



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 recurrence 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.

$$T(n) = \underbrace{\quad}_{\text{Part 1}} + \underbrace{\quad}_{\text{Part 2}} + \dots$$

$$T(n) = k_1 + k_2 + T(n-1)$$

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

Part 1 : if { Part 3 : Recursive Call
 ↑
 Part 2 (Multiplication operation)

We will take all constant-time components together (K) and write the recurrence 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)

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

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

⋮ ⋮ ⋮ ⋮ ⋮

⋮ ⋮ ⋮ ⋮ ⋮

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

$$T(1) = 1 \quad (\text{Base Case})$$

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

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

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

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

$T(n) = O(n)$

(Order of n)

② Binary Search :

Constant $\left\{ \begin{array}{l} \text{Part 1} \\ \text{Part 2} \end{array} \right. \{$

```
bool binarySearch(int arr[], int val, int l, int r) {
    if(l > r) return false;
    int mid = l + (r-l)/2;
```

```

Constant { Part 1  if(l > r) return false;
            Part 2  {
                Part 3  {
                    Part 4  {
                        Part 5  {
                            Part 6  {
                                Part 7  int mid = l + (r-l)/2;
                                Part 8  if(arr[mid] == val)
                                         return true;
                                Part 9  else if(arr[mid] > val)
                                         return binarySearch(arr, val, l, mid-1);
                                Part 10 else if(arr[mid] < val)
                                         return binarySearch(arr, val, mid+1, r);
                                Part 11
                                Part 12 return false;
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Clearly,

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

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

$$\begin{aligned}
 T(n) &= K + T(n/2) \\
 + T(n/2) &= K + T(n/4) \\
 + T(n/4) &= K + T(n/8) \\
 + \vdots & \vdots \quad \vdots \quad \vdots \quad \vdots \\
 + T(2) &= K + T(1) \\
 + T(1) &= 1
 \end{aligned}$$

$$\underline{\underline{T(n) = a * K + 1}}$$

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

What is a?

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

a times.

Using G.P :

$$\text{first term} = n$$

$$\text{last term} = 1$$

$$\text{common ratio} = 1/2$$

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

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

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

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

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

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

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

③ Merge Sort :

```

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);
}

Constant T.C:
Part 2 {
Part 3 {
Part 4 {
}
}
}

```

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

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' .

$$\begin{aligned}
T(n) &= nk' + 2T(n/2) \\
&+ [T(n/2) = \frac{nk'}{2} + 2T(n/4)] \times 2 \\
&+ [T(n/4) = \frac{nk'}{4} + 2T(n/8)] \times 4 \\
&\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
&+ [T(2) = 2k' + 2T(1)] \times (\frac{\log n}{2}) \\
&+ [T(1) = 1] \times (\log n)
\end{aligned}$$

$$\Rightarrow T(n) = nk' + 2T(\frac{n}{2})$$

$$2 \times T(\frac{n}{2}) = nk' + 4T(\frac{n}{4})$$

$$4 \times T(\frac{n}{4}) = nk' + 8T(\frac{n}{8})$$

$$8 \times T(\frac{n}{8}) = nk' + 16T(\frac{n}{16})$$

| | | | |

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

④ Fibonacci Number :

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

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

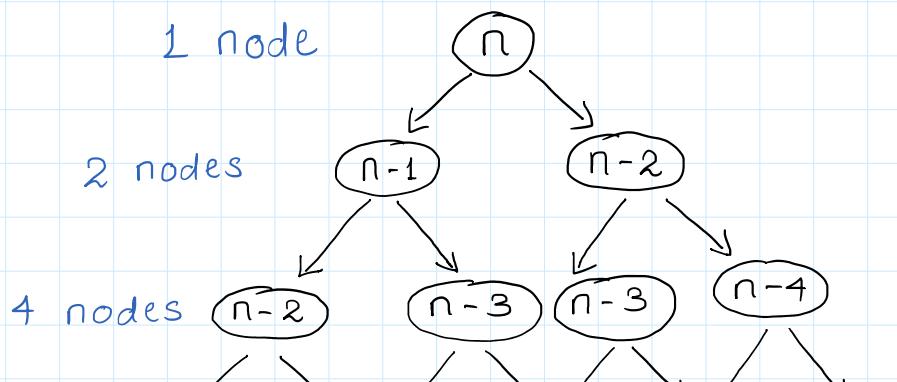
$$T(n-1) = k + T(n-2) + T(n-3)$$

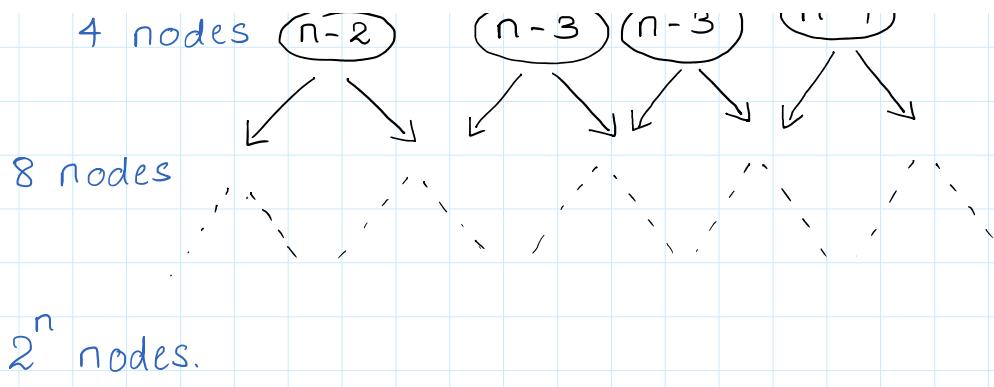
$$T(n-2) = k + T(n-3) + T(n-4)$$

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

| | | |

Use a recursion tree for calculating Time Complexity.





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

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

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

If each node takes k amount of time,

$$T.C = k \times 2 \times 2^n - k \approx 2k * 2^n \approx O(2^n)$$

A few recursive questions:

① Climb Stairs -

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

    if(nStairs == 0)
        return 1;

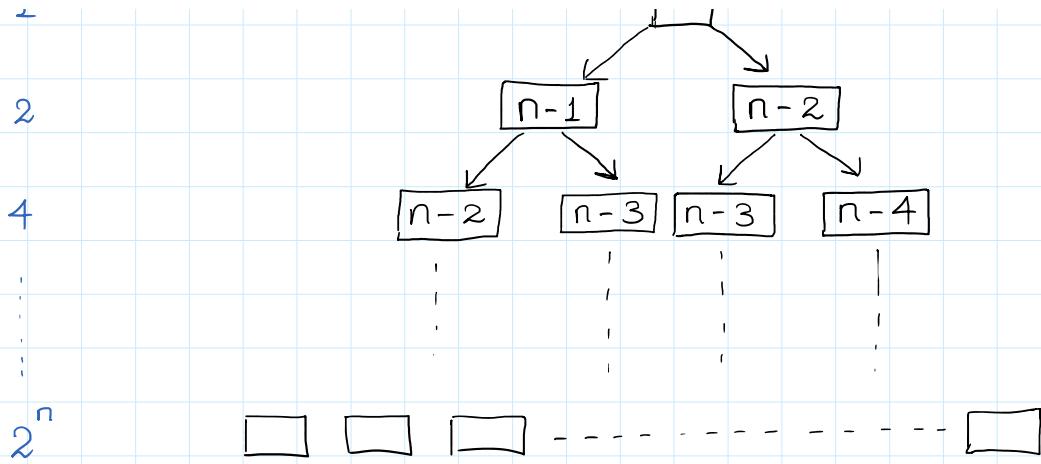
    //R.C
    int ans = countDistinctWayToClimbStair(nStairs-1)
              + countDistinctWayToClimbStair(nStairs-2);

    return ans;
}
```

Const

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





Height of the tree = n .

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

$$\begin{aligned}
 \text{Total no. of nodes} &= 1 + 2 + 4 + 8 + \dots + 2^n \\
 &= 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n \\
 &= 1 \left[\frac{2^{n+1} - 1}{2 - 1} \right] = (2^{n+1} - 1)
 \end{aligned}$$

Time Complexity = $\mathcal{O}(2^n)$ (Same as Fibonacci)

② Linear Search :

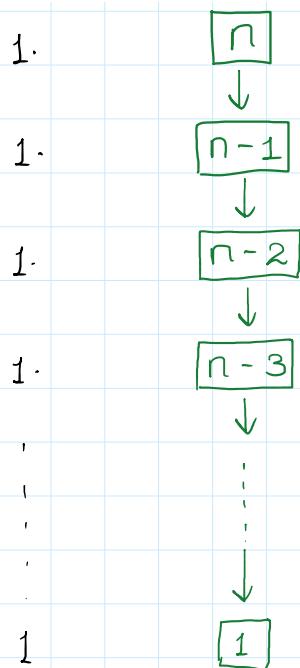
```

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 = $\underbrace{1+1+1+1+\dots+1}_{n \text{ times}}$

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

③ Finding Power :

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;
    }
}

```

Constant {

$$T(n) = k + T(n/2) \quad (\text{Same as Binary Search})$$

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

④ Quick Sort :

```

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

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

    //partition 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);

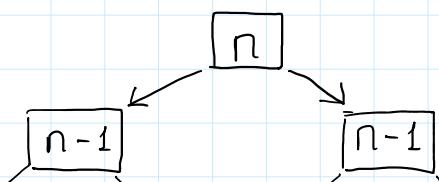
    //include
    int element = nums[index];
    output.push_back(element);
    solve(nums, output, index+1, ans);
}

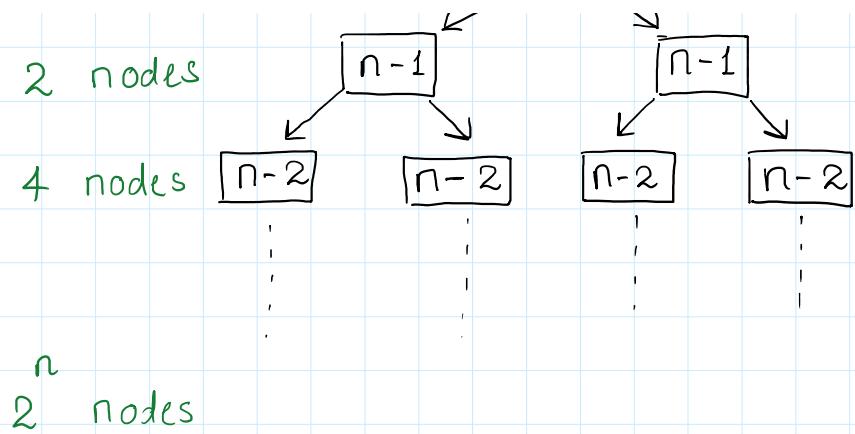
```

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

1. node.

2 nodes





$$\text{Total nodes} = 1 + 2 + 4 + \dots + 2^n$$

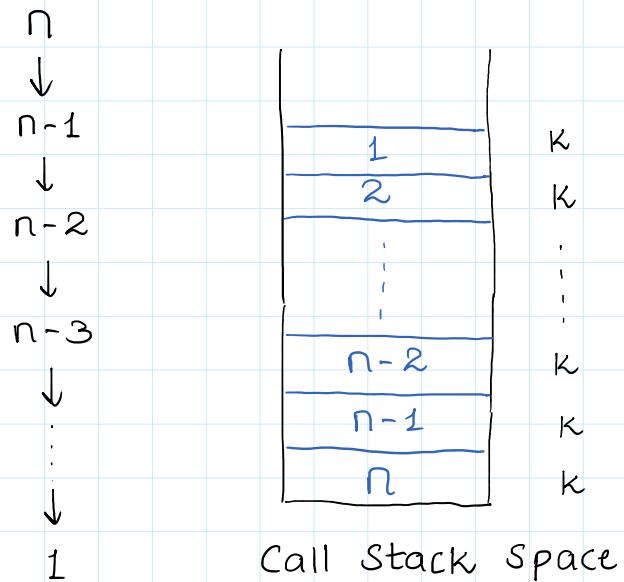
$$= 1 \left[\frac{2^{n+1} - 1}{2 - 1} \right] = (2^{n+1} - 1)$$

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

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 :



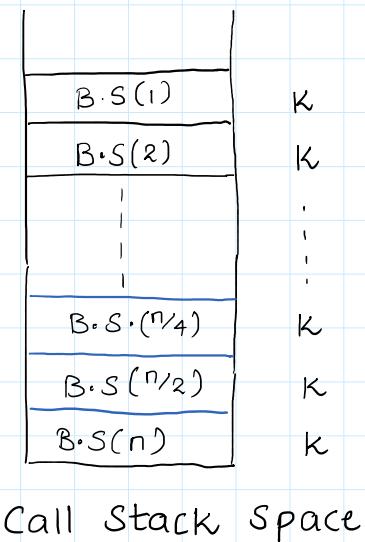
∴ Unless we reach base case $n=1$, all the calls will remain in the stack space, thus when we are calling for $n=1$, all n calls will be in the stack.

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

② Binary Search :

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

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



③ Merge Sort :

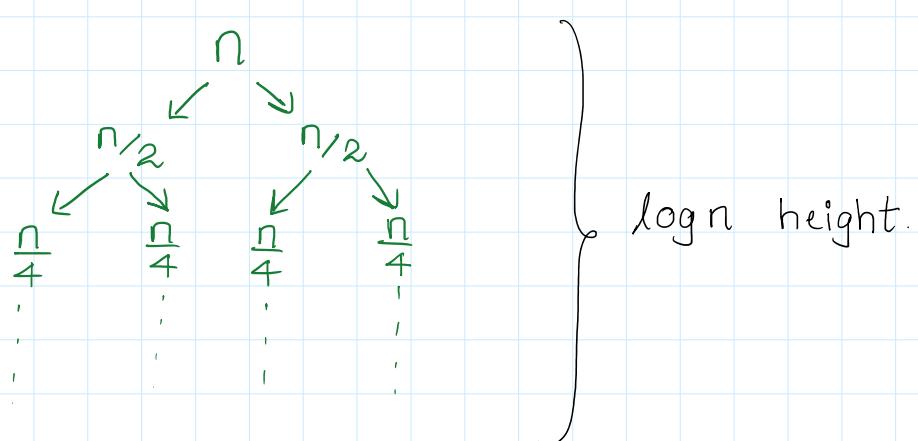
We are creating $2 - \frac{n}{2}$ sized arrays for every n sized array that is used in a call.

$$\text{Total} = n$$

$$\text{Total} = n$$

$$\text{Total} = n$$

$$\text{Total} = n$$



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