RECURSION IN C

"Recursion is a process in which a problem is define in terms of itself". In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process".

ADVANTAGES OF RECURSION:

1. Reduce unnecessary calling of function.

2. Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex. For example to reduce the code size for Tower of Honai application.

3. Extremely useful when applying the same solution.

DISADVANTAGES OF RECURSION:

1. Recursive solution is always logical and it is very difficult to trace.(debug and understand).

2. In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.

3. Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.

4. Recursion uses more processor time.

Tower of Hanoi

Use general notation:

T(N, Beg, Aux, End)

Where,

T denotes our procedure

N denotes the number of disks

Beg is the initial Tower

Aux is the auxiliary Tower

End is the final Tower

Recursive steps to solve Tower of Hanoi.

1. T(N-1, Beg, End, Aux)

2. T(1, Beg, Aux, End)

3. T(N-1, Aux, Beg, End)

Step 1 says: Move top (N-1) disks from Beg to Aux Tower.

Step 2 says: Move 1 disk from Beg to End Tower.

Step 3 says: Move top (N-1) disks from Aux to End Tower.

Pseudo code:

```
//N = Number of disks
//Beg, Aux, End are the Towers
T(N, Beg, Aux, End)
Begin
   if N = 1 then
      Print: Beg --> End;
   else
      Call T(N-1, Beg, End, Aux);
      Call T(1, Beg, Aux, End);
      Call T(N-1, Aux, Beg, End);
   endif
```

End

Moves required:

If there are N disks then we can solve the game in minimum $2^N - 1$ moves.

Example: N = 3

Minimum moves required = 23 – 1 = 7

Tower of Hanoi code in C

```
#include <stdio.h>

void t(int n, char beg, char aux, char end);

int main(){

  printf("Moves\n");

  t(3, 'a', 'b', 'c');  //N = 3 (no. of disks)  a, b, c are the three pegs

  return 0;

}

void t(int n, char beg, char aux, char end){

  if(n == 1){

    printf("%c --> %c\n", beg, end);

  }

else{

    t(n-1, beg, end, aux);

    t(1, beg, aux, end);

    t(n-1, aux, beg, end);
```

}}

FACTORIAL

Pesudo Code:

```
Fact(n)
Begin
  if n == 0 or 1 then
    Return 1;
  else
    Return n*Call Fact(n-1);
  endif
End
```

Factorial code in C

```
//factorial declaration recursive and non-recursive
#include <stdio.h>
//function declaration
int fact(int n);
int nonRecFact(int n);
int main(){
  //variable declaration
  int n, f;
  //input
```

```
  printf("Enter n:  ");
  scanf("%d", &n);
  //recursive fact
  f = fact(n);
  printf("Recursive fact: %d\n", f);
  //non-recursive fact
  f = nonRecFact(n);
  printf("Non-Recursive fact: %d\n", f);
  return 0;
}
//function definition
int fact(int n){
  if(n == 0 || n == 1)
    return 1;
  else
    return n * fact(n-1);
}
int nonRecFact(int n){
  int i, f = 1;
  for(i = 1; i <= n; i++)
    f *= i;
```

```
  return f;
}
```

FIBONACCI SERIES

Pseudo Code

```
Fibo(n)
Begin
  if n <= 1 then
      Return n;
  else
      Return Call Fibo(n-1) + Call Fibo(n-2);
  endif
End
```

Fibonacci C Code:

```
//fibonacci series recursive and non-recursive
#include <stdio.h>
//function declaration
int fibo(int n);
int nonRecFibo(int n);
int main(){
  //variable declaration
  int n, f;
```

```
  //input
  printf("Enter n: ");
  scanf("%d", &n);
  //recursive
  f = fibo(n);
  printf("Recursive Fibo: %d\n", f);
  //non-recursive
  f = nonRecFibo(n);
  printf("Non-Recursive Fibo: %d\n", f);
  return 0;
}
//function definition
int fibo(int n){
  if(n <= 1)
    return n;
  else
    return fibo(n-1) + fibo(n-2);
}
int nonRecFibo(int n){
  int i, a, b, f;
  if(n <= 1)
```

```
    return n;
  else{
    a = 0, b = 1, f = 0;
    for(i = 2; i <= n; i++){
      f = a + b;
      a = b;
      b = f;
    }
  }
  return f;
}
```

GREATEST COMMON DIVISOR

Pseudo code:

```
GCD(x, y)
Begin
    if y = 0 then
        return x;
    else
        Call: GCD(y, x%y);
    endif
End
```

GCD code in C

```
#include <stdio.h>

int gcd(int x, int y);

int main(){

  int a, b, g;

  printf("Enter a and b:\n");

  scanf("%d%d", &a, &b);

  //gcd

  g = gcd(a, b);

  //in case g is negative, then convert it into positive

  if(g < 0){

    g *= -1;

  }

  //output

  printf("GCD(%d, %d) = %d\n", a, b, g);

  return 0;

}

int gcd(int x, int y){

  if(y == 0) return x;

  else gcd(y, x%y);

}
```

Ackermann function

General notation:

$$A(n,m) = \begin{cases} M+1, & \text{If } n=0 \\ A(n-1,1), & \text{If } n>0,\ m=0 \\ A(n-1, A(n,m-1)), & \text{If } n,m>0 \end{cases}$$

C program for Ackermann function:

```
#include<stdio.h>

int A(int m, int n);

main()

{       int m,n;

        printf("Enter two numbers :: \n");

        scanf("%d%d",&m,&n);

        printf("\nOUTPUT :: %d\n",A(m,n));

}

int A(int m, int n)

{

        if(m==0)

                return n+1;

        else if(n==0)

                return A(m-1,1);

        else

                return A(m-1,A(m,n-1));

}
```