Analysis of Algorithm

Analysing the efficiency of algorithm

Efficiency of algorithm dependents on 2 factors

1] space efficiency/ complexity

2] time efficiency/ complexity

Space complexity: the amount of memory required to run the program completely and efficiently.

Time complexity: how fast a given algorithm is executed.

Mainly time complexity depends on *"choice of an algorithm"* and *"no of inputs"*.

Order of growth of an algo directly proportional to number of inputs or size of inputs to be processed.

n: no of inputs

N	Log N	N	N log N	N^2	N^3	2^N	N!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	655636	Large no.
32	5	32	160	1024	32768	Large no.	Very large no.

As size of N increases order of growth increase.

$1 < \log \log N < \log N < N^{1/3} < N^{1/2} < N < N \log N < N^2 < N^3 < N^N < 2^N < N!$

Asymptotic Notation: The value of function may increase or decrease as the value of n increases. Asymptotic notation used to compare two algorithms.

Three types

1) O (Big Oh)
2) Ω (Big Omega)
3) ϴ (Big Theta)

1) Big Oh: n indicates size of input and g(n) is function then O(g(n)) is defined as set of function with a small or same order of growth as g(n) as n goes to infinity

   E.g. $O(n^3)$ means order of growth can be n, n^2, and max n^3

2) Big Omega: n indicates size of input and g(n) is function then Ω (g(n)) is defined as set of function with larger or same order of growth as g(n) as n goes to infinity

   E.g. $\Omega(n^3)$ means order of growth can be min n^3, n^4, n^5 and so on...

3) Big Theta: n indicates size of input and g(n) is function then ϴ (g(n)) is defined as set of function that have same order of growth as g(n) as n goes to infinity

   E.g. $\Theta(n^3)$ means order of growth can be n^3, $2n^3$, $2n^3+n^2+5$ …

O(n)

```
for (int i = 1; i <= n; i += c) {
        // some O(1) expressions
   }

for (int i = n; i > 0; i -= c) {
        // some O(1) expressions
   }
```

O(n^2)

```
   for (int i = 1; i <=n; i += c) {
       for (int j = 1; j <=n; j += c) {
          // some O(1) expressions
       }
   }

   for (int i = n; i > 0; i -= c) {
       for (int j = i+1; j <=n; j += c) {
          // some O(1) expressions
   }
```

O(log(n))

```
for (int i = 1; i <=n; i *= c) {
       // some O(1) expressions
   }

   for (int i = n; i > 0; i /= c) {
       // some O(1) expressions
   }
```

Time complexity = O(n^2logn).

```
void fun(int n)
{
    for (int i = 0; i < n / 2; i++)
        for (int j = 1; j + n / 2 <= n; j++)
            for (int k = 1; k <= n; k = k * 2)
```

```
                cout << "GeeksforGeeks";
}
```

Explanation –
Time complexity of 1st for loop = O(n/2) = O(n).
Time complexity of 2nd for loop = O(n/2) = O(n).
Time complexity of 3rd for loop = O(log2n).

How to combine time complexities of consecutive loops?
When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
   for (int i = 1; i <=m; i += c) {
        // some O(1) expressions
   }

   for (int i = 1; i <=n; i += c) {
        // some O(1) expressions
   }

   Time complexity of above code is O(m) + O(n) which is O(m+n)
   If m == n, the time complexity becomes O(2n) which is O(n).
```

Practice questions with answer

https://www.geeksforgeeks.org/practice-questions-time-complexity-analysis/

https://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/?ref=lbp

https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/?ref=lbp