Unit 1

Fundamentals

What is an Algorithm?

It is a Step by step solution to a particular problem which is a Well defined Computational Procedure that takes some value or set of values as input and produce an optimized and efficient output.

It can also be viewed as a tool for solving well defined problem. Here Well defined means a problem giving proper and required set of values which is solvable in some unit time. A Procedure can be of Infinite time but Algorithm should be finite in time.

Problem ⟶ **Solution using some efficient solution** ⟶ **Implement**

Properties of an Algorithm

Input: The input must be specified. That is, the inputs that we are going to take through our algorithm must come from a specified set of elements, and the type and amount of inputs must be specified.

Output: The output must also be specified. The algorithm must specify what the output should be, and how it is related to the inputs.

Definiteness: The steps in the algorithm must be definite. In other words, the steps need to be clearly defined and detailed. The action of each step must be specified.

Effectiveness: The steps in the algorithm must be effective, so they must be doable.

Finiteness: The algorithm must be finite. That is, the algorithm must come to an end after a specific number of steps.

Unambiguousness - Every algorithm should be Unambiguous i.e. it should be Unique for a particular problem.

Programming Language :- It should not be a Programming Language dependent.

Relativeness - All steps of algorithm should be related to each other.

Instances of a problem

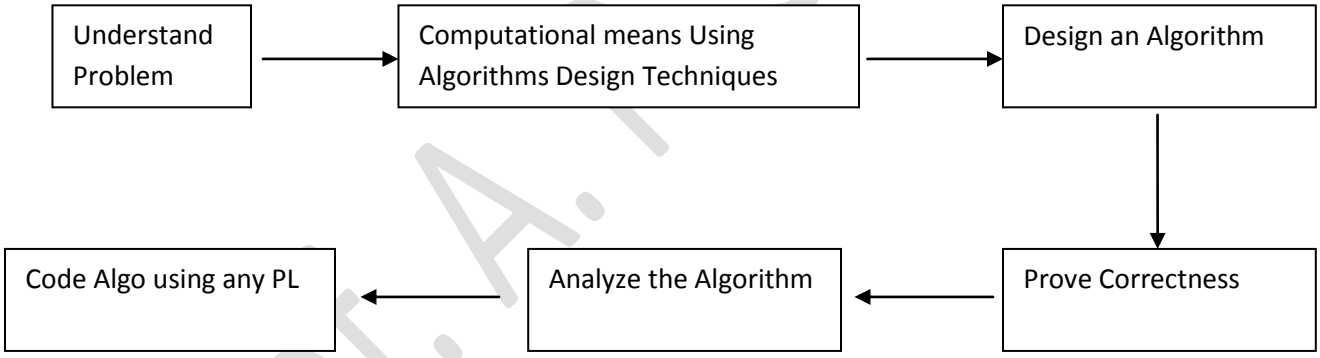
• An algorithm is said to be correct if for every input instance, it halts with the correct output

• An instance of a problem consists of all inputs needed to compute a solution to the problem

• A correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some input instance, or it might halt with other than the desired answer

Algorithms as a technology

In above mentioned flow, the Algorithm can get used in computer i.e for complex problems as well. Some problems are computer based which can get solved using some programming language. Efficiency of algorithm in terms of Time and Space is measured only.



The Need for Analysis of Algorithm

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis –

Worst-case – 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 – An average number of steps taken on any instance of size a.

Amortized – A sequence of operations applied to the input of size a averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa.

Correctness of Algorithm

- Experimental Analysis , Correctness is proved using Testing method.

- Formal Analysis - Actual proving of algorithm.

The change in the behavior of algorithm as per change in size of input is known as *Order of Growth*. According to the growth of algorithm the correctness of algorithm can be proved using *Invariant Approach*. This is the condition which needs to be TRUE for every executing statement(Before and After Execution of Loop)

It has Three Steps

- Initialization

- Maintenance

- Termination

Ex. Bubble Sort / Merge Sort

Iterative Algorithm

- What is Iterative Algoriothm?
  It is a function which Loops to repeat some part of the Code.
  Eg. For Loop is responsible for repetition of a certain block of code.
- What is Recursion?
  It is function wehich calls itself again and again to repeat the code.
  Eg. College time table

- What is Difference between Iterative and Recursion?

Iterative	Recursion
• It is Faster in Execution • It requires Less Time and Space • It solves only Simple Problems • To solve problems, it uses simple loops.	• It is Slower in Execution • It requires More Time and Space • It solves Complex Problems eg. Tower of hannoi • To solve problems by repeating procedures again and again.

Note :- Iterative Algorithm are useful when they are designed through proper channel which are mentioned as follows. (Iterative algorithm Design Issues)

A) Use of Loops

As we break down algorithm into sub-algorithms, sooner or later we shall come across some iterative construct, that is, a loop, apart from conditional statements. Experience tells us that this is a stumbling block for many novice programmers. A loop executing one more or one less time than the intended number is so common that people have stopped raising eyebrows. So we should discuss it in detail.

To construct a loop we should be aware of three aspects:

1. The initial condition that needs to be true before the loop begins execution

2. The invariant relation that must hold before, during, and after each iteration of the loop

3. The condition under which the loop must terminate

The following design process is suggested to take care of these three aspects of algorithm development.

To Establish Initial Condition Set the loop control variables, to values which are appropriate for solving the smallest instance of the problem in question. For example, suppose we want to add elements of the given array using a loop. The loop variables are, i the loop control variable which is also used as the array index and s, the current value of the sum. The smallest problem in this case is the sum of 0 number of elements. In this case s must be 0. Thus the initial condition is:

i ← 0

s ← 0

To Find the Iterative Construct Try to extend the smallest problem to the next smallest. In doing this, we will come to know what changes are to be done to the loop variables to achieve this change. For our example of summation of elements of an array, the next smallest is the case of n = 1, in which case smust be equal to a[i]. Thus the solution for n = 1 can be derived from the solution for n = 0 by first considering the solution for i = 1:

i ← 1

s ← a[1]

Then the same can be re-written as:

i ← i + 1 note: i on RHS is 0

s ← s + a[i] i is now 1

This is the general solution for any n > 0. Thus, in terms of the structured control constructs, the complete algorithm is:

```
i ← 0

s ← 0

while i < n do

    i ← i + 1

    s ← s + a[i]

endwhile
```

Loop Termination The simplest termination condition occurs when it is known in advance the number of times the loop is to be iterated. In that case we can use a FOR structured programming construct. For example, to execute a loop 20 times, we may write:

```
for i ← 1 to 20 do

    ...

    ...

endfor
```

Note that it is not necessary that the initial value assigned and the final value are constants, the only requirement is that their values should be known at the start of the loop. Thus we may have:

```
for i ← m to n

    ...

    ...

endfor
```

assuming that the values of m and n are known. Another possibility is that the loop terminates when some condition becomes false.

For example:

```
while (x > 0) and (x < 10) do

    ...
```

...

end while

For loops of this type we cannot directly predict in advance the number of times the loop will iterate before it terminates. In fact, there is no assurance that a loop of this type will terminate at all. The responsibility to achieve it rests on the algorithm designer. This is one way an error can creep into an algorithm and is one topic in algorithm correctness.

Key Points:- For use of loops, following conditions must get satisfied.

1. The initial Condition that needs to be TRUE before starting the Loop.

2. It should have Invariant relations.

3. The loops must be able to get terminated.

B) Efficiency of Algorithms

The design and implementation of algorithms have a profound influence on their efficiency. Every algorithm, when implemented must use some of the system resources to complete its task. The resources most relevant to the efficiency are the use of the central processor unit (CPU) time and internal memory (RAM). In the past, high cost of the computer resources was the driving force behind the desire to design algorithms that are economical in the use of CPU time and memory. As time passed, while on one hand, the cost of these resources has reduced and continues to do so, the requirement of more complex, and thereby time consuming algorithms has increased. Hence the need for designing efficient algorithms is present even today.

There is no standard method for designing efficient algorithms. Despite this, there are a few generalizations that can be made about the problem characteristics, while other characteristics would be specific to a particular problem. We shall discuss some means of improving the efficiency of an algorithm.

B)a) Removing Redundant Computations Outside Loops

Most of the inefficiencies that creep into the implementation of algorithms are due to redundant computations or unnecessary storage. The effect of redundant computation is serious when it

is embedded within a loop, which must be executed many a times. The most common mistake when using loops is to repeatedly re-calculate the part of an expression that remains constant throughout the execution phase of the entire loop. Let us take an example:

```
x = 0;
for i = 1 to N do
begin
    x = x + 0.01;
   y = (a * a * a + c) * x * x + b * b * x;
 writeln(' x = ', x, ' y = ', y);
end
```

This loop does twice the number of multiplications (computations) necessary to arrive at the answer. How can the unnecessary multiplications and computations be removed? Declare two new constants a3c and b2 before execution of the loop.

```
a3c = a * a * a + c;
b2 = b * b;
x = 0;
for i = 1 to N do
begin
   x = x + 0.01;
   y = a3c * x * x + b2 * x;
  writeln('x =', x, 'y =', y);
end
```

In this example, the saving is not all that significant, but in actual commercial as well as scientific applications such improvements are known to improve the performance by a reasonably good factor. Also, utmost care should be exercised in making sure that redundancies are removed in the innermost loops, as they are a major contributor to cost in terms of time.

B)b) Referencing of Array Elements

Generally, arrays are processed by iterative constructs. If care is not exercised while programming, redundant computations can creep into array processing. Consider, as an example, the following two versions of an algorithm in C, to find the maximum number and its position in an array of numbers:

Version I

```
  p = 0;
for(i = 1; i < n; i++){
    if(a[i] > a[p]){
 p = i;
 }
 }
 max = a[p]; /* p is the position of max */
```

Version II

```
  p = 0; i = 0;
 max = a[i];
for(i = 1; i < n; i++){
 if(a[i] > max){
        max = a[i];
  p = i;
```

```
    }
}
```

Which of the two versions of the algorithm is preferable? Why?

Note that indexing of any kind, whether it be of simple data elements or of structured elements requires more time. Thus it can be said that the second version of the implementation is preferable because the condition test (that is, a[i] > max) which is the dominant operation is much more efficient to perform than the corresponding test in the version I of the program. By using the variable max, only one array reference is being made each time, whereas when using the variable p as index, two array references are required. References to array elements requires address arithmetic to arrive at the correct element and this requires time. Thus more time is spent in locating the specific value for comparison. Secondly, the second version of the program is better documented and the purpose of the code is much more explicit to the reader. This is because, by adding the variable called max we are providing a clue to the reader of the program about its operation.

C) Inefficiency Due to Late Termination

Another possibility of inefficiency creeping into the implementation of an algorithm is, when considerably more tests are carried out, than are required to solve the problem at hand. The following is a good example: Suppose we have to search an alphabetically ordered list of names for a particular name using linear search.

An inefficient implementation for this case would be one where all names were examined even if a node in the list was reached where it could be definitely said that the name cannot occur later in the list. For example, suppose we are searching for Ketan, then, as soon as we reach a name that is alphabetically later than Ketan example, Lalit, we should not proceed further.

Inefficient Algorithm

1. While Name sought Current Name and not(EOF) do get name from list. Where EOF denotes End of File.

Efficient Algorithm

1. While Name sought > Current Name and not (EOF) do get name from list

2. Test if current name is equal to the name sought.

Let us take as an example, the Bubble Sort algorithm (sorting in ascending order):

Establish the array[1 . . . N] of N elements

While the array is still not sorted Do

Set the order indicator Sorted to True

For all adjacent pair of elements in the unsorted part of the array do

if the current adjacent pair is not in non-descending order then

Exchange the elements of the pair

Set sorted to false

Return sorted array

D) Early Detection of desired Output Conditions

It may sometimes happen that, due to the very nature of the input data a particular algorithm, for example, the Bubble Sort algorithm, establishes the desired output condition before the general condition for termination is met. For example, a bubble sort algorithm might receive a set of data that is already in sorted condition. When this is the case, it is obvious that the algorithm will have the data in sorted condition long before the general loop termination conditions are satisfied. It is therefore desirable to terminate the sorting as soon as it is established that the input data is already sorted. How do we determine during the course of the execution of the algorithm, whether the data is already sorted? To do this, all we need to do is to check whether there have been any exchanges in the current pass of the inner loop. If there have been no exchanges in the current pass, the data must be already sorted and the loop can be terminated early.

In general we must include additional tests or steps to determine or detect the conditions for early termination. These tests should be included only if they are inexpensive with regards to computer time as was the case in the Bubble sort algorithm. In general, it may be necessary to trade-off extra tests and storage versus CPU time to achieve early termination of algorithms.

E) Estimating and Specifying Execution Times

To arrive at a quantitative measure of an algorithm's performance it is necessary to set up a computational model that reflects its behavior under specified input conditions. This model must deal with the essence (the heart) of computation and at the same time be independent of any specific programming language.

Thus we usually characterize an algorithm's performance in terms of the size of the problem being solved. It is obvious that more computing resources are required to solve larger problems in the same class of problems.

An important question is "How does the "COST" of solving a problem vary as N increases". An intuitive response is "The cost increases linearly with an increase in N". (example, a problem with n = 200 takes twice as much time as a problem with n = 100). While this linear dependence on N is justifiable for some simple problems or algorithms, in general the relationship between complexity and the problem size follows a completely different pattern. At the lower end of the scale we have algorithms with logarithmic (that is, better) dependence on N, while on the other end of the scale we have algorithms with an exponential dependence on N. With an increase in N the relative difference in the cost of computation is enormous for these two extremes.

F) Order Notation

The Order Notation is a standard notation developed to denote the computing time (or space) of algorithms and bounds on the upper limit or the lower limit on the computing time (or space) of algorithms. It refers collectively to the Big-Oh, the Theta, the Omega and the Small-Oh notations. An algorithm in which the dominant mechanism (one which is

most executed) does not exceed c * n2times, where c is a constant and n is the problem size, is said to have an order n2 complexity which is expressed as O(N2).