Unit - 1

Chapter

1 Introduction to Algorithms

Chapter Outline

1.1 What is an Algorithm ?

To make a computer do anything, we have to write a computer program. To write a computer program, we have to tell the computer, step by step, exactly what we want it to do. The computer then "executes" the program, following each step mechanically, to accomplish the end goal.

When we are telling the computer what to do, we also get to choose how it's going to do it. That's where computer algorithms come in. The algorithm is the basic technique used to get the job done. Every problem solution starts with a plan. That plan is called an algorithm. Let's follow an example to help get an understanding of the algorithm concept.

Let's say that one of our a friend arriving at the airport, and our friend needs to get from the airport to our house. Here are four different algorithms that we might give our friend for reach our home from airport:

Algorithm 1 : Taxi Algorithm	**Algorithm 2 : Call Me Algorithm**
1. Go to the taxi stand. 2. Get in a taxi. 3. Provide address to the driver.	1. When plane arrives, call my cell phone. 2. Meet me at the entrance of the arrival gate.
Algorithm 3 : Rent a Car Algorithm	**Algorithm 4: Bus Algorithm**
1. Take the shuttle from arrival gate to the rental car place. 2. Rent a car 3. Follow the google maps directions to reach my home.	1. Take the shuttle from arrival gate to the bus terminal. 2. Catch bus number BIA 175. 3. Get down at Jayangar 4th Block. 4. Call my cell phone.

All four of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. We choose the algorithm based on the circumstances.

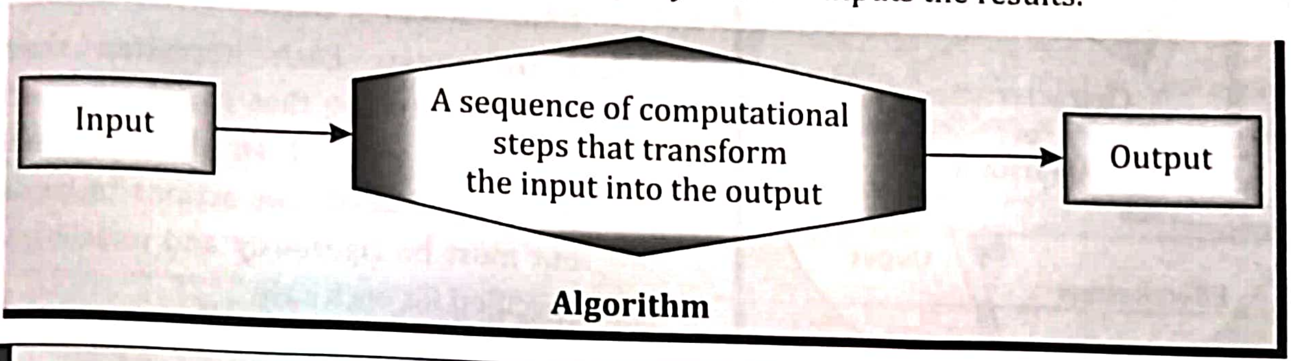
In computer programming, there are often many different algorithms to accomplish any given task. Each algorithm has advantages and disadvantages in different situations.

Problems that can be solved through the computer may range in size and complexity. Since the computer does not possess any common sense and cannot make any unplanned decisions, the problem whether it is simple or complex has to be broken into well defined set of solution steps. It should be remembered that computer don't solve problems rather they are used to implement the solutions to problems. So, the well defined set of steps for solving a problem is called an 'Algorithm'.

The word algorithm comes from the name of the 9th century Persian Muslim mathematician Abu Abdullah Muhammad ibn Musa Al-Khwarizmi and he developed systematic approach to problem solving by breaking it down into step-by-step Instructions.

In layman's language, ***an algorithm can be defined as a step-by-step procedure for accomplishing a task***. We use algorithms every day but we often do not explicitly think about the individual steps of the algorithm. For example, starting a car, logging into computer or laptop, or following a recipe for cooking a food dish, are all accomplished using an algorithm, a step-by-step series of actions.

We can also view an algorithm as a tool for solving a well-specified computational problem. For example, a search engine is an algorithm that takes a search query as an input and searches its database for items relevant to the words in the query. It then outputs the results.


Input
A sequence of computational steps that transform the input into the output
Output
Algorithm

Various Definitions of Algorithms

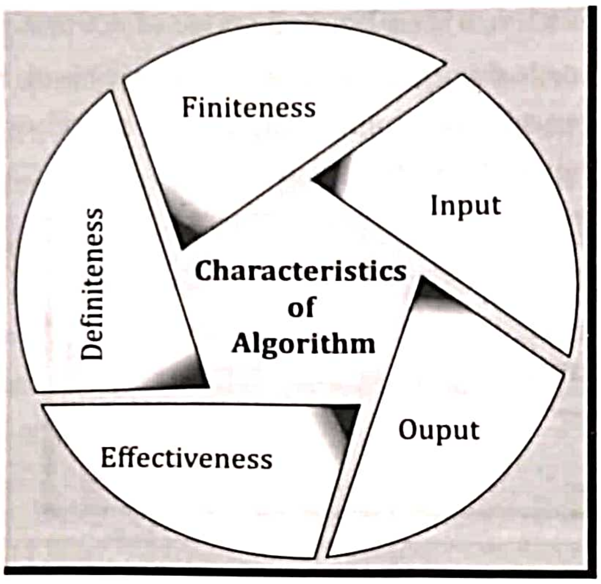
- An algorithm is a step-by-step procedure for performing some task in a finite amount of time.
- An algorithm is a sequence of unambiguous instructions for solving a problem. It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.
- An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.
- An algorithm is a well-structured computational procedure that takes some values as input and produces some values as output.
- An algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input and produces a desired output.

Examples of an Algorithm

An algorithm for making a telephone call.	An algorithm to find the average of three numbers.
Step 1 : Start Step 2 : Pick up the receiver Step 3 : Listen for the dial tone. Step 4 : Dial the phone number Step 5 : When someone answers talk Step 6 : Say "BYE" Step 7 : Hang up the receiver Step 8 : Stop	This algorithm is used to find the average of three numbers. Let a, b, c are the three numbers, find the average and store the result in the variable called as 'Average'. Step 1 : Start Step 2 : Read three numbers a, b, c Step 3 : Average = (a + b + c) /3 Step 4 : print Average Step 5 : Stop.

1.2 Characteristics of Algorithm

The following are the five important characteristics (features) of algorithm.



1. **Finiteness:** An algorithm must always terminate after a number of steps. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
2. **Definiteness:** Each operation must be definite meaning that it must be perfectly clear. Each step of an algorithm must be precisely defined. The actions to be carried out must be rigorously and unambiguously specified for each case.
3. **Input:** An algorithm has zero or more "inputs" quantities that are given to it initially before the algorithm begins, or dynamically as the algorithm runs. These inputs are taken from specified set of objects. These inputs are extremely supplied to the algorithm.
4. **Output:** An algorithm has one or more "output" quantities that have a specified relation to the inputs. An algorithm produces at least one or more outputs.
5. **Effectiveness:** Each operation should be effective i.e., the operation must be able to carryout in finite amount of time. An algorithm is generally expected to be "effective", in the sense that its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by some one using pencil and paper.

Important Note:

The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

Just like it is an important plan before working. It is important to define the algorithm before coding.

1.3 The Role of Algorithms in Computing

An algorithm is a well-defined procedure that allows a computer to solve a problem. Another way to describe an algorithm is a sequence of unambiguous instructions. In fact, it is difficult to think of a task performed by a computer that does not use algorithms.

We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

Let us consider an example to sort the numbers in ascending order. Here is how we formally define the sorting problem.

Sorting Problem

Problem : Sorting

Input : A sequence of n keys $a_1, a_2, a_3, \ldots a_n$

Output : The reordering of the input sequence such that $a_1 \leq a_2 \leq \cdots \leq a_{n-1} \leq a_n$.

For example, given the input sequence {30, 50, 40, 20, 10}, a sorting algorithm returns as output the sequence {10, 20, 30, 40, 50}. Such an input sequence is called an **instance of the sorting problem.**

An instance of sorting might be an array of names, like {Rama, Anitha, Indu, Srikanth, Sita}, or a list of numbers like {30, 50, 40, 10, 20}. Determining that we are dealing with a general problem is our first step towards solving it.

An algorithm is a procedure that takes any of the possible input instances and transforms it to the desired output.

What is an instance of a problem?

In general, **an instance of a problem** consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example : The input values like {30, 50, 40, 20, 10} in a sorting algorithm,

How to decide which algorithm is best suited?

1. It depends on how efficient the algorithm when higher values of input is given. For example, sorting algorithm might be efficient for smaller values of n such as 10 or 100 or 1000 numbers but many not be efficient for large values of n such as 1 million or 10 million numbers.
2. The possible restrictions/constraints on the input values.
3. The architecture of the computer and the kind of storage devices to be used.
4. The correctness of the algorithm

What is correctness of an algorithm?

An algorithm is said to be correct if, for every instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.

1.4 Practical Applications of Algorithms

Algorithms are heart of computing. We can find several algorithms working to solve our daily life problems like internet, social media networks, GPS applications, Google Maps, e-commerce platforms like amazon and flipkart, youtube recommendations and so on. There are many applications of algorithms in various domains. Lets look at few of the applications of algorithms.

1. **Internet :** The Internet without which it is difficult to imagine a day is the result of clever and efficient algorithms. With the help of these algorithms, various sites on the Internet are able to manage and manipulate this large volume of data. Finding good routes on which the data will travel and using search engine to find pages on which particular information is present.

2. **The Human Genome Project :** Another great milestone is the Human Genome Project which has great progress towards the goal of identification of the 100000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up the human DNA, storing this huge amount of information in databases, and developing tools for data analysis. Each of these steps required sophisticated and efficient algorithms.
3. **e-Commerce :** The day-to-day electronic commerce activities are hugely dependent on our personal information such as credit/debit card numbers, passwords, OTPs and so on. The core technologies used include public-key cryptocurrency and digital signatures which are based on numerical algorithms and number theory.
4. **PageRank:** Google's search engine uses a highly effective algorithm called PageRank to find the best matches for search terms. PageRank decides which pages are listed first when you search for something. This algorithm is highly sophisticated and played an important role in Google Search success.
5. **Weather Forecasting :** Weather forecasting algorithms to model weather patterns and make predictions.
6. **Linear Programing :** An oil company may wish to know where to place its wells in order to maximize its expected profit. An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met. An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming.
7. **Shortest Path Algorithm :** Transportation companies may have financial interest in finding shortest path through a road or rail network because taking shortest path result in lower labour or fuel costs.
8. **Other Important Applications of Algorithms :** Speech recognition, image processing, assigning fastest pick up to cab drivers using Hungarian algorithm, Facebook's friend suggestion algorithm, Resource allocation in operating system, product recommendation in e-commerce sites like amazon, duck worth Lewis method in cricket, machine learning algorithms, solving puzzles like crosswords and sudoku.

1.5 Algorithms as a Technology

We must understand that computer programs adopt different algorithms that run on computer hardware that has a processor & memory, and these components have limitations. A processor is not infinitely fast, and the memory we have is not free. They are bounded resources. They must be used wisely, and a good algorithm that is efficient in terms of time complexities and space complexities.

Despite the fact that modern processors are incredibly fast and a memory is cheap, we still have to study algorithms, design them so as to see if the solution terminates and does so with a correct result.

1.5.1 Efficiency of an Algorithm

Efficiency considerations for algorithms are inherently tied in with the design, implementation,and analysis of algorithms. Every algorithm must use up some of a computer's resources to complete its task. The resources most relevant in relation to efficiency are central processor time (CPU time) and internal memory. Because of the high cost of computing resources it is always desirable to design algorithms that are economical in the use of CPU time and memory.

Algorithms that solve the same problem can differ enormously in their efficiency. Generally speaking, we would like to select the most efficient algorithm for solving a given problem.

1.5.2 Why Analysing Efficiency is Important?

Suppose we would like to run two different sorting algorithms on two different computers A and B, where computer B is 1000 times slower than computer A. For comparing the performance, let us consider of running the slower sorting algorithm **Insertion Sort** on faster computer A and running the faster sorting algorithm Merge Sort on slower computer B.

Efficiency Comparison of Two Computers

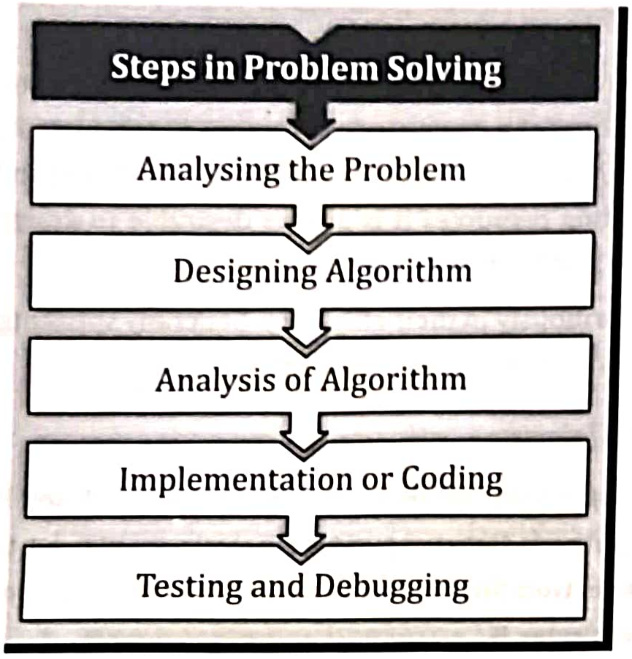
Computer A	Computer B
Execute 10 billion Instruction/Second	Execute 10 million Instruction/Second
Running Insertion Sort to Sort an array	Running Merge Sort to Sort an array
Running Time of the Insertion sort = n^2	Running Time of the Merge sort = n logn
Input Size = 10 million Numbers	
Time taken by Computer A ~ 6 Hours	Time taken by Computer B ~ 25 minutes
Input Size = 100 million Numbers	
Time taken by Computer A ~ 23 Days	Time taken by computer B ~ 4 Hours

What difference do we observe? Computer B is taking much less time than computer A, if the input size is large. This gap will increase further if we increase the input size. This would be one of the reason for analysing the efficiency of an algorithm.

It means that using efficient algorithms can be even more important than building faster computers: more efficient thinking beats more efficient hardware! And this means that algorithms are definitely worth studying.

1.6 Steps in Problem Solving

A computer cannot solve a problem on its own. One has to provide step by step solutions of the problem to the computer. In fact, the task of problem solving is not that of the computer. It is the programmer who has to write down the solution to the problem in terms of simple operations which the computer can understand and execute. The following figure shows the steps involved in problem solving.



1. **Analysing the Problem :** It is important to clearly understand a problem before we begin to find the solution for it. If we are not clear as to what is to be solved, we may end up developing a program which may not solve our purpose. Thus, we need to read and analyse the problem statement carefully in order to list the principal components of the problem and decide the core functionalities that our solution should have. By analysing a problem, we would be able to figure out what are the inputs that our program should accept and the outputs that it should produce. We should also understand the constraints on input values and output values.

2. **Designing Algorithm:** It is essential to device a solution before writing a program code for a given problem. The output of this step is an algorithm. We start with a tentative solution plan and keep on refining the algorithm until the algorithm is able to capture all the aspects of the desired solution. The design of an algorithm depends mainly on the problem and chosen design technique. There can be more than one algorithm to solve the same problem, and the choice between them will be decided by their effectiveness.

3. **Analysis of Algorithm :** An essential task in algorithm development is proving its correctness. One possible method is to run a number of data sets as inputs and compare the results against expected output. Another important task of analysing algorithms is to find out the time complexity and space complexity of an algorithm to prove that an algorithm is efficient.

4. **Implementation or Coding :** After finalising the algorithm, we need to convert the algorithm into the format which can be understood by the computer to generate the desired solution. Different programming languages can be used for writing a program. We must convert each step of the algorithm into one or more statements in a programming language such as C, C++, and Java etc.

5. **Testing and Debugging:** The program created should be tested on various parameters. The program should meet the requirements of the user. It must respond within the expected time. It should generate correct output for all possible inputs. In the presence of syntactical errors, no output will be obtained. In case the output generated is incorrect, then the program should be checked for logical errors by debugging the program. Debugging is the process of identifying and correcting or removing the Bugs (errors)

1.7 Designing Algorithms

There are many ways to design algorithms for a given problem, The following are some of the popular design approaches.

→ Linear search

1. **Brute Force Algorithm :** This is the most basic and simplest type of algorithm. A Brute Force Algorithm is the straightforward approach to a problem It is just like iterating every possibility available to solve that problem. This type of algorithms are moreover used to locate the ideal or best solution as it checks all the potential solutions.

   Example: If there is a lock of 4-digit PIN. The digits to be chosen from 0-9 then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN. In the worst case, it will take 10,000 tries to find the right combination.

2. **Recursive Algorithms:** This type of algorithm is based on recursion. In recursion, a problem is solved by breaking it into sub-problems of the same type and calling own self again and again until the problem is solved with the help of a base condition.

   Example: Some common problem that is solved using recursive algorithms are Factorial of a Number, Fibonacci Series, Tower of Hanoi, tree traversals, depth first search for Graph, etc.

3. **Divide and Conquer Technique:** The "divide and conquer" technique involves solving particular problem by dividing it into one (or) more sub problems of smaller size, recursively solving each sub problem and then "merging" the solutions to the sub-problem to produce a solution to the original problem. The divide and conquer paradigm involves three steps at each level of recursion.

   ↳ Merge sort, quick sort

   1. **Divide:** Divide the problem into a number of sub problems.
   2. **Conquer:** Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, then solve the sub problem in a straightforward manner.
   3. **Combine:** Combine the solutions to the sub problems in to the solution for the original problem.

   Example : This technique is the basis of efficient algorithms for all kinds of problems, such as sorting (Quick Sort and Merge Sort), finding maximum and minimum numbers, multiplying large numbers etc.,

4. **Greedy Approach:** A greedy algorithm is a type of algorithm that is typically used for solving optimization problems. So whenever one wishes to extract the maximum in minimum time or with minimum resources, such an algorithm is employed.

   This is an algorithm paradigm that makes the best choice possible on each iteration in the hopes of choosing the best solution. It is simple to set up and has a shorter execution time. The result is a good solution but not necessarily the best one. The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close to the optimal solution.

   Example : Some common problems that can be solved through the Greedy Algorithm are Prim's Algorithm, Kruskal's Algorithm, Huffman Coding, etc.

5. **Dynamic Programming:** A dynamic programming algorithm works by remembering the results of a previous run and using them to arrive at new results. Such an algorithm solves complex problems by breaking it into multiple simple subproblems, solving them one by one and storing them for future reference and use.

   Example : The following problems can be solved using Dynamic Programming algorithm Knapsack Problem, Weighted Job Scheduling, Floyd Warshall Algorithm, Dijkstra Shortest Path Algorithm, etc.

6. **Backtracking Algorithms:** Backtracking algorithm is one that entails finding a solution in an incremental manner. There is often recursion/ repetition involved and attempts are made to solve the problem one part at a time. At any point, if one is unsuccessful at moving forward one backtracks and comes back to start over and find another way of reaching the solution. So backtracking algorithm solves a subproblem and if and when it fails to solve the problem, the last step is undone and one starts looking for the solution again from the previous point.

Example : Some common problems that can be solved through the Backtracking Algorithm are Hamiltonian Cycle, M-Coloring Problem, N Queen Problem, Rat in Maze Problem, etc.

1.8 Qualities of Good Algorithm

There are usually many ways to solve any given problem. In computing, we are generally concerned with "good" solutions to problems. Good algorithms usually possess the following qualities and capabilities:

Qualities of Good Algorithm

1. They are simple but powerful and general solutions.
2. They can be easily understood by others, that is, the implementation is clear and concise without being "tricky".
3. They can be easily modified if necessary.
4. They are correct for clearly defined situations.
5. They are able to be understood on a number of levels.
6. They are economical in the use of computer time, computer storage and peripherals.
7. They are documented to be used by others who do not have a detailed knowledge of their inner workings.
8. They are not dependent on being run on a particular computer or particular programming language.
9. They are able to be used as a sub-procedure or functions for other problems.

The above qualitative aspects of a good algorithm are very important but it is also necessary to provide some quantitative measures to evaluate the "goodness" of an algorithm. Quantitative measures are valuable in that they can give us a way of directly predicting the performance of an algorithm and of comparing the relative performance of two or more algorithms that are intended to solve the same problem. This can be important because the use of an algorithm that is more efficient means a saving in computing resources which translates into a saving in time and money

1.9 Analysing Algorithms

1.9.1 What is Analysis or Performance Analysis of Algorithms?

Algorithm analysis refers to the task of determining the computing time and storage space requirement of an algorithm. It is also known as ***performance analysis*** or ***efficiency*** of an algorithm which enables us to select an efficient algorithm. The general idea is to take a particular algorithm and to determine its quantitative behaviour, occasionally we also study whether or not an algorithm is optional in some sense.

When we have a problem to solve, there may be many algorithms available. We would obviously like to choose the best. The selection of best algorithm is possible by analysing the algorithms in proper manner.

? What is Analysis of an Algorithm?

Analysis of algorithms or performance analyse is refers to the task of determining how much computing time and storage an algorithm requires. This is a challenging area which sometimes require great mathematical skills. An important result of this study is that it allows to make quantitative judgments about the value of one algorithm over another. Another result is that it allows to predict whether the software will meet any efficiency constraints that exist. Analysis of an algorithm is a process of making evaluative judgement about algorithms. And also Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

Generally, the performance of an algorithm depends on the following elements.

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all remaining elements.

Based on this information, performance analysis of an algorithm can also be defined as follows.

"Performance analysis of an algorithm is the process of calculating space required by that algorithm and time required by that algorithm".

We can analyse an algorithm by two ways.

1. By checking the correctness of an algorithm
2. By measuring time and space complexity of an algorithm

   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.

1.9.2 Priori and Posteriori Analysis

To compute the analysis of algorithm, two phases are required

1. Priori Analysis
2. Posteriori Analysis

1. Priori Analysis

"Priori" means "before". Hence Priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This is done usually by the algorithm designer. Algorithm complexity is determined in this phase.

In this we obtain a function which bounds the algorithms computing time. Suppose there is some statement and we wish to determine the total time that statement will spend for the execution, given some initial state of input data. This requires essentially two items of information. They are:

a. The statements frequency count.
   i.e., the number of times the statement will be executed.

b. The time taken for one execution.

The product of these two numbers is the total time.

Since the time per execution depends on both i.e., the machine being used and the programming languages used together with its compiler, a priori analysis limits itself to determine the frequency count of each statement.

Priori analysis of computing time ignores all of the factors, which are machine or programming language dependent and only concentrates on determining the order of the magnitude of the frequency of execution of the statements.

The notation used in the priori analysis are **Big-oh (O), Omega (Ω), Theta (θ)** and **Small-oh(o).**

2. Posteriori Analysis

"Posterior" means "after". Hence Posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness, space required, time consumed etc.

In this we will collect the actual statistics about the algorithm, conjunction of the time and space while executing. Once the algorithm is written it has to be tested. Testing a program consists of two major phases.

a. **Debugging :** It is the process of executing programs on sample data sets that determine whether we get proper results. If faulty results occurs it has to be corrected.

b. **Profiling :** It is the process of executing a correct program on actual data sets and measuring the time and space it takes to compute the results during execution. The actual time taken by the algorithm to process the data is called profiling.

```
Procedure: Profiling
{
        1. Read data
        2. Time (t1)
        3. Process (data)
        4. Time (t2)
        5. Write (time = t2 - t1)
}
```

Differences between Priori Analysis and Posteriori Analysis

Priori Analysis	Posteriori Analysis
1. Analysis is the process of determining how much computing time and storage an algorithm will require..	1. Profiling is the process of executing the correct program on data sets and measuring the time and space it takes to compute the results.
2. This is independent of machine programming language and won't involve the execution of program.	2. This is dependent on machine, programming language and the compiler used.

3. It will give approximate answer.	3. It will give exact answer.
4. It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.	4. It doesn't use asymptotic notations to represent the time complexity of an algorithm.

1.9.3 Complexity of Algorithms

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

Algorithm complexity can be further divided into two types: time complexity and space complexity.

1. Space required to complete the task of that algorithm **(Space Complexity)**.
2. Time required to complete the task of that algorithm **(Time Complexity)**

1.9.3.1 Space Complexity

It indicates the amount of temporary storage required for running the algorithm. i.e., "the amount of memory needed by the algorithm to run to completion".

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes.

1. Memory required to store program instructions
2. Memory required to store constant values
3. Memory required to store variable values
4. And for few other things

Definition: Space Complexity

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for three reasons. They are as follows.

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

Note:

We consider only Data Space and ignore Instruction Space and Environmental Stack while calculating space complexity. It means that we calculate only the memory required to store Variables, Constants, Structures, etc.,

Understanding Space Complexity

Let us consider Rama and Sita have written an algorithm for binary search.

Rama's algorithm: It takes 30 bytes of memory to execute.

Sita's algorithm: It takes 50 bytes of memory to execute.

Which is space efficient?

Of course it's Sita's algorithm.

To calculate the space complexity, we must know the memory required to store different data type values (according to the compiler). For example, the C Programming Language compiler requires the following.

1. 4 bytes to store Integer value,
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value,
4. 8 bytes to store Double value

In most cases, we do not count the storage required for the input / output as part of the space complexity. This is so, because the space efficiency is used to compare different algorithms for the same problem, in that case the input/output requirements are fixed. Also, we cannot move without input or output, and we want to count only the storage that may be served. We also do not count the storage required for the program itself, since it is independent of the size of the input. The space needed by an algorithm consists of the following components.

a. ***The fixed static part*** that is independent of the characteristics (eg: number size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for code), space for simple variables, space for constants and fixed size component variables. Let C_p be the space required for the code segments of a program (i.e., static part).

b. ***The variable dynamic part,*** that consists of the space needed by component variables whose size is dependent on the particular problem instance at runtime being solved, the space needed by referenced variables, and the recursion stack space (depends on instance characteristics). Let S_p be the space required for the dynamic part.

The overall space requirements for an algorithm is the sum of both the fixed static part storage and variable dynamic part storage. If P be a program, then space required for program P will be denoted by S(P).

$$\boxed{S(p) = C_p + S_p}$$

Example 1 Finding the Sum of Array Elements

```
int square(int a)
{
    return a*a;
}
```

In Example 1, it requires 4 bytes of memory to store variable 'a' and another 4 bytes of memory is used for return value.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be **Constant Space Complexity**.

$\therefore$ $S(P) = C_P + S_P$

$S(P) = 8 + 0$

$S(P) = 8$

Hence, space complexity for the above given program is O(1), or constant.

total, i 4 × 2 + 4 × n

Example 2 Finding the Sum of Array Elements

```
int ADD (int X[], int n)
{
    int total = 0, i;
    for (i = 0; i < n; i ++)
        total = total + X [i];
    return total;
}
```

In Example 2, the space needed by X is the space needed by variables of type array of integer numbers. This is at least containing n elements to be summed. Here the problem instance is characterized by n.

The code requires, 'n*2' bytes of memory to store array variable 'X[]' , 4 bytes of memory for integer parameter 'n' , 8 bytes of memory

For local integer variables 'total' and 'i' (4 bytes each) and 4 bytes of memory for return value.

$S(P) = C_P + S_P = 4 * 4 + n*2 = 16 + 2n$

$S(P) = 2n + 16$

That means, totally it requires '2n+16' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be **Linear Space Complexity.** If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be **Linear Space Complexity**.

The space complexity for the above code is **O(n)** or **linear.**

Example 3 Finding the Sum of Three Numbers

```
#include<stdio.h>
void main()
{
    int x,y,z,sum;
    printf("Enter the three numbers");
    scanf(" %d %d %d",&x,&y,&z);
    sum = x + y + z;
    printf("The sum = %d",sum);
}
```

In Example 3, there are no instance characteristics and the space needed by x, y, z and sum is independent of instance characteristics. The space for each integer variable is 4. We have 4 integer variables and space needed by x, y, z and sum are 4 × 4 = 16 bytes.

$\therefore$ $S(P) = C_P + S_P$

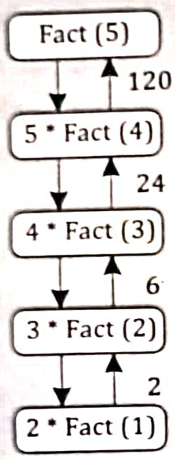
$S(P) = 16 + 0$

$S(P) = 16$

Hence, space complexity for the above given program is **O(1)**, or **constant.**

Example 4 Finding the Factorial of a Number

```
int Fact(int n)
{
    if(n < = 1)
        return 1;
    else
        return(n * Fact(n-1));
}
```



The depth of recursion = 5

$\therefore$ The depth of recursion = n

In Example 4, The program is recursive and stack space includes space for the formal parameter, local variables and the return address. Here the problem instance are characterized by 'n'.

Space occupied by n = 4 bytes

Memory for return value = 4 bytes

Total space= 8 * depth of recursion

Total space= 8 * n = 8n

The space complexity for the above code is O(n) or **linear**.

1.9.3.2 Time Complexity

The amount of time needed to run the program is termed as time efficiency or time complexity.

The total time taken by a program is the sum of the compile time and runtime. The compile time does not depend on the instance characteristics and it can be assumed as a constant factor so we concentrate on the runtime of a program. Let this runtime is denoted by t_p (instance characteristic), then

$$t_p(n) = t_a\,ADD(n) + t_s\,SUB(n) + t_m\,MUL(M) + - - - -$$

Where n indicates the instance characteristics and t_a, t_s, t_m - - - denote the time needed for an addition, subtraction, multiplication, and so on. ADD, SUB, MUL - - - represent the functions and they are performed when the code for the program is used on an instance characteristic 'n'.

Obtaining such an exact formula is itself an impossible task, since the time needed for an addition, subtraction, multiplication and so on, often depends on the numbers being added, subtracted, multiplied and so on.

The value of $t_p(n)$ for any given 'n' can be obtained only experimentally.

Definition: Time Complexity

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Understanding Time Complexity

Let us consider there's a small piece of code that takes one second on a slow computer. This piece of code will be used on a list of items for processing; something like an array waiting to be searched or sorted.

If you have designed an algorithm that is O(1), it means,

If the array contains just a single item, it will take 1 second.

If array has 10 items, it will still take 1 second to finish with all of them.

If it has 100, again 1 second only.

The algorithm we designed is great even for the large arrays.

Let's proceed to quite larger and practical time complexities. Now we have created a similar algorithm, but in O(n) this time.

If array has one item, it will take 1 second.

If we have 10 items, it will take 10 seconds.

Now it we have 100 items, it will take 100 seconds.

What will happen to the longer lists?

We can also create many algorithms that are of $O(n^2)$order.

Again, to process a single item, it will take 1 second.

to process 10 items, it will take 100 seconds to process the whole array.

And what if we have 100? It'll take 10000 seconds.

In practical cases we may have really big arrays containing millions of items. Such algorithms may not be an efficient.

So, when we design an algorithm, we should consider time and space efficiencies.

Understanding Running Time

Suppose you developed a program that finds the shortest distance between two major cities You showed the program to another friend and he/she asked you "What is the running time of your program?". You answered promptly and proudly "Only 3 seconds". It sounds more practical to say the running time in seconds or minutes but is it sufficient to say the running time in time units like seconds and minutes? Did this statement fully answer the question? The answer is NO. Measuring running time like this raises so many other questions like

What's the speed of the processor of the machine the program is running on?

What is the size of the RAM?

What is the programming language?

How experience and skillful the programmer is?

In order to fully answer your friend's question, you should say like "My program runs in 3 seconds on Intel Core i5 8-cores 4.7 GHz processor with 8 GB memory and is written in Dev C++". Who would answer this way? Of course, no one. Running time expressed in time units has so many dependencies like a computer being used, programming language, a skill of the programmer and so on. Therefore, expressing running time in seconds or minutes makes so little sense in computer programming.

You are now convinced that "seconds" is not a good choice to measure the running time. Now the question is how should we represent the running time so that it is not affected by the speed of computers, programming languages, and skill of the programmer? In another word, how should we represent the running time so that we can abstract all those dependencies away?. The answer to the question is simple which is "input size". To solve all of these dependency problems we are going to represent the running time in terms of the input size. If the input size is n (which is always positive), then the running time is some function f of n. i.e.

Running Time=f(n)

The functional value of f(n) gives the number of operations required to process the input with size n. So the running time would be the number of operations (instructions) required to carry out the given task.

Function f(n) is monotonically non-decreasing. That means, if the input size increases, the running time also increases or remains constant. Some examples of the running time would be n^2+2n, n^3, $3n$, 2^n, log n, etc. Having this knowledge of running time, if anyone asks you about the running time of your program, you would say "the running time of my program is n^2 (or 2n, nlogn etc)" instead of "my program takes 3 seconds to run".

The running time is also called a time complexity

Measuring an Input's Size

The input size is denoted by 'n' and we use 'n' in most of the algorithms. In searching and sorting **n** indicates the number of array elements, in matrix manipulation n indicates the matrix order, in polynomials **n** indicates the degree and in travelling sales man problem n indicates the number of cities. Therefore the input size **n** is very much important in analyzing the algorithm.

The analysis of an algorithm will be focussed on input size **'n'**. Some algorithms require more than one parameter to indicate the size of their inputs. For example, the number of vertices and the number of edges for algorithms on graphs represented by adjacency linked lists.

Finally, we can say that it is logical to investigate an algorithm's efficiency as a function of some parameter **n** indicating the algorithm's input size.

Units for Measuring Running Time

We cannot measure the running time by seconds, milliseconds, and so on because such a measurement depends on the type of computer, compiler and the program. We would like to have a metric that does not depend on these extraneous factors.

Following are the some of the methods of computing the time efficiency of algorithms.

- Operation Counts
- Step Counts
- Asymptotic notations (Mathematical Analysis)

1. Operations Counts

Consider an algorithm 'A' with 'n' size of the input data. The time and space are the two main measures for the efficiency of the algorithm. In operation counts, the time is measured by counting the number of **basic operations** or **key operations.**

The **basic operations** are defined that the time for the other operations is much less than or almost proportional to the time for the basic operations.

Example

Code	Description
`A = a * b;`	This code takes 1 unit of time

`for (i = 0; i < n; i ++)`
`  a = a + 1;`
	This code takes 'n' units of time because, it executes for n times.

`for (j = 0; i < n; i ++)`
`   for (i = 0; j < n; j ++)`
`      printf (" Hello");`
	This code takes 'n^2' units of time

The operation count method concentrates on certain important basic operations like multiplications, for loop or while loop where it takes considerably more time than any other operations in an algorithm.

2. Step Counts

In step counts method, we attempt to find the time spent in all parts of the program. A step is any computational unit that is independent of the selected characteristics.

Example 1

```
i.    x = a + b
                                          Step Count = 1

ii.   for (i = 1; i < = n; i++)
          x = a + b
                                          Step Count = n

iii.  for (i = 1; i < = n; i++)
          for (j = 1; j < = n; j++)
              x = a + b
                                          Step Count = n²
```

Example 2 **Finding the Sum of all Array Elements**

```
int sum(int a[], int n)
{
     int i, sum = 0;
     sum_count++;
     for (i = 0; i < n; i++)
     {
          for_count++;
          sum = sum + a [i];
          assign_count++;
     }
     for_count ++;
     return_count ++;
     return sum;
}
```

The step count for '**sum**'	= 1
The step count for '**for**' statement	= n + 1
The step count for '**assignment**'	= n
The step count for '**return**'	= 1
Total steps	**= 2n + 3**

Here we consider the everystatement rather than basic operations. The sum initialization takes 1 unit of time, the for loop executes n+1 times, the assignment statement executes n times and the return statement takes 1 unit of time.

1.9.3.3 Orders of Growth

The time complexity of an algorithm is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. Let T(n) be the complexity function with input size 'n'. The values of T(n) increases when 'n' value increases and T(n) value decreases when 'n' value decreases. Therefore, the complexity function is directly proportional to the instance characteristics 'n'.

Function	Name
1	constant
log n	logarithmic
n	linear
n log n	linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

Assume that algorithm P has complexity O(n) and algorithm Q has complexity $O(n^2)$. We can assert that algorithm P is faster than algorithm Q for sufficiently large n. Since here ($n < n^2$) and we can say that algorithm P is faster than algorithm Q. The most standard common computing time functions are shown below.

The order of growth is

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

To know how the various functions grow with "**n**", it is advised to study the following table.

n	log n	n log n	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296

It is evident from the above table that the function 2^n grows very rapidly with n. In fact, if an algorithm needs 2^n steps for execution, then when n = 32, the number of steps needed is approximately 4.2×10^9. Therefore we may conclude that the utility of algorithms with exponential complexity is limited to small n.

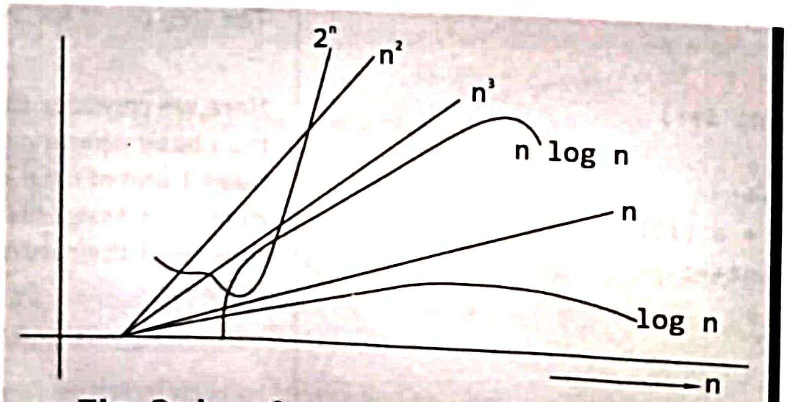


Fig: Order of growth of some functions

Understanding O(1) → Constant Time

O(1) means that it takes a constant time to run an algorithm, regardless of the size of the input.

Bookmarks are a great example of O(1) in real word. Bookmark in a book allow a reader to find the last page that we read in a quick, efficient manner. It doesn't matter if we are reading a book that has 30 pages or a book that has 1000 pages. As long as we are using a bookmark, we will find that last page in a single step.

In programming, a lot of operations are constant. Some examples include math operations, accessing an array via the index, function returing a value, pushing and popping on a stack, insertion and removal from a queue etc., Regardless of the size of n, all of these operations will take a constant amount of time.

Understanding O(n) → Linear Time

O(n) means that the runing time increases at the same pace as the input size 'n'.

Reading a book is an real world example of linear time. Let's assume that it takes exactly 1 minute to read a single page of a large print book. Given that, a book that has 50 pages will take 50 minutes to read. Likewise, a book that has 500 pages will take 500 minutes of reading time. We might not read all the pages but worst-case reading time is 500 minutes for a 500 page book.

In programming, one of the most common linear-time operations is traversing an array. In this, single loop iterating from 1 to n is an example of linear time.For example, finding the sum of n elements.

Understanding O(n^2) → Quadratic Time

O(n^2) means that the calculation runs in quadratic time, which is the squared size of the input size 'n'.

In programming, many of the more basic sorting algorithms have a worst-case run time of O(n^2). For example, Bubble Sort, Insertion Sort, Selection Sort etc.,

Generally speaking (but not always), considering two nested loops is typically a good indicator that the piece of code has a run time of O(n^2). Similarly three nested loops would indicate a run time of O(n^3).

Understanding O(log n) → Logarithmic Time

O(log n) means that the running time grows in proportion to the logarithm of the input size 'n', meaning that the run time barely increases as we exponentially increase the input size 'n'.

Finding a word in a physical dictionary by halving sample size is an excellent example of how logarithmic time works in the "real world." For example, when looking for the word "computer," we could open the dictionary precisely in the begging. Once we determine that "c" is in the first half of the book, then we can dismiss all of the pages in the second half. We then repeat the same process. By following this process, we would cut the number of pages we must search through in 1/2 every time until we find the word.

In programming, binary search operation is an example which runs in logarithmic time.

Understanding O(n log n) → Linearithmic Time

O(n log n), which is often confused with O(log n), means that the running time of an algorithm is linearithmic, which is a combination of linear and logarithmic complexity. Sorting algorithms that utilize a divide and conquer strategy are linearithmic, such as merge sort, quick sort, heapsort. When looking at time complexity, O(n log n) is between O(n^2) and O(n).

1.9.3.4 Worst Case, Best Case, and Average Case Efficiencies or Complexities

We have already studied that the algorithm efficiency is measured by the complexity function T(n) and n indicating the size of the algorithm's input. Several factors affect the running time of a program. Some, such as the compiler and computer used, are obviously beyond the scope of any theoretical model, so, although they are important, we cannot deal with them here. The other main factors are the algorithm used and the input to the algorithm. Typically, the size of the input is the main consideration.

Let us find the complexity function T(n) for certain cases.

1. **Worst Case** : It gives the maximum value of T(n) for any possible input.
2. **Best Case** : It gives the minimum value of T(n) for any possible input.
3. **Average Case** : It gives the expected value of T(n).

Complexity of average case of an algorithm is usually much more complicated to analyse.

Definitions: Worst Case, Best Case, Average Case Complexities

The **worst-case** efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size. Let us denote the worst-case efficiency of an algorithm by **T_{worst} (n)**. To determine the worst-case efficiency of an algorithm, we have to analyze the algorithm to see what kind of inputs makes the algorithm to run longest.

The **best-case** efficiency of an algorithm is its efficiency for the best-case inputs of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size. Let us denote the best-case efficiency of an algorithm by T_{best} (n). To determine the best-case efficiency of an algorithm, we have to analyze the algorithm to see what kind of inputs makes the algorithm to run fastest.

It is noticed that neither the worst-case nor the best-case yields the necessary information about the algorithm's behaviour on random input. The average-case efficiency provides that information.

Let us denote the **average-case** efficiency of an algorithm by **T_{avg} (n)**. We must make some assumptions about possible inputs of size n to determine the average-case efficiency.

Example **Sequential Search**

The following algorithm searches for a given value in a given array by sequential search.

```
algorithm sequential_search (A [0.. n-1], item)
{
      input  :  An array A [0.. n-1] and search element called item.
      output :  Returns the index of the first element of A that matches item or - 1 if there
                are no matching elements.
      i = 0;
      while (i < n and A [i] != item) do
            i = i + 1
      end while
      if i < n
         return i ;
      else
         return -1 ;
}
```

Analysis:

1. **Best Case:** The best case occurs when the element to be searched is found at the first location. In that case, the algorithms make only one comparison.

   ∴ The number of comparisons = 1

   $T_{best}(n) = 1$

2. **Worst Case:** The worst case occurs when the element to be searched is found at the last position or element to be searched is not found at any location. In both the cases, the number of comparisons required is n.

   ∴ $T_{worst}(n) = n$

3. **Average Case:** To find the average case efficiency, the following assumptions are made.

   a. The probability of successful search is equal to P $(0 \le P \le 1)$

   b. The probability of the first match occurring in the ith position of the array is the same for every i.

In the case of successful search, the probability of the first match occurring in the ith position of the array is P/n or every i. and in the case of unsuccessful search, the number of comparisons is n with the probability of such a search being (1-p). Therefore

$$T_{avg}(n) = \left[1\frac{P}{n} + 2\frac{P}{n} + - - + j.\frac{P}{n} + - - + n\frac{P}{n}\right] + n(1-p)$$

$$= \frac{P}{n}[1+2+--+n] + n(1-p)$$

$$= \frac{P}{n}\frac{n.(n+1)}{2} + n(1-p)$$

$$= \frac{p.(n+1)}{2} + n(1-p)$$

P = 1 makes the search successful and $T_{avg}(n)$ gives (n + 1) /2.

P = 0 makes the search unsuccessful and $T_{avg}(n)$ gives n.

Note:

In most of the cases, the following inequality holds good.

$$T_{best}(n) \le T_{avg}(n) \le T_{worst}(n)$$

1.10 Growth of Functions

Performing step count or operation count calculation for large algorithms is a time consuming task. A another standard way of analysing algorithms is through mathematical tools. In the previous section, we discussed that the running times are expressed in terms of the input size (n). We need to analyze the growth of the functions i.e we want to find out, if the input increases, how quickly the running time goes up.

1.10.1 Asymptotic Notations

When it comes to analysing the complexity of any algorithm in terms of time and space, we can neve provide an exact number to define the time required and the space required by the algorithm, instea we express it using some standard mathematical notations, also known as **Asymptotic Notations**

Meaning and Definition of Asymptotic Notations

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case complexities of an algorithm.

In mathematics, asymptotic analysis, is a method of describing the limiting behaviour of a function. In computing, asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance based on the input size.

For example, the running time of one operation is computed as f(n), and maybe for another operation, it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is small in value.

The word Asymptotic means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken). In Asymptotic analysis, we ignore the constant factors and insignificant parts of an expression, to device a better way of representing complexities of algorithms, in a single coefficient, so that comparison between algorithms can be done easily.

Example 1: Let us take an example, if some algorithm has a time complexity of $T(n) = (n^2 + 8n + 6)$, which is a quadratic equation. For large values of n, the 8n + 6 part will become insignificant compared to the n^2 part.

For n = 1000, n^2 will be 1000000 while 8n + 6 will be 8006.

Also, When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.

An algorithm that takes a time of $200n^2$ will be faster than some other algorithm that takes n^3 time, for any value of n larger than 200. Since we're only interested in the asymptotic behaviour of the growth of the function, the constant factor can be ignored too.

Example 2:

If we have two algorithms with the following expressions representing the time required by them for execution, then:

Expression 1: $8n^2 + 6n - 2$

Expression 2: $n^3 + 50n - 5$

Now, as per asymptotic notations, we should just worry about how the function will grow as the value of n (input) will grow, and that will entirely depend on n^2 for the Expression 1, and on n^3 for Expression 2. Hence, we can clearly say that the algorithm for which running time is represented by the Expression 2, will grow faster than the other one, simply by analyzing the highest power coefficient and ignoring the other constants(8 in $8n^2$) and insignificant parts of the expression(6n - 2 and 50n - 5).

All we need to do is, first analyze the algorithm to find out an expression to define it's time requirements and then analyze how that expression will grow as the input(n) will grow.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

The next topics gives several standard methods for simplifying the asymptotic analysis of algorithms. To analyse the algorithms, computer scientists used several asymptotic notations and important notations among them are :

Types of Asymptotic Notations:

1. **Big Oh Notation (O)**
2. **Big Omega Notation (Ω)**
3. **Big Theta Notation (θ)**

Let f(n) and g(n) are non-negative functions defined on the set of natural numbers. O(g (n)) is the set of all functions with a smaller or same order or growth as g(n). For example

$$\left.\begin{array}{r} n \in O(n^2) \\ 25n + 10 \in O(n^2) \end{array}\right\} \text{These two functions are linear and have a smaller order of growth than } g(n) = n^2$$

$$\left.\begin{array}{r} n^3 \notin O(n^2) \\ 50n^3 \notin O(n^2) \\ n^4 + 5n + 6 \notin O(n^2) \end{array}\right\} \text{These three functions are cubic and have a higher value of growth than } g(n) = n^2$$

The next notation, Ω (g (n)), stands for the set of all functions with a large or same order of growth as g (n).

$$\left.\begin{array}{r} n^3 \in \Omega(n^2) \\ \frac{1}{2}n(n-1) \notin \Omega(n^2) \end{array}\right\} \text{These functions have higher value of growth or same growth as } n^2.$$

$$\left.\begin{array}{r} 50n + 25 \notin \Omega(n^2) \\ n^2 + 50 \notin \Omega(n^3) \end{array}\right\} \text{These functions have smaller value of growth than } g(n) = n^2.$$

1.10.1.1 Big-oh Notation (O)

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm. It tells us that a certain function will never exceed a specified time for any value of input n.

Big Oh notation gives an upper bound on function f(n). The upper bound of f(n) indicates that the function f(n) will be the worst-case that it does not consume more than this computing time.

Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.

In Worst case, starting from the front of the array, we find the element or number we are searching fo at the end, which will lead to a time complexity of n, where n represents the number of total element We use the big-O notation to say that the time complexity is O(n), which means that the time complexity will never exceed n, defining the upper bound, hence saying that it can be less than or equal to n, which is the correct representation.

Definition : Big-oh Notation (O)

$f(n) = O(g(n))$ such that there exists two positive constants 'c' and 'n_0' with the constraint that.

$|f(n)| \le c\,|g(n)| \quad \forall n \ge n_0$

It is often useful to think of g as some given function and f as the function to analyze.

The important point is that f is bounded above by some constant multiple of g(n) for all large values of n. Let us assume that n is a real number.

The notation O(g(n)) is usually called "big oh of g of n". In this notation f(n) grows no more than g(n).

Example 1

Given $f(n) = 5n + 2$ Prove that $f(n) = 0\,(n)$

Here g(n) is n.

$$|f(n)| \le c\,|g(n)| \qquad \forall n \ge n_0$$

$$|5n + 2| \le c\,|n| \qquad \forall n \ge n_0$$

Now we should find the value of c and n0 such that the above inequality can be satisfied.

$$|5n + 2| \le 7\,|n| \qquad \forall n \ge 1$$

The above inequality can be satisfied by setting c = 7 and $n_0 = 1$. (always find the minimum possible value of n_0 and c).

$\therefore \quad f(n) = O(n)$

Example 2

Givenf $(n) = 3n + 2$ Prove that $g\,(n) = O(n)$

Here g (n) = n and big – oh – notation constraint is

$$|f(n)| \le c\,|g(n)| \qquad \forall n \ge n_0$$

$$|3n + 3| \le c\,|n| \qquad \forall n \ge n_0$$

$$3n + 2 \le 4n \qquad \forall n \ge 2$$

or

$$3n + 2 \le 5n \qquad \forall n \ge 1$$

The above inequality can be satisfied by setting c = 4, $n_0 = 2$ or c = 5, $n_0 = 1$

$\therefore \quad f(n) = 0\,(n)$

Example 3

Given $f(n) = 10n^2 + 4n + 2$ Prove that $f(n) = 0(n^2)$

$| f(n) | \leq c \, | g(n) | \quad \forall n \geq n_0$

$| 10n^2 + 4n + 2 | \leq c \, | n^2 | \quad \forall n \geq n_0$

$| 10n^2 + 4n + 2 | \leq 11 \, | n^2 | \quad \forall n \geq 5$

$10n^2 + 4n + 2 \leq 11n^2 \quad \forall n \geq 5$

The above inequality can be satisfied by setting $n_0 = 5$ and $c = 11$.

$\therefore$ **$f(n) = O(n^2)$**

Example 4

Given $f(n) = 10^6 n^2$ Prove that $f(n) = O(n^2)$

$|10^6 n^2 | \leq c \, | n^2 | \quad \forall n \geq n_0$

choose $c = 10^6$ and $n_0 = 1$, we get

$| 10^6 n^2 | \leq 10^6 n^2 \quad \forall n \geq 1$

$\therefore$ **$f(n) = O(n^2)$** is proved.

Example 5

Given, $f(n) = 20n^3 - 5$ Prove that $f(n) = O(n^3)$

$| 20n^3 - 5 | \leq c \, | n^3 | \quad \forall n \geq n_0$

$20n^3 - 5 \leq c \, n^3 \; \forall n \geq n_0$

choose $c = 20$ and $n_0 = 1$

$20n^3 - 5 \leq 20n^3 \quad \forall n \geq 1$

$\therefore$ **$f(n) = O(n^3)$** is proved.

Example 6

Given $f(n) = 100n + 5$ Prove that $f(n) = O(n^2)$

$| 100n + 5 | \leq c. \, |n^2| \quad \forall n \geq n_0$

$100n + 5 \leq 105.n^2 \quad \forall n \geq 1$

The above inequality can be satisfied by setting $c = 105$ and $n_0 = 1$.

$\therefore$ **$f(n) = O(n^2)$ is proved.**

1.10.1.2 Big Omega Notation (Ω)

Big Omega notation is used to define the lower bound of any algorithm or we can say the best case of any algorithm. This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big Ω, we mean that the algorithm will take atleast this much time to complete it's execution. It can definitely take more time than this too.

This notation is used to find the lower bound behaviour of f(n). The lower bound implies that below this time the algorithm cannot perform better. i.e., the algorithm will take at least this much of time (this indicates the lower bound). It is represented mathematically by notation Ω For a given function g(n), we denote by Ω (g(n) (pronounced as omega - of g of n). The function g(n) is only a lower bound on f(n). For the statement f(n) = Ω(g (n) to be informative, g(n) should be as large a function of n as possible for which the statement f(n) = Ω (g(n)) is true.

Definition: Omega Notation (Ω)

$f(n) = \Omega(g(n))$

if and only if there exists two positive constants c and n0 with the constraint that

$|f(n)| \geq c\,|g(n)| \quad \forall n \geq n_0$

Here c is some positive constant. Thus g is a lower bound (except for a constant factor c) on the value of f for all suitably large n.

Example 1

Given, $f(n) = 5n + 2$ Prove that $f(n) = \Omega(n)$

Here g(n) is n and omega notation constraint is

$|f(n)| \geq c\,|g(n)| \qquad \forall n \geq n_0$

$|5n + 2| \geq c\,|n| \qquad \forall n \geq n_0$

Since 5n + 2 is always greater than 5n we can choose c = 5 and $n_0 = 1$

$5n + 2 \geq 5n \qquad \forall n \geq 1$

The above inequality can be satisfied according to omega-notation by setting c = 5 and $n_0 = 1$

∴ **f (n) = Ω (n) is proved.**

Example 2

Given $f(n) = 6 * 2^n + n^2$ Prove that $f(n) = \Omega(2^n)$

Hence g (n) is 2^n and we have the constraint as

$|f(n)| \geq c\,|g(n)| \qquad \forall n \geq n_0$

$6.2^n + n^2 \geq c.2^n \qquad \forall n \geq n_0$

$6.2^n + n^2 \geq 1.2^n \qquad \forall n \geq 1$

The above inequality can be satisfied by setting c = 1 and $n_0 = 1$

∴ **$f(n) = \Omega(2^n)$ is proved.**

Example 3

Given $f(n) = 10n^2 + 4n + 3$ Prove that $f(n) = \Omega(n)$

$$10n^2 + 4n + 3 \geq c.n \qquad \forall n \geq n_0$$

Choosec = 1 and $n_0 = 1$, we get

$$10n^2 + 4n + 3 \geq 1.n \qquad \forall n \geq 1$$

The above inequality can be satisfied by setting $c = 1$ and $n_0 = 1$

∴ **$f(n) = \Omega(n)$ is proved.**

1.10.1.3 Theta-Notation (θ)

The theta notation can be used when the function f(n) can be bounded both from above and below by the same function g(n). For some functions, the lower bound and upper bound may be same. In finding the maximum or minimum element in an array, the computing time is O(n) and Ω(n). There exists a special notation to denote for functions having the same time complexity for lower bound and upper bound and this notation is called the theta – notation and which is denoted by θ.

The time compexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be. For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here, in the example above, complexity of $\Theta(n^2)$ means, that the avaerage time for any input n will remain in between, $k1 * n^2$ and $k2 * n^2$, where k1, k2 are two constants, therby tightly binding the expression rpresenting the growth of the algorithm.

Definition: Theta Notation (θ)

$f(n) = \theta(g(n))$

If and only if there exists three positive constants c_1, c_2 and n_0 with the constraint that

$$c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)| \qquad \forall n \geq n_0$$

Example 1 **Prove that $\frac{1}{2}n^2 - 3n = \theta(n^2)$**

To prove the above statement, we must determine positive constants c_1, c_2 and n_0 such that

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \qquad \forall n \geq n_0$$

Dividing by n^2 throughout

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \qquad \forall n \geq n_0$$

When $n_0 = 1$ and $c_2 = \frac{1}{2}$ the inequality $\frac{1}{2} - \frac{3}{n} \le c_2$ will hold good

When $n_0 = 7$ and $c_1 = \frac{1}{14}$ the inequality $c_1 \le \frac{1}{2} - \frac{3}{n}$ will hold good.

Summarizing we get

$$\frac{1}{14} \le \frac{1}{2} - \frac{3}{n} \le \frac{1}{2} \qquad \forall n \ge 7$$

$\therefore$ $f(n) = \theta(n^2)$ is proved.

Example 2 f(n) = 3n + 2 Prove that f(n) = θ(n)

The constraint is

$$c_1 |\, g(n) \,| \le |\, f(n) \,| \le c_2 \,|\, g(n) \,| \qquad \forall\, n \ge n_0$$

$$c_1 .\, n \le 3n + 2 \le c_2 .\, n \quad \forall\, n \ge n_0$$

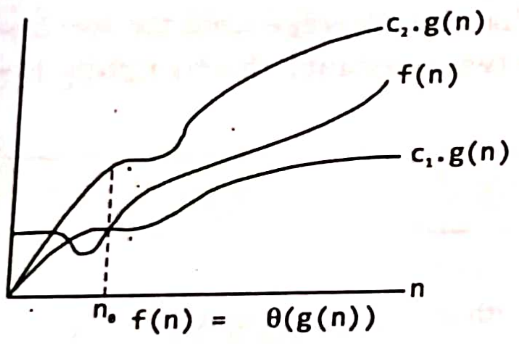
$$3.\, n \le 3n + 2 \le 4.\, n \quad \forall\, n \ge 2$$

The above inequality can be satisfied by setting

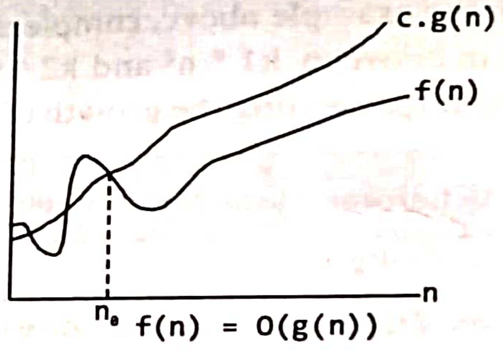
$c_1 = 3,\ c_2 = 4$ and $n_0 = 2$

$\therefore$ $f(n) = \theta(n)$ is proved.

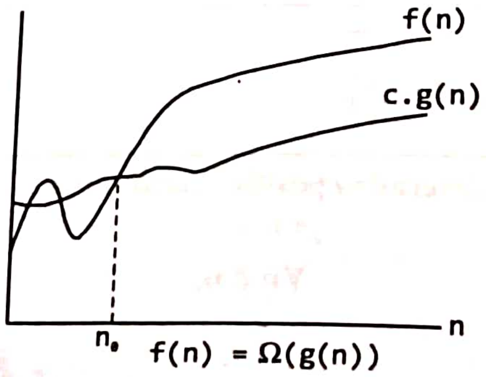
1.10.1.4 Comparison of O, Ω and θ notations



(a)



(b)



(c)

Fig: Graph examples of θ, O and Ω notations

In the above graph, the value of n_0 shown is the minimum possible value; any greater value would also work.

In figure (a) theta notation bounds a function to within a constant factor. The value of f(n) always lies between $c_1 . g(n)$ and $c_2 . g(n)$ inclusive.

The figure (b) gives an upper bound for a function to within a constant factor. The value of f(n) always lie on or below c. g(n).

The figure (c) gives an lower bound for a function to within a constant factor. The value of f(n) always lie on or above c. g(n).

1.10.1.5 Some Useful Properties

1. **Transitivity**

   $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$ imply $f(n) = \theta(h(n))$

   $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$

   $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$

2. **Reflexivity**

   $f(n) = O(f(n))$ $\quad f(n) = \Omega(f(n))$ $\quad f(n) = \theta(f(n))$

3. **Symmetry**

   $f(n) = \theta(g(n))$ iff $g(n) = \theta(f(n))$

4. **Transpose Symmetry**

   $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

We can draw an analogy between the asymptotic comparison of two functions f(n) and g(n) and the comparison of two real numbers a and b.

$f(n) = O(g(n)) \approx a \le b$ $\quad f(n) = \Omega(g(n) \approx a \ge b$ $\quad f(n) = \theta(g(n)) \approx a = b$

1.10.1.6 Using limits for Comparing Orders of Growth

The asymptotic notations like 0, Ω and θ are rarely used for comparing orders of growth of two specific functions. A much more convenient method for doing so is based on computing the limit of the ratio of two functions. There exists three cases for doing so.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} \begin{cases} 0 & \text{implies that f(n) has smaller order of growth than g(n)} \\ c & \text{implies that f(n) has the same order of growth g(n)} \\ \infty & \text{implies that f(n) has larger order of growth than g(n)} \end{cases}$$

The first two cases are for Big-oh(O), the last two cases are for omega(Ω) and only second case is for theta(θ).

Example 1 Given f(n) = 5n + 3, prove that f(n) = O(n)

$$= \lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{5n+3}{n} = \lim_{y\to 0}\frac{5\frac{1}{y}+3}{\frac{1}{y}} = \lim_{y\to 0} 5+3y = 5$$

Since 5 is a constant, we can say that f(n) is of the same order of growth of g(n).

Example 2 Given that f(n) = 20n³ − 3 Prove that f(n) = O(n³)

$$= \lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{20n^3-3}{n^3} = \lim_{y\to 0}\frac{20\frac{1}{y^3}-3}{\frac{1}{y^3}} = \lim_{y\to\infty} 20-3y^3 = 20$$

Since 20 is a constant, ∴ f(n) = O(n³) is proved.

Example 3 Given that f(n) = 2n, then prove that f(n) = O(n²)

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{2n}{n^2}\ ; \qquad \lim_{y\to 0}\frac{2.\frac{1}{y}}{\frac{1}{y^2}} = \lim_{y\to 0} 2y = 0$$

Since the result is 0, which implies that f(n) has a smaller order of growth than g(n).

∴ f(n) = O(n²) is proved.

1.10.1.7 Basic Efficiency Classes

We are concerned here about the classification of algorithms according to their asymptotic efficiencie
Some times the time efficiencies of a large number of algorithms fall into only a few classes. Thes
classes are listed in the following table in increasing order of their orders of growth along with thei
names and comments.

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since a algorithm's running time typically goes to infinity when its input size grows infinitel large.
log n	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteratio of the algorithm. Note that a logarithmic algorithm cannot take into account all its input(or even a fixed fraction of it) : any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n(e.g., sequential search) belong to this class.
n log n	*linearithmic*	Many divide-and-conquer algorithms including mergesort and quicksort in the average case, fall into this category.

n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops. Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops. Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and faster orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

1.11 Standard Mathematical Notations and Functions

To analyse an algorithm, we need to know some of the commonly used mathematical functions and notations used. Let us see some of the important functions.

1.11.1 Monotonicity

A function f(n) is monotonically increasing if $m \le n$ implies $f(m) \le f(n)$. Similarly, it is monotonically decreasing if $m \le n$ implies $f(m) \ge f(n)$. A function f(n) is strictly increasing if $m < n$ implies $f(m) < f(n)$ and strictly decreasing if $m < n$ implies $f(m) > f(n)$.

1.11.2 Floor and Ceiling Functions

If x is a real number, then floor and ceiling of x is defined as follows:

floor(x) : Returns the largest integer that is smaller than or equal to x (i.e : rounds downs the nearest integer).

ceil(x) : Returns the smallest integer that is greater than or equal to x (i.e : rounds up the nearest integer).

Example: floor(2.5)=2 floor(2.9) =2 floor(-7.2)=-8 ceil(2.5)=3 ceil(2.9)=3 ceil(-7.2)=-7

1.11.3 Remainder Function (Modular Arithmetic)

If k is any integer and M is a positive integer, then: k (mod M) gives the integer remainder when k is divided by M.

Example: 26(mod 7) = 5 30(mod 5) = 0

1.11.4 Integer and Absolute Value Functions

If x is a real number, then integer function INT(x) will convert x into integer and the fractional part is removed.

Example: INT (5.34) = 5 INT (−7.5) = −7

The absolute function ABS(x) or | x | gives the absolute value of x i.e. it gives the positive value of x even if x is negative.

Example: ABS(−99) = 99 or ABS [−99] = 99 ABS(−3.33) = 3.33 or ABS [−3.33] = 3.33

1.11.5 Summation Symbol

Here we introduce the summation symbol Σ (Sigma). Consider a sequence of n-terms $a_1 + a_2 \dots a_n$ the sums $a_1 + a_2 + \dots + a_n$ will be denoted as

$$\sum_{1 \le i \le n} a_i$$

Examples are sum of n natural numbers, square of n positive integers and many more. We can w an expression as a sum of series or sequence. We can also call summation of summation.

Consider that a function f(n) denotes the summation of n positive integers. Here function computes the order of magnitude of an algorithm.

$$f(n) = \sum_{1 \le i \le n} a_i = 1 + 2 + \dots + n = n(n+1)/2$$, and the order of complexity is $O(n^2)$

1.11.6 Factorial Function

n! denotes the product of the positive integers from 1 to n. n! is read as 'n factorial', i.e.

$n! = 1 * 2 * 3 * \dots * (n-2) * (n-1) * n$

Example: $4! = 1 * 2 * 3 * 4 = 24$

$5! = 5 * 4! = 120$

1.11.7 Permutations

Let us consider a set of n elements. A permutation of this set means the arrangement of the eleme of the set in some order.

Example: Suppose the set contains a, b and c. The various permutations of these elements can abc, acb, bac, bca, cab, cba.

If there are n elements in the set then there will be n! permutations of those elements. It means if set has 3 elements then there will be 3! = 1 * 2 * 3 = 6 permutations of the elements.

1.11.8 Exponents and Logarithms

Exponent means how many times a number is multiplied by itself. If m is a positive integer, then:

$a^m = a * a * a * \dots * a$ (m times) and $a^{-m} = 1 / a^m$

Example: $2^5 = 2 * 2 * 2 * 2 * 2 = 32$, $2^{-5} = 1 / 2^5 = 1 / 32$

The concept of logarithms is related to exponents. If b is a positive number, then the logarithm of a positive number x to the base b is written as $\log_b x$. It represents the exponent to which b should raised to get x i.e. $y = \log_b x$ and $b^y = x$

1.12 Advantages and Disadvantages of Algorithms

Advantages of Algorithms

- It is a step by step representation of a solution to a given problem, which makes it easy to understand.
- It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
- Every step in an algorithm has its own logical sequence so it is easy to debug.
- The problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program.
- An algorithm acts as a blueprint of a program and helps during program development.
- An algorithm uses a definite procedure.

Disadvantages of Algorithms

- Writing algorithm takes a long time.
- Difficult to show branching and looping in algorithms.
- Understanding and writing complex logic through algorithms can be very difficult.