What is time complexity ? (Discussed in Lecture 11)

→ Time complexity is the order of growth of a function measured against the input size.

✳ Recursive Function Examples :

① Factorial:

```
int factorial(int n) {
    if(n <= 1)
        return 1;
    return n * factorial(n-1);
}
```

Time Complexity = ??

Step 1 : Write the recurrance relation as a function.

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

Step 2 : Break down the function into components and include their respective Time complexities in a new function T(n), which is the total Time Complexity.

$$T(n) = \underbrace{\quad}_{\text{Part1}} + \underbrace{\quad}_{\text{Part 2}} + \text{----}$$

$$T(n) = K_1 + K_2 + T(n-1)$$

Part 1 : if {

```
int factorial(int n) {
    if(n <= 1)
        return 1;
    return n * factorial(n-1);
}
```

Part 3 : Recursive Call

Part 2 (Multiplication operation)

We will take all constant-time components together (k) and write the recurrance relation as a function of n.

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

Repeat this for the smaller input(s) in eq (1)

$$T(n-1) = k + T(n-2)$$
$$T(n-2) = k + T(n-3)$$
$$\vdots$$
$$T(2) = k + T(1)$$
$$T(1) = 1 \quad \text{(Base Case)}$$

If you add up all these n equations, you get:

$$\begin{aligned} T(n) &= k + T(n-1) \\ T(n-1) &= k + T(n-2) \\ + \; T(n-2) &= k + T(n-3) \\ + \; T(n-3) &= k + T(n-4) \\ + \; T(n-4) &= k + T(n-5) \\ &\vdots \\ + \; T(2) &= k + T(1) \\ + \; T(1) &= 1 \end{aligned}$$

$$T(n) = \underbrace{k + k + \dots + k}_{n \text{ times}} + 1$$

$$T(n) = nk + 1 \approx nk \quad \text{(For large values of n)}$$

$$\boxed{T(n) = O(n)} \quad \text{(Order of n)}$$

② Binary Search:

Constant { Part 1 { Part 2 {

```
bool binarySearch(int arr[], int val, int l, int r) {
    if(l>r) return false;

    int mid = l + (r-l)/2;
```

Constant { Part 1 {
Part 2 {
Part 3 {
Part 4 {
Part 5 {
Constant { Part 6 {

```
    if(l>r) return false;

    int mid = l + (r-l)/2;

    if(arr[mid] == val)
        return true;
    else if(arr[mid] > val)
        return binarySearch(arr, val, l, mid-1);
    else if(arr[mid] < val)
        return binarySearch(arr, val, mid+1, r);

    return false;
}
```

Clearly,

$$f(n) = C + f(n/2)$$

(Only one of the 2 recursive functions will be called for each call)

$$T(n) = K + T(n/2)$$
$$+ \quad T(n/2) = K + T(n/4)$$
$$+ \quad T(n/4) = K + T(n/8)$$
$$+ \quad \vdots \quad \vdots \quad \vdots \quad \vdots$$
$$+ \quad T(2) = K + T(1)$$
$$+ \quad T(1) = 1$$

$$T(n) = a * K + 1$$

[where a is the no. of times we halved n to reach to the base case]

What is a ?

$$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \ldots \rightarrow 1$$

a times.

Using G.P:

first term = n

last term = 1

common ratio = $1/2$

$$a^{th} \text{ term} = n\left(\frac{1}{2}\right)^{a-1} = 1$$

$$\Rightarrow n = 2^{(a-1)}$$

$$\Rightarrow \log n = a - 1$$

$$\Rightarrow a \approx \log n \quad \text{(For large values of n)}$$

$$T(n) = (\log n) * k + 1$$

$$T(n) \approx k * \log n$$

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

③ Merge Sort :

Constant T.C. { Part 2 { Part 3 {

Part 4 {

```
void MergeSort(vector<int> &v, int l, int r)
{
    if(l >= r) return;
    int mid = l + (r - l)/2;
    MergeSort(v, l, mid);
    MergeSort(v, mid+1, r);

    merge(v, l, mid, r);
}
```

$$T(n) = k + T(n/2) + T(n/2) + \underbrace{nk'}_{\text{Part 4}}$$

Copying the merged array in the original array

$$T(n) = \underbrace{k}_{\text{ignore constant}} + nk' + 2T(n/2)$$

as it's insignificant as compared to nk'.

$$T(n) = nk' + 2T(n/2)$$

$$+ \left[T(n/2) = \frac{n}{2}k' + 2T(n/4)\right] \times 2$$

$$+ \left[T(n/4) = \frac{nk'}{4} + 2T(n/8)\right] \times 4$$

$$\vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots$$

$$+ \left[T(2) = 2k' + 2T(1)\right] \times \left(\frac{\log n}{2}\right)$$

$$+ \left[T(1) = 1\right] \times (\log n)$$

$$\Rightarrow \quad T(n) = nk' + 2T(n/2)$$

$$2 \times T(n/2) = nk' + 4T(n/4)$$

$$4 \times T(n/4) = nk' + 8T(n/8)$$

$$8 \times T(n/8) = nk' + 16T(n/16)$$

$$T(n) = \log_2 n \times nk' = \boxed{O(n \log n)}$$

④ Fibonacci Number :

$$f(n) = K + f(n-1) + f(n-2)$$

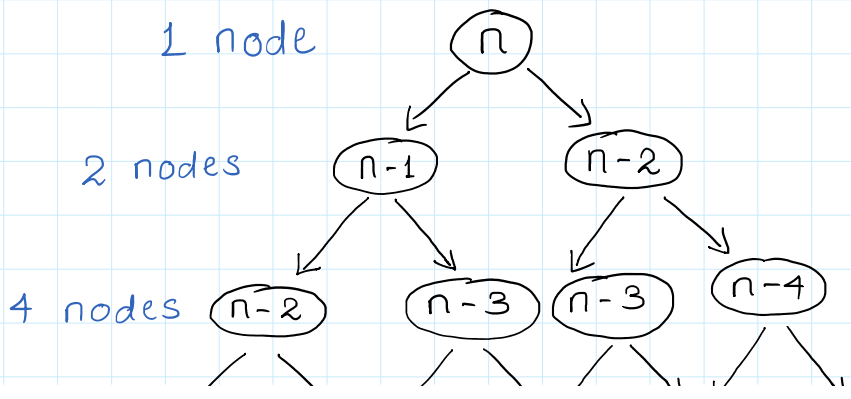
$$T(n) = K + T(n-1) + T(n-2)$$

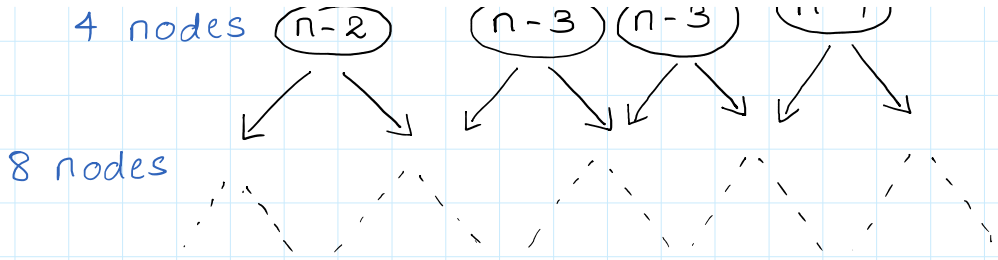
$$T(n-1) = K + T(n-2) + T(n-3)$$

$$T(n-2) = K + T(n-3) + T(n-4)$$

$$T(n-3) = K + T(n-4) + T(n-5)$$

Use a recursion tree for calculating Time Complexity.





2^n nodes.

Total number of nodes $= 1 + 2 + 4 + 8 + \ldots + 2^n$

$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^n = 1\left(\frac{2^{n+1} - 1}{2 - 1}\right)$

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

If each node takes k amount of time,

$T.C = K \times 2 \times 2^n - K \approx 2K * 2^n \approx \boxed{O(2^n)}$

A few recursive questions :

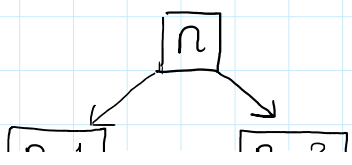
(1) Climb Stairs -

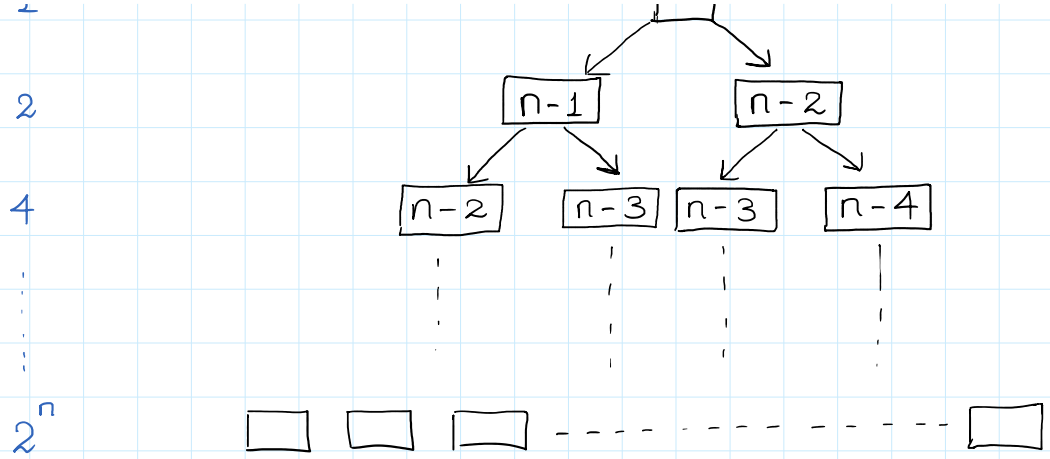
Const

```
int countDistinctWayToClimbStair(long long nStairs)
{
    //base case
    if(nStairs < 0)
        return 0;

    if(nStairs == 0)
        return 1;

    //R.C
    int ans = countDistinctWayToClimbStair(nStairs-1)
        + countDistinctWayToClimbStair(nStairs-2);

    return ans;
}
```

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

1





Height of the tree = n.

For depth = i , no. of nodes = 2^i

Total no. of nodes = $1 + 2 + 4 + 8 + \ldots + 2^n$

$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^n$

$= 1\left\{\frac{2^{n+1} - 1}{2 - 1}\right\} = (2^{n+1} - 1)$

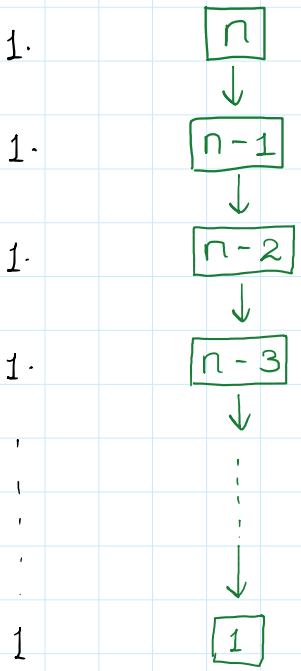
Time Complexity = $O(2^n)$ (Same as Fibonacci)

② Linear Search :

```
bool linearSearch(int arr[], int size, int k ) {

    //base case
    if(size == 0)
        return false;

    if(arr[0] == k) {
        return true;
    }
    else {
        bool remainingPart = linearSearch(arr+1, size-1, k );
        return remainingPart;
    }

}
```

Applies for `is array sorted' and other linear questions.

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



Total no. of nodes = $\underbrace{1+1+1+1+\ldots.+1}_{n \text{ times}}$

T.C. = n = $O(n)$

③ Finding Power :

Constant

Constant

```
int power(int a, int b) {
    //base case
    if( b == 0 )
        return 1;

    if(b == 1)
        return a;

    //RECURSIVE CALL
    int ans = power(a, b/2);

    //if b is even
    if(b%2 == 0) {
        return ans * ans;
    }
    else {
        //if b is odd
        return a * ans * ans;
    }
}
```

$T(n) = k + T(n/2)$ (Same as Binary Search)

T.C. = $O(\log n)$

④ Quick Sort :

```
void solve(vector<int>& arr, int s, int e) {

    //base case
    if(s >= e)
        return ;

    //partitioon karenfe
    int p = partition(arr, s, e);

    //left part sort karo
    solve(arr, s, p-1);

    //right wala part sort karo
    solve(arr, p+1, e);

}
```

In the best case, the pivot element will get placed in the middle after partitioning. This will become similar to merge sort T.C.

$$T.C. = nK + 2T(n/2)$$

$$\text{Time Complexity} = O(n \log n)$$

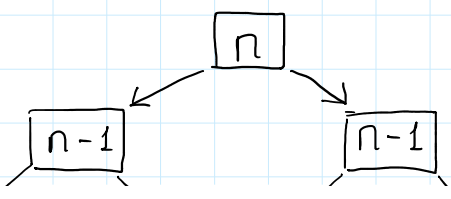
⑤ Subsequences / Subsets :

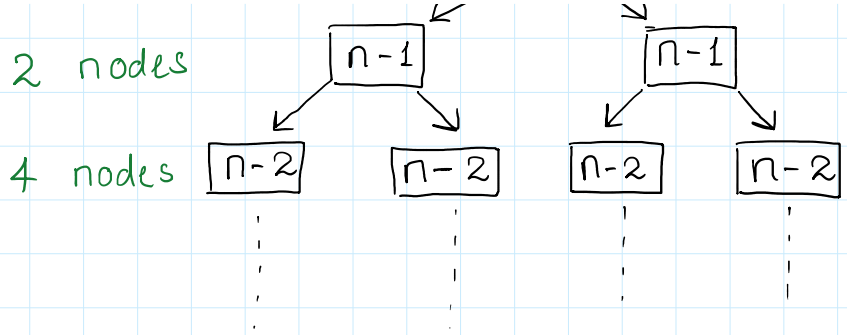
```
void solve(vector<int> nums, vector<int> output, int index, vector<vector<int> >& ans) {
    //base case
    if(index >= nums.size()) {
        ans.push_back(output);
        return ;
    }

    //exclude
    solve(nums, output, index+1, ans);

    //include
    int element = nums[index];
    output.push_back(element);
    solve(nums, output, index+1, ans);

}
```

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

1. node.

2 nodes





2^n nodes

Total nodes $= 1 + 2 + 4 + \ldots + 2^n$

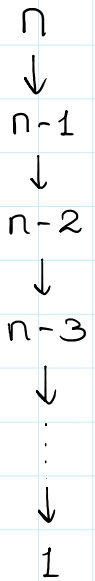
$= 1\left[\frac{2^{n+1} - 1}{2 - 1}\right] = (2^{n+1} - 1)$

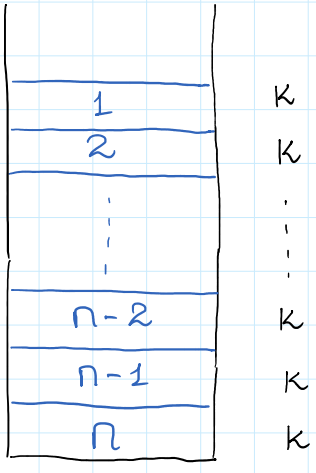
$T.C. = O(2^n)$

If you are printing / storing the elements of every subsequence, then $T.C. = O(2^n \times n)$

Space Complexity : Maximum space required at any instance.

(1) Factorial :





Call Stack Space

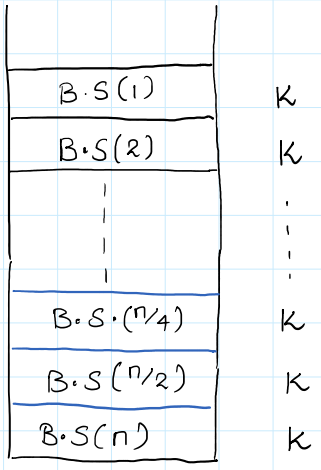
∵ Unless we reach base case $n=1$, all the calls will remain in the stack space, thus when we are calling for $n=1$, all n calls will be in the stack.

$$S.C. = k * n = O(n)$$

② Binary Search :

$S.C. = k * \log(n)$

$S.C = O(\log n)$



Call Stack Space

③ Merge Sort :

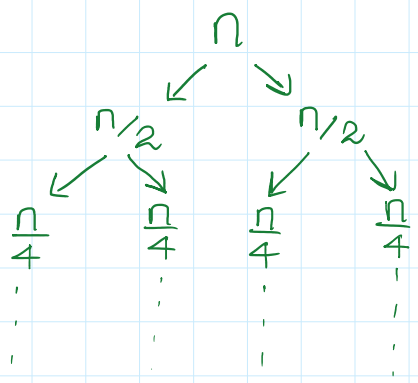
We are creating $2-\frac{n}{2}$ sized arrays for every n sized array that is used in a call.

Total = n

Total = n

Total = n

Total = n



$\log n$ height.

Space Complexity = $O(n \log n)$