Recursion :-

- First we need to understand →

1. What is Recursion?
2. Example of Recursion?
3. Tracing Recursion?
4. Stack used in Recursion
5. Time Complexity
6. Recurrence Relation

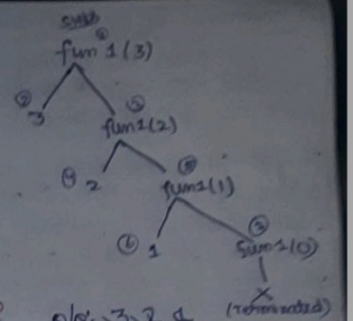
⇒ What is Recursion?

If a function is ~~called~~ calling itself then it is called as some method recursive function. In recursion there must be some condition, that will terminate recursion.

```
Type fun (param)
{
    if(<base condition>)
    {
        1. —
        2. fun (param)
        3. —
        4. —
    }
}
```

⇒ Eg of Recursion :-

① #include <iostream>
using namespace std;
int fun1(int n)
{
  if(n>0)
  {
  ① cout << n << "\n";
  ② fun1(n-1);
  }
}
int main()
{
  int x=3;
  fun1(x); —— 3
} 2
  1



Point during Ascending of Recursion

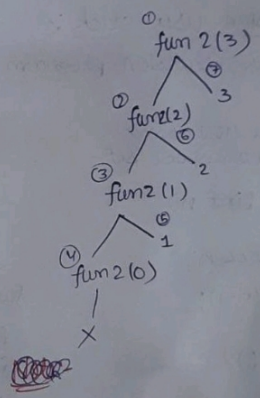
O/P: 3 2 1

(Tracing Tree of Recursive function)

⇒ Tracing Recursion :-

Recursion tracing occurs in the form of a tree like shown above.

Eg :- ② #include <iostream>
using namespace std;
int fun2(int n)
{
  if(n>0)
  {
    fun2(n-1);
    printf("%d", n);
  }
}
int main()
{
  int x=3;
  fun2(x); —— 1
} 2
  3



Point during descending of Recursion

Comparing Eg ① & ② :-

In eg ① first the printing was done & then their was a recursive call. So it first print 3, then call fun1(2), then print 2, then call fun1(1), then print 1 & then call fun1(0).

In eg ② first the calling of a function fun2(2) was done & then calling of a function fun2(1), then further calling fun2(0) & after that the printing done at the return time of the function so when fun2(1) was returned it print 1, after that fun2(2) was returned and prints 2 & then fun2(3) was returned and print 3.

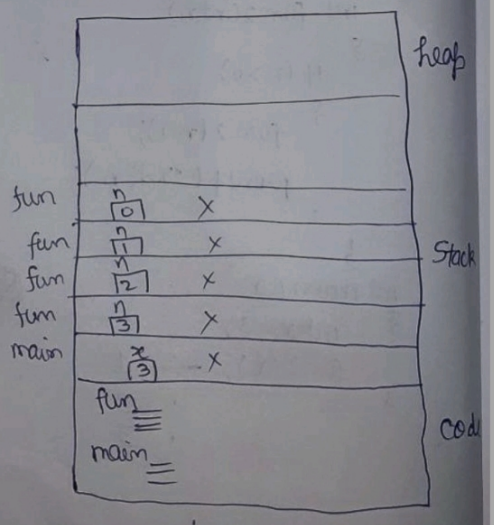
So in eg ①, first the printing done & then calling function thus, in eg ① printing was done at counting time before the function is called printing was done.
But in eg ②, the printing was done at a time when the function is returning to the previous caller, the previous call so at the reading time the printing was done after the function call.

Understanding is Imp

☆ How Recursion uses stack :-

① We use this recursion program to understand how the recursion uses stack.

```
#include<iostream>
using namespace std;
int fun(int n)
{
    cout<<n;
    fun(n-1);
}
int main()
{
    int x=3;
    fun(x);
}
```



☆ Generalising Recursion :-

```
#include <iostream>
using namespace std;
void fun(int n)
{  if(n>0)
   {
```

Ascending 1. calling
2. fun(n-1) *2
Descending 3. Returning

```
   }
}
int main()
{  int x=4;
   fun(4);
}
```

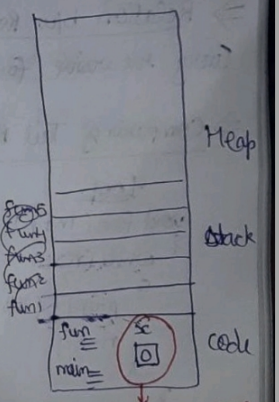
Recursion has two Phase parts :-
1. Ascending Phase
2. Descending Phase

Imp

☆ Static variables in Recursion :- x 0 1 2 3 4 5

```
#include <iostream>
using namespace std;
int fun(int n)
{
  static int x=0;
  if(n>0)
  {
    x++;
    return fun(n-1)+x;
  }
}
int main()
{
  cout<<fun(5);
}
```

① fun(5) = 25 (12)
② fun(4) + 25 (11) — 20
③ fun(3) + 20 (10) — 15
④ fun(2) + 15 (9) — 10
⑤ fun(1) + 10 (8) — 5
⑥ fun(0) + 5 (7) — 0



Global & static variables have stored data in ~~code section~~

Note ⇒ • Adding of x occurs during the descending phase.
• Same tracing true forms when we use global & variables.

Note:- As we see that space complexity of loop is better than tail recursion. So instead of writing tail recursion try to write loops.

★ Head Recursion:-

If a recursive function calling itself and that recursive call is in the first statement in the function then its known as Head Recursion.

There is no statement, no operation before the call. The function doesn't have to process or perform any operation at the time of calling & all operations are done at returning time.

⇒

```
void fun(n)
{
  if(n>0)
  {
    fun(n-1);
    ===
    ===
    ===
  }
}
```

Syntax of Head Recursion

⇒ Comparing Head Recursion with loops:-

It's not advisable to convert a ~~loop~~ head recursive func^n into loops

Loop

```
int fun(int n)
{
  int j=1;
  while(i<=n)
  {
    cout<<j;
    i++;
  }
}
```

Head Recursion

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

☆ Tree Recursion:-

To understand Tree Recursion, let's first understand linear Recursion. If a Recursive function calling itself for only one time then it is a linear recursion.
Otherwise if a recursive function calling itself for more than one time then it is known as Tree Recursion.

```
void fun(n)
{
  if(n>0)
  {
    cout<<n;
    fun(n-1);
    cout<<(n*2)+1;
  }
}
```

Eg of Linear Recursion

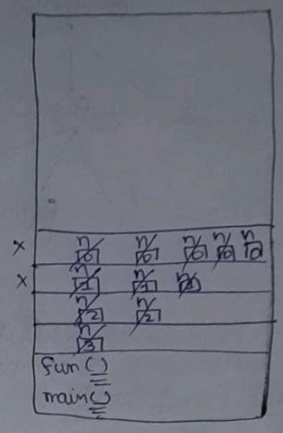
⇒ Syntax of Tree Recursion:⇒

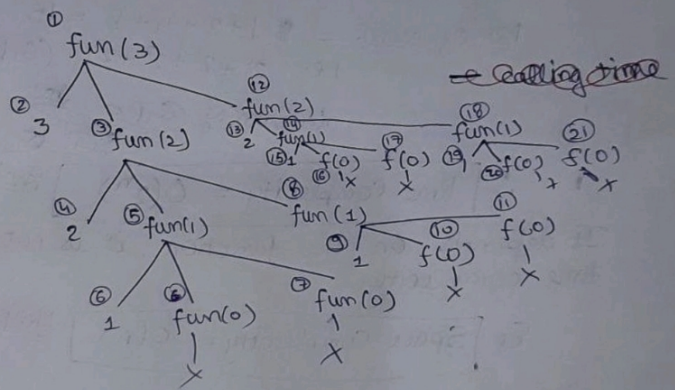
```
fun(n)
{
  if(n>0)
  {
    ==
    fun(n-1);
    ==
    ==
    fun(n-1);
    ==
    ==
  }
}
```

Example:-

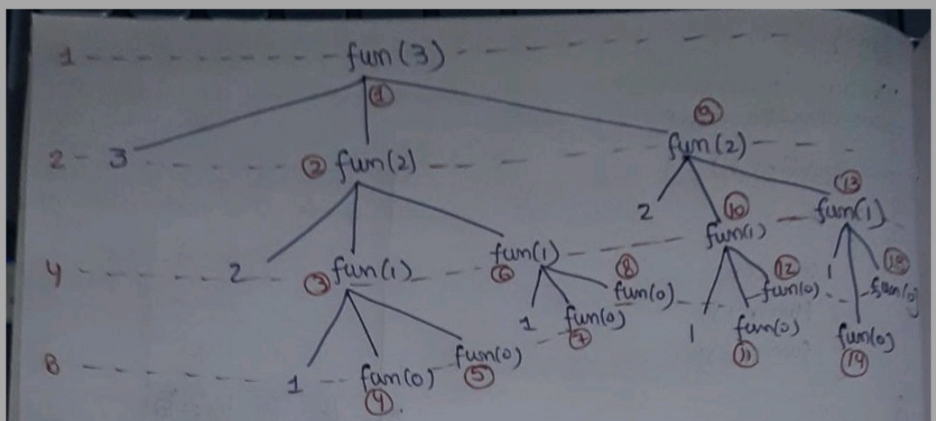
```
#include<iostream>
using namespace std;
void fun(int n)
{ if(n>0)
  {
    cout<<n;
    fun(n-1);
    fun(n-1);
  }
}
int main()
{ int x=3
  fun(3);
}
```

— 3 2 1 1 2 1 1





In Next Page This Tracing Tree was drawn Nicely



— Represents Calling number of a Function.
---- Represents Total no of call in a level.

⇒ Time Complexity :-

No. of calls = $1+2+4+8 = 15$

i.e. $2^0+2^1+2^2+2^3$ (GP)

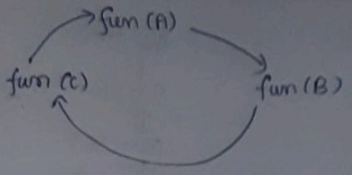
∴ Time Complexity = $O(2^n)$ Imp

It depends on the function, it is not a constant time complexity.

Eg Space Complexity = $O(n)$ Imp

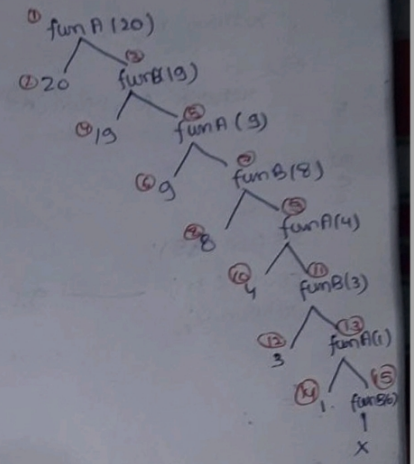
☆ Indirect Recursion:-

In this recursion, there may be more than one functions and are calling one another in a circular manner.



Example :-

```
#include<iostream>
using namespace std;
int funB(int n);
int funA(int n)
{ if (n>0)
    {
      cout<<n<<" ";
      funB(n-1);
    }
}
int funB(int n)
{
  if (n>1)
    { cout<<n<<" ";
      funA(n/2);
    }
}
int main()
{
  funA(20);
  return 0;
}
```



20 19 9 8 4 3 1

☆ Nested Recursion :-

In this recursion, a recursive function will pass the parameter as a recursive call. That means "recursion inside recursion".

for Example:

```
#include <iostream>
using namespace std;
int fun(int n)
{
    if(n>100)
    {
        return n-10;
    }
    else
        return fun(fun(n+11));
}

int main()
{
    int r;
    r=fun(95);
    cout<<r;
    return 0;
}
```

—— 91

fun(95)
fun(fun(95+11)) 96=fun(106)
fun(96)
fun(fun(107)) 97=fun(107)
fun(97)
fun(fun(108)) 98=fun(108)
fun(98)
fun(fun(109)) 99=fun(109)
fun(99)
fun(fun(110)) 100=fun(110)
fun(100)
fun(fun(111)) 101=fun(111)
fun(101)
91

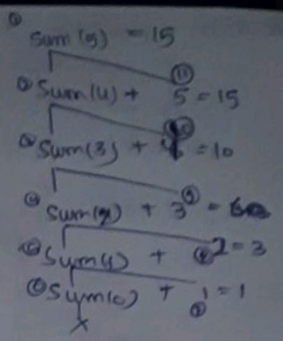
Ques) Write a recursive function to find sum of first 'n' Natural numbers?

$Sum(n) = 1+2+3+4+\dots+(n-1)+n$

$Sum(n) = Sum(n-1)+n$

$\therefore Sum(n) = \begin{cases} 0 & \text{if } n=0 \\ Sum(n-1)+n & \text{if } n>0 \end{cases}$

```
#include<iostream>
using namespace std;
int sum(int n)
{ if(n==0)
    return 0;
  else
    return sum(n-1)+n;
}
int main()
{
  cout<<sum(5);
}
```



Time complexity = O(n)
Space complexity = O(n)

Ques 2 write a recursive function to find factorial of n numbers?

$fact(n) = 1*2*3*4*\ldots*(n-1)*n$

$\therefore fact(n) = fact(n-1)*n$

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

```
#include <iostream>
using namespace std;
int fact(int n)
{
  if(n==0)
    return 1;
  else
    return fact(n-1)*n;
}
int main()
{
  int n;
  cout<<" Enter the number ";
  cin>>n;
  cout<<fact(n);
}
```

Time Comp. = O(n)
Space Comp. = O(n)

Ques: Write a program to find power by using recursion?

$m^n = m * m * m * \ldots$ for n times

pow(m,n) = m * m * m * ... * n-1 times * m

pow(m,n) = pow(m,n-1) * m

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

Method-1 :-

```
int pow(int m, int n)
{
    if(n==0)
        return 1;
    else
        return pow(m,n-1)*m;
}
```

① pow(2,9)
② pow(2,8) *
③ pow(2,7) *
④ pow(2,6) *
⑤ pow(2,5) * ⑩ 2 = $2^5 * 2 = 64$
⑥ pow(2,4) * ⑨ 2 = $2^4 * 2 = 32$
⑦ pow(2,3) * ⑧ 2 = $2^3 * 2 = 16$
⑧ pow(2,2) * ⑦ 2 = $2^2 * 2 = 8$
⑨ pow(2,1) * ⑥ 2 = $2^1 * 2 = 4$
⑩ pow(2,0) * ⑤ 2 = $1 * 2 = 2$
1

Slower Approach

∵ There are 9 multiplication has made during execution.

Time Compl. = O(n)
Space Compl = O(n)

Method-2

```
int pow(int m, int n)
{
    if(n==0)
        return 1;
    else if(n%2==0)
        return pow(m*m, n/2);
    else
        return m*pow(m*m, (n-1)/2);
}
```

① pow(2,9) $= 2^9$

② 2^1 * pow(2^2,4) $= 2 \times 2^8 = 2^9$

③ pow(2^4,2) $= 2^8$

④ pow(2^8,1) $= 2^8$

⑤ 2^8 * pow(2^{16},0) $= 2^8 \times 1 = 2^8$

Faster Approach

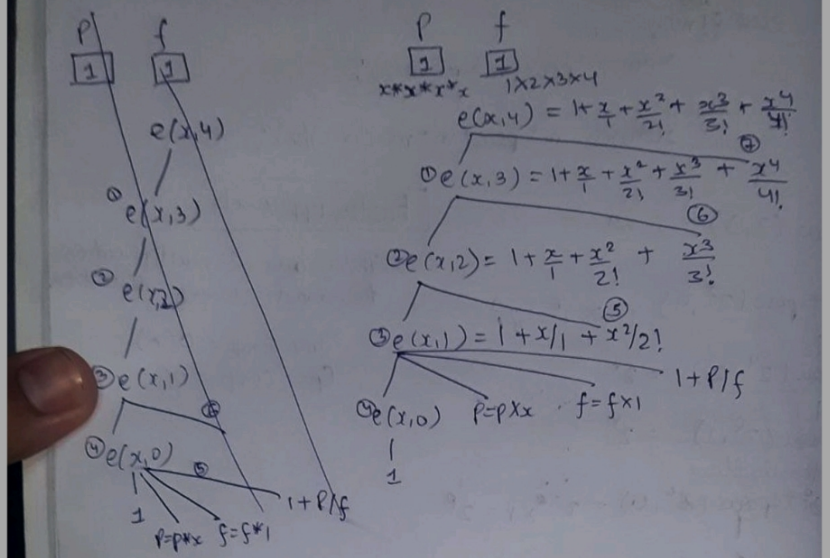
∵ There are 6 multiplication has made during execution

Time Comp. = O(n)

Space Comp. = O(n)

★ Taylor Series (e^x) :-

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + n \text{ terms}$$




```
#include <iostream>
using namespace std;
double e(int x, int n)
{
    static double p=1, f=1;
    double r;
    if (n==0)
        return 1;
    else
    {
        r = e(x, n-1);
        p = p*x;
        f = f*n;
        return r+p/f;
    }
}
int main()
{
    cout << e(2,4);
}
```

Time Comp. = $O(n^2)$
Space Comp. = $O(n)$

★ Taylor Series using Horner's Rule:-

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

Iterator (Loop)

```
#include <iostream>
using namespace std;
double e(int x, int n)
{ double s=1;
  for(; n>0; n--)
  {  s = 1 + (x/n)*s;
  }
  return s;
}
int main()
{  cout << e(4,20);
}
```

Recursion

```
#include <iostream>
using namespace std;
double e(double x, double n)
{
    static double s=1;
    if(n==0)
        return s;
    else
    { s = 1+x/n*s;
      return e(x, n-1);
    }
}
int main()
{
   cout << e(4,20);
}
```

Time complexity = O(n)

★ Fibonnacci Series :-

fib(n)	0	1	1	2	3	5	8	13
n	0	1	2	3	4	5	6	7

fib(n) = { 0 if n=0; fib(n-2)+fib(n-1) otherwise }

Iterative method :-

```
int fib(int n)
{
  int t0=0, t1=1, s;
  if(n<=1) return n;
  for(int i=2; i<=n; i++)
  {
     s = t0+t1;
     t0=t1;
     t1=s;
  }
  return s;
}
```

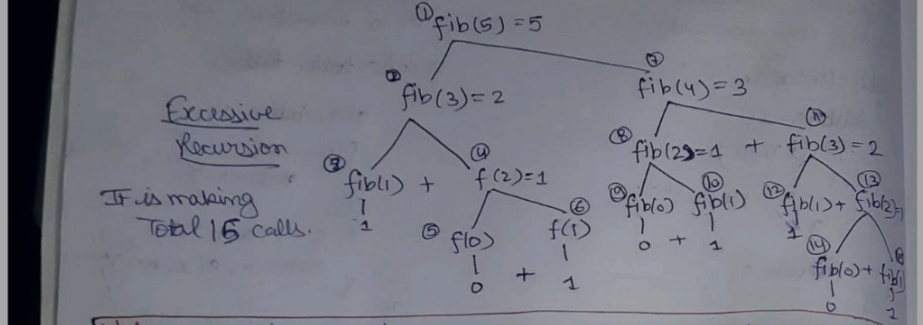
fib(n) = { 0 if n=0; 1 if n=1; fib(n-2)+fib(n-1) if n>1 }

Time complexity = O(n)

Recursion Method :-

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

Time Complexity = $O(2^n)$ approx



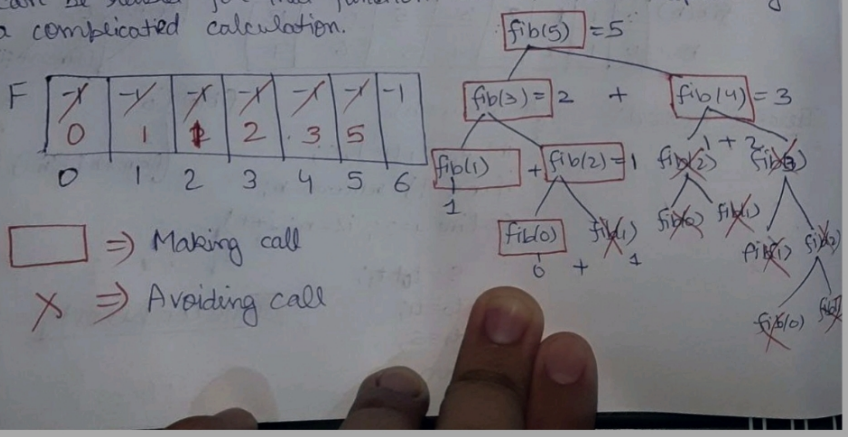
Excessive Recursion

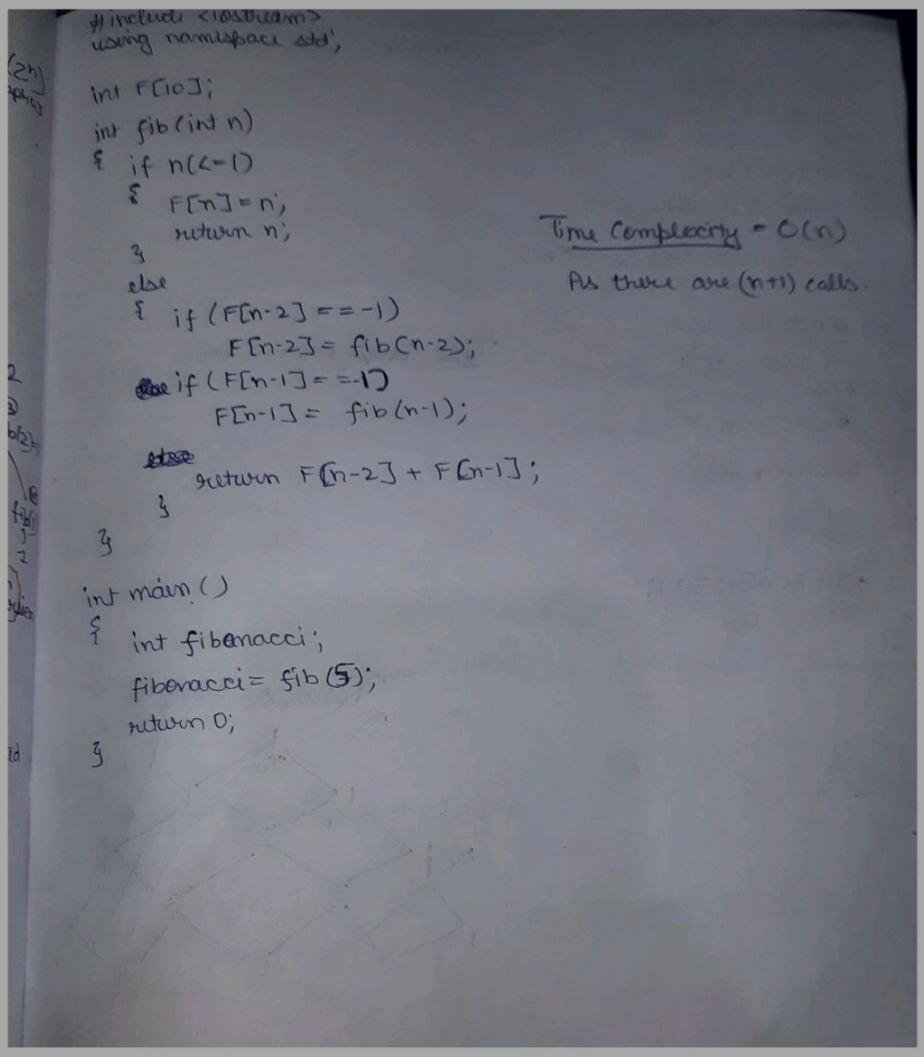
If is making Total 15 calls.

Note:- In this method a same function was called a numbers of times, and this type of recursion is known as excessive recursion

⇒ Memoization Method :-

Memoization allows a programmer to record previously calculated functions or methods ~~allowing~~ so that the same result can be reused for that function rather than repeating a complicated calculation.





```
#include <iostream>
using namespace std;

int F[10];
int fib(int n)
{ if n(<=1)
  {
    F[n] = n;
    return n;
  }
  else
  { if (F[n-2] == -1)
      F[n-2] = fib(n-2);
    if (F[n-1] == -1)
      F[n-1] = fib(n-1);

    return F[n-2] + F[n-1];
  }
}

int main()
{ int fibonacci;
  fibonacci = fib(5);
  return 0;
}
```

Time Complexity = O(n)

As there are (n+1) calls.

★ Combination Formula :- $^{n}C_{r} = \frac{n!}{r!(n-r)!}$

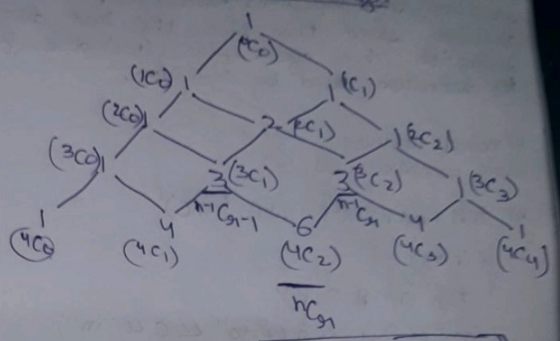
Iterator method :-

```
#include <iostream>
using namespace std;
int fact(int n)
{
   if(n==0)  return 1;
   else
       return fact(n-1)*n;
}
int nCr(int n, int r)
{
   int num, den;
   num = fact(n);
   den = fact(r) * fact(n-r);
   return num/den;
}
int main()
{
   cout << nCr(5,3);
   return 0;
}
```

Time Complexity = O(n)

Recursion Method:-

Pascal's Triangle




```
#include <iostream>
using namespace std;
int nCr(int n, int r)
{
  if(r==0 || n==r)
  {
    return 1;
  }
  else
  {
    return nCr(n-1, r-1) + nCr(n-1, r);
  }
}
int main()
{
  cout << nCr(5,3);
}
```

$nC_r = n-1C_{r-1} + n-1C_r$

Time Complexity = $O(2^n)$

☆ Tower of Hanoi Problem :-

Tower of Hanoi is a puzzle problem. Where we have three stands and n discs. Initially, discs are placed in the first stands. We have to place discs into the third or destination stand. We have to place discs into the third or destination stand, the second or auxillary stand can be used as a helping stand.

Rules:-

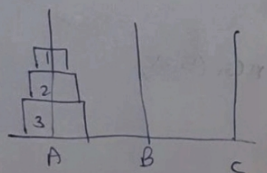
- We can transfer only one disc for each movement.
- Only the topmost disc can be picked up from a stand and no bigger disc will be placed at the top of the smaller disc.

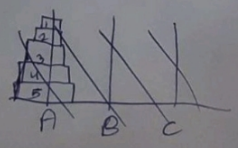
Solution Method

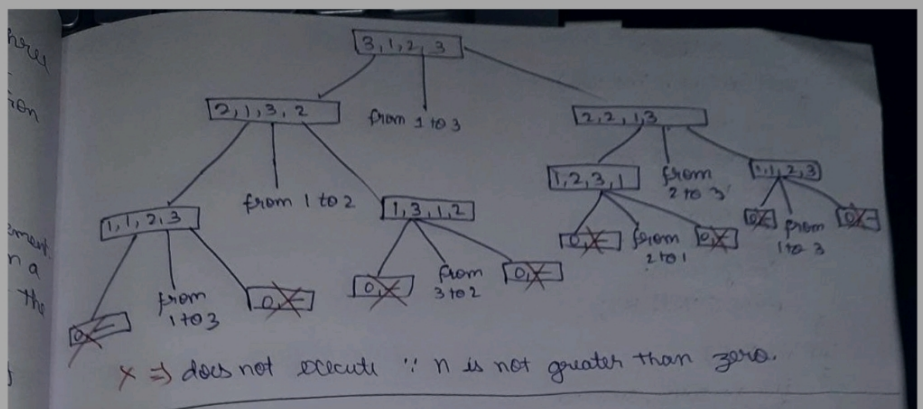
⇒This problem can be solved easily by recursion. At first, using recursion the top (n-1) discs are placed from source to auxillary stand, then place the last disc from source to destination, then again place (n-1) disc from auxillary stand to destination stand by recursion.

```
#include <iostream>
using namespace std;

int toh (int n, int A, int B, int C)
{
    if (n>0)
    { toh (n-1,A,C,B);
      cout << "from" << A << "to" << C << "\n";
      toh (n-1, B, A, C);
    }
}

int main ()
{
    toh (3, 1, 2, 3);
}
```

Time Complexity = $O(2^n)$







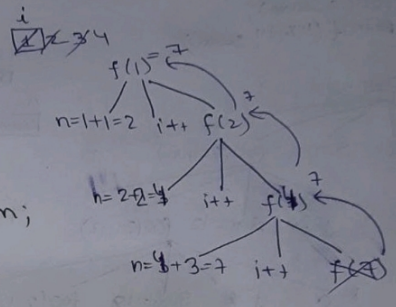
✗ ⇒ does not execute ∵ n is not greater than zero.

Quiz-1

Ques-1 Find output

```
int fun(int n)
{
  static int i = 1;
  if(n>=5) return n;
  n = n+i;
  i++;
  return fun(n);
}
int main()
{
  cout << f(1);
}
```

Time Complexity = O(n)

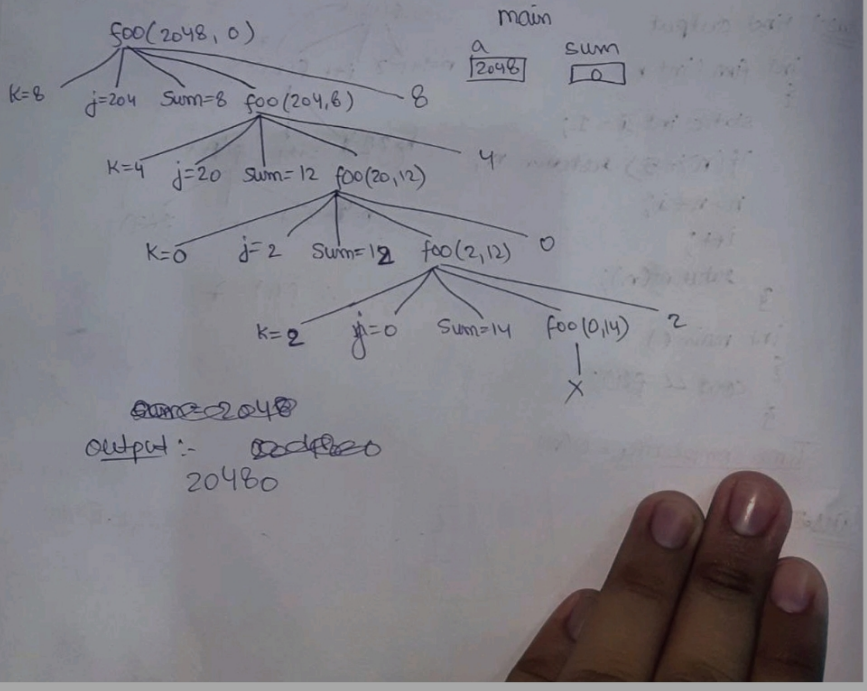


∴ f(1) = 7

Ques-2

Ques-2 Find output:-

```
void foo (int n, int sum)
{
   int k=0, j=0
   if (n==0) return;
   k= n%10;
   j= n/10;
   sum = sum + k;
   foo(j, sum);
   printf ("%d", k);
}
int main()
{
   int a=2048, sum=0
   foo(a, sum);
   printf("%d", sum);
}
```

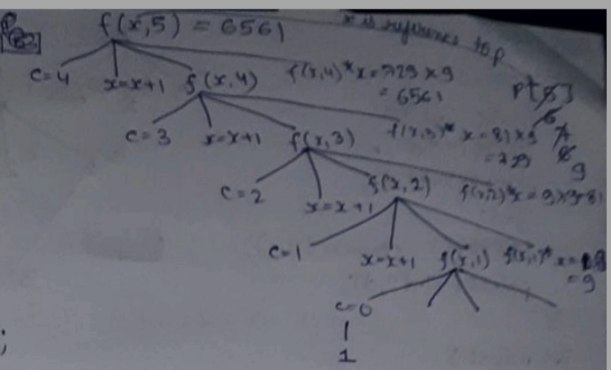


Output :-

20480

Ques 3 find output

```
int f (int &x, int c)
{
  c = c-1;
  if(c==0) return 1;
  x = x+1;
  return f(x,c)*x;
}
int main()
{ int p=5;
  printf("%d", f(p,p));
}
```



Output = 6561

Ques 4 find output :-

```
int fun (int n)
{
  int x=1, k;
  if(n==1) return x;
  for(k=1, k<n; ++k)
  {
    x = x + fun(k) * fun(n-k);
  }
  return x;
}
int main()
{
  cout << fun(5);
}
```

n	1	2	3	4	5
fun(n)	1	2	5	15	51

fun(1) = 1

fun(2):- for k=1

∴ fun(2) = x + fun(1) * fun(1)
= 1 + 1*1 = 2

fun(3):- for k=1, 2

fun(3) = x + fun(1) * fun(2) + fun(2) * fun(1)
= 1 + 1*2 + 2*1 = 5

fun(4):- for k=1,2,3

fun(4) = x + fun(1)*fun(3) + fun(2)*fun(2) + fun(3)*fun(1)
= 1 + 1*5 + 2*2 + 5*1 = 15

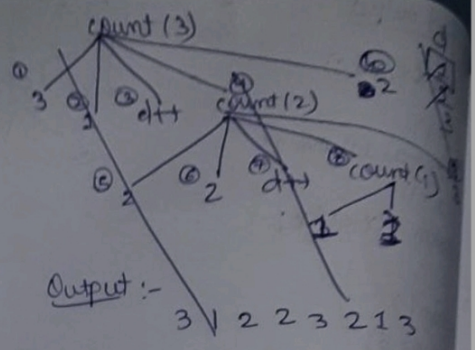
fun(5):- for k=1,2,3,4

fun(5) = x + fun(1)*fun(4) + fun(2)*fun(3) + fun(3)*fun(2) + fun(4)*fun(1)
= 1 + 1*15 + 2*5 + 5*2 + 15*1
= 51

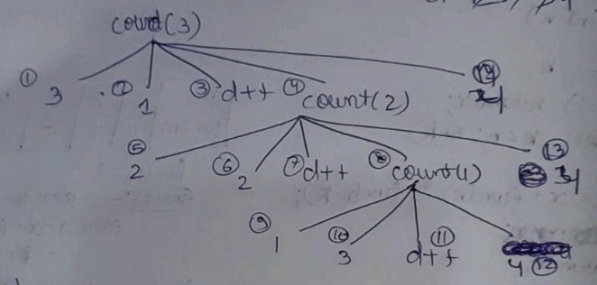
Output = 51

Ques-5 find output

```
void count(int n)
{
  static int d=1;
  printf("%d", n);
  printf("%d", d);
  d++;
  if(n>1) count(n-1);
  printf("%d", d);
}
int main()
{
  count(3);
  cout<<count(3);
}
```



Output :- 3 1 2 2 3 2 1 3



Output :- 3 1 2 2 1 3 4 4 4