What is time complexity ?

Ans : Time complexity gives the "idea" of the amount of time taken by an algorithm as a function of the input size.

It does not necessarily depict the exact execution-time taken by an algorithm, because run-time of a program depends on the following factors :

1. Computer Hardware / Architecture
2. Background Processes and load on the system during execution.
3. The exact location in memory where the program is stored.

, etc.

Thus, it's impossible to determine the execution time of a program that is applicable in all possible cases.

To compare different algorithms, we can easily calculate the time (and space) complexity of a program and judge which is faster and in general, more optimized.

How to Calculate ?

Big Oh is used to denote the worst-case time complexity (upper-bound) of an algorithm.

Eg : I take at most 15 minutes to finish my lunch.

Constant Time : O(1)
Linear Time : O(n).

Eg:
```
for(int i=0; i<10; i++){
```

```
        cout << "Hello" << endl;
    }
```

This will always run the loop 10 times irrespective of any input given. Thus, constant time complexity – $O(1)$.

Eg:
```
for(int i=0 ; i<n ; i++){
        cout << "Hello" << endl;
    }
```

This will always run the loop n times which depends on n 'linearly'. Thus, linear time complexity – $O(n)$

Order of Time Complexity :

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

← Faster / Efficient

Finding Upper Bound :

Q1: $f(n) = 4n^4 + 3n^2 + 2$

1. Remove all terms of lower power/precendence (refer to the order above).

2. Remove constants multiplied/divided with/from the highest precendence term.

① $O(f(n)) = 4n^4 + \cancel{3n^2} + \cancel{2} \approx 4n^4$

② $O(f(n)) = \cancel{4} \cdot n^4 = \boxed{n^4}$

$\Longrightarrow$ $f(n) = O(n^4)$ which means that $f(n)$ will always be upper-bounded/smaller than n^4 after some value of $n = n_0$.

Q2 $f(n) = 4\log n + 5\sqrt{n} + 17$

$O(f(n)) = \cancel{4}\log n + \cancel{5}\sqrt{n} + \cancel{17} = \boxed{\sqrt{n}}$

$f(n) = O(\sqrt{n})$

Q3 Reverse an array.

→ We swap arr[i] with arr[j] starting with $i=0$ & $j=n-1$ and increment & decrement i & j respectively until $i<j$.

Thus, we make approx. $\frac{n}{2}$ swaps for an array of length n.

$f(n) = \frac{n}{2} \Rightarrow O(f(n)) = n$

$f(n) = O(n)$

Q4 Linear Search.

We traverse the whole array, so:

$f(n) = n \Rightarrow O(f(n)) = n$

$f(n) = O(n)$

Q5.

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a = a + j;
    }
}
for (k = 0; k < N; k++) {
    b = b + k;
}
```

} Runs N times for each iteration i : 0 to N-1

} Runs N times.

Time Complexity = $O(f(N)) = O(N^2 + N) = O(N^2)$

Q6.

```
int a = 0;
for (i = 0; i < N; i++) {
```

Careful!

Q6.

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

Careful !

Runs from N to i+1 times for i: 0 to N-1.

Detailed Soln:

For i = 0 : Inner for loop runs N times (j=N; j>0; j--)

For i = 1 : Inner for loop runs N-1 times (j=N; j>1; j--)

For i = 1 : Inner for loop runs N-2 times (j=N; j>2; j--)

⋮

For i=N-1: Inner for loop runs 1 time.

Total: $N + (N-1) + (N-2) + \dots + 1$

$= 1 + 2 + 3 + \dots + N$

$= \frac{N(N+1)}{2}$

$= \frac{N^2}{2} + \frac{N}{2}$

$f(n) = \frac{N^2}{2} + \frac{N}{2} = \boxed{O(N^2)}$

10^8 Operation Rule:

Most modern machines can perform 10^8 operations/ second.

We use the constraints given in a question to determine the maximum T.C. I can have in my solution.

input size	required time complexity
$n \le 10$	$O(n!)$
$n \le 20$	$O(2^n)$
$n \le 500$	$O(n^3)$
$n \le 5000$	$O(n^2)$
$n \le 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

Source : CodeForces.

Space Complexity : Gives the 'idea' of the amount of space required by a program " with respect to " the input.

O(1) Space Complexity -

① `int a;`

② `int a, b, c, d, z;`

③ `int arr[1000];`

O(n) Space Complexity :

① `int arr[n];` // Bad practice

② `vector<int> v(n);`