GAUHATI UNIVERSITY
Centre for Distance and Online Education

SECOND SEMESTER
(under CBCS)

M.Sc.- IT

Paper: INF 2026
ALGORITHMS AND COMPLEXITY THEORY

Contents:

SLM Development Team:

HoD, Department of Computer Science, GU
Programme Coordinator, MSc-IT, GUCDOE (GUIDOL)
Prof. Shikhar Kr. Sarma, Department of IT, GU
Dr. Khurshid Alam Borbora, Assistant Professor, GUCDOE (GUIDOL)
Dr. Swapnanil Gogoi, Assistant Professor, GUCDOE (GUIDOL)
Mrs. Pallavi Saikia, Assistant Professor, GUCDOE (GUIDOL)
Dr. Rita Chakraborty, Assistant Professor, GUCDOE (GUIDOL)
Mr. Hemanta Kalita, Assistant Professor, GUCDOE (GUIDOL)

Contributors:

Dr. Tarali Kalita Asstt. Prof., NERIM, Guwahati	(Block I : Unit- 1)
Dr. Utpal Barman Asstt. Prof., GIMT, Azara	(Block I: Unit- 2)
Ms. Pinki Pathak Asstt. Prof., Pragjyotish College	(Block I: Unit: 3, Block II: Units: 1,2, & 3)
Dr. Swapnanil Gogoi Asstt. Prof., GUCDOE	(Block I : Unit- 4)
Dr. Sarat Kumar Chettri Asstt. Prof., Assam Don Bosco University, Azara	(Block III: Units- 1, 2 & 3)
Mrs. Syeda Shamim Shabnam Asstt. Prof., Pragjyotish College	(Block IV: Units- 1 & 2)
Mr. Mridul Jyoti Roy Asstt. Prof., Dept.. of Computer Science and Engineering, Assam Engineering College, Guwahati	(Block IV: Unit- 3)

Course Coordination:

Director	CDOE, Gauhati University
Prof. Anjana Kakoti Mahanta	Prof., Dept. Computer Science, G.U.
Dipankar Saikia	Editor SLM, GUCDOE

Content Editing:

Dr. Kshirod Sarmah	Asstt. Prof., Deptt. of Computer Science, PDUAM, Goalpara, Assam

Cover Page Designing:

Bhaskar Jyoti Goswami	CDOE, Gauhati University

ISBN:

May, 2023

BLOCK I:
ANALYSIS OF ALGORITHMS

UNIT 1: INTRODUCTION TO ALGORITHMS

Unit Structure:

1.1   INTRODUCTION

This unit gives an overview of algorithms and their place in modern computing systems. In this unit you will learn what an algorithm is and reasons to learn algorithms. Also you can learn advantages and disadvantages of the algorithm. In this unit, you will see the first algorithms, which solve the problem of sorting a sequence of n numbers. For sorting we examine insertion sort and pseudocode of the insertion sort written here, which is used to analyze the algorithm in

three cases namely the best case, average case and worst case. This unit also gives an idea about the rate of growth of algorithm.

1.2 UNIT OBJECTIVES

After going through this unit, you will be able to,
- Understand the concept of algorithm with its pros and cons.
- Learn about the needs of algorithm.
- Calculate the running time for algorithm
- Analyzing an algorithm in best case, average case and worst case.
- Based on running time calculate the order of growth of an algorithm.

1.3 ALGORITHMS

The word algorithm has been derived from the Persian author's name, Abu Ja 'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who has written a textbook on Mathematics. The word is taken based on providing a special significance in computer science. The algorithm is understood as a method that can be utilized by the computer as when required for providing solutions to a particular problem.

An *algorithm* is a well-defined computational procedure that takes some value or set of values as *input* and produces some value or set of values as *output*. Therefore, an algorithm is a sequence of computational steps that transform the input into the output.

In mathematics and computer science, an **algorithm** is a finite sequence of well-defined, computer-implementable instructions, used to solve a class of specific problems or to perform a computation. Algorithms are always unambiguous and are used for performing calculations, data processing, automated reasoning, and other tasks.

For example, we might need to sort a sequence of numbers into increasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools. Here we are formally define the *sorting problem*:

Input: A sequence of n elements $\{A_1, A_2, A_3, \ldots\ldots\ldots\ldots A_n\}$.
Output: A permutation (reordering) $\{a_1, a_2, a_3, \ldots\ldots\ldots\ldots a_n\}$ of the input sequence such
that $a_1 \leq a_2 \leq a_3 \leq \ldots\ldots\ldots\ldots \leq a_n$.

For example, given the input sequence (35, 42, 69, 16, 42, 78), a sorting algorithm returns as output the sequence (16, 35, 42, 42, 69, 78). Such an input sequence is called an *instance* of the sorting problem. In general, an *instance of a problem* consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. Hence as a result, we have a large number of good sorting algorithms at our disposal. For a given application which algorithm is best depends on factors like the number of items to be sorted, the extent to which the items are already somewhat sorted, possible restrictions on the item values, the architecture of the computer, and the kind of storage devices to be used (main memory, disks, or even tapes).

An algorithm is said to be *correct* if, for every input instance, it terminates with the correct output. We can say that a correct algorithm *solves* the given computational problem. An incorrect algorithm might halt with an incorrect answer or it might not halt at all on some input instances. In contrast to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate. Generally, however, we shall be concerned only with correct algorithms. An algorithm can be specified as a computer program, or even as a hardware design, written in English and the only requirement is that the specification must provide a precise description of the computational procedure to be followed.

1.3.1 Characteristics of Algorithms

- o **Input:** An algorithm should externally supply zero or more quantities.
- o **Output:** An algorithm results in at least one quantity.
- o **Definiteness:** Every instruction should be clear and ambiguous.

- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** All instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** An algorithm must be feasible enough to produce each instruction.
- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.
- **Efficient:** Efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space incorporating the acceptable limit of development time.
- **Independent:** An algorithm mainly focuses on the input and the procedure required for deriving the output instead of depending upon the language.

1.3.2 Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language using English, it becomes easy to understand the step-by-step of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors, occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by using any high-level language.

1.3.3 Disadvantages of an Algorithm

- Big and complex tasks are difficult to put in Algorithms.
- Developing algorithms for complex problems would be time-consuming.

o It is difficult to show Branching and Looping in Algorithms.

1.3.4 Need of Algorithm

Before implementing any algorithm as a program, it is better to find out a good algorithm in terms of time and memory. A good design can produce a good solution. An algorithm is needed to understand the basic idea of the problem and also to find an approach to solve the problem. It gives a clear description of requirements and goal of the problem to the designer. Using an algorithm we can measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases). Also we can analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm. An algorithm helps to convert art into a science. Overall, it is the best method of description without describing the implementation detail.

STOP TO CONSIDER

An *algorithm* is a well-defined computational procedure that takes some value or set of values as *input* and produces some value or set of values as *output*. An algorithm is said to be *correct* if, for every input instance, it terminates with the correct output.

CHECK YOUR PROGRESS - I

1) The word _____ comes from the name of a Persian mathematician Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.

2) An algorithm is a sequence of _____ steps that transform the input into the output.

3) The _____ of an algorithm results in at least one quantity.

4) Efficiency is measured in terms of _____ and _____ required by an algorithm to implement.

5) Input of an algorithm should externally supply _____ or _____ quantities.

6) It is difficult to show Branching and Looping in _____.

7) An algorithm helps to convert art into a _____.

1.4 CONCEPT IN ALGORITHM ANALYSIS

The term **"analysis of algorithms"** was coined by Donald Knuth. Theoretically in analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input.

Analysis of algorithm is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with arbitrary length of inputs. Algorithm analysis is the determination of the amount of time and space resources required to execute it.

For using an algorithm for a specific problem, we have to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm. One algorithm is often quite different from another, though the objective of this algorithm is the same. For example, we know that the sorting of a set of numbers can be done using different algorithms. For the same input, number of comparisons performed by one algorithm may vary with others. Therefore, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required for each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size of memory for storage required. However, the main concern of analysis of algorithms is the required time or performance. In general, we perform the following types of analysis −

- **Worst case** :- The function which performs maximum number of steps taken on input data of size **n**.

- **Best case** :- The function which performs minimum number of steps taken on input data of size **n.**

- **Average case** :- The function which performs An average number of steps taken on input data of size **n.**

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

1.5 TIME AND SPACE COMPLEXITY

The efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

Space Complexity

Space complexity of an algorithm means the amount of memory space needed the algorithm in its life cycle. Space needed for an algorithm is equal to the sum of the following two components:

- **A fixed part** that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), which are not dependent of the size of the problem.

- **A variable part** is a space required by variables, which is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Space complexity S(P) of an algorithm P is S(P) = A + SP(I), Where A is represented as the fixed part and SP(I) is represented as the variable part of the algorithm which depends on instance characteristic I. Consider the following example that tries to explain the concept,

Algorithm

SUM(R,Q)

Step1:- Start

Step 2:- P$\leftarrow$ R + Q + 10

Step3:- Stop

Here three variables P, Q and R and one constant are used. Therefore S(P) = 1+3. Now space is dependent on data types for given constant types and variables and it will be multiplied accordingly.

Time Complexity

Time Complexity of an algorithm means, the amount of time required by the algorithm to execute to completion. Time requirements can be denoted by numerical function T(N), where T(N) can be measured as the number of steps, provided each step takes constant time.

For example, in case of addition of two n-bit integers, N steps are used. Hence, the total computational time is T(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that T(N) grows as input size increases.

STOP TO CONSIDER

- The term **"analysis of algorithms"** was coined by Donald Knuth. Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size of memory for storage required.

- Time Complexity of an algorithm means, the amount of time required by the algorithm to execute to completion.

- Space complexity of an algorithm means the amount of memory space needed the algorithm

---

CHECK YOUR PROGRESS - II

8) The term "analysis of algorithms" was coined by _____.

9) _____ is the function which performs the maximum number of steps on input data of size n.

10) Average case is the function which performs an _____ number of steps on input data of n elements.

11) Space complexity of an algorithm means the amount of

_____ needed the algorithm in its life cycle.

12) A _____ is a space required by variables, which is totally dependent on the size of the problem.

13) _____ of an algorithm means, the amount of time required by the algorithm to execute to completion.

1.6   ANALYZING ALGORIHMS

Analyzing an algorithm means predicting the resources that the algorithm requires. Though, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, we can identify a most efficient algorithm by analyzing several candidate algorithms for a problem. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process. Before analyzing an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs.

Here we shall assume a generic uniprocessor, *random-access machine (RAM)* model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, without any concurrent operations. We should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Also we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we could sort in just one instruction. Such RAM would be unrealistic, since real computers don't have such instructions. Our guide, therefore, is how real computers are designed. RAM model contains instructions commonly found in real computers: arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Such type of instruction takes a constant amount of time.

The data types for the RAM model are integer and floating point (for storing real numbers). Here we assume a limit on the size of each

word of data. For example, when we are working with inputs of size n, typically assume that integers are represented by c lg n bits for some constant c ≥ 1. We need c ≥ 1 so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size grows arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time, clearly an unrealistic scenario.)

Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constant time instruction? In general case, no; it takes several instructions to compute xy (where x and y are real numbers). But in restricted situations, however, exponentiation is a constant-time operation. Most computers have a "shift left" instruction, which in constant time shifts the bits of an integer by k positions to the left. In many computers, shifting the bits of an integer by one position to the left is equivalent to multiplication by 2, so that shifting the bits by k positions to the left is equivalent
to multiplication by 2k. Hence, such computers can compute 2k in one constant-time instruction by shifting the integer 1 by k positions to the left, as long as k is no more than the number of bits in a computer word. We will try to avoid such gray areas in the RAM model, but we will treat computation of 2k as a constant-time operation when k is a small enough positive integer.

In the RAM model, we do not attempt to model the memory hierarchy which is common in contemporary computers. i.e., we do not model caches or virtual memory. Several computational models attempt to account for memory-hierarchy effects, and they are sometimes significant in real programs on real machines. Models that include the memory hierarchy are quite a bit more complex compared to the RAM model, and so they can be difficult to work with. However, RAM-model analyses are usually excellent predictors of performance on actual machines. Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. As the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

Though we typically select only one machine model to analyze a given algorithm, we still face many choices in deciding how to express our analysis. We would like a way, which is simple to write and manipulate, shows the important characteristics of an algorithm's resource requirements, and suppresses tedious details.
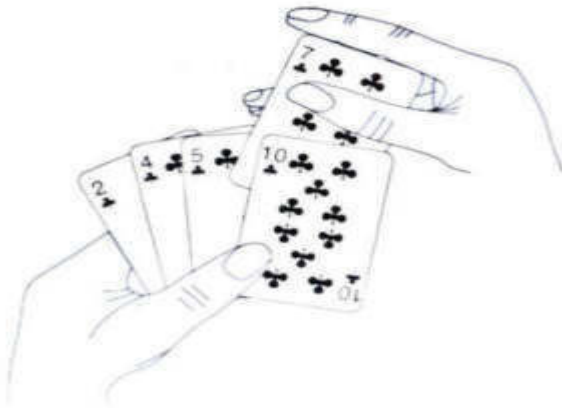
Insertion sort

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in 1.2:

Input: A sequence of n elements $\{A_1, A_2, A_3,\ldots\ldots\ldots A_n\}$.

Output: A permutation (reordering) $\{ A_1, A_2, A_3,\ldots\ldots\ldots A_n \}$ of the input sequence such

that $A_1 \leq A_2 \leq A_3 \leq \ldots\ldots\ldots \leq A_n$.

.

The elements that we wish to sort are also known as the ***keys***. Conceptually we are sorting a sequence, the input comes to us in the form of an array with n elements.


Figure 1.1 Sorting a hand of cards using insertion sort

We start with an efficient algorithm ***insertion sort***, for sorting a small number of elements. Insertion sort works like the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. Then we remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare the card with each of the cards already in the hand, from right to left, as shown in Figure 1.1. At all the times, the cards held in the left hand are sorted, and these cards were

originally the top cards of the pile on the table. Here we present a pseudocode for insertion sort as a procedure named INSERTION_SORT, which takes as a parameter an array Arr[1….n] containing a sequence of length n that is to be sorted. (In the code, the number n of elements in Arr is denoted by Arr.*length*). The algorithm sorts the input elements ***in place***: it rearranges the elements within the array Arr, with at most a constant number of them stored outside the array at any time. The input array Arr contains the sorted output sequence of elements when the INSERTION_SORT procedure is finished.
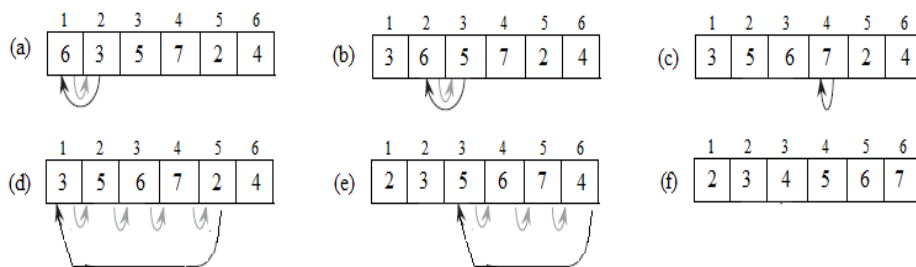
Figure 1.2 The operation of INSERTION_SORT on the array Arr = (6,3,5,7,2,4). Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. From **(a) to (e)** the iterations of the **for** loop of lines 1–8. And **(f)** The final sorted array.

INSERTION_SORT(Arr)
```
1 for j = 2 to Arr.length
2       key = Arr[j]
3       // Insert Arr[j] into the sorted sequence Arr[1….. (j -1)].
4       i = j - 1
5       while i > 0 and Arr[i] > key
6                Arr[i+1]=Arr[i]
7                i = i - 1
8       Arr[i + 1]= key
```

Loop invariants and the correctness of insertion sort

Figure 1.2 shows how this algorithm works for Arr = (6, 3, 5, 7, 2, 4). The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by j, the subarray consisting of elements Arr[1…..(j -1)] constitutes the currently sorted hand, and the remaining subarray Arr [j+1…. n] corresponds to the pile of cards still on the table. In fact, elements Arr[1…..(j - 1)] are the elements *originally* in positions 1 through (j - 1), but now in sorted order. We state these properties of Arr[1…(j − 1)] formally as a ***loop invariant***:

At the beginning of each iteration of the **for** loop of lines 1–8, the subarray Arr[1….(j − 1)] consists of the elements originally in Arr[1…..(j − 1)], but in sorted order. We use this loop invariants to help us understand why an algorithm is correct. Three things about a loop invariant we must show here:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop and it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps to show the algorithm is correct.

The first two properties hold, the loop invariant is true prior to every iteration of the loop. (Of course, we are free to use established facts other than the loop invariant itself to prove that the loop invariant remains true before starting each iteration.) The similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration corresponds to the base case, and also showing that the invariant holds from iteration to iteration corresponds to the inductive step.

The third property is the most important one, since we are using the loop invariant to show correctness. Typically, we use the loop invariant along with the condition which caused the loop to terminate. The termination property differs from how we usually use mathematical induction, in which we apply the inductive step infinitely and we stop the "induction" when the loop terminates.

Let's see how these properties are using for insertion sort.

Initialization: We start by showing the loop invariant holds before the first loop iteration, for j = 2. In this case the subarray Arr[1……(j – 1)] consists of just the single element Arr[1], which is in fact the original element in Arr[1]. Also, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Now, we consider the second property: which shows that each iteration maintains the loop invariant. Here, the body of the **for** loop works by moving Arr[j – 1], Arr[j – 2], Arr[j – 3], and so on by one position to the right until it finds the proper position for Arr[j] (lines 4–7), at which point it inserts the value of Arr[j] (line 8). The subarray Arr[1….. j] then consists of the elements originally in Arr[1…j], but in sorted order. By incrementing j for the next iteration of the **for** loop preserves the loop invariant. A more formal analysis of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. However, at this point, we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Lastly, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that j > Arr. *length* = n. Since each loop iteration increases j by 1, we must have j = n + 1 at that time. Substituting n + 1 for j in the wording of loop invariant, we have that the subarray Arr [1 …… n] consists of the elements originally in Arr [1…. n], but in sorted order. Observing that the subarray Arr [1…… n] is the entire array, so we conclude that the entire array is sorted. Therefore, the algorithm is correct.

STOP TO CONSIDER

RAM (Random Access Machine) model measures the rum time of an algorithm by summing up the number of steps needed to execute the algorithm on a set of data.

CHECK YOUR PROGRESS - III

14) _____ an algorithm means predicting the resources that the algorithm requires.

15) In the_____, instructions are executed one after another, without any concurrent operations.

16) The INSERTION_SORT algorithm sorts the input elements _____.

17) _____ means to set a starting value of a variable.

1.7.   ANALYSIS OF INSERTION SORT

The time taken by the INSERTION_SORT procedure depends on the input, for example sorting a thousand numbers takes longer than sorting three numbers. INSERTION_SORT can take different amounts of time for sorting two different input sequences of the same size depending on how nearly sorted they already are. In general, the time taken for an algorithm grows with the size of the input, so it is conventional to describe the running time of a program as a function of the size of its input. For that, we need to define the terms "*running time*" and "*size of input*" more carefully.

The ***input size*** of a problem depends on the problem being used. For sorting or computing discrete Fourier transforms, the most natural measure is the "number of items in the input" (for example, the array size n for sorting.) For other problems, like multiplying two integers, the best measure of input size is the "total number of bits" needed to represent the input in ordinary binary notation. For another instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph.

The ***running time*** of an algorithm on a particular input is the number of steps executed. Let us consider a viewpoint, a constant amount of time is required to execute each line of our pseudocode. Each line may take a different amount of time, but we assume that each execution of the ith line takes time c_i, where c_i is a constant. This viewpoint is keep in the

RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.

Here, we are presenting the INSERTION_SORT procedure with time "cost" of each statement and the number of times each statement is executed. For each $j = 2, 3, \ldots\ldots n$, where $n = $ Arr.*length*, we let tj denote the number of times the **while** loop test in line 5 is executed for that value of j. When a **for** or **while** loop exits in the usual way, the test is executed one time more than the loop body. As comments are not executable statements, and so they take no time.

INSERTION_SORT(Arr)   *cost*
      times
1. **for** $j = 2$ **to** Arr.*length* c_1
    n
2.   *key* = Arr[j] c_2
    n - 1
3.   // Insert Arr[j] into the sorted
        sequence A[1 $\ldots\ldots$ (j − 1)]   0
        n - 1
4.   i = j -1 c_4
    n - 1
5.   **while** i > 0 and A[i] > *key* c_5
    $\sum_{j=2}^{n}$ tj
6.   A[i + 1] = A[i] c_6
$\sum_{j=2}^{n}(tj - 1)$
7.   i = i - 1 c_7
$\sum_{j=2}^{n}(tj - 1)$
8. A[i + 1] = *key* c_8
    n − 1

The running time of the algorithm is the sum of running times for each statement executed. A statement that takes c_i steps to execute and executes n times will contribute c_i n to the total running time. To compute the running time T(n) for INSERTION_SORT on the input of n values, we will sum the products of the *cost* and *times* columns, and we will get as follows:-

$$T(n) = c_1 n + c_2(n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n}(t_j - 1) + c_8 (n - 1)$$

For inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. We can analyze the INSERTION_SORT in three cases, best case, worst case and average case.

1.7.1 Insertion_Sort in the Best Case

In INSERTION_SORT, the best case occurs when the array is already sorted. For each $j = 2, 3, \ldots\ldots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of j - 1. Thus $t_j = 1$ for $j = 2, 3, \ldots\ldots, n$, and the running time in the best case is,

$$T(n) = c_1 n + c_2(n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1).$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8) n - (c_2 + c_4 + c_5 + c_8).$$

We can express the above running time as An + B for *constants* A and B that depend on the statement costs c_i, it is thus a ***linear function*** of n.

1.7.2 Insertion_Sort in the Worst Case

INSERTION_SORT in the worst case means, if the array is in reverse sorted order —that is, in decreasing order. In this situation, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1, 2, \ldots\ldots (j - 1)]$, and so $t_j = j$ for $j = 2, 3, \ldots\ldots, n$. Noting that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \text{and} \qquad \sum_{j=2}^{n}(j - 1) = \frac{n(n-1)}{2}$$

Now the running time for the INSERTION_SORT in the worst case is,

$$T(n) = c_1 n + c_2(n - 1) + c_4 (n - 1) + c_5 \left(\frac{n(n+1)}{2} - 1\right) + c_6 \left(\frac{n(n-1)}{2}\right) + c_7 \left(\frac{n(n-1)}{2}\right) + c_8 (n - 1)$$

$= (c_5/2 + c_6/2 + c_7/2) n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8) n - (c_2 + c_4 + c_5 + c_8)$

We can express the above running time as $An^2 + Bn + C$ for *constants* A, B and C that again depend on the statement costs c_i, it is thus a *quadratic function* of n.

The worst-case running time of an algorithm gives us an upper bound on the running time for any input. It provides a guarantee that the algorithm will never take any longer. Moreover, we need not make some educated guess about the running time and hope that it never gets much worse.

For some algorithms, the worst case occurs differently. For example, searching for a particular piece of information in a database, the searching algorithm's worst case will often occur when the information is not present in the database. Also, in some applications, searches for absent information may be frequent.

1.7.3 Insertion_Sort in the Average Case

The "average case" is as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. On average case , half the elements in A[1…… (j − 1)] are less than A[j], and half the elements are greater than A[j]. In this case, therefore, we check half of the subarray A[1….. (j − 1)] , and so tj is about j / 2.

Here,

$$\sum_{j=2}^{n} j/2 = (\frac{n(n+1)}{2} - 1)/2 \qquad \text{and}$$
$$\sum_{j=2}^{n} (j-1)/2 = (\frac{n(n-1)}{2})/2$$

Now the running time for the INSERTION_SORT in the average case is,

$T(n) = c_1 n + c_2(n-1) + c_4 (n - 1) + c_5 (\frac{n(n+1)}{2} - 1)/2 + c_6 (\frac{n(n-1)}{2})/2 +$

$\qquad c_7 (\frac{n(n-1)}{2})/2 + c_8 (n - 1)$

$= (c_5/4 + c_6/4 + c_7/4) n^2 + (c_1 + c_2 + c_4 + c_5/4 - c_6/4 - c_7/4 + c_8) n -$

$\qquad (c_2 + c_4 + c_5/4 + c_8)$

Just like worst case here also we can express the above running time as $An^2 + Bn + C$ for *constants* A, B and C that again depend on the statement costs c_i, it is thus a ***quadratic function*** of n.

STOP TO CONSIDER

- **Best case:** Best case is the function, which performs the minimum number of steps on input data of n elements.

- **Average case:** Average case is the function, which performs the average number of steps on input data of n elements.

- **Worst case:** Worst case is the function, which performs the maximum number of steps on input data of n elements.

CHECK YOUR PROGRESS - IV

18) The _____ of a problem depends on the problem being used.

19) The _____ of an algorithm on a particular input is the number of steps executed.

20) An algorithm can take _____ amounts of time for sorting two different input sequences

1.8 RATE OF GROWTH

Suppose we are analyzing two algorithms and expressed their run times in terms of the size of the input. Algorithm A takes $n^2 + n + 1$ steps to solve a problem with size *n* and algorithm B takes $100\, n + 1$ steps. The following table shows the run time of these two algorithms for different problem sizes:

Input Size	Runtime of algorithm A	Runtime of Algorithm B
10	111	1001
100	10101	10001
1000	1001001	100001
10000	$>10^{10}$	1000001

For *n*=10, Algorithm B looks pretty bad as it takes almost 10 times longer than Algorithm A. But at *n* =100 they are about the same, and for larger values B is much better. The main reason is that for large values of *n*, any function that contains an n^2 term will grow faster than a function whose leading term is *n*. The **leading term** denotes the term with the highest exponent. Algorithm A does better than B for small n, since for Algorithm B, the leading term has a large coefficient, 100. But other than the coefficients, there will always be some value of *n* where *a* $n^2 > b\ n$.

Generally, we expect an algorithm with a smaller leading term to be a better algorithm for large problems, but for smaller problems, there may be a **crossover point** where another algorithm is better. The location of the crossover point depends on the details of the algorithms such as, the inputs, and the hardware and hence it is usually ignored for purposes of algorithmic analysis. If two algorithms have the same leading order term, it is hard to say which is better and again, the result depends on the details. Therefore for the analysis of algorithm, functions with the same leading term are considered equivalent, even if they have different coefficients.

A **rate of growth** or **order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example, 3n, 1000n and n + 1 belong to the same order of growth, which is written O(n) in Big-Oh notation and often called linear because every function in the set grows linearly with n. In another example, n^2, $2n^2 + n + 1$, $100n^2 + 1$ are belong to same order of growth, written as $O(n^2)$ and they are called quadratic for functions with the leading term n^2.

The following table shows some of the rate of growth that appear most commonly in algorithmic analysis.

Rate of growth	Name
O(1)	Constant
$O(\log_b n)$	logarithmic (for any b)
O(n)	Linear
$O(n \log_b n)$	"en log en"
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(c^n)$	Exponential (for any c)

For the logarithmic terms, the base of the logarithm doesn't matter and changing bases is the equivalent of multiplying by a constant, which doesn't change the rate of growth. Likewise, all exponential functions belong to the same order of growth regardless of the base of the exponent. Since exponential functions grow very quickly, therefore exponential algorithms are only useful for small problems.

1.9 SUMMING UP

- An Algorithm is a step-by-step process to solve a problem. Every computerized device uses algorithms, which cut the time required to do things manually.

- RAM (Random Access Machine) model measures the rum time of an algorithm by summing up the number of steps needed to execute the algorithm on a set of data.

- An analysis of algorithm is a technique that's used to measure the performance of the algorithms. Analysis of algorithm can be done in three case; best case, average case and worst case. Best case of an algorithm performs the minimum number of steps on input data of n elements. Average case of an algorithm performs the average number of steps on input data of n elements. Worst case of an algorithm performs the maximum number of steps on input data of n elements.

- Complexity of an algorithm measures the amount of time and/or space required by an algorithm for an input of a given size (n).

Time Complexity of an algorithm means, the amount of time required by the algorithm to execute to completion. Space complexity of an algorithm means the amount of memory space needed the algorithm.

- A *rate of growth* or *order of growth* is a set of functions whose asymptotic growth behavior is considered equivalent.

1.10 ANSWERS TO CHECK YOUR PROGRESS

1) Algorithm
2) Computational
3) Output
4) Time, space
5) Zero, more
6) Algorithms
7) Science
8) Donald Knuth
9) Worst case
10) Average
11) Memory space
12) Variable part
13) Time Complexity
14) Analyzing
15) RAM model
16) In place
17) Initialization
18) Input size
19) Running time
20) Different

1.11 POSSIBLE QUESTIONS AND ANSWERS

Short Answer type Questions:

1.   What do you mean by Algorithm?
2.   What is the need for an algorithm?
3.   Define time complexity of algorithm.
4.   Define space complexity of algorithm.
5.   Why analysis of algorithms required?
6.   What do you mean by order of an algorithm?

Long Answer type Questions:

1.   Define Algorithm with an example.
2.   Write a note on the advantages and disadvantages of Algorithm.
3.   What are the various cases to analyze an algorithm?
4.   Find out the order of growth for following:-
   a) $n^3 + n^2$
   b) $1000000\ n^3 + n^2$
   c) $n^3 + 1000000\ n^2$
   d) $20n^2 + n$
   e) $(n^2 + n) \cdot (n+1)$
5.   Write down the algorithm for bubble sort and analyze in best case, average case and worst case.
6.   Write down the algorithm for selection sort and analyze in best case, average case and worst case.
7.   Differentiate best case and worst case time complexity.

1.12 REFERENCES AND SUGGESTED READING

1.   Thomas H. Cormen, Charlese E . Leiserson, Ronald L. Rivest, Clifford Stein. "INTRODUCTION TO ALGORITHMS", PHI publication.
2.   Aditya Bhargava "Grokking Algorithm: An illustrated guide for programmers and other curious people".

UNIT 2: ASYMPTOTIC NOTATIONS I

Unit Structure:

2.1 INTRODUCTION

An algorithm is a collection of steps of different operations to solve a specific problem. An algorithm is an effective method to solve a problem within a finite amount of time and space. It is the best way to represent the solution of a specific problem in a very simple and well-organized way. An algorithm for a specific problem can be implemented in any programming language. Algorithm analysis is an important part of computational complexity theory and it is can be used to find the best possible. The algorithms are designed to work with inputs of arbitrary length and it is analyzed based on the amount of time and space requires to execute them. Different growth functions and notations are used to present the functional value of an algorithm such as O, o, θ, ω, Ω. The notation is used for the different case-based analyses of an algorithm using mathematical induction and other methods.

2.2 UNIT OBJECTIVES

After going through this unit, you will be able to know

i)   About different asymptotic notations.

ii) About the common mathematical function and complexity analysis

iii) About examples of complexity analysis

2.3 ASYMPTOTIC NOTATIONS (O, O, Θ, Ω, Ω)

The term algorithm complexity defines the amount of time and space required to execute the steps of an algorithm. It evaluates the order of count of operations executed by an algorithm as a function of input data size. To assess the complexity, different notations are used which are known as **Asymptotic notation**. Let's O (f) notation represents the complexity of an algorithm, the Asymptotic notation is "Big O" notation and f corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation O(f) determines in which order the resources such as CPU. The complexity of an algorithm may find in any form such as constant, logarithmic, linear, n*log (n), quadratic, cubic, exponential, etc. It is nothing but the order of constant, logarithmic, linear, and so on, the number of steps encountered for the completion of a particular algorithm. We are calling it as the term running time of the algorithm.

Generally, the running time of an algorithm falls under three different cases.

- Best Case –the Minimum time required for a program to execute its line of codes.

- Average Case − the average time required for a program to execute its line of codes.

- Worst case − the maximum time required for a program to execute its line of codes.

The following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation (Big-Oh)

- Ω Notation (Omega)

- θ Notation (Theta)

Big Oh (O) Notation

The asymptotic notation O(n) is used to prompt the upper bound of the running time of an algorithm. Generally, it is used to measure the worst-case time complexity of an algorithm but not all the time because any asymptotic notation can be used for worst-case analysis. Case-based algorithm analysis is not similar to the asymptotic notation.
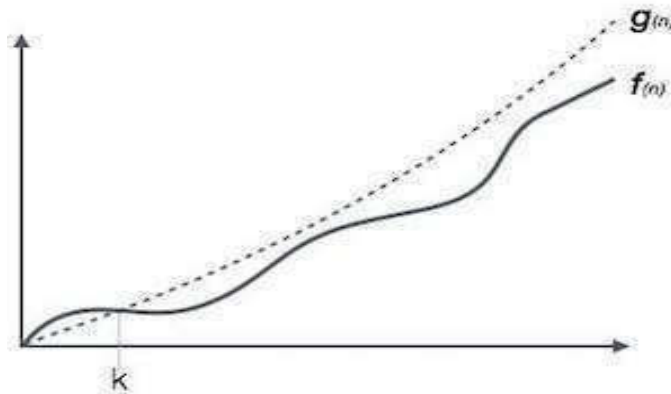

Fig 2.1. Graphical view of Big-Oh (O) Notation

Mathematically, it is explained as for a function f(n), the O(f(n)) = {g(n): there exists k > 0 and n_0 such that f(n) ≤ k.g(n) for all n > n_0.}

Omega Notation (Ω)

The asymptotic notation omega Ω(n) is used to express the lower bound of an algorithm's running time. Though it is used for the best case time complexity of an algorithm other asymptotic notations are also used for best-case analysis. As mentioned above, any asymptotic notation can be used for any case-based analysis. Mathematically, it is defined as follows. For example, for a function f(n), the omega f(n)) ≥ Ω{g(n): there exists c > 0 and n_0 such that g(n) ≤ k.f(n) for all n > n_0}
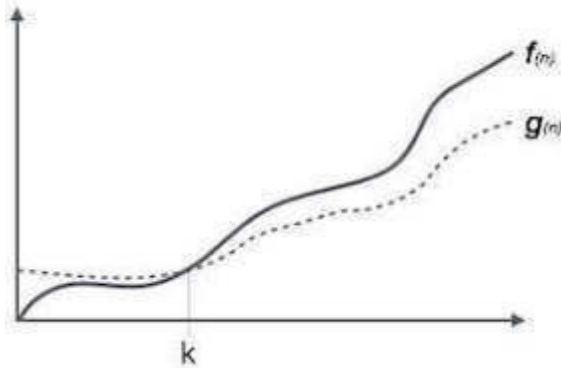
Fig 2.2. Graphical view of Omega (Ω) Notation

Theta Notation (θ)

The notation theta θ(n) is another way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows. θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > n_0.
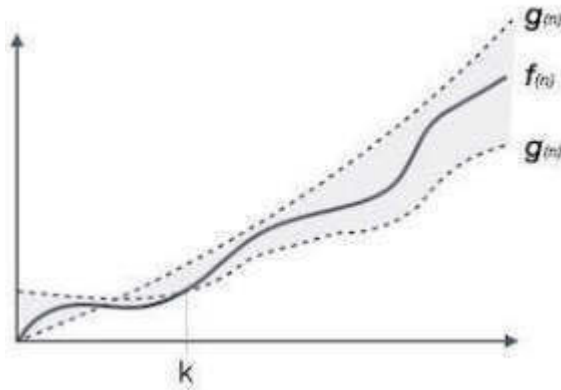

Fig 2.3. Graphical view of thetaθ(n) Notation

Along with the above three notations, another two notations such as o and **ω** are also used in complexity analysis. A O is used as a tight upper bound on the growth of an algorithm, whereas the little o notation is also used for the upper bound but it is not tight.Let f(n) and g(n) be functions and f(n) = o(g(n)) if for any real constant c > 0, there exists an integer constant n_0 ≥ 1 such that 0 ≤ f(n) < k*g(n).It means that o() means loose upper-bound of f(n). Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.
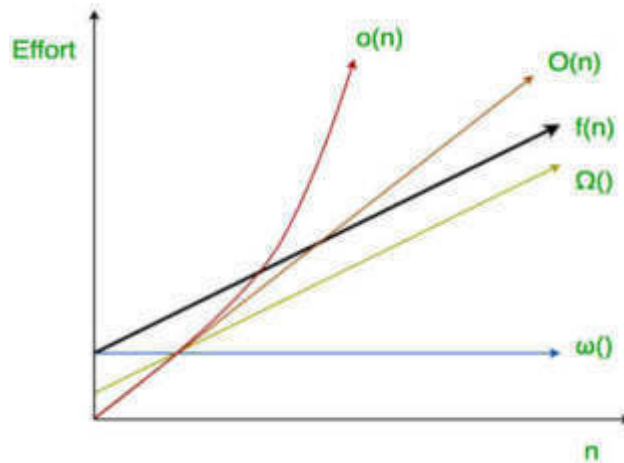
Fig 2.4. Graphical view of Small o and ωNotation

The relationship between Big Omega (Ω) and Little Omega (ω) is similar to that of Big-. The ω is looking at the lower bounds that are not asymptotically tight. Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) for any real constant c > 0, there exists an integer constant $n_0 \geq 1$ such that f(n) > c * g(n) $\geq$ 0 for every integer n $\geq$ n0. As f(n) has a higher growth rate than g(n) so main difference between Big Omega (Ω) and little omega (ω) lies in their definitions. In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=c*g(n)<f(n). We use ω notation to denote a lower bound that is not asymptotically tight. And f (n) $\in$ ω (g (n)) if and only if g(n) $\in$ o((f(n)).

CHECK YOUR PROGRESS - I

1. What is an algorithm?
2. What do you mean by asymptotic notation?
3. True or False
    i)   3n3 + 6n2 + 6000 = Θ(n3)
    ii)   f(n) = 2n²+5 is O(n²)
    iii)   If f(n) = 2n²+5 is O(n²) then 7*f(n) = 7(2n²+5) = 14n²+35 is also O(n²).

2.4 COMMON MATHEMATICAL FUNCTIONS FOR COMPLEXITY ANALYSIS

The common mathematical functions used for complexity analysis are presented below.

- constant: $\Theta(1)$
- logarithmic: $\Theta(\log N)$
- linear: $\Theta(N)$
- polynomial: $\Theta(N^\wedge 2)$
- exponential: $\Theta(2^\wedge N)$
- factorial: $\Theta(N!)$

2.5 EXAMPLES OF ASYMPTOTIC NOTATION

Let's understand the asymptotic notations with the following examples.Express the following functional value of an algorithm in terms of O, θ, Ω

i) $F(n) = 2n + 5$

ii) $F(n) = 2n^2 + 5$

Solution:

i) Here, the functional value is $F(n) = 2n + 5$

For O, the definition is the $f(n) = O\{g(n)\}$, where $f(n) \leq c.g(n)$ for all $n > n_0$.

So,

$f(n) = 2n + 5 < 3n$ ---------- (1)

Here, the $3n > 2n+5$ for some values of n. To check its validity, put n = 1, 2, 3, 4,5,6, ------

for n=1, the LHS is 2*1+5 = 7. Again RHS 3*1 = 3. For this 3<7, so it is invalidfor the O definition

for n=2, the LHS is 2*2+5 = 9. Again RHS 3*2 = 6.For this 6<9, so it is invalidfor the O definition

for n=3, the LHS is 2*3+5 = 11. Again RHS 3*3 = 9. For this 9<11, so it is invalidfor the O definition

for n=4, the LHS is 2*4+5 = 13. Again RHS 3*4 = 12. For this 12<13, so it is invalidfor the O definition

for n=5, the LHS is 2*5+5 = 15. Again RHS 3*5 = 5. For this 15 = 15, so it is valid for the O definition.

for n=6, the LHS is 2*6+5 = 17. Again RHS 3*6 = 18. For this 18 >15, so it is valid for the O definition.

If you put any value of n starting from 5 then the

$f(n) \leq c.g(n)$ where $f(n) = 2n+5$, c = 3 and n = {5, 6, …}

so, the equation is $2n+5 \leq 3n$. So, the O notation is f(n) = O (n), where g(n) = n, c= 3, and n = {5, 6, …}

Now For Ω, the definition is the$f(n) \geq \Omega$ {g(n)for all n > n0.

So,

$f(n) = 2n+ 5 > 2n$ ---------- (2)

Here, the 2n < 2n+5. To check its validity, put n = 1,2,3, 4,5,6, ------

For n=1, the LHS is 2*1+5 = 7. Again RHS 2*1 = 2. For this 2<7, so it is valid for the **Ω** definition

For n=2, the LHS is 2*2+5 = 9. Again RHS 2*2 = 4. For this 4<9, so it is valid for the **Ω** definition

If you put any value of n starting from 1then the

$f(n) \geq \Omega$ {g(n)where f(n) = 2n+5, c = 2 and n = {1,2,3, ….}

so, the equation is 2n+5> 2n. So, the **Ω** notation is f(n) = **Ω** (n), where g(n) = n, c= 2, and n = {1,2,3, ….}

Now Forθ,

$$2n < f(n) = 2n+ 5 < 3n$$

So, the n= 5 and c1= 2, and c2 = 3, the equation is valid and it cn expressed as f (n) = θ (n).

    ii)   Here, the functional value is $F(n) = 2n^2 + 5$

For O, the definition is the f(n) = O{g(n)}, where $f(n) \leq c.g(n)$ for all n > n_0.

So,

$f(n) = 2n^2 + 5 <= 3n^2$ ---------- (3)

Here, the $3n^2 > 2n^2 + 5$ for some values of n. To check its validity, put n = 1,2,3, 4,5,6, ------

for n=1, the LHS is 2*1+5 = 7. Again RHS 3*1 = 3. For this 3<7, so it is invalid for the O definition

for n=2, the LHS is $2*2^2+5 = 13$. Again RHS $3*2^2 = 12$. For this 12<13, so it is invalid for the O definition

for n=3, the LHS is $2*3^2+5 = 23$. Again RHS $3*3^2 = 27$. For this 27>23, so it is valid for the O definition

If you put any value of n starting from 3 then the $f(n) \leq c.g(n)$ where $f(n) = 2n^2 + 5$, c = 3 and n = {3, 4, …}so, the equation is $2n^2 + 5 \leq 3n^2$. So, the O notation is f(n) = O (n), where g(n) = n, c= 3, and n = {3, 4, …}

Now For Ω, the definition is the $f(n) \geq \Omega$ {g(n)for all n > n_0.

So,

$f(n) = 2n^2 + 5 >= 2n^2$ ---------- (4)

Here, the $2n^2 + 5 >= 2n^2$. To check its validity, put n = 1,2,3, 4,5,6, ------

For n=1, the LHS is 2*1+5 = 7. Again RHS 2*1 = 2. For this 7>2 so it is valid for the **Ω** definition

If you put any value of n starting from 1 then the

$f(n) \geq \Omega$ {g(n)where $f(n) = 2n^2 + 5$, c = 2 and n = {1,2,3, ….}

so, the equation is $2n^2 + 5 > 2n^2$. So, the **Ω** notation is f(n) = **Ω** (n), where g(n) = n, c= 2, and n = {1,2,3, ….}

Now Forθ,

$$2n^2 < f(n) = 2n^2 + 5 <= 3n^2$$

So, the n= 3 and c1= 2, and c2 = 3, the equation is valid and it can expressed as f (n) = θ (n).

CHECK YOUR PROGRESS - II

4. Arrange the following complexity function in ascending order.
   $O(\log n), O(n), O(2^n)$
5. True or False
   iv)   1. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$
   v)   If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $h(n) = \Omega(f(n))$
   vi) $n/100 = \Omega(n)$

2.6 SUMMING UP

i)   An algorithm is a collection of steps of different operations to solve a specific problem. An algorithm is an effective method to solve a problem within a finite amount of time and space.

ii)   The term algorithm complexity defines the amount of time and space required to execute the steps of an algorithm. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

iii)   Generally, the running time of an algorithm falls under three different cases.

   a. Best Case –the Minimum time required for a program to execute its line of codes.

   b. Average Case − the Average time required for a program to execute its line of codes.

   c. Worst case − the Maximum time required for a program to execute its line of codes.

iv)   The following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

   a. O Notation (Big-Oh)

   b. Ω Notation (Omega)

   c. θ Notation (Theta)

v)   Mathematically, Big Oh O is explained as for a function f(n), the $O(f(n)) = \{g(n)$: there exists $c > 0$ and $n0$ such that $f(n) \leq c.g(n)$ for all $n > n_0.\}$

vi)   Mathematically, omega is explained as for a function f(n), the omega $f(n)) \geq \Omega\{g(n)$: there exists $c > 0$ and $n0$ such that $g(n) \leq c.f(n)$ for all $n > n_0.\}$

vii) $\theta(f(n)) = \{\ g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0.$

viii) Let f(n) and g(n) be functions and $f(n) = o(g(n))$ if for any real constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $0 \leq f(n) < c*g(n).$

ix)   Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is $\omega(g(n))$ for any real constant $c > 0$, there exists an integer constant $n0 \geq 1$ such that $f(n) > c * g(n) \geq 0$ for every integer $n \geq n_0.$

2.7 ANSWER TO CHECK YOUR PROGRESS

1) An algorithm is a collection of steps of different operations to solve a specific problem. An algorithm is an effective method to solve a problem within a finite amount of time and space.

2) The term algorithm complexity defines the amount of time and space required to execute the steps of an algorithm. To assess the complexity, different notation are used which are known as Asymptotic notation

3) i)True ii) True iii) True

4) $O(log) < O(n) < O(2^n)$

5) i) True ii) True iii) True

2.8 POSSIBLE QUESTIONS

Short Answer type Questions:

i)   What do you mean by an algorithm?

ii)   What are the properties of a good algorithm?

iii)   What do you mean by asymptotic notation?

iv)   What are the different asymptotic notations are used?

v)   What are the different case-based analyses present?

vi)   What is the definition of Big-Oh?

vii)   What is the definition of omega?

viii)   What is the definition of theta?

Long Answer type Questions:

i)   Express the following notation in Big Oh, Small O, omega, and theta notation

   a. $F(n) = 3n + 2n$

   b. $F(n) = 2n^2 + 5$

ii)   Express the following notation in Big Oh, Small O, omega, and theta notation

   a. $F(n) = 2n / 5$

   b. $F(n) = \log n$

2.9 FURTHER READINGS

- Introduction to the algorithm- MIT press- Thomas H Coleman

UNIT 3: ASYMPTOTIC NOTATIONS II

1.1   INTRODUCTION

The efficiency and performance in a meaningful way is determined by Asymptotic Notation. Often, we get complex polynomial at the time of calculating the complexity of an algorithm. We use asymptotic notation to simplify this complex polynomial.

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N={ 0,1,2,…..}. Such notations are convenient for describing the worst case running time function T(n), which usually is defined only on integer input sizes. We sometimes find it convenient, however, to abuse asymptotic notation in a variety of ways. For example: we might extend the notation to the domain of real numbers or alternatively, restrict it to a subset of the natural numbers [1].

Primarily we use asymptotic notation to describe the running times of algorithm. When we use asymptotic notation to apply to the running time of an algorithm, we need to understand which running time we mean. Sometime we are interested in the worst- case running time. Often, we wish to characterize the running time no matter what the input.

1.2 RELATIONAL PROPERTIES OF ASYMPTOTIC NOTATIONS

In previous unit we discussed about Asymptotic Notations and its uses in calculating time complexity. Here, we will discuss various relational properties. Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that f(n) and g(n) are asymptotically positive [1].

1.2.1 General Properties

If f(n) is O(g(n)) then a*f(n) is also O(g(n)) ; where a is a constant

Example:

f(n) = 2n²+5 is O(n²)

then 2*f(n) = 2(2n²+5)= 4n²+10, is also O(n²)

Similarly, this property satisfies both Θ and Ω notation. We can say If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)); where a is a constant. If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)); where a is a constant.

1.2.2 Reflexive Properties

If f(n) is given then f(n) is O(f(n)).
Example: f(n) = n² ; O(n²) i.e O(f(n))

Similarly, this property satisfies both Θ and Ω notation. We can say
If f(n) is given then f(n) is Θ(f(n)).
If f(n) is given then f(n) is Ω (f(n)).

1.2.3 Transitive Properties

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .
Example: if f(n) = n , g(n) = n² and h(n)=n³
n is O(n²) and n² is O(n³) then n is O(n³)

Similarly this property satisfies for both Θ and Ω notation. We can say
If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

If f(n) is o(g(n)) and g(n) is o(h(n)) then f(n) = o(h(n))

If f(n) is ω(g(n)) and g(n) is ω(h(n)) then f(n) = ω(h(n))

1.2.4 Symmetric Properties

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .
Example: f(n) = n² and g(n) = n² then f(n) = Θ(n²) and g(n) = Θ(n²)

Similarly this property satisfies for both O and Ω notation. We can say

If f(n) is O (g(n)) then g(n) is O (f(n))

If f(n) is Ω (g(n)) then g(n) is Ω (f(n))

1.2.5 Transpose Symmetric Properties

If f(n) is O(g(n)) then g(n) is Ω (f(n)).
Example: f(n) = n , g(n) = n² then n is O(n²) and n² is Ω (n)
Since these properties hold for asymptotic notations, analogies can be drawn between functions f(n) and g(n) and two real numbers a and b.

g(n) = O(f(n)) is similar to a ≤ b

g(n) = Ω(f(n)) is similar to a ≥ b

g(n) = Θ(f(n)) is similar to a = b

g(n) = o(f(n)) is similar to a < b

g(n) = ω(f(n)) is similar to a > b

We can say that *f(n)* is asymptotically smaller than *g(n)* if *f(n)* = o(*g(n)*), and *f(n)* is asymptotically larger than *g(n)* if *f(n)* = ω(*g(n)*)

1.2.6 Some More Properties

1. If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))
2. If f(n) = O(g(n)) and d(n)=O(e(n))
then f(n) + d(n) = O(max(g(n), e(n)))

Example: f(n) = n   i.e. O(n)

d(n) = n²   i.e. O(n²)

then f(n) + d(n) = n + n²   i.e. O(n²)

3.If f(n)=O(g(n)) and d(n)=O(e(n))

then f(n) * d(n) = O(g(n) * e(n))

Example: f(n) = n   i.e. O(n)

d(n) = n²   i.e. O(n²)

then f(n) * d(n) = n * n² = n³   i.e. O(n³)

1.3   ASYMPTOTIC BEHAVIORS OF POLYNOMIALS

Given a nonnegative integer d, a *polynomial in n of degree d* is a function p (n) of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

Where the constants $a_0, a_1, \ldots \ldots \ldots, a_d$ are the coefficients of the polynomial and $a_d \neq 0$. A polynomial is asymptotically positive if and only if $a_d > 0$. For an asymptoticallypositive polynomial p(n) of degree d, we have p (n)= $\Theta(n^d)$. For any real constant a≥0, the function n^a is monotonically increasing, and for any real constant a≤0, the function n^a is monotonically decreasing. We say that a function f(n) is polynomially bounded if f(n)= O (n^k) for some constant k.

Let, $p(n) = \sum_{i=0}^{d} a_i n^i$

Where if $a_d > 0, be\ a\ degree\ d$ polynomial in n, and let k be a constant. By using definitions of the asymptotic notations, we can prove the following properties:

a. If k=d, then $p(n)=\Theta(n^k)$

b. If k ≥d, then $p(n)=O(n^k)$

c. If k ≤d, then $p(n)=\Omega(n^k)$

a. Theta or Asymptotic Bound:

Analytical Approach:

The largest term in the polynomial is $a_d n^d$, so the polynomial cannot grow neither slower nor faster than n^d. Hence, $p(n)=\Theta(n^k)$

Mathematical Approach:

The polynomial can be written as:

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

$$= a_d n^d + \sum_{i=0}^{d-1} a_i n^i$$

$$= a_d n^d + n^d \sum_{i=0}^{d-1} a_i n^{i-d}$$

$$= a_d n^d + n^d Q_n$$

$$= n^d (a_d + Q_{n)}$$

Where, $Q_n = \sum_{i=0}^{d-1} a_i n^{i-d}$

Now note that Q_n is the sum of powers of n multiplied by some constants, and the powers of n are all less than or equal to -1. This follows from the fact that $(i-d) \leq -1$. Now for sufficiently large n, Q_n approaches zero. And that in turn means, before it reaches zero, we can find a positive integer n_0, such that: $|Qn| \leq 0.5a_d$ for all $n \geq n_0$.

Now,

$$-0.5\, a_d \leq Q_n \leq 0.5\, a_d$$

$$a_d - 0.5\, a_d \leq a_d + Q_n \leq a_d + 0.5a_d$$

$$n^d(a_d - 0.5\, a_d) \leq n^d(a_d + Q_n) \leq n^d(a_d + 0.5\, a_d)$$

$$0.5a_d.n^d \leq n^d(a_d + Q_n) \leq 1.5a_d.n^d$$

$$0.5\, a_d.n^d \leq p(n) \leq 1.5a_d.n^d$$

So, if we pick $c_1 = 0.5\, a_d$ and $c_2 = 1.5\, a_d$, we have $c_1 n^d \leq p(n) \leq c_2 n^d$

In other words, $p(n) = \theta(n^d)$

And when $k = d$, it means $p(n) = \theta(n^k)$

b. Asymptotic Upper Bound:

Analytically as $k \geq d$, asymptotically n^k grows faster or at same rate than n^d for sufficiently large n. Hence, $p(n) = O\ (n^k)$

Mathematically, from c we can write:

$$0 \leq p(n) \leq 1.5a_d.n^d \leq 1.5\, a_d.n^k$$

So, if we pick $c_1 = 1.5\, a_d$, we have $0\ \leq p(n) \leq c_1 n^k$

In other words, $p\ (n) = O\ (n^k)$

c. Asymptotic Lower Bound:

Analytically, as $k \leq d$, asymptotically n^k grows slower or at same rate than n^d for sufficiently large n. **Hence, $p(n) = \Omega(n^k)$**

Mathematically, from c we can write:

$$0 \leq 0.5a_d.n^k \leq 0.5a_d.n^d \leq p(n)$$

So, if we pick $c_1 = 0.5\ a_d$, we have $0 \leq c_1 n^k \leq p(n)$

In other words, $p(n) = \mathbf{\Omega}(n^k)$

1.4   RELATIVE ASYMPTOTIC GROWTH

There may be many algorithms for solving any problem and obviously we would like to use the most efficient one. Analysis of algorithm is required to compare these algorithms and recognize the best one. Algorithms are generally analyzed on their time and space requirements.

One way of comparing algorithms is to compare the exact running time of all algorithms. But the running time is dependent on the language and machine used for implementing the algorithm. Even if the machine and language are kept same, calculation of exact time would be very difficult as it would require the count of instructions executed by the hardware and the time taken to execute each instruction. So the time efficiency is not measured in time units like seconds or microseconds [2].

The running time generally depends on the size of input, for example any soring algorithm will take less time to sort 10 elements and more time for 100000 elements. So the time efficiency is generally expressed in terms of size of input. If the size of input is n, then f(n) which is a function of n denotes the time complexity. Thus to compare any two algorithms we will find out this function for both algorithms and then compare the rate of growth of these two functions. It is important to compare the rates of growth because an algorithm may seem better for small input but as the input becomes large it may take more time than others [2].

The function f(n) may be found out by identifying some key operations in the algorithm which account for most of the running time. Other operations are not counted as they take very little time as compared to these key operations and not executed more often than the key operations. For example, in searching we may count the number of comparisons and in sorting we may count the swaps in addition to

comparisons. We are interested only in the growth rate of functions so the exact computation of f(n) is not necessary [2].

Let us take an example where time complexity is given by the following function:

$$f(n) = 5n^2 + 6n + 12$$

If n=10

% of running time due to the term $5n^2$: $\left(\frac{500}{500+60+12}\right) * 100 = 87.41\%$

% of running time due to the term $6n$: $\left(\frac{60}{500+60+12}\right) * 100 = 10.49\%$

% of running time due to the term 12: $\left(\frac{12}{500+60+12}\right) * 100 = 2.09\%$

The following table shows the growth rate of all the terms of function

$$f(n) = 5n^2 + 6n + 12$$

n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%
10000	99.99%	0.01%	2.4E-06%

We can see that n grows, the dominant term n^2 accounts for most of the running time and we can ignore the smaller terms. Calculating exact function f(n) for the time complexity may be difficult. So the terms which do not significantly change the magnitude of function can be dropped from the function. In this way we can get an approximation of the time efficiency and we are satisfied with this approximation because this is very close to the exact value when n becomes large. This approximate measure of complexity is known as asymptotic complexity.

There are some standard functions whose growth rates are known, we find out the complexity of our algorithm and compare it with these known functions whose growth rates are known. The growth rates of some known functions are shown in the table:

n	G(n)					
	$\log_2 n$	n	$\dfrac{n}{\log_2 n}$	n^2	n^3	2^n
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	4.29E+09
64	6	64	384	4096	262144	1.84E+19

From the table we see that some functions grow faster than others. The growth rate of $g(n) = \log_2 n$ is least and the function $g(n) = 2^n$ grows very fast. The function $g(n) = n$ grows faster than $\log_2 n$ but slower than $n\log_2 n$ or $n^2, n^3, 2^n$. To compare the growth rate of function

$f(n)$ with these standard functions, we can use big O notation.

The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. Once the input size n becomes large enough, merge sort, with its ▓(n lg n) worst-case running time, beats insertion sort, whose worst-case running time is ▓(n^2). Although we can sometimes determine the exact running time of an algorithm, the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

The **asymptotic** efficiency of algorithms is required when we look at input sizes large enough to make only the order of growth of the running time relevant. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Here we discussed several standard methods for simplifying the asymptotic analysis of algorithms.

Asymptotic growth means the rate at which the function grows. Growth rate means the complexity of function or the amount of resource it takes up to compute (i.e. time + memory). Classification of growth:

    a) Growing with the same rate

    b) Growing with the slower rate

    c) Growing with a faster rate

There are mainly three asymptotic notation are used to analyze function growth and to represent time complexity of an algorithm. The functions need not necessarily be about algorithms, and indeed asymptotic analysis is used for many other applications.

Asymptotic analysis of algorithms requires:

1. Identifying **what aspect of an algorithm we care about**, such as:

    o   runtime

    o   use of space

    o   possibly other attributes such as communication bandwidth

2. Identifying **a function that characterizes that aspect**

3. Identifying **the asymptotic class of functions that this function belongs to**, where classes are defined in terms of bounds on growth rate.

The different asymptotic bounds we use are analogous to equality and inequality relations:

- $O \approx \leq$

- $\Omega \approx \geq$

- $\Theta \approx =$

- $o \approx <$

- $\omega \approx >$

In practice, most of our analyses will be concerned with run time. Analyses may examine:

- Worst case

- Best case
- Average case (according to some probability distribution across all possible inputs)

1.4.1 Order of Growth and Big-O Notation

In estimating the running time of **insertion sort** (or any other program) we don't know what the constants c or k are. We know that it is a constant of moderate size, but other than that it is not important; we have enough evidence from the asymptotic analysis to know that a **merge sort** is faster than the quadratic **insertion sort**, even though the constants may differ somewhat [3].

We may not even be able to measure the constant c directly. For example, we may know that a given expression of the language, such as if, takes a constant number of machine instructions, but we may not know exactly how many numbers. For these reasons, we usually ignore constant factors in comparing asymptotic running times.

For hiding the constant factor, convenient notation are used. We write $O(n)$ instead of "cn for some constant c." Thus an algorithm is said to be $O(n)$ or *linear time* if there is a fixed constant c such that for all sufficiently large n, the algorithm takes time at most cn on inputs of size n. An algorithm is said to be $O(n^2)$ or *quadratic time* if there is a fixed constant c such that for all sufficiently large n, the algorithm takes time at most cn^2 on inputs of size n. $O(1)$ means *constant time*.

Polynomial time means $n^{O(1)}$, or n^c for some constant c. Thus any constant, linear, quadratic, or cubic ($O(n^3)$) time algorithm is a polynomial-time algorithm. This is called *big-O notation*. It concisely captures the important differences in the asymptotic growth rates of functions.

One important advantage of big-O notation is that it makes algorithms much easier to analyze, since we can conveniently ignore low-order terms. For example, an algorithm that runs in time

$$10n^3 + 24n^2 + 3n \log n + 144$$

is still a cubic algorithm, since

$10n^3 + 24n^2 + 3n \log n + 144$
$<= 10n^3 + 24n^3 + 3n^3 + 144n^3$
$<= (10 + 24 + 3 + 144)n^3$
$= O(n^3)$.

Since we are ignoring constant factors, any two linear algorithms will be considered equally good by this measure. There may even be some situations in which the constant is so huge in a linear algorithm that even an exponential algorithm with a small constant may be preferable in practice. This is a valid criticism of asymptotic analysis and big-O notation. However, as a rule of thumb it has served us well. Just be aware that it is *only* a rule of thumb--the asymptotically optimal algorithm is not necessarily the best one.

Some common orders of growth seen often in complexity analysis are:

$O(1)$	constant
$O(\log_2 n)$	logarithmic
$O(n)$	linear
$O(n \log_2 n)$	"n $\log_2$ n"
$O(n^2)$	quadratic
$O(n^3)$	cubic

1.4.2 Comparing Orders of Growth

Big O notation:

When we have only an ***asymptotic upper bound***, we use *O*-notation.

Let *f* and *g* be functions from positive integers to positive integers. We say
f is $O(g(n))$ if *g* is an upper bound on *f*: there exists a fixed constant *c* and a fixed n_0 such that for all $n \geq n_0$, $f(n) \leq cg(n)$. i.e.
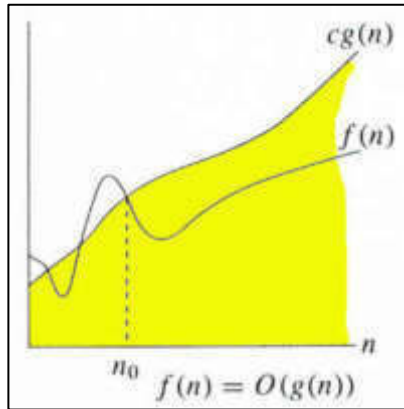
$O(g(n)) = \{f(n) : \exists$ positive constants *c* and n_0, such that $\forall n \geq n_0$,

we have $0 \leq f(n) \leq cg(n)$ }

Equivalently, *f* is $O(g(n))$ if the function $f(n)/g(n)$ is bounded above by some constant.

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of *g(n)*.

$$f(n) = O(g(n))$$

Some examples:

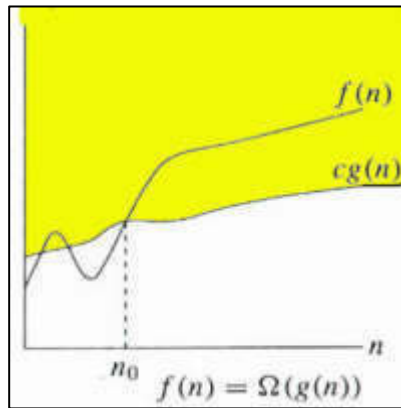| These are all O(n^2):	These are not:
• n^2 • $n^2 + 1000n$ • $1000n^2 + 1000n$ • $n^{1.99999}$ • n	• n^3 • $n^{2.00001}$ • $n^2 \lg n$

Omega Ω notation:

Just as *O*-notation provides an asymptotic *upper* bound on a function, ☒ -notation provides an **asymptotic lower bound.**

We say that *f* is Ω(*g(n)*), if *g* is a *lower* bound on *f* for large *n*. Formally, f is Ω(g) if there is a fixed constant *c* and a fixed n_0 such that for all $n > n_0, cg(n) \le f(n)$

i.e. **Ω(*g(n)*) = {*f(n)* : ∃ positive constants *c* and n_0, such that ∀*n* ≥ n_0,**

we have $0 \le cg(n) \le f(n)$}

$$f(n) = \Omega(g(n))$$

For example, any polynomial whose highest exponent is n^k is $\Omega(n^k)$. If $f(n)$ is $\Omega(g(n))$ then $g(n)$ is $O(f(n))$. If $f(n)$ is $o(g(n))$ then $f(n)$ is *not* $\Omega(g(n))$.

***Intuitively*: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.**

Some examples:

| These are all $\Omega(n^2)$:	These are not:
• n^2 • $n^2 + 1000n$ *(It's also $O(n^2)$!)* • $1000n^2 + 1000n$ • $1000n^2 - 1000n$ • n^3 • $n^{2.00001}$	• $n^{1.99999}$ • n • $\lg n$

Theta Θ notation:

Just like others, Θ notation provides an ***asymptotically tight bound.***

We say that f is $\Theta(g(n))$ if g is an accurate characterization of f for large n: it can be scaled so it is both an upper and a lower bound of f. That is, f is both $O(g(n))$ and $\Omega(g(n))$. Expanding out the definitions of Ω and O, f is $\Theta(g(n))$ if there are fixed constants c_1 and c_2 and a fixed n_0 such that for all $n > n_0, c_1 g(n) \leq f(n) \leq c_2\, g(n)$

i.e. $\Theta(g(n)) = \{f(n) : \exists$ **positive constants** c_1, c_2, **and** n_0, **such that** $\forall n \geq n_0$, **we have** $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

$$f(n) = \Theta(g(n))$$
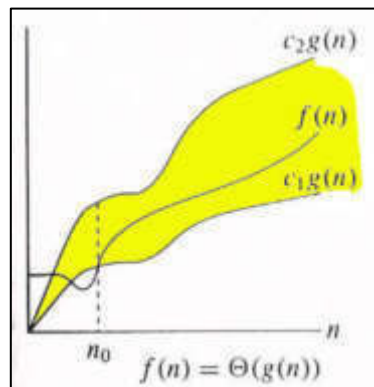
Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

For example, any polynomial whose highest exponent is n^k is $\Theta(n^k)$. If f is $\Theta(g)$, then it is $O(g)$ but not $o(g)$. Sometimes people use $O(g(n))$ a bit informally to mean the stronger property $\Theta(g(n))$; however, the two are different.

Here are some examples:

- $n + log\ n$ is $O(n)$ and $Q(n)$, because for all $n > 1$, $n < n + log\ n < 2n$.

- n^{1000} is $o(2^n)$, because $n^{1000}/2^n$ tends to 0 as n tends to infinity.

- For any fixed but arbitrarily small real number c, $n\ log\ n$ is $o(n^{1+c})$, since $n\ log\ n\ /\ n^{1+c}$ tends to 0. To see this, take the logarithm

$log(n\ log\ n\ /\ n^{1+c})$
$= log(n\ log\ n) - log(n^{1+c})$
$= log\ n + log\ log\ n - (1+c)log\ n$
$= log\ log\ n - c\ log\ n$

and observe that it tends to negative infinity.

The meaning of an expression like $O(n^2)$ is really a set of functions: all the functions that are $O(n^2)$. When we say that $f(n)$ is $O(n^2)$, we mean that $f(n)$ is a member of this set. It is also common to write this as $f(n) = O(g(n))$ although it is not really an equality.

We now introduce some convenient rules for manipulating expressions involving order notation. These rules, which we state without proof, are useful for working with orders of growth. They are really statements about sets of functions. For example, we can read #2 as saying that the product of any two functions in $O(f(n))$ and $O(g(n))$ is in $O(f(n)g(n))$.

1. $cn^m = O(n^k)$ for any constant c and any $m \leq k$.

2. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$.

3. $O(f(n))O(g(n)) = O(f(n)g(n))$.

4. $O(cf(n)) = O(f(n))$ for any constant c.

5. c is $O(1)$ for any constant c.

6. $log_b n = O(log\ n)$ for any base b.

All of these rules (except #1) also hold for Θ as well.

Some examples:

These are all $\Theta(n^2)$:	These are not
• n^2 • $n^2 + 1000n$ • $1000n^2 + 1000n + 32,700$ • $1000n^2 - 1000n - 1,048,315$	• n^3 • $n^{2.00001}$ • $n^{1.99999}$ • $n\ \lg\ n$

Small o notation:

We say f is $o(g(n))$ if for all arbitrarily small real $c > 0$, for all but perhaps finitely many n,

$f(n) \leq cg(n)$.

i.e.

$o(g(n)) = \{f(n): \forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0$, we have $0 \leq f(n) < cg(n)\}$.

$f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = 0$$

$g(n)$ is an **upper bound** for $f(n)$ that is not asymptotically tight

Equivalently, *f* is *o(g)* if the function $f(n)/g(n)$ tends to 0 as *n* tends to infinity. That is, f is small compared to g. If *f* is *o(g)* then *f* is also *O(g)*

Small ω-notation:

For a given function g(n)

$\omega(g(n))$ = {$f(n)$: $\forall c > 0$, $\exists n_0 > 0$ such that $\forall n \geq n_0$, we have$0 \leq cg(n) < f(n)$}.

f(n) becomes arbitrarily large relative to *g(n)*as *n* approaches infinity:

$$\lim_{n \to \infty} [f(n) / g(n)] = \infty.$$

g(n) is a***lower bound*** for *f(n)*that is not asymptotically tight.

Example of Relative Asymptotic Growth:

Indicate, for each pair of expressions (A,B) in the table below, whether A is O, o, Ω, ω or θ of B. Assume that $k \geq 1, \in > 0$ and $c > 1$ are constants.Now following table contain possible "yes" or "no" in the respective boxes:

A	B	O	o	Ω	ω	Θ
$\log_k n$	$n^{\in}$	Yes	Yes	No	No	No
n^k	c^n	Yes	Yes	No	No	No
$\sqrt{n}$	$n^{\sin n}$	No	No	No	No	No
2^n	$2^{n/2}$	No	No	Yes	Yes	No
$n^{\log c}$	$c^{\log n}$	Yes	No	Yes	No	Yes
$\log n!$	$\log n^n$	Yes	No	Yes	No	Yes

1.5 ORDERING FUNCTIONS BY ASYMPTOTIC GROWTH RATES

When we use asymptotic notation to express the rate of growth of an algorithm's running time in terms of the input size n. Suppose that an algorithm took a constant amount of time, regardless of the input size [4].For example, if you were given an array that is already sorted into increasing order and you had to find the minimum element, it would take constant time, since the minimum element must be at index 0. Since we like to use a function of n in asymptotic notation, we could say

that this algorithm runs in $\Theta(n^0)$, because $n^0 = 1$, and the algorithm's running time is within some constant factor of 1.

Now suppose an algorithm took $\Theta(\log_{10} n)$ time. Whenever the base of the logarithm is a constant, it doesn't matter what base we use in asymptotic notation because there's a mathematical formula that says

$$\log_a n = \frac{\log_b n}{\log_b a}$$

for all positive numbers a, b and c. Therefore, if a and b are constant, then $\log_a n \ and \ \log_b n$

Differ only by a factor of $\log_b a$ and that is a costant factor which we can ignore in asymptotic notation [4].

Worst case running time of binary search is $\theta(\log_a n)$ for any positive constant a. The no of guesses is at most $\log_2 n + 1$, generating and testing each guess takes constant time and setting up and returning take constant time. In practice we write binary search takes $\theta(\log_2 n)$ time.

Now suppose, a and b are two constant and a<b, then a running time $\theta(n^a)$ grows more slowly than a running time of n^b . For example, a running time of (n) , which is $\theta(n^1)$ grows more slowly than a running time of $\theta(n^2)$. The exponents don't have to be integers. For example, a running time of $\theta(n^2)$ grows more slowly than a running time of $\theta(n^2 \sqrt{n}, which \ is \ \theta(n^{2.5})$.
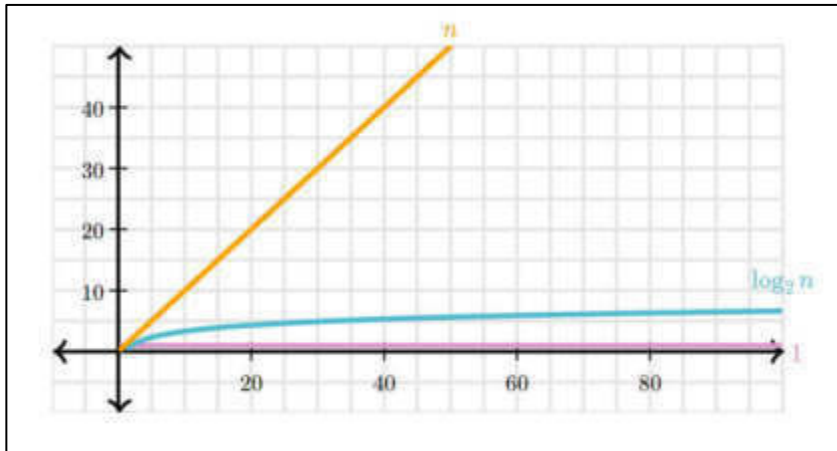
Following graph compares the growth of $n, n^2 \ and \ n^{2.5}$
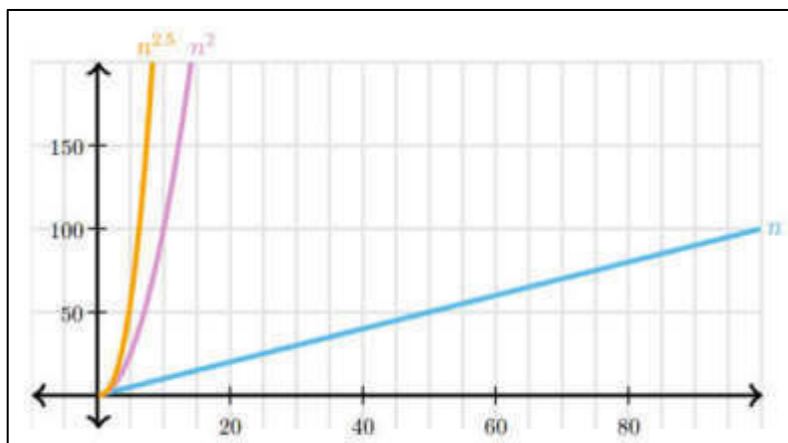
Logarithms grow more slowly than polynomials, i.e. $\theta(\log_2 n)$ grows more slowly than $\theta(n^a)$ for any positive constant a. But since the value of $\log_2 n$ increases as n increases, $\theta(\log_2 n)$ grows faster than $\theta(1)$.

Following graph compares the growth of 1, n, and $\log_2 n$:

list of functions in asymptotic notation that we often encounter when analyzing algorithms is given below (ordered by slowest to fastest growing):

1. $\theta(1)$

$$2. \theta \log_2 n$$

3. $\theta(n)$

4. $\theta(n\log_2 n)$

5. $\theta(n^2)$

6. $\theta(n^2\log_2 n)$

7. $\theta(n^3)$

8. $\theta(2^n)$

9. $\theta(n!)$

[Note that an exponential function a^n, where a>1, grows faster than any polynomial function n^b, where b is any constant].

Example: Rank these functions according to their growth, from slowest growing to fastest growing.

$8n^2$

$n \, \log_6 n$

64

$\log_2 n$

$n \, \log_2 n$

$\log_8 n$

$6n^3$

$4n$

8^{2n}

Solution:

We have several different types of functions here, so we start by thinking about the general properties of those function types and how their rate of growth compares. Here's a reminder of different function types shown here, in order of their growth:

1. Constant functions (for e.g. 64)

2. Logarithmic functions (for e.g. $\log_8 n$, $\log_2 n$)

3. Linear functions (for e.g. 4n)

4. Linearithmic functions (for e.g. $n \log_2 n, n \, \log_8 n$)

5. Polynomial functions (for e.g. $8n^2$, $6n^3$)

6. Exponential functions (for e.g. 8^{2n})

We have several types where there are multiple functions - logarithmic functions, linearithmic functions, and polynomial functions, so we have to look more closely at each of them to compare their growth within the class. Within the logarithmic functions, the lesser bases grow more quickly than the higher bases – so $\log_2 n$ will grow more quickly than $\log_8 n$. Following graph shows the scenario:
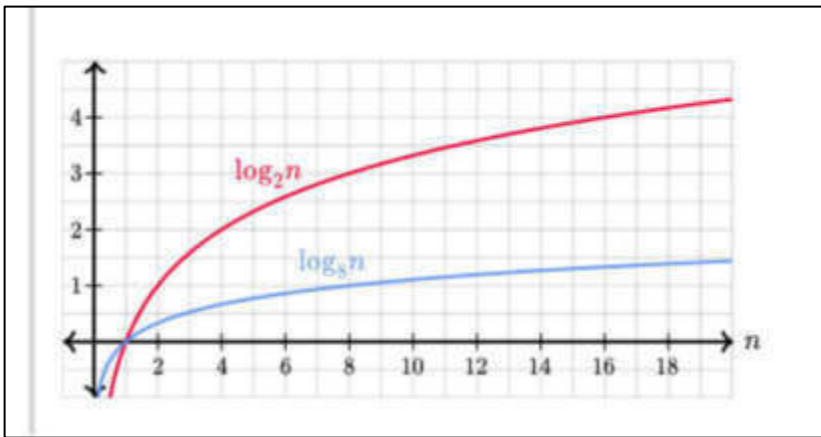
The linearithmic functions are those that multiply linear terms by a logarithm, of the form $n \log_k n$. With the n being the same in both, then the growth is dependent on the base of the logarithms. The lesser bases grow more quickly than the higher bases – so $n \log_2 n$ will grow more quickly than $n \log_6 n$. We can see that in the following graph:
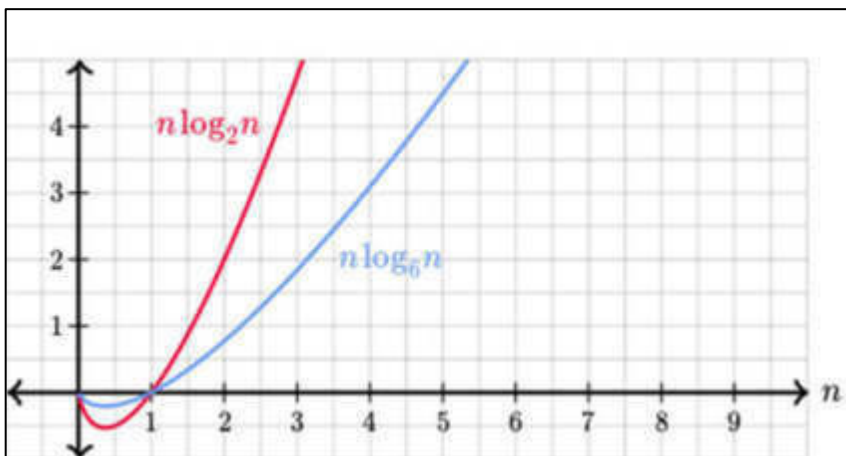
Within the polynomial functions, $8n^2$ will grow more slowly than $6n^3$, since it has a lesser exponent. We don't even have to look at the constants in front, since the exponent is more significant.

So, the correct order of the functions would be:

64

$\log_8 n$

$\log_2 n$

$4n$

$n \log_6 n$

$n \log_2 n$

$8n^2, 6n^3$

8^{2n}

CHECK YOUR PROGRESS

Answer the following:

1. For the functions, n^k and c^n, what is the asymptotic relationship between these functions? Assume that $k \geq 1$ and $c > 1$ are constant

    [A] n^k is $O(c^n)$

    [B] n^k is $\Omega(c^n)$

    [C] n^k is $\Theta(c^n)$

    [D] n^k is $o(c^n)$

2. For the functions, 8^n and 4^n, what is the asymptotic relationship between these functions:

    [A] 8^n is $O(4^n)$

    [B] 8^n is $\Omega(4^n)$

    [C] 8^n is $\Theta(4^n)$

    [D] 8^n is $o(4^n)$

3. For the functions, $\log_2 n$ and $\log_8 n$, what is the asymptotic relationship between these functions?

    [A] $\log_2 n$ is $O(\log_8 n)$

    [B] $\log_2 n$ is $\Omega(\log_8 n)$

    [C] $\log_2 n$ is $\Theta(\log_8 n)$

    [D] $\log_2 n$ is $o(\log_8 n)$

4. Consider the following functions from positive integer's real numbers:

$$10, \sqrt{n}, \ n, \log_2 n, 100/n$$

The correct arrangement of the above functions in increasing order of asymptotic complexity is:

[A] $\log_2 n, \frac{100}{n}, 10, \sqrt{n}, \ n$

[B] $\frac{100}{n}, 10, \ \log_2 n, \sqrt{n}, \ n$

[C] $10, 100/n, \sqrt{n}, \log_2 n, n$

[D] $\frac{100}{n}, \log_2 n, 10, \sqrt{n}, \ n$

5. Consider the following three functions

$$f_1 = 10^n \quad f_2 = n^{\log n} \quad f_3 = n^{\sqrt{n}}$$

Which one of the following options arranges the functions in increasing order of asymptotic growth rate?

[A] f_1, f_2, f_3

[B] f_2, f_1, f_3

[C] f_3, f_2, f_1

[D] f_2, f_3, f_1

1.6 SUMMING UP

- Asymptotic notation is used to simplify complex polynomial

- If f(n) is O(g(n)) then a*f(n) is also O(g(n)) ; where a is a constant

- If f(n) is given then f(n) is O(f(n))

- If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n))

- If f(n) is Θ(g(n)) then g(n) is Θ(f(n))

- If f(n) is O(g(n)) then g(n) is Ω (f(n))

- The *asymptotic* efficiency of algorithms is required when we look at input sizes large enough to make only the order of growth of the running time relevant.

- **Asymptotic growth** means the rate at which the function grows. Growth rate means the complexity of function or the amount of resource it takes up to compute.

- When we have only an *asymptotic upper bound*, we use O-notation

- When we have only an asymptotic *upper* bound on a function, Ω-notation provides an *asymptotic lower bound*

- Θ-notation provides an *asymptotically tight bound*

1.7 ANSWERS TO CHECK YOUR PROGRESS

1. [A]

2. [B]

3. [A], [B], [C]

4. [B]

5. [D]

1.8 POSSIBLE QUESTIONS

Solve the following problems:

1.   Let, $p(n) = \sum_{i=0}^{d} a_i n^i$

Where if $a_d > 0, be\ a\ degree\ d$ polynomial in n, and let k be a constant. By using definitions of the asymptotic notations, prove the following properties:

   a.   If k >d, then $p(n)=o(n^k)$

   b.   If k <d, then $p(n)=\omega(n^k)$

2. Rank these functions according to their growth, from slowest growing (at the top) to fastest growing (at the bottom).

   1, n3, n^2, $(3/2)^n$, n, 2^n

3. Which kind of growth best characterizes each of these functions?

	Constant	Linear	Polynomial	Exponential
$2n^3$				
$(3/2)n$				
$(3/2)^n$				
1				
2^n				
$3n$				
1000				
$3n^2$				

[Hints: 1 and 1000 are constant, 3n and (3/2)n are linear, $2n^3$ and $3n^2$ are polynomial, 2^n and $(3/2)^n$ are exponential]

1.9 REFERENCES AND SUGGESTED READINGS

[1] T.H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006

[2] Data Structure through c in Depth, by S.K. Srivastav and Deepali Srivastava

[3]https://www.cs.cornell.edu/courses/cs3110/2014sp/recitations/20/review-of-asymptotic-complexity.html#2

[4] https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/functions-in-asymptotic-notation

Space for learners:

UNIT 4:   RECURRENCES

4.1   INTRODUCTION

In Unit 1, analysis of algorithms has been already discussed where the algorithms without recursive calls to themselves are considered as examples. In case of algorithms with recursive calls to themselves, recurrences are used to express them. In this unit, we are going to learn about recurrences. We will also examine the methods to solve recurrences so that the run-time complexities of algorithms described by the recurrences can be estimated.

4.2 UNIT OBJECTIVES

After reading this unit you are expected to be able to learn:

- Definition of recurrences

- About Substitution method to solve recurrences.

- About advantages and limitations of Substitution method.

- About Recursion Tree Method to solve recurrences.

- About advantages and limitations of Recursion Tree method.

- About the Master Theorem to solve recurrences.

- About advantages and limitations of Master Theorem.

4.3 RECURRENCES

A recurrence can be defined as an inequality or equation which demonstrates the run time of an algorithm in terms of its values on lesser inputs. Algorithms with recursive calls to themselves are demonstrated by recurrences. For example: The run time of Quick sort algorithm with best case partitioning is represented by the following recurrence.

$$F(n) \leq 2 F(n/2) + C n$$

In the above recurrence, n is the number of inputs and it is greater than n1 where n1 is a constant. On the other hand C is a constant. The run time of Quick sort algorithm, F (n) is demonstrated in terms of two F (n / 2) in this recurrence.

By solving a recurrence of an algorithm, the run-time complexity of the algorithm can be estimated. In the process of solving recurrences, usually we have to make some assumptions. For example, if F (n) is the runtime of an algorithm then n must be an integer. We can also consider F (n) = Θ (1) in case of sufficiently small value of n. The floors, ceilings and boundary conditions are usually ignored to solve recurrences.

There are three methods available to solve recurrences that are Substitution method, Recursion tree method and Master method.

STOP TO CONSIDER

The run time of Quick sort algorithm with worst case partitioning is represented by the following recurrence.

$F(n) = F(n-1) + \Theta(n)$

4.4 SUBSTITUTION METHOD

The substitution method is a simple but powerful technique used for solving recurrences. It consists of two steps as mentioned below.

Step 1: At first the solution of a particular recurrence is assumed.

Step 2: Secondly, mathematical induction is used to prove that the assumed solution is correct or valid.

STOP TO CONSIDER

The substitution method can be used to estimate both upper and lower bounds on recurrences.

4.4.1 Examples of Solving Recurrences using Substitution Method

In this section, we are going to solve two recurrences using Substitution method.

(a) F (n) = 2 F (n / 2) + Θ (n)

Solution:

In step 1, we have to guess the solution of the given recurrence. So, let us guess the solution of the given recurrence as $O(n \lg n)$.

In the second step, using mathematical induction, we have to prove that $F(n) \le t\, n \lg n$, where t is a constant and it is greater than 0.

Let us assume that the solution $O(n \lg n)$ is true for n/2.

Now we can state that $F(n/2) \le t\ (n/2)\ \lg(n/2)$ is true.

Then substituting the F (n/2) in the given recurrence, we can have the following expressions.

$$F(n) \leq 2\, t\, (n/2)\, \lg(n/2) + n$$

$$\leq t\, n\, (\lg n - \lg 2) + n$$

$$\leq t\, n\, (\lg n - 1) + n$$

$$\leq t\, n \lg n - t\, n + n$$

$$\leq t\, n \lg n - n\, (t - 1)$$

For $t \geq 1$, we can have, $F(n) \leq t\, n \lg n$

Now according to mathematical induction, we also have to prove the assumed solution to be correct for the boundary conditions. In this process, we are required to prove $F(n) \leq t\, n \lg n$ for boundary conditions where $n \geq c$ and c is a constant.

Let, $F(1) = 1$, $c \geq 2$

Then for $n = 2$, we have,

$$F(2) = 2\, F(2/2) + 2$$

$$= 4$$

$$t\, n \lg n = 2 * 2 * \lg 2 \qquad [\text{For } t = 2]$$

$$= 4$$

So, it can be stated that $F(2) \leq 2 * 2 * \lg 2$

Again, for $n = 3$, we have

$$F(3) = 2\, F(3/2) + 3$$

$$= 5$$

$$t\, n \lg n = 2 * 3 * \lg 3 \qquad [\text{For } t = 2]$$

$$\cong 6$$

So, it can be stated that $F(3) \leq 2 * 3 * \lg 3$

From the above observations, it is proved that the assumed solution O (n lg n) is true for the boundary conditions of $n = 2$ and $n = 3$ with any choice of $t \geq 2$.

Finally, using mathematical induction, it is proved that $O(n \lg n)$ is the correct assumption as a solution for the recurrence $F(n) = 2F(n/2) + \Theta(n)$.

(b) $F(n) = F(n-1) + \Theta(n)$

Solution:

In step 1, we have to guess the solution of the given recurrence. So, let us guess the solution of the given recurrence as $O(n^2)$.

In the second step, using mathematical induction, we have to prove that $F(n) \le t\, n^2$, where t is a constant and it is greater than 0.

Let us assume that the solution $O(n^2)$ is true for $(n-1)$.

Now we can state that $F(n-1) \le t\,(n-1)^2$ is true.

Then substituting the $F(n-1)$ in the given recurrence, we can have the following expressions.

$$
\begin{aligned}
F(n) &\le \; t\,(n-1)^2 + n \\
&\le \; t\,(n^2 - 2n + 1) + n \\
&\le \; t\,n^2 - 2tn + t + n \\
&\le \; t\,n^2 - (2tn - t - n)
\end{aligned}
$$

For $t \ge 1$, we can have, $F(n) \le t\,n^2$

Now according to mathematical induction, we also have to prove the assumed solution to be correct for the boundary conditions. In this process, we are required to prove $F(n) \le t\,n^2$ for boundary conditions where $n \ge c$ and c is a constant.

Let, $F(1) = 1$, $c \ge 2$

Then for $n = 2$, we have,

$$F(2) = F(2-1) + 2$$
$$= 3$$
$$t\,n^2 = 2 * 2^2 \qquad [\text{For } t = 2]$$
$$= 8$$

So, it can be stated that $F(2) \le 2 * 2^2$

Again, for $n = 3$, we have

$$F(3) = F(3-1) + 3$$
$$= F(2) + 3$$
$$= 6$$
$$t\,n^2 = 2 * 3^2 \qquad [\text{For } t = 2]$$
$$= 18$$

So, it can be stated that $F(3) \leq 2 * 3^2$

From the above observations, it is proved that the assumed solution O (n^2) is true for the boundary conditions of n = 2 and n = 3 with any choice of $t \geq 2$.

Finally, using mathematical induction, it is proved that O (n^2) is the correct assumption as a solution for the recurrence $\qquad$ F (n) = F (n-1) + Θ (n).

4.4.2 Advantages and Limitations of Substitution Method

Advantages of Substitution method:

- The substitution method is a simple and powerful technique to solve recurrences.
- Appropriate solution of a recurrence can be easily estimated by using this method.
- Approximately all recurrences can be solved by Substitution method.

Limitations of Substitution method:

- Substitution method can be effective only when the assumption of the solution for particular recurrence is a correct or valid one. So, the main problem is that we don't have a standard approach to make a good assumption of an appropriate solution for a particular recurrence. As a result, it may be difficult to guess an appropriate solution for a complex recurrence.
- In some cases, it may happen that the assumption of the solution of a recurrence is correct but it may not be proved by mathematical induction. For example: O (n) is a correct assumption as the solution of the recurrence, F (n) = 2F (n/2) + 1. But using mathematical induction, it cannot be proved.

CHECK YOUR PROGRESS

1. **Multiple Choice Questions:**

A. Which of the following is not an example of recurrence?
   (i) $F(n) = 2 F(n/2) + O(n)$
   (ii) $F(n) = T(n/2) + 1$
   (iii) $F(n) = F(n-1) + O(n)$
   (iv) None of the above

B. Which of the following algorithm is not described by a recurrence?
   (i) Linear search
   (ii) Binary search
   (iii) Merge sort
   (iv) All of the above

C. Which of the recurrence can be used to describe binary search algorithm?
   (i) $F(n) = 2 F(n/2) + 1$
   (ii) $F(n) = F(n/3) + 1$
   (iii) $F(n) = F(n/2) + 1$
   (iv) None of the above

D. Which of the following is not true in case of substitution method?
   (i) Mathematical induction is used in substitution method.
   (ii) Substitution method is used to solve recurrences.
   (iii) Substitution method is a complex technique.
   (iv) None of the above.

E. Which of the following is a drawback of substitution method?
   (i) Assumption of a proper solution for complex recurrences may be difficult.
   (ii) It is a complex method for beginners.
   (iii) Appropriate solution of a recurrence cannot be easily estimated by using this method.
   (iv) None of the above

4.5 RECURSION TREE METHOD

Recursion Tree is a tree structure where each node represents the cost of a particular recursive sub-problem which is a part of an algorithm with recursive calls. So, recursion tree can be used to represent recurrences in terms of costs associated with each recursive calls. The recursion tree can be used to solve a recurrence and this technique is referred as Recursion tree method.

The steps of the Recursion tree method are stated as follows:

Step 1: At first, an appropriate recursion tree for a particular recurrence has to be drawn.

Step 2: In the second step, cost associated with each level other than the last level in the tree is estimated by adding costs represented by each node available in each level.

Step 3: In the third step, total number of levels and the total number of nodes in the last level are estimated.

Step 4: In the fourth step, cost associated at the last level is estimated.

Step 5: In the final step, summation of all costs associated with all the levels in the recursion tree is performed to obtain an expression to represent the total cost of the particular recurrence. Then asymptotic notation is determined by simplifying this estimated expression.

STOP TO CONSIDER

Recursion tree method can also be used to estimate a possible solution for a particular recurrence and this estimated solution can be verified using Substitution method.

4.5.1 Examples of Solving Recurrences using Recursion Tree Method

In this section, we are going to solve three recurrences using Recursion tree method.

(a) F (n) = 2 F (n / 2) + Θ (n)

Solution:

The given recurrence can also be written as,

$$F (n) = 2 F (n / 2) + C n,$$

where C is a constant coefficient and it is greater than 0.

At first, we have to construct an appropriate recursion tree for the given recurrence.

Construction of the recursion tree for the recurrence,

F (n) = 2 F (n / 2) + C n, is shown in figure 4.1 and 4.2.
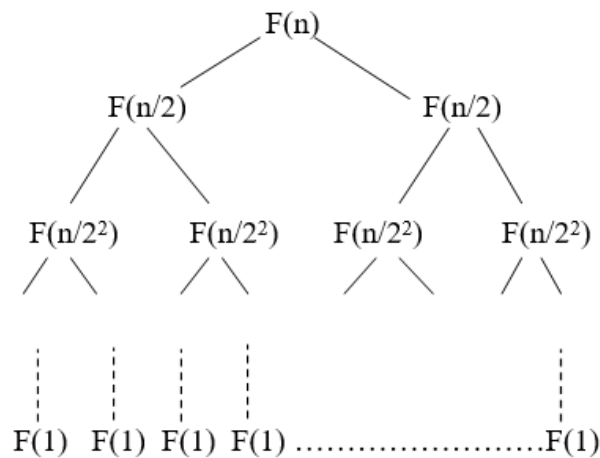
Figure 4.1: Recursion tree for the recurrence,
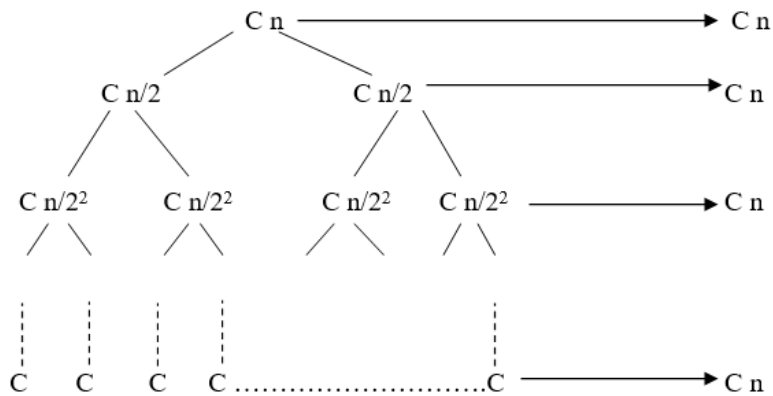F (n) = 2 F (n / 2) + Θ (n)

Figure 4.2: Recursion tree with cost at each node for the recurrence,
$$F (n) = 2 F (n / 2) + \Theta (n)$$

In the second step, we have to estimate costs associated with each level in the recursion tree.

From the given recurrence, it is observed that the cost of one sub-problem with input n is C n where C > 0.

So, from the recursion tree in figure 4.2, we can have,

The cost at the root node or the 0th level = C n

The cost at the 1st level = C n/2 + C n/2

$$= C n$$

The cost at the 2nd level = C n/2^2 + C n/2^2 + C n/2^2 + C n/2^2

$$= C n$$

It is observed that the input sizes of the sub-problems decrease as the number of levels in the recursion tree increases. As a result, the input size of each node in the last level becomes 1. Now if we let L be the number of the last level in the recursion tree then the following equation can be stated.

$$n / 2^L = 1$$
$$\Rightarrow n = 2^L$$
$$\Rightarrow \lg n = \lg 2^L$$
$$\Rightarrow \lg n = L$$

So, the total number of levels in the recursion tree = lg n +1

The total number of nodes in the last level of the recursion tree = $2^{\lg n}$

So, the cost at the last level = $\underbrace{C + C + C + \ldots\ldots\ldots + C}_{2^{\lg n}}$

$$= 2^{\lg n}\, C$$
$$= C\, n\, \lg 2$$
$$= C\, n$$

Now, finally summing up the costs of all levels in the recursion tree, we can have the following expression.

$$F(n) = \underbrace{C\,n + C\,n + C\,n + \ldots\ldots\ldots + C\,n}_{\lg n + 1}$$

$$= C\,n\,(\lg n + 1)$$
$$= C\,n\,\lg n + C\,n$$
$$= C\,n\,\lg n + \Theta(n)$$
$$= O(n\,\lg n)$$

So, we can state that $O(n \lg n)$ is the asymptotic upper bound of the recurrence $F(n) = 2\,F(n/2) + \Theta(n)$.

(c) $F(n) = 2\,F(n/2) + O(n^2)$

Solution:

The given recurrence can also be written as,

$F(n) = 2\,F(n/2) + C\,n^2$,

where C is a constant coefficient and it is greater than 0.

At first, we have to construct an appropriate recursion tree for the given recurrence.

Construction of the recursion tree for the recurrence,

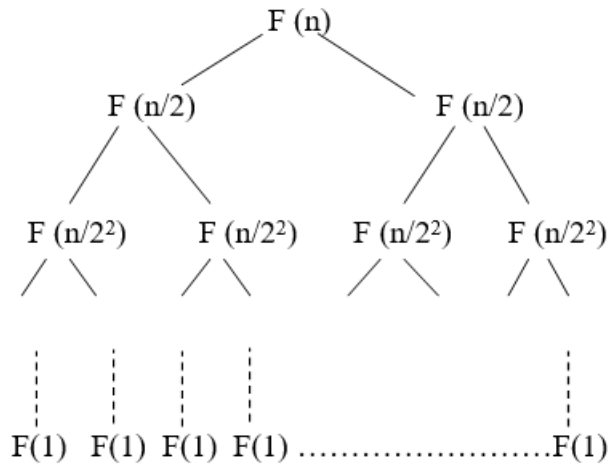$F(n) = 2\,F(n/2) + C\,n^2$, is shown in figure 4.3 and 4.4.

Figure 4.3: Recursion tree for the recurrence, $F(n) = 2 F(n/2) + O(n^2)$
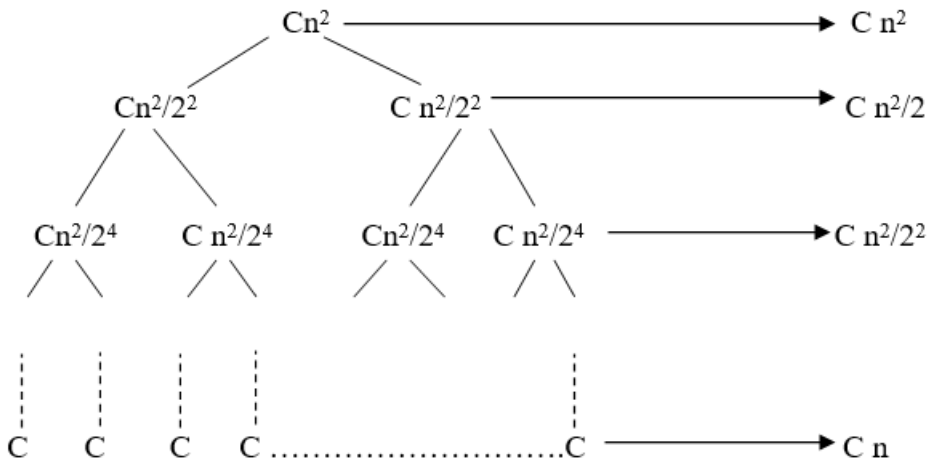

Figure 4.4:   Recursion tree with cost at each node for the recurrence, $F(n) = 2 F(n/2) + O(n^2)$

In the second step, we have to estimate costs associated with each level in the recursion tree.

From the given recurrence, it is observed that the cost of one sub-problem with input n is $C n^2$ where $C > 0$.

So, from the recursion tree in figure 4.4, we can have,

The cost at the root node or the 0th level $= C n^2$

The cost at the 1st level $= C n^2/2^2 + C n^2/2^2$

$$= C n^2/2$$

The cost at the 2nd level $= C n^2/2^4 + C n^2/2^4 + C n^2/2^4 + C n^2/2^4$

$$= C\, n^2/2^2$$

It is observed that the input sizes of the sub-problems decrease as the number of level in the recursion tree increases. As a result, the input size of each node in the last level becomes 1. Now if we let L be the number of the last level in the recursion tree then the following equation can be stated.

$$n\,/\,2^L = 1$$
$$\Rightarrow\; n\; =\; 2^L$$
$$\Rightarrow\; \lg n = \lg 2^L$$
$$\Rightarrow\; \lg n = L$$

So, the total number of levels in the recursion tree $= \lg n + 1$
The total number of nodes in the last level of the recursion tree $= 2^{\lg n}$
So, the cost at the last level $= \underbrace{C + C + C + \ldots\ldots\ldots + C}_{2^{\lg n}}$

$$= 2^{\lg n}\, C$$
$$= C\, n \lg 2$$
$$= C\, n$$

Now, finally summing up the costs of all levels in the recursion tree, we can have the following expression.

$$F\,(n) = C\,n^2 + C\,\frac{n^2}{2} + C\,\frac{n^2}{2^2} + \ldots\ldots\ldots + C\,\frac{n^2}{2^{\lg n - 1}} + C\,n$$
$$= C\,n^2 \sum_{a=0}^{\lg n - 1} \frac{1}{2^a} + C\,n$$
$$< C\,n^2 \sum_{a=0}^{\alpha} \frac{1}{2^a} + C\,n$$
$$= n^2 C\left(\frac{1}{1 - \frac{1}{2}}\right) + C\,n$$
$$= O\,(n^2)$$

So, we can state that $O\,(n^2)$ is the asymptotic upper bound of the recurrence $F\,(n) = 2\,F\,(n\,/\,2) + O\,(n^2)$.

(d) F (n) = 3 F (n / 4) + O (n)

Solution:

The given recurrence can also be written as,

$$F (n) = 3 F (n / 4) + C n,$$

where C is a constant coefficient and it is greater than 0.

At first, we have to construct an appropriate recursion tree for the given recurrence.

Construction of the recursion tree for the recurrence,

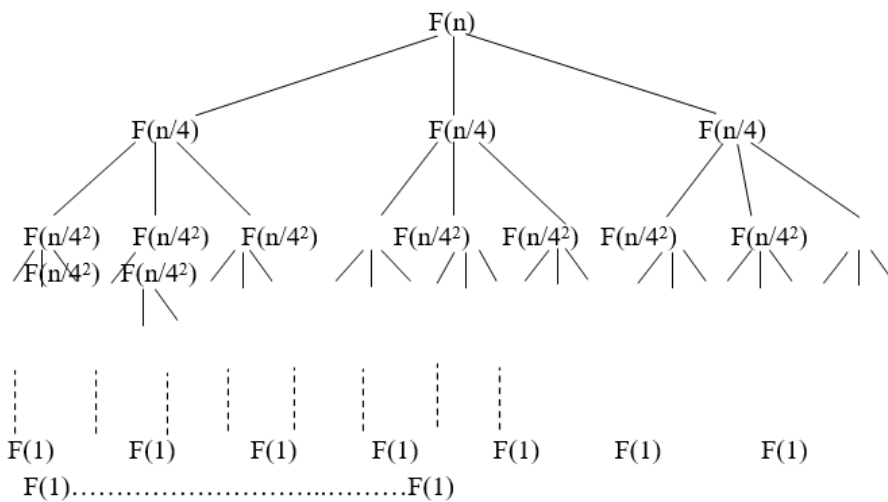$$F (n) = 3 F (n / 4) + C n,$$ is shown in figure 4.5 and 4.6.

Figure 4.5: Recursion tree for the recurrence, F (n) = 3 F (n / 4) + O (n)
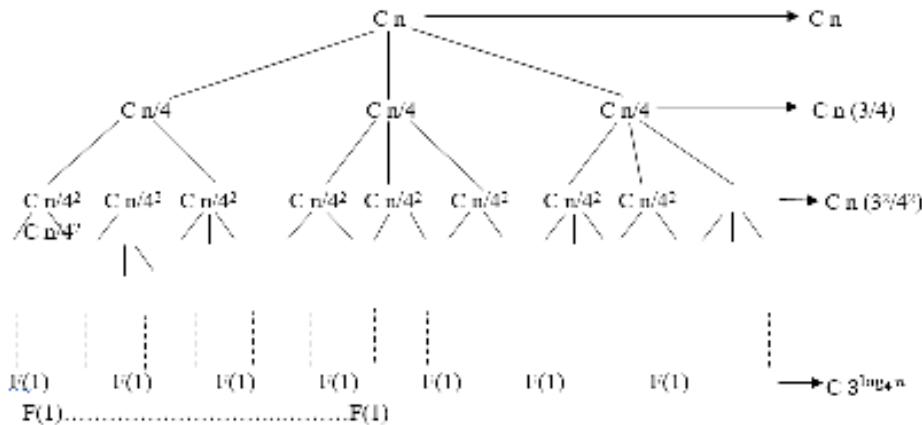
Figure 4.6: Recursion tree with cost at each node for the recurrence,
$$F(n) = 3\,F(n/4) + O(n)$$

In the second step, we have to estimate costs associated with each level in the recursion tree.

From the given recurrence, it is observed that the cost of one sub-problem with input n is C n where C > 0.

So, from the recursion tree in figure 4.6, we can have,

The cost at the root node or the 0th level = C n

The cost at the 1st level = C n/4 + C n/4 + C n/4

$$= C\,n\,(3/4)$$

The cost at the 2nd level = C n/4^2 + C n/4^2 + C n/4^2 + C n/4^2 + C n/4^2 + C n/4^2 + C n/4^2 + C n/4^2 + C n/4^2

$$= C\,n\,(3^2/4^2)$$

It is observed that the input sizes of the sub-problems decrease as the number of level in the recursion tree increases. As a result, the input size of each node in the last level becomes 1. Now if we let L be the number of the last level in the recursion tree then the following equation can be stated.

$$n / 4^L = 1$$
$$\Rightarrow n = 4^L$$
$$\Rightarrow \log_4 n = \log_4 4^L$$
$$\Rightarrow \log_4 n = L$$

So, the total number of levels in the recursion tree $= \log_4 n + 1$

The total number of nodes in the last level of the recursion tree $= 3^{\log_4 n}$

So, the cost at the last level $= C + C + C + \ldots\ldots\ldots + C$

$$\underbrace{\qquad\qquad\qquad\qquad}_{3^{\log_4 n}}$$

$$= 3^{\log_4 n}\, C$$
$$= C\, n^{\log_4 3}$$
$$= \Theta\,(n^{\log_4 3})$$

Now, finally summing up the costs of all levels in the recursion tree, we can have the following expression.

$$F(n) = C\,n + C\,n\,(3/4) + C\,n\,(3^2/4^2) + \ldots\ldots + C\,n\,(3^{\log_4 n - 1}/4^{\log_4 n - 1}) + \Theta\,(n^{\log_4 3})$$

$$= C\,n \sum_{a=0}^{\log_4 n - 1} (3^a/4^a) + \Theta\,(n^{\log_4 3})$$

$$< C\,n \sum_{a=0}^{\alpha} (3^a/4^a) + \Theta\,(n^{\log_4 3})$$

$$= n\,C\left(\frac{1}{1-\frac{3}{4}}\right) + \Theta\,(n^{\log_4 3})$$

$$= O\,(n)$$

So, we can state that O (n) is the asymptotic upper bound of the recurrence, F (n) = 3 F (n / 4) + O (n).

4.5.2 Advantages and Limitations of Recursion Tree Method

Advantage of Recursion tree method:

Recursion tree method is a straightforward and standard approach to estimate an appropriate solution or a proper assumption of solution for a particular recurrence.

Limitations of Recursion tree method:

- It may be difficult to draw a proper recursion tree for complex recurrences.

- We have to be very watchful to draw recursion trees and summing costs for recurrences. Otherwise we cannot able to estimate correct or valid solutions or proper assumptions of solutions for recurrences.

4.6 MASTER METHOD

The Master method to solve recurrences is based on the Master theorem. So, at first we have to know the Master theorem that is presented below.

Master theorem:

Let F (n) be a recurrence defined as follows:

$$F (n) = x F (n/y) + f (n) ,$$

Where x and y are two constants such that $x \geq 1$ and $y > 1$. f(n) is a function and n is a nonnegative integer. n/y can be considered as either floor of n/y or ceiling of n/y.

Now according to the Master theorem, three cases are presented below to bound F(n) with asymptotic notations.

Case 1: If $f (n) = O (n^{\log_y x - \in})$, where $\in$ is a constant and it is greater than 0 then $F (n) = \Theta (n^{\log_y x})$.

Case 2: If $f (n) = \Theta (n^{\log_y x})$, then $F (n) = \Theta (n^{\log_y x} \lg n)$.

Case 3: If $f (n) = \Omega (n^{\log_y x + \in})$, where $\in$ is a constant and it is greater than 0 and if $xf(n/y) \leq d \ f(n)$ for some constant d<1 and all adequately large n, then $F(n) = \Theta(f(n))$.

If we can establish a recurrence with any of the above cases then using the Master theorem, we can directly find out the solution of the recurrence.

4.6.1 Examples of Solving Recurrences using Master Theorem

In this section, we are going to solve four recurrences using Master method.

(a) F (n) = 4F (n/2) + n

Solution:

From the given recurrence, it is observed that x = 4, y = 2 and f (n) = n.

So, $n^{\log_y x} = n^{\log_2 4} = n^2$

$$f(n) = n = n^1 = n^{2\text{-}1} = O(n^{\log_2 4 - \epsilon}), \text{ where } \epsilon = 1$$

From the above observations, we can apply case 1 from the Master theorem.

According to case 1 from Master Theorem we can state that

$F(n) = \Theta(n^{\log_y x})$ that means $F(n) = \Theta(n^2)$.

(b) F (n) = 4F (n/2) + n²

Solution:

From the given recurrence, it is observed that x = 4, y = 2 and f(n) = n².

So, $n^{\log_y x} = n^{\log_2 4} = n^2$

$$f(n) = n^2 = \Theta(n^{\log_2 4}) = \Theta(n^{\log_y x})$$

From the above observations, we can apply case 2 from the Master theorem.

According to case 2 from Master Theorem we can state that

$F(n) = \Theta(n^{\log_y x} \lg n)$ that means $F(n) = \Theta(n^2 \lg n)$

(c) F (n) = 4 F (n/2) + n³

Solution:

From the given recurrence, it is observed that x = 4, y = 2 and f(n) = n³.

So, $n^{\log_y x} = n^{\log_2 4} = n^2$

$$f(n) = n^3 = \Omega(n^{\log_2 4 + 1}) = \Omega(n^{\log_y x + \epsilon}), \text{ where } \epsilon = 1$$

Again, $xf(n/y) = 4(n^3/2^3) = (1/2)n^3 = (1/2)f(n)$

Here d = ½ and so, d < 1

From the above observations, we can apply case 3 from the Master theorem.

According to case 3 from Master Theorem we can state that

F (n) = Θ (f (n)) that means F (n) = Θ (n^3).

(d) F (n) = F (n/2) + Θ (1)

Solution:

From the given recurrence, it is observed that x = 1, y = 2 and

$\quad$ f (n) = 1.

So, $n^{\log_y x} = n^{\log_2 1} = n^0 = 1$

$\quad$ f (n) = 1 = n^0 = Θ ($n^{\log_2 1}$) = Θ ($n^{\log_y x}$)

From the above observations, we can apply case 2 from the Master theorem.

According to case 2 from Master theorem we can state that

F (n) = Θ ($n^{\log_y x}$ lg n) that means F (n) = Θ (lg n)

4.6.2 Advantages and Limitations of Master Method

Advantage of Master method:

The Master method is a very simple and direct way to estimate solutions for recurrences.

Limitations of Master Method:

The main problem of Master method is that all types of recurrences cannot be solved by using this method. There are some recurrences available where Master theorem cannot be applicable. For example: Consider the following recurrence.

$\quad$ F (n) = 2F (n/2) + n/log n

From the above recurrence, it is observed that x = 2, y = 2 and f(n) = n/log n.

So, $n^{\log_y x} = n$ and $n^{\log_y x} /$ f(n) = log n

From the above observations, it is found that f(n) is asymptotically smaller than $n^{\log_y x}$ and so according to the Master theorem case 1 should be applicable to the above recurrence. But it is also observed that f (n) is not polynomially smaller than $n^{\log_y x}$ and as a result case 1 is not applicable in this case.

CHECK YOUR PROGRESS

2. **Multiple choice questions:**

A. Which of the following is true in case of Recursion tree method?
   (i) Construction of an appropriate recursion tree for a recurrence is the most important part to solve the recurrence using Recursion tree method.
   (ii) Cost at each level in a recursion tree is estimated by mathematical induction.
   (iii) Recursion tree method cannot be used to find out the solution of all types of recurrences.
   (iv) None of the above.

B. Which of the following recurrences can have a recursion tree where each parent node linked with three child nodes?
   (i) F (n) = 2F (n/3) + n
   (ii) F (n) = 4F (n/2) + n
   (iii) F (n) = 3F (n/4) + n
   (iv) None of the above

C. Which of the following recurrences can have a recursion tree where the cost at the root is C n^2 ?
   (i) F (n) = 2F (n/3) + O (n lg n)
   (ii) F (n) = 3F (n/3) + O (n²)
   (iii) F (n) = 4F (n/3) + O (n)
   (iv) None of the above

D. Which of the following is not true in case of Master method?
   (i) Master method is based on Master theorem.
   (ii) Master method is a direct way to solve recurrences.
   (iii) Solution estimated by Master method need not be proved by substitution method.
   (iv) All kind of recurrences can be solved by Master method.

E. Which of the following recurrences cannot be solved by Master method?
  (i) F (n) = 2F (n/2) + n lg n
 (ii) F (n) = 2F (n/2) + n
(iii) F (n) = 3F (n/4) + n lg n
 (iv) All of the above

4.7 SUMMING UP

- A recurrence is an inequality or equation which express the run time of an algorithm in terms of its values on lesser inputs. For example: F (n) = 3F (n/3) + n.

- Substitution method, Recursion tree method and Master method are the methods that can be used to solve recurrences.

- In Substitution method, at first the solution of a recurrence is assumed and then mathematical induction is used to prove that the assumed solution is correct or valid.

- In Recursion tree method, at first proper recursion tree is constructed for a recurrence. Then cost is estimated at each level of the recursion tree. Finally, all the costs associated with all the levels are added to find out the total cost of the particular recurrence.

- The Master method is based on Master theorem. All recurrences cannot be solved by Master method.

4.8 ANSWER TO CHECK YOUR PROGRESS

1 . A. (ii) , B. (i) , C. (iii) , D. (iii) , E. (i)

2. A. (i) , B. (iii) , C. (ii) , D. (iv), E. (i)

4.9   POSSIBLE QUESTIONS

1) Explain Substitution method with examples.
2) Solve the following recurrence using Substitution method.

$$F (n) = 3 F (n / 4) + O (n)$$

3) Explain Recursion tree method with examples.
4) Solve the following recurrence using Recursion tree method.

$$F (n) = F (n-1) + \Theta (n)$$

5) Write down the Master theorem.
6) Solve the following recurrences using Master method.
   (a) $F (n) = 9F (n/3) + \Theta (n)$
   (b) $F (n) = F (2n/3) + \Theta (1)$
   (c) $F (n) = 3F (n/4) + O (n \lg n)$

7) Write down the limitations of Substitution method and Master method.

4.10 REFERENCES AND SUGGESTED READINGS

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. MIT press.

UNIT 5: AMORTIZED ANALYSIS

Unit Structure:

5.1 INTRODUCTION

Several different kinds of operations are supported by data structures. Each operation has its own cost in terms of time and space. Amortized analysis says that some of these operations can be very expensive and can be performed as it does not increase the overall average cost of each operation. As we are mainly interested in the asymptotic behaviour of the algorithm so we only consider the worst-case average cost per operation in a sequence of many operations.

5.2 UNIT OBJECTIVES

After going through this unit, you will be able to :

- *understand* the fundamental concepts of amortized analysis.

- *differentiate* between amortized analysis and worst case analysis of an algorithm.

- *understand* the different approaches to amortized analysis.

- *perform* amortized analysis of a given algorithm.

5.3 AMORTIZED ANALYIS

Asymptotic analysis of algorithms gives us the worst case analysis of each operation without considering the effect of one operation over the other, whereas amortized analysis can be applied to a sequence of operations giving us precise and detailed analysis. Amortized analysis is applied on data structures that support many operations where the sequence of operations and the multiplicity of each operation is application specific or the associated algorithm specific.

As many operations are involved as part of the amortized analysis, our objective is to perform efficiently as many operations as possible without leaving very few costly operations. Generally, the worst case time of each operation is given by taking into account in calculating the average cost in the worst case. For calculating the amortized cost of an operation, we take the average over all the operations.

In an amortized analysis, the time required to perform a sequence of data-structure operations is averaged over all the operations performed. We can use amortized analysis to show if we average over a sequence of operations then the average cost of an operation is small even if a single operation within the sequence may be expensive. Amortized analysis does not involve probability making it different from average-case analysis. But amortized analysis makes sure the average performance of each operation in the worst case.

5.3.1 Features of Amortized Analysis

Features of Amortized analysis:

- Amortized analysis is applied to algorithms where most of the operations are fast but only a few occasional operations are very slow.

- Amortized analysis involves analyzing a sequence of operations and guarantees that the worst case average time will be lower than the worst case time of a particular expensive operation.

- Amortized analysis gives an upper bound as it the average performance of each operation in the worst case.

- Amortized analysis does not mention anything about the cost of a specific operation in the sequence rather it gives the overall cost of a sequence of operations.

- Amortized analysis may consist of both inexpensive and expensive operations however, amortized analysis will always give that the average cost of an operation is less expensive.

- Amortized analysis does not involve probability as different from average case analysis where inputs are modelled using probability distribution that fits the input.

- Amortized analysis takes care of the fact that some of the expensive operations may pay for future operations by somehow reducing its number or the cost of these operations that may happen in the future.

CHECK YOUR PROGRESS

1. State whether true or false:

a. In asymptotic analysis, we take the average over all the operations.

b. Amortized analysis does not include probability calculations.

c. Amortized analysis involves both inexpensive and expensive operations.

d. Amortized analysis involves calculating the cost of a specific operation in the sequence.

e. Amortized cost is less than the worst case cost of the operation.

5.4 DIFFERENT APPROACHES TO AMORTIZED ANALYSIS

We will now the study the different approaches to amortized analysis. There are mainly 3 different approaches to amortized analysis and these approaches are mainly meant for analysis purpose only.

1. Aggregate analysis method.

2. Accounting method.

3. Potential function method.

5.4.1 Aggregate Analysis method

Aggregate analysis is the simplest method which involves determining an upper bound $T(n)$ on the total cost of a sequence of n operations then dividing $T(n)$ by the number n of operations to obtain the amortized cost or the average cost in the worst case. The average cost per operation is $T(n)/n$. The average cost is taken as the amortized cost of each operation, so that all operations have the same amortized cost. For all operations the same amortized cost $T(n)/n$ is assigned, even if they are of different types.

The other two methods allows for assigning different amortized costs to different types of operations in the same sequence.

5.4.2 Accounting Method

Accounting method involves assigning different costs to different types of operations in which some of the operations will cost more and some less than the actual cost. The cost associated with each operation is called its amortized cost. If an operation's amortized cost is more than the actual cost, the difference is stored as credits in specific objects (elements) in the data structure. These accumulated credits can be used to pay for operations whose amortized cost is less than the actual cost.

Accounting method is similar to maintaining an account with '0' credits (charges). When an operation is performed, a charge / cost are associated. If we overcharge an operation, the excess charge will be deposited in the account as credit. Some operations are free operations for which we do not charge anything. For such operations we may make use of the charges which are available as credit in our account. This analysis ensures that the account is never at debit (negative balance). In this method, the amount charged for each operation type is the amortized cost for that type. The amortized cost will act an upper bound on the actual cost for any sequence of operations and it will be impossible for the account to be in debt as long as the charges are set for each operation. The amortized costs of operations must be chosen very carefully and it must be shown that these charges are sufficient to allow payment for any sequence of operations.

5.4.3 Potential Function Method

The Potential Function method of amortized analysis represents the work as "potential energy" or "potential" that can be used to pay for future operations. In the earlier method, work is stored as credit with specific objects (elements) in the data structure but this method associates potential with the whole data structure rather than with specific objects in the data structure.

The analysis is done by focusing on structural parameters of a data structure such as the number of elements, the height, etc. After

performing any operation, the change in a structural parameter is noted as a function and stored in a data structure. The function that records the changes in parameter is termed as a potential function. If the change in potential is positive, then that operation is over charged and the excess potential will be stored in the data structure similar to accounting method. If the change in potential is negative, then that operation is under charged and is compensated by excess potential available in the data structure.

Let c_i be the actual cost of the ith operation and $\hat{c}_i$ be the amortized cost of the ith operation. Then the ith operation has some positive credit if $\hat{c}_i > c_i$ and the credits $\hat{c}_i - c_i$ will be used for future operations. Now the total available credits will be positive if

$$\sum_{i=0}^{n} \hat{c}_i \geq \sum_{i=0}^{n} c_i \dots\dots (1)$$

In this method, a function maps a data structure onto a real valued non-negative number. The amortized cost is given by the actual cost added to the increase in potential due to that operation.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \quad \dots\dots (2)$$

from (1) and (2), we have

$$\sum_{i=0}^{n} \hat{c}_i \geq \sum_{i=0}^{n}(c_i + \Phi_i - \Phi_{i-1}) \dots\dots (3)$$

$$\sum_{i=0}^{n} \hat{c}_i \geq (\sum_{i=0}^{n} c_i) + \Phi_n - \Phi_0 \dots\dots (4)$$

5.5 CASE STUDIES

We will now look into two case studies, stack and binary counter, for their amortized analysis using the above discussed methods.

5.5.1 Stack

Stack is a data structure with the property of Last In First Out (LIFO). It works mainly on 2 main operations: Push(x) and Pop() where x is an element to be pushed onto the stack. Each of this 2 operations takes O(1) time respectively. Let us consider the cost of each operation to be 1. Then the total cost of a sequence of n Push(x) and Pop() operations is therefore n, and the actual running time for n operations is therefore $\theta(n)$.

Push(x)
        1. top[STACK] = top[STACK] + 1
        2. STACK [top[STACK]] = x

Pop()
        1. if top = -1
        2.   then "underflow"
        3. else
        4.   top[STACK] = top[STACK] − 1
        5. return STACK [top[STACK] + 1]

Multipop(k)
        1. While top != -1 and $k \neq 0$
        2.   Pop()
        3.   k = k − 1

Now, let us add another stack operation, Multipop(k) which removes the k elements from the top of a stack if it contains at least k elements, or it pops the entire stack if it contains fewer than k elements. The worst-case cost of a Multipop(k) operation in the sequence of n operations is O(n), since the stack size is at most n. We will now look at all three operations from the perspective of amortized analysis using all the above three methods. Let us assume that there are n such operations being performed using Push(x), Pop() and Multipop(k) in some order. The actual cost for the worst case running time of Push(x) and Pop() is O(1) and for Multipop(k) is O(n) since the stack contains at most n elements.

5.5.1.1 Aggregate analysis method

Assuming that in a sequence of n operations, we perform a sequence of n Push(x), Pop() and Multipop(k) operations in any order on an initially empty stack. So any sequence of n Push(x), Pop() and Multipop(k) operations will cost O(n) as each element can be popped at most once when it is pushed onto the stack. The number of times Pop() operation is called will be equal to the number of times Push(x) operation is called. Now, for a sequence of n Push(x), Pop() and Multipop(k) operations will take O(n) time and the average cost of an operation is O(n) / n = O(1). The amortized cost of each operation is equal to the average cost in aggregate analysis. Therefore, all three operations of Push(x), Pop() and Multipop(k) has an amortized cost of O(1).

5.5.1.2 Accounting method

In the Accounting method, credits will be assigned to the elements of the stack. '2' credits will charged for the Push(x) operation which involves inserting an element x into the stack. Out of the 2 credits, 1 credit will be used for the Push(x) operation and the remaining 1 credit will be stored at x which can be used later. We charge nothing for the Pop() operation i.e. this operation is free as we use the extra credit available for each element, x for performing this operation. So the actual cost of this operation is 1 credit which is used from the credit available for each element in the stack and hence we charge nothing for this operation for our analysis. As Pop() operation is always performed on a non empty stack , our account balance will never be in debit. Similarly for Multipop(k) operations, the overall charge for the operation will be nothing as there will always be sufficient credits available for each element, x in the stack for the k successive pop operations. Therefore, the amortized cost of Push(x) is 2 = O(1), Pop() is 0 = O(1) and Multipop(k) is 0 = O(1).

5.5.1.3 Potential function method

In this method, we need to define a potential function capturing some structural element. We can take the number of elements inside the stack as a potential function and analyze all the 3 operations viz. Push(x), Pop() and Multipop(k).

Push(x)

$$\hat{c}_{push} = c_{push} + \Phi_i + \Phi_{i-1}$$
$$= 1 + x + 1 - x, \text{ where } x = \text{Number of elements in S}$$

before push operation.

$$= 2$$

Pop()

$$\hat{c}_{push} = c_{push} + \Phi_i + \Phi_{i-1}$$
$$= 1 + x - 1 - x$$
$$= 0$$

Multipop(k)

$$\hat{c}_{push} = c_{push} + \Phi_i + \Phi_{i-1}$$
$$= k + n - k - n, \text{ where } n = |S|$$
$$= 0$$

Therefore, the amortized cost of Push(x) is 2 = O(1), Pop() is 0 = O(1) and Multipop(k) is 0 = O(1).

STOP TO CONSIDER

We can define another operation for the stack, Multipush(k) which pushes k elements into the stack. We can also analyze the cost of this operation using all the three techniques. Using aggregate analysis, the total cost is O(n) and the amortized cost is O(n) / O(1) which is equal to O(n). With accounting method, we use '2' credits for push and '0' for pop. So, the amortized cost is 2 * O(n) / O(1) which is O(n). For the potential function method, we give the amortized cost using

$$\hat{c}_{push} = c_{push} + \Phi_i + \Phi_{i-1}$$
$$= k + n + k - n, \text{ where } n = |S|$$
$$= 2k = O(n)$$

Thus, the amortized cost for Multipush(k) is O(n).

5.5.2 BINARY COUNTER

We will now look into a binary counter with k bits, initially all k bits set to 0's. The basic operations defined on the binary counter are increment, decrement and reset. Incrementing a counter adds a bit '1' to the current value. Decrementing a counter subtracts a bit '1' from the current value whereas resetting a counter makes all k bits to 0's. For example, if the current value of the counter is '0010' then on increment the new value is '0011' and on decrement the initial value of '0010', the value becomes '0001'. Whereas the reset operation makes the current value of the counter to all 0's. We will now analyze the amortized costs of the following operations:

1. A sequence of n increment operations.

2. A sequence of n increment and decrement operations.

3. A sequence of n increment, decrement and reset operations.

COUNTER VALUE	A [3]	A [2]	A [1]	A [1]	TOTAL COST
0	0	0	0	*0*	0
1	0	0	*0*	*1*	1
2	0	0	1	*0*	3
3	0	*0*	*1*	*1*	4
4	0	1	0	*0*	7
5	0	1	*0*	*1*	8
6	0	1	1	*0*	10
7	*0*	*1*	*1*	*1*	11
8	1	0	0	0	15

Fig 1: A 4 - bit binary counter with value goes from 0 to 8 by a sequence of 8 increment operations. Bits that flip to achieve the next value are underlined. The running cost for flipping bits is shown on the right as total cost.

5.5.2.1 A sequence of n increment operations

Aggregate Analysis : For aggregate analysis method, let us consider a binary counter with k bits and not all bits in the binary counter flip in each increment / decrement operation. Let us consider the 1st operation of a sequence of n increment operations in which the trivial analysis of the binary counter in the figure gives O(k) for each increment operation and for n such increment operations, the total cost will be O(nk) and the average cost will be O(k).

The 0th bit (rightmost bit , LSB) flips in each increment and there are n flips . The 1st bit is flipped alternatively and thus there are $\frac{n}{2}$ flips in total. So, the ith bit is flipped $\frac{n}{2^i}$ times. The total number of flips in a sequence of n increments will be :

$$\sum_{i=0}^{\lfloor \log n \rfloor} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

The worst case time complexity for a sequence of n increments is O(n) when all the k bits of the binary counter is initially set to all 0's.

The average cost / amortized cost of each operation is therefore O(n) / n = O(1).

Accounting method: In the accounting method, credits will be assigned for flipping the bits (0 to 1). '2' credits will assigned for each flip from 0 to 1 where 1 credit is used for the actual flip and remaining 1 credit is stored with the bit. It is done free when the bit is flipped from 1 to 0 in subsequent increments. The 1 credit stored with the bit is used to pay for this operation and hence it is termed as free. The number of 1's is accumulated as credit in the counter at the end of each increment. The cost of flipping the bit (0 to 1) is 2 = O(1) and for flipping the bit (1 to 0) is 0 = O(1). Therefore, the amortized cost of n increments is O(1).

Potential Function Method: In this method, we define the number of 1's in the counter as a potential function capturing some structural element. For the ith iteration, the last 0 is set to 1 and after the last 0 all 1's are set to 0. E.g. when a counter with value = '1001' is incremented then the new value is '1010'. Let us denote the total no of 1's before the ith operation be x and the total no of 1's after the last 0 be t . After the end of ith operation, t no of 1's are changed to 0 and the last 0 is changed to 1 so there will be x − t + 1 no of 1's. Thus, the actual increment cost is t + 1 and the amortized cost is

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + t + (x - t + 1) - x \\
&= 2 = O(1)
\end{aligned}
$$

5.5.2.2 A Sequence of N Increment and Decrement Operations

Aggregate Analysis : For the 2nd operation of a sequence of n increment and decrement operations, we take a sequence of $\frac{n}{2}$ increments followed by $\frac{n}{2}$ increments and decrements occurring alternately. Using aggregate analysis, for the 1st half of the operations the actual cost is O(n) and for the 2nd half the actual cost is $\frac{n}{2}$ * O(k) . Thus, the amortized cost is O(k) for both increment and decrement operations.

Accounting method : For the accounting method, k credits will be assigned to both increment and decrement operations. Therefore, the amortized cost is O(k) for both increment and decrement operations.

Potential Function Method : We define the number of 1's in the counter as a potential function capturing some structural element. Let us denote the total no of 1's before the ith operation be x and the total no of 1's after the last 0 be t . So, the amortized cost of increment is

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + t + (x - t + 1) - x$$
$$= 2 = O(1)$$

Let us denote the total no of 1's before the ith operation be x and the total no of 0's after the last 1 be t . So, the amortized cost of decrement is

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + t + (x + t - 1) - x$$
$$= 2t = O(k)$$

Therefore, the amortized cost of increment is O(1) and that of decrement is O(k).

5.5.2.3 A Sequence of N Increment, Decrement and Reset Operations

Aggregate Analysis : For the 3rd operation of a sequence of n increment , decrement and reset operations, we take a sequence of $\frac{n}{2}$ increments followed by one reset operation. The total cost would be O(n) + O(k). The aggregate analysis method gives O(1) amortized cost for $\frac{n}{2}$ increments and O(k) amortized cost for the reset operation. Therefore, the amortized cost is O(k) for this sequence of operations.

Accounting method : For the accounting method, k credits will be assigned to both increment and decrement operations. Reset operations requires flipping back all the 1's present in the current value of the counter back to 0's. It's a decrement operation done with k credits and so the actual cost of reset operation is O(k). Therefore, the amortized cost is O(k) for increment , decrement and reset operations.

Potential Function Method : For the potential function method, the analysis of increment and decrement is same as done previously and thus is O(1) amortized and O(k) amortized respectively. We now analyze the reset operation and the actual cost of reset operation is

O(k). Assuming that there are x 1's in the current value of the counter and so the amortized cost is

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= k + 0 - x$$
$$= O(k) \text{ , if } x = O(1)$$

5.6 SUMMING UP

- Amortized analysis is a worst-case analysis of a sequence of operations to obtain a tight bound on the overall cost per operation in the sequence.

- Amortized analysis is applied when few operations of the algorithms are slow while rest of them are fast.

- Amortized analysis does not involve probability calculations and does not give the cost of a specific operation in the sequence.

- There are 3 different approaches to perform amortized analysis of an algorithm viz. aggregate analysis, accounting method and potential function method.

- Aggregate analysis is the simplest and gives the upper bound on the total cost of a sequence of n operations.

- Accounting method gives different costs to different types of operations and the cost associated with each operation is its amortized cost. This method involves maintaining credits for each operation.

- Potential function method performs analysis as "potential energy" or "potential" which can be used to pay for future operations. The analysis is based on a structural feature of the data structure and this method also assigns different costs to different types of operations.

- Case studies are performed on stack and binary counter to perform amortized analysis using the 3 different methods.

5.7 ANSWERS TO CHECK YOUR PROGRESS

1. a. FALSE

1. b. TRUE

1. c. TRUE

1. d. FALSE

1. e. TRUE

5.8 POSSIBLE QUESTIONS

1. What is amortized analysis ?

2. Why amortized analysis of an algorithm is needed ?

3. Define the features of amortized analysis ?

4. What are the different approaches to perform amortized analysis ?

5. Which technique/techniques of amortized analysis assigns different costs to

   different operations?

6. Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as $\theta(nk)$.

7. Perform amortized analysis on dynamic tables using any one of the methods.

5.9 SUGGESTED READINGS

- Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford. *Introduction to Algorithms*. 2nd Edition : The MIT Press, 2001.

BLOCK II:
ALGORITHM DESIGN TECHNIQUES

UNIT 1: ALGORITHM DESIGN TECHNIQUES I

Unit Structure:

1.1 INTRODUCTION

The term algorithm comes from the name of a Persian author, **Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A. D.)**, who wrote a textbook of mathematics. This word has taken on a special significance in computer science, where the term "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem. This makes algorithm is different from the terms such as process, technique or method [3].

Algorithm is a set of instructions to solve a particular problem. Mainly algorithm contains a sequence of computational steps that transform the input into the output. We can think an algorithm like a tool for solving a well specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for obtaining that input/ output relationship.

All algorithms must satisfy the following criteria [1]:

- **Input:** there may be zero or more externally supplied quantities

- **Output:** at least one quantity or result is produced

- **Definiteness:** each instruction must be clear and unambiguous;

- **Finiteness:** if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

- **Effectiveness:** every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

No matter which programming language we use, but at the same time it is important to learn algorithm design techniques in data structures in order to be able to build scalable systems in an efficient manner. Selecting a proper design technique for algorithms is a complex but important task. We can choose from a wide range of algorithm design technique. Following are some of the main algorithm design techniques:

1. Divide and Conquer

2. Greedy Algorithms

3. Dynamic Programming

4. Backtracking

5. Branch and Bound Algorithm

1.2 DIVIDE AND CONQUER

The advance of scientific knowledge often involves the grouping together of similar objects followed by the abstraction and representation of their common structural and functional features. Generic properties of the objects in the class are then studied by reasoning about this abstract characterization. The resulting theory may suggest strategies for designing objects in the class which have given characteristics. One such related algorithm based on the principle of 'divide and conquer'.

1.2.1 General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements [1].

Divide and Conquer is one of the best-known general algorithm design technique. It works according to the following general plan:

- Given a function to compute on n input the divide-and-conquer strategy suggest splitting the input into k distinct subsets, 1<k<=n, yielding k subproblems.

- These subproblems must be solved and then a method must be found to combine subsolution into a solution of the whole.

- If the subproblems are still relatively large, then the divide-and–conquer strategy can possibly be reapplied.

- Often the subproblems resulting from a divide-and –conquer design are of the same type as the original problem.

- For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

**Divide and conquer strategy is as follows:**

- Divide the problem instance into two or more smaller instances of the same problem

- Solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance

- The recursion stops when an instance is reached which is too small to divide

- When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified

The Divide and conquer algorithm involves three steps at each level of recursion:

Steps	Example: Divide-and-Conquer (Input: Problem P) To Solve P:
Divide: Divide the problem into a number of sub problems. The sub problems are solved recursively.	Divide P into smaller problems P1, P2, P3.....Pk
Conquer: Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.	Conquer by solving the (smaller) subproblems recursively
Combine: Combine the solutions to the subproblems into the solution for the original problem.	Combine solutions to P1, P2, ...Pk into solution for P

A typical case with k=2is diagrammatically shown below:
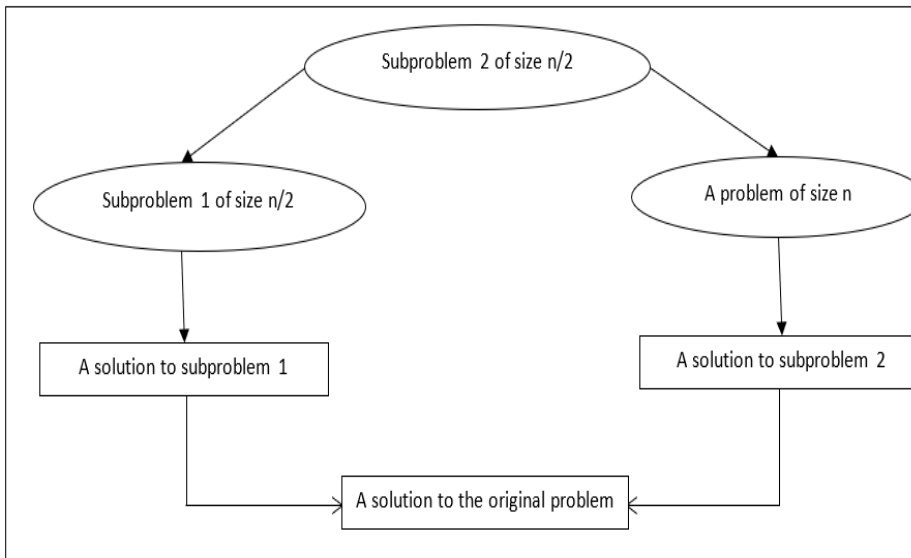
Control Abstraction for divide and conquer:

Algorithm DAndC(P)

{

      if Small(P) then return S(P)

      else

      {

          divide P into smaller instances $P_1, P_2, \ldots, P_k$, k>=1;

          Apply DAndC to each of these subproblems;

          Return

          Combine(DAndC(P_1),DAndC(P_2),....,DAndC(P_k));

      }

}

In the above specification,

- Initially **DAndC(P)** is invoked, where 'P' is the problem to be solved.

- **Small (P)** is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this so, the function '**S**' is invoked. Otherwise, the problem P is divided into smaller sub problems.

These sub problems P1, P2 ……Pk are solved by recursive application of **DAndC**.

- **Combine** is a function that determines the solution to P using the solutions to the 'k' sub problems.

1.2.2 Recurrence equation for Divide and Conquer

If the size of problem 'p' is n and the sizes of the 'k' sub problems are n_1, n_2….n_k, respectively, then the computing time of divide and conquer is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) \mid T(n_2) + \cdots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

Where,

- T(n) is the time for divide and conquer method on any input of size n and

- g(n) is the time to compute answer directly for small inputs.

- The function f(n) is the time for dividing the problem 'p' and combining the solutions to sub problems.

For divide and conquer based algorithms that produce sub problems of the same type as the original problem, it is very natural to first describe them by using recursion

More generally, an instance of size **n** can be divided into **b** instances of size **n/b**, with **a** of them needing to be solved. (Here, a and b are constants; **a>=1 and b > 1**.). Assuming that size **n** is a power of **b** (i.e. **n=b$_k$**), to simplify our analysis, we get the following recurrence for the running time T(n):

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \qquad \text{--------------(1)}$$

where, f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Analyzing divide-and-conquer algorithms:

We often use a recurrence to express the running time of a divide-and-conquer algorithm.

Let T(n) = running time on a problem of size n.

- If n is small (say n ≤ k), use constant-time brute force solution.

- Otherwise, we divide the problem into a subproblems, each 1/b the size of the original.

- Let the time to divide a size-n problem be D(n).

- Let the time to combine solutions (back to that of size n) be C(n).

We get the recurrence

$$
T(n) = \begin{cases} c & \text{if } n \leq k \\ a\, T(n/b) + D(n) + C(n) & \text{if } n > k \end{cases}
$$

Example: Merge Sort
For simplicity, assume n = 2k.
For n = 1, the running time is a constant c
For n ≥ 2, the time taken for each step is:

Divide: Compute q = (p + r)/2; so, D(n) = θ (1).

Conquer: Recursively solve 2 subproblems, each of size n/2; so, 2T(n/2)

Combine: MERGE two arrays of size n; so, C(n) = θ (n)

More precisely, the recurrence for MERGE-SORT is

$$
T(n) = \begin{cases} c & \text{if } n \leq 1 \\ 2\, T(n/2) + f(n) & \text{if } n > 1 \end{cases}
$$

where the function f(n) is bounded as d' n $\leq$ f(n) $\leq$ d n for suitable constants d, d'> 0

## 1.2.2.1 Guess-and –test (Substitution Method)

Guess an expression for the solution. The expression can contain constants that will be determined later. Use induction to find the constants and show that the solution works.

<div align="center">Or</div>

[**Substitution Method:** This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappears.]

Let us apply this method to MERGE-SORT

The recurrence of MERGE-SORT implies that there exist two constants c, d > 0 such that

$$T(n) \leq \begin{cases} c & \text{if } n = 1 \\ \\ 2\,T(n/2) + dn & \text{if } n > 1 \end{cases}$$

Guess: There is some constant a > 0 such that T(n) $\leq$ an lg n for all n $\geq$ 2 that are powers of 2

Solving the MERGE-SORT recurrence by guess-and-test

Test: For n=2^k , by induction on k
Base case: k=1

T(2) = 2c + 2d $\leq$ a 2 lg 2   if a $\geq$ c + d

Inductive step: assume T(n) $\leq$ an log n for n = 2k.
Then, for n' = 2k+1 we have:

$$T(n') \leq 2a\frac{n'}{2}\lg\frac{n'}{2} + dn'$$
$$= an'lgn' - an'\,lg\,2 + dn'$$
$$\leq an'lg\,n' \qquad if\ a \geq d$$

In summary: choosing a ≥ c + d ensures T(n) ≤ an lg n, and thus T(n) = O(n log n).

A similar argument can be used to show that $T(n) = \Omega\, n \log n$

Hence, T(n) = θ (n log n)

1.2.2.2 The Recursion Tree

Guess-and-test is great, but sometime it is difficult to guess. One way is to use the recursion tree, which exposes successive unfoldings of the recurrence. The idea is well exemplified in the case of MERGE-SORT. The recurrence is

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2\,T(n/2) + f(n) & \text{if } n > 1 \end{cases}$$

where the function f(n) satisfies the bounds $d'\,n \leq f(n) \leq d\,n$, for suitable constants d, d' > 0

Unfolding the recurrence of MERGE-SORT

Assume n = 2k for simplicity

First unfolding: cost of f(n) plus cost of two subproblems of size n/2
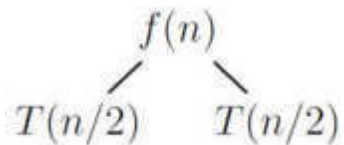
Second unfolding: for each size-n/2 subproblem, cost of f(n/2) plus cost of two subproblems of size n/4 each
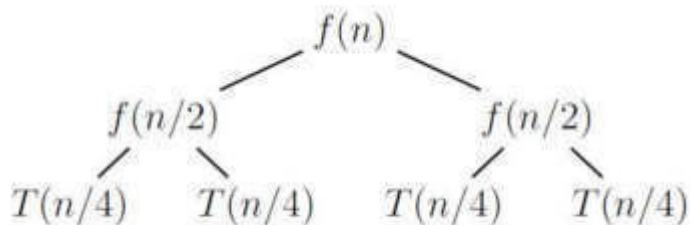
Continue unfolding, until the problem size (= node label) gets down to 1:
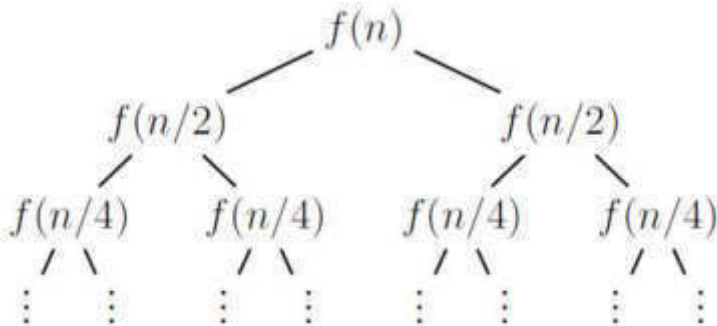
In total, there are $\lg n + 1$ levels

- Level 0 (root) has cost $C_0(n) = f(n)$
- Level 1 has cost $C_1(n) = 2f\left(\frac{n}{2}\right)$
- Level 2 has cost $C_2(n) = 4f\left(\frac{n}{4}\right)$
- For $l < \lg n$, level l has cost $c_2(n) = 2^l f\left(\frac{n}{2^l}\right)$
  Note that, since $d'n \leq f(n) \leq d\,n$, we have $d'n \leq C_l(n) \leq dn$
- The last level (consisting of n leaves) has cost **cn**

Analysing MERGE-SORT with the recursion tree

The total cost of the algorithm is the sum of the costs of all levels:

$$T(n) = \sum_{l=0}^{\lg n - 1} C_l(n) + c\,n$$

Using the relation $d'n \leq C_l(n) \leq dn$ for $< \lg n$, we obtain the bounds

$$d'n \lg n + c\,n \leq T(n) \leq dn \lg n + c\,n$$

Hence, $T(n) = \theta(n \log n)$

1.2.2.3 The Master Theorem

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the master theorem. It states that, in recurrence equation $T(n) = aT(n/b) + f(n)$, If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too.

Problems on Substitution method and Master theorem to solve the recurrence relation

Exercise 1: Solve following recurrence relation

T(n)=2T(n/2)+n, T(1)=2, using substitution method

Solution: T(n)= 2T(n/2)+n

=2[2.T(n/4)+n/2] +n

=4T(n/4)+2n

=4[2.T(n/8)+n/4]+2n= 8T (n/8)+3n

.

.

.

=2^i T(n/2^i)+ ¡n, $1 \leq i \leq \log_2 n$

The maximum value of i=log₂n [then only we get T(1)]

$= 2 \log_2 n . T\left(\dfrac{n}{2 \log_2 n}\right) + n. \log_2 n$

=n.T(1)+ $n. \log_2 n$

=$\theta\big(n\, (\log_2 n)\big)$

Solution using Master Theorem

Here, a=2, b=2, f(n)=n= $\theta n'$ d=1

Also we see that $a=b^d$ [2=2']

As per case 2 of master theorem

$$T(n)= \theta\left(n^d \; (\log_2 n)\right)$$

$$= \theta\left(n \; (\log_2 n)\right)$$

1.2.3 Advantages and Disadvantages of Divide and Conquer

Advantages

- ❖ **Parallelism:** Divide and conquer algorithms tend to have a lot of inherent parallelism. Once the division phase is complete, the sub-problems are usually independent and can therefore be solved in parallel. This approach typically generates more enough concurrency to keep the machine busy and can be adapted for execution in multi-processor machines.

- ❖ **Cache Performance:** divide and conquer algorithms also tend to have good cache performance. Once a sub-problem fits in the cache, the standard recursive solution reuses the cached data until the sub-problem has been completely solved.

- ❖ It allows solving difficult and often impossible looking problems like the Tower of Hanoi. It reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.

- ❖ Another advantage to this paradigm is that it often plays a part in finding other efficient algorithms, and in fact it was the central role in finding the quick sort and merge sort algorithms.

Disadvantages

- ❖ One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.

❖ Another concern with it is the fact that sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these group recursively, and then add the sums of the two groups together.

❖ Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times. It is solved again. In cases like these, it can often be easier to identify and save the solution to the repeated sub problem, which is commonly referred to as memorization.

1.3   GREEDY ALGORITHM

We consider problems in which a result comprises a sequence of steps or choices that have to be made to achieve the optimal solution. Greedy programming is a method by which a solution is determined based on making the locally optimal choice at any given moment. In other words, we choose the best decision from the viewpoint of the current stage of the solution. Depending on the problem, the greedy method of solving a task may or may not be the best approach. If it is not the best approach, then it often returns a result which is approximately correct but suboptimal. In such cases dynamic programming or brute-force can be the optimal approach. On the other hand, if it works correctly, its running time is usually faster than those of dynamic programming or brute-force.

A Greedy algorithm is characterized by the following two properties:

1. The algorithm works in stages, and during each stage a choice is made which is locally optimal;
2. The sum totality of all the locally optimal choices produces a globally optimal solution.

1.3.1 General Method

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function [1].

If a greedy procedure does not always lead to a globally optimal solution, then we will refer to it as a heuristic, or a greedy heuristic. Heuristics often provide a "short cut" to a solution, but not necessarily to an optimal solution. Hence fore, we will use the term "algorithm" for a method that always produces a correct/optimal solution and "heuristic" to describe a procedure that may not always produce the correct or optimal solution.

Let us consider the problem of coin change. Suppose a greedy person has some 25p, 20p, 10p, 5paise coins. When someone asks him for some change then he wants to give the change with minimum number of coins. Now, let someone requests for a change of top then he first selects 25p. Then the remaining amount is 45p. Next, he selects the largest coin that is less than or equal to 45p i.e. 25p. The remaining 20p is paid by selecting a 20p coin. So the demand for top is paid by giving total 3 numbers of coins. This solution is an optimal solution. Now, let someone requests for a change of 40p then the Greedy approach first selects 25p coin, then a 10p coin and finally a 5p coin. However, the some could be paid with two 20p coins. So it is clear from this example that Greedy approach tries to find the optimal solution by selecting the elements one by one that are locally optimal. But Greedy method never gives the guarantee to find the optimal solution.

The choice of each step is a greedy approach is done based on the following:

· It must be feasible

· It must be locally optimal

· It must be unalterable

There are two key ingredients in greedy algorithm that will solve a particular optimization problem.

1. Greedy choice property
2. Optimal substructure

1. *Greedy choice property*:

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, when a choice is to be made, without considering results from the sub-problems, it looks for best choice in the current problem. In this algorithm choice is made that seems best at the moment and solve the sub-problems after the choice is made. The choices made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choice or solution to the sub-problems. The algorithm works in a top down manner, making one greedy choice one after another, reducing each given problem instances into smaller one.

2. *Optimal substructure:*

A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solution to its sub-problem. The optimal substructure varies across problem domain in two ways-

i) How many sub-problems are used in an optimal solution to the original problem.

ii) How many choices we have in determining which sub-problem to use in an optimal solution.

In Greedy algorithm a sub-problem is created by having made the greedy choice in the original problem. Here, an optimal solution to the sub-problem, combined with the greedy choice already made, yield an optimal solution to the original problem.

1.3.2 Optimal Merge Patterns

Optimal merge patterns can be stated as follows:

- Two sorted file containing n and m records respectively could be merged together to obtain one sorted file in time O(n + m). When more than two sorted files are merged together then merge can be done by repeatedly merging the sorted files in pairs.

For example-

Problem 1: There are 5 sorted files F1,F2,F3,F4,F5 and each file has 20,30,10,5,30 records respectively.

If merge these files pair wise then-

$$M1 = F1\&F2$$

$$= 20 + 30$$

$$= 50 \text{ (i.e merging F1 and F2 requires 50 moves)}$$

$$M2 = M1\&F3$$

$$= 50 + 10$$

$$= 60$$

$$M3 = M2\&F4$$

$$= 60 + 5$$

$$= 65$$

$$M4 = M3\&F5$$

$$= 65 + 30$$

$$= 95$$

Hence Total time required to moves records is –

$$50+60+65+95 = 270$$

- Different pairing requires different amount of computing time. The problem can be stated as-

What is the optimal way to pair wise merge n sorted files? Or What is the minimum time needed to pair wise merge n sorted files?

We can solved this problem using greedy algorithm. The greedy algorithm attempt to find an optimal merge pattern.

Sorts the list of file and at each step merge the two smallest size files together.

Example: The above given problem 1 can be solved as follows-

Sort the files according to their number of records.

$$(5,10,20,30,30) = (F4,F3,F1,F2,F5)$$

Merge the first two files-

$$(5,10,20,30,30) => (15,20,30,30)$$

Merge the next two files-

$$(15,20,30,30) => (30,30,35)$$

Merge the next two files-

$$(30,30,35) => (35,60)$$

Merge the last two files-

$$(35,60) => (95)$$

Hence, total time require is 15+35+60+95= 205

This is the optimal merge pattern for the given problem instance. This merging is also called two way merge pattern because each merge step involves merging of two files.

The two way merge pattern can be represented by binary merge trees. For the above problem1 the binary merge tree representing the optimal merge pattern is as follows-

Here, the leaf nodes are denoted by square and represent the five files. These nodes are called external nodes. The remaining nodes are drawn as circle and are called internal nodes. Each internal node has exactly two children and it represent file obtained by merging the files represented by its two children. The number in the each node is the length (i.e. the number of records) of the file represented by that record.
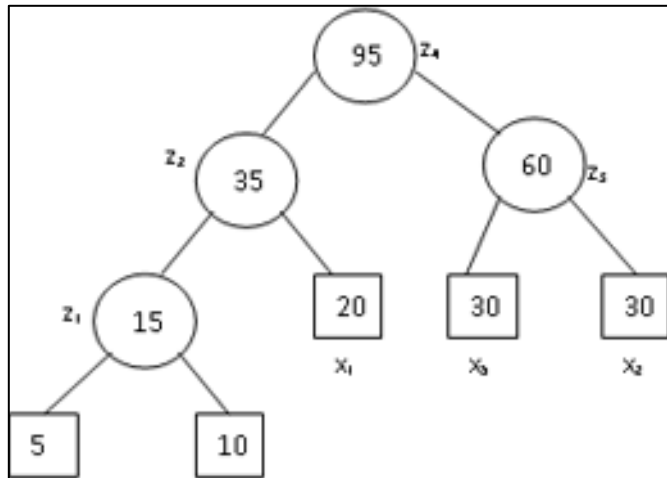
Fig 3.1 Binary merge tree representing a merge pattern

Here a node at level i is at a distance of $i - 1$ from the root (In the above tree x_4 is at a distance 3 from root z_4).

If d_i is the distance from the root to external node for a file x_i and q_i is the length of the file x_i , then the total number of records move for the binary merge tree is-

$$\sum_{i=1..n} d_i\, g_i$$

This sum is called the weighted external path length of the tree. An optimal two way merge pattern is minimum weighted external path length of a binary merge tree.

CHECK YOUR PROGRESS - I

a) _____ is a set of instructions to solve a particular problem.

b) Greedy programming is a method by which a solution is determined based on making the locally _____ choice at any given moment

c) Divide and conquer is a design strategy which is well known to breaking down _____ barriers

---

1.3.3 Minimum Spanning Tree

Before going to the definition of the minimum spanning tree let us define what a spanning tree is:

Spanning tree:

A spanning tree is a connected graph, say G = (V, E) with V as set of vertices and E as set of edges, is its connected acyclic sub- graph that contain all the vertices of the graph.

Now the minimum spanning tree can be defined as:

Minimum spanning tree:

A minimum spanning tree T of a positive weighted graph G is a minimum weighted spanning tree in which total weight of all edges are minimum

$$w(T) = \sum_{(u,v)\epsilon T} w(u, v) \; is \; minimized$$

Where w(u, v) is the cost of the edge (u, v)

For example;

Let us consider connected graph G given in fig
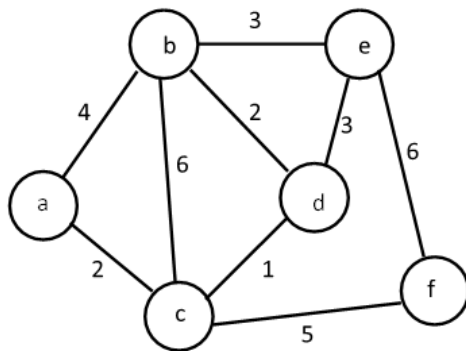
Fig. 3.2 A Connected graph G

Now, the minimum spanning trees are for the graph G is-
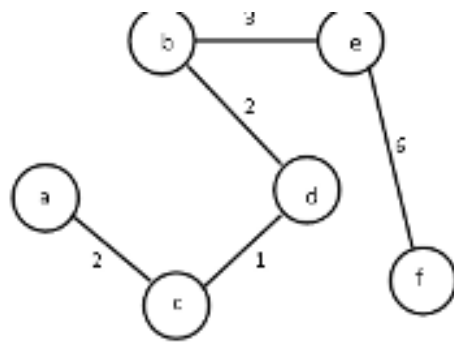
Fig. 3.3 A spanning tree the graph G

Application of Minimum spanning tree:

    i. In design of electric circuit network .

    ii. It is used in traveling salesman problem.

The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph

There are two algorithms to solve minimum spanning tree problem

    1.   Kruskal algorithm

    2.   Prim algorithm

The general approaches of these algorithms are-

    o The tree is built edge by edge

    o Let T be the set of edges selected so far

    o Each time a decision is made. Include an edge e to T s.t. Cost (T) + w (e) is minimized, and T U {e} does not create a cycle

Both these algorithms are greedy algorithm. Because at each step of an algorithm, one of the best possible choices must be made. The greedy strategy advocates making the choice that is best at the moment. Such a strategy is not generally guaranteed to globally optimal solution to a problem.

1.3.3.1 PRIM'S Algorithm

The prim's algorithm uses greedy method to build the sub-tree edge by edge to obtain a minimum cost spanning tree. The edge to include is

chosen according to some optimization criterion. Initially the tree is just a single vertex which is selected arbitrarily from the set V of vertices of a given graph G. Next edge is added to the tree by selecting the minimum weighted edge from the remaining edges and which does not form a cycle with the earlier selected edges. The tree is represented by a pair (V', E') where V' and E' represent set of vertices and set of edges of the sub-tree of minimum spanning tree.

The algorithm is as follows-

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it finds all vertices.

- *Input:* A non-empty connected weighted graph with vertices V and edges E (the weights are positive).

- Initialize: V' = {x}, where x is an arbitrary node (starting point) from V,

- E' = { }

- Repeat until V' = V;

- Choose an edge (u, v) with minimal weight such that u is in V' and v is not in V' (if there are multiple edges with the same weight, any of them may be picked)

- Add v to V' and (u, v) to E' if edge (u, v) will not make a cycle with the edges already in E'

- Output: V' and E' describe a minimal spanning tree

Example:

Let us consider the following graph G
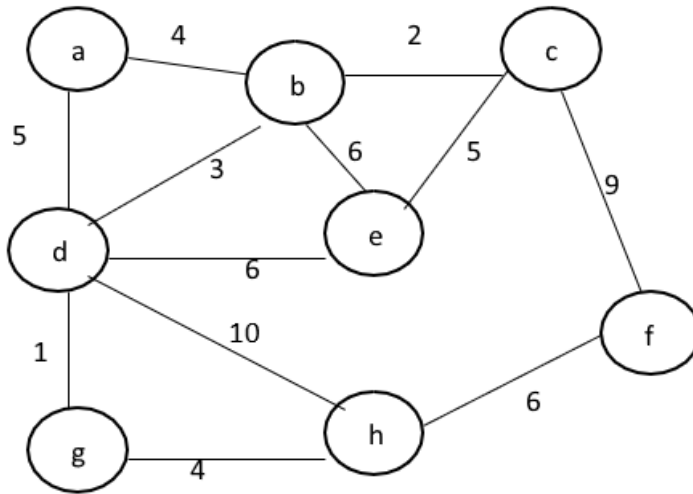
Fig. 3.4 Prim's algorithm applied on the Graph G

Initially vertex *a* is selected. So, V' will contain *a*.
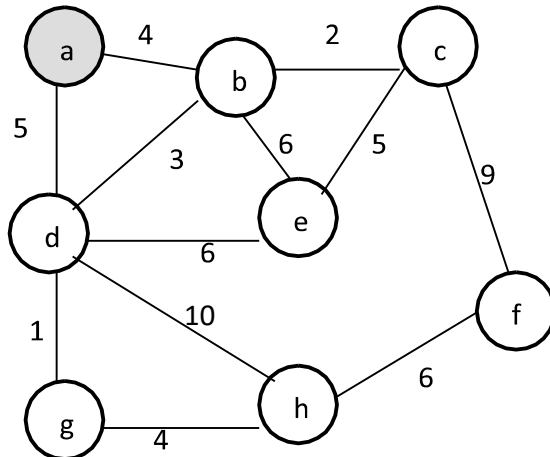
Fig. 3.5 Vertex a is selected

V' = {a}

E' = Ø

After first iteration, the minimum weight edge connected *a* and other vertices of V is selected. In this case from vertex *a* there are two edges *ab* and *ad* to vertex *b* and *d*.
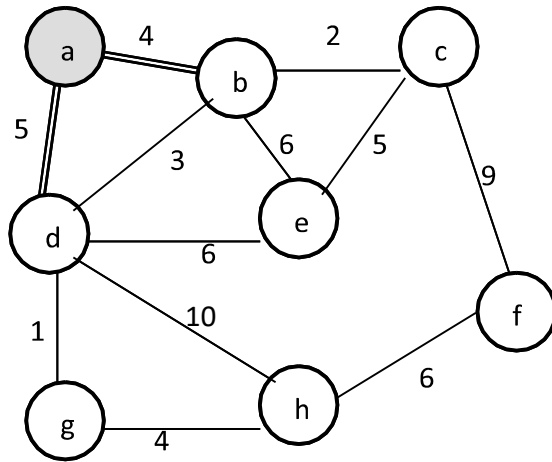
Fig. 3.6 minimum weighted edge

Between *ab* and *ad* weight of *ab* is minimum. Hence, after first iteration vertex *b* is include to V' and edge *ab* is included to E'.
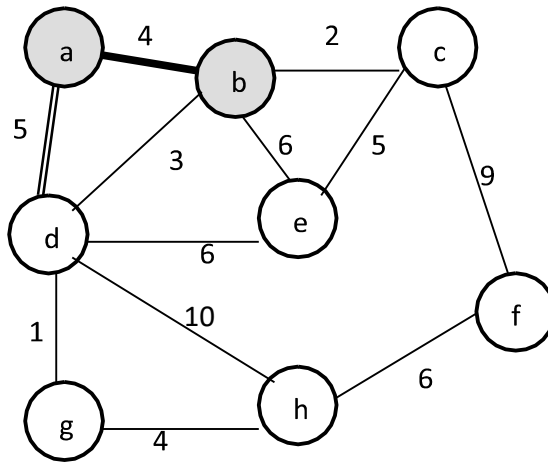
Fig. 3.7 Minimum weighted edge selected

V' = {a, b}

E' = { ab }

In the next iteration we select the minimum weight edge, which does not make a cycle with previously selected edges in E', from the edges not included in E' and edges connected   one vertex from V' and another vertex not in V'. Here edges from a and b to any other vertex.

Here, edges are ad, bd, be, bc from which we can select the minimum weight edge.
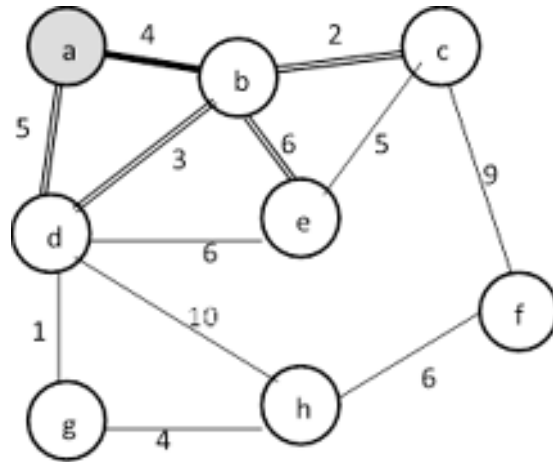
Fig. 3.8 Finds the minimum weighted edge

Here, weight of *bc* is minimum and it does not make a cycle with *ab*. Thus **bc** is selected in this iteration.
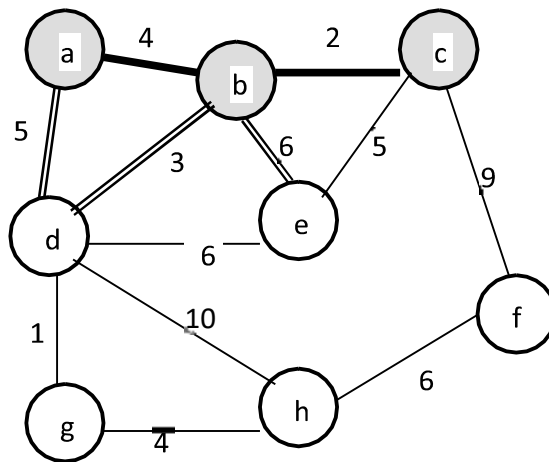
Fig. 3.9 Minimum weighted edge selected

V' = { a, b, c }

E' = { ab, bc }

In the next iteration we can consider the edges that have a,b or c as one of the vertex . Here the edges are ad, bd, be, ce, cf. we canot consider ab and bd because they are already selected.
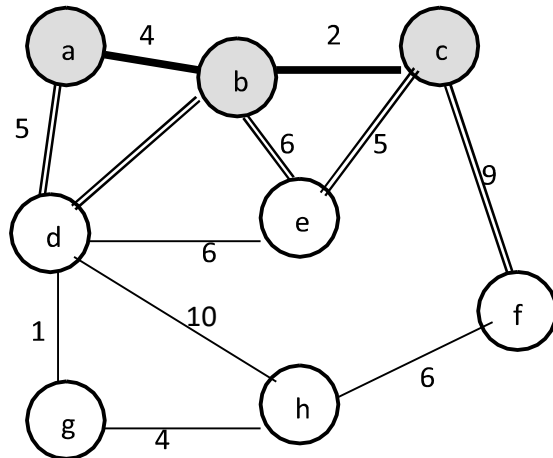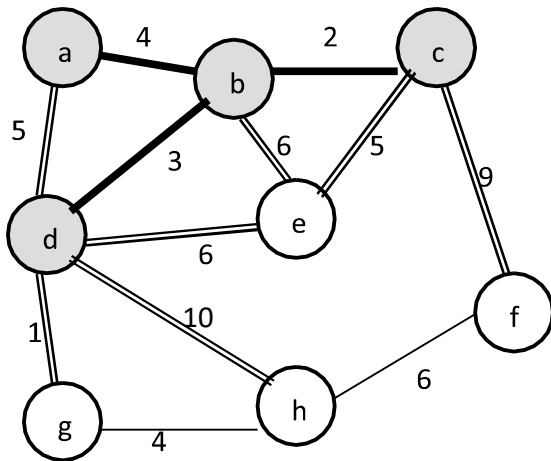
Fig. 3.10 Finds minimum weighted edge

From these edges weight of bd is minimum and it does not makea cycle with the edge in E'. Thus bd is selected.
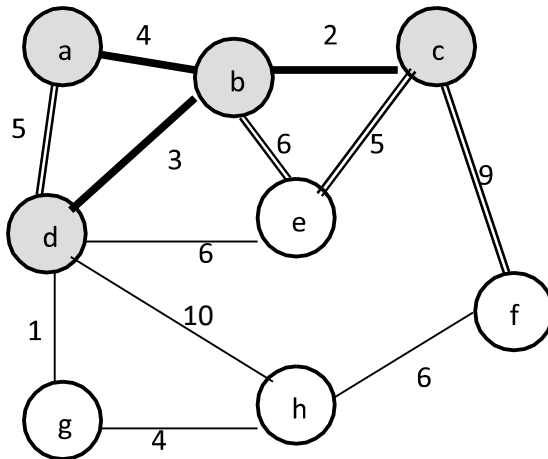
Fig. 3.11 Minimum weighted edge selected

V' = { a, b, c, d }

E' = { ab, bc, bd }

In the next iteration we consider the edges (excluding already selected edges) that have a, b, c, d as one vertex. Here edges are ad, be, ce, cf, de, dh, dg.

Fig. 3.12 Finds Minimum weighted edge

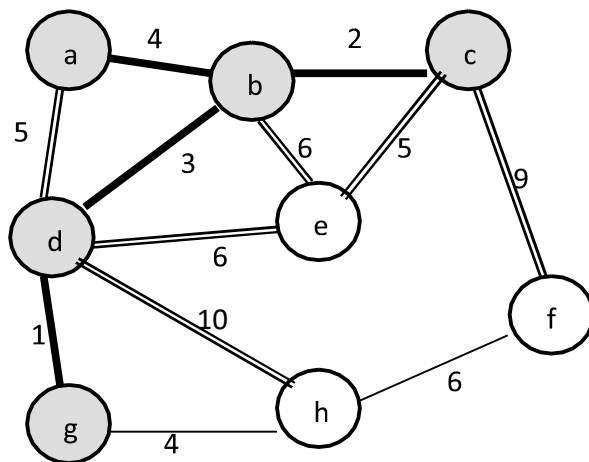The weight of dg is minimum and it does not make a cycle with the edges in E'. Thus, dg is selected.

Fig. 3.13 Minimum weighted edge selected

V' = { a, b, c, d, g }

E' = { ab, bc, bd, dg }

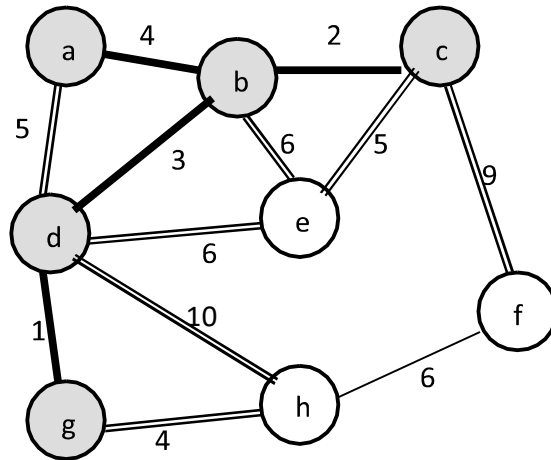In the next iteration considered edges are ad, be, de, gh, dh, ce, cf

Fig. 3.14 Finds Minimum weighted edge

Among these edges weight of gh is minimum and it does not make any cycle with already selected edges in E'. Thus, gh is selected.
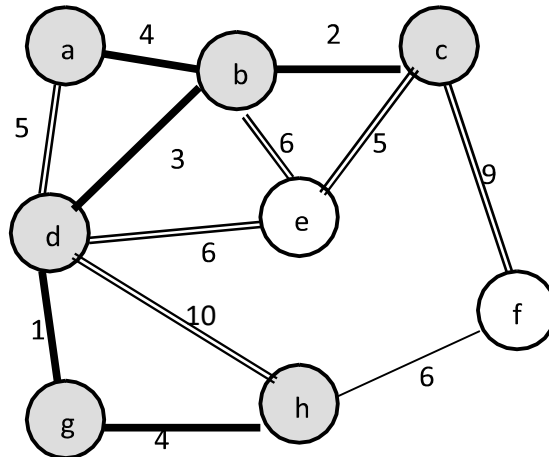

Fig. 3.15 Minimum weighted edge selected

V' = { a, b, c, d, g, h }

E' = { ab, bc, bd, dg, gh }

In the next iteration consider the edges that have one vertex from V' and connect another vertex excluding already selected edges. Here edges are ad, be, ce, cf, de, dh, hf.
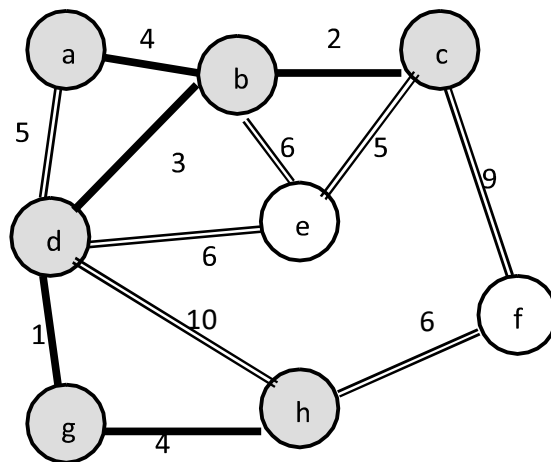
Fig. 3.16 Finds Minimum weighted edge

Among these weights, *ad* and *ce* are minimum. If select *ad* then it make a cycle with the already selected edge ab and bd of E'. So, ad cannot be selected. If we select ce it will not make a cycle with the edges of E'. Thus ce is selected.
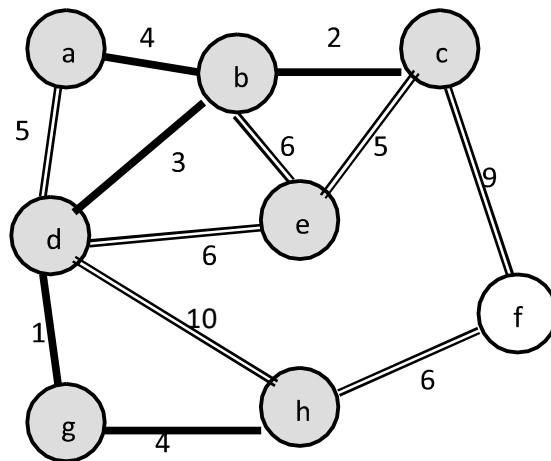

Fig. 3.17 Minimum weighted edge selected

V' = {a, b, c, d, e, g, h}

E' = {ab, bc, bd, dg, gh, ce}

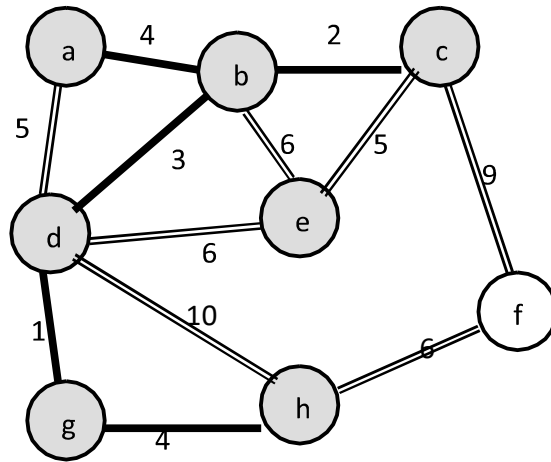In the next iteration considered edges are ad, be, de, dh, cf, hf.

Fig. 3.18 Finds Minimum weighted edge

Among these weights, ad is minimum. But it makes a cycle with already selected edges ad and ab. So, ad is rejected. Next minimum weight is of edge de, be and hf. But these two edges will also make cycle. So de and be are also rejected. hf will not make a cycle. Thus hf is considered.
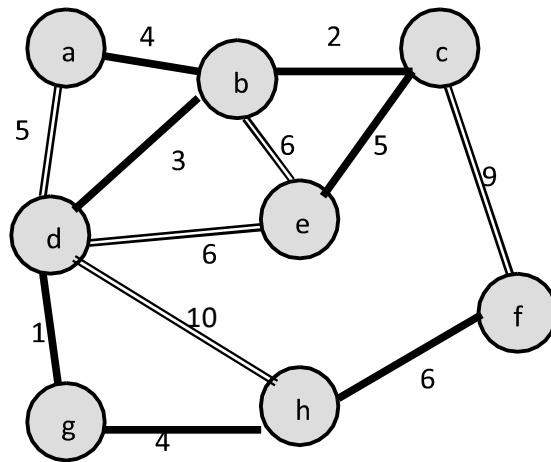
Fig. 3.19 Minimum weighted edge selected

V' = { a, b, c, d, e, f, g, h }

E' = { ab, bc, bd, dg, gh, ce, hf }

Next, the edges be, de, cf, ad, dh cannot included to form the tree because they make a cycle with already selected edges. Hence the final spanning tree is-
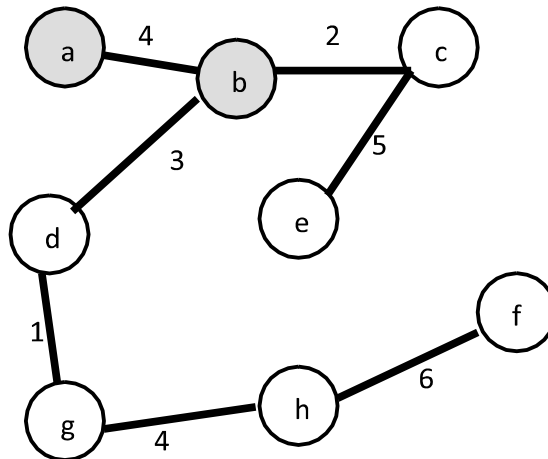
Fig. 3.20 Final spanning tree of graph G

1.3.3.2   KRUSKAL Algorithm

Another method of finding minimum spanning tree is Kruskal algorithm. In this algorithm the edges of the graph are considered in non-decreasing order. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

The algorithm is as follows-
- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty and F is not yet spanning
- remove an edge with minimum weight from S
- if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
- Otherwise discard that edge.

KRUSKAL ALGORITHM EXAMPLE:
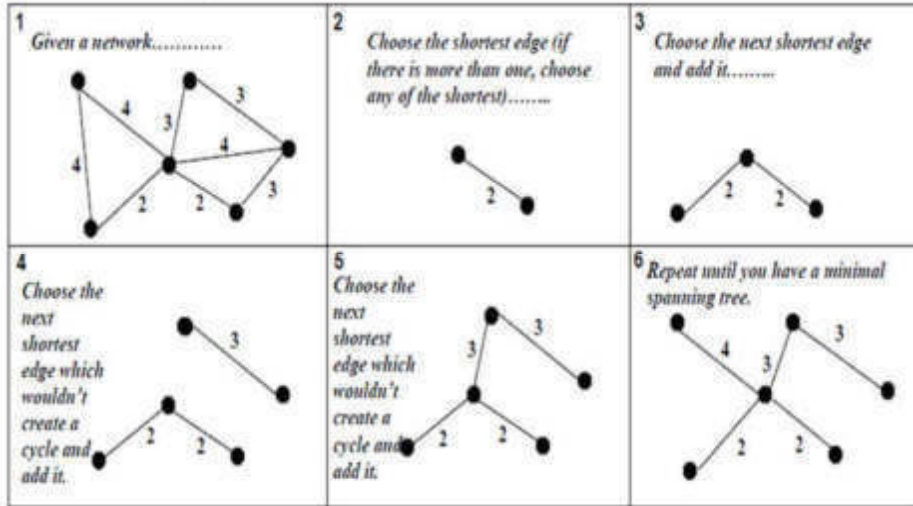
1.3.4 DIJKSTRA'S Algorithm

For a given weighted and directed graph G= (V, E), the shortest path problem is the problem of finding a shortest path between any two vertex v ∈ V in graph G. The property of the shortest path is such that a shortest path between two vertices contains other shortest path within it i.e any other sub-path of a shortest path is also a shortest path.

Single source shortest path problem:

In a single source shortest path problem, there is only one source vertex S in the vertex set V of graph G = (V, E). Now this single source shortest path problem finds out the shortest path from the source vertex S to any other vertex in v ∈ V.

Optimal substructure of a shortest path:

Optimal substructure of a shortest path can be stated that any other sub-path of a shortest path is also a shortest path. Here is the lemma-

Lemma:

Given a weighted directed graph G=(V, E) with weight function w: E → R, let p = (v_1, v_2 , v_k) be a shortest path from vertex v_1 to vertex v_k, and for any i and j such that $1 \leq i \leq j \leq k$, let $P_{ij} = (v_1, v_{i+1}, $ ------,v_j) be the

sub-path of P from vertex v_i to vertex v_j. Then P_{ij} is a shortest path from v_i to v_j.

Dijsktra algorithm solves the single source shortest path problem. But the algorithm works only on a directed and positive weighted graph. Positive weighted graph means where weights of all edges are non-negative i.e. $G = (V, E)$ is a positive weighted graph then $w(u, v) \geq 0$. Dijsktra algorithm is a greedy algorithm.

$$Dijkstra's\ Algorithm(G = (V, E), s)$$
$$D[s] \leftarrow 0$$
$$Parent[s] \leftarrow 0$$
$$V' \leftarrow \{s\}$$
$$\textbf{for}\ \ i \leftarrow 1\ \textbf{to}\ n - 1\ \textbf{do}$$
$$\text{find an edge } (u, v) \text{ such that } u \in V', v \notin V'$$
$$\text{and } D[u] + length[u, v] \text{ is minimum};$$
$$D[v] \leftarrow D[u] + length[u, v];$$
$$Parent[v] \leftarrow u;$$
$$V' \leftarrow V' \cup \{v\};$$
$$\textbf{endfor}$$

Example of Dijkstra's algorithm:

Apply Dijkstra's algorithm for the following graph G
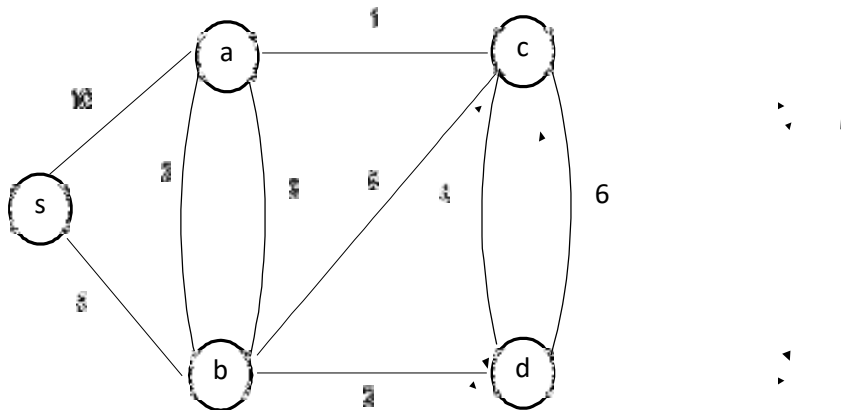
Space for learners:

Step 1

Step 2

Step 3

Step 4

Step 5

Step 6

Step 7

6

Step 8

Step 9

6
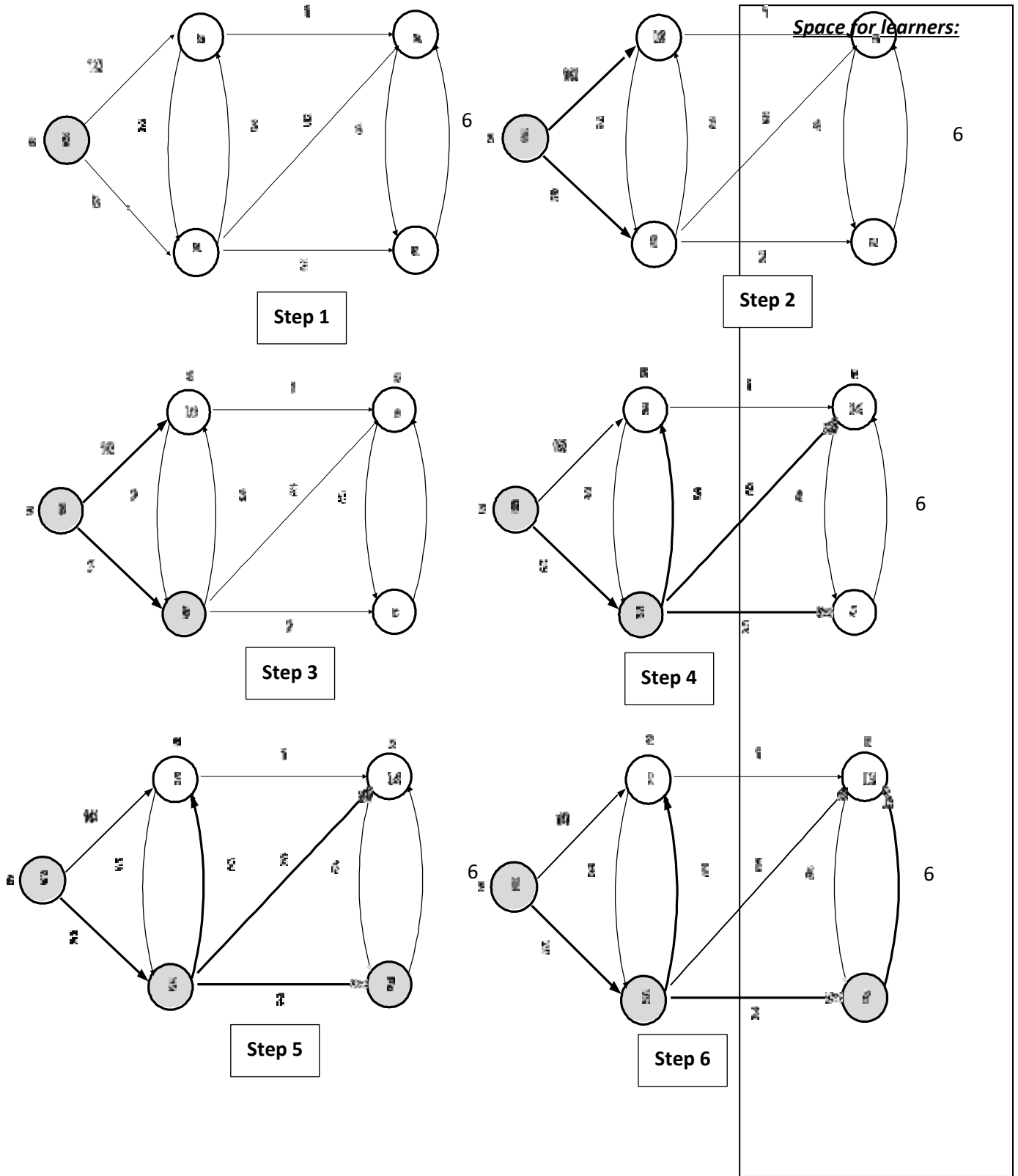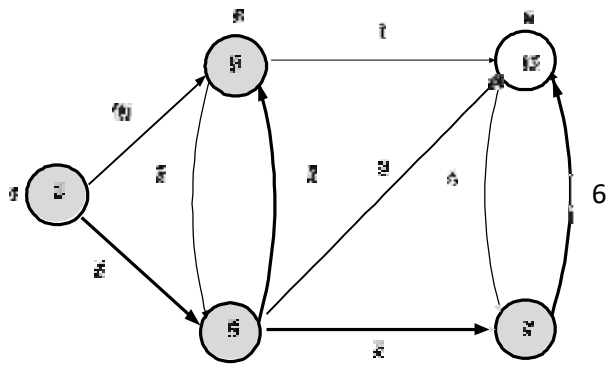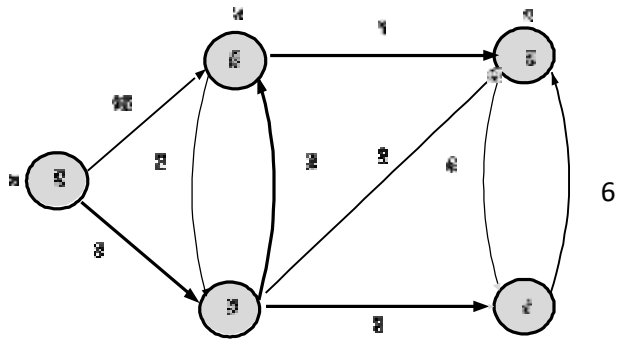
After applying Dijkstra's algorithm we found **F={s,b,d,a,c}** [distance is 9]

CHECK YOUR PROGRESS - II

d) The _____ algorithm uses greedy method to build the sub-tree edge by edge to obtain a minimum cost spanning tree.

e) In Kruskal's algorithm the edges of the graph are considered in _____ order.

f) List two applications of Minimum Spanning Trees.

1.4   DYNAMIC PROGRAMMING

The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. The term dynamic programming refers to this bottom-up approach. It was invented by mathematician named Richard Bellman inn 1950s. The DP in closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. The Dynamic Programming differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these sub-problems.

The steps of Dynamic Programming technique are:

Dividing the problem into sub-problems: The main problem is divided into smaller sub- problems. The solution of the main problem is expressed in terms of the solution for the smaller sub-problems.

Storing the sub solutions in a table: The solution for each sub-problem is stored in a table so that it can be referred many times whenever required.

Bottom-up computation: The DP technique starts with thesmallest problem instance and develops the solution to sub instances of longer size and finally obtains the solution of the original problem instance.

1.4.1 General Method

Dynamic Programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

When developing a dynamic programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically in bottom-up fashion

4. Construct an optimal solution from computed information.

Characteristics of Dynamic Programming:

- The Problem can be divided into stages, with a policy decision at each stage
- Each stage consist of a number of states associated with it
- Decision at each stage convert the current stage in to a state associated with next stage
- The state of the system at a stage is described by state variable
- When the current state is known, an optimal policy for the remaining stages is independent of the policy of the previous ones
- The solution procedure begins by finding the optimal solution of each state from the optimal solutions of its previous stage

1.4.2 Some Applications of the Dynamic-Programming

Matrix Chain Multiplication: Given a sequence of matrices that must be multiplied, parenthesize the product so that the total multiplication complexity is minimized.

0-1 Knapsack: Given items $x_1,\ldots, x_n$, where item x_i has weight w_i and profit p_i (if its placed in the knapsack), determine the subset of items to place in the knapsack in order to maximize profit, assuming that the sack has capacity M.

Longest Common Subsequence: Given an alphabet Σ, and two words X and Y whose letters belong to Σ , find the longest word Z which is a (non-contiguous) subsequence of both X and Y.

Optimal Binary Search Tree: Given a set of keys $k_1,\ldots, k_n$ and weights $w_1, \ldots, w_n$, where w_i reflects how often k_i is accessed, design a binary search tree so that the weighted cost of accessing a key is minimized.

1.5 BACKTRACKING

The name backtrack was first coined by D. H. Lehmer in the 1950s. Backtracking is a modified depth first search of a tree. Many problem which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formula. This algorithm tries to construct a solution to a computational problem incrementally, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates *every* alternative and then chooses the best one [4].

1.5.1 General Method

Many application of the backtrack method the desired solution is expressible as an n tuple $(x_1, x_2 ...x_n)$ where the x_i are chosen from some finite set s_i. Often the problem to be solved calls for finding one vector that maximizes or minimizes a criterion function $p(x_1, x_2 ...x_n)$.

Many application of the backtrack method the desired solution is expressible as an n-tuple $(x_1, x_2 ...x_n)$ where the x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector that maximizes or minimizes a criterion function $P(x_1, x_2 ...x_n)$. Sometimes it seeks all vectors that saitisfy P.

Problems, which are solving using the method backtracking required that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories [1]:

1. Explicit Constraints are rules that restrict each x_i to take on values only from a given set. The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Example:

$$x_i \geq 0 \quad or \ S_i = \{all \ nonnegative \ real \ number\}$$

$$x_i = 0 \ or \ 1 \quad S_i = \{0,1\}$$

$$l_i \leq x_i \leq u_i \quad or \ S_i = \{a: l_i \leq a \leq u_i\}$$

2. Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must related to each other.

There are two types of solution space tuple formulation:

1. *Variable size tuple:*

In this method for the solution vector (x_1, x_2,......,x_k), x_i will represent indices of ith choices for $1 \leq i \leq k$. Here size of the solution vector can varies for a problem.

2. *Fixed sized tuple:*

In this method for solution vector(x_1, x_2, x_3,......x_n), x_i € { 0,1 } and $1 \leq i \leq n$, such that x_i is 0 if i^{th} element not chosen and 1 otherwise. Here solution vector sizes are same for a problem.

The backtracking algorithm

Backtracking is quite simple. We " explore" each node, as follows"

- To "explore" node N
  - Step 1: If N is a goal node return "success"
  - Step 2: If N is a leaf node, return " failure"
  - Step 3: For each child C of N,
    - Step 3.1: Explore C
      - Step 3.1.1: If C was successful, return "success"
  - Step 4: Return "failure"

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We can say that the state space tree exists implicitly in the algorithm because it is not actually constructed. Main mechanism is that, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

Example:

Suppose we given a maze to find a path from start to finish.

To solve this, at each intersection, we have to decide between three or fewer choices:

⇒ Go straight
⇒ Go left
⇒ Go right

We don't have enough information to choose correctly.

Each choice leads to another set of choices.

One or more sequences of choices may (or may not) lead to a solution.

These types of maze problems can be solved with backtracking.
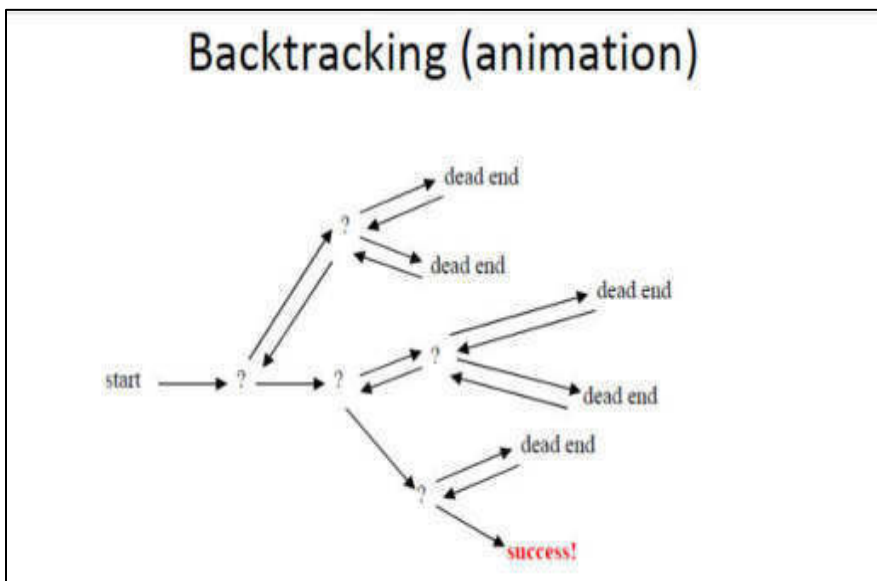
1.5.2 Tree Organization for Solution Space in Backtracking

Backtracking method determine solution of a problem by searching for the solution set in the solution space. This searching can be organized in a tree called **State Space Tree.** Terminologies used in State Space Tree are given below:

- ❖ **Problem State** is the state where each node in the DFS tree

- ❖ **Solution State** is the state are the problem states "S" for which the path from a root node to "S" defines a tuple in the solution space

- ❖ **Live node** is an node which has been generated and all of whose children have not generated yet

- ❖ **E-node** is the live node whose children are currently being generated

- ❖ **Dead node** is the node which is not expanded further or all of whose children have been generated

- ❖ **DFS** is the depth first node generation with bounding function is called backtracking

- ❖ In *state space tree*

    i.   *root* of the tree represent *0 choices*

    ii. *1st level node* represents *1st choices*

    iii. *2nd level node* represent *2nd choice*.

    iv. *nth level node* represent *nth choices*.

- ❖ **Non-promising node** is the node, if it cannot lead to a feasible solution and for this node n bounding function $B(n) = 0$. Otherwise, it is called **promising node** and bounding function $B(n)=1$. Non-promising nodes can be bounded or kill using bounding function. Then for this node its sub-trees are not generated.

- ❖ A state space tree is called **pruned state space tree** if it consist of only expanded node.
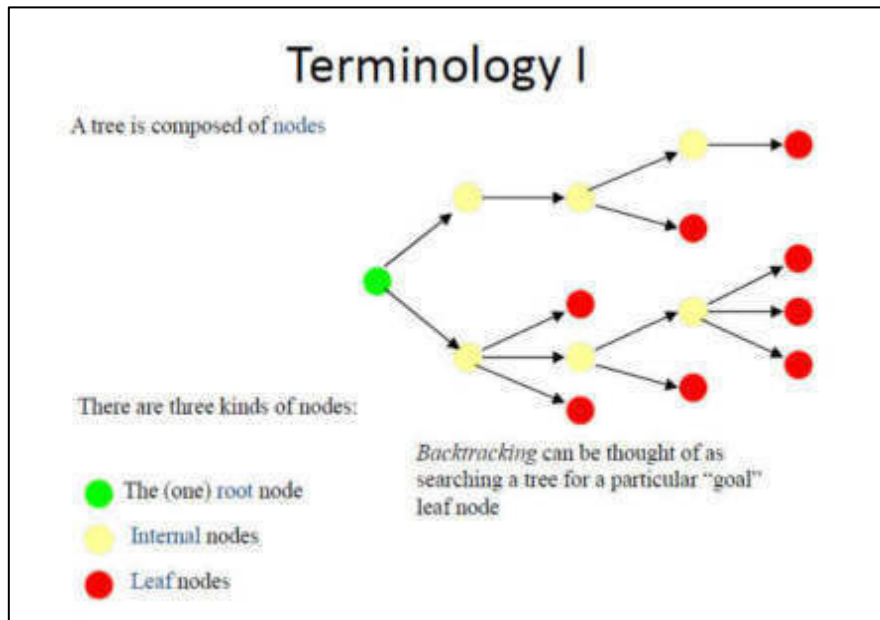
1.5.3 The N Queens Problem

Suppose we given N-Queens and NxN chess board. Now we have to find a way to place all N queens on the board show that no queens are attacking another queen. In chess, queens can move all the way horizontally, vertically or diagonally (if there is no other queen in the way).But, no two Queen can attack each other. So, due to this restriction, each queen must be on a different row and column.

Let, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples (x_1, ... , x_8), where x_i is the column of the ith row where the ith queen is placed.

The explicit constraints using this formulation are Si = {1, 2, 3, 4, 5, 6, 7, 8}, $1 < i < 8$.

The implicit constraints for this problem are that no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking strategy for 8-Queen problem is as follows-

⇒ Let, in the chess board rows and columns are numbered from 1 to 8 and also queens are numbered from 1 to 8

⇒ Without loss of generality, assume that i^{th} queen can be placed in i^{th} row, because no two queen can place in the same row

⇒ All solution can represented as 8-tuple (x_1, x_2,…. x_8), where xi is the column number of the ith row of ith queen placed

⇒ Here explicit constraints are Si = {1, 2, 3, 4, 5, 6, 7, 8}, $1 \le i \le 8$ and the solution space will consist of 8^8 8-tuple

⇒ According to the implicit constraints no two queen can on the same row

⇒ So, all solution are permutation of 8-tuple(1, 2, 3, 4, 5, 6, 7, 8)

⇒ Thus the searches is reduce to 8^8 8-tuple to 8! tuple

In 8 -Queen problem all the solution can represented as 8-tuple (x_1, x_2,….,x_8), where x_i is the column number of ith row where i^{th} queen placed. These all xi's are distinct because of the implicit constraint that no two queen can placed in same column. We assume already in no. 2 that i^{th} queen can be placed in i^{th} row only. So, no two queen can placed in same row. Now, we have only to decide whether two queens are on the same diagonal or not.

Suppose two queens are placed at position (i,j) and (k,l). then queens are on the same diagonal only if

$$i - j = k - l \quad or \; i + j = k + l$$

The first equation implies $\qquad j - l = i - k$

The second equation implies $\qquad j - i = k - i$

Therefore two queens lie on the same diagonal if and only if $|j - l| = |i - k|$

The time complexity of this approach is O(N!)

Visualization from a 4x4 chessboard solution

In this configuration, we place 2 queens in the first iteration and see that checking by placing further queens is not required as we will not get a solution in this path. Note that in this configuration, all places in the third rows can be attacked.

By using backtracking method we can bound the search of the state space tree using some constraint so that searching require less time.

For this problems to bound a node n constraints or bounding conditions B(n) are-

1. No two queens can place in same row i.e x_i always represents i^{th} queen is in i^{th} row.
2. No two queen in same column i.e values of x_i's are always distinct.
3. For two queen placed in (m, n) and (x, y) position in the chess , value of $|n - y|$ cannot same as $|m - x|$

When a node is bounded using bounding condition it will not generate any nodes in its sub-tree because nodes in its sub-tree will not give a feasible solution any more.
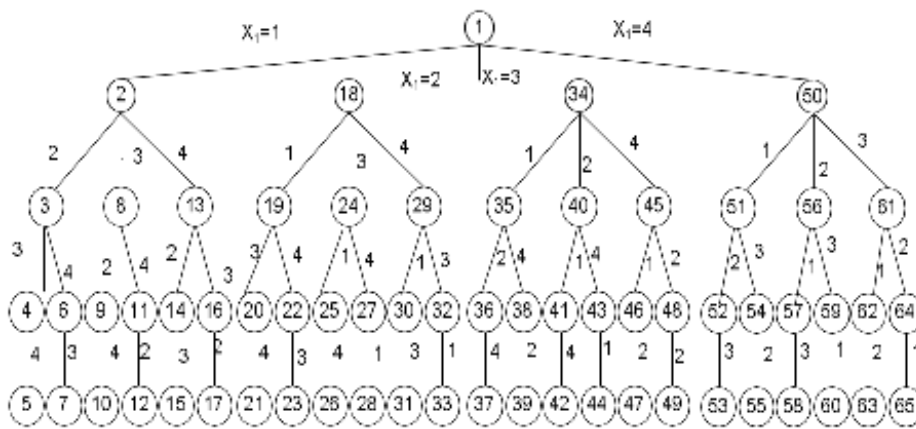

Fig: State Space Tree

The portion of pruned state space tree after applying bounding condition is as follows:
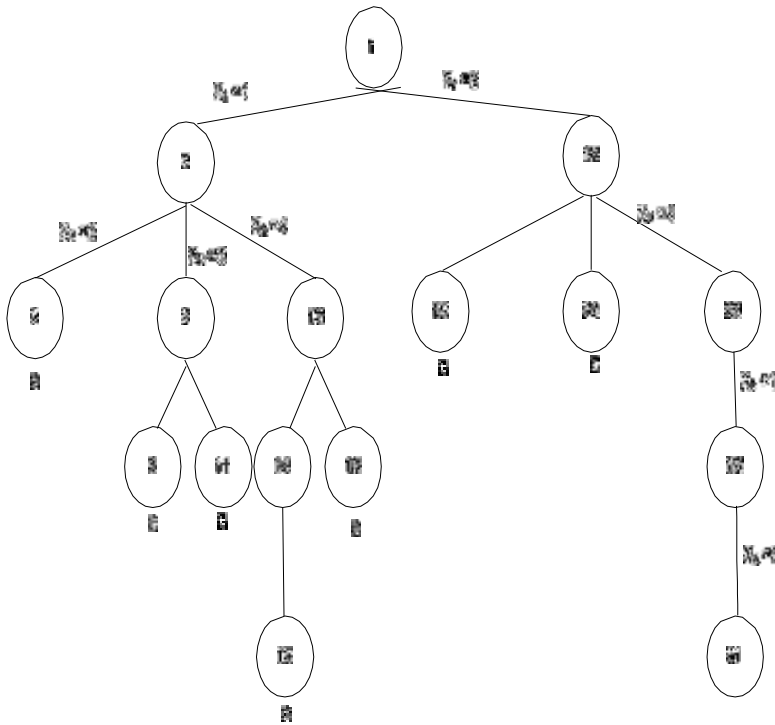
Fig: Pruned State Space Tree

Here node 3 is bounded because-

At level 1, $x_1=1$ means first time 1st queen is placed in 1st row, 1st column i.e position is (1,1)

At level 2 , $x_2=2$ means second time 2nd queen is placed in 2nd row, 2nd column i.e. position(2,2)

Thus they will place in diagonally. It will violet the implicit constraint or bounding condition. So this combination cannot give a feasible solution any more. So, the children of node 3 will not generated further. Hence node 3 will bound.

Here is a path from root 1 to leaf 31 and this will generate one feasible solution set (2, 4, 1, 3) where $x_1=2$, $x_2=4$, $x_3=1$, $x_4=3$.

Position of the 4 queens are (1,2), (2,4), (3,1) and(4,3) respectively.
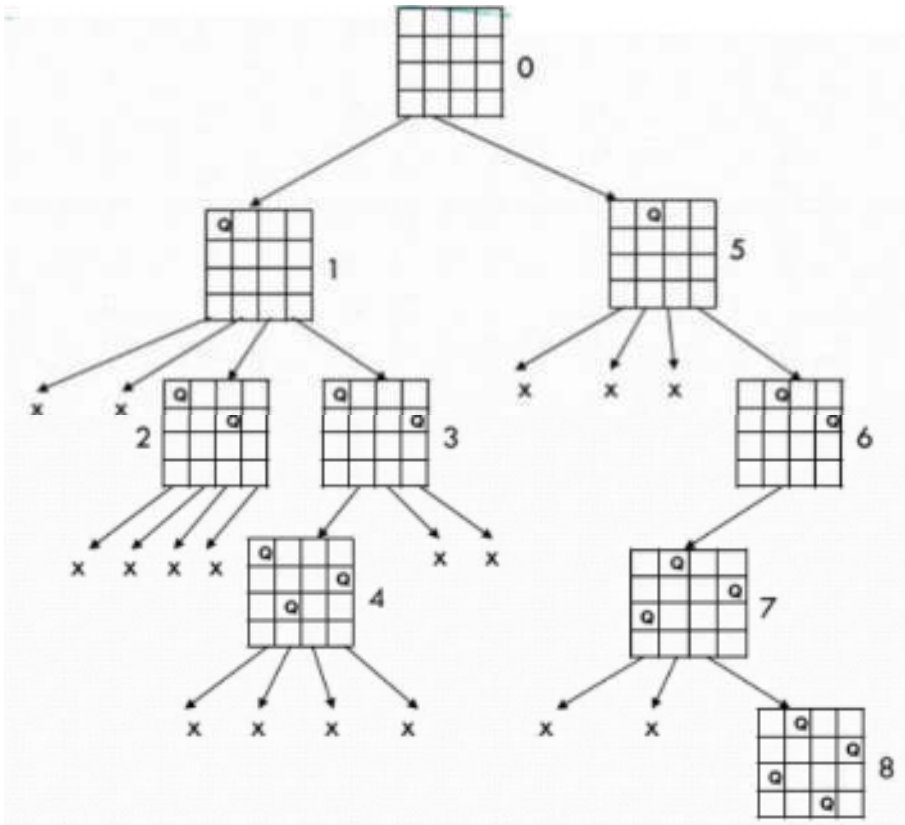
Fig: State space Tree for 4 Queens problem
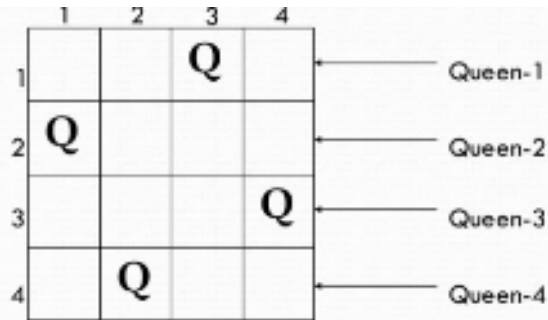
Fig: Solution of 4 Queens problem

A recursive backtracking function for n-Queen problem:

/* placed search for a new queen*/

```
bool QPlace ( int k, int i )

{

    for ( int m = 1; m < k; m++ )

    {

        if (( x [m] == i ) || (abs ( x[m] – i ) == abs ( m – k )))

            return (false);

        return (true);

    }

}
```

/* Solution to n queen*/

```
void nQueen ( int k, int i )
{
    for ( int i =1; i ≤ n;
      i++)if ( QPlace( k, i
     ))
      {
          x [k] = i;
          if ( k == n )
            {
              for ( int m = 1; m ≤ n; m++ )
              {cout << x [m] <<'
                 '<<;cout<<endl;
              }
          }
            else
              nQueen ( k + 1, n );
      }
}
```

Here QPlace (k, i) will return a boolean value true or false. The function return true if k^{th} queen can placed in i^{th} column and assigned it to x [k]. This value x [k] = i is distinct from x [1]…… x [k-1]. It also ensures that no two queens is placed in same diagonal.

Next nQueen (k, n) will solve the n-Queen problem recursively using backtracking method.

1.4.4 Hamiltonian Cycle

Hamiltonian Path is an undirected graph, which visits each vertex exactly once. A Hamiltonian Cycle is a Hamiltonian Path such that there is an edge in the graph from the last vertex to first vertex of the Hamiltonian Path, i.e. each vertex visit once in graph G and return to the starting vertex. It is named after William Hamilton.

Input: A 2D array graph[V][V]

where V is the number of vertices in graph

graph [V][V] is adjacency matrix representation of the graph

graph[i][j]=1   if there is a direct edge from i to j

graph[i][j]=0   otherwise

Output: An array path[V] that should contain the Hamiltonian Path. Path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example:

Consider the following graph and evaluate to check whether the graph is Hamiltonian or not.

Fig: A graph with 8 vertices

Fig: Hamiltonian Cycle
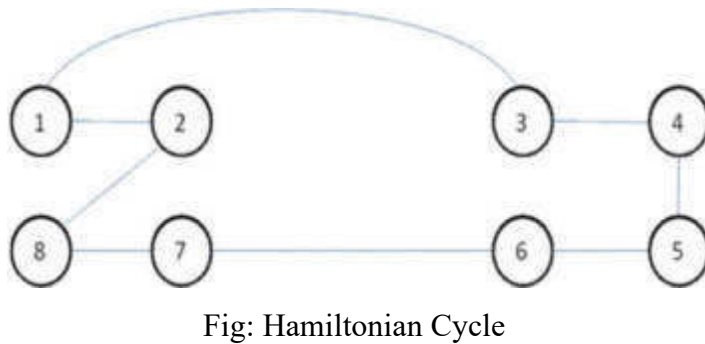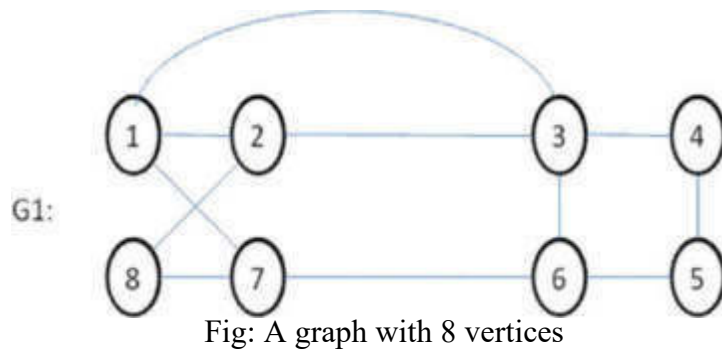
The Hamiltonian cycle of this graph is- 1, 2, 8, 7, 6, 5, 4, 3, 1

Backtracking method for Hamiltonian cycle:

Now, using backtracking method we can find out the Hamiltonian cycles in a graph which has n vertices. The solution set can represented as $(x_1, x_2, \ldots \ldots, x_n)$, where $1 \leq i \leq n$ and x_i represents the ith visited vertex of the current considered cycle.

We have to determine value of x_i i.e possible vertex to select. For i =1, x_1 can be any vertex chooses from n vertex. To determine value of x_i we have already determine $x_1, x_2, , x_{i-1}$. Hence, the x_i can be choose as

 i) any vertex v which is not assigned to $x_1, x_2 \ldots$..and x_{i-1} from the n vertices.

 ii) v is connected by an edge to x_{i-1}

The last vertex x_n must be connected to both x_{n-1} and x_1.

For example: Consider the given graph and evaluate the mechanism:
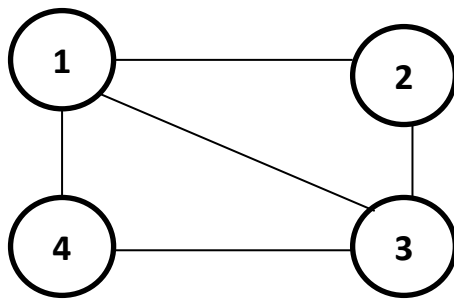
State space tree for finding the Hamiltonian Circuit is:
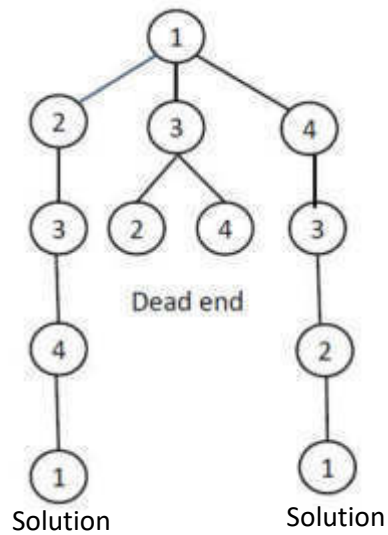
Recursive function for Hamiltonian cycle:

```
/* for finding Hamiltonian cycle*/
 void Hamiltonian ( int k )
  {
      do
       {
         NextValue ( k );
            if ( ! x [ k ] ) return;
            if ( k == n )
                {
                  for ( int i =1; i ≤ n; i++)
                     cout << x [ i ] <<" "<<"\n";
```

```
              }
          else
            Hamiltonian ( k + 1 )
      }
   while(1);
 }
```

/* generating next vertex*/

```
void NextValue ( int k )
   {
     do
         {
          x [ k ] = ( x [ k ] + 1 ) % ( n + 1 );
          if ( ! x [ k ] ) return;
          if ( G [ x [ k – 1 ] [ x [ k ] ])
            {
              if ( ( k < n ) || (( k == n ) && G [ x [ n ] ] [ x [ 1] ] ))
                  return;
            }
         } while(1);
   }
```

This program first initializes the adjacency matrix G [1:n] [1:n] and x [1] = 1 and x [2:n] = 0.

1.6   BRANCH AND BOUND

Branch and bound is an algorithm design technique that is often implemented for finding the optimal solutions in case of optimization problems; it is mainly used for combinational and discrete global optimizations of problems.

Branch and bound is composed of two main actions. Firstly, *branching,* where we define the tree structure from the set of candidates in a recursive manner. Secondly, *bounding,* where we calculates the upper and lower bounds of each node from the tree. Furthermore, there is an additional *pruning* step, where depending on the values of upper bound and lower bound some node can be discarded from the search.

Branch and bound is similar to backtracking. The main difference is that the branch and bound is used only in case the case of optimization problems, whereas backtracking can't be. Another difference is that, backtracking always picks one single successor from the candidates, while branch and bound always has the entire list of successors in the queue.

1.6.1 General Method

The branch and bound algorithm is based on an advanced breadth-first search, where breath- first search is performed with the help of apriority queue instead of the traditional list. The term branch and bound refers to all state search methods in which all children of the E-node are generated before any other live node can become the E-node. In branch-and-bound terminology, a BFS- like state space search will be called FIFO(First In First Out) search as the list of live nodes is a FIFO list or queue. A DFS search like state space search is called LIFO search as the list of live nodes is a LIFO list or a stack. To avoid the generation of subtrees that do not contain an answer node, bounding function is the best method than backtracking [4].

In branch and bound it is crucial to understand the importance of two functions: g(x) and h(x). The first function, $g(x)$, calculates the distance between the x node and the root node. Whereas, $h(x)$, is a heuristic function because it estimates how close the x node to the solution. Moreover, we can say that **f(x) = g(x) + h(x)**. The g(x) part is the path-cost function, while the h(x) part is the admissible heuristic estimate; the sum of these two is the f(x).

1.6.2 Travelling Salesman Problem

Instead of using a Queue to perform a breadth-first traversal of the state space, we will use a Priority Queue and perform a "best-first" traversal. For the TSP we first compute the minimum possible tour by finding the minimum edge exiting each vertex. The sum of these edges may not form a possible tour, but since every vertex must be visited once and only once, every vertex must be exited once. Therefore, no tour can be shorter than the sum of these minimum edges.

At each subsequent node, the lower bound for a "tour in progress" is the length of the tour to that point plus the sum of the minimum edge exiting the end vertex of the partial tour and each of the minimum edges leaving all of the remaining unvisited vertices. If this bound is less than the current minimum tour, the node is "promising" and the node is added to the queue. Initially the minTour is set to infinity. When a node whose path includes all of the vertices except one is reviewed, there is only one possible way for the tour to complete. The remaining vertex and the first are added to the path and the length of the tour is the current length plus the length of the edge to the remaining vertex and the length of the edge from there back to the starting vertex. If this tour length is better than the current minimum, it becomes the minimum tour length. Once a first complete tour is discovered, nodes whose bound is greater than or equal to this minTour are deemed "non- promising" and are pruned.

The nodes in state space must carry the following information:
- their level in the state space tree
- the length of the partial tour
- the path of the partial tour
- the bound
- (for efficiency) the last vertex in the partial tour

In a branch and bound algorithm, a node is judged to be promising before it is placed in the queue and tested again after it is removed from the queue. If a lower minTour is discovered during the time a node is in the queue, it may no longer be promising after it is removed, and it is discarded. Using a Priority Queue, the search traverses the state space tree in neither a breadth-first nor depth-first fashion, but alternates between the two approaches in a greedy, opportunistic fashion. In the example problem below, a diagram of the best-first traversal of the state space indicates by number when each of the nodes is removed from the priority queue

Example:

Let G be a fully connected directed graph containing five vertices that is represented by the following adjacency list:
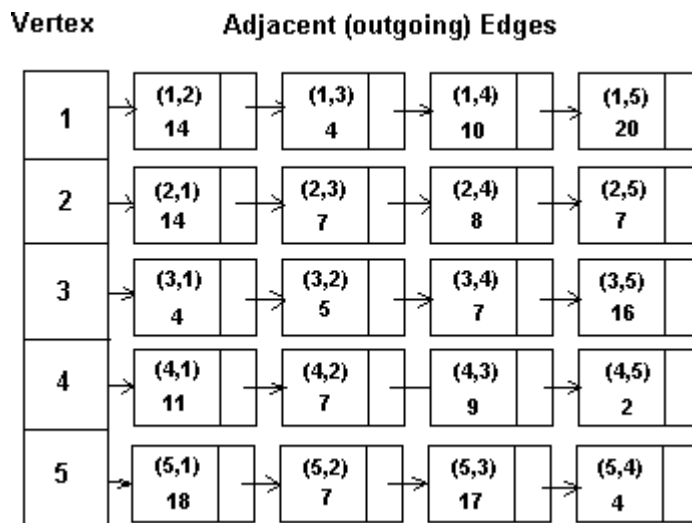
Vertex   Adjacent (outgoing) Edges

We assume in the implementation of this algorithm that vertices are labeled by an integer number and edges contain the source and sink vertices and a cost or length label. The tour will start at vertex 1, and the initial bound for the minimum tour is the sum of the minimum outgoing edges from each vertex.

$$\begin{array}{lll}
\text{Vertex 1} & \min (14, 4, 10, 20) & = 4 \\
\text{Vertex 2} & \min (14, 7, 8, 7) & = 7 \\
\text{Vertex 3} & \min (4, 5, 7, 16) & = 4 \\
\text{Vertex 4} & \min (11, 7, 9, 2) & = 2 \\
\text{Vertex 5} & \min (18, 7, 17, 4) & = 4 \\
\text{bound [1]} & & = 21
\end{array}$$

Since the bound for this node (21) is less than the initial minTour (), nodes for all of the adjacent vertices are added to the state space tree at level 1. The bound for the node for the partial tour from 1 to 2 is determined to be:

bound = length from 1 to 2 + sum of min outgoing edges for vertices 2 to 5

$= 14 + (7 + 4 + 2 + 4)$

$= 31$

After each new node is added to the PriorityQueue, the node with the best bound is removed and similarly processed. The algorithm terminates when the queue is empty.
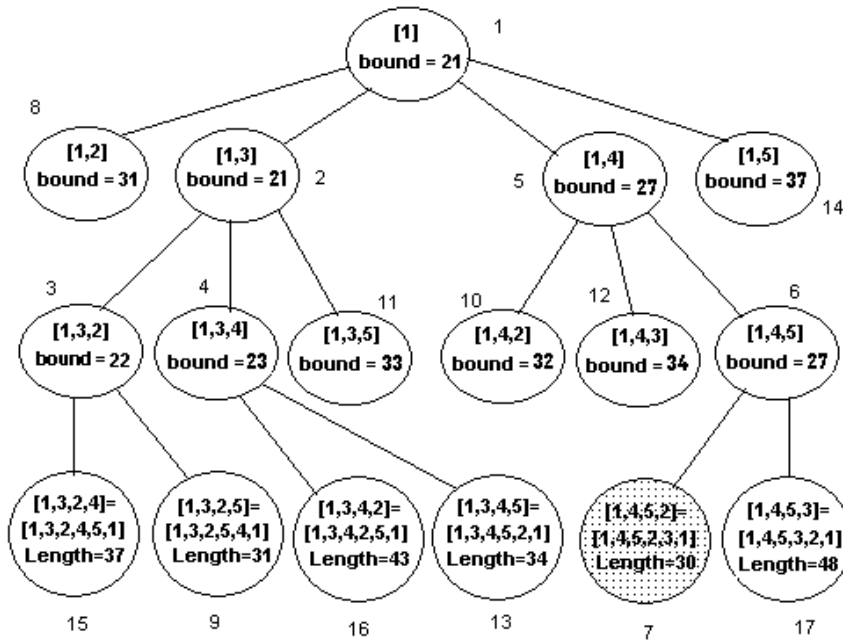
Second node placed in the priority queue, but the 8th node to be removed. By the time it is removed and examined, a tour of length 30 which turns out to be the optimal tour, has already been discovered, and, since its bound exceeds this length, it is discarded without having to check any of the possible tours that extend it.

Here is a Branch and Bound algorithm for an adjacency list representation of a graph. If the first vertex is numbered 1 instead of 0, the array bounds for *mark* and *minEdge* would have to be length N + 1 and the loops traversing these arrays would have to be from 0 to N.

CHECK YOUR PROGRESS - III

g) Backtracking is a modified _____ first search of a tree

h) Hamiltonian Path is an undirected graph, which visits each vertex exactly _____ .

1.7 SUMMING UP

- An algorithm is a sequence of computational steps that start with a set of input(s) and finish with valid output(s)

- An algorithm is correct if for every input(s), it halts with correct output(s)

- Divide and conquer algorithm has three steps

  $\Rightarrow$ Divide the problem into smaller independent sub-problems

  $\Rightarrow$ Conquer by solving these sub-problems

  $\Rightarrow$ Combine these sub-problems to together

- The sub-problems solved by a divide and conquer is non-overlapping

- Greedy algorithm is typically used in optimization problem

- Optimal solution finds a given objective function which value is either maximizes or minimizes

- A greedy algorithm always makes the choice that looks best at the moment. That is it makes a locally optimal choice that may be lead to a globally optimal solution

- In Greedy algorithm choice is made that seems best at the moment and solve the sub-problems after the choice is made

- Greedy algorithm progress in a top down manner

- A greedy algorithm gives optimal solution for all subproblems, but when these locally optimal solutions are combined it may NOT result into a globally optimal solution. Hence, a greedy algorithm cannot be used to solve all the dynamic programming problems.

- A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solution to its sub-problem

- A spanning tree is a connected graph, say G = (V, E) with V as set of vertices and E as set of edges, is its connected acyclic sub-graph that contain all the vertices of the graph

- A minimum spanning tree T of a positive weighted graph G is a minimum weighted spanning tree in which total weight of all edges are minimum

- A problem can be solved by dynamic programming only when it possesses optimal substructure

- A problem is said to satisfy the principle of optimality, if the sub solutions of an optimal solution of the problem are themselves optimal solution for their sub problems

- In dynamic programming we first solve the sub-problems and then use these solutions to get the optimal solution in recursive manner

- Backtracking is a method for searching a set of solutions or find an optimal solution for satisfy some given constraint to a problem

- In backtracking method the solution set can be represented by an n tuple $(x_1, x_2, \ldots, x_n)$, where x_i are chosen from some finite set S_i

- Backtracking method can be used for optimization problem to find one or more solution vector that maximize or minimize or satisfy a given criterion function

- In backtracking constraint to be satisfied can be divided into two categories- Implicit constraint and Explicit constraint

- Two types of tuple formulation- Variable size tuple and Fixed sized tuple

- In backtracking method searching can be organized in a tree called state space tree

- A solution state is a node s for which each node from root node to nodes together can represent a tuple in solution set

- A live node is a generated node, for which all of its children node have not yet generated.

- A E-node (Expanded node) is a live node, whose children are currently being generated

- A dead node is that , which is not expanded further and all of its children is generated

- A node n is called non-promising if it cannot lead to a feasible solution. Otherwise, it is called promising node

- A state space tree is called pruned state space tree if it consist of only expanded node

- Backtracking method do depth first search of a state space tree

- It is a generalized problem of 8-Queen problem. N Queens are placed on a chess board of size n x n, without having attack each other.

- A Hamiltonian cycle of a connected undirected graph with n vertices is a cyclic path along n edges, such that each vertex visits once in graph G and return to the starting vertex

- Branch and Bound is a state space search method in which all the children of a node are generated before expanding any of its children

- It is similar to backtracking technique but uses BFS-like search

- Branch and bound techniques uses the priority queue data structure for storing the information

- Branch and bound technique mainly based on the value $g(x) + h(x)$, where $g(x)$ is the distance from the root to the current vertex and $h(x)$ is a heuristic function.

1.8 ANSWERS TO CHECK YOUR PROGRESS

a) Algorithm

b) Optimal

c) Efficiency

d) Prim's

e) Non-decreasing

f) Two applications of Minimum Spanning Tree:

    i. In design of electric circuit network

    ii. It is used in traveling salesman problem

g) Depth

h) Once

1.9   POSSIBLE QUESTIONS

**Short Answer type Questions:**

1) What is optimal substructure?
2) How does the greedy choice property applied in optimal merge pattern problem?
3) What is minimum spanning tree?
4) What are the algorithms to solve minimum spanning tree problem ?
5) What is backtracking method?
6) Write about state space tree organization of backtracking method.
7) What is Hamiltonian cycle?

Long Answer type questions:

1) Explained the criteria that all algorithms must satisfy.
2) Solve by substitution method a=1, b=2, f(n)=c
3) Solve recurrence relation a=2, b=2, f(n)=cn
4) Solve $T(n) = q.T\left(\frac{n}{3}\right) + 4n^6; \quad n \geq 3$
5) Explain the characteristics of dynamic programming.
6) Describe the steps of dynamic programming algorithm.
7) Describe the method of solving travelling salesman problem using branch and bound strategy.
8) How does backtracking method find Hamiltonian cycle in a graph?
9) What is 8-queen problem? How can it solve using backtracking?
10) What is the bounding condition for n-queen problem?
11) What is minimum spanning tree? Find the minimum spanning tree for the following graph using Prim's and Kruskal algorithm
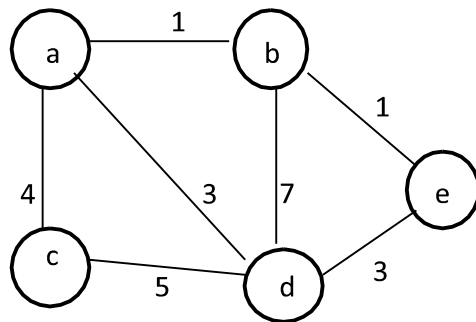
12) Find out the shortest path using Dijkstra algorithm for the following graph
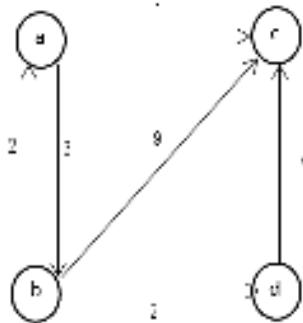
1.10 REFERENCES AND SUGGESTED READINGS

[1] Design And Analysis Of Algorithms [R18a0507], Malla Reddy College Of Engineering & Technology.

[2] T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.

[3] Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

[4] Design And Analysis Of Algorithms, by Malla Reddy College Of Engineering & Technology.

UNIT 2: ALGORITHM DESIGN TECHNIQUES II

Unit Structure:

2.1 INTRODUCTION

In the previous unit we discussed about various algorithm design techniques. We can view an algorithm as a tool for solving a well-specified computational problem. The algorithm describes a specific computational procedure for achieving that input/output relationship.

Computers may be fast, but they are not efficiently fast and memory may be inexpensive, but it is not free. Computing time is therefore bounded resource and so is space in memory. We should use those resources wisely and algorithms that are efficient I n terms of time or space will help us to do so. Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software [1]. Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

An algorithm is said to be correct if for every input instance, it halts with the correct output. Correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.

2.2 SEARCHING

In computing locating information is important and recurring problem, this problem is known as searching. Searching is considered as one of the issue involved in algorithm design [4].

The information to be searched has to first be represented (or encoded) somehow. This is where data structures come in. Of course, in a computer, everything is ultimately represented as sequences of binary digits (bits), but this is too low level for most purposes. We need to develop and study useful data structures that are closer to the way humans think, or at least more structured than mere sequences of bits.

After that we have chosen a suitable representation for information and then processed them. This is what leads to the need for algorithms. In this case, the process of interest is that of searching.

Let us assume that we want to search a collection of integer numbers. To begin with, let us consider:

1. The most obvious and simple representation.

2. Two potential algorithms for processing with that representation.

Arrays are one of the simplest possible ways of representing collections of numbers (or strings, or whatever), so we shall use that to store the information to be searched.

Suppose, for example, that the set of integers we wish to search is {1, 4, 17, 3, 90, 79, 4, 6, 81}.

We can write them in an array a as

a = [1, 4, 17, 3, 90, 79, 4, 6, 81]

If we ask where 17 is in this array, the answer is 2, the index of that element. If we ask where 91 is, the answer is nowhere. It is useful to be able to represent nowhere by a number that is not used as a possible index. Since we start our index counting from 0, any negative number would do. We shall follow the convention of using the number -1 to represent nowhere.

We can now formulate a specification of our search problem using that data structure:

Given an array a and integer x, find an integer i such that

1. If there is no j such that a[j] is x, then i is -1,

2. Otherwise, i is any j for which a[j] is x

The first clause says that if x does not occur in the array a then i should be -1, and the second says that if it does occur then i should be a position where it occurs. If there is more than one position where x occurs, then this specification allows you to return any of them.

In the previous unit we introduced with Divide-and-conquer approach that is used in the design of algorithms. This technique is the basis of designing efficient algorithms for all kinds of problems, such as sorting like insertion sort, merge sort, quick sort and in searching like binary search.

2.2.1 Binary Search

Binary search is a well-known instance of divide and conquer method. For binary search divide and conquer strategy is applied recursively for a given sorted array is as follows:

Divide: Divide the selected array at the middle. It creates two sub-arrays, one left sub-array and other right sub-array.

Conquer: Find out the appropriate sub-array.

Combine: Check for the solution to key element.

For a given sorted array of N element and for a given key element (value to be searched in the sorted array), the basic idea of binary search is as fallows –

1. First find the middle element of the array

2. Compare the middle element with the key element.

3. There are three cases

- If it is the key element then search is successful

- If it is less than key element then search only the lower half of the array

- If it is greater than key element then search only the upper half of the array

4. Repeat 1, 2 and 3 until the key element found or sub-array sizes become one.

Problem definition:

Let a_i, $1 \leq i \leq n$ be a list of elements that are sorted in non-decreasing order. The problem is to find whether a given element x is present in the list or not. If x is present we have to determine a value j (element's position) such that $a_j = x$. If x is not in the list, then j is set to zero [5].

Solution: Let $P = (n, a_{i......}a_l, x)$ denote an arbitrary instance of search problem where n is the number of elements in the list, $a_i...a_l$ is the list of elements and **x** is the key element to be searched for in the given list. **Binary search** on the list is done as follows:

Step1: Pick an index q in the middle range [i, l] i.e. $q = [(n + 1)/2]$ and compare x with a_q

Step 2: if $x = a_q$ i.e key element is equal to mid element, the problem is immediately solved.

Step 3: if $x < a_q$ in this case x has to be searched for only in the sub-list a_i, a_{i+1}, ..., a_q- Therefore, problem reduces to **(q-i, a_i...a_{q-1}, x).**

Step 4: if $x > a_q$,x has to be searched for only in the sub-list a_{q+1}, ...,., a_l . Therefore problem reduces to **(l-i, a_{q+1}...a_l, x).**

For the above solution procedure, the Algorithm can be implemented as recursive or non- recursive algorithm

Recursive binary search algorithm

```
int BinSrch(Type a[], int i, int l, Type x)
// Given an array a[i:l] of elements in nondecreasing
// order, 1<=i<=l, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
    if (l==i) { // If Small(P)
        if (x==a[i]) return i;
        else return 0;
    }
    else { // Reduce P into a smaller subproblem.
        int mid = (i+l)/2;
        if (x == a[mid]) return mid;
        else if (x < a[mid]) return BinSrch(a,i,mid-1,x);
        else return BinSrch(a,mid+1,l,x);
    }
}
```

Iterative binary search:

```
int BinSearch(Type a[], int n, Type x)
// Given an array a[1:n] of elements in nondecreasing
// order, n>=0, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
    int low = 1, high = n;
    while (low <= high){
        int mid = (low + high)/2;
        if (x < a[mid]) high = mid - 1;
        else if (x > a[mid]) low = mid + 1;
        else return(mid);
    }
    return(0);
}
```

Analysis

In binary search the basic operation is key comparison. Binary Search can be analyzed with the best, worst, and average case number of comparisons. The numbers of comparisons for the recursive and iterative versions of Binary Search are the same, if comparison counting is relaxed slightly. For Recursive Binary Search, count each pass through the if-then-else block as one

comparison. For Iterative Binary Search, count each pass through the while block as one comparison. Let us find out how many such key comparison does the algorithm make on an array of **n** elements.

Best case – Θ (1) In the best case, the key is the middle in the array. A constant number of comparisons (actually just 1) are required.

Worst case – Θ (log₂ n) In the worst case, the key does not exist in the array at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done ceiling(log_2n) times. Thus, [log_2 n] comparisons are required.

Sometimes, in case of the successful search, it may take maximum number of comparisons] log2 n]. So worst case complexity of successful binary search is Θ (log2 n).

Average case - Θ (log₂n) To find the average case, take the sum of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in array will be searched for, and that the probabilities of searching for each element are uniform.

successful searches			unsuccessful searches
$Θ(1)$,	$Θ(\log n)$,	$Θ(\log n)$	$Θ(\log n)$
best,	average,	worst	best, average, worst

Space Complexity - The space requirements for the recursive and iterative versions of binary search are different. Iterative Binary Search requires only a constant amount of space, while Recursive Binary Search requires space proportional to the number of comparisons to maintain the recursion stack.

Binary Search Example

Advantages: Efficient on very big list, Can be implemented iteratively/recursively.

Limitations:

- Interacts poorly with the memory hierarchy

- Requires sorted list as an input

- Due to random access of list element, needs arrays instead of linked list

2.2.2 Optimal Binary Search Tree

A binary search tree is a tree where the key values are stored in the internal nodes, the external nodes (leaves) are null nodes, and the keys are ordered lexicographically, i.e. For each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.

When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a binary search tree which has minimal expected cost of locating each node. In our problem, we

are not concerned with the frequency of searching for a missing node. For example:

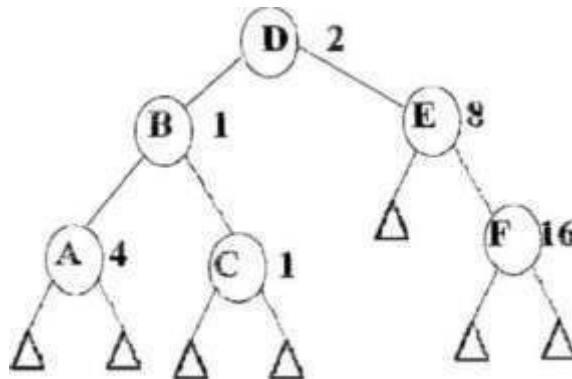| Node ID	0	1	2	3	4	5
Key	A	B	C	D	E	F
Frequency	4	1	1	2	8	16

Fig a: Optimal Binary search tree ex1.

[2*1 + (1+8)*2 + (4+1+16)*3] = 83

The expected cost of successful search is 83, is computed by multiplying each frequency by its level (starting w ith the root at 1). A different tree will have a different expected cost
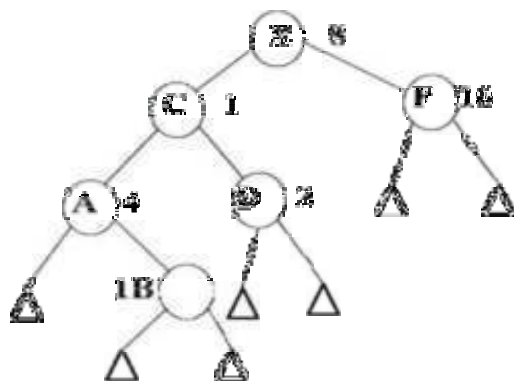

Fig b: Optimal Binary search tree ex2

[8*1 + (1+16)*2 + (4+2)*3 + 1 *4] = 64

It's clear that the tree in fig a is not optimal. - It is easy to see that the nodes having higher frequencies are closer to the root, and then the tree will have a lower expected cost.

In obtaining a cost function for binary search trees, it is useful to add an external node in place of every empty sub-tree in the search tree. If a binary search tree represents n identifiers, then there will be exactly n internal nodes and n+1 external nodes.

If a successful search terminates at an internal node at level l, then the expected cost contribution from the internal node a_i is p(i) * level(a_i).

Unsuccessful searches terminates the external nodes, let the unsuccessful searches terminates at node E_i, if the failure node is at level l, then only l-1 comparisons will be made, so the cost contribution of this node is q(I) * (level(E_i) -1)

The preceding decision leads to the following formula for the expected cost of a binary search tree.

$$\sum_{1 \le i \le n} p(i) * level(a_i) + \sum_{1 \le i \le n} q(i) * (level(E_i) - 1)$$

We define a optimal binary search tree for the identifier set { a_1, a_2, ..., a_n} to be a binary search tree for which the above equation is minimum

To solve this problem by dynamic programming we need to view the construction of such a tree as the result of a sequence of decisions and then observe that the principle of optimality holds when applied to the problem state resulting from a decision. A possible approach to this would be to make a decision as to which of the a_i's should be assigned to the root of the tree. If we choose a_k, then it is clear that the internal nodes for a_1, a_2, ..., a_{k-1} as well as external nodes for the classes E_1, E_2, ..., E_{k-1} will be in the left subtree, l, of the root. The remaining nodes will be in the right subtree, r. Define

$$cost(I) = \sum_{1 \le i < k} p(i) * level(a_i) + \sum_{1 \le i < k} p(i) * level(E_i - 1)$$

$$cost(r) = \sum_{k < i < n} p(i) * level(a_i) + \sum_{k < i < n} p(i) * level(E_i - 1)$$

In both cases the level is measured by regarding the root of the respective subtree to be at level 1.

Using w(I,j) to represent the sum q(i) + $\sum_{l=i+1}^{j}$ (q(I)+p(I)), we obtain the following as the expected cost of the search tree

p(k) + cost(I) + cost(r) + w(0, k-1) + w(k, n)

If the tree is optimal then the above equation must be minimum. Hence, cost(l) must be minimum over all the binary search trees containing $a_1, a_2, ..., a_{k-1}$ and $E_1, E_2, ..., E_{k-1}$. Similarly cost(r) must be minimum. If we use c(I,j) to represent the cost of an optimal binary search tree, t_{ij}, containing $a_{i+1}, a_{i+2}, ..., a_j$ and $E_{i+1}, E_{i+2}, ..., E_j$, then for the tree to be optimal, we must have cost(I) = c(0, k-1) and cost(r) = c(k,n). In addition k must be choose such that

p(k) + c(0, k-1) + c(k,n) + w(0, k-1) + w(k, n) is minimum. Hence c(0,n) we obtain

$$c(0,n) = \min_{i \le k \le n} \{p(k) + c(0, k-1) + c(k,n) + w(0, k-1) + w(k, n)\}$$

we can generalized this equation for any c(i,j) as follows:

$$c(i,j) = \min_{i \le k \le j} \{p(k) + c(i, k-1) + c(k,j) + w(i, k-1) + w(k, j)\}$$

$$c(i,j) = \min_{i \le k \le j} \{c(i, k - 1) + c(k, j)\} + w(i,j)$$

This equation can be solved for c(0,n) by first computing all c(i,j) such that j-i=1. Next we can compute all c(i,j) such that j-i =2, then all c(i,j) with j-i=3, etc. if during this computation we record the root r(i,j) of each tree t_{ij}, then an optimal binary search tree can be constructed from these r(i,j).

CHECK YOUR PROGRESS - I

a) Binary search is a well-known instance of _____and _____method

b) List two limitations of Binary Search

2.3   SORTING

Sorting is a fundamental operation in computer science. As a result we have a large number of good sorting at or disposal. Which algorithm is best for a given application depends on other factors also. These are:

(a) The no of items to be sorted

(b) The extent to which the items are already somewhat sorted

(c) Possible restrictions on the item values

(d) The architecture of the computer

(e) The storage device used like main memory, disks or tapes

Formal definition of sorting problem:

Input: A sequence of n numbers $< a_1, a_2, \ldots \ldots \ldots, a_n >$

Output: a permutation (reordering) $<a'_1, a'_2, \ldots \ldots \ldots, a'_n >$ **of the input sequence such that** $<a'_1 \leq a'_2 \leq \ldots \ldots \ldots \leq a'_n >$

The numbers that we wish to sort are also known as the keys. Conceptually we are sorting a sequence, but the input comes in the form of an array with n elements [2].

For example, given the input <31, 41, 59, 26, 41, 58>, a sorting algorithm returns as output the sequence <26, 31, 41, 41, 58, 59>. Such an input sequence is called instance of the sorting problem. An instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Sorting techniques can be classified into two types:

- **Internal sorting techniques:** Any sort algorithm that uses main memory exclusively during the sorting is called as internal sort algorithms. Internal sorting is faster than external sorting. Some example internal sorting algorithms are Insertion Sort, Bubble Sort, Selection Sort, Heap Sort, Shell Sort, Bucket Sort, Quick Sort, Radix Sort.

- **External sorting techniques:** Any sort algorithm that uses external memory, such as tape or disk, during the sorting is called as external sort algorithms. Merge Sort is one of the external sort algorithms.

2.3.1 Insertion Sort

We start with insertion sort, which is an efficient algorithm for sorting a small number of elements. For insertion sort we used an incremental approach. In insertion Sort the number of comparisons depends on the order of the input elements. We begin with the subarray of size 1, A[1], which is already sorted. Next, A[2] is inserted before or after A[1] depending on whether it is smaller than A[1] or not. Continuing this way, in the ith iteration, A[i] is inserted

in its proper position in the sorted subarray A[1..i − 1]. This is done by scanning the elements from index i−1 down to 1, each time comparing A[i]with the element at the current position. An element is shifted one position up to a higher index, in each iteration of the scan. This process of scanning, performing the comparison, and shifting continues until an element less than or equal to A[i] is found, or when all the sorted sequence so far is exhausted. At this point, A[i] is inserted in its proper position, and the process of inserting element A[i] in its proper place is complete [3].

Algorithm: INSERTIONSORT

Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

1. **for** $i \leftarrow 2$ **to** n
2. $x \leftarrow A[i]$
3. $j \leftarrow i - 1$
4.   **while** $(j > 0)$ **and** $(A[j] > x)$
5. $A[j + 1] \leftarrow A[j]$
6. $j \leftarrow j - 1$
7.   **end while**
8. $A[j + 1] \leftarrow x$
9. **end for**

The number of element comparisons done by Algorithm INSERTIONSORT depends on the order of the input elements. It is easy to see that the number of element comparisons is minimum when the array is already sorted in non-decreasing order. In this case, the number of element comparisons is exactly $n - 1$, as each element $A[i]$, $2 \leq i \leq n$, is compared with $A[i - 1]$ only. On the other hand, the maximum number of element comparisons occurs if the array is already sorted in decreasing order and all elements are distinct. In this case, the number of element comparisons is

$$\sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

as each element $A[i]$, $2 \leq i \leq n$, is compared with each entry in the subarray $A[1..i-1]$. As to the number of element assignments, notice that there is an element assignment after each element comparison in the **while** loop. Moreover, there are $n-1$ element assignments of $A[i]$ to x in Step 2 of the algorithm. It follows that the number of element assignments is equal to the number of element comparisons plus $n - 1$.

The number of element comparisons performed by Algorithm INSERTIONSORT is between $n-1$ and $n(n-1)/2$. The number of element assignments is equal to the number of element comparisons plus $n-1$.

Insertion Sort Example
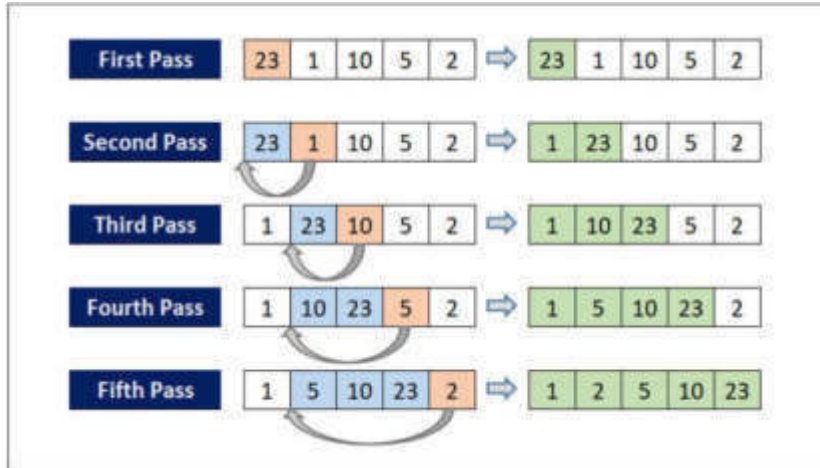
Analysis of Insertion sort:

- Simple implementation

- Efficient for small data sets

- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(n+d)$, where d is the number of inversions

- More efficient in practice than most other simple quadratic algorithms such as selection sort or bubble sort: the average running time is $n^2/4$, and the running time is linear in the best case

- Stable, i.e., does not change the relative order of elements with equal keys

- In-place, i.e., only requires a constant amount $O(1)$ of additional memory space

- Online, i.e., can sort a list as it receives it

- Worst case performance: $\theta\,(n^2)$

- Best case performance: $\theta\,(n)$

- Average case performance: $\theta\,(n^2)$

- Worst case space complexity: θ (*n*) total, θ (1) auxiliary

2.3.2 Merge Sort

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE(A, p,q,r), where A is an Array

p, q and r are indices into the array such that p<=q<r.

The procedure assume that the subarrays A[p…q] and [Aq+1…r] are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray A[p..r].

MERGE-SORT(A, p, r)

Input: An integer array A with indices p < r.

Output: The subarray A [p ... r) is sorted in non-decreasing order.

1   if r > p + 1

2      $q = \lfloor (p + r)/2 \rfloor$

3   MERGE-SORT (A, p, q)

4   MERGE-SORT (A, q, r)

5   MERGE (A, p, q, r)

Initial Call:

Merge Sort (A, 1, n+1)

Input: Array A with indices p, q, r such that

- p<q<r

- Subarrays A[p…q) and A[q…r) are both sorted

Output: The two sorted subarrays are merged into a single sorted subarray in A[p….r).

Pseudocode for MERGE

MERGE(A, p, q, r)

```
1   n₁ = q − p                    11   i = 1
2   n₂ = r − q                    12   j = 1
3   Create array L of size n₁ + 1  13   for k = p to r − 1
4   Create array R of size n₂ + 1  14       if L[i] ≤ R[j]
5   for i = 1 to n₁                15           A[k] = L[i]
6       L[i] = A[p + i − 1]        16           i = i + 1
7   for j = 1 to n₂                17       else A[k] = R[j]
8       R[j] = A[q + j − 1]        18           j = j + 1
9   L[n₁ + 1] = ∞
10  R[n₂ + 1] = ∞
```

Merge Sort example
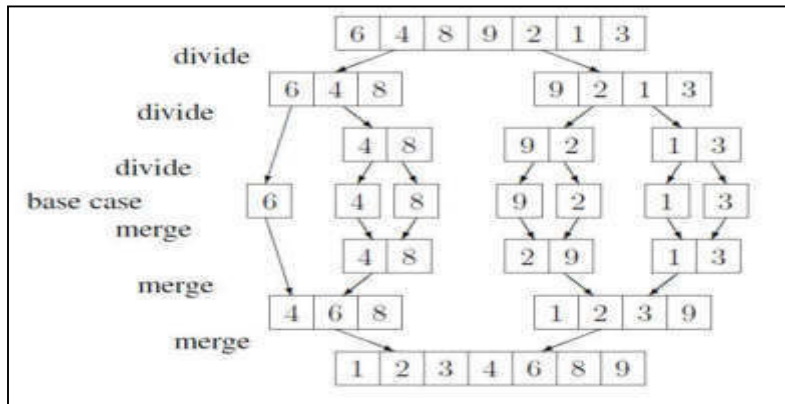
Analysis of merge sort:

- The worst-case running time of MERGE-SORT is θ (n log n), much better that the worst-case running time of INSERTION-SORT, which was θ (n^2) [11].

- MERGE-SORT is stable, because MERGE is left-biased

- MERGE and therefore MERGE-SORT is not in-place: it requires θ (n) extra space

- MERGE-SORT is not an online-algorithm: the whole array A must be specified before the algorithm starts running

2.3.3 Quick Sort

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition () makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

⇒ Repeatedly increase the pointer 'i' until a[i] >= pivot

⇒ Repeatedly decrease the pointer 'j' until a[j] <= pivot

⇒ If j > i, interchange a[j] with a[i]

⇒ Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

⇒ It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

⇒ Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . .x[high]

⇒ It calls itself recursively to sort the left sub-array x[low], x[low+1], ...x[j-1] between positions low and j-1 (where j is returned by the partition function).

⇒ It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

Algorithm QUICKSORT(low, high)

/* sorts the elements a(low), ... , a(high) which reside in the global array A(1 :
n) into ascending order a (n + 1) is considered to be defined and must be greater than all elements in a(1 : n); A(n + 1) = + ∝ */
{
    if low < high then
        {
            j := PARTITION(a, low, high+1);
               // J is the position of the partitioning
            element QUICKSORT(low, j – 1);
            QUICKSORT(j + 1 , high);
        }
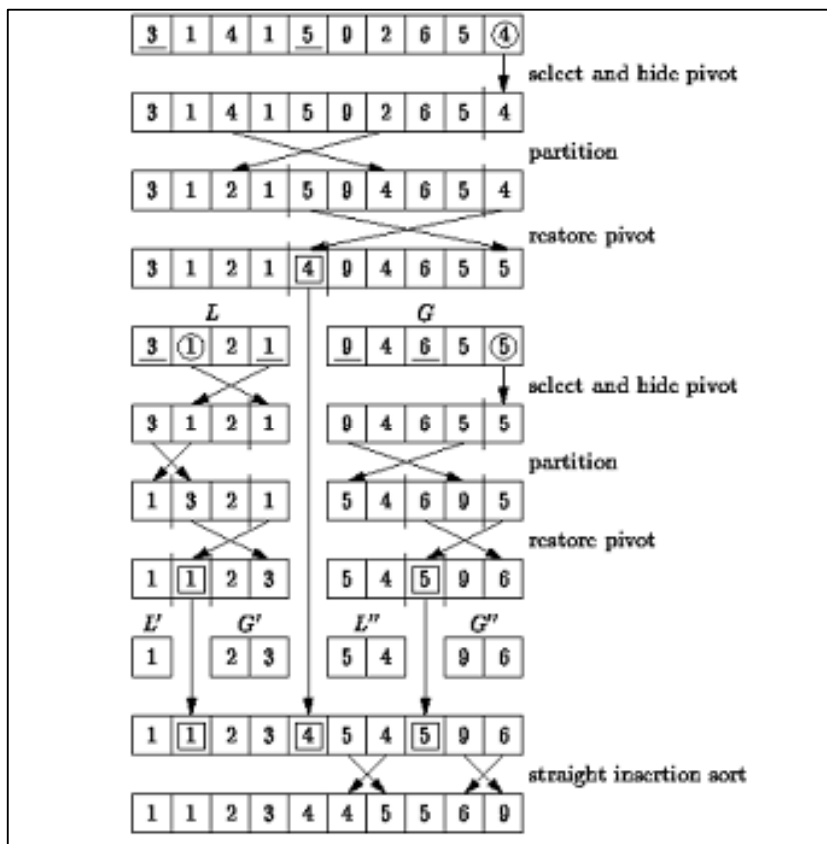}

Algorithm PARTITION(a, m, p)
{
    V ←a(m); i ←m; j← p; // A (m) is the partition element
    do
    {
        loop i := i + 1 until a(i) ≥ v // i moves left to right
        loop j := j – 1 until a(j) ≤v // p moves right to left
        if (i < j) then INTERCHANGE(a, i, j)
    } while (i ≥ j);
    a[m] :=a[j]; a[j] :=V; // the partition element belongs at
position P
    return j;
}

Algorithm INTERCHANGE(a, i, j)
{
    P:=a[i];
    a[i] := a[j];
    a[j] := p;
}

Quick Sort example

Analysis of Quicksort

- Best-case time efficiency: split in the middle — $\Theta(n \log n)$

- Worst-case time efficiency: sorted array! — $\Theta(n^2)$

- Average case time efficiency: random arrays — $\Theta(n \log n)$

- Space efficiency: not in-place — $\Theta(\log n)$ with a careful implementation

- Not stable

- Improvements:

  - better pivot selection: median-of-three partitioning

  - switch to insertion sort on small sub files or just stopping recursive calls when unsorted subarrays become small (say, <10 elements) and finish sorting with insertion sort

These yields about 20% improvement

- Considered the method of choice for sorting random files of nontrivial sizes

---

CHECK YOUR PROGRESS - II

c) In insertion Sort the number of comparisons depends on the order of the _____
d) MERGE-SORT is stable, because MERGE is _____
e) The arbitrary chosen element is termed as _____

2.4 MATRIX MANIPULATION PROBLEMS

Operations on matrices are at the heart of scientific computing. Efficient algorithms for working with matrices are therefore of considerable practical interest.

Suppose we had given a sequence of matrices to find the most efficient way to multiply those matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications. We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same.

2.4.1 Matrix Chain Multiplication

Let, we have three matrices A_1, A_2 and A_3, with order (10 x 100), (100 x 5) and (5 x 50) respectively.

Then the three matrices can be multiplied in two ways.

(i) First, multiplying A_2 and A_3, then multiplying A_1 with the resultant matrix i.e. $A_1(A_2 A_3)$.

(ii) First, multiplying A_1 and A_2, and then multiplying the resultant matrix with A_3 i.e. $(A_1 A_2) A_3$.

The number of scalar multiplications required in case 1 is

100 * 5 * 50 + 10 * 100 * 50 = 25000 + 50,000

= 75,000

and the number of scalar multiplications required in case 2 is

10 * 100 * 5 + 10 * 5 * 50 = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesize the expression in every possible fashion and count each time how many scalar multiplications are required. Thus the matrix chain multiplication problem can be stated as "*find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplications is minimized*".

For example, if we had four matrices A, B, C, and D, we would have:

$$(ABC)D = (AB)(CD) = A(BCD) = ...$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency.

For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

$(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500$ operations

$A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000$ operations.

Clearly the first parenthesization requires less number of operations.

Note that in the matrix chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest coast. Typically the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications.

2.4.2 Dynamic Programming Approach for Matrix Chain Multiplication

Dynamic programming is typically applied to *optimization problems.* In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed

to *the* optimal solution.

Our first example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication [6].

Let us consider a chain of *n* matrices $A_1, A_2.........A_n$, where the matrix A_i has dimensions $P[i-1]$ x $P[i]$

Suppose we take the parenthesisation at *k, this* results two sub chains $A_1.......A_k$ and $A_{k+1}.......A_n$. These two sub chains must each be optimal for $A_1......A_n$ to be optimal. The cost of matrix chain $(A_1....A_n)$ is calculated as

$cost(A_1......A_k) + cost(A_{k+1}.. A_n)$ + cost of multiplying two resultant matrices together i.e.

$cost(A_1......A_n) = cost(A_1......A_k) + cost(A_{k+1} A_n)$ + cost of multiplying two resultant matrices together.

Here, the cost represents the number of scalar multiplications. The sub chain $(A_1....A_k)$ has a dimension $P[0]$ x $P[k]$ and the sub chain $(A_{k+1}......A_n)$ has a dimension $P[k]$ x $P[n]$. The number of scalar multiplications required to multiply two resultant matrices is $P[0]$ x $P[k]$ x $P[n]$

Let $m[i, j]$ be the minimum number of scalar multiplications required to multiply the matrix chain $(A_i......... A_j)$. Then

    (i) $m[i, j] = 0$ if $i = j$

    (ii) $m[i, j]$ = minimum number of scalar multiplications required to multiply $(A_i....A_k)$ + minimum number of scalar multiplications required to multiply $(A_{k+1}....A_n)$ + cost of multiplying two resultant matrices i.e. $m[i, j] = m[i, k] + m[k, j] + P[i-1]$x $P[k]$x $P[j]$

However, we don't know the value of *k,* for which $m[i, j]$ is minimum. Therefore, we have to try all $j - i$ possibilities.

$$m[i, j] = \begin{cases} 0 & \text{if i=1} \\ \min\{m[i,k] + m[k, j] + P[i-1] \times P[k] \times P[j]\} & \\ \quad Otherwise & \\ \quad\quad i \leq k < j & \end{cases}$$

Therefore, the minimum number of scalar multiplications required to multiply n matrices $A_1 A_2 \ldots \ldots A_n$ is

$$m[1, n] = \min_{1 \le k < n} \left\{ m[1, k] + m[k, n] + P[0] \times P[k] \times P[n] \right\}$$

The dynamic programming approach for matrix chain multiplication is presented in the following algorithm:

Algorithm MATRIX-CHAIN-MULTIPLICATION (P)

```
// P is an array of length n+1 i.e. from P[0] to P[n]. It is assumed
that the matrix Aᵢ has the dimension P[i-1] ×P[i].
{
                for(i = 1; i<=n; i++)
                        m[i, i] = 0;
                for(l = 2; l<=n; l++){
                        for(i = 1; i<=n-(l-1); i++){
                                j = i + (l-1);
                                m[i, j] = ∞;
                        for(k = i; k<=j-1; k++)
                                q = m[i, k] + m[k+1, j] + P[i-1] P[k] P[j] ;
                                if (q<m [i, j]){
                                m[i, j] = q;
                                s[i, j] = k;
                                }
                        }
                }
        }
        return m and s
}
```

Now let us discuss the procedure and pseudo code of the matrix chain multiplication. Suppose, we are given the number of matrices in the chain is n i.e. $A_1, A_2........A_n$ and the dimension of matrix A_i is $P[i-1] \times P[i]$. The input to the matrix-chain-order algorithm is a sequence $P[n+1] = \{P[0], P[1], ...P[n]\}$. The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, ...n$ in lines 2-3. Then, the algorithm computes $m[i, j]$ for $j - i = 1$ in the first step to the calculation of $m[i, j]$ for $j - i = n$ -1 in the last step. In lines $3 - 11$, the value of

$m[i, j]$ is calculated for $j - i = 1$ to $j - i = n - 1$ recursively. At each step of the calculation of $m[i, j]$, a calculation on $m[i, k]$ and $m[k+1, j]$ for $i \leq k < j$, are required, which are already calculated in the previous steps.

To find the optimal placement of parenthesis for matrix chain multiplication A_i, A_{i+1}, …..A_j, we should test the value of $i \leq k < j$ for which $m[i, j]$ is minimum. Then the matrix chain can be divided from $(A_1 \ldots \ldots A_k)$ and $(A_{k+1} \ldots \ldots A_j)$.

Let us consider matrices $A_1, A_2 \ldots \ldots A_5$ to illustrate MATRIX-CHAIN-MULTIPLICATIONalgorithm. The matrix chain order P = {P_0, P_1, P_2, P_3, P_4, P_5} = {5, 10, 3, 12, 5, 50}. The objective is to find the minimum number of scalar multiplications required to multiply the 5 matrices and also find the optimal sequence of multiplications [7].

The solution can be obtained by using a bottom up approach that means first we should calculate m_{ii} for $1 \leq i \leq 5$. Then m_{ij} is calculated for $j - i = 1$ to $j - i = 4$.

The value of m_{ii} for $1 \leq i \leq 5$ can be filled as 0 that means the elements in the first row can be assigned 0. Then

For $j - i = 1$

$m_{12} = P_0 P_1 P_2 = 5 \times 10 \times 3 = 150$

$m_{23} = P_1 P_2 P_3 \quad = 10 \times 3 \times 12 = 360$

$m_{34} = P_2 P_3 P_4 = 3 \times 12 \times 5 = 180$

$m_{45} = P_3 P_4 P_5 = 12 \times 5 \times 50 = 3000$

For $j - i = 2$

$m_{13} \quad = \min \{m_{11} + m_{23} + P_0 P_1 P_3, m_{12} + m_{33} + P_0 P_2 P_3\}$

$= \min \{0 + 360 + 5 * 10 * 12, 150 + 0 + 5*3*12\}$

$= \min \{360 + 600,$

$150 + 180\} = \min \{960,$

$330\} = 330 m_{24} \quad =$

$\min \{m_{22} + m_{34} + P_1 P_2 P_4,$

$m_{23} + m_{44} + P_1 P_3 P_4\}$

$= \min \{0 + 180 + 10*3*5, 360 + 0 + 10*12*5\}$

$$= \min \{180 + 150,$$

$$360 + 600\} = \min \{330,$$

$$960\} = 330 \, m_{35} \qquad =$$

$$\min \{m_{33} + m_{45} + P_2 \, P_3 \, P_5,$$

$$m_{34} + m_{55} + P_2 \, P_4 \, P_5\}$$

$$= \min \{0 + 3000 + 3*12*50, \; 180 + 0 + 3*5*50\}$$

$$= \min \{3000 + 1800 + 180 + 750\} = \min \{4800, 930\} = 930$$

For $j - i = 3$

$m_{14} \qquad = \min \{m_{11} + m_{24} + P_0 \, P_1 \, P_4, \; m_{12} + m_{34} + P_0 \, P_2 \, P_4, \; m_{13} + m_{44} + P_0 \, P_3 \, P_4\}$

$$= \min \{0 + 330 + 5*10*5, \; 150 + 180 + 5*3*5, \; 330 + 0 + 5*12*5\}$$

$$= \min \{330 + 250, \; 150 + 180 + 75, \; 330 + 300\}$$

$$= \min \{580, 405, 630\} = 405$$

$m_{25} \qquad = \min \{m_{22} + m_{35} + P_1 \, P_2 \, P_5, \; m_{23} + m_{45} + P_1 \, P_3 \, P_5, \; m_{24} + m_{55} + P_1 \, P_4 \, P_5\}$

$$= \min \{0 + 930 + 10*3*50, \; 360 + 3000 + 10*12*50, \; 330 + 0 + 10*5*50\}$$

$$= \min \{930 + 1500, \; 360 + 3000 + 6000, \; 330 + 2500\}$$

$$= \min \{2430, 9360, 2830\} = 2430$$

For $j - i = 4$

$m_{15} \qquad = \min\{m_{11} + m_{25} + P_0 \, P_1 \, P_5, \; m_{12} + m_{35} + P_0 \, P_2 \, P_5, \; m_{13} + m_{45} + P_0 \, P_3 \, P_5, \; m_{14} + m_{55} + P_0 \, P_4 \, P_5 \}$

$$= \min\{0 + 2430 + 5*10*50,$$

$$150 + 930 + 5*3*50,$$

$$330 + 3000 + 5*12*50,$$

$$405 + 0 + 5*5*50\}$$

$$= \min \{2430 + 2500, \; 150 + 930 + 750, \; 330 + 3000 + 3000, \; 405 + 1250\}$$

$$= \min \{4930, 1830, 6330, 1655\} = 1655$$

Hence, minimum number of scalar multiplications required to multiply the given five matrices in 1655.

To find the optimal parenthesization of A_1..........A_5, we find the value of k is 4 for which m15 is minimum. So the matrices can be splitted to $(A_1$....$A_4)$ (A_5). Similarly, $(A_1$....$A_4)$ can be splitted to (A_1A_2) $(A_3$ $A_4)$ because for $k = 2$, m14 is minimum. No further splitting is required as the sub chains (A_1A_2) and $(A_3$ $A_4)$ has length 1. So the optimal paranthesization of A_1 ....A_5 in $((A_1$ $A_2)$ $(A_3$ $A_4)$ $)$ (A_5).

From the above solution of the given problem, we can see that all possible ways of obtaining the parenthesization of a matrix chain product using dynamic programming are performed. In other words, all possible solutions are obtained, and from those solutions, the optimal solution is taken, we have selected only those solutions that provide the least or minimum value, which can be reflected in the minimum cost table, as shown in Fig. 1. The respective k values are included in the split table, as shown in Fig. 2.
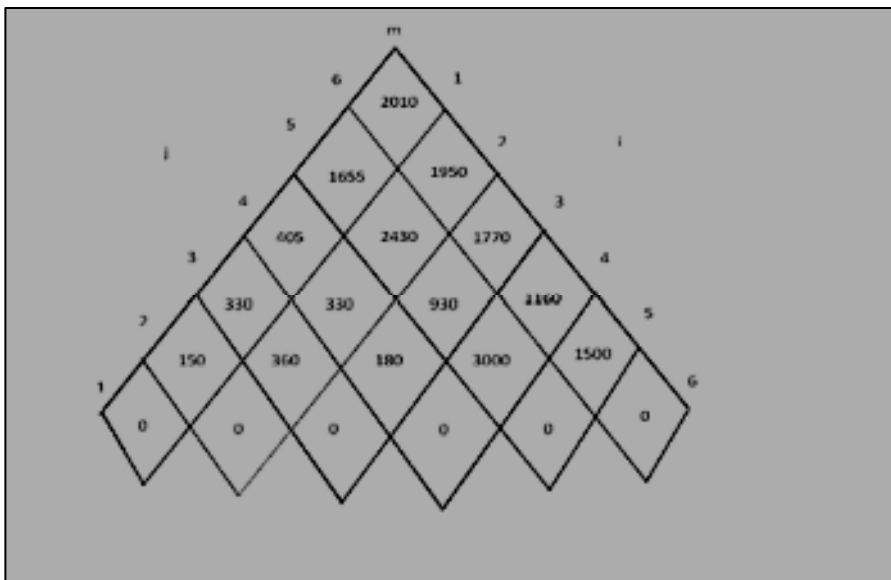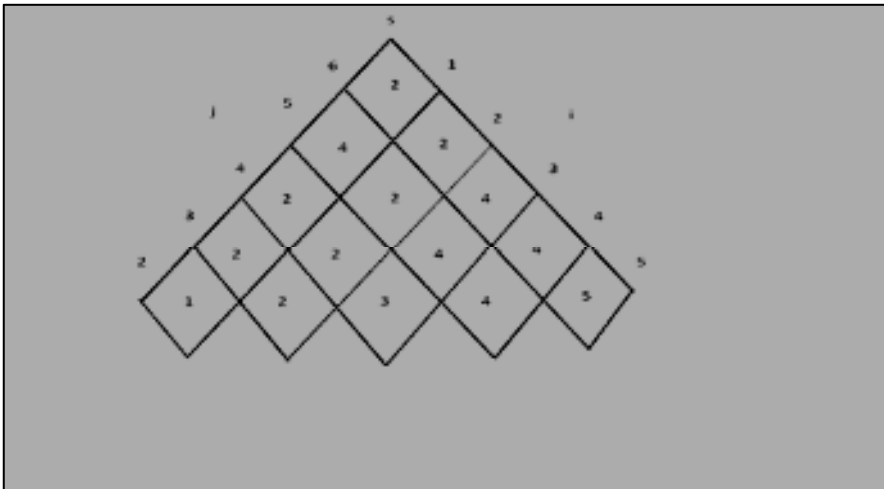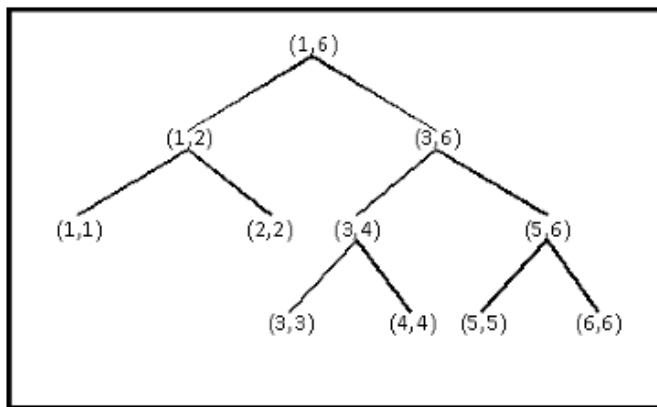

Fig 1: Minimum cost table

Fig 2: Split table

Tree for optimal parenthesization

Complexity of Matrix Chain Product:

- The time complexity of matrix chain product is $O(n^3)$
- The space complexity of matrix chain product is $O(n^2)$

Time complexity of multiplying a chain of *n* matrices

Let $T(n)$ be the time complexity of multiplying a chain of *n* matrices.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ \Theta(1) + \sum_{k=1}^{n-1}[T(k)+T(n-k)+\Theta(1)] & \text{if } n>1 \end{cases}$$

$$\Rightarrow T(n) = \Theta(1) + \sum_{k=1}^{n-1}[T(k)+T(n-k)+\Theta(1)] \quad \text{if } n>1$$

$$= \Theta(1) + \Theta(n-1) + \sum_{k=1}^{n-1}[T(k)+T(n-k)]$$

$$\Rightarrow T(n) = \Theta(n) + 2[T(1)+T(2)+\cdots\cdots+T(n-1)]\cdots\cdots(7.1)$$

Replacing *n* by *n-1*, we get

$$T(n-1) = \Theta(n-1) + 2[T(1)+T(2)+\cdots\cdots+T(n-2)]\cdots\cdots(7.2)$$

Subtracting equation 7.2 from equation 7.1, we have

$$T(n) - T(n-1) = \Theta(n) - \Theta(n-1) + 2T(n-1)$$
$$\Rightarrow T(n) = \Theta(1) + 3T(n-1)$$

$$= \Theta(1) + 3[\Theta(1)+3T(n-2)] = \Theta(1)+3\Theta(1)+3^2T(n-2)$$

$$= \Theta(1)[1+3+3^2+\cdots\cdots+3^{n-2}]+3^{n-1}T(1)$$

$$= \Theta(1)[1+3+3^2+\cdots\cdots+3^{n-1}]$$

$$= \frac{3^n-1}{2} = O(2^n)$$

2.4.3 Divide and Conquer Strategy for Matrix Multiplication

Here, we will see how divide and conquer technique work as a new algorithm for multiplying matrix.

Let X and Y be n × n matrices

$$X = \begin{pmatrix} x_{11} & x_{12} & x_{1n} \\ x_{21} & x_{22} & x_{1n} \\ x_{n1} & x_{n2} & x_{nn} \end{pmatrix}$$

$\Rightarrow$ We want to compute Z=X.Y

$$z_{ij} = \sum_{k=1}^{n} X_{ik}.Y_{kj}$$

$\Rightarrow$ Naive method uses $n^2.n=\theta(n)^3$ operations

Divide-and-Conquer solution:

$$Z = \begin{Bmatrix} A & B \\ C & D \end{Bmatrix} \cdot \begin{Bmatrix} E & F \\ G & H \end{Bmatrix} = \begin{Bmatrix} (A.E+B.G) & (A.F+B.H) \\ (C.E+D.G) & (C.F+D.H) \end{Bmatrix}$$

$\Rightarrow$ The above naturally leads to divide-and-conquer solution:
   o Divide X and Y into 8 sub-matrices A, B, C, and D
   o Do 8 matrix multiplications recursively
   o Compute Z by combining results (doing 4 matrix additions)
$\Rightarrow$ Let's assume n = 2^c for some constant c and let A, B, C and D be n/2 × n/2 matrices
   o Running time of algorithm is T(n) = 8T(n/2) + θ (n2))
                                    $\Rightarrow$ T(n) = θ (n3)

2.4.3.1 Strassen's Matrix Multiplication

By using divide-and-conquer approach proposed by Strassen in 1969, we can reduce the number of multiplications.

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

   for i := 1 to n do

         for j :=1 to n do

         c[i, j] := 0;

         for K: = 1 to n do

               c[i, j] := c[i, j] + a[i, k] * b[k, j];

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon. We apply divide and conquer to this problem.

For example, let us considers three multiplications like:

$$Z = \begin{Bmatrix} A & B \\ C & D \end{Bmatrix} \cdot \begin{Bmatrix} E & F \\ G & H \end{Bmatrix} = \begin{Bmatrix} (S_1 + S_2 - S_4 + S_6) & (S_4 + S_5) \\ (S_6 + S_7) & (S_2 + S_3 + S_5 - S_7) \end{Bmatrix}$$

S1 = (B − D) · (G + H)

S2 = (A + D) · (E + H)

S3 = (A − C) · (E + F)

S4 = (A + B) · H

S5 = A · (F − H)

S6 = D · (G − E)

S7 = (C + D) · E

Let's test that S6 + S7 is really C · E + D · G

S6 + S7   = D · (G − E) + (C + D) · E

         = DG − DE + CE + DE

         = DG + CE

This leads to a divide-and-conquer algorithm with running time T(n) = 7T(n/2) + θ (n2)

     – We only need to perform 7 multiplications recursively.

     – Division/Combination can still be performed in θ(n2) time.

Let's solve the recurrence using the iteration method

$$\begin{aligned}
T(n) &= 7T(n/2) + n^2 \\
&= n^2 + 7(7T(\tfrac{n}{2^2}) + (\tfrac{n}{2})^2) \\
&= n^2 + (\tfrac{7}{2^2})n^2 + 7^2 T(\tfrac{n}{2^2}) \\
&= n^2 + (\tfrac{7}{2^2})n^2 + 7^2(7T(\tfrac{n}{2^3}) + (\tfrac{n}{2^2})^2) \\
&= n^2 + (\tfrac{7}{2^2})n^2 + (\tfrac{7}{2^2})^2 \cdot n^2 + 7^3 T(\tfrac{n}{2^3}) \\
&= n^2 + (\tfrac{7}{2^2})n^2 + (\tfrac{7}{2^2})^2 n^2 + (\tfrac{7}{2^2})^3 n^2 .... + (\tfrac{7}{2^2})^{\log n - 1} n^2 + 7^{\log n} \\
&= \sum_{i=0}^{\log n - 1} (\tfrac{7}{2^2})^i n^2 + 7^{\log n} \\
&= n^2 \cdot \Theta((\tfrac{7}{2^2})^{\log n - 1}) + 7^{\log n} \\
&= n^2 \cdot \Theta(\tfrac{7^{\log n}}{(2^2)^{\log n}}) + 7^{\log n} \\
&= n^2 \cdot \Theta(\tfrac{7^{\log n}}{n^2}) + 7^{\log n} \\
&= \Theta(7^{\log n})
\end{aligned}$$

Now we have the following:

$$\begin{aligned}
7^{\log n} &= 7^{\frac{\log_7 n}{\log_7 2}} \\
&= (7^{\log_7 n})^{(1/\log_7 2)} \\
&= n^{(1/\log_7 2)} \\
&= n^{\frac{\log_2 7}{\log_2 2}} \\
&= n^{\log 7}
\end{aligned}$$

Or, in general: $a^{\log_k n} =: n^{\log_k n}$

So the solution is $T(n) = \theta(n^{\log 7}) = \theta(n^{2.81...})$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm.

2.5 KNAPSACK PROBLEM

The knapsack problem is an example of a combinational optimization problem, a topic in mathematics and computer science about finding the optimal object or finite solution where an exhaustive search is not possible among a set of objects. The problem can be found real-world scenarios like resource allocation in financial constraints or even in selecting investments and portfolios; and also be found in fields such as applied mathematics, complexity theory, cryptography, combinatorics and computer science.

In the knapsack problem, the given items have two attributes at minimum – an item's value, which affects its importance, and an item's weight or volume, which is its limitation aspect. Since an exhaustive search is not possible, one can break the problems into smaller sub-problems and run it recursively. This is called an optimal sub-structure. This deals with only one item at a time and the current weight still available in the knapsack. The problem solver only needs to decide whether to take the item or not based on the weight that can still be accepted. However, if it is a program, re-computation is not independent and would cause problems. This is where dynamic programming techniques can be applied. Solutions to each sub-problem are stored so that the computation would only need to happen once [8].

There are n items, i^{th} item is worth v_i dollars and weight w_i pounds, where v_i and w_i are integers. Select item to put in knapsack with total weight is less than W, So that the total value is maximized. This problem is called knapsack problem. This problem finds which items should choice from n item to obtain maximum profit and total weight is less than W.

The problem can be explained as follows-

"A thief robbing a store finds n items, the ith item is worth v_i dollar and weight w pounds, where v_i and w_i are integers. He wants to take as valuable load as possible, but he can carry atmost W pounds in his knapsack, where W is an integer. Which item should he take"

There are two types of knapsack problem.

1. ***0-1 knapsack problem****:*

In **0**-1 knapsack problem each item either be taken or left behind.

2. *Fractional knapsack problem*:

In fractional knapsack problem fractions of items are allowed to choose.

2.5.1 Greedy Strategy Applied in 0-1 KNAPSACK Problem

The greedy algorithm in 0-1 knapsack problem can be applied as follows-

1. *Greedy choice*:

Take an item with maximum value per pound.

2. *Optimal substructure*:

Consider the most valuable load that weights atmost W pounds. These W pounds can be choose from n item. If j^{th} item is choose first then remaining weight $W-w_i$ can be choose from n-1 remaining item excluding j.

2.5.2 Greedy Strategy Applied in Fractional KNAPSACK Problem

The greedy algorithm in fractional Knapsack problem can be applied as follows-

1. *Greedy choice:*

Take an item or fraction of item with maximum value per pound.

2. *Optimal substructure*:

If we choose a fraction of weight w of the item j, then the remaining weight atmost W-w can be choose from the n-1 item plus w_i-w pounds of item j.

Although, both the problems are similar, the fractional knapsack problem is solvable by greedy strategy, but 0-1 knapsack problem are not solvable by greedy algorithm.

Consider the following problem-

There are 3 items. The knapsack can hold 50 pounds. Item1 weight 10 pounds and its worth is 60 dollar, item2 weight 20 pounds and its worth 100 dollars, item3 weight 30 pounds and its weight 120 dollars. Find out the items with maximum profit which the knapsack can carry.

Solution:

Here,

W = 50 pounds

Item	Weight (w pound)	Worth (v dollar)
Item1	10	60
Item2	20	100
Item3	30	120

Let, an item I has weight w_i pounds and worth v_i dollar

Value per pound of I = v_i / w_i .

Thus, value per pound for-

$\qquad$ Item1 = w_1 / v_1

$\qquad\qquad$ = 60 dollars / 10 pounds

$\qquad\qquad$ = 6 dollars/pounds

$\qquad$ Item2 = w_2 / v_2

$\qquad\qquad$ = 100 dollars / 20 pounds

$\qquad\qquad$ = 5 dollars/pounds

$\qquad$ Item3 = w_3 / v_3

$\qquad\qquad$ = 120 dollars / 30 pounds

$\qquad\qquad$ = 4 dollars/pounds

We can select maximum of 50 pounds.

So, using greedy strategy in 0-1 knapsack problem

1st choice is Item1.

2nd choice is Item2.

Total weight = 10 + 20 pounds

= 30 pounds

Total worth = 60 +100 dollars

=160 dollars

But this is not the optimal choice.

The optimal choice will choose item 2 and 3. Then,

Total weight = 20 + 30 pounds

= 50 pounds

Total worth = 100 + 120 dollars

= 220 dollars

Hence, 0-1 knapsack problem is not solved by greedy strategy.

Now, using greedy strategy in fractional knapsack problem –

1st choice is item1.

2nd choice is item2

Total weight = 30 pounds

But the size of the knapsack is 50 pounds.

So, it will take remaining 20 pounds from item3 (fraction of item3) and its worth is 4 x 20=80 dollars.

Hence,

Total weights = 50 pounds.

Total worth = 60+100+ 80 dollars

=240 dollars.

Hence, an optimal solution can be obtained from fractional knapsack problem using greedy strategy.

2.5.3 Dynamic Programming applied in 0/1 KNAPSACK Problem

In the above topic, we have discussed about the Knapsack problem, and found that fractional knapsack problem can be solved by using greedy strategy. The 0-1 knapsack problem can only be solved by using dynamic programming. Below we will discuss methods for solving 0-1 knapsack problem.

The naive way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack. But, we can find a dynamic programming algorithm that will usually do better than this brute force technique.

Our first attempt might be to characterize a sub-problem as follows:

Let S_k be the optimal subset of elements from $\{I_0, I_1,...I_k\}$. But what we find is that the optimal subset from the elements $\{I_0, I_1,... I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1,...I_k\}$ in any regular pattern. Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

To illustrate this, consider the following example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

The maximum weight the knapsack can hold is 20.

The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$ but the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$. In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$. Instead it builds upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12.

So, now, let us rework on our example with the following idea

Let $B[k, w]$ represents the maximum total value of a subset S_k with weight w. Our goal is to find $B[n, W]$, where n is the total number of items and W is the maximal weight, the knapsack can carry.

Using this definition, we have $B[0, w] = w_0$, if $w \geq w_0$.

$= 0$, otherwise

Now, we can derive the following relationship that $B[k, w]$ obeys: $B[k, w] = B[k - 1, w]$, if $w_k > w$

$= \max \{ B[k - 1, w], B[k - 1, w - w_k] + v_k\}$

In general:

1) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ...I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight w, if weights of item k is greater than W.

   Basically, we can NOT increase the value of our knapsack with weight w if the new item we are considering weighs more than W – because it WON'T fit!!!

2) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_1, I_2, ... I_{k-1}\}$ with weight w, if item k should not be added into the knapsack.

3) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight $w-w_k$, plus item k.

You need to compare the values of knapsacks in both case 2 and 3 and take the maximal one.

Recursively, we will still have an $O(2^n)$ algorithm. But, using dynamic programming, we simply perform in just two loops - one loop running n times and the other loop running W times.

Here is a dynamic programming algorithm to solve the 0/1 Knapsack problem:

Input: S, a set of n items as described earlier, W the total weight of the knapsack. (Assume that the weights and values are stored in separate arrays named w and v, respectively.)

Output: The maximal value of items in a valid knapsack. int i, k;

for (i=0; i<= W; i++)

  B[i] = 0

  for (k=0; k<n; k++)

  {

    for (i = W; i>= w[k]; i--)

   {

     if (B[i – w[k]] + v[k]> B[i])

       B[i] = B[i – w[k]] + v[k]

  }

}

Clearly the run time of this algorithm is $O(nW)$, based on the nested loop structure and the simple operation inside of both loops. When comparing this with the previous $O(2^n)$, we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient.

2.5.4 Backtracking Method for 0-1 KNAPSACK

Problem Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of the weights such that:

$$\sum_{1 \leq i \leq n} w_i x_i \leq m \quad and \sum_{1 \leq i \leq n} P_i x_i \text{ is maximized}$$

The x_i's constitute a zero–one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's.

Bounding functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If a upper bound for a live node is not higher than the value of the best solution then the node can bounded or kill.

If we consider for a node Z the values of x_i, $1 < i < k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements $x_i = 0$ or 1.

A recursive function for 0-1 knapsack using backtracking:

/* bounding function for 0-1 knapsack*/

```
float Bound ( float cp, float cw, int k )
{
    float b = cp, c = cw;
    for ( int i = k + 1; i ≤ n; i++ )c
    = c + w [ i ];
     if ( c < m )
     b = b + p [ i ];else

 return ( b + ( 1 - ( c – m ) / w [ i ] * p [ i ] );
    }
 return(b);
}
```

/* Backtracking method for 0-1 knapsack*/

```
  void Knap( int k, float cp, float cw)
  {
    if ( cw + w [ k ] ≤ m )
      y [ k ] = 1;
       if ( k < n )
         Knap ( k + 1, cp + p [ k ], cw + w [ k ] );
       if (( cp + p [ k ] > fp ) && ( k == n ))
          {
            fp = cp + p [ k ]; fw =
            cw + w [ k ];
             for ( int j = 1; j ≤ k ; j++)x [
             j ] = y [ j ];
          }
    }
if ( Bound (cp, cw, k ) ≥ fp )
    {
      y [ k ] = 0;
       if ( k < n )
         Knap ( k + 1, cp, cw );
          if (( cp > fp ) &&( k == n ))
            {
              fp = cp; fw =
              cw ;
               for ( int j = 1; j ≤ k; j++)


            x [ j ] = y [ j ];
              }
          }
        }
```

Here,

cp = current total profit of the chosen items

cw = current total weight of all chosen items

k = index of last considered item

m = capacity of knapsack

w[i] = weight of ith item

p[i] = profit of ith item.

P[i]/w[i] ≥ p[i+1]/w[i+1], for all 1≤i<n

n= total item numbers

fw= final total weights in knapsack

fp= final maximum profit

x[k]   ==0, if w[k] is not in knapsack,

      == 1 ,Otherwise

The above method to determine an upper bound for a node at level k+1 of state space tree, function Bound(cp, cw, k) is used.

Initially fp is set to -1. This method s invoked by Knap(1,0,0). When fp != -1 , x[k],1≤k<n , is such that $\sum_{i=1...n} p[i]x[i] = fp$

The path y[j],1 ≤ j ≤ k is the path to the current node. The current weight

$$cw = i = \sum_{i=1......k-1} w[i]y[i]$$

The current profit $cp = \sum_{i=1....k=1} p[i]y[i]$

2.5.5 Solving Knapsack Problem using Branch and Bound

Let us consider a knapsack of size **K** and we want to select a set of objects from **n** objects , where the i[th] object has size s_i and value v_i such that it maximizes the value contained in the knapsack with the contents of the knapsack less than or equal to K.

Suppose that K = 16 and n = 4, and we have the following set of objects ordered by their value density.

i	v_i	s_i	v_i/s_i
1	$45	3	$15
2	$30	5	$ 6
3	$45	9	$ 5
4	$10	5	$ 2

Firstly we begin the state space tree with the root consisting of the empty Knapsack. The current weight and value are obviously. To find the maximum potential value we consider the problem as if it was the fractional knapsack problem and we were using the greedy algorithmic solution to that problem. We have already discussed that the greedy approach to the fractional knapsack problem yields an optimal solution. We place each of the remaining objects, into the knapsack until the next selected object is too big to fit into the knapsack. We then use the fractional amount of that object that could be placed in the knapsack to determine the maximum potential value.

totalSize = currentSize + size of remaining objects that can be fully placed

bound (maximum potential value) = currentValue + value of remaining objects fully placed + (K - totalSize) * (value density of item i.e partially placed)

In general, for a node at level i in the state space tree the first i items have been considered for selection and for the kth object as the one that will not completely fit into the remaining space in the knapsack, these formulae can be written:

totalsize= currentsize + $\sum_{j=j+1}^{k-1} S_i$

bound= currentvalue+ $\sum_{j=j+1}^{k-1} V_i + (K - totalsize) * (V_k \, S_k)$

For the root node currentSize = 0 and currentValue = 0

totalSize $= 0 + s_1 + s_2 = 0 + 3 + 5 = 8$

bound $= 0 + v_1 + v_2 + (K - totalSize) * (v_3/s_3)$

$= 0 + \$45 + \$30 + (16 - 8) * (\$5)$

$= \$75 + \$40$

$= \$115$

The computation of the bound and the selection criteria for promising nodes is the same as before. We must replace the depth-first traversal of the state space tree with a breadth first traversal. In the depth-first traversal the auxiliary data structure used to store the nodes was the

stack. In breath-first traversal, the auxiliary data structure is explicitly the queue.

2.6   JOB SEQUENCING WITH DEADLINE

Now, we will discuss about the job sequencing problem. The problem is stated as below-

1. There are n jobs to be processed on a machine

2. Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$

3. p_i is earned iff the job is completed by its deadline

4. To complete the job, it is processed in one machine for a unit of time

5. Only one machine is available for processing job

6. Only one job is processed at a time on the machine

7. A feasible solution is a subset of job J such that each job is completed by its deadline

8. An optimal solution is a feasible solution with a maximum profit

This problem can be solved by greedy algorithm. For the optimal solution, after choosing a job, it will add the next job to the subset such that $\sum_{i \in J} p_i$, increases and resulting subset become feasible. p_i is the total profit of ith subset of jobs. In other words we have to check all possible feasible subset J with their total profit value, for a given set of jobs.

Feasible solution for a set of job J is such that, if the jobs of set J can be processed in the order without violating any deadline then J is a feasible solution.

Algorithm for job sequencing:

Input: A is the array of jobs with deadline

1. Begin

2. Sort all the jobs based on profit P_i so

3. $P_1 > P_2 > P_3$ …………………………….>=P_n

4. d = maximum deadline of job in A

5. Create array J[1,…………………,d]

6. For i=1 to n do

   7. Find the largest job x

   8. For j=i to 1

     9. If ((J[j] = 0) and (x deadline<= d))

    10. Then

      11. J[x] = i;

      12. Break;

     13. End if

    14. End for

  15. End for

  16. End

Output: Profit J array will be the output

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution. The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

Step-01:

- Sort all the given jobs in decreasing order of their profit

Step-02:

- Check the value of maximum deadline

- Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-03:

- Pick up the jobs one by one

- Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline

Time complexity: Job sequencing problems has the time complexity of $O(n^2)$

Example :

Let , there are n=4 nos. of job and jobs are 1, 2, 3, 4

profit (p1,p2,p3,p4) = (100,10,15,27)

deadline (d1,d2,d3,d4) = (2,1,2,1)

Find the optimal solution set.

Jobs	Profit		Deadline	
1	p1	100	d1	2
2	p2	10	d2	1
3	p3	15	d3	2
4	p4	27	d4	1

Solution:

Step 1: Sorting all jobs according to profit

Jobs	Profit		Deadline	
2	p2	10	d2	1
3	p3	15	d3	2
4	p4	27	d4	1
1	p1	100	d1	2

Step 2: Here maximum deadline is 2

Now we draw a Gantt chart with maximum time on Gantt chart = 2 units.

Now,

- We take each job one by one in the order they appear in Step-01.

- We place the job on Gantt chart as far as possible from 0.

Step 3:

- We take job p4.

- Since its deadline is 1 and has the maximum profit than p2 (which has also deadline=1, so we place it in the first empty cell before deadline 1 as-

0	1
p4	
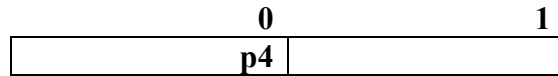
Step 4:

- We take job p1.

- Since its deadline is 2 and has the maximum profit than p3 (which has also deadline=1, so we place it in the first empty cell before deadline 2 as-

0	1
p4	**p1**
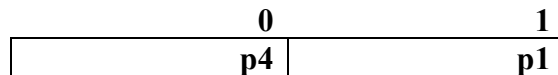
The optimal schedule is- p4, p1

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

= Profit of job p4 + Profit of job p1

= 27+100

=127 units

Following is a table with all possible feasible solution with processing sequences and profit of each sequence. From this table now we can understand that why (p4, p1) is the best optimal solution and why other combinations were not considered as optimal.

Sl. No.	Feasible Solution	Processing Sequence	Profit
1	(2,1)	(1,2)	110
2	(1,3)	(1,3) or(3,1)	115
3	(1,4)	(4,1)	127
4	(2,3)	(2,3)	25
5	(3,4)	(4,3)	42
6	(1)	(1)	100
7	(2)	(2)	10
8	(3)	(3)	15
9	(4)	(4)	27

Here solution 3 is optimal. The optimal solution is got by processing the job 1 and 4 in the order job 4 followed by job 1. The maximum profit is 127. Thus, the job 4 begins at time zero and job 1 end at time 2. Consider solution 3 i.e maximum profit job subset J = (1, 4)

Here , at first J= Ø and $\sum_{i \in J} p_i = 0$.

Job 1 is added to J as it has the largest profit and is a feasible solution.

Next add job 4 .Then also J = (1,4) is feasible because if the job processes in the sequence (4,1) then job 4 will start in zero time and job 1 will finish in 2 time within its deadline.

Next if job 3 is added then j=(1,3,4) is not feasible because all the job 1,3,4 cannot be completed within its deadline. So job 3 is not added to the set.

Similarly after adding job 2 J= (1,2,4) is not feasible.

Hence J = (1, 4) is a feasible solution set with maximum profit 127. This is an optimal solution.

2.7   SET MANIPULATION PROBLEM

The set union problem has been widely studied during the past decades. Here we will discuss the use of forests in the representation of sets [2]. Some applications involve grouping n distinct elements into a collection of disjoint sets. These applications often need to perform two operations in particular:

⇒ Finding the unique set that contains a given element

⇒ Uniting two sets

2.7.1 Disjoint-Set Operation

A disjoint-set data structure maintains a collection of set S= { $S_1, S_2 ... ... S_k$} of disjoint dynamic sets [1]. We identify each set by a representative, which is some member of the set. Some application doesn't mind about which member is used as the representative; only have to care that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the answer both times. Other application may require a pre-specified rule for choosing the representative, such as choosing the smallest member in the set.

Here we will assume that the elements of the sets are the numbers 1,2,3,….n. These numbers indicates into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (i.e. if S_i and S_j, $i \neq j$, are two sets then there is no element that is in both S_i and S_j).

For example, when n=10, the elements can be partitioned into three disjoint sets, $S_1 = \{1,7,8,9\}$, $S_2 = \{2,5,10\} and S_3 = \{3,4,6\}$.

Following fig., shows one possible representation for these sets:
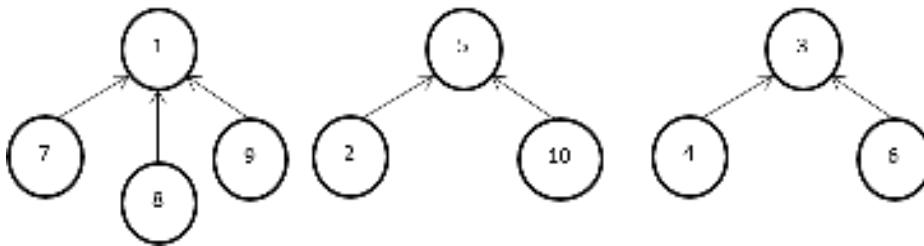
Possible tree representation of sets S_1, S_2, S_3

Here the usual method for representing child- parent relationship is not used; instead the links are maintained from child to parent. Now the operations we wish to perform on these sets are:

1. Disjoint set union: If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elemets x such that x is in $S_i or S_j$. Thus $S_1 \cup S_2 = $ {1,7,8,9,2,5,10}. Since we have assume that all sets are disjoint, we can assume that following the union of S_i and S_j , the sets $S_i and S_j$ do not exist independently, i.e. they are replaced by $S_i \cup S_j$ in the collection of sets.

2. Find(i): Given the element i , find the set containing i. Thus, 4 is in set S_3 and 9 is in set S_1.


2.7.2 Union and Find Operation

Suppose, we wish to obtain the union of S_1 and S_2 from the fig 1. Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_1$ could then have one of the representation as shown int the following fig.
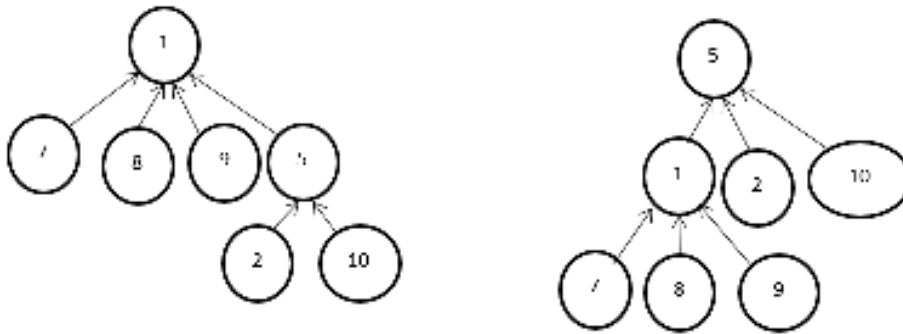
Possible tree representation of $S_1 \cup S_2$

To obtain the union of two sets, we have to set the parent field of one of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set. If each root has a pointer to the set name, them to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name. The data representation for S_1, S_2, S_3 may then take the following form shown in fig 3.
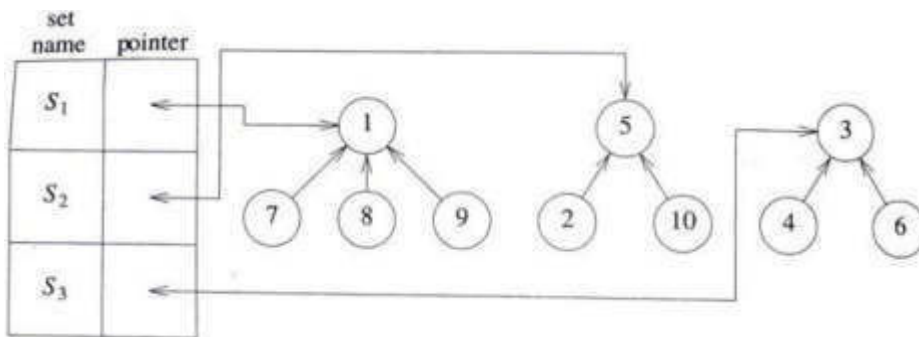
Data representation for S_1, S_2, and S_3

Since the set elements are numbered 1 through n, we can represent, the tree nodes using an array p [1 : n], where n is the maximum number of elements. The ith element of this array represents the tree node that contains element i. The array elements give the parent pointer of the corresponding tree node. Fig 4 shows representation of sets S_1, S_2, and S_3, where the root node have parent -1.

I	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Array representation of S_1, S_2, and S_3 of fig 1

Now we can implement *Find(i),* by following the indices, starting at i until we reach a node with parent value -1. For example Find(6) starts at 6 and then moves to 6's parent 3. Since p[3] is –ve, we have reached the root. The operation Union(i, j) is also equally simple, we pass the two trees root i and j, by adopting the convention first tree become the sub-tree of the second, the statement p[i]=j; accomplished the union.

An algorithm for "union and find" gives the description of the union and find operation. Although these two algorithms are very easy to state, their performance characteristics are not very good. For instance, if we start with q elements each in a set of its own (i.e. $S_i = \{i\}, 1 \le i \le q$), them the initial configuration comnsists of a forest with q nodes, and $p[i] = 0, 1 \le i \le q$.

Algorithm: Simple algorithms for union

***Step* 1**: Algorithm SimpleUnion(i,j)

Step 2: {

***Step* 3**: $p[i] = j$;

***Step* 4**: }

Algorithm: Simple algorithms for find

***Step* 1**: Algorithm SimpleFind(i)

Step 2: {

***Step* 3**: $while\ (p[i] \ge 0\ \ do\ i := p[i]$;

Step 4: return i;

Step 5: }

Since the time taken for a union is constant, the n-1 unions can be processed in time $O(n)$. However, each find requires following a sequence of parent pointers from the element to be found to the root. Since the time required to process a find for an element at level i of a tree is $O(i)$, the total time needed to process the n finds is $O(\sum_{i=1}^{n} i) = O(n^2)$.

2.8 DYNAMIC STORAGE ALLOCATION

Dynamic memory allocation has been a fundamental part of most computer systems since roughly 1960, and memory allocation is widely considered to be either a solved problem or an insoluble one.

Many algorithms require dynamic allocation of memory while the program is running. For example, a text editor may choose to store text a line at a time. Typically, each line would be allocated as it is created or modified, so that it only consumer as much space as required, without imposing arbitrary restrictions on the length of lines. Or it may choose to allocate and store the file in even smaller contiguous pieces, to minimize the number of characters that need to be moved in memory after an update. Similarly, many numerical programs will need to allocate sections of memory whose size or number depends on the particular input to the program [9].

Dynamic memory allocation is when an executing program requests that the operating system give it a block of main memory. The program then uses this memory for some purpose. Usually the purpose is to add a node to a data structure. In object oriented languages, dynamic memory allocation is used to get the memory for a new object.

The memory comes from above the static part of the data segment. Programs may request memory and may also return previously dynamically allocated memory. Memory may be returned whenever it is no longer needed. Memory can be returned in any order without any relation to the order in which it was allocated. The heap may develop

"holes" where previously allocated memory has been returned between blocks of memory still in use.
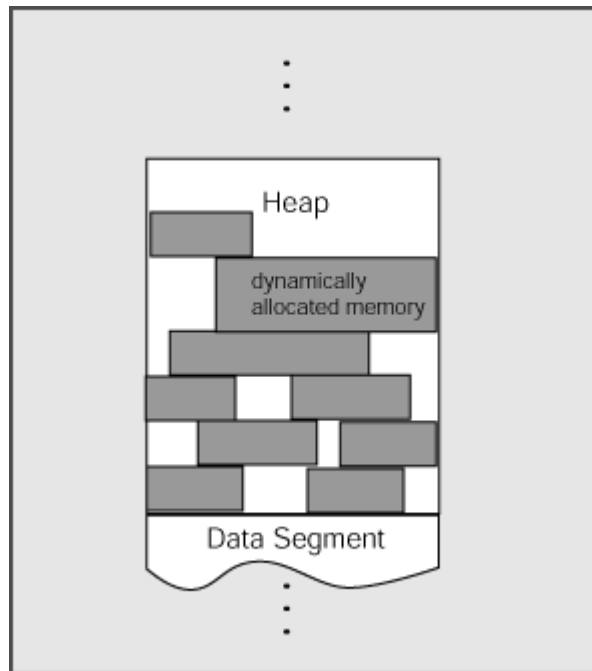
A new dynamic request for memory might return a range of addresses out of one of the holes. But it might not use up all the hole, so further dynamic requests might be satisfied out of the original hole.

If too many small holes develop, memory is wasted because the total memory used by the holes may be large, but the holes cannot be used to satisfy dynamic requests. This situation is called **memory fragmentation**. Keeping track of allocated and deallocated memory is complicated. A modern operating system does all this.

2.8.1 Garbage Collection

"Garbage Collection, also known as automatic memory management, is the automatic recycling of heap memory. Garbage Collection is performed by a garbage collector which recycles memory that it can prove will never be used again. Systems and languages which use Garbage Collection can be described as garbage-collected."

Garbage refers to those memory blocks that are allocated but not in use, these objects are dead. The garbage collection technique is used to recognize garbage blocks and automatically free them. Garbage collection is also known as automatic memory management, as the dynamically allocated memory is automatically reclaimed by the garbage collector, and there is no need for the programmer to de-allocate it explicitly. The main work of a garbage collector is to differentiate between garbage and non-garbage blocks and return the garbage blocks to the free list [10].

Advantages of Garbage Collection:

(i) Faster memory allocation

Simple pointer bumping

(ii) Increased cache performance

No need for headers

(ii) Better page locality

Compacts data and reduces fragmentation

Disadvantages of Garbage Collection:

(i) Additional process to run

(ii) Degrades cache performance

(iii) Degrades page locality

(iv) Increase in memory needs

The two common approaches of garbage collection are:

(i)   Reference counting: Each allocated block contains a reference count which indicates the number of pointers pointing to this block. This count is incremented each time we created or a copy a pointer to the block and is decremented each time when a pointer to the block is destroyed. When the reference count of an object becomes zero, it become unreachable and is considered as garbage. This garbage block is immediately made reusable by placing it on the free list.
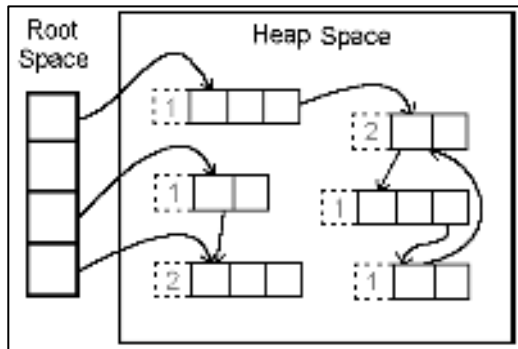
Advantages: **(i)** In reference counting a block of memory is freed as soon as it becomes garbage.

(ii) Very easy to implement.

Disadvantages: (i) Reference counting cannot handle cyclic reference correctly. A cyclic reference occurs when an object references itself indirectly, for example when some block A references block B and block B references block A. The reference count of blocks A and B will never become zero. So reference counting mechanism fails to recognize cyclic data structures as garbage and is not able to free them.

(ii) Reference counts have to be frequently updated thereby increasing processing cost.

(ii) Can be slow if a large collection is initiated.

(ii) **Mark and Sweep:** The mark and sweep garbage collector is run when the system is very low on memory and it is not possible to allocate any space for user. All the application programs come to a halt temporarily when this garbage collector runs and resume when all the garbage blocks are reclaimed. This garbage collection takes place in two phases:

(a) The first phase is the mark phase in which all the non-garbage blocks are marked

(b) Second is the sweep phase, in which the collector sweeps over the memory and returns all the unmarked i.e. garbage blocks to the free list.

A root is a program variable which directly points to a block on the heap and the set of all the roots is called the root set. These roots may be local variables on stack frames, register variables, global variables or static variables. A block is live or reachable if it is directly or indirectly accessible by the root set. The directly accessible blocks are those which are pointed to by any root, and the indirectly accessible blocks are those which are pointed to by any pointer from within a live block. Hence all the reachable blocks can be found out by following pointers from the root set

So the first task that is to be done is to find out the root set. For this all the program variables are scanned and pointers to dynamic memory (heap) are identified as roots. All the blocks that are directly and indirectly referenced by these roots are visited and marked. This is like DFS traversal of a graph and can be implemented recursively. The traversal starts from the set of roots and all the reachable blocks are visited. Whenever a block is visited, its marked field is set to true. So after the first phase all live blocks are marked and garbage blocks are not.

In the sweep phase, the garbage collector sequentially scans all the blocks on the heap and reclaims all the unmarked ones by placing them on the free list. The marked blocks are unmarked for the next run of the garbage collector. There is no movement of blocks.

In each memory block a boolean field is taken to differentiate between the marked and unmarked nodes. This mark field will be true if the block is marked and false if it is unmarked.

Advantages: (i) A mark and sweep garbage collector can recognize blocks that have already been marked and so there is no problem in the case of cyclic references.

(ii) There is no overhead of maintaining reference variables as in the reference count method.

Disadvantages: (i) This method uses a stop-the-world approach i.e. all programs need to stop when garbage collection takes place. This may be undesirable in interactive and real time applications.

(iii) Thrashing occurs when most of the memory is being used. In this case the collector is able to reclaim very less memory which is exhausted in a short duration. This causes the garbage collector to be called again, and this time also it reclaims only little space. So the garbage collector is called again and again this case.

CHECK YOUR PROGRESS

f) Dynamic programming is typically applied to _____ problem

g) In the knapsack problem, the given items have two attributes at minimum – an item's _____and _____

h) _____ is also known as automatic memory management

i) List two advantages of Garbage Collection.

2.9   SUMMING UP

- Union operation on set combine two set by making one of the root as the child of the other root

- Find operation on set returns the set-name of the set where the node belongs

- For binary search divide and conquer strategy is applied recursively for a given sorted array

- Merge sort is a recursive algorithm that splits the array into two subarrays , sorts each subarray , and then merges the two sorted arrays into a single sorted array. The array is divided until its size becomes 0 or 1.

- Merge sort is an external sorting algorithm

- In merge sort in divide step sub-problems are divided into two halves

- In conquer step sub-problems are sorted individually

- In combine Step sub-problems are combine to find the resultant sorted array

- Quick sort is an internal sorting algorithm. In its basic form it was developed by C.A.R Hoare in 1960.

- In merge sort , the list to be sorted is divided at its midpoint into subarrays which are independently sorted and later merged. In quick sort, the division to the sorted subarrays is made, so that the sorted subarrays do not need to merged later.

- The quick sort algorithm stop when there is only one element in the subarray to be sorted or if there is no element in the subarray to be sorted

- Knapsack problem: There are n items, ith item is worth vi dollars and weight wi pounds, where vi and wi are integers. Select item to put in knapsack with total weight is less than W, So that the total value is maximized

- There are two types of knapsack problem

   ⇒ 0-1 knapsack problem

   ⇒ fractional knapsack problem

- In 0-1 knapsack problem each item either be taken or left behind

- In fractional knapsack   problem fractions of items are allowed to choose

- The fractional knapsack problem is solvable by greedy strategy, but 0-1 knapsack problem are not solvable by greedy algorithm

- In the job sequencing with deadline problem, a feasible solution is a subset of job J such that each job is completed by its deadline and optimal solution is a feasible solution with a maximum profit

- the optimal way to pair wise merge n sorted files
- the optimal way to pair wise merge n sorted files
- Two algorithm to solve minimum spanning tree problem are- Kruskal algorithm and Prim algorithm.

2.10 ANSWERS TO CHECK YOUR PROGRESS

a) Divide and Conquer

b) The two limitations of Binary Search:

   a. Interacts poorly with the memory hierarchy

   b. Requires sorted list as an input

c) Input elements

d) Left-biased

e) Pivot element

f) Optimization

g) Value, weight or volume

h) Garbage collection

i) Two advantages of Garbage Collection:

   a. Faster memory allocation

   b. Simple pointer bumping

2.11 POSSIBLE QUESTIONS

Short Answer type Questions:

1) What is external and internal sorting? Give examples.

2) How does the binary search algorithm follow the divide and conquer method? Explain with an example.

3) Explain, what optimal binary search tree is.

4) Write a recursive and non-recursive function for binary search algorithm.

5) How does merge sort follow the divide and conquer strategy? Give one example.

6) What are the differences between quick sort and merge sort algorithm?

7) Write a recursive function to sort elements using merge sort.

8) Write quick sort algorithm and explain with an example.

9) What is greedy strategy for knapsack problem?

Long Answer type Questions:

1) Write briefly about knapsack problem. Explain with an example that greedy algorithm does not work for 0-1 knapsack problem.

2) What is optimal substructure for 0-1 knapsack and fractional knapsack problem?

3) With an example explain how 0/1 knapsack problem can be solved by using dynamic programming.

4) Sort the following element by using Insertion sort algorithm

        18, 19, 13, 16, 11, 9, 14, 12, 6 15, 22, 27, 3

5) Sort the following element by using Merge sort algorithm

        14, 20, 19, 13, 12, 6, 15, 22, 27, 3, 16, 11, 1

6) Sort the following element by using Quick sort algorithm

        3, 16, 11, 1, 4, 20, 16, 13, 12, 6, 15, 23, 27

7) Suppose A is an array of 7 elements. Search an element 9 in the array using binary search.

0	1	2	3	4	5	6
2	7	8	9	13	17	24

9) Consider the following job sequencing problem. Find the feasible solution set.

Job	1	2	3	4
Profit	10	20	15	5
Deadline	2	3	3	2

10) What is optimal substructure for 0-1 knapsack and fractional knapsack problem?

11) With help of an example show how 0/1 knapsack problem can be solved by using branch and bound technique.

2.12 REFERENCES AND SUGGESTED READINGS

[1] T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Third Edition, Prentice Hall of India Pvt. Ltd, 2006

[2] Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007

[3] Algorithms: Design Techniques and Analysis (Revised Edition), M H Alsuwaiyel

[4] Lecture Notes for Data Structures and Algorithms by John Bullinaria, School of Computer Science University of Birmingham Birmingham, UK

[5] Divide and conquer by Lakshmi Priya P, CSE, ACSCE

[6] LECTURE NOTES ON DESIGN AND ANALYSIS OF ALGORITHMS", VEER SURENDRA SAI UNIVERSITY OF TECHNOLOGY, BURLA, SAMBALPUR, ODISHA, INDIA – 768018

[7] Determining an Optimal Parenthesization of a Matrix Chain Product using Dynamic Programming, Vivian Brian Lobo et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 7 (2) , 2016, 786-792

[8] https://www.techopedia.com/definition/20272/knapsack-problem

[9] DYNAMIC MEMORY ALLOCATION AND GARBAGE COLLECTION Hans-J. Boehm

[10] Data structure through C in Deapth, by S.K. Srivastava and Deepali Srivastava

[11] Divide and Conquer Algorithms by Elias Koutsoupias.

UNIT 3: GENETIC ALGORITHM AND NEURAL NETWORK

Unit Structure:

3.1   Introduction
3.2   Genetic Algorithm
    3.2.1   Terms related to Genetic Algorithm
    3.2.2   Genetic Algorithm Requirements
    3.2.3   Genetic algorithms operations
    3.2.4   The Principle Structure of a Genetic Algorithm
    3.2.5   General Algorithm for Genetic Algorithm
    3.2.6   Some areas where we can use Genetic Algorithm
    3.2.7   Advantages of Genetic Algorithm
    3.2.8   Limitations of Genetic Algorithm
    3.2.9   Genetic Algorithms Application Areas
    3.2.10 Some Examples of Genetic Algorithm
3.3   Neural Network
    3.3.1   Classification based on connection type or topology
    3.3.2   Processing of information in neural network unit
    3.3.3   Learning Paradigms
    3.3.4   Artificial Neural Network
    3.3.5   Why we use Artificial Neural Network
    3.3.6   How Artificial Neuron work
    3.3.7   The main elements or blocks of an artificial neural network
    3.3.8   Properties of Artificial Neural Network
    3.3.9   Artificial Neural Network Models
    3.3.10 Applying Genetic Algorithm to neural networks
    3.3.11 Advantage and disadvantage
    3.3.12 Application of Neural Network
3.4   Summing Up
3.5   Answers to Check Your Progress
3.6   Possible Questions
3.7   References and Suggested Readings

3.1 INTRODUCTION

Before going to start Neural Network (NN) and Genetic Algorithm (GA), let's start with the concept of Optimization. Optimization is the process of having a set of inputs to finding the values of inputs in such a way that we can get the best output values i.e. making something better. The term "best" varies from time to time depending on the problem, but mathematical term, it is defined as maximizing or minimizing one or more objective functions, by varying the input parameters.

Neural networks and genetic algorithms are the techniques for optimization and learning, each with its own strengths and weaknesses. The two have generally evolved along separate paths. However, recently there have been attempts to combine the two technologies. Davis (1988) showed how any neural network can be rewritten as a type of genetic algorithm called a classifier system and vice versa.

Both Neural Network and Genetic Algorithm were invented in the spirit of a biological metaphor. The biological metaphor for neural networks is the human brain. Like the brain, this computing model consists of many small units that are interconnected. These units (or nodes) have very simple abilities. Hence, the power of the model derives from the interplay of these units. It depends on the structure of their connections.

3.2   GENETIC ALGORITHM

A genetic algorithm (or GA) is a search technique used in computing to find true or approximate solutions to optimization and search problems. Genetic algorithms are categorized as global search heuristics. GAs are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination).

Dr. David Goldberg, 1989 offered the following definition:

"Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics"[4].

This method combines Darwinian style survival of the fittest among binary string "artificial creatures" with a structured, yet randomized information exchange.

Genetic algorithms are implemented as a computer simulation in which a population of abstract representations (called **chromosomes** or the **genotype** or the **genome**) of candidate solutions (called **individuals, creatures,** or **phenotypes**) to an optimization problem evolves toward better solutions. Traditionally, solutions are represented in binary bit strings of **0s** and **1s**, but other encodings can also be taken. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the **fitness** of every individual in the population is evaluated, multiple individuals are selected from the current population (based on their fitness), and modified (recombined and possibly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm **terminates** when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached [7].

3.2.1 Terms Related to Genetic Algorithm

- **Individual** - Any possible solution

- **Population** - Group of all *individuals*

- **Search Space** - All possible solutions to the problem

- **Chromosome** - Blueprint for an *individual*

- **Trait** - Possible aspect (*features)* of an *individual*

- **Allele** - Possible settings of trait (black, blond, etc.)

- **Locus** - The position of a *gene* on the *chromosome*

- **Genome** - Collection of all *chromosomes* for an *individual*

- *Genotype -* Particular set of genes in a genome

- *Phenotype -* Physical characteristic of the genotype (smart, beautiful, healthy, etc.)
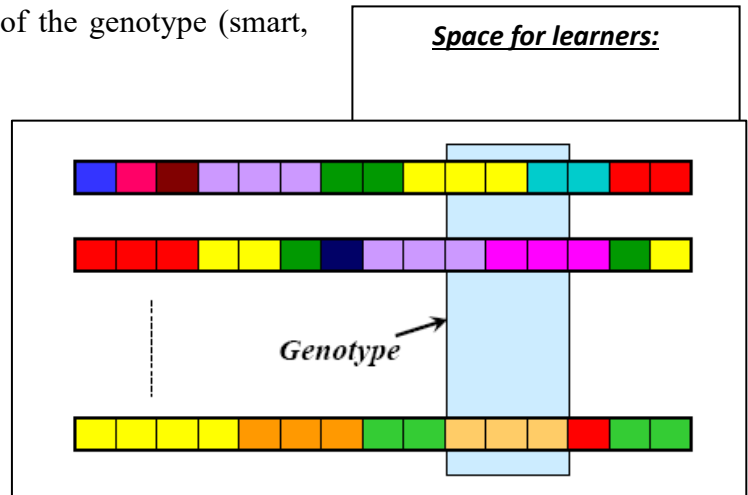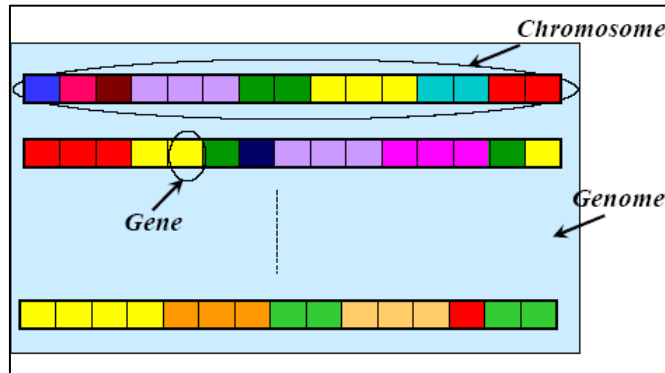
3.2.2 Genetic Algorithm Requirements

❖ A typical genetic algorithm requires two things to be defined:
  ⇒ A genetic representation of the solution domain, and
  ⇒ A fitness function to evaluate the solution domain

❖ A standard solution is represented by an array of bits. Arrays of other types and structures can also be used in essentially the same way

❖ The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operation

❖ Variable length representations may also be used, but crossover implementation is more complex in this case

❖ Tree-like representations are explored in Genetic programming

❖ Chromosomes could be:
  ⇒ Bit strings   (0101   ... 1100)
  ⇒ Real numbers   (43.2 -33.1 ... 0.0 89.2)
  ⇒ Permutations of element   (E11 E3 E7 ... E1 E15)
  ⇒ Lists of rules   (R1 R2 R3 ... R22 R23)

⇒ Program elements (genetic programming)

❖ The **fitness function** is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. For instance, in the **knapsack problem** we want to maximize the total value of objects that we can put in a knapsack of some fixed capacity.

❖ A representation of a solution might be an array of bits, where each bit represents a different object, and the value of the bit (0 or 1) represents whether or not the object is in the knapsack.

❖ Not every such representation is valid, as the size of objects may exceed the capacity of the knapsack.

❖ The fitness of the solution is the sum of values of all objects in the knapsack if the representation is valid or 0 otherwise. In some problems, it is hard or even impossible to define the fitness expression; in these cases, interactive genetic algorithms are used.
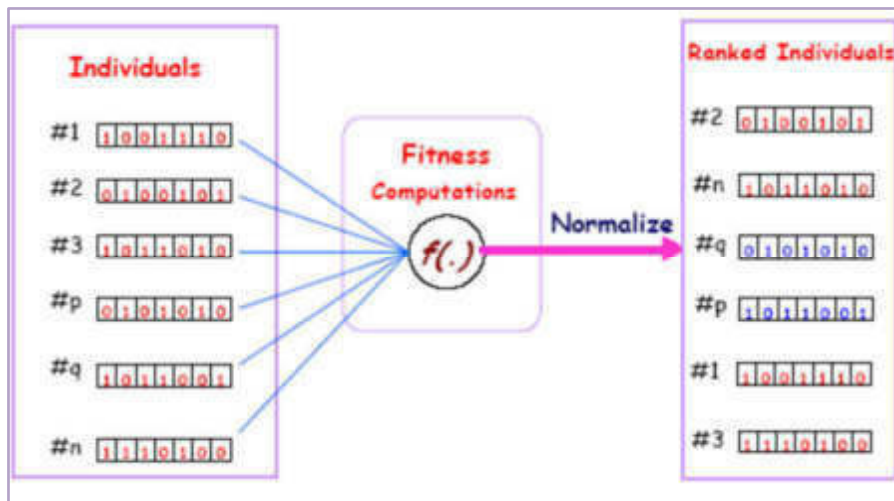
A fitness function

The GA's are used for maximization problem. For the maximization problem the fitness function is same as the objective function. But, for minimization problem, one way of defining a 'fitness function' is: F (x) = 1/f(x), where f(x) is an objective function.

3.2.3 Genetic Algorithms Operations

Genetic algorithms have three main operations:

a) Reproduction (or Selection)
b) Crossover
c) Mutation

a) **Reproduction** is a process in which individual strings are copied according to their fitness. Whose fitness value is more that is having more chances to survive in next generation.

b) **Crossover** is a process that can be divided in two steps. First, pairs of bit strings will be mated randomly to become the parents of two new bit strings. The second part consists of choosing a place (crossover site) in the bit string and exchanges all characters of the parents after that point. The process tries to artificially reproduce the mating process where the DNA of two parents determines the DNA for the newly born.
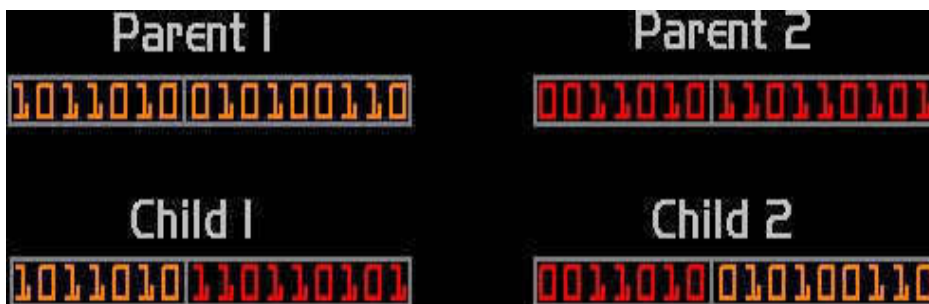
In the above figure, crossover site is 7 so after 7th bit the values of Parent1 and Parent2 get interchanged and results as Child1 and Child2.

c) **Mutation** is included, not because the previous process of reproduction and recombination are not sufficient, but because of the probability that a certain bit can't be changed by the previous operations

due to its absence from the generation, either by a random chance or because it has been discarded. It only implies the change of a 0 for a 1 and vice versa.

In the above figure, mutation takes place at bit 7 (here 7th bit's value changes from 1 to 0).

3.2.4 The Principle Structure of a Genetic Algorithm
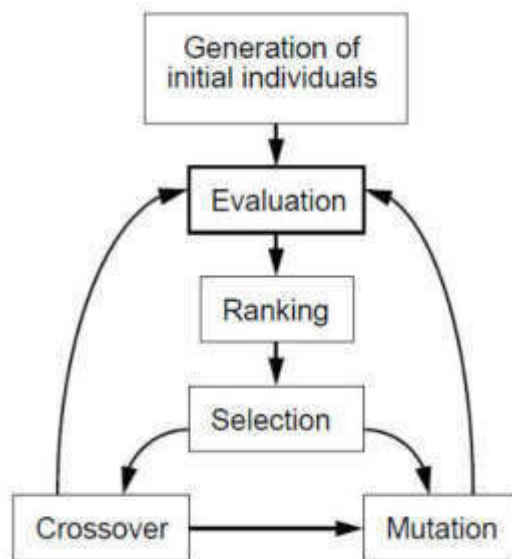
3.2.5 General Algorithm for Genetic Algorithm:

- **Initialization:** Initially many individual solutions are randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, covering the entire range of possible solutions (the search space). Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found.

- **Selection:** During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as this process may be very time-consuming. Most functions are stochastic and designed so that a small proportion of less fit solutions are selected. This helps keep the diversity of the population large, preventing premature convergence on poor solutions. Popular and well-studied selection methods include roulette wheel selection and tournament selection.

  In roulette wheel selection, individuals are given a probability of being selected that is directly proportionate to their fitness. Two individuals are then chosen randomly based on these probabilities and produce offspring.

Roulette Wheel's Selection Pseudo Code:

```
for all members of population
        sum += fitness of this individual
end for
for all members of population
        probability = sum of probabilities + (fitness / sum)
        sum of probabilities += probability
end for
loop until new population is full
        do this twice
                number = Random between 0 and 1
                for all members of population
                        if number > probability but less than next probability then
                                you have been selected
                end for
        end
        create offspring
end loop
```

- **Reproduction:** The next step is to generate a second generation population of solutions from those selected through genetic operators: crossover (also called recombination), and/or mutation. For each new solution to be produced, a pair of "parent" solutions is selected for breeding from the pool selected previously. By producing a "child" solution using the above methods of crossover and mutation, a new solution is created which typically shares many of the characteristics of its "parents". New parents are selected for each child, and the process continues until a new population of solutions of appropriate size is generated. These processes ultimately result in the next generation population of chromosomes that is different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding, along with a small

proportion of less fit solutions, for reasons already mentioned above.

- **Crossover:** The most common type is single point crossover. In single point crossover, you choose a locus at which you swap the remaining alleles from on parent to the other. This is complex and is best understood visually. As you can see, the children take one section of the chromosome from each parent. The point at which the chromosome is broken depends on the randomly selected crossover point. This particular method is called single point crossover because only one crossover point exists. Sometimes only child 1 or child 2 is created, but oftentimes both offspring are created and put into the new population. Crossover does not always occur, however. Sometimes, based on a set probability, no crossover occurs and the parents are copied directly to the new population. The probability of crossover occurring is usually 60% to 70%.

- **Mutation:** After selection and crossover, you now have a new population full of individuals. Some are directly copied, and others are produced by crossover. In order to ensure that the individuals are not all exactly the same, you allow for a small chance of mutation. You loop through all the alleles of all the individuals, and if that allele is selected for mutation, you can either change it by a small amount or replace it with a new value. The probability of mutation is usually between 1 and 2 tenths of a percent. Mutation is fairly simple. You just change the selected alleles based on what you feel is necessary and move on.

- **Termination:** This generational process is repeated until a termination condition has been reached. Common terminating conditions are:

$\Rightarrow$ A solution is found that satisfies minimum criteria

$\Rightarrow$ Fixed number of generations reached

$\Rightarrow$ Allocated budget (computation time/money) reached

$\Rightarrow$ The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results

$\Rightarrow$ Manual inspection

$\Rightarrow$ Any Combinations of the above

Genetic Algorithm Pseudo-code

Choose initial population

      Evaluate the fitness of each individual in the population

      Repeat

      Select best-ranking individuals to reproduce

      Breed new generation through crossover and mutation (genetic operations) and give birth to offspring

      Evaluate the individual fitness of the offspring

      Replace worst ranked part of population with offspring

      Until <terminating condition>

3.2.6 Some Areas of Usage of Genetic Algorithm

- Genetic Algorithms can be applied to virtually any problem that has a large search space.

- Al Biles uses genetic algorithms to filter out 'good' and 'bad' riffs for jazz improvisation.

- The military uses GAs to evolve equations to differentiate between different radar returns

- Stock companies use GA-powered programs to predict the stock market

3.2.7 Advantages of Genetic Algorithm

Genetic Algorithm has various advantages, Some of these include −

- Does not require any derivative information (which may not be available for many real-world problems)

- Faster and more efficient as compared to the traditional methods

- Very good parallel capabilities

- Optimizes both continuous and discrete functions and also multi-objective problems

- Provides a list of "good" solutions and not just a single solution

- Always gets an better answer over the time to the problem

- Useful when the search space is very large and there are a large number of parameters involved

3.2.8 Limitations of Genetic Algorithm

Genetic Algorithm also suffers from a few limitations. These include −

- Not suited for all problems, especially problems which are simple and for which derivative information is available

- Fitness value is calculated repeatedly which might be computationally expensive for some problems

- Sometimes, there are no guarantees on the optimality or the quality of the solution

- If not implemented properly, the Genetic Algorithm may not converge to the optimal solution

3.2.9 Genetic Algorithms - Application Areas

Genetic Algorithms are primarily used in optimization problems of various kinds, but they are frequently used in other application areas as

well. Here**,** some of the areas in which Genetic Algorithms are frequently used are listed below:

- **Optimization** − Genetic Algorithms are most commonly used in optimization problems wherein we have to maximize or minimize a given objective function value under a given set of constraints. The approach to solve Optimization problems has been highlighted throughout the tutorial.

- **Economics** − GAs are also used to characterize various economic models like the cobweb model, game theory equilibrium resolution, asset pricing, etc.

- **Neural Networks** − GAs are also used to train neural networks, particularly recurrent neural networks.

- **Parallelization** − GAs also have very good parallel capabilities, and prove to be very effective means in solving certain problems, and also provide a good area for research.

- **Image Processing** − GAs are used for various digital image processing (DIP) tasks as well like dense pixel matching.

- **Vehicle routing problems** − with multiple soft time windows, multiple depots and a heterogeneous fleet.

- **Scheduling applications** − GAs are used to solve various scheduling problems as well, particularly the time tabling problem.

- **Machine Learning** − as already discussed, genetics based machine learning (GBML) is a niche area in machine learning.

- **Robot Trajectory Generation** − GAs have been used to plan the path which a robot arm takes by moving from one point to another.

- **Parametric Design of Aircraft** − GAs have been used to design aircrafts by varying the parameters and evolving better solutions.

- **DNA Analysis** − GAs have been used to determine the structure of DNA using spectrometric data about the sample.

- **Multimodal Optimization** − GAs are obviously very good approaches for multimodal optimization in which we have to find multiple optimum solutions.

- **Traveling salesman problem and its applications** − GAs have been used to solve the TSP, which is a well-known combinatorial problem using novel crossover and packing strategies

3.2.10   Some Examples of Genetic Algorithm

Example 1: Encode the solution using GA:

f(x) = {MAX(x^2): 0 <= x <= 32 }

For encoding solution, initially we use 5 bits (1 or 0)

- **Step 1:** Generate initial population

A	0	1	1	0	1
B	1	1	0	0	0
C	0	1	0	0	0
D	1	0	0	1	1

- **Step 2:** Evaluate each solution against objective

Sol.	String	Fitness	% of Total
A	01101	169	14.4
B	11000	576	49.2
C	01000	64	5.5
D	10011	361	30.9

- **Step 3:** Create next generation of solutions

$\Rightarrow$ Probability of "being a parent" depends on the fitness

Ways for parents to create next generation

$\Rightarrow$ Reproduction

- Use a string again unmodified

$\Rightarrow$ Crossover

- Cut and paste portions of one string to another

$\Rightarrow$ Mutation

- Randomly flip a bit

$\Rightarrow$ COMBINATION of all of the above

Example 2: $P = (C1, C2, C3, ..., Cn)$ means the salesmen move from city C1 to C2, C2 to C3, C3 to Cn. There are five cities that a salesperson will pass. The cities are A, B, C, D, and E. The journey starts from A and ends at A as well. The distance between cities is shown in Figure below [8]:
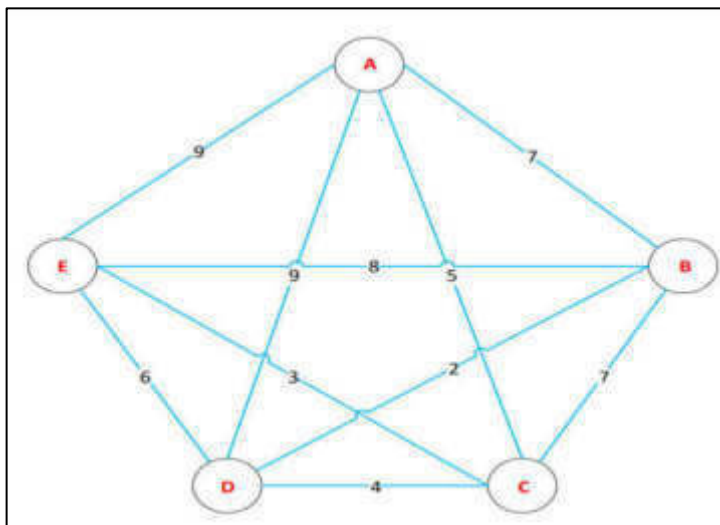
Sl. No.	City 1	City 2	Distance
1	A	B	7
2	A	C	5
3	A	D	9
4	A	E	9
5	B	C	7
6	B	D	2
7	B	E	8
8	C	D	4
9	C	E	3
10	D	E	6

Initial Chromosomes:

	Chromosomes			
1	B	D	E	C
2	D	B	E	C
3	C	B	D	E
4	E	B	C	D
5	E	C	B	D
6	C	D	E	B

Initial fitness:

	Route					Fitness
1	AB	BD	DE	EC	CA	
	7	2	6	3	5	23
2	AD	DB	BE	EC	CA	
	9	2	8	3	5	27
3	AC	CB	BD	DE	EA	
	5	7	2	6	9	29
4	AE	EB	BC	CD	DA	
	9	8	7	4	9	37
5	AE	EC	CB	BD	DA	
	9	3	7	2	9	30
6	AC	CD	DE	EB	BA	
	5	4	6	8	7	30

Selection: The chromosome selection is made because of the TSP problem desirably that chromosomes with smaller fitness will have a higher probability of being reelected.

	Chromosome		
1	1	23	0.043478
2	1	27	0.037037
3	1	29	0.034483
4	1	37	0.027027
5	1	30	0.033333
6	1	30	0.033333
Total			0.208692

	Probabilty		
1	0.043478	0.208692	0.208337
2	0.037037	0.208692	0.177472
3	0.034483	0.208692	0.165233
4	0.027027	0.208692	0.129507
5	0.033333	0.208692	0.159725
6	0.033333	0.208692	0.159725

	Cumulative		
1	0.208337	0	0.208337
2	0.208337	0.177472	0.38581
3	0.38581	0.165233	0.551043
4	0.551043	0.129507	0.68055
5	0.68055	0.159725	0.840275
6	0.840275	0.159725	1

	Random
1	0.312
2	0.112
3	0.340
4	0.744
5	0.523
6	0.421

Crossover: Crossover is done to produce children from two mothers who are mated. The resulting chromosomes are expected to increase the value of fitness. The number of chromosomes that experience crossover is determined by crossover probability. The crossover probability value is 0.25.

	Random
1	0.452
2	0.209
3	0.221
4	0.875
5	0.770
6	0.133

Old	New	Chromosomes			
1	2	D	B	E	C
2	1	B	D	E	C
3	3	C	B	D	E
4	6	C	D	E	B
5	5	E	C	B	D
6	4	E	B	C	D

Mutation: Mutation works to exchange genes for genes on other chromosomes. Expected results increase the value of fitness to be achieved. If a gene is exchanged at the end of a chromosome, this gene will be exchanged for the first gene. There is a parameter to determine how many genes will be mutated. The mutation rate is 0.2.

	Chromosomes			
1	D	B	E	C
2	B	D	E	C
3	C	B	D	E
4	C	D	E	B
5	E	C	B	D
6	E	B	C	D

Fitness Values:

Route					Fitness
1	AD	DB	BC	CE	EA	
	9	2	7	3	9	30
2	AB	BD	DE	EC	CA	
	7	2	6	3	5	23
3	AC	CE	ED	DB	BA	
	5	3	6	2	7	23
4	AE	EC	CB	BD	DA	
	9	3	7	2	9	30
5	AD	DB	BC	CE	EA	
	9	2	7	3	9	30
6	AE	ED	DB	BC	CA	
	9	6	2	7	5	29

In the first generation, it has been seen that there is the smallest fitness value that does not change. If the calculation is continued up to the Nth generation, then it is assumed that the lowest fitness value will remain unchanged. Although the calculation is sufficiently elaborated up to the 1st generation, a near-optimal solution has been found, from the genetic algorithm process above, the final result the route with the shortest optimal distance is **A, B, D, E, C, A.**

CHECK YOUR PROGRESS

a) The _____ is defined over the genetic representation and measures the quality of the represented solution
b) Define crossover
c) List two limitations of Genetic Algorithm
d) Mention two application areas of Genetic Algorithm

3.3   NEURAL NETWORK

Neural Networks are algorithms optimization and learning based loosely on the concept inspired by research into the nature of the brain. A neural network is a computational model consisting of a number of connected elements, known as neurons. A **neuron** is a processing unit that receives input from outside the network and/or from other neurons, applies a local transformation to that input, and provides a single output signal which is passed on to other neurons and/or outside the network. Each of the inputs is modified by a value associated with the connection. This value is referred to as the connection strength, or weight, and roughly speaking, represents how much importance the neuron attaches to that input source. The local transformation is referred to as the activation function and is usually sigmoidal in nature.

An important difference between Neural Network and Genetic Algorithm is that, in a genetic algorithm only those items of data that have value in predicting the outputs are retained as inputs to the system. A neural network, on the other hand, does not exclude irrelevant data inputs from the final system. It nullifies the effects of such data inputs by assigning a low weight to them in the decision process.
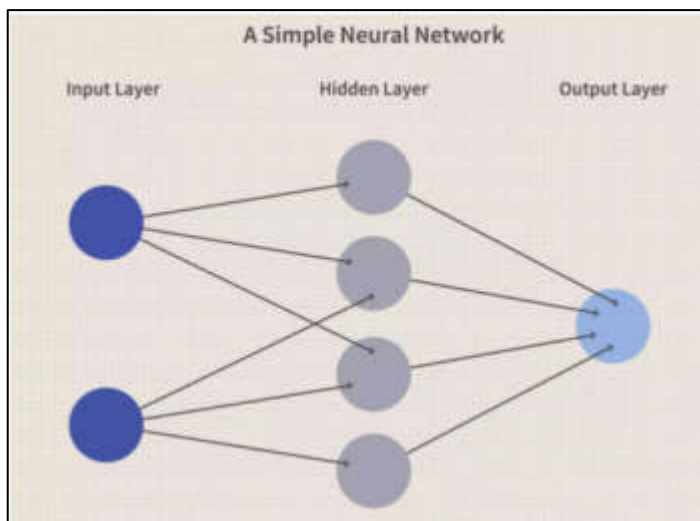
Neural Network mainly consists of five components:

1. **Directed graph**, also known as the network topology whose arcs are refer to as links

   2. **State variable** is associated with each node

   3. **Real- valued weight** associated with each link

   4. **Real- valued bias** associated with each link

5. **Transfer Function** determines the state of a node as a function of a) its bias b, b) the weights, w_t of its incoming links, and c) the states, x of the nodes connected to it by these links [1].

A simple neural network may be illustrated with the help of following figure:

Fig: A Neural Network
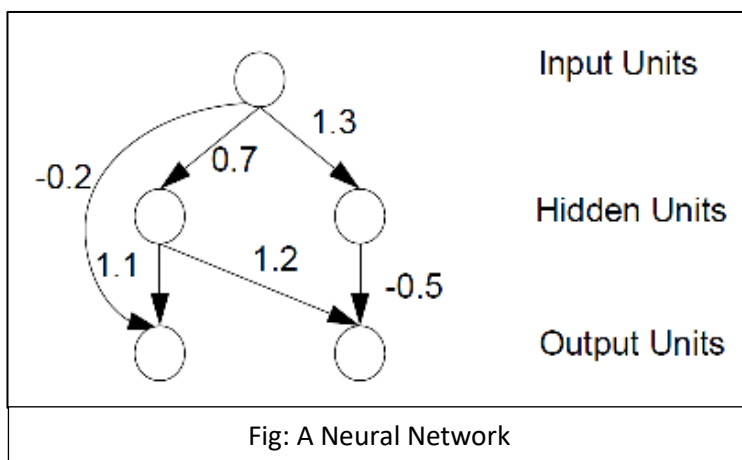
This network consists of:

- Five **units** or **neurons** or **nodes (the circles)**
- Six **connections (the arrows)**
- The number next to each connection is called **weight**; it indicates the strength of the connection
- Connections with a positive weight are called **excitatory**
- Connections with a negative weight are called **inhibitory**

The constellation of neurons and connection is called the architecture of the network, which is also called the **topology**. This is a **feed-forward**

network, because the connections are directed in only one way, from top to bottom. There are no loops or circles. [6]

In a strictly layered network, the nodes are arranged several layers. Connections may only exist to the nodes of the following layer. Yet in our case, there is a connection from the input layer to the output layer. It is, however, not a strictly layered network, but we may call it a **layered network**, because the nodes of each layer are not interconnected.

3.3.1 Classification Based on Connection Type or Topology

1. Single layer feed-forward networks:

    – Input layer projecting into the output layer
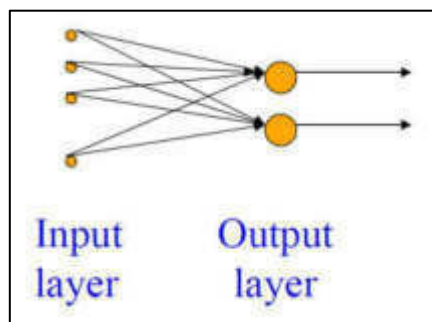
2. Multi-layer feed-forward networks:

    – One or more hidden layers.

    – Input projects only from previous layers onto a layer. Typically, only from one layer to the next
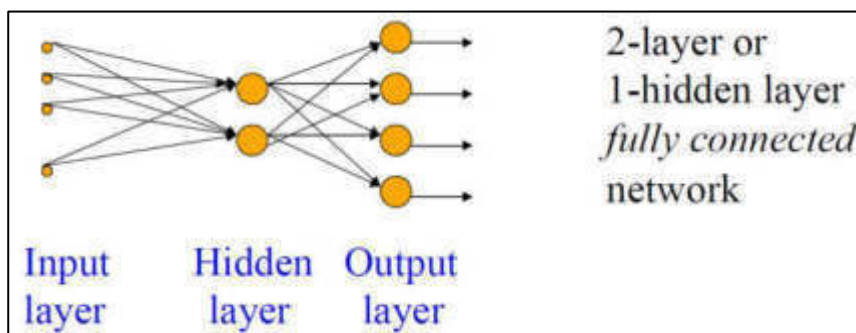
3. Recurrent networks:

– A network with feedback, where some of its inputs are connected to some of its outputs (discrete time) [2]
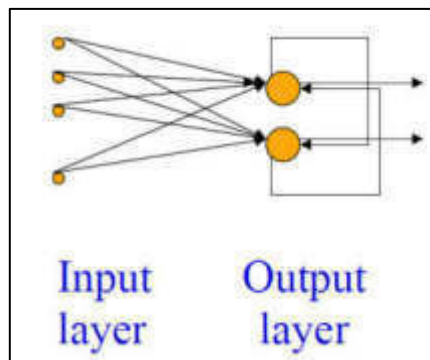
3.3.2 Processing of Information in Neural Network Unit

The node receives the weighted activation of other nodes through its incoming connections. Firstly, these are added up (summation). The result is passed through activation function; the outcome is the activation of the node. For each of the outgoing connections, this activation value is multiplied with the specific weight and transferred to the next node [5].
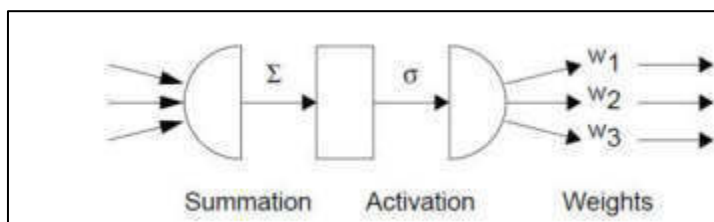
Information processing in Neural Network unit

A few different threshold functions are used. It is important that a threshold function is non-linear; otherwise a multilayer network is equivalent to a one layer net. The most widely applied threshold function is the logistic sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

There are a few other activation functions in use: scaled sigmoid, gaussian, sine, hyperbolic tangent, etc. It has some benefits for back-propagation learning, the classical training algorithm for feed-forward neural networks.

Back-Propagation Learning

At the beginning the weights of a network are randomly set or otherwise predefined. However, only little is known about the mathematical properties of neural networks. Especially, for a given problem, it is basically not possible to say which weights have to be assigned to the connections to solve the problem. Since NN follow the non-declarative programming paradigm, the network **is trained** by examples, so called **patterns**. Back-propagation is one method to train the network. The training is performed by one pattern at a time. The training of all patterns of a training set is called an **epoch.** The training set has to be a representative collection of input-output examples.

Size of the training set:
- No one-fits-all formula
- Over fitting can occur if a "good" training set is not chosen
- What constitutes a "good" training set?
    - Samples must represent the general population.
    - Samples must contain members of each class.
    - Samples in each class must contain a wide range of variations or noise effect.
- The size of the training set is related to the number of hidden neurons

Back-propagation strategy:
- N is a neuron.
- N_w is one of N's inputs weights
- N_{out} is N's output.
- $N_w = N_w + \Delta N_w$
- $\Delta N_w = N_{out} * (1 - N_{out}) * N_{ErrorFactor}$
- $N_{ErrorFactor} = N_{ExpectedOutput} - N_{ActualOutput}$

- This works only for the last layer, as we can know the actual output, and the expected output. [5]

3.3.3 Learning Paradigms

- **Supervised learning:** In supervised learning a network is fed with a set of training samples (inputs and corresponding output), and it uses these samples to learn the general relationship between the inputs and the outputs. This relationship is represented by the values of the weights of the trained network.

- **Unsupervised learning:** In unsupervised learning no desired output is associated with the training data. It is faster than supervised learning. This learning used to find out structures within data either by Clustering or by Compression

- **Reinforcement learning**: Like supervised learning, but weights adjusting is not directly related to the error value. The error value is used to randomly, shuffle weights. Relatively slow learning due to 'randomness'.

3.3.4 Artificial Neural Network

An Artificial Neural Network (ANN) is composed of many artificial neurons that are linked together according to specific network architecture.

Computational model of Artificial Neural Network is inspired by the human brain. ANN is massively parallel, distributed system and made up of simple processing units (neurons). Synaptic connection strengths among neurons are used to store the acquired knowledge. Knowledge is acquired by the network from its environment through a learning process.
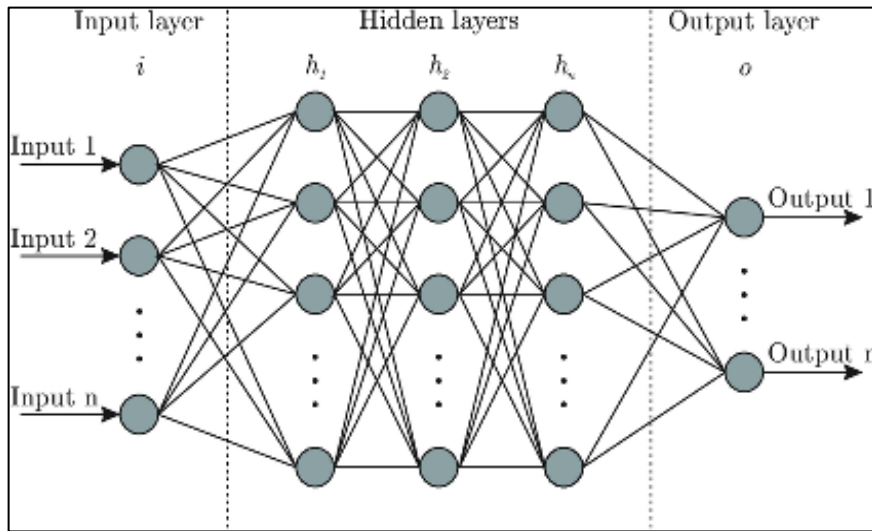
Artificial Neural Network

An ANN is either **a hardware implementation** or a **computer program** which strives to simulate the information processing capabilities of its biological exemplar. ANNs are typically composed of a great number of interconnected artificial neurons. The artificial neurons are simplified models of their biological counterparts. ANN is a technique for solving problems by constructing software that works like our brains

3.3.5 Usage Artificial Neural Network

There are basically two reasons why we are interested in building artificial neural networks (ANN):

- **Biological viewpoint**: ANNs can be used to simulate and replicate components of human or animal brain, so that it can gives us insight into natural information processing.
- **Technical viewpoint**: Character recognition or the predictions of future states of a system require massively parallel and adaptive processing. ANNs made it easy.

3.3.6 How Artificial Neuron Work

Let us start with the model of an artificial neuron
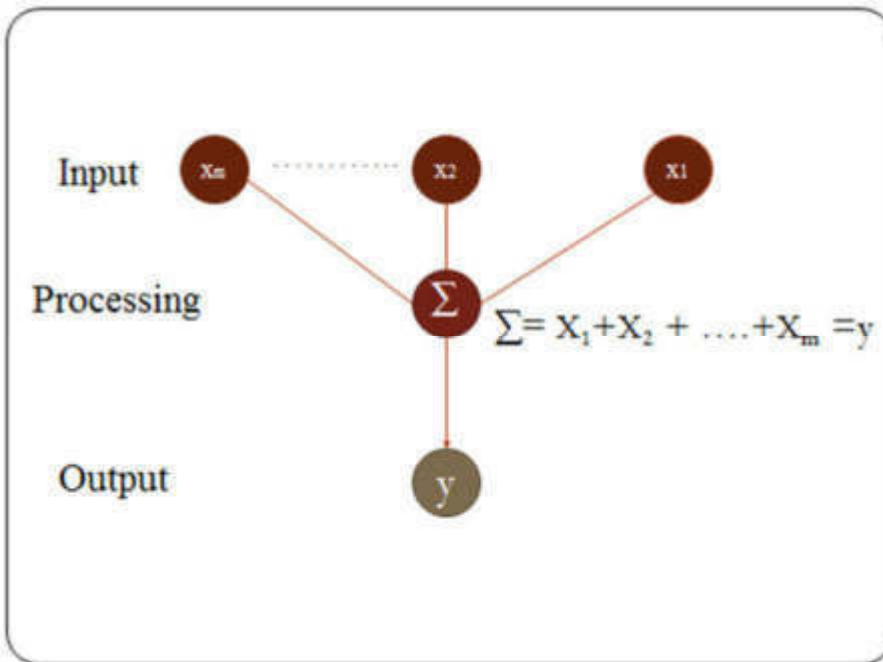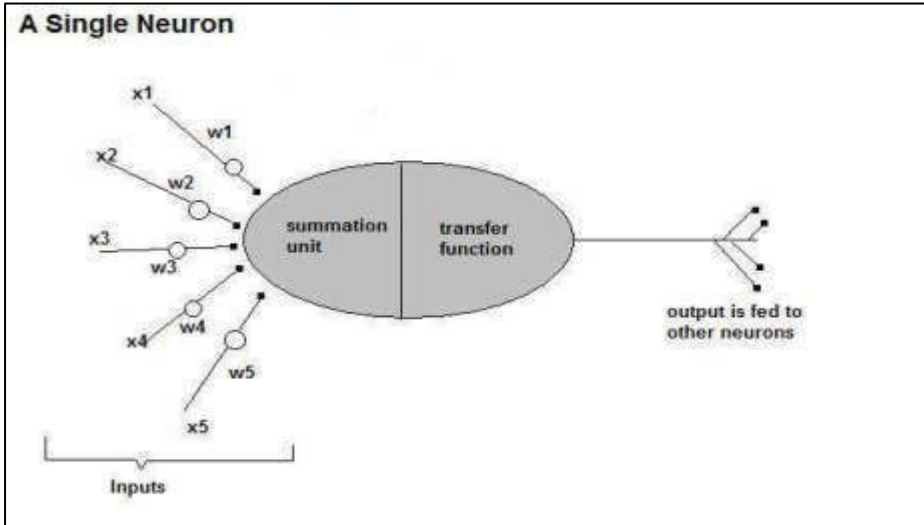
$$\Sigma = X_1 + X_2 + \ldots + X_m = y$$

Not all inputs are equal

Input

weights

Processing

$\Sigma = X_1w_1 + X_2w_2 + \ldots + X_mw_m = y$

Output

$$\sum_{i=1}^{m} X_i w_i = y$$

The signal is not passed down to the
next neuron as it is

Input

weights

Processing

Transfer Function
(Activation Function)

$$\sum_{i=1}^{m} X_i w_i = v_k$$

Output

$$y_k = f(v_k)$$
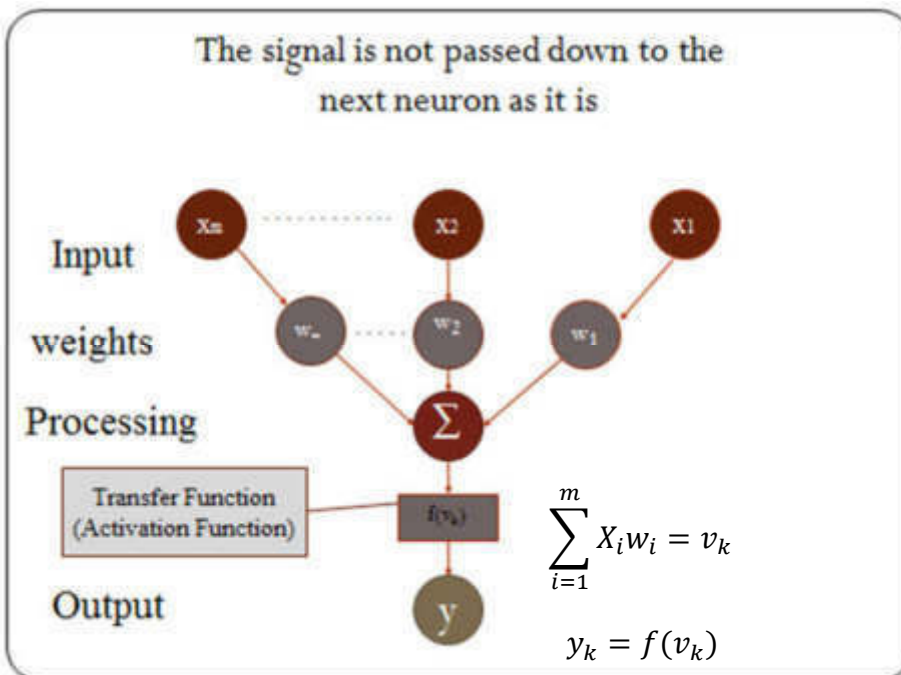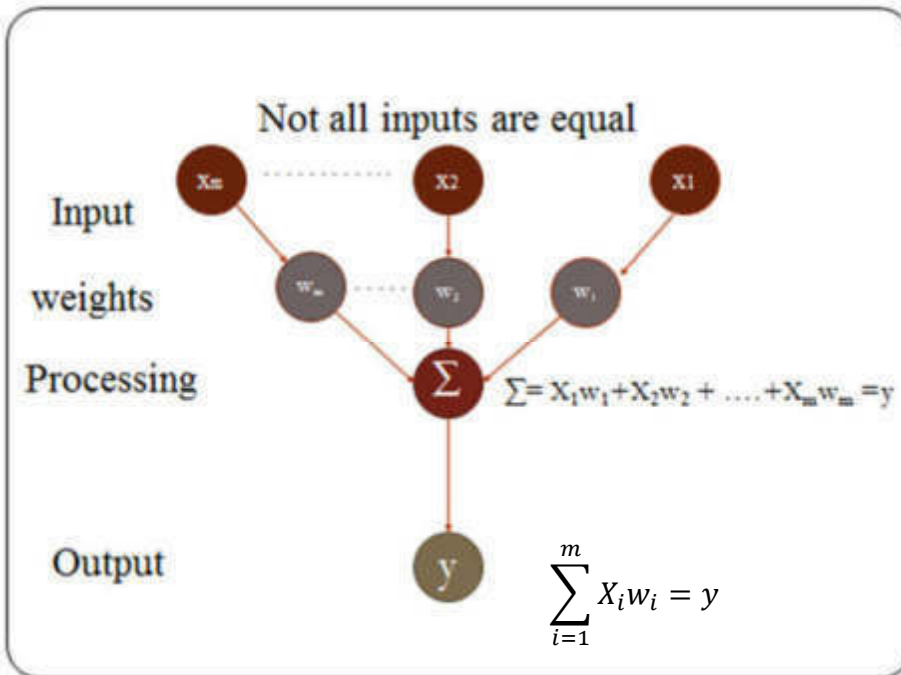
3.3.7 The Main Elements or Blocks of an Artificial Neural Network

a) The computing element (called an artificial neuron or simply neuron)

b) The connection pattern among the elements (structure or architecture)

c) The process used for training the neural network (learning algorithm)

3.3.8 Properties of Artificial Neural Network

- Learning from examples
    - labeled or unlabeled
- Adaptions
    - changing the connection strengths to learn things
- Non-linearity
    - the non-linear activation functions are essential
- Fault tolerance
    - if one of the neurons or connections is damaged, the whole network still works quite well

Thus, they might be better alternatives than classical solutions for problems characterized by high dimensionality, noisy, imprecise or imperfect data; and a lack of a clearly stated mathematical solution or algorithm.

CHECK YOUR PROGRESS

e) A neural network is a computational model consisting of a number of connected elements, known as _____

f) The training of all patterns of a training set is called an

_____

g) _____ used to find out structures within data either by Clustering or by Compression

h) Mention two properties of Artificial Neural Network

3.3.9 Artificial Neural Network Models

- Deep Learning Architectures
- Multilayer feed-forward networks (Multilayer perceptron)
- Radial Basis Function networks
- Self-Organizing Networks

3.3.10 Applying Genetic Algorithm to Neural Networks

Combining Neural Network with Evolutionary Algorithms leads to Evolutionary Artificial Neural Networks (EANNs). One can use Evolutionary Algorithms like the GA to train Neural Network, choose their structure or design related aspects like the function of their neurons.

3.3.10.1   Using Genetic Algorithm to Train Neural Network

GA will train the network no matter how it is connected - whether it's a feed-forward or a feedback network [4]. Furthermore, it can train general networks which are mixture of the two types.

a)   How to create a string or chromosome from simple neural network

Simple Neural Network
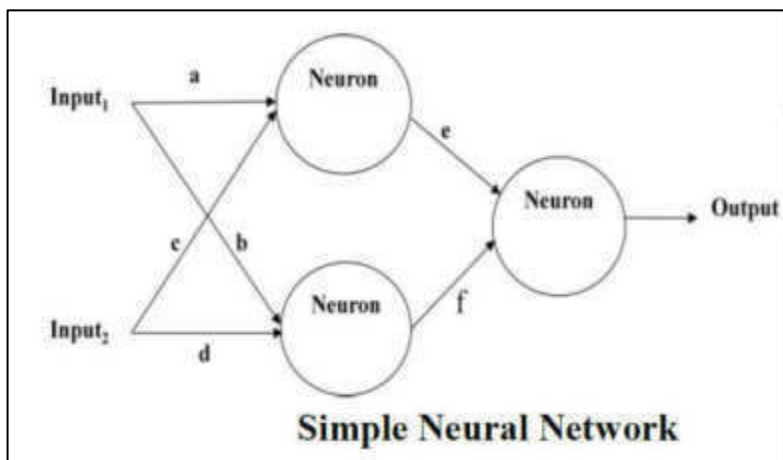
All the weights in the network are joined to make one string. This string is then used in the GA as a member of the population. Each string represents the weights of a complete network. Following figure represent the value of chromosome obtained from the above fig: simple neural network.

a	b	c	d	e	f

String or Chromosome
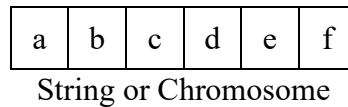
b) How to evaluate Fitness

Fitness is measured by calculating the error (target – output)

i.e. fitness= 1/error (the lower the error the higher the fitness)

Example:

The target for a network with a particular input is 1. The outputs are shown below, calculate their fitness.

Population member	Output
1	0.4
2	0.2
3	1.6
4	-0.9

One can complete the entities below by first calculating the error as described above. Then making all the errors positive and finally working out a fitness (low errors have a high fitness) by using fitness = 1 / error.

Population member	Output	Error (T-O)	Positive	Fitness
1	0.4	0.6	0.6ss	1.67
2	0.2	0.8	0.8	1.26
3	1.6	-0.6	0.6	1.67
4	-0.9	1.9	1.9	0.53

So members 1 and 3 (which are closest to the target) have the highest fitness.

3.3.10.2   Using GA to Select ANN Topology

By using genetic algorithm one can evaluate the how neurons are connected with one another in a network. [4]

Case 1: Simple Neural Network

Consider a simple neuron network. If there is a connection of one neuron with other neuron, it will be represented by 1 otherwise 0.
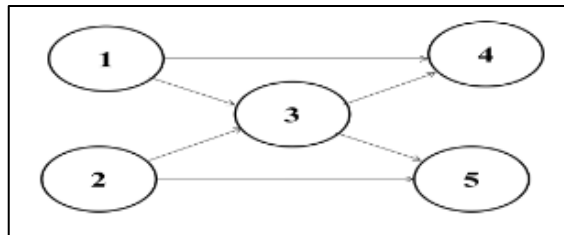
In this figure, consider the connections from neuron 1. These may be represented by the string:

<div align="center">

0   0   1   1   0

</div>

The first 0 represents the fact that neuron 1 is not connected to itself
The second 0 represent that neuron 1 is not connected to neuron 2
The third 1 means that neuron 1 is connected to neuron 3, and so on.
The complete network may be represented by the matrix shown in figure below:

00110	Neuron 1
00101	Neuron 2
00011	Neuron 3
00000	Neuron 4
00000	Neuron 5

<div align="center">Matrix representing the complete network</div>

Where matrix element M_{jk} is 0 if there is no connection between neuron j and k; if the matrix element is a 1, then there is a connection.

It is possible to concatenate the matrix into one string from this figure:

<div align="center">

0 0 1 1 0 0 0 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0

</div>

Each string represents the connection pattern of a whole network.

Case 2: Neural Network with weights

Neural networks with weights
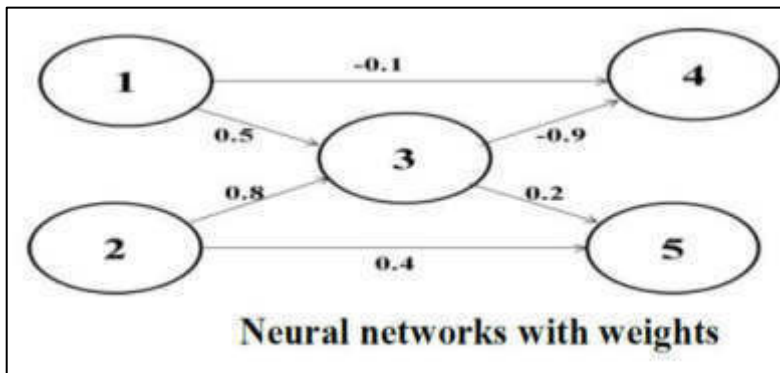
Matrix representing network as:

Connections	Weights
0 0 1 1 0	+0.0 +0.0 +0.5 -0.1 +0.0
0 0 1 0 1	+0.0 +0.0 +0.8 +0.0 +0.4
0 0 0 1 1	+0.0 +0.0 +0.0 -0.9 +0.2
0 0 0 0 0	+0.0 +0.0 +0.0 +0.0 +0.0
0 0 0 0 0	+0.0 +0.0 +0.0 +0.0 +0.0

This corresponds to the string:
0 0 0 0.5 0.1 0 0 0 0 0.8 0 0.4 0 0 0 0 -0.9 0.2 0 0 0 0 0 0 0 0 0 0 0
In this case, a weight of zero simply means that no connection exists between these neurons.

3.3.11 Advantages and Disadvantages

- **Advantages**
  - Adapt to unknown situations
  - Powerful, it can model complex functions and can perform tasks that a linear program cannot
  - Ease of use, learns by example, and very little user domain-specific expertise needed and does not need to be reprogrammed
  - It can be implemented in any application

- **Disadvantages**
  - The neural network needs training to operate
  - Not exact
  - Large complexity of the network structure as it requires high processing time for large neural networks

3.3.12 Application of Neural Network

- Pattern recognition
- Investment analysis
- Control system and monitoring
- Mobile computing
- Market and financial application
- Forecasting like sales, market research etc.

3.4 SUMMING UP

- Neural networks and genetic algorithms are the techniques for optimization and learning
- **Genetic algorithms** are search algorithms based on the mechanics of natural selection and natural genetics
- A typical genetic algorithm mainly requires two things to be defined, a genetic representation of the solution domain, and a fitness function to evaluate the solution domain
- The **fitness function** is defined over the genetic representation and measures the quality of the represented solution
- Genetic algorithms have three main operations: **Reproduction** (or Selection), **Crossover**, **Mutation**
- A **neural network** is a computational model consisting of a number of connected elements, known as neurons
- A **neuron** is a processing unit that receives input from outside the network and/or from other neurons, applies a local transformation to that input, and provides a single output signal which is passed on to other neurons and/or outside the network
- The constellation of neurons and connection is called the architecture of the network, which is also called the **topology**

- **An Artificial Neural Network (ANN)** is composed of many artificial neurons that are linked together according to specific network architecture

- Artificial Neural Networks are an **imitation of the biological neural networks**, but much simpler ones.

- The computing would have a lot to gain from neural networks. Their ability to learn by example makes them very flexible and powerful furthermore there is need to device an algorithm in order to perform a specific task.

- **Neural networks** also contribute to area of research such as **neurology** and **psychology.** They are regularly used to model parts of living organizations and to investigate the internal mechanisms of the brain.

- Many factors affect the performance of **ANNs,** such as the transfer functions, size of training sample, network topology, weights adjusting algorithm.

3.5 ANSWERS TO CHECK YOUR PROGRESS

a) Fitness function
b) Crossover is a process that can be divided in two steps. First, pairs of bit strings will be mated randomly to become the parents of two new bit strings. The second part consists of choosing a place (crossover site) in the bit string and exchanges all characters of the parents after that point. The process tries to artificially reproduce the mating process where the DNA of two parents determines the DNA for the newly born.
c) Two limitations of Genetic Algorithms:

    a. Fitness value is calculated repeatedly which might be computationally expensive for some problems

    b. Sometimes, there are no guarantees on the optimality or the quality of the solution

d) Two application areas of Genetic Algorithms:
    a. Image Processing − GAs are used for various digital image processing (DIP) tasks as well like dense pixel matching.
    b. Vehicle routing problems − with multiple soft time windows, multiple depots and a heterogeneous fleet.

e) Neurons.
f) Epoch
g) Unsupervised learning
h) Two properties of Artificial Neural Network
   a. Learning from examples whether labeled or unlabelled
   b. Fault tolerance: If one of the neurons or connections is damaged, the whole network still works quite well

3.6 POSSIBLE QUESTIONS

Short Answer type Questions:

Define the following terms:

1. Chromosome   2. Genotype   3. Genome
4. Phenotype

5. Fitness function   6. Mutation   7. Neuron
8. Topology.

Long Answer type Questions:

1. What do you mean by Genetic Algorithm? Write down the main steps involve in Genetic Algorithm.

2. Explain some areas where we can use Genetic Algorithm.

3. Let, P = (C1, C2, C3, ..., Cn) means the salesmen move from city C1 to C2, C2 to C3, C3 to Cn. There are four cities that a salesperson will pass. The cities are A, B, C and D. The journey starts from A and ends at A as well. The distance between cities is shown in Figure below:
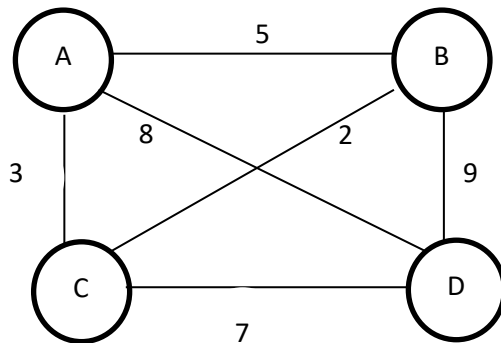
4. What do you mean by Neural Network? Explained components associated with it.

5. Explain some properties of Artificial Neural Network.

3.7 REFERENCES AND SUGGESTED READINGS

[1] Training Feedforward Neural Networks Using Genetic Algorithms: *David J. Montana and Lawrence Davis, BBN Systems and Technologies Corp., 10 Mouiton St. Cambridge, MA 02138*

[2] Artificial Neural Networks: *Slides modified from Neural Network Design by Hagan, Demuth and Beale,* Berrin Yanikoglu, DA514– Machine Learning

[3] Genetic Algorithms and Neural Networks *by D. WHITLEY*

[4] "*Neural Networks using Genetic Algorithms", International Journal of Computer Applications (0975 – 8887) Volume 77– No.14, September 2013*

[5] Artificial Neural Networks by *Ahmad Aljebaly*

[6] "*Combining Genetic Algorithms and Neural Networks: The Encoding Proble"* A Thesis The University of Tennessee, Knoxville Philipp Koehn December 1994

[7] "*Genetic Algorithms"* by *Muhannad Harrim*

[8] *"Traveling Salesman Problem Solution using Genetic Algorithm,* ISSN- 2394-5125 Vol 7, Issue 1, 2020

BLOCK III:
GRAPH ALGORITHMS

UNIT 1   INTRODUCTION TO GRAPHS

Unit Structure:

1.1   INTRODUCTION

In this unit, you will learn the concept of graph, and its different ways of representation: Adjacency matrix and Adjacency List. You will also learn the different graph traversal techniques namely Depth-first search (DFS) and Breadth-first search (BFS). How the DFS and BFS techniques along with the algorithms and the data structures used to accomplish the task. The time complexity of the algorithms will be discussed in this unit along with few demonstrations of the techniques. You will also learn about one of the popular sorting techniques i.e. Topological Sort along with its time complexity analysis.

1.2   UNIT OBJECTIVES

After going through this unit, you will be able to:

- *Understand* the fundamental concept of Graphs.
- *Know* different ways of representing a graph.
- *Define* graphs and its types.
- *Analyze* the time complexities of graph traversal techniques.
- *Describe* Topological sort.

1.3   BASIC TERMS AND THEIR DEFINITIONS

Graph is a nonlinear ADT (Abstract Data Type) which can be used as a modeling tool to represent non-hierarchical relationship. A graph is encountered often in our daily lives like social networks, road networks, computer networks and so on. For instance, road networks or social networks etc can be easily modeled using a graph (See Fig 1.1). When you say graph is a non-linear ADT, it implies that the data stored are distributed and should not be stored contiguously like Arrays etc. Unlike Trees, in a graph, there can be any number of predecessors and successors and a node can have multiple parents and descendants. To understand the graph, you need to get an introduction of the basic terms and terminologies associated with the graph. Some, of the basic terms in graph

theory are given in the subsequent subsections.
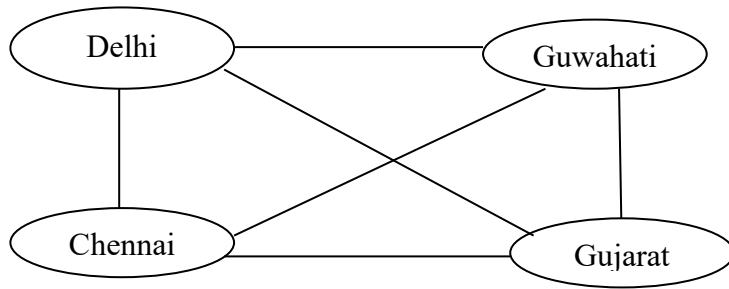
Fig. 1.1: Sample graph

i) Graph

A graph is a non-linear data structure that consists of finite set of nodes V and a finite set of edges E that connects the vertices. Mathematically, a graph can be represented as $G(V,E)$. The nodes are also called as vertices and edges as arcs. A Graph G (V, E) with 4 vertices, set of vertices $V = \{A, B, C, D\}$ and six edges i.e. set of edges $E = \{(A, B), (A, C), (A, D), (B, C), (C, D), (D, B)\}$ is shown in the Figure 1.2.

ii) Order of a Graph

The total number of nodes/vertices represents the order and edges represents the size of a graph $G(V, E)$. The size of a graph is denoted as $|E|$.

iii) Directed Graph

A directed graph is a set of vertices and a set of links between the vertices, i.e. the vertices of the graph are connected by the edges and where all the edges are directed from one vertex to another as shown in Fig. 1.2. A directed graph is also referred to as a digraph or a directed network.
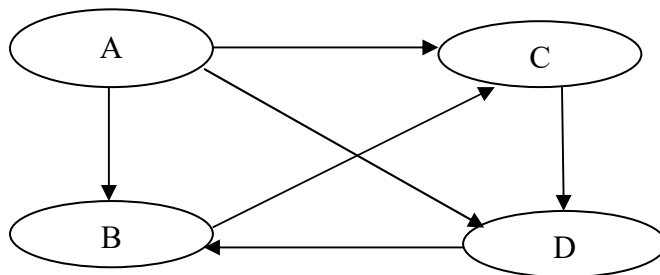
Fig. 1.2: Directed graph

iv) Undirected Graph

An undirected graph is a set of vertices and a set of links between the vertices, i.e., the vertices of the graph are connected by the edges and where the edges are not associated with the directions with them as shown in Fig. 1.3. Every edge in a undirected graph is equivalent to two edges connecting two vertices.
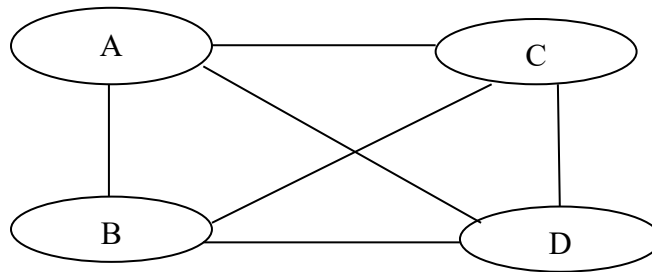
Fig. 1.3: Undirected graph

Here, $V = \{A, B, C, D\}$ and $E = \{(A,C), (A,B), (A,D), (B.A), (B,C), (B,D), (C,A), (C,B), (C,D), (D,A), (D,B), (D,C)\}$. It can be observed that and undirected graph can be replaced by two directed edges.

v) Walk

When we traverse a graph we get a walk. In other words a walk is a sequence of vertices and edges of a graph where vertex can be repeated and edges cannot be repeated. If the last vertex and first vertex are same then it is known as closed walk, otherwise the walk is referred to as open walk. There are no restrictions in the number of edges and vertices in a walk. In Fig. 1.3 the traversing of graph in a manner A-> B -> C -> D -> A is a closed walk, and A-> B -> C -> D-> is an open walk.

vi) Trail

A walk is known as a trail, if no edges appear more than once in a walk, however vertex can be repeated.

vii) Circuit

When a graph is traversed in such a manner that no edge is repeated but vertex is repeated, it is known as a circuit. It is a closed trail. For instance in Fig. 1.3 the A-> B -> C -> D -> A->C is a circuit.

viii) Path

A path can be defined as the sequence of vertices that are followed in order to reach some destination vertex v from the initial vertex u where, we do not repeat a vertex and nor we repeat an edge while we traverse the graph. A path is an open walk.

ix) Closed Path

A path is known as a closed path if the initial vertex u is same as terminal vertex v. A path will be closed path if $v = u$.

x) Simple Path

A path P is called as closed simple path if all the vertices of the graph are distinct with an exception $v = u$. then such. Where, v is the destination vertex and u is the initial vertex.

xi) Cycle

A cycle can be defined as the path where no edges are repeated neither vertices except the first and last vertices i.e. to get a cycle we can repeat the starting and ending vertex only. In simple terms, in cycle both the vertices and edges are not repeated. If all the vertices of a graph are visited only once except the first vertex, the cycle is known as Hamiltonian cycle.

xii) Loop

A loop is an edge that connects a vertex to itself. It is also known as a self loop or buckle. A graph that does not contain any loops or multiple edges is called a simple graph.

xiii) Digraph

A digraph is a directed graph in which each edge of the graph is associated with a certain direction and traversing is only possible in that direction.

xiv) Complete Graph

A complete graph is the one in which each pair of vertices is connected by an edge. A complete graph contain n(n-1)/2 edges where n is the number of vertices in the graph. The Figure 1.4 as shown is a complete graph, where there are 5 vertices and 10 edges.
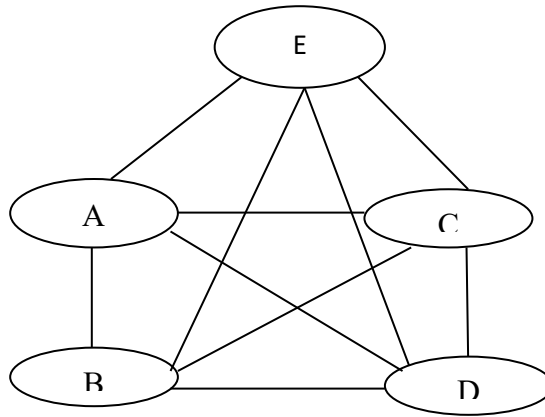
Fig. 1.4: Complete graph

xv) Degree of a Vertex

The number of edges connecting to a vertex of a graph is the degree of that vertex. A vertex with a degree zero is known as an isolated vertex.

CHECK YOUR PROGRESS

1. Graph is a _____ data structure which can be used as a modeling tool to represent _____relationship.
2. The total number of edges represents the _____ of a graph.
3. In a _____edge cannot be repeated but vertex can be repeated.
4. In a _____no edge and vertex can be repeated except the first and last vertices.
5. A _____is an edge that connects a vertex to itself in a graph.
6. If there are *n* vertices in a graph then a complete graph can contain _____ number of edges.
7. In an _____ graph, the edges are not associated with the directions with them.
8. In a _____ graph, the edges are associated with the directions with them.

1.4 GRAPH REPRESENTATIONS

When we say "graph representation", we just mean the approach that will be utilized to store a graph in the computer's memory. A graph $G(V,E)$ can be represented in the following two ways:

a) Adjacency Matrix.
b) Adjacency List.

1.4.1 Adjacency Martix

In this type of representation, a graph is represented as a matrix M of dimension n x n where $n = |V|$, number of vertices. It is a sequential representation of graph. An entry M_{ij} in the adjacency matrix representation of a graph G will be 1 if there exists an edge between any two vertices v_i and v_j. It can be represented as follows:

$$M[i,j] = \begin{cases} 1 \; if \; there \; exists \; any \; edge \; between \; v_i \; and \; v_j \\ 0 \quad otherwise \end{cases}$$

For example, consider the graph shown in Figure 1.5(a) and its corresponding adjacency matrix as shown in Figure 1.5(b).
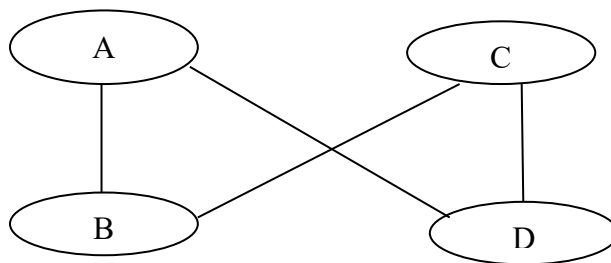
Fig, 1.5 (a): Sample undirected graph. (b): Adjacency matrix

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

It is to be observed that the sample graph shown in Figure 1.5 (a) is an undirected graph. In case of undirected graphs $M[i, j] = M[j, i]$, i.e. it holds the symmetry property.

For directed graph, the adjacency graph representation is shown in Figure 1.6 (a) and 1.6 (b).
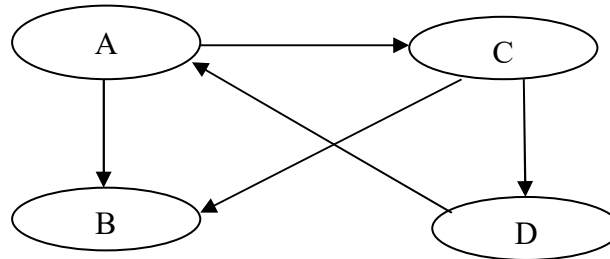
Fig. 1.6 (a): Sample directed graph

	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	1	0	1
D	1	0	0	0

Fig. 1.6 (b): Adjacency matrix representation of the sample directed graph

The entry in the matrix, $M[i, j] = 1$ only when there is an edge directed from vertex v_i and v_j otherwise it is 0.

Adjacency matrix can also be used to represent weighted graphs. In weighted graph, a weight is associated to every edge connecting any pair of vertices. The length of the path is equal to the sum of weights that fall in that path.

In case of weighted graphs the non-zero entries in adjacency matrix M are represented by the weight of respective edges instead of filling it with 1 and filled with infinite otherwise. It can be represented as follows:

$$M[i, j] = \begin{cases} w_{ij} & \text{if there exists any edge between } v_i \text{ and } v_j \\ \infty & \text{otherwise} \end{cases}$$

In Figure 1.7(a) a weighted graph is shown and its adjacency matrix representation is shown in Figure 1.7(b).
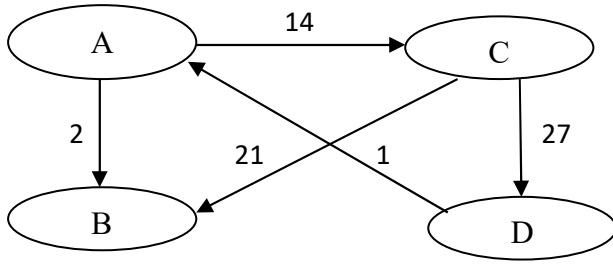
Fig. 1.7 (a): Sample weighted graph

	A	B	C	D
A	∞	2	14	∞
B	∞	∞	∞	∞
C	∞	21	∞	1
D	1	∞	∞	∞

Fig. 1.7 (b): Adjacency matrix representation of the sample weighted graph

It is observed that adjacency matrix is sparse if there are lesser number of edges compared to the maximal possible number of edges. Thus, this kind of representation consumes memory space. It is ideal if the graph is dense otherwise, the adjacency list representation of graph is preferable to represent a sparse graph.

_____

1.4.2 Adjacency List
_____

An adjacency list is another kind of graph representation where a linked list is used to represent the neighbors of a vertex.

For each node in the graph, an adjacency list is kept, which contains the node value as well as a pointer to the node's next adjacent node. If all neighbouring nodes have been traversed, store NULL in the list's last node's pointer field. In an undirected graph, the sum of the lengths of adjacency lists is equal to twice the number of edges of the graph.

Consider the directed graph shown in Figure 1.8(a) and check the adjacency list representation of the graph in the Figure 1.8(b).
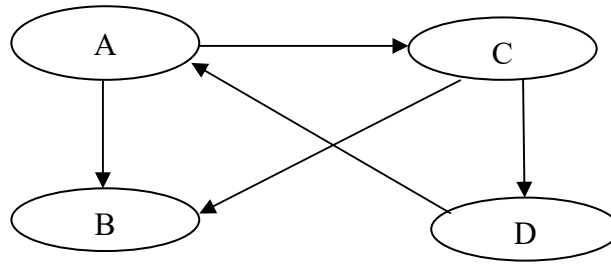
Fig. 1.8 (a): Sample directed graph
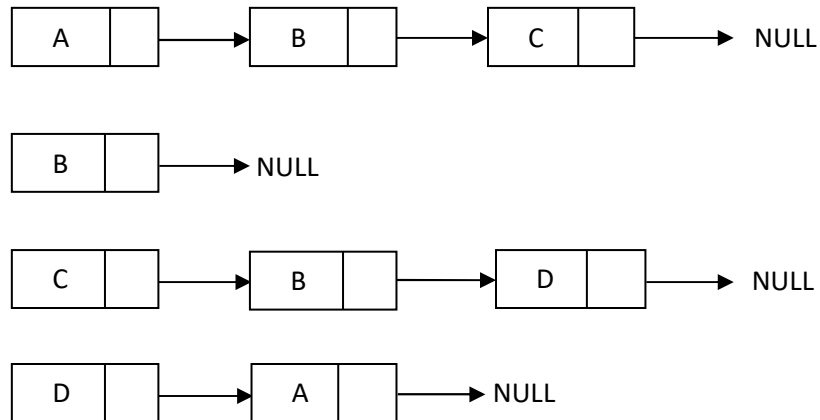
Fig. 1.8 (b): Adjacency list representation of the sample directed graph

CHECK YOUR PROGRESS

9. There are two ways of representing a graph _____ and _____.

10. Define weighted graph.

11. In adjacency list representation of graph a _____ is used to represent the neighbors of a vertex.

12. In adjacency matrix, the entry in the matrix is equal to _____ only when there is an edge directed from vertex v_i and v_j otherwise it is entered as _____.

13. In case of weighted graphs the non-zero entries in adjacency matrix are represented by the _____ of respective edges.

1.5   GRAPH TRAVERSAL TECHNIQUES
_____

Graph traversal is a way of processing the vertices/nodes of a graph such that every vertex is visited only once. There are basically two types of graph traversal techniques:

    a) Depth-first Search (DFS) and

    b) Breadth-first Search (BFS).

These techniques are used to search or traverse a graph in linear time. Both the techniques can take directed and undirected graph and produce DFS and BFS trees, respectively. The two techniques are discussed in the subsequent subsections.

_____

1.5.1 Depth-first Search
_____

Depth-first search (DFS) algorithm is a systematic way of traversing the nodes of a directed or undirected graph G. The nodes are traversed in such a way that every node is visited only once. It starts with an initial node to process and then goes deeper to process its descendants before processing the adjacent nodes. It goes deeper and deeper to process the descendants or children of subsequent nodes until the target node or the node with no descendant is found. Once a dead end is encountered, the algorithm, then backtracks towards the most recent node that is yet to be explored completely. The DFS algorithm uses stack data structure to keep track of the nodes to be used to start the search when dead ends occur. The searching is continued till the stack becomes empty which implies that all the nodes of the graph are fully explored or processed.

A DFS algorithm can be applied recursively or non-recursively. The steps followed in the iterative DFS algorithm are as follows:

Step 1: Initialize an empty stack and mark the status of every node as unvisited in the graph.
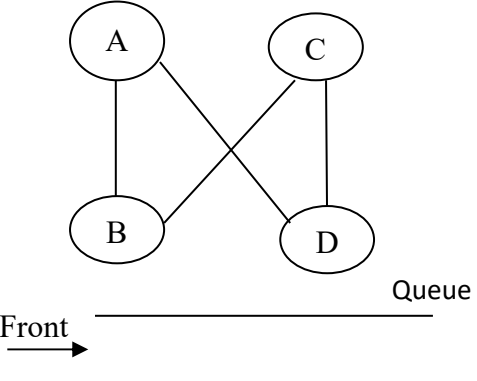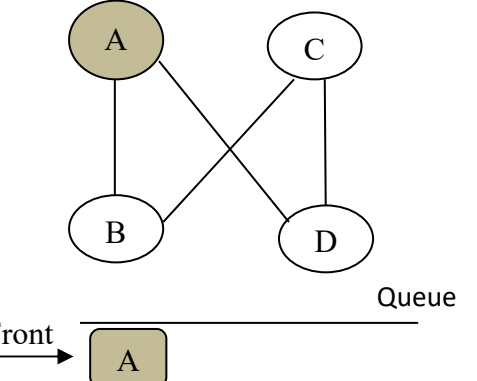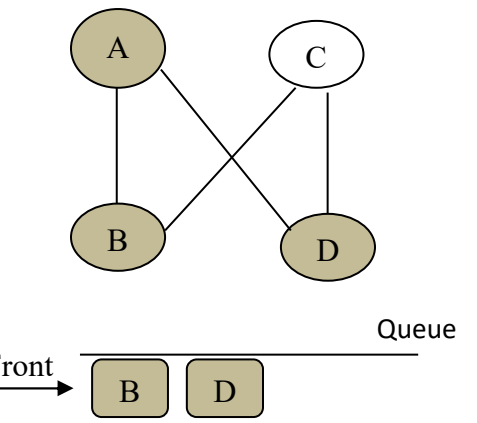
Step 2: Explore a node and mark its status as visited and push it into the
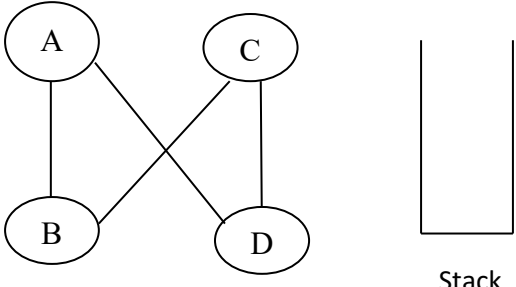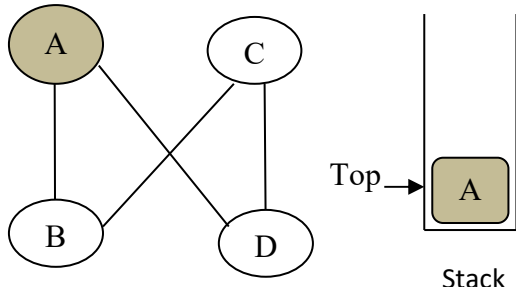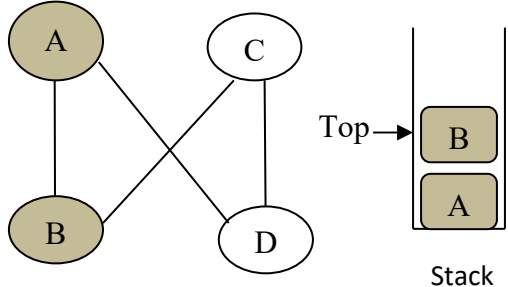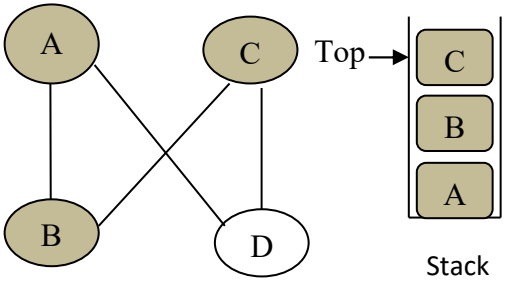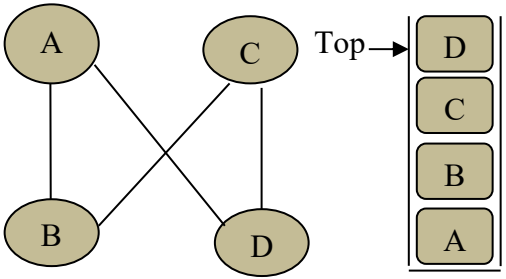
stack and display it.

Step 3: Pop a node from the stack if no adjacent node of it whose status is unvisited is found.
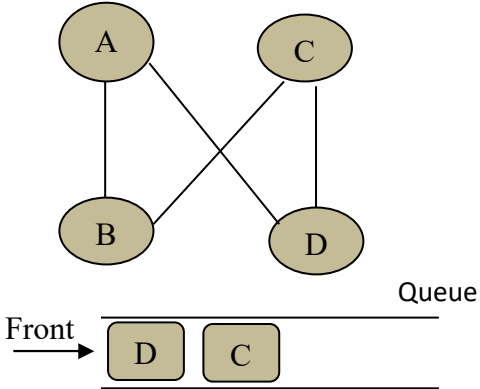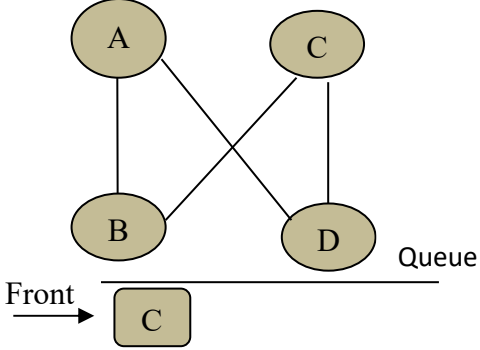
Step 4: Repeat the Steps 2 and 3 until the stack is empty.

Example: Let us try to understand with an example. Consider an undirected graph having four nodes A, B, C and D connected by the edges. We need to traverse the graph using DFS algorithm. The illustration of the algorithm is shown below:

Traversal	Description of Step
	Initialize a stack and mark the status of the nodes of the graph as unvisited.
	Take node A as starting node and push it into the stack. Mark its status as visited and display it. Explore the adjacent nodes of A whose status is unvisited. Here, there are two adjacent unvisited nodes B and D. You can take any of them to explore further.
	We took the adjacent node B to explore. Mark it as visited and push it into the stack and display B. Now, you explore the adjacent node of B. So, next explore node C.

Traversal	Description of Step
	Mark node C as visited and push it into the stack and display it. Now, you explore the adjacent nodes of C. You will get nodes B and D. Both the nodes are the adjacent nodes, but node D will be considered next to be explored due to its unvisited status.
	Mark node D as visited and push it into the stack and display it. The adjacent nodes of D are A and C but both are visited, so it cannot be explored further. So, now you backtrack with the help of the stack.

It is to be observed that the node D does not have any further adjacent nodes yet to be explored, so you need to backtrack by popping out D from the stack and keep popping until you find any node which is not yet fully explored or in other words whose status is unvisited. In this case, there are no nodes with the unvisited status, so pop the nodes one by one until the stack becomes empty. Hence, all the four nodes of the graph are traversed. The printing sequence of the graph will be as A -> B - > C -> D.

Time complexity analysis of DFS algorithm

Let us consider, a graph with n number of nodes and m number of edges (directed or undirected). As you know that in DFS algorithm every node is visited only once therefore the total time taken to traverse or visit all the nodes of the graph is n. However, the edges of the graph is traversed twice, as you can observe from the illustration, thus the total time to traverse all the edges will be $2m$. Hence, we can say that the total time taken to traverse the graph using DFS algorithm is $n + 2m$ which can be stated using asymptotic notation as $O(n + m)$.

1.5.2 Breadth-first Search

Breadth-first search is an algorithm is a systematic way of traversing the nodes of a directed or undirected graph G. It differs with the depth-first search technique is that the BFS processes the nodes of the graph in a level wise manner, in other words the nodes at the same level are explored first before exploring the nodes at the next level. The algorithm first selects a node to be a root node to start with and then explores the nodes nearer to the root node. The BFS algorithm uses a queue data structure to keep track of the nodes to be explored. The queue is used to store the sibling nodes of the current node being explored and uses the queue to get the next node to be explored once it encounters a dead-end in any iteration. The searching is continued till the queue becomes empty which implies that all the nodes of the graph are fully explored or processed.

Like a DFS algorithm, BFS also can be applied recursively or non-recursively. The steps followed in the iterative BFS algorithm are as follows:

Step 1: Initialize an empty queue and mark the status of every node as unvisited in the graph.

Step 2: Select a start node to visit, mark its status as visited and enqueue it into the queue.

Step 3: Dequeue a node from the queue, display it and explore its adjacent unvisited node.

Step 4: Enqueue the explored adjacent node it into the queue.
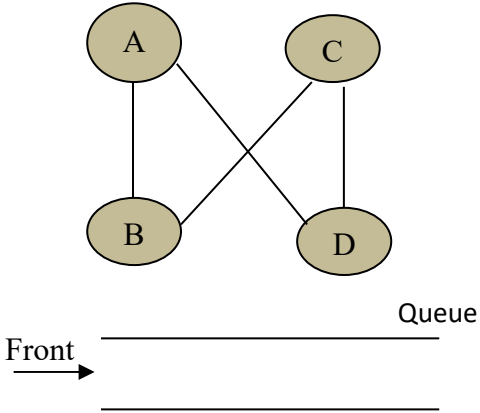
Step 5: Repeat the Steps 3 and 4 until the queue is empty.

Example: Let us try to understand with an example. Consider an undirected graph having four nodes A, B, C and D connected by the edges. We need to traverse the graph using BFS algorithm. The illustration of the algorithm is shown below:

Traversal	Description of Step
  Queue Front	Initialize a queue and mark the status of the nodes of the graph as unvisited.
  Queue Front → [A]	Take node A as starting node and enqueue it into the queue. Mark its status as visited.
  Queue Front → [B] [D]	Dequeue node A, display it and explore its adjacent nodes whose status is unvisited. Here, nodes B and D are the adjacent nodes of A. Enqueue the nodes B and D into the queue. Mark them as visited.

Traversal	Description of Step
	Dequeue node B from the queue as it is pointed by front pointer of the queue. Display it and explore its adjacent nodes. Here, nodes A and C are the adjacent nodes of B but node A is already visited. So, enqueue node C into the queue and mark it as visited.
	Dequeue node D from the queue as it is pointed by front pointer of the queue. Display it and explore its adjacent nodes. Here, nodes A and C are the adjacent nodes of D but both the nodes are already visited.
	Dequeue node C from the queue as it is pointed by front pointer of the queue. Display it and explore its adjacent nodes. Here, nodes B and D are the adjacent nodes of C but both the nodes are already visited. So, stop the exploration as queue is empty and all the nodes are fully explored.

Hence, all the four nodes of the graph are traversed. The printing sequence of the graph will be as A -> B - > D -> C. It is to be noted that in both DFS and BFS methods the printing sequence of the nodes differs with the selection of the initial starting node.

Time complexity analysis of BFS algorithm

Let us consider, a graph with *n* number of nodes and *m* number of edges (directed or undirected). As you know that in BFS algorithm every node is visited only once therefore the total time taken to traverse or visit all the nodes of the graph is *n*. However, the edges of the graph is traversed twice, as you can observe from the illustration, thus the total time to traverse all the edges will be 2*m*. Hence, we can say that the total time taken to traverse the graph using BFS algorithm is *n* + 2*m* which can be stated using asymptotic notation as *O* (*n* + *m*). Thus, the time complexity of both the DFS and BFS algorithm to traverse a given graph is same.

――――――――――――――――――――――――――――――――――――

1.5.3 Difference between DFS and BFS Algorithm
――――――――――――――――――――――――――――――――――――

Both the Depth-first search and Breadth-first search algorithms are used to traverse a graph. The time complexity of both the algorithms is $O(|V| + |E|)$, if we use adjacency list to represent the graph, otherwise it is $O(|V|^2)$ if we use adjacency matrix for graph representation. However, the main difference between them is their approach in traversing the graph. Some of the differences between the two methods are given below:

S. No	Depth-first Search	Breadth-first Search
1	DFS uses stack data structure.	BFS used queue data structure.
2	DFS explores the child nodes before the sibling nodes.	BFS explores the sibling nodes in a level by level manner. Where all nodes in a particular level are explored first before moving into the next level.
3	Best suited to find nodes which are far away from the source node.	Best suited to find nodes which are nearer to the source node.
4	The outcome can be a forest.	The outcome is a tree (BFS tree).

STOP TO CONSIDER

Breadth-first search and Depth-first search are two graph traversal algorithms. Both are useful in many graph algorithms and can be applied into directed or undirected graph. Some of the applications of these algorithms are finding single source shortest path in a graph, detect cycles in undirected graphs, job scheduling and so on.

1.6   TOPOLOGICAL SORT
_____

Topological sorting is a sorting technique of the nodes of a graph. The technique works only with directed acyclic graph (DAG). The ordering of the nodes is done in such a way that if there exist any edge directed from the node *u* and node *v,* then topological sort outputs *u* before *v.* As the graph as to be acyclic the topological sort guarantees the ordered output of the nodes however there may be more than one output and may not be unique always.

To begin with topological sort, you need to consider few of the following terms and concepts:

i). Cyclic Graph

A graph that consists of at least one cycle in a graph. We know that in a

graph a cycle is formed whenever the starting and ending vertices are repeated in a path. Figure 1.9 shows a cyclic graph with having a path from node A back to itself (A->C->B->A).
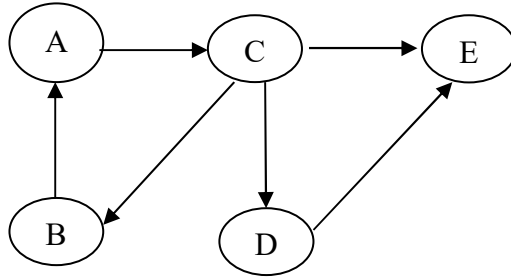
Fig. 1.9: A sample cyclic graph

ii). Acyclic Graph

A graph that do not consist of any cycle in it which means that no node of the graph can be traversed back to itself. Figure 1.10 shows an acyclic graph.
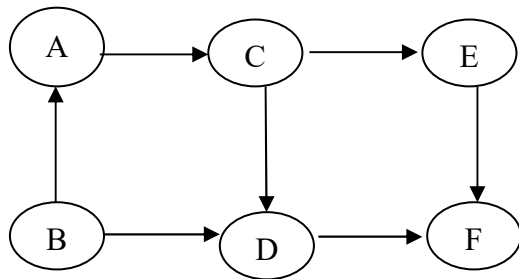
Fig. 1.10: A sample acyclic graph

iii). Directed Acyclic Graph (DAG)

A directed graph, which does not consist of any cycle, is known as directed acyclic graph (DAG). The graph shown in Figure 1.10 can also be referred to as directed acyclic graph.

iv). Indegree of a Node

In a directed graph, the number of edges leading into the node gives us the indegree of the node. If we consider the DAG shown in Figure 1.10, the indegree of the nodes are: Node A has indegree 1 as there is only one edge coming into the node from node B. The indegree of node B is 0,as there are

no edge leading into it. Similarly, the indegrees of the node C, D, E and F are 1, 2, 1 and 2 respectively.

A node with degree 0 is also known as an isolated node. And root node always has indegree 0.

v). Outdegree of a Node

In a directed graph, the number of edges leading away from the node gives us the outdegree of the node. If we consider the DAG shown in Figure 1.10, the outdegree of the nodes are: Node A has outdegree 1 as there is only one edge coming out of the node towards node C. Similarly, the outdegree of the nodes B, C, D, E and F are 2, 2, 1, 1 and 0 respectively.

Topological Sort Algorithm

Topological sort graph traversal algorithm which does the linear ordering of nodes of a graph. It works only with directed acyclic graph (DAG) and there is at least one topological ordering of a DAG. Now, given a directed acyclic graph G (V, E), to begin with topological sort perform the following steps:

Step 1: Count indegree of nodes of the graph and get the node with indegree 0 and insert into a list.
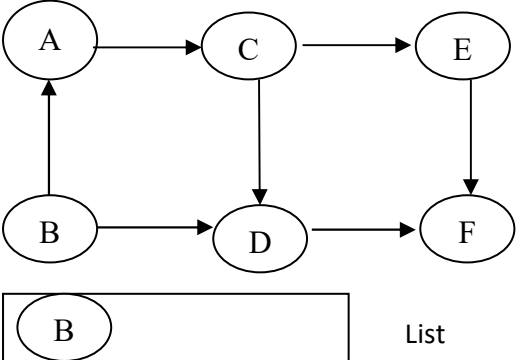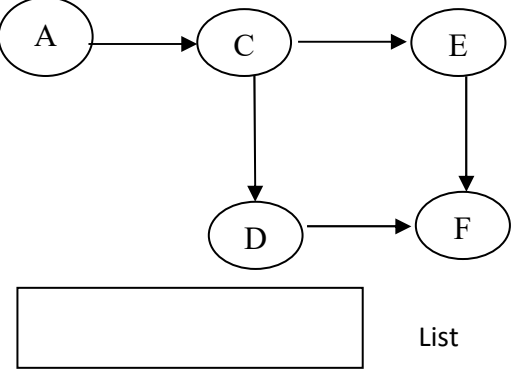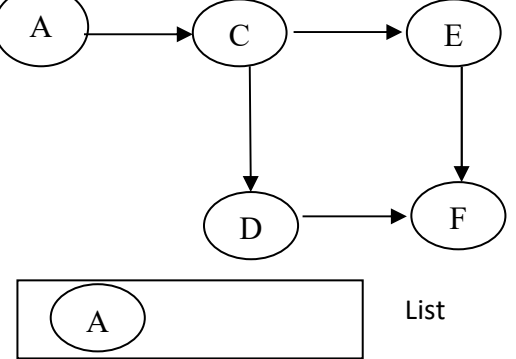
Step 2: Remove a node from the list and display it.

Step 3: Delete the node and every edge coming out of it from the graph and get a new subgraph.

Step 4: Repeat the steps 1 to 3 until all the nodes are traversed and the list becomes empty.
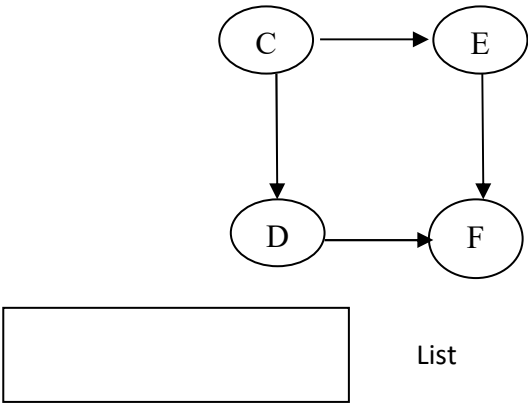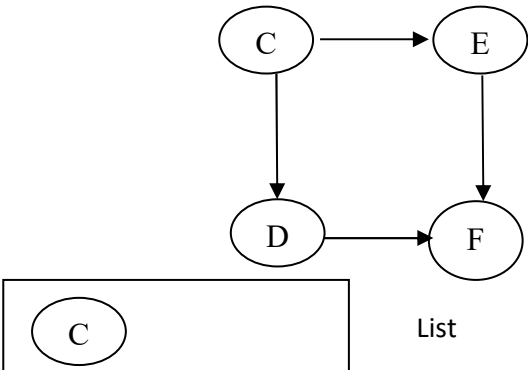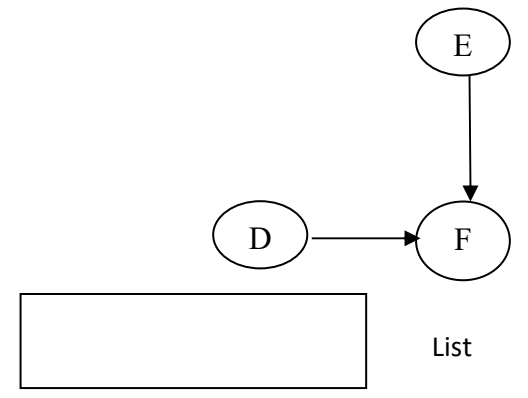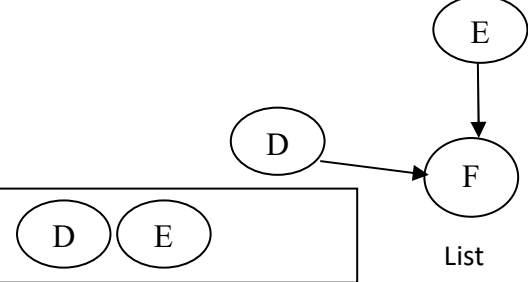
Here, a list is used to keep track of the nodes however you may use stack or queue data structure also to perform the task.
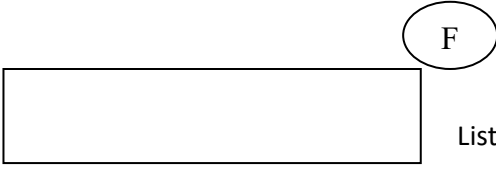
Example: Let us try to understand topological sort with an example. Consider an directed acyclic graph (DAG) having six nodes A, B, C, D, E and F and seven directed edges. We need to traverse the graph using topological sort. The illustration of the algorithm is shown below:

Traversal	Description of Step
B List	Count the indegree of nodes of the graph and get the node with indegree 0. Here, node B has indegree 0. Insert it into the list.
List	Remove node B from the list and display it. We get a new subgraph by deleting the node B and edges coming out of the node.
A List	Count the indegree of nodes of the subgraph and get the node with indegree 0. Now, node A has indegree 0. Insert it into the list.

Traversal	Description of Step
List	Remove node A from the list and display it. We get a new subgraph by deleting node A and edge coming out of the node.
List	Count the indegree of nodes of the subgraph and get the node with indegree 0. Now, node C has indegree 0. Insert it into the list.
List	Remove node C from the list and display it. We get a new subgraph by deleting node C and edges coming out of the node.
List	Count the indegree of nodes of the subgraph and get the node with indegree 0. Now, nodes D and E has indegree 0. Insert them into the list.

| **Traversal**	**Description of Step**
E → F E List	Remove node D from the list and display it. We get a new subgraph by deleting node D and edge coming out of the node.
F List	Remove node E from the list and display it. We get a new subgraph by deleting node E and edge coming out of the node.
F F List	Count the indegree of nodes of the subgraph and get the node with indegree 0. Now, node F is left and with indegree 0. Insert it into the list.
List	Remove node F from the list and display it. Stop now as we have exhausted traversing all the nodes of the graph and the list is empty.

Hence, all the six nodes of the graph are traversed. The printing sequence of the graph will be as B, A, C, D, E, F. It is to be noted that the ordered sequence of the nodes can also be B, A, C, E, D, F. Thus, we get more than one unique order of nodes using topological sort algorithm.

Time complexity analysis of Topological sort algorithm

Let us consider, a directed acyclic graph G (V, E) with $n = |V|$ and $m = |E|$. In the topological sort algorithm, every node is visited only once therefore the total time taken to traverse or visit all the nodes of the graph is n. Similarly, the edges of the graph is traversed as you can observe from the illustration, thus the total time to traverse all the edges will be m. Hence, we can say that the total time taken to traverse the graph is $n + m$ which can be stated using asymptotic notation as O $(n + m)$. Thus, the time complexity of topological sort algorithm to traverse a given graph is linear.

Why Topological sort algorithm cannot be applied to cyclic graph?

To answer this question let us consider the directed cyclic graphs shown in Figures 1.11 (a) and 1.11 (b).
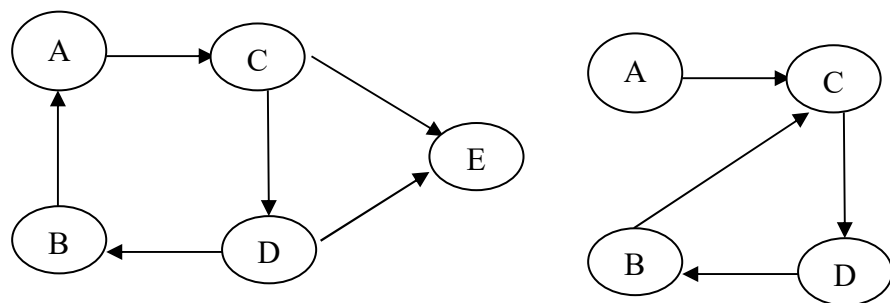
Fig. 1.11: (a) Sample cyclic graph   (b) Sample cyclic graph

If you observe carefully, the directed graph in Figure 1.11 (a), there is a cycle consisting of the nodes A, C, D, B. There is no node in the graph shown with indegree 0, so topological sort algorithm cannot be applied in it. If you consider the directed graph shown in Figure 1.11 (b), there is one node A whose indegree is 0, however, once we delete the node and the edge coming out of it, we are left with a subgraph consisting of node B, C and D forming a cycle. Hence, you cannot carry on the topological sorting further as no nodes will have an indegree 0.

14. With adjacency list representation of graph, the time complexity of BFS algorithm is _____.

15. The time complexity of topological sort algorithm is _____.

16. Define indegree and outdegree of a node.

17. DFS algorithm traverses the _____ nodes before the _____ nodes.

18. A _____ graph has at least one cycle in it.

19. State whether the following statements are true or false:

    a. Topological sort algorithm can produce only one order of the nodes of a graph.

    b. In BFS algorithm, we use queue data structure.

    c. Topological sort algorithm can be performed in directed acyclic graph (DAG) only.

1.7   SUMMING UP
_____

- A graph *G(V,E)* is a non-linear data structure that consists of finite set of nodes *V* and a finite set of edges *E* that connects the vertices.

- A non-linear data structure is one in which the data pieces are not placed in any particular order and are instead distributed across the plane.

- For implementation, we need to represent the graph in the computer's memory, and there are two ways to do so: adjacency matrix and adjacency list.

- Traversing a graph or searching a graph implies visiting every nodes or vertices of a graph.

- There are basically two types of graph traversal techniques: Depth-first Search (DFS) and Breadth-first Search (BFS) and both traverse a graph in linear time.

- A graph is represented as a matrix M of dimension n x n in adjacency matrix representation, with $n = |V|$. It represents a graph in sequential order. If an edge exists between any two vertices v_i and v_j, the item M_{ij} in the adjacency matrix representation of the graph G will be 1 otherwise 0.

- An adjacency list is a type of graph representation that uses a linked list to represent a vertex's neighbours.

- BFS explores the nodes of the graph in a level-by-level fashion, that is, nodes at the same level are explored first, followed by nodes at the next level.

- The DFS method begins by exploring an initial node then continues deeper to process its descendants before moving on to its adjacent node.

- The linear ordering of nodes in a graph is accomplished using the topological sort algorithm. It can only be used with directed acyclic graphs (DAGs) and have at least one topological ordering.

1.8 ANSWERS TO CHECK YOUR PROGRESS

_____

1. non linear, non-hierarchical

2. size

3. trail

4. cycle

5. loop

6. n(n-1)/2

7. undirected

8. directed

9. adjacency matrix, adjacency list

10. A weighted graph is a graph where every edge (directed or undirected) is assigned some numerical weight.

11. linked list

12. 1, 0

13. weight

14. $O(|V|+|E|)$

15. $O(|V|+|E|)$

16. In a directed graph, the number of edges leading into a node gives the indegree of the node and the number of edges leading away from a node gives the outdegree of the node.

17. child, adjacent

18. cyclic

19. a). False b). True c). True

_____

1.9 POSSIBLE QUESTIONS
_____

Short Answer Type Questions

1. What is a graph?
2. What is the purpose of graph representation?
3. What are the different types of graph representation?
4. Define an acyclic graph.
5. Define a complete graph.
6. What do you mean by graph traversal?
7. What are the different types of graph traversal techniques?
8. Draw a graph with five nodes and seven edges where one edge should be a loop.
9. Define a connected graph.
10. How do you get the size of a graph?
11. Define indegree of a node.
12. What is an ADT?
13. Define DAG.
14. Define outdegree of a node.
15. What is a circuit in graph?
16. What data structure is used in Depth-first search technique?

17. Why do we use queue data structure in Breadth-first search?

18. What is the output of topological sort method?

19. What is non-linear data structure?

Long Answer Type Questions

1. Explain the adjacency matrix representation of a graph with a suitable example.

2. Explain the adjacency list representation of a graph with an example.

3. Differentiate between connected and non-connected graph.

4. Consider the graphs shown in Figure 1.12 (a) (b) and do the following:

    a. Perform the BFS traversal.

    b. Perform the DFS traversal.
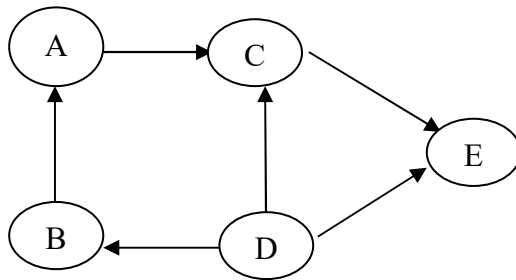
Fig. 1.12. (a): A directed graph
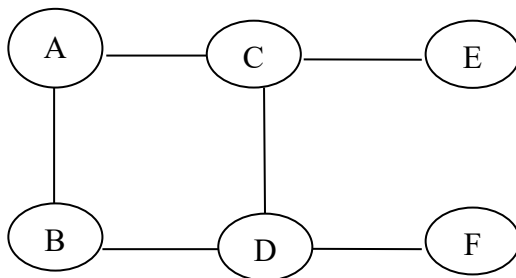
Fig. 1.12. (b): An undirected graph

5. Explain the advantages and disadvantages of adjacency matrix representation of a graph.

6. Explain the advantages and disadvantages of adjacency list representation of a graph.

7. Using the DAG as shown in Fig. 1.12 (a), perform the topological sort in the graph.

8. "Topological sort can be applied only to directed acyclic graph". Justify the statement with a suitable example.

9. Describe the various components of a graph.

10. Draw a directed graph using the following adjacency matrix:

	U	V	W	X
U	1	0	1	0
V	1	0	1	0
W	0	0	0	1
X	0	1	0	0

Also find the adjacency list representation of the graph.

11. Explain the Breadth-first search algorithm. Derive its time complexity.

12. Explain the Depth-first search algorithm. Derive its time complexity.

13. Explain the procedure to determine the presence of a cycle in a graph.

14. Differentiate between BFS and DFS algorithms.

_____

1.10   REFERENCES AND SUGGESTED
_____

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press.

- Sridhar S., Design and Analysis of Algorithms, Oxford University Press, 2014.

UNIT 2   MINIMUM SPANNING TREE

Unit Structure:

2.1 INTRODUCTION

In this unit, you will learn the concept of spanning trees, weighted graph and minimum spanning tree. You will also learn the algorithms to find minimum spanning tree: Kruskal's and Prim's algorithm. These algorithms are basically optimal graph algorithms that use the concept of Greedy technique to solve the optimization problems. You will learn the important concepts of greedy techniques and optimization problem and about the data structures used to accomplish the task. The time complexity of the algorithms will be discussed in this unit along with some demonstrations of the techniques.

2.2 UNIT OBJECTIVES

The unit is an attempt to learn the concepts of spanning trees. After going through this unit, you will be able to:

- *Understand* the fundamental concept of minimum spanning tree.
- *Know* different algorithms to find minimum spanning tree.
- *Analyze* the time complexities of algorithms used to find the minimum spanning trees.
- *Discuss* greedy approach and optimization problems.

2.3 BASIC TERMS AND THEIR DEFINITIONS

The problem of minimum spanning or MST was first formulated in the year 1926 by a Czech mathematician named Otaker Boruvka. Then in 1930 Vojtech Jarnik had developed an algorithm for the MST problem. Later, it was rediscovered by Joseph Kruskal in the year 1956 and Robert Prim in 1957 and Edsger W. Dijkstra in 1958. A spanning tree is subgraph of a graph $G(V,E)$ and minimum spanning tree is a spanning tree satisfying certain properties. The MST problem is an optimization problem in graph whose solutions uses the greedy approach. The greedy approach is also used in developing algorithms for finding the shortest path problems.

To understand the MST, you need to get an introduction of the basic terms and terminologies associated with it. Some of the basic terms and their definitions related to MST are given below:

.i) Tree

A tree is basically a data structure consisting of nodes and edges connecting the nodes in a hierarchical manner. In graph theory it is defined as an undirected graph where any two nodes are connected by only one undirected edge not forming any cycle. A tree with 5 nodes and 4 edges is shown in the Figure 2.1.
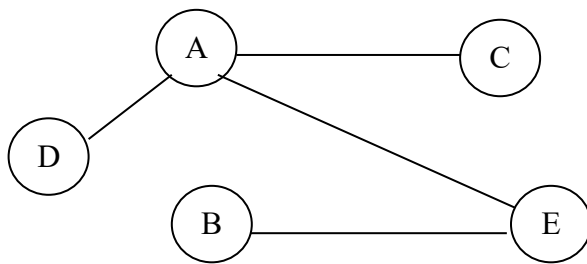
**Fig. 2.1: A tree with 5 nodes and 4 edges**

ii) Subgraph

Given a graph $G(V, E)$ a subgraph $G(V', E')$ is the subset of the graph where its edges $E' \subseteq E$ and vertices $V' \subseteq V$. One may obtain a subgraph by deleting vertices and edges of a graph. A graph may have more than one subgraphs.
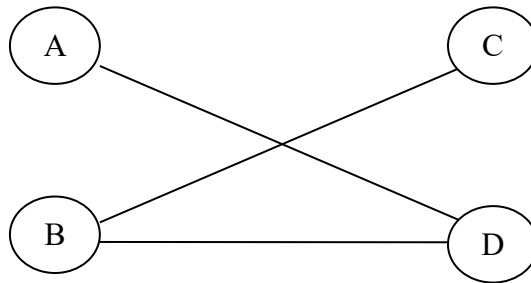
iii) Weighted graph

A weighted graph is graph, where a weight is associated to every edge connecting any pair of vertices of the graph. The graph may be directed or undirected graph.

iv) Spanning Tree

Given a graph $G(V, E)$, spanning tree is a subgraph $G(V', E')$ which includes all the vertices of the graph G connected by the edges without forming any cycle. Consider the graph shown in Figure 2.2 (a) and its corresponding subgraph in Figure 2.2 (b). A graph can have more than one spanning trees. A connected graph with n vertices can have n^{n-2} number of spanning trees. A spanning tree has n-1 edges. The graph shown in Figure 2.2 (a) has 4 vertices, so it can have $4^{4-2} = 16$ number of spanning trees.
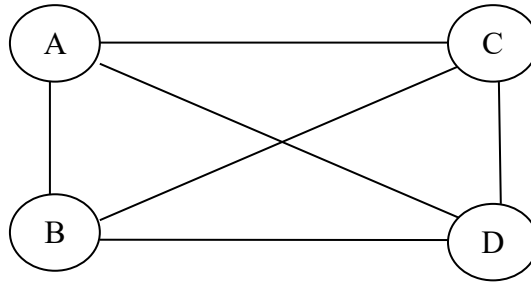
Fig. 2.2 (a): Undirected graph

Fig. 2.2 (b): Spanning tree of the graph

v) Minimum Spanning Tree

.   A minimum spanning tree (MST) is a spanning tree obtained from a weighted graph such that the sum of the weight W of its edges is the least. For a graph $G(V,E)$, the minimum spanning tree T can be represented mathematically as:

$$W(T) = min\left(\sum_{(u,v)\in V} W(u,v)\right)$$

vi) Optimization Problem

Optimization problem can be defined as a problem to get the best solutions from the available potential solutions. The main objective is to mimimize or maximizes some values. For instance to find a shortest route between two vertices of a graph etc.

vii) Forest

In graph theory, forest is collection of trees, in other words it an undirected acyclic graph whose components are connected trees.

viii) Shortest Path

A path followed to reach some destination vertex *v* from the initial vertex *u* of weighted graph *G,* such that the sum of the weights of the edges along the path is minimal. For example, look at the graph shown in Figure 2.3.
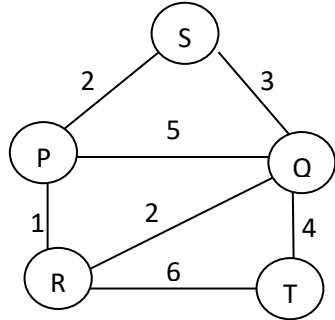
Fig. 2.3: A weighted graph

Here, if you want to find a path between vertex P and Q of the graph, there are multiple paths available with different sum of the weights. The paths are P->Q with total weight 5, path P->S->Q with total weight 5, P->R->Q with total weight 3 and path P->R->T->Q with total weight 11. Although, all these paths takes you from vertex P to Q, but the path with minimum weight is P->R->Q. Thus, it is taken as the shortest path between the two vertices.

The problem of finding the shortest path between any two vertices of a graph is known as shortest path problem or single-pair shortest path problem. It can be defined in both directed and undirected graph. The problem has variations as:

- Single-source shortest path problem.
- Single-destination shortest path problem.
- All-pairs shorted path problem.

ix) Greedy Technique

Greedy technique in an algorithmic approach where any decision is taken by considering the information available currently without thinking what would be the impact of the current decision in future. A greedy method may provide locally optimal solutions but man not be globally optimized. For example, solution to change making problem may be locally optimal solution only.

CHECK YOUR PROGRESS

1. The problem of minimum spanning or MST was first formulated in the year 1926 by _____.

2. The MST problem is an _____ problem in graph whose solutions uses the _____ approach.

3. A tree is a data structure consisting of nodes and edges connecting the nodes in a _____ manner

4. A connected graph with n vertices can have _____ number of spanning trees.

5. State whether the following statements are true or false:

   a. A spanning tree with n vertices can have n-1 edges.

   b. The problem of finding the shortest path between all the vertices of a graph is known as single-pair shortest path problem.

   c. A forest is a union of disjoint collection of trees.

   d. A graph may have more than one subgraphs.

   e. A weighted graph can be directed or undirected.

6. Define optimization problem.

7. What is globally optimal solution?

8. What is locally optimal solution?

9. Define a spanning tree.

10. Construct at least two spanning trees of the graph shown below:
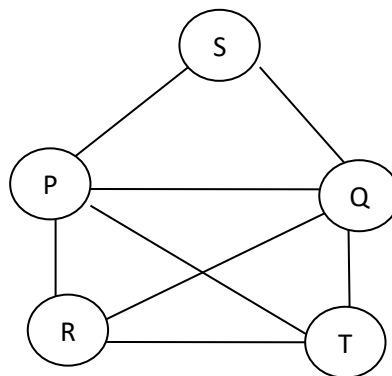
2.4   MINIMUM SPANNING TREE

A minimum spanning tree (MST) is a spanning tree obtained from a weighted graph such that the sum of the weight W of its edges is the least. The cost of an MST is calculated by summing up the weights of its edges. There can be more than one MST of a graph. For a graph $G(V,E)$, the minimum spanning tree T can be represented mathematically as:

$$W(T) = min\left(\sum_{(u,v) \in V} W(u,v)\right)$$

Here, $W(T)$ is the total weight of cost of the MST, i.e the sum of the weights of all the edges in the MST and $W(u, v)$ is the weighted associated to the edge connecting vertex u and v.

The properties of a minimum spanning tree are as follows:

1. A minimum spanning tree must have minimal cost.
2. A minimum spanning tree should not have any cycle.
3. All the vertices of the graph are present in an MST connected by the edges.

To find an MST is an optimization problem. A greedy approach is applied to find an MST. Prim's and Kruskal's algorithms are the popular algorithms to find MST. However, there are other algorithms which use greedy approach to solve some other problems such as Graph coloring problem, Job scheduling problem, Knapsack problem and so on.

Besides the use of MST in computer science, there are plenty of applications of minimum spanning tree in real life.
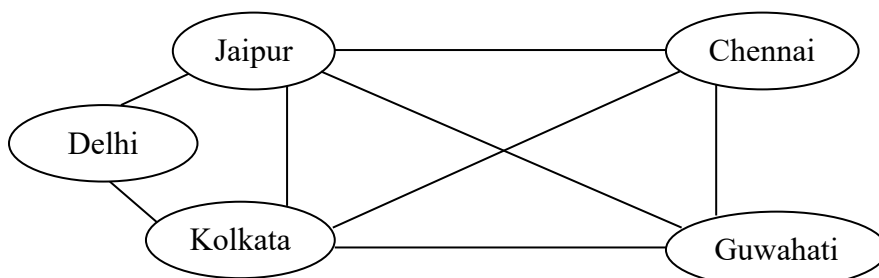
Fig. 2.4: Flight routes graph

For example, say a domestic airline company has several flights in five different cities of a country. Now if the company want to finds an optimal network connecting all the airports in those cities with the shortest distance travelled so that less possible routes need to be travelled covering all the destinations, a minimum spanning tree can be a solution to such a problem. The problem can be modeled into a graph as shown in Figure 2.4 where we have 5 vertices representing cities connected to each other by edges (flight routes). As you see, there are 5 vertices in the graph, so you can have $5^{5-2} = 125$ spanning trees. In this manner, once the number of cities grows, the number of potential spanning trees will also grow. So, the time required finding the spanning tree whose cost is minimum or MST will rise. Thus, practically, applying brute force technique to find the MST will not be feasible; in fact we need some better algorithms to construct MSTs.

Similar problems like designing a circuit board by connecting the transistors with minimum number of wires, designing networks (like cable network, water supply networks, computer networks etc), and so on can be modeled in to graph and an optimal MST can be designed to solve such problems.

_____

2.5   KRUSKAL'S ALGORITHM
_____

Kruskal's algorithm is used to find an MST using greedy approach. For a given weighted graph $G(V, E)$, the algorithm first sorts the edges in the increasing order of its weights and then adds the sorted edges one by one only if it does not form any cycle. The algorithm keeps adding the edges until all the vertices are connected and creates an acyclic graph. It is to be noted that the edge which forms a cycle is discarded.

To keep track of a cycle, the Kruskal's algorithm uses the disjoint-set data structure. The disjoint-set data structure stores the partitions of a set into non-overlapping subsets. The intersection of these subsets results in an empty set, thus they are non-overlapping subsets or in other words we can say that they have no elements in common. The find operation of the disjoint-set helps to identify whether an edge connecting two different trees in a forest forms any cycle. Thus, edges which connect only the disconnected components are considered in developing an MST.

The steps of Kruskal's algorithm are given as follows:

Algorithm Steps

1. Sort the edges of the graph based on its increasing order of weights.
2. Take a smallest edge from the sorted list and check that it does not forms any cycle then add it into the spanning tree else discard it.
3. Repeat Step 2 as long as all the vertices are connected.
4. Calculate the cost of the spanning tree by summing up all the weights of the edges to get an MST.

Let us try to understand the Kruskal's algorithm with an illustration given as under:

Example: Consider the undirected weighted graph shown in Figure 2.5. Now, apply the Kruskal's algorithm to construct a minimum spanning tree of the given graph.
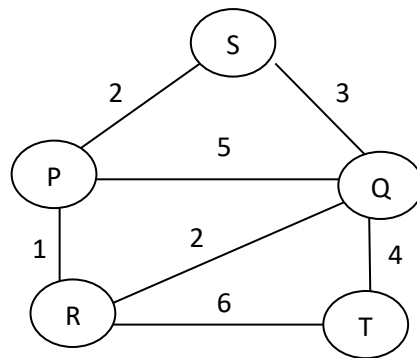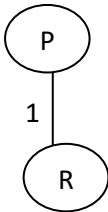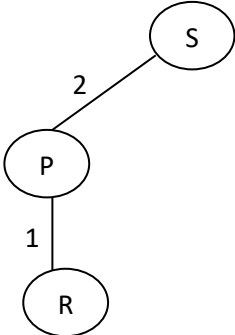

Fig. 2.5: A weighted graph
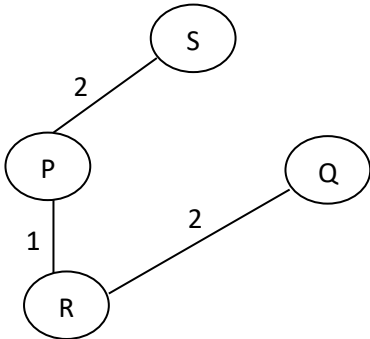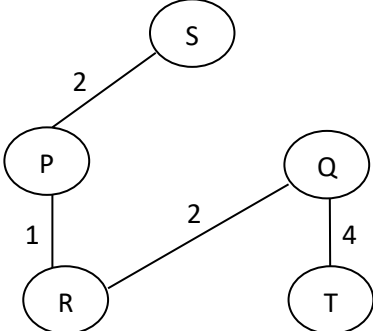
To construct an MST using Kruskal's algorithm, the first step is to sort the edges of the graph based on its weights. The sorted edges are shown in table 2.1.

Table 2.1: Sorted order of edges and their status
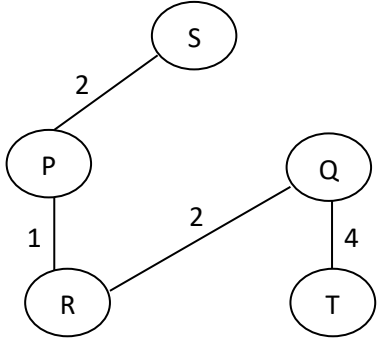
Edge	Cost (Weight)	Status
P – R	1	Accepted
P – S	2	Accepted
R – Q	2	Accepted
S – Q	3	Discarded
Q – T	4	Accepted
P – Q	5	Discarded
R – T	6	Discarded

Space for learners notes	**Step for Constructing MST**	**Description of the Step**
		Take the first edge (P – R) from the sorted list. As it does not form any cycle, so take it to construct an MST.
		Take the next edge (P – S) from the sorted list. As it does not form any cycle, so add it into the spanning tree.
		Take the next edge (R – Q) from the sorted list. As it does not form any cycle, so add it into the spanning tree.
		Take the next edge (S – Q) from the sorted list. Now if you observe it forms a cycle so discard it. Take the next edge (Q – T) from the sorted list, as it does not forms any cycle, so add it into the spanning tree.

Step for Constructing MST	Description of the Step
	Take the next edge (P – Q) from the sorted list. Now if you observe it forms a cycle so discard it. Similarly, the next edge (R – T) is discarded as it forms a cycle. Now, all the vertices are connected and |V|-1 edges are obtained the algorithm stops here. Now, obtain the cost of the obtained MST.

You may observe that for any given graph G (V, E), the number of edges in an MST is $| V | - 1$, i.e the number of edges will be one less than the total number of vertices in the MST of any given graph. So, from the above illustration, at the end we obtain an MST from the resulting spanning tree. The cost of the MST obtained in the above illustration is $1 + 2 + 2 + 4 = 9$.

Time complexity analysis of Kruskal's algorithm

One may also use priority queue data structure to keep track of the cycles. Here, we have used disjoint-set data structure in the algorithm. Given a graph G (V, E), the algorithm first sorts the edges in increasing order of the weights of the edges. So, time required to sort the edges can be expressed as O ($n \log n$), where $n = | E |$. To detect a cycle, a disjoint-set data structure is used where find operation is applied to perform the task. So, the time required determining whether a particular edge is to be taken or not by checking its cycle formation can be at most $2n$. Therefore, the time complexity of Kruskal's algorithm can be expressed as O ($| E | \log | V |$).

SAQ

1. Explain with a suitable example, how a cycle in an undirected graph can be detected in a disjoint-set data structure.

2. "A graph can have several MSTs having same cost". Justify the statement.

3. "Greedy technique provides locally optimal solutions". Justify the statement.

CHECK YOUR PROGRESS

11. Kruskal's algorithm is used to find a minimum spanning tree using _____ approach.
12. The time complexity of Kruskal's algorithm can be expressed as _____,
13. Define disjoint-set data structure.
14. Define a minimum spanning tree.
15. State whether the following statements are true or false:
    a. A minimum spanning tree must have minimal cost.
    b. In Kruskal's algorithm, the edge of the graph which forms a cycle is discarded.
    c. There cannot be more than one MST of a given graph.

_____

2.6   PRIM'S ALGORITHM
_____

Prim's algorithm is another way of constructing a minimum spanning tree which also uses the greedy approach as in the case with Kruskal's algorithm. However, the technique used in Prim's algorithm constructing an MST is different as compared with the Kruskal's algorithm. The difference between the Prim's and Kruskal's algorithm is given in Table 2.2.

The Prim's algorithm finds the spanning tree of a graph whose sum of the weights of the edges is minimum. The basic idea followed here is to start constructing an MST by initially taking any vertex of a graph. Then one by one the vertices of the graph are taken subsequently and added to the spanning tree. Before adding any new vertex to the spanning tree, it checks whether any cycle is formed or not. If it finds that a cycle is formed then it discards the edge in the same way as we have seen in the Kruskal's algorithm. It means

Table 2.2: Differences between Kruskal's and Prim's algorithm

S.No	Kruskal's Algorithm	Prim's Algorithm
1	The algorithm initially sorts the edges in the increasing order of their weights and then takes the first edge to start constructing a spanning tree.	The algorithm does not sort the edges of the graph. In fact it randomly takes any vertex to be the root and starts constructing the spanning tree.
2	It traverses a vertex only once to get the minimum cost	It traverses a vertex more than once to get the minimum cost
3	It saves time finding the next edge to be considered as it initially sorts the edges.	It saves memory space, as no sorted list of edges is required.
4	It generates forest at any iteration of the algorithm and then connects the disconnected components (trees).	It generates connected graph at any iteration of the algorithm.
5	Preferred to construct an MST for a sparse graph,	Preferred to construct an MST for a dense graph,
6	Time complexity of Kruskal's algorithm is $O\left(\lvert E \rvert \log \lvert V \rvert\right)$.	Time complexity of Prim's algorithm is $O\left(\lvert V \rvert^2\right)$, if adjacency list is used.

For a given undirected weighted graph $G\ (V,\ E)$, the Prim's algorithm arbitrarily takes any vertex v to be considered as a root and starts constructing the MST. It then explores the vertex v to find the adjacent vertices connected by an edge with minimum weight. It adds the new vertex into the minimum spanning tree by checking the presence of any cycle.

If it finds any cycle, then it discards the edge. In this way, one after other vertices are added subsequently with their connecting edges into the minimum spanning tree. The process of adding new vertices stops when all vertices are added to the minimum spanning tree.
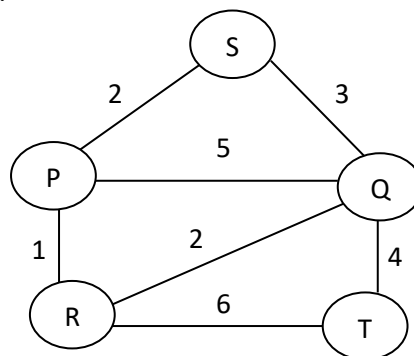
The steps in Prim's algorithm are as follows:

Algorithm Steps

1. Choose a vertex *v* as a source vertex for constructing a spanning tree.

2. Explore the vertex, to find the adjacent vertices.

3. Select the adjacent vertex connected by the edge with least weight and add it to the spanning tree by checking if the edge connecting with the adjacent vertex does not forms a cycle, otherwise discard it.

4. Consider the spanning tree as a node and explore all its adjacent vertices to construct the spanning tree.

5. Repeat steps 3 and 4 till there are *n* - 1 edges in the spanning tree, where $n = |V|$.
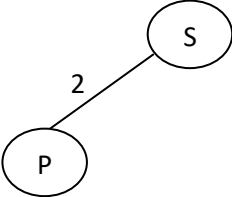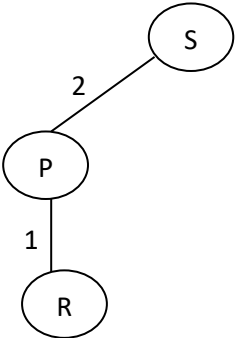
Let us try to understand the Prim's algorithm with an illustration given as under:

Example: Consider the undirected weighted graph shown in Figure 2.6. Now, apply the Prim's algorithm to construct a minimum spanning tree of the given graph.

Fig. 2.6: A weighted graph

To construct an MST using Prim's algorithm, the first step is to select a source vertex, here we have selected vertex S as the source vertex. The following steps to construct the MST with description are as follows:

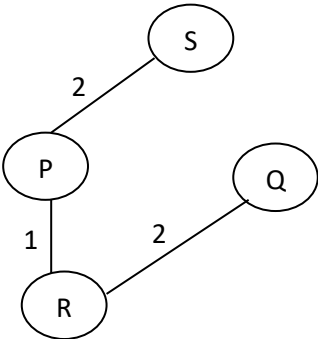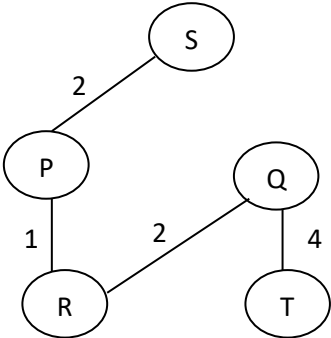| Step for Constructing MST	Description of the Step
	A vertex S has been chosen to be the source vertex, i.e. the vertex to start with constructing an MST.
	The vertex S is explored and two adjacent vertices are found: P and Q connected by the edges S – P and S – Q respectively. The weight of edge S – P is 2 and S – Q is 3. So, applying the greedy approach, where edge with minimum weight is selected, thus, edge S – P will be considered for adding to the spanning tree and it does not form any cycle. So, adjacent vertex P gets connected to the spanning tree.
	The spanning tree is considered as a node and all its adjacent vertices are explored via their connecting edges. The adjacent vertices are Q and R. The weight of the connecting edges S – Q is 3, P – R is 1 and P – Q is 5. Now, while comparing, the edge P – R has the least weight and does not form any cycle, thus it will be considered and vertex R is added to the spanning tree.

Step for Constructing MST	Description of the Step
	The spanning tree is considered as a node and all its adjacent vertices are explored via their connecting edges. The adjacent vertices are Q and T. The weight of the connecting edges R – Q is 2, P – Q is 5, S – Q is 3 and R – T is 6. Comparing all these edges, the edge R – Q has the least weight and does not form any cycle, thus it will be considered and vertex Q is added to the spanning tree.
	The spanning tree is considered as a node and all its adjacent vertices are explored via their connecting edges. The adjacent vertex is T. The weight of the connecting edges Q – T is 4 and R – T is 6. Comparing all these edges, the edge Q – T has the least weight and does not form any cycle, thus it will be considered and vertex T is added to the spanning tree. The algorithm stops here, as we have got	V	- 1 edges in the constructed minimum spanning tree.

You may observe that in Prim's algorithm too, for any given graph G (V, E), the number of edges in an MST is | V | - 1, i.e. the number of edges will be one less than the total number of vertices in the MST of any given graph. So, from the above illustration, at the end we obtain an MST from the resulting spanning tree. The cost of the MST obtained in the above illustration is $1 + 2 + 2 + 4 = 9$.

The cost of the MST for a given weighted graph will be same irrespective of the algorithm applied (Prim's or Kruskal's algorithm). Both the algorithms applies the greedy approach by considering the available edge having least weight which does not forms any cycle. However, their approach to construct an MST is different. The Prim's algorithm constructs an MST by growing a single tree whereas Kruskal's algorithm constructs an MST by growing forest of trees. The basic differences between Kruskal's and Prim's algorithms are already shown in Table 2.2.

Time complexity analysis of Prim's algorithm

The Prim's algorithm can be applied using a priority queue data structure. Given a graph G (V, E), the algorithm at first arbitrarily selects a source vertex v. It grows the spanning tree to construct an MST by adding vertex to the tree one by one. Time required to perform the insertion of vertices not yet included in the MST (or not visited vertices) in the priority queue can be expressed as O ($|V|$ log $|V|$). Other insertion and deletion operations in Prim's algorithm will take O ($|E|$ log $|E|$) time. Therefore, the time complexity of Prim's algorithm can be expressed as O ($|V|$ log $|V|$ + $|E|$ log $|E|$) = O ($|E|$ log $|V|$).

Thus, the time complexity to construct an MST is same using Prim's algorithm and the Kruskal's algorithm. You may observe that the value of $|E|$ = O ($|V|^2$) in case of dense graph and $|E|$ = O ($|V|$) in case of sparse graph, so we can consider O (log $|V|$) and O (log $|E|$) as same. Therefore, the total time complexity of Prim's and Kruskal's algorithm to construct an MST can be expressed as O ($|E|$ log $|V|$) or O ($|E|$ log $|E|$).

STOP TO CONSIDER

Prim's and Kruskal's algorithm are the two algorithms to find a minimum spanning tree for a given graph. Both the can be applied into a weighted graph which is undirected. However, Prim's algorithm is preferred if the graph is graph and Kruskal's algorithm is preferred while constructing an MST for a dense graph.

CHECK YOUR PROGRESS

16. Prim's algorithm is used to find a minimum spanning tree using _____ approach.

17. The time complexity of Prim's algorithm can be expressed as _____.

18. Define a priority queue.

19. State whether the following statements are true or false:

    a. An MST may not have minimum cost always.

    b. In Prim's algorithm, the edge of the graph which forms a cycle is discarded.

    c. A Prim's algorithm may produce more than one MST of a given graph with different costs.

    d. Like Kruskal's, Prim's algorithm also sorts the edges.

SAQ

1. Can you find an MST in a weighted graph which is directed? Justify your answer.

2. State the differences between Prim's and Kruskal's algorithm.

3. Analyse the time complexity of Prim's algorithm if Fibonacci heap is used.

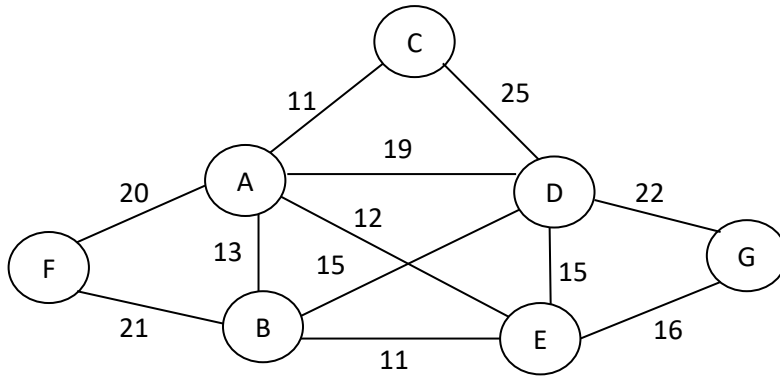_____

2.7   PRACTICE PROBLEMS
_____

Exercise 1. Construct a minimum spanning tree for the given weighted graph (Fig, 2.6) using Kruskal's Algorithm.

Fig. 2.6: A weighted graph

Solution:

Sorted list of edges:

Edge	Weight
A – C	11
B – E	11
A – E	12
A – B	13
B – D	15
D – E	15
E – G	16
A – D	19
A – F	20
B – F	21
D – G	22
C – D	25
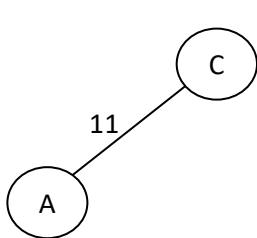
Step 1
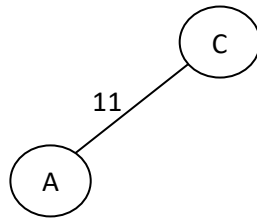
Step 2

Step 3
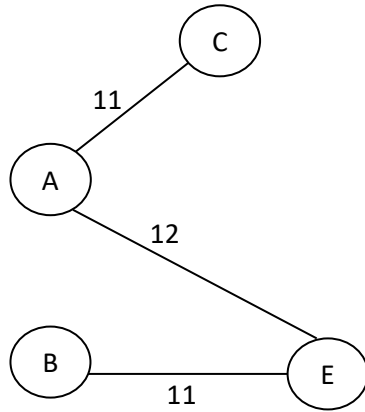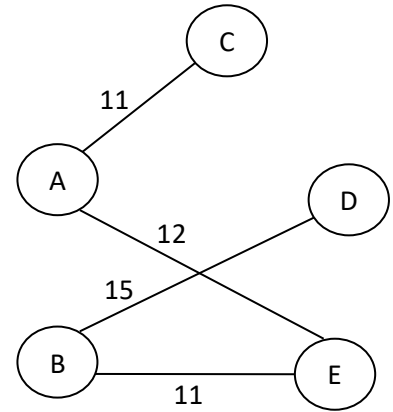
Step 4

Step 5

Step 6

Cost of the MST is 11 + 11 + 12 + 15 + 16 + 20 = 85

Exercise 1. Construct a minimum spanning tree for the given weighted graph (Fig, 2.7) using Prim's Algorithm

Fig. 2.7: A weighted graph

Solution:

Step 1
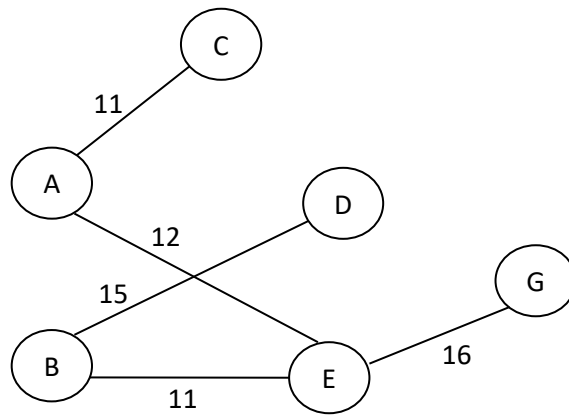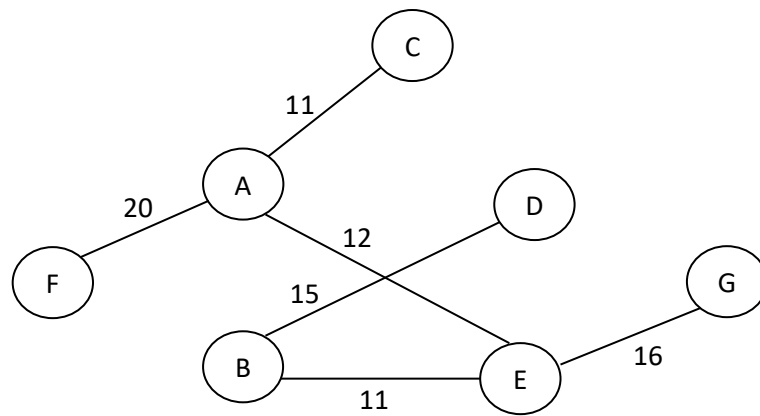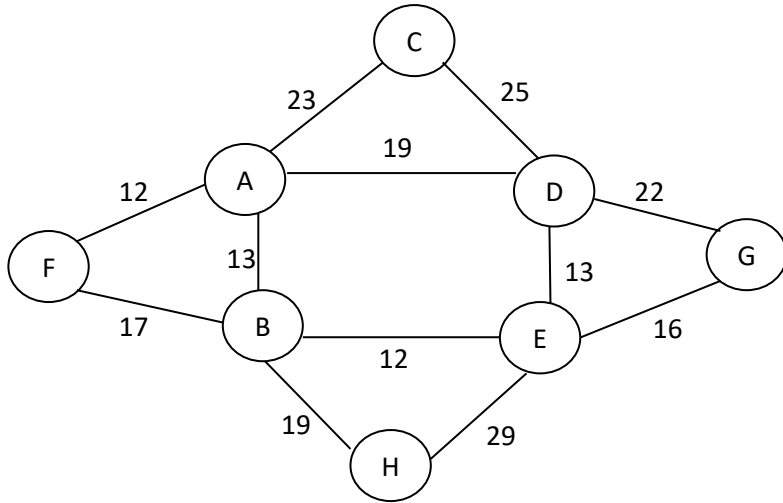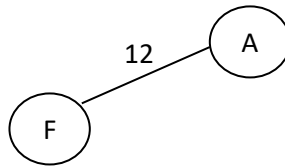
Step 2
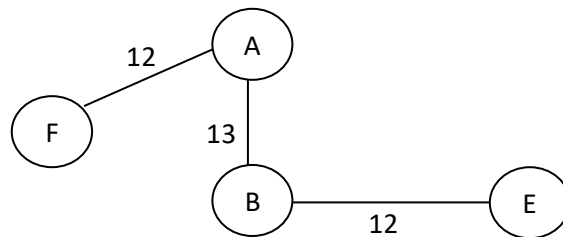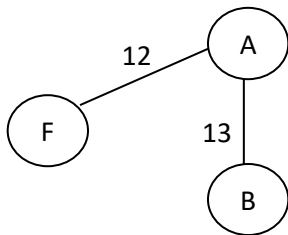
Step 3

Step 4
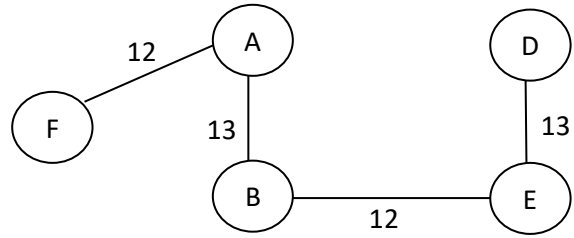
Step 5
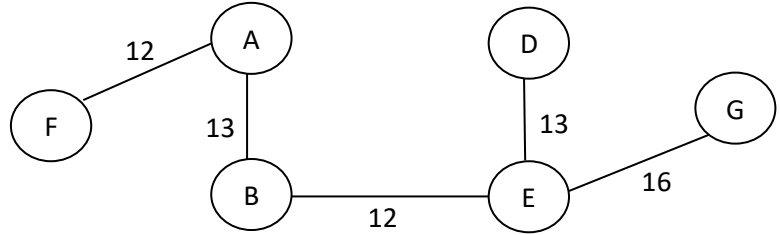
Step 6
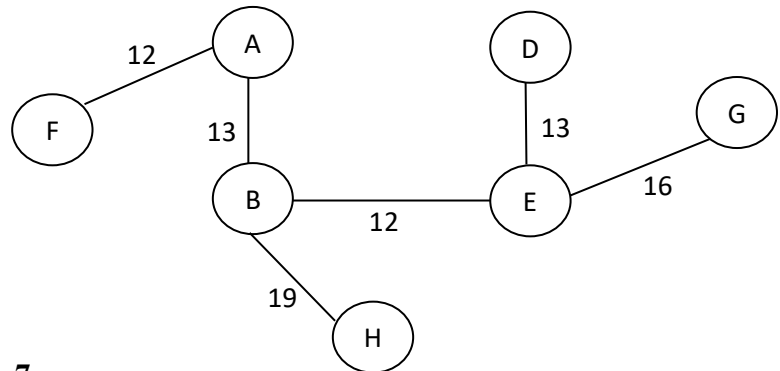
Step 7
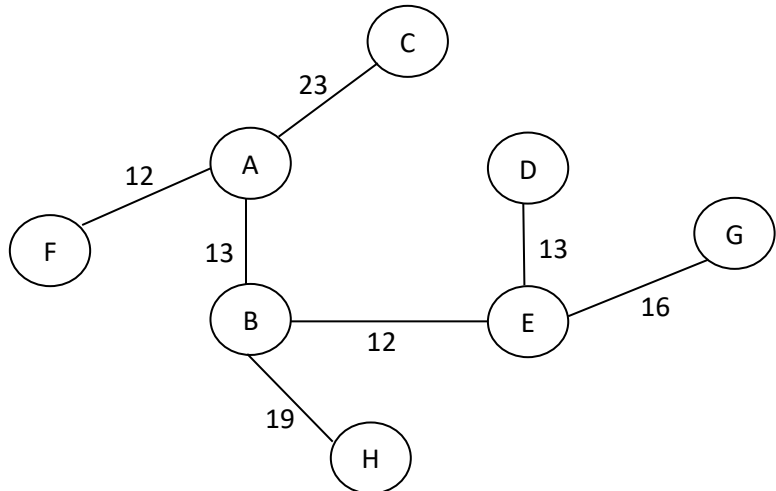
Step 8

The cost of the minimum spanning tree is 12 + 12 +13 + 13 + 16 + 19 +.23 = 108.

_____

2.8   SUMMING UP
_____

- A spanning tree is a subgraph of a graph G(*V,E*) and minimum spanning tree is a spanning tree satisfying certain properties.

- A graph can have more than one spanning trees. A connected graph with *n* vertices can have n^{n-2} number of spanning trees. A spanning tree has *n*-1 edges.

- The properties of a minimum spanning tree are as follows:

  1. A minimum spanning tree must have minimal cost.
  2. A minimum spanning tree should not have any cycle.
  3. All the vertices of the graph are present in an MST connected by the edges.

- A minimum spanning tree (MST) is a spanning tree obtained from a weighted graph such that the sum of the weight *W* of its edges is the least.

- The MST problem is an optimization problem in graph whose solutions uses the greedy approach.

- Kruskal's algorithm is used to find an MST using greedy approach. For a given weighted graph G(*V, E*), the algorithm first sorts the edges in the increasing order of its weights and then construct an MST by adding the sorted edges one by one only if it does not form any cycle.

- For a given graph G (*V, E*), the number of edges in an MST is | *V* | - 1, i.e. the number of edges will be one less than the total number of vertices in the MST of any given graph.

- For a given undirected weighted graph G (*V, E*), the Prim's algorithm arbitrarily takes any vertex *v* to be considered as a root and starts constructing the MST.

- The Prim's algorithm constructs an MST by growing a single tree whereas Kruskal's algorithm constructs an MST by growing forest of trees.

- The cost of the MST for a given weighted graph will be same irrespective of whether you apply Prim's or Kruskal's algorithm. Both the algorithms applies the greedy approach by considering the available edge having least weight which does not forms any cycle.

Subgraph: A subgraph $G(V', E')$ for a given graph $G(V, E)$ is the subset of the graph where its edges $E' \subseteq E$ and vertices $V' \subseteq V$. One may obtain a subgraph by deleting vertices and edges of a graph.

Spanning Tree: A spanning tree is a subgraph $G(V', E')$ for a given graph $G(V, E)$ that contains all the vertices of the graph connected by the edges without forming any cycle.

Minimum Spanning Tree: A minimum spanning tree (MST) is a spanning tree obtained from a weighted graph such that the sum of the weight W of its edges minimum. For a graph $G(V,E)$, the minimum spanning tree T can be represented mathematically as:

$$W(T) = min\left(\sum_{(u,v)\in V} W(u, v)\right)$$

Optimization Problem: An optimization problem is one in which the goal is to find the optimal answer from a set of viable solutions. The main goal is to minimize or maximize certain values. For example, to identify the shortest path between two vertices of a graph, etc.

Forest: A forest is collection of trees, in other words it an undirected acyclic graph whose components are connected trees.

2.9   ANSWERS TO CHECK YOUR PROGRESS

1. Otaker Boruvka

2. optimization, greedy

3. hierarchical

4. n^{n-2}

5.
    a. True.
    b. False
    c. True
    d. True
    e. True

6. An optimization problem is one in which the goal is to find the optimal answer from a set of viable solutions. The main goal is to minimize or maximize certain values.

7. A global optimal solution of an optimization problem is a solution that is most favorable or optimal among all the possible solutions.

8. A local optimal solution of an optimization problem is a solution that is most favorable or optimal in the vicinity but possibly not favorable globally.

9. A spanning tree is a subgraph *G(V', E')* for a given graph *G(V, E)* that contains all the vertices of the graph connected by the edges without forming any cycle.

10. The two spanning trees of the given graph are:
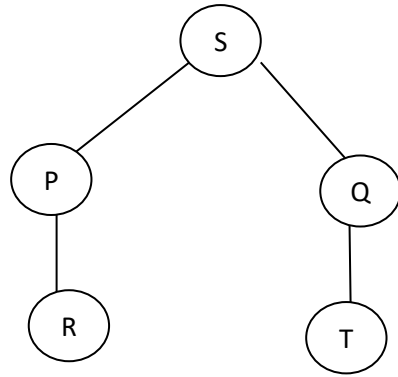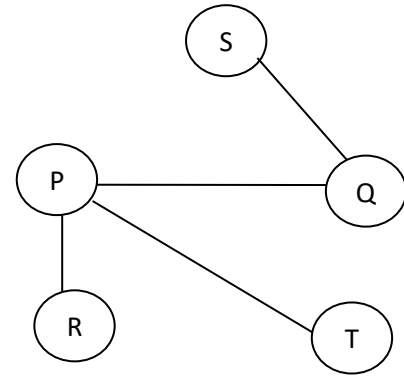
(a) Spanning Tree 1   (b) Spanning Tree 2

11. greedy

12. $O\,(|\,E\,|\log|\,E\,|)$

13. A disjoint-set data structure is a data structure that store components that have nothing in common, in other words a collection of non-overlapping sets whose intersection results in an empty set.

14. A minimum spanning tree (MST) is a spanning tree obtained from a weighted graph such whose sum of the weight of its edges is minimum.

15. :

 a. True.

 b. True

 c. False.

16. Greedy

17. $O\,(|\,E\,|\log|\,V\,|)$

18. A priority queue is a special type of queue data structure where every element has a priority associated with it. The element with high priority is given first preference followed by the element with lower priority.

19.

 a. False.

 b. True.

c. False

d. False

_____

2.10 POSSIBLE QUESTIONS
_____

Short Answer Type Questions

1. Define a tree.

2. What is a spanning tree?

3. Define minimum spanning tree.

4. What is an MST problem?

5. How do you find the cost of a spanning tree?

6. How many spanning tree can a graph have?

7. How many minimum spanning trees can a graph have?

8. What do you mean optimization problem?

9. Give some examples of optimization problem in real life.

10. Define a forest.

11. Define disjoint-set data structure.

12. What is a priority queue?

13. What is the time complexity of Kruskal's algorithm?

14. What is the time complexity of Prim's algorithm?

15. Define local optimality with a suitable example.

16. Define global optimality with a suitable example.

17. What do you mean by feasible solution?

18. What is the characteristic of greedy technique?

19. Define single-source shortest path problem.

Long Answer Type Questions

1. Explain the minimum spanning tree along with its properties with suitable example.

2. Consider the graphs shown in Figure 2.8 (a) (b) and do the following:

    a. Find the MST using Prim's algorithm.

    b. Find the MST using Kruskal's algorithm.
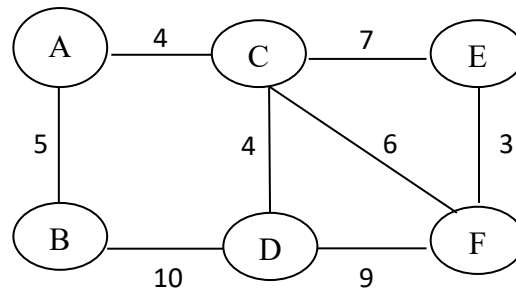
Fig. 2.8 (a): A weighted graph
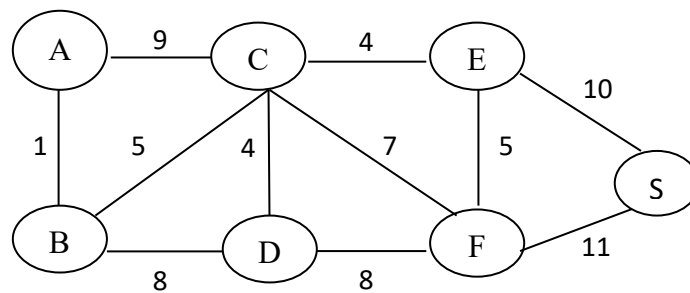
Fig. 2.8 (b): A weighted graph

3. Differentiate between Prim's and Kruskal's algorithm.

4. Explain Prim's algorithm to construct a minimum spanning tree.

5. Perform the time complexity analysis of Prim's algorithm.

6. Explain Prim's algorithm to construct a minimum spanning tree.

7. Perform the time complexity analysis of Kruskal's algorithm.

8. Give some instances of real-life problems where it can be modeled into an MST problem.

9. Explain greedy technique with a suitable example.

10. Draw a complete graph with vertices and find all its spanning trees.

11. Does Kruskal's algorithm always yield an optimal MST of a given undirected weighted graph? Explain.

12. Does Prim's algorithm always yield an optimal MST of a given undirected weighted graph? Explain.

13. Do greedy algorithms always give an optimal solution? Justify your answer.

14. Does the Prim's algorithm can construct an MST of a graph having negative weights? Explain.

_____

2.11   REFERENCES AND SUGGESTED READINGS
_____

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press.

- Sridhar S., Design and Analysis of Algorithms, Oxford University Press, 2014.

UNIT 3   SINGLE SOURCE SHORTEST PATH PROBLEM

Unit Structure:

3.1   INTRODUCTION

In this unit, you will learn the concept of directed acyclic graph, and single-source shortest path problem on a weighted graph. You will learn the concept of Dijkstra's algorithm and how the single-source shortest path problem can be solved using the Dijkstra's algorithm.   The time complexity of the algorithm will be discussed in this unit along with few demonstrations of the technique. You will also learn whether Breadth-first search and Depth-first search algorithms can solve the problem of finding the single-source shortest path on a directed weighted graph.

3.2   UNIT OBJECTIVES

After going through this unit, you will be able to:

- *Know* the directed acyclic graph.
- *Understand* the fundamental concept of single source shortest path problem.
- *Define* basic terms and terminologies associated with single-source shortest path problem.
- *Describe* Dijkstra's algorithm.
- *Analyze* the time complexities of Dijkstra's algorithm.

3.3   BASIC TERMS AND THEIR DEFINITIONS

Single source shortest path problem is a problem which is applicable in real life. For instance, one may always need to find the shortest distance to reach his destination. Suppose a traveler wants to visit a place in a city. He has got a road map, now in order to reach the place from where he is now, he need to plan a road map which will help him to reach his destination by determining the shortest route or the fastest way to reach place B from place A. The same problem can be modeled as finding a shortest path from a source vertex to any other vertex of a graph.

To understand it further, let us first look at some of the basic terms and terminologies associated to single source shortest path problem in the following subsections.

i) Graph

A graph is a non-linear data structure that consists of finite set of nodes *V* and a finite set of edges *E* that connects the vertices. Mathematically, a graph can be represented as *G (V, E)*. The nodes are also called as vertices and edges as arcs.

ii) Directed Weighted Graph

A directed weighted graph as shown in Figure 3.1 is a set of vertices and a set of weighted directed links between the vertices, i.e. the vertices of the graph are connected by the edges and where all the edges has weights associated to it and are directed from one vertex to another.
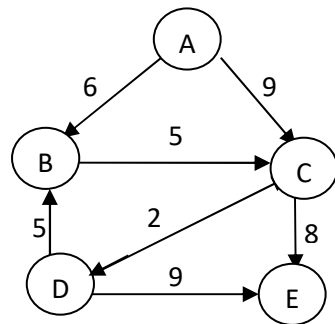
Fig. 3.1: A sample directed weighted graph

iii). Directed Acyclic Graph (DAG)

A directed graph, which does not consist of any cycle, is known as directed acyclic graph (DAG). The graph shown in Figure 3.2 is a sample DAG.
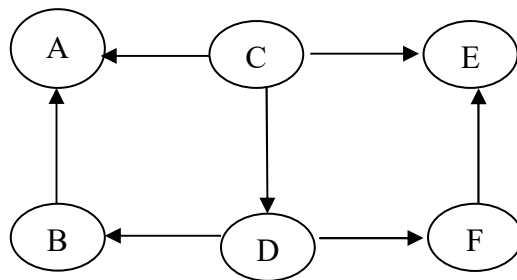
Fig. 3.2: A sample directed acyclic graph

iv). Path

A path can be defined as the sequence of vertices that are followed in order to reach some destination vertex *v* from the initial vertex *u* where, we do not repeat a vertex and nor we repeat an edge while we traverse the graph.

v). Shortest Path

A path between any two vertices in a graph such that the sum of the weights of its constituent edges is minimum.

vi). Single-source Shortest Path

The shortest path between a source vertex of a graph to all other vertices of the graph.

_____

3.4   SINGLE SOURCE SHORTEST PATH
_____

Single source shortest path problem state the problem of finding a path from the source vertex *u* to a target vertex *v* of a weighted graph (directed or undirected) such that the sum of the weights of the edge is minimum over the path. Formally, we can define it as: For a given weighted graph *G* (*V, E, W*), where *V* is the set of vertices, *E* is the set of edges and *W* is the set of real-valued weights. We need to find the shortest path from a given source vertex *u* ∈ *V* to every vertex *v* ∈ *V* where the shortest path is calculated by summing up the weights associated with edges along the path.

There are some variants of the shortest path problem:

1.   **Single pair shortest path problem:** Given a weighted Graph *G* (*V, E, W*), we need to find the shortest path from a given source vertex *u* ∈ *V* to a target vertex *v* ∈ *V*.

2.   **Single destination shortest path problem:** Given a weighted Graph *G* (*V, E, W*), we need to find the shortest path to a given target vertex *v* ∈ *V* from every source vertex *u* ∈ *V*.

3.   **All pairs shortest paths problem:** Given a weighted Graph *G* (*V, E, W*), we need to find the shortest paths for every pair of vertices *u* ∈ *V* to *v* ∈ *V*.

The shortest path problem can be solved using the following different algorithms:

- Breadth-first Search Algorithm
- Dijkstra's Algorithm
- Bellman-Ford Algorithm and
- Floyd-Warshall Algorithm

In the following section, we will discuss about an algorithm which uses the greedy approach to solve the problem. The algorithm is known as Dijkstra's algorithm. It was first proposed by Edgser Wybe Dijkstra. He was a Dutch computer Scientist and his contribution is well known in the field of computer science.

CHECK YOUR PROGRESS

1. A _____ graph is a set of vertices and a set of weighted directed links between the vertices.
2. A directed graph, which does not consist of any cycle, is known as _____.
3. The shortest path between a source vertex of a graph to all other vertices of the graph is known as _____.
4. Dijkstra's algorithm was first proposed by _____.
5. Define all pairs shortest paths problem.
6. Define single pair shortest path problem.
7. What is a shortest path problem?
8. State whether the following statements are true or false:
   a. A shortest path is a path between any two vertices in a graph such that the sum of the weights of its constituent edges is minimum.
   b. A DAG can consist of a cycle.
   c. A graph is a linear data structure.

3.5   DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is a graph search algorithm used to solve the single source shortest path problem. The algorithm is similar to Prim's algorithm however, there are basic differences between Dijkstra's and Prim's algorithm. In Prim's algorithm, the main objective is to find a minimum spanning tree where we find a subgraph of a given graph where the sum of the weights of the edges of the subgraph is minimum. Whereas, in Dijkstra's algorithm we find paths from a source vertex to every other vertex of the graph, where the path length from the source vertex to any other vertex is minimum. So, you can say that Dijkstra's algorithm constructs a shortest path tree and a shortest path tree may not be necessarily a minimum spanning tree. The cost of a shortest path tree may be much larger than the cost of an MST.

The Dijkstra's algorithm also applies greedy approach, as at every step of the algorithm, it tries finding the shortest path between the source and the target vertices of a graph. As said, the main goal of the algorithm is to construct a shortest path tree where shortest paths from a source vertex of a graph to the remaining vertices are determined. The algorithm works with both directed and undirected graph, however the weights associated to the edges have to be non-negative. As Dijkstra's algorithm cannot work with graphs (directed and undirected) having edges of negative weights. To solve the shortest path problem with graphs (directed and undirected) having negative edge weights, there are other algorithms namely Bellman-Ford algorithm and Floyd-Warshall algorithm. Breadth-first search algorithm can also solve the shortest path problem but it assumes the edge weight to be 1.

Given a graph $G(V, E, W)$, the general steps of Dijkstra's algorithm to solve the single source shortest path problem of the graph are as follows:

Step 1: Form a distance table T and allocate infinite distance values to every vertex from a source vertex s except assigning the distance value of 0 to s itself.

Step 2: Mark the vertex s as visited and all the remaining vertices as unvisited.

Step 3: Calculate the distance from the current vertex to all its adjacent vertices which are unvisited. If the calculated distance is found to be minimum than the previously stored distance than update it in the distance table T. i.e. follow the following rule to update the distance:

If $(d(u) + c(u, v) < d(v))$ Then

$d(v) = d(u) + c(u, v)$

Where, u and v are the vertices, $d(u)$ is the distance of vertex u from the source vertex, $d(v)$ is the is the distance of vertex v from the source vertex and $c(u,v)$ is the distance between vertex u and v.

Step 4: Compare the updated distances and select the vertex v with least distance and mark it as visited.

Step 5: Take the vertex v and repeat the steps 3 and 4 until all the vertices are marked as visited.

Example: Let us try to understand with an example. Consider an undirected graph shown in Figure 3.3 having four vertices A, B, C and D connected by the edges bearing positive weights. We need to find the shortest path from a source vertex to all other vertices of the graph. The illustration of the Dijkstra's algorithm is shown below:
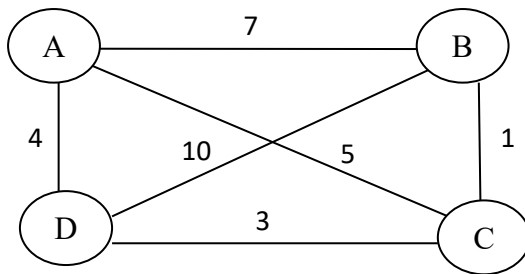
Fig. 3.3: An undirected weighted graph

Iteration 1: Let us take vertex A as source vertex and construct a distance table by assigning infinite distance values to every vertex from the source vertex A except assigning the distance value of 0 to A itself. The distance table and labeling of distance of every vertex from source vertex A in the graph is shown as follows:

	A	B	C	D
A	0	∞	∞	∞
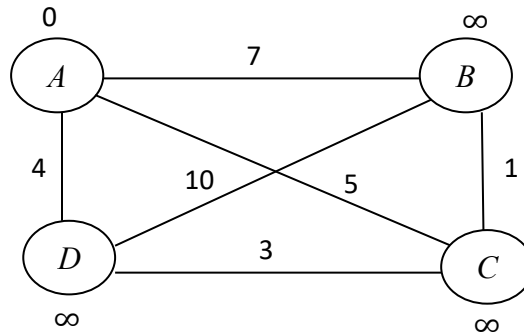
Iteration 2: Now, calculate the distances of the directly connected vertices from A which are unvisited. As the calculated distance of the adjacent vertices B, C and D are found to be lesser than the previously stored distance i.e. ∞ , we update it in the distance table. The distances from A to B, A to C and A to D are updated as 7, 5 and 4 respectively.

	A	B	C	D
A	0	∞	∞	∞
		7	5	4
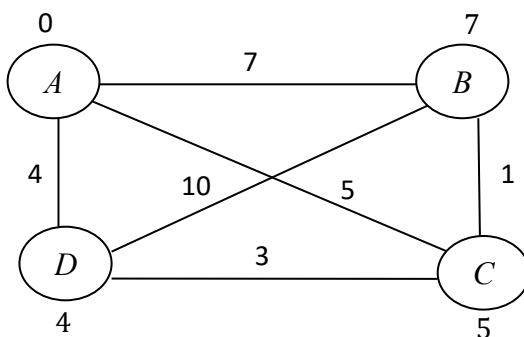
Iteration 3: Based on the current known distances, vertex D is selected as the next vertex to be explored as it is found to have least distance from A. It is marked as visited. After exploring vertex D we find, the path lengths: A – D – C is 4 + 3 = 7, A – D – B is 4 + 10 = 14. The newly calculated distances are found to be greater than previously stored distance. So, no updating of distances is done.

	A	B	C	D
A	0	∞	∞	∞
D		7	5	4
		7	5	

Iteration 4: Now the vertex C is selected as the next vertex to be explored as it is found to be nearer to A. It is marked as visited. After exploring vertex C we find, the path lengths: A – C – B is 5 + 1 = 6 and D has been already visited so it is ignored.

The newly calculated distance of vertex B from A following the path A – C – B is found to be lesser than previously stored distance i.e. 7. So, we update the distance of vertex B from A as 6.

	A	B	C	D
A	0	∞	∞	∞
D		7	5	4
C		7	5	
		6		
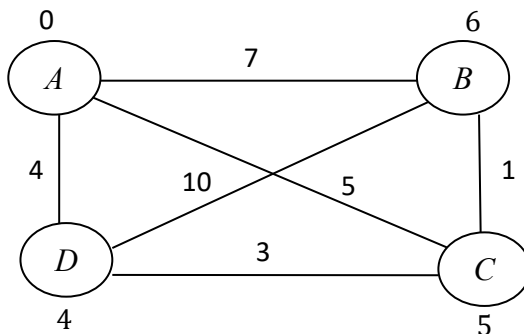
Iteration 5: Now the vertex B is selected as the next vertex to be explored.

It is marked as visited. After exploring vertex B we find, the adjacent vertices of B are already visited, so all are ignored. So, distances are not updated.

	A	B	C	D
A	0	∞	∞	∞
D		7	5	4
C		7	5	
B		6		

With this, we observe that the all the vertices are visited and thus, we have got the shortest paths from the source vertex A to all other vertices of the graph. This solves the shortest path problem of the graph using the Dijkstra's algorithm and the shortest path tree for the given graph is shown in Figure 3.4.
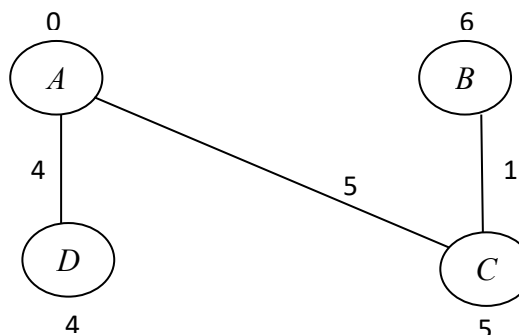
Fig. 3.4: A shortest path tree of the graph shown in Fig. 3.3

The shortest distances and shortest paths from the source vertex A to every other vertices of the given graph are shown below:

Source Vertex	Destination Vertex	Shortest Distance	Shortest Path
	B	6	A – C – B
A	C	5	A – C
	D	4	A – D

Time Complexity Analysis of Dijkstra's Algorithm

The Dijkstra's algorithm can be implemented using an array or linked list data structure. In that case the time complexity of the algorithm can be represented as $O(|V|^2 + |E|) = O(|V|^2)$.

In case of sparse graph, then it becomes efficient to represent the it using adjacency list and use priority queue or binary heap, in that case the running time of Dijkstra's algorithm becomes $O((|V|+|E|) \log V = O(|E| \log V)$.

STOP TO CONSIDER

The Dijkstra's algorithm uses a greedy approach, in which it tries to discover the shortest path between the source and target vertices of a graph at each step. The algorithm's main purpose is to construct a shortest path tree in which shortest pathways from a graph's source vertex to the remaining vertices are determined. The approach can be used with both directed and undirected graphs, but the edge weights must be non-negative.

CHECK YOUR PROGRESS

9. Dijkstra's algorithm is a graph search algorithm used to solve the _____ problem.

10. Sate at least one difference between Dijkstra's and Prim's algorithms.

11. State whether the following statements are true or false:

   a. Dijkstra's algorithm cannot work with graphs having negative weight edges.

   b. Bellman-Ford algorithm can solve the shortest path problem with graphs (directed and undirected) having edges of negative weights.

   c. Prim's and Dijkstra's algorithm both applies greedy approach.

SAQ

1. Why Dijkstra's algorithm cannot work on a graph having negative weight edges? Explain

2. How Dijkstra's algorithm is different from Breadth-first search algorithm in solving the single source shortest path problem?

3. Analyse the time complexity of Dijkstra's algorithm.

_____

3.6   PRACTICE PROBLEMS
_____

Exercise 1. Consider the graph shown in Figure 3.5 and construct a shortest path tree using A as the source vertex using Dijkstra's algorithm.
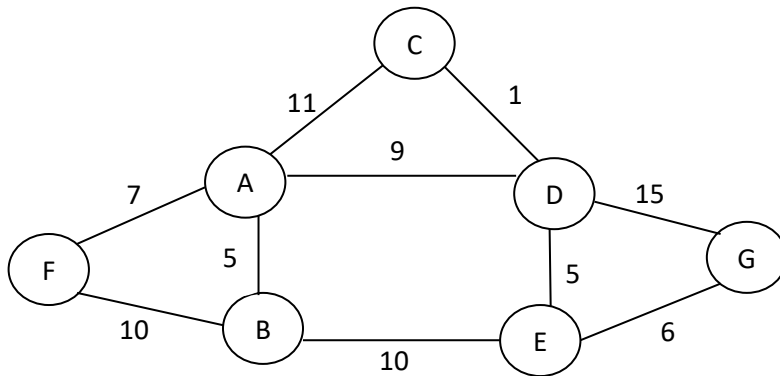
Fig. 3.5: An undirected weighted graph

Solution:

	A	B	C	D	E	F	G
A	0	∞	∞	∞	∞	∞	∞
B		5	11	9	∞	7	∞
F			11	9	15	7	∞
D			11	9	15		∞
C			10		14		24
E					14		24
G							20
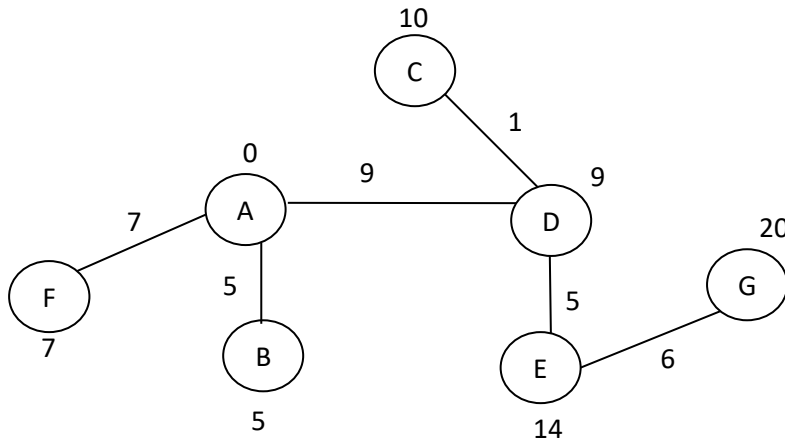
Exercise 2. Find the shortest paths from the vertex A to all other vertices of the graph as shown in Fig. 3.6 using Dijkstra's algorithm.
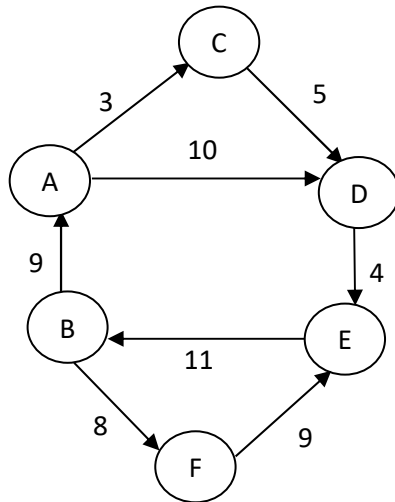
Fig. 3.6: A directed weighted graph

Solution:

	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
C		∞	3	10	∞	∞
D		∞		8	∞	∞
E		∞			12	∞
B		23				∞
F						31
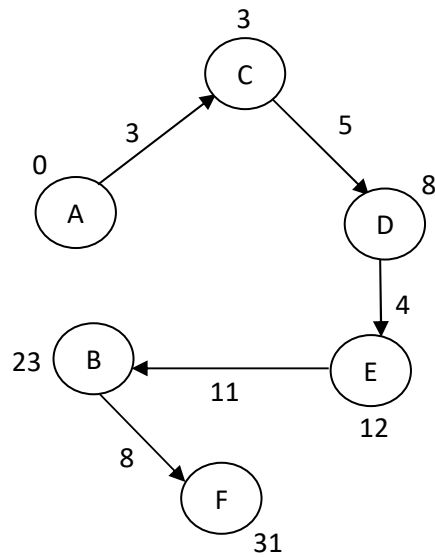
3.7   SUMMING UP

- A directed graph, which does not consist of any cycle, is known as directed acyclic graph (DAG).

- Single source shortest path problem state the problem of finding a path from the source vertex *u* to a target vertex *v* of a weighted graph (directed or undirected) such that the sum of the weights of the edge is minimum over the path.

- Dijkstra's algorithm is a graph search algorithm used to solve the single source shortest path problem.

- Dijkstra's algorithm cannot work with graphs (directed and undirected) having edges of negative weights.

- To solve the shortest path problem with graphs (directed and undirected) having negative edge weights, there are other algorithms namely Bellman-Ford algorithm and Floyd-Warshall algorithm.

Path: A path can be defined as the sequence of vertices that are followed in order to reach some destination vertex v from the initial vertex u where, we do not repeat a vertex and nor we repeat an edge while we traverse the graph.

Shortest Path: A path between any two vertices in a graph such that the sum of the weights of its constituent edges is minimum.

Single-source Shortest Path: The shortest path between a source vertex of a graph to all other vertices of the graph.

All pairs shortest paths problem: Given a weighted Graph G (V, E, W), we need to find the shortest paths for every pair of vertices $u \in V$ to $v \in V$.

_____

3.8   ANSWERS TO CHECK YOUR PROGRESS
_____

1. directed weighted.
2. directed acyclic graph.
3. single-source shortest path.
4. Edgser Wybe Dijkstra.
5. The problem can be defined as given a weighted Graph G (V, E, W), we need to find the shortest paths for every pair of vertices $u \in V$ to $v \in V$.
6. The problem can be defined as given a weighted Graph G (V, E, W), we need to find the shortest path from a given source vertex $u \in V$ to a target vertex $v \in V$.
7. The problem to find a path between any two vertices in a graph such that the sum of the weights of its constituent edges is minimum.
8.   a. True.

   b. False

12. single source shortest path .

13. Dijkstra's algorithm finds a shortest path tree of a graph whereas Prim's algorithm constructs a minimum spanning tree of a graph.

14.
   a. True.
   b. True.
   c. True.

_____

3.9   POSSIBLE QUESTIONS
_____

Short Answer Type Questions

1. Define a path.

2. What is single source shortest path problem?

3. What is Dijkstra's algorithm?

4. What data structure is used to implement Dijkstra's algorithm?

5. What is the time complexity of Dijkstra's algorithm?

6. In what kind of graph we cannot apply Dijkstra's algorithm?

7. What is all pairs source shortest path problem?

8. State two applications of Dijkstra's algorithm.

Long Answer Type Questions

1. Explain the shortest path problem and its variants.

2. Explain the Dijkstra's algorithm.

3. Differentiate between Dijkstra's algorithm and Prim's Algorithm.

4. "Dijkstra's algorithm applies the Greedy approach". Explain the statement with a suitable example.

5. Perform the time complexity analysis of Dijkstra's algorithm.

6. Consider the graph shown in Figure 3.7 to construct a shortest path tree using Dijkstra's algorithm.
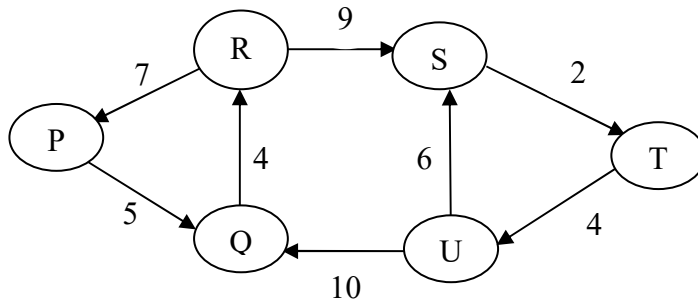
Fig. 3.7: A directed weighted graph

7. Find the shortest paths from vertex A to all other vertices of the graph shown in Figure 3.8 using Dijkstra's algorithm.
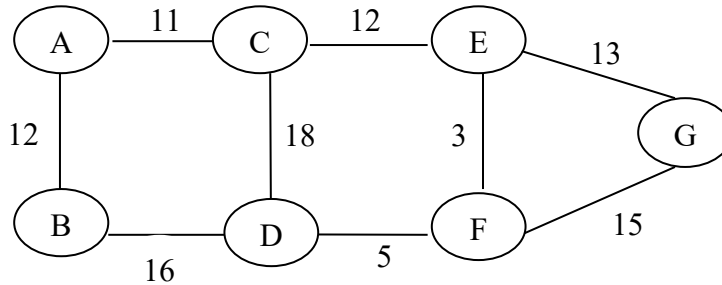
Fig. 3.8: An undirected weighted graph

8. Explain with an appropriate example, where Dijkstra's algorithm fails.

3.10 REFERENCES AND SUGGESTED READINGS

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press.
- Sridhar S., Design and Analysis of Algorithms, Oxford University Press, 2014.

BLOCK IV:
THEORY OF NP COMPLETENESS AND LOWER BOUND THEORY

UNIT 1: THEORY OF NP COMPLETENESS I

Unit Structure:

1.1 INTRODUCTION

In computer science Theory of Formal Languages has a number of applications. In early 1950's Linguists were trying hard to bring some mathematical way to describe formal languages. In 1956 Noam Chomsky an American linguist, philosopher, cognitive scientist, historian, social critic, and political activist, who is Sometimes called "The Father Of Modern Linguistics", gave a mathematical model of grammar which comes out useful for computer languages.

1.2 UNIT OBJECTIVES

To ascertain the amount of computational resources required to solve important computational problems, and to classify problems according to their difficulty is the main purpose of complexity theory. The most often discussed resource is computational time. There are certain problems that cannot be solved without expending large amounts of resources. It is much more difficult to prove that any interesting problems are hard to solve then to prove that inherently difficult problems exist. The mathematical arguments of intractability rely on the notions of Completeness and reducibility. Before understanding reducibility and completeness, one must know the notion of a complexity class. But before going to complexity first we have to know about Formal Languages.

1.3 FORMAL LANGUAGE

According to Chomsky's classification of Languages, languages can be divided into four (4) types. They are namely Unrestricted Language, Context-sensitive Language, Context-free Language and Regular Language.

Type 0 : Type 0 grammar is a phase structure grammar without any restriction. All grammars are Type 0 grammar.
For Type 0 grammar, the production rules are in the format

$\{(Lc)(NT)(Rc)\} \rightarrow$ {String of Ts or NTs or both}
L_c : Left context; R_c : Right context; NT: Non-terminal.

Type 1: Type 1 grammar is called context-sensitive grammar.

For Type 1 grammar, all production rules are in the format of context sensitive if all rules in P are of the form $\alpha A\beta \to \alpha\gamma\beta$, where $A \in NT$ (i.e. A is a single NT), α, $\beta \in (NT \cup \Sigma)*$ (i.e. α and β are strings of NTs and Ts) and γ e $(NT \cup \Sigma)^+$ (i.e. γ is a non-empty string of NTs and Ts).

Type 2: Type 2 grammar is called context-free grammar. In the left-hand side of the production, there will no left or right context.
For Type 2 grammar, all the production rules are in the format of $(NT) \to \alpha$, where $|NT| = 1$ and $\alpha \in (NT \cup T) *$, NT is non-terminal and T is terminal.

Type 3: Type 3 grammar is called regular grammar. Here all the productions will be in the following forms: $A \to \alpha$ or $A \to \alpha B$, where $A, B \in NT$ and $\alpha \in T$.

The Chomsky classification is called the **Chomsky Hierarchy**. This can be represented diagrammatically
From this diagrammatical representation, we can say that all regular grammars are context-free grammar. All context-free grammars are context-sensitive grammar. All context-sensitive grammar are unrestricted grammar.

STOP TO CONSIDER

NTs i.e Non-Terminals are those which can be replaced either by Non-Terminals or by terminals or combination of both. And denoted by A, B , C......,Z i.e. English Alphabets in uppercase.
T i.e Terminal is that which cannot be replaced and denoted by a, b, c, ...,z and 0,1 i.e e English Alphabets in lowercase and binary digits.

Type 3 or Regular

Type 2 or Context-free

Type 1 or Context-sensitive

Type 0 or Unrestricted Language
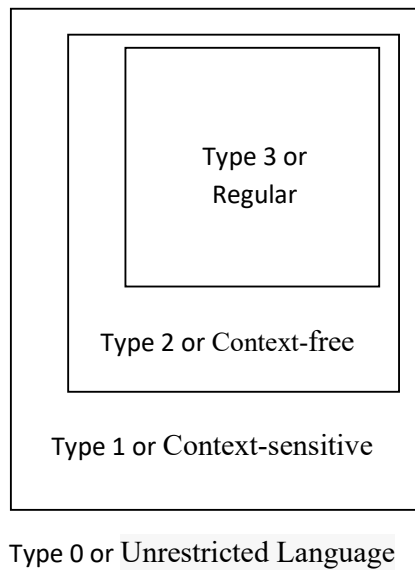
Fig 1.1: Chomsky's Classification of Languages

Grammar	Languages	Machine Format
Type 0	Unrestricted	Turing Machine(TM)
Type 1	Context-sensitive	Linear Bound Automata (LBA)
Type 2	Context-free	Push Down Automata(PDA)
Type 3	Regular	Finite Automata(FA)

Fig 1.2 : Languages and Machine Format of different Grammar

Using Type 3 Grammar, Regular language can be formed which are precisely accepted by Finite Automata. Again from Type 2 Grammar, Context Free languages can be formed which are accepted by Push down Automata(PDA) and from Type 1 Grammar, Context-sensitive languages can be formed which are accepted by Linear Bound Automata(LBA) and lastly from Type 0 Grammar, unrestricted languages can be formed which are accepted by Turing Machine(TM). From the figure 1.1 we can say that all languages are of unrestricted type.

STOP TO CONSIDER
Is there any difference between Formal language and the languages we know or we speak?
The answer is NO and YES. Formal Languages works as the languages we used to communicate. Our languages have certain sets of rules which are called grammar, and it is same in the case of Formal language. Formal Languages are used in Computer,

1.4 TURING MACHINE

In early stage of 1930's mathematicians were trying to give different effective methods of computation. Different models using the concept of Turing machines, combinatory logic ,JL-calculus, post-systems and p-recursive functions, were given by a mathematician ,Alan Turing in 1936, , S.C. Kleene in 1935, Alanzo Church in 1933, Schonfinkel

in 1965. From all these different concepts, Turing's Concept is accepted as model of computation or algorithm. According to Church-Turing's thesis any algorithm that a human or computer can carried out can be solved by a Turing Machine. Now it has been universally accepted that an ideal theoretical model of computer can be provided by Turing machine. Turing machine accepted type 0 or unrestricted languages. Turing machine can be used to determine decidability of certain languages, measuring time and space complexity of problems.

1.5 COMPLEXITY OF ALGORITHM

How many steps are required by the algorithm to solve a given problem is algorithm complexity. The function of input data size is evaluated the order of count of operations executed by an algorithm. The order of count of operation is always considered to assess the complexity instead of counting the exact steps. O(f) notation represents the complexity of an algorithm and is termed as an Asymptotic notation or **"Big O"** notation. The order in which resources such as CPU time, memory, etc. are consumed by the algorithm ids determined by the complexity of the asymptotic

computation O(f). Constant, logarithmic, linear, $n * \log(n)$, quadratic, cubic, exponential, etc etc are the form of complexity.

1.6 COMPLEXITY CLASSES

A complexity class is defined by 3 ways. They are: (1) model of computation, (2) resource (or collection of resources), and (3) function known as the complexity bound for each resource. The complexity classes can be defined by using models into two main categories: (a) machine based models, and (b) circuit-based models. Random-access machines (RAMs) and Turing machines (TMs) are the two principal families of machine models. The complexity problems that we are trying to understand are because of the model of nondeterministic Turing machines. Nondeterministic machines do model real computational problems instead of model physical computation devices.

A problem is decidable; it means that the problem is computationally solvable in principle. It may not be solvable in practical, means it may require enormous amount of memory and computation time. In this chapter we discuss the computational complexity of a problem; The proofs of decidability/undecidability are quite rigorous, as they depend solely on the definition of a rigorous mathematical techniques and Turing machine. The proof and the discussion in complexity theory based on the assumption that $P \neq NP$. The computer scientists and mathematicians strongly believe that $P \neq NP$ but this is still debatable. The class of problems that

Can be solved by a deterministic algorithm in polynomial time is represented by P
(i.e. by a Turing machine) and for the class of problems that can be solved by a nondeterministic algorithm in polynomial time is represented by **NP** (that is, by a nondeterministic TM). Here P means for polynomial and NP for nondeterministic polynomial. Another important class is the class of NP-complete problems which is a subclass of NP.

1.6.1 The Class P and NP

Time Complexity: A Turing machine M is said to be of time complexity $T(n)$ if the following holds: Given an input 'w' of length n, M halts after making at most $T(n)$ moves.

Class P: A language L is in class P if there exists some polynomial $T(n)$ such that $L = T(M)$ for some deterministic TM M of time complexity $T(n)$.

Class NP: A language L is in class **NP** if there is a nondeterministic TM M and a polynomial time complexity $T(n)$ such that $L = T(M)$ and M executes at most $t(n)$ moves for every input w of length n.

1.7 NECESSITY OF FOCUSING ON THESE CLASSES

Many familiar problems such as finding shortest paths in networks, parsing context-free grammars, sorting, matrix multiplication, and linear programming; that can be solved efficiently are in class P. In fact, P contains all problems that can be solved by programs of reasonable worst-case time complexity. There are problems whose best algorithms have time complexity $n^{10^{500}}$ are also in class P. The four important reasons for which they could be included these problems as it seems unreasonable to say that such problems are computationally feasible.

1. The main goal of proving lower bounds is that, it is sensible to have an overly generous notion of the class of feasible problems. If we show that a problem is not in P, then we have shown in a very strong way that solution via deterministic algorithms is impractical.

2. "If functions f and g are both easy to compute, then the composition of f and g should also be easy to compute", the theory of complexity-bounded reducibility is predicated on this simple notion. If we want to allow algorithms of time complexity n^2 to be considered feasible, then we are immediately led to regard running times n^4, n^8 ...as also being feasible. In other words, the choice is either to lay down an arbitrary and artificial limit on feasibility (and to forgo the desired property that the composition of easy

functions be easy), or to go with the natural and overly-generous notion given by P.

3. The intellectual boundary between feasible and infeasible problems is served well by Polynomial time. Logically, problems of time complexity $n^{10^{500}}$ do not arise, while problems of time complexity $O(n^4)$, and those proved or believed to be $\Omega(2^n)$, occur often. Moreover, once a polynomial-time algorithm for a problem is found, and a base of mathematical and algorithmic techniques can be used to improve the algorithm. The best known example is Linear programming. The breakthrough $O(n^8)$ time algorithm of [Khachiyan, 1979], for n x n instances, was impractical, but it helped in an innovation by [Karmarkar, 1984] that produced an algorithm whose running time of about $O(n^3)$ on all cases competes well commercially with the simplex method. It runs in $O(n3)$ time in most of the cases but in some cases it takes 2^n time. Of course, if it should turn out that the Hamiltonian circuit problem (or some other NP-complete problem) has complexity $n^{10^{500}}$, then the theory would need to be fixed but as of now it seems unlikely.

4. We would like our fundamental notions to be independent of arbitrary choices. It is arbitrary and historically accidental in the prevalent choice of the Turing machine as the standard model of computation. This choice does not affect the class P itself, however, because the natural notions of polynomial time for essentially all models of sequential computation that have been invented yield the same class.

By analogy to the famous Church-Turing thesis, which states that the definition of a (partial) recursive function captures the intuitive notion of a computable process, several authorities have proposed the following:-

Polynomial-Time Church-Turing Thesis : The class P captures the true notion of those problems that are computable in polynomial time by sequential machines, and is the same for any physically relevant model and minimally reasonable time measure of sequential computation that will ever be devised. This thesis extends also to

parallel models if "time" is replaced by the technologically important notion of parallel work .Another way in which the concept of P is robust is that P is characterized by many concepts from logic and mathematics that do not mention machines or time.

The class NP can also be defined by means other than nondeterministic Turing machines. NP equals the class of problems whose solutions can be verified quickly, by deterministic machines in polynomial time. Equivalently, NP comprises those languages whose membership proofs can be checked quickly.

For example, one language in NP is the set of composite numbers, written in binary.

Proof
A number z is composite can consist of two factors $z1 \geq 2$ and $z2 \geq 2$ whose product $z1z2$ equals z. This proof is quick to check if z1 and z2 are given, or guessed. Correspondingly, one can design a nondeterministic Turing machine N that on input z branches to write down "guesses" for z1 and z2, and then deterministically multiplies them to test whether $z1z2 = z$. Then L(N), the language accepted by N, equals the set of composite numbers, since there exists an accepting computation path if and only if z really is composite. Note that N does not really solve the problem; it just checks the candidate solution proposed by each branch of the computation.

Example: Construct the time complexity *T(n)* for the Turing machine that accepts $\{0^n 1^n \mid n \geq 1\}$

Solution:
The given TM step (i) consists of going through the input string $(0^n 1^n)$ forward and backward and replacing the leftmost 0 by *x* and the leftmost 1 by y. SO we require at most *2n* moves to match a 0 with a 1. Step (ii) is repetition of step (i) n times. Hence the number of moves for accepting $0^n 1^n$ is at most *(2n)(n)*. For strings not of the form $0^n 1^n$, TM halts with less than $2n^2$ steps. Hence $T(M) = O(n^2)$.

We can also define the complexity of algorithms. In the case ofalgorithms. *T(n)* denotes the running time for solving a problem with an input of size *n.* using this algorithm.

Example: Find the running time for the Euclidean algorithm for evaluating gcd(a,b) where *a and b* are positive integers expressed in binary representation.

Solution

The Euclidean algorithm has the following steps:

1. The input is *(a,b)*
2. Repeat until *b* = 0
3. Assign *a* ← *a* mod *b*
4. Exchange *a* and *b*
5. Output *a*.

Step 3 replaces *a* by *a* mod *b*. If *a/2* ≥ *b,* then *a* mod *b*< *b* ≤*a/2.* If *a/2* < *b,* then *a* < *2b.* Write *a* = *b* + *r* for some *r* < *b.* Then *a* mod *b* = *r* < *b* < *a/2.* Hence a mod *b* ≤ *a/2.* So *a* is reduced by at least half in size anther application of step 3. Hence one iteration of step 3 and step 4 reduces *a* and *b* by at least half in size. So the maximum number of times the steps 3 and 4 reduces *a* and *b* by at least half in size. So the maximum number of times the steps 3 and 4 are executed is min $\{[\log_2 a], [\log_2 b]\}$. If *n* denotes the maximum of the number of digits of *a* and *b.* that is max$\{\{[\log_2 a], [\log_2 b]\}$ then the number of iterations of steps 3 and 4 is O(n). We have to perform step 2 at most min $\{\{[\log 2a], [\log 2b]\}$ times or *n* times. Hence $T(n) = nO(n) = O(n)$

Note: The Euclidean algorithm is a polynomial algorithm.

We know that for a deterministic TM M_1 simulating a non deterministic TM *M* exists .If *T(n)* is the complexity of *M,* then the complexity of the equivalent deterministic TM M_1 is $2^{0(T(n))}$. This can be justified as follows.

The processing of an input string w of length *n* by *M* is equivalent to a 'tree' of computations by M_1. Let *k* be the maximum of the number of choices forced by the nondeterministic transition function. (It is max| δ(q, x)| , the maximum taken over all states *q* and all tape symbol *X).* Every branch of the computation tree has a length *T(n)* or less. Hence the total number of leaves is at most *kT(n).* Hence the complexity of M_1 is at most $2^{0(T(n))}$.

It is not known whether the complexity of M_1 is less than $2^{0(T(n))}$. Once again an answer to this question will prove or disprove P ≠

NP. But there do algorithms exist where *T(n)* lies between a polynomial and an exponential function.

CHECK YOUR PROGRESS

a) Type 1 grammar is called _____.
b) According to Church-Turing's thesis any algorithm that a human or computer can be carried out can be solved by a _____
c) _____ and _____ are the two principal families of machine models.

1.8 POLYNOMIAL TIME REDUCTION AND NP-COMPLETENESS

If P_1 and P_2 are two problems and $P_2 \in$ P, then we can decide whether

$P_1 \in$ P by relating the two problems P_1 and P_2. If there is an algorithm for obtaining an instance of P_2 given any instance of P_1, then we can decide about

the problem P_1. Intuitively if this algorithm is a polynomial one, then the problem P_1 can be decided in polynomial time.

Theorem 1.1 : If there is a polynomial time reduction from P_1 to P_2 and if P_2 is in **P** then P_1 is in **P.**

Proof :

Let *m* denote the size of the input of P_1. As there is a polynomial time reduction of P_1 to P_2 the corresponding instance of P_2 can be got in polynomial-time. Let it be $O(n^k)$, So the size of the resulting input of P_2 is at most $C n^k$ for some constant c. As P_2 is in **P,** the time taken for deciding the membership in P_2 is $O(m^j)$, n being the size of the input of P_2. So the total

time taken for deciding the membership of m-size input of P_1 is the sum of the time taken for conversion into an instance of P_2, and the time for decision

of the corresponding input in P_2. This is $O[n^k + (n^k)^j]$ which is the same as $O(n^{ij})$. So P_1 is in **P.**

Definition 1.1: Let L be a language or problem in **NP.** Then L is *NP*complete
if
1. L is in **NP**
2. For every language L' in NP there exists a polynomial-time reduction of L' to L.

Theorem 1.2 : If P_1 is *NP-complete,* and there is a polynomial-time reduction of P_1 to P_2, then P_2 is *NP-complete.*

Proof:
 If L is any language in NP, we show that there is a polynomial-time reduction of L to P_2. As P_1 is *NP-complete,* there is a polynomial-time reduction of L to P_1. So the time taken for converting an n-size input string w in L to a string x in P_1 is at most p_1(n) for some polynomial p_1 . As there is a polynomial-time reduction of P1 to P_2 , there exists a polynomial P_2 such that the input x to P_1 is transferred into input y to P_2 in at most $P_2(n)$ time. So the time taken for transforming w to y is at most $p1(n) + p2\left(p1(n)\right)$. As $p1(n) + p2\left(p1(n)\right)$is a polynomial, we get a polynomial-time reduction of L to P_2. Hence *P2* is *NP-complete.*

Theorem1.3 : If some NP-complete problem is in P, then P =NP.

Proof :
        Let P be an NP-complete problem and PϵP. Let L be any NP-complete problem. By definition, there is a polynomial-time reduction of L to P. As P is in P, L is also in P by Theorem 1. Hence NP = P.

1.9 SATISFIABILITY PROBLEM (SAT) is NP-COMPLETE

Another important language in NP is the set of satisfiable Boolean formulas, called **SAT**.

Boolean formula Φ is satisfiable if there exists a way of assigning true or false to each variable such that under this truth assignment, the value of Φ is true. For example, the formula $x \wedge (\bar{x} \vee y)$ is satisfiable, but $x \wedge \bar{y} \wedge (\bar{x} \vee y)$ is not satisfiable. A nondeterministic Turing machine N, after checking the syntax of Φ and counting the number n of variables, can non deterministically write down an n-bit 0-1 string a on its tape, and then deterministically (and easily) evaluate Φ for the truth assignment denoted by a. The computation path corresponding to each individual a accepts if and only if Φ (a) = true, and so N itself accepts Φ if and only if Φ is satisfiable; i.e. $L(N) = SAT$.

Again, this checking of given assignments differs significantly from trying to find an accepting assignment.

The above characterization of NP as the set of problems with easily verified solutions is formalized as follows:

$A \in NP$ if and only if there exist a language $A' \in P$ and a polynomial p such that for every x, $x \in A$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in A'$. Here, whenever $x \in A$, y is interpreted as a positive solution to the problem represented by x, or equivalently, as a proof that $x \in A$. NP represents all sets of theorems with proofs that are short (i.e., of polynomial length), and P represents the statements that can prove or disproved quickly from scratch.

The theory of NP-completeness, together with the many known NP-complete problems, is perhaps the best justification for interest in the classes P and NP. All of the other canonical complexity classes listed above have natural and important problems that are complete for them. Further motivation for studying L, NL, and PSPACE, comes from their relationships to P and NP. L and NL are the largest space-bounded classes known to be contained in P, and PSPACE is the smallest space-bounded class known to contain NP. (It is worth mentioning here that NP does not stand for "non-polynomial time"; the class P is a subclass of NP.)

The satisfiability problem for boolean expressions (whether a boolean expression is satisfiable) is NP-complete and it was the first problem to be proved NP-complete.

1.9.1 Cook's Theorem

Theorem 1.4 (Cook's theorem) SAT is *NP-complete.*

Proof:

SAT ϵ NT

If the encoded expression E is of length n, then the number of variables is *[n/2]*. Hence, for guessing a truth assignment t we can use multi tape TM for E. The time taken by a multi tape NTM M is $O(n)$. Then M evaluates the value of E for a truth assignment t. This is done in $O(n^2)$ time. An equivalent single-tape TM takes $0(n4)$ time. Once an accepting truth assignment is found, M accepts E and M and halts. Thus we have found a polynomial time NTM for SAT. Hence SAT ϵ NP.

1.9.2 Other NP-Complete Problems

It is difficult to prove the NP-completeness of any problem. But after getting one NP-complete problem such as SAT *P' by* obtaining a polynomial reduction of SAT to *P'* we can prove the NP-completeness of the problem. The polynomial reduction of SAT to *P'* is relatively easy. Here we will have a list of *NP-complete* problems without proving their NP-completeness. Many of the *NP-complete* problems are of practical interest.

1. CSAT- Given a Boolean expression in CNF (conjunctive normal form *), is it satisfiable?

We can prove that CSAT is *NP-complete* by proving that CSAT is in **NP** and getting a polynomial reduction from SAT to CSAT

* A formula is in conjunctive normal form(CNF) if it is a product of elementary sums.
    If a is in disjunctive normal form, then $\neg a$ is in conjunctive normal form. (This can be seen by applying the DeMorgan's laws.) So to obtain the conjunctive normal form of a, we construct the disjunctive normal form of $\neg a$ and use negation.

2. Hamiltonian circuit problem - Does G have a Hamiltonian circuit (i.e.a circuit passing through each edge of G exactly once)?

3. Travelling salesman problem (TSP)-Given *n* cities, the distance between them and a number *D,* does there exist a tour programme for a salesman to visit all the cities exactly once so that the distance travelled is at most *D?*

4. Vertex cover problem-Given a graph G and a natural number *k,* does there exist a vertex cover for G with *k* vertices? (A subsets C of vertices of G is a vertex cover for G if each edge of G has an odd vertex in C.)

5. Knapsack problem-Given a set $A = \{a1, a2, a3, \ldots\ldots, an\}$ of nonnegative integers. and an integer *K,* does there exist a subset *B* of *A* such that $\sum_{bj \epsilon B} bj = K$?

This list of *NP-complete* problems can be expanded by having a polynomial reduction of known *NP-complete* problems to the problems which are in **NP** and which are suspected to be *NP-complete.*

1.10 USE OF NP-COMPLETENESS

NP-complete prevent us from wasting our time and energy over finding polynomial or easy algorithms for that problem. Also we may not need the full generality of an *NP-complete* problem. Particular cases may be useful and they may admit polynomial algorithms. Also there may exist polynomial algorithms for getting an approximate optimal solution to a given NP-complete problem.

For example, the travelling salesman problem satisfying the triangular inequality for distances between cities (i.e.$dij \leq dik + dki$ for all i, j, k) has approximate polynomial algorithm such that the ratio of the error to the optimal values of total distance travelled is less than or equal to 1/2.

1.11 ANSWERS TO CHECK YOUR PROGRESS

a) Context-sensitive grammer
b) Turing Machine
c) Random Access Memory, Turing Machine

1.12 POSSIBLE QUESTIONS

1) Define Language and grammar.

2) Describe Chomsky Hierarchy with examples.

3) The set of all languages whose complements are in NP is called **CO-NP.** Prove that **NP = CO-NP** if and only if there is some *NP-complete* problem whose complement is in **NP.**

4) Is $f(a, b, c, \mathrm{d}) = (a \vee b \vee c) \wedge (a \vee b \vee d)$ satisfiable?

1.13 REFERENCES AND SUGGESTED READINGS

1. Atallah, M. J., & Blanton, M. (Eds.). (2009). *Algorithms and theory of computation handbook, volume 2: special topics and techniques*. CRC press.

2. Allender, E., Loui, M. C., & Regan, K. W. (2014). Complexity Theory.

3. Mishra, K. L. P., & Chandrasekaran, N. (2006). *Theory of Computer Science: Automata, Languages and Computation*. PHI Learning Pvt. Ltd..

4. Yates, D. F., Templeman, A. B., & Boffey, T. B. (1984). The computational complexity of the problem of determining least capital cost designs for water supply networks. *Engineering Optimization*, *7*(2), 143-155.

5. Nothen, E., & de Fátima Mastroianni, M. Data structures, from theory to bits: Using theory of formal languages to analyze structured data. In *IEEE CACIDI 2016-IEEE Conference on Computer Sciences* (pp. 1-6). IEEE.

Space for learners:

UNIT 2: THEORY OF NP COMPLETENESS II

Unit Structure:

2.1 INTRODUCTION

Whether the most useful tool that is delivered by complexity theory is the notion of reducibility or not is a little doubtful. Many computational problems such as Travelling Salesman Problem, their deterministic time or space complexity are still not briefly known. Here we cannot say whether class P and NP are distinct till now. But still it's useful for new problem whose complexity is needed to calculate, say X and can be say that the complexity of X and Travelling Salesman are same by showing some efficient ways of reducing each problem o the other. In the cases where the exact complexity cannot be pinpoint, there showing problems equivalent in such way can solve the problem. According to reducibility

relation the number of real world computational problem which are similar must be of a large number but surprisingly the number is very less. Therefore complexity of any problem can be classified as it must be equivalent to any of the sort listed representative problem, which was originally not expected. The complexity classes of these representative problems were discussed in the previous chapter using Turing machines, a small set of abstract machine concepts.

With some time defining and space bounding simple functions, the complexity of major computational problems are possible to characterise though most problems have no similarity with any questions of Turing machines. This technique is way more successful than anyone can expect. Because of this we are forced to believe that, the problems being placed under one class is not an accident and all the classes are distinct in nature and the classification is real. Nondeterministic Turing machines, has ability to soar through huge search spaces, and it is appear to be way more powerful than mundane deterministic machines, and this strengthen our belief. Until P vs. NP and other long-standing questions of complexity theories are completely resolved, to understand the complexity of real-world problems the best way will be reducibility of classification.

2.2 REDUCIBILITY RELATIONS

In mathematics, the simplest way to solve a new problem, the problem must reduce to a problem which has been solved previously. To interpret the solution of a new problem, the problem must express in terms of a prior problem. This type of reduction is called **many-one reducibility**.

Using the subroutine of the prior problem the new problem can be solved. For example, we can solve a optimisation problem that has a feasible solution and it maximises the value of an objective function f by repeatedly calling a subroutine that solves the corresponding decision problem of whether there exists a feasible solution y whose value $f(y)$ satisfies $f(y) \geq k$. This reduction is called **Turing reducibility**,

Let L1 and L2 be languages. L1 is many-one reducible to L2, andL1 $\leq$ L2, if there exists a total recursive function f such that for all y, where y $\in$ L1 if and only iff(y) $\in$ L2. The function f is called the **transformation function**. L1 is Turing reducible to L2, writtenL1 $\leq$ T L2, if L1 can be decided by a deterministic oracle Turing machine M using L2 as its oracle, i.e., L1 = L (MA2) (The oracle for L2 models a hypothetical efficient subroutine for L2).

In above case if M or f consumes too much space or time then the computed reduction will not helpful. The study of complexity classes which are defined by bounds on time and space resources, resource-bound reducibility must be consider. Let L1 and L2 be languages.

❖ L1 is Karp reducible to L2 and written as L1 $\leq \frac{p}{m}$ L2, if L1 is many-one reducible to L2 via a transformation function which is computable deterministically in polynomial time.

❖ L1 is Cook reducible to L2, and written as L1$\leq \frac{P}{T}$ L2, if L1 is Turing reducible to L2 via a deterministic oracle Turing machine of polynomial time complexity.

The term "polynomial-time reducibility" usually refers to Karp reducibility. If $L_1 \leq \frac{p}{m} L_2$ and $L_2 \leq \frac{p}{m} L_1$, then we can say that L1 and L2 are equivalent under Karp reducibility. Similarly the Equivalence under Cook reducibility is defined.

To find the relationships between languages of high complexity, Karp and Cook reductions are useful but in case of distinguishing between problems in P Karp and Cook reductions are not at all useful as all the problems which are in P, are equivalent under Karp (and hence Cook) reductions.

The preserve polynomial-time feasibility is the key property of Cook and Karp reductions. Suppose $L_1 \leq \frac{p}{m} L_2$ via a transformation g. If M_2 decides L_2, and Mg computes g, then to decide whether an input word y is in L_1, we may use M_g to compute g(y), and then run M2 on input g(y). If the time complexities of M_2 and M_g are bounded by polynomials r_2 and r_g , respectively, then on inputs y of length n

=|y|, the time taken by this method of deciding L_1 is at most $r_g(n)+r_2(r_g(n))$, which is also a polynomial in n. It can be summaries as, if L2 is feasible, and there is an efficient reduction from L1 to L2, then L1 is also feasible. Even though this is just a simple observation, but this fact is significant enough to state as a theorem. Here we need the concept of "closure."

A class of languages A is closed under a reducibility $\leq_r$ if for all languages L1 and L2, Whenever L1 $\leq_r$ L2 and L2 ϵ A, necessarily L1 ϵ A.

Theorem 2.1 P is closed under both Cook and Karp reducibility.

This is an instance of an idea that motivated our identification of P with the class of "feasible" problems, that the composition of two feasible functions should be feasible.

Theorem 2.2 Karp reducibility and Cook reducibility are transitive, that is :

1. If $L_1 \leq_m^p L_2$ and $L_2 \leq_m^p L_3$, then $L_1 \leq_m^p L_3$.
2. If $L_1 \leq_T^P L_2$ and $L_2 \leq_T^P L_3$, then $L_1 \leq_T^P L_3$.

3 Complete Languages and Cook's Theorem

Let A be a class of languages that represent computational problems. A language L_0 is C-hard under a reducibility $\leq_r$ if for all L in A, L $\leq_r$ L_0. A language L_0 is C-complete under $\leq_r$ if L0 is C-hard, and L_0 ϵ A. Informally, if L_0 is C-hard, then L_0 represents a problem that is at least as difficult to solve as any problem in A. If L_0 is C-complete, then in a sense, L_0 is one of the most difficult problems in A.

Completeness can be viewed in other ways. Completeness provides the tight lower bounds on complexity of problems. For complexity class A, if a language L is complete, then there will be a lower bound on its complexity. L is as hard as the most difficult problem in A, assuming that the complexity of the reduction itself is small enough. As L is in A, the lower bound is tight i.e, the upper bound matches the lower bound.

In the case A = NP, the reducibility $\leq_r$ is usually taken to be Karp reducibility unless stated otherwise .Thus we can say:

- ❖ A language L0 is NP-hard if L0 is NP-hard under Karp reducibility.
- ❖ A0 is NP-complete if A0 is NP-complete under Karp reducibility.

However, many sources take the term "NP-hard" to refer the *Cook reducibility*.

Some implications of the statement "L_0 is NP-complete," and also some things this statement doesn't mean.

If there exists a deterministic Turing machine that decides L_0 in polynomial time, i.e if $L_0 \in P$, then as P is closed under Karp reducibility, and it would follow that NP $\subseteq$ P, therefore P = NP. In real, the question that whether P is the same as NP reduced to the question whether any particular NP-complete language is in P or not. Or in other word we can say that, if any one of all NP-complete languages which are fall together is in P, then all the remaining NP-complete languages are in P and similarly if one language is not in P then others are also not in P. Another implication, that follows by a almost similar closure argument applied to co-NP, is that if L0 $\in$ co-NP then NP = co-NP. But it is also believed that it's unlikely to be NP=co-NP. A theorem given by Lander [Ladner, 1975b] shows that N$\neq$NP if and only if there exist a language L_0in NP-P such that L_0 is not NP-complete. Therefore, if P $\neq$ NP, then L_0 is a contradiction to the "definition".

Another misconception that arises from the statement "If L0 is NP complete, then L0 is one of the most difficult problems in NP" due to the misinterpretation of the statement. This interpretation is true up to one level. The NP-complete language L_0 is a kind of problem where it is not in P but is in NP.

2.3 NP-COMPLETE PROBLEMS AND COMPLETENESS PROOFS

To solve a computational problem where we do not know how to solve it but want to know how hard it is, the following steps will let us know the answer and may help to know the problem that are well brought-up even if the problem is NP-hard for general cases. The steps are:

1. State the problem in general mathematical terms, and formalize the statement.

2. Verify whether the problem belongs to NP.

3. If yes, then try to find it in a compendium of known NP-complete problems.

4. If no, then try to construct a reduction from a related problem that is known to be NP-complete or NP-hard.

5. Try to identify special cases of your problem that are (i) hard, (ii) easy, and/or (iii) the ones you need.

6. Even if your cases are NP-hard, they may still be amenable to direct attack by sophisticated methods on high-powered hardware.

These steps are combined with a traditional "theorem-proof" presentation and several long examples, but the same sequences are to be maintained

Step 1: Give a formal statement of the problem.
It must State without using any terms that are specific to one particular discipline. It is advisable to use common terms from mathematics and data objects in computer science, e.g. graphs, trees, matrices, vectors, alphabets, strings, logical formulas, mathematical equations. For example, a problem in evolutionary biology which will state by a Phylogenist would state in terms of "species" and "characters" and "cladograms" can be stated in terms of trees and strings, using an alphabet that represents the taxonomic characters. Standard notions of size, depth, and distance in trees can express the objectives of the problem.

If the problem involves computing a function that produces a lot of output, look for associated yes/no decision problems, because

decision problems have been easier to characterize and classify. For example, if we need to compute matrices of a certain kind, we have to look for whether the essence of the problem can be captured by yes/no questions about the matrices or not. Many optimization problems who looks for a solution of a certain minimum cost or maximum value can be turned into decision problems by including a target cost/value "x" as an input parameter, and phrasing the question of whether a there exists a solution of cost less than (or value greater than) the target x.

By ignoring or removing some particular elements from the problem, the problem can be present in over simplified form. It can help in verifying the category which is closest to the problem. In the process, we may learn some useful information about the problem.

Step 2.
When we have an adequate formalization, then first we must ask ourselves whether our decision problem belongs to NP? This is true if and only if candidate solutions that would bring a "yes" answer can be tested in polynomial time. If it answer belongs to NP, then that's good. If not, we may proceed to determine if it is NP-hard. The problem may be complete for a class such as PSPACE that contains NP.

Step 3.
Check whether our problem is already listed in a compendium of (NP-) complete problems. The book [Garey and Johnson, 1988] lists hundreds of NP-complete problems arranged according to category. A small example from Garey and Johnson, 1988:

VERTEX COVER
Instance: A graph G_1 and an integer *n*.
Question: Does G_1 have a set Z of *n* vertices such that every edge in G_1 is incident on a vertex in Z?

CLIQUE
Instance: A graph G_1 and an integer *n*.
Question: Does G_1 have a set N of n vertices such that every two vertices in N are adjacent in G_1?

HAMILTONIAN CIRCUIT

Instance: A graph G_1.

Question: Does G_1 have a circuit that includes every vertex exactly once?

3-DIMENSIONAL MATCHING

Instance: Sets A, B, C with $|A| = |B| = |C| = p$ and a subset $W \subseteq A \times B \times C$

Question: Is there a subset $W' \subseteq W$ of size q such that no two triples in W' agree in any coordinate?

PARTITION

Instance: A set W of positive integers.

Question: Is there a subset $W' \subseteq W$ such that the sum of the elements of W' equals the sum of the elements of $W - W'$?

INDEPENDENT SET

Instance: A graph G_1 and an integer n.

Question: Does G_1 have a set A of n vertices such that no two vertices in A are adjacent in G_1?

GRAPH COLOURABILITY

Instance: A graph G_1 and an integer n.

Question: Is there an assignment of colours to the vertices of G_1 so that no two adjacent vertices receive the same colour, and at most *n* colours are used overall?

TRAVELLING SALESPERSON (TSP)

Instance: A set of n "cities" $C_1, C_2, C_3, ... C_m$, with a distance $d(j,k)$ between every pair of cities C_j and C_k , and an integer K.

Question: Is there a tour of the cities whose total length is at most K, i.e., a permutation $c_1, c_2, ..., c_n$ of $\{12, ..., n\}$ such that $d(c_1, c_2) + ... + d(c_{n-1}, c_n) + d(c_n, c_1) \leq K$?

KNAPSACK

Instance: A set $F = \{f_1, f_2, ..., f_n\}$ of objects, each with an integer size $size(f_j)$ and an integer profit $profit(u_j)$, a target size t_0, and a target profit q_0.

Question: Is there a subset $F' \subseteq F$ whose total cost and total profit satisfy

$$\sum_{fj \in F'} size\ (fj) \leq t_0 \text{ and } \sum_{fj \in F'} profit\ (fj) \geq q_0$$

The languages that are in these problems are seen to belong to NP. For example, to show that TSP is in NP, one can build a nondeterministic Turing machine that simply guesses a tour and checks that the tour's total length is at most K.

Some comments on the last two problems are relevant to steps 1 and 2 above. Travelling Salesperson provides a single abstract form for many concrete problems about sequencing a series of test examples so as to minimize the variation between successive items. The Knapsack problem models the filling of a knapsack with items of various sizes, with the goal of maximizing the total value (profit) of the items. Many scheduling problems for multiprocessor computers can be expressed in the form of Knapsack instances, where the "size" of an item represents the length of time a job takes to run, and the size of the knapsack represents an available block of machine time.

If our problem is on the list of NP-complete problems then we can skip Step 4, and the compendium may give us further information for Steps 5 and 6. We may still wish to pursue Step 4 if we need more study of particular transformations to and from our problem.
If our problem is not on the list, it may still be close enough to one or more problems on the list to help with the next step.

Step 4.
Construct a reduction from an already-known NP-complete problem. Karp reductions come in three kinds.
- ❖ A restriction from your problem to a special case that is already known to be NP-complete.
- ❖ A minor adjustment of an already-known problem
- ❖ A combinatorial transformation.

The first two kinds of reduction are usually quite easy to do, and we give several examples before proceeding to the third kind.

Example. Partition $\leq^p_m$ Knapsack, by restriction: Given a Partition instance with integers a_j, the corresponding instance of Knapsack takes size(f_j) = profit(f_j) = a_j (for all j), and sets the targets a_0 and p_0 both equal to $(\sum_j a_j)/2$. The condition in the definition of the Knapsack problem of not exceeding a_0 nor being less than p_0 requires that the sum of the selected items meet the target $(\sum_j a_j)/2$ exactly, which is possible if and only if the original instance of Partition is solvable.

In this way, the Partition problem can be regarded as a restriction or special case of the Knapsack problem. Note that the reduction itself goes from the more-special problem to the more general problem, even though one thinks of the more-general problem as the one being restricted. The implication is that if the restricted problem is NP-hard, then the more-general problem is NP-hard as well, not vice-versa.

Example. Hamiltonian Circuit $\leq^p_m$ TSP by restriction: Let a graph G be given as an instance of the Hamiltonian Circuit problem, and let G have n vertices v_1, v_2 ,..., v_n. These vertices become the "cities" of the TSP instance that we build. Now define a distance function as follows:

$$d(i,j) = \{ \begin{matrix} 1 & \text{if } (vi, vj) \text{ is an edge in G} \\ n+1 & otherwise \end{matrix}$$

Set K = n. Clearly, k and K can be computed in polynomial time from G. If G has a Hamiltonian circuit, then the length of the tour that corresponds to this circuit is exactly n. Conversely, if there is a tour whose length is at most m, then each step of the tour must have distance 1, not n+1. Then each step corresponds to an edge of G, so the corresponding sequence of vertices forms a Hamiltonian circuit in G. Thus the function f defined by f (G) = $(\{d(i,j): 1 \leq i, j \leq n\}, K)$ is a polynomial-time transformation from Hamiltonian Circuit to TSP.

<div style="border:1px solid black; padding:10px;">

CHECK YOUR PROGRESS

a) What is many one reducibility?

b) What is a transformation function?

c) Completeness provides the tight _____ bounds on complexity of problems.

d) A language L0 is NP-hard if L0 is NP-hard under_____

</div>

2.4 NP-COMPLETENESS BY COMBINATORIAL TRANSFORMATION

Theorem 2.3 Independent Set is NP-complete. Hence also Clique and Vertex Cover are
NP-complete.

Proof. We have remarked already that the languages of these three problems belong to NP, and shown already that Independent Set $\leq \frac{p}{m}$ Clique and Independent $\leq \frac{p}{m}$ Vertex Cover.

It suffices to show that $3SAT \leq \frac{p}{m}$ Independent Set.

Construction. Let the Boolean formula Φ be a given instance of 3SAT with variables $a_1, a_2, \ldots, a_m$ and clauses $C_1, C_2, \ldots, C_n$. The graph G_{Φ} we build consists of a "ladder" on 2m vertices labelled $y_1, \bar{y}_1, y_2, \bar{y}_2, \ldots, y_m, \bar{y}_m$ with edges $(y_i, \bar{y}_i)$ for $1 \leq i \leq m$ forming the "rungs," and n "clause components."

 Here the component for each clause C_j has one vertex for each literal y_i or $\bar{y}_i$ in the clause, and all pairs of vertices within each clause component are joined by an edge. Finally, each clause component node with a label y_i is connected by a "crossing edge" to the node with the opposite label $\bar{y}_i$ in the ith "rung," and similarly each occurrence of $\bar{y}_i$ in a clause is joined to the rung node y_i. This finishes the construction of G_{Φ}.

 Also set p = m + n. Then the reduction function f is defined for all arguments Φ by $f(\Phi) = (G\Phi; p)$.

Complexity. It is not hard to see that f is computable in polynomial time given (a straightforward encoding of) Φ.
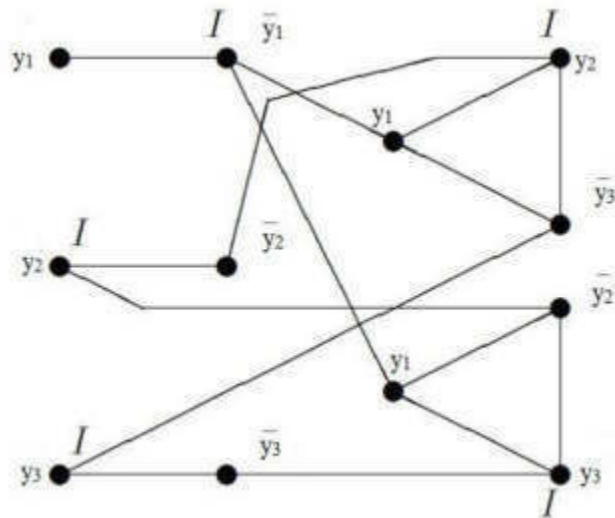
Figure 2.1: Construction in the proof of NP-completeness of Independent Set for the formula $(y1 \lor y2 \lor \bar{y}3) \land (y1 \lor \bar{y}2 \lor y3)$. The independent set of size 5 corresponding to the satisfying assignment y_1 = false, y_2 = true, and y_3 = true is shown by nodes marked I.

Correctness. To complete the proof, we need to argue that Φ is satisfiable if and only if G_Φ has an independent set of size m + n. To see this, first note that any independent set I of that size must contain exactly one of the two nodes from each "rung," and exactly one node from each clause component - because the edges in the rungs and the clause component prevent any more nodes from being added. And if I selects a node labelled y_i in a clause component, then I must also select y_i i in the ith rung. If I selects $\bar{y}_i$ in a clause component, then I must also select $\bar{y}_i$ in the rung. In this manner I induces a truth assignment in which y_i= true and y_i = false, and so on for all variables. This assignment satisfies Φ, because the node selected from each clause component tells how the corresponding clause is satisfied by the assignment. Going the other way, if Φ has a satisfying assignment, then that assignment yields an independent set I of size m +n in like manner.

Since the Φ in this proof is a 3SAT instance, every clause component is a triangle. The idea, however, also works for CNF formulas with any number of variables in a clause, such as the Φ_y in the proof of Cook's Theorem.

Now we modify the above idea to give another example of an NP-completeness proof by combinatorial transformation.

Theorem 2.4 Graph Colorability is NP-complete

Proof.
Construction: Given the 3SAT instance Φ, we build G_Φ similarly to the last proof, but with several changes. See Figure 4.2. On the left, we add a special node labelled "B" and connect it to all 2n rung nodes. On the right we add a special node "G" with an edge to B. In any possible 3-coloring of G_Φ, without loss of generality B will be coloured "blue" and the adjacent G will be coloured "green." The third colour "red" stands for literals made true, whereas green stands for falsity.

Now for each occurrence of a positive literal xi in a clause, the corresponding clause component has two nodes labelled y_i and y_i' with an edge between them; and similarly an occurrence of a negated literal $\bar{y}_j$ gives nodes $\bar{y}_j$ and $\bar{y}_j'$ with an edge between them. The primed ("inner") nodes in each component are connected by edges into a triangle, but the unprimed ("outer") nodes are not. Each outer node of each clause component is instead connected by an edge to G. Finally, each outer node y_i is connected by a "crossing edge" to the rung node $\bar{y}_i$ and each outer node $\bar{y}_j$ to rung node y_j, exactly as in the Independent Set reduction. This finishes the construction of G_Φ.
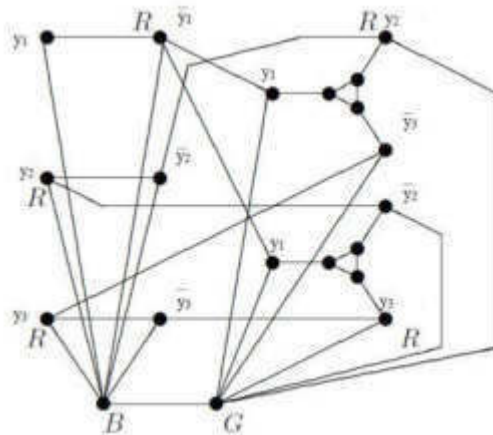
Figure 2.2 : Construction in the proof of NP-completeness of Graph Colourability for the formula (y1 ∨ y2 ∨ ȳ3) ∧ (y1 ∨ ȳ2 ∨ y3) . The

nodes shown colour R correspond to the satisfying assignment $y_1 =$ false, $y_2 =$ true, and $y_3 =$ true, and these together with G and B essentially force a 3-coloring of the graph, which the reader may complete. Note the resemblance to Figure 2.1.

Complexity. The function f that given any Φ outputs G_Φ also fixing $k = 3$, is clearly computable in polynomial time.

Correctness. The key idea is that every three-colouring of B, G, and the rung nodes, which corresponds to a truth assignment to the variables of Φ, can be extended to a 3-coloring of a clause component if and only if at least one of the three crossing edges from the component goes to a green rung node. If all three of these edges go to red nodes, then the links to G force each outer node in the component to be coloured blue, but then it is impossible to three-color the inner triangle since blue cannot be used. Conversely, any crossing edge to a green node allows the outer node y_i or $\bar{y}_j$ to be coloured red, so that one red and two blues can be used for the outer nodes, and this allows the inner triangle to be coloured as well. Hence G_Φ is 3-colorable if and only if Φ is satisfiable.

2.4.1 Disjoint Connecting Paths

Instance: A graph G with two disjoint sets of distinguished vertices v_1, v_2,....., v_j and u_1, u_2,....., u_j where $j \geq 1$.

Question: Does G contain paths $P_1 P_2$,......, P_j with each P_k going from v_k to u_k, such that no two paths share a vertex?

Theorem 4.3 Disjoint Connecting Paths is NP-complete.

Proof. First, it is easy to see that Disjoint Connecting Paths belongs to NP: one can design a polynomial-time nondeterministic Turing machine that simply guesses j paths and then deterministically checks that no two of these paths share a vertex. Now let Φ be a given instance of 3SAT with n variables and m clauses. Take $j = n + m$.

Construction and complexity. The graph G_Φ we build has distinguished path-origin vertices v_1, v_2,....., v_n for the variables and

$V_1, V_2, \ldots, V_m$ for the clauses of Φ. G_Φ also has corresponding sets of path-destination nodes $u_1, u_2, \ldots, u_n$ and $U_1, U_2, \ldots, U_m$. The other vertices in G_Φ are nodes b_{ij} for each occurrence of a positive literal y_i in a clause C_j, and nodes a_{ij} for each occurrences of a negated literal $\bar{y}_i$ in C_j. For each i, $1 \leq i \leq n$, G_Φ is given the edges for a directed path from v_i through all b_{ij} nodes to u_i, and another from v_i through all a_{ij} nodes to u_i. (If there are no occurrences of the positive literal x_i in any clause then the former path is just an edge from v_i right to u_i, and likewise for the latter path if the negated literal $\bar{y}_i$ does not appear in any clause.) Finally, for each j, $1 \leq j \leq m$, G_Φ has an edge from V_j to every node b_{ij} or a_{ij} for the jth clause, and edges from those nodes to U_j. Clearly these instructions can be carried out to build G_Φ in polynomial time given Φ. (See Figure 2.3.)

Correctness. The first point is that for each i, no path from v_i to u_i can go through both a "b-node" and a "a-node." Setting y_i true corresponds to avoiding b-nodes, and setting y_i false entails avoiding a-nodes. Thus the choices of such paths for all i represent a truth assignment. The key point is that for each j, one of the three nodes between V_j and U_j will be free for the taking if and only if the corresponding positive or negative literal was made true in the assignment, thus satisfying the clause. Hence G_Φ has the $n + m$ required paths if and only if Φ is satisfiable.
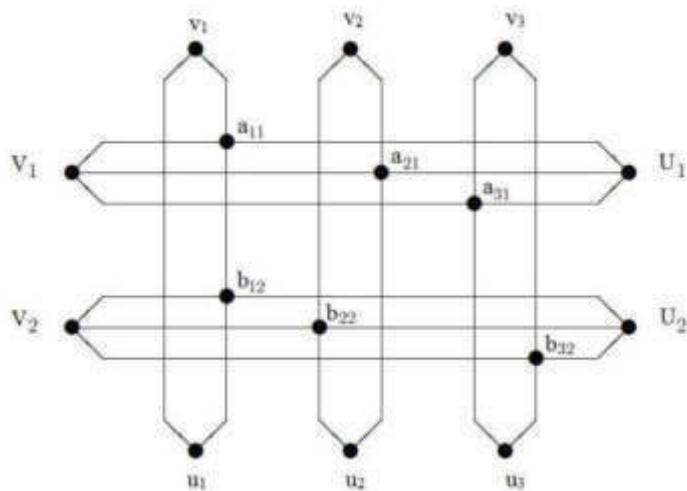

Figure 2.3: Construction in the proof of NP-completeness of Disjoint Connecting Paths for the formula (y1 ∨ y2 ∨ ȳ3) ∧ (y1 ∨ ȳ2 ∨ y3).

2.5 SIGNIFICANCE OF NP-COMPLETENESS

Suppose that you have proved that your problem is NP-complete. What does this mean, and how should you approach the problem now?

Exactly what it means is that your problem does not have a polynomial-time algorithm, unless every problem in NP has a polynomial-time algorithm; i.e., unless NP $\neq$ P. We have discussed above the reasons for believing that NP $\neq$ P. In practical terms, you can draw one definite

Conclusion: Don't bother looking for a "magic bullet" to solve the problem. A simple formula or an easily-tested deciding condition will not be available; otherwise it probably would have been spotted already during the thousands of person-years that have been spent trying to solve similar problems. For example, the NP-completeness of Graph 3-Colorability effectively ended hopes that an efficient mathematical formula for deciding the problem would pop out of research on "chromatic polynomials" associated to graphs. Notice that NP-hardness does not say that one needs to be "extra clever" to find a feasible solving algorithm, it says that one probably does not exist at all.

The proof itself means that the combinatorial mechanism of the problem is rich enough to simulate Boolean logic. The proof, however, may also unlock the door to finding saving graces in Steps 5 and 6.

Step 5. Analyze the instances of our problem that are in the range of the reduction. We may tentatively think of these as "hard cases" of the problem. If these differ markedly from the kinds of instances that you expect to see, then this difference may help you refine the statement and conditions of your problem in ways that may actually define a problem in P after all.

To be sure, avoiding the range of one reduction still leaves wide-open the possibility that another reduction will map into your instances of interest. However, it often happens that special cases of

NP-complete problems belong to P- and often the boundary between these and the NP-complete cases is sudden and sharp. For one example, consider SAT. The restricted case of three variables per clause is NP-complete, but the case of two variables per clause belongs to P.

For another example, note that the proof of NP-completeness for Disjoint Connecting Paths given above uses instances in which j = m+n; i.e., in which j depends on the number of variables.

The case j = 2, where you are given G and v_1, v_2, u_1, u_2 and need to decide whether there are vertex- disjoint paths from v_1 to u_1 and from v_2 to u_2, belongs to P. (The polynomial-time algorithm for this case is nontrivial and was not discovered until 1978, as noted in [Garey and Johnson, 1988].)

However, one must also be careful in one's expectations. Suppose we alter the statement of Disjoint Connecting Paths by requiring also that no two vertices in two different paths may have an edge between them. Then the case j = 2 of the new problem is NP-complete. (Showing this is a nice exercise; the idea is to make one path climb the "variable ladder" and send the other path through all the clause components.)
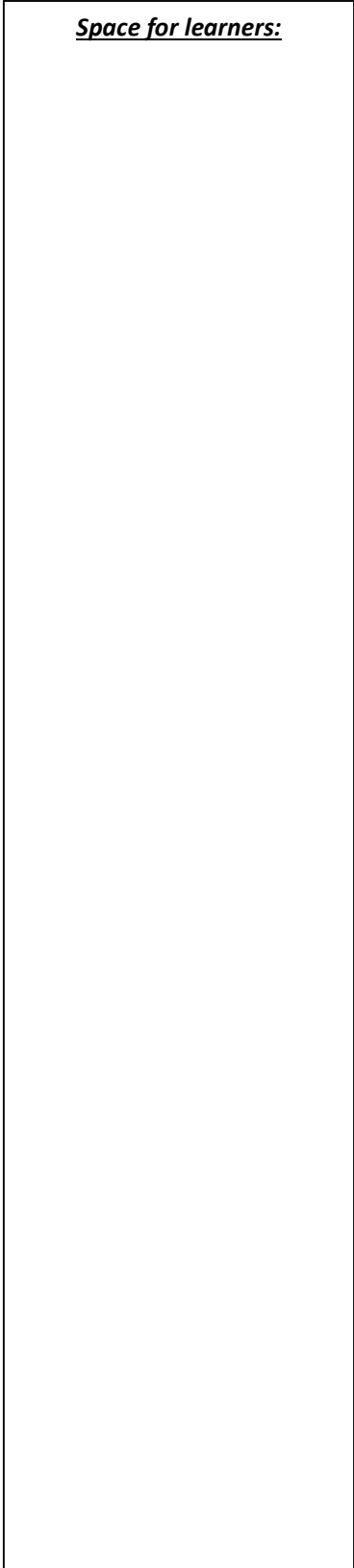
2.6 STRONG NP-COMPLETENESS FOR NUMERICAL PROBLEMS

An important difference between hard and easy cases applies to certain NP-complete problems that involve numbers. For example, above we stated that the Partition problem is NP-complete; thus, it is unlikely to be solvable by an efficient algorithm. Clearly, however, we can solve the Partition problem by a simple dynamic programming algorithm, as follows.

For an instance of Partition, let S be a set of positive $\{s_1, s_2, ..., s_m\}$ and let s* be the total, $s^* = \sum_{i=1}^{m} s_i$. Initialize a linear array B of Boolean values so that B[0] = true, and each other entry of B is false. For i = 1 to m, and for t = s* down to 0, if B[t] = true, then set B[t + s_i] to true. After the ith iteration, B[t] is true if and only if a

subset of $\{s_1, s_2,....., s_i\}$ sums to t. The answer to this instance of Partition is "yes" if B[s*/2] is ever set to true.

The running time of this algorithm depends critically on the representation of S. If each integer in S is represented in binary, then the running time is exponential in the total length of the representation. If each integer is represented in unary - that is, each s_i is represented by s_i consecutive occurrences of the same symbol- then total length of the representation would be greater than s*, and the running time would be only a polynomial in the length. Put another way, if the magnitudes of the numbers involved are bounded by a polynomial in m, then the above algorithm runs in time bounded by a polynomial in m. Since the length of the encoding of such a low-magnitude instance is O(mlogm), the running time is polynomial in the length of the input. The bottom line is that these cases of the Partition problem are feasible to solve completely.

A problem is NP-complete in the strong sense if there is a fixed polynomial p such that for each instance x of the problem, the value of the largest number encoded in x is at most $p(|x|)$. That is, the integer values are polynomial in the length of the standard representation of the problem. By definition, the 3SAT, Vertex Cover, Clique, Hamiltonian Circuit, and 3-Dimensional Matching problems defined in Section 4 are NP-complete in the strong sense, but Partition and Knapsack are not. The Partition and Knapsack problems can be solved in polynomial time if the integers in their statements are bounded by a polynomial in n- for instance, if numbers are written in unary rather than binary notation.

The concept of strong NP-completeness reminds us that the representation of information can have a major impact on the computational complexity of a problem.

2.7 COPING WITH NP-HARDNESS

Step 6. Even if we cannot escape NP-hardness, the cases we need to solve may still respond to sophisticated algorithmic methods, possibly needing high-powered hardware.

There are two broad families of direct attack that have been made on hard problems. Exact solvers typically take exponential time in the worst case, but provide feasible runs in certain concrete cases. Whenever they halt, they output a correct answer- and some exact solvers also output a proof that their answer is correct. Heuristic algorithms typically run in polynomial time in all cases, and often aim to be correct only most of the time, or to find approximate solutions. They are more common. Popular heuristic methods include genetic algorithms, simulated annealing, neural networks, relaxation to linear programming, and stochastic (Markov) process simulation. Experimental systems dedicated to certain NP-complete problems have recently yielded some interesting results- an extensive survey on solvers for Travelling Salesperson is given by [Johnson and McGeogh, 1997].

There are two ways to attempt to use this research. One is to find a problem close to yours for which people have produced solvers, and try to carry over their methods and heuristics to the specific features of your problem. The other (much more speculative) is to construct a Karp reduction from your problem to their problem, ask to run their program or machine itself on the transformed instance, and then try to map the answer obtained back to a solution of your problem.

The hitches are (1) that the currently-known Karp reductions f tend to lose much of the potentially helpful structure of the source instance x when they form f(x), and (2) that approximate solutions for f(x) may map back to terribly sub-optimal or even infeasible answers to x. All of this indicates that there is much scope for further research on important practical features of relationships between NP-complete problems

2.8 BEYOND NP-HARDNESS

If our problem belongs to NP and we cannot prove that it is NP-hard, it may be an '"NP intermediate" problem; i.e., neither in P nor NP-complete. According to the theorem of Ladner that NP-intermediate problems exist, assuming NP ≠ P. However, very few natural problems are currently counted as good candidates for such intermediate status: factoring, discrete logarithm, graph-

isomorphism, and several problems relating to lattice bases form a very representative list. The vast majority of natural problems in NP have resolved themselves as being either in P or NP-complete. Unless we uncover a specific connection to one of those four intermediate problems, it is more likely offhand that our problem simply needs more work.

The observed tendency of natural problems in NP to "cluster" as either being in P or NP complete, with little in between, reinforces the arguments made early in this chapter that P is really different from NP.

Finally, if our problem seems not to be in NP, or alternatively if some more-stringent notion of feasibility than polynomial time is at issue, then we may desire to know whether our problem is complete for some other complexity class.

2.9 ANSWERS TO CHECK YOUR PROGRESS

a) In mathematics, the simplest way to solve a new problem, the problem must reduce to a problem which has been solved previously. To interpret the solution of a new problem, the problem must express in terms of a prior problem. This type of reduction is called many-one reducibility.
b) Let L1 and L2 be languages. L1 is many-one reducible to L2, andL1 $\leq$ L2, if there exists a total recursive function f such that for all y, where y $\in$ L1 if and only iff(y) $\in$ L2. The function f is called the transformation function
c) Lower
d) Karp reducibility

2.10 MODEL QUESTIONS

1. Is $f(y,y_1,y_2,y_3)$= (y1 $\lor$ y2 $\lor$ $\bar{y}$3) $\land$ (y1 $\lor$ $\bar{y}$2 $\lor$ y3) satisfiable?
2. What does reducibility mean in NP-problems and why is it required?
3. What was the first problem proved as NP-Complete?
4. Multiple Choice Questions

i) Ram and Laxman have been asked to show that a certain problem $\prod$ is in NP-complete. Ram shows a polynomial time reduction rom 3-SAT problem to $\prod$, and Laxman shows a polynomial time reduction from $\prod$ to 3-SAT. Which of the following can be inferred from these reductions?
a) $\prod$ is NP-complete
b) $\prod$ is NP, but is not NP-complete
c) $\prod$ is neither NP-hard, nor in NP
d) $\prod$ is NP-hard not NP-complete

ii) Consider the following two problems of graph 1) given a graph; find if the graph has a cycle that visits every vertex once except the first visited vertex which must visit again to complete the cycle. 2) Given a graph, find if the graph has a cycle that visits every edge exactly once. Which of the following is true about above two problems?

a) Both problems belongs to P set
b) Both problem belongs to NP-complete set
c) Problem 1 belongs to P and problem 2 belongs to NP- complete
d) Problem 1 belongs to NP- complete and problem 2 belongs to P

iii) _____ is the class of decision problems that can be solved by non-deterministic polynomial algorithms?
a) NP   b) P   c) Hard   d) Complete

iv) How many conditions have to be met if an NP- complete problem is polynomially reducible?
a) 1   b) 2   c) 3   d) 4

v) Which of the following problems is not NP complete?
a) Hamiltonian Circuit   b) Bin packing
c) Partition problem   d) Halting problem

vi) To which class does Euler's circuit problem belongs
a) Decidable   b) Unpredictable
c) Complete   d) Trackable

vii) Halting is an example for?

a) Decidable   b) Undecidable

c) Trackable   d) Untrackable

1. Atallah, M. J., & Blanton, M. (Eds.). (2009). *Algorithms and theory of computation handbook, volume 2: special topics and techniques*. CRC press.

2. Allender, E., Loui, M. C., & Regan, K. W. (2014). Complexity Theory.

3. Mishra, K. L. P., & Chandrasekaran, N. (2006). *Theory of Computer Science: Automata, Languages and Computation*. PHI Learning Pvt. Ltd..

4. Yates, D. F., Templeman, A. B., & Boffey, T. B. (1984). The computational complexity of the problem of determining least capital cost designs for water supply networks. *Engineering Optimization*, *7*(2), 143-155.

5. Nothen, E., & de Fátima Mastroianni, M. Data structures, from theory to bits: Using theory of formal languages to analyze structured data. In *IEEE CACIDI 2016-IEEE Conference on Computer Sciences* (pp. 1-6). IEEE.

UNIT 3: LOWER BOUND THEORY

3.1 INTRODUCTION

For many problems, algorithms are designed and analysed to give correct and efficient solutions. However, a problem can be solved in various ways and we can have different algorithms for the same problem where one algorithm may be better than the others based on comparing the time complexities. But we cannot claim that this is the best algorithm for the problem. There might exist a faster algorithm with a better time complexity.

Lower Bound Theory provides those techniques to establish a given algorithm is an efficient one or not. We use a function g(n) as a lower bound on the time complexity and any algorithm that solves the given problem should have this bound otherwise asymptotically we do not have a better solution. Lower bounds of various problems are still unknown as finding the lower bound of a particular problem is harder.

3.2 UNIT OBJECTIVES

After going through this unit, you will be able to :

- *understand* the fundamental concepts of lower bound theory.

- *analyze* lower bound of an algorithm using different techniques.

- *define* and *create* comparison trees for comparison based algorithms.

- *develop* state space diagram for different algorithms.

3.3 LOWER BOUND THEORY

In problem solving, we design algorithms with the purpose of getting a correct and efficient solution. But the search for a fast and better algorithm continues even after discovering an efficient algorithm. We use the concept of Lower Bound Theory to establish that a given algorithm is the most efficient one. Lower Bound Theory gives the minimum time required for executing an algorithm.

Using Lower Bound Theory, our main aim is to find out the minimum number of comparisons required while executing an algorithm. Lower Bound Theory uses a number of techniques or methods to find the lower bound of a problem.

Lower bound, $L(n)$ can be defined as the property of a particular problem i.e. searching, sorting, matrix multiplication. Lower Bound Theory says that no algorithm can take lesser time than the time of $L(n)$ for any given input, n. So, lower bound theory is a method to check that the given algorithm for the problem solution is the most efficient one. To do so, we take a function $g(n)$ which is a lower bound on the time that any algorithm will take to solve the given problem. If our algorithm takes the same computing time in the same order as $g(n)$ then asymptotically we cannot be better.

Let the time complexity of some algorithm for solving a given problem be $f(n)$ then the lower bound on $f(n)$ is given by $f(n) = \Omega(g(n))$ i.e. there exists positive constants c and n_0 such that $|f(n)| >= c|g(n)|$ for all $n > n_0$. It is easier to write efficient algorithms but more

challenging to derive good lower bounds. As the lower bound gives a fact about all possible algorithms for solving a given problem. So, lower bound proofs are difficult to obtain as we cannot analyze all such algorithms.

Self Asking Questions (SAQ)

1. What is lower bound theory.

2. What sort of problems requires us to find the lower bound of a problem.

3. Why is difficult to find the lower bound of a problem.

3.4 TECHNIQUES TO FIND LOWER BOUND THEORY

There are various techniques to find the lower bound theory is:

1. Comparisons Trees.

2. Oracle and adversary argument.

3. State Space Method.

3.4.1 Comparisons Trees

This method is very easy and a popular computational model for determining the lower bounds for a given problem. Sorting and searching problems uses this method since comparison trees for sorting and searching problems is based on comparison of the elements and models all possible outcomes. This model is designed to work on a large number of sorting and searching problems.

3.4.1.1 Sorting Algorithm

Assuming that all input elements are distinct in the list containing n elements and the input sequence be $<a_1, a_2, a_3, \ldots\ldots, a_n>$. Any comparison between a_i and a_j between any two elements in the

input sequence will be given by $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i > a_j$, $a_i \geq a_j$ to determine the relative ordering between them forming a binary comparison tree. Each internal node in the comparison tree is represented by an ellipse and represents comparison between a_i and a_j whereas the leaf node is represented by a rectangle and contains elements in sorted order either in increasing or decreasing order. The leaves also indicate the terminating state of the algorithm with root being the starting state.
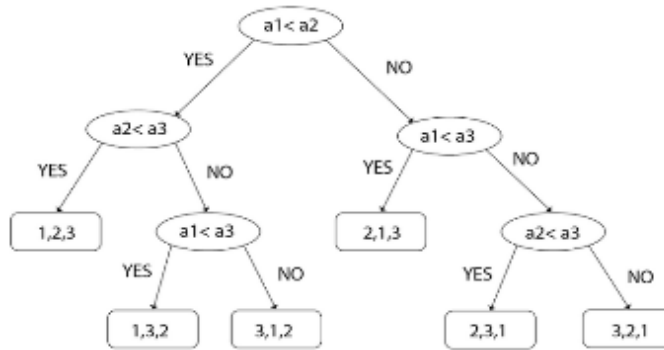
Fig 3.1: Decision Tree (sort 3 elements)

Analysis:

Let,

T(n) = Minimum number of comparisons to sort n numbers in the worst case.

K = Maximum height of the tree.

Now, let us assume that all the internal nodes are at level < K. Therefore, maximum numbers of leaves are 2^K. Then we have

$$2T(n) \geq n! \qquad \text{(Stirling's formula)}$$

$$\Rightarrow T(n) \geq \log n!$$

$$\Rightarrow T(n) = n\log n - \frac{n}{\log n} + \frac{1}{2}\log n + O(1)$$

$$\Rightarrow T(n) \geq n\log n$$

$$\Rightarrow T(n) \geq \Omega(n\log n)$$

Thus, any comparison based sorting algorithm for n elements runs in **Ω (n log n)** time.

3.4.1.2 Linear Searching

In linear searching, an unordered list L contains n elements. If an element,x is to be found in L then we need to compare x with every element of (L – x) i.e. x is compared with L[1], L[2], L[3], ... L[n]. If x is present at the ith position i.e. x = L[i] then searching terminates after i^{th} comparisons. Otherwise, searching continues as x is compared with rest of the elements in L. When x is found, search is successful otherwise it's a failure. In this case also, the comparison tree is a binary tree too. The leaves contain either failure(F) or success(S) and the internal nodes give the position,i of x in L. Depending on the number of comparisons, we can derive 3 different cases.

Best Case: If x is present at the root, then time complexity is minimum and it is Ω (1).

Average Case: If x is present at any other position except at the root node and the leaf node then search algorithm gives average time complexity. So, the average number of comparisons are $\frac{n}{2}$ and the time complexity, T(n) = Ω (n).

Worst Case: In this case, search continues till the end of the list i.e. x is present at the last position of the list or it is not present in L. So, total number of comparisons = n – 1 and the worst case time complexity,

T(n) = Maximum number of comparisons = n – 1 = O(n)

Also, any algorithm that searches a sorted sequence of n elements must perform at least (log n + 1) comparisons in the worst case.

<div style="text-align:center">

STOP TO CONSIDER

</div>

We can also use the comparison tree to find the lower bound of a binary search. The minimum number of comparisons needed to perform a search on n elements using binary search,$T(n) \geq log_2(n + 1)$.

3.4.2 Oracle and Adversary Argument

The second method for finding lower bound takes the help of oracles and adversaries. The oracle will tell us the outcome of each comparison for some model of estimation e.g. comparison trees. The oracles make the algorithm to work hard in order to get a good lower bound. The outcome of the next analysis is decided to determine the final answer. A worst case lower bound of the problem can be derived by keeping track of the work that has been finished.

The job of the adversary is almost the same as it also makes the algorithms cost high. The adversary is allowed to reorder the input to the algorithm in order to drive the cost of the algorithm higher. The adversary is not an entity in the program nor does it modify the program, it simply used for analysis only.

Suppose there are two sorted lists given to us , M[1 : m] and N[1 : n] where the elements are in ascending order. Let L be the merged list of M and N containing m + n elements also in the ascending order. We know that there are C_2^{m+n} ways to merge M and N i.e. it is possible that every element of M can be interleaved with every element of N in all possible ways in L and vice versa.

Assuming, M = <a,b> and N = <c,d,e> then there are C_2^{2+3} ways = C_2^5 ways = 10 ways to merge M and N. These are :

(a) a, b, c, d, e   (f) c, a, b, d, e

(b) a, c, b, d, e   (g) c, a, d, b, e

(c) a, c, d, b, e   (h) c, a, d, e, b

(d) a, c, d, e, b   (i) c, d, a, b, e

(e) c, d, e, a, b   (j) c, d, a, e, b

Now, if we use comparison tree as model of computation for merging M and N then there are C_m^{m+n} external nodes and atleast $log(C_m^{m+n})$ comparisons on the tree that uses any comparison based algorithms. However, using conventional technique of merging requires only (m + n - 1) comparisons. Therefore, $log(C_m^{m+n}) \leq merge(m,n) \leq (m+n-1)$ where merge(m,n) is the minimum number of comparisons required to merge m elements with n elements.

If m = 1, we need fewer comparisons.

If m = n, we have optimal number of comparisons for conventional merging and that's the lower bound too. Therefore, merge(m,n) = 2m – 1, $m \geq 1$.

3.4.3 State Space Method

This method gives a set of rules of an algorithm from a single comparison of a given state showing all possible states (n-tuples). We can now derive the lower bounds once the state transitions are known. This is possible because the finished state cannot be reached with lesser transactions. This approach requires counting the number of changes in state where a state is a collection of attributes. This will help us in finding out the smallest and biggest items using this method by comparison.

Our algorithm is modelled to define a state that an algorithm will be in at any given instant. In this way, we can define the start state, the end state and the transition states that the algorithm will traversal while moving from the start state to the end state. Thus, we can derive the minimum number of states the algorithm goes through from the start to the end to have a state space lower bound.

CHECK YOUR PROGRESS

1. State whether true or false :

a. Mostly searching and sorting algorithms use state space method for finding the lower bound.

b. Comparison trees are the most common method for finding the lower bound.

c. State space method shows all the possible states of an algorithm starting from the start to the end.

d. Oracle method for finding the lower theory gives a prediction of each comparison outcome.

e. Finding the lower bound is considered to be an easier task as compared to finding the worst case time of an algorithm.

3.5 SUMMING UP

- Lower bound for a problem is the tightest (highest) lower bound that can be proved for all possible algorithms solving the given problem.

- There can be theoretically infinite number solutions to a problem but it is not possible to know all the algorithms for any problem.

- We try to give a simple lower bound based on the amount of input that can be examined.

- Lower bound of an algorithm can be found using 3 techniques: Comparison trees, Oracle and adversary arguments; and state space method.

3.6 ANSWERS TO CHECK YOUR PROGRESS

1. a. FALSE

1. b. TRUE

1. c. TRUE

1. d. TRUE

1. e. FALSE

3.7 POSSIBLE QUESTIONS

1. Define lower bound of a problem.

2. What is the difference between worst case lower bound and average case lower bound?

3. Give the trivial lower bounds for the following:

    (a) Finding the transpose of a m x n matrix.

(b) Finding the median of n elements.

4. Prove that the lower bound of sorting a sequence of n elements using comparison based sorting algorithm is n log n.

5. Draw the decision tree for the following algorithms:

    (a)   Linear search on seven elements.

    (b)   Merge sort on five elements.

6. Let X and Y be two sorted lists of n elements each. Find the minimum number of comparisons to merge X and Y in the worst case.

3.8 REFERENCES SUGGESTED READINGS

1. Aho, A. V.; Hopcroft, J. E. & Ullman, J. D. ,*The Design and Analysis of Computer Algorithms* , Pearson .