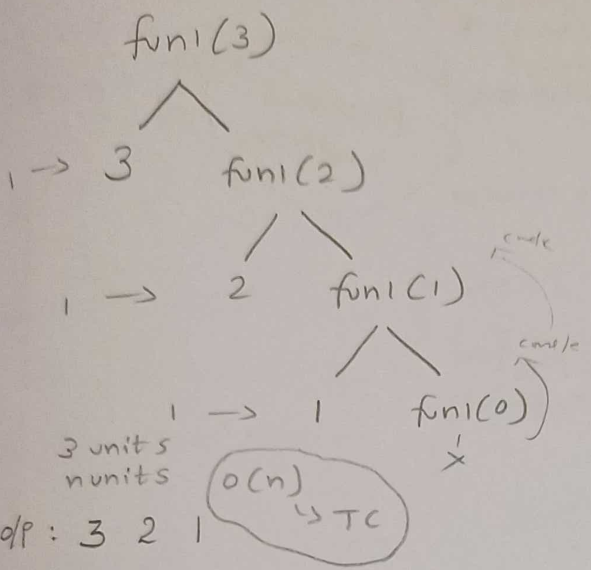
Recursion

-> function calling itself.

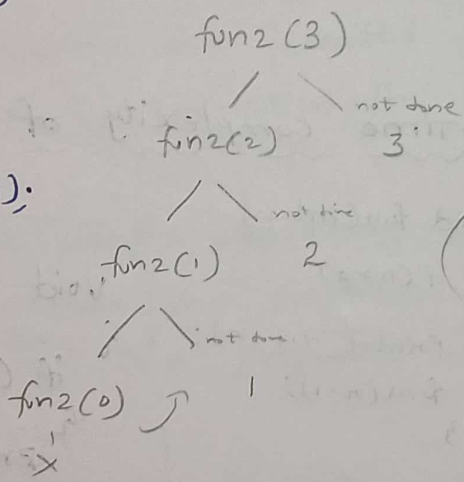
-> There must be some base condition That will Terminate recursion.

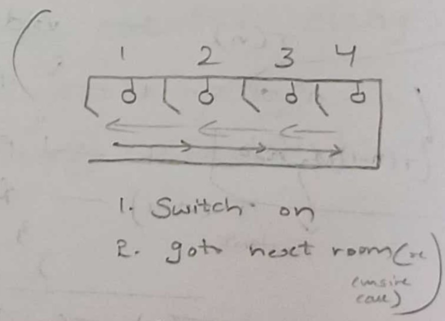



```
void fun1(int n)
{
   if(n>0)
   {
      printf("%d", n);
      fun1(n-1);
   }
}
void main()
{
   int x=3;
   fun1(x);
}
```

o/p : 3 2 1

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



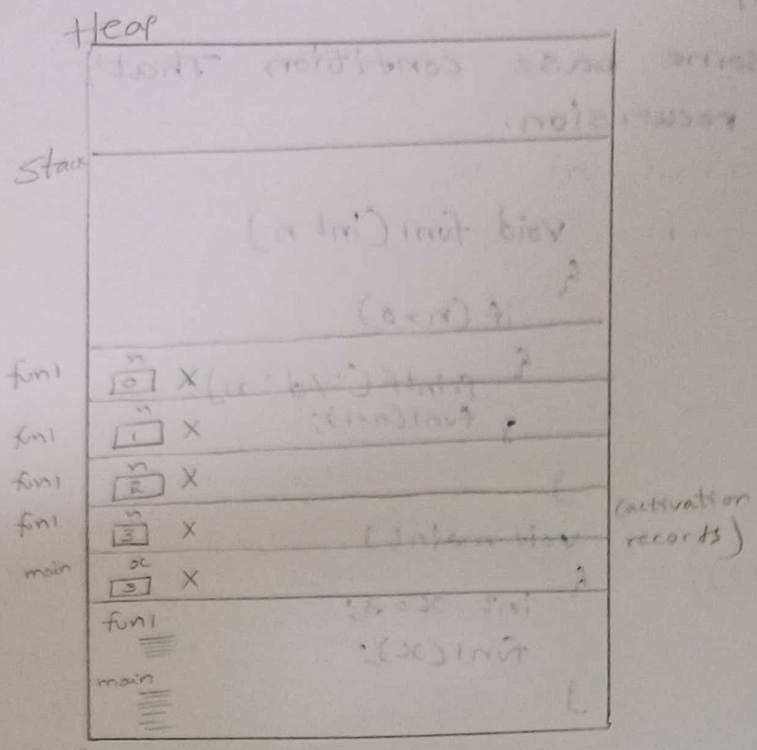


o/p : 1 2 3

-> recursion having two phases : calling & returning phases.

```
Void fun(int n)
{
   if(n>0)
   {
     1. Calling (Ascending)
     2. fun(n-1)*2
     3. Returning (Descending)
   }
}
```

How Recursion uses Stack



Previous function

$\boxed{\text{calls} \quad O(n)}$ → space

$\underbrace{\text{calls}}_{n+1}$

Recurrence Relation - Time complexity of Recursion.

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

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

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

constant

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

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

fun1(3)
1 — 3 fun1(2)
1 — 2 fun1(1)
1 — 1 fun1(0)
X

3 units

n units o/p 3, 2, 1

$O(n)$

$T(n) = T(n-1) + 1$ — 1

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

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

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

$T(n) = T(n-2) + 2$ — 2

$T(n-3) + 1$

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

$T(n) = T(n-3) + 3$ — 3

⋮

$T(n) = T(n-k) + k$ — 4

till '0'

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

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

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

$T(n) = 1 + n$ — $O(n)$

Program code ✓

debugging ✓

Static variables in Recursion → & global

fun(5) = 15

fun(4) + 5 → 15

fun(3) + 4 → 10

fun(2) + 3 → 6

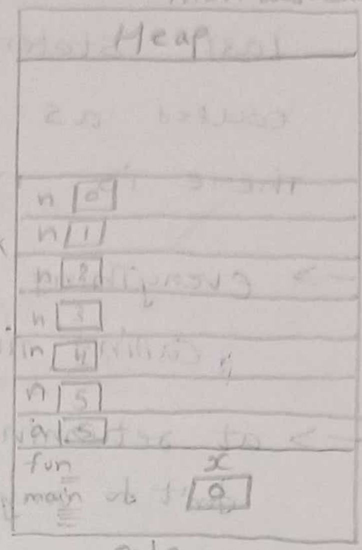
fun(1) + 2 → 3

fun(0) + 1 → 1

0

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

if remove This & writing it globally Then also same



code section

-> static variables are created in codesection (There will be a subsection for global & static variables).

-> will these static variables be created every time whenever the function called -> no (at loading time of The program only created.)

x

~~0~~ 1 2 3 4 5

fun(5) = 25

fun(4) + 5 → 25

fun(3) + 5 → 20

fun(2) + 5 → 15

fun(1) + 5 → 10

fun(0) + 5 → 5

0

Program code ✓

debugging ✓

Tail Recursion

-> if a Recursive function is calling itself and That Recursive call, i.e. That call is The last statement in function; Then it is called as Tail Recursion, after That call There is nothing (not performing anyThing).

-> everyThing is done at calling Time only.

-> at returning Time don't do anyThing.

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

TC
$O(n)$
SC
$O(n+1)$
$O(n)$

Comparison Tail Recursion with loops.

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

TC $O(n)$ SC $O(1)$ (constant Time).

(Tail Recursion can be easily converted into loops)

Head Recursion

-> no statements before Recursive call
-> everyThing is done at Returning Time only.
-> at calling Time don't do anyThing.

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

Comparison Head Recursion with loops.

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

X

(So we cannot easily convert)

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

Tree Recursion

```
fun(n)
{ if(n>0)
  {
     =
①->  fun(n-1);
     =
②->  fun(n-1);
     =
  }
}
```

if a f^n calling more Than '1' Time

Linear Recursion

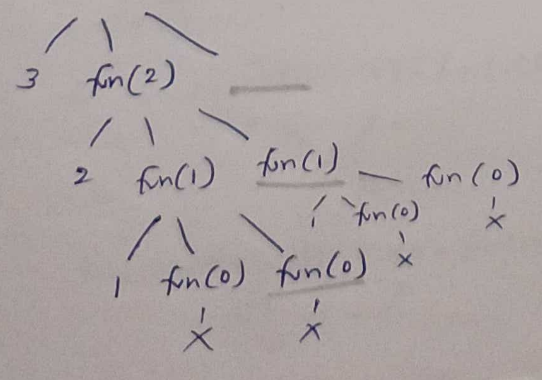
```
fun(n)
{
   if(n>0)
   {
      ≡
①->   fun(n-1);
      ≡
   }
}
```

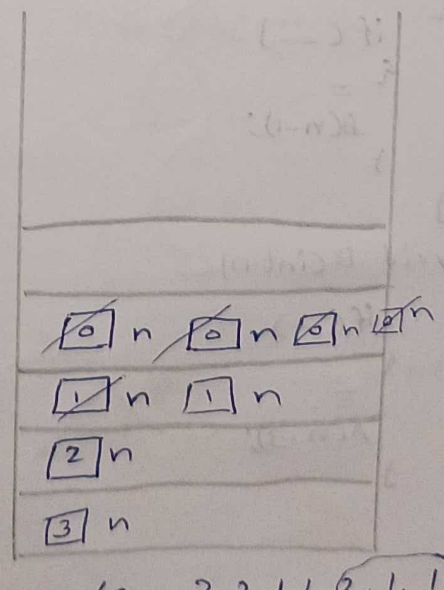
if it is calling it self '1' Time Then linear Recursion.

ex:

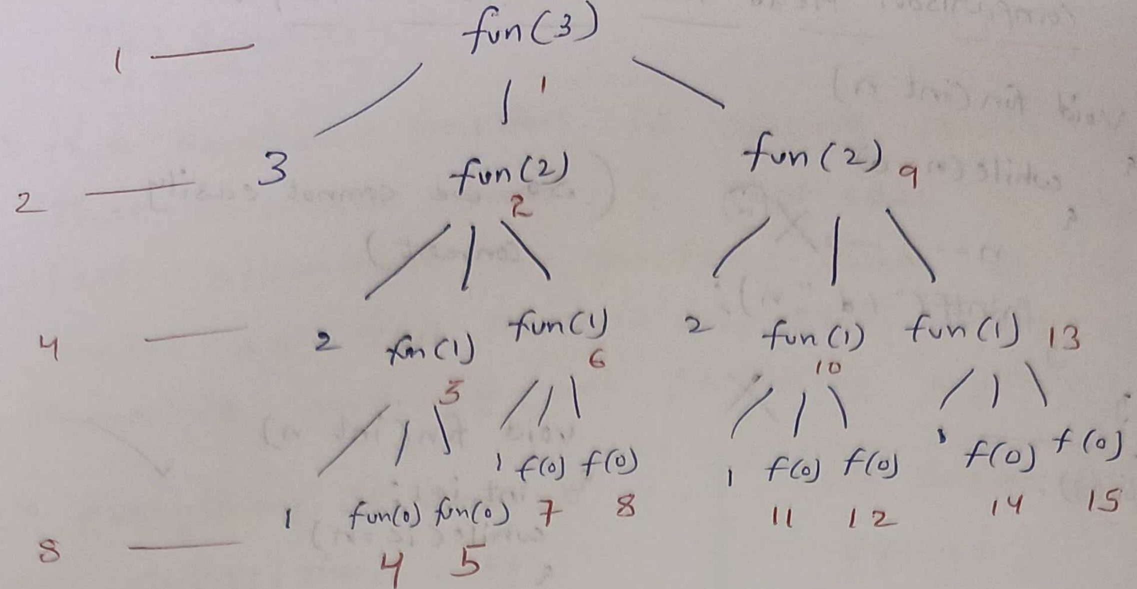
```
void fun(int n)
{ if(n>0)
  { printf("%d ",n);
    fun(n-1);
    fun(n-1);
  }
}
fun(3);
```

Tracing





o/p 3, 2, 1, 1, 2, 1, 1



→ Sum = 15

$2^0 + 2^1 + 2^2 + 2^3 = 15,$ → $= 2^{3+1} - 1$

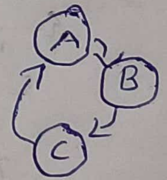
$\boxed{\text{TC } O(2^n) \text{ calls}}$

SC not $O(2^n)$
its less only,
bcz same place
deleting & creating

Code for Tree

indirect Recursion (iR)

in (iR) There may be more Than one function & They are calling one another in a circular fashion, so That if The first f^n calls second one. & Second call Third & The Third one again call back first function. Then it becomes a cycle, so it becomes a recursion.




```
void A(int n)
{
    if(__)
    {
        =
        B(n-1);
    }
}
void B(int n)
    if(-)
    {
        =
        A(n-3);
    }
}
```

funA(20)

20 — funB(19)

19 — funA(9)

9 — funB(8)

8 — funA(4)

4 — funB(3)

3 — funA(1)

1 — funB(0)

X

```
void funA(int n)
{ if(n>0)
  {
    printf(n);
    funB(n-1);
  }
}
void funB(int n)
{ if(n>1)
  {
    printf(n);
    funA(n/2);
  }
}
```

Program code for IR ✓

Nested Recursion (NR)

In NR a Recursive f^n will pass parameter as a Recursive call.

```
void fun(int n)
{ if(—)
  {
    =
    fun(fun(n-1));
  }
}
```

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

fun(95);

fun(95)

fun(fun(95+11)) — $\underline{106}$ — 96 = fun(106)

fun(96)

fun(fun(96+11)) — $\underline{107}$ — 97 = fun(107)

fun(97)

fun(fun(97+11)) — $\underline{108}$ — 98 = fun(108)

fun(98)

fun(fun(98+11)) — $\underline{109}$ — 99 = fun(109)

fun(99)

fun(fun(99+11)) — $\underline{110}$

fun(100)

fun(fun(100+11)) — $\underline{111}$

fun(101)

~~fun~~ 91

Program code for NR ✓

Sum of first n Natural no's

Sum(n) = 1+2+3... +(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;
}
```

O(n) TC
O(n) SC

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

TC = O(1)
SC

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

O(n) TC
O(1) SC

Program code ✓

Factorial ✓
code ✓

Power using Reccursion.

Exponent $(m)^n$

$2^5 = \underbrace{2*2*2*2}_{4\ times}*2$

$m^n = m*m*m*m*m.$ for n times.

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

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

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

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

$2^8 = (2^2)^4 = (2\times2)^4$

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

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

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

  if(n%2==0)
    return pow(m*m, n/2);

  else
    return m*pow(m*m, (n-1)/2);

}
```

code ✓

pow(2, 9) 2^9

↓

$2 * pow(2^2, \frac{(9-1)}{2})$ 2^9

|

$pow(2^2 * 2^2, 2)$ 2^8

|

$pow(2^4 * 2^4, 1)$ 2^8

|

$2^4 * 2^4 * pow(2^{16}, 0);$

↓

1

Taylor series using Recursion

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

	p	f	
x^0	1	1	0!
	x	1	
	↳ x*x	↳ 1*2	
	↳ x*x*x	↳ 1*2*3	
	↳ x*x*x*x	↳ 1*2*3*4	

e(x, 4)

|

e(x, 3)

|

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

|

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

|

e(x, 0) = 1 $1 + \frac{x}{1}$

↓

1

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

/

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

/

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

$f = f * 2$ $1 + \frac{x}{1} + \frac{p}{f}$

$p = p * x$

/

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

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

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;
  }
}
```

code ✓

Taylor series using Horner's Rule

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

$$0 \quad 0 \quad \frac{x*x}{1*2} \quad \frac{x*x*x}{1*2*3} \quad \frac{3}{3}$$

$$0 \quad 0 \quad 2+4+6+8+10$$

$$2(1+2+3+4\dots)$$

$$\cancel{2}\left(\frac{n(n+1)}{\cancel{2}}\right)$$

$n(n+1)$ (no. of multiplic'ns required).

TC $O(n^2)$

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

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

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

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

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

$O(n)$ multiplic'ns

if we consider

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

e(x, 4)

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

using loop.

changed code

```
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);
}
```

recursion.

Fibonacci Series

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

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

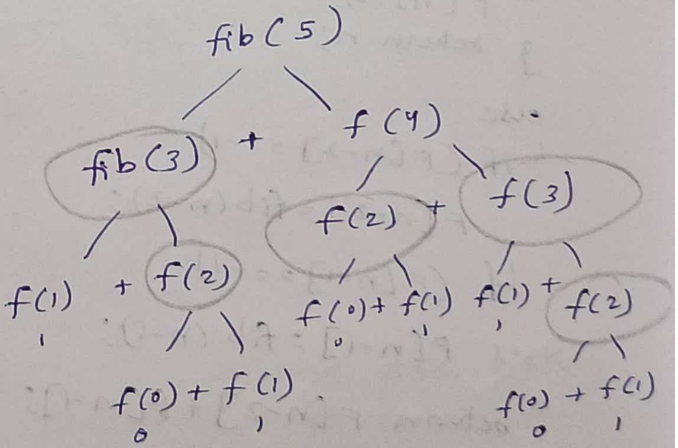
Recursive

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

iterative

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

o(n)



calls = 15 for n=5

calls = 9 for n=4

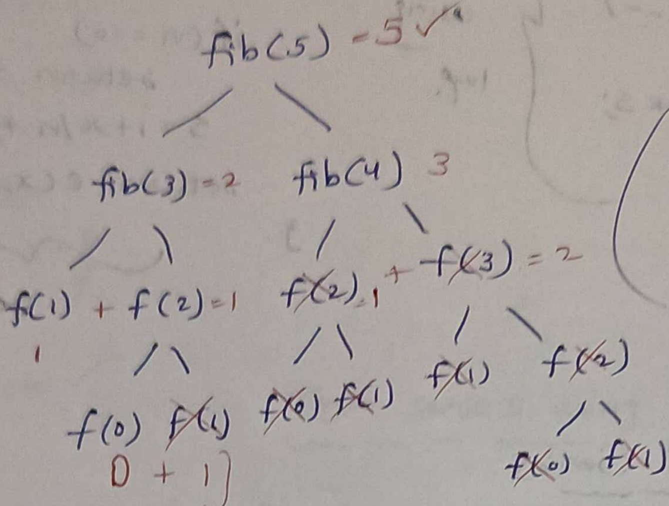
calls = 5 for n=3

approx o(2^n)

assume fib(n-2) as fib(n-1) Then 2fib(n-1)

→ So a recursive function is calling itself multiple times for the same values. So, such a recursive f^n is called excessive recursion.

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



avoid This call we know answer.

6 calls only O(n)

→ This Process called Memoization.

```
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];
   }
```

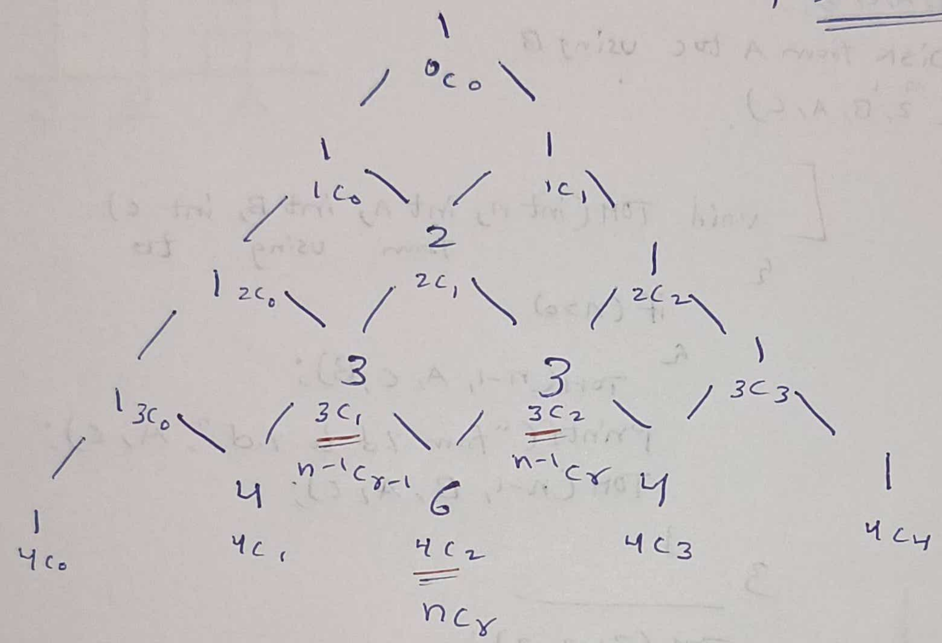
Code ✓

Combination formula

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

```
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);    3n
}                                o(n)
```

Pascal's Δ^{le}



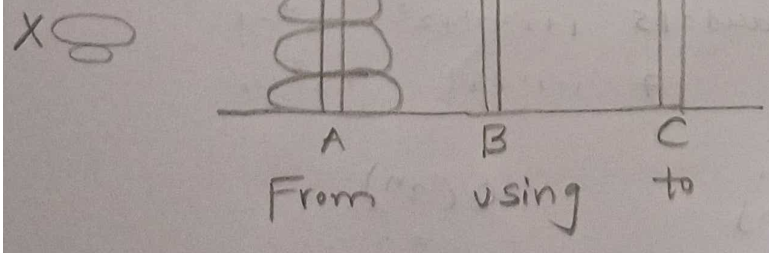

```
int c (int n, int r)
{  if (r==0 || n==r)
      return 1;
   else
      return c(n-1, r-1) + c(n-1, r);
}
```

code

Tower of Hanoi

→ move one disk at a Time

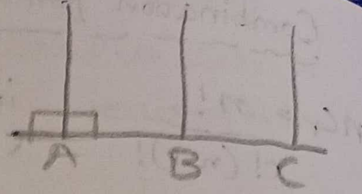
→ no, larger disk should be kept over a smaller disk.



TOH(1, A, B, C)

more Disk from A to C using B

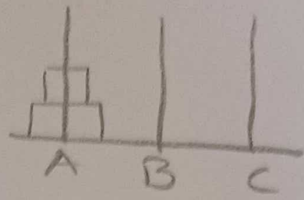
1 Disk



TOH (2, A, B, C)

1. TOH(1, A, C, B)
2. More Disk from A to C using B
3. TOH (1, B, A, C)

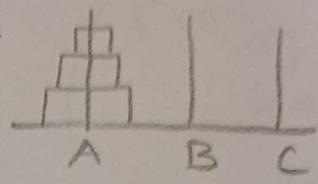
2 Disk



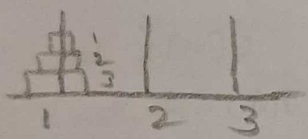
TOH($\overset{n}{3}$, A, B, C)

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

3 Disk

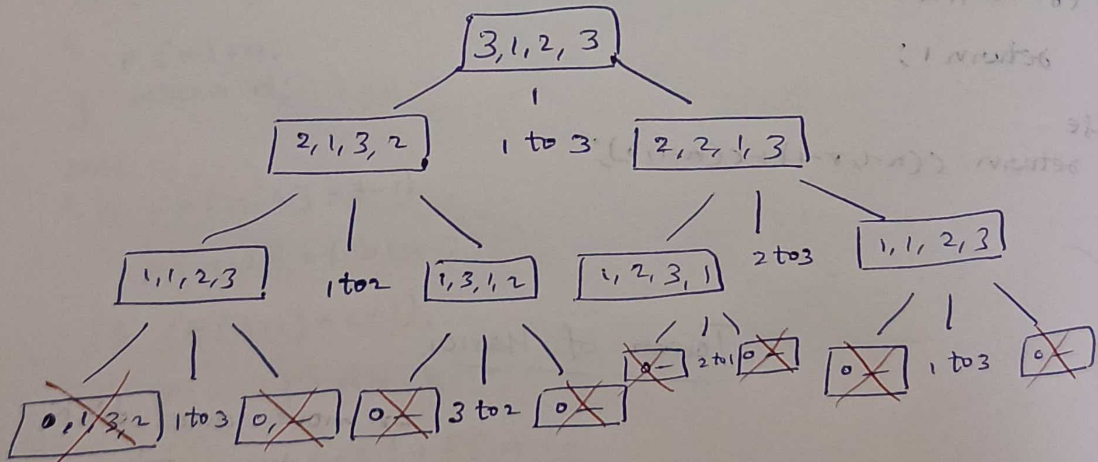



```
void TOH(int n, int A, int B, int C)
                 from     using   to
{
    if (n>0)
    {
        TOH(n-1, A, C, B);
        printf("from %d to %d", A, C);
        TOH (n-1, B, A, C);
    }
}
```



3

TOH (3,1,2,3);
n A B C



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

disks $\underline{n=3}$ $\underline{\text{calls}=15}$ $1+2^1+2^2+2^3 = 2^4-1$

$n=2$ 7 $1+2^1+2^2 = 2^3-1$

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

$\underline{O(2^n)}$

Code ✓