Data Structures by Tilak Dave

Based on academic syllabus copy (180 OB) of Government Polytechnic Pune

UNIT I

Introduction to data structures and Arrays

1.1 Introduction, Basic Terminology, Elementary data structure, Organization, Classification of data structure.

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

For now, we have only one important terminology, we will have more further in the course

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, if our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e., once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Elementary Data Structures

Elementary data structures such as stacks queues lists and heaps will be the "of-the-shelf" components we build our algorithm from. There are two aspects to any data structure:

The abstract operations which it supports

The implementation of these operations.

Note - *** Please don't worry if you do not understand this topic. I too worry that why this is in the first unit. Just focus on the two aspects of any data structure. The operations which are on the 4th page and implementation that is the need and Advantages which are explained above. ***

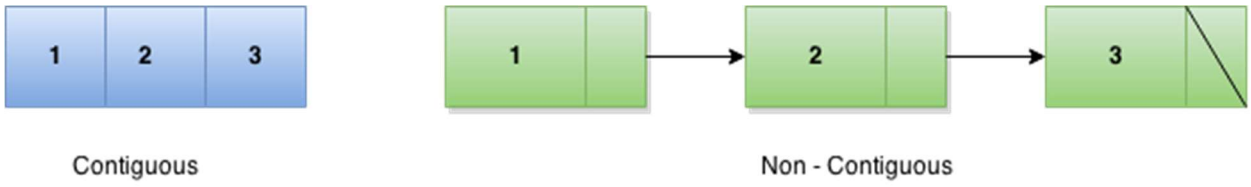
Organization of data In Data Structures

The collection of data you work with in a program have some kind of structure or organization of data In Data Structures. No matter how complex your data structures are they can be broken down into two fundamental types.

1. Contiguous
2. Non-Contiguous

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements.

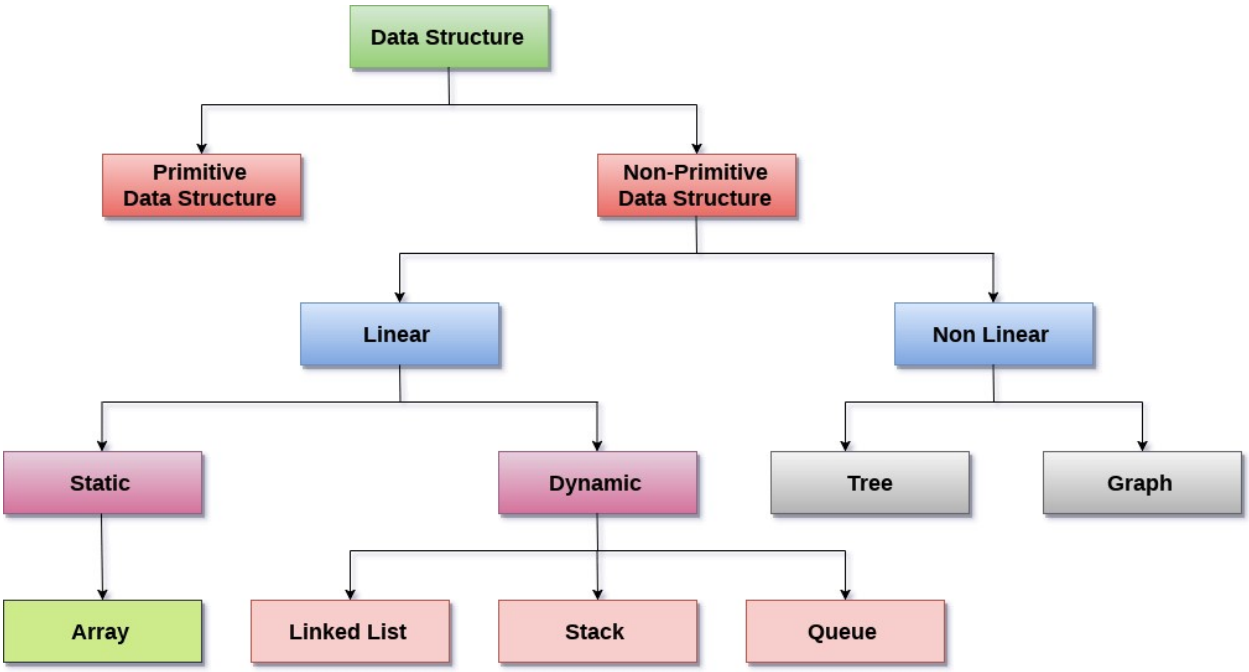
In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node.



Please follow this link to study more about organization of data in data structure.

https://www.onlinetutorialspoint.com/data-structures/organization-of-data-in-data-structures.html

Data Structure Classification



More Terminologies

Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array **age** are:

age [0], age [1], age [2], age [3], ... age [98], age [99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non-Linear Data Structures: This data structure does not form a sequence i.e.; each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non-Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have almost one parent except the root node.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Note - *** No need to learn these terminologies. Just you should know meaning of all those. Don't go directly very deep in any of these. You will learn these more in upcoming pdfs. ***

1.2 Operations on data structures: Traversing, Inserting, deleting, Searching, sorting, and merging.

Operations on data structure

Note - *** These are just introduction to the operations. Operations are mentioned well ahead. ***

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

   Example: If we need to calculate the average of the marks obtained by a student in 6 different subjects, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e., 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

   If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

   If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them in upcoming pdfs.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

1.3 Complexity: Time Complexity, Space Complexity, Big 'O' Notation.

Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e., simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A dynamic part is a space required by variables; whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Time Complexity

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time.

For example, in case of addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.

Note - *** Time and space complexity seems easy to understand but understanding it correctly and deeply is very important. Specially time complexity. ***

I recommend watching separate videos on time complexity on YouTube.

Complexities – Let us understand better

Let us imagine that you and your friend are going to Pizza Hut for eating pizzas. You placed an order of two pizzas. You got the order in about 20 minutes. Great right?

Now let us imagine you are with your family and ordered 4 to 5 pizzas and also some siders. Obviously, it will take more time to deliver.

Which means that as the number of input pizzas increases; so, increases the time for the order to become ready. Also, more the order is more space it will take in the Pizza Hut's Kitchen.

Then what's the solution???

Solution is NOTATIONS; Asymptotic Notations. We will start notation after some time. Before that…

Let us say you have 1GB daily data then you obviously cannot download PUBG but you can download a smaller game like Pokémon Go. So, you will call your friend and transfer the game using any sharing app. But for downloading any small game or app we will not call our friend or visit him.

Because for games < 1GB downloading is fast and more efficient

But for games > 1GB visiting a friend is more efficient.

As the file size grows, time taken by online downloading increases linearly – Time(n)

As the file size grows further, time taken by physical sending remains constant because sharing is done very fast at a negligible speed. – Time (1)

Here, n is the size of your app which is input

and Time is constant for downloading speed and visiting time to a friend.

Calculating order in terms of input size.

As in mathematics, the term which has the most order is most impactful and is taken into account.

In order to calculate the order, the most impactful term containing n is taken into account. (here n is size of input (size of app))

Let us create two algorithms one for downloading app and second for transferring app. (note we are calculating time)

Algo 1 (downloading)

Total time = k n (size of app in Mb)

Here k is the constant which says "it requires 5 sec for downloading 1MB"

Which means if my app is of 1000MB it will take 5000 seconds.

We can conclude that **in Algo 1 time increases as input increases.**

So Algo1 is very much better for small games sizes as it will download the game in just some seconds. But if the game is large it will take too much time. Hence for small input size my algo 1 is efficient. Let us look to the algo 2.

Algo 2 (physical visit and transfer the game)

Total time = k1 + k2 + k3

Here k1 is the time to travel.

K2 is the time you talk with your friend. transferring time is included in k2

K3 is the time to return back to your home.

Which means what may my app size it will require same time.

We can conclude that **in Algo 2 time remains constant for any input size.**

We can now say that for input smaller than 1000MB we will use Algo1 else we will use algo 2.

Now let’s go deeper because it is important:

Note - *** this is very mathematical and takes time to understand. Please read two to three times. ***

How to find the Time Complexity or running time for performing the operations?

The measuring of the actual running time is not practical at all. The running time to perform any operation depends on the size of the input. Let's understand this statement through a simple example.

Suppose we have an array of five elements, and we want to add a new element at the beginning of the array. To achieve this, we need to shift each element towards right, and suppose each element takes one unit of time. There are five elements, so five units of time would be taken. Suppose there are 1000 elements in an array, then it takes 1000 units of time to shift. It concludes that time complexity depends upon the input size.

Therefore, if the input size is n, then f(n) is a function of n that denotes the time complexity.

How to calculate f(n)?

Calculating the value of f(n) for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their f(n) values. We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes. Now, how to find f(n).

Let's look at a simple example.

$f(n) = 5n^2 + 6n + 12$

where n is the number of instructions executed, and it depends on the size of the input.

When n=1

% of running time due to $5n^2$ = $\frac{5}{5+6+12}$ * 100 = 21.74%

% of running time due to 6n = $\frac{6}{5+6+12}$ * 100 = 26.09%

% of running time due to 12 = $\frac{12}{5+6+12}$ * 100 = 52.17%

From the above calculation, it is observed that most of the time is taken by 12. But we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of f(n).

n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

As we can observe in the above table that with the increase in the value of n, the running time of $5n^2$ increases while the running time of 6n and 12 also decreases. Therefore, it is observed that for larger values of n, the squared term consumes almost 99% of the time. As the n^2 term is contributing most of the time, so we can eliminate the rest two terms.

Therefore,

$f(n) = 5n^2$

Here, we are getting the approximate time complexity whose result is very close to the actual result. And this approximate measure of time complexity is known as an Asymptotic complexity. Here, we are not calculating the exact running time, we are eliminating the unnecessary terms, and we are just considering the term which is taking most of the time.

In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical boundation of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.

It is used to mathematically calculate the running time of any operation inside an algorithm.

Example: Running time of one operation is x(n) and for another operation, it is calculated as f(n2). It refers to running time will increase linearly with an increase in 'n' for the first operation, and running time will increase exponentially for the second operation. Similarly, the running time of both operations will be the same if n is significantly small.

Usually, the time required by an algorithm comes under three types:

Worst case: It defines the input for which the algorithm takes a huge time.

Average case: It takes average time for the program execution.

Best case: It defines the input for which the algorithm takes the lowest time

Note - *** further coming is asymptotic notations. Please go forward with a fresh mind. Don’t hurry ***

Asymptotic Notations

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (O)
- Omega Notation (Ω)
- Theta Notation (θ)

Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:

For example:

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)≤c.g(n) for all n≥no

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

In simple words big oh notation says that time required for f(n) will be always smaller than g(n). hence g(n) upper bounds f(n). As we are calculating the highest time for f(n) we are indirectly calculating the worst time complexity of a function.

That’s it. If not understood “message me I will explain more clearly”

Let's understand through examples

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

For n=1, the above condition is true.

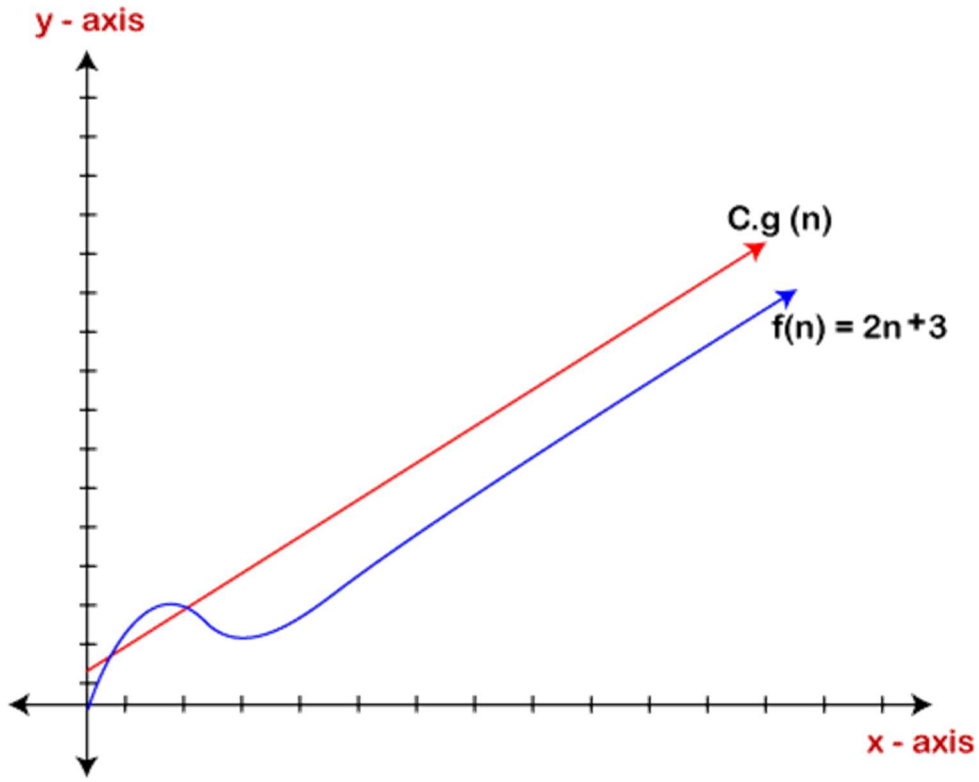
If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e., the Greek letter "omega". It is used to bound the growth of running time for large input size.

If **f(n)** and **g(n)** are the two functions defined for positive integers,

then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)>=c.g(n) for all n≥no and c>0

This implies that f(n) will always grow faster than g(n), or g(n) is a lower bound on the function f(n). In this case, we are calculating the best amount of time of the function which eventually calculates the best time complexity of a function, i.e., how best an algorithm can perform.

In simple words big omega notation says that time required for f(n) will be always greater than g(n). hence g(n) lower bounds f(n). As we are calculating the best time for f(n) we are indirectly calculating the best time complexity of a function.

That's it. If not understood "message me I will explain more clearly"

Let's consider a simple example.

If f(n) = 2n+3, g(n) = n,

Is f(n)= **Ω** (g(n))?

It must satisfy the condition:

f(n)>=c.g(n)

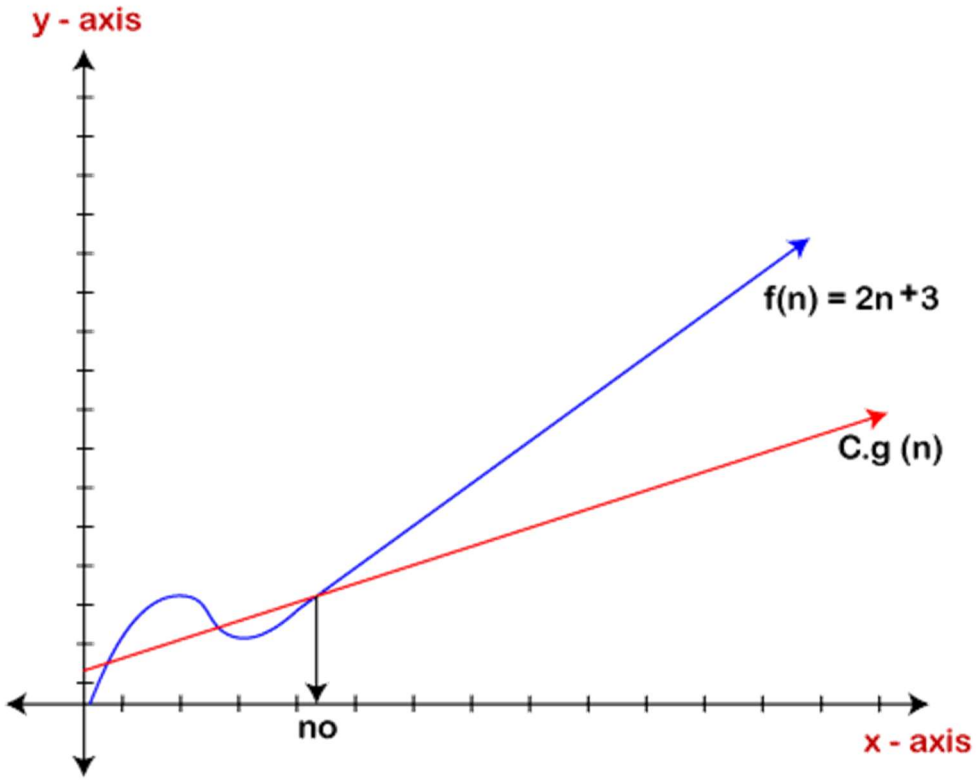
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

2n+3>=c*n

Suppose c=1

2n+3>=n (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.



As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time.

But we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

  Let's understand the big theta notation mathematically:

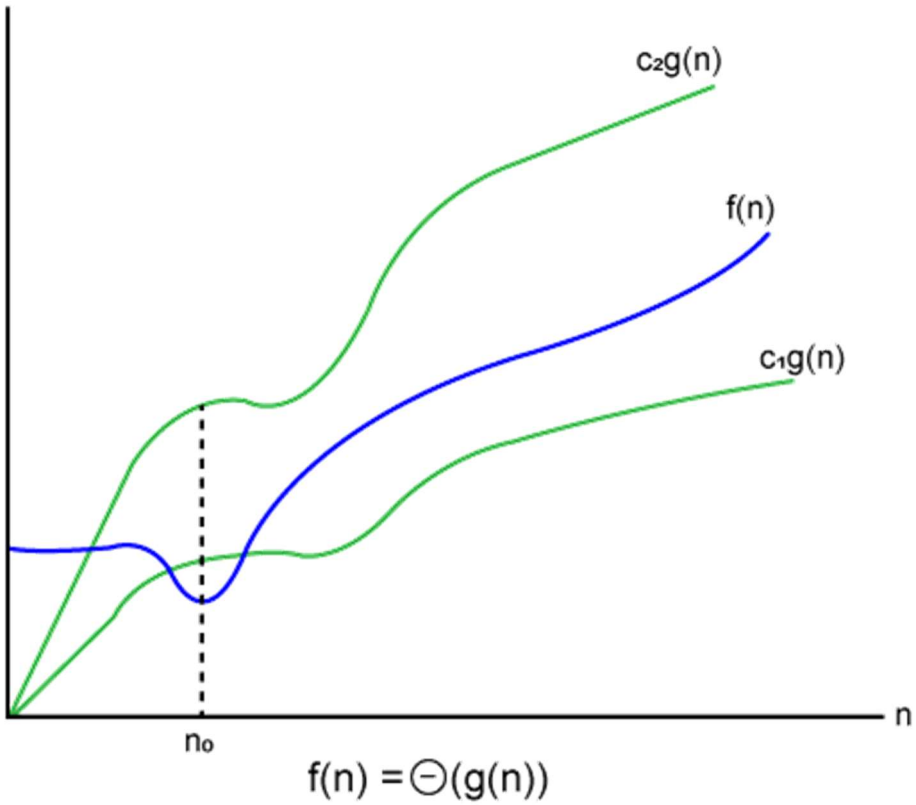
  Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

  f(n)= θg(n)

  The above condition is satisfied only if when

  c1.g(n)<=f(n)<=c2.g(n)

where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



Let's consider the same example where
f(n)=2n+3
g(n)=n

As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).

c1.g(n)<=f(n)<=c2.g(n)

Replace g(n) by n and f(n) by 2n+3

c1.n <=2n+3<=c2.n

if c1=1, c2=2, n=1

1*1 <=2*1+3 <=2*1

1 <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

If n=2

1*2<=2*2+3<=2*2

2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

Why we have three different asymptotic analysis?

As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm.

Suppose we have an array of n numbers, and we want to find the particular element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose, if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst case would be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.

So, three different analysis provide the proper bounding between the actual functions. Here, bounding means that we have upper as well as lower limit which assures that the algorithm will behave between these limits only, i.e., it will not go beyond these limits.

Common Asymptotic Notations

constant	T(1)
linear	T(n)
logarithmic	T(log n)
n log n	T(n log n)
exponential	2T(n)
cubic	T(n3)
polynomial	nT(1)
quadratic	T(n2)