Time and Space Complexity.

What is time and space complexity?

→ Time and space complexity are two fundamentals concepts we use to analyse the efficiency of an algorithm.

↳ efficiency is how much space and time a program took to run a specific task.

What is Time Complexity?

Time complexity tells us the growth in runtime of algorithm as the input increases.

In simple words.

↳ The time required by the algorithm to solve given problem is called time complexity of the algorithm.

It tells us how the number of operation increases as the size of input increases.

↳ we measure Time Complexity by counting its operations in a program.

It is measured in **Big-O Notation**.

more simple definition

↳ time complexity ~~is a way to~~ tells us how the time to run an algorithm increases with the size of the ~~ip~~ input data.

The three Notations.

Notations are used to describe how an algorithm performs as the input increases.

There are three types of Notation.

① Big O (O) → worst case performance.

② Big Theta (Θ) → Avrage or typical performance.

③ Big Omega (Ω) → Best-case performance.

① Big O Notation (O)

→ It tells us the maximum time or ~~it~~ memory our code might use.

This is also called as the worst case ~~(worst~~ Scenario. and we typically takes ~~it~~ use this.

$O(1)$, $O(N)$, $O(N^2)$

→ upper Bound.

② Big Theta Notation (Θ)

→ It shows the typical time or memory used — not too fast or not too slow.

→ It tells us how the algorithm performs on average.

③ Big Omega Notation (Ω)

→ It shows the minimum time or memory your code might use.

↳ It tells how fast the algorithm can be in the best situation, lower bound.

Now, in depth of Big O (O) Notation

① **O(1)** → Constant

↳ The execution time is constant regardless of input size.

Example →

```
int a = arr[5];
```

→ O(1)

② **O(n)** → Linear

↳ The time grows directly with input size.

Example →

```
for (int i = 0; i < n; i++) {
    System.out.println(i);
}
```

→ O(n)

* the loop runs once for each item.
  ↳ you visit every element once.
  ↳ It is very common in loops over arrays or lists.

⑧ $O(\log n)$ — Logarithmic Time.

→ when an algorithm cuts the input size in half (or reduces it significantly) with each step, it runs in logarithmic time.
O(log n)
↳ the time grows very slowly even as input gets large.
↳ It is like divide and conquer.

~~↳ It div~~

→ Common in binary search, trees.

④ $O(n \log n)$ — linearithmic Time

→ It is a combination of linear and logarithmic — It is faster than $O(n^2)$, slower than $O(n)$.

→ It is like doing a linear loop, but inside each step, you're also doing a log n operation (like dividing).

→ but for comparison-based sorting.
→ shows up in merge sort, quick sort, heap sort.

```
int[] arr = {5, 3, 8, 2, 1, 4};
Array.sort(arr);
for (int num : arr) {
    System.out.println(num + " ");
}
```

(5) **O(n²)** — Quadratic Time

→ An algorithm is said to have O(n²) time complexity when the running time increases proportionally to the square of the input size.

↳ As the input grows, time grows faster — n multiplied by n. (usually) caused by nested loops.

↳ It is very common in nested loop.
↳ Mostly used in Bubble sort, selection sort, comparing each element pair of
↳ Brute force solution.

⑥ $O(2^n)$ — Exponential Time.

↳ Time doubles with each added input

↳ each time the input increase by 1.

the operations : doubles ... grows extremly fast

mostly used in :

↳ Recursion

↳ Subset generation

↳ Solving the Fibonacci number naively.

↳ Backtracking algorithm.

↳ for input size n, steps = $2 \times 2 \times 2 \times \ldots$ (n time)

$= 2^n$

```
Public static int fib(int n) {
    if (n<=1) return n;
    return fib(n-1) + fib(n-2);
}

Public PSVM (String[] args){
    int n = 10;
    Sout.println(n + " " + fib(n));
}
```

It is very slow if the input size is large.

O(n!) — Factorial Time.

→ An algorithm has O(n!) time complexity when the number of operations grows factorially with the size of input n.

→ If you increase the input size even a little, the time explodes!

$$n! = n \times (n-1) \times (n-2) \times \dots$$

↳ usually used in Permutations and Combinations. where all possibilities must be explored.

→ It tries every possible Permutations.

for n = 5, you try 5! = 120 combinations.

In Short

one loop = O(n)

nested loop = $O(n^2)$

cut in half each time $O(\log n)$

divide and conquer with merge : $O(n \log n)$

Recursion doubling → $O(2^n)$

every combo, permutation & combination → O(n!)

Calculating the Time Complexity

① Break the code into Parts

- → loops.
- → function calls
- → conditionals
- → Basic operations (like +, -, =, print)

Rules to Calculate

① If single loop → Add O(N)

② If Nested loop → Add $O(n^2)$

③ Drop constants

→ means remove all the constant value while calculating the time complexity.

EXAMPLE

```
PSV print(int n){
    for(int i=0; i<n; i++){
        S.out.println(n);
    }                          } O(n)
    for(int j=0; j<n; j++){
        s.out.println(j);      } O(n)
    }
```

So, we will have, $n + n = 2n$

$\Rightarrow O(2n)$

Now, According to the rule we have to drop the constant

$\Rightarrow O(n^2)$

④ Dropping Non Dominant Term

→ Remove all the non-Dominant terms while Calculating the time complexity.

EXAMPLE

Example code

```
Public Static void printNum (int n) {
    for (int i=0; i<n; i++) {          → O(n)
        S.out.println(i);
        for (int j=0; j<n; j++) {      → O(n)
            S.out.println(j);
        }
    }
    for (int k=0; k<n; k++) {          → O(n)
        S.out.println(k);
    }
}
```

$O(n^2)$

So, we have $O(n^2) + O(n)$

So, according to the rule we have to drop the non dominant term. So drop $O(n)$

$\Rightarrow$ final $\Rightarrow O(n^2)$

⑤ Add and Multiply

① Adding the Sequential code blocks

→ when the code blocks are sequential not inside the code block itself, but outside it.

Example-

```
O(n)    { for (int i = 0; i < n; i++) {
              S.out.println(i);
          }

O(n)    { for (int j = 0; j < n; j++) {
              S.out.println(j);
          }
```

Since both of them are different and here are sequential, so the rule is to add it.

$$O(n) + O(n)$$

② Multiplication

↳ When the code is inside each other, means operations are nested. There we have to do multiply.

EXAMPLE

```
for(int i=0; i<n;i++){ ──→ O(n)
    S.out.println(i);
    for(int j=0; j<n; j++){ ──→ O(n)
        S.out.println(j);
    }
}
```

n x n = O(n²)

So, as it is nested, we have to multiply it.

$n \times n = n^2$

So the final time = $O(n^2)$

Space Complexity:

→ Space complexity is the total amount of memory algorithm or program needs based on the size of the input.

It include:

↳ Memory for input.
↳ Memory for variable
↳ Memory for function use.
↳ Memory for data structure like Arrays, list, hash map etc.

① Count memory used by variable.

- int, bool, string, variable → O(1) space
- Array of size n → O(n) space.
- 2D matrix of size n × n → O(n²) space

② Count memory used in function calls (like recursion)

↳ Each recursive call uses stack space

↳ Recursive depth of n = O(n)

space

Basic Concepts to Remember.

Integer.Max_value → It is the highest integer

Integer.Min_value → It is the lowest integer

when to use for or while loop?

→ while loop is used when we don't know weather we have to increament the counter in first or anywhere in the code.

But when we know that the increament can be done in the starting of the ment execution.