FUNCTIONS

- Concept of function, using functions, call by value and call by reference mechanism to working with functions-example programs, passing arrays to functions, scope and extent, storage classes, recursion.

Recursion

A function calling itself is known as **Recursion.** But while using recursion programmers need to be careful to define an exit condition from the function, otherwise it will go in infinite loop.

Example:

```
void recursion()
{
        recursion(); /* function calling itself */
}
```

- ✓ In the above example we have not return the exit condition, so the function is in infinite loop. So, we need to be careful to define an exit function when we are implementing recursive functions.
- ✓ Recursive functions are used to calculate many mathematical functions like factorial of a give number and Fibonacci series.

Example 1:

```
 /******* Factorial of a given number using Recursion ******/
#include<stdio.h>
int fact(int m);
void main()
{
        int n,result;
        printf("enter number:");
        scanf("%d",&n);
        result=fact(n);
        printf("the factorial of a number is:%d",result);
        getch();
}
```

```
int fact(int m)
{
        if(m==1)
                return 1;
        else
                return (m*fact(m-1));
}
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! Or 15! Depending on the system you compile it.

Any recursive function has two elements:

a) Base case

b) General case

Base case:

- ✓ The statement that solves the problem is called base case.
- ✓ Every recursive function must have at least one base case. It is a special case whose solution can be obtained without using recursion.

  A base case serves two purposes:

  i). It act as a terminating condition.

  ii). the recursive function obtains the solution from the base case it reaches.

  In the factorial problem, the base case is 0!=1

$$\boxed{n!=1 \text{ if } n=0}$$

General case:

- ✓ The statement that reduces the size of the problem is called general case. This is done by calling the same function with reduced size.
  - ✓ In the factorial problem, we compute 5! Initial size of the problem is 5 and then the problem is reduced to find 4!, 3!, 2!, 1! and finally we reached the base case 0! Where 0! is 1.

- ✓ 5!= 5 x 4!

In general

$$\boxed{n! = n * (n-1)! \quad \text{If } n \; != 0}$$

In general recursive function of factorial problem can be written as

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n=0 \quad \text{// Base case} \\ n * \text{fact}(n-1)! & \text{Otherwise} \quad \text{// General case} \end{cases}$$

Example 2:

Fibonacci numbers

- ✓ Fibonacci numbers are a series of numbers such that each number is the sum of the two previous numbers except the first and second number.
  - In this problem Base case: 0, 1
  - General case: 1, 2, 3, 5, 8,…..

In general recursive function of Fibonacci problem can be written as

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n=0 \quad \text{// Base case} \\ 1 & \text{if } n=1 \quad \text{// Base case} \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n>2 \quad \text{// General case} \end{cases}$$

Example 3:

Tower of Hanoi:

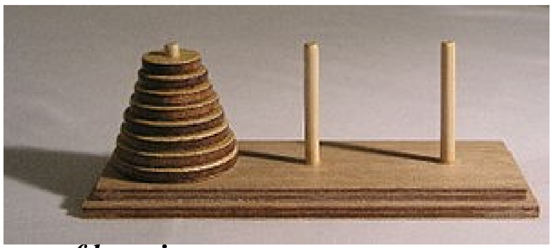
The **Tower of Hanoi** (also called the **Tower of Brahma** or **Lucas' Tower**). It consists of three rods, and a number of disks of different sizes which can slide onto any rod. It starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.

2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

With three disks, the puzzle can be solved in seven moves. The minimum number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.



Recursive solution for tower of hanoi

A key to solving this puzzle is to recognize that it can be solved by breaking the problem down into a collection of smaller problems and further breaking those problems down into even smaller problems until a solution is reached.

For Example:

- label the pegs A, B, C — these labels may move at different steps
- let n be the total number of discs
- number the discs from 1 (smallest, topmost) to n (largest, bottommost)

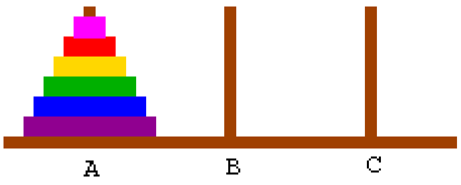
To move n discs from peg A to peg C:

1. move $n-1$ discs from A to B. This leaves disc n alone on peg A
2. move disc n from A to C
3. move $n-1$ discs from B to C so they sit on disc n

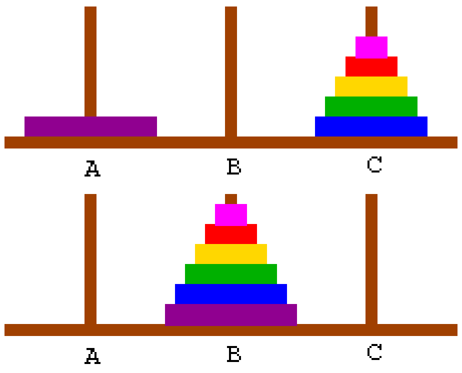
The above is a recursive algorithm, to carry out steps 1 and 3, apply the same algorithm again for $n-1$. The entire procedure is a finite number of steps, since at some point the algorithm will be required for $n = 1$. This step, moving a single disc from peg A to peg B, is trivial. This approach can be given a rigorous mathematical formalism with the theory of dynamic programming and is often used as an example of recursion.

In our Towers of Hanoi solution, we recurse on the largest disk to be moved. That is, we will write a recursive function that takes as a parameter the disk that is the largest disk in the tower we want to move. Our function will also take three parameters indicating from which peg the tower should be moved (*source*), to which peg it should go (*dest*), and the other peg, which we can use temporarily to make this happen (*spare*).

At the top level, we will want to move the entire tower, so we want to move disks 5 and smaller from peg A to peg B. We can break this into three basic steps.



- Move disks 4 and smaller from peg A (*source*) to peg C (*spare*), using peg B (*dest*) as a spare. How do we do this? By recursively using the same procedure. After finishing this, we'll have all the disks smaller than disk 4 on peg C. (Bear with me if this doesn't make sense for the moment - we'll do an example soon.)



In pseudocode, this looks like the following. At the top level, we'll call

```
MoveTower with disk=5, source=A, dest=B, and spare=C.

FUNCTION MoveTower(disk, source, dest, spare):

IF disk == 0, THEN:
        move disk from source to dest
ELSE:
        MoveTower(disk - 1, source, spare, dest)   // Step 1 above
        move disk from source to dest              // Step 2 above
        MoveTower(disk - 1, spare, dest, source)   // Step 3 above
END IF
```

Note:

1. Recursion is not possible, where are called thousands of time.
2. Function can be used in expressions, more than once; provided functions should not have *void* return type.
3. Even we can identify the function in mathematical view

   Ex: (6, 4), (1, 0), (2,3), (6,10) : This is not a function because here 6 is appeared 2 times

Tips and common errors

- Several possible errors related to passing parameters:
  - It is a compiler error if the types in the prototype declaration and function definition are incompatible.
  - It is a compiler error to have a different number of actual parameters in the function call then there are in the prototype statement.
  - It is logic error if you code the parameters in the wrong order. Their meaning will be inconsistence
  - In the called program.
- It is compiler error to define local variables with the same identifiers as formal parameters.
- Using void return with a function that excepts a return value or using a return value with a function that excepts a void return is a compiler error.
- Each parameters type must be individually specified: you cannot use multiple definitions like in variables.
- Forgetting the semicolon at the end or a function prototype statement is a compiler error. Similarly, using a semicolon at the end of the header in a function definition is a compiler error.
- It is most likely a logic error to call a function from within itself or one of its called functions.
- It is a compiler error to attempt to define a function within the body of another function.
- It is a run time error to code a function call without the parentheses, even when function has no parameters.
- It is a compiler error if the type of data in the return statement does not match the function return type.
- It is logic error to call srand every time you call rand.