Recursion - when a function calls itself directly / indirectly we call it recursion.

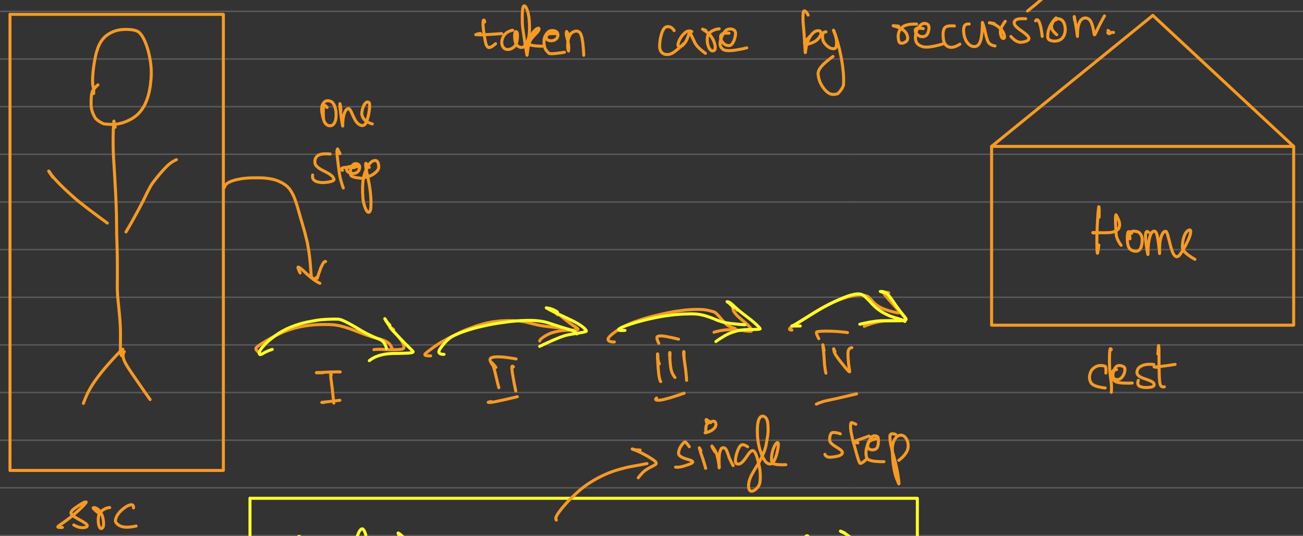
if solution of bigger problem

depends on

solution of smallest problem of same type

then we can apply recursion

important thing to understand is to solve one case and the rest would be taken care by recursion.



single step

$$\boxed{step(4) = 1 + step(3)}$$

bigger problem ← depends on → smaller problem } of same type

Mathematical Problem Statement

Ex: 1. Solve $(n) \longrightarrow$ find 2^n

if solve $(n) = 2^n$

then solve $(n-1) = 2 * 2^{n-1}$

$\therefore$ $\boxed{\text{solve}(n) = 2 * \text{solve}(n-1)}$

bigger problem

Smaller problem of same type

2. solve (n) — print counting from n to 1

solve $(n) = n, (n-1, n-2, n-3, \ldots\ldots, 1)$

if solve $(n) = (n \longrightarrow 1)$ print counting

then solve $(n-1) = (n-1 \longrightarrow 1)$

This would then be

$\boxed{\text{solve}(n) = n, \text{solve}(n-1)}$

Factorial

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

$fact(5) = 5!$

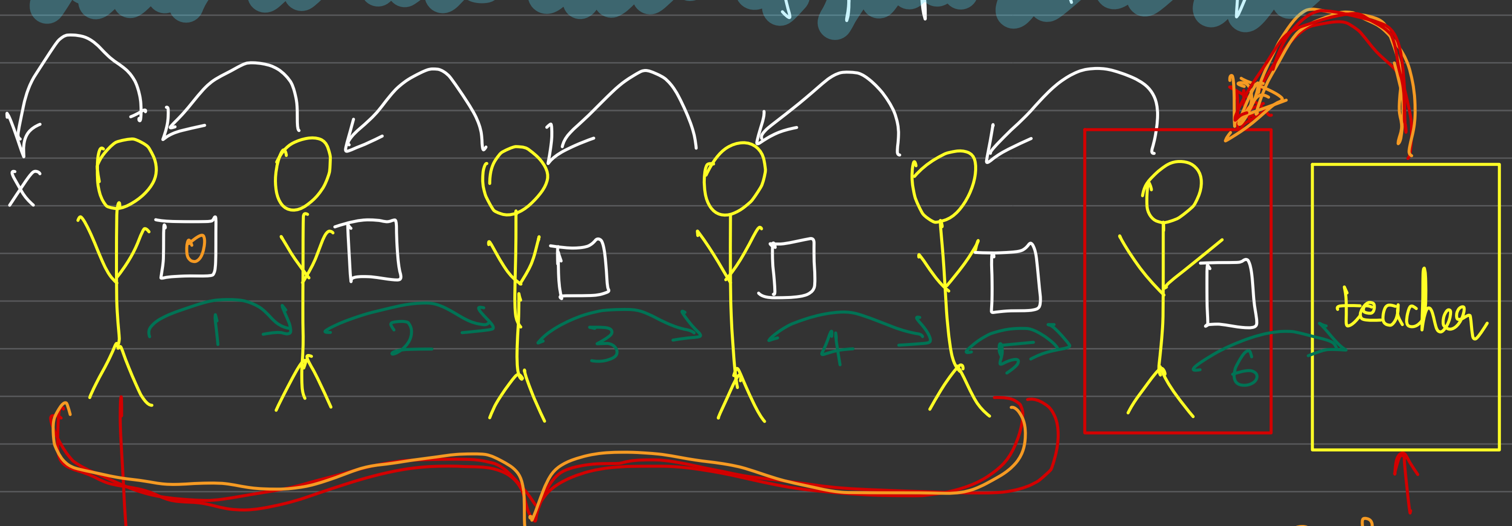
$fact(5) = 5 * 4 * 3 * 2 * 1$

$fact(4) = 4 * 3 * 2 * 1$

$\therefore \boxed{fact(5) = 5 * fact(4)}$

bigger problem depends on Smaller problem of same type.

Problem to count number of people in the queue



Stop here

These are the same function call take care by RECURSION

this is the first function made by you

Rules:

1. If someone is present behind a person pass the paper.
2. If no one present then stop passing the paper.
3. Person standing behind will pass the paper back to the person standing in front of him by adding 1.

Recursion

- Base case - mandatory
- Recursive call - mandatory
- Processing → Optional

```
int fact (int N){
    if (N==0 || N==1){
        return 1;
    }                           } Base case
    int temp = fact (N-1);      } Recursive call
    return N * temp;
}
```

Processing

multiplication or addition etc.

fact(5) → n=5

```
if(n==0||n==1)   False
    return 1;
int recAns = fact(n-1);
             24
int finalAns = n* recAns;
               5*24=120
return finalAns
```

120

fact(4) → n=4

```
if(n==0||n==1)   false
    return 1;
int recAns = fact(n-1);
             6
int finalAns = n* recAns;
               4*6=24
return finalAns;
               24
```

fact(3) → n=3

```
if(n==0||n==1)   False
    return 1;
int recAns = fact(n-1);
             2
int finalAns = n* recAns;
               3*2=6
return finalAns;
               6
```

fact(1) → n=1

```
if(n==0||n==1)   true
    return 1;
int recAns = fact(n-1);
int finalAns = n* recAns;
return finalAns;
```

fact(2) → n=2

```
if(n==0||n==1)   false
    return 1;
int recAns = fact(n-1);
             1
int finalAns = n* recAns;
               2 * 1
return finalAns;
               2
```

Print counting from N to 1 using recursion

```java
void printCounting(int N){
    if (N==1) {
        System.out.println(N);
        return;
    }
    System.out.println(N);
    printCounting(N-1);
}
//output:5 4 3 2 1
```

Base case

Processing

Recursive relation.

If a function ends with a recursive relation, that is called as a tail recursion.

if we change the ordering, where the recursion relation is called we get a different output.

```java
void printCounting(int N){
    if (N==1) {
        System.out.println(N);
        return;
    }
    printCounting(N-1);
    System.out.println(N-1);
}
//Output : 1 2 3 4 5
```

If a function has a recursive relation first and the processing comes later then it is called a head recursion

Find the value of 2^N.

$$pow(N) = 2^N$$

$$pow(N) = 2 * 2^{N-1}$$

$$pow(N) = 2 * pow(N-1);$$

2^5 32 → final output

$2 * 2^4$ 16

$2 * 2^3$ 8

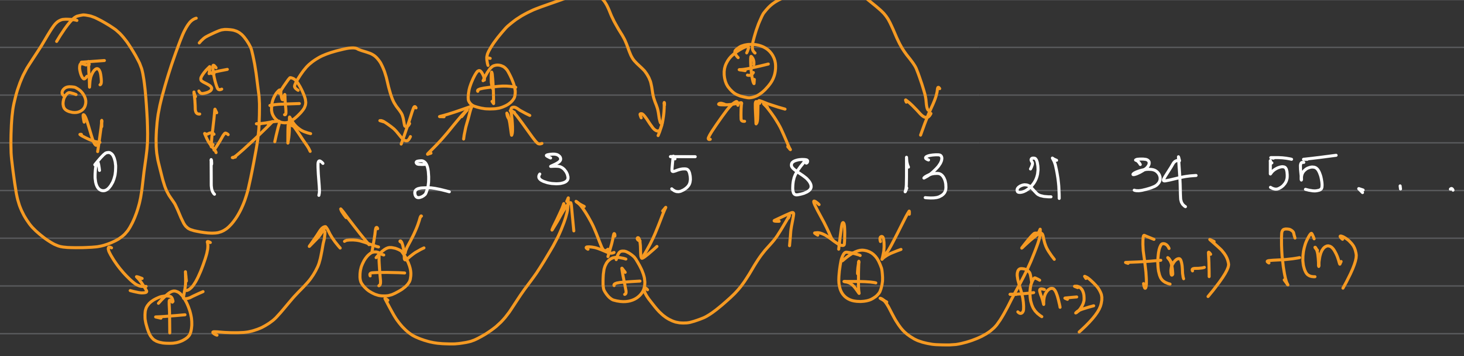
$2 * 2^2$ 4

$2 * 2^1$ 2

$2 * \boxed{2^0}$ $\boxed{=1}$

```
int pow (int N){
    if(N==0){
        return 1;
    }
    int temp = pow(N-1);
    return 2 * temp;
}
```

Fibonacci Series



find n^{th} term in a fibonacci series

$$\boxed{f(n) = f(n-1) + f(n-2)}$$

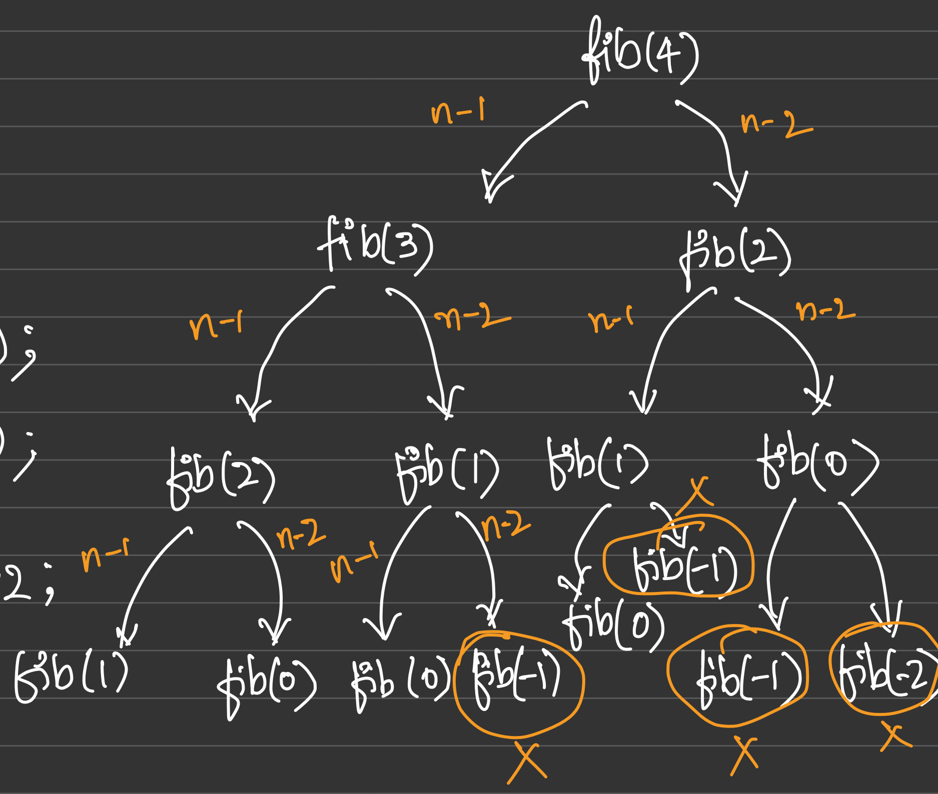
RECURSIVE RELATION

```
if(n==0)
    return 0;
if(n==1)
    return 1;
```

BASE CASE

code:

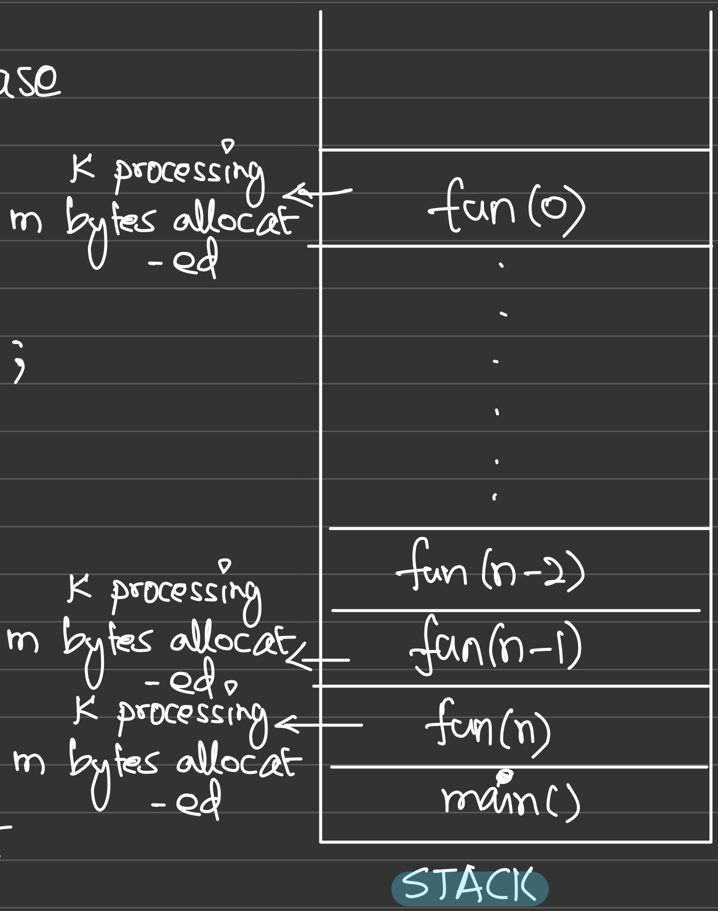
```
int fib(int n) {
    if (n==0) return 0;
    if(n==1) return 1;

    int ans1 = fib(n-1);
    int ans2 = fib(n-2);

    return ans1 + ans2;
}
```



Time & Space Complexity of Recursive Solutions

```
main( ){
   fun(n);
   return 0;
}
```

```
fun(int n){
     //base case
     if (n==0)
        return;
     //processing
     int a,b,c;

     fun(n-1);
```



for every recursive call memory is allocated in the stack until we hit the base case. Once base case is hit we unwind the stack.

Print array recursively (linear traversal of an array)

```
void  printArray (int[] arr, int n){
       if(n==0) return;
      System.out.println(n);
      printArray (arr+1, n-1);
}
```

Since time complexity is the time taken by any alogorithm with respect to input n, we will consider only n in the recursive relation.

Formula Method:

i.e., $F(n) = \underbrace{K}_{\text{processing}} + \underbrace{\cancel{F(n-1)}}_{\text{function called for (n-1) times.}}$

Likewise,
$$\cancel{F(n-1)} = K + \cancel{F(n-2)}$$
$$\cancel{F(n-2)} = K + \cancel{F(n-3)}$$
$$\vdots$$
$$\cancel{F(1)} = K + \cancel{F(0)}$$
$$\cancel{F(0)} = K_1 \rightarrow$$ K_1 because in base condition we don't call recursive relation.

---

adding → $F(n) = n \cdot K + K_1$

$O(n \cdot K + K_1) \simeq O(n)$ time complexity using formula method.

Recursive Tree Method

$P.A(n) \rightarrow K \cdot \text{times}$

$P.A(n-1) \rightarrow K \cdot \text{times}$

$P.A(n-2) \rightarrow K \cdot \text{times}$

$P.A(n-3) \rightarrow K \cdot \text{times}$

$\vdots$

$P(0) \rightarrow K1 \text{ times}$

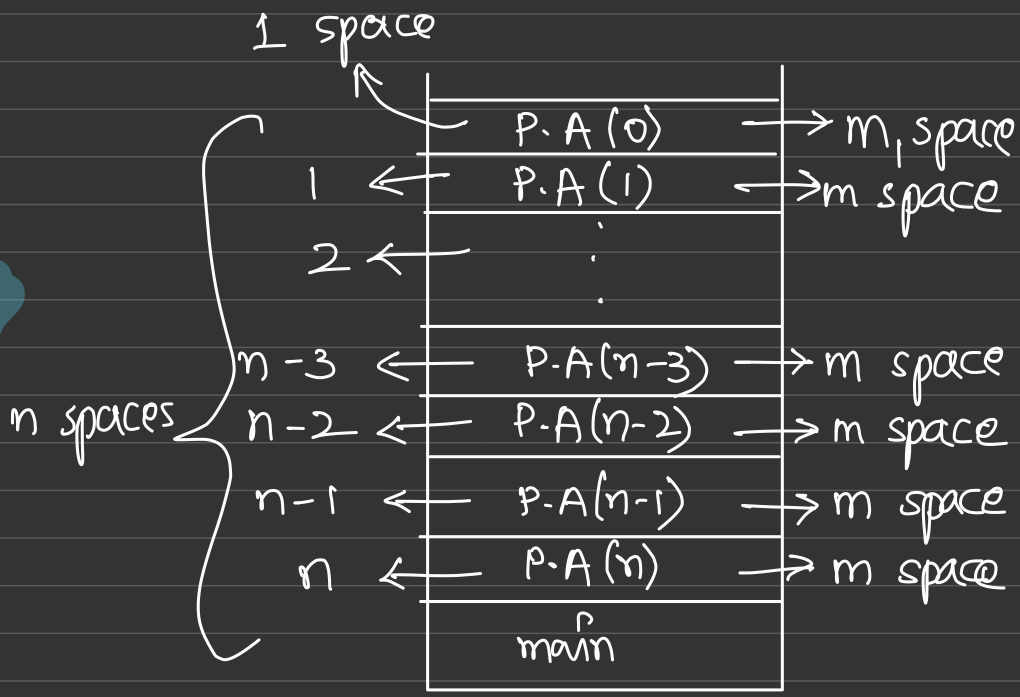
K1

$\therefore F(n) = n \cdot K + K$

n K's

$O(n \cdot K + n) \simeq O(n)$

when we us recursive relation on O.S level a call stack is maintained for all function calls allocated same amount of memory for every recursive calls.



Total space complexity:

$O(n+1) = O(n)$

Ex: Factorial of a number

```
int fact (int n){
    if (n==1)
        return 1;

    return n * fact(n-1);
}
```

$F(n) = \underbrace{n *}\, F(n-1)$

$T(n) = K + \cancel{T(n-1)}$

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

$\cancel{T(n-2)} = K + \cancel{T(n-3)}$

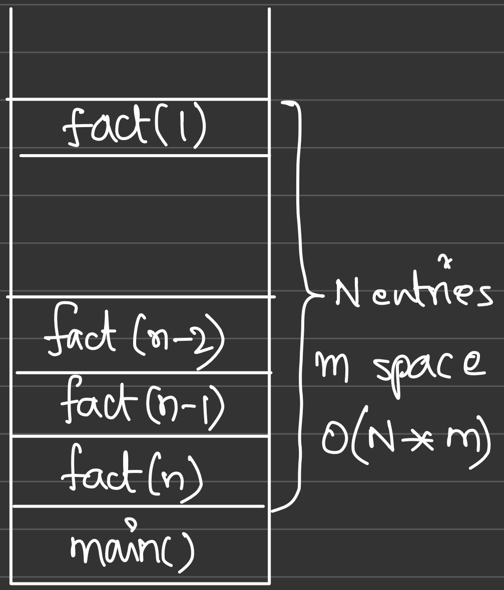
$\vdots$

$\cancel{T(1)} = K$

$T(n) = n.K$

$O(T(n)) = O(n.K) = O(n)$ time complexity

Space complexity of factorial

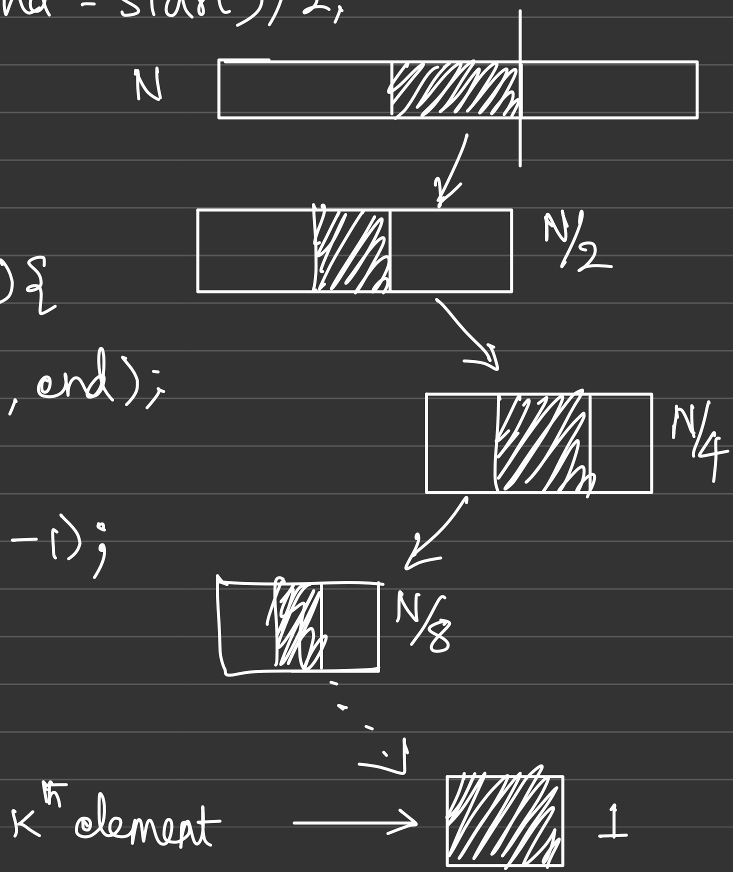


$O(N * m)$ → constant

space complexity = O(N)

Binary Search using recursion

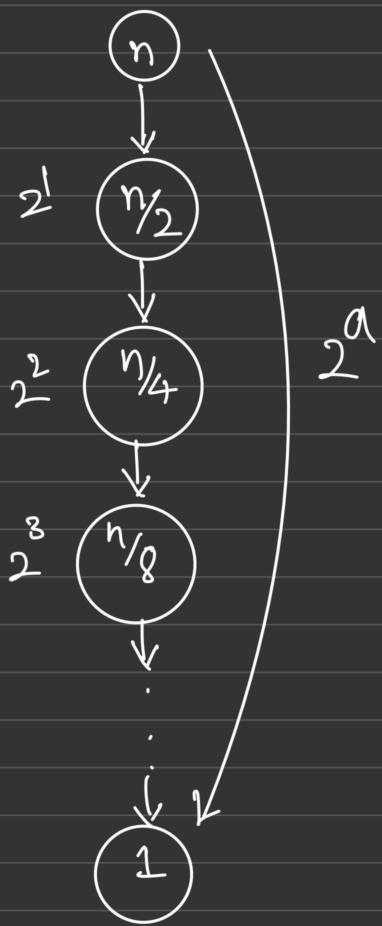
```
int  bsr(int[] a, int k, int start, int end){
    if(start > end)
        return -1;

    int mid = start + (end - start)/2;
    if( a[mid] == k)
        return mid;
    else if (k > a[mid]){
        bsr( a, k, mid+1, end);
    else
        bsr(a, k, start, mid-1);
}
```



$$F(n) = \underbrace{K}_{\text{processing}} + \underbrace{F(n/2)}_{\text{reducing input by half everytime}}$$

$$T(n) = K + \cancel{T(n/2)}$$
$$\cancel{T(n/2)} = K + \cancel{T(n/4)}$$
$$\cancel{T(n/4)} = K + \cancel{T(n/8)}$$
$$\vdots$$
$$\cancel{T(2)} = K + \cancel{T(1)}$$
$$\cancel{T(1)} = K$$

$$T(n) = a * K$$



To reduce to one sized input we require 2^a operations because everytime we divide by 2^a

$$\therefore \frac{n}{2^a} = 1 \rightarrow$$ size of the array when we reduce array elements (n) by half ($1/2^a$) everytime

$$n = 2^a$$

$$a = \log_2 n$$

$$T(n) = \log n * K$$ $\rightarrow$ constant will be ignored

$$\therefore T(n) = \log n \Rightarrow O(T(n)) = O(\log n)$$

time complexity

space complexity

we recursively call the same function again & again.

$a = \log n$

$K * \log(n)$

$O(T(n)) = O(K \cdot \log n)$

$O(n) = O(\log n)$ space complexity

bsr(1)	$n/2^a = 1$ - K space
⋮	
bsr(n/8)	n/8 - K space
bsr(n/4)	n/4 - K space
bsr(n/2)	n/2 - K space
bsr(n)	n - K space
main()	

Fibonacci series recursive algorithm

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

c time for processing

fib(n) → $2^0 * C$

(n-1) (n-2) → $2^1 * C$

(n-2) (n-3) (n-3) (n-4) → $2^2 * C$

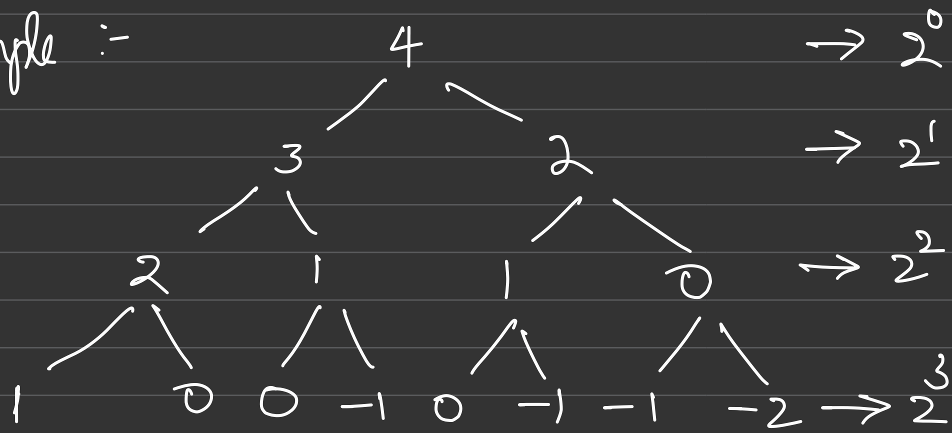
(n-3) (n-4) (n-4) (n-5) (n-4) (n-5) (n-5) (n-6) → $2^3 * C$

⋮

$2^{n-1} * C$

why 2^{n-1} ?

Example :-



for $n=4$, we go till 2^3 or 2^{n-1}

$$T(n) <= 2^0 + 2^1 + 2^2 + 2^3$$

we are not having all the nodes in 2^3 level because of negatives hence we say <=

$$\therefore \quad T(n) \leq 2^0 * c + 2^1 * c + 2^2 * c + 2^3 * c$$

$$T(n) \leq c\left(2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{n-1}\right)$$

geometric series

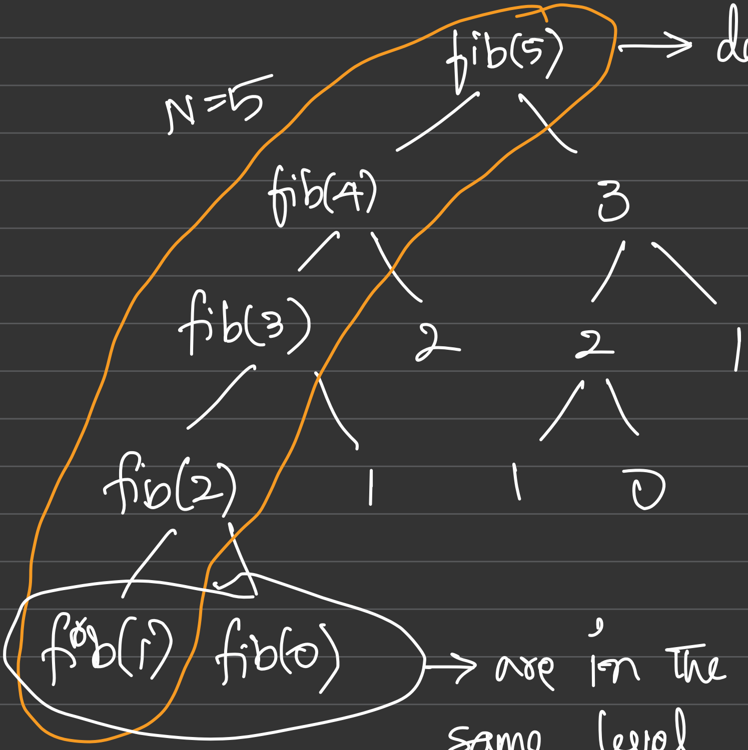
$$T(n) \leq c\left(2^n - 1\right)$$

$$\boxed{T(n) \leq 2^n - 1}$$

$T(n) = O(2^n)$ for upper bound

space complexity of fibonacci

N=5



→ depth = 5

→ are in the same level

fib(1)
fib(2)
fib(3)
fib(4)
fib(5)
main

∴ S.C => O(n)