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.).

1.1 Algorithmic Thinking 1-3

1.2 What is sn Algorithm ? 1-3

1.2.1 Characteristics of an Algorithm 1-3

UQ. Explain the characteristics of a good algorithm? List out the problems solved by the algorithm.
SPPU - Q. 1(b), March 19, 5 Marks 1-3

1.2.2 Algorithms as Technology 1-4

UQ. Write a short note on the algorithm as a technology with example.
SPPU - Q. 2(b), March 19, 5 Marks 1-4

1.2.3 Evolution of Algorithms 1-5

1.3 Classification of Problems 1-5

1.4 Stages in Problem-Solving 1-6

1.5 Applying Different Algorithmic Strategies 1-7

1.6 Design of Algorithm 1-8

1.6.1 Basic Steps to Design an Algorithm 1-8

1.6.2 Writing Pseudocode of an Algorithm 1-8

1.6.3 Pseudocode Conventions 1-8

1.7 The Correctness of an Algorithm 1-9

1.7.1 Need for the Correctness of an Algorithm 1-9

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** 1-9

1.7.2 Confirming Correctness of Algorithm - Sample Examples 1-9

UQ. How to confirm the correctness of an Algorithm? Explain with example. **SPPU - Q. 2(a), March 19, 5 Marks** 1-9

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** 1-9

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** 1-9

1.8 Iterative Algorithm Design Issues 1-11

UQ. Explain issues related to iterative algorithm design. **SPPU - Q. 1(a), May 19, 6 Marks** 1-11

1.8.1 Iterations using the Loop Control Structure 1-11

1.8.2 Improving the Efficiency of the Algorithms 1-11

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** 1-11

1.9 Classification of Time Complexities 1-13

▶ **Chapter Ends** 1-14

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

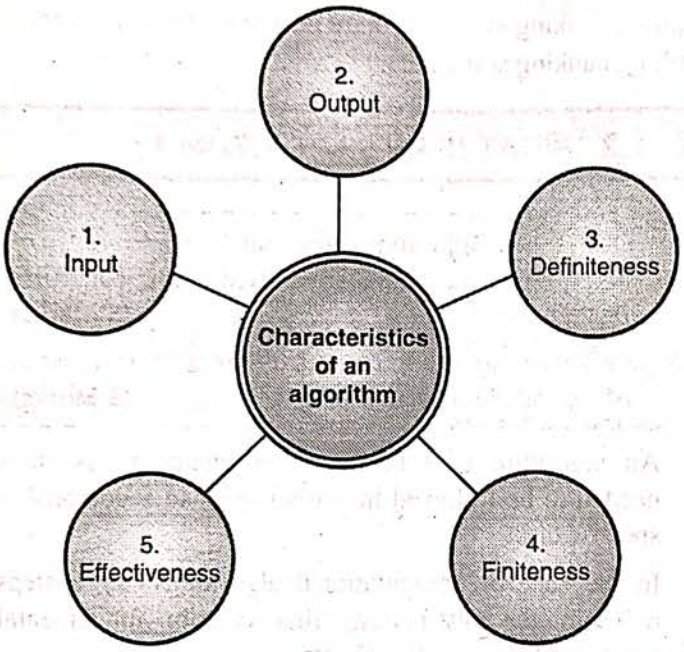
(1) **Input : An algorithm has zero or more inputs.** Each instruction that contains a fundamental operator must accept zero or more inputs.

(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

UNIT II

CHAPTER 2 Analysis of Algorithms and Complexity Theory

Syllabus

Analysis : Input size, best case, worst-case, average-case Counting Dominant operators, Growth rate, upper bounds, asymptotic growth, O, Ω, , Θ, o and ω notations, polynomial and non-polynomial problems, deterministic and non-deterministic algorithms, P-Class problems, NP-Class of problems, Polynomial problem reduction NP-Complete problems - vertex cover and 3-SAT and NP-hard problem - Hamiltonian cycle.

2.1 Analysis of Algorithms 2-3

RQ. Give the significance of analysis of algorithms. Also, compare a priori analysis and a posteriori analysis of algorithms. **(Ref. Q. 1(a), Nov. 18-12 course, 6 Marks)** 2-3

RQ. How do we analyze and measure the time complexity of the algorithm? What are the basic components, which contribute to space complexity? In what way the asymmetry between Big-Oh and Big-Omega notation is helpful? **(Ref. Q. 1(a), Dec. 18, 5 Marks)** 2-3

2.1.1 A Priori and a Posteriori Analysis 2-4

RQ. Compare a priori and posterior analysis of algorithms. **Ref. Q. 2(a) - Dec. 19, 4 Marks** 2-4

2.2 Efficiency-Analysis Framework 2-5

RQ. What is the framework for the analysis of algorithms? Discuss all the components. **(Ref. Q. 1(a), May 13, 8 Marks)** 2-5

2.2.1 The Growth Rate Function 2-5

2.2.2 Estimate of Time Complexity 2-6

2.2.3 Best-case, Worst-case and Average-case Analysis 2-8

RQ. Define best-case, worst-case and average-case efficiency? Is an average-case efficiency is an average of best-case, worst-case efficiencies? **Ref. Q. 2(a), May 14, 6 Marks,Q. 1(a), Feb. 15, 5 Marks** 2-8

2.2.4 Asymptotic Efficiency 2-8

2.3 Asymptotic Notations 2-9

UQ. Explain Big Oh(O), Omega(Ω) and Theta (θ) notations in detail along with suitable examples. **SPPU - Q. 1(a), May 17, 6 Marks** 2-9

UQ. Define asymptotic notations. Explain their significance in analyzing algorithms. **SPPU - Q. 2(b), Aug. 17, 4 Marks** 2-9

UQ. Explain Asymptotic notations with example. **SPPU - Q. 3(a), May 18, 8 Marks** 2-9

UQ. Define asymptotic notation. What is their significance in analyzing algorithms? Explain Big oh, Omega and Theta notations. **SPPU - Q. 4(a), May 19, 8 Marks.** 2-9

UQ. Write short note on : (i) P class and NP class (ii) Big 'oh' and theta **SPPU - Q. 4(a), Dec 19, 8 Marks** 2-9

2.3.1 Types of Asymptotic Notations 2-9

2.3.2 Properties of Asymptotic Notations 2-12

UQ. List the properties of various asymptotic notations. **(SPPU - Q. 1(a), Dec. 19, 5 Marks)** 2-12

2.4 Computational Complexity 2-14

2.4.1 Basic Terminologies of Computational Complexity 2-14

UQ. Explain Polynomial and non-polynomial problems. Explain its Computational complexity. **SPPU - Q. 4(b), May 18, 8 Marks** 2-14

UQ. What are deterministic and non-deterministic algorithms? Explain with example. **SPPU - Q. 3(b), Dec. 17, Q. 3(a), May 19, Q. 3(b), Dec 19, 8 Marks.** 2-14

UQ. Write one example each of deterministic and non-deterministic algorithm for searching. **SPPU - Q. 4(a), May 16, 8 Marks.** 2-15

2.5 Computational Complexity Classes 2-16

UQ. Give and explain the relationship between P, NP, NP-Complete and NP-Hard. **SPPU - Q. 3(a), May 17, 8 Marks.** 2-16

UQ. Explain following with relations with each other.
(i) Polynomial Algorithms (ii) Non-Polynomial Hard Algorithms
(iii) Non-polynomial complete Algorithms **SPPU - Q. 3(b), Dec. 16, 8 Marks.** 2-16

UQ. What are P and NP classes? What is their relationship? Give examples of each class. **SPPU - Q. 4(b), May 16, 8 Marks.** 2-16

2.5.1 The classes: P, NP, NP-Hard, NP-Complete 2-16

2.6 Theory of Reducibility 2-17

2.6.1 Polynomial-time Reduction 2-18

2.6.2 Reducibility and NP-Completeness 2-18

2.7 NP-Complete Problems 2-18

UQ. Write a short note on NP-Completeness of algorithm and NP-Hard. **SPPU - Q. 3(b), May 18, 8 Marks** 2-18

UQ. What are the steps to prove the NP-Completeness of a problem? Prove that the vertex cover problem is NP-Complete. **SPPU - Q. 4(b), May 19, 8 Marks** 2-18

UQ. What is NP-Complete Algorithm? How do we prove that an algorithm is NP-Complete? (Give an example) **SPPU - Q. 4(b), Dec. 16, 6 Marks** 2-18

2.7.1 The 3-SAT Problem 2-19

UQ. What is SAT and 3-SAT problem? Prove that 3-SAT problem is NP-Complete. **SPPU - Q. 4(a), May 18, 8 Marks** 2-20

UQ. What is Boolean Satisfiability Problem? Explain 3-SAT problem. Prove 3-SAT is NP-Complete. **SPPU - Q. 3(b), May 19, 8 Marks** 2-20

UQ. Explain in brief NP-Complete problem. Prove that the 3-SAT problem is NP-Complete. **SPPU - Q. 4(b), Dec. 17, 8 Marks** 2-20

UQ. What is SAT and 3-SAT problem? Prove that the 3-SAT problem is NP-Complete. **SPPU - Q. 3(b), May 16, 8 Marks** 2-20

2.7.2 The Clique Problem 2-22

2.7.3 The Vertex Cover Problem 2-24

UQ. State Vertex cover problem and prove that Vertex Cover problem is NP-Complete. **SPPU - Q. 3(b), Dec. 19, 8 Marks, Q. 3(a), Dec. 17, 8 Marks.** 2-24

UQ. Prove that Vertex cover problem is NP-Complete. **SPPU - Q. 4(b), May 17, 8 Marks.** 2-24

2.7.4 The Hamiltonian Cycle Problem 2-26

UQ. Explain NP-Hard Hamiltonian cycle problem. **(SPPU - Q. 4(b), Dec. 19, 8 Marks)** 2-28

2.7.5 Redúcibility Structure of NP-Complete Problems 2-28

▶ **Chapter Ends** 2-29

2.1 ANALYSIS OF ALGORITHMS

- For many routine activities we, human beings, intuitively follow algorithmic thinking or without applying any conscious thought we do many activities mechanically or habitually.
- Most of the time while doing such activities we are less concerned with the efficiency but for computational algorithms, we must consciously work for its efficiency considering the scarcity of resources and their costs.
- We should think about the wise usage of the computing resources while performing any computational task. We should design efficient algorithms that require lesser resources and execution time.

RQ. Give the significance of analysis of algorithms. Also, compare a priori analysis and a posteriori analysis of algorithms.

(Ref. Q. 1(a), Nov. 18-12 course, 6 Marks)

RQ. How do we analyze and measure the time complexity of the algorithm? What are the basic components, which contribute to space complexity? In what way the asymmetry between Big-Oh and Big-Omega notation is helpful?

(Ref. Q. 1(a), Dec. 18, 5 Marks)

GQ. Define an algorithm. List various criteria used for analyzing an algorithm. **(5 Marks)**

GQ. Define Algorithm, Time Complexity and Space Complexity. **(5 Marks)**

GQ. Define space complexity. **(2 Marks)**

- Analysis of algorithm investigates the efficiency of an algorithm concerning computing resources. It involves the analysis of the time and space complexity of an algorithm.

▶ **Time Complexity :** It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

▶ **Space Complexity :** It is the amount of memory needed for the completion of an algorithm. To estimate the memory requirement we need to focus on two parts:

(1) **A fixed part :** It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

(2) **A variable part :** It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

Consider the following algorithm

Step No.	Step Description	Step Cost	Frequency	Total Count
1.	Algorithm Product(int a[], int n)	-	-	-
2.	{	-	-	-
3.	prod :=1;	1 (for assignment operation)	1	1
4.	for (i :=1; i ≤ n; i++) {	1; (for i : = 1) 1; (for i ≤ n) 1; (for i++)	1; (n+1); n	1 +(n+1) +n
5.	prod := prod*a[i];	1 (for assignment and multiplication operation)	n	n
6.	}	-	-	-
7.	return prod;	1	1	1
8.	}	-	-	
			Total time complexity	**3n + 4**
	Space requirement : 1.	Fixed part: memory to store 3 variables- prod, n, and i.		3
	Space requirement : 2.	Variable part: memory to store an array t of size n		n
			Total space complexity	**n + 3**

- As computational time is more crucial than memory (currently memory is cheaper, but time is precious!) we focus on time complexity in all further discussions. The same theory applies to space complexity, too.

GQ. Determine the frequency couhts for all statements in the following algorithm segment.

```
I = 1;
while(I ≤ n)
{
 X = X + I;
 I = I + 1;
}
```

(3 Marks)

☑ **Ans. :** The frequency counts for all statements in the given algorithm segment are calculated as below :

Step No.	Step Description	Step Cost	Frequency	Total Count
1	I = 1;	1	1	1
2	while(I ≤ n)	1	n + 1	n + 1
3	{	--	--	--
	X = X + I;	1	n	n
4	I = I + 1;	1	n	n
5	}	--	--	--
			Total time complexity	**3n + 2 = O(n)**

2.1.1 A Priori and a Posteriori Analysis

RQ. Compare a priori and posterior analysis of algorithms. **Ref. Q. 2(a) - Dec. 19, 4 Marks**

- The complexity of an algorithm is typically analyzed in two different ways :
  (1) A priori analysis (2) A posteriori analysis

Comparison of a priori and a posteriori analysis

Sr. No.	A priori Analysis	A posteriori analysis
1.	It is made before the execution of an algorithm.	It is made after the execution of an algorithm.
2.	It is based on knowledge that is independent of experimental evidence.	It needs justification through experience and hence it is based on the knowledge that depends on experimental evidence.
3.	All the values of different evaluation metrics are estimated values.	All the values of different evaluation metrics are exact values recorded during experimentation.
4.	All the values of various performance metrics are uniform values independent of actual input size.	All the values of various performance metrics are non-uniform values w.r.t. the actual input given during execution.
5.	It is independent of the computational power of the CPU, operating system, system architecture, programming language, and other environmental aspects.	It is dependent on the computational power of the CPU, operating system, system architecture, programming language, and other environmental aspects.
6.	It is also known as "performance estimation" or "non-empirical analysis".	It is also known as "performance testing" or "performance measurement" or "empirical analysis".
7.	It needs a strong knowledge base of mathematical principles for the analysis of algorithms.	It does not require a strong knowledge base of mathematics.
8.	E.g. x ← x + y needs estimated running time = O(1) irrespective of machine specification.	E.g. x ← x + y runs in say, 0.01 ns on the machine-I, in 0.02 ns on the machine-II, in 0.0005 ns on the machine-III.

2.2 EFFICIENCY-ANALYSIS FRAMEWORK

RQ. What is the framework for the analysis of algorithms? Discuss all the components.
(Ref. Q. 1(a), May 13, 8 Marks)

- There are four major components of the framework for efficiency analysis of an algorithm as given below:

(1) **The growth rate function :** The time and space complexities are defined as functions of the number of input instances of an algorithm.

(2) **An estimate of time complexity :** The time complexity is estimated by calculating the frequency count of fundamental instructions in an algorithm. Each fundamental instruction needs one unit of time.

(3) **Best-case, worst-case and average-case analysis :** Though the input size remains the same, some algorithms perform differently with specific inputs. This leads to the best-case, worst-case and average-case analysis of an algorithm.

(4) **Asymptotic Efficiency :** The efficiency of an algorithm is investigated when the input size increases without any bound.

2.2.1 The Growth Rate Function

- The running time of an algorithm is the function of an input size given to an algorithm. The growth of this function with the increasing input size characterizes the efficiency of an algorithm.
- By comparing the growth rates of the running time of algorithms, we can evaluate the relative performance of alternative solutions to the given problem.

Table 2.2.1 : Growth Rates of some Typical Functions of n

Note : Below the dark line depicted in Table 2.2.1, the algorithm will be impossibly slow]

Input	log n	n	n log n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7				
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}					

Ex. 2.2.1 : Suppose you have algorithms with running time listed below (Assume these are exact running time). How much slower do each of these algorithms get when you a) Double the input size b) increase the input size by 1?

(i) $100\,n^2$ (ii) $n \log n$ (iii) 2^n (iv) n^2 **(8 Marks)**

Soln. :

(A) Double the input size

(i) **$100\,n^2$:** If n = 2n then running time = $100\,(2n)^2 = 400\,n^2$. It implies that the algorithm **gets slower by a factor of 4.**

(ii) **n log n :** If n = 2n then running time = 2n log (2n). It implies that the algorithm **gets slower by a factor of 2, plus an additive 2n.**

(iii) **2^n :** If n = 2n then running time = $2^{(2n)}$. It implies that the algorithm **gets slower by a factor equal to the square of the previous running time i.e. 2^n.**

(iv) **n^2 :** If n = (2n) then running time = $(2n)^2 = 4n^2$. It implies that the algorithm **gets slower by a factor of 4.**

(B) Increase the input size by 1

(i) **$100\,n^2$:** If n = n + 1 then running time = $100\,(n + 1)^2 = 100\,(n^2 + 2n + 1) = 100\,n^2 + 200\,n + 100$. It implies that the algorithm gets **slower by an additive 200 n + 100.**

(ii) **n log n :** If n = n + 1 then running time = (n + 1) log (n + 1). It implies that algorithm gets **slower by an additive log (n + 1) + n [log (n + 1) – log n].**

(iii) **2^n :** If n = n + 1 then running time = $2^{(n+1)} = 2 \times 2^n$. It implies that the algorithm **gets slower by a factor of 2.**

(iv) **n^2 :** If n = n + 1 then running time = $(n + 1)^2 = n^2 + 2n + 1$. It implies that the algorithm gets **slower by an additive 2n + 1.**

Ex. 2.2.2 : Suppose you have algorithms with running time listed below (Assume these are exact running time). How much slower do each of these algorithms get when you a) Double the input size b) increase the input size by 1?

(i) n^2 (ii) n^3 (iii) 2^n (iv) $n \log n$ **(8 Marks)**

Soln. :

(A) Double the input size

(i) **n^2 :** If n = 2n then the running time = $(2n^2) = 4n^2$. It implies that the algorithm gets **slower by a factor of 4.**

(ii) **n^3 :** If n = 2n then the running time = $(2n)^3 = 8n^3$. It implies that the algorithm gets **slower by a factor of 8.**

(iii) 2^n : If n = 2n then the running time = $2^{(2n)}$. It implies that the algorithm **gets slower by a factor equal to the square of the previous running time i.e. 2^n.**

(iv) **n log n :** If n = 2n, then the running time = 2n log (2n). It implies that the algorithm **gets slower by a factor of 2 plus an additive 2n.**

(B) Increase the input size by 1

(i) n^2 : If n = n + 1, then the running time = $(n + 1)^2$ = $n^2 + 2n + 1$. It implies that the algorithm **gets slower by an additive 2n + 1.**

(ii) n^3 : If n = n + 1 then the running time = $(n + 1)^3$ = $n^3 + 3n^2 + 3n + 1$. It implies that the algorithm gets **slower by an additive $3n^2 + 3n + 1$.**

(iii) 2^n : If n = n + 1 then the running time = $2^{(n+1)} = 2 * 2^n$. It implies that the algorithm **gets slower by a factor of 2.**

(iv) **n log n :** If n = (n + 1) then running time = (n + 1) log (n). It implies that the algorithm **gets slower by an additive log(n + 1) + n [log(n + 1) – log n].**

2.2.2 Estimate of Time Complexity

As computational time is more crucial than memory (currently memory is cheaper but time is precious!) we focus on the time complexity in all further discussions. The same theory applies to the space complexity, too.

Counting the dominant operators

- An algorithm describes a finite set of unambiguous steps to be followed in a certain sequence to complete a particular task.
- In the case of computational algorithms, these steps refer to the instructions that contain the fundamental operators *like* +, –, *, /, % etc.
- To estimate the computational time of an algorithm, we can concentrate on dominant fundamental operations in an algorithm.
- These operations can adequately give a notion of the time requirement of execution of any algorithm.
- E.g., In a recursive algorithm, we can focus on the frequency of recursive calls. Also, in an iterative algorithm, we can focus on the frequency of execution of the loop body.

Examples

Ex. 1 : A simple *for* loop.

Step No.	Step Description	Step Cost	Frequency	Total Count
1	Algorithm Sum1(n)	-	-	-
2	{	-	-	-
3	sum:=0;	1	1	1
4	for (i:=1; i≤ n; i++)	1;(for i:=1) 1;(for i ≤ n) 1;(for i++)	1; (n + 1); n	1 + (n + 1) + n
5	sum:= sum+i;	1	n	n
6	}	-	-	-
7	return sum;	1	1	1
8	}	-	-	
			Total time complexity	**3n+4**
Space requirement: 1.Fixed part: memory to store 3 variables- sum, n and i.				3
			Total space complexity	**3 (Constant)**

Ex. 2 : A nested *for* loop.

Step No.	Step Description	Step Cost	Frequency	Total Count
1	Algorithm Sum2(n)	-	-	-
2	{	-	-	-
3	sum:=0;	1	1	1
4	for (i:=1; i≤ n; i++)	1;(for i:=1) 1;(for i≤ n) 1;(for i++)	1; (n+1); n	1 +(n+1) +n
5	for (j:=1; j≤ n; j++)	1;(for j:=1) 1;(for j≤ n) 1;(for j++)	1*n; (n+1)*n; n*n;	n $+n^2+n$ $+n^2$
6	sum:= sum+j;	1	n*n	n^2
7	}	-	-	-
8	}	-	-	-
9	return sum;	1	1	1
10	}	-	-	-
			Total time complexity	**$3n^2+4n+4$**
Space requirement: 1.Fixed part: memory to store 3 variables- sum, n, i and j.				4
			Total space complexity	**4(Constant)**

- In this example, the outer *for* loop on i is executed for frequency = n.
- For each value of i, the inner *for* loop on j is executed for frequency = n giving total frequency count of n^2.

Ex. 3 : A nested *for* loop (Inner loop frequency is based on the arithmetic progression)

Step No.	Step Description	Step Cost	Frequency	Total Count
1	Algorithm Sum3(n)	-	-	-
2	{	-	-	-
3	sum:=0;	1	1	1
4	for (i:=1; i≤ n; i++)	1;(for i:=1) 1;(for i≤ n) 1;(for i++)	1; (n+1); n	1 +(n+1) +n
5	for (j:=1; j≤ n; **j:=i+2**)	1;(for j:=1) 1;(for j≤ n) 1;(for j+2)	1*n; ((n+3)/2)*n; ((n+1)/2)*n;	n $+(n^2+3n)/2$ $+(n^2+n)/2$
6	sum:= sum+j;	1	((n+1)/2)*n	$+(n^2+n)/2$
7	}	-	-	-
8	}	-	-	-
9	return sum;	1	1	1
10	}	-	-	
			Total time complexity	**$(3n^2+11n+8)/2$**
Space requirement: 1.Fixed part: memory to store 3 variables- sum, n, i and j.				4
			Total space complexity	**4 (Constant)**

- In this example, the outer *for* loop on i is executed for frequency = n.
- However, for each value of i, the inner *for* loop on j is executed for frequency = $\frac{(n+1)}{2}$ as j = i + 2 that follows the arithmetic progression.

2.2.3 Best-case, Worst-case and Average-case Analysis

RQ. Define best-case, worst-case and average-case efficiency? Is an average-case efficiency is an average of best-case, worst-case efficiencies?

Ref. Q. 2(a), May 14, 6 Marks, Q. 1(a), Feb. 15, 5 Marks

Though the input size remains the same, the efficiency of certain algorithms varies with specific inputs. It demonstrates the three cases of efficiency analysis as described below

(i) Best-case efficiency

- It is achieved when the best-case input of size n is given to an algorithm. With such input, an algorithm runs to its completion in the least amount of time than that with other inputs of the same size, n.
- E.g., In a sequential search, if the first element of a given input of size n is the same as the search key, then the algorithms ends with a successful search in just one comparison.

∴ O(1) is the best-case efficiency of a sequential search.

(ii) Worst-case efficiency

- It is achieved when the worst-case input of size n is given to an algorithm. With such input, an algorithm runs for the longest among all inputs of the same size n. It gives an upper bound on running time.
- E.g., In a sequential search if the last element of a given input of size n is equal to the search key, then the algorithm ends with a successful search by performing n comparison O(n) is the worst-case efficiency of a sequential search.

(iii) Average-case efficiency

- It is recorded when neither best-case nor the worst-case but any "random" input of size n is given to an algorithm.
- It is not calculated by averaging the worst-case and the best-case efficiencies. Sometimes coincidently the average case efficiency of an algorithms matches with the mean of its worst-case and the best-case of efficiencies.
- Estimation of average-case efficiency is based on some probabilistic assumptions and is comparatively more complex than the best-case and worst-case analysis.
- E.g., for any random input of size n, sequential search performs O(n) comparisons. Hence its average-case efficiency is O(n).

2.2.4 Asymptotic Efficiency

Unless we evaluate the performance of an algorithm by taking a sufficiently large input size, we cannot justify its growth rate. Hence we need to study the **"asymptotic efficiency"** of an algorithm where the performance of an algorithm is verified against the increasing input size boundlessly.

Table 2.2.2 : Basic asymptotic efficiency classes

Sr. No.	Asymptotic efficiency class	Description
1.	1	Defines **"constant"** growth rate irrespective of input size.
2.	log n	Defines **"logarithmic"** or **"sublinear"** growth rate achieved by reducing an input size by a constant factor on subsequent iterations of the algorithm. E.g. Binary search algorithm.
3.	n	Defines **"linear"** growth rate. E.g. Sequential search algorithm.
4	n log n	Defines **"quasi-linear"** growth rate. E.g. Merge sort by divide and conquer strategy.
5.	n^2	Defines **"quadratic"** growth rate. It is generally found in an analysis of algorithms with nested loops. E.g. Matrix addition.
6.	n^3	Defines **"cubic"** growth rate. It is generally found in an analysis of algorithms with 3 nested loops. E.g. Floyd-Warshall's algorithm.
7.	2^n	Defines **"exponential"** growth rate. It characterizes the algorithms that generate all possible subsets of n-item sets. E.g. Travelling salesperson problem by dynamic programming.
8.	n!	Defines **"factorial"** growth rate. It characterizes the algorithms that produce all permutations of an n-item set. E.g. Travelling salesman problem by brute-force method.

2.3 ASYMPTOTIC NOTATIONS

UQ. Explain Big Oh(O), Omega(Ω) and Theta (θ) notations in detail along with suitable examples. **SPPU - Q. 1(a), May 17, 6 Marks**

UQ. Define asymptotic notations. Explain their significance in analyzing algorithms. **SPPU - Q. 2(b), Aug. 17, 4 Marks**

UQ. Explain Asymptotic notations with example. **SPPU - Q. 3(a), May 18, 8 Marks**

UQ. Define asymptotic notation. What is their significance in analyzing algorithms? Explain Big oh, Omega and Theta notations. **SPPU - Q. 4(a), May 19, 8 Marks**

UQ. Write short note on :

(i) P class and NP class

(ii) Big 'oh' and theta **SPPU - Q. 4(a), Dec 19, 8 Marks**

- To compare two algorithms with reference to their growth rates, we use notations called **"asymptotic notations"**.
- The asymptotic notation describes the **"asymptotic efficiency"** of an algorithm.

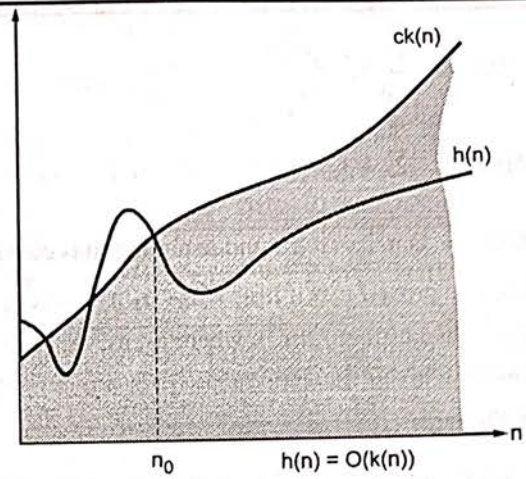
2.3.1 Types of Asymptotic Notations

(1) Big "Oh" (O)
(2) Big Omega (Ω)
(3) Theta (Θ)
(4) Little "Oh" (o)
(5) Little Omega (ω)

(1) Big "Oh" (O)

Let h(n) and k(n) are two functions. The function h(n) = O(k(n)) iff there exist some positive constants c and n_0 such that $h(n) \le c * k(n)$, $\forall\, n \ge n_0$.

Here, k(n) is having the same or higher growth rate than h(n). So it gives the **upper bound** on h(n).



(1A2)**Fig. 2.3.1(a) : Upper Bound(O-Notation)**

Example

(i) The function $5n + 16 \ne O(1)$ as there are no positive constants c and n_0 so that $5n + 16 \le c \cdot 1\ \forall\, n \ge n_0$.

- The function $5n + 16 = O(n)$ as $5n + 16 \le 21\,n$ for all $n \ge 1$. Here $n_0 = 1$ and $c = 21$.
- The function $5n + 16 = O(n^2)$ as $5n + 16 \le 7n^2$ for all $n \ge 2$. Here $n_0 = 2$ and $c = 7$.
- The function $5n + 16 = O(n^3)$ as $5n + 16 \le n^3$ for all $n \ge 4$. Here $n_0 = 4$ and $c = 1$.
- The function $5n + 16 = O(2^n)$ as $5n + 16 \le 2^n$ for all $n \ge 6$. Here $n_0 = 6$ and $c = 1$.
- Even though 5n + 16 can be expressed as O(n), $O(n^2)$, $O(n^3)$, $O(2^n)$, we must select the Least Upper Bound. Since, $O(1) \le O(n) \le O(n^2) \le O(n^3) \le O(2^n)$ for sufficient large value of n, 5n + 16 = O(n) is the most appropriate expression.

Theorem 2.3.1

If $h(n) = a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m$ then $h(n) = O(n^m)$, where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

Proof

- To prove $h(n) = O(n^m)$ we check the inequality,

$$h(n) \le a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m;$$

where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

$$\therefore h(n) \le |a_i| \cdot n^i$$

$$\therefore \quad h(n) \le \sum_{i=0}^{m} |a_i| \cdot n^{i-m}$$

$$\therefore \quad h(n) \le \sum_{i=0}^{m} |a_i|, \forall n \ge 1$$

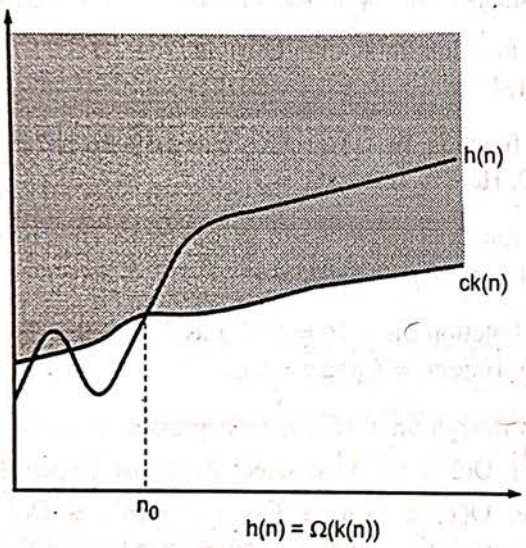
$\therefore$ $h(n) = O(n^m)$, $\forall n \ge 1$ and assuming m is constant.

- Thus, it is proved that if $h(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_m n^m$ then $h(n) = O(n^m)$, where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

(2) Big Omega (Ω)

Let h(n) and k(n) are two functions. The function $h(n) = \Omega(k(n))$ iff there exists some positive constants c and n_0 such that $h(n) \ge c * k(n)$, $\forall n \ge n_0$.

Here k(n) is having either the same or lower growth rate than h(n). So it defines the **lower bound** on h(n).



(1A3)**Fig. 2.3.1(b) : Lower Bound (Ω-Notation)**

☞ Examples

(i) The function $5n + 16 = \Omega(n)$ as $5n + 16 \ge n$ for all $n \ge 1$. Here $n_0 = 1$ and $c = 1$. [It holds for n = 0, but as per definition $n_0 \ge 0$]

(ii) The function $8n^2 + 5n + 16 = \Omega(1)$ as $8n^2 + 5n + 16 \ge 1$ for all $n \ge 1$. Here $n_0 = 1$ and $c = 1$.

- The function $8n^2 + 5n + 16 = \Omega(n)$ as $8n^2 + 5n + 16 \ge n$ for all $n \ge 1$. Here $n_0 = 1$ and $c = 1$.
- The function $8n^2 + 5n + 16 = \Omega(n^2)$ as $8n^2 + 5n + 16 \ge n^2$ for all $n \ge 1$. Here $n_0 = 1$ and $c = 1$.
- Even though $8n^2 + 5n + 16$ can be expressed as $\Omega(1)$, $\Omega(n)$ and $\Omega(n^2)$. $\Omega(1)$ is not much informative as a lower bound so it is never used though it is valid. We must select the Highest Lower Bound. Since $\Omega(1) \le \Omega(n) \le \Omega(n^2)$ function $8n^2 + 5n + 16 = \Omega(n^2)$.

☞ Theorem 2.3.2

If $h(n) = a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m$ then $h(n) = \Omega(n^m)$, where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

☑ Proof

- To prove $h(n) = \Omega(n^m)$ we check the inequality,

$$h(n) \ge a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m;$$

where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

$$\therefore h(n) \ge |a_i| \cdot n^i$$

$$\therefore h(n) \ge \sum_{i=0}^{m} |a_i| \cdot n^{i-m}$$

$$\therefore h(n) \ge \sum_{i=0}^{m} |a_i|, \forall n \ge 1$$

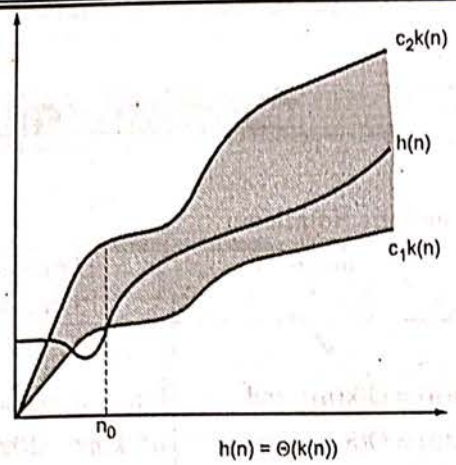
$\therefore$ $h(n) = O(n^m)$, $\forall n \ge 1$ and assuming m is constant.

- Thus, it is proved that if $h(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_m n^m$ then $h(n) = \Omega(n^m)$, where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

(3) Theta (Θ)

Let h(n) and k(n) are two functions. The function $h(n) = \Theta(k(n))$ iff there exists some positive constants c_1, c_2 and n_0 such that $c_1 * k(n) \le h(n) \le c_2 * k(n)$, $\forall n \ge n_0$.

In other words, the function $h(n) = \Theta(k(n))$ iff $h(n) = O(k(n))$ and $h(n) = \Omega(k(n))$, $\forall n \ge n_0$. Here k(n) defines both an upper and lower bound on h(n). It defines a **tight bound** on h(n).



(1A4)**Fig. 2.3.1(c) : Tight Bound (Θ -Notation)**

Examples

(i) The function $5n + 16 \neq \Theta(1)$ as $5n + 16 \neq O(1)$ though $5n + 16 = \Omega(1)$.

(ii) The function $5n + 16 = \Theta(n)$ as $5n + 16 \leq 21n$ for all $n \geq 1$ and $5n + 16 \geq n$ for all $n \geq 1$. Here, $c_1 = 1$, $c_2 = 21$ and $n_0 = 1$.

Theorem 2.3.3

If $h(n) = a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m$ then $h(n) = \Theta(n^m)$, where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

Proof

- By the definition of an asymptotic notation Θ, to prove $h(n) = \Theta(n^m)$ we must prove that $h(n) = O(n^m)$ and $h(n) = \Omega(n^m)$.
- Referring to Theorem 2.3.1 we can show $h(n) = O(n^m)$.
- Referring to Theorem 2.3.2 we can show $h(n) = \Omega(n^m)$.
- As $h(n) = O(n^m)$ and $h(n) = \Omega(n^m)$ we have $h(n) = \Theta(n^m)$. Thus, it is proved that if $h(n) = a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m$ then prove that $h(n) = \Theta(n^m)$, where $a_0, a_1, a_2, \ldots, a_m$ are constants and $a_m > 0$.

Theorem 2.3.4

If Let h(n) and k(n) be asymptotically nonnegative functions then h(n) + k(n) = Θ(maximum (h(n), g(n))).

Proof

- As h(n) and k(n) are asymptotically nonnegative functions, there exists n_0 such that $h(n) \geq 0$ and $k(n) \geq 0$ for all $n \geq n_0$.
- Thus, for $n \geq n_0$, $h(n) + k(n) \geq h(n) \geq 0$ and $h(n) + k(n) \geq k(n) \geq 0$.
- Let f(n) = maximum (h(n), g(n))

$$\therefore\ f(n) = \begin{cases} h(n) \text{ if } h(n) \geq h(n), \\ k(n) \text{ if } h(n) < k(n) \end{cases}$$

- Thus, for any n, f(n) is either h(n) or k(n).
- So, we have $0 \leq h(n) \leq f(n)$ and $0 \leq k(n) \leq f(n)$ for all $n \geq n_0$. Adding these 2 inequalities we get, $0 \leq h(n) + k(n) \leq 2f(n)$, which shows that $(h(n) + k(n)) = O(f(n)) = O(\text{maximum}(h(n), k(n)))$ for all $n \geq n_0$ and $c_2 = 2$. .....(By definition of Big-Oh).
- Similarly, we have $h(n) + k(n) \geq f(n) \geq 0$, which shows that $(h(n) + k(n)) = \Omega(f(n)) = \Omega(\text{maximum}(h(n), k(n)))$ for all $n \geq n_0$ and $c_1 = 1$.

...(By definition of Big-Omega).

- As $(h(n) + k(n)) = O(f(n)) = O(\text{maximum}(h(n), k(n)))$ and $(h(n) + k(n)) = \Omega(f(n)) = \Omega(\text{maximum}(h(n), k(n)))$ we can show that $(h(n) + k(n)) = \Theta(f(n)) = \Theta(\text{maximum}(h(n), k(n)))$ for all $n \geq n_0$, $c_1 = 1$, $c_2 = 2$.

...(By definition of Theta).

(4) Little 'Oh' (o)

Let h(n) and k(n) are two functions. The function h(n) = o(k(n)) iff

$$\lim_{n \to \infty} \frac{h(n)}{k(n)} = 0$$

Here the bound $0 \leq h(n) < c * k(n)$ holds for all $c > 0$ and for all $n \geq n_0$.

Examples

(i) The function $5n + 16 = o(n^2)$ as

$$\lim_{n \to \infty} \frac{5n + 16}{n^2} = 0$$

(ii) The function $5n + 16 \neq o(n)$ as

$$\lim_{n \to \infty} \frac{5n + 16}{n} \neq 0$$

(5) Little Omega (ω)

Let h(n) and k(n) are two functions. The function h(n) = ω(k(n)) iff

$$\lim_{n \to \infty} \frac{k(n)}{h(n)} = 0 \quad \text{OR} \quad \lim_{n \to \infty} \frac{h(n)}{k(n)} = \infty$$

Here the bound $h(n) > c * k(n) \geq 0$ holds for all $c > 0$ and for all $n \geq n_0$.

Example

(i) The function $8n^2 + 5n + 16 = \omega(n)$ as

$$\lim_{n \to \infty} \frac{n}{8n^2 + 5n + 16} = 0$$

2.3.2 Properties of Asymptotic Notations

UQ. List the properties of various asymptotic notations. **(SPPU - Q. 1(a), Dec. 19, 5 Marks)**

Consider functions h(n) and k(n) are asymptotically positive.

Table 2.3.1 : Relational Properties of Asymptotic Notations

Sr. No.	Property / Notation	Reflexivity	Symmetry	Transitivity	Transpose Symmetry
1.	O	✓ E.g. $h(n) = O(h(n))$	× if $h(n) = O(k(n))$ then $k(n) \neq O(h(n))$	✓ $h(n) = O(k(n))$ and $k(n) = O(l(n))$ $\Rightarrow h(n) = O(l(n))$	✓ $h(n) = O(k(n))$ iff $k(n) = \Omega(h(n))$
2.	Ω	✓ E.g. $h(n) = \Omega(h(n))$	× if $(h(n)) = \Omega(k(n))$ then $k(n) \neq \Omega(h(n))$	✓ $h(n) = \Omega(k(n))$ and $k(n) = \Omega(l(n)) \Rightarrow$ $h(n) = \Omega(l(n))$	✓ $h(n) = \Omega\ (k(n))$ iff $k(n) = O((h(n))$
3.	Θ	✓ E.g. $h(n) = \Theta(h(n))$	✓ $h(n) = \Theta(k(n))$ iff $k(n) = \Theta(h(n))$	✓ $h(n) = \Theta(k(n))$ and $k(n) = \Theta(l(n))$ $\Rightarrow h(n) = \Theta(l(n))$	✓ $h(n) = O(k(n))$ iff $k(n) = \Omega(h(n))$ and $h(n) = \Omega(k(n))$ iff $k(n) = O((h(n))$
4.	o	× E.g. $h(n) \neq o(h(n))$	× if $h(n) = o(k(n))$ then $k(n) \neq o(h(n))$	✓ $h(n) = o(k(n))$ $k(n) = o(l(n))$ $\Rightarrow h(n) = o(l(n))$	✓ $h(n) = o(k(n))$ iff $k(n) = \omega(h(n))$
5.	ω	× E.g. $h(n) \neq \omega(h(n))$	× if $h(n) = \omega(k(n))$ then $k(n) \neq \omega(h(n))$	✓ $h(n) = \omega(k(n))$ $k(n) = \omega(l(n))$ $\Rightarrow h(n) = \omega(l(n))$	✓ $h(n) = \omega(k(n))$ iff $k\ (n) = o(h(n))$

Note :

1. Dominance ranking of growth rate functions ;

$$1 \leq \frac{\log n}{\log \log n} \leq \log \log n \leq (\log n)^k \leq n \leq n \log n \leq n^2 \leq 2^n \leq 4^n \leq n!$$

2. Lower growth rate function = O (Same or higher growth rate function).
3. Higher growth rate function = Ω (Same or lower growth rate function).
4. Always select the least upper bound and the highest lower bound to express the asymptotic efficiency.
5. If $h(n) = a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m$, where $a_m \neq 0$ then,

$$h(n) = \begin{cases} O(n^m) \\ \Omega(n^m) \\ \Theta(n^m) \end{cases}$$

6. If $h(n) = O(k(n))$ then $a * h(n) = O(k(n))$ where $a > 0$.
7. If $h(n) = O(f(n))$ and $k(n) = O(g(n))$ then
   (i) $h(n) + k(n) = O(\text{maximum}(f(n), g(n)))$
   (ii) $h(n) * k(n) = O(f(n) * g(n))$
8. $\lim_{n \to \infty} \frac{h(n)}{k(n)} = \begin{cases} 0 \Rightarrow h(n) = o(k(n)), \text{ i.e. } h(n) \text{ has smaller growth rate than } k(n). \\ \infty \Rightarrow h(n) = \omega(k(n)), \text{ i.e. } h(n) \text{ has larger growth rate than } k(n). \\ c > 0 \Rightarrow h(n) \text{ has the same growth rate as that of } k(n). \end{cases}$

Ex. 2.3.1 : Reorder from the smallest to the largest. Justify your answer.

(i) $n \log_2 n$, $n+n^2+n^3$, 2^4, sqrt(n).

(ii) n^2, 2^n, $n \log_2 n$, $\log_2 n$, n^3.

(iii) $n \log_2 n$, n^8, $n^2/\log_2 n$, $(n^2 - n + 1)$

(iv) $n!$, 2^n, $(n+1)!$, 2^{2n}, n^n, $n^{\log n}$ **(10 Marks)**

☑ **Soln. :**

Let us take $n = 1024 = 2^{10}$

Note : Take any sufficiently large value of n.

(i)
$$n \log_2 n = 1024 \log_2 1024 = 2^{10} \times 10$$
$$n + n^2 + n^3 = 2^{10} + 2^{20} + 2^{30}$$
$$2^4 = \text{constant}$$
$$\text{sqrt}(n) = \sqrt{2^{10}} = 2^5$$

As $2^4 < 2^5 < 2^{10} \times 10 < 2^{10} + 2^{20} + 2^{30}$ we get,

$$\mathbf{2^4 \le \text{sqrt}(n) \le n \log_2 n \le n + n^2 + n^3}$$

(ii)
$$n^2 = (2^{10})^2 = 2^{20}$$
$$2^n = (2^{1024})$$
$$n \log_2 n = 2^{10} \times \log_2 2^{10} = 2^{10} \times 10$$
$$\log_2 n = \log_2 2^{10} = 10$$
$$n^3 = (2^{10})^3 = 2^{30}$$

As $2^{10} \times 10 < 2^{20} < 2^{30} < 2^{1024}$ we get,

$$\mathbf{\log_2 n \le n \log_2 n \le n^2 \le n^3 \le 2^n}$$

(iii)
$$n \log_2 n = 2^{10} \log_2 2^{10} = 2^{10} \times 10$$
$$n^8 = (2^{10})^8$$
$$\frac{n^2}{\log n} = \frac{(2^{10})^2}{\log_2 2^{10}} = \frac{2^{20}}{10}$$
$$n^2 - n + 1 = (2^{10})^2 - 2^{10} + 1 = 2^{20} - 2^{10} + 1$$

As $2^{10} \times 10 < \frac{2^{20}}{10} < 2^{20} - 2^{10} + 1 < 2^{80}$ we get,

$$\mathbf{n \log_2 n \le n^2 / \log_2 n \le n^2 - n + 1 \le n^8}$$

(iv)
$$n! = 1024!$$
$$2^n = 2^{1024}$$
$$(n+1)! = 1025!$$
$$2^{2n} = 2^{2 \times 1024} = 2^{2048}$$
$$n^n = 1024^{1024}$$
$$n^{\log n} = 1024^{\log_2 1024} = 1024^{10}$$

As $1024^{10} < 2^{1024} < 2^{2048} < 1024! < 1025! < (1024)^{1024}$

We get,

$$\mathbf{n^{\log n} \le 2^n \le 2^{2n} \le n! \le (n+1)! \le n^n}$$

Ex. 2.3.2 : Reorder from the smallest to the largest. Justify your answer.

(i) $n \log_2 n$, $n + n^2 + n^3$, 2^4, sqrt(n).

(ii) $n!$, 2^n, $(n+1)!$, 2^{2n}, n^n, $n^{\log n}$ **(5 Marks)**

☑ **Soln. :**

Refer solution to sub-questions (i) and (iv) in the solution of Ex. 2.3.1.

Ex. 2.3.3 : Compare the following complexities and Reorder from the smallest to the largest. Justify your answer.

(i) n^2, 2^n, $n \log_2 n$, $\log_2 n$, n^3.

(ii) $n \log_2 n$, n^8, $n^2/\log_2 n$, $(n^2 - n + 1)$ **(5 Marks)**

☑ **Soln. :** Refer solution to sub-questions (ii) and (iii) in the solution of Ex. 2.3.1.

2.4 COMPUTATIONAL COMPLEXITY

- Computational complexity theory refers to the lower bounds on the efficiency of the algorithms in a pessimistic way to describe the difficulty level of those algorithms.
- It categorizes numerous computational problems based on their inherent difficulty and defines the relations between them.
- It identifies different models of efficient computation, their strengths and weaknesses, and their relationships.

2.4.1 Basic Terminologies of Computational Complexity

UQ. Explain Polynomial and non-polynomial problems. Explain its Computational complexity.

SPPU - Q. 4(b), May 18, 8 Marks

UQ. What are deterministic and non-deterministic algorithms? Explain with example.

SPPU - Q. 3(b), Dec. 17, Q. 3(a), May 19, Q. 3(b), Dec 19, 8 Marks

Optimization problem

- It is a computational problem that determines the optimal value of a specified cost function.
- It includes minimization problems like optimal storage on tapes problem, minimum spanning tree (MST) problem, TSP, optimal binary search tree (OBST) problem, vertex cover problem.
- It includes maximization problems like knapsack problem, job sequencing problem, max-clique problem.

Decision problem

- It is a restricted type of a computational problem that produces only two possible outputs ("yes" or "no") or ("TRUE" or "FALSE") for each input..
- A decision problem only verifies whether the input satisfies a specific property.
- E.g., Primality test, Hamiltonian cycle: Whether a given graph has any Hamiltonian cycle in it?
- **A decision algorithm** answers the correct truth value TRUE or FALSE for each input instance of a given decision problem in the finite amount of time.
- The decision problems are easier to solve than the optimization problems.
- If a decision problem is "hard" to solve, then it implies that the corresponding optimization problem is also "hard". Thus, the theory of computational complexity is built around the decision problems that have implications for the optimization problems.
- An optimization problem can be formulated as a decision problem. E.g., Clique decision problem: Is there any clique of size at least s, for some s in a given graph.

Decidable problem

- A decision problem is said to be **decidable** if there exists a decision algorithm to solve it.
- The decidable problems get the correct TRUE or FALSE answer for a given input either in polynomial time or in non-polynomial time. Thus, the decidable problems can be either tractable or intractable.
- E.g., Primality test, TSP, knapsack problem, Hamiltonian cycle problem.

Undecidable problem

- A decision problem is said to be **undecidable** if there is no decision algorithm to solve it.
- Such problems do not get the correct TRUE or FALSE answer for a given input by any algorithm.
- E.g., Halting problem: For a given input instance whether a computer program will halt or continue its execution indefinitely?

Tractability

- It defines the feasibility of an algorithm to complete its execution in a reasonable time.
- The problems are said to be **tractable** if they get solved in polynomial time using the deterministic algorithms. E.g., Time complexities $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$.
- Some examples of the tractable problems: linear search: $O(n)$, binary search: $O(\log n)$, merge sort: $O(n \log n)$, Prim's algorithm: $O(n^2)$.

Intractability

- It defines the infeasibility of an algorithm to complete its execution in a reasonable time.
- The problems are said to be **intractable** if they cannot be solved in polynomial time using the deterministic algorithms.
- There are some intractable but decidable problems with exponential runtime. E.g., Time complexities $O(2^n)$, $O(n^n)$, $O(n!)$

- Some examples of the intractable problems: 0/1 knapsack problem: $O(2^{n/2})$, TSP: $O(n^2 2^n)$, m-colouring problem: $O(nm^n)$.

Deterministic algorithm

- The algorithm is said to be **deterministic** if it generates the same result for the same set of inputs.
- The deterministic algorithms uniquely define the outcomes for specific legitimate input.
- All computer programs are deterministic in nature.
- E.g., Addition of first n numbers, sorting algorithms, searching algorithms.

Non-deterministic algorithm

- The algorithm is said to be **non-deterministic** if it generates different results for the same set of inputs on different executions.
- The non-deterministic algorithms arbitrarily define the outcomes for specific legitimate input.
- Though there is a degree of randomness to the outcomes of a non-deterministic algorithm, they are restricted to a certain set of possibilities.
- A non-deterministic algorithm follows two stages :
  - (i) **Non-deterministic stage of guessing** the outcome for a given input instance.
  - (ii) **Deterministic stage of verifying** the outcome whether it satisfies a particular property.
- In a non-deterministic algorithm, the next step of execution cannot be decided.
- In practice, no such algorithm is implemented on a computer.
- E.g., A non-deterministic searching algorithm, a non-deterministic sorting algorithm.

Complexity classes

- In the theory of computational complexity, the computational problems are classified as per their complex nature and the requirements of computing resources. These classes of problems are known as **complexity classes.**
- The problems with a similar range of time and space requirements to get their solutions are included in a single complexity class.
- Complexity classes help in the organization of similar types of problems.
- The major four complexity classes are :
  - (1) P-Class problems,
  - (2) NP-Class problems,
  - (3) NP-Hard problems and
  - (4) NP-Complete problems.

UQ. Write one example each of deterministic and non-deterministic algorithm for searching.

SPPU - Q. 4(a), May 16, 8 Marks

Ans. :

- Refer to the definitions mentioned in section 2.4.1 Basic Terminologies of Computational Complexity
- E.g., A deterministic searching algorithm and a non-deterministic searching algorithm.

Deterministic searching algorithm

```
Algorithm Search_D(X, y, n)
/*Input: An array X[1 : n], (n ≥ 1 is a set of n elements in
which an element y is to be searched.
Output: An index i in an array X, such that X[i] = y or i = 0
if y is not found in X.*/
{
  for ( i:=1; i ≤ n; i++)
  {
     if (X[i] = y)
     {
        write(i) ;          /*An element y found at
                             position i. Successful
                             search.*/
        break;
     }
  }
  if (i = n+1)
     write (0) ;            /*An element y is not present in
                             X[1:n]. Unsuccessful search.*/
}
```

- Here, if an element y is found at index i, $1 \le i \le n$ in an array X[1: n] then the algorithm has a successful search otherwise it is an unsuccessful search.
- **Complexity analysis of the deterministic searching algorithm :** It takes O(n) time.

Non-deterministic searching algorithm

```
Algorithm Search_ND(X, y, n)
/*Input: An array X[1 : n], (n ≥ 1 is a set of n elements in
which an element y is to be searched.
Output: An index i in an array X, such that X[i] = y or i = 0
if y is not found in X.*/
{
    i : = Choice(1, n) ;          // Guessing stage
    if (X[i] = y)                 // Verification stage
    {
        write(i) ;
        Success ( ) ;             /*Indicates a successful
                                    execution.*/
    }
    else
    {   write (0) ;
        Failure ( ) ;             /*Indicates an unsuccessful
                                    execution.*/
    }
}
```

- Here, a function Choice(1, n) randomly selects any index in the range [1, n].
- If an element y is found at this randomly selected index i in an array X then the algorithm has a successful execution indicating the same by a function Success ( ).
- However, if an element y is not found at this randomly chosen index i in an array X then the algorithm has an unsuccessful execution indicating the same by a function Failure( ).
- **Complexity analysis of the non-deterministic searching algorithm:** Each of the three functions Choice(1, n), Success( ) and Failure( ) takes a constant execution time O(1).
- Hence non-deterministic complexity of this non-deterministic searching algorithm is O(1).
- Every deterministic searching algorithm has time complexity $\Omega(n)$ to search an element in an unordered set of n elements.
- Thus, the searching can be done in O(1) time only if there exists any machine to run a non-deterministic algorithm. Practically no such machine exists.

2.5 COMPUTATIONAL COMPLEXITY CLASSES

UQ. Give and explain the relationship between P, NP, NP-Complete and NP-Hard.

SPPU – Q. 3(a), May 17, 8 Marks

UQ. Explain following with relations with each other.

(i) Polynomial Algorithms

(ii) Non-Polynomial Hard Algorithms

(iii) Non-polynomial complete Algorithms

SPPU – Q. 3(b), Dec. 16, 8 Marks

UQ. What are P and NP classes? What is their relationship? Give examples of each class.

SPPU - Q. 4(b), May 16, 8 Marks

Computational complexity classes describe the categories of computational problems based on their algorithmic complexities. There are four broad categories of problems: (1) P-Class problems, (2) NP-Class problems, (3) NP-Hard problems and (4) NP-Complete problems.

2.5.1 The classes: P, NP, NP-Hard, NP-Complete

GQ. When do you claim that an algorithm is of a polynomial complexity? **(2 Marks)**

GQ. Explain in brief NP-Complete problem. **(4 Marks)**

1. P-Class

- It is the class of decision problems that can be solved in polynomial time by deterministic algorithms.

A polynomial-time algorithm

- It is an algorithm whose running time is polynomially dependent on the input size of a problem instance.
- Thus, a **polynomial-time algorithm** for a problem instance of size n has its worst-case complexity O(p(n)) where p(n) is polynomial of input size n. E.g., Time complexities $O(n^2)$, $O(n^3)$, O(1), O(n log n).
- **"P"** stands for the Polynomial-time algorithm.
- P-Class problems are decidable and tractable problems.
- Some examples of P-Class problems: linear search: O(n), binary search: O(log n), merge sort: O(n log n), Prim's algorithm: $O(n^2)$, Floyd-Warshall's algorithm: $O(n^3)$.

2. NP-Class

- It is the class of decision problems that can be solved in polynomial time by non-deterministic algorithms.
- "NP" stands for the Non-deterministic Polynomial-time algorithm and not for non-polynomial time algorithms.

▶ **Non-deterministic polynomial-time algorithms** produce possible solutions to given problems in a non-deterministic way and verify the correctness of those solutions in polynomial time.

- If somebody tells us a solution to the problem, then we can verify it in polynomial time. Such problems are NP-Class problems. E.g., Sudoku puzzle.
- NP-Class problems are decidable.
- NP-Class problems are tractable or intractable.
- As the definition of NP-Class problems applies to all P-Class problems, $P \subseteq NP$; but it is an open question in computational complexity: does P = NP?
- All P-Class problems and NP-Complete problems are NP-Class problems, but its inverse is not true by assuming $P \neq NP$.
- Some classic examples of NP-Class problems: 0/1 knapsack problem, TSP, graph colouring problem, vertex-cover problem, clique problem.

3. NP-Hard Class

▶ A decision problem P_1 is **NP-Hard** if each NP-Class problem is polynomially reducible to P_1 (i.e. for each problem $P_2 \in NP$, $P_2 \alpha P_1$).

- Thus, it implies that an NP-Hard problem is at least as hard as the hardest NP-Class problem.
- All NP-Complete problems are NP-Hard problems, but all NP-Hard problems are not NP-Complete problems.
- Though the name is **NP**-Hard, all problems in this class do not belong to NP-Class.
- NP-Hard problems are decidable or undecidable.

  E.g.,

  (a) Decidable NP-Hard problems: vertex-cover decision problem, clique decision problem, m-coloring decision problem, TSP decision problem.

  (b) Undecidable NP-Hard problems: Halting problem.

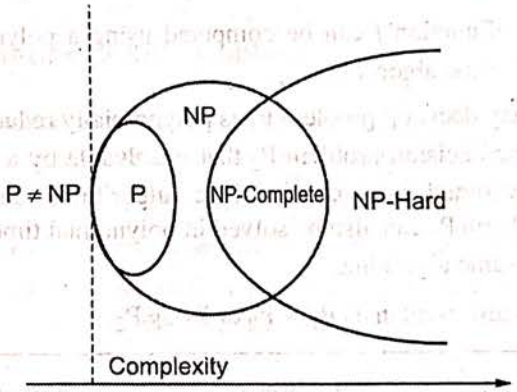
4. NP-Complete Class

▶ A decision problem P_1 is **NP-Complete** if,

(i) P_1 is an NP-Class problem (i.e. $P_1 \in NP$) and

(ii) Each NP-Class problem is polynomially reducible to P_1 (i.e., for each problem $P_2 \in NP$, $P_2 \alpha P_1$).

- Thus, P_1 is said to be NP-Complete if $P_1 \in NP$ and $P_1 \in$ NP-Hard.
- An NP-Complete problem is solvable in polynomial time iff P = NP. [This is the implication of Cook's theorem that says, "Satisfiability is in P iff P = NP."]
- Consequently, if an NP-Complete problem gets a solution in polynomial time, then all NP-Class problems could get their solutions in polynomial time. Thus, NP-Complete problems are the hardest problems in NP-Class.
- NP-Complete problems can have possible solutions only by applying the non-deterministic algorithms and their solutions can be verified in polynomial time.
- E.g., Circuit-satisfiability problem, vertex-cover decision problem, clique decision problem, m-coloring decision problem, TSP decision problem.
- The relationship among P, NP, NP-Hard and NP-Complete problems (assuming $P \neq NP$) is depicted in Fig. 2.5.1.



(TG14)**Fig. 2.5.1 : Generally assumed relationship among P, NP, NP-Hard and NP-Complete problems**

2.6 THEORY OF REDUCIBILITY

GQ. Consider two problems A & B. If $A \leq_p B$ and if B can be solved in polynomial time then prove that A can also be solved in polynomial time. **(6 Marks)**

GQ. Comment on the statement: "Two problems L1 and L2 are polynomially equivalent iff L1 α L2 and L2 α L1". **(4 Marks)**

GQ. What is Reduction in NP-Completeness proofs? **(6 Marks)**

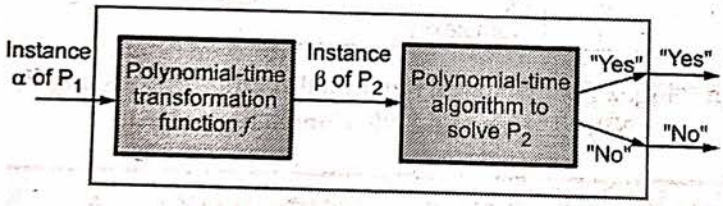
GQ. Write a short note on polynomial-time reduction. **(6 Marks)**

Suppose a kid has not yet learnt the procedure of multiplying two numbers, but he knows the procedure of

calculating a sum of two numbers. If someone defines the problem of multiplication as a problem of repetitive addition and asks that kid to solve it, then the kid can solve the problem of multiplication using the procedure of addition. This example resembles the theory of reducibility.

2.6.1 Polynomial-time Reduction

- Suppose P_1 and P_2 are two decision problems and a problem P_2 can be solved by a polynomial-time deterministic algorithm.
- A decision problem P_1 is polynomially reducible to a decision problem P_2 if there exists a transformation function f that converts all instances of P_1 to all instances of P_2 such that :
  - Function f maps all "yes" or "accept" instances of P_1 to all "yes" or "accept" instances of P_2 and all "no" or "reject" instances of P_1 to "no" or "reject" instances of P_2.
  - Function f can be computed using a polynomial-time algorithm.
- If any decision problem P_1 is polynomially reducible to some decision problem P_2 that is solvable by a known polynomial-time deterministic algorithm, then the problem P_1 can also be solved in polynomial time using the same algorithm.
- It is also written as $P_1 \propto P_2$ or $P_1 \leq_P P_2$.



(1G1)**Fig. 2.6.1 : Polynomial-time reduction**

- Thus, through polynomial-time reduction one can prove the easiness of computations for problem P_1 by using the easiness of computations for problem P_2.
- Two problems P_1 and P_2 are considered to be polynomially equivalent iff $P_1 \propto P_2$ and $P_2 \propto P_1$ (or it can be also written as iff $P_1 \leq_P P_2$ and $P_2 \leq_P P_1$).

2.6.2 Reducibility and NP-Completeness

- Theory of NP-Completeness describes the hardness or the complexity level of a problem.
- NP-Completeness proofs use polynomial-time reductions in a pessimistic way.
- Instead of proving the 'easiness' of a problem, NP-Completeness follows the theory of reducibility to prove the 'hardness' of a problem.
- Thus using the theory of reducibility it proves the non-existence of a polynomial-time algorithm to solve a particular problem.
- E.g., Suppose we want to prove that there is no polynomial-time algorithm to solve a decision problem P_2 and we know that there is no polynomial-time algorithm to solve another decision problem P_1.

☑ Proof

Let us assume a problem P_2 can be solved by a known polynomial-time algorithm.

(i) By applying polynomial-time reduction depicted in above Fig. 2.6.1 we would have a solution to P_1 using a polynomial-time solution to P_2. But it contradicts the known fact of the non-existence of any polynomial-time algorithm to solve P_1.

(ii) It falsifies our assumption that there exists a polynomial-time algorithm to solve P_2. Thus, using this proof by contradiction we can prove that there is no polynomial-time algorithm to solve P_2.

2.7 NP-COMPLETE PROBLEMS

UQ. Write a short note on NP-Completeness of algorithm and NP-Hard.
SPPU – Q. 3(b), May 18, 8 Marks

UQ. What are the steps to prove the NP-Completeness of a problem? Prove that the vertex cover problem is NP-Complete.
SPPU - Q. 4(b), May 19, 8 Marks

UQ. What is NP-Complete Algorithm? How do we prove that an algorithm is NP-Complete? (Give an example)
SPPU - Q. 4(b), Dec. 16, 6 Marks

- If a decision problem P_1 is known to be NP-Complete, then one can prove that any other decision problem P_2 is also NP-Complete using the theory of polynomial-time reduction (i.e. by showing $P_1 \propto P_2$).
- Thus, to prove the NP-Completeness of any decision problem there must be the "first" NP-Complete problem.
- The **circuit-satisfiability problem is the first proven NP-Complete problem** and hence it is used to prove

the NP-Completeness of any other decision problem by referring to the theory of polynomial-time reduction.

- The circuit-satisfiability problem: A combinational circuit formed by using OR, AND, NOT gates produce Boolean output based on a set of Boolean inputs. The circuit-satisfiability problem determines some set of inputs to this Boolean circuit to produce the output = 1(or TRUE).

Proofs for NP-Hard and NP-Complete problems

GQ. How do you prove that a problem is NP-hard? (4 Marks)

Proofs for NP-Hard and NP-Complete problems are based on the theory of reducibility.

Proof for the NP-Hardness

- Let a decision problem P_1 is to be proved as an NP-Hard problem.
- We know that an NP-Hard problem is at least as hard as the hardest NP-Class problem and NP-Complete problems are the hardest problems in NP-Class.
- Thus, if any NP-Complete problem P_2 is polynomially reducible to P_1 (i.e. $P_2 \alpha P_1$), then P_1 is NP-Hard.
- As per the Cook's theorem, all NP-Complete problems are polynomially reducible to the satisfiability (SAT) problem. So, if SAT α P_1 then P_1 is said to be an NP-Hard problem.

Proof for the NP-Completeness

- Let a decision problem P_1 is to be proved as an NP-Complete problem and we have another decision problem P_2 which is NP-Complete.
- If $P_1 \in NP$, $P_2 \alpha P_1$ then P_1 is NP-Complete.
- Since NP-Complete class = NP-Class $\cap$ NP-Hard class, if $P_1 \in NP$ and $P_1 \in$ NP-Hard, then P_1 is NP-Complete.
- Thus, if $P_1 \in NP$ and SAT α P_1 then P_1 is said to be an NP-Complete problem.

2.7.1 The 3-SAT Problem

- 3-SAT problem is a particular case of the satisfiability (SAT) problem.
- The satisfiability problem is also known as the **"Boolean satisfiability"** or **"propositional satisfiability"** problem.
- In 1971, Stephen Cook proved the first time that the satisfiability problem is NP-Complete.

Problem description

- An expression (or a formula) in the propositional calculus is formed by using Boolean variables $x_1, x_2, \ldots$ and the operations OR, AND, NOT. E.g. $(\neg x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ where $\vee$ denotes OR, $\wedge$ denotes AND, $\neg$ denotes NOT.
- The expression is considered to be **'satisfiable'** if there is some set of truth assignments to the variables in it that makes the expression TRUE. E.g. $[\neg x_1 \wedge x_2]$ is satisfied if x_1 = FALSE and x_2 = TRUE.
- **The satisfiability (SAT) problem** is to ascertain whether the given Boolean expression is satisfiable. E.g. $(\neg x_1 \wedge x_1)$ is un-satisfiable for any truth values of x_1.

Varieties of SAT problem

- **The CNF-satisfiability problem** is the SAT problem for Conjunctive Normal Form (CNF) expressions which are represented as $\wedge_{i=1}^{m} C_i$ where C_i's are the clauses each expressed as $\vee b_{ij}$; m is the number of clauses, b_{ij}'s are literals (variables or their negations) and j is the number of literals in each clause. E.g. $(\neg x_1 \vee \neg x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3 \vee x_4)$.
- **The 3-SAT problem** is the SAT problem for CNF expressions in which each clause contains at most three literals.

  E.g. $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$
- The 3-SAT problem is generalised to **k-SAT problem** in which CNF formula consists of clauses with at most k-literals in each clause.
- **The DNF-satisfiability problem** is the SAT problem for Disjunctive Normal Form (DNF) expressions which are represented as $\vee_{i=1}^{m} C_i$ where C_i's are the clauses each expressed as $\wedge b_{ij}$; m is the number of clauses, b_{ij}'s are literals and j is the number of literals in each clause. E.g. $(x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3)$
- The DNF SAT problem has a trivial solution if at least one of the clauses in a DNF expression is satisfiable.

- A clause in a DNF expression is a conjunction and it is satisfiable only if it does not have both x_i and $\neg x_i$ for some variable x_i in it.

☞ Non-deterministic algorithm

```
Algorithm 3-SAT_ND (E, n)
/* Input: E is a Boolean propositional expression composed of
3 variables x1, x2 and x3.
Output: Indication of success if E is satisfiable, otherwise
signal of failure. */
{
  //Guessing stage:
  for (i := 1; i ≤ 3 ; i+ +)
  {
    xi:= Choice(TRUE, FALSE) ;
                    /* Random choice of truth
                       assignments to x1, x2 and
                       x3.*/
  }
  //Verification stage:
  if E(x1, x2, x3)
    Success( ) ;
                /* Indicates success, if E is satisfiable
                   for a randomly selected choice of
                   truth assignments to x1, x2 and x3.*/
  else Failure( ) ;
                /*Results failure if E is un-satisfiable for
                  a randomly selected choice of truth
                  assignments to x1, x2, x3.*/
}
```

- The non-deterministic algorithm for a 3-SAT problem randomly assigns TRUE or FALSE value to each of the three variables in the given Boolean propositional expression.
- Thus, it non-deterministically determines one of the 2^3 possible truth assignments to 3 variables x_1, x_2, x_3 and checks whether an expression E (x_1, x_2, x_3) is TRUE for that random choice of truth assignments to x_1, x_2 and x_3.

☞ Complexity analysis

- To make a random choice of truth assignments to n variables (x_1, x_2,.... x_n), the non-deterministic time O(n) is needed. So to make a random choice of truth assignments to 3 variables (x_1, x_2, x_3), the non-deterministic time O(3) => constant time O(1) is needed.
- To verify the satisfiability of an expression E(x_1, x_2, x_3) for the random choice of truth assignments to x_1, x_2 and x_3 the required deterministic time directly proportional to the length 3 of the expression E is constant time O(1).

☞ Proof of the NP-Completeness

UQ. What is SAT and 3-SAT problem? Prove that 3-SAT problem is NP-Complete.

SPPU - Q. 4(a), May 18, 8 Marks

UQ. What is Boolean Satisfiability Problem? Explain 3-SAT problem. Prove 3-SAT is NP-Complete.

SPPU - Q. 3(b), May 19, 8 Marks

UQ. Explain in brief NP-Complete problem. Prove that the 3-SAT problem is NP-Complete.

SPPU – Q. 4(b), Dec. 17, 8 Marks

UQ. What is SAT and 3-SAT problem? Prove that the 3-SAT problem is NP-Complete.

SPPU - Q. 3(b), May 16, 8 Marks

GQ. What is 3-SAT problem? Prove that the 3-SAT problem is NP-Hard. **(8 Marks)**

☑ Proof:

By definition of NP-Complete problem, the 3-SAT problem is NP-Complete if (i) 3-SAT ∈ NP and (ii) 3-SAT ∈ NP-Hard.

(i) To prove 3-SAT ∈ NP

- For a given 3-CNF expression, we can randomly choose a set of truth assignments to the variables in it.
- Then in polynomial time, we can verify the given 3-CNF formula is evaluated to be TRUE for that chosen set of truth assignments.

(ii) To prove 3-SAT ∈ NP-Hard:

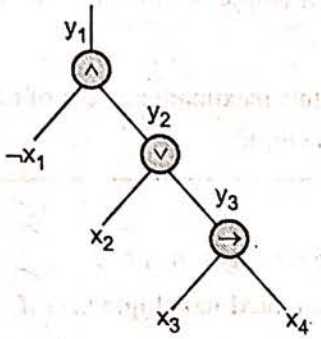
We show that SAT α 3-SAT

Step (1) :

- We represent a propositional Boolean expression E1 as a binary "parse" tree in which leaves represent literals and internal nodes represent connectives.
- Using the property of associativity, we fully parenthesize E1 such that each internal node in a corresponding sparse tree has 1 or 2 children.

- Let y_i represents the output at each internal node in a parse tree.
- We express E1 as the AND of output y_i at the root and the conjunction of the other outputs at each node. Let the resultant expression be E2.
- E.g. Let $E1 = \neg x_1 \wedge x_2 \vee (x_3 \rightarrow x_4)$
  - o We parenthesize E1 to get,

    $E1 = \neg x_1 \wedge (x_2 \vee (x_3 \rightarrow x_4))$
  - o We represent E1 as a binary parse tree as given in Fig. 2.7.1.



(1G)**Fig. 2.7.1**

$$\therefore\ E2 = y_1 \wedge (y_1 \leftrightarrow (\neg x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow (x_2 \vee y_3) \wedge (y_3 \leftrightarrow (x_3 \rightarrow x_4))$$

Step (2) :

- We convert each clause in the resultant expression E2 into CNF.
- For the same, we evaluate all truth assignments to all variables in E2 using a truth table.
- Based on it we construct a Disjunctive Normal Form (DNF) for truth assignments evaluating to 0.
- Then we convert this DNF formula into a CNF formula using DeMorgan's laws :

  (1) $\neg (a \wedge b) = \neg a \vee \neg b$ and

  (2) $\neg (a \vee b) = \neg a \wedge \neg b$

  Let this resultant CNF expression be E3 which is equivalent to original expression E1.
- E.g., In step (1) we have $E2 = y_1 \wedge (y_1 \leftrightarrow (\neg x_1 \wedge y_2))$ which is equivalent to E1.
  - o We construct a truth table for the same E2 as below :

Table 2.7.1

y_1	y_2	x_1	$y_1 \leftrightarrow (\neg x_1 \wedge y_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

  - o Considering the truth assignments evaluating to 0 in the truth table we form a DNF formula E3 equivalent to $\neg$ E2 as below :

$$E3 = (y_1 \wedge y_2 \wedge x_1) \vee (y_1 \wedge \neg y_2 \wedge x_1) \vee (y_1 \wedge \neg y_2 \wedge \neg x_1) \vee (\neg y_1 \wedge y_2 \wedge \neg x_1)$$

  - o We apply DeMorgan's law to E3 to get a CNF formulae E4 as below :

$$E4 = (\neg y_1 \vee \neg y_2 \vee \neg x_1) \wedge (\neg y_1 \vee y_2 \vee \neg x_1) \wedge (\neg y_1 \vee y_2 \vee x_1) \wedge (y_1 \vee \neg y_2 \vee x_1)$$

Step (3) :

- Finally, we convert the resultant CNF formula E4 into 3-CNF formula E5 such that each clause in it has exactly 3 distinct literals.
- Any clause C_i in a CNF expression E4 is added to a 3-CNF expression E5 by the following rules:

  (i) If C_i has exactly 3 distinct literals, then directly add it to E5.

  E.g., In the above example, all clauses in E4 have exactly 3 distinct literals so they are directly added to E5.

  Thus, we get a 3-CNF expression E5 equivalent to the original expression.

$$E1 = \neg x_1 \wedge x_2 \vee (x_3 \rightarrow x_4)$$

as below :

$$E5 = (\neg y_1 \vee y_2 \vee \neg x_1) \wedge (\neg y_1 \vee y_2 \vee \neg x_1) \wedge (\neg y_1 \vee y_2 \vee x_1) \wedge (y_1 \vee \neg y_2 \vee x_1)$$

(ii) If C_i contains 2 different literals i.e., $C_i = (p \vee q)$ then add third literal as r and $\neg$ r to make it $(p \vee q \vee r) \wedge (p \vee q \vee \neg r)$ as clauses in a 3-CNF expression E5. For any truth value of r (r = 0 or r = 1) one of these clauses results to 1 and other results to $(p \vee q)$.

(iii) If C_i contains only 1 literal i.e., $C_i = (p)$ then add 2 more literals as q, $\neg$ q, r and $\neg$ r to make clauses $(p \vee q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge (p \vee q \vee r) \wedge (p \vee \neg q \vee \neg r)$ in a 3-CNF expression E5.

For any truth values of q and r, one of these clauses results to (p) and the remaining 3 clauses result to 1.

- Thus, in polynomial time by following the aforementioned steps (1), (2), (3) any propositional Boolean expression can be converted to an equivalent 3-CNF expression. So, SAT $\propto$ 3-SAT $\Rightarrow$ 3-SAT $\in$ NP-Hard.
- Since both the conditions: (i) 3-SAT $\in$ NP and (ii) 3-SAT $\in$ NP-Hard, the 3-SAT problem is proved to be an NP-Complete problem.

2.7.2 The Clique Problem

It is a classic example of an NP-Complete problem.

- A **clique** is a complete subgraph of a given undirected graph. In a complete subgraph, all nodes are connected.
- The **size of a clique** is defined by the number of nodes in it.
- A clique with a maximum number of nodes is known as a **maximum clique**.

Maximal clique

(i) Consider an undirected graph G = (V, E) where V is a set of nodes and E is a set edges in G.

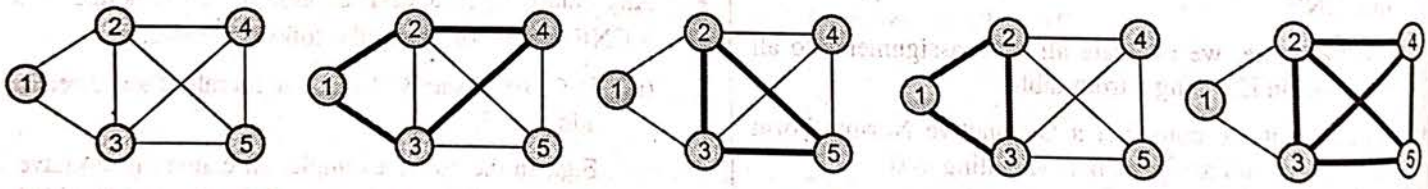
(ii) A subgraph $G' = (V', E')$ of a graph G where $V' \subseteq V$ and $E' \subseteq E$ is said to be a **maximal clique** in G if

(iii) G′ is a clique (all nodes in G′ are connected) and

(iv) There is no node $v \in (V - V')$ that connects all nodes in G′.

- Maximal cliques cannot be expanded by adding any more nodes into them.
- E.g., In graph G shown in Fig. 2.7.2(a):
  - There are 5 cliques of size 1 (i.e., all nodes),
  - There are 8 cliques of size 2 (i.e., all edges),
  - There are 5 cliques of size 3,

  There is one clique of size 4 i.e., a maximum clique in G.



(1G2)(a) Given graph G (1G3)(b) Not a clique as nodes 1 and 4 are not adjacent (1G4)(c) Non-maximal clique as node 4 connects to all nodes in a subgraph (1G5)(d) A maximal clique of size 3 (1G6)(e) A maximal clique of size 4. This is also a maximum clique.

Fig. 2.7.2 : Clique examples

Problem description

There are two general formulations of a clique problem:

(i) Clique optimization problem and (ii) Clique decision problem

The max-clique problem **is a type of optimization problem of finding a maximum clique with the highest number of nodes in a given undirected graph.**

- **The clique decision problem (CDP) is to check whether a given undirected graph has a clique of size at least s for some given s.**
- The theory of NP-Completeness refers to the clique decision problem (CDP).

☞ Non-deterministic algorithm for CDP

```
Algorithm CDP_ND (G, n, s)
/* Input: G = (V, E) is an undirected graph where V is a set of
nodes and E is a set of edges. n is the number of nodes in G. s
is the size of a clique in G.
Output: Indicates success if a graph has a clique of size at
least s, otherwise signals failure.*/
{
   V' := Φ;              /* V' is an initially empty set to store
                            nodes of a subgraph of G.*/
/*Guessing stage: Random choice of s distinct nodes forming a
subgraph of G.*/
   for (i := 1 ; i ≤ s ; i ++)
   {
      v := Choice(1, n) ;
      if (v∈ V')
         Failure( );
      V' := V' ∪ { v } ;          /*Adds a distinct node v
                                    to a set V'.*/
   }
   //Verification stage:
   for (all pairs (v0, v1) such that v0, v1 ∈ V' and v0 ≠ v1)
      if (an edge <v0, v1> ∉ E)
         Failure( );      /*If each pair of nodes in a set V'
                            is not adjacent, then that subgraph
                            is not a clique. If so, it indicates
                            failure. */
   Success( );          /*Indicates success if randomly selected
                          subgraph is a clique of size s. */
}
```

☞ Complexity analysis

- To make a random choice of a subgraph with s vertices among total n vertices of a given graph, the non-deterministic time O(n) is needed.
- To check whether a randomly chosen subgraph of s vertices is a clique in a given graph, the required deterministic time is $O(s^2)$.
- Thus the total non-deterministic time is $O(n + s^2) = O(n^2)$ as s has an upper bound O(n).

☞ Proof of the NP-Completeness

GQ. Prove that Clique problem is NP-Complete.

GQ. CNF-Satisfiability is polynomially transformable to the clique problem. Therefore, prove that the clique problem is NP-Complete. **(8 Marks)**

☑ Proof

By definition of NP-Complete problem, the Clique Decision Problem (CDP) is NP-Complete if (i) CDP ∈ NP and (ii) CDP ∈ NP-Hard.

(i) To prove that CDP ∈ NP

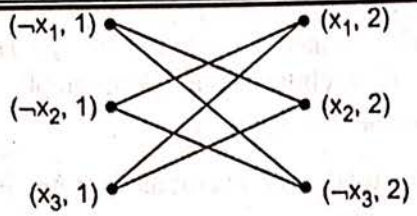
For a given undirected graph G = (V, E) we can randomly choose a subgraph G′ = (V′, E′) of G such that $V' \subseteq V$ and $E' \subseteq E$. Then in polynomial time we can verify that a subgraph G′ is a clique in G by testing whether, for each pair $v_0, v_1 \in V'$, there is an edge $< v_0, v_1 > \in E$.

(ii) To prove CDP ∈ NP-Hard

- We show that CNF-SAT ∝ CDP. To prove that CNF-SAT ∝ CDP, we can represent a CNF-formula F of length m as a graph G = (V, E) such that G has a clique of size at least m iff F is satisfiable.
- Let $F = \wedge_{i=1}^{m} c_i$ be a CNF-formula of length m. Let x_i, $1 \le i \le n$ be the variables in F. Then any F can be represented as G = (V, E) as given below :

  $V = \{(b, i) \mid b$ is a literal in clause C_i in a CNF-formula F, $1 \le i \le m\}$

  $E = \{<(b, i), (d, i)> \mid i \ne j$ and $b \ne \neg d\}$
- This graph G = (V, E) can be easily constructed from a CNF- formula F in polynomial time.
- E.g., $F = (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. The corresponding graph is depicted in Fig. 2.7.3. F has a length of 2 and G has six cliques of size 2.
- Consider a clique with vertices $\{(\neg x_1, 1), (x_2, 2)\}$. By assigning a TRUE value to all 3 variables x_1, x_2 and x_3, the CNF- formula F is satisfiable.



(1G7)**Fig. 2.7.3 : A graph G constructed from a CNF formula E**

- We must show that the construction of graph G from formula F is a polynomial-time reduction: **A formula F is satisfiable iff a graph G has a clique of size ≥ s.**
  - (i) If F is satisfiable, then there must be at least one literal in each clause C_i with value TRUE. Let $S = \{(b, i) \mid b = \text{TRUE in } C_i\}$ be a set of exactly one such (b, i) for each i, $1 \le i \le m$. Any two nodes (b, i) and (d, i) in S are connected by an edge in G since $i \ne j$ and both b and d have the value TRUE. Thus, S is a clique of size m in graph G.
  - (ii) Similarly, consider $G' = (V', E')$ is a clique of size m in G. Since no edges in G connect nodes from the same clause, G′ has exactly one node per clause. As G does not have edges connecting a literal and its negation, we can assign TRUE values to all literals corresponding to vertices in a clique G′. Thus we get at least one literal per clause with value TRUE. That makes each clause satisfiable and hence a CNF-expression F is satisfied.
- Since CNF-SAT ∝ CDP, CDP is an NP-hard problem.
- Thus, both the conditions: (i) CDP ∈ NP and (ii) CDP ∈ NP-Hard, are proved. So, the CDP is proved to be NP-Complete.
- If the number of clauses in F is m then the corresponding G having a clique of size at least m is constructed in O(m) time.

2.7.3 The Vertex Cover Problem

UQ. State Vertex cover problem and prove that Vertex Cover problem is NP-Complete.

SPPU - Q. 3(b), Dec. 19, 8 Marks, Q. 3(a), Dec. 17, 8 Marks

UQ. Prove that Vertex cover problem is NP-Complete.

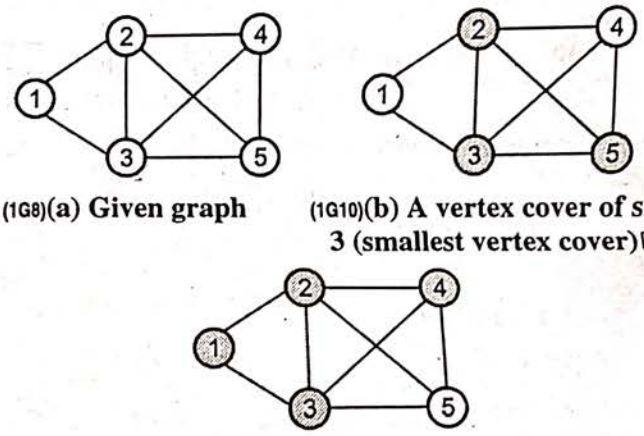
SPPU - Q. 4(b), May 17, 8 Marks

- It is a classic example of an NP-Complete problem.
- It is also known as **"node cover problem"**.

▶ **A vertex cover** of a given undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ iff each edge $< v_0, v_1 > \in E$, $v_0, v_1 \in V$ is incident to at least one node in V' that means either $v_0 \in V'$ or $v_1 \in V'$ or $v_0, v_1 \in V'$.

▶ **The size of a vertex cover** (|V′|) is the number of vertices in it.

- Since each node in a vertex cover V′ "covers" its incident edges, all nodes in V′ covers all the edges in E of a given graph G = (V, E).



(1G8)**(a) Given graph** (1G10)**(b) A vertex cover of size 3 (smallest vertex cover)**□

(1G9)**(c) A vertex cover of size 4**

Fig. 2.7.4 : Vertex Cover Examples

☞ Problem description

- A vertex cover problem has two general variants as below :
  - (i) Vertex cover optimization problem
  - (ii) Vertex cover decision problem

▶ **The vertex cover optimization problem** is to determine a vertex cover with the minimum number of nodes for a given undirected graph.

▶ **The vertex cover decision problem** (VCDP) is to check whether a given undirected graph has a vertex cover of size at most s for some givens.

- The theory of NP-Completeness refers to the vertex cover decision problem (VCDP).

☞ Non-deterministic algorithm of VCDP

```
Algorithm VCDP _ ND (G, n, s)
/* Input: G = (V, E) is an undirected graph where V is a set of
vertices and E is a set of edges. n = | V | = the number of
vertices in G. s is a size of a vertex cover of G.
Output: Indicates success if a graph has a vertex cover of size
at most s, otherwise signals failure. */
{
  V' := Φ;   /* V' is an initially empty set to store randomly
                selected nodes of G. V' describes the
                cover of G. */
/* Guessing stage: Random choice of s distinct nodes forming
a cover of graph G. */
  for (i := 1 ; i ≤ s ; i ++)
  {
     v := Choice(1, n);
     if (v ∈ V')
        Failure( ) ;
     V' := V' ∪ {v}; // Adds a distinct node v to a set V'.
  }
     // Verification stage:
  for (each edge < v0 , v1 > ∈ E and v0 , v1 ∈ V )
  {
     if ((v0 ∈ V') || (v1 ∈ V') || (v0, v1 ∈ V'))
        Success( ); //
     else Failure();      /*If all nodes in randomly selected
                            set V' covers all edges of the graph
                            G then it indicates success,
                            otherwise signals failure*/
  }
}
```

☞ Complexity analysis

- To make a random choice of a set with S nodes among total n nodes of a given graph, the non-deterministic time O(n) is needed.
- To check whether a randomly chosen set of nodes is a vertex cover of a given graph, the required deterministic time is O(|E|).
- Thus, the total non-deterministic time is O (|V| + |E|).

☞ Proof of NP-Completeness

GQ. Prove that vertex cover problem is NP-Complete.

☑ Proof :

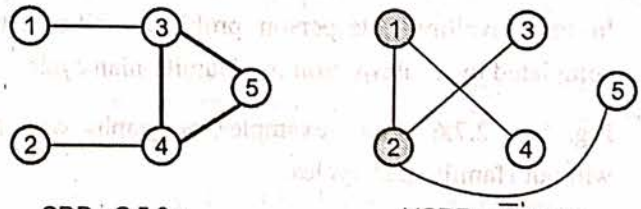
By definition of NP-Complete problem, the vertex cover decision problem (VCDP) is NP-Complete if (i) VCDP ∈ NP and (ii) VCDP ∈ NP-Hard.

(i) To prove that VCDP ∈ NP

For a given undirected graph G = (V, E), we can randomly choose a set of vertices $V' \subseteq V$ and $|V'| = s$ for some given size s. Then in polynomial time, we can verify that set V' is a vertex cover of G by testing whether each edge $< v_0, v_1 > \in E$ and $v_0, v_1 \in V$ is incident to at least one node in V'.

(ii) To prove that VCDP ∈ NP-Hard

- We will show that CDP (clique decision problem) ∝ VCDP.
- The proof of CDP ∝ VCDP refers the concept of "complement" of a graph. For a undirected graph G = (V, E) its complement is given by $\bar{G} = (V, \bar{E})$ where $\bar{E} = \{< v_0, v_1 > \mid v_0, v_1 \in V \text{ and } < v_0, v_1 > \notin E\}$.
- Let < G, n, s > be an instance of CDP where G is a given undirected graph with n vertices and a clique of size at least s.
- This instance of CDP can be reduced to an instance of VCDP in polynomial time by constructing the complement $\bar{G} = (V, \bar{E})$ of a graph G. Then $\bar{G}$ has a vertex cover of size at most (n – s). So, by polynomial-time reduction a CDP instance < G, n, s > is transformed to a VCDP instance $< \bar{G}, n, n - s >$.
- E.g., Consider a CDP instance < G, 5, 3 > as depicted in Fig. 2.7.5(a). G has 5 nodes and a clique of size 3 including nodes 3, 4 and 5.
- This CDP instance is polynomially reducible to a VCDP instance $< \bar{G}, 5, 2 >$ as depicted in Fig. 2.7.5(b) below $\bar{G}$ has 5 nodes a vertex cover of size 2 including nodes 1 and 2.



CDP <G,5,3>
Clique of size 3 = {3, 4, 5}

VCDP < $\bar{G}$, 5, 2>
Node cover of size 2 = {1, 2}

(1G11) (1G12)

Fig. 2.7.5 : Example CDP ∝ VCDP

- We must show that the construction of $\bar{G} = (V, \bar{E})$ of VCDP instance from G = (V, E) of CDP instance is a polynomial-time reduction: **A graph G has a clique of size ≥ s iff the graph $\bar{G}$ has a vertex cover of size ≤ (|V| – s).**

(i) If G = (V, E) has a clique G′ = (V′, E′) where V′ ⊆ V, E′ ⊆ E and | V′ | = s. $\bar{E}$ does not contain the edges connecting nodes in a clique G′, that means $\bar{E} \cap E' = \Phi$. Hence, the remaining (|V| – |V′|) = (n – s) vertices in $\bar{G}$ must cover all edges in $\bar{E}$.

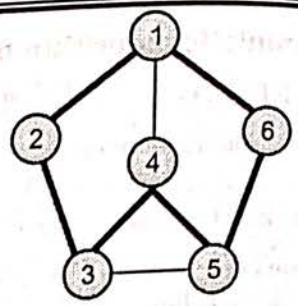
(ii) Similarly, V′ ⊆ V is a vertex cover of $\bar{G} = (V, \bar{E})$ then V – V′ must form a clique in G.

- Thus we have shown that CDP ∝ VCDP. Also, we know that CNF-SAT ∝ CDP. So, by transitivity CNF-SAT ∝ VCDP. Thus it is proved that VCDP ∈ NP-Hard.
- Since both the conditions: (i) VCDP ∈ NP and (ii) VCDP ∈ NP-Hard are proved, the vertex cover decision problem is NP-Complete.

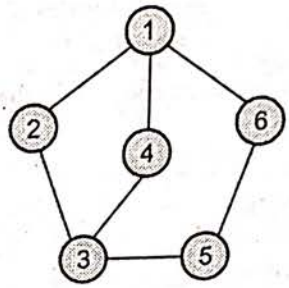
2.7.4 The Hamiltonian Cycle Problem

Problem description

- **A Hamiltonian cycle or circuit** of a given graph is a round-trip path that starts at a specific node, visits all nodes excluding the starting node in the graph exactly once, and ends at the starting node.
- A graph can have zero, one or multiple Hamiltonian cycles in it.
- It applies to both directed and undirected graph.
- In the travelling salesperson problem (TSP), a tour completed by a salesperson is a Hamiltonian cycle.
- E.g. Fig. 2.7.6 shows examples of graphs with and without Hamiltonian cycles.



(1E33)(a) G with Hamiltonian cycles: (1-2-3-4-5-6-1) and (1-6-5-4-3-2-1)



(1E34)(b) G with no Hamiltonian cycle

Fig. 2.7.6 : Examples of graphs with and without Hamiltonian cycles

- **Hamiltonian cycle decision problem (HCDP)** is to find whether a given undirected graph has any Hamiltonian cycle in it.

Non-deterministic algorithm

```
Algorithm HCD_ND (G, n, s)
/* Input: G = (V, E) is an undirected graph where V is a set of
nodes and E is a set of edges. n is the number of nodes in G.
Output: Indicates success if a graph has a Hamiltonian cycle
otherwise signals failure.*/
{
    V′ := Φ;                /* V′ is an initially empty set to store
                               nodes of a subgraph of G.*/
/* Guessing stage: Random choice of a sequence of n vertices
of G. */
    for (i := 1 ; i ≤ n ; i ++)
    {
        v := Choice (1, n) ;
        if (v ∈ V′)
            Failure( );
        V′ := V′ ∪ {v} ;    //Adds a distinct node v to a set V′
    }
```

```
//Verification stage:
if ( an edge <v_n, v_1> ∉ E , v_1, v_n ∈ V'))
        Failure( );      /* If there is no path from last
                            node to the first node of selected
                          random sequence to complete a
                          cycle then it indicates failure. */
else
{
    for (edge <v_i, v_i+1> such that v_i, v_i+1 ∈ V', v_i ≠ v_i+1,
        1 ≤ i < n)
        if (<v_i, v_i+1> ∉ E)
          Failure( );
                          /*If each ordered pair of nodes in a
                          set V' is not adjacent then there is no
                          Hamiltonian path. If so, it indicates
                          failure. */
    Success( );           /*Indicates success if randomly
                          selected sequence of n vertices form
                          a Hamiltonian cycle. */
}
}
```

☞ Complexity analysis

- To make a random choice of a sequence of n vertices of a given graph, the non-deterministic time O(n) is needed.
- To check whether a randomly chosen sequence of n vertices is a Hamiltonian cycle in a given graph, the required deterministic time is O(|E|).

– Thus, the total non-deterministic time is O (|V| + |E|).

☞ Proof of the NP-Completeness

GQ. Prove that the Hamiltonian Cycle decision problem is NP-Complete. (8 Marks)

☑ Proof:

By definition of NP-Complete problem, the Hamiltonian cycle decision problem (HCDP) is NP-Complete if (i) HCDP ∈ NP and (ii) HCDP ∈ NP-Hard.

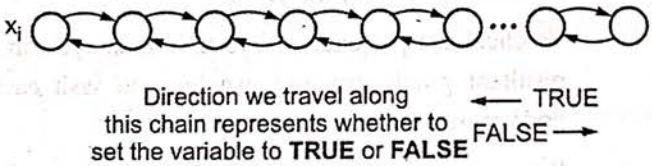
(i) To prove that HCDP ∈ NP:

For a given undirected graph G = (V, E), we can randomly choose a sequence of n vertices. Then in polynomial time, we can verify that a selected sequence of n vertices forms a Hamiltonian cycle by testing whether each node of G is visited exactly once excluding the first node which is visited twice to complete a cycle.

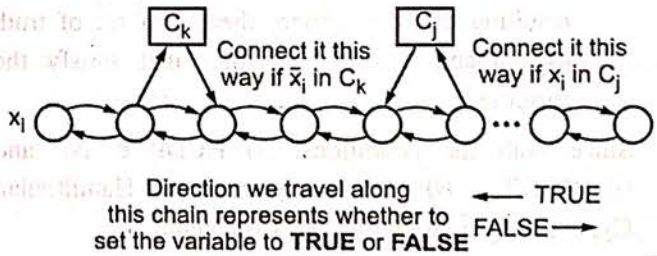
(ii) To prove that HCDP ∈ NP-Hard:

- We can show that either VCDP (vertex cover decision problem) ∝ HCDP or SAT ∝HCDP.
- Here we show SAT ∝HCDP.
- Consider a propositional Boolean formula E representing an instance of the SAT problem with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$.
- **Reduction Steps** (very superficial level):
  (1) Construct a specific graph structure (a "gadget") that represents the variables $x_1, \ldots, x_n$ of a given formula E.
  (2) Use some another graph structure to represent the clauses $C_1, \ldots, C_k$ of a given formula E.
  (3) Connect these two graph structures up such that the resulting graph structure encodes the given formula E.
  (4) Show that the resulting graph structure has a Hamiltonian cycle iff the given propositional Boolean formula E is satisfiable.

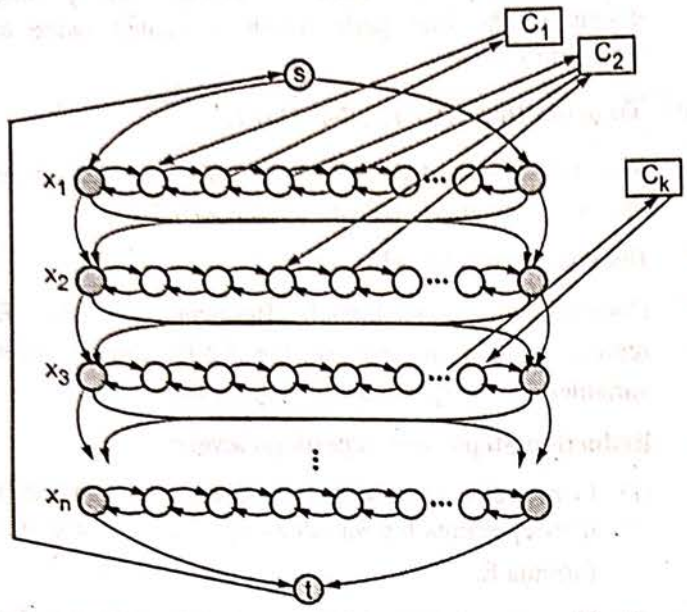
Fig. 2.7.7(a),(b),(c) depict the different reduction steps of SAT ∝HCDP.



(2D1)(a) : A gadget representing the variables $x_1, \ldots, x_n$ of a formula E



(2D1)(b) : Add a new vertex for each clause $C_1, \ldots, C_k$ of a formula E



(2D1)(c) : Connecting up the paths

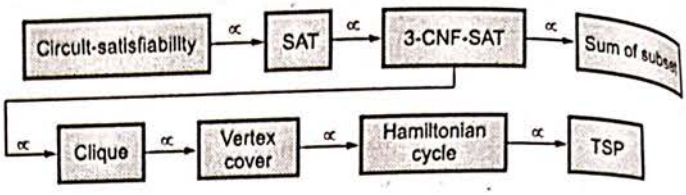
Fig. 2.7.7 : (a),(b),(c) Reduction steps of SAT ∝ HCDP

- Now we show that the resulting graph structure has a Hamiltonian cycle iff the given propositional Boolean formula E is satisfiable.
  - A Hamiltonian path in the resulting graph structure encodes a set of truth assignments for the variables.
  - These truth values depend on the direction in which each chain is traversed.
  - To check the presence of a Hamiltonian cycle in a resultant graph structure, we have to visit each node representing a clause.
  - We can only visit a clause if we satisfy it by assigning the TRUE value to one of its terms.
  - Thus, if there is a Hamiltonian cycle present in the resulting graph structure, there is a set of truth assignments for the variables that satisfy the formula E.
- Since both the conditions: (i) HCDP ∈ NP and (ii) HCDP ∈ NP-Hard are proved, the Hamiltonian Cycle decision problem is NP-Complete.

UQ. Explain NP-Hard Hamiltonian cycle problem.

SPPU - Q. 4(b), Dec. 19, 8 Marks

2.7.5 Reducibility Structure of NP-Complete Problems



(1G13)**Fig. 2.7.8 : Reducibility structure of some NP-Complete problems**

- All proofs of the NP-Completeness typically follow the reducibility structure based on the NP-Completeness of the circuit-satisfiability problem as shown in Fig. 2.7.8.
- It obeys the transitive property. So if a problem $P_1 \propto$ problem P_2 and problem $P_2 \propto$ problem P_3 then the problem $P_1 \propto$ problem P_3.

Summary

- **A priori analysis** is made before the execution of an algorithm and is independent of experimental evidence.
- **A posteriori analysis** is made after the execution of an algorithm and is dependent on experimental evidence.
- **Time Complexity** is the time needed for the completion of an algorithm.
- **Space Complexity** is the amount of memory needed for the completion of an algorithm.
- **The framework for efficiency analysis** of an algorithm has four major components: the growth rate function, estimate of time complexity, best-case, worst-case and average-case analysis and asymptotic efficiency.
- The efficiency of an algorithm is expressed in terms of asymptotic notations.
- O-notation gives an upper bound, Ω-notation gives a lower bound and Θ-notation gives a tight bound.
- **Computational complexity** categorizes numerous computational problems based on their inherent difficulty and defines the relations between them.
- **An optimization problem** determines the optimal value of a specified cost function.
- A decision problem produces only two possible outputs ("yes" or "no") or ("TRUE" or "FALSE") for each input..
- **A decidable problem** can be solved by a decision algorithm either in polynomial time or in non-polynomial time.