# Unit 6 Recursion

## 6.1 Introduction

### ➢ *Recursion*

Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied. Inorder to solve a problem recursively, two conditions must be satisfied.

a) The problem must be written in recursive form.
b) The problem statement must include a stopping condition.

### ➢ *Principle of Recursion*

For designing a good recursive program, we must make certain assumption such as

1.Base case is the terminating condition for the problem while designing any recursive function.
2. There must be recursive case and should not generate in infinite sequence of calls on itself.
3. Function/object must be so that it can be expressed in terms of previous result.

### ➢ *Types of Recursion*

**1. Direct Recursion** :

The recursion in which the function calls itself is called direct recursion. In this type, only one function is invoked.

```
int xyz( )
{
……..;
……..;
xyz( );
}
```

*Example: Direct Recursion*

**2. Indirect Recursion** :

If two function calls one another mutually then this type of recursion is called indirect recursion. The indirect recursion doesnot make any overhead as direct recursion. When control exit from one function, the local variables of former function are destroyed. Hence, memory is not engaged.

```
int abc( )
{
……..;
……..;
    xyz( );
}

int xyz( )
{
    abc( );
……..;
}
```

### ➢ *Comparision between Recursion and Iteration*

| Recursion | Iteration |
|---|---|
| 1.Recursion is the technique of defining anything in terms of itself. | 1.Iteration allows to repeat the task till the given condition is satisfied. |
| 2.All problems maynot have recursive solution. | 2.Any recursive problem can be solved iteratively. |
| 3. Recursive programming technique conserves considerable execution time. | 3. Iterative programming technique takes less execution time in comparision to recursion. |
| 4. Recursive problems are not efficient for memory utilization and execution speed of problems. | 4. Iterative problems are effective in terms of memory consumption and execution speed. |
| 5. Recursive technique is a top down approach where the given program is divided into modules. | 5. Iterative technique is a bottom up approach where it constructs the solution step by step. |

### ➢ *Advantages of Recursion*

1. Avoidance of unnecessary calling of function.
2. A substitute for iteration where the iterative solution is very complex.
3. Extremely useful when applying same solution.
4. It doesnpt use system stack.
5. Fast at execution time.

- ***Disadvantages of Recursion***

1.It occupies lot of memory space.
2. A recursive function is often confusing.
3. It is difficult to trace the logic of the function.
4. The computer may run out of memory if the recursive calls are not checked.
5. The exit point must be explicitly coded.

## 6.2 Examples of Recursion: factorial, fibonacci sequence, Tower of Hanoi (TOH)

### 6.2.1 Factorial of a number:

Factorial is used to compute the number of permutations (combinations) of arranging a set of ***n*** numbers.

- Factorial of a number n = n! = 1*2*3*…..*n

Examples:

- 3! = 1*2*3 = 6
- 6! = 1*2*3*4*5*6 = 720
- 1! =1
- 0! = 1 (because an empty set can only be ordered in 1 way)

**Input**
Number for which we have to calculate the factorial i.e. ***n***

**Output**
*n! = 1*2*3*….*n*

As we can see we are doing a repetitive task of multiplying numbers from 1 to n, we can use recursion to calculate factorials.

1. n!=1*2*3*….(n-1)*n
2. (n-1)! = 1*2*3*…..*(n-1)
   In the same way
3. n!=(n-1)!*n
4. f(n!)= f(n-1)!*n

Program 1: Write a program to calculate factorial using recursion.

```
#include<stdio.h>
long int factorial(int n);
void main()
{
 int num;
 printf("Enter an integer number\n");
 scanf("%d",&num);
 printf("The factorial is %ld", factorial(num));
 }

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

**6.2.2 Fibonacci Sequence:**

A Fibonacci sequence is the sequence of integer.

- Examples
  1,1,2,3,5,8,13,21,34,55,89 ……
- A Fibonacci number is nothing but a positive integer which is a summation of the two previous Fibonacci number. A Fibonacci number is defined as

$$Fib(n) = \begin{cases} 0 & \text{for } n=0 \text{ or } n=1 \quad [\textit{Base Case}] \\ 1 & \\ fib(n-1)+fib(n-2) & \text{for } n>=2 \quad [\textit{Recursive Case}] \end{cases}$$

Program 2: Write a program to print Fibonacci series.

```
#include<stdio.h>
int fibon(int f);
void main()
{
  int i,n;
  printf("Enter the number for displaying the fibonacci sequence\n");
  scanf("%d",&n);
  for(i=0;i<n;i++)
        printf("%d\t", fibon(i));
}
int fibon(int f)
{
        if(f==0)
  return 0;
  else if (f==1)
  return 1;
  else
  return(fibon(f-1)+fibon(f-2));
}
```

***6.2.3 Tower of Hanoi problem***

Tower of Hanoi is a game played with three poles and a number of different sized disks. Each disk has a hole in the center that makes easy to placeinto poles.

**Initial State** :

a) There are three poles named as origin, intermediate and destination.
b) 'n' number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.
c) The disks are numbered as 1,2,3,4……n.

**Objective:**

Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

**Rules:**

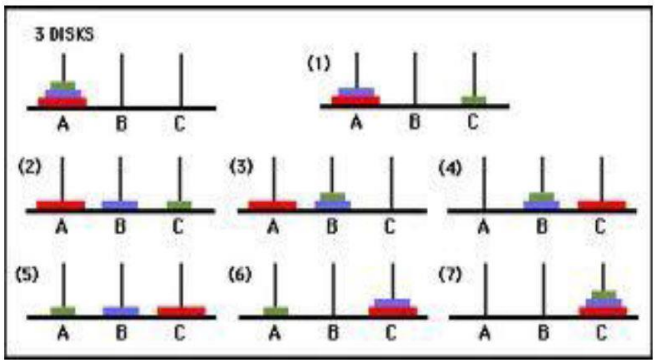
a) Move only one disk at a time.
b) Each disk must always be placed around one of the pole.
c) Never place larger disk on the top of smaller disk.

- ***Algorithm***

To move a tower of 'n' disks from origin to destination pole (where 'n' is positive integer)
1) If n==1, move a single disk from origin to destination pole.
2) If n>1,
a) Let 'intermediate' be the remaining pole other than origin and destination.
b) Move a tower of (n-1) disks from origin to intermediate.
c) Move a single disk from origin to destination.
d) Move a tower of (n-1) disks from intermediate to destination.

- ***Solution to the Tower of Hanoi problem when number of disks 'n'=3.***

Let, the origin, intermediate and destination pole or peg be denoted as Peg 'A', Peg 'B' and Peg 'C' respectively. Let the disks be denoted as Disk1, Disk2 and Disk3.




Program 3: WAP to solve the problem of TOH.

```
#include<stdio.h>
void toh(int n,char a,char b,char c);
void main()
{
 int n;
  printf("Enter the number of disks\t");
 scanf("%d",&n);
          toh(n,'A','B','C');
   }

 void toh(int n,char a,char b,char c)
  {
    if(n<=0)
     printf("Number of disks can't be zero (or) less than zero");
   else if(n==1)
     printf("Move disk from %c to %c\n",a,c);
   else
    {
     toh(n-1,a,c,b);
     toh(1,a,b,c);
     toh(n-1,b,a,c);
    }
  }
```

## 6.3 Applications and Efficiency of recursion

- ***Applications of Recursion***

Recursion is applied to problems which can be broken into smaller parts, each part looking similar to the original problem. Some application areas of recursion are listed below.

1) Algorithms on trees and sorted lists can be implemented recursively.
2) Recursion is also useful to guarantee the correctness of an algorithm.
3) Recursion is used heavily in backtracking algorithms e.g The Sudoko Solver.
4) Recursion is used in several sorting algorithms like Quick sort, Merge Sort etc.
5) Parsers and compilers also use recursion heavily.

- ***Efficiency of Recursion***

In general, a non-recursive version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a block is avoided in the non-recursive version. Thus, the use of recursion is not necessarily the best way to approach problem, even though the problem definition may be recursive. The use of recursion may involve a trade off in between simplicity and performance. Therefore, each problem must be judged on its own individuals merit.