

- Function calls itself
- Solving a large (input size) problem by solving smaller related sub-problems

## Conditions

- Base condition (terminating condition)
- Recursive/inductive condition

# Recursive definition of factorial

## Function elements

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot \text{factorial}(n - 1) & \text{if } n > 1 \end{cases}$$

# Factorial using recursion

```
factorial(n) = n x factorial(n - 1)
              = n x (n - 1) x factorial(n - 2)
              ...
              = n x (n - 1) x ... x factorial(1)
              = n x (n - 1) x ... x 1
```

n = 4

```
factorial(4) = 4 x factorial(3)
              = 4 x 3 x factorial(2)
              = 4 x 3 x 2 x factorial(1)
              = 4 x 3 x 2 x 1
              = 24
```

# Factorial using recursion (program)

## Program

```
#include <stdio.h>

int fact(int n) //Iterative version
{
    int i, f = 1;
    for(i = 1 ; i <= n; i++)
    {
        f = f*i;
    }
    return f;
}

int fact_recursive(n) //Recursive version
{
    int x;
    if(n == 1)
    {
        printf("Base condition return: n = %d\n", n);
        return 1;
    }
    else
    {
        printf("n = %d: before recursive call!\n", n);
        x = n*fact_recursive(n-1);
        printf("n = %d: after returning from call!\n", n);
        printf("Current value of factorial (x) = %d\n", x);
        return x;
    }
}
```

# Factorial using recursion (program) (contd.)

## Program

```
int main()
{
    int n = 4, fct, fct_rec;
    printf("Call from main\n");
    fct = fact(n);
    fct_rec = fact_recursive(n);
    printf("Return to main\n");
    printf("Factorial(%d) = %d (recursive)\n", n, fct_rec);
    printf("Factorial(%d) = %d (iterative)\n", n, fct);
    return 0;
}
```

# Factorial using recursion (program) (contd.)

## Output

Call from main

n = 4: before recursive call!

n = 3: before recursive call!

n = 2: before recursive call!

Base condition return: n = 1

n = 2: after returning from call!

Current value of factorial (x) = 2

n = 3: after returning from call!

Current value of factorial (x) = 6

n = 4: after returning from call!

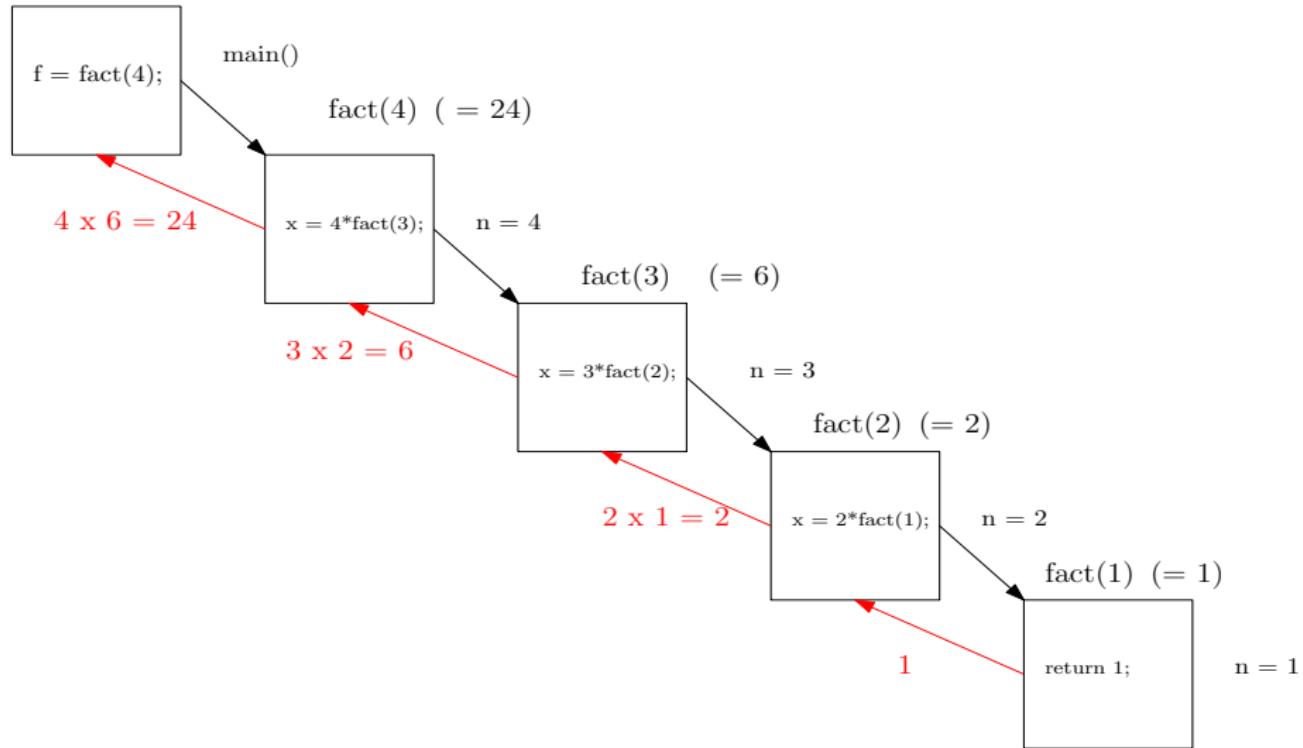
Current value of factorial (x) = 24

Return to main

Factorial(4) = 24 (recursive)

Factorial(4) = 24 (iterative)

# Factorial using recursion (illustration)



# Factorial: decomposition and recomposition

Base condition:  $n = 1$

if  $n == 1$ , return value is 1

Inductive condition:  $n > 1$

- Decomposition: `fact_recursive(n - 1)`
- Recomposition:  $x = n * \text{fact\_recursive}(n - 1)$

# Recursive definition of Fibonacci series

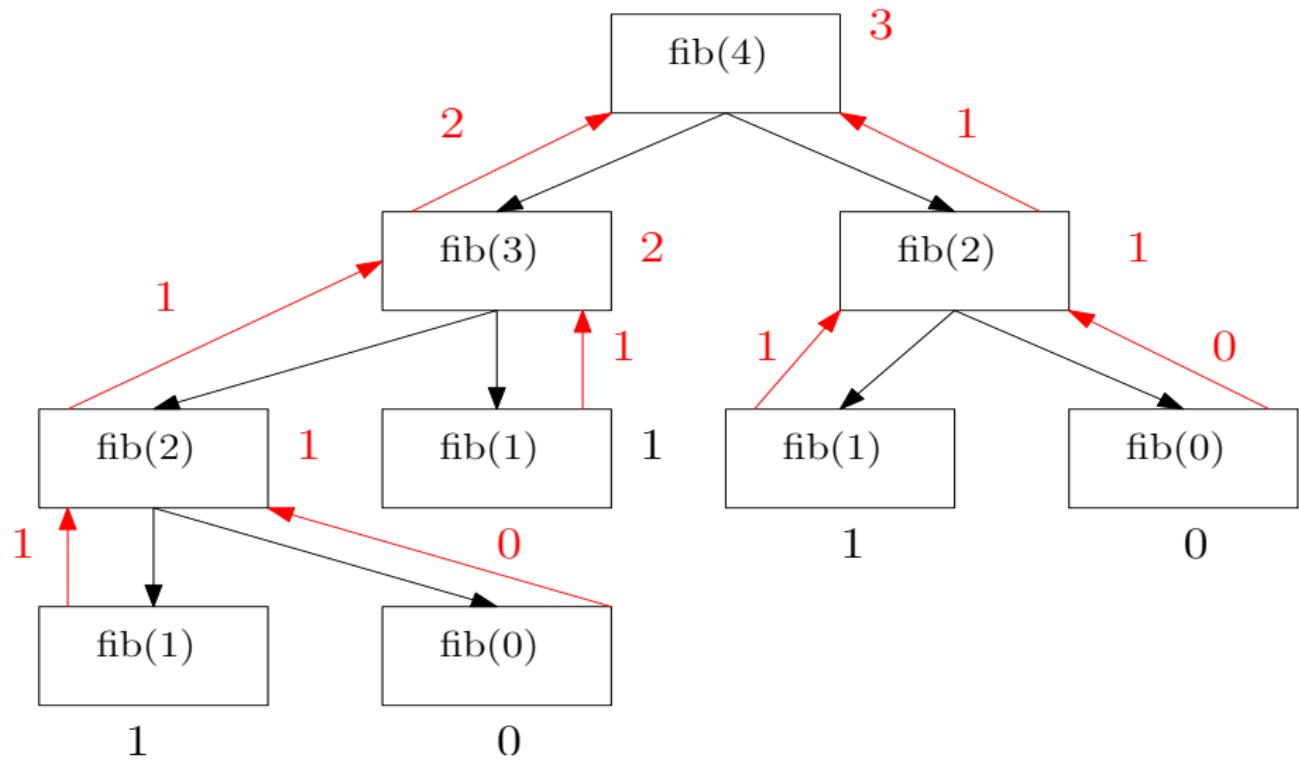
## Function elements

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

## Series

0 1 1 2 3 5 ...

# Fibonacci series using recursion (illustration)



# Fibonacci series using recursion (program)

## Program

```
#include <stdio.h>

void fib(int n) //Iterative version
{
    int f0 = 0, f1 = 1, f, i;
    printf("%d %d ", f0, f1);
    for(i = 2 ; i <= n; i++)
    {
        f = f0 + f1;
        printf("%d ", f);
        f0 = f1;
        f1 = f;
    }
    printf("\n");
}

int fib_recursive(n) //Recursive version
{
    int x;
    if(n == 0)
    {
        return 0;
    }
    else
    {
        if(n == 1)
        {
            return 1;
        }
        else
        {
            x = fib_recursive(n-1) + fib_recursive(n - 2);
            return x;
        }
    }
}
```

# Fibonacci series using recursion (program) (contd.)

## Program

```
int main()
{
    int n = 4, i;
    printf("Recursive: \n");
    for(i = 0 ; i <= n ; i++)
    {
        printf("%d ", fib_recursive(i));
    }
    printf("\n");
    printf("Iterative: \n");
    fib(n);
    return 0;
}
```

## Output

Recursive:

0 1 1 2 3

Iterative:

# GCD: Euclid's algorithm

## Algorithm: GCD( $m, n$ )

- **Step 1:** Divide  $m$  by  $n$  and let  $r$  be the remainder
- **Step 2:** If  $r = 0$ , end;  $n$  is the answer.
- **Step 3:** Let  $m = n$ ,  $n = r$  and go to Step 1.

# GCD: Euclid's algorithm

## Iterative implementation

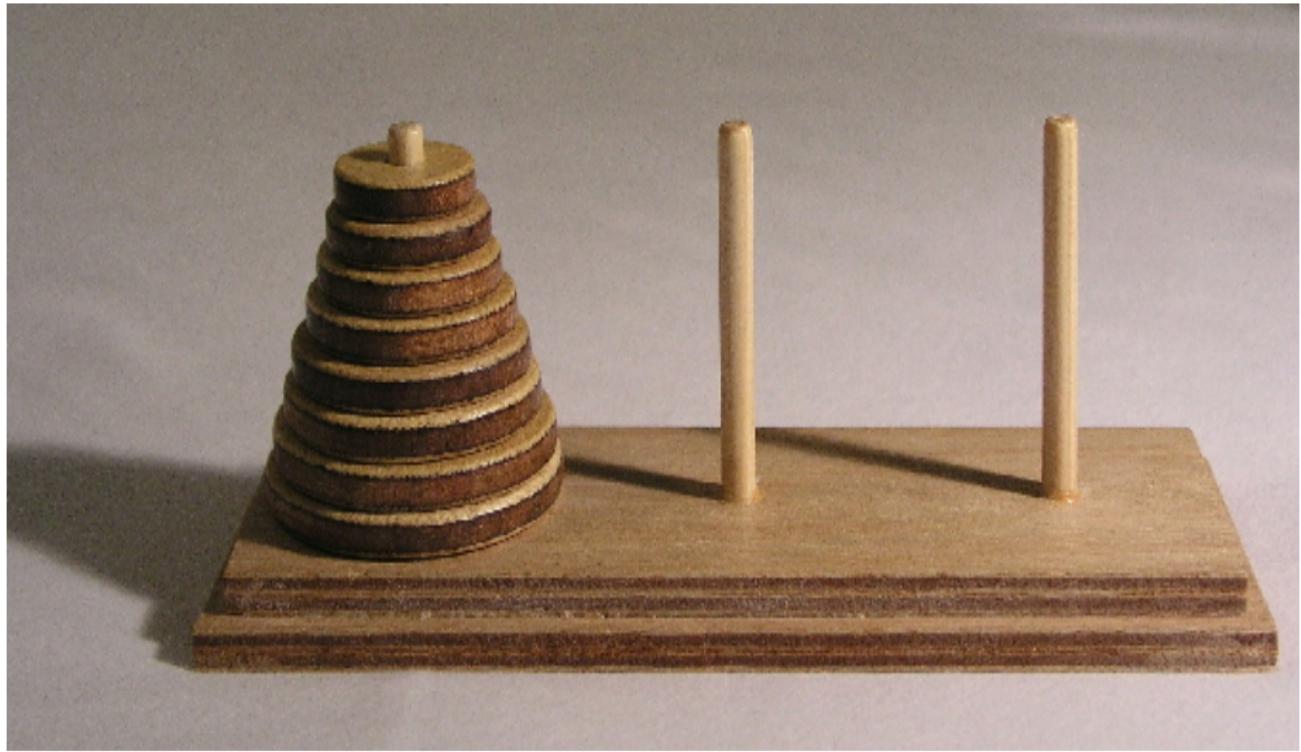
```
int gcd(int a, int b)
{
    int rem;
    while(b != 0) //Step 2
    {
        rem = a%b;//Step 1
        a = b;//Step 3
        b = rem; //Step 3
    }
    return a;
}
```

# GCD: Euclid's algorithm

## Recursive implementation

```
int gcd_rec(int a, int b)
{
    if(b == 0) //Step 2
    {
        return a;
    }
    return gcd_rec(b, a%b); //Step 1, 3
}
```

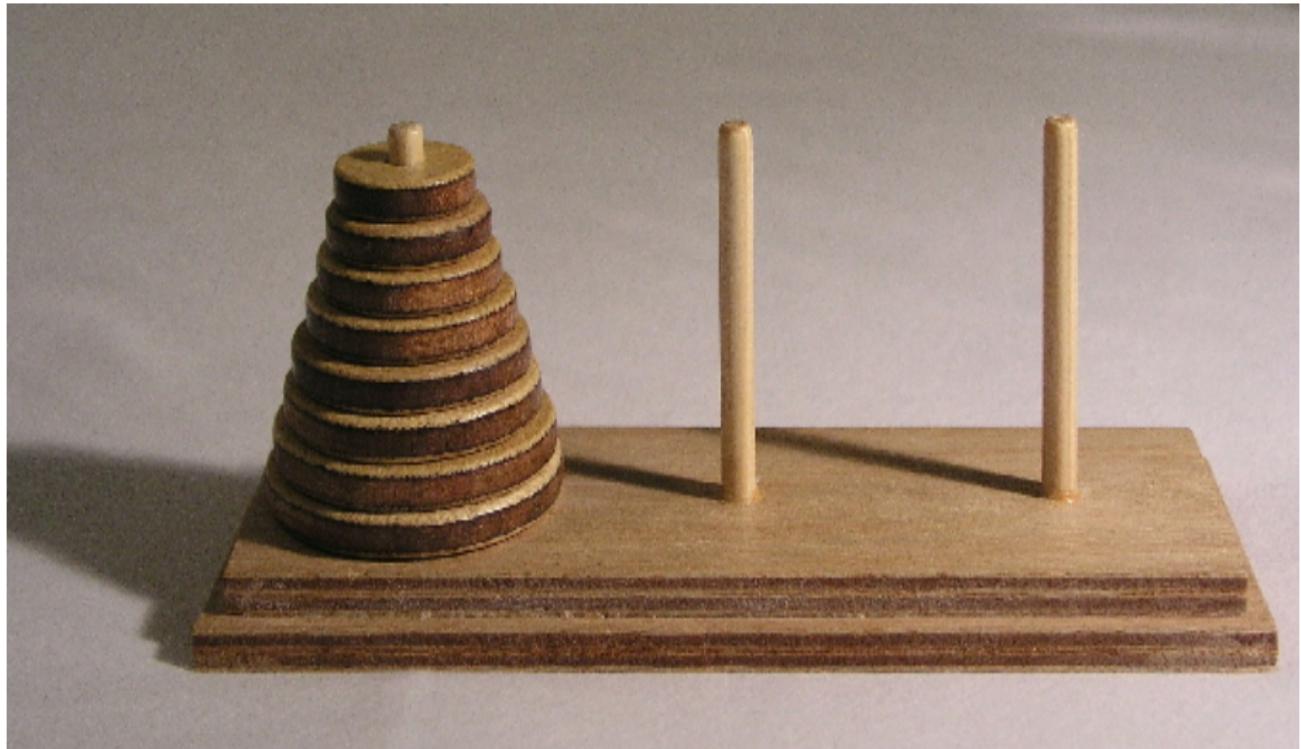
# Tower Of Hanoi (Tower of Brahma)<sup>1</sup>



---

<sup>1</sup>Source: Wikipedia

# Tower Of Hanoi (Tower of Brahma)<sup>2</sup>



---

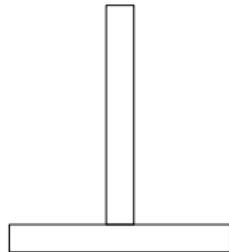
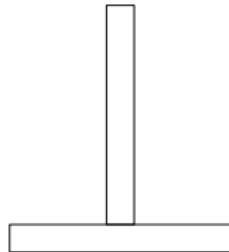
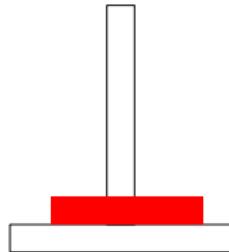
<sup>2</sup>Source: Wikipedia

# Tower Of Hanoi

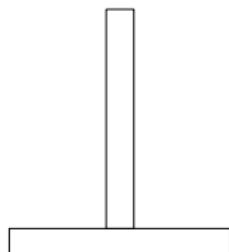
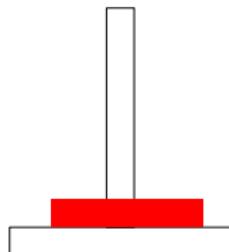
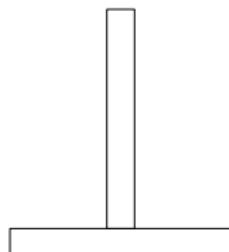
## Rules

- Only one disk can be moved at a time
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod
- No larger disk may be placed on top of a smaller disk

# Tower of Hanoi: n = 1 (illustration)



$n = 1$

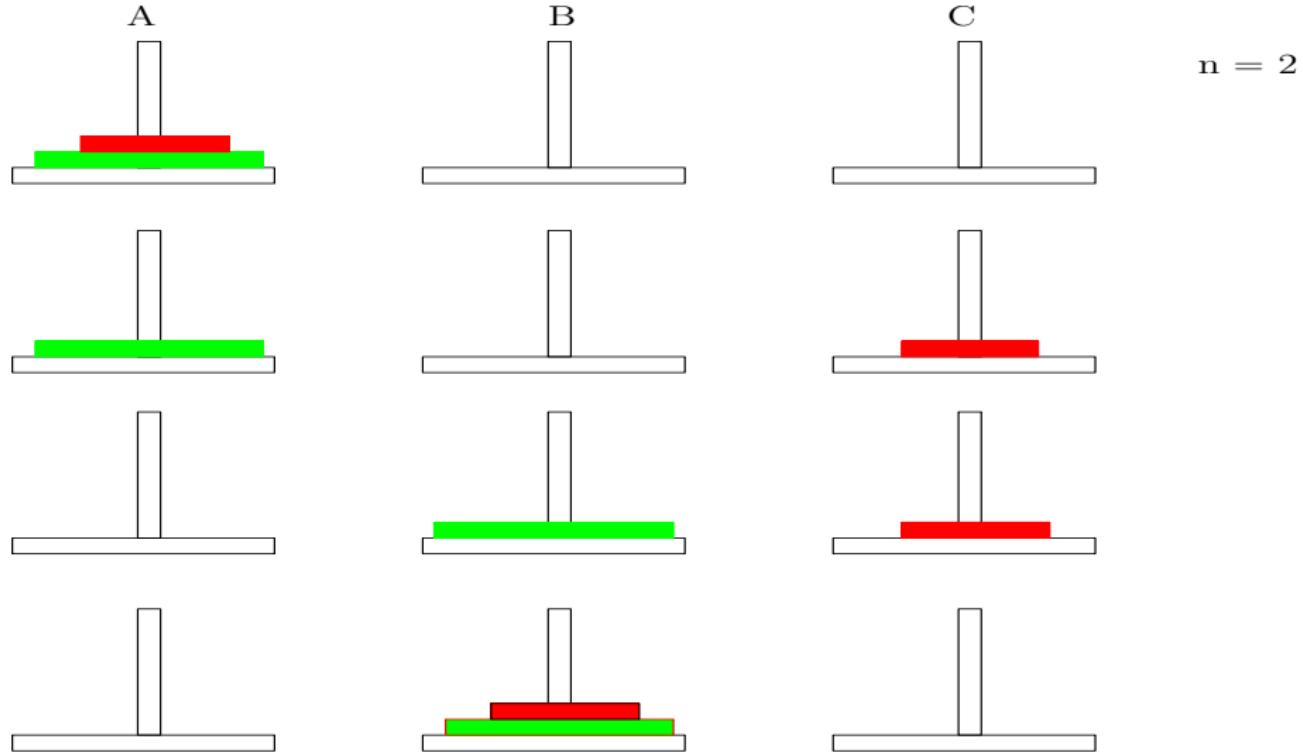


A

B

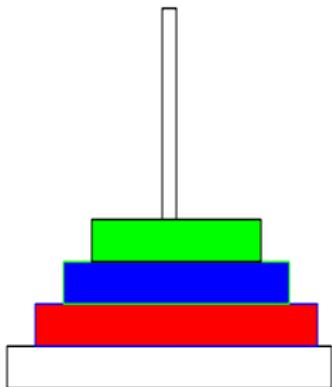
C

# Tower of Hanoi: $n = 2$ (illustration)

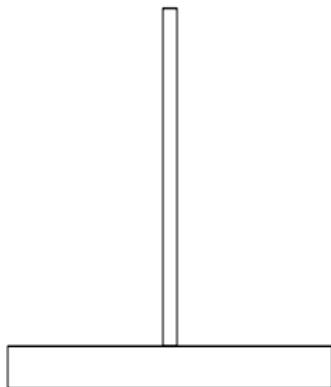


# Tower of Hanoi: n = 3 (illustration): zero step

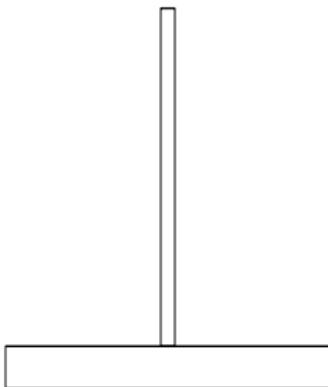
A



B

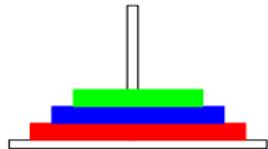


C



# Tower of Hanoi: $n = 3$ (illustration)

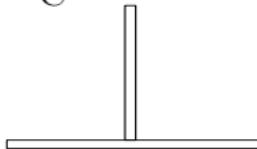
A



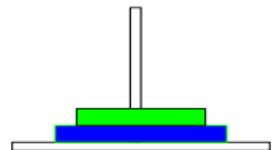
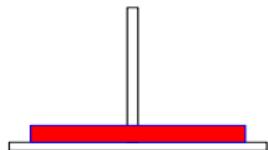
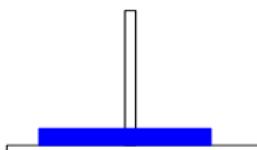
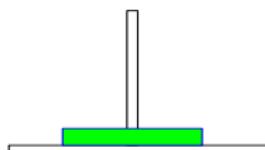
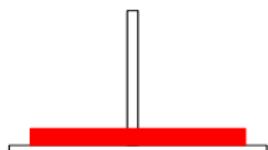
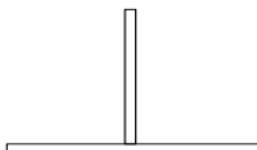
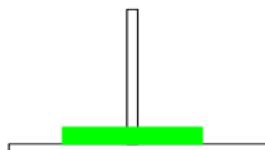
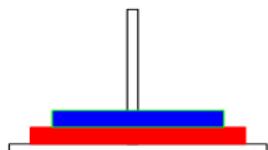
B



C



$n = 3$



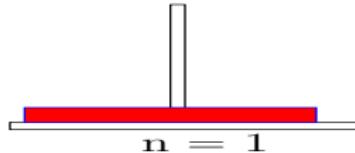
$n = 2$

# Tower of Hanoi: $n = 3$ (contd.) (illustration)

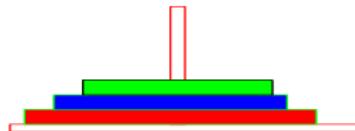
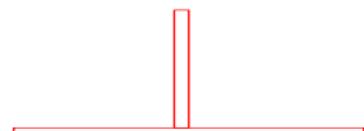
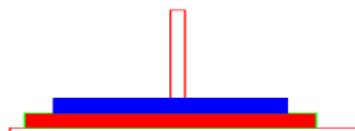
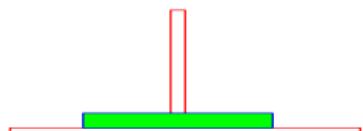
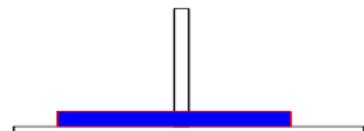
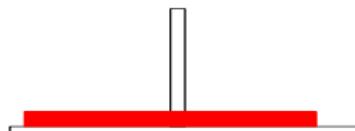
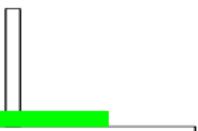
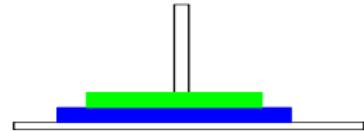
A



B



C



# Tower Of Hanoi: formal solution

Format

TOWERS( $n$ , from, to, via)

Problem

TOWERS( $n$ , A, B, C)

# Tower Of Hanoi: formal solution (contd.)

## Base condition

( $n = 1$ ): TOWERS(1, A, B, C)

## Inductive condition: Decomposition

- $L1 = \text{TOWERS}(n - 1, A, C, B)$
- $L2 = \text{TOWERS}(1, A, B, C)$
- $L3 = \text{TOWERS}(n - 1, C, B, A)$

## Inductive condition: Recomposition

$L = \text{APPEND}(L1, L2, L3)$

# Tower Of Hanoi: implementation

## Program

```
#include<stdio.h>
void tofh(int ndisk, char source, char temp, char dest);
int main(void)
{
    char source = 'A', temp = 'C', dest = 'B';
    int ndisk;
    printf("Enter the number of disks : ");
    scanf("%d", &ndisk );
    printf("Sequence is :\n");
    tofh(ndisk, source, dest, temp);
    return 0;
}
```

# Tower Of Hanoi: implementation (contd.)

## Program

```
void tofh(int ndisk, char source, char dest, char temp)
{
    if(ndisk==1)
    {
        printf("Move Disk %d from %c->%c\n", ndisk, source, dest);
        return;
    }
    tofh(ndisk-1, source, temp, dest);
    printf("Move Disk %d from %c->%c\n", ndisk, source, dest);
    tofh(ndisk-1, temp, dest, source);
}
```

# Tower Of Hanoi: implementation (contd.)

## Output

Enter the number of disks : 3

Sequence is :

Move Disk 1 from A—>B

Move Disk 2 from A—>C

Move Disk 1 from B—>C

Move Disk 3 from A—>B

Move Disk 1 from C—>A

Move Disk 2 from C—>B

Move Disk 1 from A—>B



**THANK  
YOU!**