UNIT 1 BASICS OF AN ALGORITHM AND ITS PROPERTIES

1.0 INTRODUCTION

Studying algorithms is an exciting subject in computer science discipline. We come across a large number of interesting problems and techniques to solve to solve these problems. Not every problem can be solved with the existing techniques but majority of them can be. But let us define what is an algorithm first. The word *algorithm* is derived from the mathematician of the ninth century *Abdullah Jafar Muhammad ibn Musa Al-khowarizmi*. The word 'al-khowarizmi' is 'Algorismus'in Latin which became Algorithm after his name.

He defined Algorithm as:

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a well-defined computational procedure that takes input and produces output.
- An algorithm is a finite sequence of instructions or steps (i.e. inputs) to achieve some particular output.

In this unit, the basics of the algorithms and its designing process will be discussed. Section 1.3 will define the algorithm and its uses with suitable example. An algorithm is designed with five basic building blocks, namely sequencing, selection, and iteration. A detailed discussion about these building blocks of an algorithm is presented in Section 1.4.

The solution of a problem can be achieved through a number of algorithms. To check which algorithm is better than the others, a parameter, known as time complexity, is used. Therefore, time complexity is one of the important concepts related to algorithm which are discussed in Section 1.5. Section 1.6 deals with the analysis of Algorithms. To compare Algorithms, complexity is the parameter to be considered. Computing problems are categorized according to their solving approach. These are discussed in section 1.7. Section 1.8 comprises the solving techniques of various computing problems. In section 1.9 Deterministic and Stochastic Algorithm are discussed. An algorithm is deterministic if the next output can be predicted/ determined from the input and the state of the program, whereas stochastic algorithms are random in nature.

Chapter is summarized in section 1.10. In section 1.11 review questions of the chapter are covered for check pointing purpose. In Section 1.12 a list of reference material is enlisted for further readings.

1.1 OBJECTIVES

After studying this unit, you should be able to:

- Define and list properties of an algorithm.
- List basics building blocks of algorithms.
- Explain fundamental techniques to design an algorithm.
- Define the time and space complexity of an algorithm.
- Differentiate between deterministic and stochastic algorithms.

1.2 EXAMPLE OF AN ALGORITHM

An algorithm is not a coding instruction rather it is a sequence of tasks written in common language, if executed produces certain output within a time frame. An algorithm is completely independent of programming language.

For a good algorithm, it must satisfy the following characteristics or properties:

1. **Input**: There must be a finite number of inputs for the algorithm.
2. **Output**: There must be some output produced as a result of execution of the algorithm.
3. **Definiteness**: There must be a definite sequence of operations for transformation of input into output.
4. **Effectiveness**: Every step of the algorithm should be basic and essential.

5. **Finiteness**: The transformation of input to output must be achieved in finite steps, i.e. the algorithm must stop, eventually! Stopping may mean that it should produce the expected output or a response that no solution is possible.

Following are desirable characteristics of an algorithm:

- The **algorithm should be general** and is able to solve several cases.
- The **algorithms should use resources efficiently**, i.e. takes less time and memory in producing the result.
- The **algorithms should be understandable** so that anyone can understand and apply it to own problem.
- The **algorithm should follow the uniqueness** such that each instruction of the algorithm is unambiguous and clear.

The "analysis" of an algorithm finds the suitability of it in terms of the time and space (memory) complexity also known as the performance evaluation of the algorithm.
Before proceeding further to discuss about writing the algorithm and its analysis, let's first see how an algorithm looks like. For the same, let us consider a well-known algorithm to find the Greatest Common Divisor (GCD) of two given integers. We can write an algorithm in any natural language. Here, we are presenting the algorithm in English language.

To find GCD of two given Integers, we are using an efficient **Euclid's algorithm** which is named after the ancient Greek mathematician Euclid.

The pseudo code for computing GCD (a, b) by Euclid's method is as follows:

// a and b are two positive numbers where a is dividend and b is a divisor

1. If b=0, return a and exit
2. else go to step 3
3. Divide a by b and assign remainder to r
4. Assign the value of b to a and the value of r to b and go back to step 1

To validate the algorithm, it must produce the desired result within finite number of steps. The above-mentioned algorithm has two inputs and one output. The algorithm is also definiteness and written in basic and effective sentences. The algorithm is also finite as it terminates in finite steps. To observe the same, let us find the GCD of $\boldsymbol{a}$ = 1071 and $\boldsymbol{b}$ = 462 using Euclid's algorithm.

Iteration 1:

1. Divide **a**=1071 by **b**=462 and store the remainder in **r**.

   r= 1071 % 462 (here, % represents the remainder operator)

r = 147

2. If **r** = 0, the algorithm terminates and **b** is the GCD. Otherwise, go to Step 3.

   Here, **r** is not zero, so we will go to Step 3.

3. The integer will get the current value of integer **b** and the new value of integer **b** will be the current value of **r**.

   Here, **a**=462 and **b**=147

4. Go back to Step 1.

Iteration 2:

1. Divide **a**= 462 by **b**= 147 and store the remainder in **r**.

   r= 462 % 147 (here, % represents the remainder operator)

   r = 21

2. If **r** = 0, the algorithm terminates and **b** is the GCD. Otherwise, go to Step 3.

   Here, **r** is not zero, so we will go to Step 3.

3. The integer **a** will get the current value of integer **b** and the new value of integer **b** will be the current value of **r**.

   Here, **a**= 147 and **b**= 21

4. Go back to Step 1.

Iteration 3:

1. Divide **a**=147 by **b**=21 and store the remainder in **r**.

   r= 147 % 21 (here, % represents the remainder operator)

   r = 0

2. If **r** = 0, the algorithm terminates and **b** is the GCD. Otherwise, go to Step 3.

   Here, **r** is zero, so the algorithm terminates and b is the answer, i.e. 21.

Let us write the **Euclid's Algorithm**

```
Algorithm GCD-Euclid (a, b)
begin [start of Algorithm]
{
        while b ≠ 0 do
        {
                r ← a mod b;
                a ←b;
                b← r;
        } [end of while loop]
return (b)
}  [end of algorithm]
```

Algorithm 1: Finding GCD of (a, b) with Euclidian Method.

1.3 BASICS BUILDING BLOCKS OF ALGORITHMS

An algorithm is a procedural way to write the solution of a problem. It is designed with five basic building blocks, namely: sequencing, selection, iteration procedure and recursion. In the first subsection we discuss sequencing, iteration and selection and in the subsequent subsection procedure & recursion will be explained.

Sr. N.	Building Block	Common Name
1.	Sequencing	Step by step actions
2.	Selection	Decision
3.	Iteration	Repetition or Loop
4.	Procedure	
5.	Recursion	

1.3.1 Sequencing, Selection and Iteration

A solution of a problem mainly comprises these three basic blocks only. In an Algorithm there may be actions to be performed linearly or sequentially as they written in text. At some times the next action/ statement to be executed is

decided based on some condition called the selection of next action to be performed. Some set of actions/statements are to be executed more than once called repetition or the loop.

Let's consider the example of finding GCD of (a, b) with Euclidian method to understand the basic building blocks of an algorithm(Algorithm 1)

Sequencing: A problem can be solved by performing some actions in a sequence [called algorithm], and the order of execution of those actions is important to ensure the correctness of an algorithm.

If the order of steps of algorithm changes and does not follow the steps as specified, it will not produce the correct output as expected.

Selection: As an algorithm has to be generalized to solve many cases, there may be situations where the sequence of execution of actions may depend on some condition. That is, some of the instructions will only be executed if a given condition satisfies.

It is like the next action to be performed is dependent upon some Boolean expression. So, using selection, next step to be executed is determined.
Iteration: While solving a problem certain actions may be required to execute a certain number of times or until a certain condition is met.

Let us consider Algorithm 1 again to understand these concepts: Finding GCD of (a, b) with Euclidian Method.

Algorithm GCD-Euclid (a, b)

Step1: begin [start of Algorithm]

Step2: {

Step3: while b ≠ 0 do

Step4: {

Step5: r ← a mod b;

Step6: a ←b;

Step7: b← r;

Step8: } [end of while loop]

Step9: return (b)

Step10: } [end of algorithm]

In above algorithm Step5, Step6 and Step7 are example of sequencing, as these statements are always executed in sequence as written the text.

Step3 is selection step as it decides which next step is to be executed among Step4 and Step9 according to the condition of the loop.

Step3 is also acts as iteration or the looping statements. Based on the while loop condition, the Step4 to Step8 are executed in repeatedly manner.

1.3.2 Procedure & Recursion

Though the above-mentioned three control structures, viz., direct sequencing, selection and repetition, are sufficient to express any algorithm, yet the following two advanced control structures have proved to be quite useful in facilitating the expression of complex algorithms viz.

(i) Procedure
(ii) Recursion

Let us first take the advanced control structure *Procedure*.

Procedure

Among a number of terms that are used, instead of procedure, are subprogram and even function. These terms may have shades of differences in their usage in different programming languages. However, the basic idea behind these terms is the same, and is explained next.

It may happen that a sequence frequently occurs either in the same algorithm repeatedly in different parts of the algorithm or may occur in different algorithms. In such cases, writing repeatedly of the same sequence, is a wasteful activity. *Procedure* is a mechanism that provides a method of checking this wastage. For example we can define GCD(a, b) as a procedure/function only once and can call it a number of times in a main function with different values of a and b

Under this mechanism, the sequence of instructions expected to be repeatedly used in one or more algorithms, is written only once and outside and independent of the algorithms of which the sequence could have been a part otherwise. There may be many such sequences and hence, there is need for an identification of each of such sequences. For this purpose, each sequence is prefaced by statements in the following format:

Procedure <Name> (<parameter-list>) [*: < type>*]
<declarations>
<sequence of instructions expected to be occurred repeatedly>
end;

In cases of procedures which pass a value to the calling program another basic construct (in addition to *assignment, read and write*) viz., *return (x)* is used, where x is a variable used for the value to be passed by the procedure.

There are various mechanisms by which values of *a and b* are respectively associated with or transferred to *x and y*. *The variables like a and b*, defined in the calling algorithm to pass data to the procedure (*i.e., the called algorithm*), which the procedure may use in solving the particular instance of the problem, are called **actual parameters** or **arguments**.

Recursion

Next, we consider another important control structure namely recursion. In order to facilitate the discussion, we recall from Mathematics, one of the ways

in which the factorial of a natural number n is defined:

factorial (1) = 1

factorial (n) = n*factorial(n─1).

For those who are familiar with recursive definitions like the one given above for factorial, it is easy to understand how the value of (n!) is obtained from the above definition of factorial of a natural number. However, for those who are not familiar with recursive definitions, let us compute factorial (4) using the above definition.

By definition
factorial (4) = 4 * factorial (3).
Again by the definition
factorial (3) = 3 * factorial (2)
Similarly
factorial (2) = 2* factorial (1)
And by definition
factorial (1) = 1

Substituting back values of factorial (1), factorial (2) etc., we get factorial (4) = 4.3.2.1=24, as desired.

This definition suggests the following procedure/algorithm for computing the factorial of a natural number n:

In the following procedure factorial (n), let *fact* be the variable which is used to pass the value by the procedure *factorial* to a calling program. The variable fact is initially assigned value 1, which is the value of factorial (1).

Procedure factorial (n)

```
    fact: integer;
begin
    fact ←1
    if n equals 1 then return fact
    else begin
    fact ←n * factorial (n –1)
    return (fact)
    end;
end;
```

In order to compute *factorial (n ─1)*, procedure *factorial* is called by itself, but this time with (simpler) argument (n ─1). The repeated calls with simpler arguments continue until factorial is called with argument 1. Successive multiplications of partial results with 2,3, …..up to n finally deliver the desired result.

Definition: A procedure, which can call itself, is said to be **recursive procedure/algorithm**. For successful implementation of the concept of recursive procedure, the following conditions should be satisfied.

(i) There must be in-built mechanism in the computer system that supports the calling of a procedure by itself, e.g, there may be in-built stack operations on a set of stack registers.

(ii) There must be conditions within the definition of a recursive procedure under which, after finite number of calls, the procedure is terminated.

(iii) The arguments in successive calls should be simpler in the sense that each succeeding argument takes us towards the conditions mentioned in (ii).

Recursion is an important construct which will be used extensively to solve sorting algorithms, searching algorithm, matrix multiplications, etc.

1.4 A SURVEY OF COMMON RUNNING TIME

For a given problem, more than one algorithm can be designed. However, one algorithm may be better than the other. To compare two algorithms for a problem, running time is generally used which is defined as the time taken by an algorithm in generating the output. An algorithm is better if it takes less running time. The "time" here is not necessarily the clock time. However, this measure should be invariant to any hardware used. Therefore, the running time of an algorithm can be represented in terms of the number of operations executed for a given input. More the number of operations, the larger the running time of an algorithm. So, if we can find the number of operations required for a given input in an algorithm then we can measure the running time. This running time of an algorithm for producing the output is also known as **time complexity**. Time here is not the clock time

Therefore, two algorithms can be compared in terms of time complexity. an Algorithm is better compared to others having smaller running time (time complexity).

Running time of an algorithm is represented as a function $T(n)$, where n is the input size. Let, an algorithm has a running time $T(n) = cn$, where c is a constant. The running time for this algorithm is linearly dependent on the size of the input. The unit of $T(n)$ is unspecified.

Following are the generalized form of running time for the algorithms:

1. **Constant Time($O(1)$):** If the running time does not depend on the input size (n) then it is known as constant running time. It can be represented as

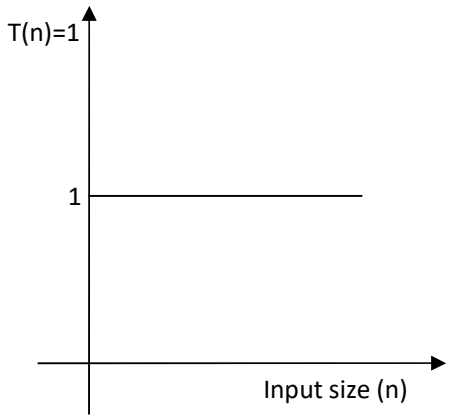

Figure (a): *T(n) = O(1)*

2. **Linear Time *O(kn)*:** If the time complexity is at most a constant factor times the size of the input, then it is known as linear time complexity and is presented as *T(n) <=kn* where k is a constant or T(n) =*O(n)*. An algorithm of this type of complexity generally completes the execution in a single pass. For example to search for minimum value of *a* given *n* numbers in an array the processing can be completed just in one pass. The following program fragment demonstrates. In this way we perform constant amount of work in processing each element of an array.

```
minimum = a[1]
for í = 2 to n
if a[í] < minimum
minimum = a[í]
end
end if
```

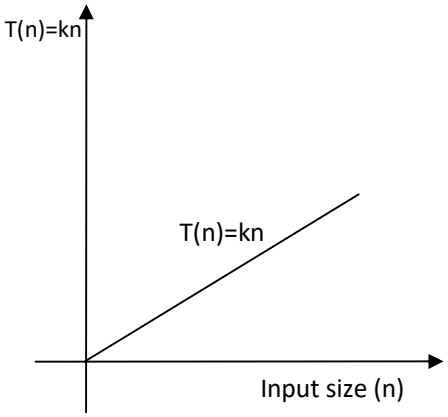

Figure (b): T(n) = O(n)

3. **Logarithmic Time(*log(n)*)**: If the time complexity of an algorithm is proportional to the logarithm of the input size, then it is known as logarithmic time complexity and depicted as *O(log n)*time. For example running time of binary search algorithm is *O(log n)*.O(n log n) is a very common running time for many algorithms which are solved through divide and conquer technique such as Merge sort ,Quick sort algorithms, etc., The common operations among all these problems are in splitting of the array in equal sized sub-arrays and then solve it recursively.

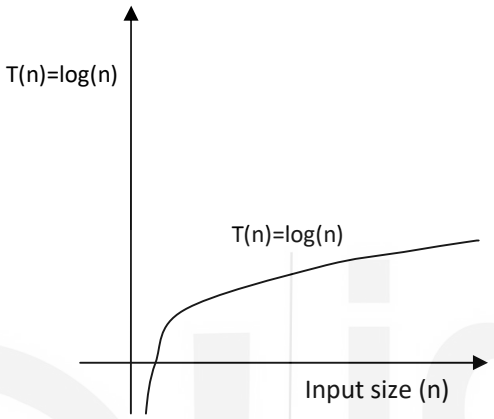

Figure (c): T(n) =O(log(n))

Quadratic Time: (*T(n)= O(n)^2)*- It occurs when the algorithm is having a pair of nested loops. The outer loop iterates *O(n)* time and for each iteration the inner loop takes *O(n)* time so we get *O(n^2)* by multiplying these two factors of *n*. Practically this is useful for problem for small input size or elementary sorting algorithms. The worst case time complexity for Bubble sort, Insertion sort, Selection sort and insertion sort running time complexities are O(n^2)

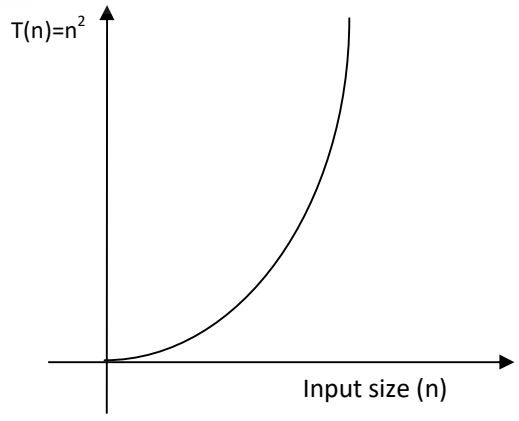

Figure (d): T(n) = O=O(n^2)

4. **Cubic Time:** (*T(n)= O(n^3)):* It often occurs when the algorithm is having there nested loops and each loop has a maximum *n* iterations. Let us have one interesting example which requires cubic time complexity. Suppose we are given n sets: S_1, S_2,….S_n. Size of each set is n (ie each set is having n elements). The problem is to find whether some pairs of these sets are disjoint, i.e there are no common elements in these pairs and what is the time complexity ?

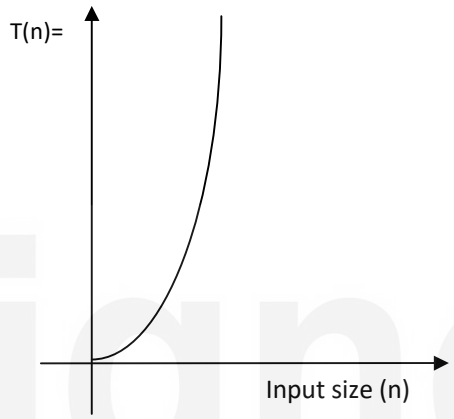

Figure (e): T(n) =O(n^3)

Pseudo-code for finding common elements in pair of sets:

```
for each set S_i of n elements
  for each other set S_j of n elements
    for each element x of S_i
      check whether x also belongs to S_j
    end for
    if x belongs to both S_i and S_j
      print "  S_j and S_j are not disjoint"
    end if
  end for
end for
```

Time Complexity- The innermost loop will be executed for n(n+1)(n+2)/6 times, which clearly states that the time complexity for this algorithm is O(n^3).

5. **Polynomial Time:** (0*(n^k)*):-This running time is obtained when the search over all subsets of a set of *a* size *k*in performed. To understand

the complexity of running time we have to find how many distinct subsets of a size *k* of *n* elements of a set can be chosen. For that we have to take a combination of n elements taken k at a time .As an example let us consider a problem to find an independent set in a graph which can be defined as a set of nodes in which no pair of nodes have an edge between them. Let us formulate the independent set problem in the following way: given a constant *k* and a graph *G* having *n* nodes (vertices) find out an independent set of a size *k*.

The brute force method to solve this problem would require searching for all subsets of *k* nodes and for each subset it would examine whether there is an edge connecting any two nodes for each subset s of a size *k* .Below is a pseudo-code for finding an independent set.

Pseudo-code

```
for each subset s of a size k in a graph G
   check whether s is an independent set
   if yes, print " s is an independent set
   else stop
```

In this case the outer loop will iterate $O(n^k)$ times and it selects all *k*-node subsets of *n* node of the graph. In the inner loop within each subset it loops for each pair of nodes to find out whether there is an edge between the pair which will require *O(* 2 out of k) pairs of search i.e. O(k^2) search. Therefore the total time now is $O(k^2n^k)$. Since k is a constant, it can be dropped, finally it is $O(n^k)$.

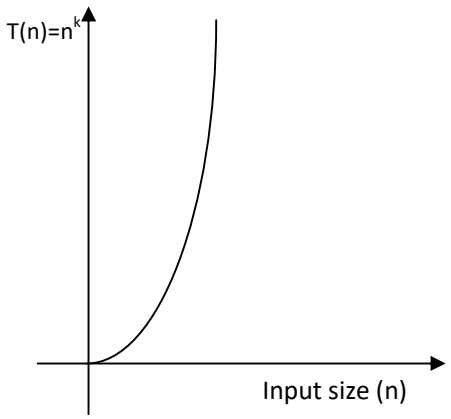

Figure (f):T(n) =O(n^k)

6. **Exponential Time:** *($O(k^n)$)* Beyond the polynomial time complexity there are other two types of bounds :exponential time *$O(2^N)$*and factorial time *O(n!):* let us refine the independent set problem that we are given a graph of a size *n* and want to find out an independent of a maximum value instead of some constant *k* which is less than *n*. The modified version of the pseudo-code is presented below.

Pseudo-code : Pseudo-code for finding an Independent Set of a graph

```
Input G(V,E)
{
for each subset s of n number of nodes
verify whether s is an independent set
if s is the largest among all the subsets examined so for
  print "s is the largest independent set "
end if
end for
}end of code fragment
```

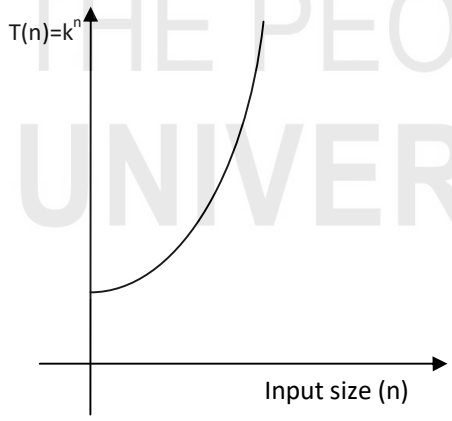

Figure (g): T(n) =O(k^n)

In this case the total number of subsets of n elements would be 2^n, so the outer loop will execute 2^n times instead of n^k times

Verification of all pairs of subsets i.e. (2^n) whether these subsets are having edges or not and then selecting the maximum will be O(n^2) i.e the total number of pair of subsets. The total running time would be

$O(n^2*2^n)$.$O(2^n)$running time complexity arises when a search algorithm considers all subsets of *n* elements.

Factorial time (O(n!):In comparison to the growth of exponential running time, the growth of factorial time (n!) is more rapid. The running time of this type of complexity arises in two types of algorithms:

(i)Algorithm for Matching, for example bipartite matching algorithm. Suppose there are *n* number boys and *n* number of girls. To find perfect matching between n number of boys &n number of girls, the first boy will be compared with *n* numbers of girls. The second boy will be left with *(n-1)* choices among girls for comparison. There will be only (n-2) options for matching for the third boy, and so forth. After array girls multiplying all these options for n boys we obtain n! ie. n(n-1) (n-2) .....(2) (1)

(ii) O(n!) also occurs where the algorithm requires arranging n elements into a particular order (i.e. a permutation of *n* numbers). A classic example is travelling salesman problem. Given a n number of cities with distance between all pairs of cities with the following conditions (i) the salesman can start the tour with any city but must conclude the tour with the starting city only (ii) all cities must be visited only once except the one where from the tour starts. The problem is to find out the shortest tour covering all n cities. Applying a brute force approach to find out the solution, a salesman has to explore n! searches which will take O(n!). Note that a salesman can pick up any city among n cities to start the tour. Next it will have (n-1) cities to pickup the second city on the tour. There will be (n-2) cities to pick up the third city at the next stage and so forth. Multiplying all these choices we get n! i.e. n (n-1) (n-2) ....(2) (1)

1.5 ANALYSIS & COMPLEXITY OF ALGORITHM

The term "analysis of algorithms" was introduced by Donald Knuth. It has become now an important computer science discipline whose overall objective is to understand the complexity of an algorithm in terms of time complexity and storage requirement.

System performance is directly dependent on the efficiency of algorithm in terms of both the time complexity as well the memory. An algorithm designed for time sensitive application takes too long to run can render its results of no use.

Suppose M is an algorithm with n the input data size. The time and space used by the algorithm M are the two main measures for the efficiency of M. The time is measured by counting the number of key operations, for example, in case of sorting and searching algorithms, the number of comparisons is the

number of key operations. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.

The ***complexity*** of an algorithm M is the *function f(n)*, which give the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. In general the term "***complexity***" given anywhere simply refers to the running time of the algorithm. There are 3 cases, in general, to find the complexity function f(n):

- Worst-case − The maximum number of steps taken on any instance of size a.
- Best-case − The minimum number of steps taken on any instance of size a.
- Average case –The number of steps taken on average for all instances of size n

The analysis of the average case assumes a certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The Average case also uses the concept of probability theory. Suppose the numbers $N_1, N_2, \ldots N_k$ occur with respective probabilities $p_1, p_2, \ldots p_k$. Then the expectation or average value of E is given by $E = N_1 p_1, N_2 p_2 \ldots . N_k p_k$

To understand the Best, Worst and Average cases of an algorithm, consider a linear array $A[1 \ldots . n]$, where the array A contains n-elements. Students may you are having some problem in understanding. Suppose you want *either* to find the location LOC of a given element (say x) in the given array A or to send some message, such as LOC=0, to indicate that x does not appear in A. Here the linear search algorithm solves this problem by comparing given x, one-by-one, with each element in A. That is, we compare with A[1], then A[2], and so on, until we find x LOC such that $x = A[LOC]$.

Algorithm: (**Linear search)**
/* **Input:** A linear list A with n elements and a searching element x.
Output: Finds the location LOC of x in the array A (by returning an index)
or return LOC=0toindicate x is not present in A.*/

1. [Initialize]: Set K=1 and LOC=0.
2. Repeat step 3 and 4 while (LOC = = 0 && K ≤ n)
3. *If (x = = A[K])*
4. {
5. LOC=K
6. K=K+1;
7. }
8. *If (LOC = = 0)*
9. *Print (" x is not present in the given array A);*
10. *Else*
11. *Print f("x is present in the given array A at location A [LOC]);*
12. *Exit [end of algorithm]*

Analysis of linear search algorithm

The complexity of the search algorithm is given by the number C of comparisons between x and array elements A[K].

Best case: Clearly the best case occurs when x is the first element in the array A. That is $x = A[LOC]$. In this case $C(n) = 1$

Worst case: Clearly the worst case occurs when x is the last element in the array A or x is not present in given array A (to ensure this we have to search entire array A till last element). In this case, we have $C(n) = n$.

Average case: Here we assume that searched element x appear in array A, and it is equally likely to occur at any position in the array. Here the number of comparisons can be anyofnumbers*1,2,3,…,n,* and each number occurs with the probability p=1/n then

$$C(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \cdots \ldots \ldots + n.\frac{1}{n}$$
$$= (1 + 2 + \cdots \ldots \ldots + n).\frac{1}{n}$$
$$= \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2}$$

It means the average number of comparisons needed to find the location of x is approximately equal to half the number of elements in array A. From above discussion, it may be noted that the complexity of an algorithm in the average case is much more complicated to analyze than that of worst case. Unless otherwise stated or implied, we always find and write the complexity of an algorithm in the worst case.

There are three basic asymptotic(i.e., when input size n → infinity) notations which are used to express the running time of an algorithm in terms of function, whose domain is the set of natural numbers N={1,2,3,…..}. These are:

- *0(Big – 'Oh')* [This notation is used to express Upper bound (maximum steps) required to solve a problem]
- *Ω(Big – 'Oh')* [This notation is used to express Lower bound i.e. minimum (at least) steps required to solve a problem]
- Θ ('Theta') Notations.[Used to express both Upper & Lower bound, also called tight bound]

Asymptotic notation gives the *rate of growth*, i.e. performance, of the run time for "*sufficiently large input sizes"* $(i.e.\ n \to \infty)$and is **not** a measure of the *particular* run time for a specific input size (which should be done empirically). O-notation is used to express the Upper bound (worst case); Ω-notation is used to express the Lower bound (Best case) and Θ- Notations is used to express both upper and lower bound (i.e. tight bound) on a function.

We generally want to find either or both an asymptotic *lower bound* and *upper bound* for the growth of our function.

1.6 TYPES OF PROBLEMS

The types of problems in computing are limitless, and are categorized into a few areas to make it easy for researchers to address types of problems while addressing the algorithm field.

Following are the some commonly known problem types:

- Sorting
- Searching
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

For the above-mentioned categories, certain standard input sets are defined as benchmarking sets to analyse the algorithms.

Sorting

The *sorting* is the process to arrange the given set of items in a certain order, assuming that the nature of the items allow such an ordering. For example, sorting a set of numbers in increasing or decreasing order and sorting the character strings, like names, in an alphabetical order.

Researchers have published a large number of different sorting algorithms, targeting various types of items. A sorting algorithm does not necessarily work optimally for all types of items list. Some of them are good in terms of resource usage, while some are fast in terms of computing. The efficiency of a sorting algorithm also depends on the type of input, some work well on randomly ordered inputs, whereas others perform better on already almost-sorted lists. Some of the sorting algorithms perform well for lists residing in the memory, while others perform optimally for sorting large files stored on a secondary disk.

As of now in common, the best sorting algorithm takes *nlogn* comparisons to sort an item list of n items.

For any sorting algorithm following two characteristics are desirable:

1. Stability
2. In-place.

A sorting algorithm is called stable if it does not change the relative positions of any two equal items of input list. Say, in an input list, there are two equal

item sat positions *i* and *j* where *i< j,* then the final position of these items in the sorted list should also be k and *l* respectively, such that k<*l.* That is there should not be any swapping among these equal items and should not interchange their position with each other.

A sorting algorithm needs extra memory space to store elements during the swapping process. For small set of items in a list, this constraint is not observable but, for an input list of large elements the required storage space is considerable large. An algorithm is said to be *in-place* if the required extra memory is not markable.

Searching

Searching is finding an element, referred as *search key*, in a given set of items (may have the redundant value). Searching is one of the most important and frequently performed operation on any dataset/database.

Searching is one of the most favourite areas of researches in the field of algorithm analysis. No single searching performs optimally to all situations. Some algorithms are faster but consume more memory; some are very fast but only with specific input set; and so on.

While designing an algorithm for searching problem, it is highly influenced by the nature of underlying data. The data, static in nature, has to be addressed differently than the dynamic one in nature with addition or deletion from the data set of an item.

String Processing

Exponential increase in the textual data due to the various applications over social media and blogs, string-handling algorithms become a current area of research. Another reason for blooming strings rather text processing is the kind of data available. Now day's the business paradigms are totally changed from offline to online. According to Grant Thornton, e-commerce in India is expected to be worth US$ 188 billion by 2025.Most of the text data is used to predict the interest of people involving direct or indirect monetary benefits for commercial organizations specially e-commerce sectors. One of the most widely used search engine (Google) is also based on string processing.

Sting matching is one of the string processing problems.

Graph Problems

It is always favourable for researchers to map a computational problem to a graph problem. Many computational problems can be solved using graph. Most of the computer network problems can be solved using graph algorithms efficiently. Problems like: visiting all the nodes of a graph (broadcasting in network), routing in networks (finding the minimum cost path, i.e. the shortest

path, path with minimum delay etc. can be solved efficiently with graph algorithms.

At the same time some of the graph problems are computationally not easy, like the travelling salesman and the graph-colouring problems. The *Travelling Salesman Problem (TSP)* is used to cover *n* cities by taking the shortest path and not visiting any of the city more than once. The *graph-colouring problem* seeks to colour all the vertices of a graph with minimum number colours such that, no two adjacent vertices having the same colour. While solving TSP cities can be considered as the vertices of the graph. Event scheduling could be one of the problems which can be solved using graph colouring algorithm. Considering events to be represented by the vertices, there exists an edge between two events only if the corresponding events cannot be scheduled at the same time.

Combinatorial Problems

These types of problems have a combination of solutions i.e. more than one solution are possible. The aim of the combinatorial problems is to find permutations, combinations, or subsets, satisfying the given conditions. The travelling salesman problem, independent set and the graph-coloring problems can be categorized as examples of *combinatorial problems.* From both theoretical as well as practical point of view, the combinatorial problems are considered to be one of the most difficult problems in computing. Due to the combinatorial type of solutions, it becomes very difficult to handle the problems with big size inputs sets. The number of combinatorial objects (the output solution) grows rapidly with the problem's size.

Geometric Problems

Some of the applications of *Geometric algorithms* are computer graphics, robotics and tomography. These algorithms are based upon geometric objects such as points, lines, and polygons. The geometry procedures are developed to solve various geometric problems, like construction shapes of geometric objects, triangles, circles, etc., using ruler and compass.

Following are widely known classic problems of computational geometry:

1. The closest-pair problem
2. The convex-hull problem

The *closest-pair problem* is to find the closest pair out of a given set of points in the plane.

In the *convex-hull problem,* the smallest convex polygon is to be constructed so that it includes all the points of a given set.

Numerical Problems

Problems of numerical computing nature are simultaneous linear equations (linear algebra), differential equations, definite integration, and statistics. Most of the numerical problems could be solved approximately.

The biggest drawback of numerical algorithms is the accumulation of errors over the multiple iterations, due to rounding off the approximated result at each iteration.

1.7 PROBLEM SOLVING TECHNIQUES

Divide and Conquer Approach

This is one of the popular approaches in which a problem is divided into smaller subproblems. These subproblems are further divided into smaller subproblems until they can no longer be divided. It is a top down approach in which the algorithm *logically* progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.

An algorithm, following divide & conquer technique, involves following steps:

Step 1. Divide the problem (top level) into a set of sub-problems (lower level).

Step 2. Solve every sub-problem individually by recursive approach.

Step 3. Merge the solution of the sub-problems into a complete solution of the problem.

Following are the examples of the problems that can efficiently be solved using divide and conquer approach.

- Binary Search.
- Quick Sort.
- Merge Sort.
- Strassen's Matrix Multiplication.
- Closest Pair of Points.

Greedy Technique

Using Greedy approach, optimization problems are solved efficiently. In an optimization problem, the given set of input values are either to be maximized or minimized (called as objective), subject to some constraints or conditions.

- Greedy algorithm always picks the best choice (greedy approach) out of many at a particular moment to optimize a given objective.

- The greedy method chooses the local optimum at each step and this decision may result in overall non-optimum or optimum solution.
- The greedy approach doesn't always produce the optimal solution rather produces very nearby solution to the optimal solution.

Consequently, Greedy algorithms are often very easy to design for the optimisation problems. Following are some of the examples of the greedy approach.

- Kruskal's Minimum Spanning Tree
- Prim's Minimal Spanning Tree
- Dijkstra's shortest path
- Knapsack Problem

Dynamic Programming

Dynamic Programming approach is a bottom-up approach which involves finding solution of all sub-problems, saving these partial results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained. *Reusing* the results of sub-problems (by maintaining a table of results) is the major advantage of dynamic programming because it avoids the ***re-computations*** **(**computing results twice or more**)** of the same problem. Thus Dynamic programming approach takes much less time than naïve or straightforward methods.

The working style of dynamic programming is similar to divide and conquer approach. Both solve a problem by breaking it down into a several sub problems that can be solved recursively. The drawback of divide and conquer method is the calling of a recursive function with same output/result repeatedly which is overcome in dynamic programming by maintaining a table to store the results. It is dynamically decided whether to call a function or retrieve values from the table, that's why the word dynamic is used in it. The Dynamic programming approach is faster than the divide and conquer method as the redundancy of calling functions with same result are omitted in it.0-1 Knapsack and subset-sum problem are the examples of dynamic programming.

Branch and Bound

Branch and bound algorithm efficiently solves the discrete and combinatorial optimization problems. In branch-and-bound algorithm, a rooted tree is formed with the full solution set at the root. The algorithm explores the branches of this tree, representing the subsets of the solution set. A candidate solution of a root node is considered as a branch only if it is better than the already explored solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm. Branch and Bound algorithm are methods

for solving global optimization problems. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded (i.e., their children generated).

Randomized Algorithms

In a randomized algorithm, a random number is selected at any stage of the solution and is used for computation of the solution, that's why it is called as randomized algorithm. In other words it can be said that algorithms that make random choices for faster solutions are known as randomized algorithms. For example, in the Quick sort algorithm, a random number can be generated and considered as a pivot. In other example, a random number can be chosen as possible divisor to factor a large number.

Backtracking Algorithm

Backtracking algorithm is like creating checkpoints while exploring new solutions. It works analogues to depth-first search. It searches all the possible solutions. During the exploration of solutions, if a solution doesn't work, it back-track to the previous place and then find the other alternatives to get the solution. If there are no more choice points the search fails.

1.8 DETERMINISTIC AND STOCHASTIC ALGORITHM

Algorithms can be categorized either deterministic or stochastic in nature. An algorithm is deterministic if the next output can be predicted/ determined from the input and the state of the program, whereas stochastic algorithms are random in nature. Problems with unpredictable result cannot be solved using deterministic approach. For example, the next output of a card shuffling program of blackjack game should not be predictable by players even if the source code of the program is visible. Pseudorandom number generator method can be compromised and is often not sufficient to ensure the true randomness. The random number generated using Pseudorandom number generator method might be precisely predicted. To avoid this problem, use of a cryptographically secure pseudo-random number generator with an unpredictable random seed to initialize the generator can be better way. A hardware random number generator is the best way for achieving real randomness.

1.10 SUMMARY

In computation, an algorithm is independent from a programming language. Algorithm is designed to understand and analyze the solution of a computational problem. In an algorithm, statements may be used to perform an action called as sequencing, or to take decision (selection) or to repeat certain actions (iterations).

Running time of an algorithm is one of the most widely used parameter to judge an algorithm. Running time is computed as the number of instructions executed. Space complexity is measurement of memory storage used during the execution. For a computational problem, there may exist multiple algorithms to solve which leads to analyze these Algorithms to get the best solution as per the requirement.

An algorithm can be analyzed in terms of time complexity and space complexity. To evaluate algorithms of a problem, time and space complexities are considered. Algorithm, taking less time to produce the desired output, is desirable. There has to be a tradeoff between these two parameters, an algorithm with less space complexity may not be desirable if it runs for a long time.

In computing, based on the nature of the problem, it can be assigned any one of the commonly known categories, namely sorting, searching, string processing, graph problems, combinatorial problems, geometric problems, numerical problems.

Similar type of problems can be solved with similar approach. Some of the commonly used problems solving techniques are Brute Force and Exhaustive search approach, Divide and Conquer approach, Greedy technique, Dynamic Programming, Branch and Bound, Randomized algorithms, and Backtracking algorithm.

Another widely known categories of Algorithms, based on the type of the inputs used, are Deterministic and Stochastic Algorithms. If the output of an algorithm can be predicted by looking at the input, such algorithms are called as deterministic in nature. While stochastic algorithms are random in nature, means the output cannot be determined from the input.

1.11 SOLUTION TO CHECK YOUR PROGRESS

Q1. What is an Algorithm? What are the important characteristics of an algorithm? Solution: An Algorithm is a set of steps to solve a problem or a set of problems. Also, an algorithm is a step by step procedure to solve logical and mathematical or computational problems. A recipe is a good example of Algorithm. To cook a dish a recipe says what to be done step by step.

Important characteristics of an algorithm are: input, output, definiteness, effectiveness and finiteness

Q2.What are the building blocks of an Algorithm?

Selection, selection, iteration, procedure and recursion

Q3. How to judge an algorithm, whether it is efficient or not?

Solution: An algorithm should be both correct and efficient. The efficiency of an Algorithm is defined in terms of the resource usage to yield a correct answer.

In general the efficiency of an Algorithm is considered in terms of the computational complexity, which is: how hard is it to execute the algorithm. The execution of an Algorithm is very much depends on the input. An Algorithm may perform differently depending on how input looks like. For example: The efficiency of Insertion sort Algorithm depends on the input array. Here, insertion sort has a linear running time (i.e., O(n)). While it is quadratic running time (i.e., O(n2)) for an array sorted in reverse order known as the worst case.

Q4. Map the Set B with corresponding problems listed in Set B.

S. N.	Set A	S.N.	Set B
1	Sorting Problem	A	Arranging a list of numbers in ascending order.
2	Geometric Problem	B	Finding an item in set items.
3	Graph Problem	C	Finding the shortest path between two nodes.
4	Numerical Problem	D	Finding Euler graph for a given graph.
5	Searching Problem	E	Finding the solutions of a given set of linear equations.
6	String Processing	F	Finding the pair of points (from a set of points) with the smallest distance between them.
7		G	Match a word in a paragraph.

Solution: The correct match is as follows:

1 – A, 2 – F, 3 – C, D, 4 – E, 5 – B, 6 – G.

Q5. When should the deterministic approach of problem solving to be avoided? Explain with an example.

Solution: An algorithm is deterministic if the next output can be predicted/ determined from the input and the state of the program. Deterministic approach of problem solving technique is not suitable for the problems with unpredictable result.

For example, the next output of a card shuffling program of blackjack game should not be predictable by players even if the source code of the program is visible.

Q6. Differentiate Dynamic programming and backtracking problem solving approach. What problems can be solved by each technique?

Solution: Dynamic programming is a technique widely used to solve optimization problem. Optimization problem is used to find the either minimum or maximum result (a single result) out of all possible outcomes. In backtracking method, a brute force approach is used, hence it is not used for optimization problem. Backtracking approach is suitable for solving problems having multiple results and out of which, all or some of them are acceptable.

Following problems can be solved using backtracking approach:

- Eight queen puzzle
- Map coloring
- Sudoku

Following problems can be solved using dynamic programming approach:

Ans

- All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford
- 0/1 knapsack problem
- Chain matrix multiplication
- Traveling salesman problem

Q7 What problems can be solved through greedy technique?

Ans

- Fractional knapsack problem
- Minimum cost spanning tree
- Single source shortest path algorithm

Q9 What are the common running times for the algorithms?

Ans. Constant time, linear time, logarithmic time, polynomial time, exponential time and factorial time

Q10Define independent set problem.

Ans. Independent set problem can be defined as a set of nodes which are not joined by any edge. One way to formulate this problem is that given a constant k and a graph G having n nodes (vertices) find out an independent set of a size k.

1.12 FURTHER READINGS

1. Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. Algorithm Design, Jon Kleinberg and Eva Tardos, Pearson