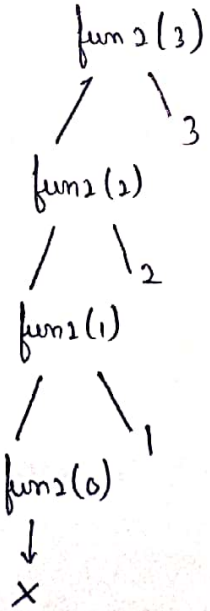
RECURSIONS

★ Syntax for Recursive function :-

```
Type fun (Parameters)
{
    if (< base Condition >)
    {
        1. ——
        2. fun (Parameter);
        3. ——
    }
}
```

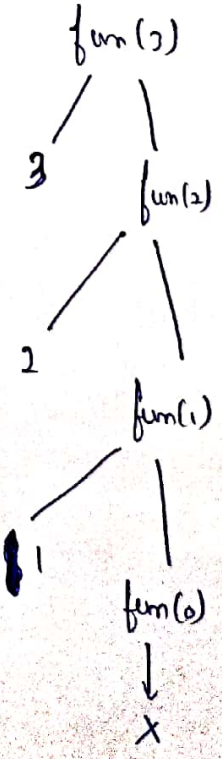
Examples :- Tracing of Recursion :-

```
Void fun2 (int n)
{ if (n>0)
  {
    1. fun2(n-1);
    2. printf ("%d", n);
  }
}
Void main()
{
    int x=3;
    fun2 (x);
}
```



Output :- 1 2 3

```
void fun1 (int n)
{ if (n>0)
  { printf ("%d ", n);
    fun1 (n-1);
  }
}
Void main()
{ int x=3;
  fun1(x);
}
```



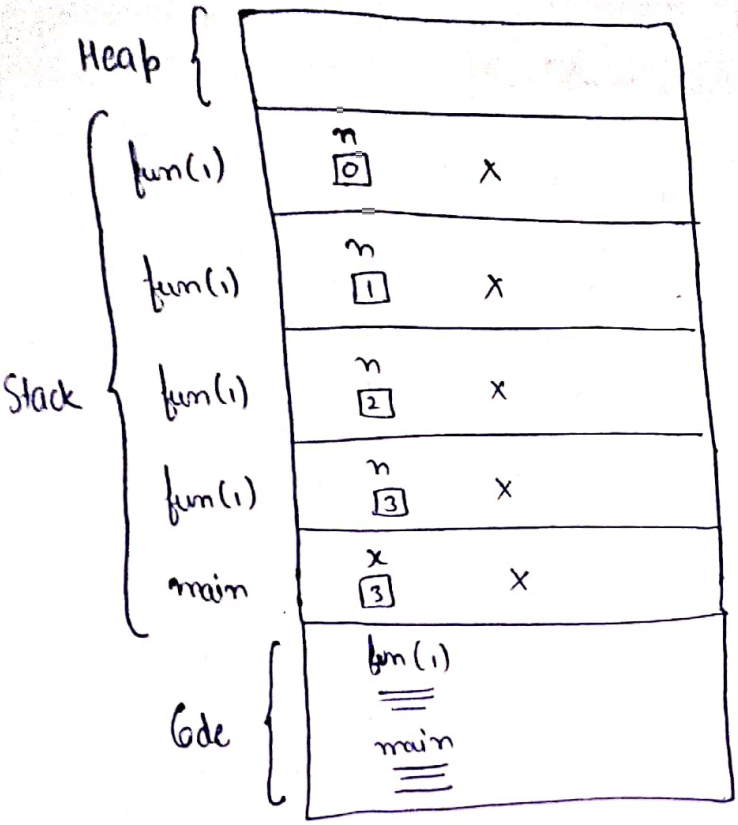
Output :- 3 2 1

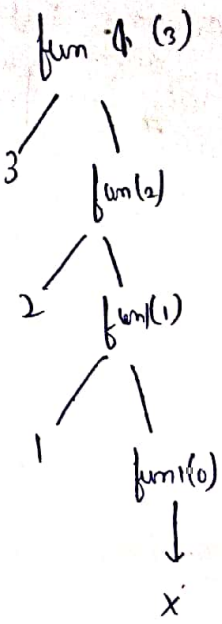
★ Generalizing Recursion :-

```
void fun(int n)
{
    if (n>0)
    {
        1. Calling
        2. fun(n-1) * 2
        3. returning ↑
    }
}
```

" There is a diff. b/w Loops & Recursions, loops & Recursions both are repeating but Loops are repeating only in Ascending / Descending order but Recursion performs both."

☆ How Recursion uses Stack :-






```
Void fun1(int n)
{
  if (n>0)
  {
    printf("%d",n);
    fun1(n-1);
  }
}
Void main()
{
  int x = 3;
  fun1(x);
}
```

Size of Stack :- Leaving main function, it has 4 activation records, so size of stack is 4.

Memory of Consumed :- $(4 * \text{size of (int)})$ or $(n * \text{size of (int)}) = O(n)$

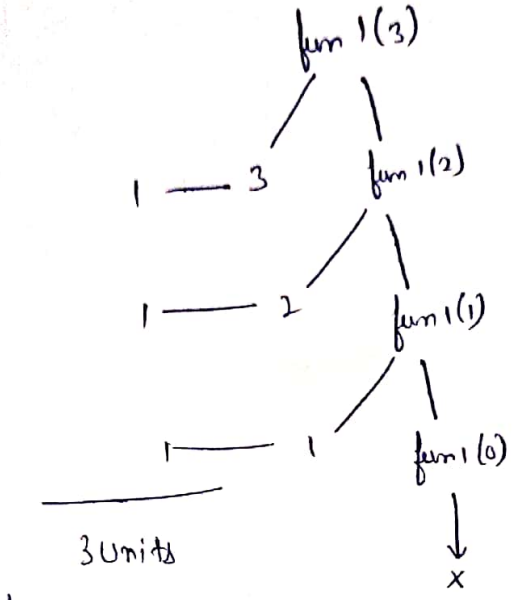
"No. of activation records depends upon no. of calls".

So, Recursive functions are memory Consuming.

Note :- It is same for Example-2

★ Recurrence Relation | Time Complexity of Recursion :-

We will assume that Every statement in a program takes one unit of time during Execution. for finding Time Complexity.



3 units

for n → n units

O(n).

```
Void fun1 (int n)
{  if (n>0)
   {
     printf ("%d", n);
     fun1(n-1);
   }
}
Void main()
{
   int x = 3;
   fun1(x);
}
```

Recurrence Formula | Relation :-

T(n) → Time taken

$$T(n) = T(n-1) + 2$$

So, $$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+2 & n>0 \end{cases}$$

Here '2' is Constant, ∴

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

T(n) —— Void fun1 (int n)

```
1 ——  {
      if (n>0)
1 ——  {
        printf ("%d", n);
T(n-1) —— fun1 (n-1);
      }
}
```

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

$$T(n) = T(n-1) + 1 \quad \text{— (1)}$$

Now, $\because T(n) = T(n-1) + 1$

$$T(n-1) = T(n-2) + 1$$

$$\therefore T(n) = T(n-2) + 2 \quad \text{— (2)}$$

Also, $T(n-2) = T(n-3) + 1$

$$\therefore T(n) = T(n-3) + 3 \quad \text{— (3)}$$

$$\vdots$$

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

Now, Assume, $n-k=0$; $n=k$

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

$$T(n) = T(0) + n$$

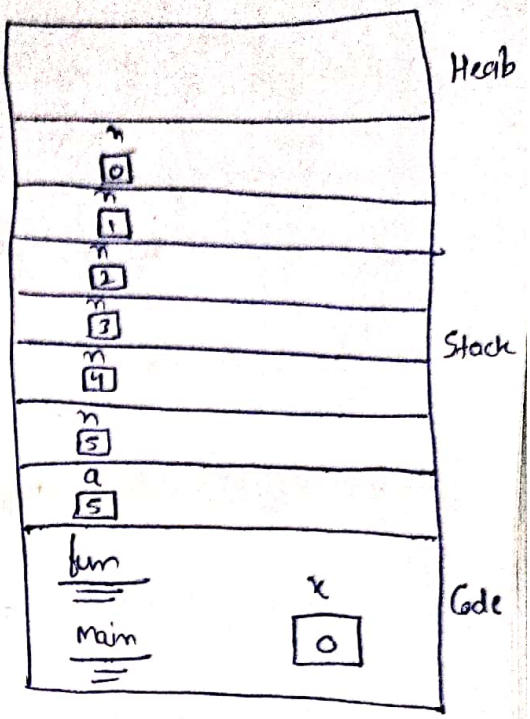
$$\boxed{T(n) = 1 + n \quad \text{—} \quad O(n)}$$

So, Time Complexity is order of 'n'.

☆ Static / Global Variables In Recursion :-

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

int main ()
{
    int a = 5;
    printf ("%d", fun (a));
}
```



fun (5) = 15

fun (4) ⑤ + 5 = 15

fun (3) ④ + 4 = 10

fun (2) ③ + 3 = 6

fun (1) ② + 2 = 3

fun (0) ① + 1 = 1

0

- Now, if we declare a variable 'x' as an static', then create function as and this 'x' will stored only once in code Section.

```
int fun1(int n)
{
static int x = 0;
    if(n>0)
    {
    x++;
    return fun1(n-1) + x;
    }
    return 0;
}
int main()
{
    int a = 5;
    printf("%d", fun1(a));
```

x
0 1 2 3 4 5

fun(5)

fun(4) (20) + 5 = 25

fun(3) (15) + 5 = 20

f(2) (10) + 5 = 15

f(1) (5) + 5 = 10

f(0) + 5 = 5

0

This will be the same if we declare 'x' as an global variable.

★ TYPES OF RECURSION :-

① Tail Recursion :- If a recursive function calls itself and that recursive call is the last statement in the function.

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

In Tail Recursion, nothing can be processed at returningtime.

For example :-

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

This is not a tail recursion.

- **Comparision of Tail Recursion with loops :-**

Tail Recursion can easily be converted in the form of loops.

In Tail Recursion :-

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

fun (3); // Calling
```

→ output :-

3 2 1

In Loops :-

```
Void fun (int n)
{
  while (n>0)
  {
    printf ("%d", n);
    n--;
  }
}

fun(3); // Calling
```

→ output :-

3 2 1

Regarding Space Complexity

Loops are more efficient than Tail Recursion.

output will be same and also the time taken by both is same of O(n) but recursive function takes space of O(n) as it creates n 'activation records' but loops only one activation record; O(1).

(2) Head Recursion :- If a recursive function calls itself and before that recursive call there is nothing to execute.

It means the function doesn't have to perform any operation during calling time, it will perform only during returning time.

```
Void fun (int n)
{ if (n>0)
  {
  → fun (n-1);
    printf ("%d", n);
  }
}
fun (3);
```

→ output :- 1 2 3

- Comparison with loops :- It will not easily converted to loops but can be converted.

```
Void fun (int n)
{
  int i=1;
  while (i<=n)
  {
    printf ("%d", i);
    i++;
  }
}
fun (3);
```

→ output :- 1 2 3

③ Tree Recursion :-

Linear Recursion :-

If a recursive function calls itself only one time than it will be Linear Recursion.

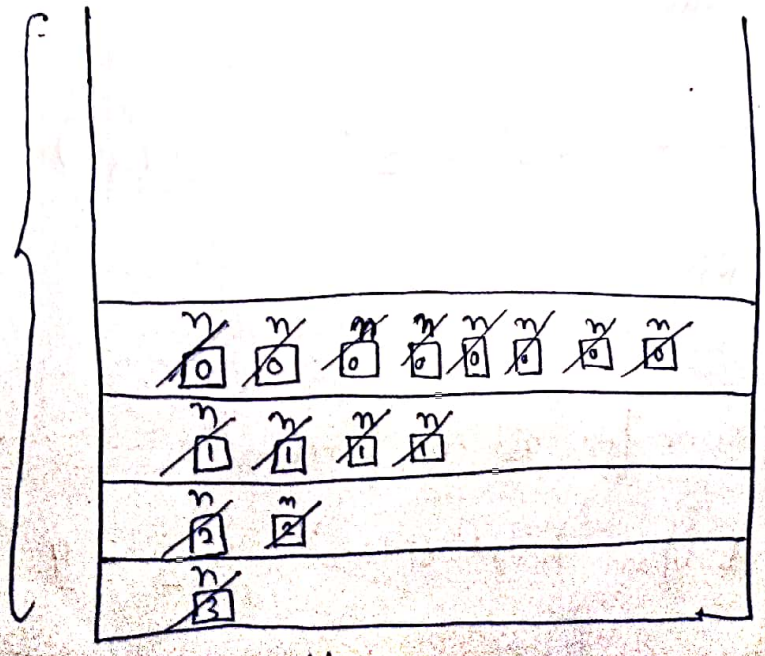
```
int fun(n)
{
  if(n>0)
  {
    ===
    ===
    fun(n-1);
    ===
  }
}
```

Tree Recursion :-

If a recursive function calls itself more than one time than it will be Tree Recursion.

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

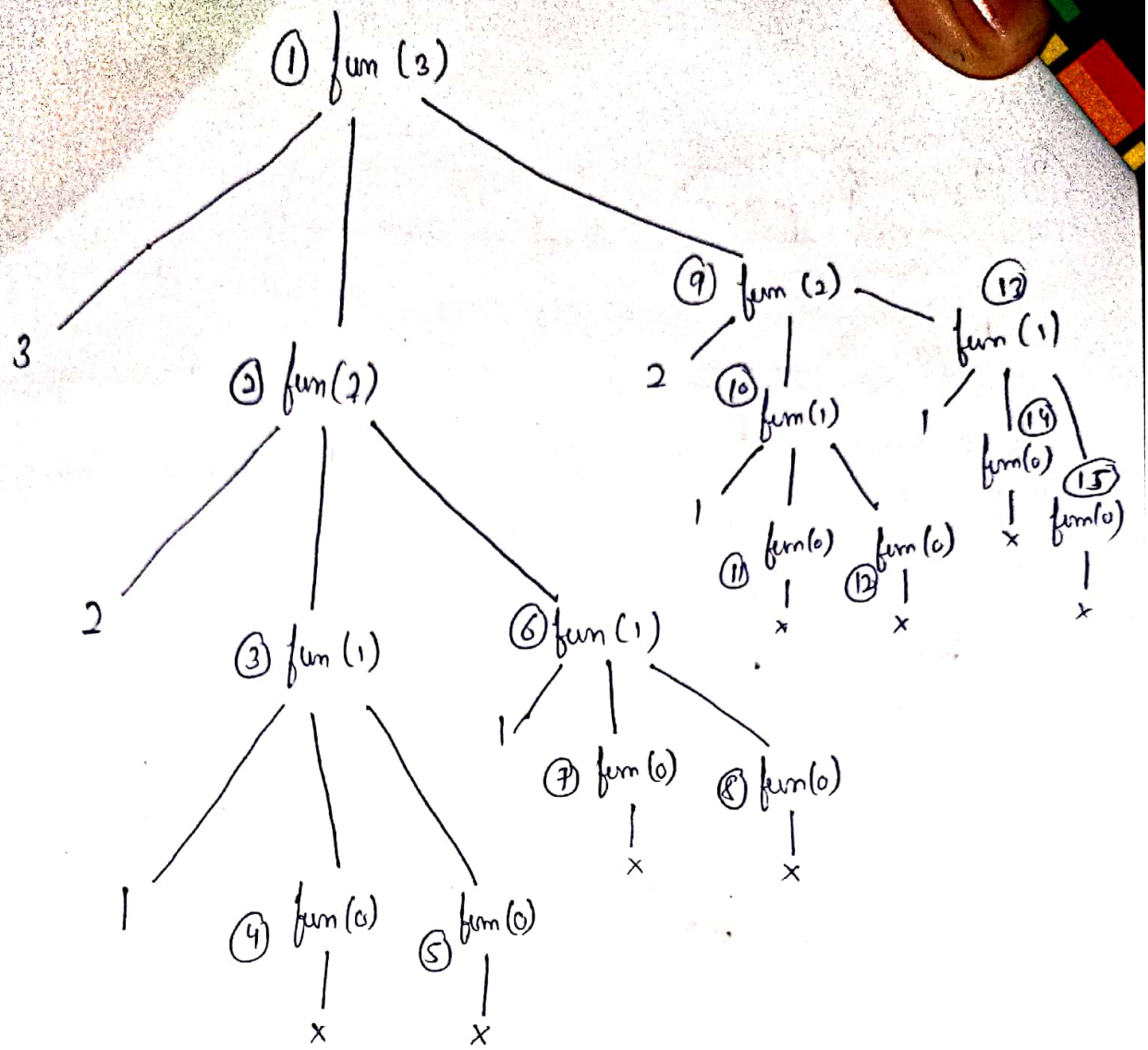
How it used Stack :-




```
void fun(int n)
{
  if (n>0)
  {
1→ printf("%d",n);
2→ fun(n-1);
3→ fun(n-1);
  }
}
---
fun(3);
```

Output :- 3, 2, 1, 1, 2, 1, 1

Tracing :-



For n = 3 :-

At level 1 :- 1 call $= 2^0$
At level 2 :- 2 calls $= 2^1$
At level 3 :- 4 calls $= 2^2$
At level 4 :- 8 calls $= 2^3$

} G.P Series

For n :- $2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n = 2^{n+1} = O(2^n)$

So, time Complexity is $O(2^n)$

Note :- It is not Same for all the functions.

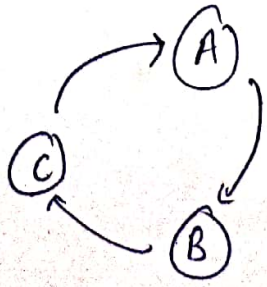
- "Space Complexity depends upon height of tree", for 3 :- 4 (i.e. n+1)
  ∴ $= O(n)$

(4) **Indirect Recursion** :- In indirect Recursion there may be more than one function and calling one another in a circular fashion.

Syntax :-

```
Void A (int n)
{ if (< Base Conditions>)
  {
    ==
    B (n-1);
  }
}
Void B (int n)
{
  if (< Base Conditions)
  {
    ==
    A (n-3);
  }
}
```

Example :-



→ Skelton of indirect function.

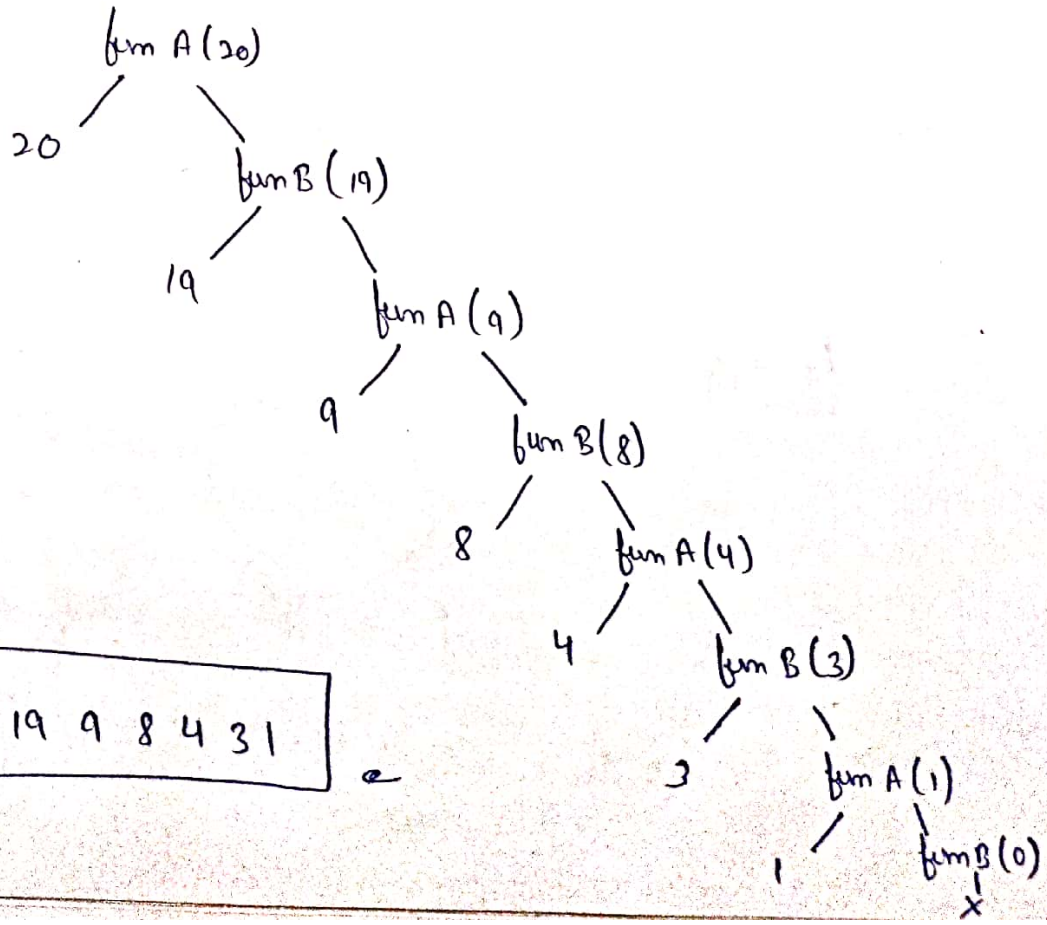
[Conditions must be satisfied i.e. base conditions]

Code :-

```
void funB (int n); // Prototype declaration bcz must be declared first then in program
Void funA( int n)
{ if (n>0)
  {
    printf ("%d", n);
    funB (n-1);
  }
}
Void funB (int n)
{
  if (n>1)
  {
    printf ("%d",n);
    funA (n/2);
  }
}
```

funA(20); // Calling

Tracing :-

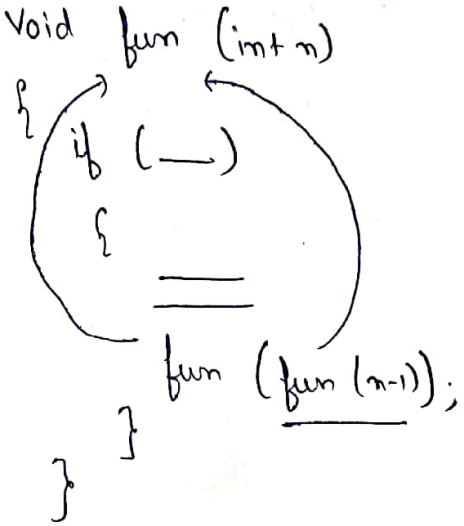


Output :- 20 19 9 8 4 3 1

(5) Nested Recursion :-

In this, a Recursive function pass as a parameter as a recursive call.

Syntax :-



This is recursion inside recursion, thats why known as Nested Recursion.

Example :-

```
fun (95)
  |
fun (fun (95+11)) ← 96 = fun (106)
  ↕
fun (96)
  |
fun (fun (96+11))
  ↕
fun (97)
  |
fun (fun (97+11))
  ↕
fun (98)
```

```
fun (99)
  |
fun (100)
  |
fun (101) = 91 → output
```

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

fun (95); // calling
```

Program - 1 :- write a program for sum of 'n' natural numbers.

⇒ $1+2+3+4+\cdots+n$

$Sum(n) = \underline{1+2+3+4+\cdots+(n-1)}+n$

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

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

Code :-

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

int main()
{
  int H;
  H = Sum (35);
  printf ("%d", H);
}
```

we can do this with iteration method (i.e. loops) also.

```
int Sum (int n)
{
    int i, s=0;
    for (i=1; i<=n; i++)
        s=s+i;
    return s;
}
int main ()
{
    int s;
    s= Sum (30);
    printf ("%d", s);
```

Time Complexity for recursion & iterative will be same of O(n) but actually for mathematical purposes recursions are easy to understand but loops takes less space than Recursion.

- One more Alternative Method and actually fast :-

```
int Sum (int n)
{
    return n* (n+1) /2;
}
int main ()
{
int M = Sum (30);
    printf ("%d", M);
}
```

Time Complexity :- O(1)

Program - 2 :- Write a program for factorial of N.

⇒ $n! = 1 * 2 * 3 * \ldots * n$

$5! = 1 * 2 * 3 * 4 * 5 = 120$

$fact(n) = \underline{1 * 2 * 3 * \ldots * (n-1)} * n$

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

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

Code :-

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

int main ()
{
  int u;
  u = fact (7);
  printf ("%d", u);
}
```

Here, if $n < 1$, then it will go to infinte loop but a recursive function stops at one time when it go to infinite loop bcz of stack overflow, so to avoid this we have to make a Condition for that if $(n < 1)$

return 0;

We Can do this by using loops also

```
int Ifact (int n)
{
    int f = 1;
    int i;
    for (i = 1; i <= n; i++)
        f = f * i;
    return f;
}
int main()
{
    int f;
    f = Ifact (5);
    printf ("%d", f);
}
```

Time Complexity for both :- $O(n)$

Program - 3 :- Write a program for Exponent using Recursion.

⇒ $2^5 = 2 * 2 * 2 * 2 * 2$

$m^n = m * m * m * \text{---- for } n \text{ times}$

$Pow(m,n) = \underline{(m * m * m * \text{--- } (n-1) \text{ times})} * m$

$Pow(m,n) = Pow(m, n-1) * m$

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

Code :-

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

int main()
{
  int u = Power(2,9);
  printf("%d", u);
}
```

Tracing :-

Pow(2,9)
|
Pow(2,8) * 2
|
Pow(2,7) * 2
|
Pow(2,6) * 2
|
Pow(2,5) * 2
|
Pow(2,4) * 2
|
Pow(2,3) * 2
|
Pow(2,2) * 2 = $2^2 * 2 = 2^3$
|
Pow(2,1) * 2 = $2 * 2 = 2^2$
|
Pow(2,0) * 2 = $1 * 2 = 2$

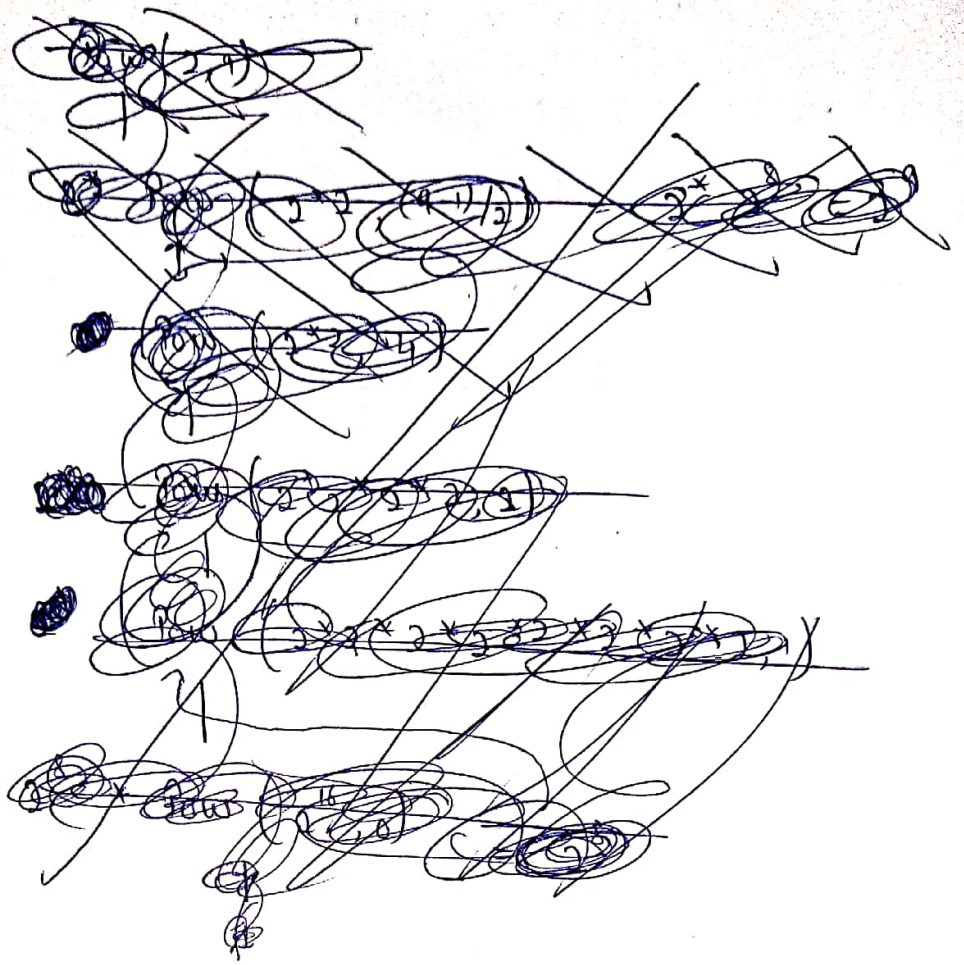
Time & Space Complexity :- O(n)

Now, we see it performs total 9 multiplications if we want to reduce, then we can do as follows

if, n is even → $2^8 = (2^2)^4 = (2*2)^4$

n is odd → $2^9 = 2 * (2*2)^4$

~~Trace for (2,9) = 2^9~~



Tracing :-

Pow (2, 9)

$2 * \text{Pow}(2^2, 4) = 2 * 2^8 = 2^9$

Pow (2^4, 2)

Pow (2^8, 1)

$2^8 * \text{Pow}(2^{16}, 0) = 2^8$

⇒ This is the faster version

So, there are total of 6 multiplications.

Code :-

```
int Power1 (int m, int n)
{
    if (n == 0)
        return 1;
    else if (n % 2 == 0)
        return Power1 (m * m, n/2);
    else
        return m * Power1 (m * m, (n-1)/2);
}
int main ()
{
    int H = Power1 (2, 8);
    printf ("%d", H);
}
```

Program-4 :- Write a program for Taylor Series Using Recursion.

Taylor Series e^x

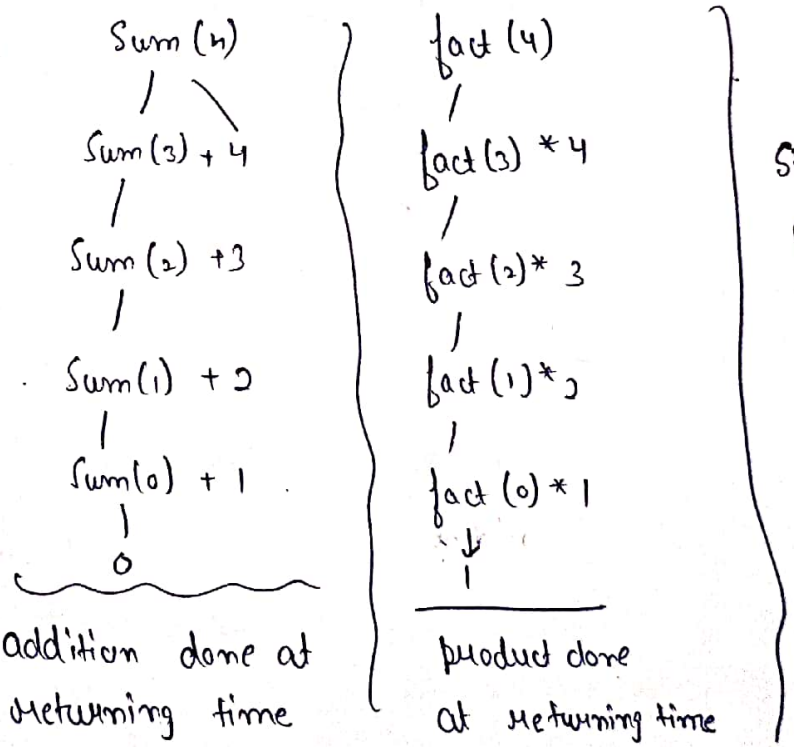
$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + n \text{ terms}$$

We know, $Sum(n) = 1+2+3+\dots+n = Sum(n-1)+n$

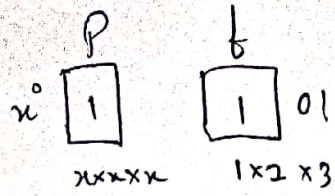
$fact(n) = 1*2*3*\dots*n = fact(n-1)*n$

$Pow(x,n) = x*x*\dots \text{ n times} = Pow(x,n-1)*x$

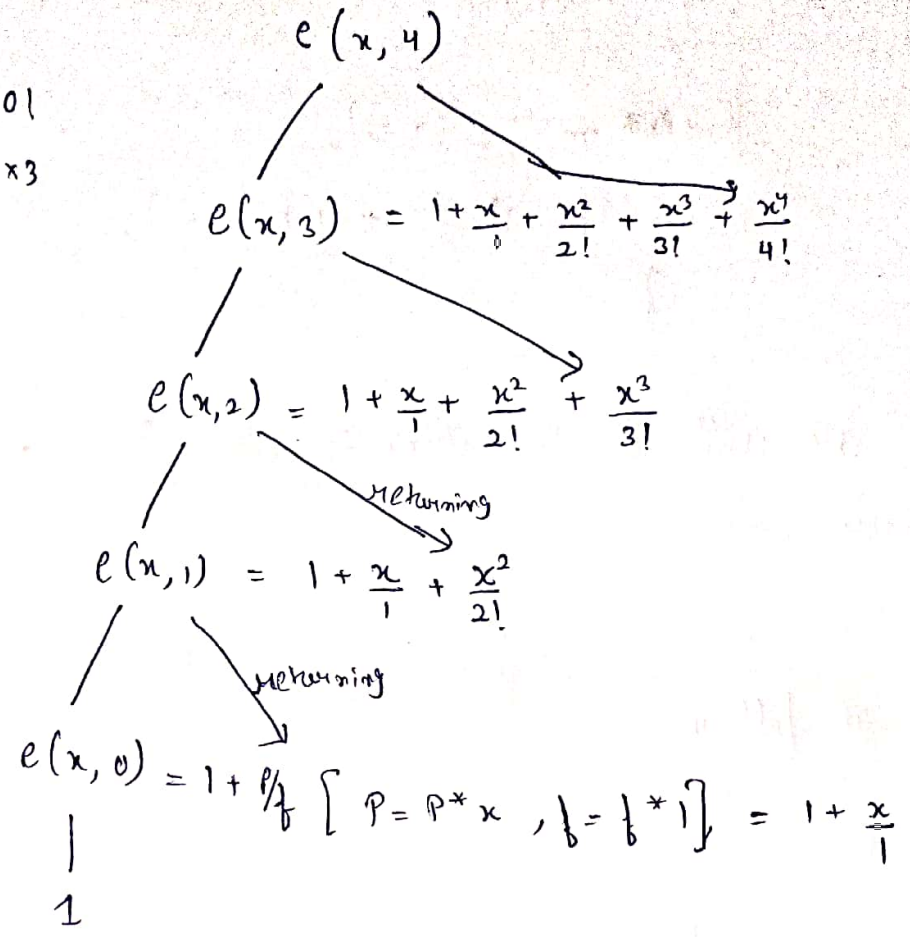
Now, Let us take



For Taylor Series, a function must perform three functions but it have to return only one function, so, when we have to involve multiple values then we can use static variable.



Here, P & f are static variables



Code :-

```
float e (int x, int n)
{
    static double P=1, f=1;
    double H;
    if (n==0)
        return 1;
    else
    {
        H = e(x, n-1);
        P = P*x;
        f = f*n;
        return H + P/f;
    }
}
```

```
int main()
{
float m = e(1, 10);
    printf("%.1f", m);
}
```

Now, We will do this using Horner's Rule, and this method is faster.

Now, We know, with previous method, there will no. of multiplications as follows :-

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

For upto 4 terms :- no. of multiplications ⇒

$$0 \quad 0 + \frac{x \times x}{1 \times 2} + \frac{x \times x \times x}{1 \times 2 \times 3} + \frac{x \times x \times x \times x}{1 \times 2 \times 3 \times 4}$$

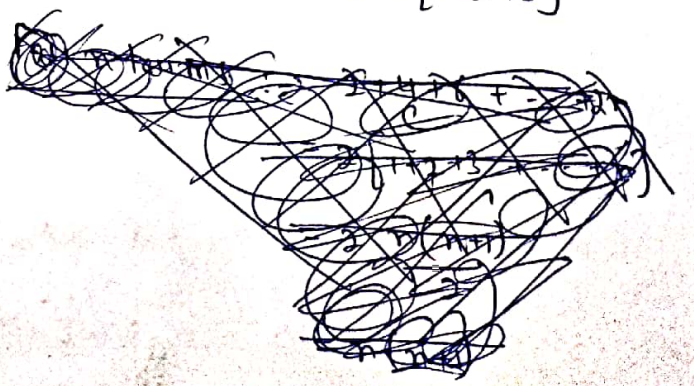
no. of multiplications :- $2 + 4 + 6 + \cdots$

$\Rightarrow 2[1+2+3]$

For 4 terms :- 12

For 5 terms :- 20

For n terms :- $n(n-1)$



So, Time Complexity for previous method will be $O(n^2)$

Now, To Reduce No. of Multiplications,

$$1 + \frac{x}{1} + \frac{x^2}{1\times2} + \frac{x^3}{1\times2\times3} + \frac{x^4}{1\times2\times3\times4}$$

$$1 + \frac{x}{1}\left[1 + \frac{x}{2} + \frac{x^2}{2\times3} + \frac{x^3}{2\times3\times4}\right]$$

$$1 + \frac{x}{1}\left[1 + \frac{x}{2}\left[1 + \frac{x}{3} + \frac{x^2}{3\times4}\right]\right]$$

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

* * * *

So, it has done only 4 multiplications.

So, for n :- O(n)

Code :-

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

int main ()
{
    printf ("%.f \n", e(1, 10));
}
```

Code for iterative :-

```
double e (int x, int n)
{
    double S=1;
    int i;
    double num=1;
    double den=1;
    for (i=1; i<=n; i++)
    {
        num = num*x;
        den = den*i;
        S = S + num/den;
    }
    return S;
}
int main()
{
    printf ("%f", e(1,10));
}
```

Program - 5 :- Write a program for fibbonacci Series Using Recursion.

⇒

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

$$fib(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-2) + fib(n-1) & n>1 \end{cases}$$

Code for iterative :-

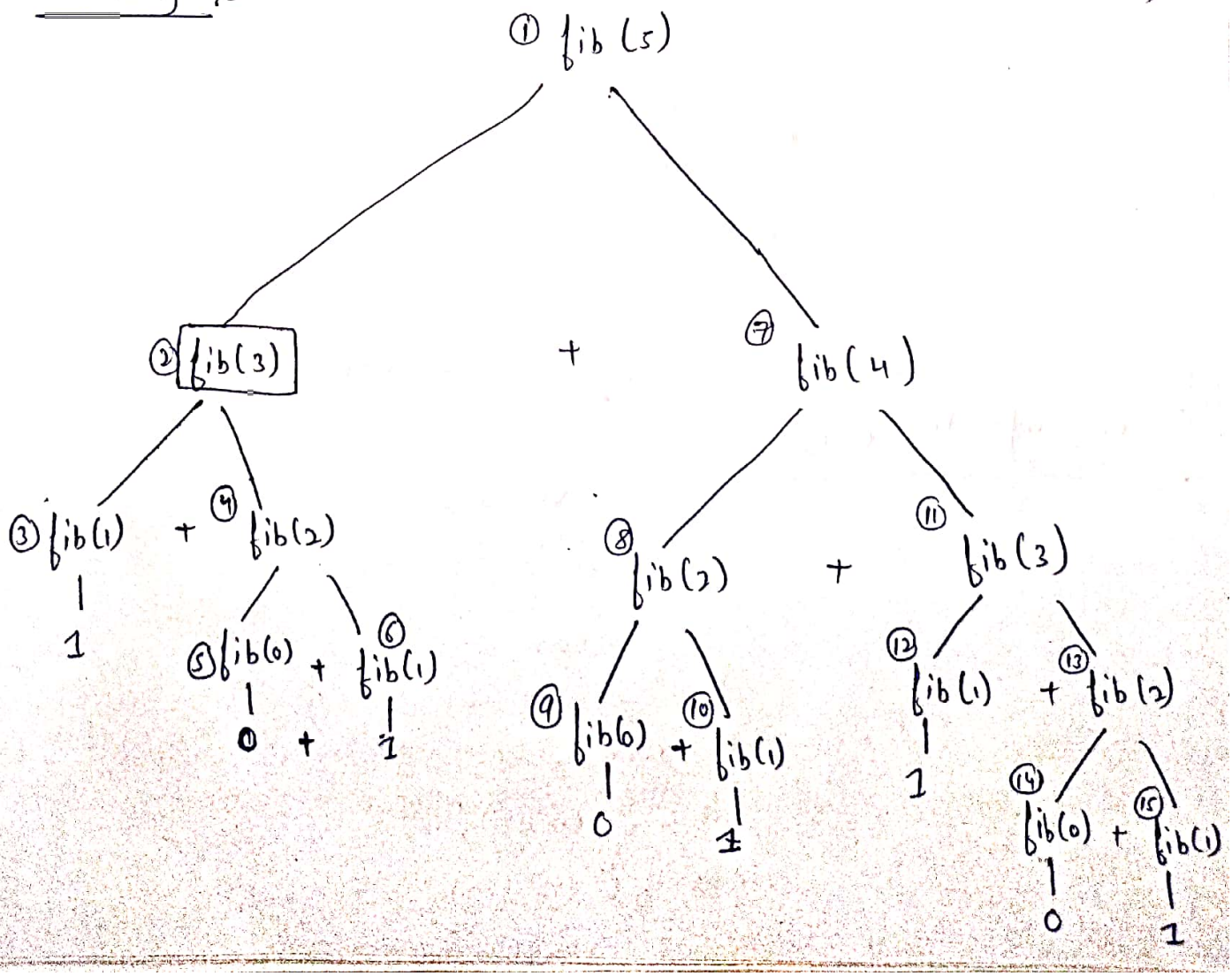
```
int fib (int n)
{ int t0 = 0 ; t1 = 1 ; s = 0 ; i
  if (n <= 1)
   return n ;
  else
  { for ( i=2 ; i<=n ; i ++)
    { s = t0 + t1 ;
      t0 = t1 ;
      t1 = s ;
    }
  } return s ;
}
int main ( )
{
  int n = fib (5) ;
  printf ("%d", n) ;
}
```

Time Complexity :- O(n)

Code for Recursion :-

```
int fib(int n)
{
    if (n<=1)
        return n;
    else
        return fib(n-2) + fib(n-1);
}
int main()
{
    int n = fib(5);
    printf("%d", n);
}
```

Tracing :-



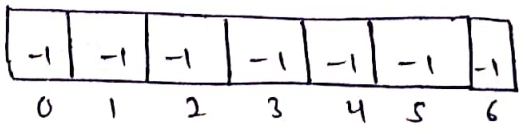
for fib(05) → 15 Calls
fib (4) → 9 Calls
fib (3) → 5 Calls

For this, **Time Complexity :- $O(2^n)$**

This is so time Consuming, so we will find a new way to make this faster.

Here we can see that Function call itself multiple times for the same value (i.e. fib(3), fib(2), fib(1), fib(0)) which we called as Excessive Recursion, so, fibonnaci series is Excessive Recursion.

Now, we will reduce this Excessive, to do this we need Static | Global Array of any length. as follows.



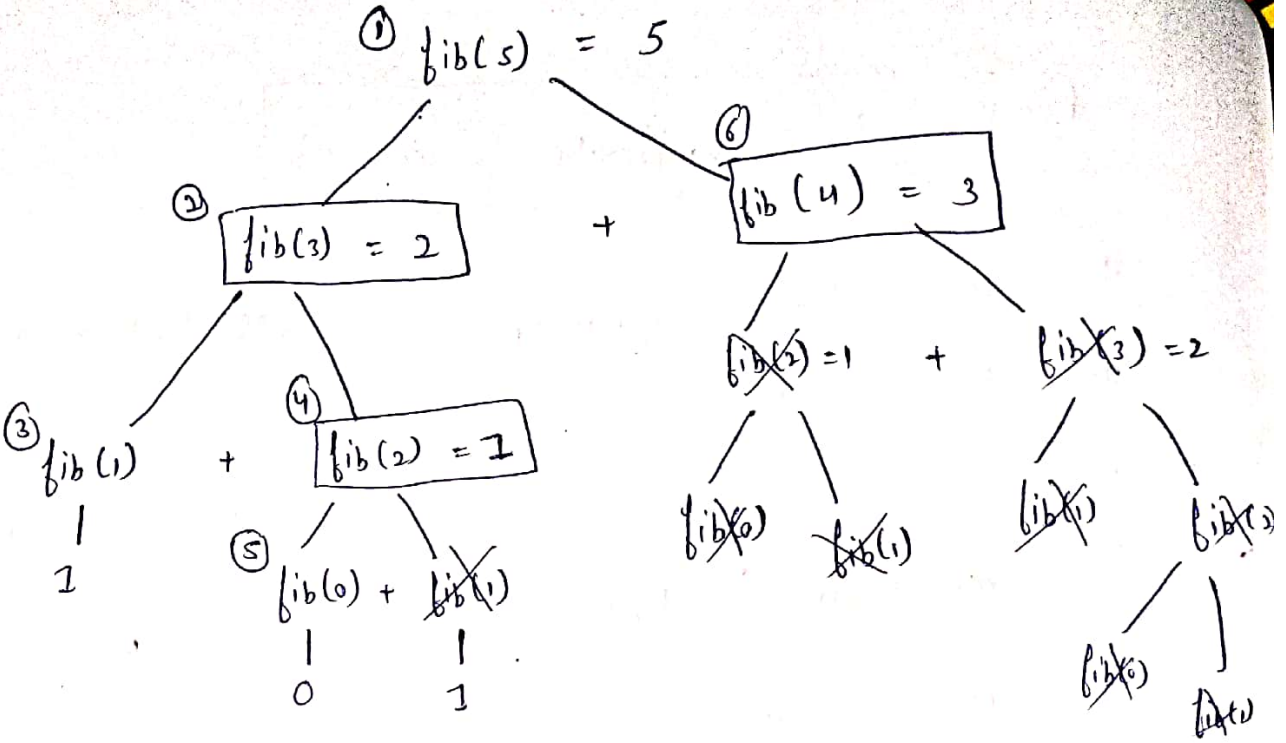
Let us initialize with (-1), so we dont know the value here

So, this Reduction of time is bcz of Storing values in an array holding the results, so that avoid Excessive Calls, so this approach is known as Memoization. Storing the results of function Call, so that they can be Utilized again when we need the same call Called Memoization.

Tracing :-

f

0	1	2	3	4	5	6
~~0~~	~~1~~	~~1~~	~~2~~	~~3~~	~~5~~	-1



So, it is making 6 calls for fib(5)

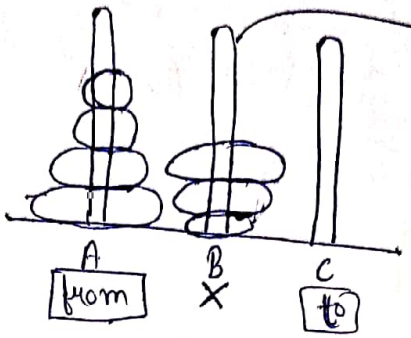
So, for n :- (n+1) calls ⇒ Time complexity :- O(n)

Code :-

```
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);
    f[n] = fib(n-2) + fib(n-1);
    return f[n-2] + f[n-1];
  }
}
int main()
{
  int i;
  for(i=0; i<10; i++)
    f[i] = -1;
  printf("%d", fib(5));
}
```

· Here we make a call.

Program-6 :- Write a program for Tower of Hanoi Problem.



We have to transfer these discs from A to C one by one. And the another Condition is that no larger disc will kept on Smaller Disc.

Recursion

TOH(1, A, B, C)

★ Move disk from A to C using B.

TOH(2, A, B, C)

★ TOH(1, A, C, B) → Using tower Here.

★ Move Disk from A to C Using B.

★ TOH(1, B, A, C)

Recursion

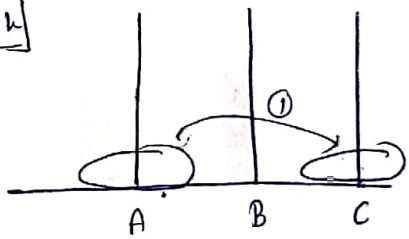
TOH(3, A, B, C)

★ TOH(2, A, C, B)

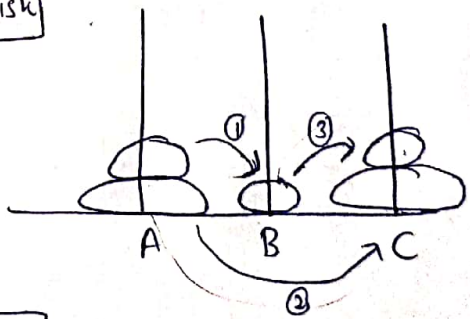
★ Move Disk from A to C using B

★ TOH(2, B, A, C) → It is also recursive

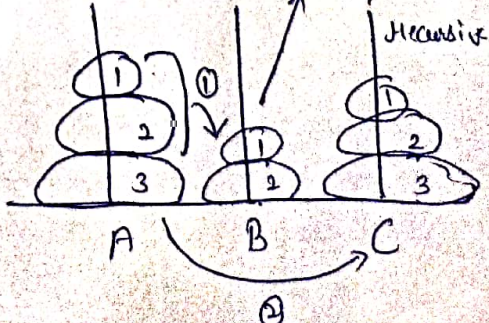
1 Disk



2 Disk



3 Disk

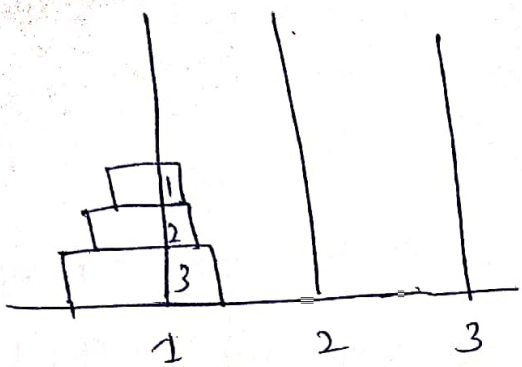


for n-disk :- TOH(n, A, B, C)

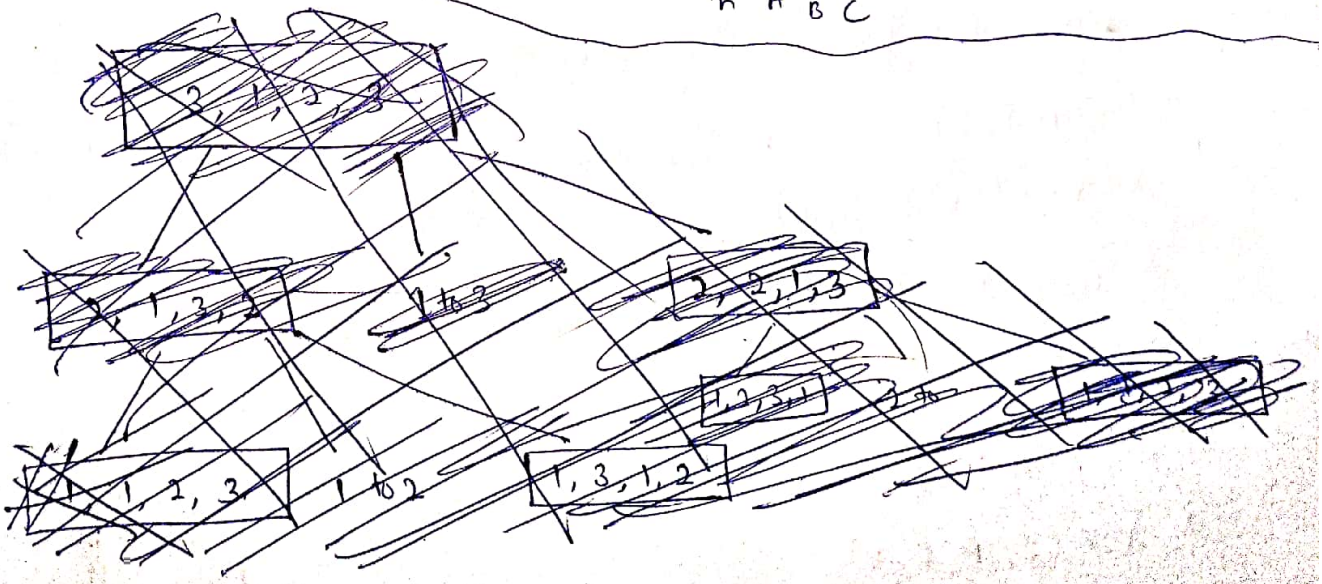
1. TOH(n-1, A, C, B)
2. move Disk from A to C using B
3. TOH(n-1, B, A, C)

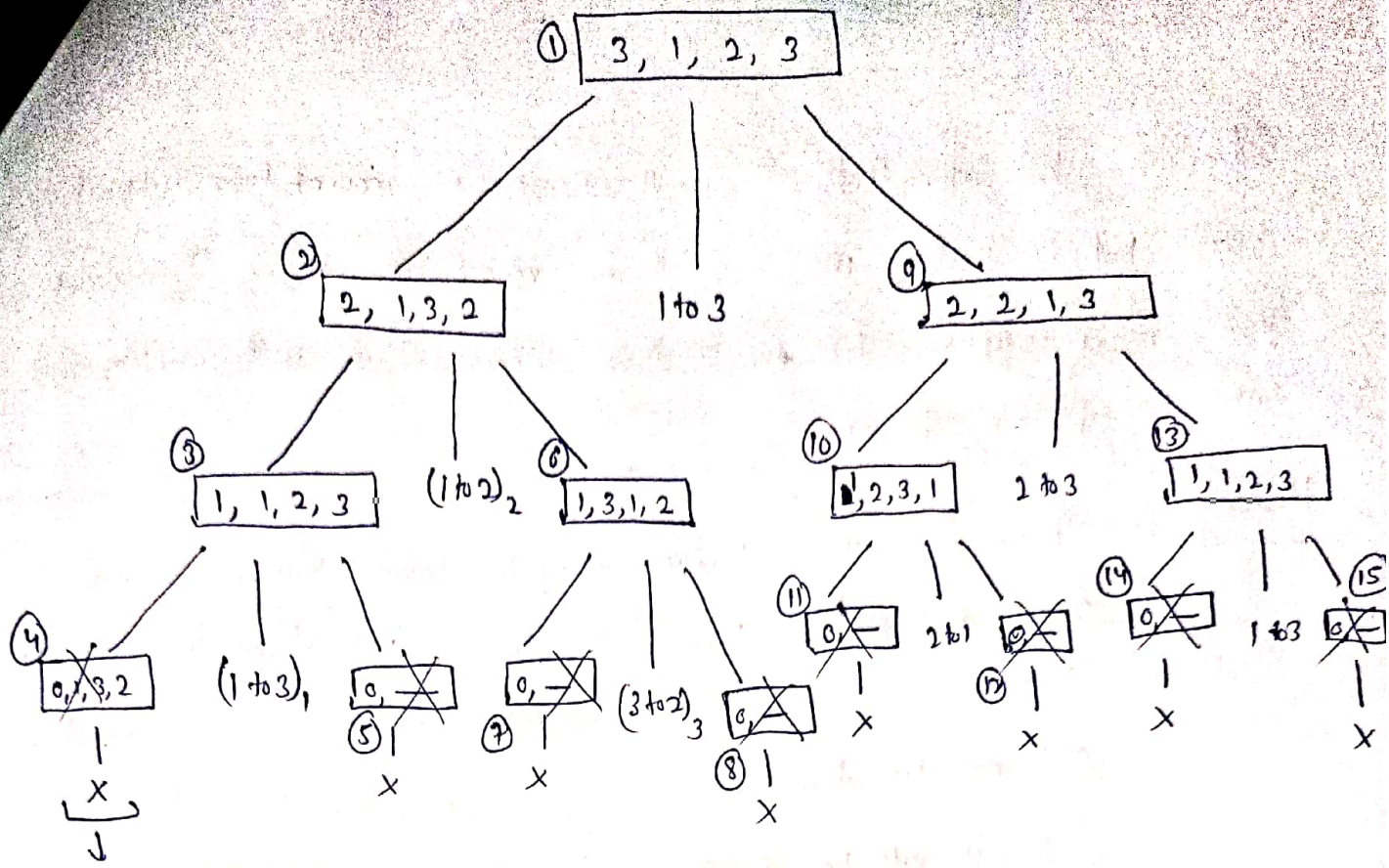
Tracing :-

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



ToH (3, 1, 2, 3);
↑ ↑ ↑ ↑
n A B C





bcz these steps are useless as $n = 0$

(1,3), (1,2) (3,2) (1,3) (2,1) (2,3) (1,3)

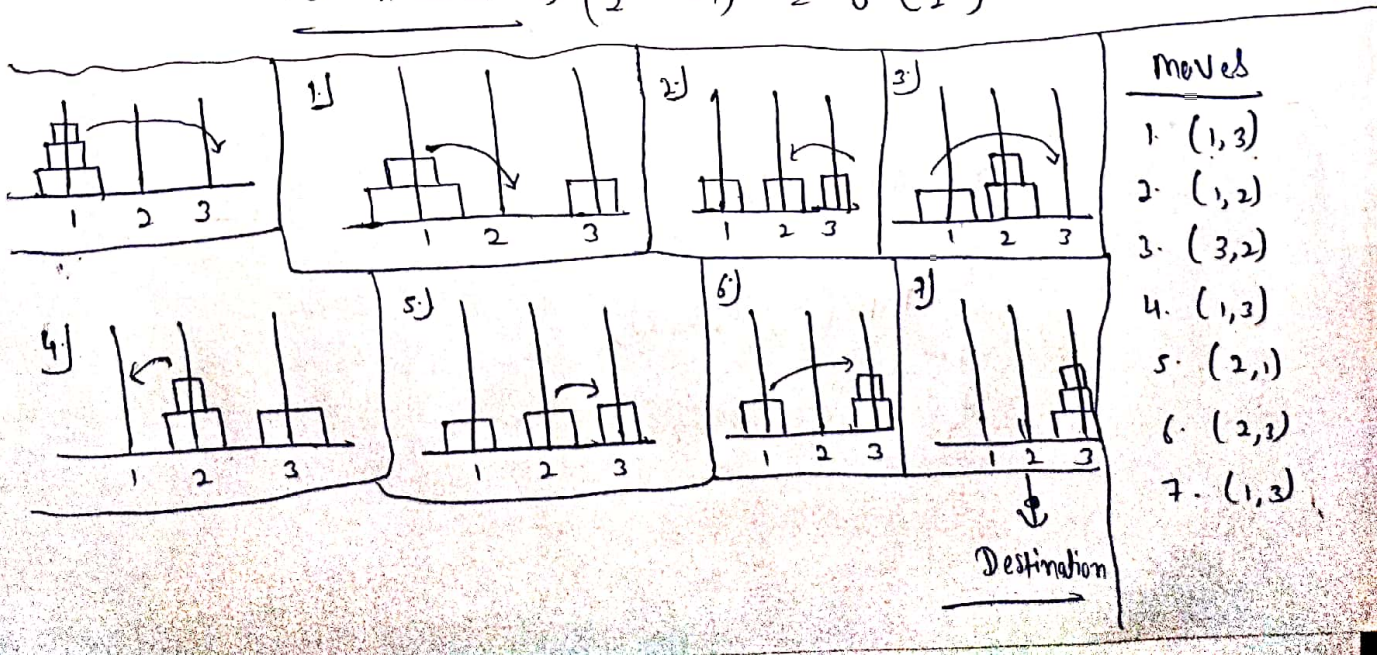
↓

These are the steps if we follow then Problem will be solved.

For 3 Disks → 15 Calls $= 2^4 - 1$

For 2 Disks → 7 Calls $= 2^3 - 1$

∴ For n Disks → $(2^{n+1} - 1) \doteq O(2^n)$



Definition was saying that, this is recursive (i.e. moving two disks at a time) but in reality we move them one by one. Hence Recursion gives no. of steps to solve this problem otherwise we will wasting our time by repeating steps again.

History :- There is a Hanoi tower, may be somewhere in china near a temple, Some People try to Put plates from tower 1 to tower 3 so that larger disk doesn't come over smaller one and transfering one by one and when it will done then it will be doom's day (i.e. last day on earth).

Code :-

```
Void TOH (int n, int a, int b, int c)
{
  if (n>0)
  {
    TOH (n-1, a, c, b);
    printf (" move %d to %d \n", a, c);
    TOH (n-1, b, a, c);
  }
}
int main()
{
  TOH (3, 1, 2, 3);
}
```

Program - 7 :- Write a program for ${}^{n}C_{r}$ using Recursion.

(Selection formula) → no. of ways for finding no. of subsets in a set.

Eg :- A B C D E F G → Select only 3

Same → A B C
A B D
X A C B → This is Permutation.

$$ {}^{n}C_{r} = \frac{n!}{r!\,(n-r)!} $$

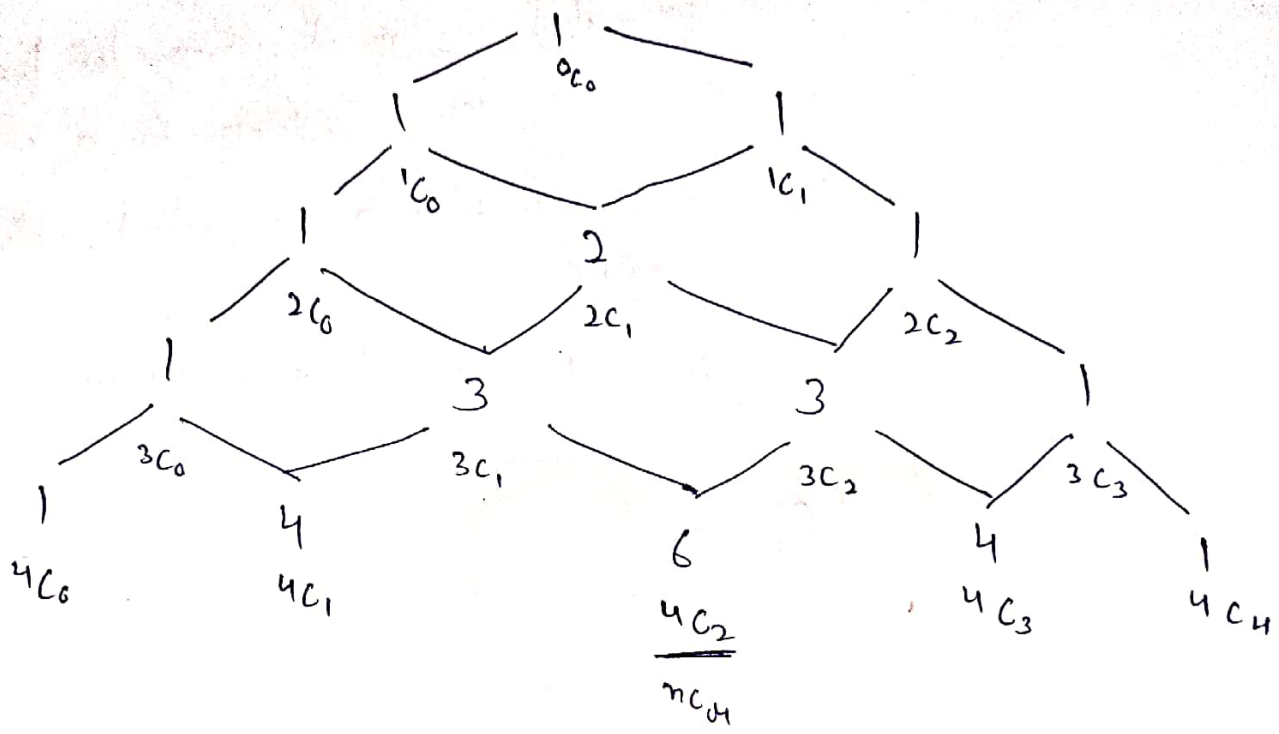
$${}^{5}C_{r=0-5} = {}^{5}C_{0}\ {}^{5}C_{1}\ {}^{5}C_{2} \cdots {}^{5}C_{5}$$

Simple function for this :-

```
int C (int n, int r)
{ int t1, t2, t3;
  t1 = fact (n);
  t2 = fact (r);
  t3 = fact (n-r);
  return t1 / (t2 * t3);
}
```

Time Complexity :- $O(n)$

Now, Recursive function for this, first we have to understand Pascals triangle law.



$^4C_2 \rightarrow {}^3C_1 + {}^3C_2$

for n :- $^nC_H \rightarrow {}^{n-1}C_{H-1} + {}^{n-1}C_H$ ←

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

For Simple function :-

Code :-

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

int nCH (int n, int H)
{
    int num, dem;
    num = fact(n);
    dem = fact(H) * fact(n-H);
    return num/dem;
}
int main()
{
    printf("%d", nCH(5,1));
}
```

Code for Recursive function :-

```
int NCR (int n, int H)
{
    if (n==H || H==0)
        return 1;
    else
        return NCR(n-1, H-1) + NCR(n-1, H);
}
int main()
{
    printf("%d", NCR(5,2));
}
```

→ This is a recursive function for $^{n}C_{R}$ problem

Practise Set

Q-① Consider the following 'C' function

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

The value returned by f(1) is

A-① :-

f(1)

/

n=2; i=1

/i++

n=4; i=2

/i++

n=7; i=3

/

return (7) ✓

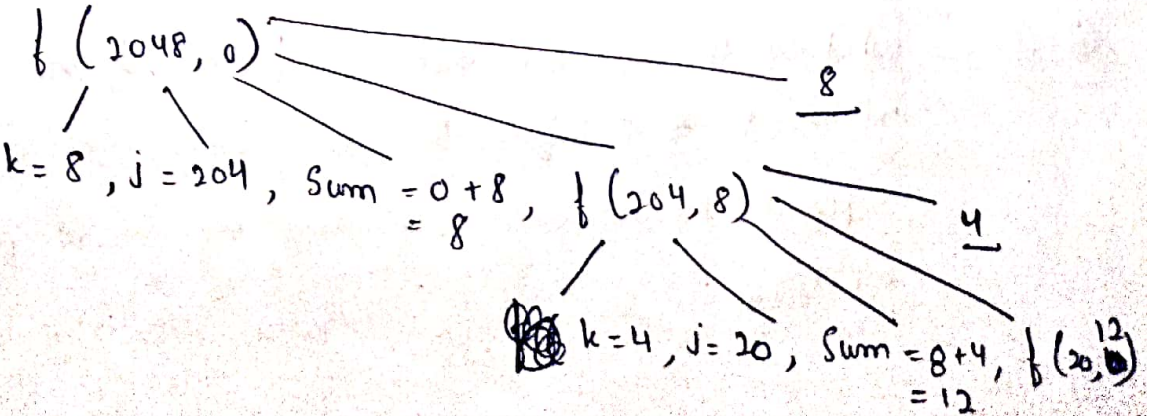
∴ f(1) ⇒ output = 7

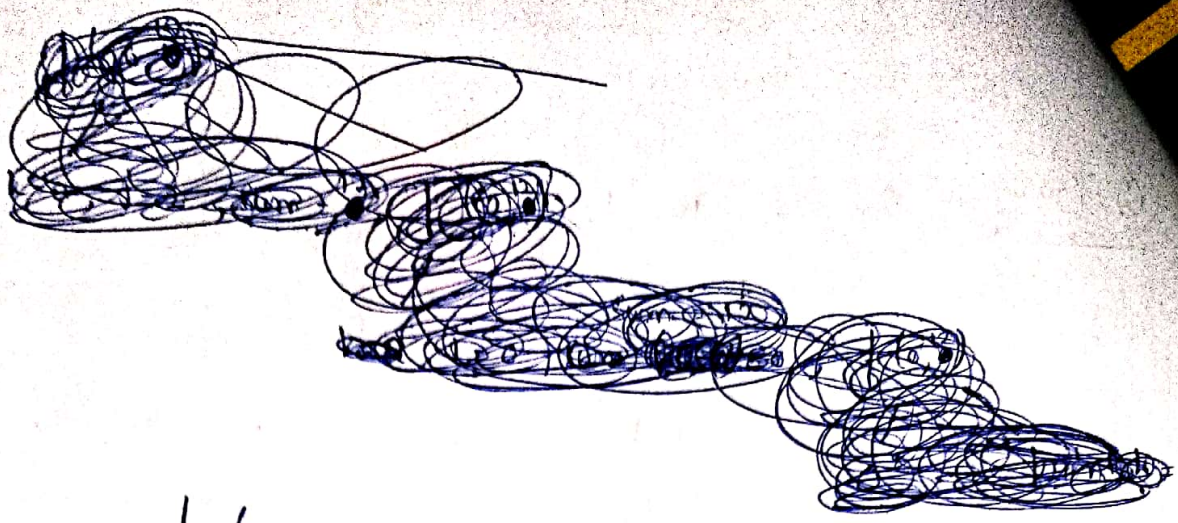
Q-(2) :- Consider the following C- program

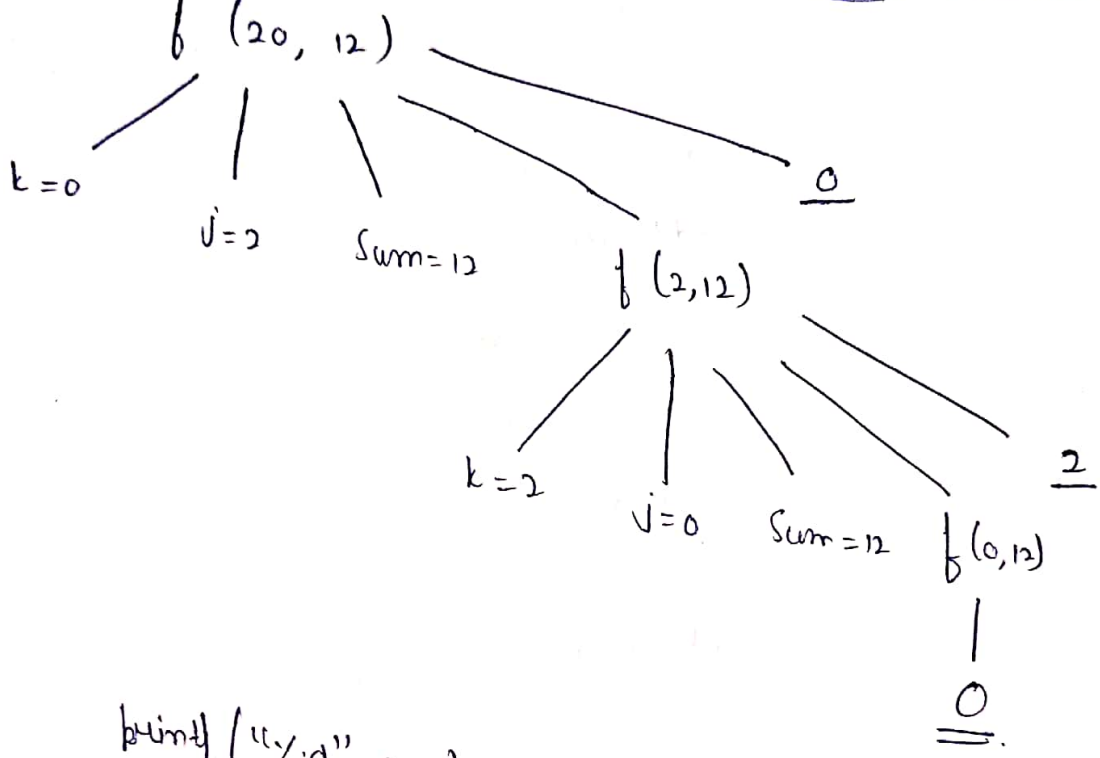
```
void foo (int n, int sum)
{
    int k=0, j=0;
    if (n==0)
        return 0;
    k = n%10;
    j = n/10;
    sum = sum+k;
    foo (j, sum);
    printf ("%d", k)
}
int main ()
{
    int a = 2048, sum = 0
    foo (a, sum);
    printf ("%d \n", sum);
}
```

what does the above program print.

A-(2) :-







printf ("%d", sum); → 0

↓

bcz this sum is sum from main

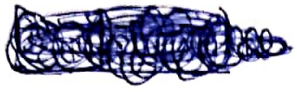
So, the ans. is 20480

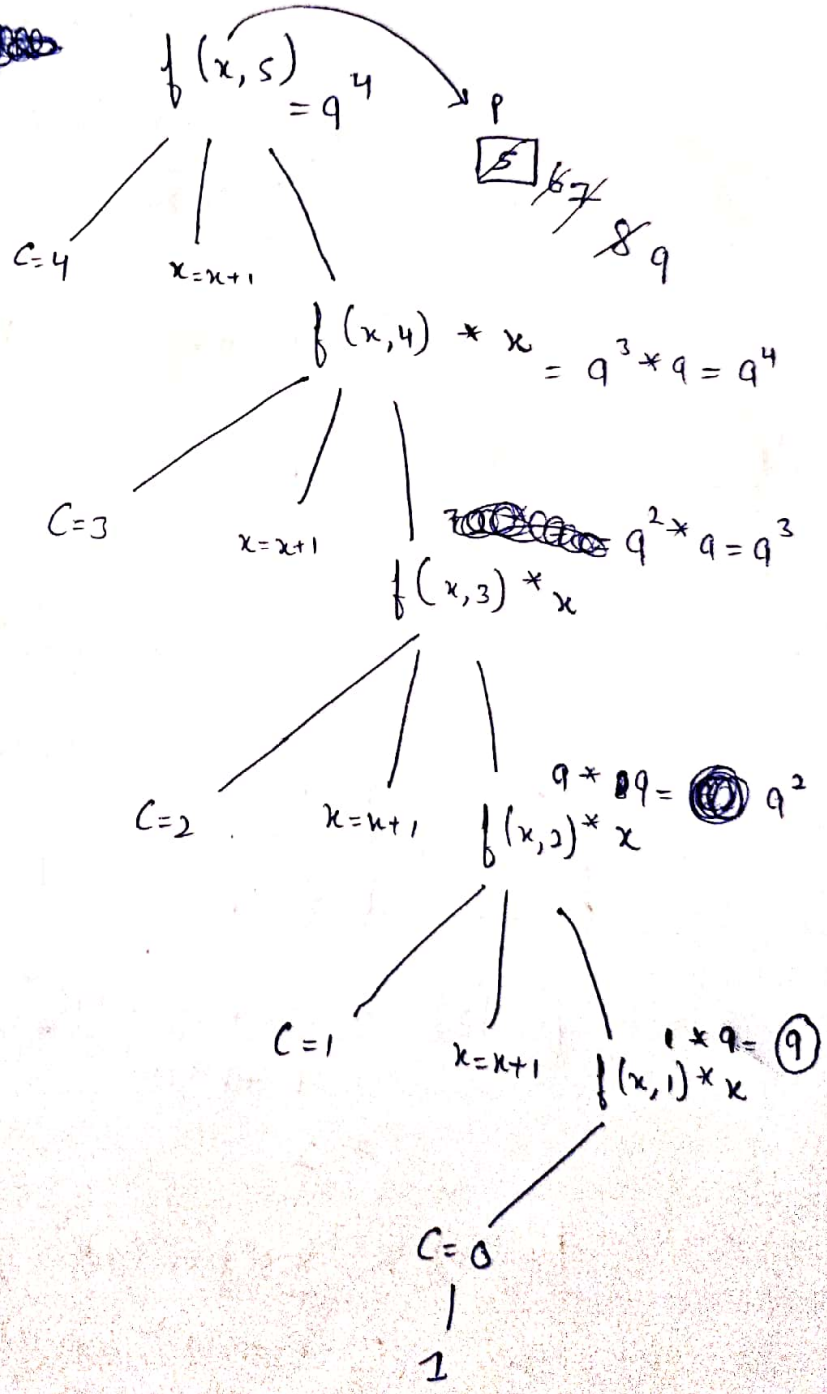
Q-③ :- what is the return value of f(P,P), if the value of P is initialised to 5 before the call? Note that the first Parameter is passed by reference, whereas the second parameter is passed by value:

```
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));
}
```

Now,





So, the answer is

$q^4 = 6561$

Q-4 :- Consider the following function

```
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;
}
```

The return value of fun(5) is

A-4 :-

fun(5)

$x = x + fun(1) * fun(4) + fun(2) * fun(3) + fun(3) * fun(2) + fun(4) * fun(1)$

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

$fun(2) = x + fun(1) * fun(2-1)$ → from Loop ; (n=2; k=1)

$= 1 + (1 * 1)$

$= 2$

$fun(3) = x + fun(1) * fun(3-1) + fun(2) * fun(3-2)$

$= 1 + 1 * fun(2) + fun(2) * fun(1)$

$= 1 + (1 * 2) + (2 * 1)$

$= 5$

Now, $fun(4) = x + fun(1) * fun(4-1) + fun(2) * fun(4-2) + fun(3) * fun(4-3)$

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

$= 1 + (1 * 5) + (2 * 2) + (5 * 1)$

$= 15$

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

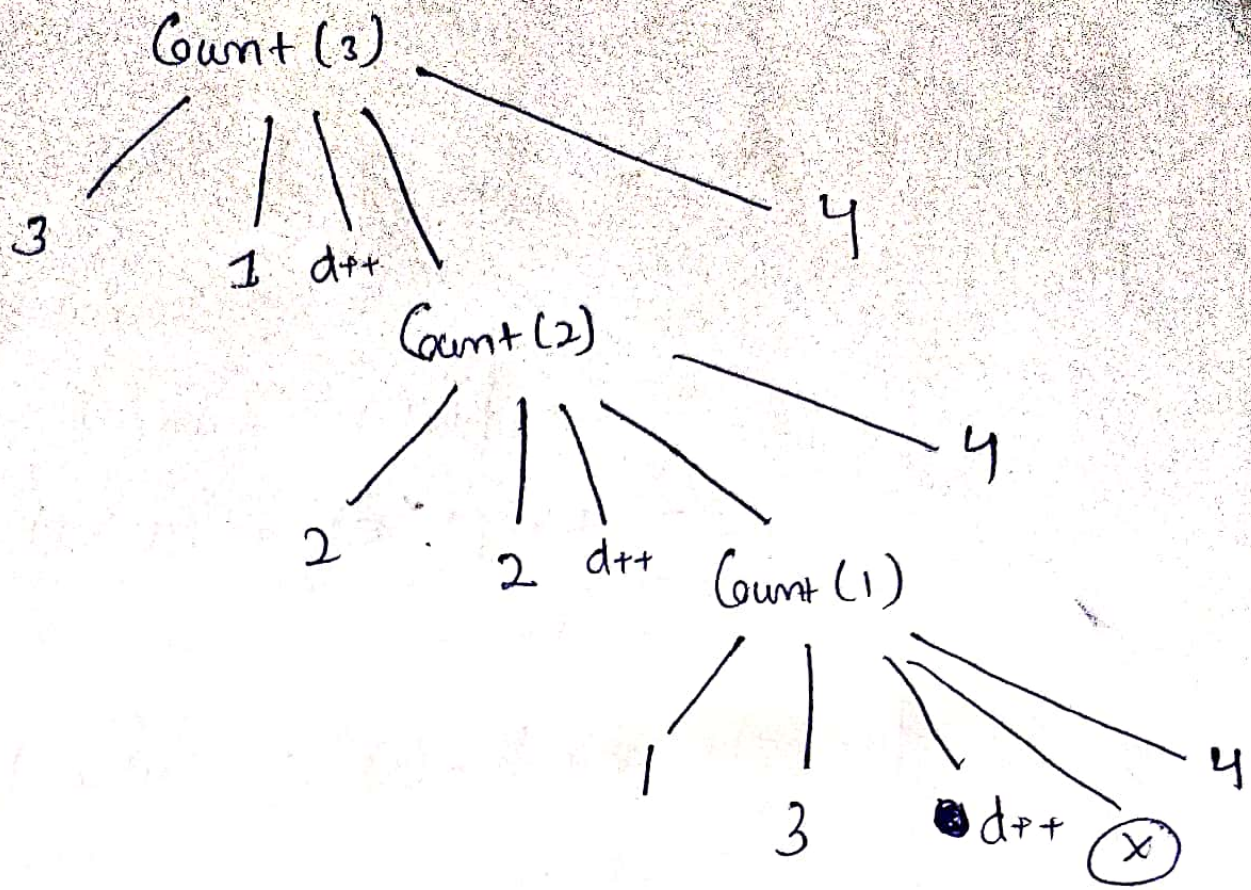
$= 1 + (1 * 15) + (2 * 5) + (5 * 2) + (15 * 1)$

$fun(5) = 51$

Q-5 :- What will be the output of a C Program ?

```
void Count (int n)
 {
   Static int d=1;
   printf ("%d", n);
   printf ("%d", d);
   d++;
   if (n>1)
      Count (n-1);
   printf ("%d", d);
   }
   Void main()
    {
      Count (3);
    }
```

A-5 :-



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