UNIT I

CHAPTER 1

Algorithms and Problem Solving

Syllabus

Algorithm : The Role of Algorithms in Computing - What are algorithms, Algorithms as technology, Evolution of Algorithms, Design of Algorithm, Need of Correctness of Algorithm, Confirming correctness of Algorithm - sample examples, Iterative algorithm design issues.

Problem Solving Principles : Classification of problems, Problem solving strategies, Classification of time complexities (linear, logarithmic etc.).

Algorithms are everywhere and are involved in our lives at every moment whether we notice it explicitly or not. To cook a delicious dish we need a correct recipe, to get a passport we need to fulfil defined procedures, to solve any numerical we complete certain computational steps, to manufacture a product it requires typical processing, to cure a disease we obey a prescription and so on. Even to breath, we follow certain steps unknowingly. All these terminologies like a recipe, procedure, process, prescription refer to an algorithm.

1.1 ALGORITHMIC THINKING

- Presently computers are effectively used to perform numerous tasks to improve profitability and proficiency.
- **Computational thinking** is necessary to make use of computers for accomplishing our variety of activities.
- Apart from the basic skills of computer usage, computer professionals should be capable of developing specific computer programs to perform numerous computer-automated tasks.
- Algorithmic thinking is required for writing such computer programs.
- **Algorithmic thinking** is an essential analytical skill for solving any problem. It provides the strategy to get the solution to a problem.
- It is highly needed in the development of effective computer programs to solve computational problems.
- Algorithmic thinking is pertinent in all disciplines of study and not only in the field of computer science.
- Suppose you want to borrow a DAA book from the library. Then before approaching the library, you will analyse the requirements like valid library membership card, correct details of the book (title, author, publication, edition, etc.), and library working hours.
- Based on it you should plan for a certain sequence of steps to borrow a DAA book from the library. If you fail in such algorithmic thinking, you will not get the expected results for your task either you may forget library membership card or may borrow the wrong book or miss the library working hours.
- So, to accomplish any predefined task we need to develop a strategic plan to do it. Accordingly, we have to follow certain steps in a specific order. In this example we will follow the steps below :

(1) Carry your valid library membership card.

(2) Go to the respective library.

(3) Show your library membership card to the library staff and get his/her permission to access the book records.

(4) Search for a required DAA book.

(5) If the book is available, then ask the library staff to issue the book to you.

(6) If the book is not available then fill the requirement slip and submit it to the library staff to register your claim for that book.

Consider another example of developing a banking software. The only syntax of programming languages or some snippets of codes from varied sources are not sufficient to develop the correct computer programs for it.

Unless you apply algorithmic thinking to design a computational model and define the correct workflow of the required banking system you are unable to develop correctly working banking software.

1.2 WHAT IS AN ALGORITHM ?

GQ. Define an algorithm and name two types of algorithmic complexities based on computer resources. **(5 Marks)**

GQ Define an algorithm. Discuss different characteristics of a good algorithm. **(5 Marks)**

- An algorithm is a finite set of unambiguous steps needed to be followed in certain order to accomplish a specific task.
- In the case of computational algorithms, these steps refer to the instructions that contain fundamental operators like +, –, *, /, %, etc.
- Algorithms provide a precise description of the procedure to be followed in certain order to solve a well-defined computational problem.

1.2.1 Characteristics of an Algorithm

UQ. Explain the characteristics of a good algorithm? List out the problems solved by the algorithm.

SPPU - Q. 1(b), March 19, 5 Marks

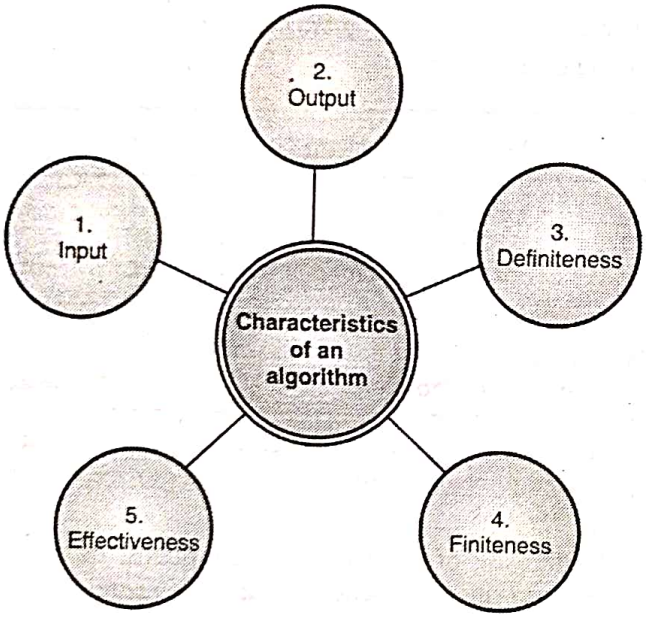
(1) **Input : An algorithm has zero or more inputs.** Each instruction that contains a fundamental operator must accept zero or more inputs.

(2) **Output : An algorithm produces at least one output.** Every instruction that contains a fundamental operator must accept zero or more inputs.

(3) **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.

(4) **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.

(5) **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.



(1A1)**Fig. 1.2.1 : Characteristics of an Algorithm**

- A good algorithm never compromises the precise and unambiguous requirements at every step.
- The same problem can be solved by different algorithmic strategies.
- Also, the same algorithm can be specified in multiple ways.
- The legitimate inputs of an algorithm should be well-specified.

1.2.2 Algorithms as Technology

UQ. Write a short note on the algorithm as a technology with example. **SPPU - Q. 2(b), March 19, 5 Marks**

- Computing time and memory space are limited resources, so we should use them sensibly.
- It is achieved by the usage of efficient algorithms that need less space and time.

Efficiency

- The same problem can be solved by different algorithms. Such algorithms exhibit radical differences in their efficiency.
- Many times, these variances are much more significant than those due to runtime environment (hardware and software).
- E.g., consider the following scenario where Computer1 is 100 times faster than Computer 2. Two different sorting algorithms are executed on these two computers to sort 10^6 numbers.
- It shows that even with the 100 times slower compiler the Computer2 completes the sorting of 10^6 numbers 20 times faster than Computer 1. This is achieved because of the usage of an efficient algorithm heap sort.

	Processing speed	Sorting Algorithm	No. of instructions to sort n numbers	Time to sort $n=10^6$
Comp.1	10^9 instructions per second	Bubble Sort	$2n^2$, where a constant $C_1 = 2$.	$\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^9 \text{ instructions/second}}$ = 2000 seconds
Comp.2	10^7 instructions per second	Heap Sort	50nlogn, where a constant $C_2 = 50$.	$\frac{50 \cdot 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/second}}$ ≈ 100 seconds

Algorithms and other technologies

Many other computer technologies need knowledge of algorithms :

E.g.,

(i) Computer organization and architecture: high-speed hardware with superpipelining, superscalar architectures, graphics processors, parallel architectures etc.

(ii) Networking: LAN (local area network), WAN (wide area network)

(iii) System programming: language processors, operating systems, linkers, loaders etc.

(iv) Web designing.

(v) Image processing: graphics/image/video processing

(vi) Artificial intelligence

(vii) Embedded systems

(viii) Robotics

1.2.3 Evolution of Algorithms

An algorithm is a precisely defined, self-contained sequence of instructions needed to complete a particular job.

- The invention of zero and decimal number systems by ancient Indians gave rise to basic algorithms for number systems, arithmetic operations.
- Ancient Indian Sanskrit Grammarian Panini designed data structures like *Maheshwar Sutra* describing algorithms and rules for Sanskrit phonetics, morphology, language syntax.
- Some theories say that the Babylonian clay tablets (300 BC) were the first algorithms used by the ancient people to record their grain stock and cattle.
- Originally, algorithms were contributed to algebra, calculus etc. by many mathematicians like Euclid, Eratosthenes.
- The term **"Algorithm"** is attributed to Persian mathematician, scientist and astronomer Abu Abdullah Muhammad ibn Musa Al-Khwarizmi. His name was translated in Latin as **"Algorithmic"** from which the word **"Algorithm"** was coined.
- In 1847, English mathematician George Boole designed binary algebra, the base of today's computing logic.
- Decades later, algorithms of the present form came into practice with Alan Turing's computing machine. In 1936, he proposed the concept of **effective procedure**. This steered the evolution of **structured programming**.
- Later there was a concept of the correctness of algorithms. To verify the correctness of algorithms different proof techniques were used.
- As the field of algorithms evolved, it gave rise to the need for efficient algorithms.
- Many algorithmic strategies were proposed like divide and conquer, decrease and conquer, greedy method, dynamic programming, backtracking, branch and bound etc.
- Prof. D. Knuth tossed the term **"algorithm analysis"**. Many researchers studied the efficiency of algorithms by considering time and space trade-off.
- Then the theory of complexity classes was developed based on the **tractability** and **intractability** of problems.
- For many intractable important problems advanced algorithms like approximation algorithms, randomized algorithms are designed.
- Algorithms are used in many applications, in many technologies.
- Currently, many streams of algorithms are developed like genetic algorithms, online algorithms, parallel algorithms, distributed algorithms, optimization algorithms, fuzzy algorithms etc.

1.3 CLASSIFICATION OF PROBLEMS

There are different types of problems in computation. Some of the most important problems are listed below:

- **Searching problems :** These problems include searching of any item or a search key in given data. E.g., retrieving information from large databases, searching an element in the list.
- **Sorting problems :** These problems include rearrangement of items in given data in an ascending or descending order. E.g., arranging names in alphabetical order, ranking internet search results, ranking students as per their CGPA.
- **String processing :** These problems include the computations on strings. A string is defined as a sequence of characters from an alphabet i.e., letters, numbers, and special characters. E.g., String matching problems, string encoding, parsing.

- **Graph problems :** These problems include processing of graphs. A graph is a collection of points (nodes/ vertices) and some line segments (edges) connecting them. E.g., graph-traversal, graph colouring problem, minimum spanning tree (MST) problem.
- **Combinatorial problems :** These problems explicitly or implicitly ask to find a combinatorial object like a permutation, a combination, or a subset that satisfies the specified constraints. A desired combinatorial object may also be needed to possess some additional property such as a maximum or a minimum value. E.g., 8-queens problem, 15-puzzle problem, tiling problem.
- **Geometric problems :** These problems deal with geometric objects like points, lines, and polygons. Computational geometry problems have numerous applications such as computer graphics, tomography, and robotics. E.g., convex-hull problem, closest-pair problem.
- **Numerical problems :** These problems include mathematical objects of continuous nature. They include multiplication problems, computing definite integrals, solving equations and systems of equations, evaluating functions, and so on. E.g., larger integer multiplication, matrix chain multiplication, Gaussian elimination problem.
- Considering the complexity theory, the problems are broadly classified as :
- **Optimization problem :** It is a computational problem that determines the optimal value of a specified cost function. E.g., travelling salesman problem (TSP), optimal binary search tree (OBST) problem, vertex cover problem.
- **Decision problem :** It is a restricted type of a computational problem that produces only two possible outputs ("yes" or "no") for each input. E.g., primality test, Hamiltonian cycle: Whether a given graph has any Hamiltonian cycle in it?
- **Decidable problem :** It is a decision problem which gets the correct "yes" or "no" answer for a given input either in polynomial time or in non-polynomial time. E.g., primality test, Hamiltonian cycle problem
- **Undecidable problem :** It is a decision problem which does not get the correct "yes" or "no" answer for a given input by any algorithm. E.g., halting problem
- **Tractable problem :** It is solved in polynomial time using the deterministic algorithms. E.g., binary search, merge sort
- **Intractable problem :** It cannot be solved in polynomial time using the deterministic algorithms. E.g., knapsack problem, graph colouring problem

1.4 STAGES IN PROBLEM-SOLVING

GQ. State and explain different stages in problem solving. **(4 Marks)**

The following stages are essential to solve any real-world problem :

(1) **Identifying the problem :** To get the solution to any real-world problem we must thoroughly understand the problem and its constraints described in a natural language.

(2) **Designing a computational or mathematical model :** It presents the abstraction of a real-world problem. It removes unnecessary and irrelevant data from the problem description and simplifies it to get a precise computational model.

(3) **Data organization :** The essential data to solve the problem must be organized effectively. We should select an appropriate data structure to store the necessary data.

(4) **Algorithm designing :** By analyzing the problem we should design a finite set of unambiguous steps to get the solution.

(5) **Algorithm specification :** It is the way of describing the algorithmic steps for the programmer. Generally, these steps are written in the form of a pseudo-code and conveyed to the programmer.

(6) **Algorithm validation :** After defining these algorithmic steps, we should validate our logic. It checks whether the algorithm produces the correct output in a finite amount of time for all legal test inputs.

(7) **Analysis of an algorithm :** The performance of the correct algorithm is analyzed and its efficiency is checked by considering different performance metrics like usage of memory, time, etc.

(8) The correctness and the efficiency of an algorithm for all legitimate inputs are verified through mathematical proof.

(9) **Implementation :** By referring to the specifications of a verified algorithm a correct computer program is written using a specific programming language and technology.

(10) **Testing and debugging :** The computer program written for a specific algorithm is executed on a machine. It is tested for all legitimate inputs and is debugged to trace the expected workflow. The performance of an algorithm is tested by the experimentation results.

(11) Documentation : The details of the solved problem, its algorithm, analysis of algorithm, proofs of the correctness of an algorithm, implementation, test cases etc. is well documented for the future applications and research.

1.5 APPLYING DIFFERENT ALGORITHMIC STRATEGIES

GQ. List types of algorithms and classic problems solved by each of them. **(4 Marks)**

- The same problem can be unravelled by different algorithmic strategies. These algorithmic solutions may differ in the requirements of computing resources.
- Thus, the efficiency of an algorithm depends on an algorithmic strategy used to design it.
- Some of the popular algorithmic strategies are as below:
  - **Brute force method :** It enumerates all possible solutions to a given problem without applying any heuristics.
  - **Exhaustive search :** It generates all candidate solutions to a given combinatorial problem and identifies the feasible solution.
  - **Divide and conquer :** It follows three steps as given below:
    - **(i) Divide :** Apply a top-down approach to divide a large problem into smaller and distinct sub-problems.
    - **(ii) Conquer :** Solve the sub-problems independently and recursively by invoking the same algorithm.
    - **(iii) Combine :** Apply a bottom-up approach to combine the solutions of all sub-problems to get a final solution to the original problem.
  - **Greedy method :** It builds a solution in stages. At every stage, it selects the best choice concerning the local considerations.
  - **Dynamic programming :** It is suitable for solving optimization problems with overlapping sub-problems. It applies to problems when the optimal decision sequence cannot be generated through stepwise decisions based on locally optimal criteria.
  - **Backtracking :** It is an algorithmic strategy that explores all solutions to a given problem and abandons them if they are not fulfilling the specified constraints. It follows depth first search (DFS).
  - **Branch and bound :** It is a state-space algorithm where an E-nodes remains an E-node until it is dead.
  - **Exotic algorithms** like genetic algorithms, simulated annealing, online algorithms, parallel algorithms, distributed algorithms, optimization algorithms, fuzzy algorithms, etc.
- By analysing the characteristic nature of the problem, the appropriate algorithmic strategy is to be applied to solve it.

Some of the classic problems that can be solved by different algorithms are as follows.

Algorithmic Strategy	Problems
Brute force method	(1) Sequential search (2) Bubble sort (3) N-queens problem (4) 15-puzzle problem (5) Closest-pair problem (6) Container loading problem
Divide and conquer	(1) Binary search (2) Merge sort (3) Quicksort (4) Large integer multiplication problem (5) Closest-pair problem (6) Finding Max-Min
Greedy algorithms	(1) Fractional knapsack problem (2) Job scheduling problem (3) Activity selection problem (4) Minimum spanning tree problem (5) Single source shortest paths problem (6) Optimal storage on tapes problem (7) Huffman code generation problem

Algorithmic Strategy	Problems
Dynamic programming	(1) Binomial coefficients problem, (2) Optimal Binary Search Tree (OBST) problem (3) 0/1 knapsack problem (4) Matrix chain multiplication problem (4) Multistage graph problem (5) All pairs shortest paths (6) Longest common subsequence problem (7) Travelling salesperson problem
Backtracking	(1) N-queens problem (2) Sum of subsets problem (3) Graph coloring problem (4) Hamiltonian cycle problem (5) 0/1 knapsack problem
Branch and Bound	(1) 0/1 knapsack problem (2) 15-puzzle problem (3) Travelling salesperson problem

1.6 DESIGN OF ALGORITHM

- Designing an efficient algorithm to solve a computational problem is a skill that needs good algorithmic thinking.
- A good algorithm designer needs the knowledge of:
  - Mathematics,
  - Discrete mathematics,
  - Numerical methods for Simulation and Modeling,
  - Computer programming languages,
  - Data structures and file handling,
  - Computer organization and architecture,
  - Database management systems,
  - Systems programming.
- A good algorithm should possess five major characteristics: (1) Input, (2) Output, (3) Definiteness, (4) Finiteness and (5) Effectiveness.
- The same problem can be answered by different algorithmic strategies. These algorithmic solutions may differ in requirements of computing resources.
- Also, the same algorithm can be written in multiple ways.

1.6.1 Basic Steps to Design an Algorithm

(1) Understand and analyse the problem to be solved.

(2) Mention the distinctive name to an algorithm.

(3) Select the appropriate algorithmic strategy to solve the problem by analysing the characteristic nature of the problem.

E.g., If a problem can be divided into independent sub-problems of the same nature then divide and conquer algorithmic strategy can be used to solve it. If a problem has overlapping sub-problems then dynamic programming can be used to solve it.

(4) Identify the legitimate inputs of an algorithm.

(5) Identify the expected output of an algorithm.

(6) Decide the suitable data structure to define inputs and to present the output.

(7) Describe a finite set of well-ordered unambiguous instructions to produce the expected output.

1.6.2 Writing Pseudocode of an Algorithm

- Once the algorithm is designed it can be specified in different forms like flowchart, pseudocode etc.
- Pseudocode is the most suitable way of conveying the algorithmic steps to the programmer.
- **Pseudocode** presents a high-level unambiguous description of an algorithm.
- It does not need the knowledge of the syntax of a specific programming language.
- Since pseudocode represents the algorithm as a sequential composition of fundamental operations, it helps to compute frequency counts of these operations to estimate the running time of an algorithm.
- The algorithm specification by pseudocode facilitates structured programming that helps to easy understanding, debugging, modifications and confirming the correctness of algorithms.

1.6.3 Pseudocode Conventions

- Pseudocodes can be written by following different conventions.
- This book follows the pseudocode convention that is similar to programming language C.
- Comments are given by // (single line) or /*.....*/ (multiline).
- Each collection of simple statements is described as a block enclosed in { }.

Statements have delimiter ;

- Simple data types like integer, float, char, boolean etc. are assumed and hence are not specified explicitly.
- To assign a value to a variable the assignment operator := is used.
- The arithmetic operators are: +, –, *, /, % etc.
- The logical operators are: && (logical and), || (logical or), ¬ (logical not) etc.
- The relational operators are: <, ≤, ≥, ≠, =, >.
- An array is given by [ ].
- The control structures *for, while, repeat-until, if-else, case* have a similar representation and interpretation as per C language.
- Boolean variables have TRUE and FALSE values.
- Input and output are presented by the instructions **read** and **write**.
- Each procedure is written as :

```
Algorithm name (parameter lists)
/* Description, input and output of an algorithm */
{
   body of an algorithm
}
```

1.7 THE CORRECTNESS OF AN ALGORITHM

Once we are done with an algorithm specification, we need to check its **correctness.** By testing the correctness of an algorithm we confirm that the algorithm produces an expected output for all legitimate inputs in a finite amount of time.

1.7.1 Need for the Correctness of an Algorithm

UQ. Why the correctness of an algorithm is important. What is loop Invariant property? Explain with example. **SPPU - Q. 1(c), May 19, 8 Marks**

- For any problem, we first get its correct solution by performing any valid logical computations.
- Once the problem is cracked then only we can think of the better solution to solve the same problem in a more efficient way.
- If an algorithm is incorrect then the efforts for improving its efficiency will be in vain.
- So before improving the efficiency of any algorithm we first confirm its correctness.
- To test the correctness of an algorithm we can give several sets of valid inputs to an algorithm and compare the resulting outputs with known outputs or manually computed results.
- This simple procedure is insufficient and sometimes impractical to check the correctness of an algorithm for all possible inputs.
- Hence it needs rigorous proofs based on mathematical and logical reasoning for confirming the correctness of algorithms.
- Such proofs not only provide us with more confidence in the correct working of our algorithms but also help us to rectify subtle errors in the algorithms.

1.7.2 Confirming Correctness of Algorithm - Sample Examples

UQ. How to confirm the correctness of an Algorithm? Explain with example. **SPPU - Q. 2(a), March 19, 5 Marks**

UQ. Explain the concept of Principle of Mathematical Induction and prove the correctness of an algorithm to find factorial of a number. **SPPU - Q. 1(a), May 18, 6 Marks**

UQ. Explain the concept of PMI and prove the correctness of an algorithm to find factorial of a number using PMI. **SPPU - Q. 3(b), March 19, 5 Marks**

- Tracing the algorithm's working for a specific set of inputs can be a very useful and simple procedure to test whether it produces the desired output, but it cannot prove its correctness always.
- Thus we need to use mathematical proofs to convince the correctness of an algorithm.
- However, to demonstrate the incorrect working of an algorithm, we require just a single input instance for which the algorithm fails.
- For some algorithms, confirming their correctness is quite simple; but for others, it becomes a quite complex procedure.

▶ **The basic steps** for confirming the correctness of an algorithm:

(1) Give the statement or a postcondition to be satisfied. **A postcondition** is a predicate based on an expected output of the algorithm.

(2) Assume the necessary preconditions or assumptions to be TRUE. A **precondition** is a predicate based on a set of valid input to the algorithm.

(3) Apply a chain of logical and mathematical reasoning from preconditions to satisfy the postcondition.

- Different proof techniques can be used to prove the correctness of various algorithms. Some of them are explained below.

☞ Testing of the loop invariant property

- **A loop invariant property** is a property of an iterative algorithm. These algorithms use a loop control structure to define iterations in it.
- The loop invariant relation is true before, after and during all iterations of an iterative algorithm.
- It defines the goal or the desired output of an iterative algorithm.
- After each iteration of a loop, it gives an idea about the current progress towards the final output.
- Thus a loop invariant property is highly essential in understanding the outcome of a loop.
- By testing a loop invariant property of an iterative algorithm we can confirm its correct working.
- To prove the correctness of an iterative algorithm we need to demonstrate that a postcondition is satisfied upon the termination of a given loop. For the same we need to show the following:

(1) **Initialization :** The loop invariant holds before the first iteration of a loop.

(2) **Maintenance :** If the loop invariant holds at the beginning of any loop iteration, then it must also hold at the termination of that iteration.

(3) **Termination :** The loop always terminates after a finite number of iterations.

(4) **Correctness :** Whenever the loop invariant and the termination condition of a loop both hold, then the postcondition must hold.

- E.g. Consider an iterative algorithm to calculate factorial of any number n.

```
int Iterative_fact (int n)
{
    fact := 1;
   for (i := 1; i ≤ n ; i++)
        fact := fact *i;
   return fact;
}
```

- Here we define a postcondition as: fact = n!= product of 1 to n numbers.
- A loop invariant is: fact := fact * i.
- Now we test whether the loop invariant holds for initialization, maintenance and termination. Consider an input = n = 4.
- **Initialization:** If i = 1, fact = 1 = 1!
- **Maintenance:** For any iteration with i = 1 to 4 we have: fact = 1*1 = 1! for i = 1; fact = 2*1= 2! for i =2; fact = 2*3 = 3! for i = 3; fact = 6*4 = 1*1*2*3*4 = 4! for i = 4. Thus loop invariant holds at the beginning and even at the termination of all iterations.
- **Termination:** The loop terminates when i= n = 4.
- **Correctness:** Whenever the termination condition i = n = 4 reaches fact = 24 = 1*1*2*3*4 = 4! Thus the postcondition (fact = n! = product of 1 to n numbers) holds when the loop invariant and the termination condition of a loop hold.
- It implies that the algorithm is correctly computing n! in the finite number of steps.

☞ Proof by mathematical induction (PMI)

- It is a common method to confirm the correctness of an algorithm.
- It is generally used to prove the correctness of recursive algorithms.
- It is a very powerful technique of verifying an infinite number of facts in a finite amount of space. It uses recursion to demonstrate the same.
- It proves the truth of the statement A(n) $\forall n \geq n_0$, by showing individually the truth of each of the statements $A(n_0)$, $A(n_0 + 1)$, $A(n_0 + 2)$,... . Using the principle of mathematical induction we need to establish the truth of only two statements, namely (i) $A(n_0)$ and (ii) A(n + 1) if A(n) holds for an arbitrary integer $n > n_0$.

The general procedure of PMI

(1) **Basis step :** Prove directly that statement $A(n_0)$ is true for some base case n_0 where n_0 is some integer. It is generally a very easy step.

(2) **Induction Hypothesis :** Assume that A(n) holds for an arbitrary integer $n > n_0$ and prove its implication that A(n + 1) holds for $\forall n \geq n_0$.

(3) **Inductive Step :** Then by using the principle of induction, the statement A(n) is true $\forall n \geq n_0$.

- E.g., Consider an iterative algorithm to calculate factorial of any number n.

```
int Iterative_fact (int n)
{
  fact := 1;
  for (i := 1; i ≤ n ; i++)
      fact := fact *i;
  return fact;
}
```

- **Proof of correctness of algorithm Iterative_fact(n) by mathematical induction :**
  - Let a proposition A(n) be true if for each positive number n , fact = n!= 1*2*....*n.
  - **Basis Step :** Let n = 1. When n = 1, the loop enters in the first iteration, fact := 1 * 1 = 1 and i:=1+1 = 2. Since 1! = 1, fact = n! and i = n+ 1 hold. Thus A(n_0) is true for some base case n_0 = 1.
  - **Induction Hypothesis :** Assume that A(n) is true for some n = k. So, fact = k! and i = k + 1 hold after k number of iterations of the loop. Now we prove that A(n+1) is true.
  - We need to show that for k+1 iteration, fact = (k+1)! and i = k+1+1 hold. When the loop enters $(k+1)^{st}$ iteration, fact = k! and i = (k+1) at the beginning of the loop. Inside the loop, fact := k!* (k + 1) and i := (k + 1) + 1 producing fact = (k + 1)! and i = (k + 1) + 1.
  - **Inductive Step:** Hence by inductive reasoning, it can be claimed that A(n) holds for any n i.e. fact = n! and i = n + 1 hold for any positive integer n.
  - Now, when the algorithm terminates, i = n + 1. Hence the loop has been executed at the most n times.
  - Thus fact = n! is returned. This implies that the algorithm Iterative_fact(n) is correct.

1.8 ITERATIVE ALGORITHM DESIGN ISSUES

UQ. Explain issues related to iterative algorithm design.

SPPU - Q. 1(a), May 19, 6 Marks

- Iterative algorithms are non-recursive by nature.
- In these types of the algorithm, the functions do not call themselves.
- The various design issues of the iterative algorithms are as below :
  (1) Iterations using the loop control structure
  (2) Improving the efficiency of the algorithms
  (3) Estimation of time and space requirements
  (4) Expressing the complexities using order notations
  (5) Applying different algorithmic strategies

1.8.1 Iterations using the Loop Control Structure

- The iterative algorithm uses a looping control structure to implement the iterations. E.g. *for, while, repeat-until.*
- For any loop we specify:
  (1) The **initial condition** that is set to be TRUE before the beginning of the loop.
  (2) The **invariant relation** that must hold before, after and during each iteration of the loop.
  (3) The **terminating condition** that specifies the condition for which the loop must terminate.

E.g.,

```
i := 1;      // Initial condition
sum := 0;
while (i ≤ n)      // Terminating condition
{
    sum := sum + i;      // Invariant relation
    i:= i + 1;
}
```

1.8.2 Improving the Efficiency of the Algorithms

UQ. Explain different means of improving the efficiency of an algorithm.

SPPU - Q. 1(a), March 19, 5 Marks, Q. 2(b), Dec. 19, 6 Marks

Considering the limited computing resources we should incorporate different techniques to design an efficient algorithm that consumes lesser memory and time.

☞ Techniques to improve the efficiency of an algorithm

1. Eliminating redundant computations in a loop

- Many times, redundant computations of a constant part in a loop cause the inefficiency.
- Such constant part is unnecessarily recalculated each time a loop iterates causing the wastage of computing resources.
- E.g., Consider the following snippet of a code.

```
sum := 0;
  for (i := 1; i ≤ n; i++)
  {
      sum := sum + i * (a * b + b * b + a * a)
  }
```

- Here expression (a * b + b * b + a * a) has a constant value. Since it is present inside a loop, it is unnecessarily recalculated n times.
- If we eliminate it from a loop and compute it only once outside a loop, it improves the efficiency. So the same code can be rewritten as below :

```
z:= (a * b + b * b + a * a);
sum := 0;
  for (i := 1; i ≤ n; i++)
  {
      sum := sum + i * z;
  }
```

- As the statements in the innermost loop have a maximum frequency of execution they contribute more to the running time of an algorithm. So we must focus on such statements and eliminate the redundancies from the innermost loops.

2. Reducing the references to the array elements in a loop

- To reach any array element the program performs address arithmetic. It consumes time.
- If references to an array element are present in a loop, it costs more in terms of time.
- E.g., Consider the following code of finding the smallest element in an array A [0 : n – 1].

```
j:=0;
  for (i:= 1; i < n; i++)
  {
      if (A[i] < A[j])
      {
          j := i;          /* j is an index of the smallest
                              element in A */
      }
  }
  smallest := A[j];
```

- Here the dominant operation in a loop is a comparison between A[i] and A[j] to find the smallest one. Since both the values are obtained by accessing the array elements, each comparison is more time-consuming.
- If we reduce a reference to an array element in a loop, it will result in lesser time for the comparison. The same code can be rewritten as below :

```
j:= 0;          /* j is an index of the smallest element in A */
smallest := A[j];
  for (i :=1; i < n;  i++)
  {
      if (A[i] < smallest)      /* here, one reference to an
                                   array element is reduced */
      {
          smallest := A[i];
          j :=i;
      }
  }
```

- Thus, by reducing the references to the array elements in the loop, the efficiency of the algorithm can be improved.

3. Avoiding the late termination of a loop

- Sometimes more conditions are tested than required to terminate a loop. This causes the inefficiency of an algorithm.
- E.g., Suppose we want to search the smallest and the largest element in an array A[0:n – 1].

```
smallest := largest := A[0];
for (i := 1; i < n; i++)
{
    if (A[i] < smallest)
        smallest := A[i];
}
```

- Here, two comparisons – (A[i] < smallest and A[i] > largest) are independently tested by two *if* statements inside a *for* loop, so the number of comparisons are 2(n – 1).
- If we replace two *if* statements by *if-else if* statements inside a *for* loop, then the number of comparisons will be (n – 1). This replacement makes sense because an element A[i] is found to be smaller than the current smallest element then there is no need to compare it again with the current largest element. This modification is reflected in the code below :

```
smallest := largest := A[0];
  for (i:= 1; i < n; i++)
  {
     if (A[i] <smallest)
          smallest := A[i];
     else if (A[i] > largest)
          largest := A[i];
  }
```

- Thus, by avoiding the late termination of a loop the performance of an algorithm can be improved.

4. Early detection of the expected output conditions

- For the certain input instances of a problem, the algorithm can reach the expected output condition before its regular condition of termination is reached.
- Identifying such conditions, the algorithm can have an early exit from a loop. It saves the computation time.
- E.g., Suppose we want to search for at least a single even number in an array A[0:n – 1]. Consider an inefficient code for the same as given below :

```
for (i :=0;  i < n ; i++)
{
   if ((A[i] %2) = 0)
       write (A[i]);
   else write ("no even element is present in A");
}
```

- Here, though an even element is found prior to reach the last element in given array A, the loop continues till the end of an array A.
- As we want to search for at least one even element in an array A, if we find any even element before reaching the last element in A, there is no need to scan the further elements in the array.
- On the first occurrence of an even element, (if any) in A, the algorithm can have early termination. The efficient code for the same can be given as below :

```
for (i :=0;  i < n ; i++)
{
   if ((A[i] %2) = 0)
   {
      write (A[i]);
      break;          /* on the occurrence of an
                         even element the algorithm
                         exits early */
   }
}
   if (i =n)
write ("no even element is present in A");
```

- To check for the early exit conditions some additional steps are needed to be added in an algorithm. These steps should be added only if they are inexpensive in terms of computing time.
- There is always a trade-off additional steps and memory space versus execution time to have early exit of an algorithm.

1.9 CLASSIFICATION OF TIME COMPLEXITIES

✓ **Important Note : Kindly refer to subsection 2.2.5 in Chapter 2.**

Summary

- Algorithmic thinking is an essential analytical skill for solving any problem.
- **An algorithm** is a finite set of unambiguous steps needed to be followed in certain order to accomplish a specific task.
- A good algorithm possesses five characteristics: **input, output, definiteness, finiteness** and **effectiveness.**
- Some of the popular algorithmic strategies for problem solving are as below:
  - Brute force method
  - Exhaustive search
  - Divide and conquer
  - Greedy method
  - Dynamic programming

- Backtracking
- Branch and bound
- Exotic algorithms like genetic algorithms, simulated annealing, online algorithms, parallel algorithms, distributed algorithms, optimization algorithms, fuzzy algorithms etc.

- Basic steps to design an algorithm :
  (1) Understand and analyse the problem to be solved.
  (2) Mention the distinctive name to an algorithm.
  (3) Select the appropriate algorithmic strategy.
  (4) Identify the legitimate inputs of an algorithm.
  (5) Identify the expected output of an algorithm.
  (6) Decide the suitable data structure to define inputs and to present the output.
  (7) Describe a finite set of well-ordered unambiguous instructions to produce the expected output.
- **Pseudocode** is the most suitable way of conveying the algorithmic steps to the programmer.
- By testing the **correctness of an algorithm,** we confirm that the algorithm produces an expected output for all legitimate inputs in a finite amount of time.
- **A loop invariant property** of an iterative algorithm defines the goal or the desired output of the algorithm.
- Testing of a loop invariant property: It shows the following:
  (1) Initialization : The loop invariant holds before the first iteration of a loop.
  (2) Maintenance : If the loop invariant holds at the beginning of any loop iteration, then it must also hold at the termination of that iteration.
  (3) Termination : The loop always terminates after a finite number of iterations.
  (4) Correctness : Whenever the loop invariant and the termination condition of a loop both hold, then the postcondition must hold.
- **Proof of the correctness by Mathematical induction :**
  (1) **Basis step :** Prove directly that statement $A(n_0)$ is true for some base case n_0 where n_0 is some integer. It is generally a very easy step.
  (2) **Induction Hypothesis :** Assume that A(n) holds for an arbitrary integer $n > n_0$ and prove its implication that A(n + 1) holds for $\forall n \geq n_0$.
  (3) **Inductive Step :** Then by using the principle of induction, the statement A(n) is true $\forall n \geq n_0$.
- The various **design issues of iterative algorithms** are as below:
  (1) Iterations using the loop control structure
  (2) Improving the efficiency of the algorithms
  (3) Estimation of time and space requirements
  (4) Expressing the complexities using order notations
  (5) Applying different algorithmic strategies
- Techniques to improve the efficiency of the algorithms:
  (1) Eliminating the redundant computations in a loop
  (2) Reducing the references to the array elements in a loop
  (3) Avoiding the late termination of a loop
  (4) Early detection of the expected output conditions

Chapter Ends...

❑❑❑