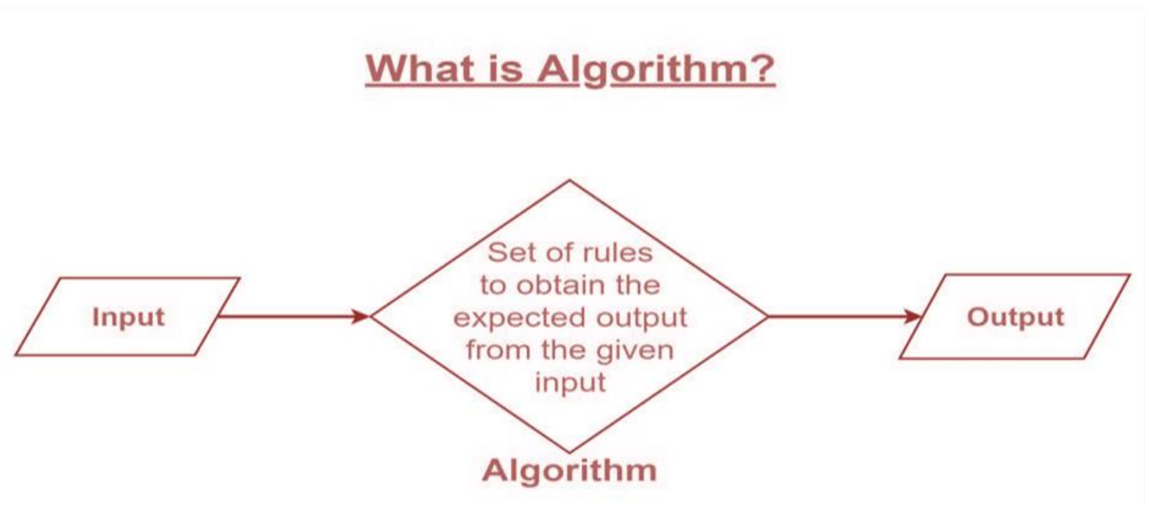
Introduction

The word Algorithm means" A set of finite rules or instructions to be followed in calculations or other problem-solving operations"

Solving a mathematical problem in a finite number of steps that frequently involves recursive operations".

Therefore, Algorithm refers to a sequence of finite steps to solve a particular problem.


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

Use of the Algorithms:

1. Computer Science: Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
2. Mathematics: Algorithms are used to solve mathematical problems,

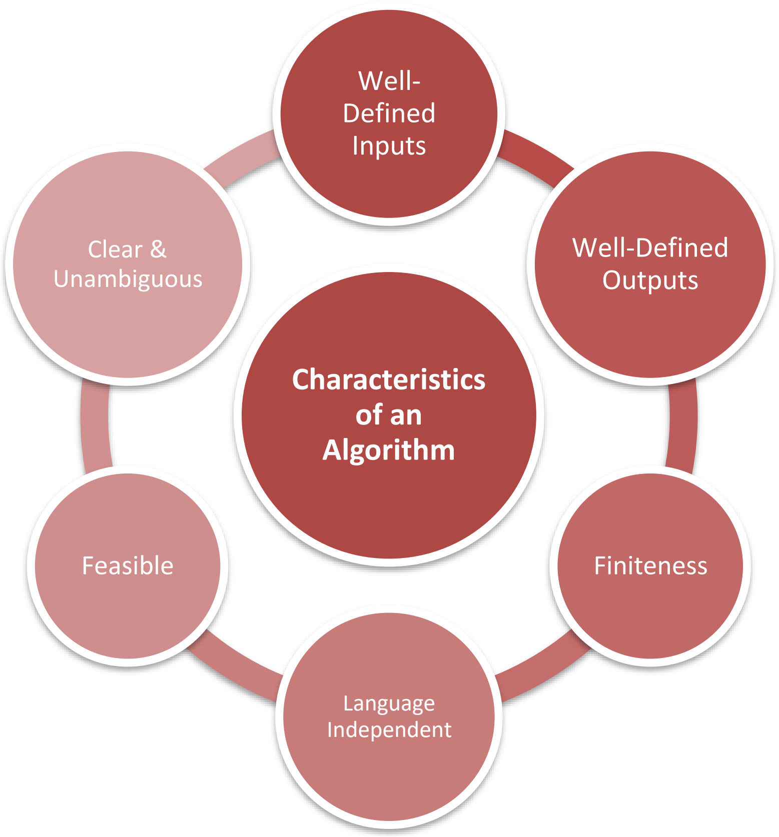
such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.

3. Operations Research: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.
4. Artificial Intelligence: Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision- making.
5. Data Science: Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

What is the need for algorithms?

1. Algorithms are necessary for solving complex problems efficiently and effectively.
2. They help to automate processes and make them more reliable, faster, and easier to perform.
3. Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
4. They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

What are the Characteristics of an Algorithm?



For some instructions to be an algorithm, it must have the following characteristics:

- **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well- defined inputs. It may or may not take input.

- **Well-Defined Outputs**: The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.

- **Finite-ness**: The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible**: The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- Language Independent: The Algorithm designed must be language-independent,
  i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
- **Input**: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.
- **Output**: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.
- **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.
- **Finiteness**: An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
- **Effectiveness**: An algorithm must be developed by using very

basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do

some work.

Types of Algorithms:

- Brute Force Algorithm
- Recursive Algorithm
- Backtracking Algorithm
- Searching Algorithm:
- Divide and Conquer Algorithm

Advantages of Algorithms:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In an Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actualprogram.

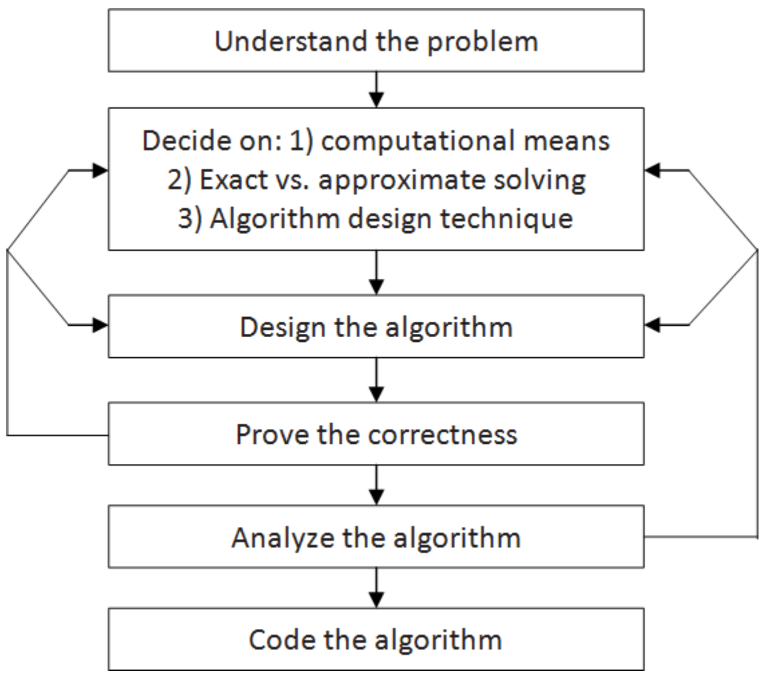
Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms

Fundamentals of Algorithmic Problem Solving:

A sequence of steps involved in designing and analyzing an algorithm is shown in the figure below.

Figure: Flow diagram of designing and analyzing an algorithm



1.Understanding the Problem

This is the first step in designing of algorithm.

Read the problem's description carefully to understand the problem statement completely.Ask questions for clarifying the doubts about the problem.

Identify the problem types and use existing algorithm to find solution.Input (instance) to the problem and range of the input get fixed.

(b) Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly and solving it approximately.

An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.

If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. ie., produces an approximate answer. E., extracting square roots, solving non linear equations, and evaluating definite integrals.

(c) Algorithm Design Techniques

An algorithm design technique (strategy) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Analyzing an Algorithm

a. For an algorithm the most important is efficiency. In fact, there are two kinds of algorithmefficiency. They are:

b. Time efficiency, indicating how fast the algorithm runs, and

c. Space efficiency, indicating how much extra memory it uses.

d. The efficiency of an algorithm is determined by measuring both time efficiency and spaceefficiency.

e. So factors to analyze an algorithm are:

1.Time efficiency of an algorithm 2.Space efficiency of an algorithm 3.Simplicity of an algorithm

2.Coding an Algorithm

a. The coding / implementation of an algorithm is done by a suitable programming languagelike C, C++, JAVA.

b. The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power shouldnot reduce by inefficient implementation.

c. Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common sub-expressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.

d. Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude.

e. It is very essential to write an optimized code (efficient code) to reduce the burden of compiler. Role of algorithms in computing The most important problem types related to computing are:

Fundamental of The Analysis Of Algorithm Efficiency

Complexity:

- The Performance of a computer program is the amount of the memory and time needed to a run a program
- We study it in terms of complexity of the program
- Complexity of a algorithm is a function which is defined in terms of input size and gives the running time and space required by the algorithm

There are two kinds of efficiency:

- **Time efficiency**, also called time complexity, indicates how quickly an algorithm runs.
- **Space efficiency**, also known as space complexity, refers to the number of memory units that the algorithm takes, in addition to the space needed for its input and output.
- Amount of memory space required

- The space complexity of an algorithm is the amount of space (memory) it needs till completion.
- The space needed by a program has following components:
- Instruction space
- Data Space
- Environment stack
- S = S(fix) + S(variable)

Example:Adding two numbers

```
Algorithm(A,N)
{
S=0;   -1 Unit for(i=0;i<n;i++)---- n+1
{
   S=S+A[i];-____________________n
}
return s;------------------------------- 1
}
                                   ------------------------------------------
                                        f(n)=2n+3
                                        Time Complexity:O(n)
```

Space Complexity: A ---- $\rightarrow$n

n---- $\rightarrow$1

s---- $\rightarrow$1

i ----$\rightarrow$1

________________-

n+3

S(n)=n+3

Space Complexity(n)

Example 02:

Sum of two matrices

```
Algorithm Add (A,B,n)
{
for(i=0;i<n;i++) ---------------- n+1
{
for(j=0;j<n;j++) ---------------- n*(n+1)
{
C[i,j]=A[i,j]+B[I,j];---------------n*n
                                ----------
```

$F(n)=2\,n^2+2n+1$

Time Complexity=O(n^2)

f(n) 3n2+3

Space Complexity:=O($n^{2)}$

Analysis Framework

1. **Measuring an input size**
2. **Units for measuring runtime**
3. **Orders of Growth**
4. **Worst case, Best case and Average case**
5. **Time Complexity**
6. **Space Complexity**

01. Measuring an input size

An algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

In most cases, selecting such a parameter is quite straight forward.

For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.

For the problem of evaluating a polynomial p(x) = a n x n+ . . . + a 0 of degree n, it will be the polynomial's degree or the number of its coefficients, which is larger by one than its degree.

There are situations, of course, where the choice of a parameter indicating an input size does matter.

Example - computing the product of two n-by-n matrices.

02. Units for measuring runtime

- We can simply use some standard unit of time measurement-a second, a millisecond, and so on-to measure the running time of a program implementing the algorithm.

- There are obvious drawbacks to such an approach. They are dependence on the speed of a particular computer

- Dependence on the quality of a program implementing the algorithm the compiler used in generating the machine code

- The difficulty of clocking the actual running time of the program.

- Since we are in need to measure algorithm efficiency, we should have a metric that does not depend on these extraneous factors.

- One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both difficult and unnecessary.

- The main objective is to identify the most important operation of the algorithm, called the basic operation,
- The operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

Time Complexity:

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

Definition

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

Space complexity:

Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

We often speak of extra memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

We can use bytes, but it's easier to use, say, the number of integers used, the number of fixed-sized structures, etc.

In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.

Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important issue as time complexity

Worst Case Analysis:
Most of the time, we do worst-case analyses to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

Average Case Analysis

The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

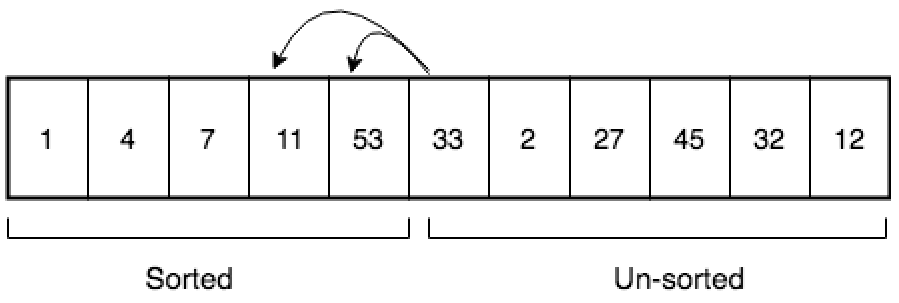
Best Case Analysis
The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn’t provide any information as in the worst case, an algorithm may take years to run.

Consider the example of Linear Search where we search for an item in an array. If the item is in the array, we return the corresponding index, otherwise, we return -1. The code for linear search is given below.

```
int search(int a, int n, int item) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] == item) {
            return a[i]
        }
    }
    return -1
}
```

Variable a is an array, n is the size of the array and item is the item we are looking for in the array. When the item we are looking for is in the very first position of the array, it will return the index immediately.

The for loop runs only once. So the complexity, in this case, will be O(1). This is the called the best case.
Consider another example of insertion sort. Insertion sort sorts the items in the input array in an ascending (or descending) order. It maintains the sorted and un-sorted parts in an array. It takes the items from the un-sorted part and inserts into the sorted part in its appropriate position. The figure below shows one snapshot of the insertion operation.



In the figure, items [1, 4, 7, 11, 53] are already sorted and now we want to place 33 in its appropriate place.

The item to be inserted are compared with the items from right to left one-by-one until we found an item that is smaller than the item we are trying to insert.

We compare 33 with 53 since 53 is bigger we move one position to the left and compare 33 with 11. Since 11 is smaller than 33, we place 33 just after 11 and move 53 one step to the right.

Here we did 2 comparisons. It the item was 55 instead of 33, we would have

Performed only one comparison.

That means, if the array is already sorted then only one comparison is necessary to place each item to its appropriate place and one scan of the array would sort it. The code for insertion operation is given below.

```
void sort(int a, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
       j = i-1;
       key = a[i];
       while (j >= 0 && a[j] > key)
       {
           a[j+1] = a[j];
           j = j-1;
       }
       a[j+1] = key;
    }
}
```

When items are already sorted, then the while loop executes only once for each item
There are total n items, so the running time would be O(n)
So the best case running time of insertion sort is O(n)
The best case gives us a lower bound on the running time for any input.

If the best case of the algorithm is O(n) then we know that for any input the program needs *at least* O(n) time to run. In reality, we rarely need the best case for our algorithm. We never design an algorithm based on the best case scenario.

Worst Case Analysis

In real life, most of the time we do the worst case analysis of an algorithm. Worst case running time is the longest running time for any input of size n.

In the linear search, the worst case happens when the item we are searching is in the last position of the array or the item is not in the array.

In both the cases, we need to go through all n items in the array. The worst case runtime is, therefore, O(n). Worst case performance is more important than the best case performance in case of linear search because of the following reasons.

1. The item we are searching is rarely in the first position. If the array has 1000 items from 1 to 1000. If we randomly search the item from 1 to 1000, there is 0.001 percent chance that the item will be in the first position.
2. Most of the time the item is not in the array (or database in general). In which case it takes the worst case running time to run.

Similarly, in insertion sort, the worst case scenario occurs when the items are reverse sorted. The number of comparisons in the worst case will be in the order of n2 2 and hence the running time is O(n2)

Average Case Analysis

Sometimes we do the average case analysis on algorithms. Most of the time the average case is roughly as bad as the worst case.

In the case of insertion sort, when we try to insert a new item to its appropriate position, we compare the new item with half of the sorted item on average.

The complexity is still in the order of n2 2 which is the worst-case running time.

It is usually harder to analyze the average behavior of an algorithm than to analyze its behavior in the worst case.

This is because it may not be apparent what constitutes an “average” input for a particular problem.

A useful analysis of the average behavior of an algorithm, therefore, requires a prior knowledge of the distribution of the input instances which is an unrealistic requirement.

Therefore often we assume that all inputs of a given size are equally likely and do the probabilistic analysis for the average case.

Order of growth:

It is described by the highest degree term of the formula for running time. (Drop lower-order terms. Ignore the constant coefficient in the leading term.)

Example: We found out that for insertion sort the worst-case running time is of the form an2 + bn + c.

Drop lower-order terms. What remains is an2.Ignore constant coefficient. It results in n2.But we cannot say that the worst-case running time T(n) equals n2 .Rather It grows like n2 . But it doesn’t equal n2.We say that the running time is Θ (n2) to capture the notion that the order of growth is n2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.