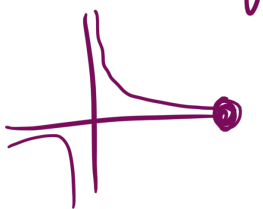
Time & Space Complexity

Topics in Today's class

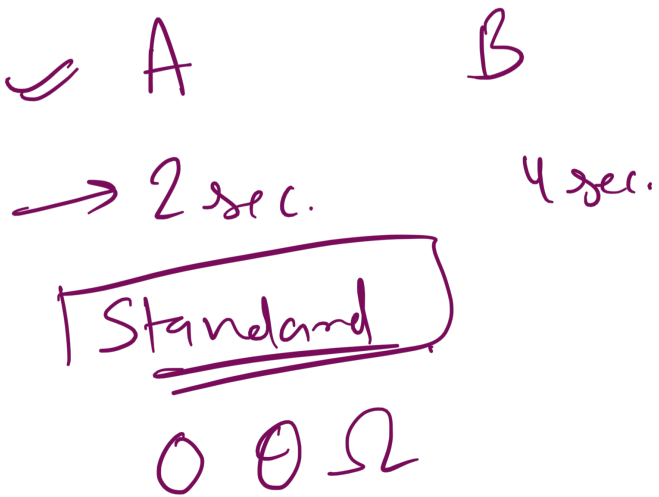
What is Time Complexity

Asymptotic Notations

- Big O notations
- Omega Notation
- Theta Notations



Time Complexity & Constraints

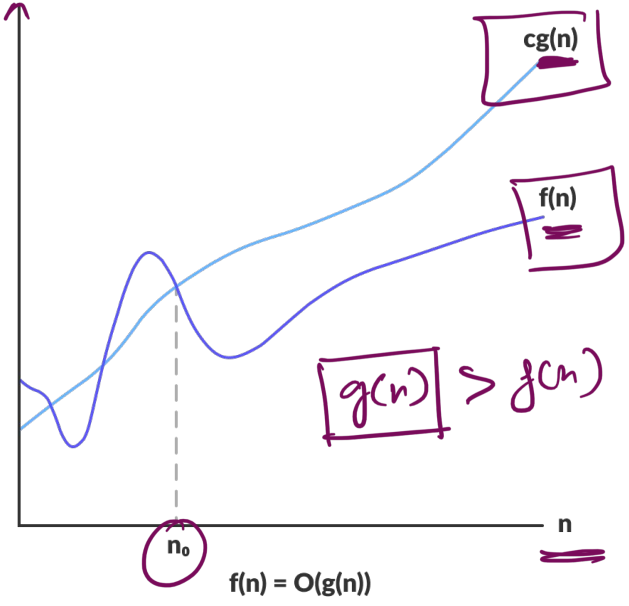


Big O Notation

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.



$O(n^2)$ ← (1) $f(n) = an^2 + bn + cn$

$O(n)$ ← (2) $f(n) = \frac{an'}{4} + bn'$

$O(1)$ ← (3) $f(n) = 100n^0$

Omega (Ω) Notation

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

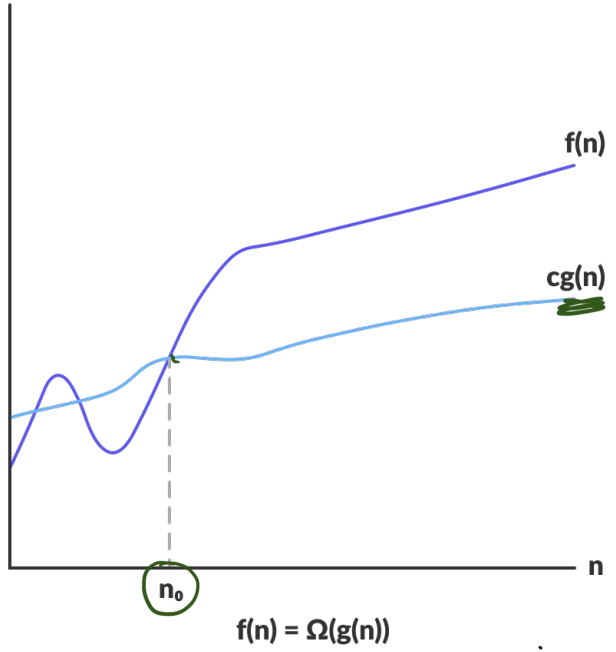
For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

best.

$\Omega(n)$

a[]

key.



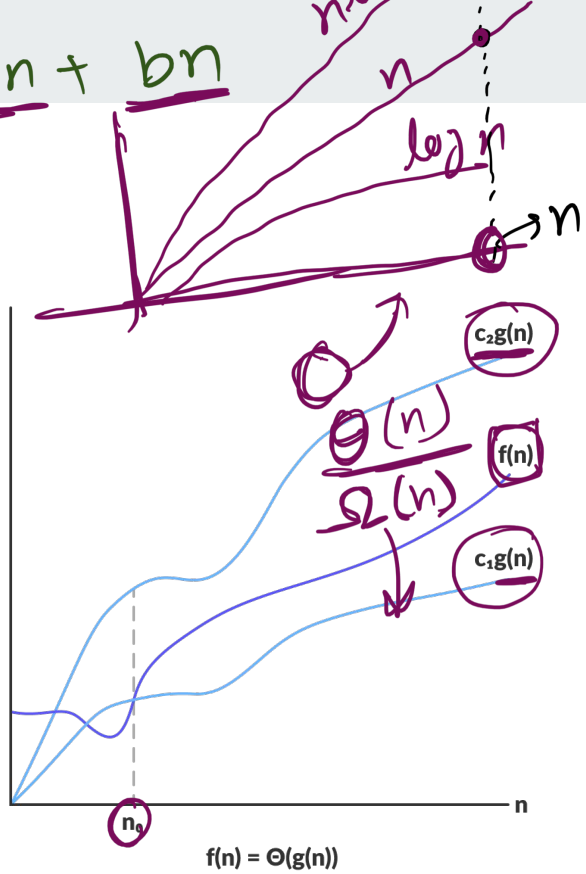
$an^2 + bn'$

n^n

Theta (Θ) Notation

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

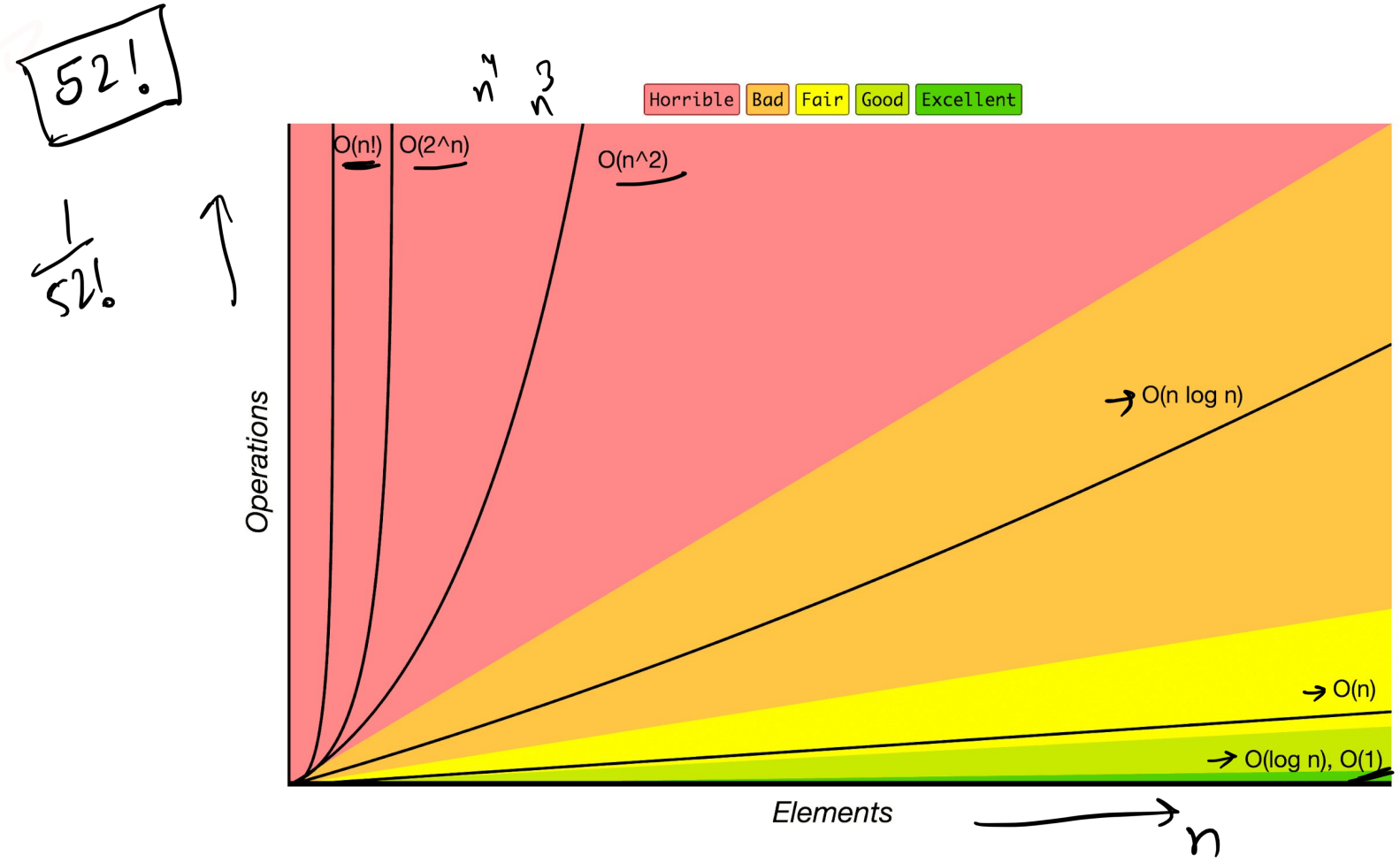


Insertion Sort Example

Each of the following statements are true:

- Insertion Sort's worst case rate of growth is at most O(n^2)
- Insertion Sort's worst case rate of growth is at least Ω(n)
- Insertion Sort's worst case rate of growth is exactly Θ(n^2)

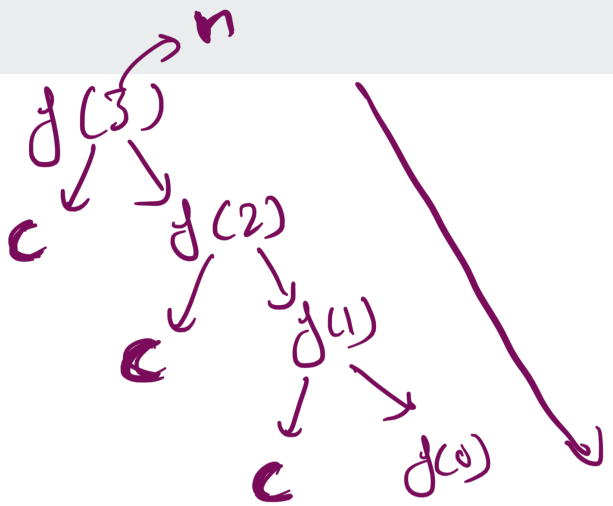
```
const insertionSort = arr => {
  for (let i = 1; i < arr.length; i++) {
    let index = i;
    while (index > 0 && arr[index - 1] > arr[index]) {
      let temp = arr[index];
      arr[index] = arr[index - 1];
      arr[index - 1] = temp;
      index = index - 1;
    }
  }
  return arr;
}
```


52!
1
52!
n^4
n^3
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements
n

Recurrence Problems

1. $T(n) = T(n-1) + c$
2. $T(n) = 2T(n-1) + c$
3. $T(n) = 2T(n/2) + nc$

```
f(int n) {
  c [
    f(n-1);
  ]
```



operation = nc

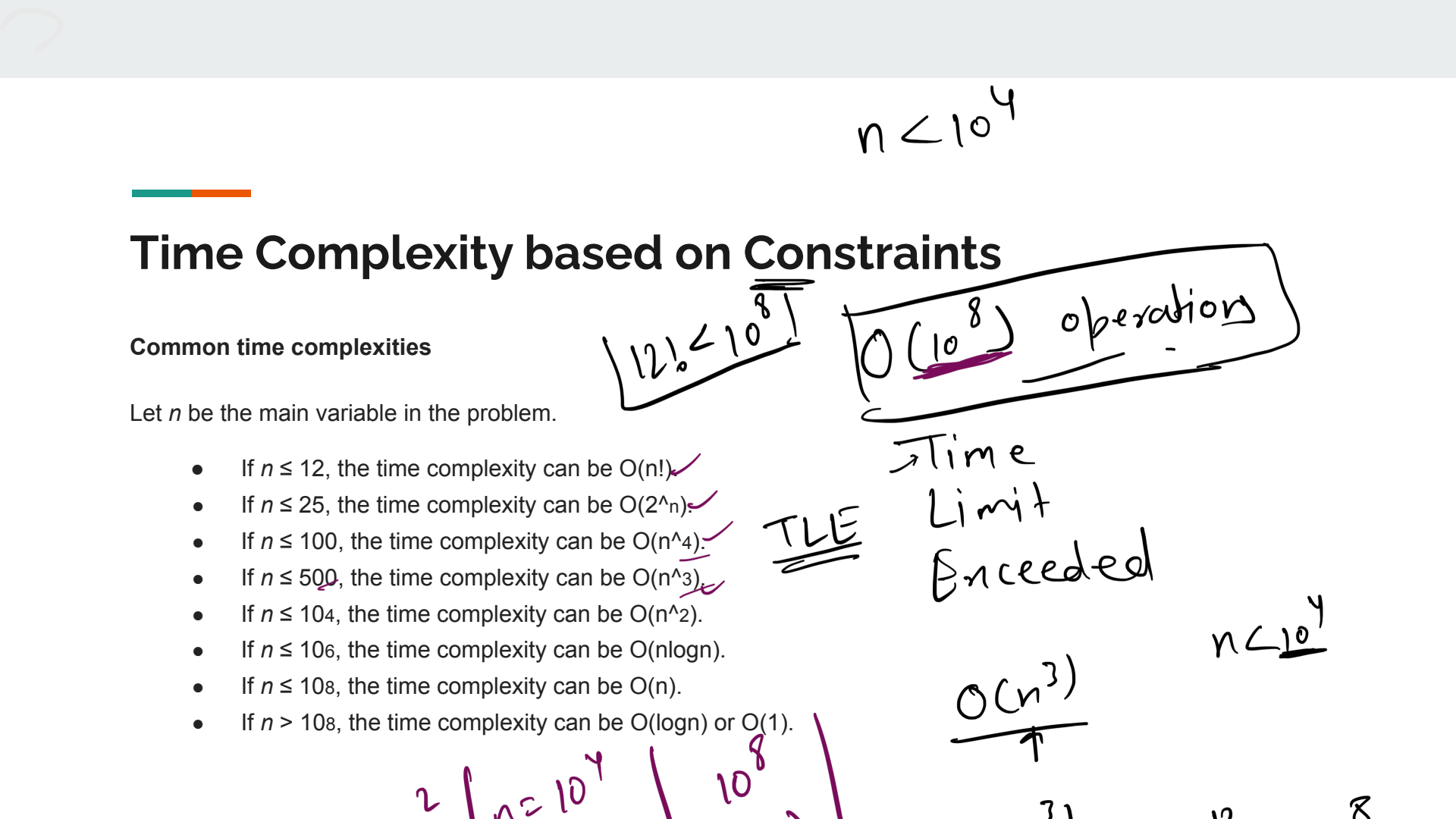
$O(n)$

Time Complexity based on Constraints

Common time complexities

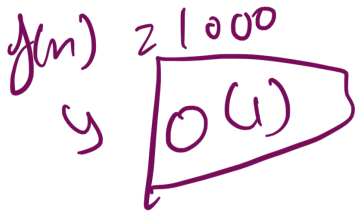
Let *n* be the main variable in the problem.

- If *n* ≤ 12, the time complexity can be O(n!).
- If *n* ≤ 25, the time complexity can be O(2^n).
- If *n* ≤ 100, the time complexity can be O(n^4).
- If *n* ≤ 500, the time complexity can be O(n^3).
- If *n* ≤ 10^4, the time complexity can be O(n^2).
- If *n* ≤ 10^6, the time complexity can be O(nlogn).
- If *n* ≤ 10^8, the time complexity can be O(n).
- If *n* > 10^8, the time complexity can be O(logn) or O(1).



Examples of each common time complexity

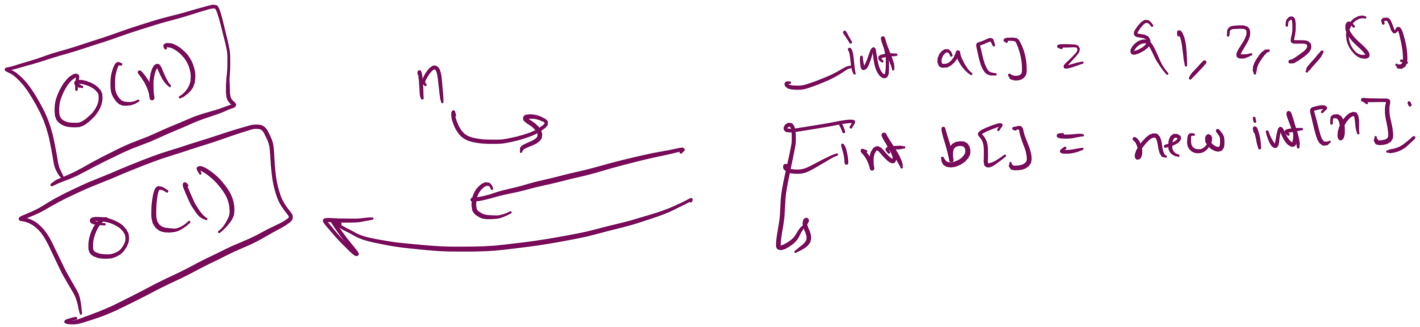
- O(n!) [Factorial time]: Permutations of 1 ... *n*
- O(2^n) [Exponential time]: Exhaust all subsets of an array of size *n*
- O(n^3) [Cubic time]: Exhaust all triangles with side length less than *n*
- O(n^2) [Quadratic time]: Slow comparison-based sorting (eg. Bubble Sort, Insertion Sort, Selection Sort)
- O(nlog n) [Linearithmic time]: Fast comparison-based sorting (eg. Merge Sort)
- O(n) [Linear time]: Linear Search (Finding maximum/minimum element in a 1D array), Counting Sort
- O(log n) [Logarithmic time]: Binary Search, finding GCD (Greatest Common Divisor) using Euclidean Algorithm
- O(1) [Constant time]: Calculation (eg. Solving linear equations in one unknown)



Space Complexity

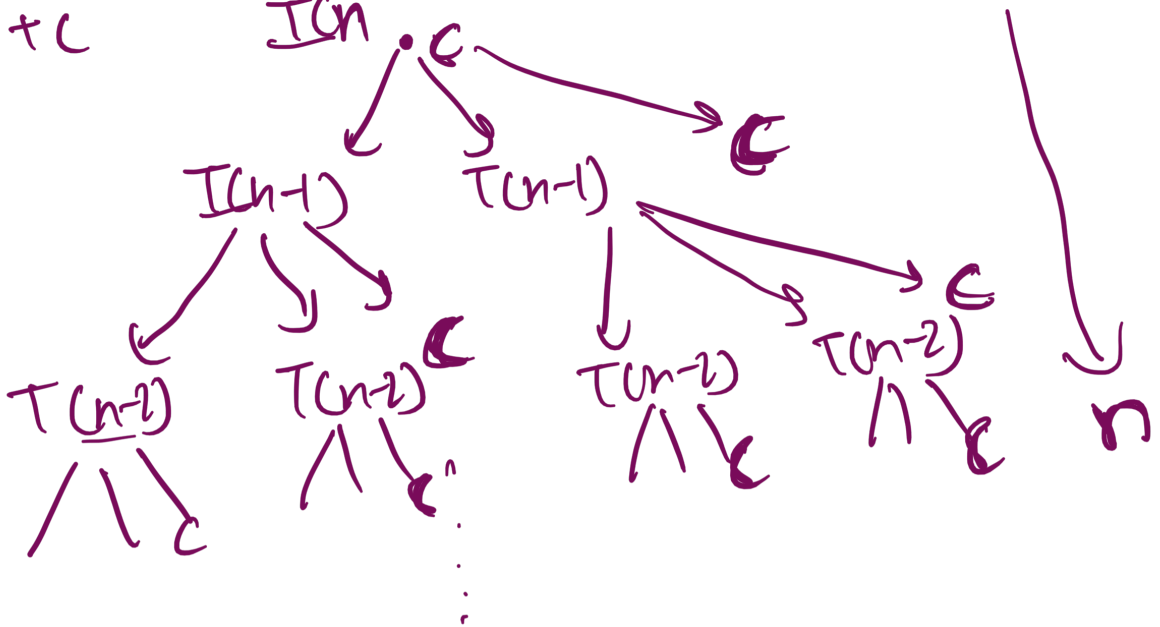
Space complexity refers to the total amount of memory space used by an algorithm/program, including the space of input values for execution.

Calculate the space occupied by variables in an algorithm/program to determine space complexity.



Q → $T(n) = 2T(n-1) + c$

```
f(int n) {
  c[
    f(n-1)
    f(n-1)
  ]
}
```



$T(1) \longrightarrow n$

$c + 2c + 4c + 8c + \ldots$ G.P.

$$\text{Sum} \Rightarrow \frac{a(r^n - 1)}{(r-1)} = \frac{c(2^n - 1)}{1} \Rightarrow c2^n - c$$

$$O(2^n)$$

```
int a[] = new int[n];
int b[] = new int[n];
:
int d[][] = new int[n][n]
```

$5n$ → $O(n)$

$O(n^2)$

$5n + n^2$ → $O(n^2)$

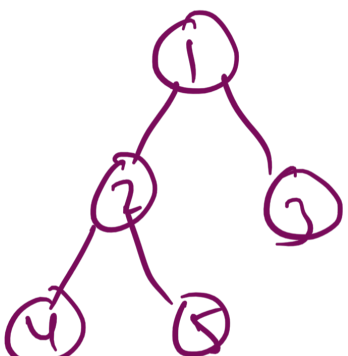
n

int temp = 23

int temp2 = 41

⋮

50,000 → O(1)



→ O(n)