UNIT I ROLE OF ALGORITHMS IN COMPUTING & COMPLEXITY ANALYSIS

Algorithms – Algorithms as a Technology – Time and Space complexity of algorithms – Asymptotic analysis – Average and worst-case analysis – Asymptotic notation – Importance of efficient algorithms – Program performance measurement – Recurrences: The Substitution Method – The Recursion – Tree Method – Data structures and algorithms.

TIME AND SPACE COMPLEXITY OF ALGORITHM

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

- Independent of the machine and its configuration, on which the algorithm is running on.
- Shows a direct correlation with the number of inputs.
- Can distinguish two algorithms clearly without ambiguity.

Time Complexity

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. The time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running. In order to calculate time complexity on an algorithm, it is assumed that a constant time c is taken to execute one operation, and then the total operations for an input length on N are calculated. Consider an example to understand the process of calculation: Suppose a problem is to find whether a pair (X, Y) exists in an array *A* of *N* elements whose sum is *z*. The simplest idea is to consider every pair and check if it satisfies the given condition or not.

The pseudo-code is as follows:

```
int a[n];
for(int i = 0;i < n;i++)
        cin >> a[i];
for(int i = 0;i < n;i++)
        for(int j = 0;j < n;j++)
                if(i!=j && a[i]+a[j] == z)
                        return true
return false
```

Assume that each of the operations in the computer takes approximately constant

time c. The number of lines of code executed actually depends on the value of *z*. During analyses of the algorithm, mostly the worst-case scenario is considered, i.e., when there is no pair of elements with sum equals z. In the worst case,

- *N**c operations are required for input.
- The outer loop i, runs *N* times.
- For each i, the inner loop j loop runs *n* times.

So total execution time is ***N**c + *N**N**c + c**. Now ignore the lower order terms since the lower order terms are relatively insignificant for large input, therefore only the highest order term is taken (without constant) which is ***N**N*** in this case. Different notations are used to describe the limiting behavior of a function, but since the worst case is taken so big-O notation will be used to represent the time complexity.

Hence, the time complexity is **O(N^2)** for the above algorithm. Note that the time complexity is based on the number of elements in array A i.e the input length, so if the length of the array will increase the time of execution will also increase.

Order of growth is how the time of execution depends on the length of the input. In the above example, it is clearly evident that the time of execution quadratically depends on the length of the array. Order of growth will help to compute the running time with ease.

Another Example: Let's calculate the time complexity of the below algorithm:

```
count = 0
for(int i =N;  i > 0; i /= 2)
        for (int j = 0; j < i; j++)
                count++;
```

It seems like the complexity is **O(N * log N)**. **N** for the **j′s** loop and **log(N)** for **i′s** loop. But it's wrong. Let's see why.

Think about how many times **count++** will run.

- When i = N, it will run N times.
- When i = N / 2, it will run N / 2 times.
- When i = N / 4, it will run N / 4 times.
- And so on.

The total number of times **count++** will run is **N + N/2 + N/4+…+1= 2 * N**. So the time complexity will be **O(N)**. Some general time complexities are listed below with the input range for which they are accepted in competitive programming:

Input Length	Worst Accepted Time Complexity	Usually type of solutions
10 -12	O(N!)	Recursion and backtracking
15-18	O(2N * N)	Recursion, backtracking, and bit manipulation
18-22	O(2N * N)	Recursion, backtracking, and bit manipulation
30-40	O(2N/2 * N)	Meet in the middle, Divide and Conquer
100	O(N4)	Dynamic programming, Constructive
400	O(N3)	Dynamic programming, Constructive
2K	O(N2* log N)	Dynamic programming, Binary Search, Sorting, Divide and Conquer
10K	O(N2)	Dynamic programming, Graph, Trees,
1M	O(N* log N)	Sorting, Binary Search, Divide and Conquer
100M	O(N), O(log N), O(1)	Constructive, Mathematical, Greedy Algorithms

Space Complexity

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.

The pseudo-code is as follows:

```
int freq[n];
int a[n];
```

```
for(int i = 0; i<n; i++)
{
        cin>>a[i];
        freq[a[i]]++;
}
```

Here two arrays of length N, and variable i are used in the algorithm so, the total space used is N * c + N * c + 1 * c = 2N * c + c, where c is a unit space taken. For many inputs, constant c is insignificant, and it can be said that the space complexity is O(N).

There is also auxiliary space, which is different from space complexity. The main difference is where space complexity quantifies the total space used by the algorithm, auxiliary space quantifies the extra space that is used in the algorithm apart from the given input. In the above example, the auxiliary space is the space used by the freq[] array because that is not part of the given input. So total auxiliary space is N * c + c which is O(N) only.