DEPARTMENT OF COMPUTER APPLICATIONS
ADVANCED DATA STRUCTURES AND ALGORITHMS

INTRODUCTION TO ALGORITHM

Algorithm Definition:

An Algorithm is a set of well-defined instructions designed to perform a specific set of tasks. Algorithms are used in Computer science to perform calculations, automatic reasoning, data processing, computations, and problem-solving.

Design and Analysis of Algorithm Definition:

Design and Analysis of Algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology. It also helps to design and analyze the logic on how the program will work before developing the actualcode for a program.

ANALYSIS OF ALGORITHM

The analysis is a process of estimating the efficiency of an algorithm. There are two fundamental parameters based on which wecan analysis the algorithm:

- **Space Complexity:** The space complexity can be understood as the amount of space required by an algorithm to run to completion.

- **Time Complexity:** Time complexity is a function of input size **n** that refers to the amount of time needed by an algorithm to run to completion.

if there is a problem **P1**, then it may have many solutions, such that each of these solutions is regarded as an algorithm. So, there may be many algorithms such as A_1, A_2, A_3, …, A_n.

Before you implement any algorithm as a program, it is better to find out which among these algorithms are good in terms of time and memory It would be best to analyze every algorithm in terms of **Time** that relates to which one could execute faster and **Memory** corresponding to which one will take less memory.

So, the Design and Analysis of Algorithm talks about how to design various algorithms and how to analyze them. After designing and analyzing, choose the best algorithm that takes the least time and the least memory and then implement it as a program in C.

Memory is relatively more flexible. We can increase the memory as when required by simply adding a memory card. So, we will focus on timethan that of the space.

The *running time* is measured in terms of a particular piece of hardware, not a robust measure. When we run the same algorithm on a

different computer or use different programming languages, we willencounter that the same algorithm takes a different time.
Generally, we make three types of analysis, which is as follows:

- **Worst-case time complexity:** For '**n**' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the maximum number of steps performed on an instance having an input size of n.
- **Average case time complexity:** For '**n**' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.
- **Best case time complexity:** For '**n**' input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

Complexity of Algorithm:

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

To assess the complexity, the order (approximation) of the count of operation is always considered instead of counting the exact steps.

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "**Big O**" notation. Here the f corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation O(f) determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.

The complexity can be found in any form of the following forms. It is nothing but the order of constant, logarithmic, linear and so on, the number of steps encountered for the completion of a particular algorithm. To make it even more precise, we often *call the complexity ofan algorithm as "running time"*.

Typical Complexities of an Algorithm:

- **Constant Complexity:**

  It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.
- **Logarithmic Complexity:**

  It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.
- **Linear Complexity:**

It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.

- **Quadratic Complexity:**

  It imposes a complexity of **$O(n^2)$**. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.
- **Cubic Complexity:**

  It imposes a complexity of **$O(n^3)$**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.
- **Exponential Complexity:**

  It imposes a complexity of **$O(2^n)$, $O(N!)$, $O(n^k)$, ….** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.

ASYMPTOTIC NOTATIONS

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm islinear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation):

(n) is bounded above by some constant multiple of g(n) for all large n,

i.e., if there exist some positive constant c and some nonnegative integern_0 such that t (n) ≤ cg(n) for all n ≥ n_0

$$O(g(n)) = \{ t(n): \text{there exist positive constants } c \text{ and } n0 \text{such that } 0 \le t(n) \le cg(n) \text{ for all } n \ge n0 \}$$

Example:

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$100n + 5 \le 100n + n$ (for all $n \ge 5$) = $101n \le 101n^2$.

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. For example, we could also

reason that 100n + 5 ≤ 100n + 5n (for all n ≥ 1) = 105n to complete the proofwith c = 105 and n_0 = 1

Omega Notation (Ω-notation)

Omega notation represents the *lower bound* of the running time ofan algorithm. Thus, it provides the *best case complexity of an algorithm*.

A function t(n) is said to be in (g(n)), denoted t(n) ∈ (g(n)), if t(n) is

bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integern_0 such that t (n) ≥ cg(n) for all n ≥ n_0.

Ω(g(n)) = { t(n): there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ t(n) for all n ≥ n0 }

Example

Here is an example of the formal proof that n3 ∈ (n2):

n3 ≥ n2 for all n ≥

0, i.e.,we can select c = 1 and n_0 =

0.

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Sinceit represents *the upper and the lower bound of the running time* of an

algorithm, it is used for analyzing the *average-case complexity of analgorithm.*

A function t(n) is said to be in (g(n)), denoted t(n) ∈ (g(n)), if t(n) is

bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n_0 such that $c2g(n) \le t(n) \le c1g(n)$ for all

$n \ge n_0$.

$\Theta(g(n)) = \{ t(n)$: there exist positive constants c1, c2 and n_0 such that $0 \le c1g(n) \le t(n) \le c2g(n)$ for all $n \ge n_0 \}$

Example

For example, let us prove that $\frac{1}{2}\, n(n-1) \in (n^2)$. First, we prove the right inequality (the upper bound):

$$\frac{1}{2}\, n(n-1) = \frac{1}{2}\, n^2 - \frac{1}{2}\, n \le \frac{1}{2}\, n^2 \text{ for all } n \ge 0.$$

Second, we prove the left inequality (the lower bound):

$\frac{1}{2}$	$n(n-1) =$	$\frac{1}{2}$
	$\ge$	$\frac{1}{2}$
	$=$	$\frac{1}{4}$

$n^2 - \frac{1}{2}\, n$

$n^2 - 1/2\ n \geq 1/2\ n$ (for all $n \geq 2) n^2$.

Hence, we can select $c_2 = 1/4$, $c_1 = 1/2$, and $n_0 = 2$.

IMPORTANCE OF EFFICIENT ALGORITHMS

In real-world applications, efficiency is very important.

- To avoid large program or application to become slow and unresponsive.
- The efficiency of an algorithm often gets worse rapidly as the size of the dataset increases. Big data requires us to come up with new algorithms with good efficiency.
- To avoid delays.
- For time critical applications like weather predictions.
- Many modern businesses rely on having the fastest algorithms,such as search and comparison engines, medical analysis etc…
- Efficiency is only one factor in choosing or designing an algorithm.
- For fetching results with high quality and speed.

PROGRAM PERFORMANCE MEASUREMENT

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are

multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. *Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem*. When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

Algorithm performance Definition:

Performance of an algorithm is a process of making evaluative judgement about algorithms. Thus performance of an algorithm means *predicting the resources which are required to an algorithm to perform its task*.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem. We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,
Generally, the performance of an algorithm depends on the following elements...

- Whether that algorithm is providing the exact solution for the problem?
- Whether it is easy to understand?

- Whether it is easy to implement?
- How much space (memory) it requires to solve the problem?
- 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 the remaining elements.

Performance analysis of an algorithm definition:

Performance analysis of an algorithm is the process of calculatingspace and time required by that algorithm.

Performance analysis of an algorithm measures:

- Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space
- Time required to complete the task of that algorithm (Time Complexity)

RECURRENCES: THE SUBSTITUTION METHOD – THE RECURSION – TREE METHOD

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time

complexity of Merge Sort can be written as $T(n) = 2T(n/2) + cn$. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

- **Substitution Method:** We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

  For example consider the recurrence $T(n) = 2T(n/2) + n$

  We guess the solution as $T(n) = O(nLogn)$. Now we use induction to prove our guess.

  We need to prove that $T(n) <= cnLogn$. We can assume that it is truefor values smaller than n.

  $$T(n) = 2T(n/2) + n$$

  $$<= 2cn/2Log(n/2) + n$$

  $$= cnLogn - cnLog2 + n$$

  $$= cnLogn - cn + n$$

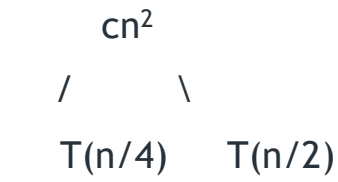
  $$<= cnLogn$$

- **Recurrence Tree Method:** In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep

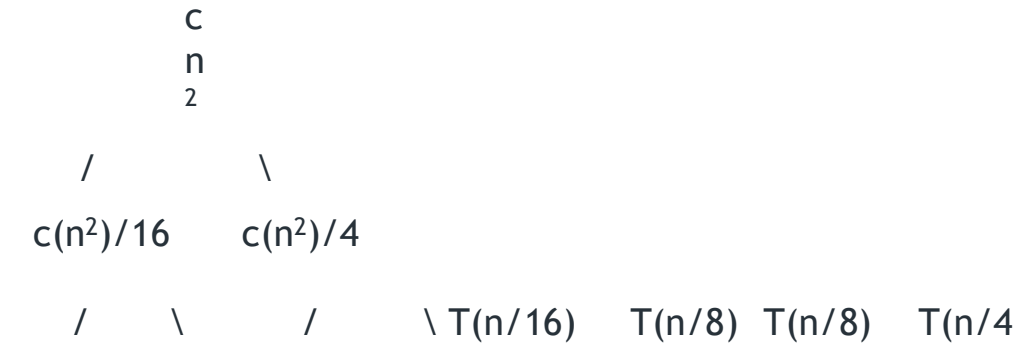
drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

For example consider the recurrence relation

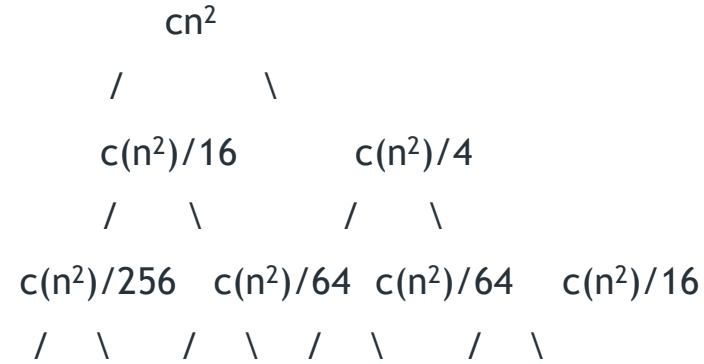
$$T(n) = T(n/4) + T(n/2) + cn^2$$



If we further break down the expression T(n/4) and T(n/2), we getfollowing recursion tree.



Breaking down further gives us following



To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series

T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...

The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

- **Master Method:** Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$$T(n) = aT(n/b) + f(n) \text{ where } a >= 1 \text{ and } b > 1$$

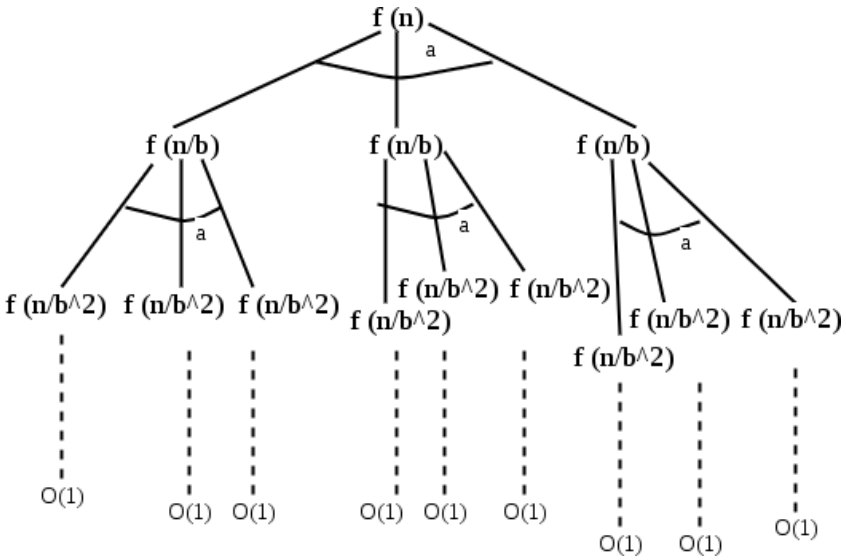
There are following three cases:

- If $f(n) = O(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
- If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
- If $f(n) = \Omega(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all

leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$.



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at

root is asymptotically more, then our result becomes work done at root(Case 3).

Examples of some standard algorithms whose time complexity can beevaluated using Master Method.

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $\text{Log}_b a]$is also 1. So the solution is $\Theta(n \text{ Log}n)$.

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $Log_b a$ is also 0. So the solution is $\Theta(Logn)$.

DATA STRUCTURES AND ALGORITHMS

Data Structure Definition:

A data structure is a particular way of organizing data in a computer so that it can be used efficiently. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

Types of Data Structures:

- Primitive Data Structures (Built-In Data Structures)

Primitive data structures are those which are predefined way of storing data by the system. And the set of operations that can be performed on these data are also predefined. Primitive data structures are

- char
- int
- float
- double
- pointer

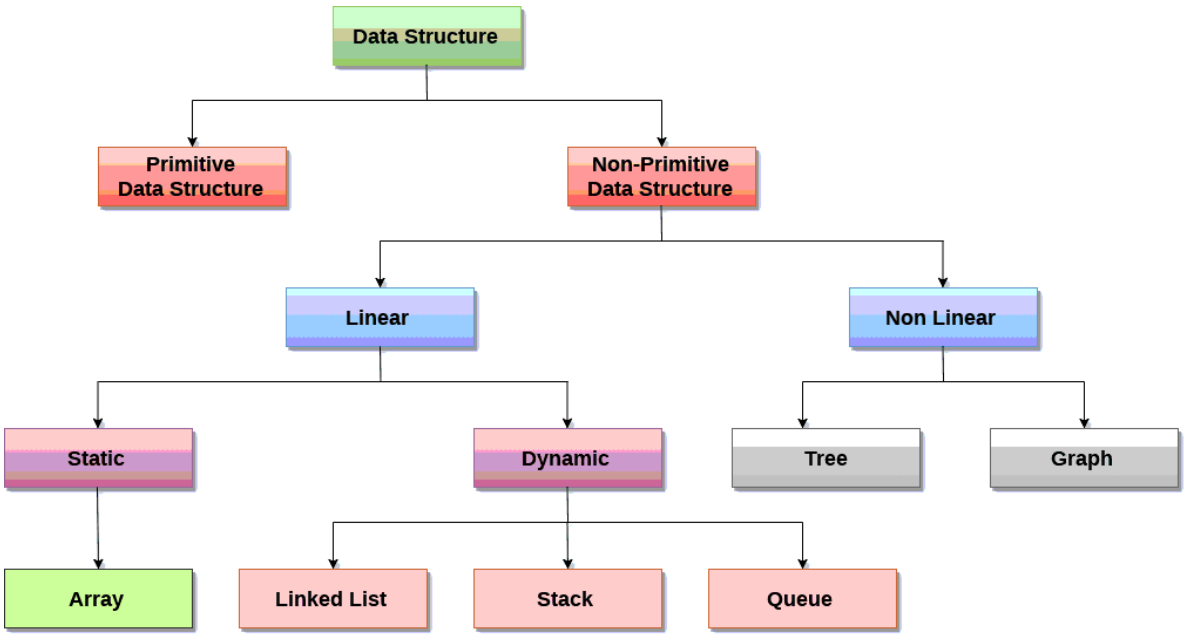
Characters are internally considered as int and floats also falls under double and the predefined operations are addition, subtraction, etc.

- Non-primitive Data Structures (User Defined Data Structures)

Non-primitive data structures are more complicated data structures and they are derived from primitive data structures. Non-primitive data structures are used to store large and connected data. Some example of Non-primitive data structures are:

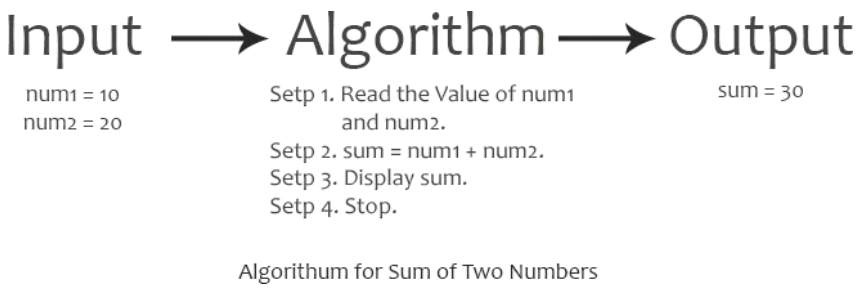
- Linked List
- Tree
- Graph
- Stack
- Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operationis required.



Algorithm Definition:

An algorithm is a finite set of unambiguous instructions, written in order, to accomplish a certain predefined task or obtaining a required output for any legitimate input in a finite amount of time. Algorithm is not the complete code or program, it is just the core logic of a problem.



Algorithum for Sum of Two Numbers

Data Definition:

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

Time Complexity & Space Complexity:

Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.

Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. Space needed by an algorithm is equal to the sum of the following two components

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

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

Good Program = Good Algorithm + Good Data Structures