Recursion

- Defining a function in terms of itself is called *recursion*. We call a method that calls itself a *recursive method*. Recursive algorithms can be done iteratively

Each recursive call has its own distinct set of parameters and local variables. A recursive call is a separate entry on the execution stack.
Parts of recursion:

1. The Basis Case

- In order for recursion to work correctly, every recursive method must have a *basis case*.
- The basis case is an input for which the method does not make a recursive call. The basis case prevents *infinite recursion*.
- It's not enough for there to simply *be* a basis case; the values of the input must reliably *approach* the basis value.

2. Reduced case that works to the base case

A Recursive Example

```
#include<iostream.h>
int power(int,int); //prototype
void main()
{
     int testpower;
     testpower  = power(5, 2);
     cout<<testpower;
}
                 //function definition
int power(int base, int exp)
{
     if (exp == 0)
          return 1;
     else
          return (base * power(base, exp-1));
}
```

Designing a Recursive Algorithm:

- Every recursive call must either solve part of the problem or reduce the size
- Reduction must lead to a solution
- Steps
  - first determine the base case
  - determine the general case
  - combine the two to produce the algorithm

<u>The Factorial Function</u>

The factorial function is:

`n! = n(n -1) ………1`
We can define the factorial function recursively as:
Basis case:
`1! = 1`
Recursive case (n > 1):
`n! = n *(n -1)!`
So factorial function can be defined recursively as
```
 fact(n) = n*fact(n-1);
```

Factorial : The Iterative Algorithm
Calculates the factorial of a number using a loop

INPUT	**n is the number to be raised factorially**
RETURN	**n! is returned**

```
1. i = 1
2. factN = 1
3. Loop ( i <= n)
     1. factN = factN * i
     2. i = i + 1;
4. end loop
5. return factN
```

Running example:
```
0! = 1
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 *3 *2 * 1
n! = n * (n-1) * (n-2) *… … … …* 3 * 2 *1
```

Factorial: Formal Definition Recursively

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fact(n-1) & \text{if } n > 0 \end{cases}$$

Algorithm:
Calculates the factorial of a number using recursion

Input	**n is a number to be raised factorially**
Return	**n! is returned**

```
  1. if (n is 0)
          return 1
  2. else
          return n *  recursiveFactorial(n-1)
  3. end if
```
<u>Recursive Version:</u>

0! = 1 (base case)

1! = 1	= 1 * 0!
2! = 2 * 1	= 2 * 1!
3! = 3 * 2 * 1	= 3 * 2!
4! = 4 *3 *2 * 1	= 4 * 3!
n! =	= n * (n-1)!

The Fibonacci Sequence

A Fibonacci sequence is the sequence of integers:
0,1,1,2,3,5,8,13,21,34,………….. in which each element in the sequence is the sum of the two preceding elements.

In the Fibonacci sequence, the basis cases are n = 0 and n = 1. Since the sequence is only defined for nonnegative integers n, the recursive definition will always approach 0.

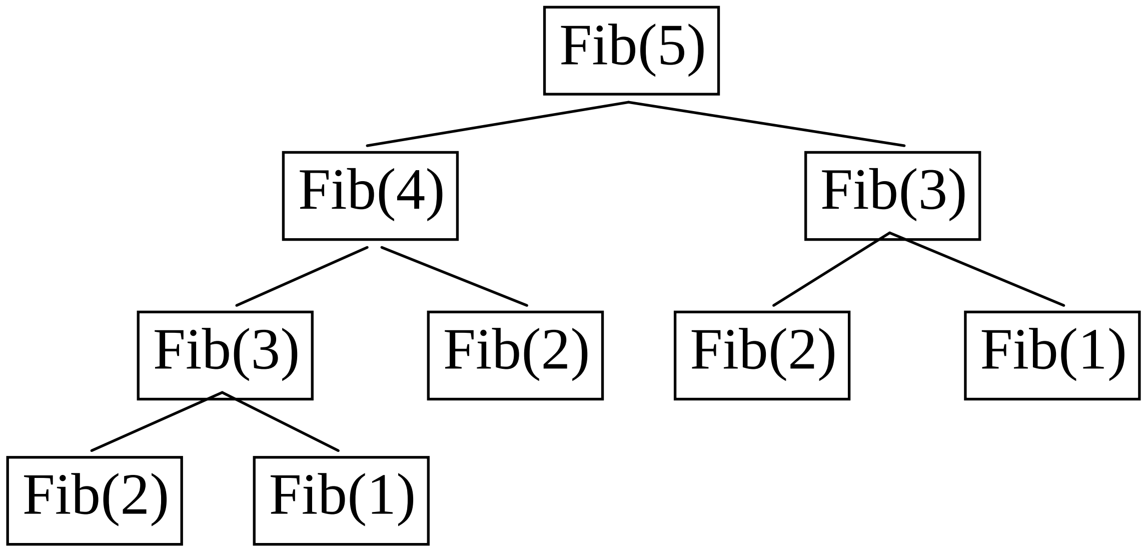
Basis cases:
fibo(0) = 0 , fibo(1) = 1
Recursive case (n > 2):
fibo(n) = fibo(n-1) + fibo(n-2)

The non recursive Version of function that returns the nth Fibonacci sequence:

```
int fibo(int n)
{
  int i, Fn_1, Fn_2, Fn;
  if(n==0 ||n==1)
     return n;
  Fn_2 = 1;
  Fn_1 = 0;
  for (i = 2; i <= n; i++)
  {
    Fn = Fn_1 + Fn_2;
    Fn_2 = Fn_1;
    Fn_1 = Fn;
  }
  return Fn;
}
```

A recursive Version:

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



<u>The Multiplication of Natural numbers</u>

Another example of recursive definition is the definition of multiplication of natural numbers. The product a*b, where a and b are positive integers , may be defined as a added to itself b times. This is an iterative definition. An equivalent recursive definition can be made as below:

<u>The basis:</u>

```
a*b= a if b==1
a*b  =0 , if  a==0 or b==0
```

<u>Recursive case: (b>1)</u>

```
a*b = a+(a*(b-1));
```

The recursive function:

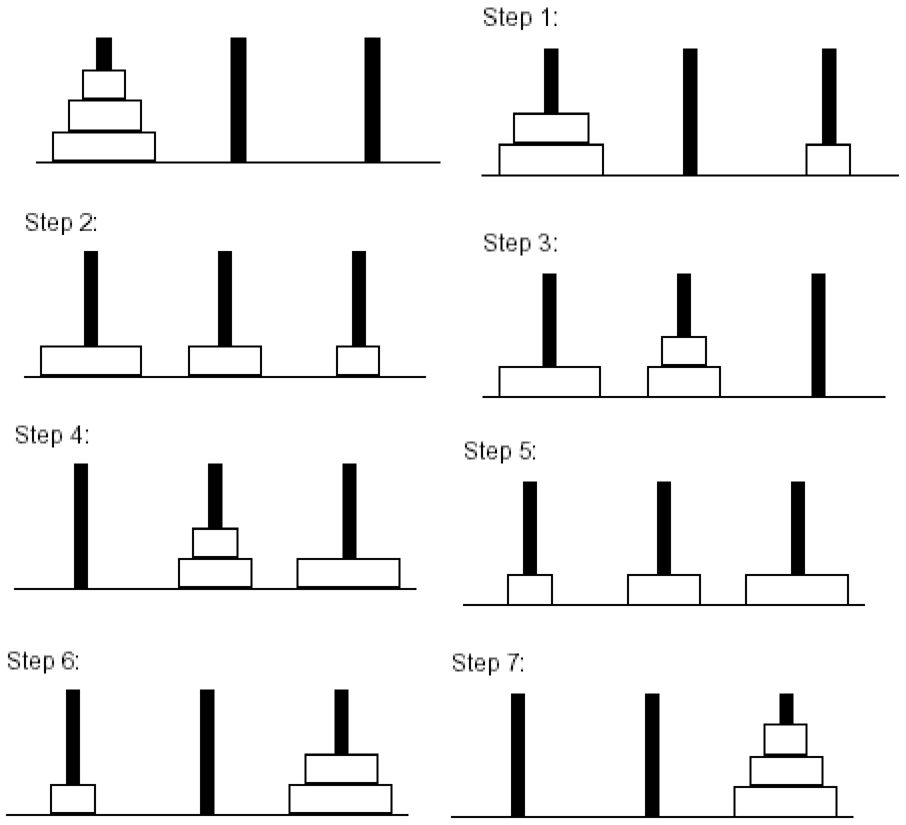
```
int multi(int a, int b)
{
  if(a==0 || b==0 )
   return 0;
  else if(b==1)
    return a;
  else return (multi(a,b-1) +a);
}
```

The Towers of Hanoi

A puzzle developed in 1883 by Edouard Lucas. Three poles upon which a certain number of round discs, increasing in size, are placed (all of the discs initially start on the first pole). The object of the puzzle is to move all of the discs from one pole to another pole. Only one disc can be removed from the poles at any one time, and no disc can be placed upon a larger disc.

- The Towers of Hanoi problem is to move a stack of plates from the first post to the third. We may only move one plate at a time and a larger plate cannot be stacked on top

The Towers of Hanoi



The Towers of Hanoi is a classic example of a recursive problem. To solve it for n plates: i.e. to move n disks from first peg (A) to Third peg (C) using second intermediate peg (B) as auxiliary:

1. if n== 1, move single disk from A to C and stop - A basis case.
2. Move top n-1 disks from A to B, using C as auxiliary.
3. Move remaining disk from A to C.
4. Move the n-1 disks from B to C , using A as auxiliary.

The Recursive Program for TOH problem:

```
#include<iostream.h>
void move(int , char,char,char);
void main()
{
  int disks;
  cout<<"No of disks ?:";
  cin>>disks;
  move(disks,  'A', 'C', 'B');
}

/* move() definition */
void move(int n, char speg, char tpeg, char axpeg)
```

```
{
   /* if n==1, make the move and return */
  if( n==1)
{
  cout<<”\nMove disk 1 from peg “<<speg<<” to peg “<<tpeg;
   return;
}
 /*move top n-1 disks from A to B , using C as auxiliary */
move(n-1, speg,axpeg,tpeg);
/* move remaining disk from A to C */
cout<<“\nmove disk “<<n<<” from peg “<<speg<< “ to peg
“<<tpeg;
 /* move n-1 disks from B to C using A as Auxiliary */
 move(n-1,axpeg,tpeg,speg);
}
```

The out put of program looks like:

```
No of disks ?:3

Move disk 1 from peg A to peg C
move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
move disk 2 from peg B to peg C
Move disk 1 from peg A to peg C
```