**TIME AND SPACE COMPLEXITY**

**What is an Algorithm?**

An algorithm is the step-by-step unambiguous instruction to solve a given problem.

**example**

Let us consider the problem of preparing an *omelette*. To prepare an omelette, we follow the steps given below:

1) Get the frying pan.

2) Get the oil.

a. Do we have oil?

i. If yes, put it in the pan.

ii. If no, do we want to buy oil?

1. If yes, then go out and buy.

2. If no, we can terminate.

3) Turn on the stove, etc...

**Why the Analysis of Algorithms?**

Algorithm analysis helps us to determine which algorithm is the most efficient in terms of time and space consumed.

**Example**

To go from city *“A”* to city *“B”*, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more).

**Goal of the Analysis of Algorithms**

The goal of the *analysis of algorithms* is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

**What is Running Time Analysis?**

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. The following are the common types of inputs.

* Size of an array
* Polynomial degree
* Number of elements in a matrix
* Number of bits in the binary representation of the input
* Vertices and edges in a graph.

**How to Compare Algorithms**

To compare algorithms, let us define a *few objective measures:*

**Execution times?** *Not a good measure* as execution times are specific to a particular computer.

**Number of statements executed?** *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

**Ideal solution?** Let us assume that we express the running time of a given algorithm as a function of the input size *n* (i.e., *f*(*n*)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

**What is Rate of Growth?**

The rate at which the running time increases as a function of input is called *rate of growth*. Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say *buying a car*. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).



**Commonly used Rates of Growth**

Below is the list of growth rates you will come across in the following chapters.

**The diagram below shows the relationship between different rates of growth**



**Types of Analysis**

There are three type of analysis.

* Worst case
* Best case
* Average case

**Worst case**

* Defines the input for which the algorithm takes a long time (slowest time to complete).
* Input is the one for which the algorithm runs the slowest.

**Best case**

* Defines the input for which the algorithm takes the least time (fastest time to complete).
* Input is the one for which the algorithm runs the fastest.

**Average case**

* Provides a prediction about the running time of the algorithm.
* Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
* Assumes that the input is random.

For a given algorithm, we can represent the best, worst and average cases in the form of expressions. As an example, let *f*(*n*) be the function, which represents the given algorithm.

Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

**Asymptotic Notation**

Having the expressions for the best, average and worst cases, for all three cases we need to identify the upper and lower bounds. To represent these upper and lower bounds, we need some kind of syntax, and that is the subject of the following discussion. Let us assume that the given algorithm is represented in the form of function *f*(*n*).

**Big-O Notation**

This notation gives the *tight* upper bound of the given function. Generally, it is represented *as f*(*n*)= O(*g*(*n*)). That means, at larger values of *n*, the upper bound *of f*(*n*) is *g*(*n*).



* For example, if *f*(*n*) = n4 + 100*n*2 + 10*n* + 50 is the given algorithm, then *n*4 *is g*(*n*). That means *g*(*n*) gives the maximum rate of growth for *f*(*n*) at larger values of *n*.
* Let us see the O–notation with a little more detail. O–notation defined as O(*g*(*n*)) = {*f*(*n*)*:* there exist positive constants *c* and *n*0 such that 0 ≤ *f*(*n*) *≤ cg*(*n*) for all *n* ≥ *n*0}*. g*(*n*) is an asymptotic tight upper bound for *f*(*n*). Our objective is to give the smallest rate of growth *g*(*n*) which is greater than or equal to the given algorithms’ rate of growth *f*(*n*).
* Generally we discard lower values of *n*. That means the rate of growth at lower values of *n* is not important. In the figure, *n*0 is the point from which we need to consider the rate of growth for a given algorithm. Below *n*0, the rate of growth could be different. *n*0 is called threshold for the given function.

**Big-O Visualization**

* O(*g*(*n*)) is the set of functions with smaller or the same order of growth as *g*(*n*).
* For example; O(*n*2) includes O(1), O(*n*), O(*nlogn*), etc.

**Big-O Examples**

**Example-1** Find upper bound for *f*(*n*) = 3*n* + 8

**Solution:** 3*n* + 8 ≤ 4*n*, for all *n ≥* 8

∴ 3*n* + 8 = O(*n*) with c = 4 and *n*0 = 8

**Example-2** Find upper bound for *f*(*n*) *= n*2 + 1

**Solution:** *n*2 + 1 ≤ 2*n*2, for all *n ≥* 1

∴ *n2* + 1 = O(*n*2) with *c =* 2 and *n*0 = 1

**Example-3** Find upper bound for *f*(*n*) *= n*4 + 100*n*2 + 50

**Solution:** *n*4 + 100*n*2 + 50 ≤ 2*n*4, for all *n ≥* 11

∴ *n*4 + 100*n*2 + 50 = O(*n*4 ) with *c =* 2 and *n*0 = 11

**Example-4** Find upper bound for *f*(*n*) = 2*n*3 *–* 2*n*2

**Solution:** 2*n*3 – 2*n*2 ≤ 2*n*3, for all *n ≥* 1

∴ 2*n*3 *–* 2*n*2 = O(*n*3 ) with *c =* 2 and *n*0 = 1

**Example-5** Find upper bound for *f*(*n*) = *n*

**Solution:** *n ≤ n*, for all *n ≥* 1

∴ *n =* O(*n*) with *c =* 1 and *n*0 = 1

**Example-6** Find upper bound for *f*(*n*) = 410

**Solution:** 410 ≤ 410, for all *n ≥* 1

∴ 410 = O(1) with *c =* 1 and *n*0 = 1