UNIT II

Introduction to Algorithms,

An algorithm is a method for solving a computational problem. A formula or set of steps for solving a problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. Algorithms can be expressed in any language, from natural languages like English or French to programming languages likeFORTRAN.

We use algorithms every day. For example, a recipe for baking a cake is an algorithm. Most programs, except for some artificial intelligence applications, consist of algorithms. Inventing elegant algorithms that are simple and require the fewest steps possible is one of the principal challenges in programming.
For example, we might be able to say that our algorithm indeed correctly solves the problem in question and runs in time at most f(n) on any input of size n.
Definition. An algorithm is a finite set of instructions for performing a computation or solving a problem. Types of Algorithms Considered. In this course, we will concentrate on several different types of relatively simple algorithms, namely:
Selection -- Finding a value in a list, counting numbers;

Sorting -- Arranging numbers in order of increasing or decreasing value; and

Comparison -- Matching a test pattern with patterns in a database.

Properties of Algorithms. It is useful for an algorithm to have the following properties:

Input -- Values that are accepted by the algorithm is called input or arguments.

Output -- The result produced by the algorithm is the solution to the problem that the algorithm is designed to address.

Definiteness -- The steps in each algorithm must be well defined.

Correctness -- The algorithm must perform the task it is designed to perform, for all input combinations.

Finiteness -- Output is produced by the algorithm after a finite number of computational steps.

Effectiveness -- Each step of the algorithm must be performed exactly, infinite time.

Generality -- The procedure inherent in a specific algorithm should be applicable to all algorithms of the same general form, with minor modifications permitted.

Complexities of Algorithm:

Algorithmic complexity is concerned about how fast or slow algorithm performs.

We define complexity as a numerical function T(n) - time versus the input size n. We want to define time taken by an algorithm without depending on the implementation details. But you agree that T(n) does depend on the implementation. A given algorithm will take different amounts of time on the same inputs depending on such factors as processor speed; instruction set, disk speed, a brand of compiler etc. The way around is to estimate the efficiency of each algorithm asymptotically. We will measure time T(n) as the number of elementary "steps" (defined in any way), provided each such step takes constant time.
Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is time taken by addition of two bits. On different computers, the addition of two bits might take different time, say c1 and c2, thus the addition of two n-bit integers takes $T(n) = c1 * n$ and $T(n) = c2* n$ respectively. This shows that different machines result in different slopes, but time T(n) grows linearly as input size increases.
The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

We use algorithms to introduce salient concepts and to concentrate on the analysis of algorithm performance, especially computational complexity.

Concept. To facilitate the design of efficient algorithms, it is necessary to estimate the bounds on the complexity of candidate algorithms.

Representation. Complexity is typically represented via the following measures, which are numbered in the order that they are typically computed by a system designer:

Work W(n) -- How many operations of each given type are required for an algorithm to produce a specified output given n inputs?
Space S(n) -- How much storage (memory or disk) is required for an algorithm to produce a specified output given n inputs?

Time T(n) -- How long does it take to compute the algorithm (with n inputs) on a given architecture?

Cost C(n) = T(n) · S(n) -- Sometimes called the space-time bandwidth product, this measure tells a system designer what expenditure of aggregate computational resources is required to compute a given algorithm with n inputs.

Procedure. Analysis of algorithmic complexity generally proceeds as follows:

Step 1. Decompose the algorithm into steps and operations.
Step 2. For each step or operation, determine the desired complexity measures, typically using Big-Oh notation, or other types of complexity bounds discussed below.
Step 3. Compose the component measures obtained in Step 2 via theorems presented below, to obtain the complexity estimate(s) for the entire algorithm.

Example. Consider the following procedure for finding the maximum of a sequence of numbers. An assessment of the work requirement is given to the right of each statement.
Let $\{a_n\} = (a_1, a_2, ..., a_n)$

```
{ max = a1                         # One I/O operation
for i = 2 to n do:                 # Loop iterates n-1 times
{ if ai        max then            # One comparison per iteration max := ai }
}
```

Analysis : -

(1) In the preceding algorithm, we note that there are n-1 comparisons within the loop.
(2) In a randomly ordered sequence, half the values will be less than the mean, and a_1 would be assumed to have the mean value (for purposes of analysis). Hence, there will be an average of n/2 I/O operations to replace the value max with a_i. Thus, there are n/2 + 1 I/O operations.
(3) This means that the preceding algorithm is O(n) in comparisons and I/O operations. More precisely, we assert that in the average case:
W(n) = n-1 comparisons + (n/2 - 1) I/O operations.

Tractable -- An algorithm belongs to class P, such that the algorithm is solvable in polynomial time (hence the use of P for polynomial). This means that complexity of the algorithm is $O(n^d)$, where d may be large (which typically implies slow execution).

Intractable -- The algorithm in question requires greater than polynomial work, time, space, or cost. Approximate solutions to such algorithms are often more efficient than exact solutions and are preferred in such cases.

Solvable -- An algorithm exists that generates a solution to the problem addressed by the algorithm, but the algorithm is not necessarily tractable.

Unsolvable -- No algorithm exists for solving the given problem. Example: Turing showed that it is impossible to decide whether a program will terminate on a given input.

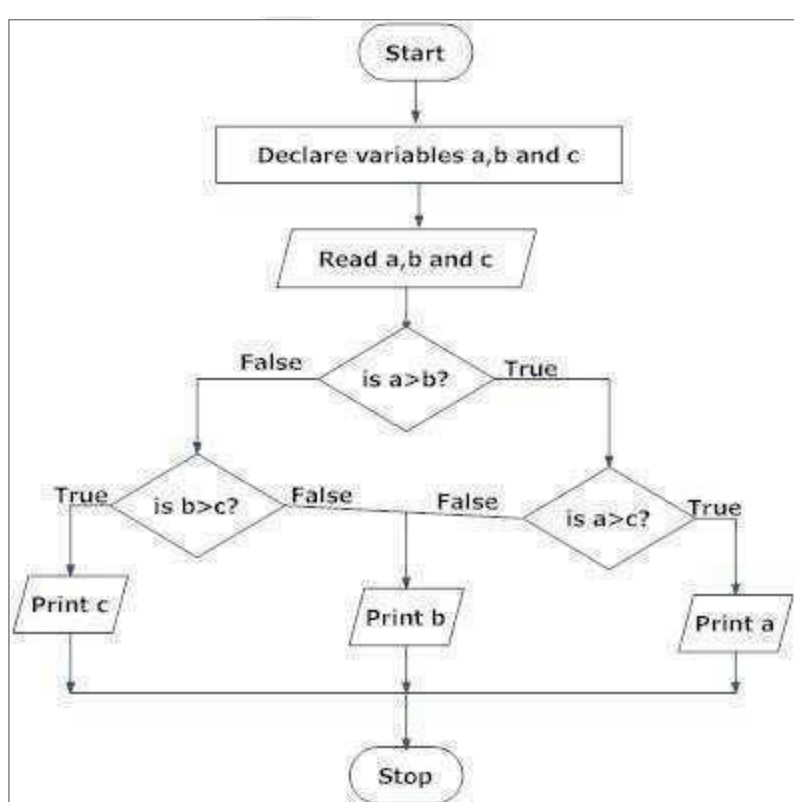
Class NP -- If a problem is in Class NP (non-polynomial), then no algorithm with polynomial worst-case complexity can solve this problem.

Class NP-Complete -- If any problem is in Class NP-Complete, then any polynomial-time algorithm that could be used to solve this problem could solve all NP-complete problems. Also, it is generally accepted, but not proven, that no NP-complete problem can be solved in polynomial time.

FLOWCHART

1. The flowchart is a type of diagram (graphical or symbolic) that represents an algorithm or process.
2. Each step in the process is represented by a different symbol and contains a short description of the process step.
3. The flowchart symbols are linked together with arrows showing the process flow direction.
4. A flowchart typically shows the flow of data in a process.
5. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.
6. Flowcharts are generally drawn in the early stages of formulating computer solutions.
7. Flowcharts often facilitate communication between programmers and business people.

Example: - Draw a flowchart to find the largest among three different numbers entered by the user.



Advantages of Using Flowcharts

1. Communication: a better way of logical communicating.
2. Effective analysis: the problem can be analyzed in more effective way.
3. Proper documentation.
4. Efficient Coding: a guide or blueprint during the systems analysis and program development phase. Proper Debugging: helps in debugging process.
5. Efficient Program Maintenance: easy with the help of flowchart.

Limitations of Using Flowcharts

1. Complex logic: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. Alterations and Modifications: If alterations are required the flowchart may require redrawing completely.
3. Reproduction: As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

Flowchart Symbols

1. Terminator: ""tart" or "End"
2. Process
3. Decision: Yes/No question or True/False
4. Connector: jump in the process flow
5. Data: data input or output (I/O)
6. Delay
7. Arrow: flow of control in a process.

Symbol	Purpose	Description
	Flowline	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of the flowchart.
	Input/output	Used for input and output operation.
	Processing	Used for arithmetic operations and data- manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect flowchart portion on a different page.
	Predefined Process/Function	Used to represent a group of statements performing one processing task.