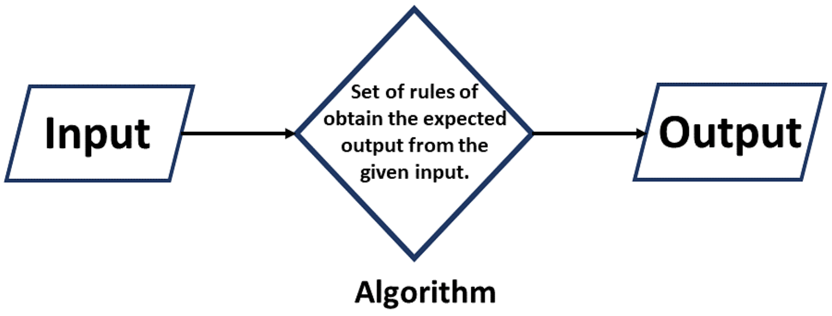
UNIT -1

INTRODUCTION

What is an algorithm?

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

- An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.
- According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task.
- It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudocode.


Input
Set of rules of obtain the expected output from the given input.
Output
Algorithm

- Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.
- Algorithm: An algorithm is defined as a step-by-step process that will be designed for a problem.
- Input: After designing an algorithm, the algorithm is given the necessary and desired inputs.

- Processing unit: The input will be passed to the processing unit, producing the desired output.
- Output: The outcome or result of the program is referred to as the output.

What is the Need for Algorithms?

- You require algorithms for the following reasons:
- Scalability
- It aids in your understanding of scalability. When you have a sizable real-world problem, you must break it down into small steps to analyze it quickly.
- Performance
- The real world is challenging to break down into smaller steps. If a problem can be easily divided into smaller steps, it indicates that the problem is feasible.

TYPES OF ALGORITHMS

1. Brute Force Algorithm: A straightforward approach that exhaustively tries all possible solutions, suitable for small problem instances but may become impractical for larger ones due to its high time complexity.

2. Recursive Algorithm: A method that breaks a problem into smaller, similar subproblems and repeatedly applies itself to solve them until reaching a base case, making it effective for tasks with recursive structures.

3. Encryption Algorithm: Utilized to transform data into a secure, unreadable form using cryptographic techniques, ensuring confidentiality and privacy in digital communications and transactions.

4. Backtracking Algorithm: A trial-and-error technique used to explore potential solutions by undoing choices when they lead to an incorrect outcome, commonly employed in puzzles and optimization problems.

5. Searching Algorithm: Designed to find a specific target within a dataset, enabling efficient retrieval of information from sorted or unsorted collections.

6. Sorting Algorithm: Aimed at arranging elements in a specific order, like numerical or alphabetical, to enhance data organization and retrieval.

7. Divide and Conquer Algorithm: Breaks a complex problem into smaller subproblems, solves them independently, and then combines their solutions to address the original problem effectively.

8. Greedy Algorithm: Makes locally optimal choices at each step in the hope of finding a global optimum, useful for optimization problems but may not always lead to the best solution.

FUNDAMENTALS OF ALGORITHM PROBLEM SOLVING

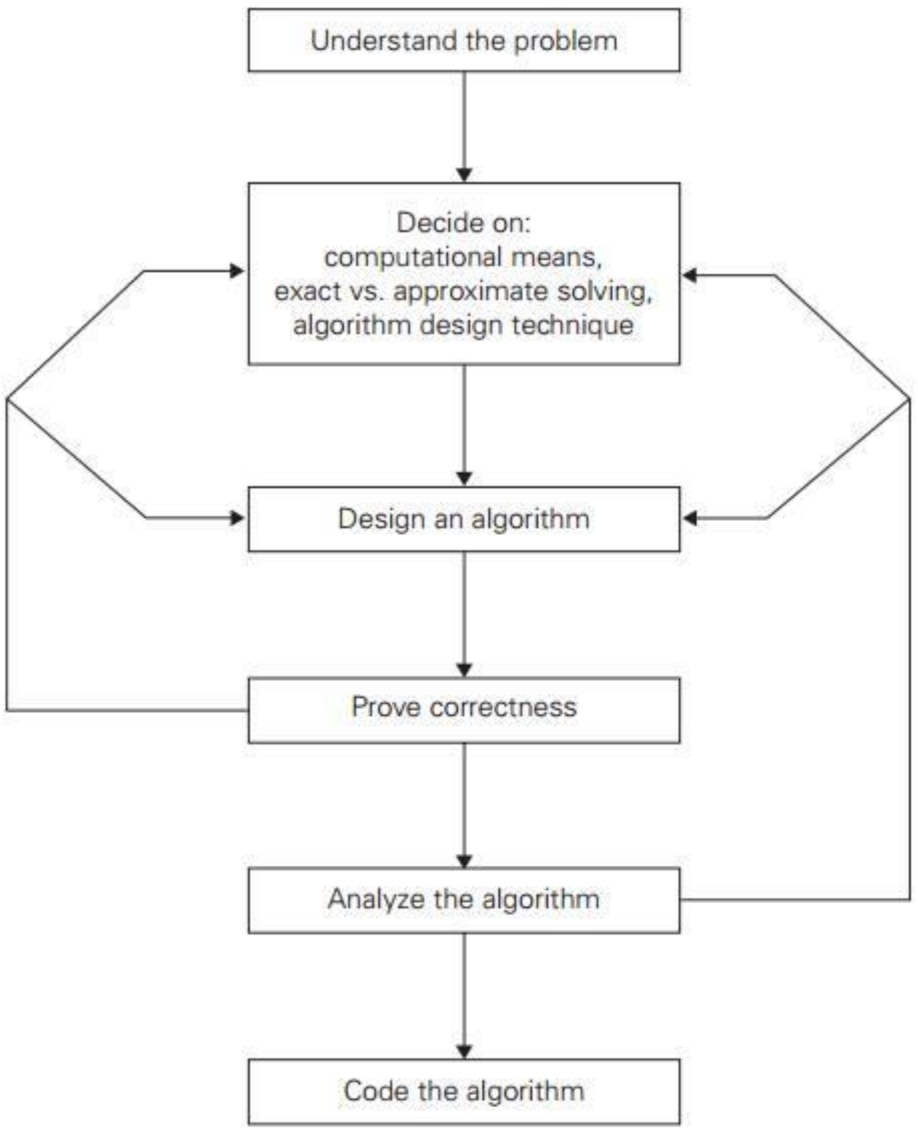


FIGURE 1.2 Algorithm design and analysis process.

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

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.

Ascertaining the Capabilities of the Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. The vast majority of

algorithms in use today are still destined to be programmed for a computer.

The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***

Choosing between Exact and Approximate Problem Solving

→The next principal decision is to choose between solving the problem exactly or 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. i.e., produces an

→Approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

Algorithm Design Techniques

What is an algorithm design technique?

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

• Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.

• Implementation of algorithm is possible only with the help of Algorithms and Data Structures

• Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and soon.

Designing an Algorithm and Data Structures

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question

Methods of Specifying an Algorithm There are three ways to specify an algorithm.

They are: a. Natural language

b. Pseudocode

Natural Language :

It is very simple and easy to specify an algorithm using natural language.

But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers.

Step 1: Read the first number, say a.

Step 2: Read the first number, say b.

Step 3: Add the above two numbers and store the result in c.

Step 4: Display the result from c

Such a specification creates difficulty while actually implementing it.

Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b) Pseudocode:

• Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.

• For Assignment operation left arrow "←", for comments two slashes "//",if condition, for, while loops are used. Natural Language Pseudocode

```
ALGORITHM
Sum(a,b)
//Problem Description: This algorithm performs addition of two numbers
//Input: Two integers a and b
//Output: Addition of two integers
c←a+b
return c
```

PROVING AN ALGORITHM'S CORRECTNESS

• Once an algorithm has been specified then its correctness must be proved

• An algorithm must yield a required result for every legitimate input in a finite amount of time.

For Example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n)

• A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

• The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The error produced by the algorithm should not exceed a predefined limit.

ANALYZING AN ALGORITHM

• For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are: • Time efficiency, indicating how fast the algorithm runs, and

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

• The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
• **So factors to analyze an algorithm are:**

▪ Time efficiency of an algorithm

Space efficiency of an algorithm

▪ Simplicity of an algorithm

▪ Generality of an algorithm

(VI) CODING AN ALGORITHM

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

• 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 should not reduce by in efficient implementation.

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

• 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. But once an algorithm is selected, a 10–50% speedup may be worth an effort.

• It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

FUNDAMENTALS OF ANALYSIS OF ALGORITHM EFFICIENCY

Fundamentals of Analysis of Algorithm:

1 Analysis of Framework

2 Measuring an input size

3 Units for measuring runtime

4 Worst case, Best case and Average case

5 Asymptotic Notations

1 ANALYSIS FRAME WORK

♦ there are two kinds of efficiency

♦ **Time efficiency** - indicates how fast an algorithm in question runs.

♦ **Space efficiency** - deals with the extra space the algorithm requires

2 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 straightforward.

- ♦ 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 $p(x) = a_n x^n + \dots + a_0$ 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.

  - ♦ There are two natural measures of size for this problem.

    - ♦ The matrix order n.

    - ♦ The total number of elements N in the matrices being multiplied.

- ♦ Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of the two measures we use.

3 UNITS FOR MEASURING RUN TIME:

- 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.

4 WORST CASE, BEST CASE AND AVERAGE CASE EFFICIENCIES

- It is reasonable to measure an algorithm's efficiency as a function of a parameter indicating the size of the algorithm's input.

- But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.

- **Example, sequential search**. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

- Here is the algorithm's pseudo code, in which, for simplicity, a list is implemented as an array. (It also assumes that the second condition A[i] i= K will not be checked if the first one, which checks that the array's index does not exceed its upper bound, fails.)

WORST CASE:

```
Sequential Search(A[0..n -1], K)

    //Searches for a given value in a given array by sequential search //Input: An
    array A[0..n -1] and a search key K

    //Output: Returns the index of the first element of A that matches K // or -1
    ifthere are no matching elements

    i←0

    while i < n and A[i] ≠ K do

        i←i+1

    if i < n return i else return -1
```

- The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

- ♦ In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$\mathbf{C_{worst}\ (n) = n.}$$

- ♦ The way to determine is quite straightforward

- ♦ To analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst-case value $\mathbf{C_{worst}\ (n)}$

- ♦ The worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size n, the running time will not exceed $\mathbf{C_{worst}\ (n)}$ its running time on the worst-case inputs.

BEST CASE EFFICIENCY

- ♦ The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.

- ♦ We can analyze the best case efficiency as follows.

- ♦ First, determine the kind of inputs for which the count C (n) will be the smallest among all possible inputs of size n. (Note that the best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.)

- ♦ Then ascertain the value of C (n) on these most convenient inputs.

- ♦ Example- for sequential search, best-case inputs will be lists of size n with their first elements equal to a search key; accordingly, $C_{best}(n) = 1$.

AVERAGE CASE EFFICIENCY

- To analyze the algorithm's average-case efficiency, we must make some assumptions about possible i*n*puts of size n.

- The investigation of the average case efficiency is considerably more difficult than investigation of the worst case and best case efficiency.

- It involves dividing all instances of size n .into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same.

- let us consider again sequential search. The standard assumptions are,

- In the case of a successful search, the probability of the first match occurring in the ith position of the list is pin for every i, and the number of comparisons made by the algorithm in such a situation is obviously i.

- In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being (1 - p). Therefore,

$$C_{avg}(n) = \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \dots + i \cdot \frac{p}{n} + \dots + n \cdot \frac{p}{n} \right] + n \cdot (1 - p)$$

$$= \frac{p}{n} [1 + 2 + 3 + \dots + i + \dots + n] + n(1 - p)$$

$$= \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1 - p)$$

$$= \frac{p(n+1)}{2} + n(1 - p)$$

- ♦ Example, if p = 1 (i.e., the search must be successful), the average number of key comparisons made by sequential search is (n + 1) /2.

- ♦ If p = 0 (i.e., the search must be unsuccessful), the average number of key comparisons will be n because the algorithm will inspect all n elements on all such inputs.

ORDERS OF GROWTH

The order of growth of an algorithm is an approximation of the time required to run a computer program as the input size increases. The order of growth ignores the constant factor needed for

Order	**Big-Theta**	**Example**
Constant	$\Theta(1)$	Indexing an item in a list
Logarithmic	$\Theta(\lg N)$	Repeatedly halving a number
Linear	$\Theta(n)$	Summing a list
Quadratic	$\Theta(n^2)$	Summing each pair of numbers in a list
Exponential	$\Theta(2^n)$	Visiting each node in a binary tree

fixed operations and focuses instead on the operations that increase proportional to input size. For example, a program with a linear order of growth generally requires double the time if the input doubles.

The order of growth is often described using either Big-Theta or Big-O notation, but that notation is out of scope for this course.

This table summarizes the most common orders of growth:

CONSTANT TIME

When an algorithm has a constant order of growth, it means that it always takes a fixed number of steps, no matter how large the input size increases

LOGARITHMIC TIME

When an algorithm has a logarithmic order of growth, it increases proportionally to the logarithm of the input size.

The binary search algorithm is an example of an algorithm that runs in logarithmic time.

LINEAR TIME

When an algorithm has a linear order of growth, its number of steps increases in direct proportion to the input size.

The aptly-named linear search algorithm runs in linear time.

QUADRATIC TIME

When an algorithm has a quadratic order of growth, its steps increase in proportion to the input size squared.

Several list sorting algorithms run in quadratic time, like selection sort. That algorithm starts from the front of the list, then keeps finding the next smallest value in the list and swapping it with the current value.

EXPONENTIAL TIME

When an algorithm has a superpolynomial order of growth, its number of steps increases faster than a polynomial function of the input size.

An algorithm often requires superpolynomial time when it must look at every permutation of values. For example, consider an algorithm that generates all possible numerical passwords for a given password length.