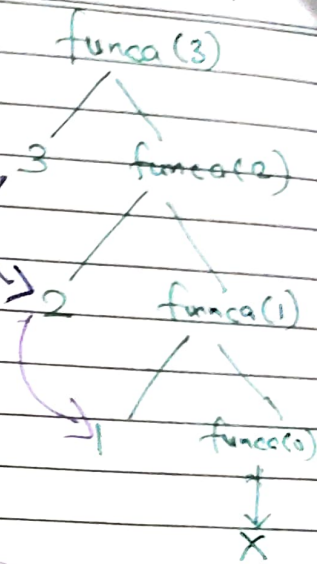
papergrid
Date: / /

Recursion

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

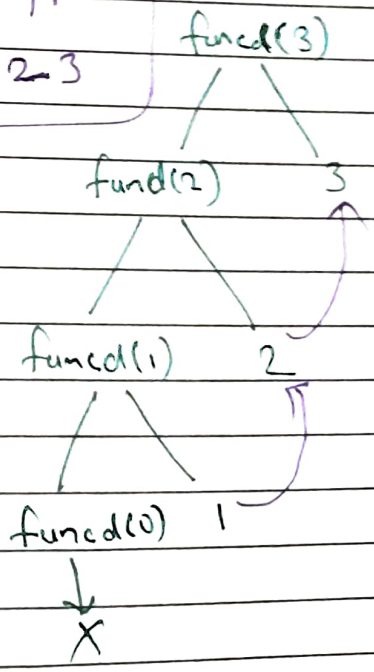
O/P
3 2 1



~~int~~

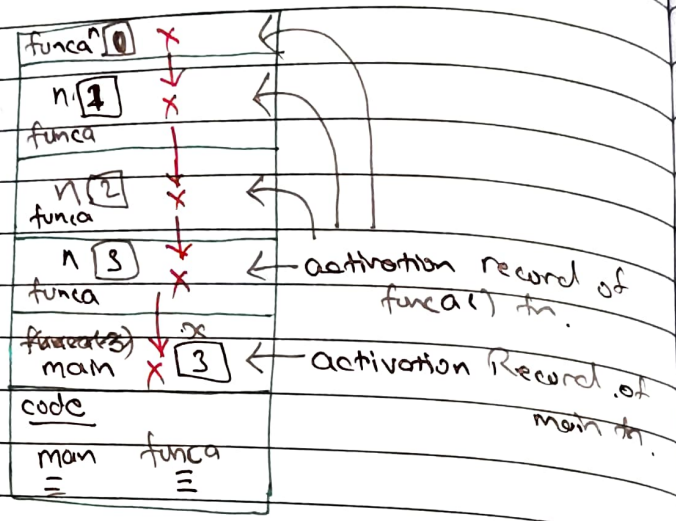
```
void funcd(int n)
{
  if(n>0)
  {funcd(n-1)
  cout << n << endl;
  }
}
```

O/P
1 2 3




```
int main()
{
  int x = 3;
  funca(3);
  funcd(3);
}
```

Recursion with Stack



funca() is called n+1 times, here n=x.

Eg

for x=3 funca() is called 4 times. (n+1)

n+1 => O(n)

memory consumed by funcac) fn is

= (n+1)(funca variable numbers & its size)

Time & Space Complexity

A	0	1	2	3	4	5	6	7	8	9

n=10 elements (but n can be 1 to ∞)

(1) use case 1, Eg adding all the elements

code

```
for (i=0; i<n ; i++)
{
    ≡ } this code will get executed n times.
}.
```

} O(n) ← Order of n or Big O

degree of 1

(2) Use case 2

Eg:- Executing/performing operations on all the element twice in the array

code

```
for (i=0; i<n ; i++)
{
    for (j=0; j<n ; j++)
    {
        ≡ } This will get executed n times when each time this for loop is triggered.
    }
}
```

O(n) = n × n = n²

n O(n²)

degree of 2

③ Use case 3

Eg:- Performing operations on all the elements with rest of the elements.

code

```
for(j=0; j<n; j++)
{
  for(j=0, j<n-1; j++)
  {
    ≡   } run(n-1) times for each for loop trigger
  }
}
```

n

$O(n) = n(n-1)$
$= n^2 - n$

$O(n^2)$

④ Use core 4

Eg:- Performing 1st element over 2 to n & Performing 2nd element over 3 to n

code

```
for(j=0; j<n; j++)
{
  for(j=0; j<n; j++)
  {
    ≡
  }
}
```


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

$= \frac{n(n-1)}{2}$

$O(n^2)$

⑤ Use case 5

Eg:- Processing the array from the middle and dividing the left by 2 and 2 again and so on.

code

```
for(j=0; j>1; j=j/2)
{
  ≡
}
```

When something is diride by x until it reache 1 then it can be ~~x~~represented by => $\log_x n$

for the above ex => $\log_2 n$

becouse we are dividing by 2 again and again

$2^3 = 8 \Rightarrow \log_2 8 = 3$

$O(\log_2 n) = O(n)$

Exact time taken for processing.

~~for~~ void.

```
int sum(int A[], int n){

  for int s, i;
    s=0;          -> 1

  for(i=0; i<n; i++)      -> n+1
  {   1   n+1   n
      s = s + A[i];       -> n
  }
  return s      -> 1
}
```

$f(n) = 1+1+n+n+1 = 2n+3$

$O(n)$


```
int Add(int n)
{
  int i,j

  for(i=0; i<n; i++)  -> n+1
  {
    for(j=0; j<n; j++)  -> n+1
    {
      c[i][j] = A[i][j] + B[i][j];   n <-> n
    }
  }
}
```

$f(n) = (n+1) + n(n+1) + n^2$

$= n+1+n^2+n+n$

$= 2n^2+2n+1$

$= O(n^2)$

Static & Global variables in Recursion.

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

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

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

* if x is static & global

x=0

fun(5) → 20 + 5 = 25 | 5 1
fun(4) → 15 + 5 = 20 | 5 2
fun(3) → 10 + 5 = 15 | 5 3
fun(2) → 5 + 5 = 10 | 5 4
fun(1) → 0 + 5 = 5 | 5 5
fun(0) → 0

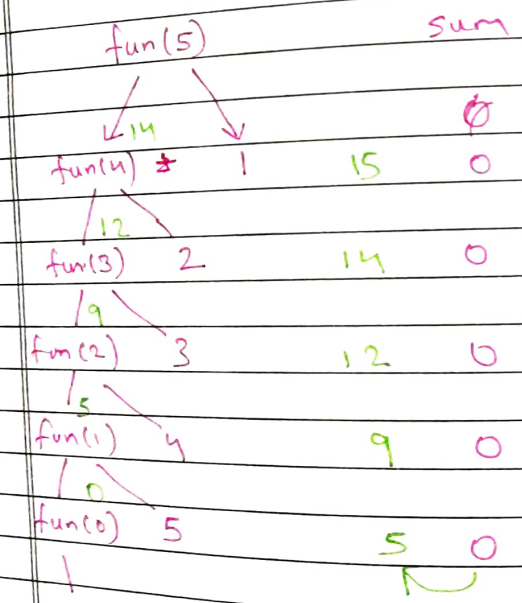
```
int fun(int n){
    static int x=0;
    if(n>0){
      return fun(n-1)+(++x);
    }
    return 0;
}
```

This can be written as:-

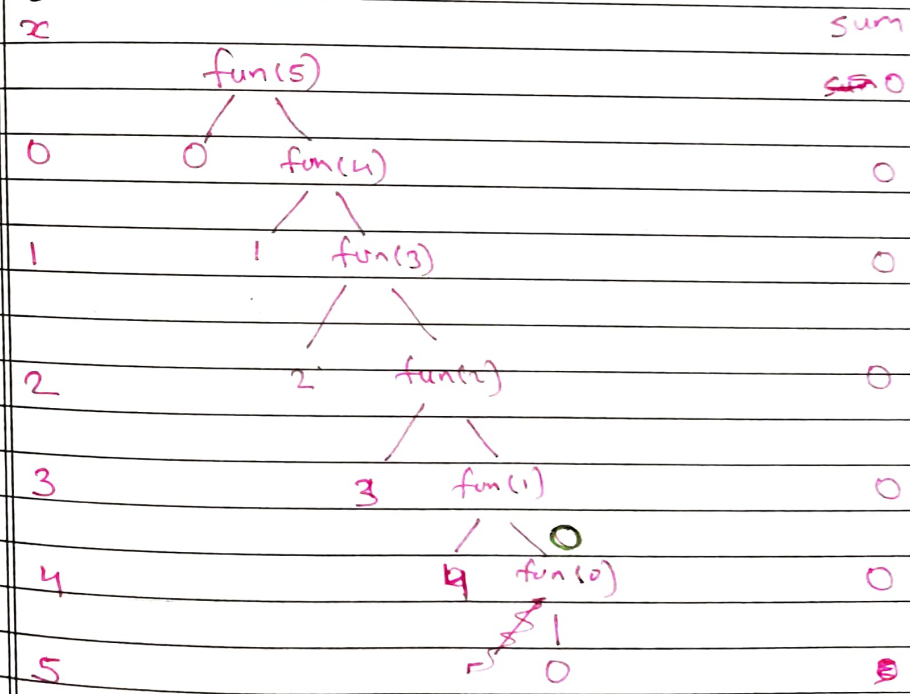
```
sum = fun(n-1) + (++x);
return sum;
```

	x
fun(5) → 10 + 5 = 15	5 0
fun(4) → 6 + 4 = 10	4 0
fun(3) → 3 + 3 = 6	3 0
fun(2) → 1 + 2 = 3	2 0
fun(1) → 0 + 1 = 1	1 0
fun(0) → 0	

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




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



Tail Recursion

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

If there is no call below the recursive function, then this is called as Tail Recursion.

In other words there should not be any thing to perform in the return phase.

Tail recursion and its loop.

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

=> For fun(3) for the o/p: 3, 2, 1

Space - O(n)

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

Space - O(1)

* Time taken by both the function is same.
* But in terms of Space. recursion occupies "n" number of activation records for input "n"

But for loop takes only one activation records.

So for loop is more efficient than tail recursion.

Head Recursion.

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

If there is no call above the recursive function, then this is called as Head recursion.

In other words there should not any thing to perform in the calling phase

Head recursion and its loop.

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

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

for fun(3) for both

0/P = 1,2,3

Linear

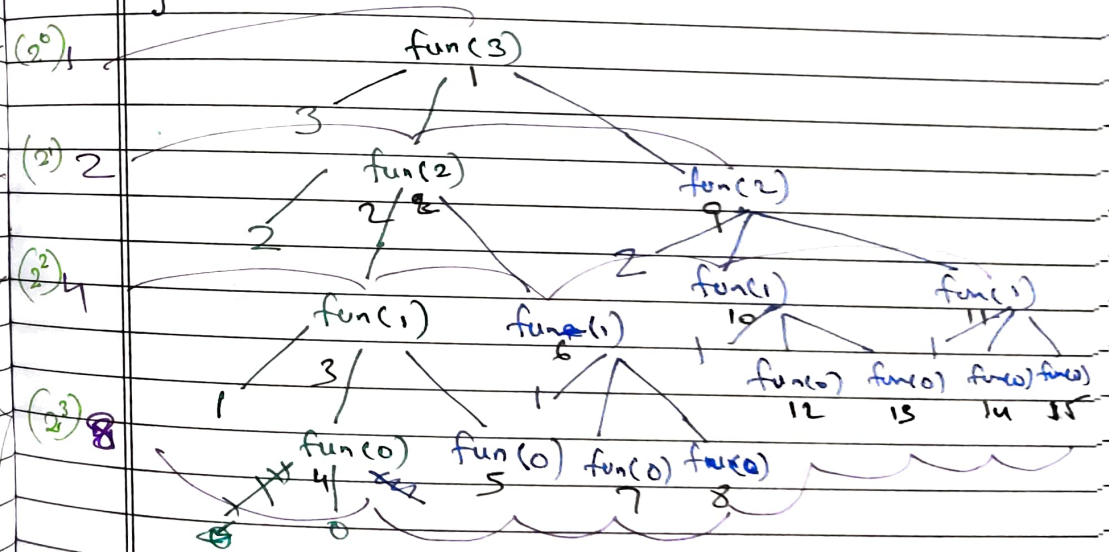
Tree recursion

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


Tree recursion

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

if a recursive function is calling itself more than one time then it is called as Tree function.



$1 + 2 + 4 + 8 = 15$ (Total number of calls)

$2^0 + 2^1 + 2^2 + 2^3 = 2^{n+1} - 1 = O(2^n)$

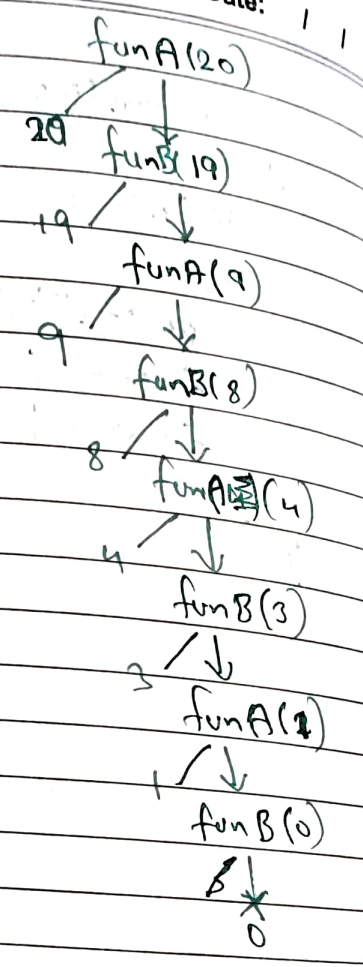
for fun(n), $2^{n+1} - 1$ calls will be made.

space is = maximum number activation records in stack

$= 4 \Rightarrow$ for fun(n) = $n+1 = O(n)$

Indirect Recursion

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

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



Ex1

Sum of first n natural numbers

For Ex:

if n=7

= 7+6+5+4+3+2+1

= 28

so for n = n+(n-1)+(n-2)+(n-3)+.....+3+2+1

Sum(n) = n+(n-1)+(n-2)+(n-3)+....+3+2+1

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)

In mathematics the problems are solved using recursive.

Opt 1: O(1)

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

Opt 2: O(n)

```
int Sum (int n)
{
   Int i, s=0;
   for(i=1; i<=n; i++)
      s=s+i;

   return s;
}.
```

Ex 2: Factorial

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

$5! = 5\times 4\times 3\times 2\times 1$

$= 120$

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

$fact(n) = n \times fact(n-1)$

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

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

Ex 3: Power Recursion.

$2^5 = 2\times 2\times 2\times 2\times 2$

$2^5 = 2^4 \times 2$

$m^n = m^{n-1} \times m$

$Pow(m,n) = Pow(m,n-1) \times m$

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

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

To reduce the steps

$2^8 = (2^2)^4$ or $2^9 = 2 \times (2^2)^4$

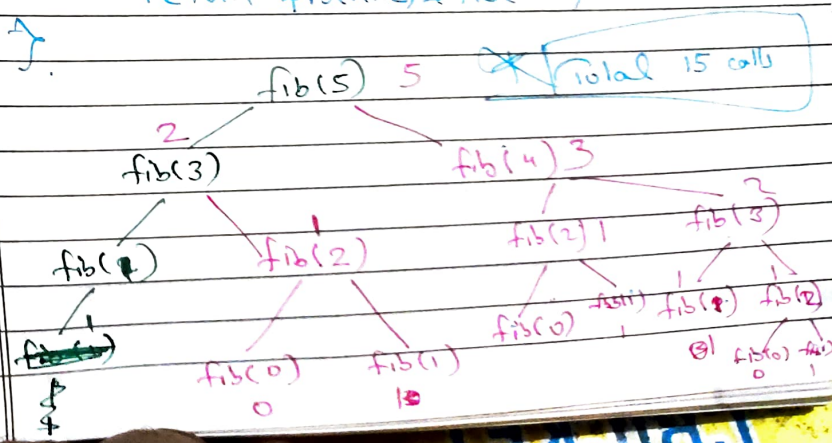
```
int pow (int m, int n)
{
    if (n==0)
        return 1;
    if (n%2==0)
        return pow(m*m, n/2)
    else
        return pow(m*m, (n-1)/2) * m
}
```

Ex: Fibonacci Series

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

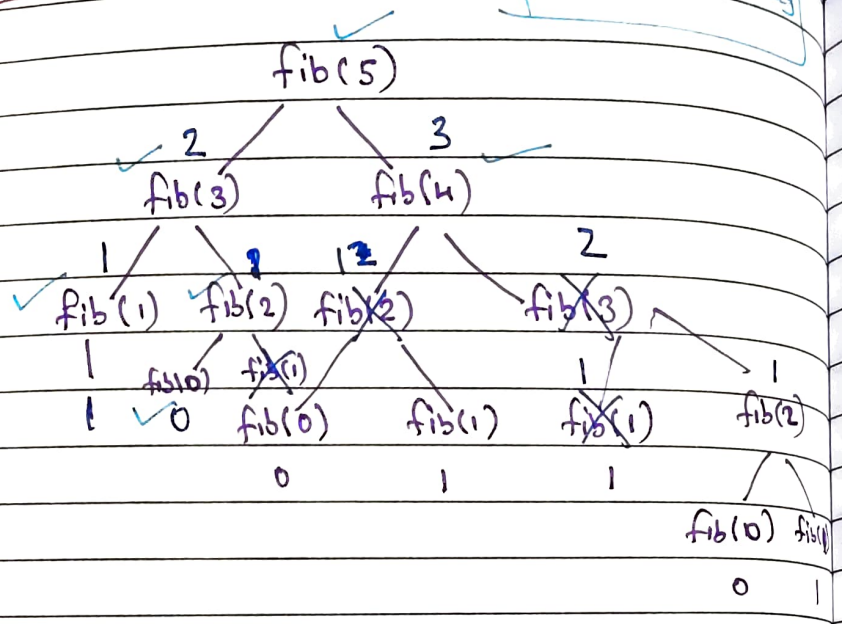
$$fib(n) = \begin{cases} 0 & n==0 \\ 1 & n==1 \\ fib(n-2) + fib(n-1) & n>1 \end{cases}$$

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



memoization.

Total 6 calls



F	0	1	1	2	3	5	-1	-1
	0	1	2	3	4	5	6	7

```
int F[10] = {-1};

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];
      }.
}
```

papergrid
Date: / /

Ex5: ^nC_r & ^nP_r

Permutation: ^nP_r

The number of possibilities for choosing an ordered set of r objects (permutation) from a total of n objects

Ex: ~~ABC~~ Formula: $$^nP_r = \frac{n!}{(n-r)!}$$

Ex: ABC => number of objects $n = 3$
What are the possible ways of ordering with only 2 objects.

AB, BC, CA, AC, CB, BA => 6 ways.

$$^nP_r = {}^3P_2 = \frac{3!}{(3-2)!} = \frac{3!}{1!} = \frac{1\times2\times3}{1} = 6 \text{ ways}$$

Combination: ^nC_r

The number of unordered, different combinations of r objects from a set of n objects.

papergrid
Date: / /

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

For ABC,
AB, BC, AC = 3 ways.

$$^nC_r = {}^3C_2 = \frac{3!}{2!(3-2)!} = \frac{3\times2\times1}{2\times1(1)} = \frac{6}{2} = 3 \text{ ways}$$

Program for combination

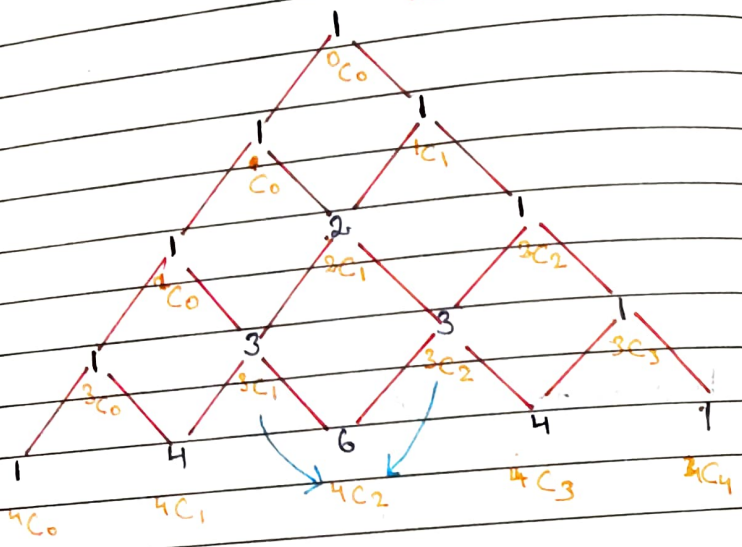
```
int C (int n, int r)
{
    int t1, t2, t3;
    t1 = fact(n);
    t2 = fact(r);
    t3 = fact(n-r);

    return t1/(t2 x t3)
}
```

papergrid

Date: / /

^nC_r Using Recursion.

Pascals Triangle



For $^4C_2 = {}^3C_1 + {}^3C_2$

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

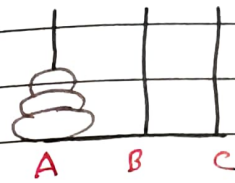
for r==0 or r==n value is 1.

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

papergrid

Date: / /

Tower of Hanoi



Rules

① One stone at a time

② Big stone should not stand on top of small stone.

Move the stones from A to C tower

B is a auxillary Tower

One Disk

TOH(1, A, B, C)

Move Disk A to C using B

Two DISK

TOH(2, A, B, C)

TOH(1, A, C, B)

Move Disk A to C using B

TOH(1, B, A, C)

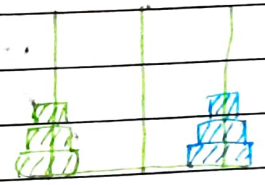
Three Disk

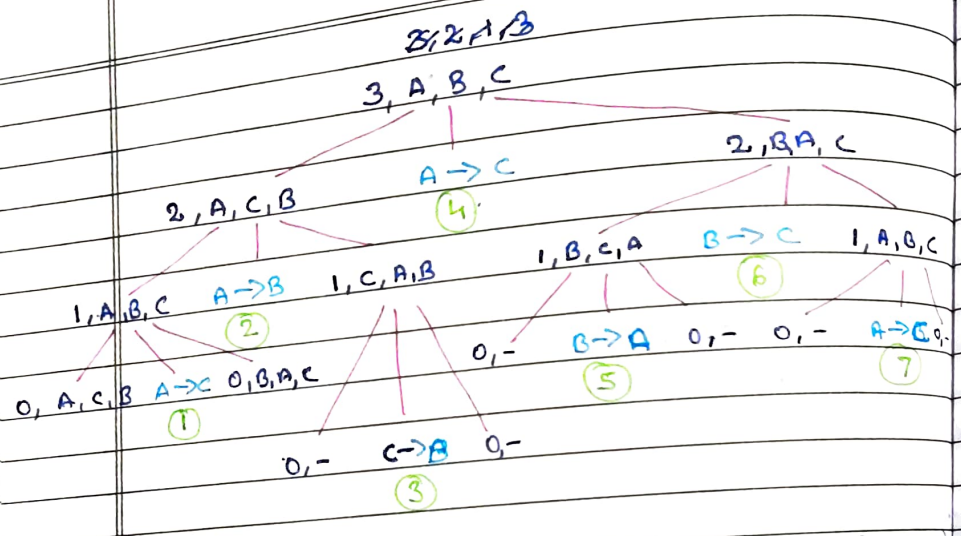
TOH(3, A, B, C)

TOH(n-1 = 2, A, C, B)

Move Disk from A to C using B // TOH(2, A

TOH(n-1 = 2, B, A, C)





(A → C) (A → B) (C → B) (A → C) (B → A) (B → C) (A → C)

Program:

```
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)
    }
}
```


Factors of a number

When a divident is divided by a divisor without a remainder then it is called or the divisor is called Factor of the divident number

Ex:-

No : 8 — Factors

1 => 8 % 1 => 0 => 1
2 => 8 % 2 = 0 => 2
3 => 8 % 3 ≠ 0 => -
4 => 8 % 4 ≠ 0 => 4
5 => 8 % 5 ≠ 0 => -
6 => 8 % 6 ≠ 0 => -
7 => 8 % 7 ≠ 0 => -
8 => 8 % 8 = 0 => 8

Here the Factors of 8 is 1, 2, 4, 8

Perfect Numbers

If the sum of the factors is equal to twice the given number, then the given number is a perfect number.