TIME COMPLEXITY

★ **Execution Time :** A measure used to estimate time taken for the execution of a program wholly.

→ **How do we calculate this execution time of a program?**
Usually when we talk about time, first units to calculate that comes to mind are seconds, minutes etc. But, Here in order to calculate execution time, we look for a Machine Independent Measure: **BASIC UNIT OPERATIONS**.

→ **What are Basic Unit operations / Single Unit Operations?**
Our programs perform multitude of operations like:

- Taking Input, `cin >> x;`
- Printing Output, `cout << x;`
- Assignment operations, `a = b;`
- Arithmetic operations, `a++;`

A program is a collection of basic unit operations. Each instruction in itself can be called basic unit operation. ∴ TIME COMPLEXITY / EXECUTION TIME is simply measured by calculating how many instructions will be executed in a program.

For the sake of simplicity, we assign unit operation a constant value of 1 (indicating one unit operation)

★ **Measure of Time Complexity :**

Big O : measure of worst case → when algo inputs reaches the tightest bound.
For a basic unit operation like, a++, The Time Complexity can be represented as O(1) → Big O of 1
↓ representing 1 unit operation.

We should try keep the total no. of instrcs in our prgrm $< 10^8$

→ On platforms like codechef etc., we are given a Time limit of 1s for our program execution. That 1s usually hints at max 10^8 operations

1s → 10^8 operations allowed

→ One thing to note is that while basic unit operation is always one liner, It doesn't necessarily mean that they will only be executed once. Introduction of loops, function calls can increase the number of times a single unit operation is executed.

For example

```
for(int i=0; i<n; i++){
    cout<<i*i;
}
```

Here while the basic operation is simply (i*i) but it will happen n times because the condition lies within a loop.

So we can denote the occurence of i*i with

$\Rightarrow O(n)$

represent occurence of i*i (n times)

Hence, variables like n can affect the occurence of Basic unit operations and ultimately the time complexity.

Note :→ Time Complexity is Machine Independent & Abstracted.
→ It hints more towards algorithmic estimation.

★ EXAMPLES :

1

```
for (int i=1 ; i<=n ; i++){
        cout <<n ;          // print
}
```

(1) (2) (3) (4)

In total, let us calculate the number of unit operations.

① Initialization, $i=1$ → 1 time

② Comparison, $i \leq n$ → happens $(n+1)$ times

$i=1$ $(\leq n)$

$i=2$ $(\leq n)$

! keeps on happening

$i=n+1$ $\leq n$ → termination

From 1 to $n+1$, all values will be compared

③ Increment, $i++$ → happens n times

i starts with 1, increment to 2 - - -

Keeps on happening till n.

④ Print operation → happens n times

(explained previously)

Combining ①, ②, ③ & ④ :

$$\Rightarrow 1 + n + 1 + n + n$$
$$= 3n + 2$$

We will remove the constants here, as they are not the growing factor.

$$\Rightarrow O(n)$$

→ Based on $O(n)$, what is the largest value of n for which the algo will work?

$n \lesssim 10^8$ [as 10^8 operations are allowed in 1s which is usually our Time limit].

To be safe :

$n <= 5 \times 10^7$

Using the above information, we can calculate optimized solution for any problem.

① Look at constraints of variable. Ex : N.

If constraint is something like $0 \leq N \leq 10^6$, we know that we want to keep our operations strictly less than 10^8. Looking at the max value of N which is $10^6 (< 10^8)$ we can code a solution of $O(N)$.

But what if our constraint was $0 \leq N \leq 10^{18}$.

Here, we cannot code a linear solution $O(N)$.

But, we will have to rely on $O(\log n)$ like Solution.

2.

```
for(int i=1; i<=n ; i++){
    for(int j=1; j<=n ; j++){
        cout<<j;
}}
```

Previously we had seen O(n) was relying on how many iterations we are performing.
Let's break down the iteration of our loops:

i=1 : j is iterating till n.
i=2 : j→1 - - - - n.
i=3 : j→1 - - - - n.
⋮
i=n : j→1 - - - - n.

i → iterating n times
j → iterating n times

In total, the number of iterations are n*n
∴ O(n²)

→ If O(n²) is the T.C, then what is the max value n can have?

$O(n^2) \leq 10^8$

$n < 10^4$

[Refrain from using nested loops whenever n has gone above 10^4]

3.

```
for (int i ---)
    for (int j ---)
        for (int k ---)
```

Here there are 3 layers of nesting. Taking results from previous examples, T.C here will be $\Rightarrow O(n^3)$

4.

```
for (int i=1; i<=n; i++) {
    for (int j=1; j<=i; j++) {
        print j;
    }
```

For $i=1$, j is iterating till 1 $\rightarrow 1$

$i=2$, $j: 1 \rightarrow 2$ $\rightarrow 2$

$i=3$, $j: 1, 2, 3$ $\rightarrow 3$

$\vdots$

$i=n$, $j: 1, 2, 3 \text{ ---- } n$ $\rightarrow n$

In total, no. of iterations

$$1+2+3+ \text{ ---} +n$$

$$= \frac{n(n+1)}{2} = \frac{n^2+n}{2} \Rightarrow O(n^2)$$

Here we take n^2 as n^2 grows faster than n.

Highest exponent is the determining factor for Time Complexity.

Ex: $O(N^3+N)$ N^3 grows faster than N for the same value. Hence

$\Rightarrow O(N^3)$

5. for (i=1, i<=n, i*=2)
   print i;

i=1)*2
i=2)*2
i=4)*2
i=8)*2
.
.
.
$i = 2^x \leq n$

$2^0\ 2^1\ 2^2 \_ \_ \_ 2^x$

Max value of x:

$2^x = n$

Taking log both sides

$\log(2^x) = \log n$

$x \cdot \log_2 2 = \log_2 n$

$x \cdot 1 = \log n$

$\boxed{x = \log n}$

$\Rightarrow O(\log n)$

↳ no. of iterations happening

6.

```
While(n>0){
    n/=2;
}
```

$\Rightarrow O(\log n)$

$$N \to \frac{N}{2} \to \frac{N}{4} \to \dots \Rightarrow 1$$

$$\frac{N}{2^0} \to \frac{N}{2^1} \to \frac{N}{2^2} \to \dots \to \frac{N}{2^x}$$

Equating,

$$\frac{N}{2^x} = 1$$

$$N = 2^x$$

From previous question, we can derive here $x = \log n$.

★ For cases like:

- $i \to i * K$
- $n \to n/K$

$\Rightarrow O(\log_K(n))$

If value of i is increasing by a factor of 2 then the base of the log becomes 2.
If the value of i is increasing by a factor of 3, then the base of the log becomes 3 and so on.

7.

```
for (i → 1 to N){        ← O(N)
}
for (i 1 to M){          ← O(M)
}
```

Now, we can separately calculate the T.C of these individual loops and since they are sequential, we can simply add these.

$\Rightarrow O(N+M)$

Rule of Thumb:

Nested loops → Multiply

Sequential → Add