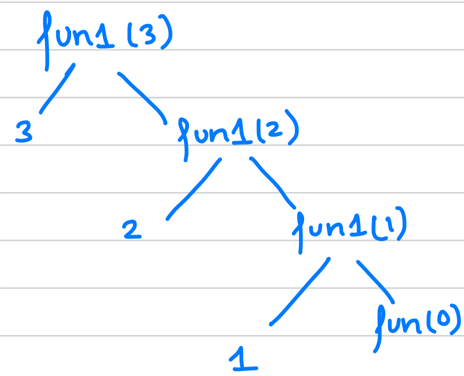
RECURSION

When a function calls itself

EXAMPLE #1

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

void main()
{
        int x = 3;
        fun1 (x);
}
```

TRACING TREE



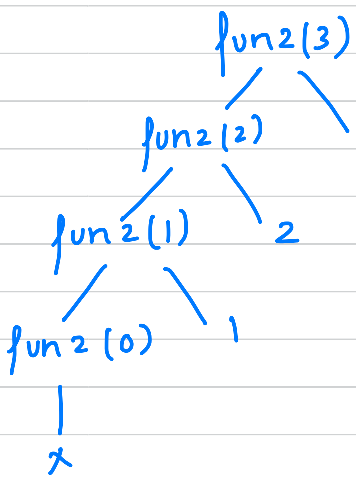
OUTPUT

3 2 1

EXAMPLE #2

```
void fun2 (int n)
{
        if (n>0)
        {
                fun2 (n-1);
                printf ("%d", n);
        }
}

void main()
{
        int x = 3;
        fun2 (x);
}
```

TRACING TREE



OUTPUT

1 2 3

```
void fun (int n)
{
        if (n>0)
        {
ASCENDING   1. calling
            2. fun(n-1)
DESCENDING  3. returning
        }
}
```

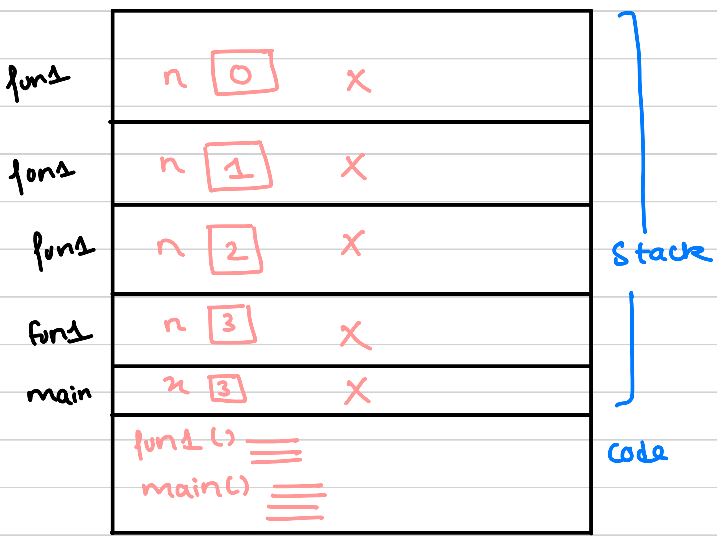
DIFFERENCE BETWEEN LOOP AND RECURSION

The main difference between loop and recursion is that recursion allows two phases i.e ascending and descending while loop allows only ascending

HOW STACK IS UTILISED IN RECURSIVE FUNCTIONS?

EXAMPLE #1

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




```
void main()
{
        int x = 3;
        fun1 (x);
}
```

Here, value of n was 3, so there were 4 calls (Tracing tree).

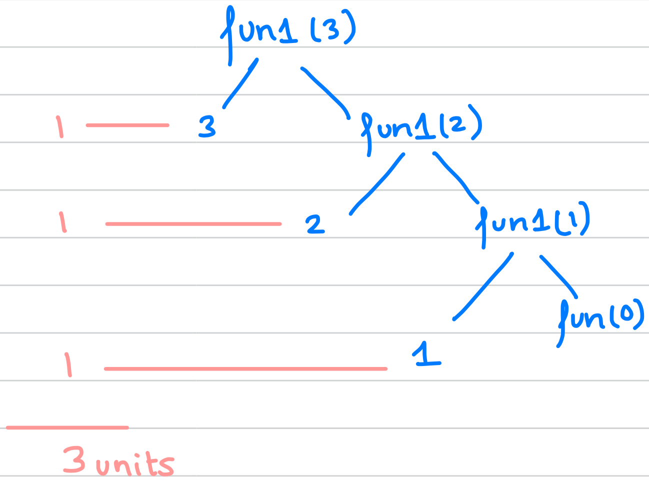
So for value of n, there will be n+1 calls

TIME COMPLEXITY (using Tree)

EXAMPLE #1

```
void fun1 (int n)
{
        if (n>0)
        {
    →         printf ("%d", n);
              fun1 (n-1);
        }
}

void main()
{
        int x = 3;
        fun1 (x);
}
```

TRACING TREE



Thus, for n calls → n units of time.

$\boxed{O(n)}$

TIME COMPLEXITY (using recurrence relation)

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

Assume it as 1 for constant

$T(n)$ — This function takes n time.

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

As it is similar to $T(n)$

$\underline{T(n) = T(n-1) + 2}$

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

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

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

$T(n) = \underbrace{T(n-2)}_{T(n-3)+1} + 1 + 1$

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

$\vdots$

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

Assume $n - k = 0$ $\therefore n = k$

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

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

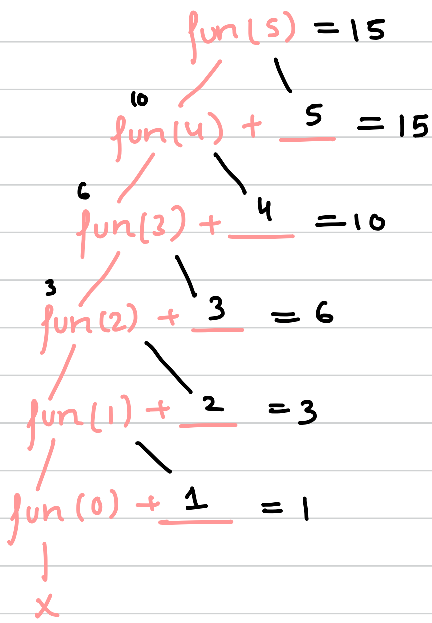
$T(n) = 1 + n$

$\boxed{O(n)}$

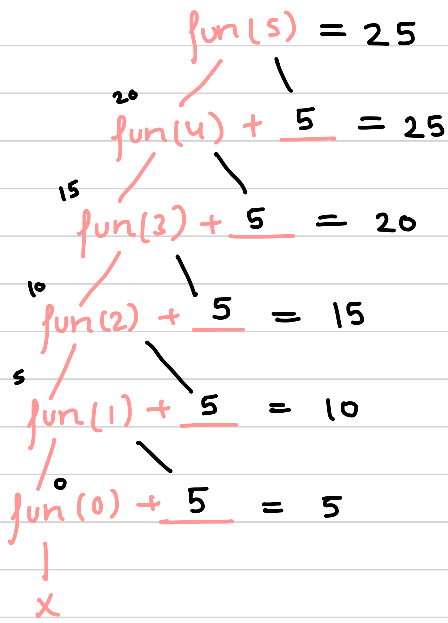
STATIC VARIABLES IN RECURSION

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

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

TRACING TREE




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

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



TYPES OF RECURSION

1. Tail Recursion
2. Head Recursion
3. Tree Recursion
4. Indirect Recursion
5. Nested Recursion

1. TAIL RECURSION

When the function is calling itself and that call is the last call in the function.

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

1. Every operation is performed at calling time.
2. Tail recursions can easily converted into loops.

TAIL RECURSION AND LOOPS

Some compilers convert your program to loop if you have used tail recursion as they are more effecient

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

fun(3);
```

space O(1)

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

fun(3);
```

O(n)

2. HEAD RECURSION

It means that the the function does not need to perform any operation at the time of calling. It has to do all operations at returning time.

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

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

Head recursions cannot be so easily converted into loops.

3. TREE RECURSION

LINEAR RECURSION

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

When the function calls itself only once.

TREE RECURSION

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

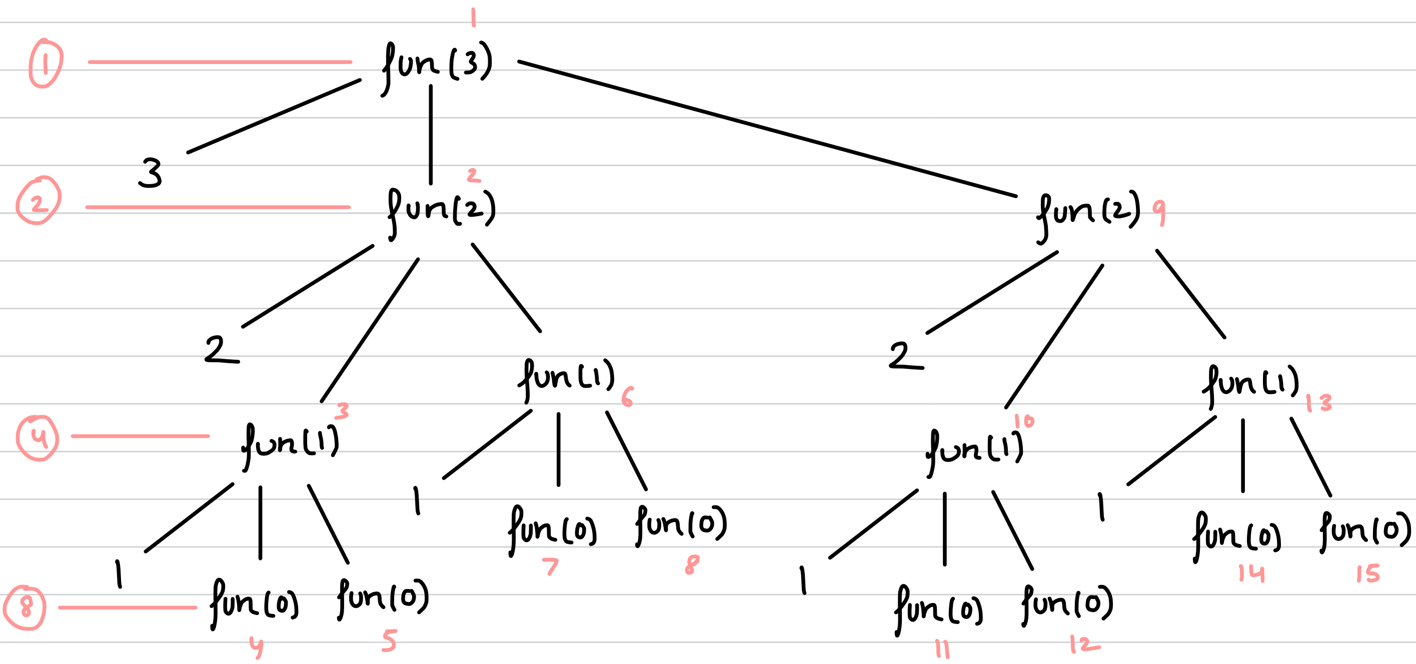
When the function calls itself more than one time.

EXAMPLE OF TREE RECURSION

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

fun(3);

• ACTIVATION RECORDS



$1 + 2 + 4 + 8 = 15$

$2^0 + 2^1 + 2^2 + 2^3 = 2^{3+1} - 1$ $\quad 2^{n+1} - 1$ $\quad$ GP Series

Time $= O(2^n)$

Space $= O(n)$ $\quad$ [No of levels of activation records $n+1$]

3 – parameter

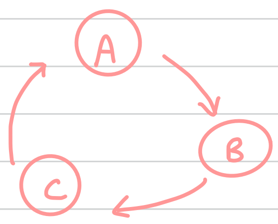
4 – levels

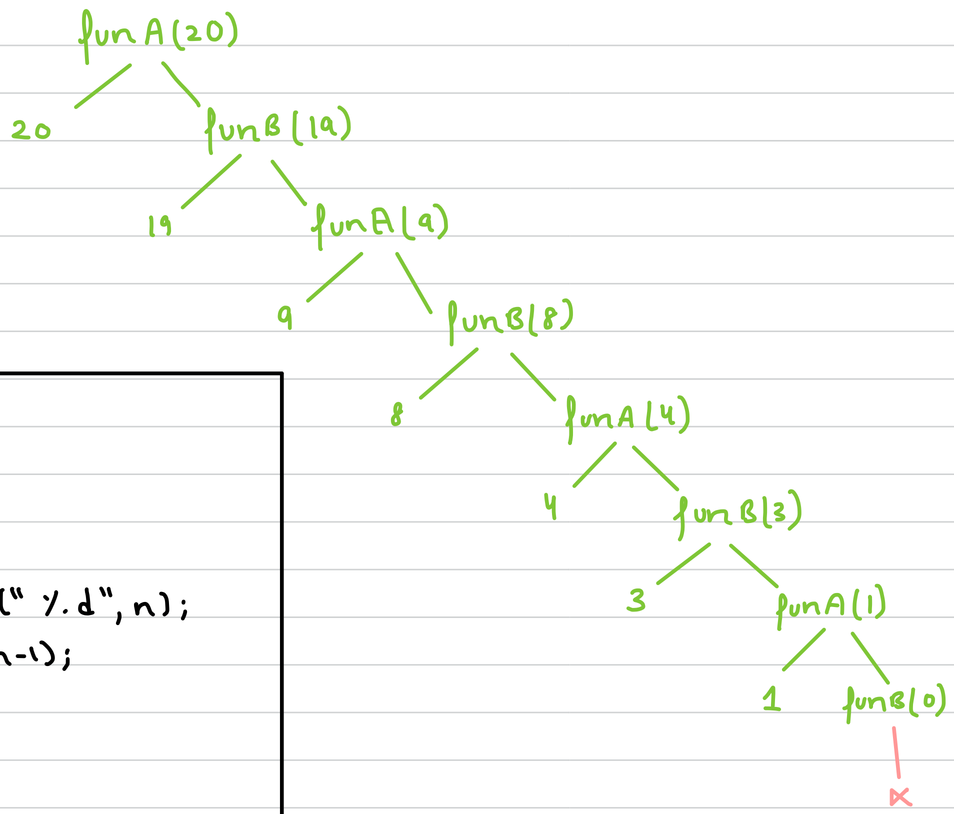
OUTPUT

3 2 1 1 2 1 1

4. INDIRECT RECURSION

In indirect recursion, there are more than one function and they are calling one another in a circular manner.






```
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);
    }
}
```

5. NESTED RECURSION

In a nested recursion, the recursive function will pass parameter as a recursive call.

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

        return fun(fun(n+11));
}

fun(95);
```

fun(95) = 91

fun(fun(95+11)) 106 — 96 = fun(106)

fun(96)

fun(fun(96+11)) 107 — 97 = fun(107)

fun(97)

fun(fun(97+11)) 108 — 98 = fun(108)

fun(98)

fun(fun(98+11)) 109 — 99 = fun(109)

fun(99)

fun(fun(99+11)) 110 — 100 = fun(110)

fun(100)

fun(fun(100+11)) 111 — 101 = fun(111)

fun(101)

91

SUM OF FIRST N NATURAL NUMBERS

$1+2+3+4 \ldots\ldots + n$

$sum(n) = 1+2+3+4+\ldots\ldots+(n-1)+n$

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

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

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

Time $= O(n)$
Space $= O(n)$

sum(5)
sum(4) + 5 = 15
sum(3) + 4 = 10
sum(2) + 3 = 6
sum(1) + 2 = 3
sum(0) + 1 = 1

```
int sum (int n)
{
    return n*(n+1)/2;
}
```

$O(1)$

```
Int sum (int n)
{
    int i, s=0;            ——— 1
    for (i=1; i<=n; i++)   ——— n+1
        s=s+i;             ——— n
    return s;              ——— 1
}
```

$2n+3$

$O(n)$

FACTORIAL USING RECURSION

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

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

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

POWER USING RECURSION

$m^n = m * m * m * m \ldots\ldots$ for n times
$pow(m,n) = m * m * m \ldots\ldots * (n-1)$ times $* m$
$pow(m,n) = pow(m, n-1) * m$

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

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

Here 10 calls are being performed

But this way is taking longer

$2^8 = (2^2)^4$
$= (2*2)^4$

$2^9 = 2 * (2^2)^4$

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 = 6
|
pow (2,1)*2 = 4
|
pow (2,0)*2 = 2
|
1

REWRITING POWER FUNCTION

FASTER METHOD

```
int pow(m,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);
}
```

pow(2, 9) $= 2^9.$

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

pow(2^4, 2)

pow(2^8, 1)

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

1

TAYLOR'S SERIES

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

$$e(x,4) = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!}$$

$$e(x,3) = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!}$$

$$e(x,2) = 1 + \frac{x}{1} + \frac{x^2}{2}$$

$$e(x,1) = 1 + \frac{x}{1}$$

$p = p * x$ $\quad f = f * 1$ $\quad 1 + p/f$

$$e(x,0) = 1$$

```
int  e(int x, int n)
{
     static int p=1, f=1;
     int r;

     if (n==0)
          return 1;

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

p	f
$\boxed{1}$	$\boxed{1}$
x*x*x*x	1×2×3×4

TAYLOR'S SERIES USING HORNER'S RULE

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

$$\frac{x \times x}{2 \times 1} \qquad \frac{x \times x \times x}{3 \times 2 \times 1}$$

No of multiplications

0 0 1+1 2+2

2 4 6 8 10

$2(1+2+3+\ldots n)$

$2\frac{(n)(n+1)}{2}$

$O(n^2)$ Quadratic

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

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

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

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

$O(n)$ Linear

```
int e(int x, int n)
{
    int s = 1;

    for( ;n>0; n--)
        s = 1+ x/n*s;

    return s;
}
```

USING FOR LOOP

```
int e(int x, int n)
{
    static int s = 1;

    if(n==0)
        return s;
    else
        s = 1+ x/n*s;

    return e(x, n-1);
}
```

USING RECURSION

FIBONACCI SERIES

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

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

PROGRAM USING ITERATION

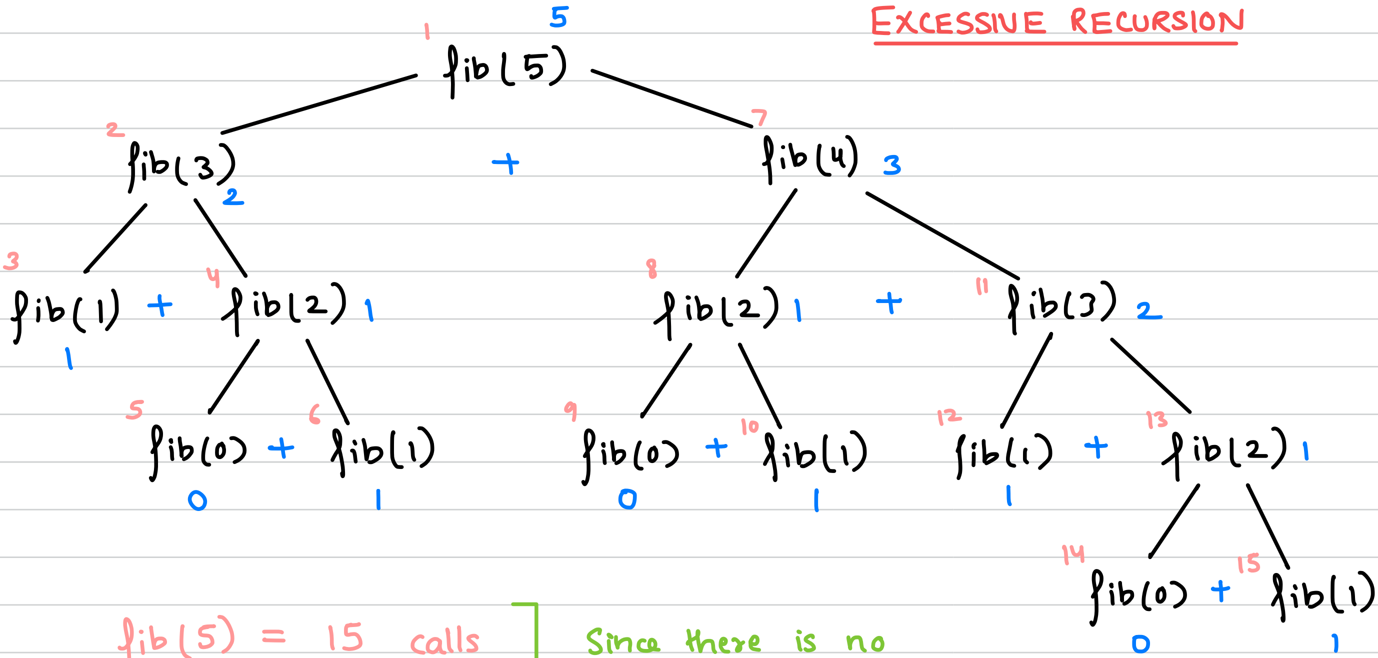
```
int fib (int n)
{
    int t0=0, t1=1, s, i;  ——  1
    if (n<=1)
        return n;  ——————  1
    for (i=2; i<=n; i++)  ——  n
    {
        s = t0 + t1;  ——  n-1
        t0 = t1;  ——————  n-1
        t1 = s;  ——————  n-1
    }

    return s;  ———————————  1
}                          ----
                            4n
```

$O(n)$

PROGRAM USING RECURSION

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

EXCESSIVE RECURSION



fib (5) = 15 calls
fib (4) = 9 calls
fib (3) = 5 calls

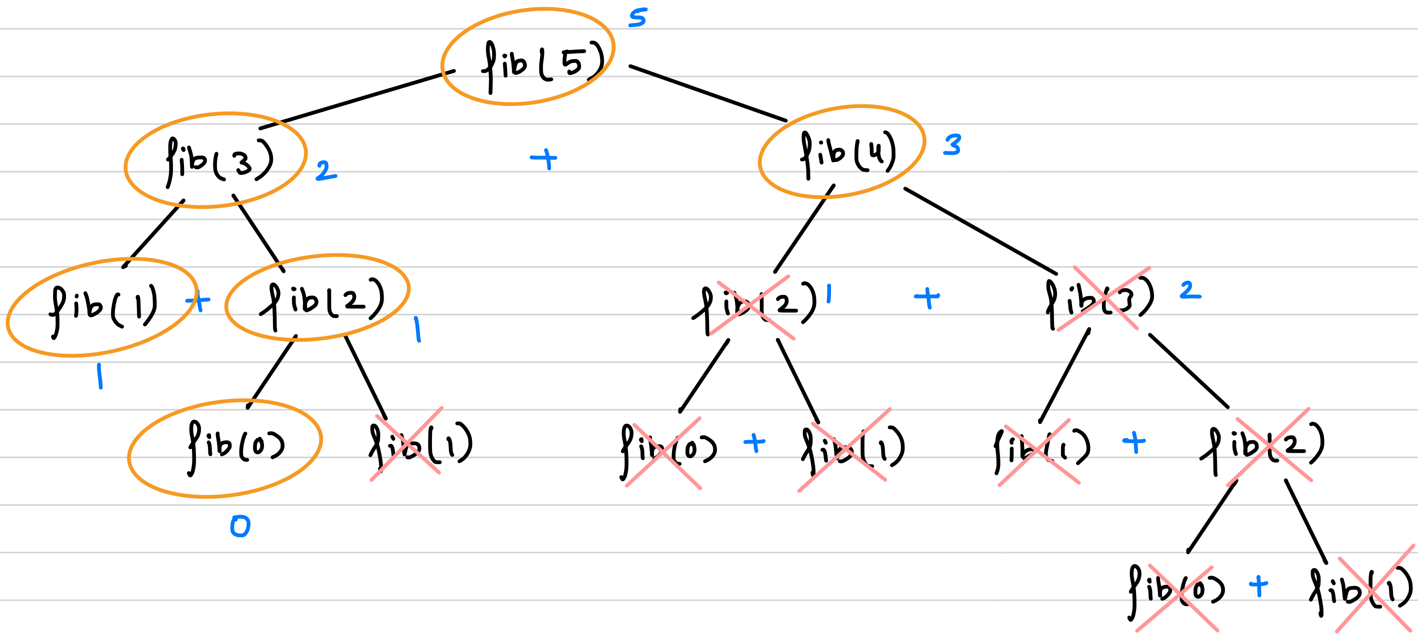
Since there is no definite pattern, then we have to assume that the function fib(n-2) + fib(n-1); is calling itself 2 fib(n-1)

order of (2^n)

This tree lies in the category of excessive recursion because it calls itself multiple times for the same parameters.

To reduce the order of this function, we will write another program using static array and initialize all its values with -1.

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



So, for 5 as n, 6 calls are made
∴ for n, n+1 calls are made

O(n)

This approach of storing result in an array is called MEMOIZATION.

↳ Storing the result of function calls, so they can be utilized again for avoiding excessive calls

```
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];
    }
}
```

COMBINATION FORMULA

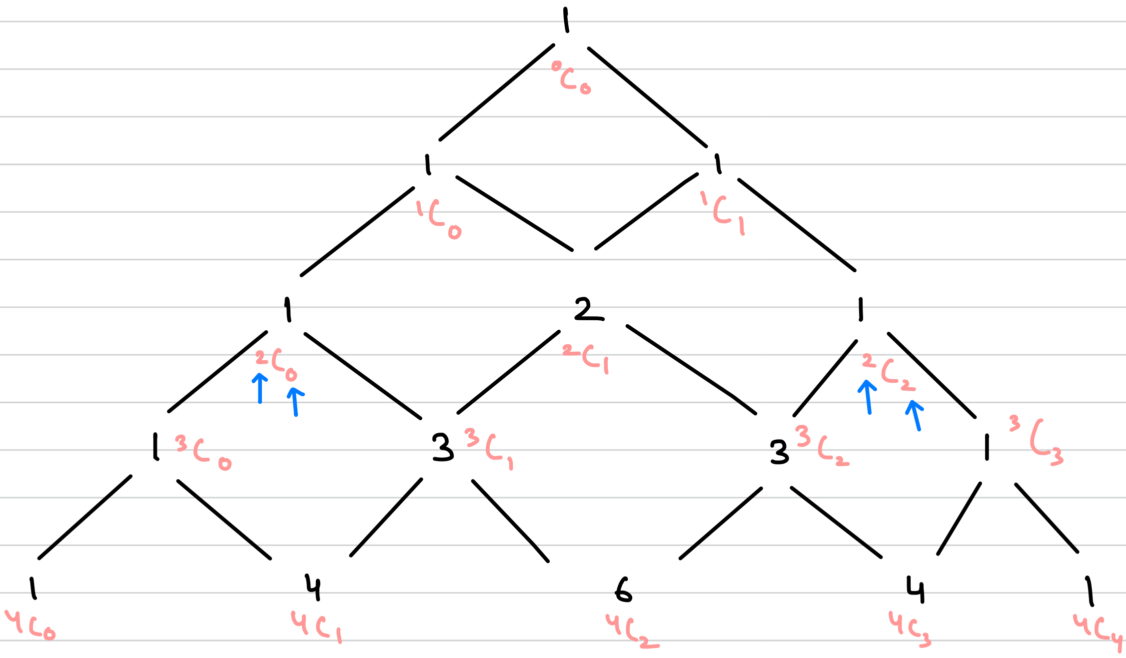
$$^nC_r = \frac{n!}{(n-r)!\,r!}$$

```
int C (int n, int r)
{
    int t1, t2, t3;
    t1 = fact (n);          ——— n
    t2 = fact (r);          ——— n
    t3 = fact (n-r);        ——— n

    return t1 /(t2 * t3);   ——— 1
}
```

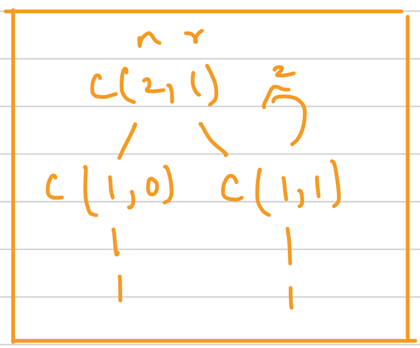
$3n$

$O(n)$

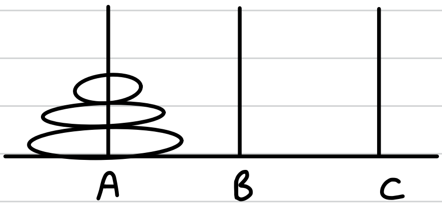
PASCAL'S TRIANGLE




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

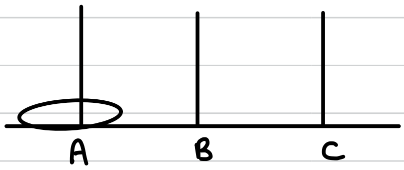


TOWER OF HANOI

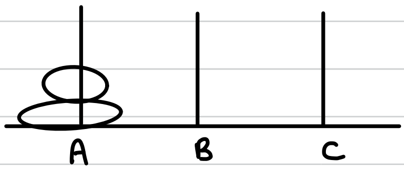


1. Move one disk at a time.
2. No bigger disk can be there above a smaller one.

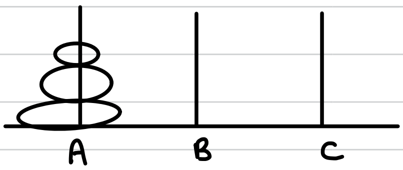
TOH (1, A, B, C)
Move disk A to C using B.



TOH (2, A, B, C)
1. TOH (1, A, C, B)
2. Move disk A to C using B.
3. TOH (1, B, A, C)



TOH (3, A, B, C)
1. TOH (2, A, C, B)
2. Move disk from A to C using B
3. TOH (2, B, A, C).



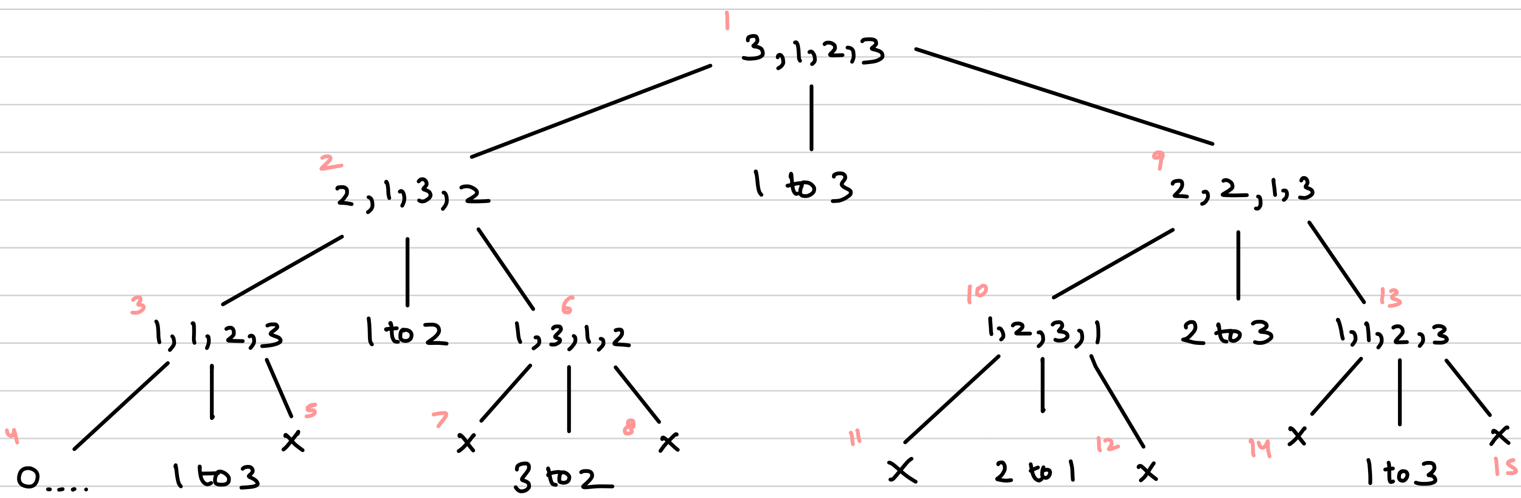
FOR n number of disk.

TOH ($\overset{n}{\cancel{3}}$, A, B, C)
1. TOH ($\overset{n-1}{\cancel{2}}$, A, C, B)
2. Move disk from A to C using B
3. TOH ($\overset{n-1}{\cancel{2}}$, B, A, C).

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

$A \rightarrow 1$
$B \rightarrow 2$
$C \rightarrow 3$

TOH (3,1,2,3)



OUTPUT

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

	Calls		
$n = 3$	15	$1+2+2^2+2^3$	$= 2^4-1$
$n = 2$	7	$1+2+2^2$	$= 2^3-1$

$\boxed{2^{n+1}-1}$

$O(2^n)$