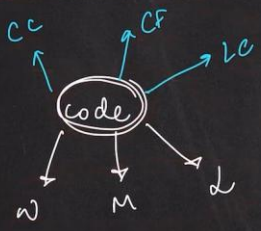
Chapter-12

Time & Space Complexity

Time Complexity

NOT the actual time taken but amount of time taken as function of input size (n).



Time Complexity

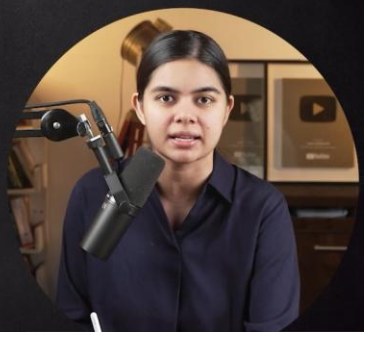
NOT the actual time taken but amount of time taken as function of input size (n).

operation

O(4)

```
int x=10;
for(___) {

}
```

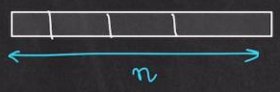
operation



Time Complexity

NOT the actual time taken but amount of time taken as function of input size (n).

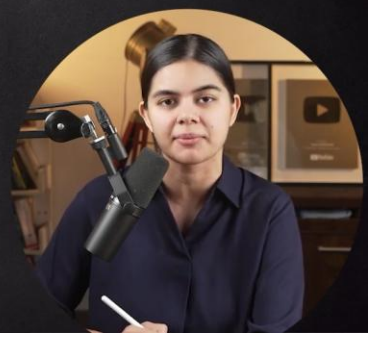
Linear Search



target

$n=1 \approx 1$ op
$n=10 \approx 10$ op
$n=100 \approx 100$ op
$n=10^5 \approx 10^5$ op

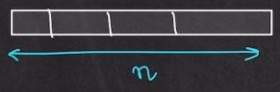
```
for(i=0; i<n; i++) {
    if(arr[i] == target)
        return i;
}
return -1;
```



Time Complexity

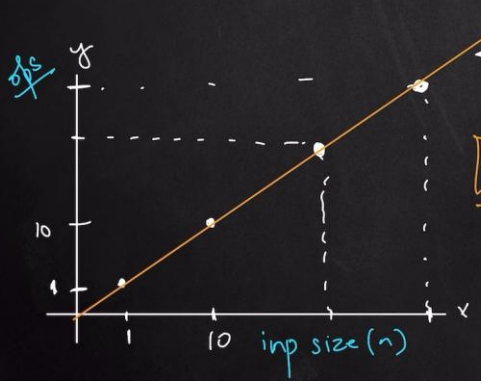
NOT the actual time taken but amount of time taken as function of input size (n).

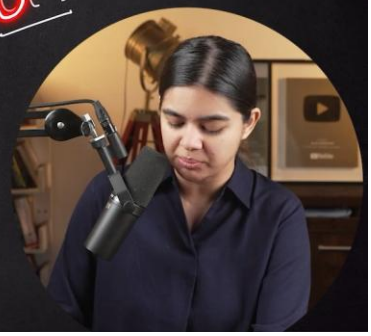
Linear Search



target

$n=1 \approx 1$ op
$n=10 \approx 10$ op
$n=100 \approx 100$ op
$n=10^5 \approx 10^5$ op





Time Complexity

5, 100, 1000

10^5, 10M, 1Bn

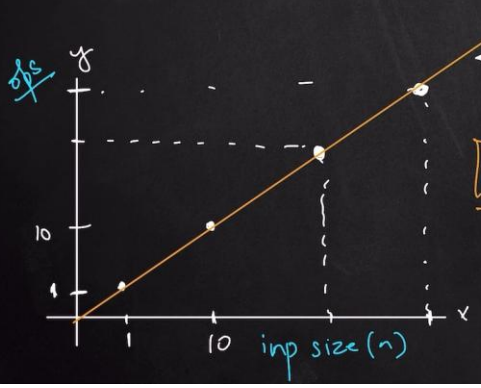
worst case scenarios

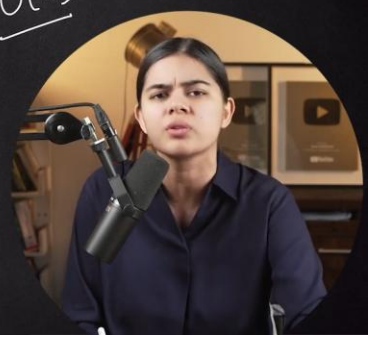
inp size → n

NOT the actual time taken but amount of time taken as function of input size (n).

Linear Search

target





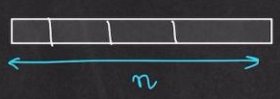
Time Complexity

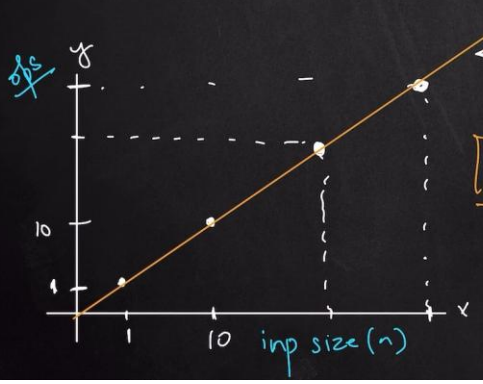
O(n) O(n²)

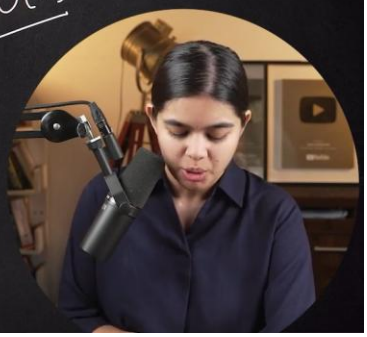
worst case scenarios → inp size → n

NOT the actual time taken but amount of time taken as function of input size (n).

Linear Search

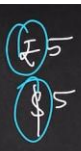




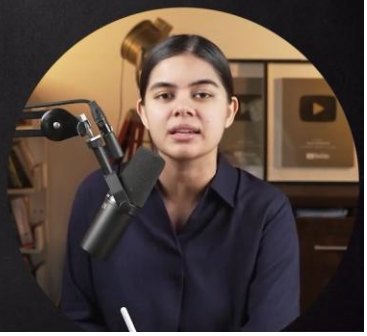


Big O Notation

└ symbol



O(n)

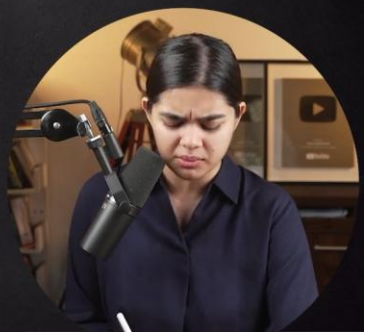


Big O Notation

└→ worst case comp (upper bound)

O(n)

for = 1



Big O Notation

↳ worst case comp (upper bound)

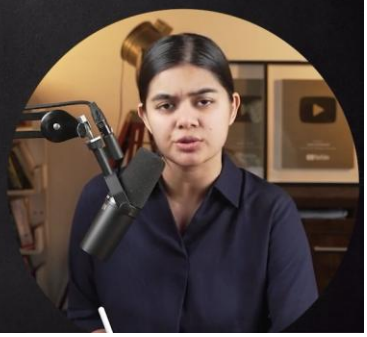
1. ignore constants
2. largest term

$f(n) = 4n^2 + 3n + 5$

$n = 10^5$

$n^2 \; n \; 1$

$10^{10} \quad 10^5 \quad 1$



Big O Notation

↳ worst case comp (upper bound)

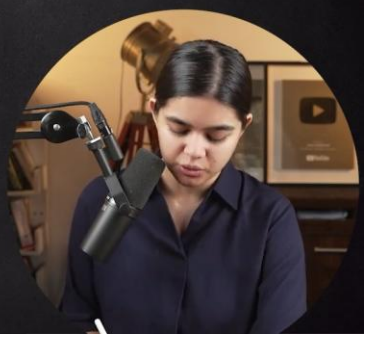
1. ignore constants
2. largest term

$f(n) = 4n^2 + 3n + 5$

$n = 10^5$

$n^2 + n + 1$

$O(n^2)$



Big O Notation

↳ worst case comp (upper bound)

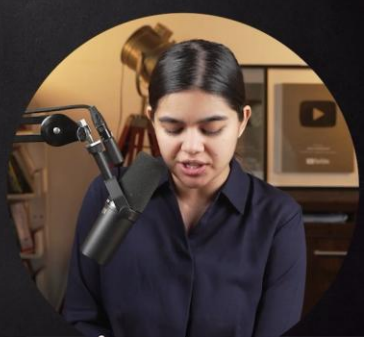
1. ignore constants
2. largest term

$f(n) = 100 + 5n^2 + \sqrt{n}$

$n = 10^6$

$1 + n^2 + \sqrt{n}$

$O(n^2)$



Big O Notation

↳ worst case comp (upper bound)

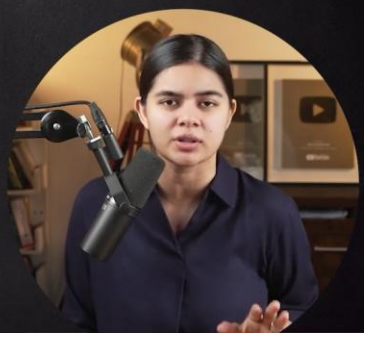
1. ignore constants
2. largest term

→ Cormen
Master's theorem

worst case
O
upper bound

avg.
Θ (theta)

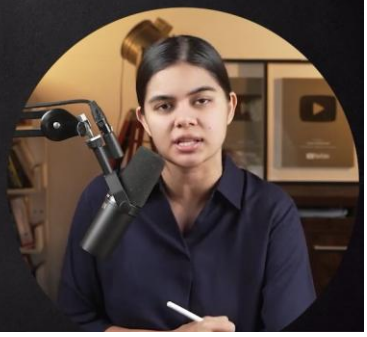
best
Ω (omega)
lower bound



Space Complexity

Amount of space taken by an algorithm as function of input size (n).

code → input (vec, arr, string ...)
code → auxiliary space



Space Complexity

SC = O(n)

Amount of space taken by an algorithm as function of input size (n).

inp → arr[ ]
out → squarednos[ ]

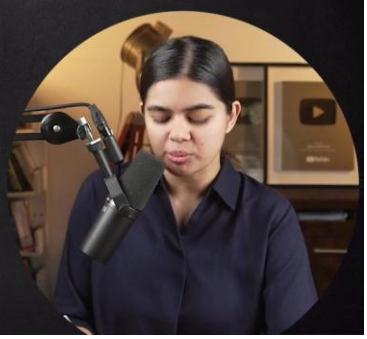
n → arr input

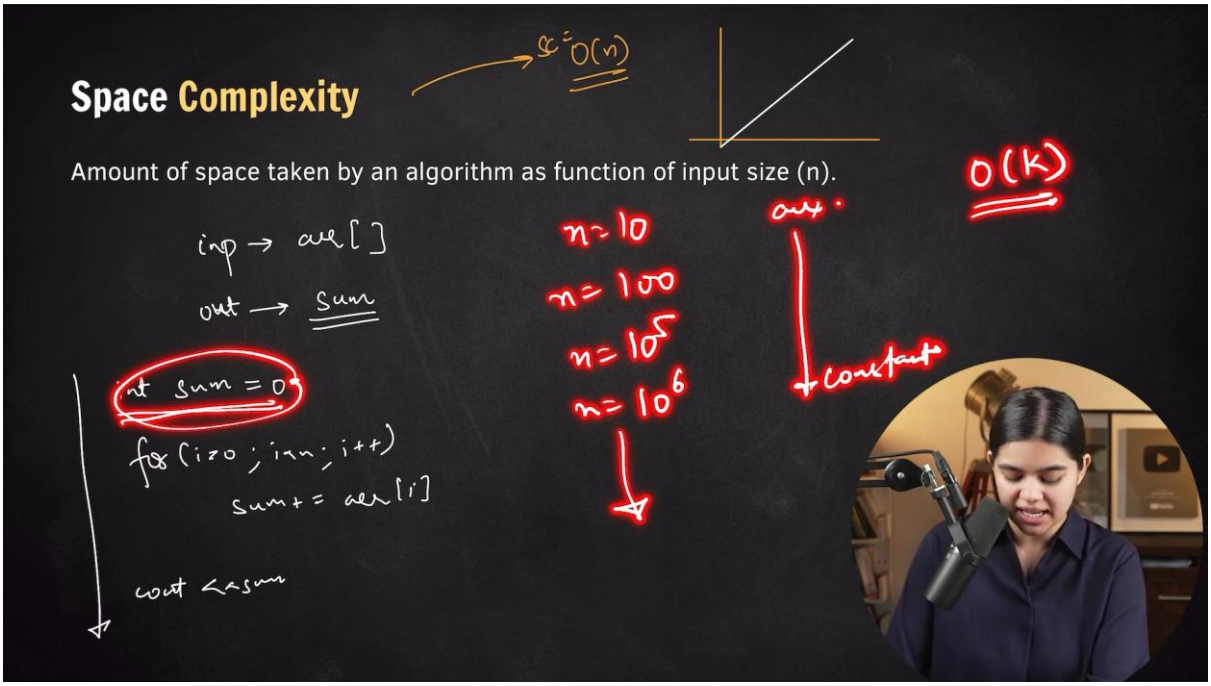
1 | 2 | 3 | 4 | 5 | 6 | 7

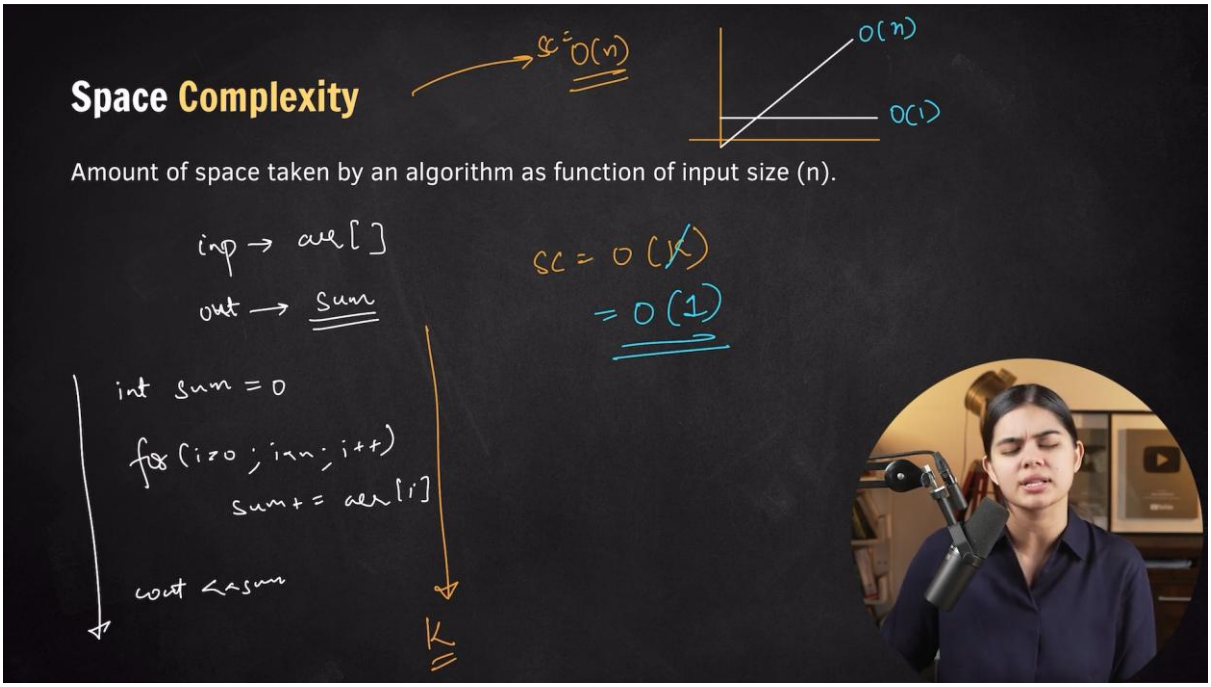
n

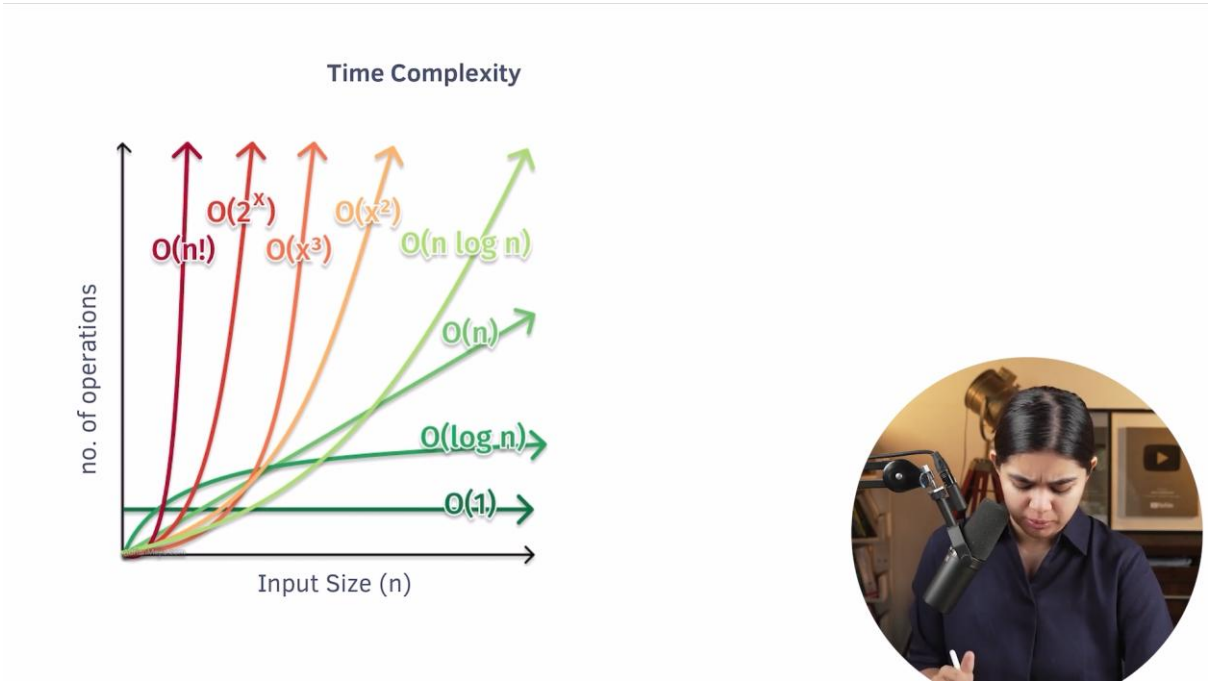
1 | 4 | 9 | 16 | 25

extra ✓




Space Complexity
SC = O(n)
Amount of space taken by an algorithm as function of input size (n).
O(K)
inp → arr[ ]
out → sum
n = 10
n = 100
n = 10^5
n = 10^6
out
constant
int sum = 0
for (i=0 ; i<n ; i++)
sum += arr[i]
cout << sum


Space Complexity
SC = O(n)
O(n)
O(1)
Amount of space taken by an algorithm as function of input size (n).
inp → arr[ ]
out → sum
SC = O(K)
= O(1)
int sum = 0
for (i=0 ; i<n ; i++)
sum += arr[i]
cout << sum
K


Time Complexity
O(2^x)
O(x^2)
O(n!)
O(x^3)
O(n log n)
O(n)
O(log n)
O(1)
no. of operations
Input Size (n)

Common Time Complexities

O(1) — constant

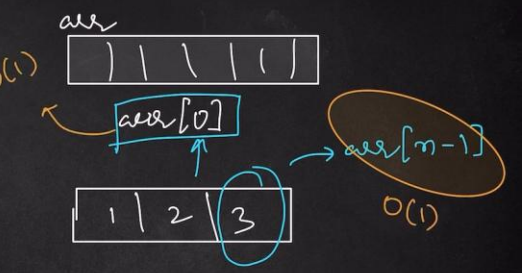
Sum of numbers from 1 to N

```
int n;
cin >> n;
int ans = n * (n+1)/2;
```

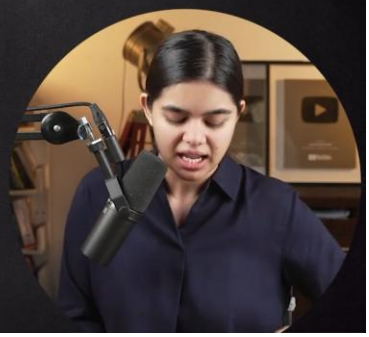
K ops

O(K)

O(1)



Sum = n*(n+1)/2



Common Time Complexities

O(1) — constant

Sum of numbers from 1 to N

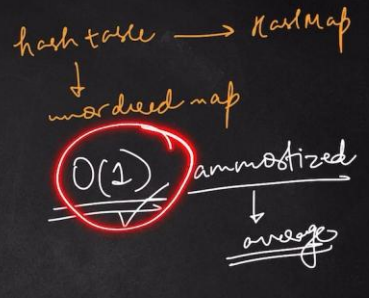
```
int n;
cin >> n;
int ans = n * (n+1)/2;
```

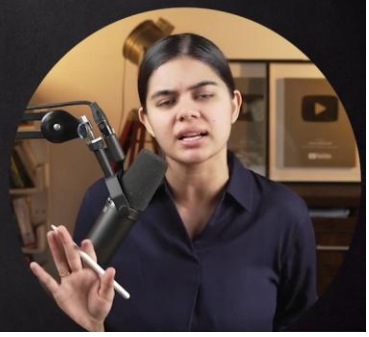
K ops

O(K)

O(1)

Sum = n*(n+1)/2





Common Time Complexities

O(1) — constant

Sum of numbers from 1 to N

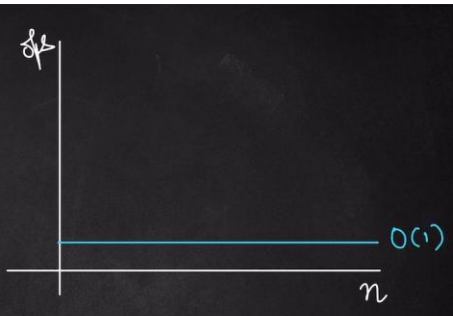
```
int n;
cin >> n;
int ans = n * (n+1)/2;
```

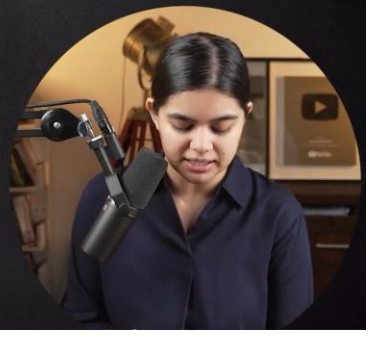
K ops

O(K)

O(1)

Sum = n*(n+1)/2





Common Time Complexities

O(n) → Linear

N Factorial

```
int fact = 1;
for(int i=1; i<=n; i++) {
    fact *= i;
}
```

Nth Fibonacci - DP

```
for(int i=2; i<=n; i++) {
    dp[i] = dp[i-1] + dp[i-2];
}
```

Kadane's Algorithm

```
int currSum = 0, ans = INT_MIN;

for(int i=0; i<n; i++) {
    currSum += arr[i];
    ans = max(currSum, ans);
    currSum = currSum < 0 ? 0 : currSum;
}
```

Common Time Complexities

O(n) → Linear

N Factorial

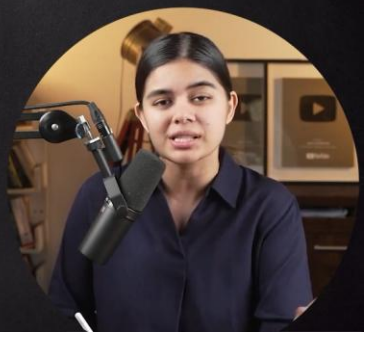
```
int fact = 1;
for(int i=1; i<=n; i++) {
    fact *= i;
}
```

Nth Fibonacci - DP

```
for(int i=2; i<=n; i++) {
    dp[i] = dp[i-1] + dp[i-2];
}
```

Kadane's Algorithm

```
int currSum = 0, ans = INT_MIN;

for(int i=0; i<n; i++) {
    currSum += arr[i];
    ans = max(currSum, ans);
    currSum = currSum < 0 ? 0 : currSum;
}
```

Common Time Complexities

O(n) → Linear

N Factorial

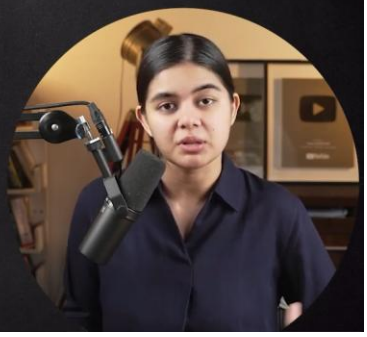
```
int fact = 1;
for(int i=1; i<=n; i++) {
    fact *= i;
}
```

Nth Fibonacci - DP

```
for(int i=2; i<=n; i++) {
    dp[i] = dp[i-1] + dp[i-2];
}
```

Kadane's Algorithm

```
int currSum = 0, ans = INT_MIN;

for(int i=0; i<n; i++) {
    currSum += arr[i];
    ans = max(currSum, ans);
    currSum = currSum < 0 ? 0 : currSum;
}
```

Common Time Complexities

O(n^2) & O(n^3)

selection, insertion
Bubble sort

```
for(int i=0; i<n-1; i++) {
    for(int j=0; j<n-i-1; j++) {
        if(arr[j] > arr[j+1]) {
            swap(arr[j], arr[j+1]);
        }
    }
}
```

K

n=4

i=0 j=0,1,2

i=1 j=0,1

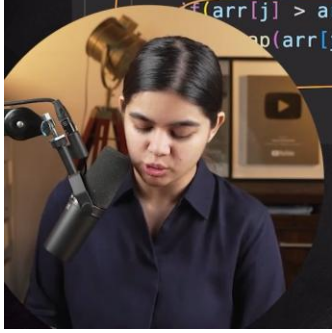
i=2 j=0

n

3*K + 2*K + 1*K

ops

(n-1)K + (n-2)K + (n-3)K + ... 2K + K



Common Time Complexities

O(n^2) & O(n^3)

selection, insertion
Bubble sort

```
for(int i=0; i<n-1; i++) {
    for(int j=0; j<n-i-1; j++) {
        if(arr[j] > arr[j+1]) {
            swap(arr[j], arr[j+1]);
        }
    }
}
```

K

(n-1)K + (n-2)K + (n-3)K + ... 2K + K

K[(n-1) + (n-2) + (n-3) + ... - 2 + 2]

operations

K * [n*(n-1) / 2]

O(Kn^2/2 - Kn/2) ⇒ O(n^2)

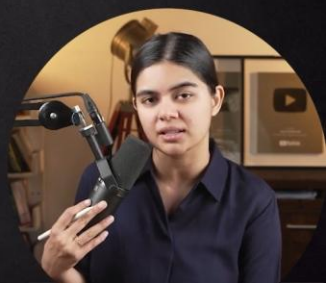
Common Time Complexities

patterns → nested loops O(n^2)

O(n^2) & O(n^3)

selection, insertion
Bubble sort

```
for(int i=0; i<n-1; i++) {
    for(int j=0; j<n-i-1; j++) {
        if(arr[j] > arr[j+1]) {
            swap(arr[j], arr[j+1]);
        }
    }
}
```

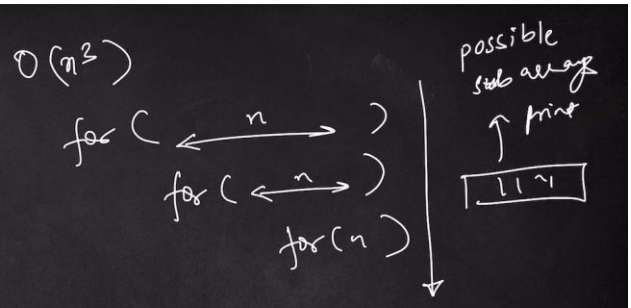


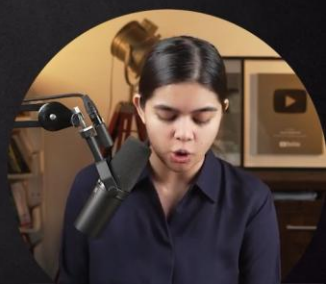
Common Time Complexities

O(n^2) & O(n^3)

selection, insertion
Bubble sort

```
for(int i=0; i<n-1; i++) {
    for(int j=0; j<n-i-1; j++) {
        if(arr[j] > arr[j+1]) {
            swap(arr[j], arr[j+1]);
        }
    }
}
```



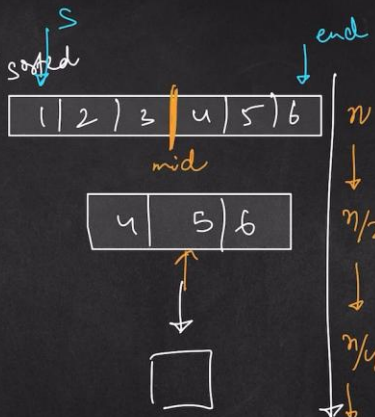


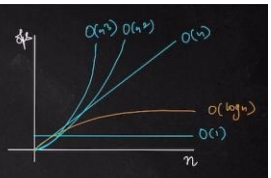
Common Time Complexities

O(logn)

Binary Search

```
int s=0, e=n-1;
while(s <= e) {
    int mid = s + (e-s)/2;
    if(arr[mid] < target) {
        s = mid + 1;
    } else if(arr[mid] > target) {
        e = mid - 1;
    } else {
        return mid; //ans
    }
}
```



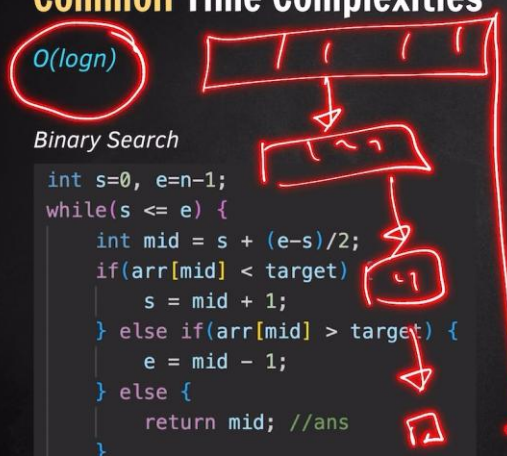


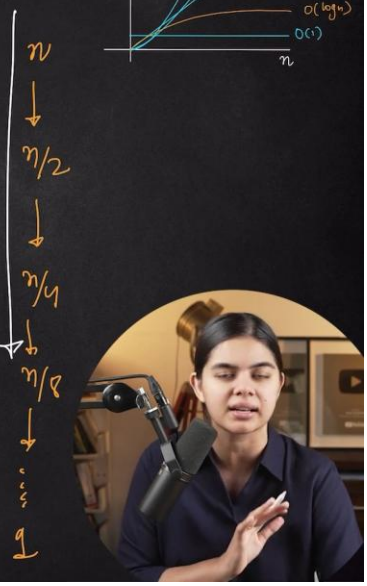
Common Time Complexities

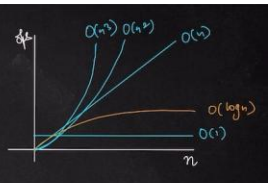
O(logn)

Binary Search

```
int s=0, e=n-1;
while(s <= e) {
    int mid = s + (e-s)/2;
    if(arr[mid] < target)
        s = mid + 1;
    } else if(arr[mid] > target) {
        e = mid - 1;
    } else {
        return mid; //ans
    }
}
```







Common Time Complexities

O(nlogn)

↳ sorting

MergeSort

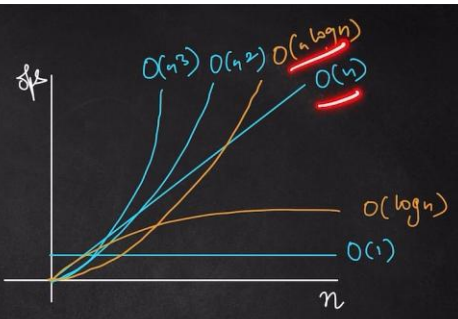
QuickSort (avg)

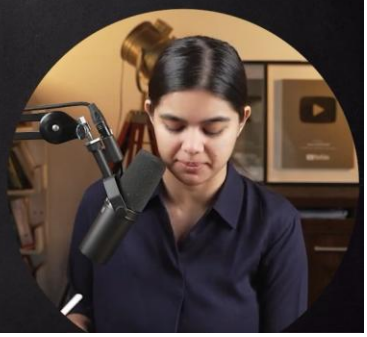
Greedy algos ✓

O(logn) ← better — O(n)

xn xn

O(nlogn) ← Better — O(n²)

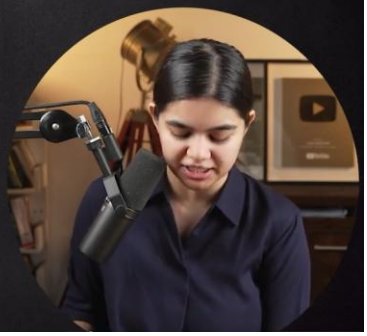




Common Time Complexities

O(2^n) → exponential O(3^n) O(4^n)

↳ Recursion (Brute Force)



Common Time Complexities

O(nlogn)

↳ sorting

MergeSort

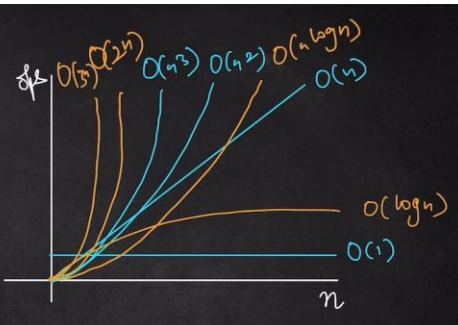
QuickSort (avg)

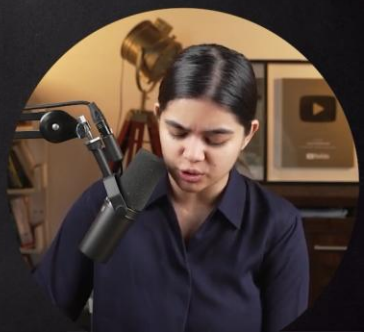
Greedy algos ✓

O(logn) ← better — O(n)

xn xn

O(nlogn) ← Better — O(n²)





Common Time Complexities

O(n!) //not so common

$2^n = 2 \times 2 \times 2 \times 2 \cdots \times 2$ (n times)

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

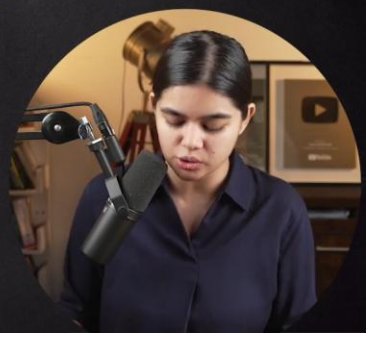
- n Queens ✓
- Knights Tour ✓
- String → all possible permutation ✓ O(n!) — rec

"abc" n=3

abc
acb
bac
bca
cab
cba

6 = 3! = n!

O(n!)



Common Time Complexities

O(n!) //not so common

$2^n = 2 \times 2 \times 2 \times 2 \cdots \times 2$ (n times)

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

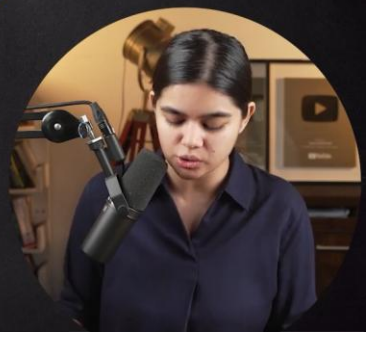
- n Queens ✓
- Knights Tour ✓
- String → all possible permutation ✓ O(n!) — rec

"abc" n=3

abc
acb
bac
bca
cab
cba

6 = 3! = n!

O(n!)



Solve

Prime Number - Time Complexity

O(√n)

```
for(int i=2; i*i<=n; i++) {
    if(n % i == 0) {
        cout << "Non prime";
        break;
    }
}
```

K

$i = 2$ to $i^2 \le n$

$i^2 = n$

$i = \sqrt{n}$

$i = 2$ to $\sqrt{n}$

0 to $\sqrt{n}$

O(√n)

① O(√n) O(n)

② O(√n) O(log n)

Solution

① $O(\sqrt{n})$ $O(n)$

② $O(\sqrt{n})$ $O(\log n)$

Solve

$O(n^2)$

Selection Sort - Time Complexity

```
for(int i=0; i<n-1; i++) {
    int minIdx = i;
    for(int j=i+1; j<n; j++) {
        if(arr[j] < arr[minIdx]) {
            minIdx = j;
        }
    }
    swap(arr[i], arr[minIdx]);
}
```

K

outer loop ⇒ n times

i=0

i=1

i=2

j = 1 to n → n

j = 2 to n → n-1

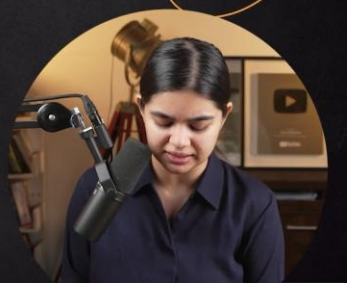
j = 3 to n → n-2

⋮

1

$n + (n-1) + (n-2) + (n-3) \cdots + 1$

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



Recursion

Time Complexity

$f(n) = K*n + K_2$

$TC = O(n)$

TC

① $f(n) = K + f(n-1)$

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

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

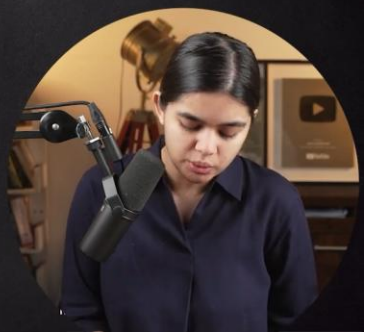
⋮ = ⋱ ⋮

$f(1) = K + f(0)$ K_2

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

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

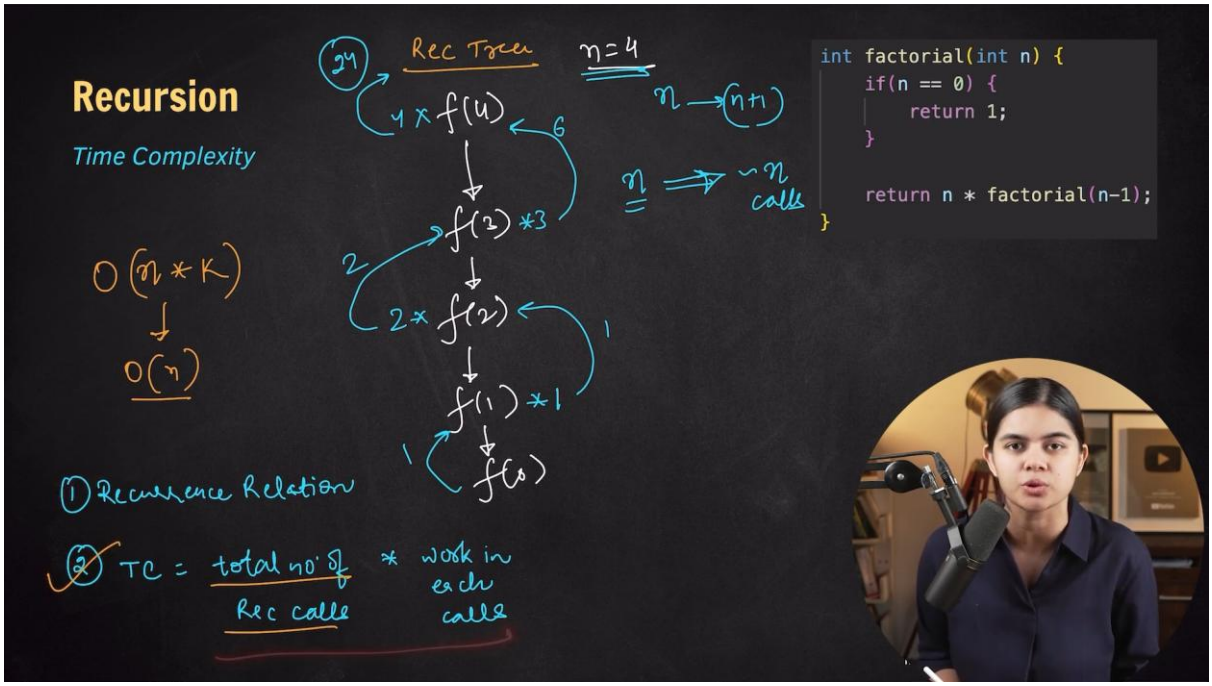
① Recurrence Relation

② TC = total no of Rec calls * work in each calls



Recursion

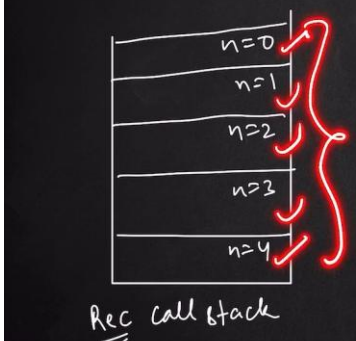
Time Complexity



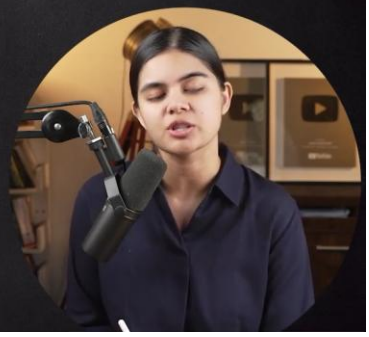

```
int factorial(int n) {
    if(n == 0) {
        return 1;
    }

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


Recursion

Space Complexity

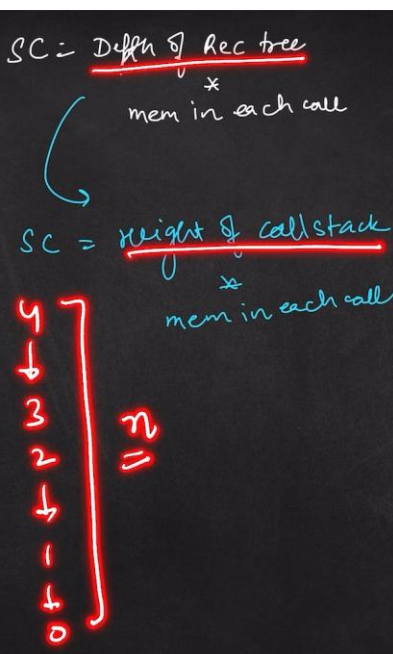



```
int factorial(int n) {
    if(n == 0) {
        return 1;
    }

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

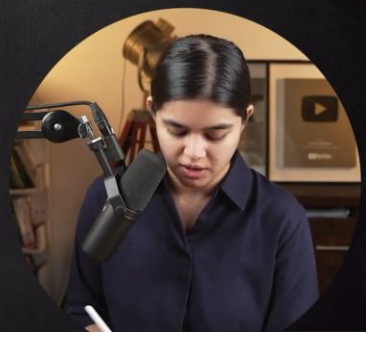


Recursion

Space Complexity




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

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



Recursion

Space Complexity

SC = Depth of Rec tree * mem in each call

SC = height of callstack * mem in each call

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

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

Rec call stack

$SC = n * K$

$SC = O(n)$ ✓

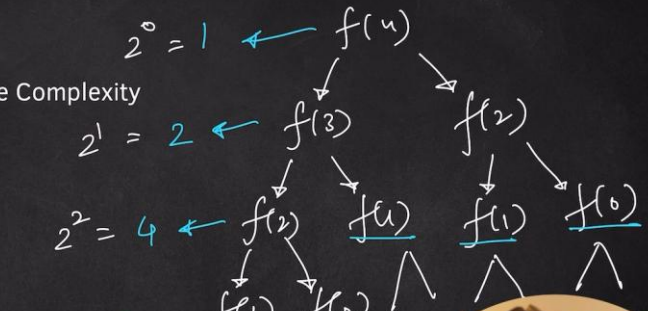
Solve

Recursive Fibonacci - Time & Space Complexity

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

    return fib(n-1) + fib(n-2);
}
```

Sum = $a\left(\frac{r^n - 1}{r-1}\right)$

TC ⇒ n = 4 n



TC = total calls * WD in each

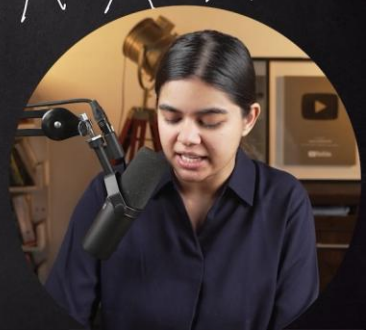
GP

$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{n-1}$

$= 2^0 * \frac{(2^n - 1)}{2-1}$

$\frac{x * 2^n - 1}{x}$

$(2^n - 1)$

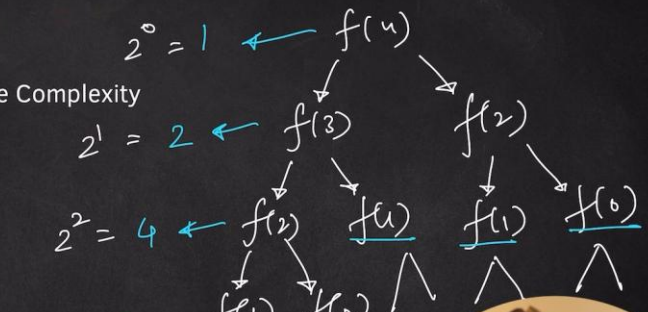


Solve

Recursive Fibonacci - Time & Space Complexity

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

    return fib(n-1) + fib(n-2);
}
```

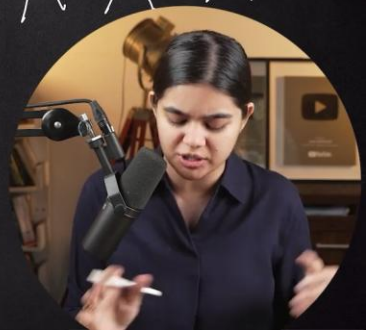
TC ⇒ n = 4 n



TC = total calls * WD in each

$(2^n - 1)$

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



Solve

$O((1.618)^n)$ GR

Recursive Fibonacci - Time & Space Complexity

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

    return fib(n-1) + fib(n-2);
}
```

TC ⇒ n = 4 n

$2^0 = 1$ ← f(4)

$2^1 = 2$ ← f(3) f(2)

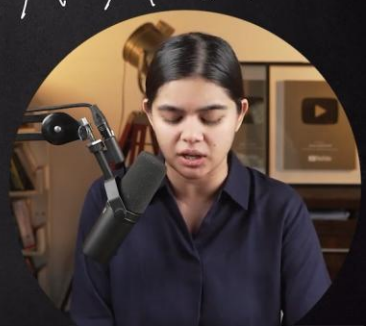
$2^2 = 4$ ← f(2) f(1) f(1) f(0)

$2^3 = 8$ ← f(1) f(0)

$(2^n - 1)$

TC = total calls * WD in each

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



Solve

Recursive Fibonacci - Time & Space Complexity

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

    return fib(n-1) + fib(n-2);
}
```

TC ⇒ n = 4 n

$2^0 = 1$ ← f(4)

$2^1 = 2$ ← f(3) f(2)

$2^2 = 4$ ← f(2) f(1) f(1) f(0)

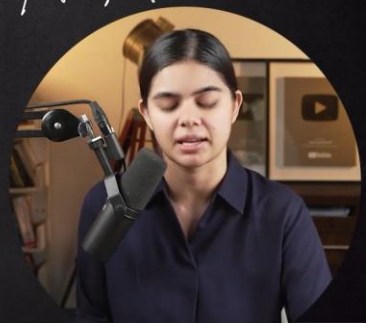
$2^3 = 8$ ← f(1) f(0)

$(2^n - 1)$

SC = Depth of RT * mem

$= n * 1$

SC = $O(n)$ ✓

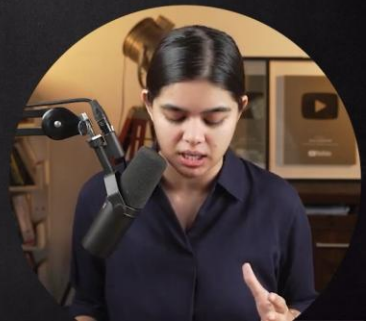


Solve

Rec. Relⁿ for TC ⟶ $T(n) = T(n-1) + T(n-2) + O(1)$

Recursive Fibonacci - Time & Space Complexity

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

    return fib(n-1) + fib(n-2);
}
```

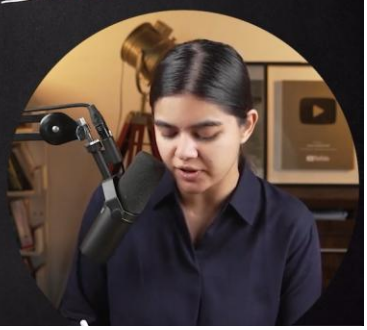


Merge Sort - Time & Space Complexity

```
void merge(int arr[], int si, int mid, int ei) {
    vector<int> temp;
    int i=si, j=mid+1;

    while(i <= mid && j <= ei) {
        if(arr[i] <= arr[j]) {
            temp.push_back(arr[i++]);
        } else {
            temp.push_back(arr[j++]);
        }
    }

    while(i <= mid) {
        temp.push_back(arr[i++]);
    }

    while(j <= ei) {
        temp.push_back(arr[j++]);
    }

    //copy back to original
    for(int idx=si, x=0; idx<=ei; idx++) {
        arr[idx] = temp[x++];
    }
}
```

```
void mergeSort(int arr[], int si, int ei) {
    if(si >= ei) {
        return;
    }

    int mid = si + (ei - si)/2;
    mergeSort(arr, si, mid);
    mergeSort(arr, mid+1, ei);

    merge(arr, si, mid, ei);
}
```

TC = total no of calls * WD each call



Merge Step - Time & Space Complexity

O(n)

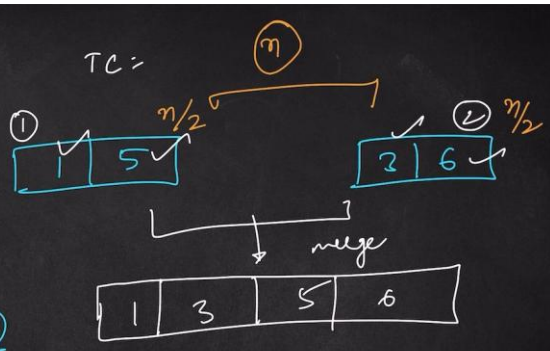
```
void merge(int arr[], int si, int mid, int ei) {
    vector<int> temp;
    int i=si, j=mid+1;

    while(i <= mid && j <= ei) {
        if(arr[i] <= arr[j]) {
            temp.push_back(arr[i++]);
        } else {
            temp.push_back(arr[j++]);
        }
    }
```

① ② ③ O(n)

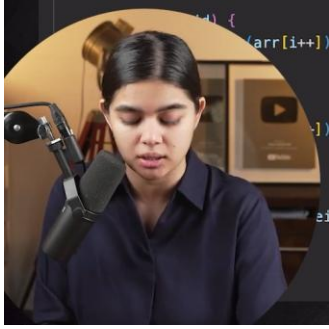
```
    while(i <= mid) {
        temp.push_back(arr[i++]);
    }

    while(j <= ei) {
        temp.push_back(arr[j++]);
    }

    //copy back to original
    for(int idx=si, x=0; idx<=ei; idx++) {
        arr[idx] = temp[x++];
    }
}
```

④ O(n)

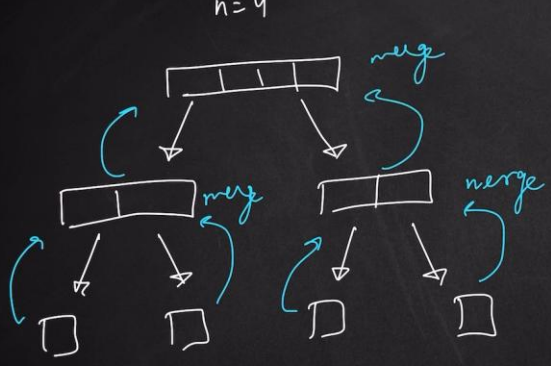
TC :



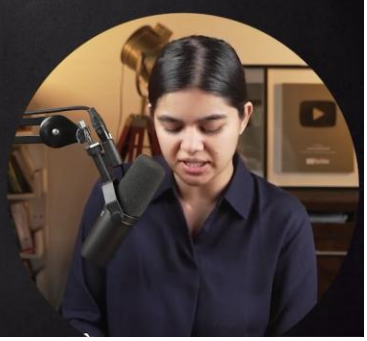
$= O(n+n) = O(2n) \Rightarrow O(n)$



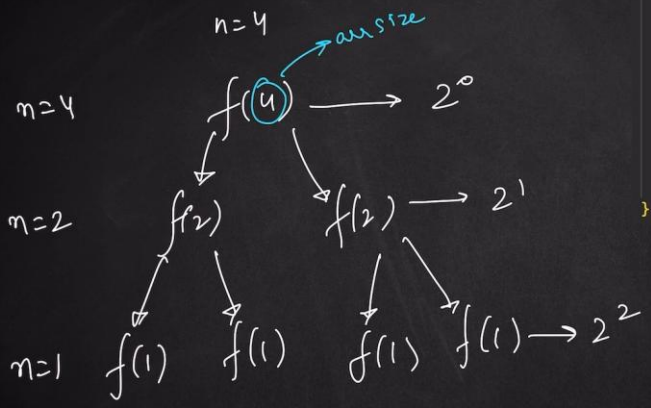
Merge Sort - Time & Space Complexity



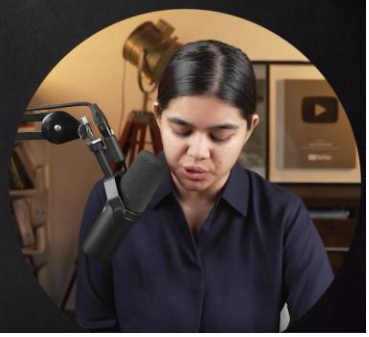

```
void mergeSort(int arr[], int si, int ei) {
    if(si >= ei) {
        return;
    }

    int mid = si + (ei - si)/2;
    mergeSort(arr, si, mid);
    mergeSort(arr, mid+1, ei);

    merge(arr, si, mid, ei); // O(n) O(n)
}
```



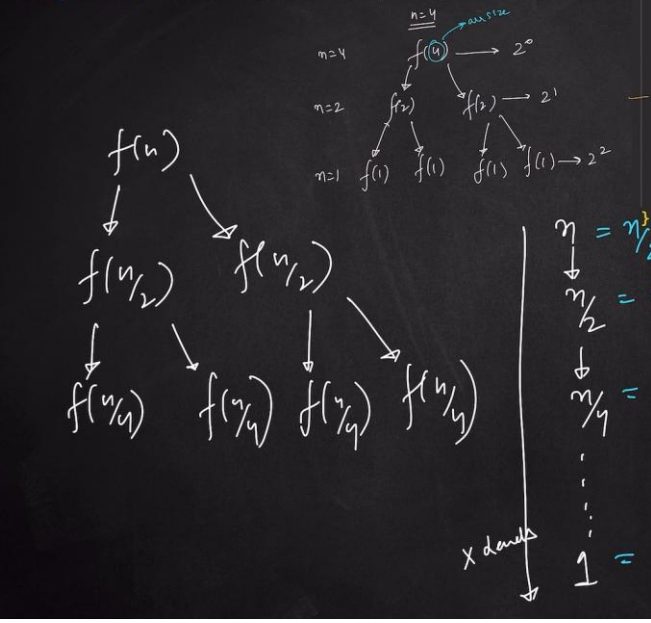
Merge Sort - Time & Space Complexity



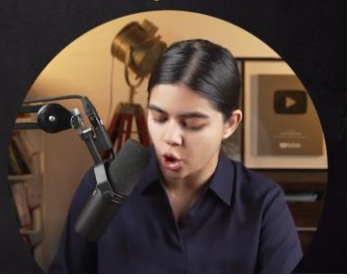

```
void mergeSort(int arr[], int si, int ei) {
    if(si >= ei) {
        return;
    }

    int mid = si + (ei - si)/2;
    mergeSort(arr, si, mid);
    mergeSort(arr, mid+1, ei);

    merge(arr, si, mid, ei); // O(n) O(n)
}
```



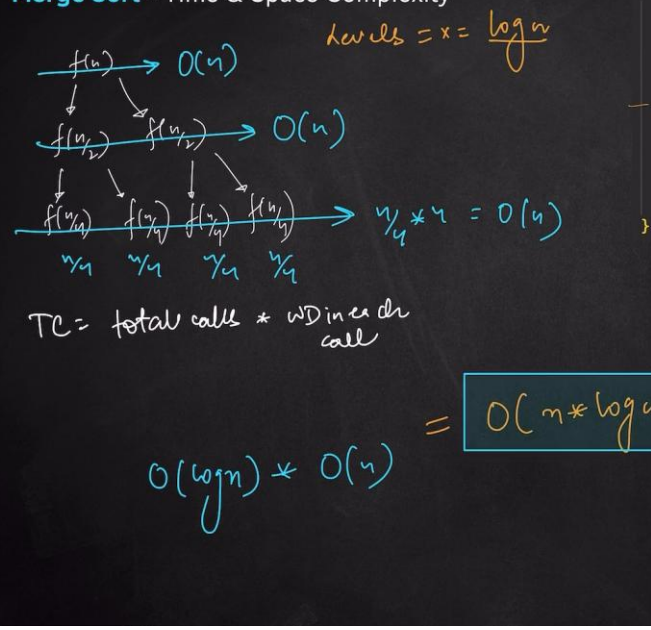
Merge Sort - Time & Space Complexity



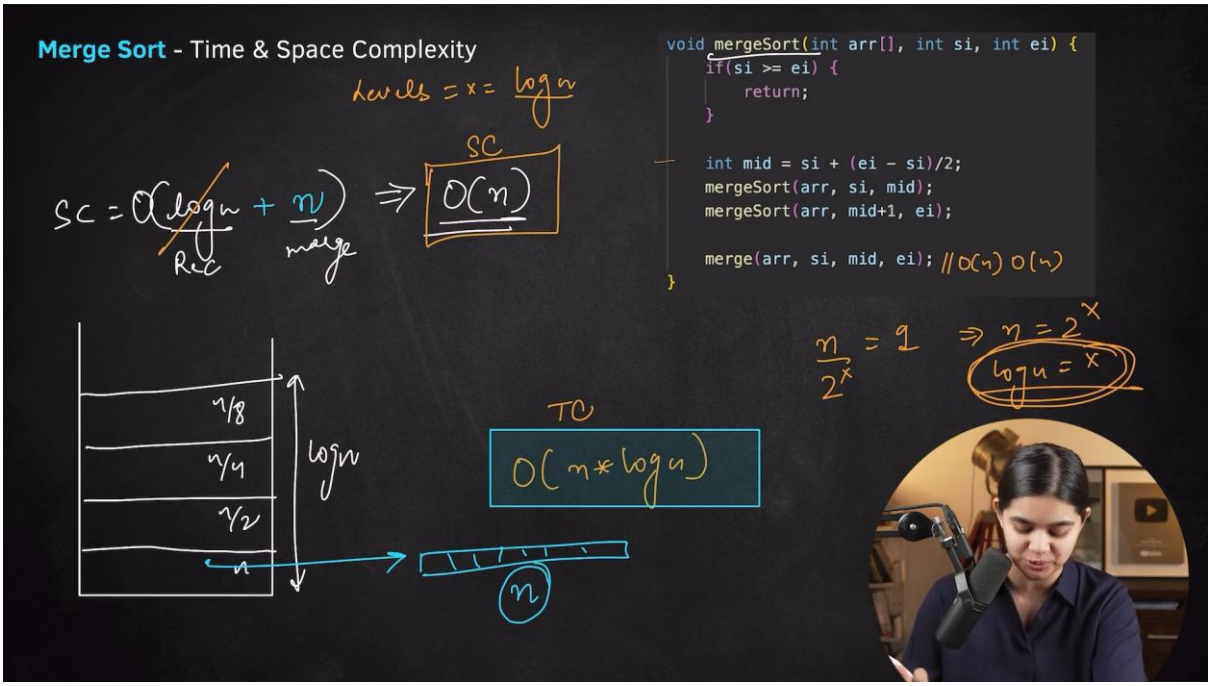

```
void mergeSort(int arr[], int si, int ei) {
    if(si >= ei) {
        return;
    }

    int mid = si + (ei - si)/2;
    mergeSort(arr, si, mid);
    mergeSort(arr, mid+1, ei);

    merge(arr, si, mid, ei); // O(n) O(n)
}
```

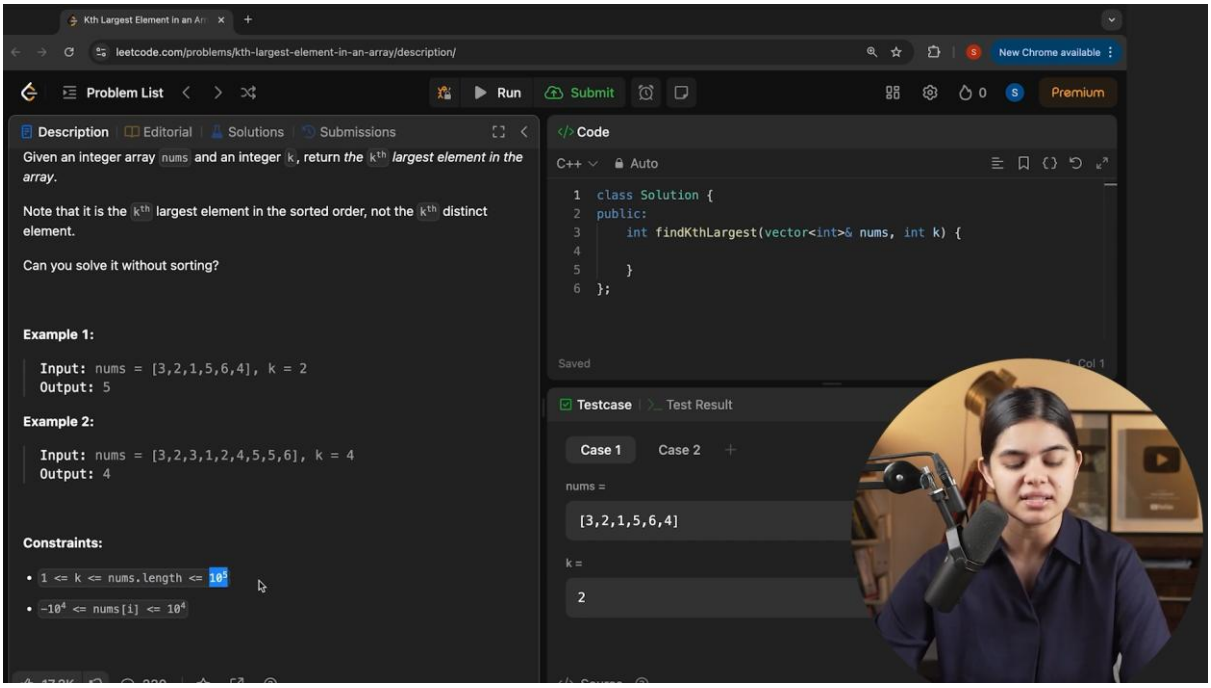


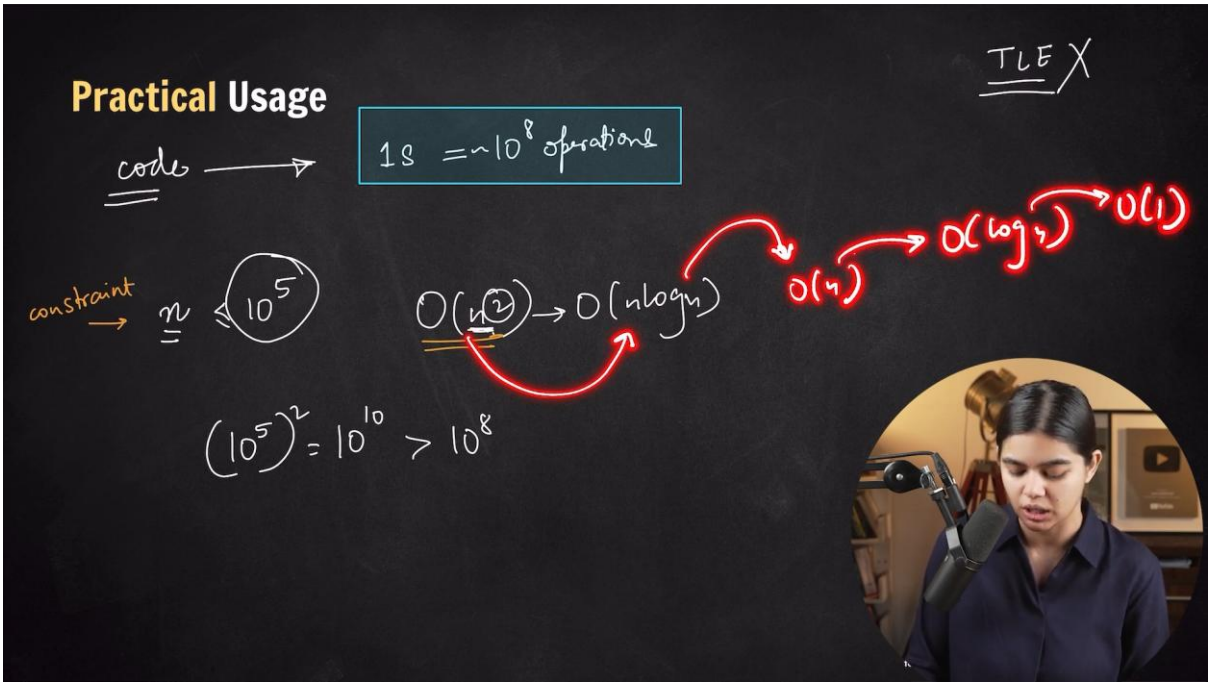
Merge Sort - Time & Space Complexity




```
void mergeSort(int arr[], int si, int ei) {
    if(si >= ei) {
        return;
    }

    int mid = si + (ei - si)/2;
    mergeSort(arr, si, mid);
    mergeSort(arr, mid+1, ei);

    merge(arr, si, mid, ei); // O(n) O(n)
}
```


Merge Sort - Time & Space Complexity
levels = x = log n
SC = O(log n + n)
Rec
merge
SC
O(n)
void mergeSort(int arr[], int si, int ei) {
if(si >= ei) {
return;
}
int mid = si + (ei - si)/2;
mergeSort(arr, si, mid);
mergeSort(arr, mid+1, ei);
merge(arr, si, mid, ei); // O(n) O(n)
}
n/8
n/4
n/2
n
log n
TC
O(n * log n)
n = 2^x
log n = x


leetcode.com/problems/kth-largest-element-in-an-array/description/
Problem List
Run
Submit
Premium
Description
Editorial
Solutions
Submissions
Given an integer array nums and an integer k, return the kth largest element in the array.
Note that it is the kth largest element in the sorted order, not the kth distinct element.
Can you solve it without sorting?
Example 1:
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
Example 2:
Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4
Constraints:
1 <= k <= nums.length <= 10^5
-10^4 <= nums[i] <= 10^4
Code
C++
Auto
class Solution {
public:
int findKthLargest(vector<int>& nums, int k) {
}
};
Saved
Testcase
Test Result
Case 1
Case 2
nums =
[3,2,1,5,6,4]
k =
2


Practical Usage
TLE X
code
1s = ~10^8 operations
constraint
n <= 10^5
O(n^2) → O(n log n)
O(n)
O(log n)
O(1)
(10^5)^2 = 10^10 > 10^8

Practical Usage

code → 1s = ~10^8 operations

TLE ✗

$\log_2 10 = 0.3$

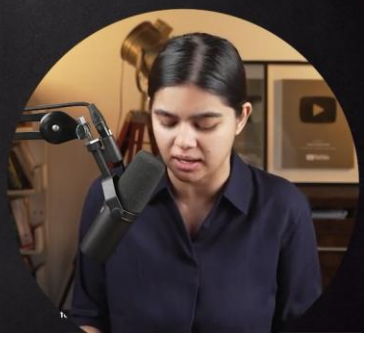
constraint → $n \leq 10^5$

$O(n^3)$ ✗ → $O(n)$ ✓ → $O(n \log n)$ → $\log n$, n, 1

$10^5 * \log 10^5$

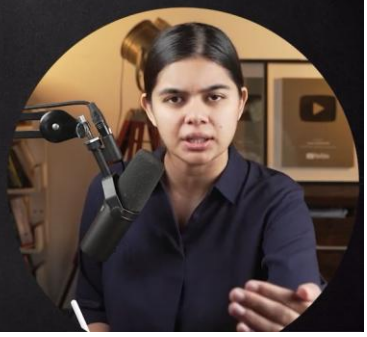
$= 10^5 * 5 \log 10$

$10^5 * 5 * 0.3 < 10^8$



Practical Usage

1)	$n > 10^8$	$O(\log n)$ $O(1)$
2)	$n \leq 10^8$	$O(n)$
3)	$n \leq 10^6$	$O(n \log n)$ → sorting
4)	$n \leq 10^4$	$O(n^2)$
5)	$n \leq 500$	$O(n^3)$
6)	$n \leq 25$	$O(2^n)$ → Recursion, Brute Force
7)	$n \leq 12$	$O(n!)$ → "



- Time Comp
- Space Comp
- Common TC → $O(1)$, $\log n$, $n \log n$, n, n^2, n^3, 2^n, $n!$
- graphs
- Recursion TC, SC
- Sorted ✓
- Practical Usage