RECURSION

The process of a function calling itsey, to solve a small instance of the problem is known as recursion.

A recursive function should contain atleast 1 base condition to terminate it, otherwise, the function will enter an infinite loop of calling itsey again and again.

basic syntax :

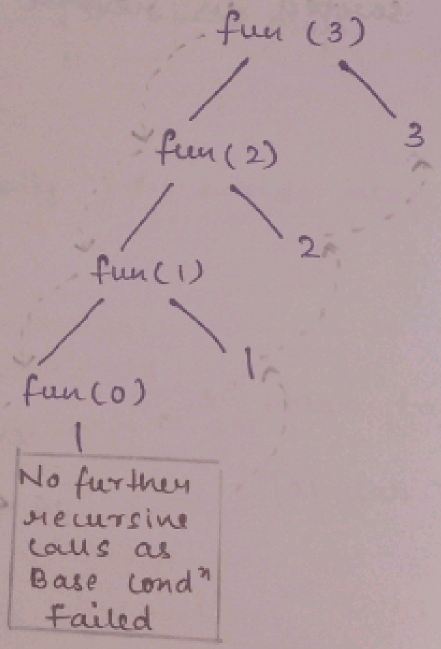
```
Type function (parameters) {
        if (base conditions) {
              1. - - - -
              2. function (parameter)
              3. - - - .
        }
}
```

Q. write a recursive function to print first N numbers.

```
void fun (int n) {
    if (n > 0) {
        fun (n - 1);
        cout << n;
    }
}.

void main () {
    fun (3);
}
```

Output : 1 2 3

Time complexity : O(N)



Note : we can apply Recursion in many algorithms like the Fibonacci sequence, Tower of Hanoi, binary search, tree Traversal, etc.

execution and Different phase in recursion

```
              Type fun (params) {
                   if (<base conditions>) {
Ascending    ←        1. - - - -   executed at calling phase./Time
phase
                      2. fun (param) × ((some other expression))
Descending   ←        3. - - - - -   executed during
phase.            }                  returning
              }                      phase/Ti
```

Q. what is the main difference between loops and recursion as both of them are repeating staments?

→ As Both Recursion and loops are repeating statement but the main difference between them is that loops only have Ascending phase. where as recursion have both Ascending (calling) phase as well as Descending (returning) phase.

Time complexity of recursion

```
vaid  fun(n) {                                   T(n)
    if (n>0) {                  - - - - .         1
        printf('%d', n);  - - - - - .             1
        fun(n);       - - - - -           T(n-1)
    }
}
```

using recurrence Relation

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1)+1 & n > 0 \end{cases}$$

$T(n) = T(n-1) + 1$ — (I)

$T(n-1) = T(n-2) + 1$ — (II)

from (I) and (II)

$T(n) = T(n-2) + 2.$

we can also say,

$T(n) = T(n-k) + k$

let $k = n$

$T(n) = T(n-n) + n.$

$T(n) = T(0) + n$ [as $T(0) = 1$].

$T(n) = n + 1 = n$

Static Variables in Recursion (working)

```
int  fun (int n) {
     static int x = 0;
     if (n >0) {
          x++;
          ~~fun~~ return fun(n-1) + x;
     }
     return 0;
}
```

let n = 5

$\overset{x}{\boxed{\cancel{0}}}$ ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5

fun(5) = 25

/ 20

fun(4) + 5 = 20+5 = 25

/ 15

fun(3) + 5 = 15+5 = 20

/ 10

fun(2) + 5 = 10+5 = 15

/ 5

fun(1) + 5 = 5+5 = 10

/ 0

Types of Recursion

1. Tail Recursion

If a recursive function is calling itself and ~~&~~ in that recursive function, is the last operation then it is known as Tail Recursion.

eg.

```
void  fun( int n) {
        if (n>0) {
            printf('%d', n);
            fun(n-1);
        }
  }
```

In Tail Recursion, every operation is done at calling time (i.e. Ascending phase) with no descending phase.

example of what is not a Tail recursion.

```
void  fun (int n) {
        if (n>0) {
            printf("%d", n);
            fun(n-1) + n;
        }
  }
```

This will be done at descending phase.

The Difference b/w Tail recursion and loop is that the Stack will ~~create~~ have n activation record in case of Tail recursion and 1 in case of loop.

	Tail recursion	Loop
Time complexity	O(n)	O(n)
Space complexity	O(n)	O(1)

2. Head Recursion

If the recursive call is the first operation in a recursive function, then the recursion is called Head Recursion.

eg.

```
void  fun (int n) {
        if (n>0) {
            fun(n-1);
            Printf ("%d", n);
        }
  }
```

In Head ~~opera~~ Recursion, every operation is done at Returning Time (i.e. descending phase)

3. Indirect Recursion

In Indirect Recursion, First function calls second function and the second function calls the first function again, creating a cycle.

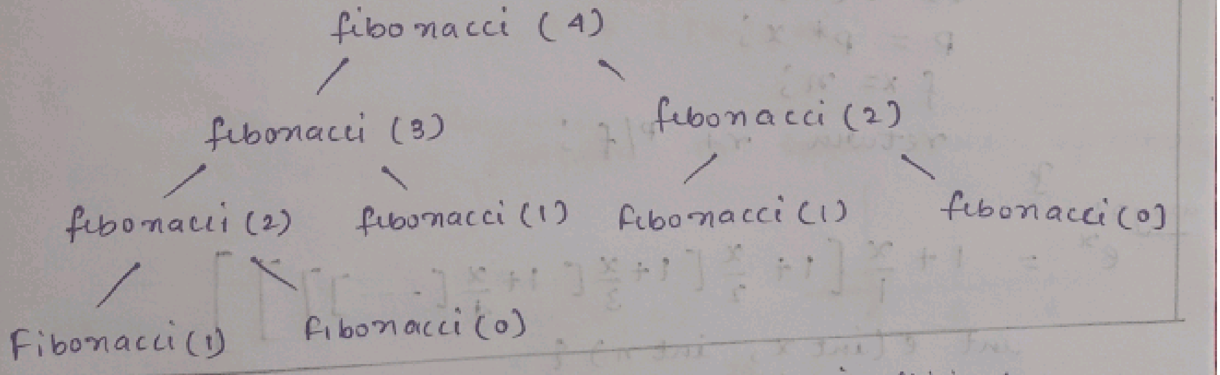
eg.
```
void   function A ( int n) {
          if (n==0) return;
          function B (n-1);
   }

   void function B (int n) {
          if (n==0) return;
          function A ( ~~int~~ n -1);
   }
```

4. Tree Recursion

If a function makes more than one recursive call within itsely, which results in Branching structure is called Tree Recursion.

eg.
```
int fibonacci (int n) {
     if (n<=1) return n;
     return  fibonacci (n-1) + fibonacci (n-2);
}
```

let n = 4



Pros: - Simple

Cons: - inefficient
- can cause memory stack overflow.

5. Nested Recursion

It is a Type of recursion where the recursive function call is made within another recursive call. In other words, the parameter of a recursive function itsely is a recursive call.

```
eg.   int   fun ( int n )
      {
        if ( n >100 ) {
             return  n-10;
        } else {
             freturn  fun ( fun ( n+11 ) );
      }
```

let n = 98

fun (109) = 99

fun (98)

↓

fun (fun (98+11))

↕

fun (99)

↓

fun (110) = 100

fun (fun (99+11))

↕

fun (100)

↓

90

Q. Write a recursive function for taylor series:

Taylor series $(e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \text{---})$

```
int e( int x, int n) {
    Static int p=1, f=1;
    int r;
    if (n==0) return 1;
    r= e(x, n-1);
    p = p* x;
    f x= n;
    return r+ p/f;
}
```

way2:

$$e^x = 1 + \frac{x}{1}\left[1 + \frac{x}{2}\left[1 + \frac{x}{3}\left[1 + \frac{x}{4}\left[\text{- - -}\right]\right]\right]\right]$$

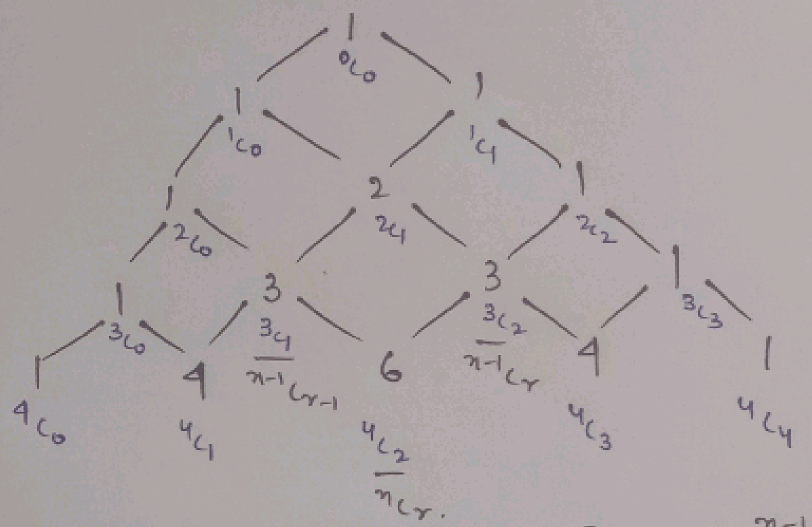
```
int e (int x, int n) {
      Static int s=1;
      if (n==0) return s;
      s = 1+ x/n * s;
      return e (x , n-1);
}
```

Q2. write a recursive function for finding $^nC_r = \frac{n!}{r!(n-r)!}$

way 1: use of recursive factorial function

```
int factorial (int n){
    if (n <= 0) return 1;
    return n* factorial (n-1);
}

int C (int n, int r) {
    int a = factorial (n), b= factorial (r);
    int c = factorial (n-r);
    return a/(b*c);
}
```

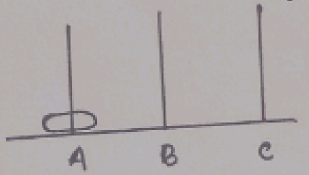
way 2: with the help | study of pascal triangle



from this we can say that $^nC_r = {}^{n-1}C_{r-1} + {}^{n-1}C_r$.

```
int C (int n, int r) {
    if (r==0 || n==r) return 1;
    return C(n-1, r-1) + C(n-1, r);
}
```

Q3. write a recursive function to solve Tower of Hanoi



when n = 1. TOH(1, A, B, C)

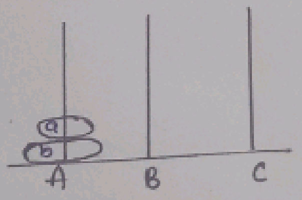
Here we can simply move disk from A to C

TOH (1, A, B, C)

from mediary to

when n = 2 TOH(2, A, B, C)

1. move a from A to B   TOH(1, A, C, B)
2. move b from A to C — Same as when n=1
3. move a from B to C   TOH(1, B, A, C)



From this we can say the psuedcode will be

```
ToH (n, A, B, C)
    1. ToH (n-1, A, C, B)
    2. move DISK from A to C using B
    3. ToH (n-1, B, A, C)
```

<u>code</u> :

```
Void ToH ( int n, int A, int B, int C) {
        if (n > 0) {
                ToH (n-1 , A , C , B);
                printf ("from %d to %d \n", A , C);
                ToH (n-1 , B , A , C);
        }
}
```