**Data Structure**

*Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. A *data structure* is a special format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

1) *Linear data structures:* Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). *Examples:* Linked Lists, Stacks and Queues.

2) *Non – linear data structures:* Elements of this data structure are stored/accessed in a non-linear order. *Examples:* Trees and graphs.

**Abstract Data Types (ADTs)**

Before defining abstract data types, let us consider the different view of system-defined data types. We all know that, by default, all primitive data types (int, float, etc.) support basic operations such as addition and subtraction. The system provides the implementations for the primitive data types. For user-defined data types we also need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general, user defined data types are defined along with their operations.

To simplify the process of solving problems, we combine the data structures with their operationsand we call this *Abstract Data Types* (ADTs). An ADT consists of *two parts:*

1. Declaration of data

2. Declaration of operations

1. Commonly used ADTs include: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others. For example, stack uses a LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations are: creating the stack, push an element onto the stack, pop an element from the stack, finding the current top of the stack, finding the number of elements in the stack, etc.

**What is an Algorithm?**

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?

1. If yes, put it in the pan.
2. If no, do we want to buy oil?

1. If yes, then go out and buy.

2. If no, we can terminate.

1. Turn on the stove, etc...

What we are doing is, for a given problem (preparing an omelette), we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as:

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

**Why the Analysis of Algorithms?**

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

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 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).



As an example, in the case below, *n*4, 2*n*2, 100*n* and 500 are the individual costs of some function and approximate to *n*4 since *n*4 is the highest rate of growth.



**Types of Analysis**

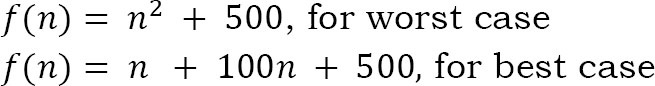
To analyse the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for the case where it takes less time and another for the case where it takes more time.

In general, the first case is called the *best case* and the second case is called the *worst case* for the algorithm. To analyse an algorithm, we need some kind of syntax, and that forms the base for asymptotic analysis/notation. There are three types of analysis:

1. **Worst case**
   1. Defines the input for which the algorithm takes a long time (slowest time to complete).
   2. Input is the one for which the algorithm runs the slowest.
2. **Best case**
   1. Defines the input for which the algorithm takes the least time (fastest time to complete).
   2. Input is the one for which the algorithm runs the fastest.
3. **Average case**
   1. Provides a prediction about the running time of the algorithm.
   2. 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.
   3. 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.



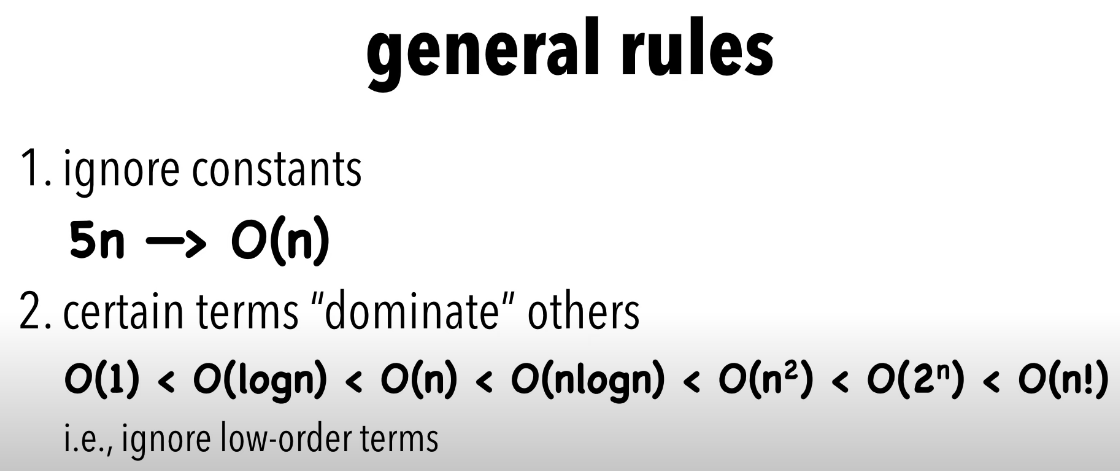
Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

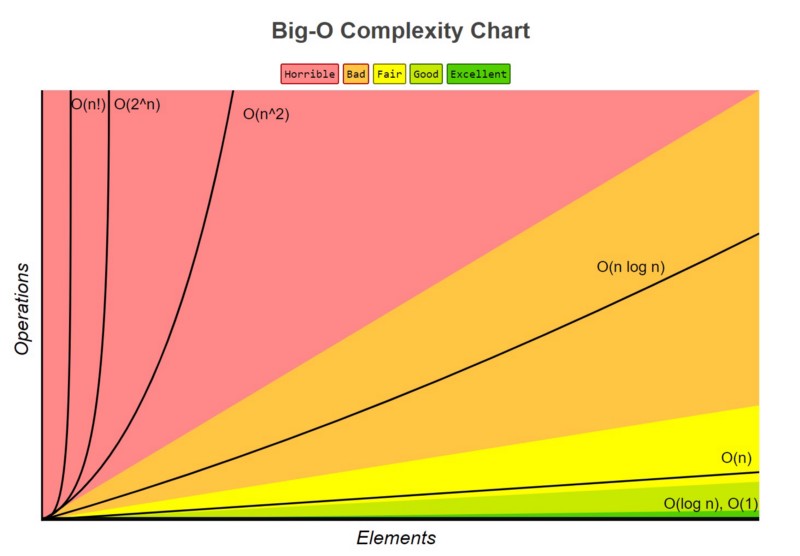
**Asymptotic Notation**

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*).

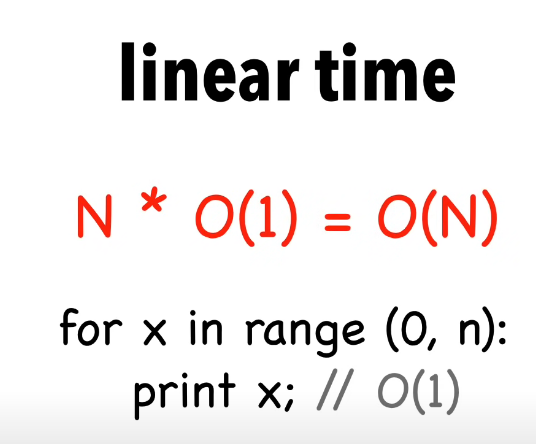
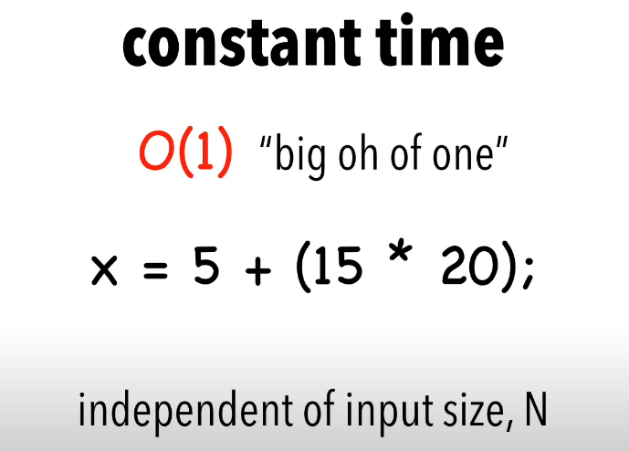
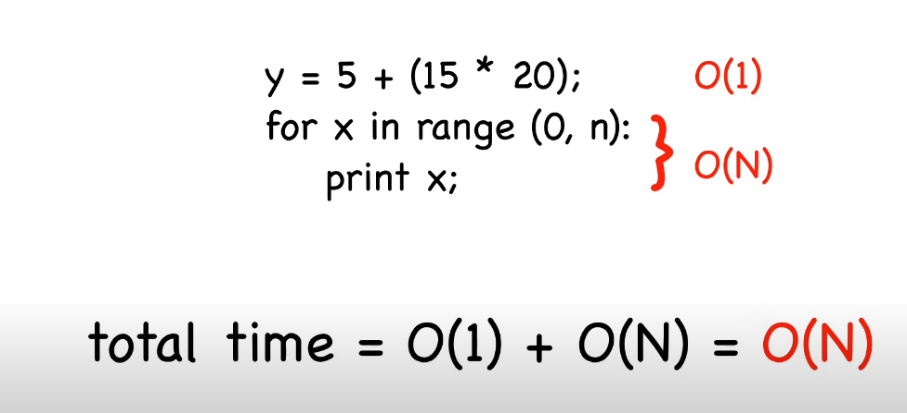
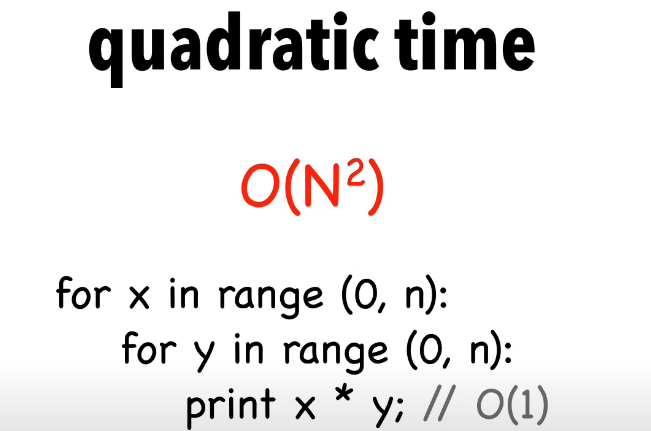
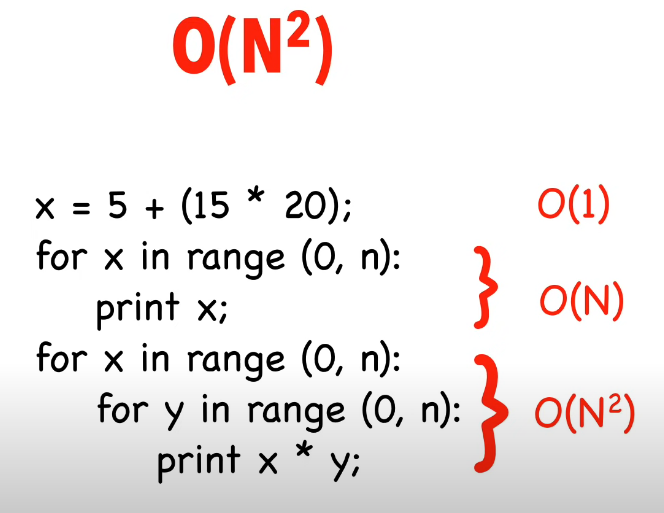
Big O

Simplified analysis of an algorithm’s efficiency. Big O gives the algorithms complexity in terms of Input size, N. It gives us a way to abstract the efficiency of the algorithm code from the machines they run on. We can use Big O to analyse both Time and Space.









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*).

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**.

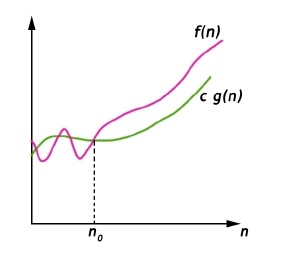


**Big Ω**

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

We write f(n) = Ω(g(n)), If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or above c\*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ c g(n) ≤ f(n), for all n ≤ n0}



**Big Theta Notation**

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.

We write f(n) = Θ(g(n)), If there are positive constants n0 and c1 and c2such that, to the right of n0 the f(n) always lies between c1\*g(n) and c2\*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2g(n), for all n ≥ n0}

