1. Philosophy of Data Structure

You might think that with ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still continue to improve. Won't today's efficiency problem be solved by tomorrow's hardware?

As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Unfortunately, as tasks become more complex, they become less like our everyday experience. So today's computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

In the most general sense, a data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term "data structure" to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring. These ideas are explored further in a discussion of Abstract Data Types.

Given sufficient space to store a collection of data items, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. The most obvious example is an unsorted array containing all of the data items. It is possible to perform all necessary operations on an unsorted array. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days. For example, searching for a given record in a hash table is much faster than searching for it in an unsorted array.

A solution is said to be efficient if it solves the problem within the required resource constraints. Examples of resource constraints include the total space available to store the data—possibly divided into separate main memory and disk space constraints—and the time allowed to perform each subtask. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements. The cost of a solution is the amount of resources that the solution consumes. Most often, cost is measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

- **Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.

- **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Data Structure is a branch of Computer Science. The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program. Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized in the future when required.

Some examples of Data Structures are Arrays, Linked Lists, Stack, Queue, Trees, etc. Data Structures are widely used in almost every aspect of Computer Science, i.e., Compiler Design, Operating Systems, Graphics, Artificial Intelligence, and many more.

1.1 Need of Data Structures

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

1. **Data Search** − Consider an inventory of 1 million items of a store. If the application is to search an item, it has to search an item in 1 million items every time slowing down the search. As data grows, search will become slower.
2. **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
3. **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

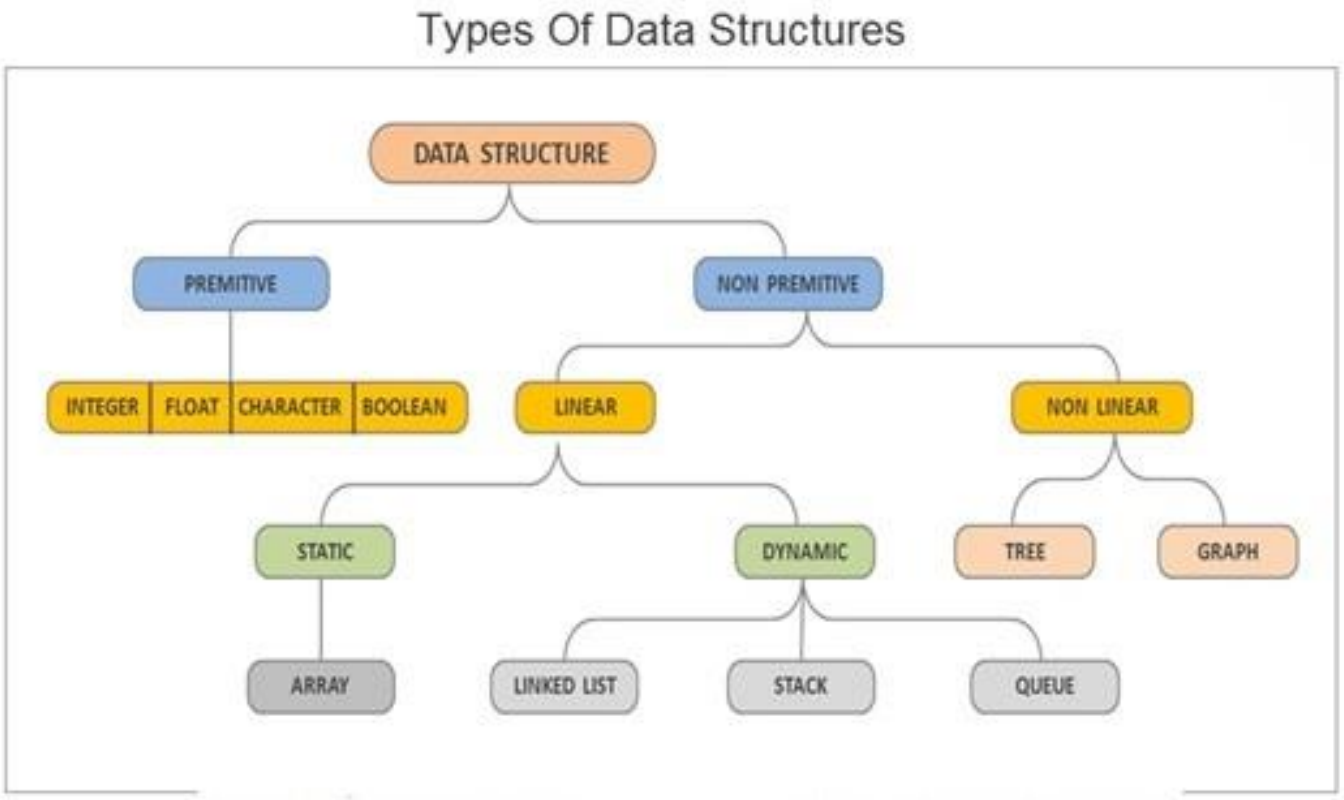
1.2 Characteristics and Types

Characteristics

1. **Correctness** − Data structure implementation should implement its interface correctly.
2. **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
3. **Space Complexity** − Memory usage of a data structure operation should be as little as possible.
4. **Access Methods**: Data structures provide specific methods for accessing and manipulating the stored data, such as insertion, deletion, search, or retrieval operations.

5. **Storage Organization**: Data structures determine how data is stored in memory, such as arrays, linked lists, trees, or hash tables.
6. **Extensibility**: Data structures can be extended or customized to meet specific requirements by adding new operations or modifying existing ones.

Types



2. Abstract Data Type (ADT) and Data Structures

An abstract data type is an **abstraction** of a data structure that provides **only the interface** to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

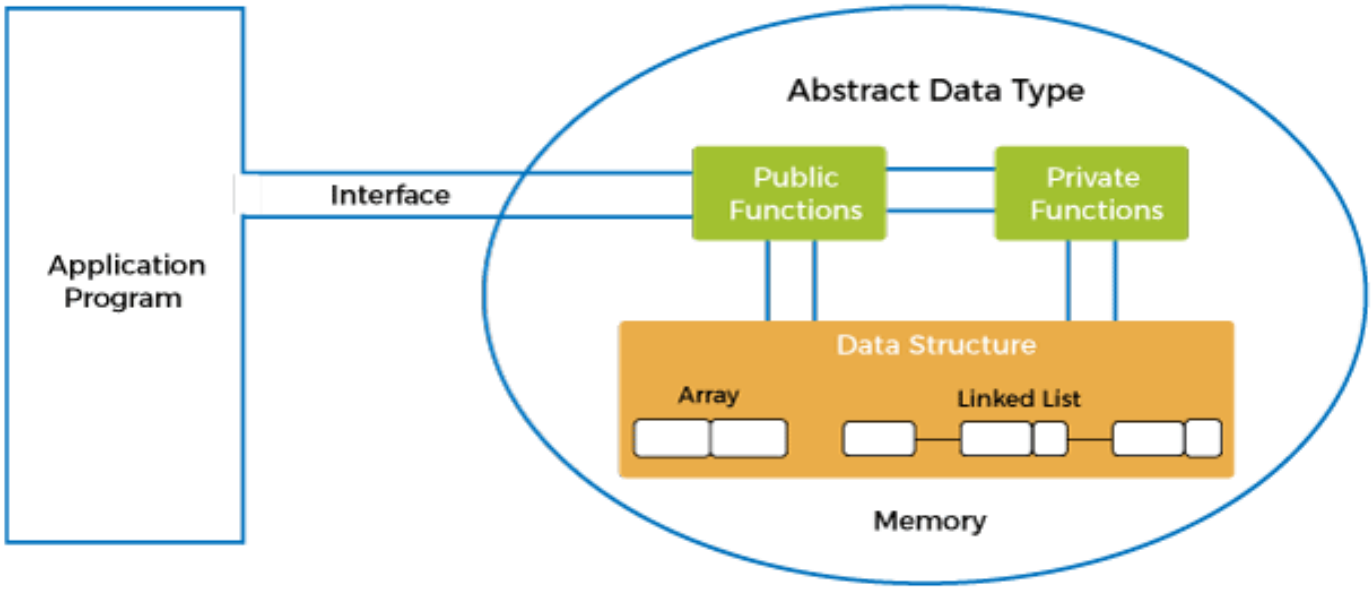
In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.

Abstract data type model

Before knowing about the abstract data type model, we should know about abstraction and encapsulation.

Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.

Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.



The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program.

The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Some examples are List ADT, Stack ADT, Queue ADT.

3. Algorithm Design Techniques

An Algorithm is a procedure to solve a particular problem in a finite number of steps for a finite-sized input.

Algorithm design techniques refer to various approaches and strategies employed to create efficient and effective algorithms.

Here are some commonly used algorithm design techniques:

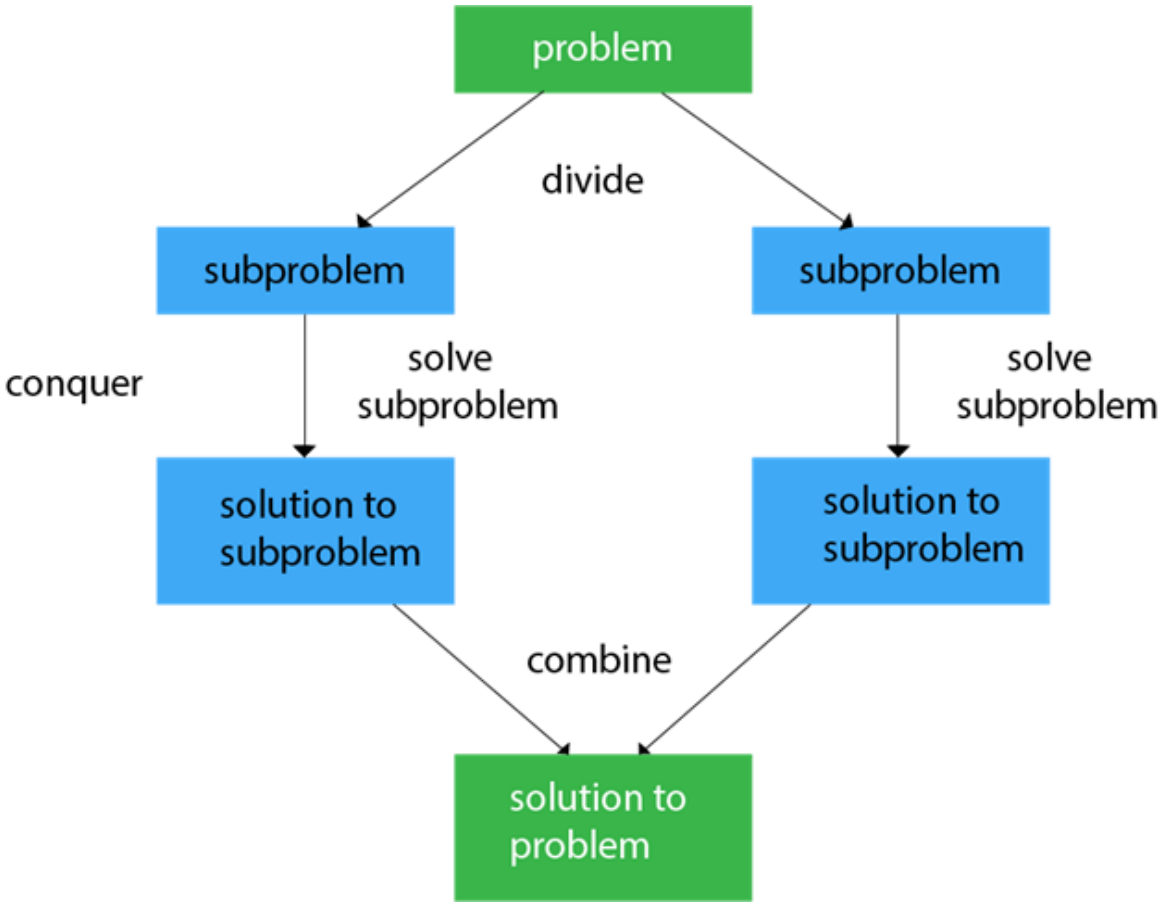
- Divide and Conquer
- Greedy Algorithms
- Dynamic Programming
- Backtracking
- Brute Force
- Randomized Algorithms

3.1 Divide and Conquer

Divide and conquer approach breaks down a problem into multiple sub-problems recursively until it cannot be divided further. These sub-problems are solved first and the solutions are merged together to form the final solution.

The common procedure for the divide and conquer design technique is as follows −

- **Divide** − We divide the original problem into multiple sub-problems until they cannot be divided further.
- **Conquer** − Then these subproblems are solved separately with the help of recursion
- **Combine** − Once solved, all the subproblems are merged/combined together to form the final solution of the original problem.



There are several ways to give input to the divide and conquer algorithm design pattern. Two major data structures used are − arrays and linked lists.

Some algorithms based on divide-and-conquer programming approach are

- Merge Sort
- Quick Sort
- Binary Search
- Tower of Hanoi
- Closest Pair

Advantages of DAC:

1. **Efficiency**: Divide and Conquer algorithms offer high efficiency by breaking down complex problems and utilizing parallelism.
2. **Scalability**: These algorithms can efficiently distribute workload across multiple processors or threads, making them suitable for parallel and distributed computing.
3. **Simplified Problem Solving**: The approach simplifies problem-solving by breaking complex problems into manageable subproblems.
4. **Code Reusability**: Recursive nature allows for code reusability across similar problems.

Disadvantages of DAC:

1. **Overhead: (**It represents the extra work or resources needed to implement the divide and conquer approach compared to other algorithms or methods.) Recursive function calls and merging subproblem solutions can introduce additional overhead which can lead to increased memory consumption and slower execution compared to iterative approaches.
2. **Memory Requirements**: In certain cases, divide and conquer algorithms may require additional space to store intermediate results or subproblem solutions. This extra space requirement can be a disadvantage when dealing with large input sizes or limited memory resources.
3. **Problem Dependency**: Not all problems can be effectively solved using Divide and Conquer approach due to strong dependencies between subproblems.
4. **Complexity**: Analyzing time and space complexity can be complex due to intricate recurrence relations in these algorithms.

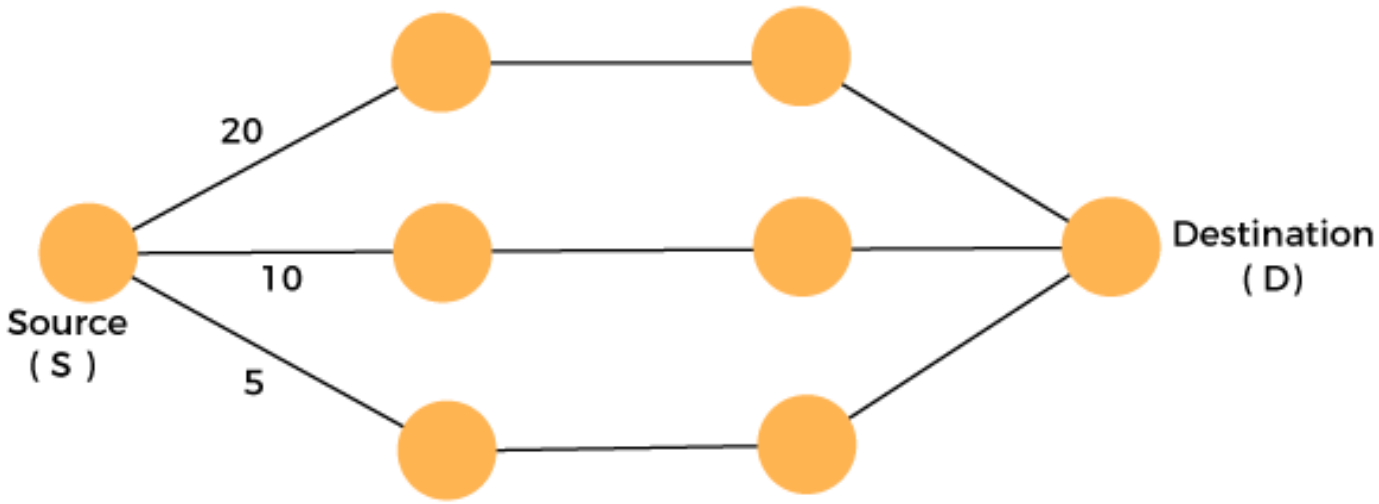
3.2 Greedy Algorithm

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

Greedy method is one of the strategies used for solving the optimization problems. An optimization problem is a problem that demands either maximum or minimum results.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as the feasible, whereas the optimal solution is the best solution among all the solutions.

Consider the graph which is given below:



We have to travel from the source to the destination at the minimum cost. Since we have three feasible solutions having cost paths as 10, 20, and 5. 5 is the minimum cost path so it is the optimal solution. This is the local optimum, and in this way, we find the local optimum at each stage in order to calculate the global optimal solution.

Applications

- **Minimum Spanning Tree**: Greedy algorithms like Kruskal's or Prim's algorithm are used to find the minimum spanning tree in a connected weighted graph.
- **Shortest Path**: Algorithms like Dijkstra's and Bellman-Ford use greedy strategies to find the shortest path between two nodes in a graph.
- **Huffman Coding**: Greedy algorithms are employed to construct optimal prefix codes for data compression, such as in Huffman coding.
- **Job Scheduling**: Greedy algorithms can be used to schedule jobs or tasks based on priority, deadlines, or other criteria.
- **Knapsack Problem**: Certain variations of the knapsack problem can be solved using greedy algorithms, where items are selected based on their value-to-weight ratios.

Advantages

1. **Simplicity**: Greedy algorithms are easy to understand and implement.
2. **Efficiency**: They often have a time complexity that is less than or equal to other algorithms.
3. **Intuitive Approach**: They make locally optimal choices based on current information.

Disadvantages

1. **Lack of Global Optimal Solution**: Greedy algorithms can produce suboptimal or incorrect solutions in some cases.
2. **Dependency on Problem Constraints**: The effectiveness of greedy algorithms relies on the problem's constraints and structure. In problems with complex dependencies or constraints, greedy algorithms may fail to produce the desired solution.
3. **Difficulty in Problem Selection**: Identifying suitable problems for the greedy approach can be challenging. Greedy algorithms require careful analysis and understanding of the problem's characteristics and constraints. Applying the greedy approach incorrectly to an unsuitable problem can lead to incorrect or inefficient solutions.
4. **Lack of Backtracking**: Greedy algorithms lack a backtracking mechanism to reconsider previous choices.

3.3 Backtracking

Backtracking is a systematic algorithmic technique used to solve problems by incrementally building a solution candidate and exploring different paths until a valid solution is found or all possibilities have been exhausted.

It involves a depth-first search approach where choices are made at each step and backtracking is performed when a choice leads to an invalid or undesirable outcome.

Example Backtracking Approach

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. Constraint: Girl should not be on the middle bench.

Solution: There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.

All the possibilities are:

The following state space tree shows the possible solutions.

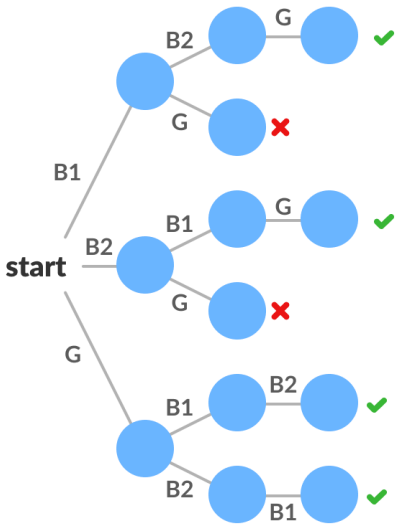


Fig: State tree with all the solutions

The terms related to the backtracking are:

- **Live node**: The nodes that can be further generated are known as live nodes.
- **E node**: The nodes whose children are being generated and become a success node.
- **Success node**: The node is said to be a success node if it provides a feasible solution.
- **Dead node**: The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Backtracking Algorithm Applications

- To find all Hamiltonian Paths present in a graph.
- Sum of subset
- N Queen problem.
- Maze solving problem.
- The Knight's tour problem.

4. Algorithm Analysis

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Algorithm analysis is the process of evaluating and understanding the performance characteristics of an algorithm. It involves studying the behavior of an algorithm in terms of its time complexity, space complexity, and other relevant factors to assess its efficiency and scalability.

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- **A Priori Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- **A Posterior Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We will learn about a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required 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 that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- **A variable part** is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

4.1 Best, Worst and Average Case Analysis

Best, worst, and average case analysis are methods used to analyze the performance of algorithms based on different scenarios:

1. **Best-case analysis**: This involves evaluating the algorithm's performance under the most favorable or optimal conditions. It determines the minimum amount of resources (time, space, etc.) required by the algorithm when the input is specifically tailored to produce the best outcome. The best-case analysis provides an understanding of the algorithm's lower bound or best possible efficiency.
2. **Worst-case analysis**: This involves evaluating the algorithm's performance under the most unfavorable or challenging conditions. It determines the maximum amount of resources required by the algorithm when the input is specifically designed to create the worst outcome. The worst-case analysis provides an understanding of the algorithm's upper bound or worst possible efficiency.
3. **Average-case analysis:** This involves evaluating the algorithm's performance on an average or expected input. It considers the distribution of inputs and calculates the average resource usage over all possible inputs. The average-case analysis provides a more realistic estimation of the algorithm's efficiency in practical scenarios.

By considering these different cases, we can gain insights into how an algorithm performs in various scenarios. The best-case analysis gives us a lower bound or optimistic view, the worst-case analysis provides an upper bound or pessimistic view, and the average-case analysis provides a more representative estimation. These analyses help in understanding the algorithm's behavior and making informed decisions regarding its suitability for different use cases.

4.2 Rate of Growth

Rate of growth is defined as the rate at which the running time of the algorithm is increased when the input size is increased.

Let us assume that you went to a shop to buy a car and a cycle. If your friend sees you there and asks what you are buying then in general we say buying a car. This is because. cost of a car is too big compared to cost of cycle (approximating the cost of cycle to the cost of a car).

Total Cost = cost_of_car + cost_of_cycle
Total Cost ≈ cost_of_car (approximation)

For the above example, we can represent the cost of car and cost of cycle in terms of function and for a given function ignore the low order terms that are relatively insignificant (for large values of input size, n). As an example in the below case, n^4 , $2n^2$, 100n, and 500 are the individual costs of some function and approximate it to n^4. Since, n^4 is the highest rate of growth.

$n^4 + 2n^2 + 100n + 500n \approx n^4$

Commonly used Rate of Growth.

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
log n	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in a unsorted array
n log n	Linear Logarithmic	Sorting n items by 'Divide and Conquer'
n^2	Quadratic	Shortest path between 2 nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

The rate of growth is important for assessing the efficiency and scalability of algorithms. Algorithms with slower rates of growth (e.g., logarithmic or linear) are generally considered more efficient and scalable than those with faster rates of growth (e.g., quadratic or exponential).

By understanding the rate of growth, we can compare different algorithms, estimate their performance, and make informed decisions when selecting the most suitable algorithm for a given problem or dataset size.

The growth rate could be categorized into two types: linear and exponential. If the algorithm is increased in a linear way with an increasing in input size, it is **linear growth rate**. And if the running time of the algorithm is increased exponentially with the increase in input size, it is **exponential growth rate**.

4.3 Asymptotic Notations- Big Oh, Big Omega and Big Theta

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e, as some sort of limit is taken)

Asymptotic notation is a **mathematical notation** used to describe the limiting behavior of functions as the **input size approaches infinity**. It provides a concise way to represent the growth rate of an algorithm's time or space complexity without getting into precise details.

The simplest example is a function $f(n) = n^2+3n$, the term 3n becomes insignificant compared to n^2 when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \rightarrow \infty$", and here is written symbolically as $f(n) \sim n^2$.

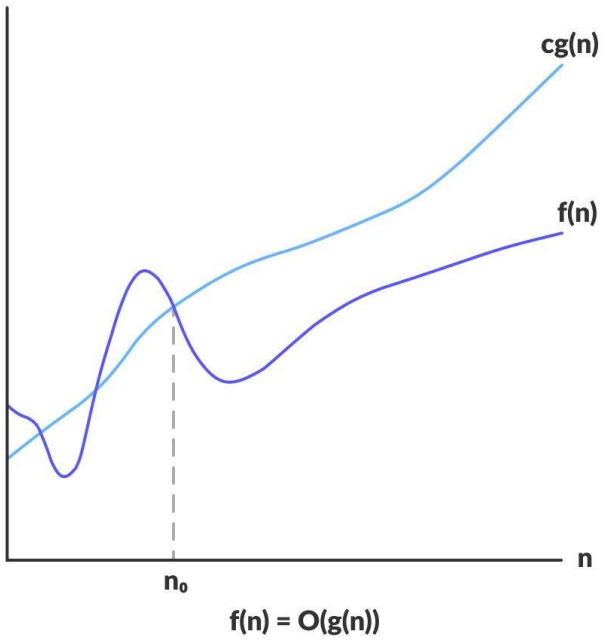
Asymptotic notations are used to write fastest and slowest possible running time for an algorithm.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- O − Big Oh
- Ω − Big omega
- θ − Big theta

The three most commonly used asymptotic notations are:

1. **Big O notation (O):** It represents the upper bound or worst-case scenario of an algorithm's time or space complexity. It describes the maximum growth rate of the algorithm's resource usage as the input size increases. For example, an algorithm with a time complexity of O(n^2) means that the runtime of the algorithm grows quadratically with the input size.



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 n_o such that:

$f(n) \leq c.g(n)$ for all $n \geq n_0$

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.

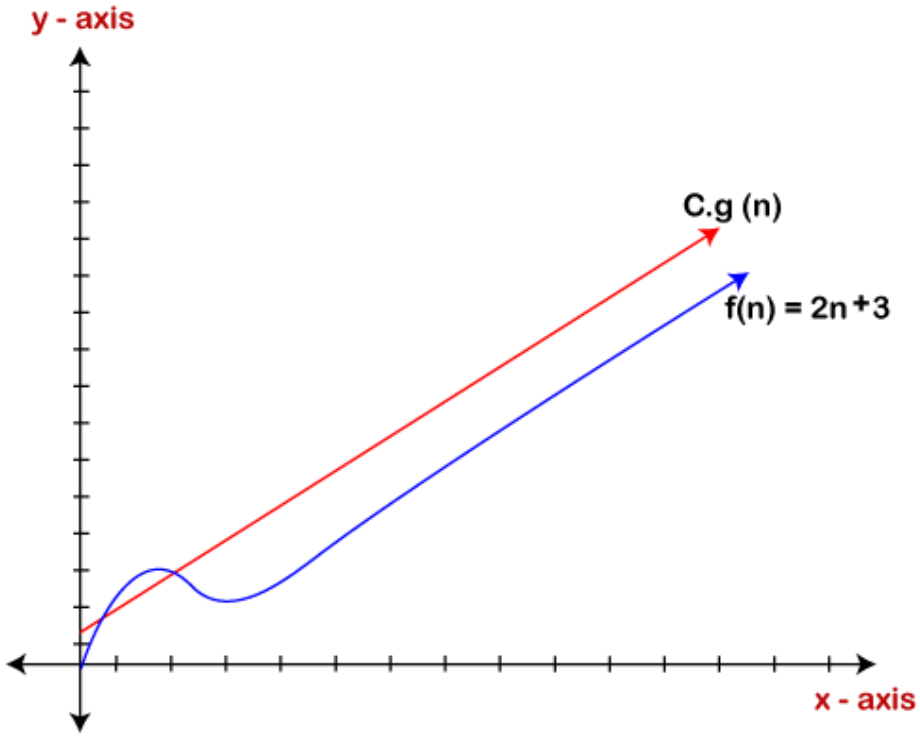
If n=2

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.

2. **Omega notation (Ω):** It represents the lower bound or best-case scenario of an algorithm's time or space complexity. It describes the minimum growth rate of the algorithm's resource usage as the input size increases. For example, an algorithm with a time complexity of Ω(n) means that the runtime of the algorithm grows at least linearly with the 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 $n \geq n_o$ and $c > 0$

   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)

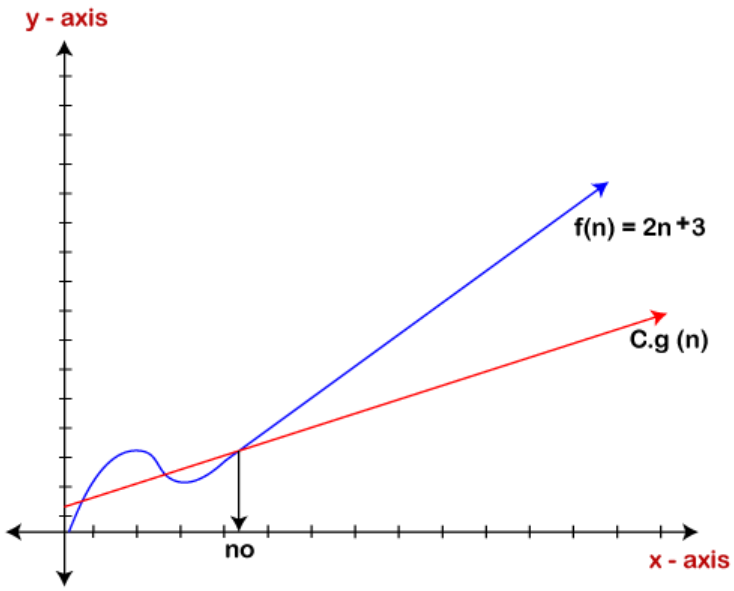
   To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

   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.

3. **Theta notation (Θ):** It represents both the upper and lower bounds of an algorithm's time or space complexity, providing a tight bound on the growth rate. It describes the growth rate of the algorithm's resource usage within a constant factor as the input size increases. For example, an algorithm with a time complexity of Θ(n) means that the runtime of the algorithm grows linearly with the input size.

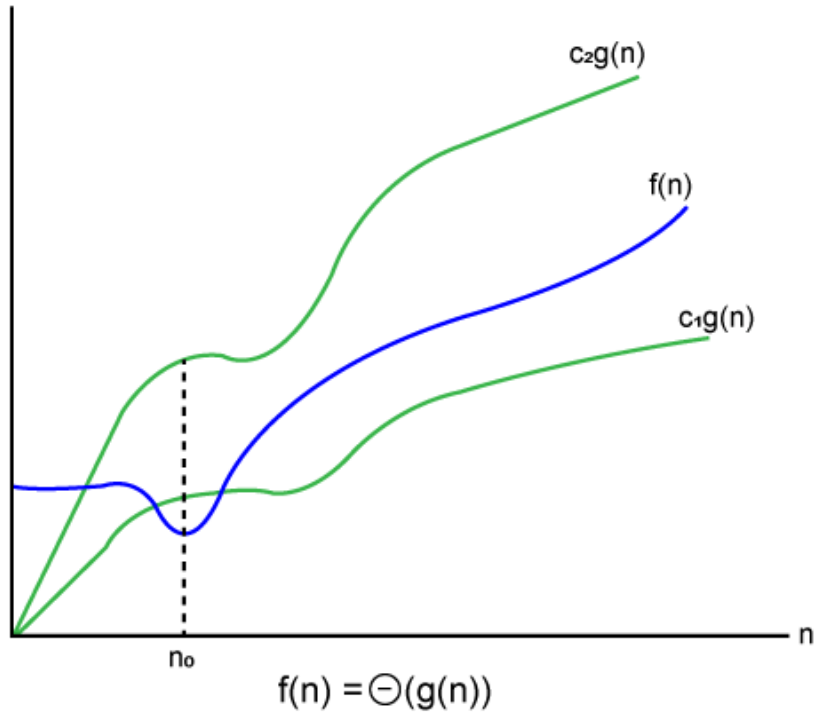
   Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

   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.