

Output:

```
Enter the number of elements
3
Enter the profit and weights of the elements
For item no 1
10 30
For item no 2
20 15
For item no 3
30 50

Enter the capacity
45
Items included are
S1.no weight profit
1 30 10
2 15 20
Total profit = 30
```

Practical 3.3: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's Algorithm

Solution:

```
#include<stdio.h>
#include<conio.h>
#define infinity 999
void dij(int n,intv,int cost[10][10],int dist[100])
{
    int i,u,count,w,flag[10],min;
    for(i=1;i<=n;i++)
        flag[i]=cost[v][i];
    count=2;
    while(count<=n)
    {
        min=99;
        for(w=1;w<=n;w++)
            if(dist[w]<min && !flag[w])
```



```

min=dist[w],u=w;
flag[u]=1;
count++;
for(w=1;w<=n;w++)
if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
dist[w]=dist[u]+cost[u][w];
}

void main()
{
int n,v,i,j,cost[10][10],dist[10];
clrscr();
printf("\n Enter the number of nodes:");
scanf("%d",&n);
printf("\n Enter the cost matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=infinity;
}
printf("\n Enter the source matrix:");
scanf("%d",&v);
dij(n,v,cost,dist);
printf("\n Shortest path:\n");
for(i=1;i<=n;i++)
if(i!=v)
printf("%d->%d,cost=%d\n",v,i,dist[i]);
getch();
}

```

Output:

```
Enter the number of nodes:5
Enter the cost matrix:
 0   5   12   17   999
 999   0   999   8   ?
 999   999   0   9   999
 999   999   999   0   999
 999   999   999   999   0

Enter the source matrix:1

Shortest path:
1->2, cost=5
1->3, cost=12
1->4, cost=13
1->5, cost=12
```

Practical 3.4: Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's Algorithm

Solution:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
    clrscr();
    printf("\n\n\nImplementation of Kruskal's algorithm\n\n");
    printf("\nEnter the no. of vertices\n");
    scanf("%d",&n);
    printf("\nEnter the cost adjacency matrix\n");
    for(i=1;i<=n;i++)
    {
```



```

for(j=1;j<=n;j++)
{
    scanf("%d",&cost[i][j]);
    if(cost[i][j]==0)
        cost[i][j]=999;
}
printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
while(ne<n)
{
    for(i=1,min=999;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            if(cost[i][j]<min)
            {
                min=cost[i][j];
                a=u=i;
                b=v=j;
            }
        }
    }
    u=find(u);
    v=find(v);
    if(uni(u,v))
    {
        printf("\n%d edge (%d,%d) = %d\n",ne++,a,b,min);
        mincost +=min;
    }
    cost[a][b]=cost[b][a]=999;
}
printf("\nMinimum cost = %d\n",mincost);
getch();

```

```
}  
int find(int i)  
{  
    while(parent[i])  
        i=parent[i];  
    return i;  
}  
int uni(int i,int j)  
{  
    if(i!=j)  
    {  
        parent[j]=i;  
        return 1;  
    }  
    return 0;  
}
```

Output:

```
Implementation of Kruskal's algorithm  
Enter the no. of vertices  
4  
Enter the cost adjacency matrix  
0 20 10 50  
20 0 60 999  
10 60 0 40  
50 999 40 0  
The edges of Minimum Cost Spanning Tree are  
1 edge (1,3) =10  
2 edge (1,2) =20  
3 edge (3,4) =40  
Minimum cost = 70
```



Practical 3.5(a): Print all the nodes reachable from a given starting node in a digraph using BFS method

Solution:

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{
    for(i=1;i<=n;i++)
        if(a[v][i] && !visited[i])
            q[++r]=i;
    if(f<=r)
    {
        visited[q[f]]=1;
        bfs(q[f++]);
    }
}
void main()
{
    int v;
    clrscr();
    printf("\n Enter the number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        q[i]=0;
        visited[i]=0;
    }
    printf("\n Enter graph data in matrix form:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);
}
```

```
printf("\n Enter the starting vertex:");
scanf("%d",&v);
bfs(v);

printf("\n The node which are reachable are:\n");
for(i=1;i<=n;i++)
if(visited[i])
printf("%dt",i);
getch();
}
```

Output:

```
Enter the number of vertices:4
Enter graph data in matrix form:
0 1 1 1
0 0 0 1
0 0 0 0
0 1 0 0
Enter the starting vertex:1
The node which are reachable are:
2 3 4 -
```

Practical 3.5(b): Check whether a given graph is connected or not using DFS method

Solution:

```
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;

voiddfs(int v)
{
    int i;
    reach[v]=1;
```



```

for(i=1;i<=n;i++)
{
    if(a[v][i] && !reach[i])
    {
        printf("\n %d->%d",v,i);
        dfs(i);
    }
}

void main()
{
    int i,j,count=0;
    clrscr();
    printf("\n Enter number of vertices:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        reach[i]=0;
        for(j=1;j<=n;j++)
            a[i][j]=0;
    }
    printf("\n Enter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    dfs(1);
    printf("\n");
    for(i=1;i<=n;i++)
    {
        if(reach[i])
            count++;
    }
    if(count==n)
        printf("\n Graph is connected");
    else
}

```

```
printf("\n Graph is not connected");
getch();
}
```

Output:

```
Enter number of vertices:4
Enter the adjacency matrix:
0 1 0 1
0 0 0 0
0 0 0 0
0 0 1 0
1->2
2->4
4->3
Graph is connected
```

3.6 References

References of this unit have been given at the end of the book.

REFERENCES

1. ‘*Introduction to the Design and Analysis of Algorithms*’, Anany Levitin: 2nd Edition, 2009. Pearson.
2. ‘*Computer Algorithms/C++*’, Ellis Horowitz, Satraj Sahni and Rajasekaran, 2nd Edition, 2014, Universities Press.
3. ‘*Introduction to Algorithms*’, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, 3rd Edition, PHI.
4. ‘*Design and Analysis of Algorithms*’, S. Sridhar, Oxford (Higher Education).
5. ‘*Design and Analysis of Computer Algorithms*’ by AHO.
6. ‘*Introduction to Algorithms (Eastern Economy Edition)*’ by Thomas H. Cormen and Charles E. Leiserson.
7. ‘*Introduction to the Design and Analysis of Algorithms*’ by Anany Levitin.
8. ‘*Introduction to Design and Analysis of Algorithms*’ (Anna University) by Anany Levitin.
9. ‘*Design And Analysis of Algorithms*’ by Shweta Bajaj Mundra.
10. C++ Programming Examples on Hard Graph Problems and Algorithms.





www.cuidol.in

1800-1213-88800

INSTITUTE OF DISTANCE & ONLINE LEARNING

NH-95, Chandigarh-Ludhiana Highway, Gharuan, Mohali (Punjab)
Phone:- 7527009635 | Email: info@cuidol.in

FOLLOW US ON:





CHANDIGARH
UNIVERSITY

Discover. Learn. Empower.

**INSTITUTE OF
DISTANCE & ONLINE LEARNING**



MASTER OF COMPUTER APPLICATIONS

**DESIGN AND ANALYSIS
OF ALGORITHMS
(THEORY AND PRACTICAL)**
MCA632/MCA637

Self Learning Material

R101

**MASTER OF COMPUTER
APPLICATIONS**

**DESIGN AND ANALYSIS OF
ALGORITHMS**
(THEORY AND PRACTICAL)

MCA632/MCA637

Dr. Abdullah Khan
Solomon Aregawi



**CHANDIGARH
UNIVERSITY**

Discover. Learn. Empower.



CHANDIGARH UNIVERSITY
Institute of Distance and Online Learning
Course Development Committee

Chairman

Prof. (Dr.) R.S. Bawa

Vice Chancellor, Chandigarh University, Punjab

Advisors

Prof. (Dr.) Bharat Bhushan, Director, IGNOU

Prof. (Dr.) Majulika Srivastava, Director, CIQQA, IGNOU

Programme Coordinators & Editing Team

Master of Business Administration (MBA) **Bachelor of Business Administration (BBA)**

Co-ordinator - Prof. Pragya Sharma

Master of Computer Applications (MCA)

Co-ordinator - Dr. Deepti Rani Sindhu

Master of Commerce (M.Com.)

Co-ordinator - Dr. Shashi Singhal

Master of Arts (Psychology)

Co-ordinator - Dr. Samerjeet Kaur

Master of Arts (English)

Co-ordinator - Dr. Ashita Chadha

Master of Arts (Mass Communication and Journalism)

Co-ordinator - Dr. Chanchal Sachdeva Suri

Academic and Administrative Management

Prof. (Dr.) Pranveer Singh Satvat

Pro VC (Academic)

Prof. (Dr.) H. Nagaraja Udupa

Director – (IDOL)

Prof. (Dr.) S.S. Sehgal

Registrar

Prof. (Dr.) Shiv Kumar Tripathi

Executive Director – USB

© No part of this publication should be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise without the prior written permission of the author and the publisher.

**SLM SPECIALLY PREPARED FOR
CU IDOL STUDENTS**

Printed and Published by:

Himalaya Publishing House Pvt. Ltd.,

E-mail: himpub@bharamail.co.in, Website: www.himpub.com

For: **CHANDIGARH UNIVERSITY**

Institute of Distance and Online Learning



CU IDOL SELF LEARNING MATERIAL (SLM)



Design and Analysis of Algorithms (Theory and Practical)

Course Code: MCA632/MCA637

Credits: 3 (Theory)/1(Practical)

Course Objectives:

- To demonstrate a familiarity with major algorithms and data structures.
- To analyse the performance of algorithms.
- To apply important algorithmic paradigms and methods of analysis.

Syllabus

Unit 1 - Introduction: Algorithm Specification, Analysis Framework, Performance Analysis: Space Complexity, Time Complexity.

Unit 2 - Asymptotic Notations: Big-Oh Notation (O), Omega Notation (Ω), Theta Notation (Θ), and Little-oh Notation (o), Mathematical Analysis of Non-recursive and Recursive Algorithms with Examples.

Unit 3 - Important Problem Types: Sorting, Searching, String Processing, Graph Problems, Combinatorial Problems.

Unit 4 - Fundamental Data Structures: Stacks, Queues, Graphs, Trees, Sets and Dictionaries.

Unit 5 - Divide and Conquer: General Method, Binary Search, Merge Sort, Quick Sort, Advantages and Disadvantages of Divide and Conquer, Decrease and Conquer Approach: Topological Sort.

Unit 6 - Greedy Method 1: General Method, Coin Change Problem, Knapsack Problem, Job Sequencing with Deadlines, Minimum Cost Spanning Trees: Prim's Algorithm, Kruskal's Algorithm.

Unit 7 - Greedy Method 2: Single Source Shortest Paths: Dijkstra's Algorithm, Optimal Tree Problem: Huffman Trees and Codes, Transform and Conquer Approach: Heaps and Heap Sort.



Unit 8 - Dynamic Programming 1: General Method with Examples, Multistage Graphs, Transitive Closure: Warshall's Algorithm.

Unit 9 - Dynamic Programming 2: All Pairs Shortest Paths: Floyd's Algorithm, Optimal Binary Search Trees, Bellman-Ford Algorithm, Travelling Salesperson Problem, Reliability Design.

Unit 10 - Backtracking 1: General Method: N-Queens Problem, Sum of Subset Problem, Graph Colouring, Hamilton Cycles.

Unit 11 - Backtracking 2: Branch and Bound: Assignment Problem, Travelling Salesperson Problem.

Text Books:

1. Levitin, A. (2009), *Introduction to the Design and Analysis of Algorithms*, 2nd Edition, Delhi: Pearson Education.
2. Horowitz, E., Sahni, S. and Rajasekaran (2014), *Computer Algorithms/C++*, 2nd Edition, Hyderabad: Universities Press.

Reference Books:

1. Thomas, H., Cormen, Charles, E., Ronald, L., Rivest, L. and Stein, C. (2010), *Introduction*

Design and Analysis of Algorithms Practical

Course Code: MCA637

Credits: 1

Course Objectives:

- To indicate computational solution to well-known problems like searching, sorting, etc.
- To understand and estimate the computational complexity of different algorithms.
- To design an algorithm using appropriate design strategies for problem-solving.

Syllabus

Unit 1 - DAA Basics:

1. Determine whether a particular element is available in a list of elements entered by the user at runtime.
2. Obtain the topological ordering of vertices in a given digraph.

Unit 2 - Divide and Conquer and Greedy Method:

1. Sort a given set of elements using the Quick Sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
2. Using Open, implement a parallelised Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
3. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

Unit 3 - Dynamic Programming and Backtracking:

1. Compute the transitive closure of a given directed graph using Warshall's algorithm.
2. Implement 0/1 Knapsack Problem using Dynamic Programming.
3. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's Algorithm.
4. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.
5. (a) Print all the nodes reachable from a given starting node in a digraph using BFS method.
(b) Check whether a given graph is connected or not using DFS method.

Text Books:

1. Levitin, A. (2009). *Introduction to the Design and Analysis of Algorithms*, 2nd Edition, Delhi: Pearson Education.
2. Horowitz, E., Sahni, S., Rajasekaran (2014). *Computer Algorithms/C++*, 2nd Edition. Hyderabad: University Press.

Reference Books:

1. Thomas, H., Cormen, Charles, E., Ronald, L., Rivest, L. and Stein, C. (2010), *Introduction to Algorithms*, 3rd Edition, Delhi: PHI.
2. Sridhar, S. (2014), *Design and Analysis of Algorithms*, UK: Oxford (Higher Education).

CONTENTS

Unit 1:	Introduction	1 - 14
Unit 2:	Asymptotic Notations	15 - 34
Unit 3:	Important Problem Types	35 - 60
Unit 4:	Fundamentals of Data Structures	61 - 119
Unit 5:	Divide and Conquer	120 - 146
Unit 6:	Greedy Method 1	147 - 185
Unit 7:	Greedy Method 2	186 - 208
Unit 8:	Dynamic Programming 1	209 - 223
Unit 9:	Dynamic Programming 2	224 - 251
Unit 10:	Backtracking 1	252 - 272
Unit 11:	Backtracking 2	273 - 292
Practical Unit 1:	DAA Basics	293 - 301
Practical Unit 2:	Divide and Conquer and Greedy Method	302 - 309
Practical Unit 3:	Dynamic Programming and Backtracking	310 - 323
References		324

UNIT 1 INTRODUCTION

Structure:

- 1.0 Learning Objectives
- 1.1 Introduction
- 1.2 Algorithm Specification
- 1.3 Analysis Framework
 - 1.3.1 The Need for Analysis
 - 1.3.2 Characteristics of Algorithms
- 1.4 Performance Analysis
 - 1.4.1 Space Complexity and Time Complexity
 - 1.4.2 Time Complexity of an Algorithm
- 1.5 Summary
- 1.6 Key Words/Abbreviations
- 1.7 Learning Activity
- 1.8 Unit End Questions (MCQ and Descriptive)
- 1.9 References

1.0 Learning Objectives

After studying this unit, you will be able to:

- Define algorithm specifications
- Describe analysis framework



- Illustrate performance complexity
- Elaborate space complexity, time complexity
- Describe the difference between algorithm and pseudo code

1.1 Introduction

An Algorithm is a sequence of steps to solve a problem. Design and analysis of algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology. This chapter introduces the fundamental concepts of designing strategies, complexity analysis of algorithms, followed by problems on Graph Theory and sorting methods. This chapter also includes the basic concepts on Complexity Theory.

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space. An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm is independent from any programming language**.

Algorithm: An algorithm can be defined as a well-defined computational procedure that takes some values, or the set of values, as an input and produces some value, or the set of values, as an output. An algorithm is, thus, a sequence of computational steps that transforms the input into output. It describes specific computational procedures for achieving the input-output relationship.

For example, we need to sort the sequence of number into ascending order. Here is how we define the sorting problem.

Input: A sequence of n number (a_1, a_2, \dots, a_n)

Output: A permutation (reordering) (a'_1, a'_2, \dots, a'_n) of the input sequence such that

$$(a'_1 \leq a'_2 \leq \dots \leq a'_n)$$



Need of Algorithm

1. To understand the basic idea of the problem.
2. An approach to solve the problem.
3. To improve the efficiency and performance of existing techniques.
4. To understand the basic principles of designing the algorithms.
5. It is the best method of description without describing the implementation detail.
6. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
7. To measure the behaviour (or performance) of the methods in all cases (best cases, worst cases, average cases).
8. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
9. With the help of algorithm, we convert art into a science.
10. To understand the principle of designing.
11. We can measure and analyse the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

Algorithm vs. Program

A finite set of instructions that specifies a sequence of operations to be carried out to solve a specific problem of a class of problem is called an algorithm.

On the other hand, the program doesn't have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a 'wait' loop until more jobs are entered. Such a program doesn't terminate unless the system crashes.

Given a problem to solve, the design phase produces an algorithm, and the implementation phase then generates a program that expresses the designed algorithm. So, the concrete expression of an algorithm in a particular programming language is called a program.

1.2 Algorithm Specification

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimised simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

1.3 Analysis Framework

The following steps are involved in solving computational problems:

- Problem definition
- Development of a model
- Specification of an algorithm
- Designing an algorithm
- Checking the correctness of an algorithm
- Analysis of an algorithm
- Implementation of an algorithm
- Program testing
- Documentation

Algorithm analysis 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 inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources require to execute it.

Usually, 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**.

1.3.1 The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms. By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm. Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analysing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**, bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment where memory is very limited.

1.3.2 Characteristics of Algorithms

The main characteristics of algorithms are as follows:

- Algorithms must have a unique name.
- Algorithms should have explicitly defined set of inputs and outputs.
- Algorithms are well ordered with unambiguous operations.

- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point.

The algorithm insertion-sort could be described in a more realistic way.

```
for i <- 1 to length(A)
    x <- A[i]
    j <- i
    while j > 0 and A[j-1] > x
        A[j] <- A[j-1]
        j <- j - 1
    A[j] <- x
```

In this section, algorithms will be presented in the form of pseudocode that is similar in many respects to C, C++, Java, Python, and other programming languages.

1.4 Performance Analysis

In this section, we will discuss the complexity of computational problems with respect to the amount of space an algorithm requires. Space complexity shares many of the features of time complexity and serves as a further way of classifying problems according to their computational difficulties.

1.4.1 Space Complexity and Time Complexity

What is Space Complexity?

Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. We often speak of **extra memory** needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed length) units to measure this.

We can use bytes, but it's easier to use, say, the number of integers used, the number of fixed sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes an important issue as time complexity.



1.4.2 Time Complexity of an Algorithm

The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm. This calculation is totally independent of implementation and programming language.

1. **Time complexity of a simple loop when the loop variable is incremented or decremented by a constant amount:**

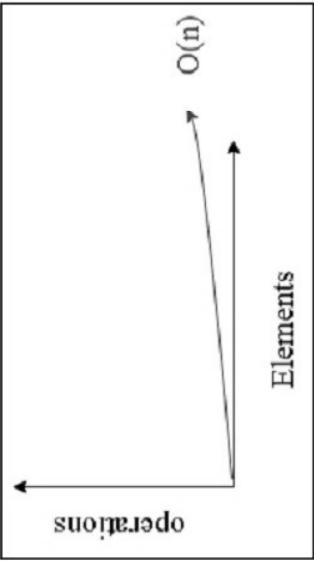


Fig. 1.1: Time Complexity of a Simple Loop

In the above scenario, loop is executed ‘n’ times. Therefore, time complexity of this loop is $O(n)$.

2. **Time complexity of a loop when the loop variable is divided or multiplied by a constant amount:**

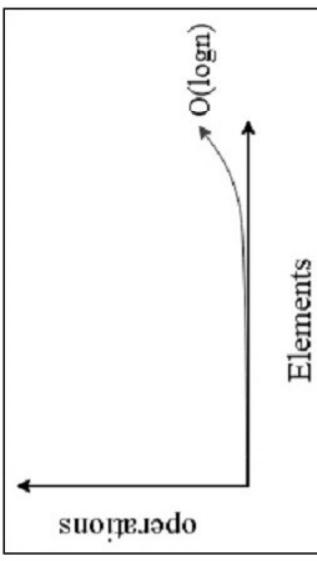


Fig. 1.2: Time Complexity of a Loop when the Loop Variable is Divided or Multiplied by a Constant Amount

In this case, time complexity is $O(\log n)$.

3. Time complexity of a nested loop:

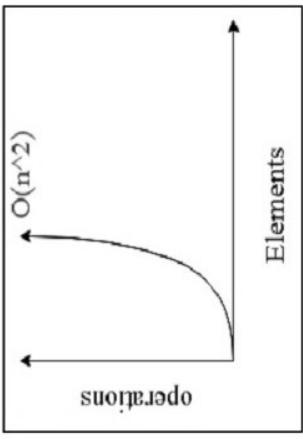


Fig. 1.3: Time Complexity of a Nested Loop

Time complexity of different loops is equal to the sum of the complexities of individual loop.

Therefore, Time complexity = $O(m) + O(n)$

Relationship among Complexity Classes

The following diagram depicts the relationship among different complexity classes:

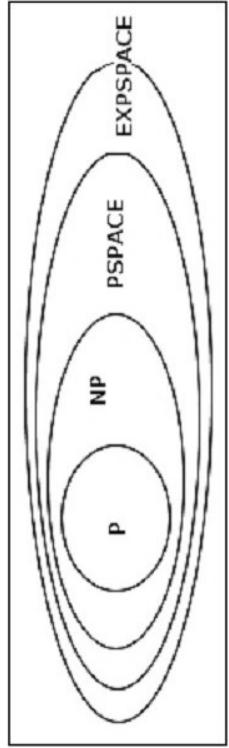


Fig. 1.4: Relationship among Different Complexity Classes

Till now, we have not discussed P and NP classes in this section. These will be discussed later.

1.5 Summary

An algorithm can be defined as a well-defined computational procedure that takes some values, or the set of values, as an input and produces some value, or the set of values, as an output. An algorithm is, thus, a sequence of computational steps that transform the input into output. It describes specific computational procedures for achieving the input-output relationship. It is very convenient to classify algorithm based on the relative amount of time or relative amount of space they require and specify the growth of time/space requirement as a function of input size.

Time Complexity: Running time of a program as a function of the size of the input.

Space Complexity: Some forms of analysis could be done based on how much space an algorithm needs to complete its task. This space complexity analysis was critical in the early days of computing when storage space on the computer was limited. When considering this algorithm are divided into those that need extra space to do their work and those that work in place.

But, nowadays, problem of space rarely occurs because space on the computer (internal or external) is enough.

Broadly, we achieve the following types of analysis:

Worst-case: $f(n)$ defined by the maximum number of steps taken on any instance of size n .

Best-case: $f(n)$ defined by the minimum number of steps taken on any instance of size n .

Average case: $f(n)$ defined by the average number of steps taken on any instance of size n .

An algorithm must have the following properties:

Correctness: It should produce the output according to the requirement of the algorithm.

Finiteness: Algorithm must complete after a finite number of instructions have been executed.

An Absence of Ambiguity: Each step must be defined, having only one interpretation.

Definition of Sequence: Each step must have a unique defined preceding and succeeding step. The first step and the last step must be noted.

Input/output: Number and classification of needed inputs and results must be stated.

Feasibility: It must be feasible to execute each instruction.

Flexibility: It should also be possible to make changes in the algorithm without putting so much effort on it.

Efficient: Efficiency is always measured in terms of time and space required for implementing the algorithm, so the algorithm uses a little running time and memory space as possible within the limits of acceptable development time.

Independent: An algorithm should focus on what are inputs, outputs and how to derive output without knowing the language it is defined. Therefore, we can say that the algorithm is independent of language.

1.6 Key Words/Abbreviations

- **Worst-case** – The maximum number of steps taken on any instance of size a.
- **Best-case** – The minimum number of steps taken on any instance of size a.
- **Average case** – An average number of steps taken on any instance of size a.
- **Amortised** – A sequence of operations applied to the input of size a averaged over time.

1.7 Learning Activity

1. What is the time complexity for the following C module? Assume that $n > 0$.

```
int module(int n)
```

```
{  
    if (n == 1)  
        return 1;  
    else  
        return (n + module(n-1));  
}
```

2. The time complexity of the following C function is (assume $n > 0$):

```
int recursive (int n)  
{  
    if (n == 1)  
        return 1;  
    else  
        return (recursive(n-1) + recursive(n-1));  
}
```

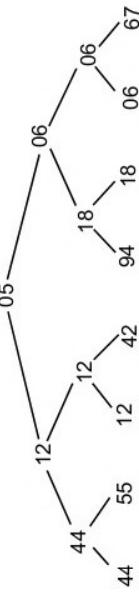
```
        else  
            return (recursive (n-1) + recursive (n-1));  
    }  
}
```

3. If one uses straight two-way merge sort algorithm to sort the following elements in ascending order:

20, 47, 15, 8, 9, 4, 40, 30, 12, 17

then what is the order of these elements after second pass of the algorithm?

4. The figure below represent a _____ sort



5. Given the array: 12,40,3,2 and intermediate states of the array while sorting

Stage (i) 12,3,40,2

Stage (ii) 12,3,2,40

Stage (iii) 3,12,2,40

Stage (iv) 3,2,12,40

Stage (v) 2,3,12,40

The array has been sorted using

6. You have to sort a list L, consisting of a sorted list followed by a few ‘random’ elements. Which of the sorting method would be most suitable for such a task?
- -----

1.8 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. Explain what is an algorithm in computing.
2. Explain what is time complexity of algorithm.
3. Mention what are the types of notation used for time complexity.
4. What is time complexity? Give example.
5. What is space complexity? Give example.
6. Explain what is a recursive algorithm.
7. Explain what is the difference between best-case scenario and worst-case scenario of an algorithm.
8. Define performance analysis.
9. Mention what are the three laws of recursion algorithm.
10. Explain what is space complexity of insertion sort algorithm.

B. Multiple Choice/Objective Type Questions

1. The word _____ comes from the name of a Persian mathematician Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.
 - (a) Flowchart
 - (b) Flow
 - (c) Algorithm
 - (d) Syntax
2. In computer science, algorithm refers to a special method usable by a computer for the solution to a problem.
 - (a) True
 - (b) False

3. This characteristic often draws the line between what is feasible and what is impossible.
 - (a) Performance
 - (b) System evaluation
 - (c) Modularity
 - (d) Reliability
4. The time that depends on the input: an already sorted sequence that is easier to sort.
 - (a) Process
 - (b) Evaluation
 - (c) Running
 - (d) Input

5. Which of the following is incorrect?

Algorithms can be represented:

- (a) As pseudocodes
 - (b) As syntax
 - (c) As programs
 - (d) As flowcharts
6. When an algorithm is written in the form of a programming language, it becomes a _____
 - (a) Flowchart
 - (b) Program
 - (c) Pseudocode
 - (d) Syntax
 7. Any algorithm is a program.
 - (a) True
 - (b) False
 8. A system wherein items are added from one end and removed from the other end.
 - (a) Stack
 - (b) Queue
 - (c) Linked list
 - (d) Array
 9. Another name for 1-D arrays.
 - (a) Linear arrays
 - (b) Lists
 - (c) Horizontal array
 - (d) Vertical array
 10. A data structure that follows the FIFO principle.
 - (a) Queue
 - (b) LL
 - (c) Stack
 - (d) Union

Answer:

1. (c), 2. (a), 3. (a), 4. (c), 5 (b), 6. (b), 7. (b), 8. (b), 9. (a), 10. (a)

1.9 References

References of this unit have been given at the end of the book.

UNIT 2 ASYMPTOTIC NOTATIONS

Structure:

- 2.0 Learning Objectives
- 2.1 Introduction
- 2.2 Asymptotic Notations
- 2.3 Mathematical Analysis of Non-recursive and Recursive Algorithms with Examples
- 2.4 Summary
- 2.5 Key Words/Abbreviations
- 2.6 Learning Activity
- 2.7 Unit End Questions (MCQ and Descriptive)
- 2.8 References

2.0 Learning Objectives

After studying this unit, you will be able to:

- Explain asymptotic notations
- Exemplify mathematical analysis of non-recursive and recursive algorithms
- Describe Big-O notation (O)
- Describe Omega notation (Ω)
- Implement Theta notation (Θ)
- Elaborate little-o notation (o)



2.1 Introduction

The word asymptotic means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Asymptotic Analysis

It is a technique of representing limiting behaviour. The methodology has the applications across science. It can be used to analyse the performance of an algorithm for some large data set.

1. In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input data sets.

The simplest example is a function $f(n) = n^2 + 3n$, the term $3n$ becomes insignificant compared to n^2 when n is very large. The function ' $f(n)$ ' is said to be **asymptotically equivalent** to n^2 as $n \rightarrow \infty$ ', and here is written symbolically as $f(n) \sim n^2$.

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as ‘best-case’ and ‘worst-case’ scenarios, respectively.

‘In asymptotic notations, we derive the complexity concerning the size of the input. (example in terms of n)’.

These notations are important because without expanding the cost of running the algorithm we can estimate the complexity of the algorithms.

Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm’s efficiency.
2. They allow the comparisons of the performances of various algorithms.

Asymptotic Notations

Asymptotic notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

1. **Big-O Notation:** Big-O is the formal method of expressing the upper bound of an algorithm’s running time. It is the measure of the longest amount of time. The



function $f(n) = O(g(n))$ [read as ‘ f of n is big- o of g of n'] if and only if exists positive constant c and such that

$$f(n) \leq k \cdot g(n) \leq k \cdot g(n) \text{ for } n > n_0 > n_0 \text{ in all case}$$

Hence, function $g(n)$ is an upper bound for function $f(n)$, as $g(n)$ grows faster than $f(n)$

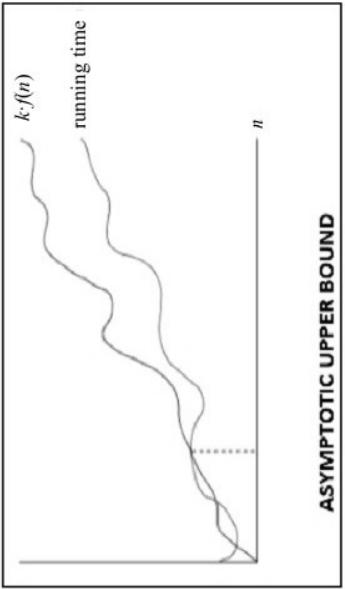


Fig. 2.1: Asymptotic Upper Bound

For example:

1. $3n+2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$
2. $3n+3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$

Hence, the complexity of $f(n)$ can be represented as $O(g(n))$.

- 2. Omega (Ω) Notation:** The function $f(n) = \Omega(g(n))$ [read as ‘ f of n is omega of g of n'] if and only if there exists positive constant c and n_0 such that
- $$F(n) \geq k^* g(n) \text{ for all } n, n \geq n_0$$

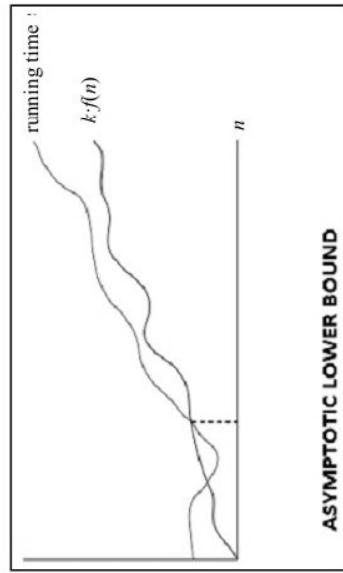


Fig. 2.2: Asymptotic Lower Bound



For example:

$$\begin{aligned} f(n) &= 8n_2 + 2n - 3 \geq 8n_2 - 3 \\ &= 7n_2 + (n_2 - 3) \geq 7n_2 \quad (g(n)) \end{aligned}$$

Thus, $k_1 = 7$

Hence, the complexity of $f(n)$ can be represented as $\Omega(g(n))$

- 3. Theta (Θ):** The function $f(n) = \theta(g(n))$ [read as ‘f is the theta of g of n’] if and only if there exists positive constant k_1, k_2 and n_0 such that

$$k_1 * g(n) \leq f(n) \leq k_2 * g(n) \text{ for all } n, n \geq n_0$$

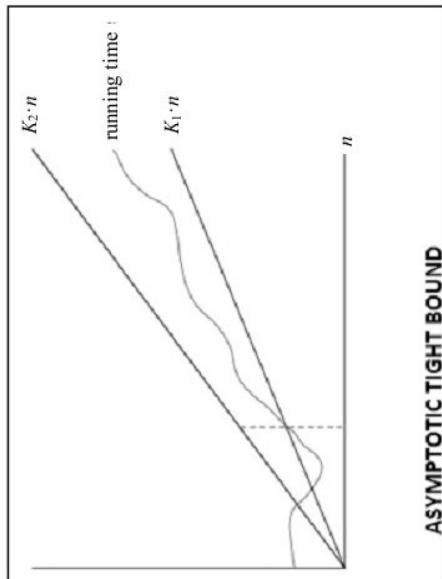


Fig. 2.3: Asymptotic Tight Bound

For example:

$$3n+2 = \theta(n) \text{ as } 3n+2 \geq 3n \text{ and } 3n+2 \leq 4n, \text{ for } n$$

$$k_1 = 3, k_2 = 4, \text{ and } n_0 = 2$$

Hence, the complexity of $f(n)$ can be represented as $\theta(g(n))$.

The Theta notation is more precise than both the Big-O and Omega notation. The function $f(n) = \theta(g(n))$ if $g(n)$ is both an upper and lower bound.

Analysing Algorithm Control Structure

To analyse a programming code or algorithm, we must notice that each instruction affects the overall performance of the algorithm, and therefore each instruction must be analysed separately

to analyse overall performance. However, there are some algorithm control structures which are present in each programming code and have a specific asymptotic analysis.

Some Algorithm Control Structures are:

1. Sequencing
2. If-then-else
3. for loop
4. While loop

2.2 Asymptotic Notations

In designing of algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works. The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of memory required to process the data and the processing time.

Complexity of an algorithm is analysed in two perspectives: **Time and Space**.

Time Complexity

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. ‘Time’ can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

Space Complexity

It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of ‘extra’ memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed length) units to measure this. Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by $T(n)$, where n is the input size.

Different types of asymptotic notations are used to represent the complexity of an algorithm.

Following asymptotic notations are used to calculate the running time complexity of an algorithm:

- O – Big-O
- Ω – Big Omega
- Θ – Big Theta
- o – Little-o
- ω – Little-omega

Types of Functions, Limits, and Simplification

- Logarithmic Function - $\log n$
- Linear Function - $an + b$
- Quadratic Function - $an^2 + bn + c$
- Polynomial Function - $an^z + \dots + an^2 + a*n^1 + a*n^0$, where z is some constant
- Exponential Function - a^n , where a is some constant

These are some basic function growth classifications used in various notations. The list starts at the slowest growing function (logarithmic, fastest execution time) and goes on to the fastest growing (exponential, slowest execution time). Notice that as ' n ', or the input, increases in each of those functions, the result clearly increases much quicker in quadratic, polynomial, and exponential, compared to logarithmic and linear.

One extremely important note is that for the notations about to be discussed you should do your best to use simplest terms. This means to disregard constants, and lower order terms, because as the input size (or n in our $f(n)$ example) increases to infinity (mathematical limits), the lower order terms and constants are of little to no importance. That being said, if you have

constants that are 2^{9001} , or some other ridiculous, unimaginable amount, realise that simplifying will skew your notation accuracy.

Since we want simplest form, lets modify our table a bit...

- Logarithmic - $\log n$
- Linear - n
- Quadratic - n^2
- Polynomial - n^z , where z is some constant
- Exponential - a^n , where a is some constant

2.3 Mathematical Analysis of Non-recursive and Recursive Algorithms with Examples

The complexity of an algorithm is often analysed to estimate the resources it will demand given a specific execution. These resources can basically be expressed in terms of execution time (known as **time complexity**) and memory needs (known as **space complexity**).

When you have a non-recursive algorithm the complexity analysis is simply the analysis of its iterations (basically loops), but when you have a recursive algorithm you have to pay attention to how the recursion evolves. Therefore, for this lesson, it is assumed that you are familiar with the concept of iterative complexity. Nevertheless, it is worth to remind that the **Big-O** notation stands for the asymptotic value of an expression, i.e., Big-O of an expression means the upper bound of that expression considering only what really matters in its numerical computation. For example, the example below states a complex expression, and its Big-O representation.

$$\frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n \rightarrow O(n^3)$$

In this example, what really gives the order of magnitude for the expression is the n cubed, i.e., it does not really matter the coefficient of the n cubed, and even less the lesser parts of the equation (n squared and n).

Time Complexity of Recursive Algorithms

The time complexity of an algorithm is basically how many operations are necessary to compute it. While for non-recursive algorithms the reasoning is quite straightforward; for recursive algorithms, it is a little trickier, but still easy to follow.

Let's take a simple example, the recursive algorithm to compute the factorial of a natural number n as in the C language code below.

```
1. int Factorial(int n) {  
2.     if(n == 0)  
3.         return 1;  
4.     else  
5.         return n*Factorial(n-1);  
6. }
```

First, we compute how many operations are there in a basic call, in this example, it will be three operations:

- one for the *if* condition (is n equal to 0?)
- one for the product of n by the output of the next call of Factorial()
- one to subtract 1 from n in the

Space Complexity of Recursive Algorithms

To compute the space complexity it is important to observe not the operations, but how much memory is needed in each program execution. Using the factorial example, in each call of Factorial(), it is necessary a fixed amount k of memory. The amount of memory needed would be directly proportional to the number of calls that the function makes, and since for values of $n > 0$ this function calls Factorial() just once for $n - 1$. The space complexity $S(n)$ is expressed as:

$$S(n) = kn \rightarrow O(n)$$

Since k is a constant value (a fixed amount of memory for each function call), this complexity is once again a direct function of n , i.e., in terms of space complexity the recursive factorial program is also **linear**.

EXAMPLE 1: Compute the factorial function $F(\mathbf{n}) = \mathbf{n}!$ for an arbitrary non-negative integer \mathbf{n} .

Since

$$\mathbf{n}! = 1 \cdot \dots \cdot (\mathbf{n} - 1) \cdot \mathbf{n} = (\mathbf{n} - 1)! \cdot \mathbf{n} \quad \text{for } \mathbf{n} \geq 1$$

and $0! = 1$ by definition, we can compute $F(\mathbf{n}) = F(\mathbf{n} - 1) \cdot \mathbf{n}$ with the following recursive algorithm.

ALGORITHM $F(\mathbf{n})$

//Computes $\mathbf{n}!$ recursively //Input: A nonnegative integer \mathbf{n} //Output: The value of $\mathbf{n}!$

```
if  $\mathbf{n} = 0$  return 1
```

```
else return  $F(\mathbf{n} - 1) * \mathbf{n}$ 
```

For simplicity, we consider \mathbf{n} itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication, whose number of executions we denote as $M(\mathbf{n})$. Since the function $F(\mathbf{n})$ is computed according to the formula

$$F(\mathbf{n}) = F(\mathbf{n} - 1) \cdot \mathbf{n} \quad \text{for } \mathbf{n} > 0,$$

the number of multiplications $M(\mathbf{n})$ needed to compute it must satisfy the equality

$$M(n) = M(n-1) + \begin{cases} 1 & \text{to multiply} \\ \text{to compute } F(n-1) \text{ by } n & \end{cases} \quad \text{for } \mathbf{n} > 0.$$

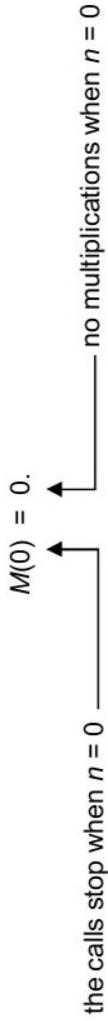
Indeed, $M(\mathbf{n}-1)$ multiplications are spent to compute $F(\mathbf{n}-1)$, and one more multiplication is needed to multiply the result by n .

The last equation defines the sequence $\mathbf{M}(\mathbf{n})$ that we need to find. This equation defines $\mathbf{M}(\mathbf{n})$ not explicitly, i.e., as a function of \mathbf{n} , but implicitly as a function of its value at another point, namely $\mathbf{n}-1$. Such equations are called *recurrence relations* or, for brevity, *recurrences*. Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics. They are usually studied in detail in courses on discrete mathematics or discrete structures; a very brief tutorial on them is provided in Appendix B. Our goal now is to solve the recurrence relation $\mathbf{M}(\mathbf{n}) = \mathbf{M}(\mathbf{n}-1) + 1$, i.e., to find an explicit formula for $\mathbf{M}(\mathbf{n})$ in terms of \mathbf{n} only.

Note, however, that there is not one but infinitely many sequences that satisfy this recurrence. (Can you give examples of, say, two of them?) To determine a solution uniquely, we need an *initial condition* that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

```
if  $\mathbf{n} = 0$  return 1.
```

This tells us two things. First, since the calls stop when $\mathbf{n} = 0$, the smallest value of \mathbf{n} for which this algorithm is executed and hence $\mathbf{M}(\mathbf{n})$ defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when $\mathbf{n} = 0$, the algorithm performs no multiplications. Therefore, the initial condition we are after is



Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{for } n > 0, \\ M(0) &= 0. \end{aligned} \tag{2.1}$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function $F(n)$ itself; it is defined by the recurrence

$$\begin{aligned} F(n) &= F(n-1) \cdot n && \text{for every } n > 0. \\ F(n) &= 0 \end{aligned}$$

The second is the number of multiplications $M(n)$ needed to compute $F(n)$ by the recursive algorithm whose pseudocode was given at the beginning of the section.

As we just showed, $M(n)$ is defined by recurrence (2.1). And it is recurrence (2.1) that we need to solve now.

Though it is not difficult to 'guess' the solution here (what sequence starts with 0 when $n = 0$ and increases by 1 on each step), it will be more useful to arrive at it in a systematic fashion. From the several techniques available for solving recurrence relations, we use what can be called the **method of backward substitutions**. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3. \end{aligned}$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern:

$M(n) = M(n - i) + i$. Strictly speaking, the correctness of this formula should be proved by mathematical induction. But, it is easier to get to the solution as follows and then verify its correctness.

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n - 1) + 1 = \dots = M(n - i) + i = \dots = M(n - n) + n = n.$$

You should not be disappointed after exerting so much effort to get this ‘obvious’ answer. The benefits of the method illustrated in this simple example will become clear very soon, when we have to solve more difficult recurrences. Also, note that the simple iterative algorithm that accumulates the product of n consecutive integers requires the same number of multiplications, and it does so without the overhead of time and space used for maintaining the recursion’s stack.

The issue of time efficiency is actually not that important for the problem of computing $n!$, however. As we saw in Section 2.1, the function’s values get so large so fast that we can realistically compute exact values of $n!$ only for very small n ’s. Again, we use this example just as a simple and convenient vehicle to introduce the standard approach to analysing recursive algorithms.

Generalising our experience with investigating the recursive algorithm for computing $n!$, we can now outline a general plan for investigating recursive algorithms.

General Plan for Analysing the Time Efficiency of Recursive Algorithms

Decide on a parameter (or parameters) indicating an input’s size.

Identify the algorithm’s basic operation.

Check whether the number of times the basic operation is executed can vary on different inputs of the same size. If it can, the worst-case, average case, and best-case efficiencies must be investigated separately.

Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

Solve the recurrence or, at least, ascertain the order of growth of its solution.

Mathematical Analysis of Non-recursive Algorithms

In this section, we systematically apply the general framework outlined in Section 2.1 of analysing the time efficiency of non-recursive algorithms. Let us start with a very simple example that demonstrates all the principal steps typically taken in analysing such algorithms.

EXAMPLE 1: Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem:

```
ALGORITHM MaxElement( $A[0..n - 1]$ )
```

```
//Determines the value of the largest element in a given array
```

```
//Input: An array  $A[0..n - 1]$  of real numbers
```

```
//Output: The value of the largest element in  $A$  maxval  $\leftarrow A[0]$ 
```

```
for  $i \leftarrow 1$  to  $n - 1$  do
```

```
    if  $A[i] > maxval$  maxval  $\leftarrow A[i]$ 
```

```
    return maxval
```

The obvious measure of an input's size here is the number of elements in the array, i.e., n . The operations that are going to be executed most often are in the algorithm's **for** loop. There are two operations in the loop's body: the comparison $A[i] > maxval$ and the assignment $maxval \leftarrow A[i]$. Which of these two operations should we consider basic? Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. Note that the number of comparisons will be the same for all arrays of size n ; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.

Let us denote $C(n)$ the number of times this comparison is executed and try to find a formula expressing it as a function of size n . The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and $n - 1$, inclusive. Therefore, we get the following sum for $C(n)$:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated $n - 1$ times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Here is a general plan to follow in analysing non-recursive algorithms.

General Plan for Analysing the Time Efficiency of Non-recursive Algorithms

Decide on a parameter (or parameters) indicating an input's size.

Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)

Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average case, and, if necessary, best-case efficiencies have to be investigated separately.

Set up a sum expressing the number of times the algorithm's basic operation is executed.

Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

Before proceeding with further examples, you may want to review Appendix A, which contains a list of summation formulas and rules that are often useful in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation.

$$\sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i, \quad (\text{R1})$$

$$\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i, \quad (\text{R2})$$

and two summation formulas

$$\sum_{i=l}^u 1 = u - l + 1 \text{ where } l \leq u \text{ are some lower and upper integer limits,} \quad (\text{S1})$$



$$\sum_{i=0}^n i = \sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \quad (\text{S2})$$

Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.

EXAMPLE 2: Consider the *element uniqueness problem*. Check whether all the elements in a given array of n elements are distinct. This problem can be solved by the following straightforward algorithm:

ALGORITHM *UniqueElements ($A[0..n - 1]$)*

```
//Determines whether all the elements in a given array are distinct //Input: An array  $A[0..n - 1]$ 
//Output: Returns ‘true’ if all the elements in  $A$  are distinct // and ‘false’ otherwise
for  $i \leftarrow 0$  to  $n - 2$  do
    for  $j \leftarrow i + 1$  to  $n - 1$  do
        if  $A[i] = A[j]$  return false return true
```

The natural measure of the input’s size here is again n , the number of elements in the array. Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm’s basic operation. Note, however, that the number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

By definition, the worst-case input is an array for which the number of element comparisons $C_{\text{worst}}(n)$ is the largest among all arrays of size n . An inspection of the innermost loop reveals that there are two kinds of worst-case inputs — inputs for which the algorithm does not exit the loop prematurely: arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits $i + 1$ and $n - 1$; this is



repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and $n - 2$. Accordingly, we get

$$\begin{aligned} C_{\text{worst}}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2). \end{aligned}$$

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \dots + 1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2). Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all $n(n - 1)/2$ distinct pairs of its n elements.

2.4 Summary

Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

Big-O Notation: Big-O (O) notation gives an upper bound for a function $f(n)$ to within a constant factor.

Big-Omega Notation: Big-Omega (Ω) notation gives a lower bound for a function $f(n)$ to within a constant factor.

Big-Theta Notation: Big-Theta (Θ) notation gives bound for a function $f(n)$ to within a constant factor.

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine-specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

2.5 Key Words/Abbreviations

- O – Big-O
- Ω – Big-Omega
- θ – Big-Theta
- o – Little-o
- ω – Little-omega

2.6 Learning Activity

1. What does asymptotically mean in layman terms?

2. What is an asymptotic notation?

3. What is an asymptotic complexity?

4. Why an asymptotic notation is called 'asymptotic'?

5. What is an asymptotic notation in an algorithm?

6. How can I understand asymptotic notations?

7. What are the asymptotic notations for the analysis of algorithms?

8. What is the term asymptotic?

9. How can the BNF notation be explained with examples?

10. Why do we use an asymptotic notation?

11. What is a recurrence relation? How is it related to an asymptotic notation?

12. What are some asymptotic functions?

13. Which asymptotic notation is most frequently used for algorithms and why?

14. What are explanations on asymptotic notations with examples of algorithms?

2.7 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. What is the significance of an asymptotic notation?
 2. What is an asymptotic notation data structure?
 3. What is Big-O notation and why is it useful?
 4. What is Big-O notation? Explain by giving some examples.
 5. What are non-recursive and recursive algorithms.
 6. Define omega notation.
 7. What is meant by theta notation? Give example.
 8. What is little-o notation? Give example.

B. Multiple Choice/Objective Type Questions

1. If $f(x) = (x^3 - 1) / (3x + 1)$, then $f(x)$ is:
 - (a) $O(x^2)$
 - (b) $O(x)$
 - (c) $O(x^2 / 3)$
 - (d) $O(1)$
 2. If $f(x) = 3x^2 + x^3 \log x$, then $f(x)$ is:
 - (a) $O(x^2)$
 - (b) $O(x^3)$
 - (c) $O(x)$
 - (d) $O(1)$
 3. The Big-O notation for $f(n) = (n \log n + n^2)(n^3 + 2)$ is:
 - (a) $O(n^2)$
 - (b) $O(3^n)$
 - (c) $O(n^4)$
 - (d) $O(n^5)$

4. The Big-Theta notation for function $f(n) = 2n^3 + n - 1$ is:

- (a) n
- (b) n^2
- (c) n^3
- (d) n^4

5. The Big-Theta notation for $f(n) = n\log(n^2 + 1) + n^2\log n$ is:

- (a) $n^2\log n$
- (b) n^2
- (c) $\log n$
- (d) $n\log(n^2)$

6. The Big-Omega notation for $f(x, y) = x^5y^3 + x^4y^4 + x^3y^5$ is:

- (a) x^5y^3
- (b) x^5y^5
- (c) x^3y^3
- (d) x^4y^4

7. If $f_1(x)$ is $O(g(x))$ and $f_2(x)$ is $o(g(x))$, then $f_1(x) + f_2(x)$ is:

- (a) $O(g(x))$
- (b) $o(g(x))$
- (c) $O(g(x)) + o(g(x))$
- (d) None of the mentioned

8. The little-o notation for $f(x) = x\log x$ is:

- (a) x
- (b) x^3
- (c) x^2
- (d) $x\log x$

9. The Big-O notation for $f(n) = 2\log(n!) + (n^2 + 1)\log n$ is:

- (a) n
- (b) n^2
- (c) $n\log n$
- (d) $n^2\log n$

10. The Big-O notation for $f(x) = 5\log x$ is:

- (a) 1
- (b) x
- (c) x^2
- (d) x^3

11. The Big-Omega notation for $f(x) = 2x^4 + x^2 - 4$ is:

- (a) x^2
- (b) x^3
- (c) x
- (d) x^4

Answers:

1. (a), 2. (b), 3. (d), 4. (c), 5. (a), 6. (c), 7. (a), 8. (c), 9. (d), 10. (b), 11. (d)

2.8 References

References of this unit have been given at the end of the book.



UNIT 3 IMPORTANT PROBLEM TYPES

Structure:

- 3.0 Learning Objectives
- 3.1 Introduction
- 3.2 Sorting Algorithms
- 3.3 Searching
- 3.4 String Processing
- 3.5 Graph Problems
- 3.6 Combinatorial Problems
- 3.7 Summary
- 3.8 Key Words/Abbreviations
- 3.9 Learning Activity
- 3.10 Unit End Questions (MCQ and Descriptive)
- 3.11 References

3.0 Learning Objectives

After studying this unit, you will be able to:

- Explain sorting algorithm in practical way
- Describe the concept and implementation of all searching algorithm
- Define and discuss graph problems



- Describe string processing concept
- Describe the theory of combinatorial problems

3.1 Introduction

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimised to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios:

- 1. Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- 2. Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

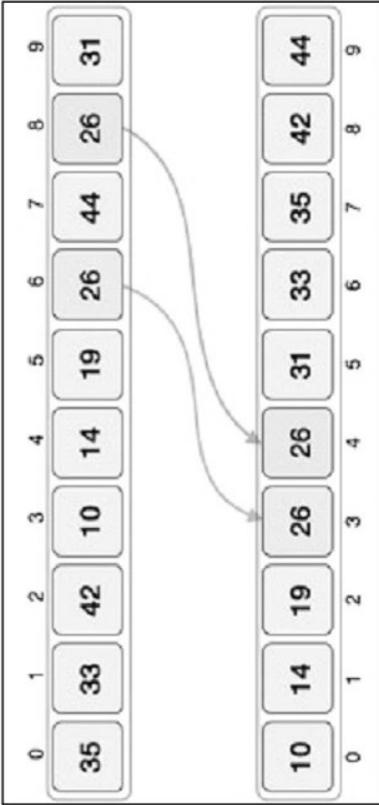
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of **in-place sorting**.

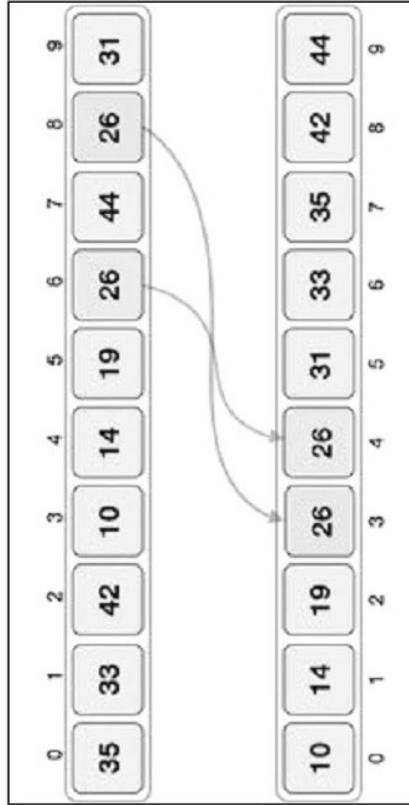
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge sort is an example of **not-in-place sorting**.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple, for example.

Adaptive and Non-adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive if it takes advantage of already ‘sorted’ elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to reorder them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be reordered to confirm their sortedness.

Important Terms

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them –

Increasing Order

A sequence of values is said to be in **increasing order**, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Decreasing Order

A sequence of values is said to be in **decreasing order**, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Non-increasing Order

A sequence of values is said to be in **non-increasing order**, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

Non-decreasing Order

A sequence of values is said to be in **non-decreasing order**, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Table 3.1: Comparisons of Different Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case
Bubble	$O(n22)$	$O(n22)$	$O(n22)$
Selection	$O(n22)$	$O(n22)$	$O(n22)$

Insertion	O(n)	O(n22)	O(n22)
Quick	O(n log n)	O(n log n)	O(n22)
Merge	O(n log n)	O(n log n)	O(n log n)
Heap	O(n log n)	O(n log n)	O(n log n)
Radix	$\Omega(nk)$	$\theta(nk)$	O(nk)

The searching algorithms are used to search or find one or more than one element from a dataset. These type of algorithms are used to find elements from a specific data structures.

Searching may be sequential or not. If the data in the dataset are random, then we need to use sequential searching. Otherwise, we can use other different techniques to reduce the complexity.

3.2 Sorting Algorithms

Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. For example, consider an array $A = \{A1, A2, A3, A4, ?An\}$, the array is called to be in ascending order if element of A are arranged like:

$A1 > A2 > A3 > A4 > A5 > ? > An$.

Consider an array;

```
int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9 }
```

The array sorted in ascending order will be given as;

```
A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }
```

There are many techniques by using which sorting can be performed. In this section of the tutorial, we will discuss each method in detail.

Sorting algorithms are described in the following table along with the description:

Table 3.2: Sorting Algorithms

SN	Sorting Algorithms	Description
1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
2	Bucket Sort	Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. In this sorting algorithms, buckets are sorted individually by using different sorting algorithm.
3	Comb Sort	Comb sort is the advanced form of bubble sort. Bubble sort compares all the adjacent values, while comb sort removes all the turtle values or small values near the end of the list.
4	Counting Sort	It is a sorting technique based on the keys, i.e., objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning it to objects.
5	Heap Sort	In the heap sort, min-heap or max-heap is maintained from the array elements descending upon the choice and the elements are sorted by deleting the root element of the heap.
6	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
7	Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.

8	Quick Sort	Quick sort is the most optimised sort algorithm which performs sorting in $O(n \log n)$ comparisons. Like merge sort, quick sort also works by using divide and conquer approach.
9	Radix Sort	In radix sort, the sorting is done as we do sort the names according to their alphabetical order. It is the linear sorting algorithm used for integers.
10	Selection Sort	Selection sort finds the smallest element in the array and places it on the first place on the list, then it finds the second smallest element in the array and places it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time $O(n^2)$ which is worst than insertion sort.
11	Shell Sort	Shell sort is the generalisation of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

Bubble Sort

In bubble sort, each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires $n-1$ passes for sorting. Consider an array A of n elements whose elements are to be sorted by using bubble sort. The algorithm processes like following:

1. In Pass 1, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$, $A[2]$ is compared with $A[3]$, and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$, and so on. At the end of Pass 2, the second largest element of the list is placed at the second highest index of the list.
3. In pass $n-1$, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$, and so on. At the end of this pass, the smallest element of the list is placed at the first index of the list.

Algorithm

Step 1: Repeat Step 2 For i = 0 to N-1

Step 2: Repeat For J = i + 1 to N - 1

Step 3: IF A[J] > A[i]

SWAP A[J] and A[i]

[END OF INNER LOOP]

[END OF OUTER LOOP]

Step 4: EXIT

Table 3.3: Complexity of Bubble Sort

Scenario	Complexity
Space	$O(1)$
Worst Case Running Time	$O(n^2)$
Average Case Running Time	$O(n)$
Best-case Running Time	$O(n^2)$

Bucket Sort

Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. Buckets are sorted individually by using different sorting algorithm.

Complexity of Bucket Sort

Algorithm	Complexity
Space	$O(1)$
Worst Case	$O(n^2)$
Best Case	$\Omega(n+k)$
Average Case	$\theta(n+k)$

Algorithm**Step 1:** START**Step 2:** Set up an array of initially empty ‘buckets’.**Step 3:** Scatter: Go over the original array, putting each object in its bucket.**Step 4:** Sort each non-empty bucket.**Step 5:** Gather: Visit the buckets in order and put all elements back into the original array.**Step 6:** STOP**Comb Sort**

Comb sort is the advance form of bubble sort. Bubble sort compares all the adjacent values, while comb sort removes all the turtle values or small values near the end of the list.

Factors affecting comb sort are:

- It improves on bubble sort by using gap of size more than 1.
- Gap starts with large value and shrinks by the factor of 1.3.
- Gap shrinks till value reaches 1.

Table 3.4: Complexity of Comb Sort

Algorithm	Complexity
Worst-case Complexity	$O(n^2)$
Best-case Complexity	$\Theta(n \log n)$
Average Case Complexity	$\Omega(n^{2/2p})$ where p is number of increments.
Worst-case Space Complexity	$O(1)$

Algorithm

Step 1: START

Step 2: Calculate the gap value if gap value==1 go to step 5 else go to step 3.

Step 3: Iterate over data set and compare each item with gap item, then go to step 4.

Step 4: Swap the element if required else go to step 2.

Step 5: Print the sorted array.

Step 6: STOP

Counting Sort

It is a sorting technique based on the keys, i.e., objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning them to objects.

Complexity

Time Complexity: $O(n+k)$ is worst case where n is the number of element and k is the range of input.

Table 3.5: Space Complexity: $O(k)$ k is the Range of Input

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$\Omega(n+k)$	$\theta(n+k)$	$O(n+k)$
Space Complexity			$O(k)$

Limitation of Counting Sort

- It is effective when range is not greater than number of object.
- It is not comparison based complexity.
- It is also used as sub-algorithm for different algorithm.

- It uses partial hashing technique to count the occurrence.
- It is also used for negative inputs.

Algorithm**Step 1:** START**Step 2:** Store the input array.**Step 3:** Count the key values by number of occurrence of object.**Step 4:** Update the array by adding previous key elements and assigning to objects.**Step 5:** Sort by replacing the object into new array and key= key-1.**Step 6:** STOP

3.3 Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then location of the item is returned, otherwise the algorithm returns NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted.

The algorithm of linear search is given as follows:

Algorithm

LINEAR_SEARCH(A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1**Step 2:** [INITIALIZE] SET I = 1**Step 3:** Repeat Step 4 while I<=N**Step 4:** IF A[I] = VAL

SET POS = I

PRINT POS

Go to Step 6

[END OF IF]

SET I = I + 1

[END OF LOOP]

Step 5: IF POS = -1

PRINT " VALUE IS NOT PRESENTIN THE ARRAY "

[END OF IF]

Step 6: EXIT**Table 3.6: Complexity of Algorithm**

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

3.4 String Processing

Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus, a null-terminated string contains the characters that comprise the string followed by a null.



The following declaration and initialisation create a string consisting of the word ‘Hello’. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word ‘Hello’.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialisation, then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ –

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the ‘\0’ at the end of the string when it initialises the array. Let us try to print the above-mentioned string –

```
#include<stdio.h>
int main ()
{
    char greeting[6]={'H','e','l','l','o','\0'};
    printf("Greeting message: %s\n", greeting);
    return0;
}
```

When the above code is compiled and executed, it produces the following result:

Greeting message: Hello

Table 3.7: C supports a Wide Range of Functions that Manipulate Null-terminated Strings

Sr. No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

3.5 Graph Problems

Graph

A graph G can be defined as a group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (nodes) maintain any complex relationship among them instead of having a parent-child relationship.

Definition

A graph G can be defined as an ordered set $G(V, E)$ where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges which are used to connect these vertices.

A graph $G(V, E)$ with five vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure:



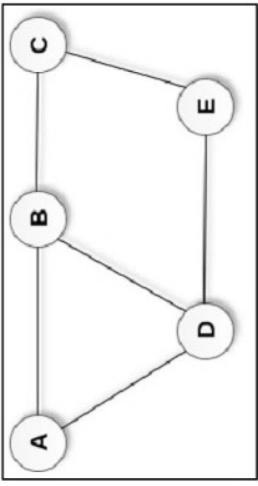


Fig. 3.1: Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B, then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node, while node B is called terminal node.

A directed graph is shown in the following figure:

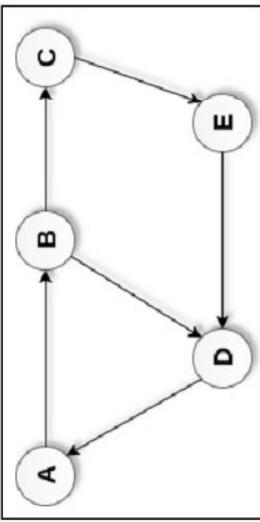


Fig. 3.2: Directed Graph

Graph Representation

By graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

There are two ways to store graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

1. Sequential Representation

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices will have a dimension $n \times n$.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1, if there exists an edge between V_i and V_j .

An undirected graph and its adjacency matrix representation is shown in the following figure:

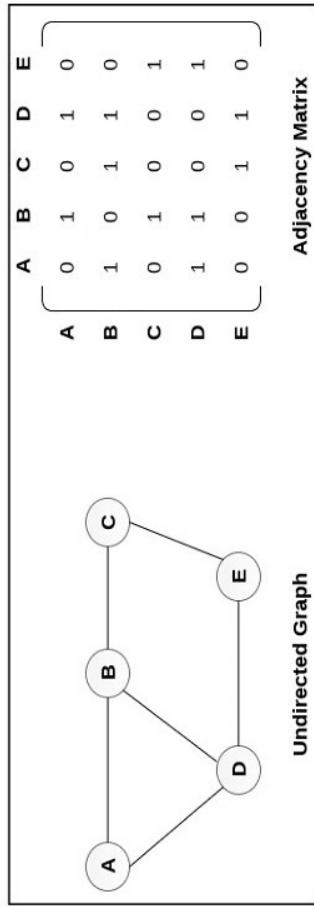


Fig. 3.3: Undirected Graph and its Adjacency Matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j .

A directed graph and its adjacency matrix representation is shown in the following figure:

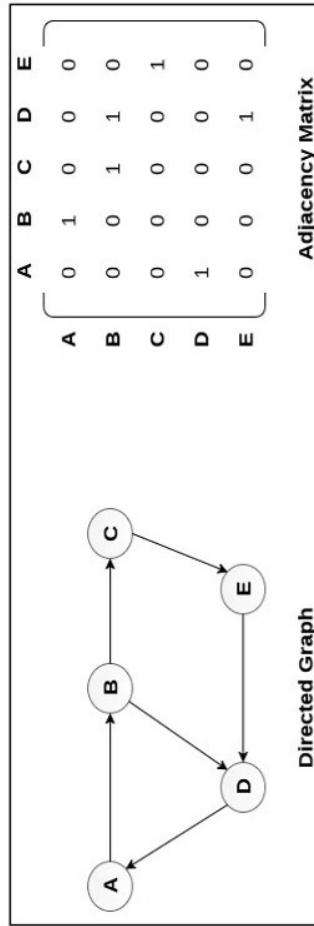


Fig. 3.4: Directed Graph and its Adjacency Matrix



Representation of weighted directed graph is different. Instead of filling the entry by 1, the nonzero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure:

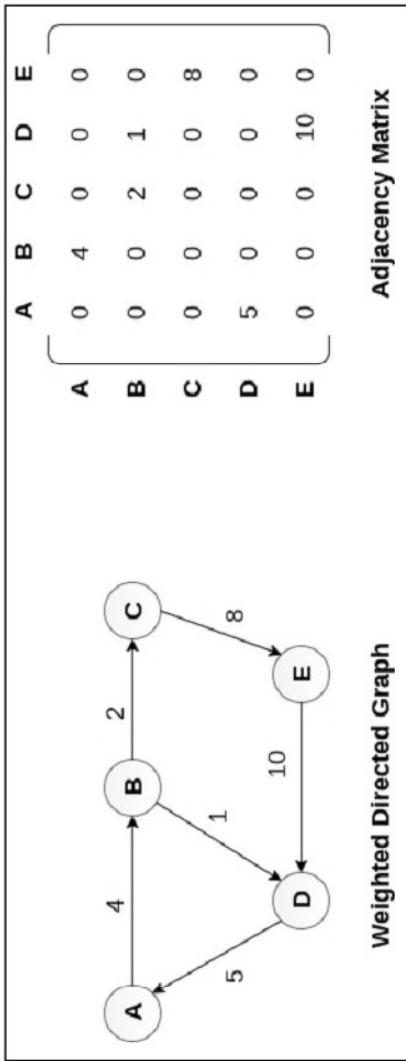


Fig. 3.5: Weighted Directed Graph along with the Adjacency Matrix

2. Linked Representation

In the linked representation, an adjacency list is used to store the graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.

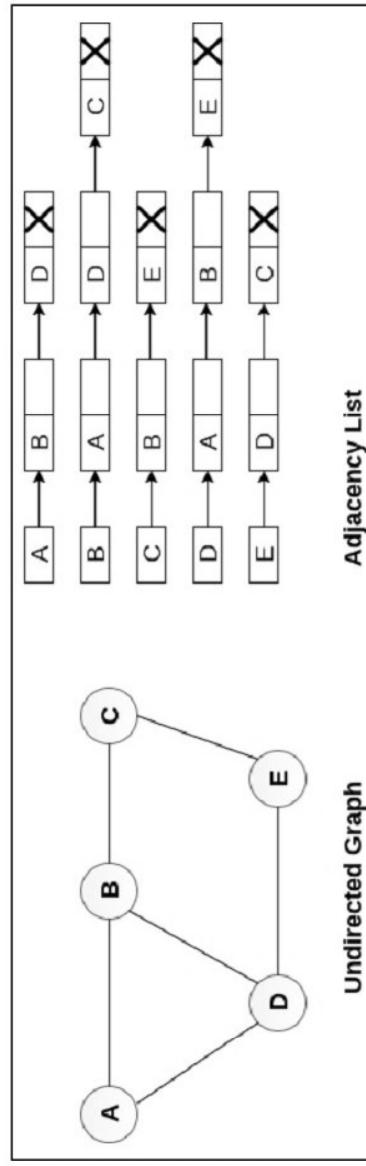


Fig. 3.6: Undirected Graph with Adjacency List

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph:

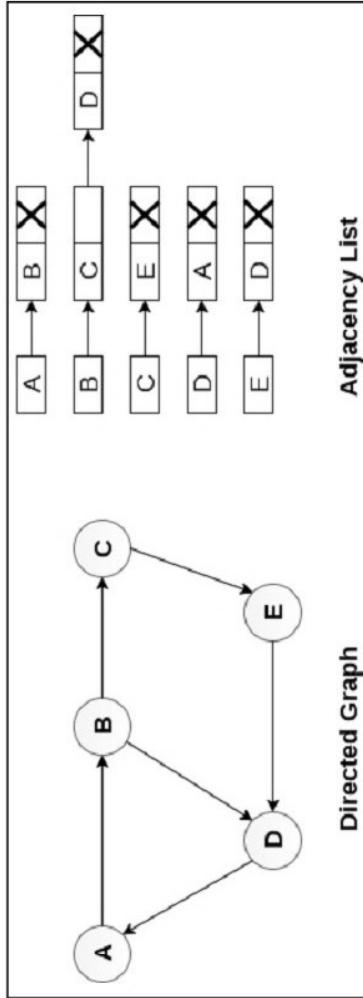


Fig. 3.7: Directed Graph with Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure:

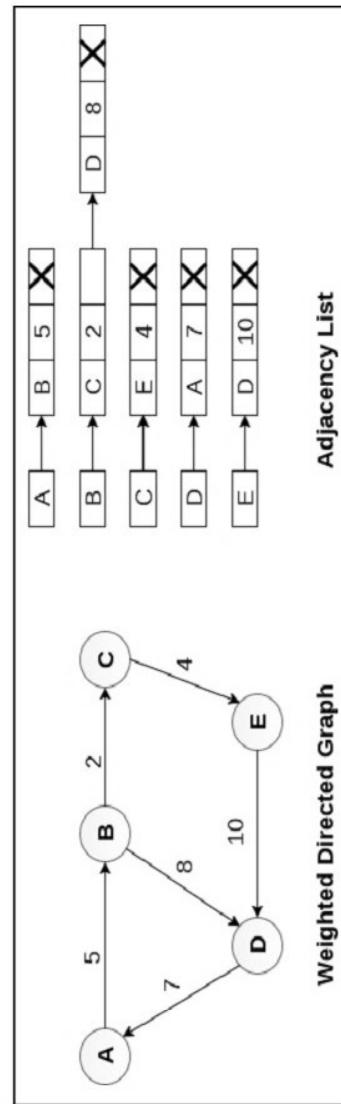


Fig. 3.8: Weighted Directed Graph with Adjacency List

3.6 Combinatorial Problems

In this section, we consider several classic algorithmic problems of a purely combinatorial nature. These include sorting and permutation generation, both of which were among the first non-numerical problems arising on electronic computers. Sorting, searching, and selection can all be classified in terms of operations on a partial order of keys. Sorting can be viewed as identifying or imposing the total order on the keys, while searching and selection involve identifying specific keys based on their position in the total order.

The rest of this section deals with other combinatorial objects, such as permutations, partitions, subsets, calendars, and schedules. We are particularly interested in algorithms that *rank* and *unrank* combinatorial objects, i.e., those which map each distinct object to and from a unique integer. Once we have rank and unrank operations, many other tasks become simple, such as generating random objects (pick a random number and unrank) or listing all objects in order (iterate from 1 to n and unrank).

The following is a collection of JavaScript for computing permutations and combinations counting with or without repetitions:

Many disciplines and sciences require the answer to the question: How many? In finite probability theory we need to know how many outcomes there would be for a particular event, and we need to know the total number of outcomes in the sample space.

Combinatorics, also referred to as **Combinatorial Mathematics**, is the field of mathematics concerned with problems of selection, arrangement, and operation within a finite or discrete system. Its objective is: How to count without counting. Therefore, one of the basic problems of combinatorics is to determine the number of possible configurations of objects of a given type.

You may ask, why combinatorics? If a sample spaces contains a finite set of outcomes, determining the probability of an event often is a counting problem. But, often the numbers are just too large to count in the 1, 2, 3, 4 ordinary ways.

A Fundamental Result: If an operation consists of two steps, of which the first can be done in n_1 ways and for each of these the second can be done in n_2 ways, then the entire operation can be done in a total of $n_1 \times n_2$ ways.

This simple rule can be generalised as follows: If an operation consists of k steps, of which the first can be done in n_1 ways and for each of these the second step can be done in n_2 ways, for each of these the third step can be done in n_3 ways and so forth, then the whole operation can be done in $n_1 \times n_2 \times n_3 \times n_4 \times \dots \times n_k$ ways.

Numerical Example: A quality control inspector wishes to select one part for inspection from each of four different bins containing 4, 3, 5 and 4 parts, respectively. The total number of ways that the parts can be selected is $4 \times 3 \times 5 \times 4$ or 240 ways.

Factorial Notation: The notation $n!$ (read as, n factorial) means by definition the product:

$$n! = (n)(n-1)(n-2)(n-3)\dots(3)(2)(1).$$

Notice that by convention, $0! = 1$, (i.e., 0 !). For example, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$

Permutations versus Combination: A permutation is an arrangement of objects from a set of objects. That is, the objects are chosen from a particular set and listed in a particular order. A combination is a selection of objects from a set of objects, that is objects are chosen from a particular set and listed, but the order in which the objects are listed is immaterial.

The number of ways of lining up k objects at a time from n distinct objects is denoted by ${}_n P^k$, and by the preceding we have:

$${}_n P^k = (n)(n-1)(n-2)(n-3)\dots(n-k+1)$$

Therefore, The number of permutations of n distinct objects taken k at a time can be written as:

$${}_n P^k = n! / (n - k)$$

Combinations: There are many problems in which we are interested in determining the number of ways in which k objects can be selected from n distinct objects without regard to the order in which they are selected. Such selections are called combinations or k sets. It may help to think of combinations as a committee. The key here is without regard for order.

The number of combinations of k objects from a set with n objects is nCk . For example, the combinations of $\{1, 2, 3, 4\}$ taken $k = 2$ at a time are $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{3, 4\}$, for a total of $6 = 4 / [(2)(4-2)]$ subsets.

The general formula is:

$$nC^k = n / [k (n-k)].$$

Permutation with Repetitions: How many different letter arrangements can be formed using the letters P E P P E R?

In general, there are multinomial coefficients:

$$n! / (n1! n2! n3! \dots nr!)$$

Different permutations of n objects, of which $n1$ are alike, $n2$, are alike, $n3$ are alike,..., nr are alike. Therefore, the answer is $6! / (3! 2! 1!) = 60$ possible arrangements of the letters P E P P E R.

3.7 Summary

Sorting is the process of arranging the elements of an array, so that they can be placed either in ascending or descending order. For example, consider an array $A = \{A1, A2, A3, A4, ?An\}$, the array is called to be in ascending order if the element of A are arranged like $A1 > A2 > A3 > A4 > A5 > ? > An$. There are two popular search methods that are widely used in order to search some items from the list. However, choice of the algorithm depends upon the arrangement of the list.

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then location of the item is returned otherwise the algorithm returns NULL.

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (nodes) maintain any complex relationship among them instead of having parent-child relationship.

Combinatorics, also referred to as combinatorial mathematics, is the field of mathematics concerned with problems of selection, arrangement, and operation within a finite or discrete system. Its objective is: How to count without counting. Therefore, one of the basic problems of combinatorics is to determine the number of possible configurations of objects of a given type.

3.8 Key Words/Abbreviations

- **Path:** A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
- **Closed Path:** A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0 = V_N$.
- **Simple Path:** If all the nodes of the graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.
- **Cycle:** A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
- **Connected Graph:** A connected graph is the one in which some path exists between every two vertices (u, v) in V.
- **Complete Graph:** A complete graph is the one in which every node is connected with all other nodes.
- **Weighted Graph:** In a weighted graph, each edge is assigned with some data such as length or weight.
- **Digraph:** A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- **Loop:** An edge that is associated with the similar end points can be called as loop.
- **Adjacent Nodes:** If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- **Degree of the Node:** A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

3.9 Learning Activity

1. Match List 1 with List 2 and choose the correct answer from the code given below:

List I

(Graph Algorithm)

- (a) Dijkstra's algorithm
- (b) Kruskal's algorithm
- (c) Floyd-Warshall algorithm
- (d) Topological sorting

where V and E are the number of vertices and edges in graph, respectively.

List II

(Time Complexity)

- (i) $O(E \lg E)$
- (ii) $\Theta(V^3)$
- (iii) $O(V^2)$
- (iv) $\Theta(V + E)$

2. Which sorting algorithms have the least best-case complexity?

3. Suppose there are 11 items in sorted order in an array. How many searches are required on the average, if binary search is employed and all searches are successful in finding the item?

4. The question is based on the following program fragment:

First iteration:

$$I=0, j=9$$

$$K=(0+9)/2=4$$

Elements with duplicate values of “2”
the condition if ($Y[K]$)

Second iteration:

$$I=0, j=k=4$$

$$K=(0+4)/2=2.$$

Third iteration:

$$I=0, j=k=2$$

$$K=(0+2)/2=1$$



Fourth iteration:

I=0, j=k=1

$$K = (0+1)/2 = 0$$

On which of the following contents of 'Y' and 'X' does the program fail?

5. Consider an array representation of an n element binary heap where the elements are stored from index 1 to index n of the array. For the element stored at index i of the array ($i \leq n$), then what is the index of the parent?

6. A list of n strings, each of length n , is sorted into lexicographic order using merge-sort algorithm. What is the worst-case running time of this computation?

3.10 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. Write a program to implement linear search algorithm.
 2. Write a program to implement a binary search algorithm.
 3. Implement the bubble sort algorithm.
 4. What is the difference between a stable and unstable sorting algorithm?
 5. What is depth-first search algorithm for a binary tree?
 6. How do you implement a counting sort algorithm?
 7. Write algorithms to check if two string are anagram.
 8. Implement the quick sort algorithm in your favourite programming language
 9. How to check if two string is rotation of each other?
 0. How do you implement a bucket sort algorithm?



B. Multiple Choice/Objective Type Questions

1. What is an external sorting algorithm?
 - (a) Algorithm that uses tape or disk during the sort
 - (b) Algorithm that uses main memory during the sort
 - (c) Algorithm that involves swapping
 - (d) Algorithm that are considered ‘in-place’
2. What is an internal sorting algorithm?
 - (a) Algorithm that uses tape or disk during the sort
 - (b) Algorithm that uses main memory during the sort
 - (c) Algorithm that involves swapping
 - (d) Algorithm that are considered ‘in-place’
3. What is the worst-case complexity of bubble sort?
 - (a) $O(n\log n)$
 - (b) $O(\log n)$
 - (c) $O(n)$
 - (d) $O(n^2)$
4. What is the advantage of selection sort over other sorting techniques?
 - (a) It requires no additional storage space
 - (b) It is scalable
 - (c) It works best for inputs which are already sorted
 - (d) It is faster than any other sorting technique
5. What is the average case complexity of selection sort?
 - (a) $O(n\log n)$
 - (b) $O(\log n)$
 - (c) $O(n)$
 - (d) $O(n^2)$
6. What is the disadvantage of selection sort?
 - (a) It requires auxiliary memory
 - (b) It is not scalable
 - (c) It can be used for small keys
 - (d) It takes linear time to sort the elements



7. Which of the following statements for a simple graph is correct?
- (a) Every path is a trail
 - (b) Every trail is a path
 - (c) Every trail is a path as well as every path is a trail
 - (d) Path and trail have no relation
8. Binary search can be categorised into which of the following?
- (a) Brute force technique
 - (b) Divide and conquer
 - (c) Greedy algorithm
 - (d) Dynamic programming
9. Given an array arr = {5,6,77,88,99} and key = 88; how many iterations are done until the element is found?
- (a) 1
 - (b) 3
 - (c) 4
 - (d) 2
10. Given an array arr = {45,77,89,90,94,99,100} and key = 100; what are the mid values (corresponding array elements) generated in the first and second iterations?
- (a) 90 and 99
 - (b) 90 and 100
 - (c) 89 and 94
 - (d) 94 and 99
11. What is the time complexity of binary search with iteration?
- (a) O(nlogn)
 - (b) O(logn)
 - (c) O(n)
 - (d) O(n²)

Answers:

1. (a), 2. (b), 3. (d), 4. (a), 5. (d), 6. (b), 7. (a), 8. (b), 9. (d), 10. (a), 11. (b)

3.11 References

References of this unit have been given at the end of the book.



UNIT 4 FUNDAMENTALS OF DATA STRUCTURES

Structure:

4.0 Learning Objectives

4.1 Introduction

4.2 Stack

4.3 Queue

4.4 Graph Data Structure

4.5 Graphs

4.6 Tree

4.7 Sets and Dictionaries

4.8 Summary

- Explain queues in a practical way
- Describe all types of graphs
- Implement trees algorithm
- Elaborate sets and dictionaries

4.1 Introduction

Data structure can be defined as the group of data elements which provide an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of data structures are arrays, linked list, stack, queue, etc. Data structures are widely used in almost every aspect of Computer Science, i.e., operating system, compiler design, artificial intelligence, graphics and many more.

Data structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Need of Data Structures

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

- **Processor Speed:** To handle very large amount of data, high-speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.
- **Data Search:** Consider an inventory size of 106 items in a store. If our application needs to search for a particular item, it needs to traverse 106 items every time, resulting in slowing down the search process.
- **Multiple Requests:** If thousands of users are searching the data simultaneously on a web server, then there are chances that a very large server can fail during that process. In order to solve the above problems, data structures are used. Data is organised to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Advantages of Data Structures

- **Efficiency:** Efficiency of a program depends upon the choice of data structure. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organise our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.
- **Reusability:** Data structures are reusable, i.e., once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
- **Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification

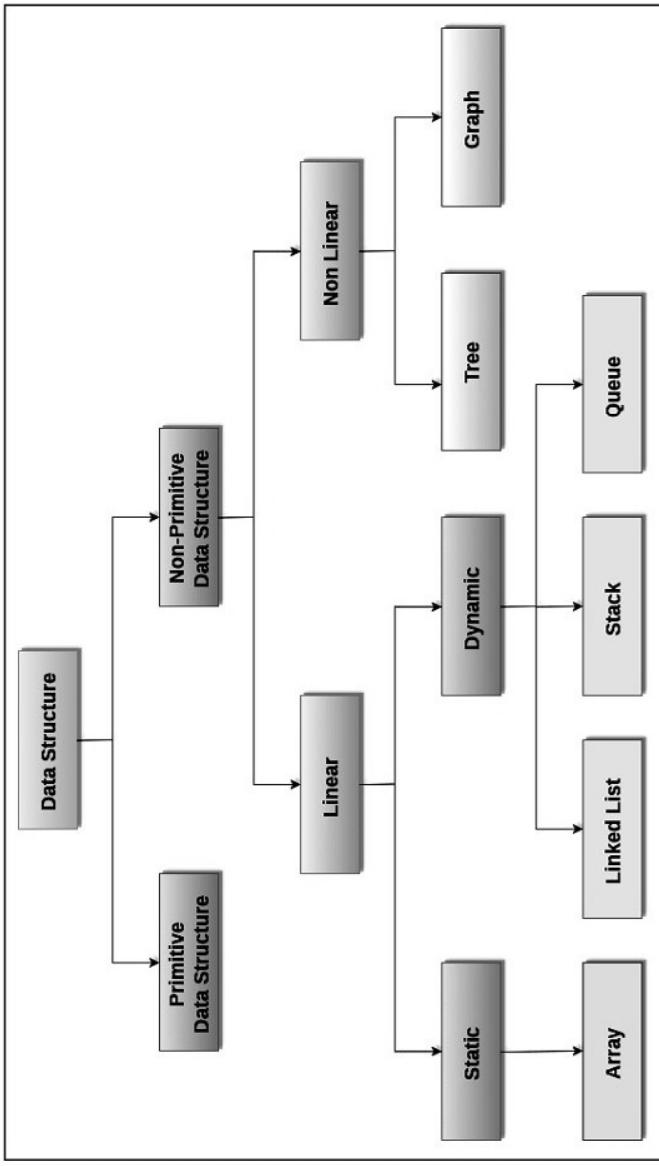


Fig. 4.1: Data Structure Classification

Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

- **Arrays:** An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double. The elements of array share the same variable name, but each one carries a different index number known as subscript. The array can be one-dimensional, two-dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3], age[98], age[99].

- **Linked List:** Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.
- **Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**. A stack, an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example, piles of plates or deck of cards, etc.
- **Queue:** Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.

It is an abstract data structure, similar to stack. Queue is opened at both end, therefore it follows First in First out (FIFO) methodology for storing the data items.

Non-linear Data Structures

This data structure does not form a sequence, i.e., each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in a sequential structure.

Types of Non-linear Data Structures are given below:

- **Trees:** Trees are multilevel data structures with a hierarchical relationship among its elements known as **nodes**. The bottom-most nodes in the hierarchy are called leaf nodes, while the topmost node is called **root node**. Each node contains pointers to point to adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child, except the leaf nodes, whereas each node can have at the most one parent, except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

- **Graphs:** Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle, while the tree cannot have one.

Operations on Data Structure

- 1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in six different subjects, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects, i.e., six, in order to find the average.

- 2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.
If the size of data structure is n , then we can only insert **$n-1$** data elements into it.
- 3) **Deletion:** The process of removing an element from the data structure is called deletion.
We can delete an element from the data structure at any random location.
If we try to delete an element from an empty data structure, then **underflow** occurs.



- 4) **Searching:** The process of finding the location of an element within the data structure is called searching. There are two algorithms to perform searching, linear search and binary search. We will discuss each one of them later in this book.
- 5) **Sorting:** The process of arranging the data structure in a specific order is known as sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.
- 6) **Merging:** When two lists List A and List B of size M and N, respectively, of similar type of elements, are clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

4.2 Stack

1. Stack is an ordered list in which insertion and deletion can be performed only at one end, that is called **top**.
2. Stack is a recursive data structure having a pointer to its top element.
3. Stacks are sometimes called as Last in First out (LIFO) lists, i.e., the element which is inserted first in the stack will be deleted last from the stack.

Applications of Stack

1. Recursion
2. Expression evaluations and conversions
3. Parsing
4. Browsers
5. Editors
6. Tree traversals

Operations on Stack

There are various operations which can be performed on stack.

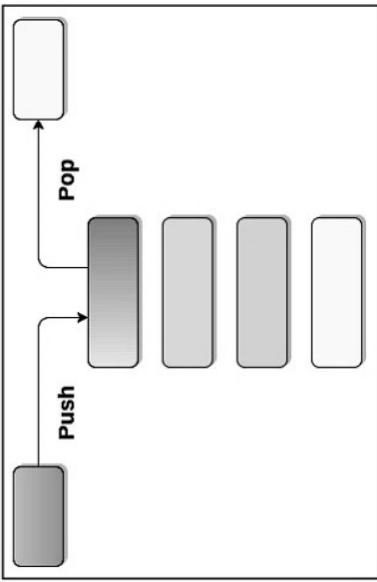


Fig. 4.2: Operations on Stack

1. **Push:** Adding an element onto the stack.

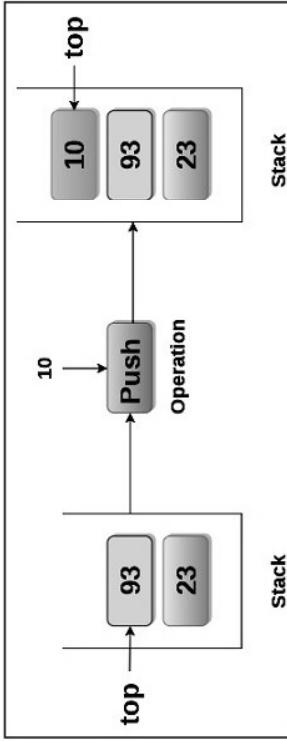


Fig. 4.3: Push Operation

2. **Pop:** Removing an element from the stack.

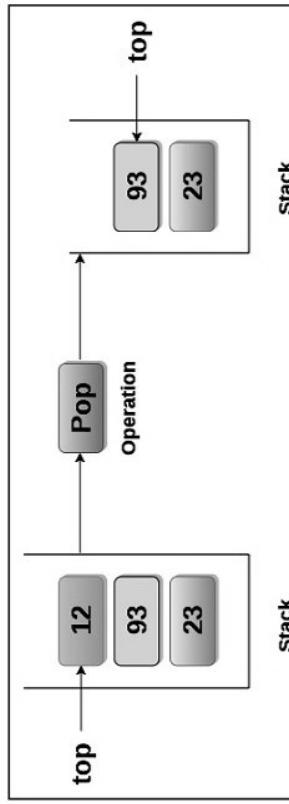


Fig. 4.4: Pop Operation

3. **Peek:** Looking at all the elements of stack without removing them.

How the Stack Grows?**Scenario 1: Stack is empty.**

The stack is called empty if it doesn't contain any element inside it. At this stage, the value of variable top is -1.

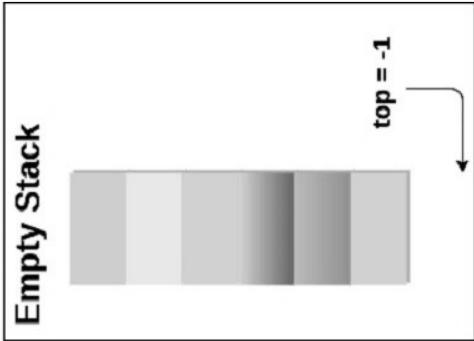


Fig. 4.5: Empty Stack

Scenario 2: Stack is not empty.

Value of top will get increased by 1 every time when we add any element to the stack. In the following stack, after adding first element, top = 2.

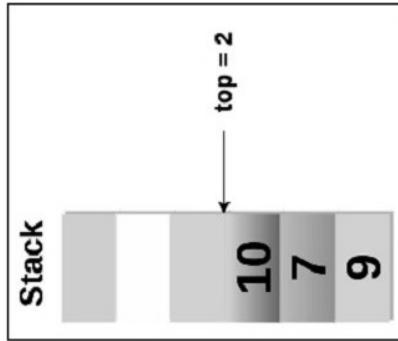


Fig. 4.6: Stack is Not Empty

Scenario 3: Deletion of an element.

Value of top will get decreased by 1 whenever an element is deleted from the stack.

In the following stack, after deleting 10 from the stack, top = 1.

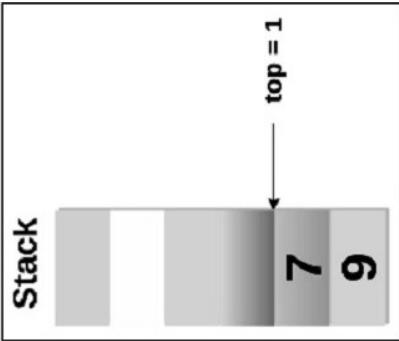


Fig. 4.7: Deletion of an Element

Table 4.1: Top and its Value

Top Position	Status of Stack
-1	Empty
0	Only one element in the stack
N-1	Stack is full
N	Overflow

Array Implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an Element onto the Stack (Push Operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves the following two steps:

1. Increment the variable top so that it can now refer to the next memory location.

2. Add element at the position of incremented top. This is referred to as adding a new element at the top of the stack.

Stack is overflowed when we try to insert an element into a completely filled stack. Therefore, our main function must always avoid stack overflow condition.

Algorithm

```

1. begin
2.   if top == n then stack full
3.   top = top + 1
4.   stack (top) : = item;
5. end

```

Time Complexity: o(1)

Implementation of Push Algorithm in C Language

```

1. void push (int val,int n) //n is size of the stack
2. {
3.   if (top == n )
4.     printf("\n Overflow");
5.   else
6.   {
7.     top = top +1;
8.     stack[top] = val;
9.   }
10. }

```

Deletion of an Element from a Stack (Pop Operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The topmost element of the stack is stored in another variable and then the top is decremented by 1. The operation returns the deleted value that was stored in another variable as the result.



The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm

```
1. begin
2.   if top = 0 then stack empty;
3.   item := stack(top);
4.   top = top - 1;
5. end;
```

Time Complexity: $O(1)$

Visiting Each Element of the Stack (Peek Operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm

```
PEEK (STACK, TOP)
```

```
1. Begin
2.   if top = -1 then stack empty
3.   item = stack[top]
4.   return item
5. End
```

Time Complexity: $O(n)$

Linked List Implementation of Stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenarios is same for all the operations, i.e., push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflow if the space left in the memory heap is not enough to create a node.



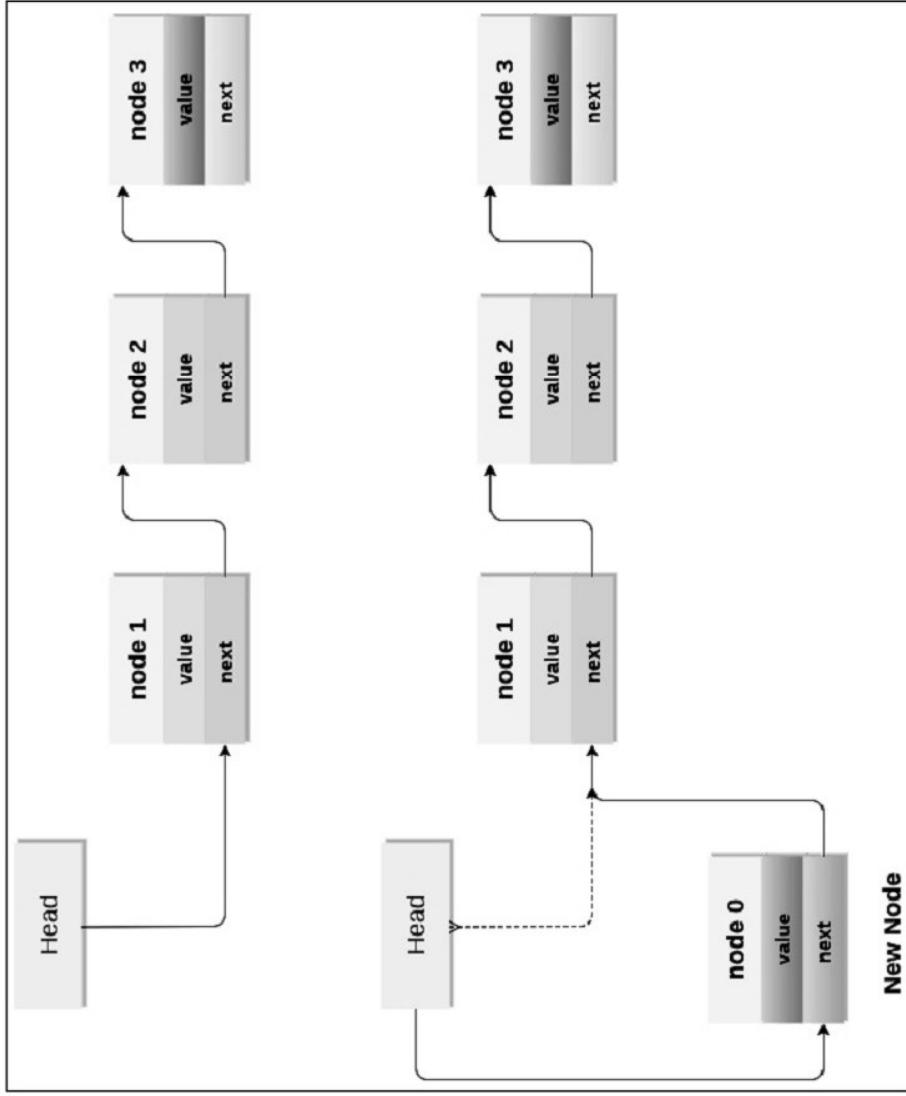


Fig. 4.9: Adding a Node to the Stack

Deleting a Node from the Stack (POP Operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps:

- 1. Check for the Underflow Condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
- 2. Adjust the Head Pointer Accordingly:** In stack the elements are popped only from one end. Therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity: $O(n)$

Display the Nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organised in the form of stack. For this purpose, we need to follow the following steps:

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity: $O(n)$

4.3 Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First in First out list.
3. For example, people waiting in line for a rail ticket form a queue.

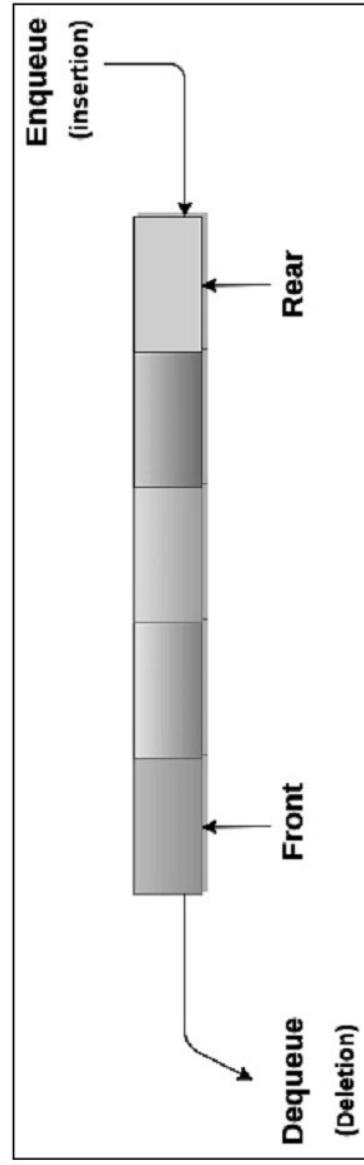


Fig. 4.10: Queue

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.



2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for e.g., pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queues are used to maintain the playlist in media players in order to add and remove the songs from the playlist.
5. Queues are used in operating systems for handling interrupts.

Array Representation of Queue

We can easily represent a queue by using linear arrays. There are two variables, i.e., front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing five elements along with the respective values of front and rear, is shown in the following figure:

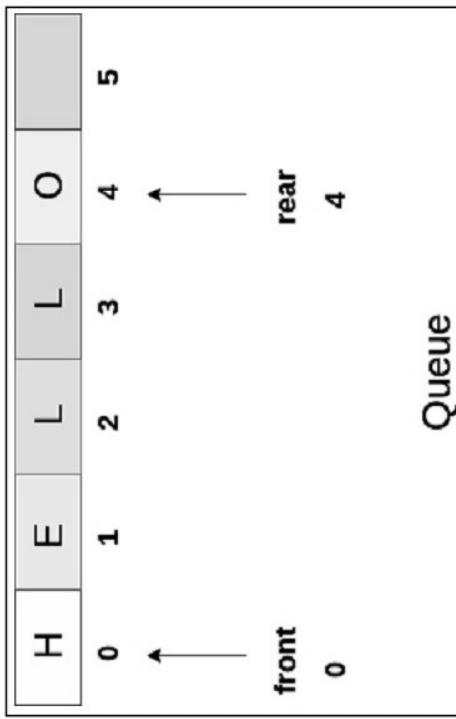


Fig. 4.11: Array Representation of Queue

The above figure shows the queue of characters forming the English word ‘HELLO’. Since, no deletion is performed in the queue till now, therefore the value of front remains -1. However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5, while the value of front remains same.

H	E	L	O	G
0	1	2	3	4

↑
front
0

↑
rear
5

Fig. 4.12: After Inserting an Element into the Queue

After deleting an element, the value of front will increase from -1 to 0. However, the queue will look something like following:

	E	L	O	G
0	1	2	3	4

↑
front
1

↑
rear
5

Fig. 4.13: After Deleting an Element into the Queue

Algorithm to Insert any Element in a Queue

Check if the queue is already full by comparing rear to max - 1. If so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise, keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

Step 1: IF REAR = MAX - 1

 Write OVERFLOW

 Go to step

 [END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

 SET FRONT = REAR = 0

 ELSE

 SET REAR = REAR + 1

 [END OF IF]

Step 3: Set QUEUE[REAR] = NUM

Step 4: EXIT

Algorithm to Delete an Element from the Queue

If the value of front is -1 or value of front is greater than rear, then write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

Step 1: IF FRONT = -1 or FRONT > REAR

 Write UNDERFLOW

 ELSE

 SET VAL = QUEUE[FRONT]

 SET FRONT = FRONT + 1

 [END OF IF]

Step 2: EXIT

Linked List Implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation cannot be used for the large-scale applications where the queues are implemented. One of the alternatives of array implementation is linked list implementation of queue.

The storage requirement of linked representation of a queue with n elements is $O(n)$, while the time requirement for operations is $O(1)$.

In a linked queue, each node of the queue consists of two parts, i.e., data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory, i.e., front pointer and rear pointer. The front pointer contains the address of the starting element of the queue, while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end, respectively. If front and rear both are null, it indicates that the queue is empty.

The linked representation of queue is shown in the following Figure:

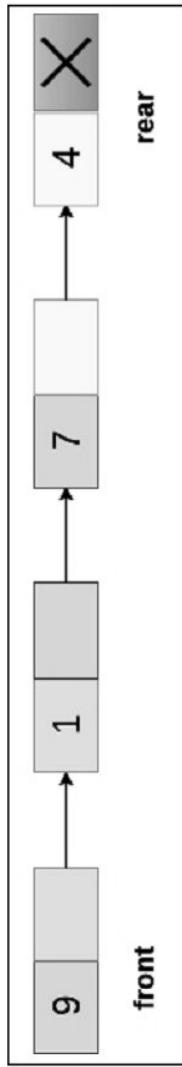


Fig. 4.14: Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are insertion and deletion.

Insert Operation

The insert operation appends the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement:

```
1. ptr = (struct node *) malloc (sizeof(struct node));
```

There can be two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front** = **NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to **NULL**.

```
1. ptr->data = item;
2. if(front == NULL)
3. {
4.     front = ptr;
5.     rear = ptr;
6.     front->next = NULL;
7.     rear->next = NULL;
8. }
```

In the second case, the queue contains more than one element. The condition **front** = **NULL** becomes false. In this scenario, we need to update the end pointer **rear** so that the next pointer of **rear** will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node ptr. We also need to make the next pointer of **rear** point to null.

```
1. rear->next = ptr;
2. rear = ptr;
3. rear->next = NULL;
```

```
Step 3: IF FRONT = NULL  
    SET FRONT = REAR = PTR  
    SET FRONT -> NEXT = REAR -> NEXT = NULL  
ELSE  
    SET REAR -> NEXT = PTR  
    SET REAR = PTR  
    SET REAR -> NEXT = NULL  
[END OF IF]  
  
Step 4: END
```

Deletion Operation

Deletion operation removes the element that is first inserted among all the queue elements.

Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make an exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements:

1. ptr = front;
2. front = front -> next;
3. free(ptr);

The algorithm and C function is given as follows:

Algorithm

Step 1: IF FRONT = NULL

 Write "Underflow"

 Go to Step 5

[END OF IF]



Step 2: SET PTR = FRONT

Step 3: SET FRONT = FRONT -> NEXT

Step 4: FREE PTR

Step 5: END

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets (V, E) , where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph:

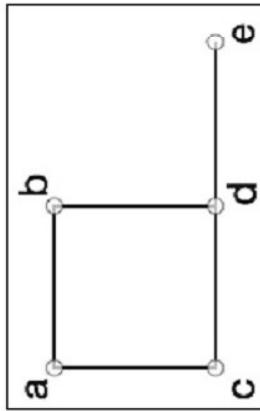


Fig. 4.15: Graph

In the above graph,

$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, bd, cd, de\}$$

4.4 Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarise ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labelled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1, and so on.

- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2, and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

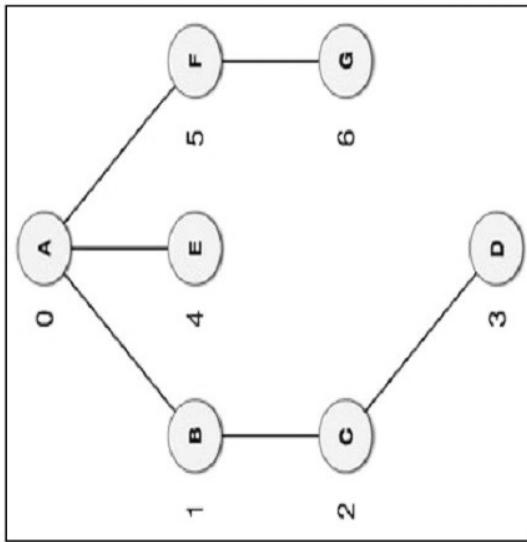


Fig. 4.16: Graph Data Structure

Basic Operations

Following are basic primary operations of a graph:

- **Add Vertex** – Adds a vertex to the graph.
- **Add Edge** – Adds an edge between the two vertices of the graph.
- **Display Vertex** – Displays a vertex of the graph.

4.5 Graphs

Graph Traversal Algorithm

In this part of the tutorial we will discuss the techniques by using which, we can traverse all the vertices of the graph.

Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs. Let's discuss each one of them in detail.

- Breadth-First Search
- Depth-First Search

Breadth First Search (BFS) Algorithm

Breadth-first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explores all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth-first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and the process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

Algorithm

Step 1: SET STATUS = 1 (ready state)

for each node in G

Step 2: Enqueue the starting node A

and set its STATUS = 2

(waiting state)

Step 3: Repeat Steps 4 and 5 until

QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example: Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.

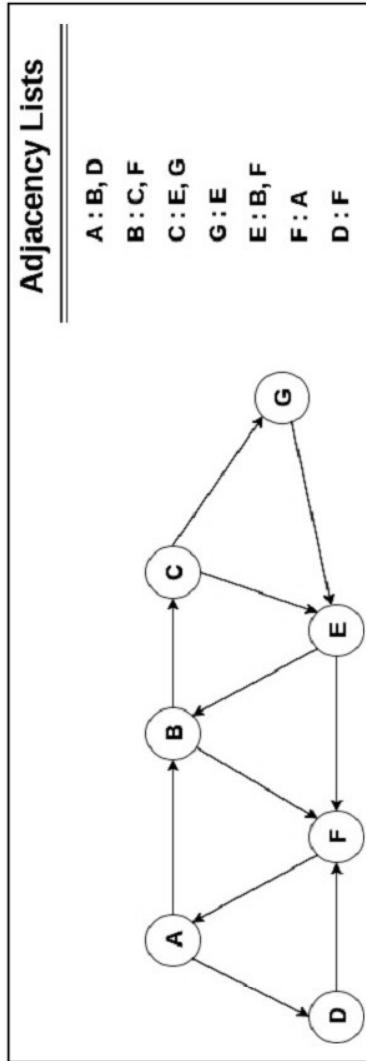


Fig. 4.17: Graph G

Solution:

Minimum path P can be found by applying breadth-first search algorithm that will begin at node A and will end at E. The algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed, while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

Lets start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.
 1. QUEUE1 = {A}
 2. QUEUE2 = {NULL}
2. Delete the node A from QUEUE1 and insert all its neighbours. Insert node A into QUEUE2.
 1. QUEUE1 = {B, D}
 2. QUEUE2 = {A}
3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.
 1. QUEUE1 = {D, C, F}
 2. QUEUE2 = {A, B}
4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.
 1. QUEUE1 = {C, F}
 2. QUEUE2 = {A, B, D}
5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.
 1. QUEUE1 = {F, E, G}
 2. QUEUE2 = {A, B, D, C}
6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours have already been added, we will not add them again. Add node F to QUEUE2.
 1. QUEUE1 = {E, G}
 2. QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours have already been added to QUEUE1, therefore we will not add them again. All the nodes are visited and the target node, i.e., E is encountered into QUEUE2.

1. QUEUE1 = {G}

2. QUEUE2 = {A, B, D, C, F, E}

Now, backtrack from E to A using the nodes available in QUEUE2.

The minimum path will be A → B → C → E.

Depth First Search (DFS) Algorithm

Depth-first search (DFS) algorithm starts with the initial node of the graph G, and then goes deeper and deeper until we find the goal node or the node which has no children. The algorithm then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that lead to an unvisited node are called discovery edges, while the edges that lead to an already visited node are called block edges.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G.

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state).

Step 3: Repeat Steps 4 and 5 until STACK is empty.

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state).

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example: Consider the graph G along with its adjacency list given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth-first search (DFS) algorithm.

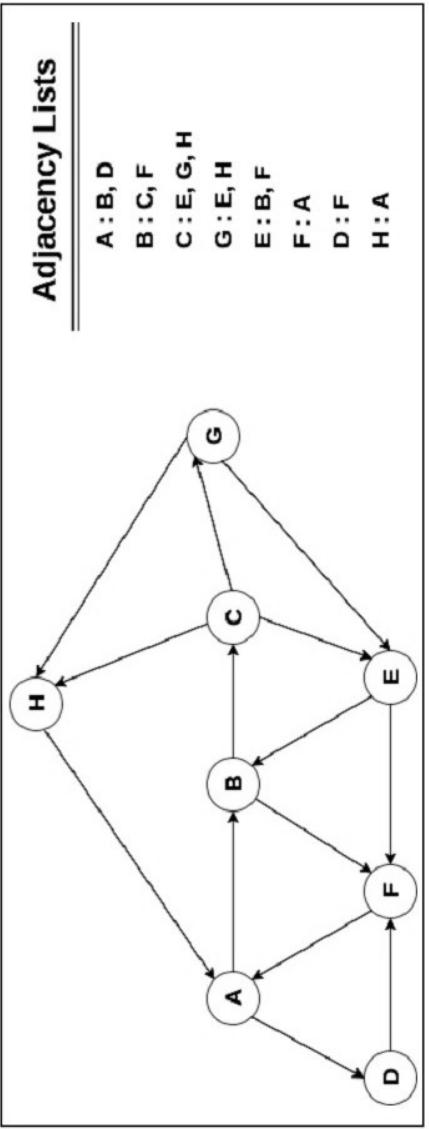


Fig. 4.18: Graph G Along with its Adjacency

Solution:

1. Push H onto the stack
 1. STACK: H
2. POP the top element of the stack, i.e., H, print it and push all the neighbours of H onto the stack that are in ready state.
 1. Print H
 2. STACK: A
3. Pop the top element of the stack, i.e., A, print it and push all the neighbours of A onto the stack that are in ready state.
 1. Print A
 2. Stack: B, D
4. Pop the top element of the stack, i.e., D, print it and push all the neighbours of D onto the stack that are in ready state.
 1. Print D
 2. Stack: B, F

5. Pop the top element of the stack, i.e., F, print it and push all the neighbours of F onto the stack that are in ready state.
 1. Print F
 2. Stack: B
6. Pop the top of the stack, i.e., B and push all the neighbours.
 1. Print B
 2. Stack: C
7. Pop the top of the stack, i.e., C and push all the neighbours.
 1. Print C
 2. Stack: E, G
8. Pop the top of the stack, i.e., G and push all its neighbours.
 1. Print G
 2. Stack: E
9. Pop the top of the stack, i.e., E and push all its neighbours.
 1. Print E
 2. Stack:
10. Hence, the stack now becomes empty and all the nodes of the graph have been traversed.
11. The printing sequence of the graph will be :
 1. H → A → D → F → B → C → G → E

4.6 Tree

- A tree is a recursive data structure containing the set of one or more data nodes where one node is designated as the root of the tree, while the remaining nodes are called as the children of the root.
- The nodes other than the root node are partitioned into the non-empty sets where each one of them is to be called a subtree.



- Nodes of a tree either maintain a parent-child relationship between them or they are sister nodes.
- In a general tree, a node can have any number of children nodes, but it can have only a single parent.
- The following image shows a tree where the node A is the root node of the tree, while the other nodes can be seen as the children of A.

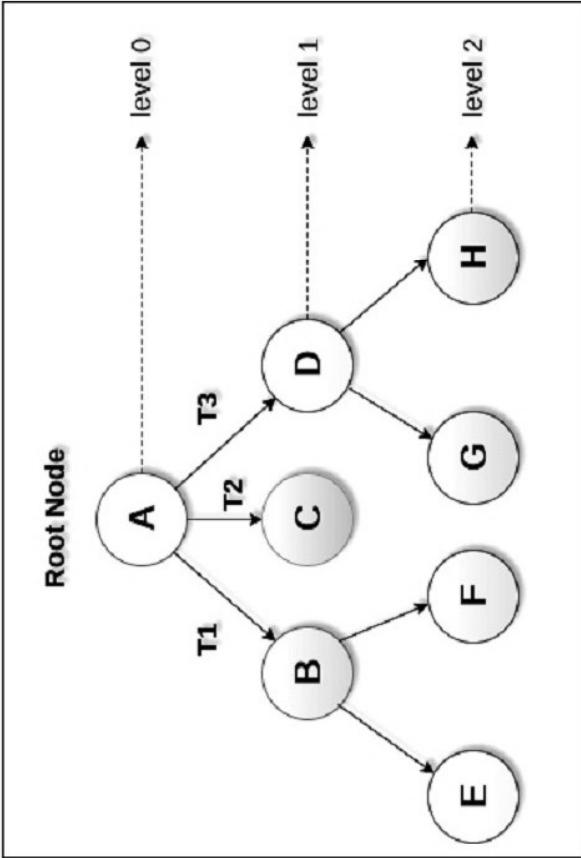


Fig. 4.19: Tree

Basic Terminology

- **Root Node:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.
- **Subtree:** If the root node is not null, the tree T1, T2 and T3 are called subtrees of the root node.
- **Leaf Node:** The node of tree, which doesn't have any child node, is called a leaf node. Leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

- **Path:** The sequence of consecutive edges is called path. In the tree shown in the above image, path to the node E is A → B → E.
- **Ancestor Node:** An ancestor of a node is any predecessor node on a path from root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, the node F has the ancestors, B and A.
- **Degree:** Degree of a node is equal to number of children a node have. In the tree shown in the above image, the degree of node B is 2. Degree of a leaf node is always 0, while in a complete binary tree, degree of each node is equal to 2.
- **Level Number:** Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. Root node of the tree is always present at level 0.

Static Representation of Tree

```

1. #define MAXNODE 500
2. struct treenode {
3.     int root;
4.     int father;
5.     int son;
6.     int next;
7. }
```

Dynamic Representation of Tree

```

1. struct treenode
2. {
3.     int root;
4.     struct treenode *father;
5.     struct treenode *son
6.     struct treenode *next;
7. }
```

Types of Tree

The tree data structure can be classified into six different categories.

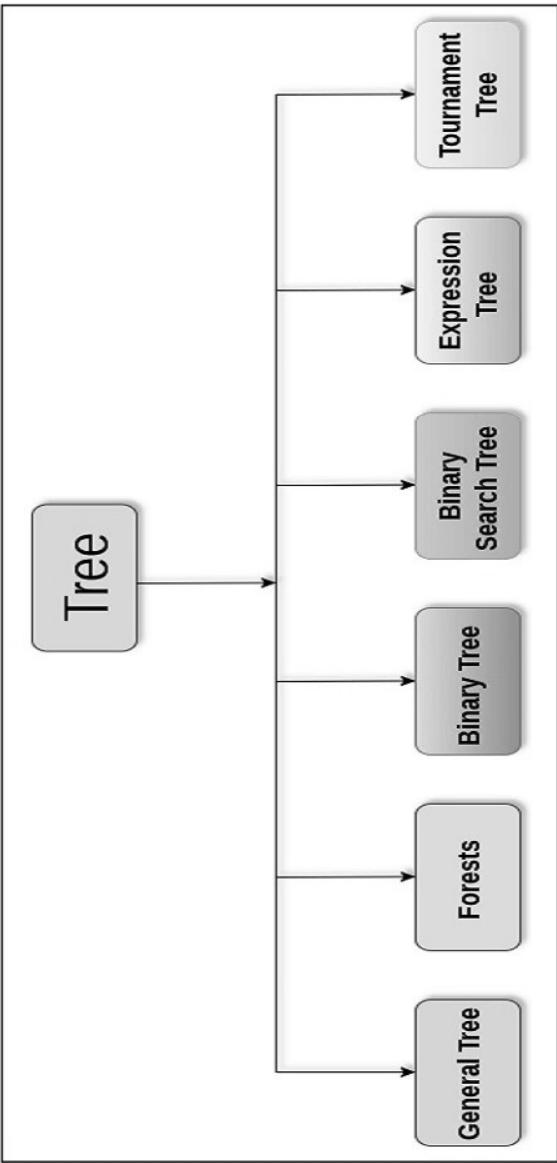


Fig. 4.20: Types of Tree

General Tree

General tree stores the elements in a hierarchical order in which the top-level element is always present at level 0 as the root element. All the nodes except the root node are present at number of levels. The nodes which are present on the same level are called siblings, while the nodes which are present on the different levels exhibit the parent-child relationship among them. A node may contain any number of subtrees. The tree in which each node contains three subtrees is called ternary tree.

Forests

Forests can be defined as the set of disjoint trees which can be obtained by deleting the root node and the edges which connects root node to the first level node.

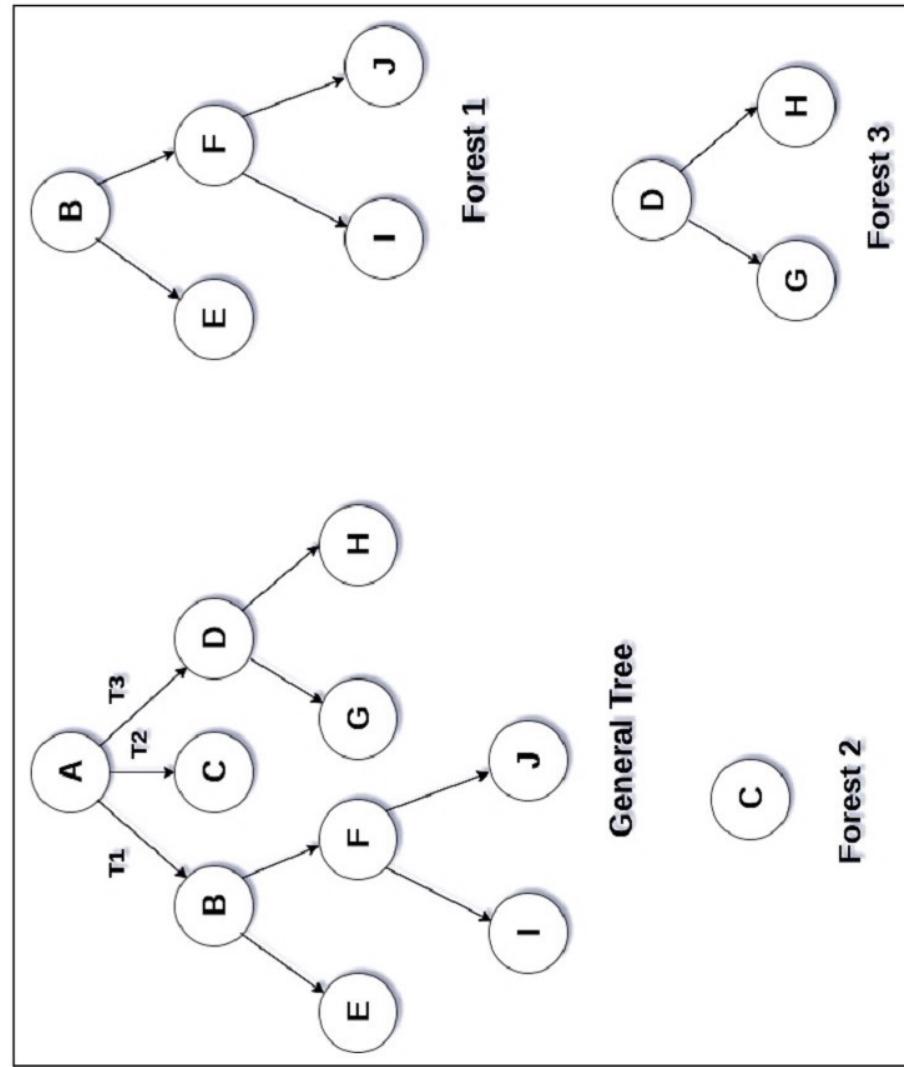


Fig. 4.21: General Tree

Binary Tree

Binary tree is a data structure in which each node can have at most two children. The node present at the topmost level is called the root node. A node with the 0 children is called leaf node. Binary trees are used in the applications like expression evaluation and many more. We will discuss binary tree in detail, later in this tutorial.

Binary Search Tree

Binary search tree is an ordered binary tree. All the elements in the left subtree are less than the root, while elements present in the right subtree are greater than or equal to the root node element. Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.

Expression Tree

Expression trees are used to evaluate the simple arithmetic expressions. Expression tree is basically a binary tree where internal nodes are represented by operators, while the leaf nodes are represented by operands. Expression trees are widely used to solve algebraic expressions like $(a+b)*(a-b)$. Consider the following example:

Q. Construct an expression tree by using the following algebraic expression.

$$(a + b) / (a * b - c) + d$$

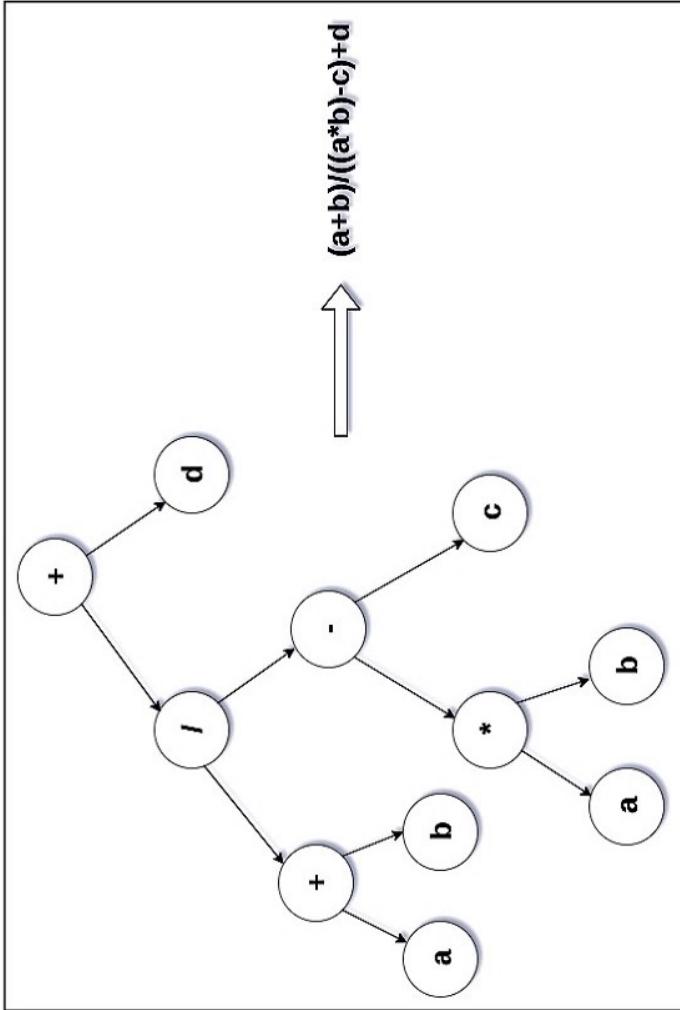


Fig. 4.22: Expression Tree

Tournament Tree

Tournament tree are used to record the winner of the match in each round being played between two players. Tournament tree can also be called as selection tree or winner tree. External nodes represent the players among which a match is being played, while the internal nodes represent the winner of the match played. At the topmost level, the winner of the tournament is present as the root node of the tree.

For example, tree of a chess tournament being played among four players is shown as follows. However, the winner in the left subtree will play against the winner of right subtree.

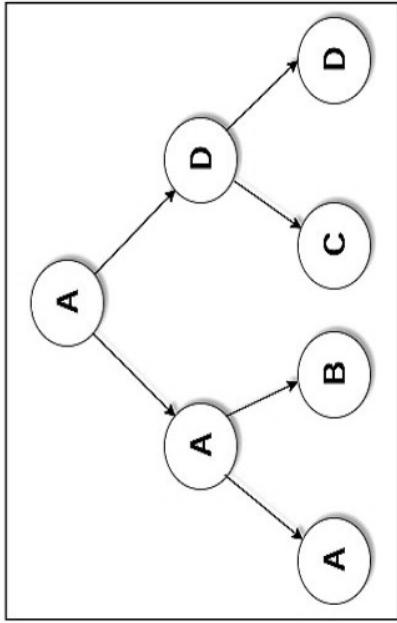


Fig. 4.23: Tournament Tree

Binary Tree

Binary tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets.

1. Root of the node.
2. Left subtree which is also a binary tree.
3. Right binary subtree.

A binary tree is shown in the following image:

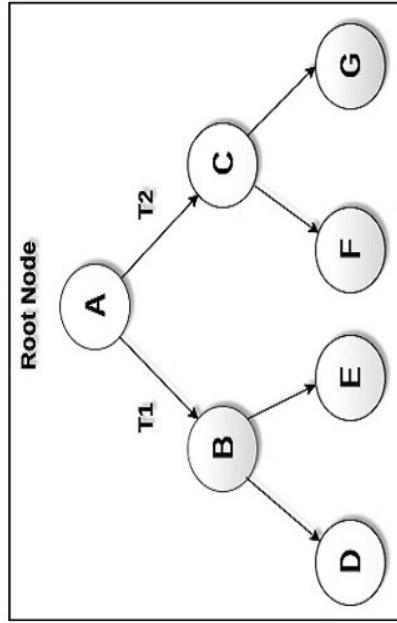


Fig. 4.24: Binary Tree

Types of Binary Tree

1. **Strictly Binary Tree:** In strictly binary tree, every non-leaf node contain non-empty left and right subtrees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves will have $(2n - 1)$ nodes.

A strictly binary tree is shown in the following figure:

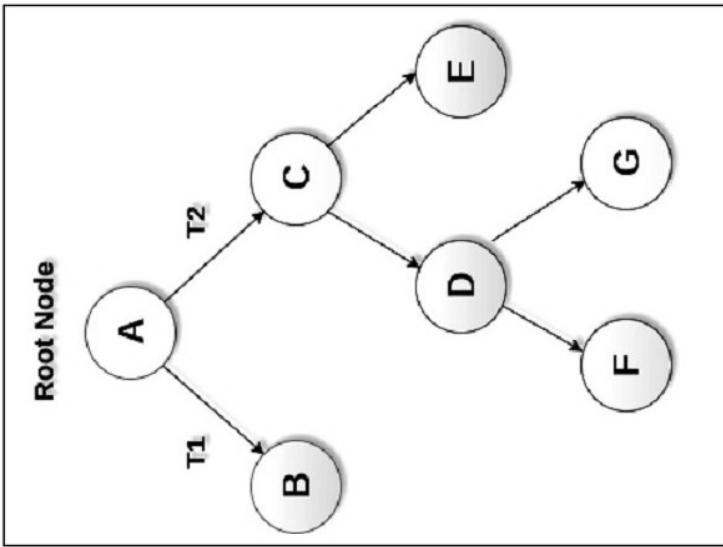
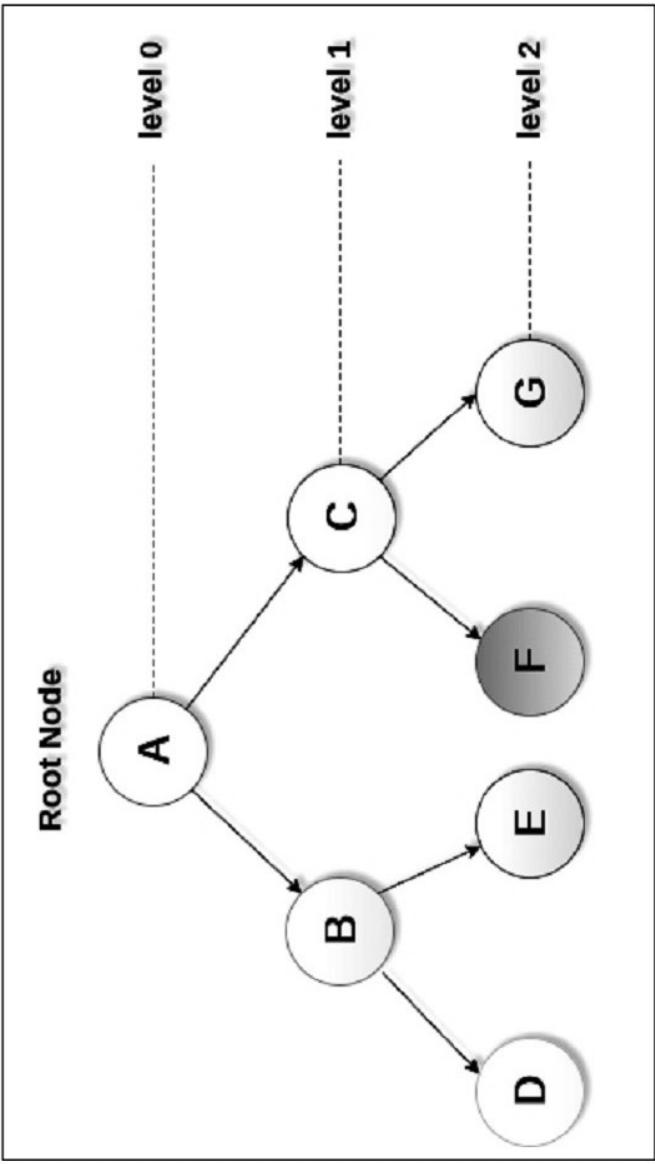


Fig. 4.25: Strictly Binary Tree

2. **Complete Binary Tree:** A binary tree is said to be a complete binary tree if all of the leaves are located at the same level d . A complete binary tree is a binary tree that contains exactly $2^{\lfloor d \rfloor}$ nodes at each level between level 0 and d . The total number of nodes in a complete binary tree with depth d is $2^{d+1}-1$ where leaf nodes are 2^d , while non-leaf nodes are 2^d-1 .

**Fig. 4.26: Complete Binary Tree****Table 4.2: Binary Tree Traversal**

SN	Traversal	Description
1	Pre-order Traversal	Traverse the root first, then traverse into the left subtree and right subtree, respectively. This procedure will be applied to each subtree of the tree recursively.
2	Inorder Traversal	Traverse the left subtree first, and then traverse the root and the right subtree, respectively. This procedure will be applied to each subtree of the tree recursively.
3	Postorder Traversal	Traverse the left subtree and then traverse the right subtree and root, respectively. This procedure will be applied to each subtree of the tree recursively.

Binary Tree Representation

There are two types of representation of a binary tree:

1. **Linked Representation:** In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent-child relationship like a tree. Every node contains three parts: pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.

Consider the binary tree given in the figure below.

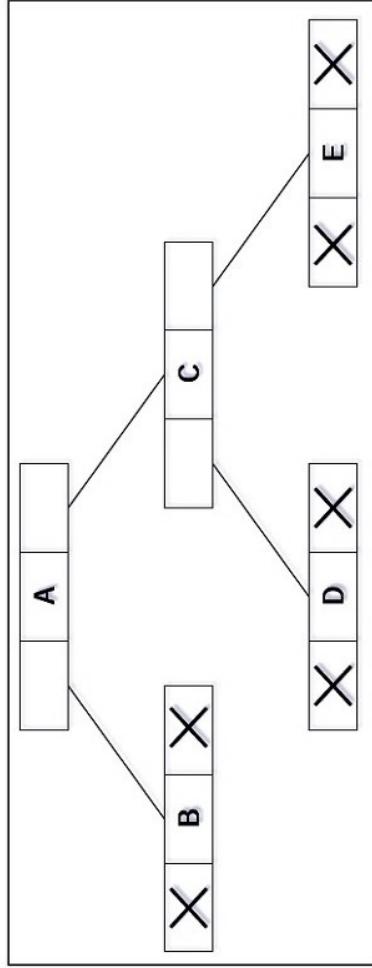


Fig. 4.27: Binary Tree Representation

In the above figure, a tree is seen as the collection of nodes where each node contains three parts: left pointer, data element and right pointer. Left pointer stores the address of the left child, while the right pointer stores the address of the right child. The leaf node contains null in its left and right pointers.

The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains null in its left and right pointers.

root

In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the first index of the array. If a node is stored at i th index, then its left and right children will be stored at $2i$ and $2i+1$ location. If the first index of the array, i.e., $\text{tree}[1]$ is 0, it means that the tree is empty.

AVL Tree

AVL tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL tree can be defined as height balanced binary search tree in which each node is associated with a balance factor, which is calculated by subtracting the height of its right subtree from that of its left subtree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

$$\text{Balance Factor (k)} = \text{height(left(k))} - \text{height(right(k))}$$

If balance factor of any node is 1, it means that the left subtree is one level higher than the right subtree.

If balance factor of any node is 0, it means that the left subtree and right subtree contain equal height.

If balance factor of any node is -1, it means that the left subtree is one level lower than the right subtree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. Therefore, it is an example of AVL tree.

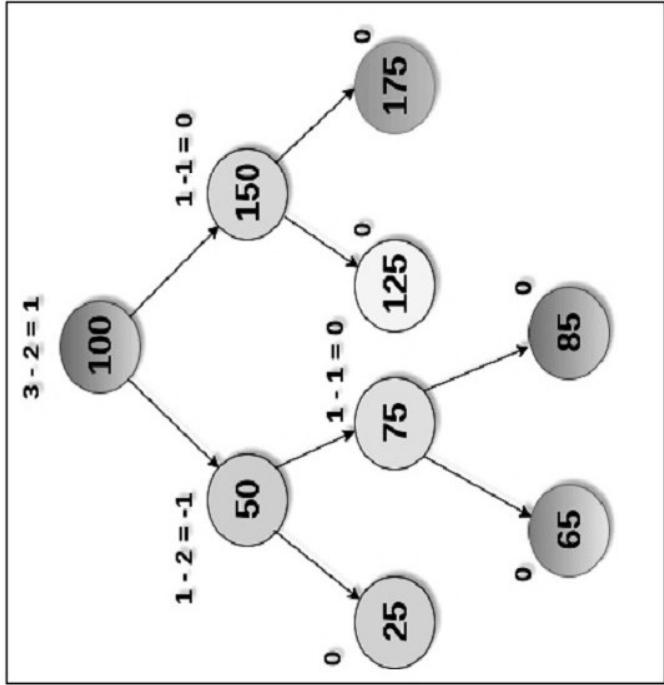


Fig. 4.30: AVL Tree

Table 4.3: Complexity

Algorithm	Average case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Operations on AVL Tree

Due to the fact that AVL tree is also a binary search tree, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property, and therefore they need to be revisited.

Table 4.4: Operations on AVL Tree

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree, therefore various types of rotations are used to rebalance the tree.

Why AVL Tree ?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is $O(h)$. However, it can be extended to $O(n)$ if the BST becomes skewed (i.e., worst case). By limiting this height to $\log n$, AVL tree imposes an upper bound on each operation to be $O(\log n)$ where n is the B tree.

B tree is a specialised m-way tree that can be widely used for disk access. A B tree of order m can have at most $m-1$ keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties:

1. Every node in a B tree contains at most m children.
2. Every node in a B tree except the root node and the leaf node contain at least $m/2$ children.
3. The root nodes must have at least two nodes.
4. All leaf nodes must be at the same level.

It is not necessary that all the nodes contain the same number of children, but each node must have $m/2$ number of nodes.

Number of Nodes

A B tree of order 4 is shown in the following image:

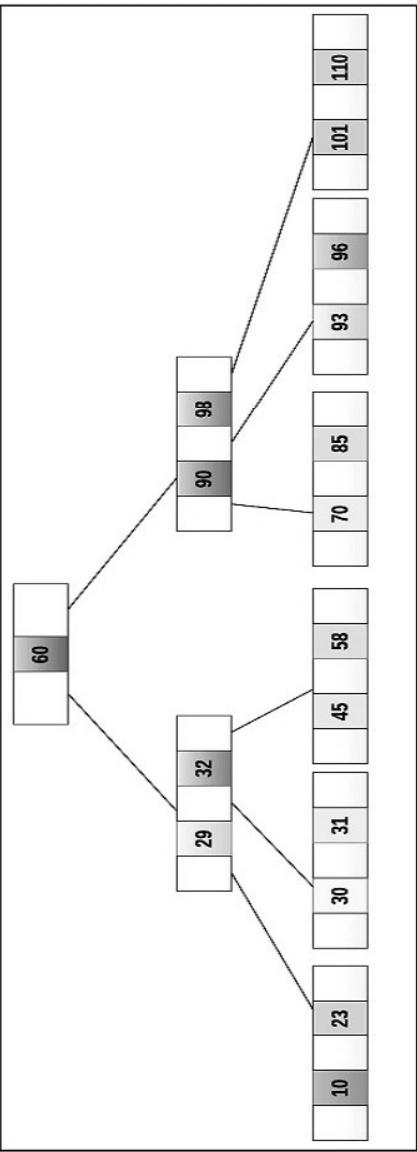


Fig. 4.31: B Tree of Order 4

While performing some operations on B tree, any property of B tree may violate, such as number of minimum children a node can have. To maintain the properties of B tree, the tree may split or join.

Operations

Searching

Searching in B trees is similar to that in binary search tree. For example, if we search for an item 49 in the following B tree, the process will be something like following:

1. Compare item 49 with root node 78. Since $49 < 78$, hence move to its left subtree.
2. Since, $40 < 49 < 56$, traverse right subtree of 40.
3. $49 > 45$, move to right. Compare 49.
4. Match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes $O(\log n)$ time to search any element in a B tree.

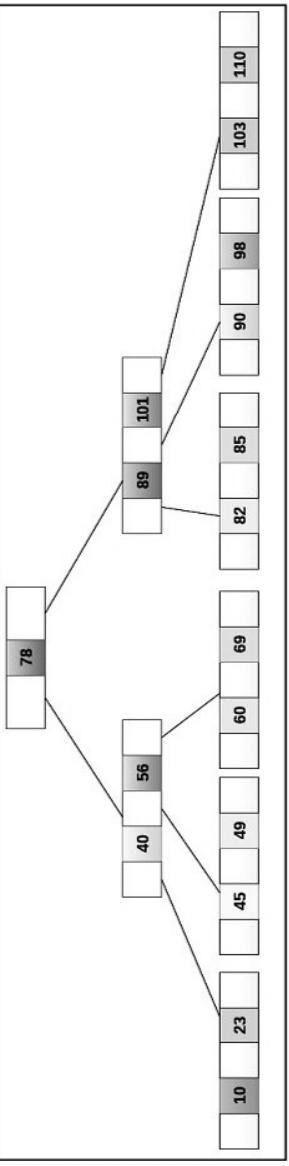


Fig. 4.32: Searching in B Trees

Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B tree:

1. Traverse the B tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than $m-1$ keys, then insert the element in the increasing order.
3. Else, if the leaf node contains $m-1$ keys, then follow the following steps:
 - Insert the new element in the increasing order of elements.
 - Split the node into the two nodes at the median.
 - Push the median element up to its parent node.
 - If the parent node also contains $m-1$ number of keys, then split it too by following the same steps.

Example: Insert the node 8 into the B tree of order 5 shown in the following image:

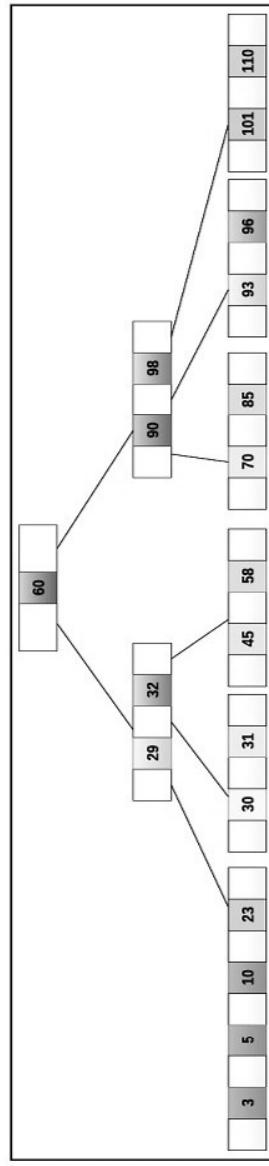


Fig. 4.33: B Tree of Order 5

8 will be inserted to the right of 5, therefore insert 8.

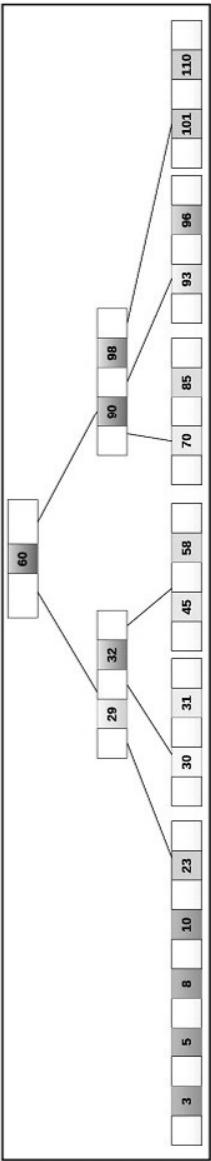


Fig. 4.34: Insertion into the B Tree

The node, now contains 5 keys which are greater than $(5 - 1 = 4)$ keys. Therefore, split the node from the median, i.e., 8 and push it up to its parent node shown as follows:

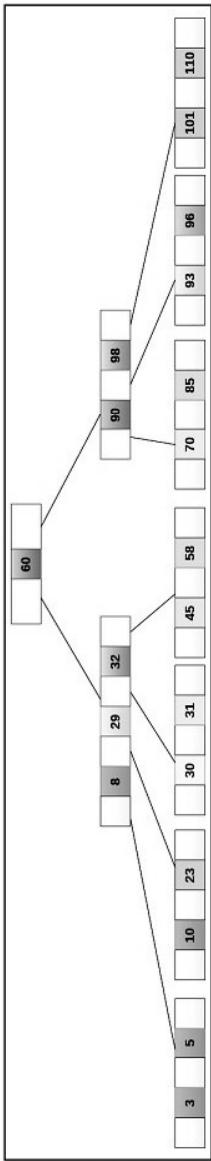


Fig. 4.35: B Tree after Insertion

Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree:

1. Locate the leaf node.
2. If there are more than $m/2$ keys in the leaf node, then delete the desired key from the node.
3. If the leaf node doesn't contain $m/2$ keys, then complete the keys by taking the element from right or left sibling.
 - If the left sibling contains more than $m/2$ elements, then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
 - If the right sibling contains more than $m/2$ elements, then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

4. If neither of the sibling contain more than $m/2$ elements, then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
 5. If parent is left with less than $m/2$ nodes, then apply the above process on the parent, too.
- If the the node which is to be deleted is an internal node, then replace the node with its inorder successor or predecessor. Since, successor or predecessor will always be on the leaf node, hence the process will be similar as the node is being deleted from the leaf node.

Example 1: Delete the node 53 from the B tree of order 5 shown in the following figure:

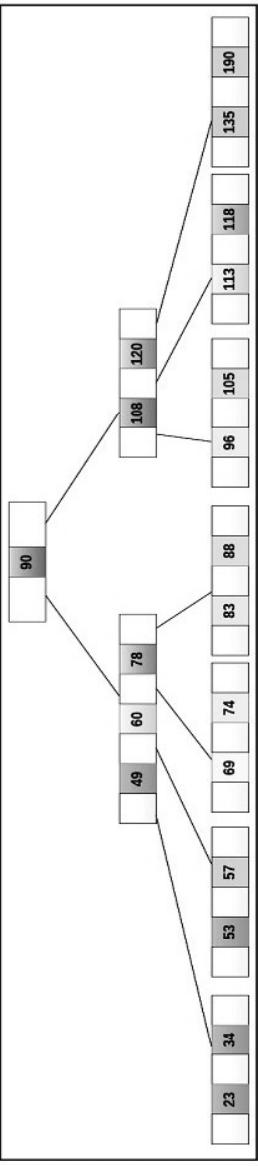


Fig. 4.36: Deletion Operation from the B Tree

53 is present in the right child of element 49. Delete it.

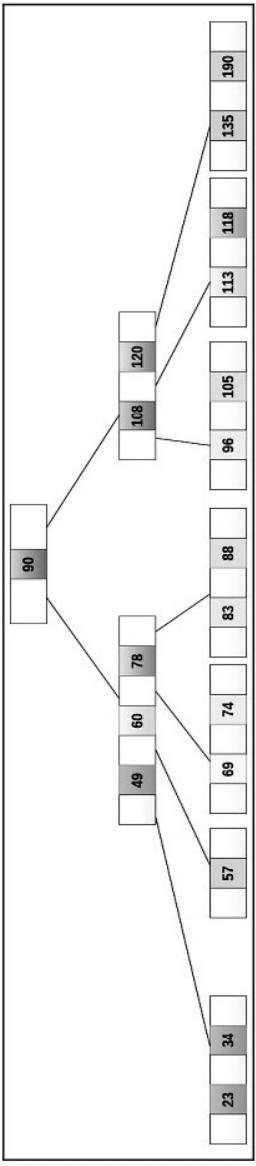


Fig. 4.37: Deletion the Node 53 from the B Tree

Now, 57 is the only element which is left in the node. The minimum number of elements that must be present in a B tree of order 5 is 2. It is less than that. The elements in its left and right subtree are also not sufficient, therefore merge it with the left sibling and intervening element of parent, i.e., 49.

The final B tree is shown as follows:

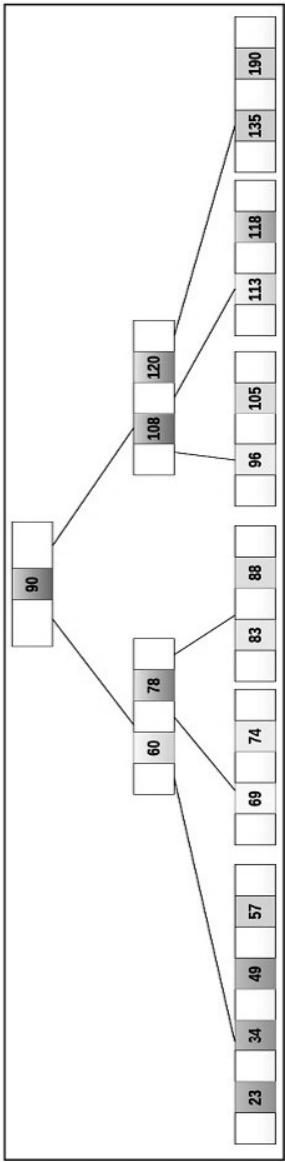


Fig. 4.38: Final B Tree

Application of B Tree

B tree is used to index the data and provides fast access to the actual data stored on the disks, since the access to value stored in a large database, that is stored on a disk, is a very time-consuming process.

Searching an unindexed and unsorted database containing n key values needs $O(n)$ running time in worst case. However, if we use B tree to index this database, it will be searched in $O(\log n)$ time in worst case.

B+ Tree

B+ tree is an extension of B tree which allows efficient insertion, deletion and search operations.

In B tree, keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes, while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked list to make the search queries more efficient.

B+ tree are used to store the large amount of data which cannot be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory, whereas leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure:

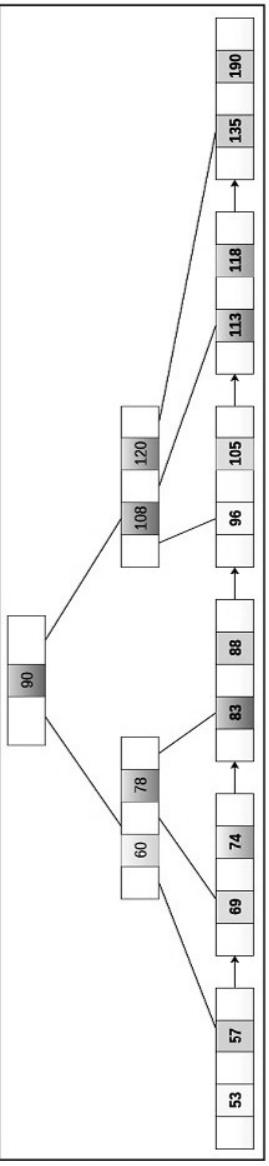


Fig. 4.39: B+ Tree

Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

Table 4.5: B Tree vs. B+ Tree

SN	B Tree	B+ Tree
1	Search keys cannot be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes.	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process, since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time-consuming.	Deletion will never be a complex process, since element will always be deleted from the leaf nodes.
5	Leaf nodes cannot be linked together.	Leaf nodes are linked together to make the search operations more efficient.

4.7 Sets and Dictionaries

Sets

A set is an unordered collection of (unique) items.

Examples:

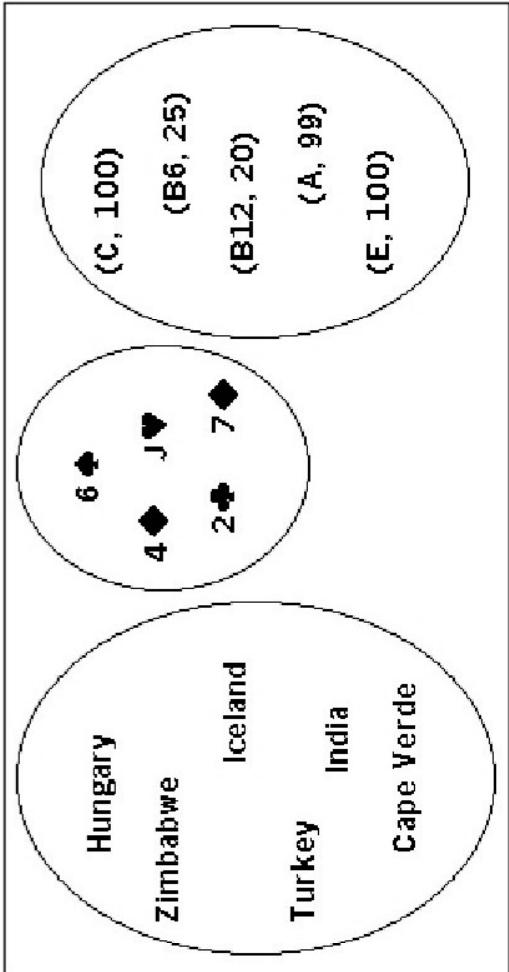


Fig. 4.40: Sets and Dictionaries

Operations

`s.add(e)`

Add element e to the set s.

`s.remove(e)`

Remove element e from the set s.

`s.contains(e)`

Returns whether e is a member of s.

`s.size()`

Returns the number of elements in s.

`s.isEmpty()`

Returns whether there are no elements in s.

s.addAll(t)

Adds all the elements of set t to set s.

s.removeAll(t)

Removes all the elements of t from s.

s.retainAll(t)

Removes all the elements from s that are not in t.

s.union(t)

Returns a new set which contains all the elements of set s and all the elements of set t, and no others.

s.intersect(t)

Returns a new set which contains all and only those elements in both s and t.

s.minus(t)

Returns a new set which contains all and only those elements in s but not in t.

s.symmetricMinus(t)

Returns a new set which contains all and only those elements in either set s or set t but not both.

Dictionaries

A dictionary (also called a map, associative array, hash, or lookup table) is a collection of key value pairs where every key is unique. We don't care as much about the pairs as we do about *looking up the value associated with a given key*.

Example:**Table 4.6: Dictionaries**

www.nato.int	152.152.96.1
cr.ypt.to	131.193.178.175
losangeles.citysearch.com	209.104.35.15
www.cl.cam.ac.uk	128.232.0.20
mit.edu	18.7.22.69
www.altan.ie	212.108.64.74



securitysearch.net	64.94.136.5
linuxassembly.org	64.202.189.170
regular-expressions.info	66.39.67.31
jpl.nasa.gov	137.78.160.180
groups.google.com	216.239.57.147
script.aculo.us	86.59.5.67

Say: www.nato.int maps to 152.152.96.1, or www.nato.int is bound to 152.152.96.1.

Note that a set is just a kind of dictionary in which the value part is ignored.

Operations

m.put(k,v)

Adds new key value pair to map m.

m.get(k)

Returns the value associated with key k in map m.

m.containsKey(k)

Returns whether there's a key value pair in map m with key k.

m.containsValue(v)

Returns whether there's a key value pair in map m with value v.

m.putAll(otherMap)

Adds all the key value pairs of otherMap to m.

m.keySet()

Returns the set of keys of m.

m.values()

Returns the values of m in some collection (not a set, since they don't have to be unique).

m.entrySet()

Returns the set of key value pairs in m.

m.remove(k)

Removes the key value pair with key k from map m.

m.clear()

Removes all the key value pairs from m.

m.size()

Returns the number of key value pairs in m.

m.isEmpty()

Returns whether there are zero pairs in map m.

Representation of Sets and Dictionaries

There are dozens of ways to implement these.

- Association Lists
 - Unsorted association lists
 - Sorted association lists
- Search Trees
 - General search trees
 - (a,b) trees
 - B trees and B+ trees
 - Binary search trees
 - Unbalanced binary search trees
 - Self-balancing binary search trees
 - AVL trees
 - Scapegoat trees
 - Red-Black trees
 - AA trees
 - Splay trees
 - Cartesian trees
- Skip Lists
- Tries and Radix (Patricia) Trees
- Hash Tables



- Bitsets
- Judy Arrays
- Bloom Filters

No one representation is best for all situations. You need to take into account:

- Will the collection be loaded once and only read and never written?
- Will insertions and deletions be frequent or uncommon, compared to lookups?
- Will the collection contain only a very small number of elements or can it be huge?
- Are there any restrictions on the keys? For example, are keys only strings? Or only integers?
- Will the keys come from a type that can be ordered?
- Is the collection subject to algorithmic complexity attacks?

4.8 Summary

A stack is a container of objects that are inserted and removed according to the Last in First out (LIFO) principle. In the push down stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. Push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A queue is a container of objects (a linear collection) that are inserted and removed according to the First in First out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line are made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

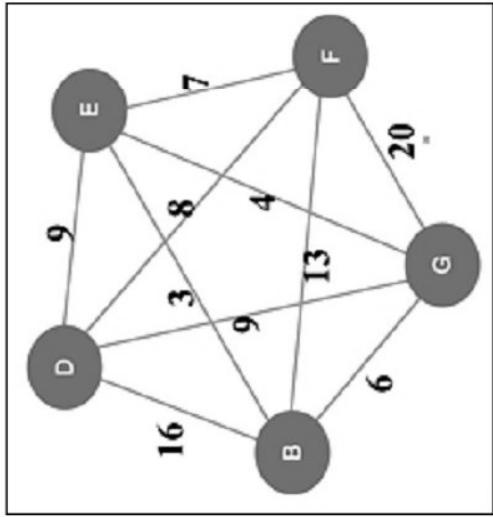


Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style. Tree is one of the most powerful and advanced data structures. It is a non-linear data structure compared to arrays, linked lists, stack and queue. It represents the nodes connected by edges.

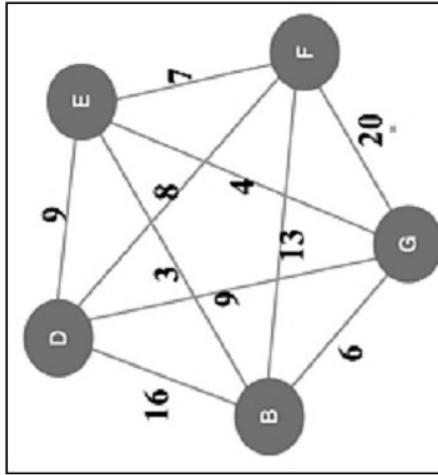
4.9 Key Words/Abbreviations

- **Data:** Data can be defined as an elementary value or the collection of values.

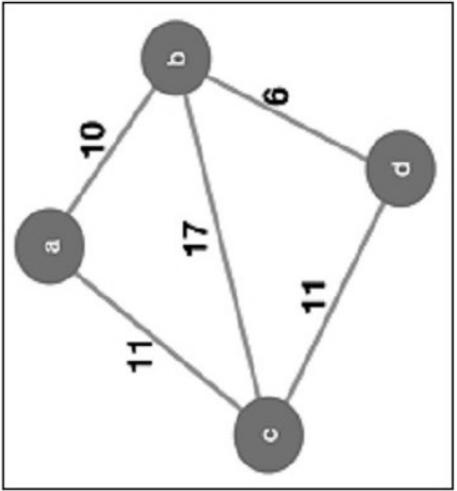
2. Consider the following graph. Using Kruskal's algorithm, which edge will be selected first?



3. Which of the following edges form minimum spanning tree on the graph using Kruskal's algorithm?

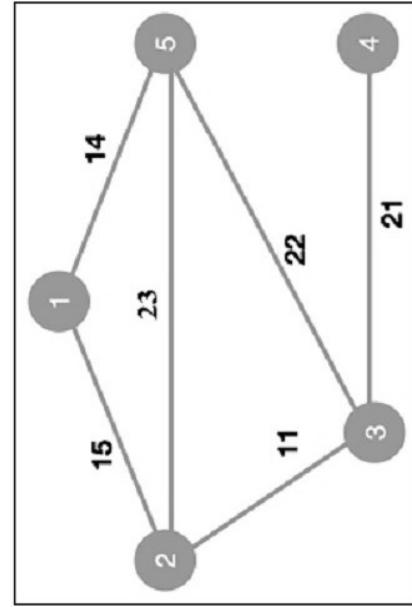


4. Consider the given graph.



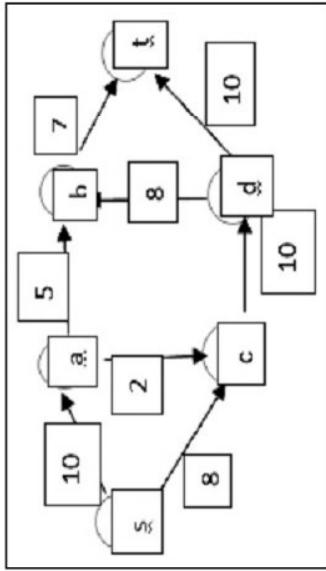
What is the weight of the minimum spanning tree using the Prim's algorithm, starting from vertex a?

5. Consider the graph shown below.



Which of the following edges form the MST of the given graph using Prim's algorithm, starting from vertex 4.

6. Find the maximum flow from the following graph:



7. What is a queue? How it is different from stack and how is it implemented?
8. What is stack and where can it be used?
9. What are infix, prefix, postfix notations?
10. What is a linked list and what are its types?

4.11 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. Entries in a stack are ‘ordered’. What is the meaning of this statement?
2. What is data structure?

3. When is a binary search best applied?
4. What is a linked list?
5. In what areas are data structures applied?
6. What is LIFO?
7. What is a queue?
8. What are binary trees?
9. What is a stack?
10. Explain binary search tree.
11. What is an ordered list?
12. What is the difference between a PUSH and a POP?
13. What is a linear search?

B. Multiple Choice/Objective Type Questions

1. A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a _____.
 - (a) Queue
 - (b) Stack
 - (c) Tree
 - (d) Linked list
2. The data structure required for breadth-first traversal on a graph is _____.
 - (a) Stack
 - (b) Array
 - (c) Queue
 - (d) Tree
3. A queue follows _____.
 - (a) FIFO (First in First out) principle
 - (b) LIFO (Last in First out) principle
 - (c) Ordered array
 - (d) Linear tree
4. Circular queue is also known as _____.
 - (a) Ring buffer
 - (b) Square buffer
 - (c) Rectangle buffer
 - (d) Curve buffer





10. Here is an infix expression: $4 + 3 * (6 * 3 - 12)$. Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation.

What is the maximum number of symbols that will appear on the stack at one time during the conversion of this expression?

- (a) 1 (b) 2
 (c) 3 (d) 4

Answers:

1. (a), 2. (c), 3. (a), 4. (a), 5. (a), 6. (d), 7. (c) 8. (b), 9. (d) 10. (d)

4.12 References

References of this unit have been given at the end of the book.



UNIT 5 DIVIDE AND CONQUER

Structure:

- 5.0 Learning Objectives
- 5.1 Introduction
- 5.2 Max - Min Problem
- 5.3 Binary Search
- 5.4 Merge Sort
- 5.5 Quick Sort
- 5.6 Advantages and Disadvantages of Divide and Conquer
- 5.7 Decrease and Conquer Approach
- 5.8 Topological Sort
- 5.9 Summary
- 5.10 Key Words/Abbreviations
- 5.11 Learning Activity
- 5.12 Unit End Questions (MCQ and Descriptive)
- 5.13 References

5.0 Learning Objectives

After studying this unit, you will be able to:

- Describe the concept of divide and conquer
- Explain various sorting and searching operations



- Explain merge sort in a practical way
- Describe binary search method
- Implement quick sort algorithm
- Elaborate advantages and disadvantages of divide and conquer method
- Understand topological sort

5.1 Introduction

Divide and conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide and Conquer Strategy.

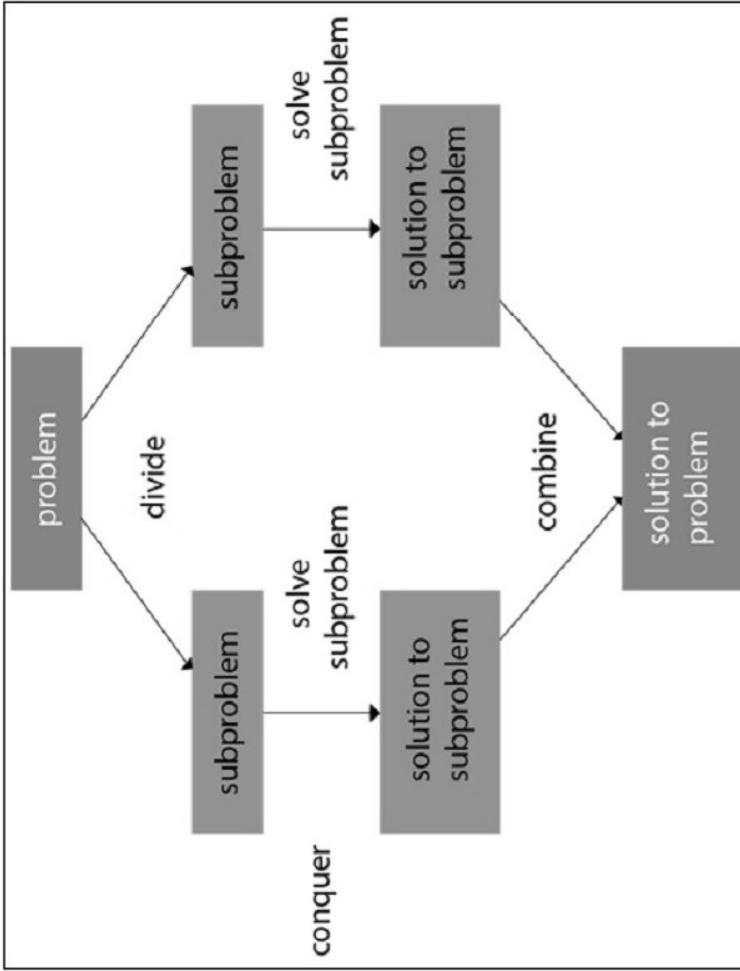


Fig. 5.1: Divide and Conquer Strategy

Generally, we can follow the divide and conquer approach in a three-step process.

Examples: The specific computer algorithms are based on the divide and conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamentals of Divide and Conquer Strategy

There are two fundamentals of divide and conquer strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of formula we apply D&C strategy, i.e., we break the problem recursively and solve the broken subproblems.

2. Stopping Condition: When we break the problem using divide and conquer strategy, we need to know that for how much time, we need to apply divide and conquer. So the condition where the need to stop our recursion steps of D&C is called as stopping condition.

5.2 Max - Min Problem

Problem: Analyse the algorithm to find the maximum and minimum element from an array.

Algorithm: Max-Min Element (a [])

```
Max: a [i]
Min: a [i]
For i= 2 to n do
If a[i]> max then
max = a[i]
if a[i] < min then
min: a[i]
return (max, min)
```



Analysis:

Method 1: If we apply the general approach to the array of size n , the number of comparisons required are $2n-2$.

Method 2: In another approach, we will divide the problem into subproblems and find the max and min of each group, now max. Of each group we will compare with the only max of another group and min with min.

Let $n =$ be the size of items in an array

Let $T(n) =$ time required to apply the algorithm on an array of size n . Here, we divide the terms as $T(n/2)$.

2 here tends to the comparison of the minimum with minimum and maximum with maximum as in above example.

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2 \\ T(n) &= 2 T\left(\frac{n}{2}\right) + 2 \end{aligned} \quad \dots \text{(Equation 1)}$$

$T(2) = 1$, time required to compare two elements/items. (Time is measured in units of the number of comparisons.)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + 2 \quad \dots \text{(Equation 2)}$$

Put eq (ii) in eq (i)

$$\begin{aligned} T(n) &= 2 \left[2T\left(\frac{n}{2^2}\right) + 2 \right] + 2 \\ &= 2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2 \end{aligned}$$

Similarly, apply the same procedure recursively on each subproblem or anatomy.

{Use recursion means, we will use some stopping condition to stop the algorithm.}

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + \dots + 2 \quad \dots \text{ (Equation 3)}$$

Recursion will stop, when

$$\frac{n}{2^i} = 2 \Rightarrow n = 2^{i+1} \quad \dots \text{ (Equation 4)}$$

Put the equation 4 into equation 3.

$$\begin{aligned} T(n) &= 2^i T(2) + 2^i + 2^{i-1} + \dots + 2 \\ &= 2^i \cdot 1 + 2^i + 2^{i-1} + \dots + 2 \\ &= 2^i + \frac{2(2^i - 1)}{2-1} \\ &= 2^{i+1} + 2^i - 2 \\ &= n + \frac{n}{2} - 2 \\ &= \frac{3n}{2} - 2 \end{aligned}$$

Number of comparisons requires applying the divide and conquering algorithm on n elements/items = $\frac{3n}{2} - 2$

Number of comparisons requires applying general approach on n elements = $(n - 1) + (n - 1)$

$$= 2n - 2$$

From this example, we can analyse how to reduce the number of comparisons by using this technique.

Analysis: Suppose we have the array of size 8 elements.

Method 1: Requires $(2n - 2), (2 \times 8) - 2 = 14$ comparisons.

Method 2: Requires $\frac{3 \times 8}{2} - 2 = 10$ comparisons.

It is evident that we can reduce the number of comparisons (complexity) by using a proper technique.

5.3 Binary Search

1. In binary search technique, we search an element in a sorted array by recursively dividing the interval in half.
2. Firstly, we take the whole array as an interval.
3. If the pivot element (the item to be searched) is less than the item in the middle of the interval, then we discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.
4. If the pivot element (the item to be searched) is greater than the item in the middle of the interval, then we discard the first half of the list and work recursively on the second half by calculating the new beginning and middle element.
5. Repeatedly, check until the value is found or interval is empty.

Analysis:

1. **Input:** An array A of size n, already sorted in the ascending or descending order.
2. **Output:** Analyse to search an element item in the sorted array of size n.
3. **Logic:** Let T(n) = number of comparisons of an item with n elements in a sorted array.
 - Set BEG = 1 and END = n
 - Find mid = $\text{int} \left(\frac{\text{beg} + \text{end}}{2} \right)$
 - Compare the search item with the mid item.

Case 1: item = A [mid], then LOC = mid, but it the best case and $T(n) = 1$

Case 2: item \neq A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$.

And again find the midpoint of the half-sorted array and compare it with the search element.

Repeat the same process until a search element is found.

$$T(n) = T\left(\frac{n}{2}\right) + 1 \quad \dots(\text{Equation 1})$$

{Time to compare the search element with mid element, then with half of the selected half part of array.}

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1, \text{ putting } \frac{n}{2} \text{ in place of } n.$$

$$\text{Then we get: } T(n) = \left(T\left(\frac{n}{2^2}\right) + 1 \right) + 1 \dots \text{ By putting } T\left(\frac{n}{2}\right) \text{ in (1) equation}$$

$$T(n) = T\left(\frac{n}{2^2}\right) + 2 \quad \dots(\text{Equation 2})$$

$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1 \dots \text{ Putting } \frac{n}{2} \text{ in place of } n \text{ in equation (1).}$$

$$T(n) = T\left(\frac{n}{2^3}\right) + 1 + 2$$

$$T(n) = T\left(\frac{n}{2^3}\right) + 3 \quad \dots(\text{Equation 3})$$

$$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1 \dots \text{ Putting } \frac{n}{3} \text{ in place of } n \text{ in equation (1)}$$

$$\text{Put } T\left(\frac{n}{2^3}\right) \text{ in equation (3)}$$

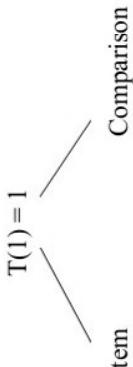
$$T(n) = T\left(\frac{n}{2^4}\right) + 4$$

Repeat the same process ith times

$$T(n) = T\left(\frac{n}{2^i}\right) + i \dots$$

Stopping Condition: $T(1) = 1$

At least there will be only one term left that's why that term will compare out, and only one comparison be done that's why



Is the last term of the equation and it will be equal to 1

$$\frac{n}{2^i} = 1$$

$\frac{n}{2^i}$ is the last term of the equation and it will be equal to 1

$$\frac{n}{2^i} = 1$$

Applying log both sides

$$\log n = \log_2 i$$

$$\log n = i \log 2$$

$$\frac{\log n}{\log 2} = 1$$

$$\log_2 n = i$$

$$T(n) = T\left(\frac{n}{2^i}\right) + 1$$

$\frac{n}{2^i} = 1$ as in equation 5

$$= T(1) + i$$

$T(1) = 1$ by stopping condition

$$= 1 + \log_2 n$$

$$= \log_2 n$$

(1 is a constant that's why ignore it)

Therefore, binary search is of order $O(\log_2 n)$.

5.4 Merge Sort

It closely follows the divide and conquer paradigm.

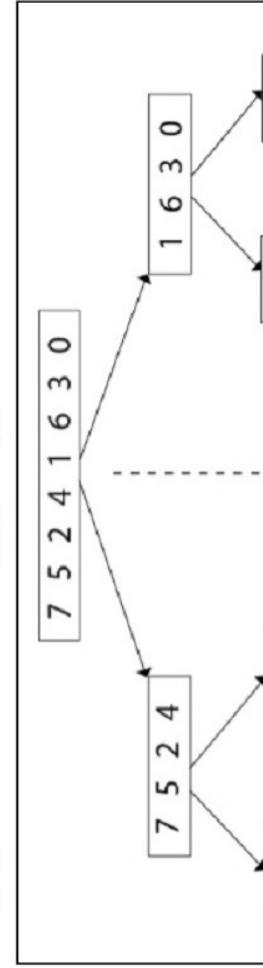
Conceptually, it works as follows:

1. **Divide:** Divide the unsorted list into two sublists of about half the size.
 2. **Conquer:** Sort each of the two sublists recursively until we have list sizes of length 1, in which case the list items are returned.
 3. **Combine:** Join the two sorted sublists back into one sorted list.
- The main purpose is to sort the unsorted list in non-decreasing order.

ALGORITHM - MERGE SORT

1. If $p < r$
2. Then $q \leftarrow (p + r)/2$
3. MERGE SORT (A, p, q)
4. MERGE SORT ($A, q + 1, r$)
5. MERGE (A, p, q, r)

The following figure illustrates the dividing (splitting) procedure.



FUNCTIONS:

MERGE (A, p, q, r)

1. $n1 = q - p + 1$
2. $n2 = r - q$
3. create arrays [1 $n1 + 1$] and R [1 $n2 + 1$]
4. for $i \leftarrow 1$ to $n1$
5. do $[i] \leftarrow A[p + i - 1]$
6. for $j \leftarrow 1$ to $n2$
7. do $R[j] \leftarrow A[q + j]$
8. $L[n1 + 1] \leftarrow \infty$
9. $R[n2 + 1] \leftarrow \infty$
10. $I \leftarrow 1$
11. $J \leftarrow 1$
12. for $k \leftarrow p$ to r
13. do if $L[i] \leq R[j]$
14. then $A[k] \leftarrow L[i]$
15. $i \leftarrow i + 1$
16. else $A[k] \leftarrow R[j]$
17. $j \leftarrow j + 1$



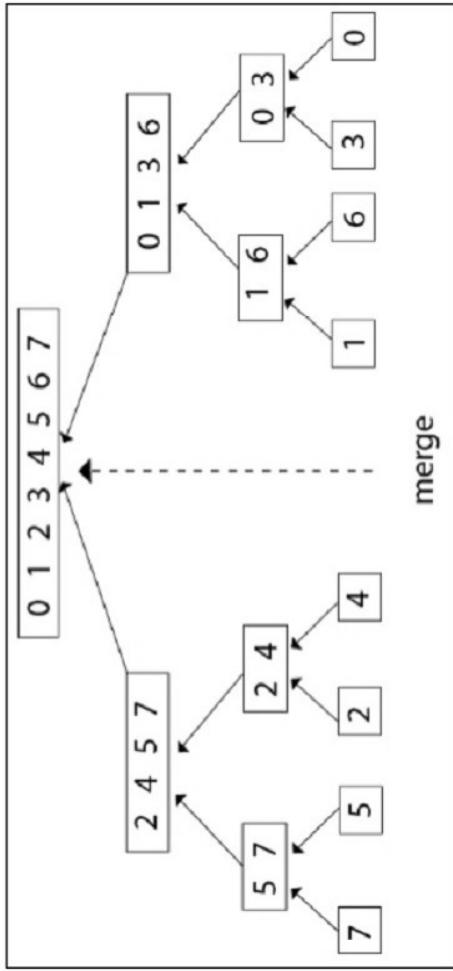


Fig. 5.3: Merge Sort: Combine Phase

In this method, we split the given list into two halves. Then recursively analyse merge sort and dividing. We get many sorted lists.

At last, we combine the sorted lists.

Let $T(n)$ be the total time taken in merge sort

1. Sorting two halves will be taken at the most $2T \frac{n}{2}$ time
 2. When we merge the sorted lists, we have a total $n-1$ comparison because the last element which will be left will just need to be copied down in the combined list and there will be no comparison.

So, the relational formula becomes

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But, we ignore ‘-1’ because the element will take some time to be copied in merge lists.

$$S_0, T(n) = 2T\left(\frac{n}{2}\right) + n \quad \dots \text{(Equation 1)}$$

Note: Stopping condition $T(1) = 0$ because at last there will be only one element left which need to be copied and there will be no comparison.

Putting $n = \frac{n}{2}$ in place of n in (Equation 1)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} \quad \dots(\text{Equation 2})$$

Put equation 2 in equation 1

$$\begin{aligned} T(n) &= 2 \left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2} \right] + n \\ &= 2^2 T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n \\ T(n) &= 2^2 T\left(\frac{n}{2^2}\right) + 2n \end{aligned} \quad \dots(\text{Equation 3})$$

Putting $n = \frac{n}{2^2}$ in equation 1

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} \quad \dots(\text{Equation 4})$$

Putting equation 4 in equation 3

$$\begin{aligned} T(n) &= 2^2 \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} \right] + 2n \\ T(n) &= 2^3 T\left(\frac{n}{2^3}\right) + n + 2n \\ T(n) &= 2^3 T\left(\frac{n}{2^3}\right) + 3n \end{aligned} \quad \dots(\text{Equation 5})$$

From equation 1, equation 3, equation 5, we get

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in \quad \dots(\text{Equation 6})$$

From stopping condition:

$$\frac{n}{2^i} = 1 \quad \text{and} \quad T\left(\frac{n}{2^i}\right) = 0$$

$$n = 2^i$$

Apply log both sides:

$$\log n = \log_2 i$$

$$\log n = i \log_2$$

$$\frac{\log n}{\log_2} = i$$

$$\log_2 n = i$$

From equation 6

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n \cdot n$$

$$= T(n) = n \cdot \log n$$

5.5 Quick Sort

It is an algorithm of divide and conquer type.

Divide: Rearrange the elements and split arrays into two subarrays and an element in between search that each element in left subarray is less than or equal to the average element and each element in the right subarray is larger than the middle element.

Conquer: Recursively, sort two subarrays.

Combine: Combine the already sorted array.

Algorithm:

QUICK SORT (array A, int m, int n)

```
1   if (n > m)
2     then
3       i ← a random index from [m,n]
4       swap A [i] with A[m]
5       o ← PARTITION (A, m, n)
6       Quicksort (A, m, o - 1)
7       Quicksort (A, o + 1, n)
```

Partition Algorithm

Partition algorithm rearranges the subarrays in a place.

PARTITION (array A, int m, int n)

```
1   x ← A[m]
2   o ← m
3   for p ← m + 1 to n
4     do if (A[p] < x)
5       then o ← o + 1
6       swap A[o] with A[p]
7       swap A[m] with A[o]
8   return o
```

The following figure shows the execution trace partition algorithm.

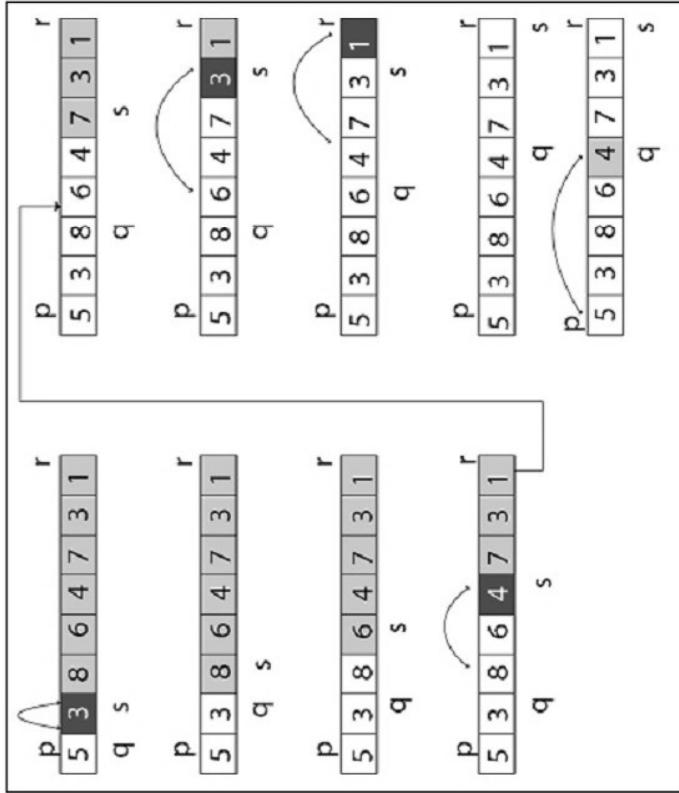


Fig. 5.4: Execution Trace Partition Algorithm

Example of Quick Sort

44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left-side elements and the elements must be **greater** than **44**, then swap them. As **55** is greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 55 88

Recursively, repeating step 1 and step 2 until we get two lists; one left from pivot element **44** and one right from pivot element.

22 33 11 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22	33	11	40	44	90	77	60	99	55	88
----	----	----	----	-----------	----	----	----	----	----	----

Now, the element on the right side and left side are greater than and smaller than **44**, respectively.

Now we get two sorted lists:

Sublist1				Sublist2							
22	33	11	40	44	90	77	60	99	55	88	

And these sublists are sorted under the same process as done above.

These two sorted sublists are given side by side:

22	33	11	40	44	90	77	60	99	55	88
11	33	22	40	44	88	77	60	99	55	90
11	22	33	40	44	88	77	60	90	55	99

First sorted list

Sublist3				Sublist4							
55	77	60	88	90	99						
55	60	77									

Sorted

55 77 **60**

55 60 77

Sorted

Merging Sublists:

11	22	33	40	44	55	60	77	88	90	99
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------



Is the average case complexity of quick sort for sorting n elements?

Quick Sort [Best Case]:

In any sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

Table 5.1: Complexity of Quick Sort for Sorting n Elements

Method Name	Equation	Stopping Condition	Complexities
1. Quick Sort [Worst Case]	$T(n) = T(n - 1) + T(0) + n$	$T(1) = 0$	$T(n) = n^2$
2. Quick Sort [Average Case]	$T(n) = n + 1 + \frac{1}{n} \left(\sum_{k=1}^n T(k-1) + T(n-k) \right)$		$T(n) = n \log n$

5.6 Advantages and Disadvantages of Divide and Conquer

Pros and Cons of Divide and Conquer Approach

Divide and conquer approach supports parallelism as subproblems are independent. Hence, an algorithm which is designed using this technique can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

5.7 Decrease and Conquer Approach

As divide and conquer approach is already discussed, which include the following steps:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough; however, just solve the subproblems in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

The decrease and conquer approach works similarly. It also includes following steps:

- Decrease or reduce problem instance to smaller instance of the same problem and extend solution.

- Conquer the problem by solving smaller instance of the problem.
- Extend solution of smaller instance to obtain solution to original problem.

Basic idea of the decrease and conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. This approach is also known as incremental or inductive approach.

‘Divide and Conquer’ vs. ‘Decrease and Conquer’:

As per Wikipedia, some authors consider that the name ‘divide and conquer’ should be used only when each problem may generate two or more subproblems. The name decrease and conquer has been proposed instead for the single subproblem class. According to this definition, merge sort and quick sort comes under divide and conquer (because there are two subproblems) and binary search comes under decrease and conquer (because there is one subproblem).

Implementations of Decrease and Conquer:

This approach can be either implemented as top-down or bottom-up.

Top-down Approach: It always leads to the recursive implementation of the problem.

Bottom-up Approach: It is usually implemented in an iterative way, starting with a solution to the smallest instance of the problem.

Variations of Decrease and Conquer:

There are three major variations of decrease and conquer.

1. Decrease by a constant
2. Decrease by a constant factor
3. Variable size decrease

Decrease by a Constant: In this variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one, although other constant size reductions do happen occasionally. Below are example problems:

- Insertion sort
- Graph search algorithms: DFS, BFS
- Topological sorting
- Algorithms for generating permutations, subsets

Decrease by a Constant Factor: This technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. A reduction by a factor other than two is especially rare.

Decrease by a constant factor algorithms are very efficient, especially when the factor is greater than two as in the fake coin problem. Below are example problems:

- Binary search
- Fake coin problems
- Russian peasant multiplication

Variable Size Decrease: In this variation, the size reduction pattern varies from one iteration of an algorithm to another.

As, in problem of finding GCD of two numbers though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor. Below are example problems:

- Computing median and selection problem
- Interpolation search
- Euclid's algorithm

There may be a case that a problem can be solved by decrease by constant as well as decrease by factor variations, but the implementations can be either recursive or iterative. The iterative implementations may require more coding effort. However, they avoid the overload that accompanies recursion.

5.8 Topological Sort

Topological sort is an ordering of the vertices that is not contrary to the dependencies. In other words, if you were to perform one job at a time, a topological sort is one order that you could perform the jobs.

How to generate the topological sort?

From the definition (same method as your DS textbook), there must be a source vertex. The vertex with no dependencies, i.e., in-degree is zero.

Why must there be a source? Otherwise, the graph would have a cycle or be infinite.

The algorithm uses an `incounter` initially equal to the in-degree of the vertices. When `incounter` goes to zero for a vertex, it is placed on the list for the topological sort and all dependent adjacent vertices' `incounter` is reduced by one.

Basically, the algorithm reduces the size of the graph by one each time a source is removed.

Algorithm $\text{TopoSource}(G(V, E))$

```
make empty Container of vertices, C
make empty List of vertices, S
for each vertex v in V do
    incounter(v) ← in-deg(v)
    if incounter(v) == 0 then C.add(v)
while C is not empty do
    u ← C.remove()
    S.add(u)
    for each edge (u, w) do // edge is out from u to w
        incounter(w) ← incounter(w)-1
        if incounter(w) == 0 then C.add(w)
    if L.size() == V.size() then return S
    else return "G has a cycle"
```

What is the cost? The size of the data structure for G.

There is another technique with use of a DFS of the DAG.

When the vertices are popped off the recursive stack, they are added to the list of the topological sort. Then the list is read in reverse order.

Is the algorithm correct? What is last the vertex to be popped off the stack? The source.

What is the first vertex to be popped off the stack? A dead end.

The next vertex to be popped off is a dead end of the reduced graph (graph less the vertex that has been popped off).

This is always the case or the vertex.

5.9 Summary

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

The **quick sort** is internal sorting method where the data is **sorted** in main memory. The **merge sort** is external **sorting** method in which the data that is to be **sorted** cannot be accommodated in the memory and needs auxiliary memory for **sorting**.

Merge sort is a divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The **merge()** function is used for merging two halves. The merge (arr, l, m, r) is the key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.

The approach decrease and conquer include the following steps:

1. **Decrease** or reduce problem instance to smaller instance of the same problem and extend the solution.
2. **Conquer** the problem by solving smaller instance of the problem.
3. **Extend** solution of smaller instance to obtain solution to original problem.

Basic idea of the decrease and conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. This approach is also known as incremental or inductive approach.

5.10 Key Words/Abbreviations

- **Divide:** Divide the original problem into a set of subproblems.
- **Conquer:** Solve every subproblem individually, recursively.
- **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.
- **Relational Formula:** It is the formula that we generate from the given technique.
- **Stopping Condition:** It is when we break the problem using divide and conquer strategy.

5.11 Learning Activity

1. Choose the correct code for merge sort.

(a)

```
void merge_sort(int arr[], int left, int right)
{
    if (left > right)
    {
        int mid = (right-left)/2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid+1, right);

        merge(arr, left, mid, right); //function to merge sorted arrays
    }
}
```

(b)

```
void merge_sort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left+(right-left)/2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid+1, right);

        merge(arr, left, mid, right); //function to merge sorted arrays
    }
}
```

(c)

```
void merge_sort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left+(right-left)/2;
        merge(arr, left, mid, right); //function to merge sorted arrays
        merge_sort(arr, left, mid);
        merge_sort(arr, mid+1, right);
    }
}
```

(d)

```
void merge_sort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = (right-left)/2;
```



```
merge(arr, left, mid, right); //function to merge sorted arrays  
merge_sort(arr, left, mid);  
merge_sort(arr, mid+1, right);  
}  
}
```

2. What is recurrence for worst case of quick sort and what is the time complexity in worst-case?

3. Suppose we have a $O(n)$ time algorithm that finds median of an unsorted array. Now consider a quick sort implementation where we first find median using the above algorithm, then use median as pivot. What will be the worst-case time complexity of this modified quick sort.

4. Given an unsorted array, the array has this property that every element in array is at most k distance from its position in sorted array, where k is a positive integer smaller than size of array. Which sorting algorithm can be easily modified for sorting this array and what is the obtainable time complexity?

5. Consider the following program fragment for reversing the digits in a given integer to obtain a new integer. Let $n = D_1D_2\dots D_m$

```
int n, rev;  
rev = 0;
```

```

while (n > 0)
{
    rev = rev*10 + n%10;
    n = n/10;
}

```

The loop invariant condition at the end of the i^{th} iteration is:

6. What is the worst-case time complexity of insertion sort where position of the data to be inserted is calculated using binary search?

5.12 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. What will be the best-case time complexity of merge sort?
2. What is the worst-case time complexity of a quick sort algorithm?
3. Find the pivot element from the given input using median of three partitioning method.
8, 1, 4, 9, 6, 3, 5, 2, 7, 0.
4. Define quick sort algorithm?
5. What are the advantages and disadvantages of divide and conquer method?
6. What is meant by topological sort? Give an example.
7. What is binary search? Give an example.
8. What are the advantages of merge sort?

B. Multiple Choice/Objective Type Questions

1. Merge sort uses which of the following technique to implement sorting?
 - (a) Backtracking
 - (b) Greedy algorithm
 - (c) Divide and conquer
 - (d) Dynamic programming
2. What is the average case time complexity of merge sort?
 - (a) $O(n \log n)$
 - (b) $O(n^2)$
 - (c) $O(n^2 \log n)$
 - (d) $O(n \log n^2)$
3. What is the auxiliary space complexity of merge sort?
 - (a) $O(1)$
 - (b) $O(\log n)$
 - (c) $O(n)$
 - (d) $O(n \log n)$
4. Merge sort can be implemented using $O(1)$ auxiliary space.
 - (a) True
 - (b) False
5. What is the worst-case time complexity of merge sort?
 - (a) $O(n \log n)$
 - (b) $O(n^2)$
 - (c) $O(n^2 \log n)$
 - (d) $O(n \log n^2)$
6. Which is the safest method to choose a pivot element?
 - (a) Choosing a random element as pivot
 - (b) Choosing the first element as pivot
 - (c) Choosing the last element as pivot
 - (d) Median of three partitioning method
7. What is the average running time of a quick sort algorithm?
 - (a) $O(N^2)$
 - (b) $O(N)$
 - (c) $O(N \log N)$
 - (d) $O(\log N)$



8. Which of the following sorting algorithms is used along with quick sort to sort the subarrays?
 - (a) Merge sort
 - (b) Shell sort
 - (c) Insertion sort
 - (d) Bubble sort
9. Quick sort uses joint operation rather than merge operation.
 - (a) True
 - (b) False
10. How many subarrays does the quick sort algorithm divide the entire array into?
 - (a) One
 - (b) Two
 - (c) Three
 - (d) Four

Answers:

1. (c), 2. (a), 3. (c), 4. (a), 5. (a), 6. (a), 7. (c), 8. (c), 9. (a), 10. (b)

5.13 References

References of this unit have been given at the end of the book.



UNIT 6 GREEDY METHOD 1

Structure:

- 6.0 Learning Objectives
- 6.1 Introduction
- 6.2 General Method
- 6.3 Coin Change Problem
- 6.4 Knapsack Problem
- 6.5 Job Sequencing with Deadlines
- 6.6 Minimum Cost Spanning Trees
- 6.7 Methods of Minimum Spanning Tree
 - 6.7.1 Kruskal's Algorithm
 - 6.7.2 Prim's Algorithm
- 6.8 Summary
- 6.9 Key Words/Abbreviations
- 6.10 Learning Activity
- 6.11 Unit End Questions (MCQ and Descriptive)
- 6.12 References

6.0 Learning Objectives

After studying this unit, you will be able to:

- Define coin change problem



- Describe knapsack problem
- Define job sequencing with deadlines
- Elaborate minimum cost spanning tree using Prim's algorithm and Kruskal's algorithm
- Understand job sequencing with deadlines

6.1 Introduction

In an algorithm design there is no one ‘silver bullet’ that is a cure for all computation problems. Different problems require the use of different kinds of techniques. A good programmer uses all these techniques based on the type of problem. Some commonly used techniques are:

1. Divide and conquer
2. Randomised algorithms
3. Greedy algorithms (This is not an algorithm, it is a technique.)
4. Dynamic programming

What is a ‘Greedy Algorithm’?

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How do you decide which choice is optimal?

Assume that you have an objective function that needs to be optimised (either maximised or minimised) at a given point. A greedy algorithm makes greedy choices at each step to ensure that the objective function is optimised. The greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

Greedy algorithms have some advantages and disadvantages:

1. It is quite easy to come up with a greedy algorithm (or even multiple greedy algorithms) for a problem.



2. Analysing the run time for greedy algorithms will generally be much easier than for other techniques (like divide and conquer). For the divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of it gets smaller and the number of subproblems increases.
3. The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Note: Most greedy algorithms are not correct. An example is described later in this section.

How to create a greedy algorithm?

Being a very busy person, you have exactly T time to do some interesting things and you want to do maximum of such things.

You are given an array A of integers, where each element indicates the time a thing takes for completion. You want to calculate the maximum number of things that you can do in the limited time that you have.

This is a simple greedy algorithm problem. In each iteration, you have to greedily select the things which will take the minimum amount of time to complete while maintaining two variables **currentTime** and **numberOfThings**. To complete the calculation, you must:

1. Sort the array A in a non-decreasing order.
2. Select each to-do item one by one.
3. Add the time that it will take to complete that to-do item into **currentTime**.
4. Add one to **numberOfThings**.

Repeat this as long as the **currentTime** is less than or equal to T.

Let A = {5, 3, 4, 2, 1} and T = 6

After sorting, A = {1, 2, 3, 4, 5}

After the 1st iteration:

- **currentTime** = 1
- **numberOfThings** = 1

After the 2nd iteration:

- **currentTime** is $1 + 2 = 3$
- **numberOfThings** = 2

After the 3rd iteration:

- **currentTime** is $3 + 3 = 6$
- **numberOfThings** = 3

After the 4th iteration:

- **currentTime** is $6 + 4 = 10$, which is greater than T.

Therefore, the answer is 3.

Where to use greedy algorithms?

A problem must comprise these two components for a greedy algorithm to work:

1. It has **optimal substructures**. The optimal solution for the problem contains optimal solutions to the subproblems.
2. It has a **greedy property** (hard to prove its correctness!). If you make a choice that seems the best at the moment and solve the remaining subproblems later, you still reach an optimal solution. You will never have to reconsider your earlier choices.

For example:

1. Activity selection problem
2. Fractional knapsack problem
3. Scheduling problem

Examples:

The greedy method is quite powerful and works well for a wide range of problems. Many algorithms can be viewed as applications of the greedy algorithms, such as (includes but is not limited to):

1. Minimum spanning tree
2. Dijkstra's algorithm for shortest paths from a single source
3. Huffman codes (data compression codes)

6.2 General Method

“Greedy Method finds out many options, but you have to choose the best option.”

In this method, we have to find out the best method/option out of many present ways.

In this approach/method, we focus on the first stage and decide the output. Don't think that this method may or may not give the best output.

Greedy algorithm solves problems by making the best choice that seems best at the particular moment. Many optimisation problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:

1. **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating ‘greedy’ choices.
2. **Optimal Substructure:** Optimal solutions contain optimal sub-solutions. In other words, answers to subproblems of an optimal solution are optimal.

Example:

1. Machine scheduling
2. Fractional knapsack problem
3. Minimum spanning tree



4. Huffman code
5. Job sequencing
6. Activity selection problem

Steps for achieving a greedy algorithm are:

1. **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
2. **Local Optimal Choice:** In this the choice should be the optimum which is selected from the currently available choices.
3. **Unalterable:** Once the decision is made, at any subsequent step that option is not altered.

6.3 Coin Change Problem

The Coin Change Problem is considered by many to be essential for understanding the paradigm of programming known as **Dynamic Programming**. The two often are always paired together because the coin change problem encompasses the concepts of dynamic programming.

In other words, dynamic problem is a method of programming that is used to simplify a problem into smaller pieces. For example, if you were asked simply what is $3 * 89?$, you perhaps would not know the answer off your head as you probably know what is $2 * 2$. However, if you knew what was $3 * 88$ (264), then certainly you can deduce $3 * 89$. All you would have to do is add 3 to the previous multiple and you would arrive at the answer of 267. Thus, that is a very simple explanation of what is dynamic programming and perhaps you can now see how it can be used to solve large time complexity problems effectively.

By keeping the above definition of dynamic programming in mind, we can now move forward to the coin change problem. The following is an example of one of the many variations of the coin change problem. Given a list of coins, i.e., 1 cent, 5 cents and 10 cents, can you determine the total number of permutations of the coins in the given list to make up the number N?



Example 1: Suppose you are given the coins 1 cent, 5 cents, and 10 cents with $N = 8$ cents, what are the total number of permutations of the coins you can arrange to obtain 8 cents?

Input:	N=8
Coins:	1, 5, 10
Output:	2
Explanation: 1 way: $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8$ cents.	
2 way: $1 + 1 + 1 + 5 = 8$ cents.	

All you're doing is determining all of the ways you can come up with the denomination of 8 cents. Eight 1 cents added together is equal to 8 cents. Three 1 cent plus one 5 cents added is 8 cents. So, there are a total of two ways given the list of coins 1, 5 and 10 to obtain 8 cents.

Example 2: Suppose you are given the coins 1 cent, 5 cents and 10 cents with $N = 10$ cents what

N = 12
Index of Array: [0, 1, 2]
Array of Coins: [1, 5, 10]

This is a array of coins, 1 cent, 5 cents, and 10 cents. The N is 12 cents. So we need to come up with a method that can use those coin values and determine the number of ways by which we can make 12 cents.

Thinking dynamically, we need to figure out how to add to previous data. So what that means is we have to add to previous solutions instead of recalculating over the same values. Clearly, we have to iterate through the entire array of coins. We also need a way to see if a coin is larger than the N value.

One way to do this is having an array that counts all the way up to the Nth value.

So ...

Array of Ways

[0, 0, 0 Nth value] in our case it would be up to 12.
--

The reason for having an array up to the Nth value is so that we can determine the number of ways the coins make up the values at the index of array of ways. We do this because if we can determine a coin is larger than that value at the index, then clearly we can't use that coin to determine the permutations of the coins because that coin is larger than that value. This can be better understood with an example.

Using the above numbers as example.

N = 12
Index of Array of Coins: [0, 1, 2]
Array of Coins: [1, 5, 10]

Index of Array of Ways: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Array of Ways: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]



Before we start iterating, we have to give a predefined value to our array of ways. We must set the first element at index 0 of the array of ways to 1. This is because there is one way to make the number 0, using 0 coins.

So, if we started iterating through all the array of coins and compare the elements to the array of ways, we will determine how many times a coin can be used to make the values at the index of the array of ways.

For example, first set ways[0] = 1.

Lets compare the first coin, 1 cent.

```
N = 12  
Index of Array of Coins: [0, 1, 2]  
Array of Coins: [1, 5, 10]  
Index of Array of Ways: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]  
Array of Ways: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Then compare coins[0] to all of the index's of ways array. If the value of the coin is less than or equal to the ways index, then ways[j]-coins[i]+ways[j] is the new value of ways[j]. We do this because we are trying to break each part down into smaller pieces. You will see what is happening as you continue to read. So comparing each value of the ways index to the first coin, we get the following.

```
Index of Array of Ways: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]  
Array of Ways: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Lets now compare the second coin, 5 cents.

```
N = 12  
Index of Array of Coins: [0, 1, 2]  
Array of Coins: [1, 5, 10]  
Index of Array of Ways: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]  
Array of Ways: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Comparing 5 cents to each of the index and making that same comparison, if the value of the coin is smaller than the value of the index at the ways array, then $\text{ways}[j]-\text{coins}[j]+\text{ways}[j]$ is the new value of $\text{ways}[j]$. Thus, we get the following.

Index of Array of Ways: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Array of Ways: [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3]

We are determining how many times the second coin goes into all of the values leading up the Nth coin. Why are we using all of the coins? It is to check our previous result dynamically and update our answer instead of recalculating all over again.

For example, take the element at index 10, the answer is 3 so far. But how did we get 3? We know that the value of $10-5$ is 5 so that is our $j-\text{coins}[i]$ value, that is the difference of what needs to be made up to make the amount 10. So we look at index 5 of the ways array and see it has the value 2, for the same reason as above, there are so far two ways to obtain the value 5. So, if there are two ways to obtain the value 5, then those ways plus the current number of ways is the new updated value of the total ways to get the value at index 10.

Lets now compare the third coin, 10 cents.

$N = 12$

Index of Array of Coins: [0, 1, 2]

Array of Coins: [1, 5, 10]

Comparing 10 cents to each of the index and making that same comparison, if the value of the coin is smaller than the value of the index at the ways array, then $\text{ways}[j]-\text{coins}[j]+\text{ways}[j]$ is the new value of $\text{ways}[j]$.

Thus, we get the following.

Index of Array of Ways: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Array of Ways: [1, 1, 1, 1, 2, 2, 2, 2, 4, 4, 4]

So, the answer to our example is $\text{ways}[12]$ which is 4.

6.4 Knapsack Problem

Knapsack problem can be further divided into two types:

The 0/1 Knapsack Problem

In this type, each package can be taken or not taken. Besides, the thief cannot take a fractional amount of a taken package or take a package more than once. This type can be solved by dynamic programming approach.

Brief Introduction of Dynamic Programming

In the divide-and-conquer strategy, you divide the problem to be solved into subproblems. The subproblems are further divided into smaller subproblems. That task will continue until you get subproblems that can be solved easily. However, in the process of such division, you may encounter the same problem many times.

The basic idea of dynamic programming is to use a table to store the solutions of solved subproblems. If you face a subproblem again, you just need to take the solution in the table without having to solve it again. Therefore, the algorithms designed by dynamic programming are very effective.

Problem --> Sub-problem --> Smallest Sub-problem

To solve a problem by dynamic programming, you need to do the following tasks:

- Find solutions of the smallest subproblems.
- Find out the formula (or rule) to build a solution of subproblem through solutions of even smallest subproblems.
- Create a table that stores the solutions of subproblems. Then calculate the solution of subproblem according to the found formula and save to the table.
- From the solved subproblems, you find the solution of the original problem.

Analyze the 0/1 Knapsack Problem

When analyzing this type, you can find some noticeable points. The value of the knapsack algorithm depends on two factors:



1. How many packages are being considered
2. The remaining weight which the knapsack can store.

Therefore, you have two variable quantities.

With dynamic programming, you have useful information:

1. The objective function will depend on two variable quantities
2. The table of options will be a 2-dimensional table.

If calling $B[i][j]$ is the maximum possible value by selecting in packages $\{1, 2, \dots, i\}$ with weight limit j .

- The maximum value when selected in n packages with the weight limit M is $B[n][M]$. In other words: When there are i packages to choose, $B[i][j]$ is the optimal weight when the maximum weight of the knapsack is j .

- The optimal weight is always less than or equal to the maximum weight: $B[i][j] \leq j$.

For example: $B[4][10] = 8$. It means that in the optimal case, the total weight of the selected packages is 8, when there are 4 first packages to choose from (1st to 4th package) and the maximum weight of the knapsack is 10. It is not necessary that all 4 items are selected.

Formula to Calculate $B[i][j]$

Input, you define:

- $W[i]$, $V[i]$ are in turn the weight and value of package i , in which $i \in \{1, \dots, n\}$.
- M is the maximum weight that the knapsack can carry.

In the case of simply having only 1 package to choose. You calculate $B[1][j]$ for every j : which means the maximum weight of the knapsack \geq the weight of the 1st package

$$B[1][j] = W[1]$$

With the weight limit j , the optimal selections among packages $\{1, 2, \dots, i-1, i\}$ to have the largest value will have two possibilities:



- If package i is not selected, $B[i][j]$ is the maximum possible value by selecting among packages $\{1, 2, \dots, i-1\}$ with weight limit of j. You have:

$$B[i][j] = B[i-1][j]$$

- If package i is selected (of course only consider this case when $W[i] \leq j$) then $B[i][j]$ is equal to the value $V[i]$ of package i plus the maximum value can be obtained by selecting among packages $\{1, 2, \dots, i-1\}$ with weight limit $(j - W[i])$. That is, in terms of the value you have:

$$B[i][j] = V[i] + B[i-1][j - W[i]]$$

Due to the creation of $B[i][j]$, which is the maximum possible value, $B[i][j]$ will be the max of the above 2 values.

Basis of Dynamic Programming

So, you have to consider if it is better to choose package i or not. From there you have the recursive formula as follows:

$$B[i][j] = \max(B[i-1][j], V[i] + B[i-1][j - W[i]])$$

It is easy to see $B[0][j] = \text{maximum value possible by selecting from 0 package} = 0$.

Calculate the Table of Options: You build a table of options based on the above recursive formula. To check if the results are correct (if not exactly, you rebuild the objective function $B[i][j]$). Through the creation of the objective function $B[i][j]$ and the table of options, you will orient the tracing.

Table of options B includes $n + 1$ lines, $M + 1$ columns,

- Firstly, filled with the basis of dynamic programming: Line 0 includes all zeros.
- Using recursive formulas use line 0 to calculate line 1, use line 1 to calculate line 2, etc. ... until all lines are calculated.



	0	1	...	M
0	0	0	0	0
1				
2				
...	...			
n				

Fig. 6.1: Table of Options

Trace

When calculating the table of options, you are interested in $B[n][M]$ which is the maximum value obtained when selecting in all n packages with the weight limit M .

- If $B[n][M] = B[n - 1][M]$ then package n is not selected, you trace $B[n - 1][M]$.
- If $B[n][M] \neq B[n - 1][M]$, you notice that the optimal selection has the package n and trace $B[n - 1][M - W[n]]$.

Continue to trace until reaching row 0 of the table of options.

Algorithm to look up the table of options to find the selected packages

Note: If $B[i][j] = B[i - 1][j]$, the package i is not selected. $B[n][W]$ is the optimal total value of package put into the knapsack.

Steps for tracing:

- **Step 1:** Starting from $i = n, j = M$.
- **Step 2:** Look in column j , up from bottom, you find the line i such that $B[i][j] > B[i - 1][j]$. Mark selected package i : Select $[i] = \text{true}$;
- **Step 3:** $j = B[i][j] - W[i]$. If $j > 0$, go to step 2, otherwise go to step 4
- **Step 4:** Based on the table of options to print the selected packages.

Java Code

```
public void knapsackDyProg(int W[], int V[], int M, int n)
{
    int B[][] = new int[n + 1][M + 1];
    for (int i=0; i<=n; i++)
        for (int j=0; j<=M; j++)
    {
        B[i][j] = 0;
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= M; j++)
        {
            B[i][j] = B[i - 1][j];
            if ((j >= W[i - 1]) && (B[i][j] < B[i - 1][j - W[i - 1]] + V[i - 1]))
            {
                B[i][j] = B[i - 1][j - W[i - 1]] + V[i - 1];
            }
            System.out.print(B[i][j] + " ");
        }
        System.out.print("\n");
    }
    System.out.println("Max Value:\t" + B[n][M]);
    System.out.println("Selected Packs: ");
    while (n != 0)
    {
        if (B[n][M] != B[n - 1][M])
        {
            System.out.println("\tPackage " + n + " with W = " + W[n - 1] + " and
Value = " + V[n - 1]);
            M = M - W[n - 1];
        }
    }
}
```

```

public void knapsackDyProg(int w[], int v[], int m, int n) {
    int B[][] = new int[n + 1][m + 1]; 1

    for (int i=0; i<=n; i++) {
        for (int j=0; j<=m; j++) {
            B[i][j] = 0;
        }
    }

    for (int i = 1; i <= n; i++) { 2
        for (int j = 0; j <= m; j++) {
            B[i][j] = B[i - 1][j]; 3

            if ((j >= w[i - 1]) && (B[i][j] < B[i - 1][j - w[i - 1]] + v[i - 1])) {
                B[i][j] = B[i - 1][j - w[i - 1]] + v[i - 1]; 4
            }
        }

        System.out.print(B[i][j] + " ");
    }

    System.out.print("\n");
}

System.out.println("Max Value:\t" + B[n][m]);

System.out.println("Selected Packs: ");
while (n != 0) { 5
    if (B[n][m] != B[n - 1][m]) {
        System.out.println("\tPackage " + n + " with w = " + w[n - 1] + " and value = " + v[n - 1]);
        m = m - w[n - 1]; 6
    }
    n--;
}
}

```

Fig. 6.2: Function knapsackDyProg() in Java

Explanation of code:

1. Create table B[][]]. Set default value for each cell is 0.
2. Build table B[][] in bottom-up manner. Calculate the table of options with the retrieval formula.
3. Calculate B[i][j]. If you do not select package i.
4. Then evaluate: if you select package i, it will be more beneficial then reset B[i][j].
5. Trace the table from row n to row 0.
6. If you choose package n. Once select package n, can only add weight M - W[n - 1].



In this section, you have two examples. Here is java code to run the above program with two examples:

```
public void run() {
    /*
     * Pack and Weight - Value
     */
    //int W[] = new int[]{3, 4, 5, 9, 4};
    int W[] = new int[]{12, 2, 1, 1, 4};
    //int V[] = new int[]{3, 4, 4, 10, 4};
    int V[] = new int[]{4, 2, 1, 2, 10};
    /*
     * Max Weight
     */
    //int M = 11;
    int M = 15;
    int n = V.length;
    /*
     * Run the algorithm
     */
    knapsackDyProg(W, V, M, n);
}
```

You have the output:

- First Example:

```
0 0 0 3 3 3 3 3 3 3 3  
0 0 0 3 4 4 7 7 7 7  
0 0 0 3 4 4 7 7 8 8  
0 0 0 3 4 4 7 7 10 10  
0 0 0 3 4 4 7 8 10 11  
Max Value: 11
```

Selected Packs:

```
Package 5 with W = 4 and V = 4  
Package 2 with W = 4 and V = 4  
Package 1 with W = 3 and V = 3
```

- Second Example:

```
0 0 0 0 0 0 0 0 0 0 4 4 4  
0 0 2 2 2 2 2 2 2 2 4 6 6  
0 1 2 3 3 3 3 3 3 3 4 5 6 7  
0 2 3 4 5 5 5 5 5 6 7 8  
0 2 3 4 10 12 13 14 15 15 15 15 15 15
```



Max Value: 15

Selected Packs:

Package 5 with W = 4 and V= 10

Package 4 with W = 1 and V= 2

Package 3 with W = 1 and V= 1

Package 2 with W = 2 and V= 2

Fractional Knapsack

Fractions of items can be taken rather than having to make binary (0-1) choices for each item.

Fractional knapsack problem can be solvable by greedy strategy, whereas 0-1 problem is not.

Steps to Solve the Fractional Problem:

1. Compute the value per pound for each item.
2. Obeying a greedy strategy, we take as much as possible of the item with the highest value per pound.
3. If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.
4. Sorting, the items by value per pound, the greedy algorithm run in $O(n \log n)$ time.

Fractional Knapsack (Array v, Array w, int W)

1. for i = 1 to size (v)
2. do p [i] = v [i] / w [i]
3. Sort Descending (p)
4. i \leftarrow 1
5. while (W>0)
6. do amount = min (W, w [i])
7. solution [i] = amount
8. W= W-amount
9. i \leftarrow i + 1
10. return solution

Example: Consider five items along their respective weights and values:

$$I = (I_1, I_2, I_3, I_4, I_5)$$

$$w = (5, 10, 20, 30, 40)$$

$$v = (30, 20, 100, 90, 160)$$

The capacity of knapsack $W = 60$

Now fill the knapsack according to the decreasing value of v_i .

First, we choose the item I_1 whose weight is 5.

Then choose item I_3 whose weight is 20. Now, the total weight of knapsack is $20 + 5 = 25$.

Now the next item is I_5 , and its weight is 40, but we want only 35, so we chose the fractional part of it,

$$\text{i.e., } 5 \times \frac{5}{5} + 20 \times \frac{20}{20} + 40 \times \frac{35}{40}$$

$$\text{Weight} = 5 + 20 + 35 = 60$$

Maximum Value:

$$30 \times \frac{5}{5} + 100 \times \frac{20}{20} + 160 \times \frac{35}{40}$$

$$= 30 + 100 + 140 = 270 \text{ (Minimum Cost)}$$

Solution:

Table 6.1: 5 Items Along their Respective Weights

ITEM	w_i	v_i
I1	5	30
I2	10	20
I3	20	100
I4	30	90
I5	40	160

Taking value per weight ratio, i.e., $p_i = \frac{v_i}{w_i}$



Table 6.2: Per Weight Ratio

ITEM	w _i	v _i	p _i = $\frac{v_i}{w_i}$
11	5	30	6.0
12	10	20	2.0
13	20	100	5.0
14	30	90	3.0
15	40	160	4.0

Now, arrange the value of p_i in decreasing order.

Table 6.3: Arrange the Value of p_i in Decreasing Order

ITEM	w _i	v _i	p _i = $\frac{v_i}{w_i}$
11	5	30	6.0
13	20	100	5.0
15	40	160	4.0
14	30	90	3.0
12	10	20	2.0

6.5 Job Sequencing with Deadlines

Problem Statement

In job sequencing problem, the objective is to find a sequence of jobs, which are completed within their deadlines and give maximum profit.

Solution

Let us consider a set of n given jobs, which are associated with deadlines and profit is earned if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of ith job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.



Thus, $\$D(i) > 0\$$ for $\$1 \leq i \leq n\$$.

Initially, these jobs are ordered according to profit, i.e., $\$p_1 \geq p_2 \geq \dots \geq p_n\$$.

Algorithm: Job Sequencing With Deadline (D, J, n, k)

```

D(0) := J(0) := 0
k := 1
J(1) := 1 // means first job is selected
for i = 2 ... n do
    r := k
    while D(J(r)) > D(i) and D(J(r)) ≠ r do
        r := r - 1
    if D(J(r)) ≤ D(i) and D(i) > r then
        for l = k ... r + 1 by -1 do
            J(l + 1) := J(l)
            J(r + 1) := i
            k := k + 1

```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is $\$O(n^2)\$$.

Example: Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Table 6.4: Set of Given Jobs

Job	J ₁	J ₂	J ₃	J ₄	J ₅
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution:

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table:

Table 6.5: Sorted Jobs According to their Profit

Job	J ₂	J ₁	J ₄	J ₃	J ₅
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J₂, as it can be completed within its deadline and contributes maximum profit.

- Next, J₁ is selected as it gives more profit compared to J₄.
 - In the next clock, J₄ cannot be selected as its deadline is over, hence J₃ is selected as it executes within its deadline.
 - The job J₅ is discarded as it cannot be executed within its deadline.
- Thus, the solution is the sequence of jobs (J₂, J₁, J₃), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is $100 + 60 + 20 = 180$.

6.6 Minimum Cost Spanning Trees

Introduction of Minimum Spanning Tree

Tree: A tree is a graph with the following properties:

1. The graph is connected (can go from anywhere to anywhere).
2. t is not cyclic (acyclic).

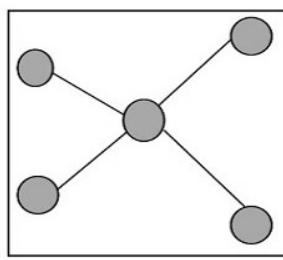


Fig. 6.3: Tree

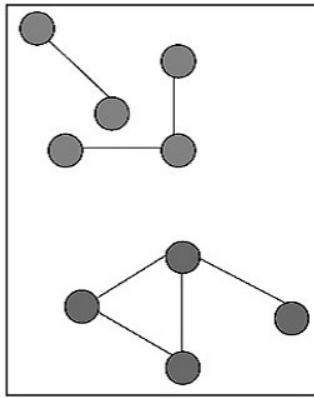


Fig. 6.4: Graph That are Not Trees

Spanning Tree:

Given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and joining all vertices. A single graph can have many spanning trees.

For Example:

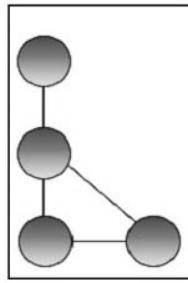


Fig. 6.5: Connected Undirected Graph

For the above connected graph there can be multiple spanning trees like these:

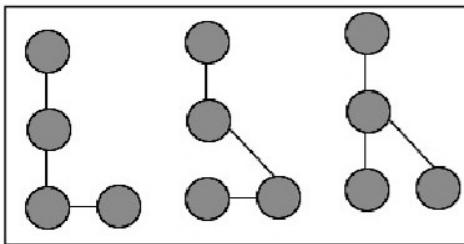


Fig. 6.6: Spanning Trees

Properties of Spanning Tree

1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't span the tree, or it can't span all the vertices.
6. Spanning tree doesn't contain cycles.
7. Spanning tree has **(n-1) edges** where n is the number of vertices.

Addition of even one single edge results in the spanning tree losing its property of acyclicity and elimination of one single edge results in its losing the property of connectivity.

Minimum Spanning Tree

Minimum Spanning tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combined with each edge, then the cost of the spanning tree would be the sum of the cost of its edges.

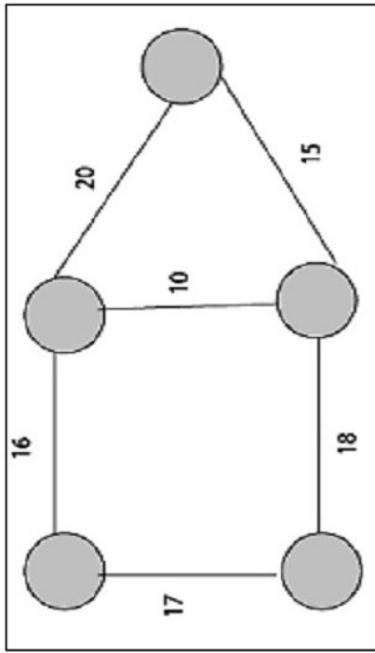
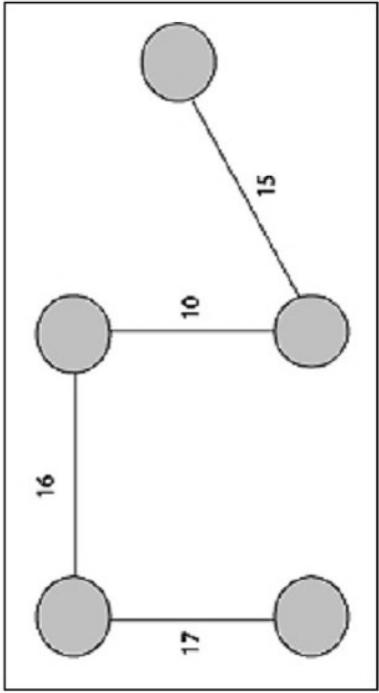


Fig. 6.7: Connected, Undirected Graph



$$\text{Total Cost} = 17 + 16 + 10 + 15 = 58$$

Fig. 6.8: Minimum Cost Spanning Tree

6.7 Methods of Minimum Spanning Tree

There are two methods to find minimum spanning tree

1. Kruskal's Algorithm
2. Prim's Algorithm

6.7.1 Kruskal's Algorithm

It is an algorithm to construct a minimum spanning tree for a connected weighted graph. It is a greedy algorithm. The greedy choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

If the graph is not linked, then it finds a minimum spanning tree.

Steps for Finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially, add each edge which does not result in a cycle, until $(n - 1)$ edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

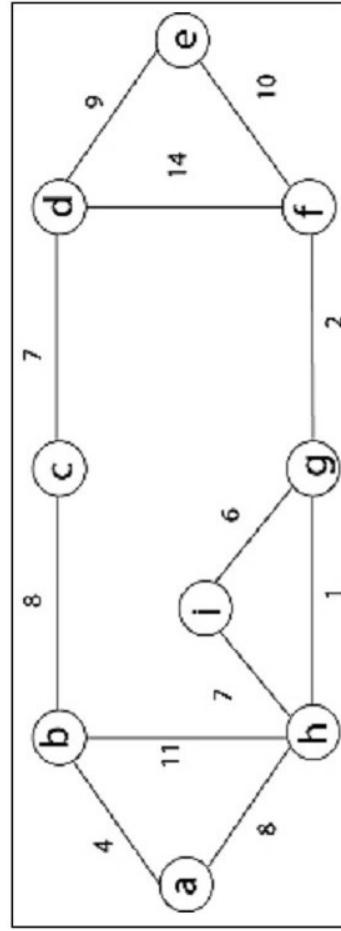
1. A $\leftarrow \emptyset$
2. for each vertex $v \in V [G]$
3. do MAKE - SET (v)
4. sort the edges of E into non-decreasing order by weight w
5. for each edge $(u, v) \in E$, taken in non-decreasing order by weight w
6. do if FIND-SET (μ) \neq if FIND-SET (v)
7. then A $\leftarrow A \cup \{(u, v)\}$
8. UNION (u, v)
9. return A

Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or simply, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is $O(\log V)$.
- If we ignore isolated vertices, which will each be the components of the minimum spanning tree, $V \leq 2E$, so $\log V$ is $O(\log E)$.

Thus, the total time is $O(E \log E) = O(E \log V)$.

For Example: Find the minimum spanning tree of the following graph using Kruskal's algorithm.



Solution: First we initialise the set A to the empty set and create $|v|$ trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed $(9-1) = 8$ edges

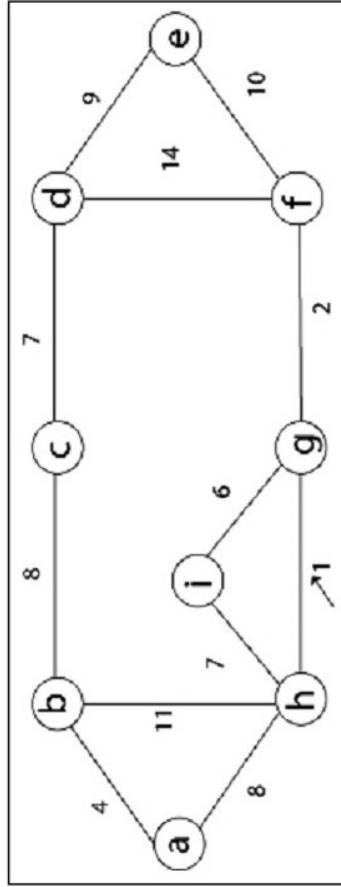
Table 6.6: MST

Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	j	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

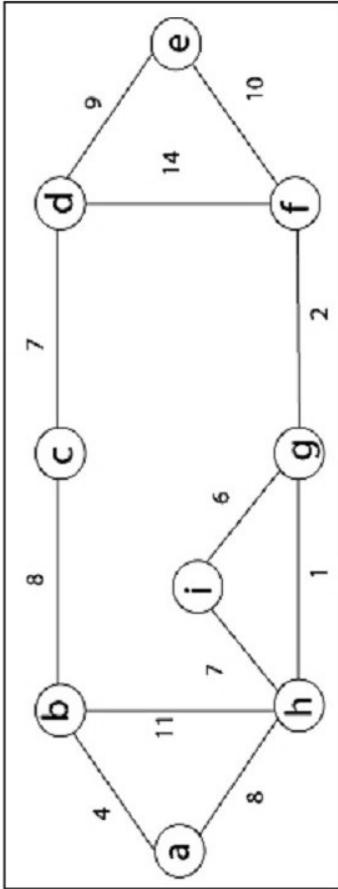
Now, check for each edge (u, v) whether the end points u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

Step 1: First take (h, g) edge

Step 2: Then (g, f) edge.

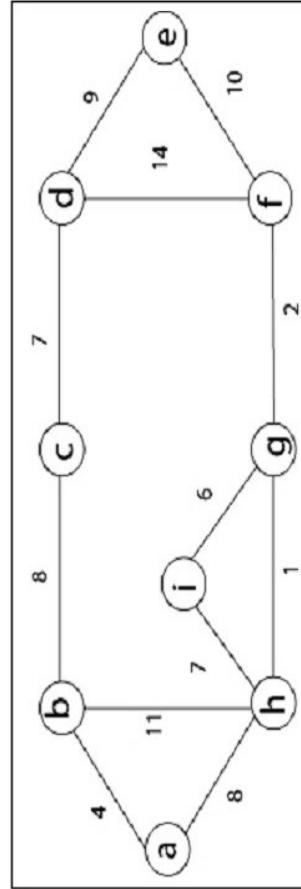


Step 3: Then (a, b) and (i, g) edges are considered, and the forest becomes:



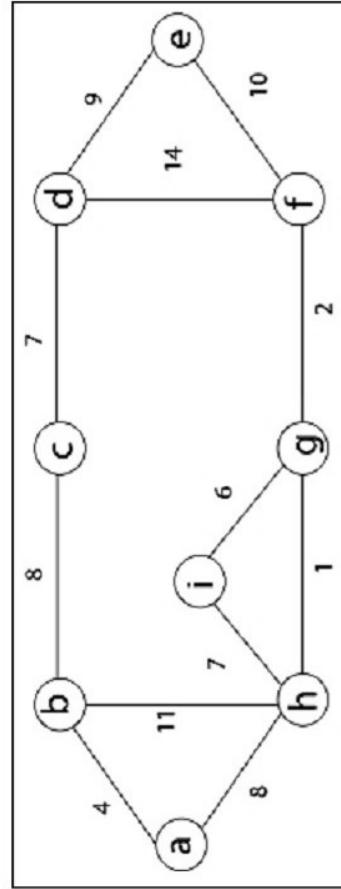
Step 4: Now, take edge (h, i). Both h and i vertices are in the same set. Thus, it creates a cycle.
So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes:



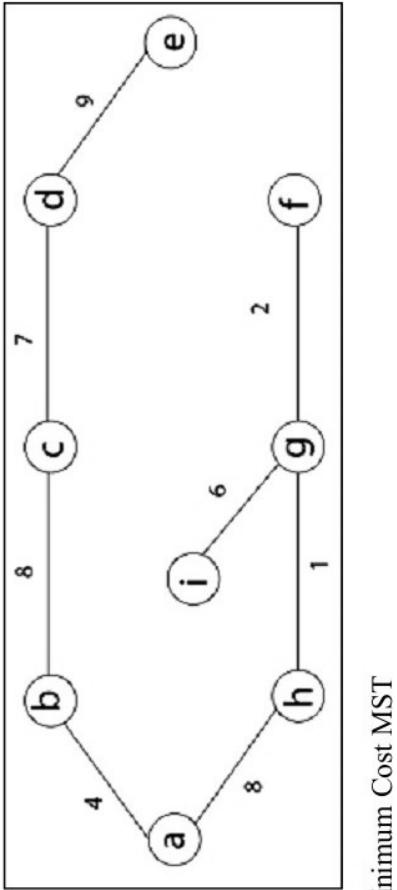
Step 5: In (e, f) edge both end points e and f exist in the same tree so discard this edge. Then (b, h) edge, which also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required minimum spanning tree because it contains all the 9 vertices and $(9 - 1) = 8$ edges

$e \rightarrow f$, $b \rightarrow h$, $d \rightarrow f$ [cycle will be formed]



Minimum Cost MST

6.7.2 Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other end point of edge to set containing MST.

Steps for Finding MST using Prim's Algorithm:

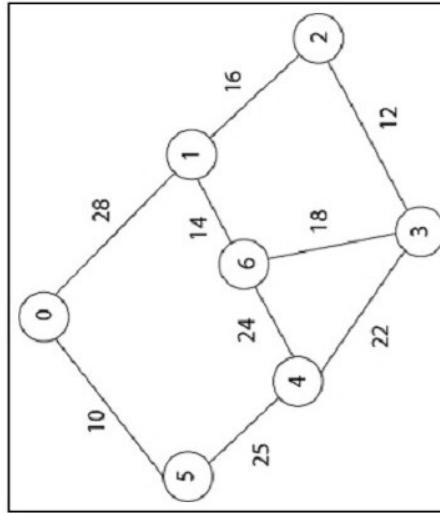
1. Create MST set that keeps a track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialise all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
 - (a) Pick vertex u which is not there in MST set and has minimum key value. Include 'u' to MST set.

- (b) Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v is less than the previous key value of v, update key value as a weight of u.v.

MST-PRIM (G, w, r)

1. for each $u \in V[G]$
2. do $\text{key}[u] \leftarrow \infty$
3. $\pi[u] \leftarrow \text{NIL}$
4. $\text{key}[r] \leftarrow 0$
5. $Q \leftarrow V[G]$
6. while $Q \neq \emptyset$
7. do $u \leftarrow \text{EXTRACT-MIN}(Q)$
8. for each $v \in \text{Adj}[u]$
9. do if $v \in Q$ and $w(u, v) < \text{key}[v]$
10. then $\pi[v] \leftarrow u$
11. $\text{key}[v] \leftarrow w(u, v)$

Example: Generate minimum cost spanning tree for the following graph using Prim's algorithm.



Solution: In Prim's algorithm, first we initialise the priority queue Q to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT-MIN (Q) procure, now $u = r$ and $\text{Adj}[u] = \{5, 1\}$.

Removing u from set Q and add it to set $V - Q$ of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Table 6.7: Minimum Cost Spanning Tree

Vertex	0	1	2	3	4	5	6
Key	0	∞	∞	∞	∞	∞	∞
Value							
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Taking 0 as starting vertex

Root = 0

$\text{Adj}[0] = 5, 1$

Parent, $\pi[5] = 0$ and $\pi[1] = 0$

Key [5] = ∞ and key [1] = ∞

$w[0, 5] = 10$ and $w(0, 1) = 28$

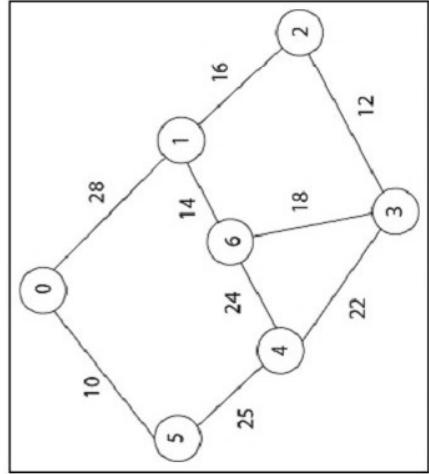
$w(u, v) < \text{key}[5]$, $w(u, v) < \text{key}[1]$

Key [5] = 10 and key [1] = 28

So update key value of 5 and 1 is:

Table 6.8: Update Key Value of 5 and 1

Vertex	0	1	2	3	4	5	6
Key	0	28	∞	∞	∞	10	∞
Value							
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN(Q) Remove 5 because key [5] = 10 which is minimum so u = 5.

Adj [5] = {0, 4} and 0 is already in heap

Taking 4, key [4] = ∞ $\pi[4] = 5$

$(u, v) < \text{key}[v]$ then key [4] = 25

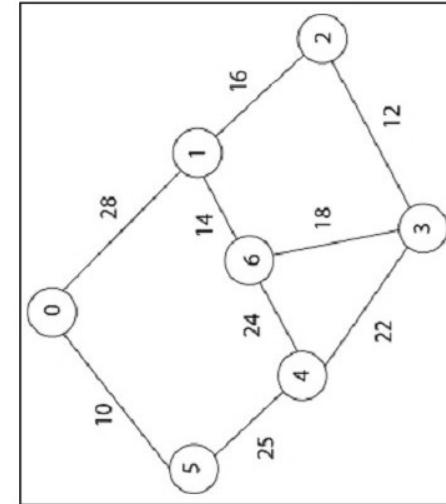
w(5,4) = 25

w(5,4) < key [4]

date key value and parent of 4.

Table 6.9: EXTRACT_MIN (Q)

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u = 4

Adj [4] = {6, 3}	key [6] = ∞
Key [3] = ∞	key [6] = ∞
w (4,3) = 22	w (4,6) = 24
w (u, v) < key [v]	w (u, v) < key [v]
w (4,3) < key [3]	w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

And the parent of 3, 6 as 4.

$\pi[3]=4$ $\pi[6]=4$

u = EXTRACT_MIN (3, 6)	[key [3] < key [6]]
u = 3	i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u = 3.

Adj [3] = {4, 6, 2}	
4 is already in heap	
4 \neq Q	key [6] = 24 now becomes key [6] = 18
Key [2] = ∞	key [6] = 24
w (3, 2) = 12	w (3, 6) = 18
w (3, 2) < key [2]	w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

$\pi[2]=3$ $\pi[6]=3$

Now by EXTRACT_MIN (Q) Remove 2, because key [2] = 12 is minimum.

u = EXTRACT_MIN (2, 6)	
u = 2	[key [2] < key [6]]
12 < 18	

Now the root is 2



$\text{Adj}[2] = \{3, 1\}$
 3 is already in a heap
 Taking 1, key [1] = 28
 $w(2,1) = 16$
 $w(2,1) < \text{key}[1]$

So update key value of key [1] as 16 and its parent as 2.

$\pi[1] = 2$

Now by EXTRACT_MIN(Q) Remove 1 because key [1] = 16 is minimum.

$\text{Adj}[1] = \{0, 6, 2\}$
 0 and 2 are already in heap.
 Taking 6, key [6] = 18
 $w[1, 6] = 14$
 $w[1, 6] < \text{key}[6]$

Update key value of 6 as 14 and its parent as 1.

$\Pi[6] = 1$

Now all the vertices have been spanned using the above table we get minimum spanning tree.

$0 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 6$

[Because $\Pi[5] = 0, \Pi[4] = 5, \Pi[3] = 5, \Pi[2] = 4, \Pi[1] = 3, \Pi[6] = 2, \Pi[0] = 1$]

Thus, the final spanning tree is

Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

6.8 Summary

The coin change Problem is considered by many to be essential for understanding the paradigm of programming known as **Dynamic Programming**. The two often are always paired together because the coin change problem encompasses the concepts of dynamic programming.



Knapsack Problem: In the supermarket, there are n packages ($n \leq 100$), the package i has weight $W[i] \leq 100$ and value $V[i] \leq 100$. A thief breaks into the supermarket, the thief cannot carry weight exceeding $M(M \leq 100)$. The problem to be solved here is: which packages the thief will take away to get the highest value?

Input:

Maximum weight M and the number of packages n .

Array of weight $W[i]$ and corresponding value $V[i]$.

Output:

Maximise value and corresponding weight in capacity.

Which packages the thief will take away.

Knapsack problem can be further divided into two types:

The 0/1 Knapsack Problem: In this type, each package can be taken or not taken. Besides, the thief cannot take a fractional amount of a taken package or take a package more than once. This type can be solved by dynamic programming approach.

Fractional Knapsack Problem: This type can be solved by greedy strategy.

Minimum-Cost Spanning Trees. Let $G = (V, E)$ be a connected graph in which each edge $(u, v) \in E$ has an associated cost $C(u, v)$. A spanning tree for G is a subgraph of G , that it is a free tree connecting all vertices in V . The cost of a spanning tree is the sum of costs on its edges.

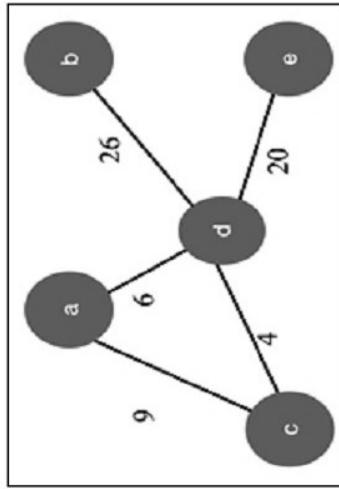
6.9 Key Words/Abbreviations

- **Tree:** A tree is a collection of elements called nodes.
- **Root:** This is the unique node in the tree in which further subtrees are attached.
- **Degree of Node:** The total number of subtrees attached to that node is called the degree of the node.
- **Leaf Nodes:** These are the terminal nodes of the tree.

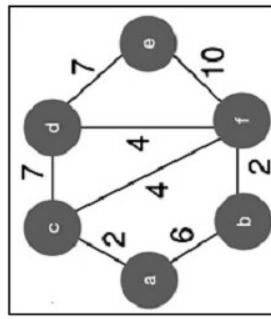
- **Internal Nodes:** The nodes in the tree which are other than leaf nodes and the root node are called as internal nodes.
- **Parent Node:** The node which is having further sub-branches is called the parent node of those sub-branches.
- **Height of the Tree:** The maximum level is the height of the tree.

6.10 Learning Activity

1. Consider the graph shown below. Which of the following are the edges in the MST of the given graph?

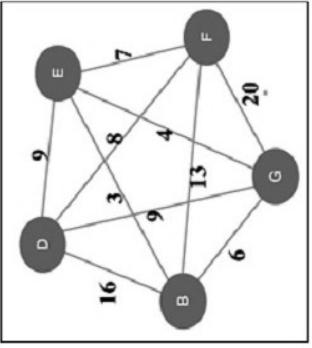


2. Consider the given graph.

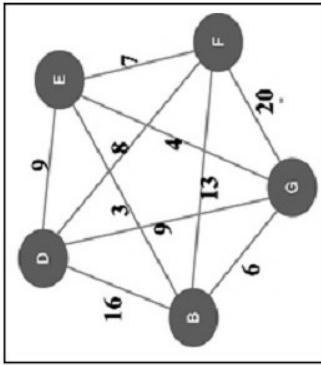


What is the weight of the minimum spanning tree using the Kruskal's algorithm?

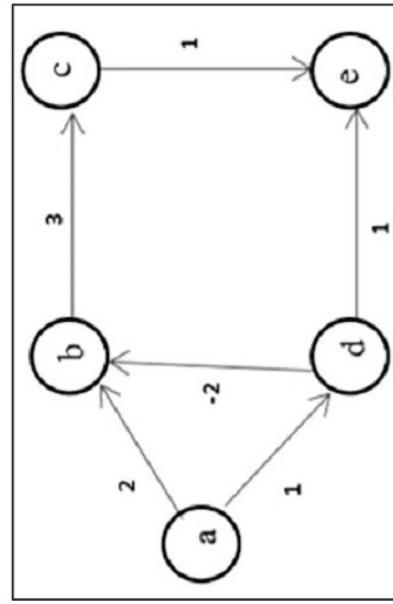
3. Consider the following graph. Using Kruskal's algorithm, which edge will be selected first?



4. Which of the following edges form minimum spanning tree on the graph using Kruskal's algorithm?



5. Consider the following graph:



What is the minimum cost to travel from node A to node C?

6.11 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. Define coin change Problem.
2. What is job sequencing with deadlines?
3. What is minimum cost spanning trees? Give example.
4. What is Prim's algorithm?
5. Define Knapsack problem.
6. Define Kruskal's algorithm.
7. What is coin change problem?

B. Multiple Choice/Objective Type Questions

1. The Knapsack problem is an example of _____.
 - (a) Greedy algorithm
 - (b) 2D dynamic programming
 - (c) 1D dynamic programming
 - (d) Divide and conquer
2. Which of the following methods can be used to solve the knapsack problem?
 - (a) Brute force algorithm
 - (b) Recursion
 - (c) Dynamic programming
 - (d) Brute force, recursion and dynamic programming
3. You are given a knapsack that can carry a maximum weight of 60. There are four items with weights {20, 30, 40, 70} and values {70, 80, 90, 200}. What is the maximum value of the items you can carry using the knapsack?
 - (a) 160
 - (b) 200
 - (c) 170
 - (d) 90

4. What is the time complexity of the brute force algorithm used to solve the knapsack problem?
- (a) $O(n)$
 - (b) $O(n!)$
 - (c) $O(2n)$
 - (d) $O(n^3)$
5. The 0-1 Knapsack problem can be solved using greedy algorithm.
- (a) True
 - (b) False
6. Which of the following is true?
- (a) Prim's algorithm initialised with a vertex
 - (b) Prim's algorithm initialised with edge
 - (c) Prim's algorithm initialised with a vertex which has the smallest edge
 - (d) Prim's algorithm initialised with a forest
7. Prim's algorithm is a _____.
- (a) Divide and conquer algorithm
 - (b) Greedy algorithm
 - (c) Dynamic Programming
 - (d) Approximation algorithm
8. Prim's algorithm resembles Dijkstra's algorithm.
- (a) True
 - (b) False
9. Kruskal's algorithm is best suited for the sparse graphs than Prim's algorithm.
- (a) True
 - (b) False
10. Prim's algorithm is also known as _____.
- (a) Dijkstra–Scholten algorithm
 - (b) Boruvka's algorithm
 - (c) Floyd–Warshall algorithm
 - (d) DJP Algorithm

Answers:

1. (b), 2. (d). 3. (a), 4. (c), 5. (b), 6. (a), 7. (b). 8. (a), 9. (a), 10. (d)

6.12 References

References of this unit have been given at the end of the book.



UNIT 7 GREEDY METHOD 2

Structure:

- 7.0 Learning Objectives
- 7.1 Introduction
- 7.2 Single-source Shortest Paths: Dijkstra's Algorithm
- 7.3 Optimal Tree Problem: Huffman Trees and Codes
- 7.4 Transform and Conquer Approach: Heaps and Heap Sort
- 7.5 Summary
- 7.6 Key Words/Abbreviations
- 7.7 Learning Activity
- 7.8 Unit End Questions (MCQ and Descriptive)
- 7.9 References

7.0 Learning Objectives

After studying this unit, you will be able to:

- Explain optimal tree problems
- Define heaps and explain heap sort
- Understand single-source shortest paths
- Describe Dijkstra's algorithm
- Describe all pairs shortest paths concepts
- Understand Huffman trees and codes
- Ensure conquer approach



7.1 Introduction

The greedy approach is an algorithm strategy in which a set of resources are recursively divided based on the maximum, immediate availability of that resource at any given stage of execution.

To solve a problem based on the greedy approach, there are two stages:

1. Scanning the list of items
2. Optimisation

These stages are covered parallelly, on course of division of the array.

To understand the greedy approach, you will need to have a working knowledge of recursion and context switching. This helps you to understand how to trace the code. You can define the greedy paradigm in terms of your own necessary and sufficient statements.

Two conditions define the greedy paradigm which are as follows:

- Each stepwise solution must structure a problem towards its best accepted solution.
- It is sufficient if the structuring of the problem can halt in a finite number of greedy steps.

With the theorising continued, let us describe the history associated with the greedy approach.

History of Greedy Algorithms

Here is an important landmark of greedy algorithms:

- Greedy algorithms were conceptualized for many graph walk algorithms in the 1950s.
- Edsger Dijkstra conceptualised the algorithm to generate minimal spanning trees. He aimed to shorten the span of routes within the Dutch capital, Amsterdam.
- In the same decade, Prim and Kruskal achieved optimisation strategies that were based on minimising path costs along weighed routes.
- In the '70s, American researchers, Cormen, Rivest, and Stein proposed a recursive substructuring of greedy solutions in their classical introduction to algorithms book.

- The greedy paradigm was registered as a different type of optimisation strategy in the NIST records in 2005.
- Till date, protocols that run the web, such as the open shortest path first (OSPF) and many other network packet switching protocols use the greedy strategy to minimise time spent on a network.

Greedy Strategies and Decisions

Logic in its easiest form was boiled down to ‘greedy’ or ‘not greedy’. These statements were defined by the approach taken to advance in each algorithm stage.

For example, Dijkstra’s algorithm utilised a stepwise greedy strategy identifying hosts on the internet by calculating a cost function. The value returned by the cost function determined whether the next path is ‘greedy’ or ‘non-greedy’.

In short, an algorithm ceases to be greedy if at any stage it takes a step that is not locally greedy. The problem halts with no further scope of greed.

Characteristics of the Greedy Approach

The important characteristics of a greedy method are:

- There is an ordered list of resources, with costs or value attributions. These quantify constraints on a system.

- You will take the maximum quantity of resources in the time a constraint applies.
- For example, in an activity scheduling problem, the resource costs are in hours, and the activities need to be performed in serial order.

Table 7.1: Ordered List of Resources

Activity	Start(hours)	Finish(hours)	Resource costs as activity times
1	1	5	
2	6	9	
3	8	9	
4	4	8	
Orderlines			

Why Use the Greedy Approach?

Here are the reasons for using the greedy approach:

- The greedy approach has a few trade offs, which may make it suitable for optimisation.
- One prominent reason is to achieve the most feasible solution immediately. In the activity selection problem (explained below), if more activities can be done before finishing the current activity, these activities can be performed within the same time.
- Another reason is to divide a problem recursively based on a condition, with no need to combine all the solutions.
- In the activity selection problem, the ‘recursive division’ step is achieved by scanning a list of items only once and considering certain activities.

How to Solve the Activity Selection Problem?

In the activity scheduling example, there is a ‘start’ and ‘finish’ time for every activity. Each activity is indexed by a number for reference. There are two activity categories.

1. **Considered Activity:** It is the activity, which is a reference from which the ability to do more than one remaining activity is analysed.
2. **Remaining Activities:** They are activities at one or more indexes ahead of the considered activity.

The total duration gives the cost of performing the activity. That is (finish - start) gives us the durational as the cost of an activity.

You will learn that the greedy extent is the number of remaining activities you can perform in the time of a considered activity.

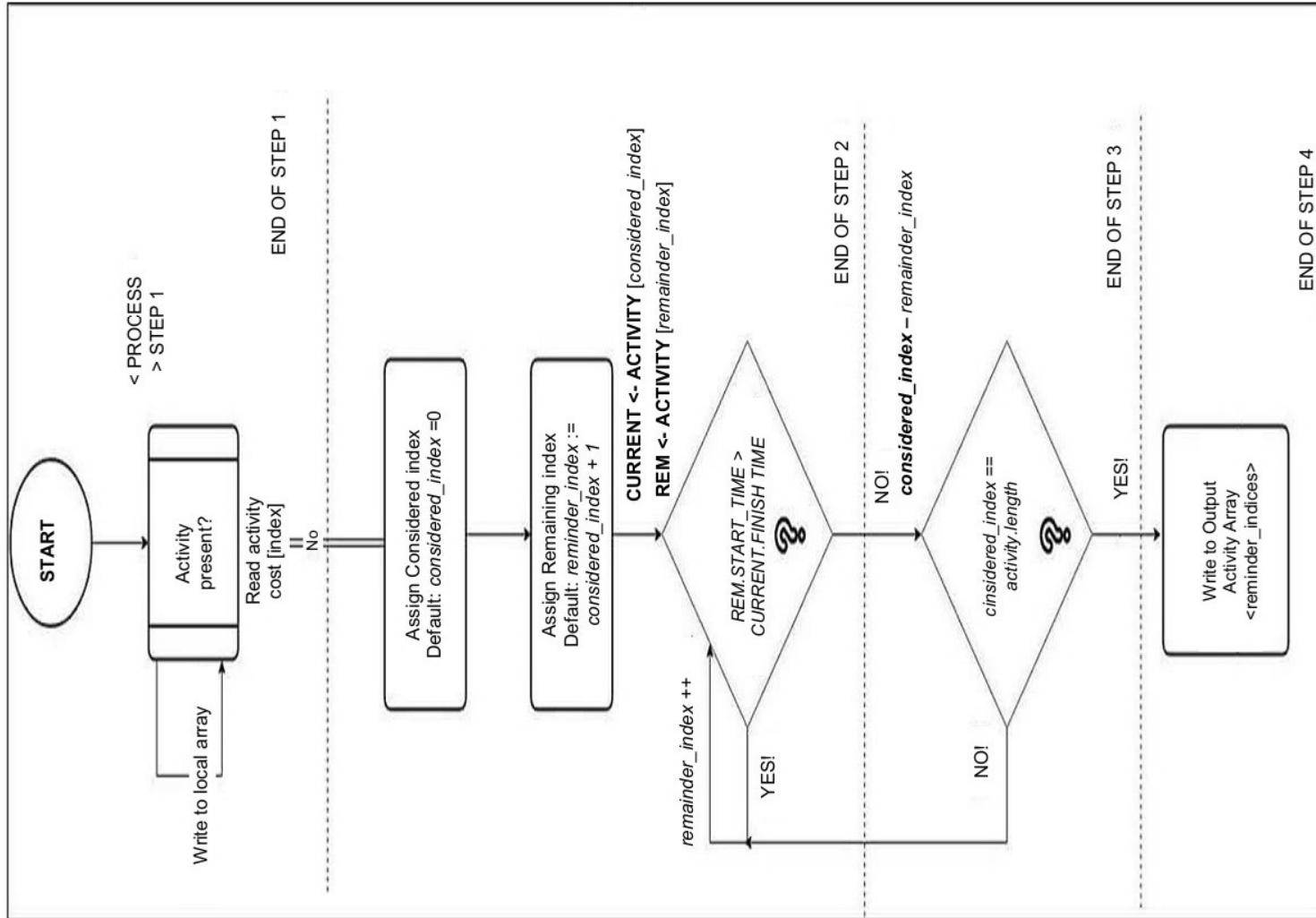
Architecture of the Greedy Approach

Step 1: Scan the list of activity costs, starting with index 0 as the considered index.

Step 2: When more activities can be finished by the time the considered activity finishes, start searching for one or more remaining activities.

Step 3: If there are no more remaining activities, the current remaining activity becomes the next considered activity. Repeat step 1 and step 2, with the new considered activity. If there are no remaining activities left, go to step 4.

Step 4: Return the union of considered indices. These are the activity indices that will be used to maximise throughput.



Disadvantages of Greedy Algorithms

It is not suitable for problems where a solution is required for every subproblem like sorting.

In such problems, the greedy strategy can be wrong; in the worst case even lead to a non-optimal solution.

Therefore, the disadvantage of greedy algorithms is using not knowing what lies ahead of the current greedy state. Below is a depiction of the disadvantage of the greedy approach.

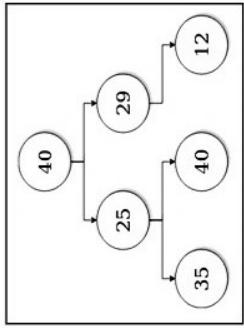


Fig. 7.2: Disadvantage of the Greedy Approach

In the greedy scan shown here as a tree (higher value higher greed), an algorithm state at value: 40, is likely to take 29 as the next value. Further, its quest ends at 12. This amounts to a value of 41. However, if the algorithm took a suboptimal path or adopted a conquering strategy then 25 would be followed by 40, and the overall cost improvement would be 65, which is valued 24 points higher as a suboptimal decision.

Examples of Greedy Algorithms

Most networking algorithms use the greedy approach. Here is a list of few of them –

- Prim's Minimal Spanning Tree Algorithm
- Travelling Salesman Problem
- Graph - Map Colouring
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

7.2 Single-source Shortest Paths: Dijkstra's Algorithm

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest paths problem on a directed weighted graph $G = (V, E)$, where all the edges are non-negative (i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$).

In the following algorithm, we will use one function **Extract-MinQ**, which extracts the node with the smallest key.

Algorithm: Dijkstra's - Algorithm (G, w, s)

```

for each vertex v   G.V
    v.d := ∞
    v.Π := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
    u := Extract-Min (Q)
    S := S U {u}
    for each vertex v   G.adj[u]
        if v.d > u.d + w(u, v)
            v.d := u.d + w(u, v)
            v.Π := u

```

Analysis

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If Extract-Min function is implemented using linear search, the complexity of this algorithm is $O(V_2 + E)$. In this algorithm, if we use min-heap on which **Extract-MinQ** function works to return the node from Q with the smallest key, the complexity of this algorithm can be reduced further.

Example: Let us consider vertex 1 and 9 as the start and destination vertex, respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by 0.

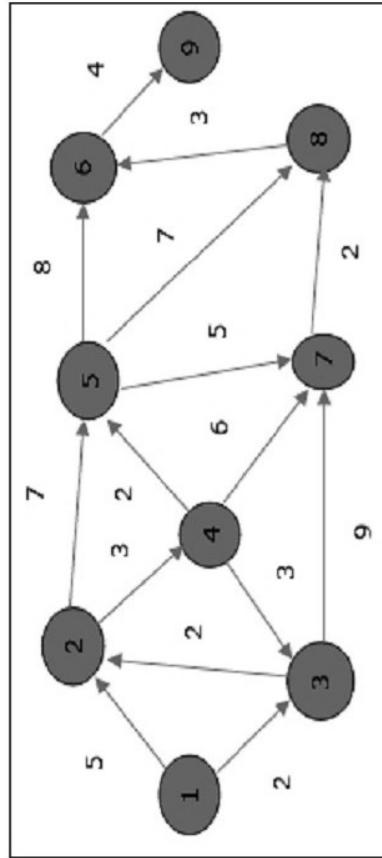
Table 7.2: Solution Example

Vertex	Initial	Step 1 V_1	Step 2 V_3	Step 3 V_2	Step 4 V_4	Step 5 V_5	Step 6 V_7	Step 7 V_8	Step 8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex 9 from vertex 1 is 20. And the path is

1 → 3 → 7 → 8 → 6 → 9

This path is determined based on predecessor information.

**Fig. 7.3: Minimum Distance Path**

7.3 Optimal Tree Problem: Huffman Trees and Codes

Huffman Codes

- (i) Data can be encoded efficiently using Huffman codes.
 - (ii) It is a widely used and beneficial technique for compressing data.
 - (iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.
- Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8×10^5 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

Table 7.3: Frequencies of Occurrences of Each Character

	a	b	c	d	e	f	Total
Frequency	45	13	12	16	9	5	100

How Can we Represent the Data in a Compact Way?

- (i) **Fixed Length Code:** Each letter is represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

For Example:

a	000	b	001
c	010	d	011
e	100	f	101

For a file with 10^5 characters, we need 3×10^5 bits.

- (ii) **A Variable Length Code:** It can do considerably better than a fixed length code, by giving many characters short code words and infrequent character long code words.

For Example:

a	0	b	101
c	100	d	111
e	1101	f	1100



$$\begin{aligned}
 \text{Number of bits} &= (45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 \\
 &= 2.24 \times 10^5 \text{ bits}
 \end{aligned}$$

Thus, 224,000 bits to represent the file, a saving of approximately 25 per cent. This is an optimal character code for this file.

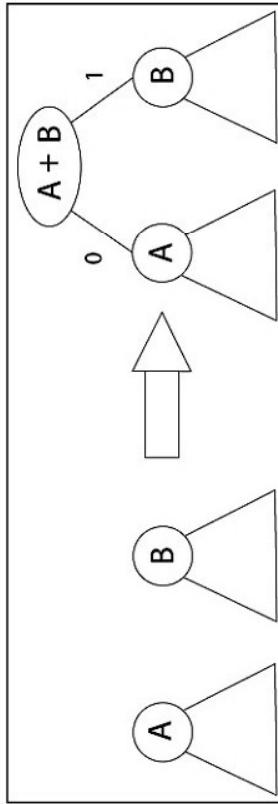
Prefix Codes:

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no code word is a prefix of any other, the code word that starts with an encoded data is unambiguous.

Greedy Algorithm for Constructing a Huffman Code:

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman code.



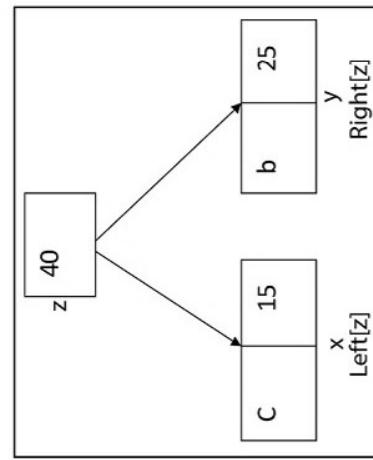
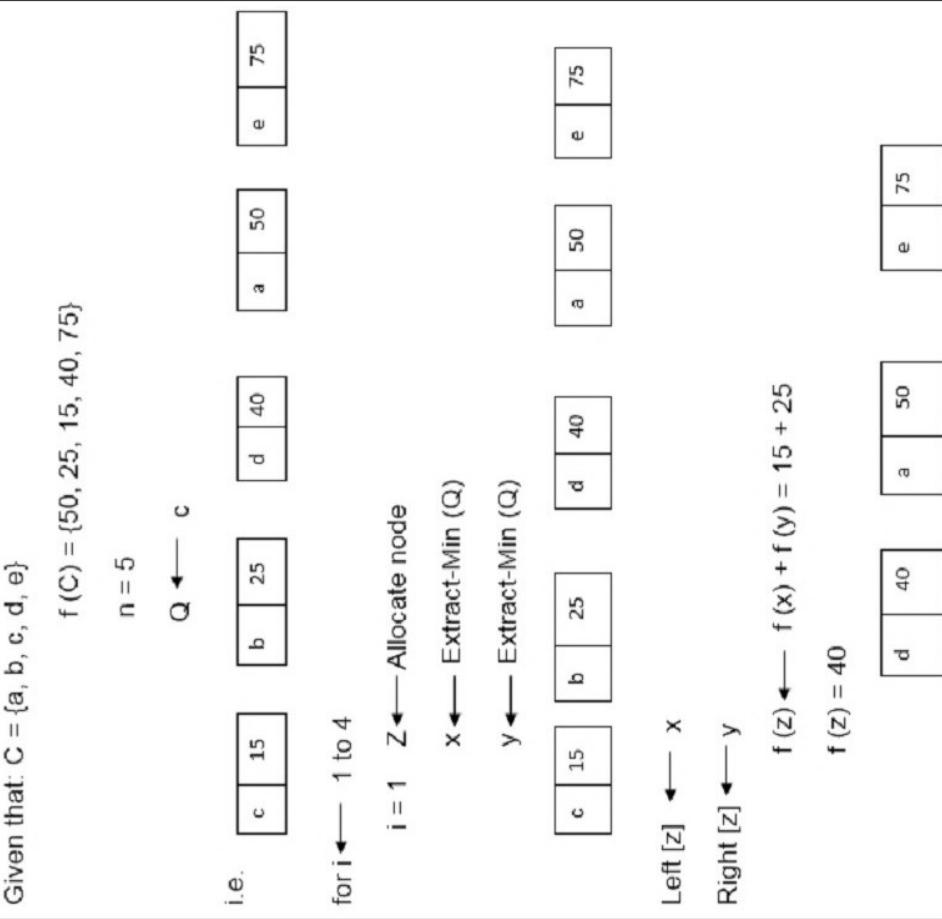
The algorithm builds the tree T analogous to the optimal code in a bottom-up manner. It starts with a set of $|C|$ leaves (C is the number of characters) and performs $|C| - 1$ ‘merging’ operations to create the final tree. In the Huffman algorithm ‘n’ denotes the quantity of a set of characters, z indicates the parent node, and x and y are the left and right child of z , respectively.

Algorithm of Huffman Code**Huffman (C)**

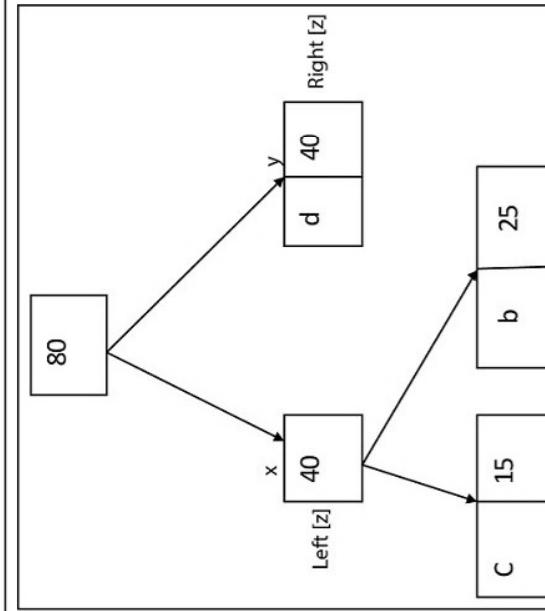
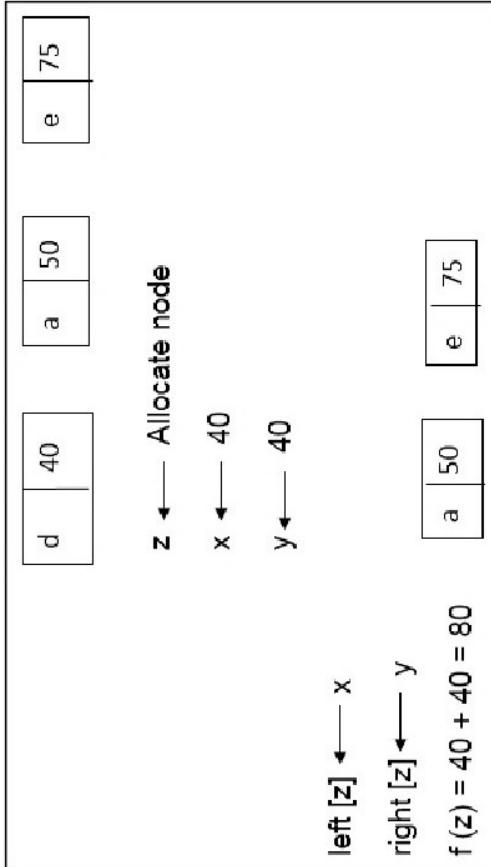
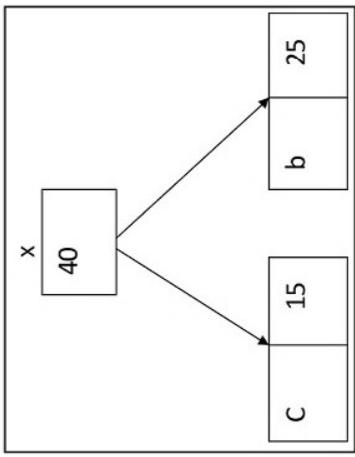
1. n = |C|
2. Q \leftarrow C
3. for i = 1 to n - 1
4. do
5. z = allocate-Node ()
6. x = left[z] = Extract-Min(Q)
7. y = right[z] = Extract-Min(Q)
8. f[z] = f[x] + f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)

Example: Find an optimal Huffman code for the following set of frequencies:

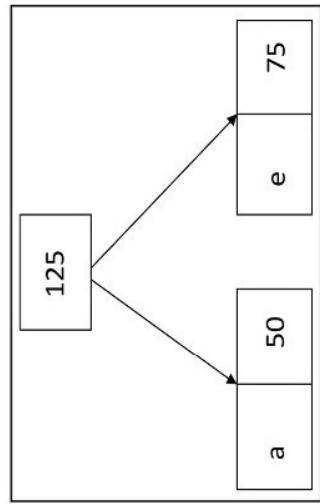
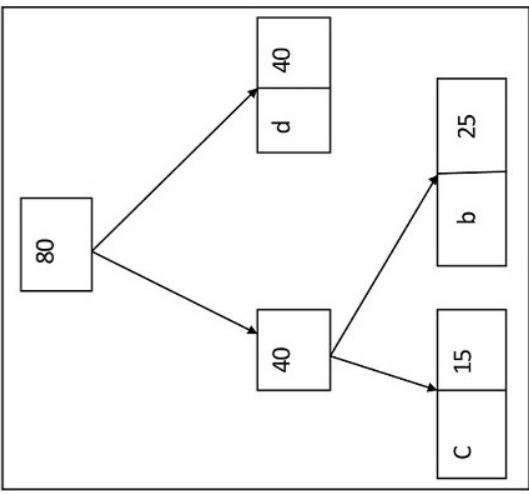
1. a: 50 b: 25 c: 15 d: 40 e: 75

Solution:

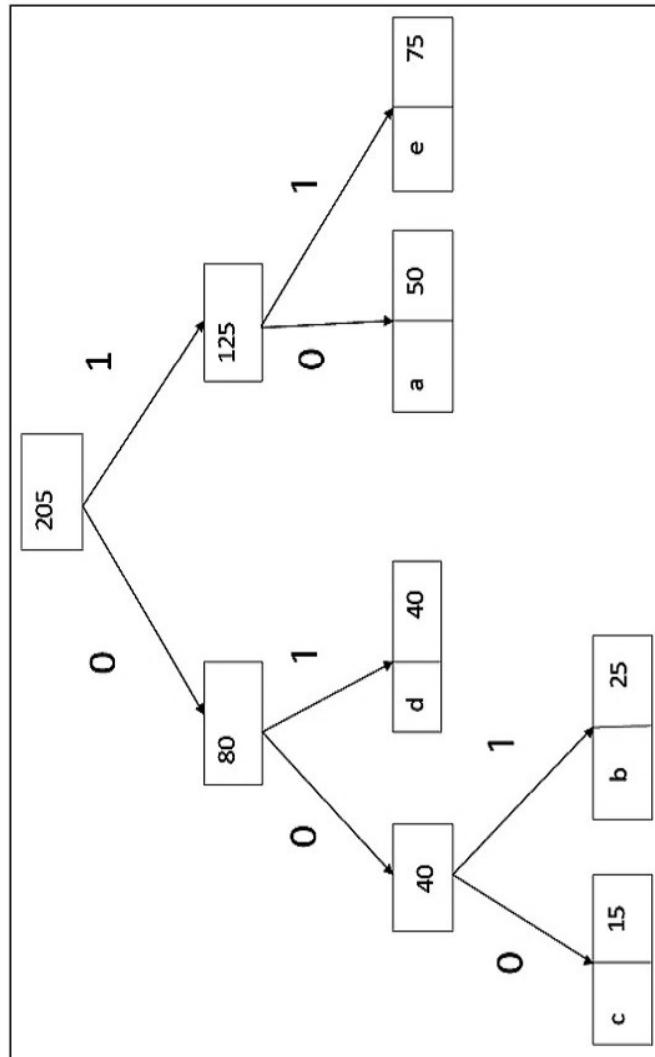
Again for i=2



Similarly, we apply the same process and we get:



Thus, the final output is:



7.4 Transform and Conquer Approach: Heaps and Heap Sort

Heap is a special case of balanced binary tree data structure where the root node key is compared with its children and arranged accordingly. If α has child node β then –

$$\text{key}(\alpha) \geq \text{key}(\beta)$$

As the value of parent is greater than that of child, this property generates **max-heap**. Based on this criteria, a heap can be of two types –

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap – Where the value of the root node is less than or equal to either of its children.

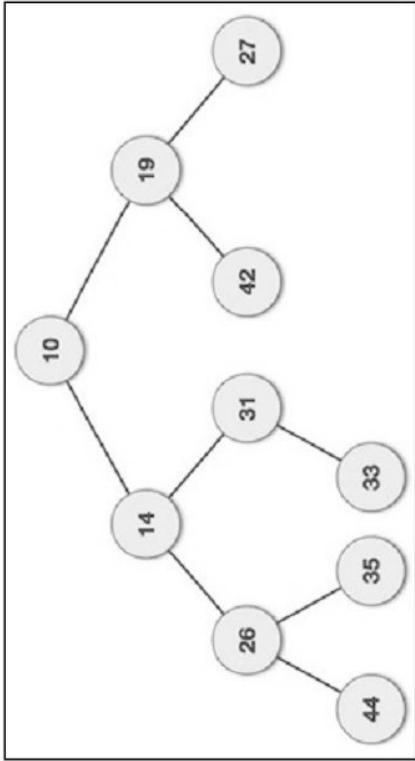


Fig. 7.4: Min-Heap

Max-Heap – Where the value of the root node is greater than or equal to either of its children.

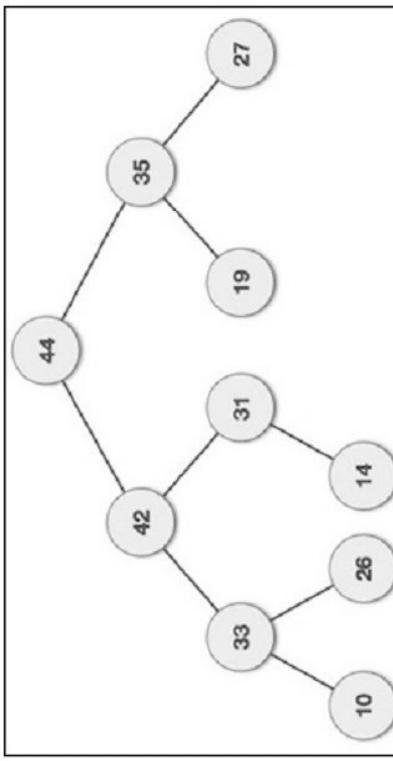


Fig. 7.5: Max-Heap

Both trees are constructed using the same input and order of arrival.

Max-Heap Construction Algorithm

We shall use the same example to demonstrate how a max-heap is created. The procedure to create min-heap is similar, but we go for minimum values instead of maximum values.

We are going to derive an algorithm for max-heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 – Create a new node at the end of the heap.

Step 2 – Assign new value to the node.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 and 4 until heap property holds.

Note – In min-heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand max-heap construction by an animated illustration. We consider the same input sample that we used earlier.

Input → 35 33 42 10 14 19 27 44 26 31

Max-Heap Deletion Algorithm

Let us derive an algorithm to delete from max-heap. Deletion in max (or min)- heap always happens at the root to remove the maximum (or minimum) value.

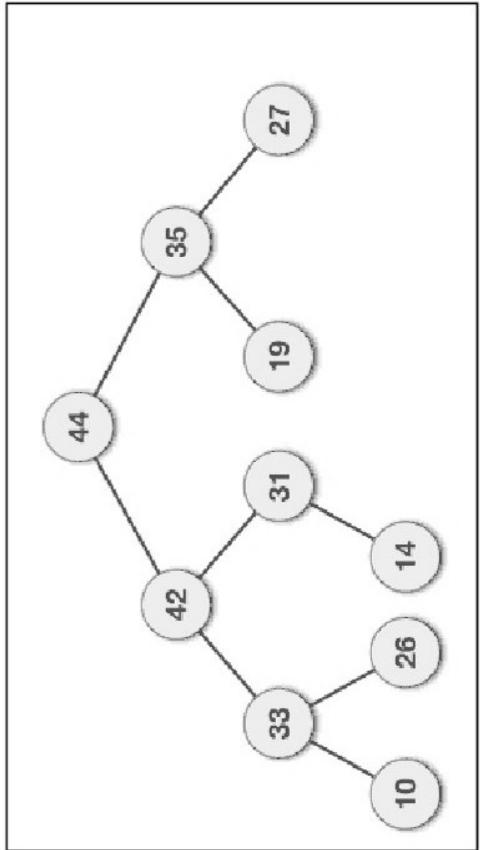
Step 1 – Remove root node.

Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 and 4 until heap property holds.

**Fig. 7.6: Max-Heap Deletion**

Heap Sort

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

Complexity

Table 7.4: Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Space Complexity			$O(1)$

Algorithm

HEAP_SORT(ARR, N)

Step 1: [Build Heap H]

 Repeat for i=0 to N-1

 CALL INSERT_HEAP(ARR, N, ARR[i])

 [END OF LOOP]

Step 2: Repeatedly delete the root element

 Repeat while N > 0

 CALL Delete_Heap(ARR,N,VAL)

 SET N = N+1

 [END OF LOOP]

Step 3: END

7.5 Summary

Single-source Shortest Paths, Arbitrary Weights: The single-source shortest path algorithm (for arbitrary weight positive or negative) is also known Bellman-Ford algorithm and is used to find minimum distance from source vertex to any other vertex.

Dijkstra's algorithm (or Dijkstra's shortest path first algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. For a given source node in the graph, the algorithm finds the shortest path between that node and every other.

Given a sorted array keys[0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts, where freq[i] is the number of searches to keys[i]. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. Level of root is 1.



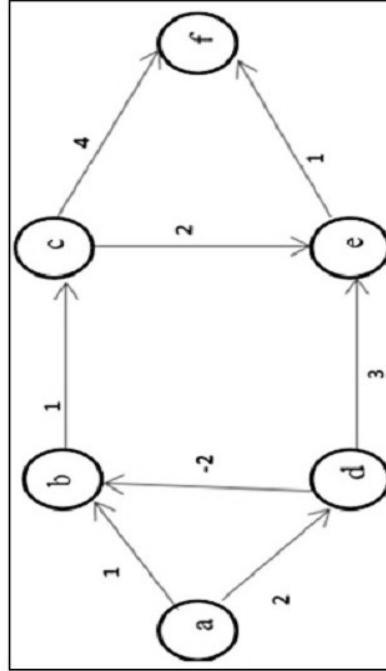
Heap sort is a comparison based sorting technique based on binary heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

7.6 Key Words/Abbreviations

- **Fixed Length Code:** Each letter is represented by an equal number of bits.
- **A Variable Length Code:** It can do considerably better than a fixed length code.
- **Prefix Codes:** The prefixes of an encoding of one character must not be equal to complete encoding of another character.
- **Min-heap:** Where the value of the root node is less than or equal to either of its children.
- **Max-heap:** Where the value of the root node is greater than or equal to either of its children.

7.7 Learning Activity

1. In the given graph:

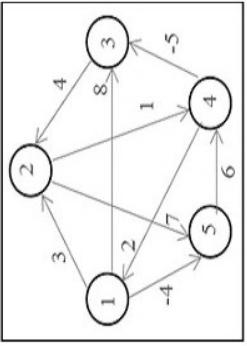


Identify the path that has minimum cost to travel from node a to node f?

2. Complete the program.

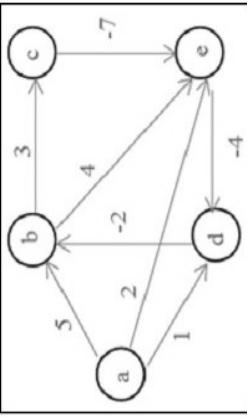
```
n=rows[W]
D(0)=W
for k=1 to n
    do for i=1 to n
        do for j=1 to n
            _____
    return D(n)
```

3. In the given graph:



What is the minimum cost to travel from vertex 1 to vertex 3?

4. In the given graph:



How many intermediate vertices are required to travel from node a to node e at a minimum cost?



5. Consider an array representation of an n element binary heap where the elements are stored from index 1 to index n of the array. For the element stored at index i of the array ($i \leq n$), what is index of the parent?

.....

6. A list of n strings, each of length n , is sorted into lexicographic order using merge sort algorithm. What is the worst-case running time of this computation?

.....

.....

.....

.....

7.8 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. How many arrays are required to perform deletion operation in a heap?
2. What is the average number of comparisons used in a heap sort algorithm?
3. What is the running time of the Huffman encoding algorithm?
4. Define Huffman trees and codes?
5. What is heaps and heap sort?
6. What is meant by single-source shortest paths?
7. What is optimal tree problem?
8. Given a directed weighted graph, you are also given the shortest path from a source vertex ‘s’ to a destination vertex ‘t’. If weight of every edge is increased by 10 units, does the shortest path remain same in the modified graph?
9. Given a directed graph where every edge has weight as either 1 or 2, find the shortest path from a given source vertex ‘s’ to a given destination vertex ‘t’. Expected time complexity is $O(V+E)$.
10. Given a directed acyclic weighted graph, how to find the shortest path from a source s to a destination t in $O(V+E)$ time?

11. How does the algorithm find new paths and do the relaxation?
12. In which order does the algorithm process the vertices one by one?

B. Multiple Choice/Objective Type Questions

1. Dijkstra's algorithm is used to solve _____ problems.
 - (a) All pair shortest path
 - (b) Single source shortest path
 - (c) Network flow
 - (d) Sorting
2. Which of the following is the most commonly used data structure for implementing Dijkstra's algorithm?
 - (a) Max-priority queue
 - (b) Stack
 - (c) Circular queue
 - (d) Min-priority queue
3. What is the time complexity of Dijkstra's algorithm?
 - (a) $O(N)$
 - (b) $O(N_3)$
 - (c) $O(N_2)$
 - (d) $O(\log N)$
4. Dijkstra's algorithm cannot be applied on _____.
 - (a) Directed and weighted graphs
 - (b) Graphs having negative weight function
 - (c) Unweighted graphs
 - (d) Undirected and unweighted graphs
5. How many times the insert and extract minimum operations are invoked per vertex?
 - (a) 1
 - (b) 2
 - (c) 3
 - (d) 0
6. The maximum number of times the decrease key operation performed in Dijkstra's algorithm will be equal to _____.
 - (a) Total number of vertices
 - (b) Total number of edges
 - (c) Number of vertices - 1
 - (d) Number of edges - 1
7. On which algorithm is heap sort based on?
 - (a) Fibonacci heap
 - (b) Binary tree
 - (c) Priority queue
 - (d) FIFO



8. In what time can a binary heap be built?
 - (a) $O(N)$
 - (b) $O(N \log N)$
 - (c) $O(\log N)$
 - (d) $O(N_2)$
9. Heap sort is faster than Shell sort.
 - (a) True
 - (b) False
10. What is the typical running time of a heap sort algorithm?
 - (a) $O(N)$
 - (b) $O(N \log N)$
 - (c) $O(\log N)$
 - (d) $O(N_2)$

Answers:

1. (b), 2. (d), 3. (c), 4. (b), 5. (a), 6. (b), 7. (c), 8. (a), 9. (b), 10. (b)

7.9 References

References of this unit have been given at the end of the book.

UNIT 8 DYNAMIC PROGRAMMING 1

Structure:

- 8.0 Learning Objectives
- 8.1 Introduction
- 8.2 General Method with Examples
- 8.3 Multistage Graphs
- 8.4 Transitive Closure
- 8.5 Warshalls Algorithms
- 8.6 Summary
- 8.7 Key Words/Abbreviations
- 8.8 Learning Activity
- 8.9 Unit End Questions (MCQ and Descriptive)
- 8.10 References

8.0 Learning Objectives

After studying this unit, you will be able to:

- Exemplify dynamic programming general method
- Define multistage graphs
- Describe Floyd-Warshall algorithm

8.1 Introduction

1. Dynamic programming is the most powerful design technique for solving optimisation problems.
2. Divide and conquer algorithm partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solution to solve the original problems.
3. Dynamic programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same subproblem multiple times.
4. Dynamic programming solves each subproblem just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
5. Dynamic programming is a bottom-up approach. We solve all possible small problems and then combine to obtain solutions for bigger problems.
6. Dynamic programming is a paradigm of algorithm design in which an optimisation problem is solved by a combination of achieving subproblem solutions and appearing to the ‘principle of optimality’.

Characteristics of Dynamic Programming:

Dynamic programming works when a problem has the following features:

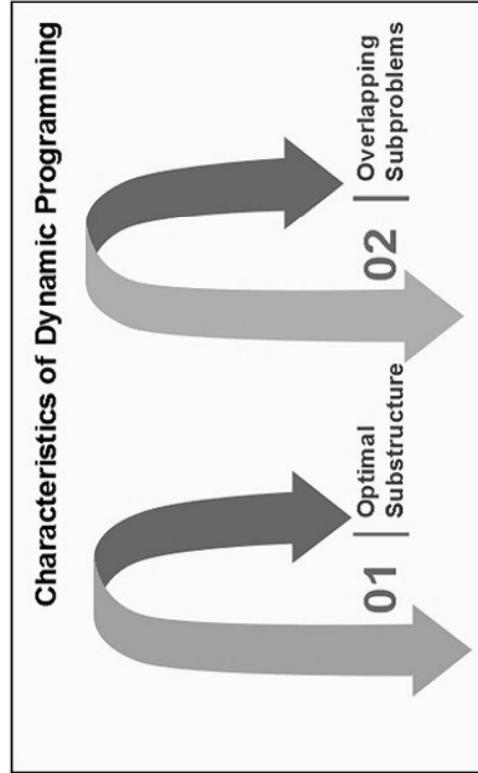


Fig. 8.1: Characteristics of Dynamic Programming

Optimal Substructure: If an optimal solution contains optimal sub-solutions, then a problem exhibits optimal substructure.

Overlapping Subproblems: When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping subproblems, we don't have anything to gain by using dynamic programming.

If the space of subproblems is enough (i.e., polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming

There are basically three elements that characterise a dynamic programming algorithm:

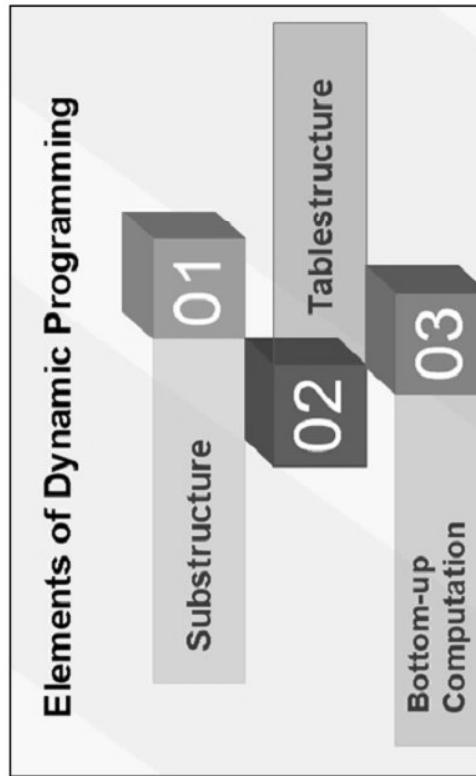


Fig. 8.2: Elements of Dynamic Programming

Substructure: Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.

Table Structure: After solving the subproblems, store the results to the subproblems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.

Bottom-up Computation: Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrive at a solution to complete problem.

Note: Bottom-up means:

- (i) Start with smallest subproblems.
- (ii) Combining their solutions obtain the solution to subproblems of increasing size.
- (iii) Keep solving until you arrive at the solution of the original problem.

Components of Dynamic Programming

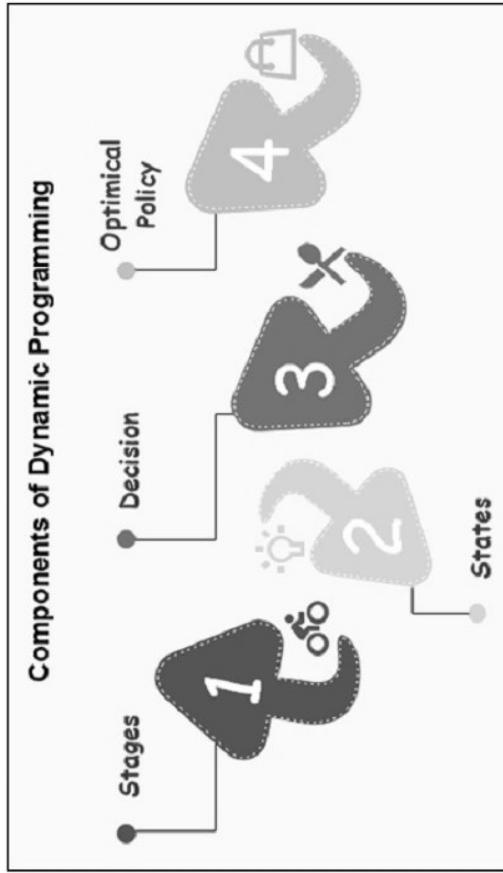


Fig. 8.3: Components of Dynamic Programming

- 1. Stages:** The problem can be divided into several subproblems which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
- 2. States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.

3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal Policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exists a recursive relationship that identifies the optimal decisions for stage j , given that stage $j+1$ has already been solved.
7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterise the structure of an optimal solution.
2. Recursively define the value of the optimal solution. Like divide and conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom-up (starting with the smallest subproblems).
4. Construct the optimal solution for the entire problem from the computed values of smaller subproblems.

Applications of Dynamic Programming

1. 0/1 Knapsack problem
2. Mathematical optimisation problem
3. All pair shortest path problem

4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximise CPU usage.

8.2 General Method with Examples

Dynamic programming is also used in optimisation problems. Like divide and conquer method, dynamic programming solves problems by combining the solutions of subproblems. Moreover, dynamic programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using dynamic programming. These properties are overlapping subproblems and optimal substructure.

Overlapping Subproblems

Similar to divide and conquer approach, dynamic programming also combines solutions to subproblems. It is mainly used where the solution of one subproblem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be recomputed. Hence, this technique is needed where overlapping subproblem exists.

For example, binary search does not have overlapping subproblems whereas recursive program of Fibonacci numbers have many overlapping subproblems.

Optimal Substructure

A given problem has optimal substructure property, if the optimal solution of the given problem can be obtained using optimal solutions of its subproblems.

For example, the shortest path problem has the following optimal substructure property:

If a node x lies in the shortest path from a source node u to destination node v , then the shortest path from u to v is the combination of the shortest path from u to x , and the shortest path from x to v .

The standard all pair shortest path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of dynamic programming.

8.3 Multistage Graphs

A multistage graph $G = (V, E)$ is a directed graph where vertices are partitioned into k (where $k > 1$), number of disjoint subsets $S = \{s_1, s_2, \dots, s_k\}$ such that edge (u, v) is in E , then $u \in s_i$ and $v \in s_{i+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.

The vertex $s \in s_1$ is called the source and the vertex $t \in s_k$ is called sink.

G is usually assumed to be a weighted graph. In this graph, cost of an edge (i, j) is represented by $c(i, j)$. Hence, the cost of path from source s to sink t is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source s to sink t .

Example:

Consider the following example to understand the concept of multistage graph.

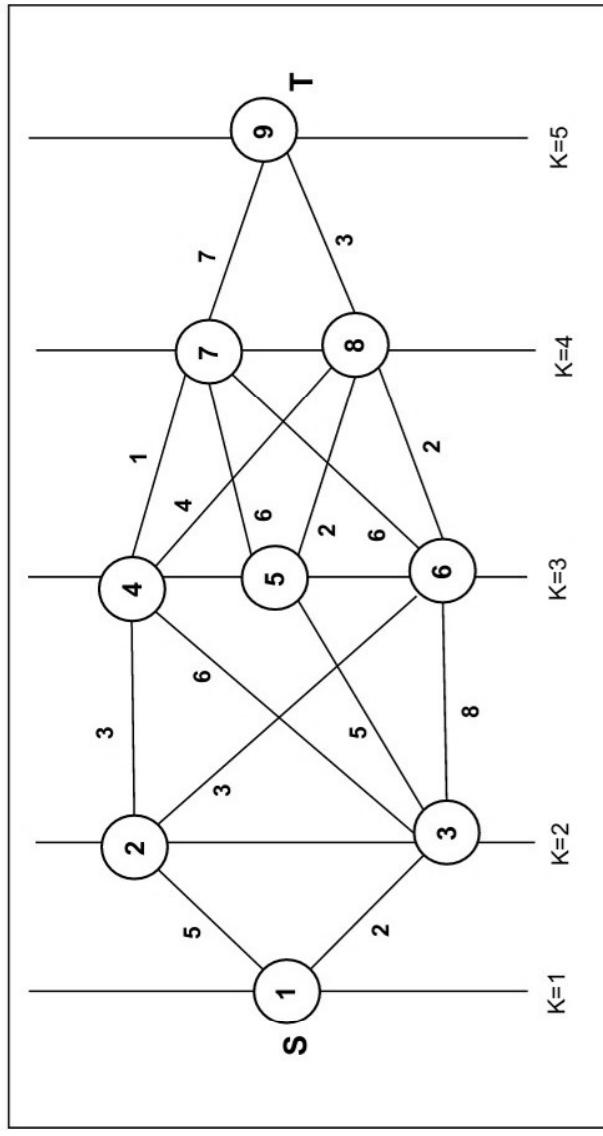


Fig. 8.4: Multistage Graph

According to the formula, we have to calculate the cost (i, j) using the following steps:

Step-1: Cost (K-2, j)

In this step, three nodes (node 4, 5, 6) are selected as j. Hence, we have three options to choose the minimum cost at this step.

$$\text{Cost}(3, 4) = \min \{c(4, 7) + \text{Cost}(7, 9), c(4, 8) + \text{Cost}(8, 9)\} = 7$$

$$\text{Cost}(3, 5) = \min \{c(5, 7) + \text{Cost}(7, 9), c(5, 8) + \text{Cost}(8, 9)\} = 5$$

$$\text{Cost}(3, 6) = \min \{c(6, 7) + \text{Cost}(7, 9), c(6, 8) + \text{Cost}(8, 9)\} = 5$$

Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage $k - 3 = 2$ there are two nodes, 2 and 3. So, the value $i = 2$ and $j = 2$ and 3.

$$\begin{aligned} \text{Cost}(2, 2) &= \min \{c(2, 4) + \text{Cost}(4, 8) + \text{Cost}(8, 9), c(2, 6) + \text{Cost}(6, 8) + \text{Cost}(8, 9)\} \\ &= 8 \end{aligned}$$

$$\begin{aligned} \text{Cost}(2, 3) &= \{c(3, 4) + \text{Cost}(4, 8) + \text{Cost}(8, 9), c(3, 5) + \text{Cost}(5, 8) + \text{Cost}(8, 9), c(3, \\ &\quad 6) + \text{Cost}(6, 8) + \text{Cost}(8, 9)\} = 10 \end{aligned}$$

Step-3: Cost (K-4, j)

$$\begin{aligned} \text{Cost}(1, 1) &= \{c(1, 2) + \text{Cost}(2, 6) + \text{Cost}(6, 8) + \text{Cost}(8, 9), c(1, 3) + \text{Cost}(3, 5) + \\ &\quad \text{Cost}(5, 8) + \text{Cost}(8, 9)\} = 12 \end{aligned}$$

$$c(1, 3) + \text{Cost}(3, 6) + \text{Cost}(6, 8 + \text{Cost}(8, 9))\} = 13$$

Hence, the path having the minimum cost is $1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 9$.

8.4 Transitive Closure

Transitive Closure of a Graph

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here, reachable means that there is a path from vertex i to j. The reachability matrix is called transitive closure of a graph.



For example, consider below graph:

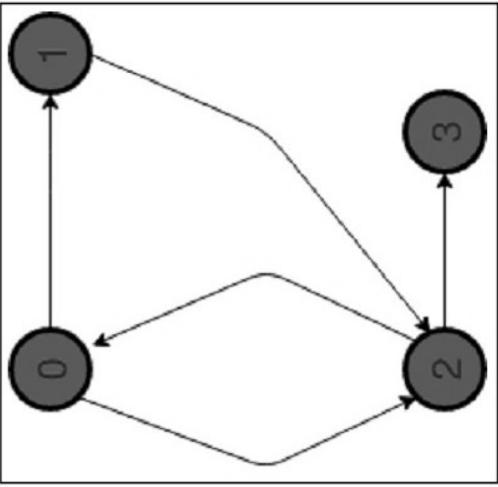


Fig. 8.5: Directed Graph

Transitive closure of above graphs is:

1	1	1	1
1	1	1	1
1	1	1	1
0	0	0	1

The graph is given in the form of adjacency matrix say ‘ $\text{graph}[V][V]$ ’ where $\text{graph}[i][j]$ is 1 if there is an edge from vertex i to vertex j or i is equal to j , otherwise $\text{graph}[i][j]$ is 0.

Floyd-Warshall algorithm can be used; we can calculate the distance matrix $\text{dist}[V][V]$ using Floyd-Warshall, if $\text{dist}[i][j]$ is infinite, then j is not reachable from i , otherwise j is reachable and value of $\text{dist}[i][j]$ will be less than V .

Instead of directly using Floyd-Warshall, we can optimise it in terms of space and time, for this particular problem. Following are the optimisations:

- 1) Instead of integer resultant matrix ($\text{dist}[V][V]$ in Floyd-Warshall), we can create a boolean reach-ability matrix $\text{reach}[V][V]$ (we save space). The value $\text{reach}[i][j]$ will be 1, if j is reachable from i , otherwise 0.

- 2) Instead of using arithmetic operations, we can use logical operations. For arithmetic operation ‘+’, logical and ‘&&’ is used, and for min, logical or ‘||’ is used. (We save time by a constant factor. Time complexity is same though.)

8.5 Warshalls Algorithms

Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

At first, the output matrix is the same as the given cost matrix of the graph. After that, the output matrix will be updated with all vertices k as the intermediate vertex. The time complexity of this algorithm is $O(V^3)$, where V is the number of vertices in the graph.

Input and Output

Table 8.1: Input: The Cost Matrix of the Graph

0	3	6	∞	∞	∞
3	0	2	1	∞	∞
6	2	0	1	4	2
∞	1	1	0	2	∞
∞	∞	4	2	0	2
∞	∞	2	∞	2	0
∞	∞	∞	4	1	1
					0

Table 8.2: Output: Matrix of All Pair Shortest Path

0	3	4	5	6	7	7
3	0	2	1	3	4	4
4	2	0	1	3	2	3
5	1	1	0	2	3	3
6	3	3	2	0	2	1
7	4	2	3	2	0	1
7	4	3	3	1	1	0

Algorithm

Floyd-Warshall(cost)

Input: The cost matrix of given graph.

Output: Matrix for shortest path between any vertex to any vertex.

```

Begin
    for k := 0 to n, do
        for i := 0 to n, do
            for j := 0 to n, do
                if cost[i,k] + cost[k,j] < cost[i,j], then
                    cost[i,j] := cost[i,k] + cost[k,j]
                done
            done
        done
    display the current cost matrix
End

```

8.6 Summary

Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory based data structure (array, map, etc).

A multistage graph is a directed graph in which the nodes can be divided into a set of stages, such that all edges are from a stage to next stage only. (In other words, there is no edge between vertices of same stage and from a vertex of current stage to previous stage.) We are given a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.

The Floyd-Warshall algorithm is for solving the all pairs shortest path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed graph.

8.7 Key Words/Abbreviations

- **Substructure:** Decompose the given problem into smaller subproblems.
- **Table Structure:** After solving the subproblems, store the results to the subproblems in a table.
- **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems.

8.8 Learning Activity

1. What is the time complexity of Floyd-Warshall algorithm to calculate all pair shortest path in a graph with n vertices?

2. In a competition, four different functions are observed. All the functions use a single for loop and within the for loop, same set of statements are executed. Consider the following for loops:

A) `for(i = 0; i < n; i++)`

B) `for(i = 0; i < n; i += 2)`

C) `for(i = 1; i < n; i *= 2)`

D) `for(i = n; i > -1; i /= 2)`

3. If n is the size of input (positive), which function is most efficient (if the task to be performed is not an issue)?

4. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?
.....
.....
5. What is the basic difference between fractional and binary knapsack algorithm?
.....
.....
6. What is the 'Knapsack problem'?
.....
.....

8.9 Unit End Questions (MCQ and Descriptive)

A. Descriptive Types Questions

1. Define dynamic programming
2. What is a multistage graph?
3. What is the need for transitive closure?
4. Define Warshall algorithm.
5. What is the optimal substructure?
6. Define characteristics of dynamic programming.

B. Multiple Choice/Objective Type Questions

1. Floyd-Warshall algorithm can be used for finding _____.
(a) Single-source shortest path (b) Topological sort
(c) Minimum spanning tree (d) Transitive closure
2. What procedure is being followed in Floyd-Warshall algorithm?
(a) Top-down (b) Bottom-up
(c) Big bang (d) Sandwich



3. Floyd-Warshall algorithm was proposed by _____.

 - (a) Robert Floyd and Stephen Warshall
 - (b) Stephen Floyd and Robert Warshall
 - (c) Bernad Floyd and Robert Warshall
 - (d) Robert Floyd and Bernad Warshall

4. Who proposed the modern formulation of Floyd-Warshall algorithm as three nested loops?

 - (a) Robert Floyd
 - (b) Stephen Warshall
 - (c) Bernard Roy
 - (d) Peter Ingerman

5. Which of the following statements for a simple graph is correct?

 - (a) Every path is a trail
 - (b) Every trail is a path
 - (c) Every trail is a path as well as every path is a trail
 - (d) Path and trail have no relation

6. What are the number of edges present in a complete graph having n vertices?

 - (a) $(n*(n+1))/2$
 - (b) $(n*(n-1))/2$
 - (c) n
 - (d) Information given is insufficient

7. In a simple graph, the number of edges is equal to twice the sum of the degrees of the vertices.

 - (a) True
 - (b) False

8. A connected planar graph having 6 vertices, 7 edges contains _____ regions.

 - (a) 15
 - (b) 3
 - (c) 1
 - (d) 11

9. If a simple graph G , contains n vertices and m edges, the number of edges in the graph G' (Complement of G) is _____

 - (a) $(n*n-n-2*m)/2$
 - (b) $(n*n+n+2*m)/2$
 - (c) $(n*n-n-2*m)/2$
 - (d) $(n*n+n+2*m)/2$



10. Which of the following properties does a simple graph not hold?
 - (a) Must be connected
 - (b) Must be unweighted
 - (c) Must have no loops or multiple edges
 - (d) Must have no multiple edges
11. What is the maximum number of edges in a bipartite graph having 10 vertices?
 - (a) 24
 - (b) 21
 - (c) 25
 - (d) 16
12. Which of the following is true?
 - (a) A graph may contain no edges and many vertices
 - (b) A graph may contain many edges and no vertices
 - (c) A graph may contain no edges and no vertices
 - (d) A graph may contain no vertices and many edges
13. For a given graph G having v vertices and e edges which is connected and has no cycles, which of the following statements is true?
 - (a) $v = e$
 - (b) $v = e + 1$
 - (c) $v + 1 = e$
 - (d) $v = e - 1$

Answers:

1. (d), 2. (b), 3. (a), 4. (d), 5. (a), 6. (b), 7. (b), 8. (b), 9. (a), 10. (a), 11. (c), 12. (b), 13. (b)

8.10 References

References of this unit have been given at the end of the book.

UNIT 9 DYNAMIC PROGRAMMING 2

Structure:

- 9.0 Learning Objectives
- 9.1 Introduction
- 9.2 All Pairs Shortest Paths
- 9.3 Floyd-Warshall Algorithm
- 9.4 Optimal Binary Search Trees
- 9.5 Bellman-Ford Algorithm
- 9.6 Travelling Salesman Problem
- 9.7 Reliability Design
- 9.8 Summary
- 9.9 Key Words/Abbreviations
- 9.10 Learning Activity
- 9.11 Unit End Questions (MCQ and Descriptive)
- 9.12 References

9.0 Learning Objectives

After studying this unit, you will be able to:

- Understand Floyd-Warshall algorithm
- Describe TSP problems



- Explain Bellman-Ford algorithm
 - Define reliability design
 - Describe all pairs shortest paths concepts
 - Explain assignment problems
 - Understand Bellman-Ford algorithm
 - Ensure travelling sales person problem
-

9.1 Introduction

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of and optimal substructure. When applicable, the method takes far less time than naive methods that don't take advantage of the subproblem overlap.

The idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often when using a more naive method, many of the subproblems are generated and solved many times. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations. Once the solution to a given subproblem has been computed, it is stored. The next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems experience exponential growth as a function of the size of the input.

Linear programming adopts an intentionally simple model. Dynamic programming concerns itself with a class of functional relations that arise from multistage decision processes possessing certain definite structural characteristics. The characteristic properties are exploited to effect a reduction in the dimensionality of the mathematical problem, thereby making some complex processes amenable to analytic or computational techniques.

Since linear programming is an allocation problem, various subproblems (states) may be defined as the amount of resources to be allocated to the current stage and the succeeding stages.

This will result in a backward functional (recursive) equation, which, when obtained, can be used to solve the linear programming problem by the dynamic programming approach.

9.2 All Pairs Shortest Paths

Introduction

It aims to figure out the shortest path from each vertex v to every other u . Storing all the paths explicitly can be very memory-expensive indeed, as we need one spanning tree for each vertex. This is often impractical regarding memory consumption, so these are generally considered as all pairs shortest distance problems, which aim to find just the distance from each to each node to another. We usually want the output in tabular form: the entry in u 's row and v 's column should be the weight of the shortest path from u to v .

Three approaches for improvement:

Table 9.1: Algorithm with Cost

Algorithm	Cost
Matrix Multiplication	$O(N^3 \log V)$
Floyd-Warshall	$O(N^3)$
Johnson O	$(N_2 \log^2 N + VE)$

Unlike the single-source algorithms, which assume an adjacency list representation of the graph, most of the algorithm uses an adjacency matrix representation. (Johnson's algorithm for sparse graphs uses adjacency lists.) The input is a $n \times n$ matrix, W representing the edge weights of an n vertex directed graph $G = (V, E)$. That is, $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i, j) & \text{if } i \neq j \text{ and } (i, j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E. \end{cases}$$

9.3 Floyd-Warshall Algorithm

Let the vertices of G be $V = \{1, 2, \dots, n\}$ and consider a subset $\{1, 2, \dots, k\}$ of vertices for some k . For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate



vertices are all drawn from $\{1, 2, \dots, k\}$, and let p be a minimum weight path from amongst them. The Floyd-Warshall algorithm exploits a link between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \dots, k-1\}$. The link depends on whether or not k is an intermediate vertex of path p .

If k is not an intermediate vertex of path p , then all intermediate vertices of path p are in the set $\{1, 2, \dots, k-1\}$. Thus, the shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \dots, k-1\}$ is, also, the shortest path i to j with all intermediate vertices in the set $\{1, 2, \dots, k\}$.

If k is an intermediate vertex of path p , then we break p down into $i \rightarrow k \rightarrow j$.

Let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \dots, k\}$.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

FLOYD-WARSHALL(W)

1. $n \leftarrow \text{rows}[W]$.
2. $D^0 \leftarrow W$
3. for $k \leftarrow 1$ to n
4. do for $i \leftarrow 1$ to n
5. do for $j \leftarrow 1$ to n
6. do $d_{ij}^{(k)} \leftarrow \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)})$
7. return $D^{(n)}$

The strategy adopted by the Floyd-Warshall algorithm is dynamic programming. The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes $O(1)$ time. The algorithm, thus, runs in time $\theta(n^3)$.

Example: Apply Floyd-Warshall algorithm for constructing the shortest path. Show that matrices $D(k)$ and $\pi(k)$ are computed by the Floyd-Warshall algorithm for the graph.

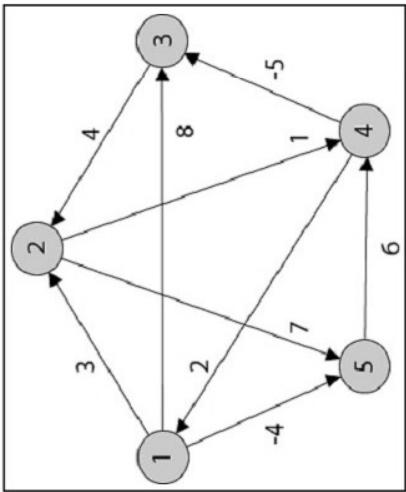


Fig. 9.1: Graph

Solution:

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Step (i) When $k = 0$

$D^{(0)}$ = 0	3	8	∞	-4	$\pi^{(0)} = \text{NIL}$	1	1	NIL	1
∞	0	∞	1	7	NIL	NIL	2	2	
∞	4	0	-5	∞	NIL	3	NIL	3	NIL
2	∞	∞	0	∞	4	NIL	NIL	NIL	NIL
∞	∞	∞	6	0	NIL	NIL	5	NIL	

Step (ii) When $k = 1$

$$\begin{aligned} d_{ij}^{(k)} &= \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \\ d_{14}^{(1)} &= \min(d_{14}^{(0)}, d_{11}^{(0)} + d_{14}^{(0)}) \\ d_{14}^{(1)} &= \min(\infty, 0 + \infty) = \infty \end{aligned}$$



$$d_{15}^{(1)} = \min (d_{15}^{(0)}, d_{11}^{(0)} + d_{15}^{(0)})$$

$$d_{15}^{(1)} = \min (-4, 0 + -4) = -4$$

$$d_{21}^{(1)} = \min (d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$$

$$d_{21}^{(1)} = \min (\infty, \infty + 0) = \infty$$

$$d_{23}^{(1)} = \min (d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)})$$

$$d_{23}^{(1)} = \min (\infty, \infty + 8) = \infty$$

$$d_{31}^{(1)} = \min (d_{31}^{(0)}, d_{31}^{(0)} + d_{11}^{(0)})$$

$$d_{31}^{(1)} = \min (\infty, \infty + 0) = \infty$$

$$d_{35}^{(1)} = \min (d_{35}^{(0)}, d_{31}^{(0)} + d_{15}^{(0)})$$

$$d_{35}^{(1)} = \min (\infty, \infty + (-4)) = \infty$$

$$d_{42}^{(1)} = \min (d_{42}^{(0)}, d_{41}^{(0)} + d_{12}^{(0)})$$

$$d_{42}^{(1)} = \min (\infty, 2 + 3) = 5$$

$$d_{43}^{(1)} = \min (d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)})$$

$$d_{43}^{(1)} = \min (\infty, 2 + 8) = 10$$

$$d_{45}^{(1)} = \min (d_{45}^{(0)}, d_{41}^{(0)} + d_{15}^{(0)})$$

$$d_{45}^{(1)} = \min (\infty, 2 + (-4)) = -2$$

$$d_{51}^{(1)} = \min (d_{51}^{(0)}, d_{51}^{(0)} + d_{11}^{(0)})$$

$$d_{51}^{(1)} = \min (\infty, \infty + 0) = \infty$$



$D_{ij}^{(1)}$	0	3	8	∞	-4	$\pi^{(1)} = \text{NIL}$	1	1	NIL	1
∞	0	∞	1	7	NIL	NIL	2	2	NIL	2
∞	4	0	-5	∞	NIL	3	NIL	3	NIL	3
2	5	10	0	-2	NIL	4	1	1	NIL	1
∞	∞	6	0	NIL	NIL	NIL	5	NIL	5	NIL

Step (iii) When k = 2

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{14}^{(2)} = \min(d_{14}^{(1)}, d_{12}^{(1)} + d_{24}^{(1)})$$

$$d_{14}^{(2)} = \min(\infty, 3 + 1) = 4$$

$$d_{21}^{(2)} = \min(d_{21}^{(1)}, d_{22}^{(1)} + d_{21}^{(1)})$$

$$d_{21}^{(2)} = \min(\infty, 0 + \infty) = \infty$$

$$d_{34}^{(2)} = \min(d_{34}^{(1)}, d_{32}^{(1)} + d_{24}^{(1)})$$

$$d_{34}^{(2)} = \min(-5, 4 + 1) = -5$$

$$d_{35}^{(2)} = \min(d_{35}^{(1)}, d_{32}^{(1)} + d_{25}^{(1)})$$

$$d_{35}^{(2)} = \min(\infty, 4 + 7) = 11$$

$$d_{43}^{(2)} = \min(d_{43}^{(1)}, d_{42}^{(1)} + d_{23}^{(1)})$$

$$d_{43}^{(2)} = \min(10, 5 + \infty) = 10$$

$$D_{ij}^{(2)} = 0 \quad 3 \quad 8 \quad 4 \quad -4 \quad \pi^{(2)} = \text{NIL} \quad 1 \quad 1 \quad 2 \quad 1$$

$$\infty \quad 0 \quad \infty \quad 1 \quad 7 \quad \text{NIL} \quad \text{NIL} \quad \text{NIL} \quad 2 \quad 2$$

$$\infty \quad 4 \quad 0 \quad -5 \quad 11 \quad \text{NIL} \quad 3 \quad \text{NIL} \quad 3 \quad 2$$

$$2 \quad 5 \quad 10 \quad 0 \quad -2 \quad 4 \quad 1 \quad 1 \quad \text{NIL} \quad 1$$

$$\infty \quad \infty \quad 6 \quad 0 \quad \text{NIL} \quad \text{NIL} \quad \text{NIL} \quad 5 \quad \text{NIL}$$



Step (iv) When $k = 3$

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{14}^{(3)} = \min(d_{14}^{(2)}, d_{13}^{(2)} + d_{34}^{(2)})$$

$$d_{14}^{(3)} = \min(4, 8 + (-5)) = 3$$

		0	3	8	3	-4		$\pi^{(3)}$ =	NIL	1	1	3	1
∞	0	∞	1	7			NIL	NIL	NIL	2	2		
∞	4	0	-5	11			NIL	3	NIL	3	2		
2	5	10	0	-2			4	1	1	NIL	1		
∞	∞	∞	6	0			NIL	NIL	NIL	5	NIL		

Step (v) When $k = 4$

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{21}^{(4)} = \min(d_{21}^{(3)}, d_{24}^{(3)} + d_{41}^{(3)})$$

$$d_{21}^{(4)} = \min(\infty, 1 + 2) = 3$$

$$d_{23}^{(4)} = \min(d_{23}^{(3)}, d_{24}^{(3)} + d_{43}^{(3)})$$

$$d_{23}^{(4)} = \min(\infty, 1 + 10) = 11$$

$$d_{25}^{(4)} = \min(d_{25}^{(3)}, d_{24}^{(3)} + d_{45}^{(3)})$$

$$d_{25}^{(4)} = \min(7, 1 + (-2)) = -1$$

$$d_{31}^{(4)} = \min(d_{31}^{(3)}, d_{34}^{(3)} + d_{41}^{(3)})$$

$$d_{31}^{(4)} = \min(\infty, -5 + 2) = -3$$

$$d_{32}^{(4)} = \min(d_{32}^{(3)}, d_{34}^{(3)} + d_{42}^{(3)})$$

$$d_{32}^{(4)} = \min(4, -5 + 5) = 0$$



$$d_{51}^{(4)} = \min(d_{51}^{(3)}, d_{54}^{(3)} + d_{41}^{(3)})$$

$$d_{51}^{(4)} = \min(\infty, 6 + 2) = 8$$

$$d_{52}^{(4)} = \min(d_{52}^{(3)}, d_{54}^{(3)} + d_{42}^{(3)})$$

$$d_{52}^{(4)} = \min(\infty, 6 + 5) = 11$$

$$d_{53}^{(4)} = \min(d_{53}^{(3)}, d_{54}^{(3)} + d_{43}^{(3)})$$

$$d_{53}^{(4)} = \min(\infty, 6 + 10) = 16$$

	0	3	8	3	-4	$\pi^{(4)} =$	NIL	1	1	3	1
	3	0	11	1	-1		4	NIL	4	2	2
	-3	0	0	-5	-7		4	4	NIL	3	4
	2	5	10	0	-2		4	1	1	NIL	1
	8	11	16	6	0		4	4	4	5	NIL

Step (vi) When k = 5

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

$$d_{25}^{(5)} = \min(d_{25}^{(4)}, d_{25}^{(4)} + d_{55}^{(3)})$$

$$d_{25}^{(5)} = \min(-1, -1 + 0) = -1$$

$$d_{23}^{(5)} = \min(d_{23}^{(4)}, d_{25}^{(4)} + d_{53}^{(3)})$$

$$d_{23}^{(5)} = \min(11, -1 + 16) = 11$$

$$d_{35}^{(5)} = \min(d_{35}^{(4)}, d_{35}^{(4)} + d_{55}^{(3)})$$

$$d_{35}^{(5)} = \min(-7, -7 + 0) = -7$$



$D_{ij}^{(5)} =$	0	3	8	3	-4	$\pi^{(5)} =$	NIL	1	1	5	1
	3	0	11	1	-1		4	NIL	4	2	4
	-3	0	0	-5	-7		4	4	NIL	3	4
	2	5	10	0	-2		4	1	1	NIL	1
	8	11	16	6	0		4	4	4	5	NIL

TRANSITIVE-CLOSURE (G)

1. $n \leftarrow |V[G]|$
2. for $i \leftarrow 1$ to n
3. do for $j \leftarrow 1$ to n
4. do if $i = j$ or $(i, j) \in E[G]$
5. the $t_{ij}^{(0)} \leftarrow 1$
6. else $t_{ij}^{(0)} \leftarrow 0$
7. for $k \leftarrow 1$ to n
8. do for $i \leftarrow 1$ to n
9. do for $j \leftarrow 1$ to n
10. $dod_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
11. Return $T^{(n)}$.

9.4 Optimal Binary Search Trees

Binary Search Trees

A binary search tree is organised in a binary tree. Such a tree can be defined by a linked data structure in which a particular node is an object. In addition to a key field, each node contains field left, right, and p that point to the nodes corresponding to its left child, its right child, and its



parent, respectively. If a child or parent is missing, the appropriate field contains the value NIL.
The root node is the only node in the tree whose parent field is NIL.

Binary Search Tree Property

Let x be a node in a binary search tree.

- If y is a node in the left subtree of x , then $\text{key}[y] \leq \text{key}[k]$.
- If z is a node in the right subtree of x , then $\text{key}[x] \leq \text{key}[y]$.

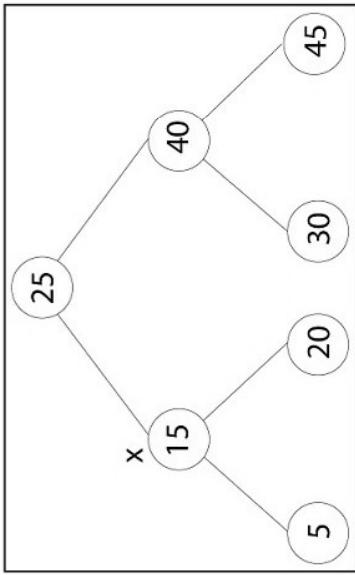


Fig. 9.2: Binary Search Tree

In this tree $\text{key}[x] = 15$

- If y is a node in the left subtree of x , then $\text{key}[y] = 5$,
i.e., $\text{key}[y] \leq \text{key}[x]$.
- If y is a node in the right subtree of x , then $\text{key}[y] = 20$,
i.e., $\text{key}[x] \leq \text{key}[y]$.

Traversal in Binary Search Trees

1. In-Order-Tree-Walk (x): Always print keys in binary search tree in sorted order.

INORDER-TREE-WALK (x) - Running time is $\Theta(n)$

1. If $x \neq \text{NIL}$.
2. then INORDER-TREE-WALK (left [x])
3. print key [x]

4. INORDER-TREE-WALK (right [x])
2. **PREORDER-TREE-WALK (x):** In which we visit the root node before the nodes in either subtree.

PREORDER-TREE-WALK (x):

1. If $x \neq \text{NIL}$.
2. then print key [x]
3. PREORDER-TREE-WALK (left [x]).
4. PREORDER-TREE-WALK (right [x]).

3. **POSTORDER-TREE-WALK (x):** In which we visit the root node after the nodes in its subtree.

POSTORDER-TREE-WALK (x):

1. If $x \neq \text{NIL}$.
2. then POSTORDER-TREE-WALK (left [x]).
3. POSTORDER-TREE-WALK (right [x]).
4. print key [x]

Querying a Binary Search Trees

1. **Searching:** The TREE-SEARCH (x, k) algorithm searches the tree node at x for a node whose key value is equal to k . It returns a pointer to the node if it exists otherwise NIL.

TREE-SEARCH (x, k)

1. If $x = \text{NIL}$ or $k = \text{key}[x]$.
2. then return x .
3. If $k < \text{key}[x]$.
4. then return TREE-SEARCH (left [x], k)
5. else return TREE-SEARCH (right [x], k)



Clearly, this algorithm runs in $O(h)$ time where h is the height of the tree. The iterative version of the above algorithm is very easy to implement.

ITERATIVE-TREE-SEARCH (x, k)

1. while $x \neq \text{NIL}$ and $k \neq \text{key}[k]$.
2. do if $k < \text{key}[x]$.
3. then $x \leftarrow \text{left}[x]$.
4. else $x \leftarrow \text{right}[x]$.
5. return x .

2. Minimum and Maximum: An item in a binary search tree whose key is a minimum can always be found by following left child pointers from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x .

TREE-MINIMUM (x)

1. While $\text{left}[x] \neq \text{NIL}$.
2. do $x \leftarrow \text{left}[x]$.
3. return x .

TREE-MAXIMUM (x)

1. While $\text{left}[x] \neq \text{NIL}$.
2. do $x \leftarrow \text{right}[x]$.
3. return x .

3. Successor and Predecessor: Given a node in a binary search tree, sometimes we used to find its successor in the sorted form determined by an inorder tree walk. If all keys are specific, the successor of a node x is the node with the smallest key greater than $\text{key}[x]$. The structure of a binary search tree allows us to rule the successor of a node without ever comparing keys. The following action returns the successor of a node x in a binary search tree if it exists, and NIL if x has the greatest key in the tree:

TREE SUCCESSOR (x)

1. If right [x] ≠ NIL.
2. Then return TREE-MINIMUM (right [x]))
3. $y \leftarrow p[x]$
4. While $y \neq \text{NIL}$ and $x = \text{right}[y]$
5. do $x \leftarrow y$
6. $y \leftarrow p[y]$
7. return y.

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node x is non-empty, then the successor of x is just the leftmost node in the right subtree, which we find in line 2 by calling TREE-MINIMUM (right [x]). On the other hand, if the right subtree of node x is empty and x has a successor y , then y is the lowest ancestor of x whose left child is also an ancestor of x . To find y , we quickly go up the tree from x until we encounter a node that is the left child of its parent; lines 3-7 of TREE-SUCCESSOR handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$ since we either follow a simple path up the tree or follow a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

4. Insertion in Binary Search Tree: To insert a new value into a binary search tree T , we use the procedure TREE-INSERT. The procedure takes a node for which key [z] = v, left [z] NIL, and right [z] = NIL. It modifies T and some of the attributes of z in such a way that it inserts into an appropriate position in the tree.

TREE-INSERT (T, z)

1. $y \leftarrow \text{NIL}$
2. $x \leftarrow \text{root}[T]$
3. while $x \neq \text{NIL}$



4. do $y \leftarrow x$
5. if $\text{key}[z] < \text{key}[x]$
6. then $x \leftarrow \text{left}[x]$
7. else $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. if $y = \text{NIL}$
10. then root $[T] \leftarrow z$
11. else if $\text{key}[z] < \text{key}[y]$
12. then left $[y] \leftarrow z$

For Example:

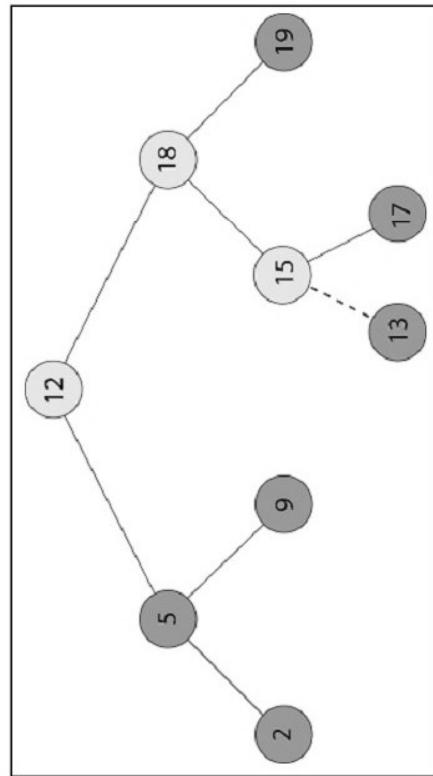


Fig. 9.3: Working of TREE-INSERT

Suppose, we want to insert an item with key 13 into a binary search tree.

- | | |
|----|----------------------------------|
| 1. | $x = 1$ |
| 2. | $y = 1$ as $x \neq \text{NIL}$. |
| 3. | $\text{Key}[z] < \text{key}[x]$ |
| 4. | $13 <$ not equal to 12. |
| 5. | $x \leftarrow \text{right}[x]$. |
| 6. | $x \leftarrow 3$ |

- ```

7. Again x ≠ NIL
8. y ← 3
9. key [z] < key [x]
10. 13 < 18
11. x←left [x]
12. x←6
13. Again x ≠ NIL, y←6
14. 13 < 15
15. x←left [x]
16. x←NIL
17. p [z]←6

```

Now our node z will be either left or right child of its parent (y).

- ```

1. key [z] < key [y]
2. 13 < 15
3. Left [y] ← z
4. Left [6] ← z

```

So, insert a node in the left of node index at 6.

5. Deletion in Binary Search Tree: When deleting a node from a tree, it is essential that any relationships implicit in the tree can be maintained. The deletion of nodes from a binary search tree will be considered.

There are three distinct cases:

1. **Nodes with no children:** This case is trivial. Simply set the parent's pointer to the node to be deleted to NIL and delete the node.
2. **Nodes with one child:** When z has no left child, then we replace z by its right child which may or may not be NIL. And when z has no right child, then we replace z with its right child.



3. Nodes with both Children: When z has both left and right child, we find z's successor y, which lies in right z's right subtree and has no left child (the successor of z will be a node with minimum value its right subtree and so it has no left child).

- If y is z's right child, then we replace z.
- Otherwise, y lies within z's right subtree but not z's right child. In this case, we first replace z by its own right child and the replace z by y.

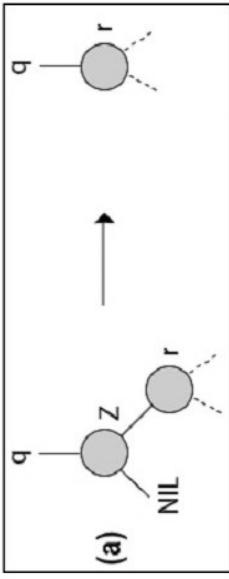
TREE-DELETE (T, z)

1. If left [z] = NIL or right [z] = NIL
2. Then y \leftarrow z
3. Else y \leftarrow TREE-SUCCESSOR (z)
4. If left [y] \neq NIL
5. Then x \leftarrow left [y]
6. Else x \leftarrow right [y]
7. If x \neq NIL
8. Then p[x] \leftarrow p[y]
9. If p[y] = NIL
10. Then root [T] \leftarrow x
11. Else if y = left [p[y]]
12. Then left [p[y]] \leftarrow x
13. Else right [p[y]] \leftarrow y
14. If y \neq z
15. Then key [z] \leftarrow key [y]
16. If y has other fields, copy them, too
17. Return y

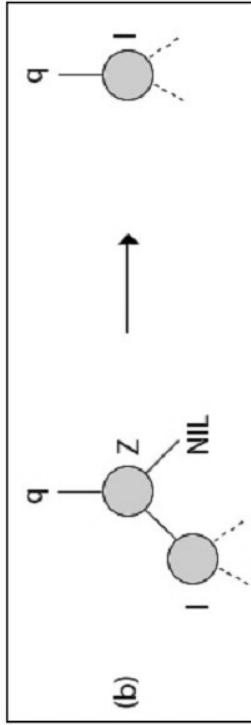
The procedure runs in O(h) time on a tree of height h.



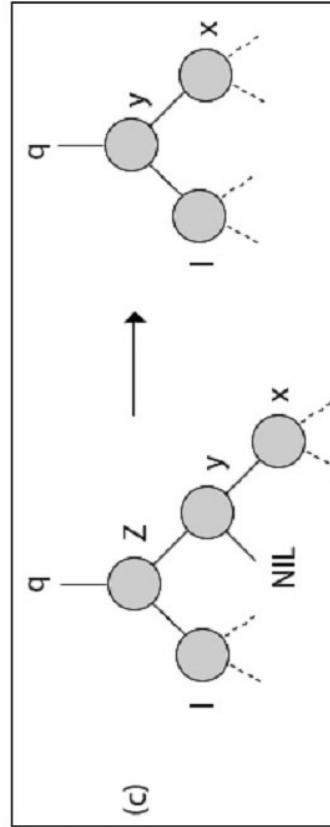
For Example: Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.



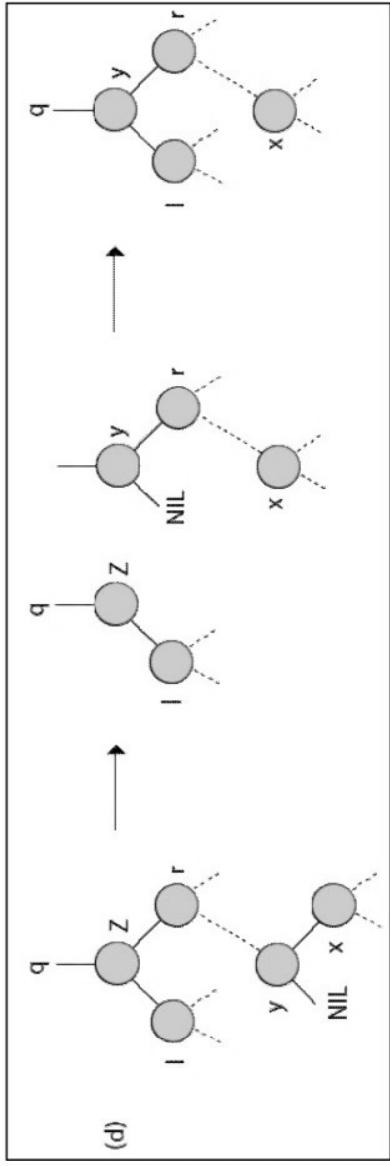
Z has the only right child.



Z has the only left child. We replace z by l.



Node z has two children; its left child is node l, its right child is its successor y, and y's right child is node x. We replace z by y, updating y's left child to become l, but leaving x as y's right child.



Node z has two children (left child l and right child r), and its successor y \neq r lies within the subtree rooted at r. We replace y with its own right child x, and we set y to be r's parent. Then, we set y to be q's child and the parent of l.

9.5 Bellman-Ford Algorithm

Solves single shortest path problem in which edge weight may be negative, but no negative cycle exists.

This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination.

It is slower than Dijkstra's algorithm but more versatile, as it capable of handling some of the negative weight edges.

This algorithm detects the negative cycle in a graph and reports their existence.

Based on the '**Principle of relaxation**' in which more accurate values gradually recover an approximation to the proper distance by until eventually reaching the optimum solution.

Given a weighted directed graph $G = (V, E)$ with source s and weight function $w: E \rightarrow R$, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative weight cycle that is attainable from the source. If there is such a cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if a graph contains no negative weight cycles that are reachable from the source.

Recurrence Relation

$\text{dist}^k[u] = [\min[\text{dist}^{k-1}[u], \min[\text{dist}^{k-1}[i] + \text{cost}[i, u]]]$ as i except u.

k → k is the source vertex

u → u is the destination vertex

i → no. of edges to be scanned concerning a vertex.

BELLMAN-FORD (G, w, s)

1. INITIALISE - SINGLE - SOURCE (G, s)
2. for i ← 1 to |V[G]| - 1
3. do for each edge (u, v) ∈ E [G]
4. do RELAX (u, v, w)
5. for each edge (u, v) ∈ E [G]
6. do if d[v] > d[u] + w(u, v)
7. then return FALSE.
8. return TRUE.

Example: Here first we list all the edges and their weights.

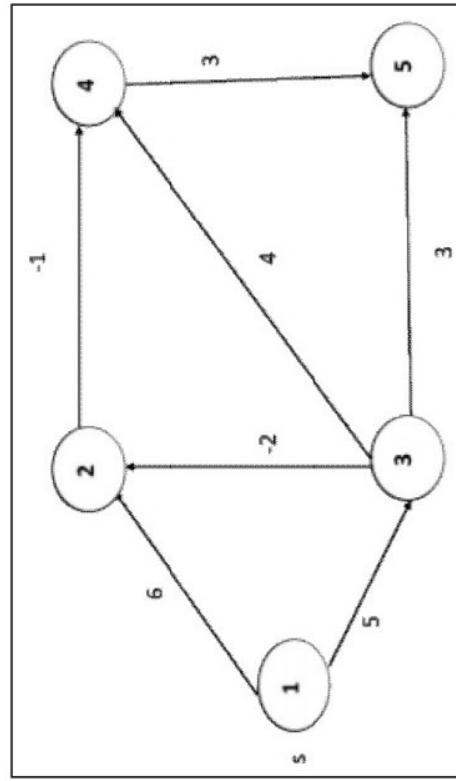


Fig. 9.4: Graph and their Weights

Solution: $\text{dist}^k[u] = [\min[\text{dist}^{k-1}[u], \min[\text{dist}^{k-1}[i] + \text{cost}[i,u]]]$ as $i \neq u$.

Table 9.2: Solution

No of Edges Traversed	Vertices				
	1	2	3	4	5
1	0	6	5	∞	∞
2	0	3	5	5	8
3	0	3	5	2	8
4	0	3	5	2	5

$\text{dist}^2[2] = \min[\text{dist}^1[2], \min[\text{dist}^1[1] + \text{cost}[1,2], \text{dist}^1[3] + \text{cost}[3,2], \text{dist}^1[4] + \text{cost}[4,2], \text{dist}^1[5] + \text{cost}[5,2]]]$

Min = [6, 0 + 6, 5 + (-2), $\infty + \infty$, $\infty + \infty$] = 3

$\text{dist}^2[3] = \min[\text{dist}^1[3], \min[\text{dist}^1[1] + \text{cost}[1,3], \text{dist}^1[2] + \text{cost}[2,3], \text{dist}^1[4] + \text{cost}[4,3], \text{dist}^1[5] + \text{cost}[5,3]]]$

Min = [5, 0 + ∞ , 6 + ∞ , $\infty + \infty$, $\infty + \infty$] = 5

$\text{dist}^2[4] = \min[\text{dist}^1[4], \min[\text{dist}^1[1] + \text{cost}[1,4], \text{dist}^1[2] + \text{cost}[2,4], \text{dist}^1[3] + \text{cost}[3,4], \text{dist}^1[5] + \text{cost}[5,4]]]$

Min = [∞ , 0 + ∞ , 6 + (-1), 5 + 4, $\infty + \infty$] = 5

$\text{dist}^2[5] = \min[\text{dist}^1[5], \min[\text{dist}^1[1] + \text{cost}[1,5], \text{dist}^1[2] + \text{cost}[2,5], \text{dist}^1[3] + \text{cost}[3,5], \text{dist}^1[4] + \text{cost}[4,5]]]$

Min = [∞ , 0 + ∞ , 6 + ∞ , 5 + 3, $\infty + 3$] = 8

$\text{dist}^3[2] = \min[\text{dist}^2[2], \min[\text{dist}^2[1] + \text{cost}[1,2], \text{dist}^2[3] + \text{cost}[3,2], \text{dist}^2[4] + \text{cost}[4,2], \text{dist}^2[5] + \text{cost}[5,2]]]$

Min = [3, 0 + 6, 5 + (-2), 5 + ∞ , 8 + ∞] = 3

$\text{dist}^3[3] = \min[\text{dist}^2[3], \min[\text{dist}^2[1] + \text{cost}[1,3], \text{dist}^2[2] + \text{cost}[2,3], \text{dist}^2[4] + \text{cost}[4,3], \text{dist}^2[5] + \text{cost}[5,3]]]$

$\text{Min} = [5, 0 + \infty, 3 + \infty, 5 + \infty, 8 + \infty] = 5$

$\text{dist}^3[4] = \min[\text{dist}^2[4], \min[\text{dist}^2[1] + \text{cost}[1,4], \text{dist}^2[2] + \text{cost}[2,4], \text{dist}^2[3] + \text{cost}[3,4], \text{dist}^2[5] + \text{cost}[5,4]]]$

$\text{Min} = [5, 0 + \infty, 3 + (-1), 5 + 4, 8 + \infty] = 2$

$\text{dist}^3[5] = \min[\text{dist}^2[5], \min[\text{dist}^2[1] + \text{cost}[1,5], \text{dist}^2[2] + \text{cost}[2,5], \text{dist}^2[3] + \text{cost}[3,5], \text{dist}^2[4] + \text{cost}[4,5]]]$

$\text{Min} = [8, 0 + \infty, 3 + \infty, 5 + 3, 5 + 3] = 8$

$\text{dist}^4[2] = \min[\text{dist}^3[2], \min[\text{dist}^3[1] + \text{cost}[1,2], \text{dist}^3[3] + \text{cost}[3,2], \text{dist}^3[4] + \text{cost}[4,2], \text{dist}^3[5] + \text{cost}[5,2]]]$

$\text{Min} = [3, 0 + 6, 5 + (-2), 2 + \infty, 8 + \infty] = 3$

$\text{dist}^4[3] = \min[\text{dist}^3[3], \min[\text{dist}^3[1] + \text{cost}[1,3], \text{dist}^3[2] + \text{cost}[2,3], \text{dist}^3[4] + \text{cost}[4,3], \text{dist}^3[5] + \text{cost}[5,3]]]$

$\text{Min} = [5, 0 + \infty, 3 + \infty, 2 + \infty, 8 + \infty] = 5$

$\text{dist}^4[4] = \min[\text{dist}^3[4], \min[\text{dist}^3[1] + \text{cost}[1,4], \text{dist}^3[2] + \text{cost}[2,4], \text{dist}^3[3] + \text{cost}[3,4], \text{dist}^3[5] + \text{cost}[5,4]]]$

$\text{Min} = [2, 0 + \infty, 3 + (-1), 5 + 4, 8 + \infty] = 2$

$\text{dist}^4[5] = \min[\text{dist}^3[5], \min[\text{dist}^3[1] + \text{cost}[1,5], \text{dist}^3[2] + \text{cost}[2,5], \text{dist}^3[3] + \text{cost}[3,5], \text{dist}^3[5] + \text{cost}[4,5]]]$

$\text{Min} = [8, 0 + \infty, 3 + \infty, 8, 5] = 5$

9.6 Travelling Salesman Problem

In the travelling salesman problem, a salesman must visit n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost $c(i, j)$ to travel from the city i to city j . The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called travelling salesman problem (TSP).



We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

If we assume the cost function c satisfies the triangle inequality, then we can use the following approximate algorithm:

Triangle Inequality

Let u, v, w be any three vertices, we have

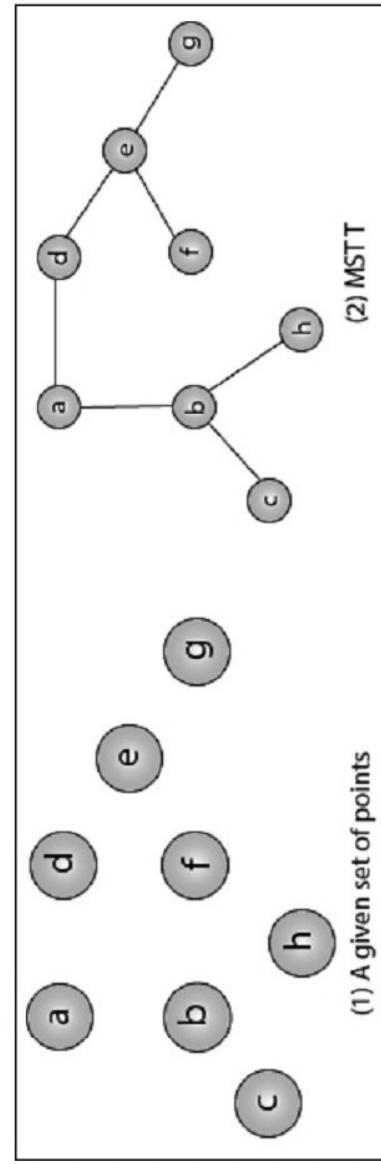
$$c(u, w) \leq c(u, v) + c(v, w)$$

One important observation to develop an approximate solution is if we remove an edge from H^* , the tour becomes a spanning tree.

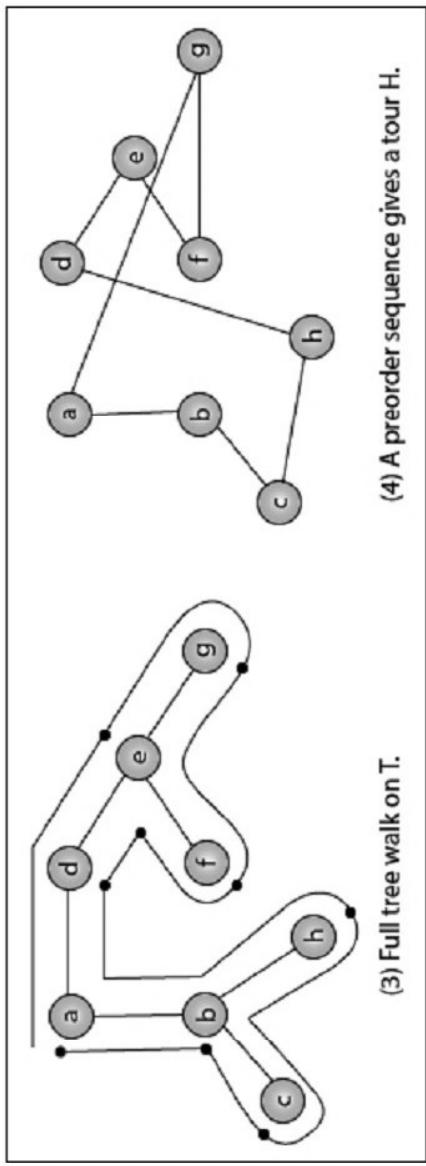
- ```

1. Approx-TSP (G=(V, E))
2. {
3. 1. Compute a MST T of G;
4. 2. Select any vertex r is the root of the tree;
5. 3. Let L be the list of vertices visited in a preorder tree walk of T;
6. 4. Return the Hamiltonian cycle H that visits the vertices in the order L;
7. }
```

### Travelling Salesman Problem



**Fig. 9.5 (a): Travelling Salesman Problem**



(3) Full tree walk on T.

**Fig. 9.5 (b): Full Tree Walk on T**

Intuitively, Approximate TSP first makes a full walk of MST T, which visits each edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut).

## 9.7 Reliability Design

Algorithm design is a specific method to create a mathematical process in problem-solving processes. Applied algorithm design is algorithm engineering. In computer science, the analysis of algorithms is the determination of the computational complexity of algorithms, that is the amount of time, storage and/or other resources necessary to execute them.

## 9.8 Summary

**All Pairs Shortest Path:** The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be solved using applications of Dijkstra's algorithm or all at once using the Floyd-Warshall algorithm.

**Binary search tree** is a node based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

In the travelling salesman problem, a salesman must visit  $n$  cities. We can say that the salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost  $c(i, j)$  to travel from the city  $i$  to city  $j$ . The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called travelling salesman problem (TSP).

## 9.9 Key Words/Abbreviations

- **Cycle** is a path that starts and ends at the same vertex.
- **Simple path** is a path with distinct vertices.
- **Directed path** is a path of only directed edges.
- **Directed cycle** is a cycle of only directed edges.
- **Subgraph** is a subset of vertices and edges.
- **Spanning subgraph** contains all the vertices.

## 9.10 Learning Activity

1. How do you use Warshall's algorithm?

.....  
.....

2. How do you solve Floyd's algorithm?

.....  
.....

3. How do you find the optimal binary search tree?

.....  
.....

4. What is the difference in Kruskal's and Prim's algorithm?

.....  
.....

5. What is meant by optimal binary search tree?  
.....  
.....
6. What is optimal binary search tree in dynamic programming?  
.....  
.....

7. What is the time complexity of optimal binary search?  
.....  
.....

8. How do you solve Bellman-Ford algorithm?  
.....  
.....

9. What is the running time of Bellman-Ford algorithm?  
.....  
.....

10. Is Bellman-Ford algorithm greedy?  
.....  
.....

11. Why is the travelling salesman problem important?  
.....  
.....
12. What is travelling salesman problem and how is it modeled as a graph problem?  
.....  
.....

## 9.11 Unit End Questions (MCQ and Descriptive)

### A. Descriptive Types Questions

1. What is the specialty about the inorder traversal of a binary search tree?
2. How many solution/solutions are available for a graph having negative weight cycle?
3. What is the running time of Bellman-Ford algorithm?
4. What is the basic principle behind Bellman-Ford algorithm?
5. What procedure is being followed in Floyd-Warshall algorithm?
6. What happens when the value of k is 0 in the Floyd-Warshall algorithm?

### B. Multiple Choice/Objective Type Questions

1. Floyd-Warshall's algorithm is used for solving \_\_\_\_\_.  
 (a) All pair shortest path problems    (b) Single-source shortest path problems  
 (c) Network flow problems                (d) Sorting problems
2. Floyd-Warshall's algorithm can be applied on \_\_\_\_\_.  
 (a) Undirected and unweighted graphs (b) Undirected graphs  
 (c) Directed graphs                        (d) Acyclic graphs
3. What is the running time of the Floyd-Warshall algorithm?  
 (a) Big-O ( $V$ )                            (b) Theta ( $V^2$ )  
 (c) Big-O ( $VE$ )                            (d) Theta ( $V^3$ )
4. What approach is being followed in Floyd-Warshall algorithm?  
 (a) Greedy technique                        (b) Dynamic programming  
 (c) Linear programming                      (d) Backtracking
5. Which of the following is false about a binary search tree?  
 (a) The left child is always lesser than its parent  
 (b) The right child is always greater than its parent  
 (c) The left and right subtrees should also be binary search trees  
 (d) Inorder sequence gives decreasing order of elements



6. What are the worst-case and average case complexities of a binary search tree?
  - (a)  $O(n)$ ,  $O(n)$
  - (b)  $O(\log n)$ ,  $O(\log n)$
  - (c)  $O(\log n)$ ,  $O(n)$
  - (d)  $O(n)$ ,  $O(\log n)$
7. Which of the following is false in the case of a spanning tree of a graph G?
  - (a) It is tree that spans G
  - (b) It is a subgraph of the G
  - (c) It includes every vertex of the G
  - (d) It can be either cyclic or acyclic
8. Every graph has only one minimum spanning tree.
  - (a) True
  - (b) False
9. Consider a complete graph G with 4 vertices. The graph G has \_\_\_\_\_ spanning trees.
  - (a) 15
  - (b) 8
  - (c) 16
  - (d) 13
10. The travelling salesman problem can be solved using \_\_\_\_\_.
  - (a) A spanning tree
  - (b) A minimum spanning tree
  - (c) Bellman – Ford algorithm
  - (d) DFS traversal

**Answers:**

1. (a), 2. (c), 3. (d), 4. (b), 5. (d), 6. (d), 7. (d), 8. (b), 9. (c), 10. (b)

---

## 9.12 References

References of this unit have been given at the end of the book.



---

## **UNIT 10 BACKTRACKING 1**

---

### **Structure:**

- 10.0 Learning Objectives
- 10.1 Introduction
- 10.2 General Method: N-Queens Problem
- 10.3 Sum of Subset Problem
- 10.4 Graph Colouring
- 10.5 Hamilton Cycles
- 10.6 Summary
- 10.7 Key Words/Abbreviations
- 10.8 Learning Activity
- 10.9 Unit End Questions (MCQ and Descriptive)
- 10.10 References

---

### **10.0 Learning Objectives**

---

After studying this unit, you will be able to:

- Explain n-queens problem
- Describe graph colouring
- Illustrate Hamiltonian cycles
- Describe introduction of backtracking
- Solve subset sum problem



- Ensure applications of graph colouring
- Gain the confidence about graph colouring

## 10.1 Introduction

---

Backtracking is an algorithmic method to solve a problem in an additional way. It uses a recursive approach to explain the problems. We can say that backtracking is needed to find all possible combinations to solve an optimisation problem. **Backtracking** is a systematic way of trying out different sequences of decisions until we find one that ‘works.’ Generally, however, we draw our trees downward, with the root at the top. A tree is composed of nodes.

**Backtracking can be understood of as searching a tree for a particular ‘goal’ leaf node.**

Backtracking is undoubtedly quite simple - we ‘explore’ each node, as follows:

To ‘explore’ node N:

1. If N is a goal node, return ‘success’,
2. If N is a leaf node, return ‘failure’,
3. For each child C of N,  
Explore C.  
If C was successful, return ‘success’,  
4. Return ‘failure’.

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a **depth-first search** with any bounding function. All solutions using backtracking are needed to satisfy a complex set of constraints. The constraints may be explicit or implicit.

**Explicit constraint** is ruled, which restricts each vector element to be chosen from the given set.

**Implicit constraint** is ruled, which determines each of the tuples in the solution space, actually satisfying the criterion function.

## 10.2 General Method: N-Queens Problem

N-queens problem is to place n-queens in such a manner on an  $n \times n$  chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for  $n = 1$ , the problem has a trivial solution, and no solution exists for  $n = 2$  and  $n = 3$ . So, first we will consider the 4-queens problem and then generate it to n-queens problem.

Given a  $4 \times 4$  chessboard and number the rows and column of the chessboard 1 through 4.

**Table 10.1: 4×4 Chessboard**

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

**4x4 chessboard**

Since, we have to place 4 queens such as  $q_1$   $q_2$   $q_3$  and  $q_4$  on the chessboard, such that no two queens attack each other. In such a condition, each queen must be placed on a different row, i.e., we put queen ‘i’ on row ‘i’.

Now, we place queen  $q_1$  in the very first acceptable position (1, 1). Next, we put queen  $q_2$  so that both these queens do not attack each other. We find that if we place  $q_2$  in column 1 and 2, then the dead end is encountered. Thus, the first acceptable position for  $q_2$  in column 3, i.e., (2, 3) but then no position is left for placing queen ‘ $q_3$ ’ safely. So, we backtrack one step and place the queen ‘ $q_2$ ’ in (2, 4), the next best possible solution. Then we obtain the position for placing ‘ $q_3$ ’ which is (3, 2). But later this position also leads to a dead end, and no place is found where ‘ $q_4$ ’ can be placed safely. Then we have to backtrack till ‘ $q_1$ ’ and place it to (1, 2) and then all other queens are placed safely by moving  $q_2$  to (2, 4),  $q_3$  to (3, 1) and  $q_4$  to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4-queens problems is (3, 1, 4, 2), i.e.,

**Table 10.2: 4-Queens Problems**

|   | 1              | 2              | 3              | 4              |
|---|----------------|----------------|----------------|----------------|
| 1 |                |                | q <sub>1</sub> |                |
| 2 | q <sub>2</sub> |                |                |                |
| 3 |                |                |                | q <sub>3</sub> |
| 4 |                | q <sub>4</sub> |                |                |

The implicit tree for 4-queens problem for a solution (2, 4, 1, 3) is as follows:

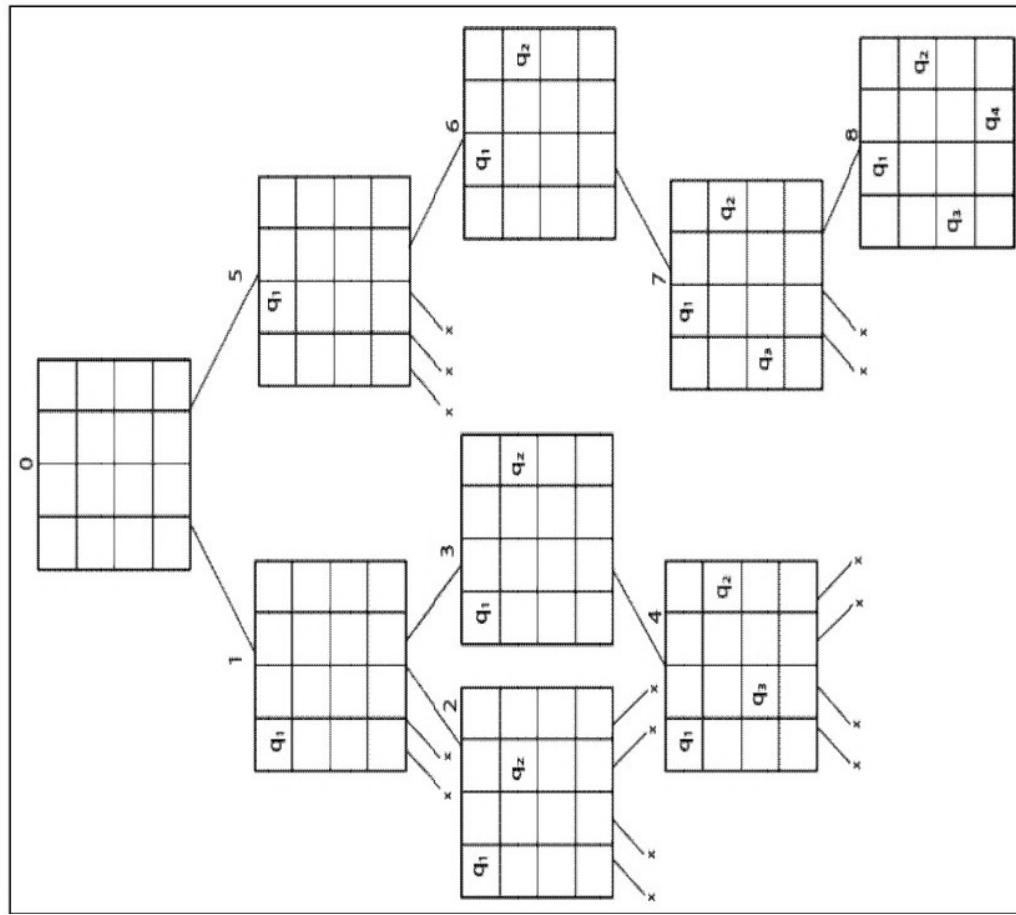
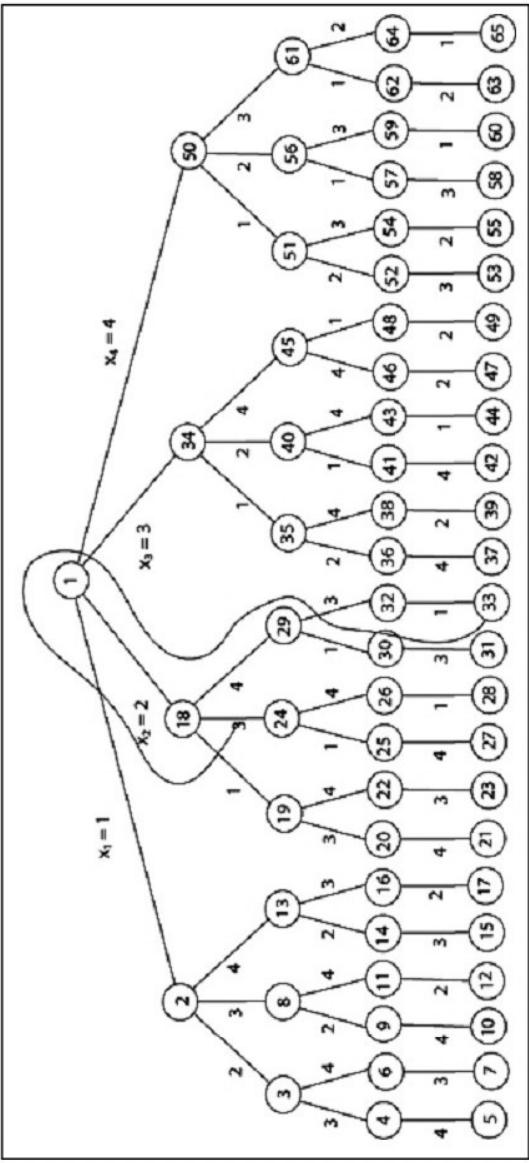
**Fig. 10.1: Implicit Tree for 4-Queen Problem**

Figure bellow shows the complete state space for 4-queens problem. But, we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



**Fig. 10.2: 4-Queens Solution Space with Nodes Numbered in DFS**

It can be seen that all the solutions to the 4-queens problem can be represented as 4 tuples  $(x_1, x_2, x_3, x_4)$  where  $x_i$  represents the column on which queen ' $q_i$ ' is placed.

One possible solution for 8-queens problem is shown in Table:

**Table 10.3: 8-Queens Problem**

|   | 1 | 2        | 3     | 4     | 5        | 6     | 7     | 8 |
|---|---|----------|-------|-------|----------|-------|-------|---|
| 1 |   |          | $q_1$ |       |          |       |       |   |
| 2 |   |          |       |       | $q_{12}$ |       |       |   |
| 3 |   |          |       |       |          |       |       |   |
| 4 |   |          |       | $q_4$ |          |       |       |   |
| 5 |   |          |       |       |          |       | $q_5$ |   |
| 6 |   | $q_{16}$ |       |       |          |       |       |   |
| 7 |   |          |       |       | $q_7$    |       |       |   |
| 8 |   |          |       |       |          | $q_8$ |       |   |

1. Thus, the solution for 8-queens problem for (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only if  $(i - j) = k - l$  or  $i + j = k + l$ .
4. The first equation implies that  $j - l = i - k$ .
5. The second equation implies that  $j - l = k - i$ .
6. Therefore, two queens lie on the duplicate diagonal if and only if  $|j - l| = |i - k|$

Placed (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs  $x_1, x_2, \dots, x_{k-1}$  and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n-queens problem.

```

1. Place (k, i)
2.
3. For j \leftarrow 1 to k - 1
4. do if ($x[j] = i$)
 or ($Abs\ x[j] - i = (Abs\ (j - k))$)
6. then return false;
7. return true;
8. }
```

Placed (k, i) returns true if a queen can be placed in the kth row and ith column is otherwise return is false.  
 $x[]$  is a global array whose final  $k - 1$  values have been set. Abs (r) returns the absolute value of r.

```

1. N - Queens (k, n)
2.
3. For i \leftarrow 1 to n
4. do if Place (k, i) then
5. {
6. $x[k] \leftarrow i;$
```

```

7.. if (k ==n) then
8. write (x [1....n));
9. else
10. N - Queens (k + 1, n);
11. }
12. }

```

### 10.3 Sum of Subset Problem

#### Subset Sum Problem

The subset sum problem is to find a subset 's' of the given set  $S = (S_1 S_2 S_3...S_n)$  where the elements of the set  $S$  are  $n$  positive integers in such a manner that  $s' \in S$  and sum of the elements of subset 's' is equal to some positive integer 'X'.

The subset sum problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that it represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in increasing order:

$$S_1 \leq S_2 \leq S_3 \dots \leq S_n$$

The left child of the root node indicates that we have to include ' $S_1$ ' from the set ' $S$ ' and the right child of the root indicates that we have to execute ' $S_1'$ . Each node stores the total of the partial solution elements. If at any stage the sum equals to 'X', then the search is successful and terminates.

The dead end in the tree appears only when either of the two inequalities exist:

- The sum of  $s'$  is too large, i.e.,

$$s' + S_i + 1 > X$$

- The sum of  $s'$  is too small, i.e.,

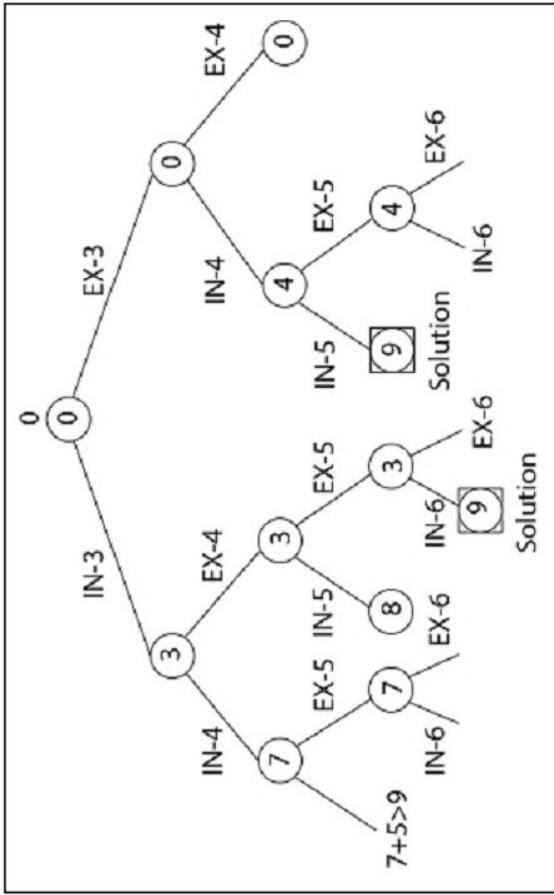
$$s' + \sum_{j=i+1}^n S_j < X$$

**Example:** Given a set  $S = \{3, 4, 5, 6\}$  and  $X = 9$ . Obtain the subset sum using backtracking approach.

**Solution:**

- Initially  $S = \{3, 4, 5, 6\}$  and  $X = 9$ .
  - $S' = \{\emptyset\}$

The implicit binary tree for the subset sum problem is as shown in following figure:



**Fig. 10.3:** Implicit Binary Tree for the Subset Sum Problem

The number inside a node is the sum of the partial solution elements at a particular level.

Thus, if our partial solution elements sum is equal to the positive integer 'X' then at that time search will terminate, or it continues if all the possible solutions need to be obtained.

---

**10.4 Graph Colouring**

Graph colouring is nothing but a simple way of labelling graph components such as vertices, edges, and regions under some constraints. In a graph, no two adjacent vertices, adjacent edges, or adjacent regions are coloured with minimum number of colours. This number is called the **chromatic number** and the graph is called a **properly coloured graph**.

While graph colouring, the constraints that are set on the graph are colours, order of colouring, the way of assigning colour, etc. A colouring is given to a vertex or a particular region. Thus, the vertices or regions having same colours form independent sets.

Vertex Colouring

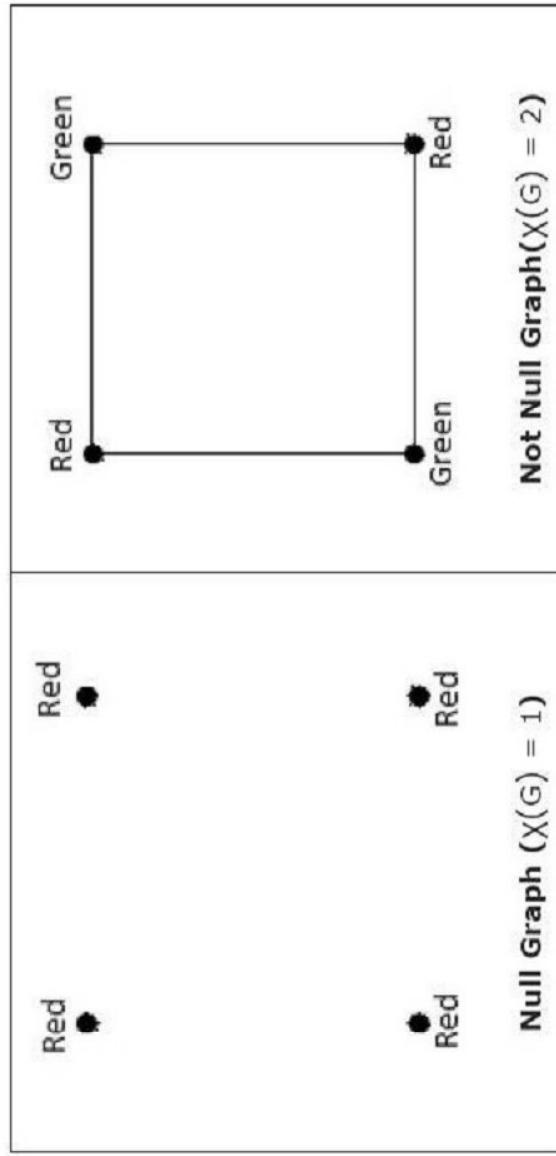
Vertex colouring is an assignment of colours to the vertices of a graph ‘G’ such that no two adjacent vertices have the same colour. Simply put, no two vertices of an edge should be of the same colour.

Chromatic Number

The minimum number of colours required for vertex colouring of graph ‘G’ is called as the chromatic number of G, denoted by  $X(G)$ .

$\chi(G) = 1$  if and only if ' $G$ ' is a null graph. If ' $G$ ' is not a null graph, then  $\chi(G) \geq 2$ .

### Example:



**Fig. 10.4:** Vertex Colouring of Graph

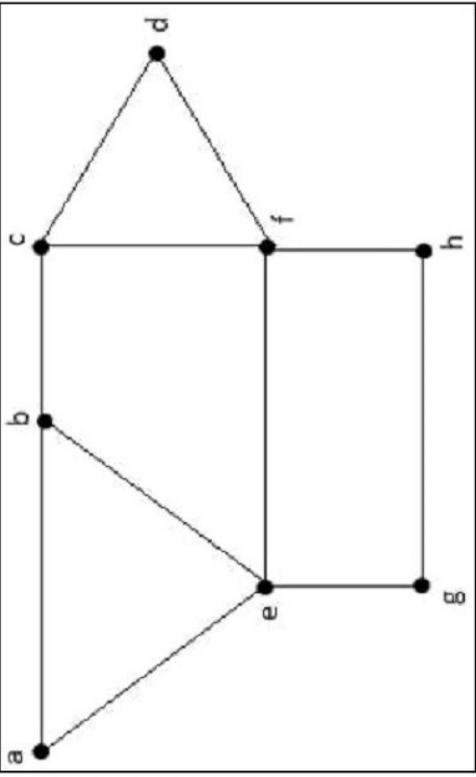
**Note:** A graph ‘G’ is said to be  $n$  coverable if there is a vertex colouring that uses at most  $n$  colours, i.e.,  $X(G) \leq n$ .

**Region Colouring**

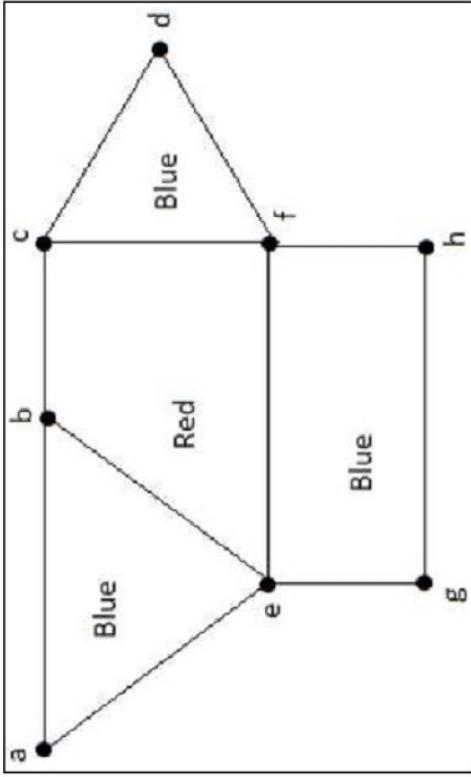
Region colouring is an assignment of colours to the regions of a planar graph such that no two adjacent regions have the same colour. Two regions are said to be adjacent if they have a common edge.

**Example:**

Take a look at the following graph. The regions ‘aeb’ and ‘befc’ are adjacent, as there is a common edge ‘be’ between those two regions.



Similarly, the other regions are also coloured based on the adjacency. This graph is coloured as follows:

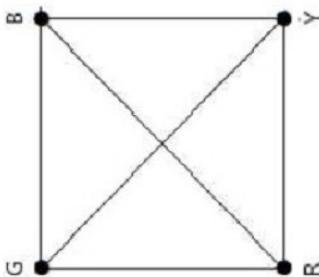


**Example:**

The chromatic number of  $K_n$  is:

- a) n
- b)  $n-1$
- c)  $[n/2]$
- d)  $[n/2]$

Consider this example with  $K_4$ .



In the complete graph, each vertex is adjacent to remaining  $(n - 1)$  vertices. Hence, each vertex requires a new colour. Hence, the chromatic number of  $K_n = n$ .

### **Applications of Graph Colouring**

Graph colouring is one of the most important concepts in graph theory. It is used in many real-time applications of computer science such as –

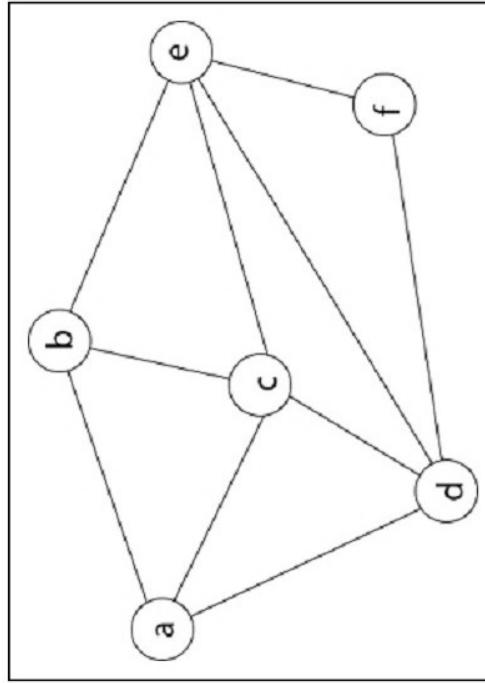
- Clustering
- Data mining
- Image capturing
- Image segmentation
- Networking
- Resource allocation
- Processes scheduling

## 10.5 Hamilton Cycles

### Hamiltonian Circuit Problems

Given a graph  $G = (V, E)$  we have to find the Hamiltonian circuit using backtracking approach. We start our search from any arbitrary vertex say 'a'. This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian cycle that is to be constructed. The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that dead end is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

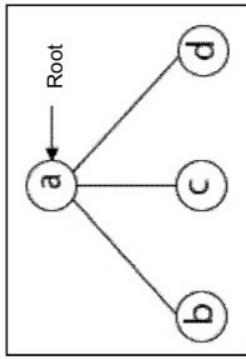
**Example:** Consider a graph  $G = (V, E)$  shown in figure below. We have to find a Hamiltonian circuit using backtracking method.



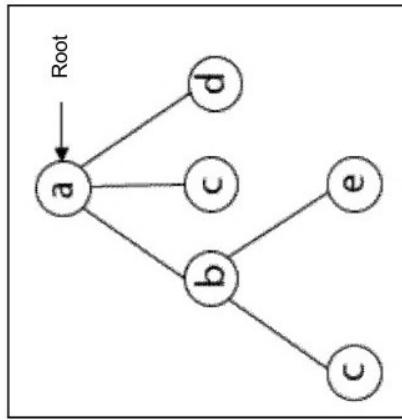
**Solution:** Firstly, we start our search with vertex 'a'. This vertex 'a' becomes the root of our implicit tree.



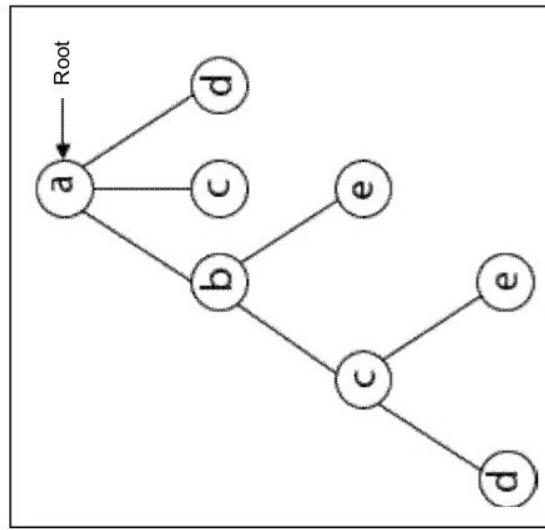
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



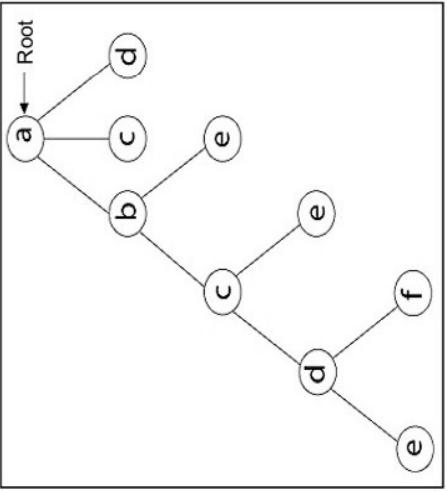
Next, we select 'c' adjacent to 'b'.



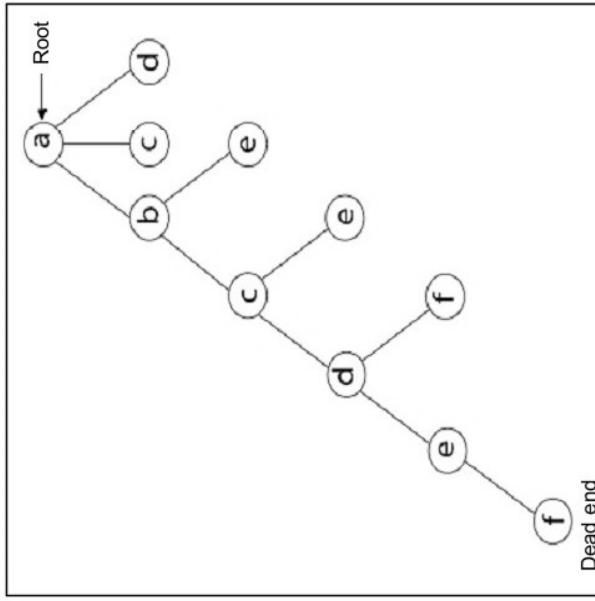
Next, we select 'd' adjacent to 'c'.



Next, we select 'e' adjacent to 'd'.

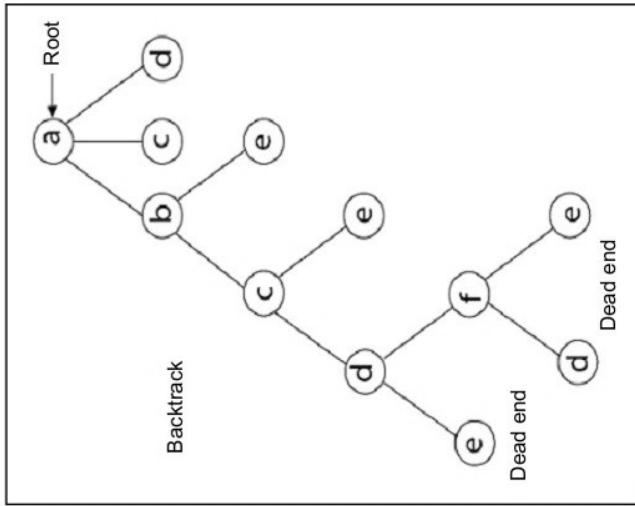
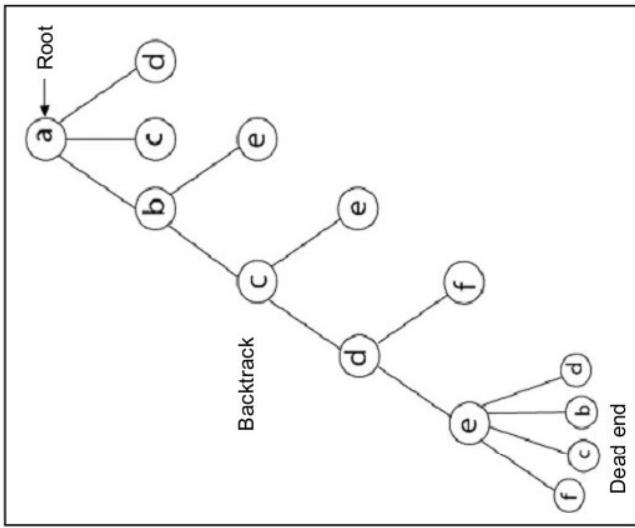


Next, we select vertex 'f' adjacent to 'e'. The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to 'd' are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a'. Here, we get the Hamiltonian cycle as all the vertex other than the start vertex 'a' is visited only once (a - b - c - e - f - d - a).



Again backtrack.



Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

## 10.6 Summary

---

Backtracking is a systematic way of trying out different sequences of decisions until we find one that ‘works’. Generally, however, we draw our trees downward, with the root at the top. A tree is composed of nodes. Backtracking can be understood of as searching a tree for a particular ‘goal’ leaf node.

N-queens problem is to place n-queens in such a manner on an  $n \times n$  chessboard that no queens attack each other by being in the same row, column or diagonal. It can be seen that for  $n = 1$ , the problem has a trivial solution, and no solution exists for  $n = 2$  and  $n = 3$ . So, first we will consider the 4-queens problem and then generate it to n-queens problem.

The subset sum problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that it represents that no decision is yet taken on any input.

Graph colouring is nothing but a simple way of labelling graph components such as vertices, edges, and regions under some constraints. In a graph, no two adjacent vertices, adjacent edges, or adjacent regions are coloured with minimum number of colours. This number is called the **chromatic number** and the graph is called a **properly coloured graph**.

## 10.7 Key Words/Abbreviations

---

- A set,  $V$ , of vertices (nodes).
- A collection,  $E$ , of pairs of vertices from  $V$  called **edges** (arcs).
- **Directed**, if the pairs are ordered  $(u, v)$   $u$  the **origin**,  $v$  the **destination**.
- **Directed graph** (di-graph), if all the edges are directed.
- **Undirected graph** (graph), if all the edges are undirected.
- **Mixed graph**, if edges are both directed or undirected.

- End vertices of an edge are the **end points** of the edge.
- An edge is **incident** on a vertex if the vertex is an end point of the edge.
- **Outgoing edges** of a vertex are directed edges that the vertex is the origin.
- **Incoming edges** of a vertex are directed edges that the vertex is the destination.
- Degree of a vertex,  $v$ , denoted  $\deg(v)$  is the number of incident edges.
- Out-degree,  $\text{outdeg}(v)$ , is the number of outgoing edges.
- In-degree,  $\text{indeg}(v)$ , is the number of incoming edges.
- Parallel edges or multiple edges are edges of the same type and end vertices
- Self-loop is an edge with the end vertices of the same vertex.

## 10.8 Learning Activity

1. What is vertex colouring of a graph?  
.....  
.....
2. How many edges will a tree consisting of  $N$  nodes have?  
.....  
.....
3. How many unique colours will be required for proper vertex colouring of an empty graph having  $n$  vertices?  
.....  
.....
4. How many unique colours will be required for proper vertex colouring of a bipartite graph having  $n$  vertices?  
.....  
.....

5. How many unique colours will be required for proper vertex colouring of a line graph having  $n$  vertices?

.....

6. How many unique colours will be required for proper vertex colouring of a complete graph having  $n$  vertices?

.....

7. What is a chromatic index?

.....

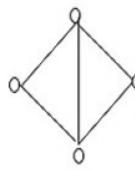
8. What will be the chromatic number of the following graph?



9. How many unique colours will be required for vertex colouring of the following graph?



10. How many unique colours will be required for vertex colouring of the following graph?



## 10.9 Unit End Questions (MCQ and Descriptive)

### A. Descriptive Types Questions

1. How many possible solutions occur for a 10-queen problem?
2. How many possible solutions exist for an 8-queen problem?
3. What is vertex colouring of a graph?
4. How many edges will a tree consisting of N nodes have?
5. How many unique colours will be required for proper vertex colouring of an empty graph having n vertices?
6. Who formulated the first ever algorithm for solving the Hamiltonian path problem?
7. In what time can the Hamiltonian path problem be solved using dynamic programming?
8. What is the time complexity for finding a Hamiltonian path for a graph having n vertices (using permutation)?

### B. Multiple Choice/Objective Type Questions

1. In how many directions do queens attack each other?
  - (a) 1
  - (b) 2
  - (c) 3
  - (d) 4
2. Placing n-queens so that no two queens attack each other is called:
  - (a) N-queens problem
  - (b) 8-queens problem
  - (c) Hamiltonian circuit problem
  - (d) Subset sum problem
3. Where is the n-queens problem implemented?
  - (a) Carom
  - (b) Chess
  - (c) Ludo
  - (d) Cards
4. Not more than 2 queens can occur in n-queens problem.
  - (a) True
  - (b) False

5. In n-queens problem, how many values of n does not provide an optimal solution?
  - (a) 1
  - (b) 2
  - (c) 3
  - (d) 4
6. Which of the following methods can be used to solve n-queens problem?
  - (a) Greedy algorithm
  - (b) Divide and conquer
  - (c) Iterative improvement
  - (d) Backtracking
7. Of the following given options, which one of the following is a correct option that provides an optimal solution for 4-queens problem?
  - (a) (3,1,4,2)
  - (b) (2,3,1,4)
  - (c) (4,3,2,1)
  - (d) (4,2,3,1)
8. Under what condition any set A will be a subset of B?
  - (a) If all elements of set B are also present in set A
  - (b) If all elements of set A are also present in set B
  - (c) If A contains more elements than B
  - (d) If B contains more elements than A
9. What is a subset sum problem?
  - (a) Finding a subset of a set that has sum of elements equal to a given number
  - (b) Checking for the presence of a subset that has sum of elements equal to a given number and printing true or false based on the result
  - (c) Finding the sum of elements present in a set
  - (d) Finding the sum of all the subsets of a set
10. Which of the following is true about the time complexity of the recursive solution of the subset sum problem?
  - (a) It has an exponential time complexity
  - (b) It has a linear time complexity
  - (c) It has a logarithmic time complexity
  - (d) It has a time complexity of  $O(n^2)$

11. What is the worst-case time complexity of dynamic programming solution of the subset sum problem (sum=given subset sum)?
- (a) O(n)
  - (b) O(sum)
  - (c) O(n<sup>2</sup>)
  - (d) O(sum\*n)
12. Subset sum problem is an example of NP complete problem.
- (a) True
  - (b) False
13. Recursive solution of subset sum problem is faster than dynamic problem solution in terms of time complexity.
- (a) True
  - (b) False
14. Which of the following is not true about subset sum problem?
- (a) The recursive solution has a time complexity of O(2n)
  - (b) There is no known solution that takes polynomial time
  - (c) The recursive solution is slower than dynamic programming solution
  - (d) The dynamic programming solution has a time complexity of O(n log n)

**Answers:**

1. (c), 2. (a), 3. (b), 4. (b), 5. (b), 6. (d), 7. (a), 8. (b), 9. (b), 10. (a), 11. (d), 12. (a), 13. (b), 14. (d)

---

## 10.10 References

References of this unit have been given at the end of the book.

---

## **UNIT 11 BACKTRACKING 2**

---

### **Structure:**

- 11.0 Learning Objectives
- 11.1 Introduction
- 11.2 Branch and Bound
- 11.3 Assignment Problem
- 11.4 Travelling Salesperson Problem
- 11.5 Summary
- 11.6 Key Words/Abbreviations
- 11.7 Learning Activity
- 11.8 Unit End Questions (MCQ and Descriptive)
- 11.9 References

---

### **11.0 Learning Objectives**

---

After studying this unit, you will be able to:

- Describe branch and bound concepts
- Explain assignment problems
- Gain confidence to solve job assignment problem using branch and bound
- Implementation of 0/1 knapsack using branch and bound



- Ensure travelling salesperson problem
- Gain confidence by providing the complete algorithm

## **11.1 Introduction**

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimisation problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and bound solve these problems relatively quickly.

Let us consider below 0/1 knapsack problem to understand branch and bound.

Given two integer arrays  $val[0..n-1]$  and  $wt[0..n-1]$  that represent values and weights associated with  $n$  items, respectively. Find out the maximum value subset of  $val[]$  such that sum of the weights of this subset is smaller than or equal to knapsack capacity  $W$ .

### **Let us explore all approaches for this problem.**

1. A greedy approach is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.
2. We can use dynamic programming (DP) for 0/1 knapsack problem. In DP, we use a 2D table of size  $n \times W$ . The DP solution doesn't work if item weights are not integers.
3. Since DP solution doesn't always work, a solution is to use brute force. With  $n$  items, there are  $2n$  solutions to be generated, check each to see if they satisfy the constraint, save maximum solutions that satisfy constraint. This solution can be expressed as tree.

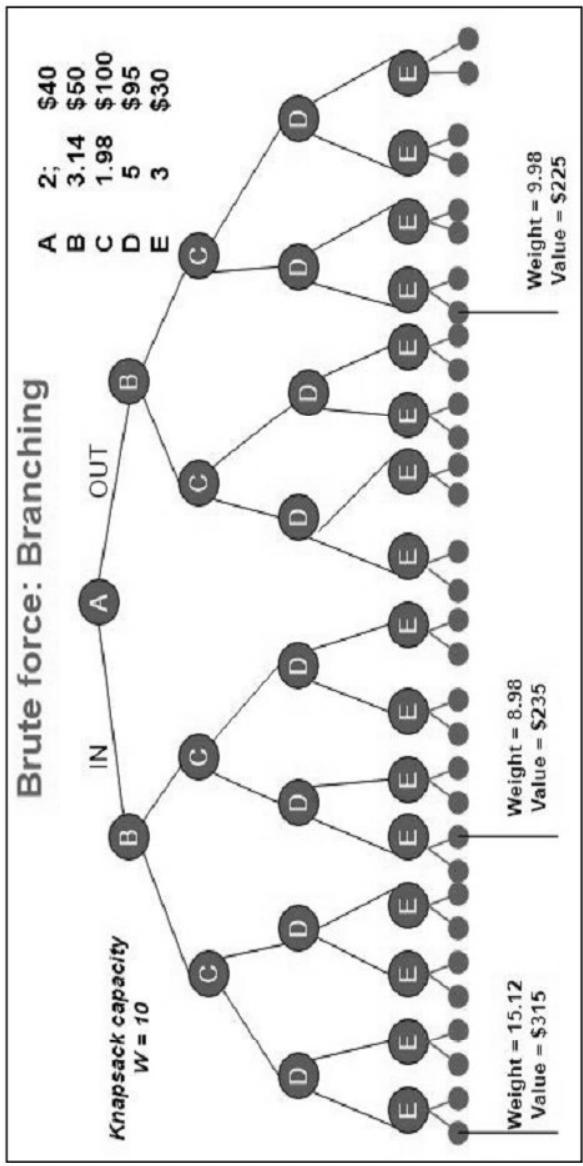


Fig. 11.1: Brute Force- Branching

4. We can use backtracking to optimise the brute force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution is no longer feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.

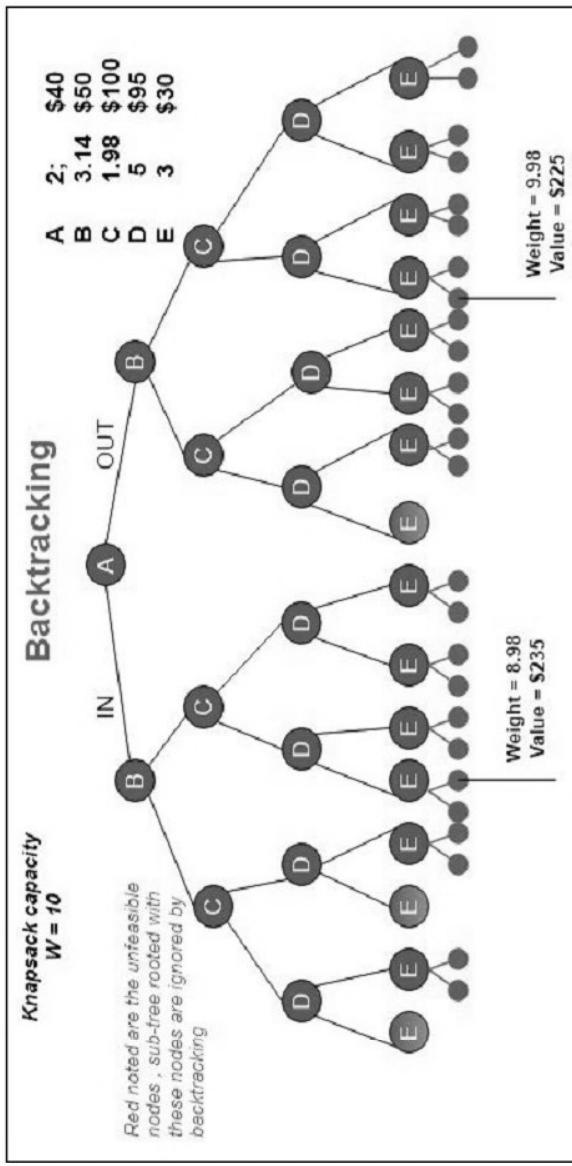


Fig. 11.2: Backtracking

The backtracking based solution works better than brute force by ignoring infeasible solutions. We can do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So, we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

**Example:** Bounds used in below diagram are, A down can give 315, B down can 315, B down can 275, C down can 225, D down can 225, and E down can \$30. In the next section, we have discussed the process to get these bounds.

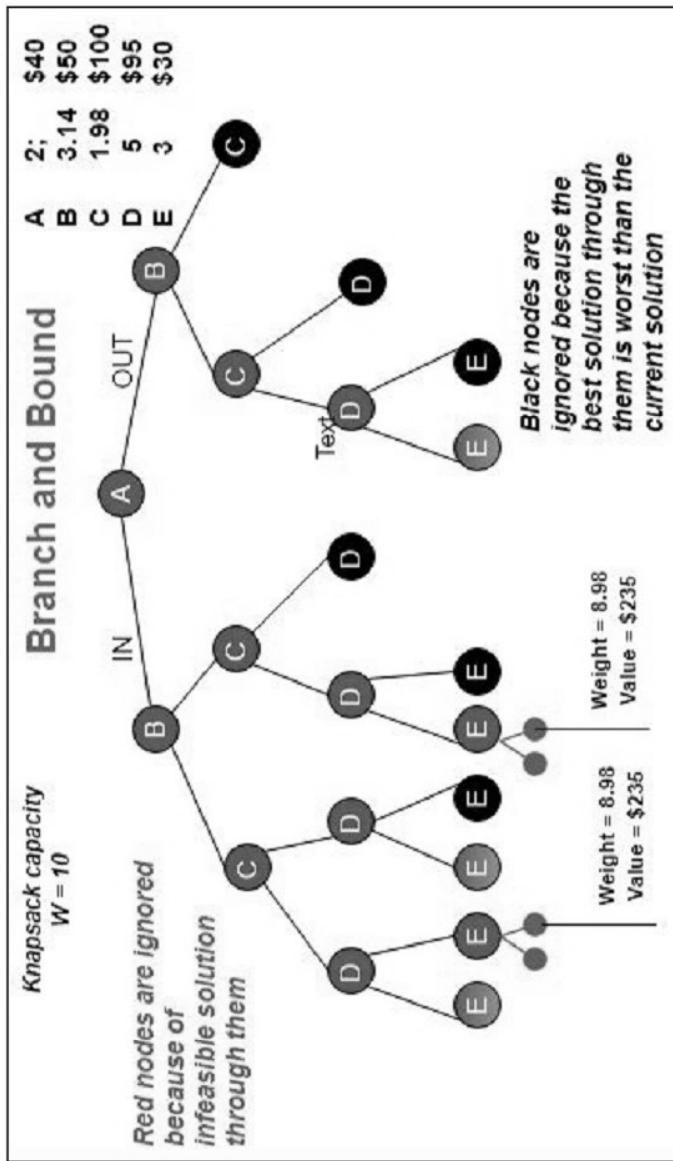


Fig. 11.3: Branch and Bound

Branch and bound is a very useful technique for searching a solution, but in worst case we need to fully calculate the entire tree. At best, we only need to fully calculate one path through the tree and prune the rest of it.

### Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In

the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

- (1) Start in the leftmost column.
- (2) If all queens are placed, return true.
- (3) Try all rows in the current column. Do the following for every tried row:
  - (a) If the queen can be placed safely in this row, then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
    - (b) If placing the queen in [row, column] leads to a solution then return true.
    - (c) If placing queen doesn't lead to a solution, then unmark this [row, column] (backtrack) and go to step (a) to try other rows.
  - (3) If all rows have been tried and nothing worked, return false to trigger backtracking.

---

## 11.2 Branch and Bound

Branch and bound is a general technique for improving the searching process by systematically enumerating all candidate solutions and disposing off obviously impossible solutions.

Branch and bound usually applies to those problems that have finite solutions, in which the solutions can be represented as a sequence of options. The first part of branch and bound branching requires several choices to be made so that the choices will branch out into the solution space. In these methods, the solution space is organised as a treelike structure. Branching out to all possible choices guarantees that no potential solutions will be left uncovered. But, because the target problem is usually NP complete or even NP hard, the solution space is often too vast to traverse.

An important advantage of branch and bound algorithms is that we can control the quality of the solution to be expected, even if it is not yet found.

### 11.3 Assignment Problem

#### Job Assignment Problem using Branch and Bound

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimised.

**Table 11.1: Job Assignment Problem using Branch and Bound**

|   | Job 1 | Job 2 | Job 3 | Job 4 | Worker A takes 8 units of time to finish job 4. |
|---|-------|-------|-------|-------|-------------------------------------------------|
| A | 9     | 2     | 7     | 8     |                                                 |
| B | 6     | 4     | 3     | 7     |                                                 |
| C | 5     | 8     | 1     | 8     |                                                 |
| D | 7     | 6     | 9     | 4     |                                                 |

An example job assignment problem. Green values show optimal job assignment that is A-Job4, B-Job1 C-Job3 and D-Job1

Let us explore all approaches for this problem.

#### Solution 1: Brute Force

We generate  $n!$  possible job assignments and for each such assignment, we compute its total cost and return the less expensive assignment. Since the solution is a permutation of the n jobs, its complexity is  $O(n!)$ .

#### Solution 2: Hungarian Algorithm

The optimal assignment can be found using the Hungarian algorithm. The Hungarian algorithm has worst-case runtime complexity of  $O(n^3)$ .

**Solution 3: DFS/BFS on State Space Tree**

A state space tree is a N-ary tree with property that any path from root to leaf node holds one of many solutions to given problem. We can perform depth-first search on state space tree, but successive moves can take us away from the goal rather than bringing us closer. The search of state space tree follows leftmost path from the root regardless of initial state. An answer node may never be found in this approach. We can also perform a breadth-first search on state space tree. But, no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

**Solution 4: Finding Optimal Solution using Branch and Bound**

The selection rule for the next node in BFS and DFS is ‘blind’ i.e. the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. The search for an optimal solution can often be speeded by using an ‘intelligent’ ranking function, also called an approximate cost function to avoid searching in subtrees that do not contain an optimal solution. It is similar to BFS like search but with one major optimisation. Instead of following FIFO order, we choose a live node with least cost. We may not get optimal solution by following node with least promising cost, but it will provide very good chance of getting the search to an answer node quickly.

There are two approaches to calculate the cost function:

1. For each worker, we choose job with minimum cost from a list of unassigned jobs (take minimum entry from each row).
2. For each job, we choose a worker with lowest cost for that job from a list of unassigned workers (take minimum entry from each column).

In this article, the first approach is followed.

Let’s take below example and try to calculate promising cost when Job 2 is assigned to worker A.

**Table 11.2: Example**

|          | <b>Job 1</b> | <b>Job 2</b> | <b>Job 3</b> | <b>Job 4</b> |
|----------|--------------|--------------|--------------|--------------|
| <b>A</b> | 9            | 2            | 7            | 8            |
| <b>B</b> | 6            | 4            | 3            | 7            |
| <b>C</b> | 5            | 8            | 1            | 8            |
| <b>D</b> | 7            | 6            | 9            | 4            |

Since Job 2 is assigned to worker A (marked in green), cost becomes 2 and Job 2 and worker A becomes unavailable (marked in red).

**Table 11.3: Assign Job 2 to Worker**

|          | <b>Job 1</b> | <b>Job 2</b> | <b>Job 3</b> | <b>Job 4</b> |
|----------|--------------|--------------|--------------|--------------|
| <b>A</b> | 9            | 2            | 7            | 8            |
| <b>B</b> | 6            | 4            | 3            | 7            |
| <b>C</b> | 5            | 8            | 1            | 8            |
| <b>D</b> | 7            | 6            | 9            | 4            |

Now, we assign Job 3 to worker B as it has minimum cost from the list of unassigned jobs. Cost becomes  $2 + 3 = 5$  and Job 3 and worker B also becomes unavailable.

**Table 11.4: Assign Job 3 to Worker**

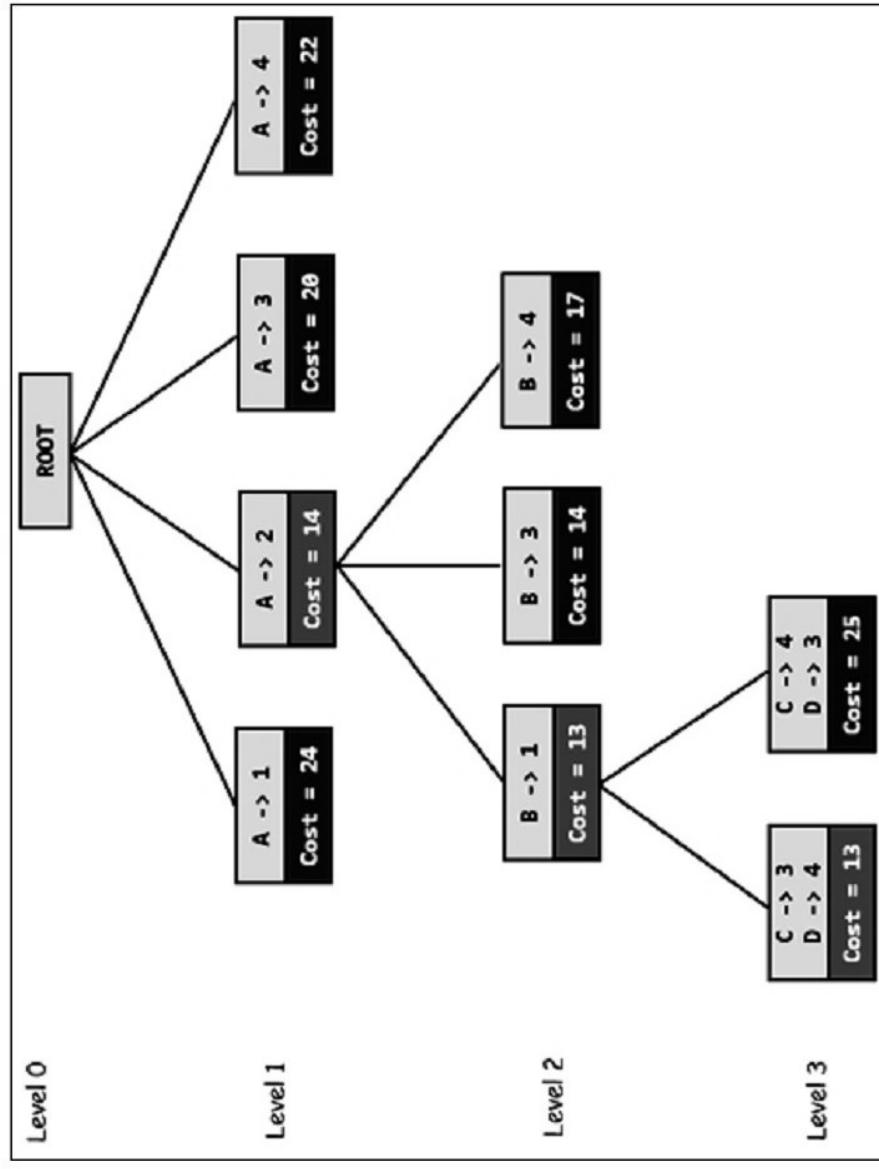
|          | <b>Job 1</b> | <b>Job 2</b> | <b>Job 3</b> | <b>Job 4</b> |
|----------|--------------|--------------|--------------|--------------|
| <b>A</b> | 9            | 2            | 7            | 8            |
| <b>B</b> | 6            | 4            | 3            | 7            |
| <b>C</b> | 5            | 8            | 1            | 8            |
| <b>D</b> | 7            | 6            | 9            | 4            |

Finally, Job 1 gets assigned to worker C as it has minimum cost among the unassigned jobs and Job 4 gets assigned to worker C as it is the only job left. Total cost becomes  $2 + 3 + 5 + 4 = 14$ .

**Table 11.5: Job 1 Gets Assigned to Worker C**

|   | Job 1 | Job 2 | Job 3 | Job 4 |
|---|-------|-------|-------|-------|
| A | 9     | 2     | 7     | 8     |
| B | 6     | 4     | 3     | 7     |
| C | 5     | 8     | 1     | 8     |
| D | 7     | 6     | 9     | 4     |

Below diagram shows complete search space diagram showing optimal solution path in green.

**Fig. 11.4: Complete Search Space Diagram**

### Complete Algorithm:

```

/* findMinCost uses Least() and Add() to maintain the
list of live nodes

Least() finds a live node with least cost, deletes
it from the list and returns it

Add(x) calculates cost of x and adds it to the list
of live nodes

Implements list of live nodes as a min heap */

// Search Space Tree Node

node
{
 int job_number;
 int worker_number;
 node parent;
 int cost;
}

// Input: Cost Matrix of Job Assignment problem
// Output: Optimal cost and Assignment of Jobs
algorithm findMinCost (costMatrix mat[][])
{
 // Initialize list of live nodes(min-heap)
 // with root of search tree, i.e., a dummy node
 while (true)

 {
 // Find a live node with least estimated cost
 E = Least();

 // The found node is deleted from the list
 // of live nodes
 if (E is a leaf node)

 {
 printSolution();
 return;
 }
 }
}

```

```

 }

 for each child x of E

 {

 Add(x); // Add x to list of live nodes;

 x->parent = E; // Pointer for path to root
 }
}

```

## 11.4 Travelling Salesperson Problem

## Travelling Salesman Problem using Branch and Bound

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

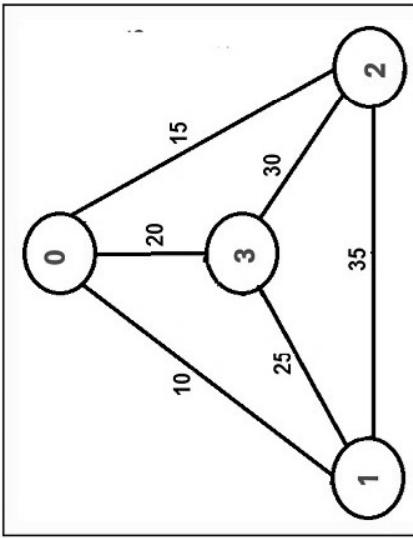


Fig. 11.5: Set of Cities

For example, consider the graph shown in figure above. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is  $|0+25+30+15|$  which is 80.

We have discussed the following solutions:

- 1) Naive and dynamic programming
  - 2) Approximate solution using MST

## Branch and Bound Solution

As seen in the previous articles, in branch and bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Note that the cost through a node includes two costs.

- 1) Cost of reaching the node from the root. (When we reach a node, we have this cost computed.)
  - 2) Cost of reaching an answer from current node to a leaf. (We compute a bound on this cost to decide whether to ignore subtree with this node or not).
- In cases of a **maximisation problem**, an upper bound tells us the maximum possible solution if we follow the given node. For example in 0/1 knapsack we used Greedy approach to find an upper bound.
  - In cases of a **minimisation problem**, a lower bound tells us the minimum possible solution if we follow the given node. For example, in job assignment problem, we get a lower bound by assigning least cost job to a worker.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for travelling salesman problem.

Cost of any tour can be written as below.

Cost of a tour  $T = (1/2) * \text{Sum}(\text{Sum of cost of two edges adjacent to } u \text{ and in the tour } T)$

where  $u \in V$

For every vertex  $u$ , if we consider two edges through it in  $T$ , and sum their costs. The overall sum for all vertices would be twice of cost of tour  $T$  (We have considered every edge twice.)

(Sum of two tour edges adjacent to  $u$ )  $\geq$  (sum of minimum weight two edges adjacent to  $u$ )



|                                                                                              |
|----------------------------------------------------------------------------------------------|
| Cost of any tour $\geq 1/2 * \&Sum;$ (Sum of cost of two minimum weight edges adjacent to u) |
| where $u \in V$                                                                              |

For example, consider the above shown graph. Below are minimum cost two edges adjacent to every node.

| Node | Least cost edges | Total cost |
|------|------------------|------------|
| 0    | (0, 1), (0, 2)   | 25         |
| 1    | (0, 1), (1, 3)   | 35         |
| 2    | (0, 2), (2, 3)   | 45         |
| 3    | (0, 3), (1, 3)   | 45         |

Thus, a lower bound on the cost of any tour =  
 $1/2(25 + 35 + 45 + 45)$   
 $= 75$

Refer this for one more example.

Now we have an idea about computation of lower bound. Let us see how to apply it state space search tree. We start enumerating all possible nodes (preferably in lexicographical order).

**1. The Root Node:** Without loss of generality, we assume we start at vertex '0' for which the lower bound has been calculated above.

**Dealing with Level 2:** The next level enumerates all possible vertices we can go to (keeping in mind that in any path a vertex has to occur only once) which are, 1, 2, 3... n (note that the graph is complete). Consider we are calculating for vertex 1. Since we moved from 0 to 1, our tour has now included the edge 0-1. This allows us to make necessary changes in the lower bound of the root.

|                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------|
| Lower Bound for vertex 1 =<br>$Old\ lower\ bound - ((minimum\ edge\ cost\ of\ 0 +$<br>$minimum\ edge\ cost\ of\ 1)/2) + (edge\ cost\ 0-1)$ |
|--------------------------------------------------------------------------------------------------------------------------------------------|

How does it work? To include edge 0-1, we add the edge cost of 0-1, and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. Clearly, the edge subtracted can't be smaller than this.

**Dealing with Other Levels:** As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, ...n.

Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and alter the new lower bound for this node.

$$\begin{aligned} \text{Lower bound}(2) = & \\ & \text{Old lower bound} - (\text{second minimum edge cost of 1} + \\ & \text{minimum edge cost of 2})/2 \\ & + \text{edge cost 1-2} \end{aligned}$$

**Note:** The only change in the formula is that this time we have included second minimum edge cost for 1, because the minimum edge cost has already been subtracted in the previous level.

## 11.5 Summary

Branch and bound usually applies to those problems that have finite solutions, in which the solutions can be represented as a sequence of options. The first part of branch and bound branching requires several choices to be made, so that the choices will branch out into the solution space. An important advantage of branch and bound algorithms is that we can control the quality of the solution to be expected, even if it is not yet found.

The travelling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and return to the same city. The challenge of the problem is that the travelling salesman needs to minimise the total length of the trip.

Suppose the cities are  $x_1, x_2, \dots, x_n$  where cost ( $c_{ij}$ ) denotes the cost of travelling from city  $x_i$  to  $x_j$ . The travelling salesperson problem is to find a route starting and ending at  $x_1$  that will take in all cities with the minimum cost.

## Knapsack Problem

Knapsack is basically means a bag. A bag of given capacity.

We want to pack n items in your luggage.

The ith item is worth  $v_i$  dollars and weight  $w_i$  pounds.

Take as valuable a load as possible, but cannot exceed W pounds.

$v_i$   $w_i$  W are integers.

$W \leq$  capacity

Value  $\leftarrow$  Max

**Knapsack problem can be further divided into two parts:**

1. **Fractional Knapsack:** Fractional knapsack problem can be solved by **greedy strategy** whereas 0 / 1 problem is not.

It cannot be solved by **dynamic programming approach**.

2. **0/1 Knapsack Problem:** In this, item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the greedy approach because it is unable to fill the knapsack to capacity.
- **Greedy approach** doesn't ensure an optimal solution.

## 11.6 Key Words/Abbreviations

- Branch and bound (BB, B&B, or BnB)
- **FIFO** (first in, first out)
- **LIFO** (last in, first out)
- **LC** (lowest cost)
- Greedy algorithm for fractional knapsack

- DP solution for 0/1 knapsack
- Backtracking solution for 0/1 knapsack

## 11.7 Learning Activity

1. What is the assignment problem?

---

---

2. Give mathematical form of assignment problem.

---

---

3. What is the difference between assignment problem and transportation problem?

---

---

4. Three jobs A, B and C are to be assigned to three machines U, V and W. The processing cost for each job-machine combination is shown in the matrix given below. Determine the allocation that minimises the overall processing cost.

|     |   | Machine |    |    |
|-----|---|---------|----|----|
|     |   | U       | V  | W  |
| Job | A | 17      | 25 | 31 |
|     | B | 10      | 25 | 16 |
|     | C | 12      | 14 | 11 |

(cost is in ₹ per unit)

5. A computer centre has got three expert programmers. The centre needs three application programmes to be developed. The head of the computer centre, after studying carefully the programmes to be developed, estimates the computer time in minutes required by the experts to the application programme as follows.

|             |   | Programmes |     |     |
|-------------|---|------------|-----|-----|
|             |   | P          | Q   | R   |
| Programmers | 1 | 120        | 100 | 80  |
|             | 2 | 80         | 90  | 110 |
|             | 3 | 110        | 140 | 120 |

Assign the programmers to the programme in such a way that the total computer time is least.

---

---

6. A departmental head has four subordinates and four tasks to be performed. The subordinates differ in efficiency and the tasks differ in their intrinsic difficulty. His estimates of the time each man would take to perform each task is given below.

|              |   | Tasks |    |    |    |
|--------------|---|-------|----|----|----|
|              |   | 1     | 2  | 3  | 4  |
| Subordinates | P | 8     | 26 | 17 | 11 |
|              | Q | 13    | 28 | 4  | 26 |
|              | R | 38    | 19 | 18 | 15 |
|              | S | 9     | 26 | 24 | 10 |

How should the tasks be allocated to subordinates so as to minimise the total man-hours?

---

---

7. Find the optimal solution for the assignment problem with the following cost matrix:

|          |   | Area |    |    |    |
|----------|---|------|----|----|----|
|          |   | 1    | 2  | 3  | 4  |
| Salesman | P | 11   | 17 | 8  | 16 |
|          | Q | 9    | 7  | 12 | 6  |
|          | R | 13   | 16 | 15 | 12 |
|          | S | 14   | 10 | 12 | 11 |

8. Assign four trucks 1, 2, 3 and 4 to vacant spaces A, B, C, D, E and F so that distance travelled is minimised. The matrix below shows the distance.

|       |   | 1 | 2 | 3 | 4 |
|-------|---|---|---|---|---|
| Truck | A | 4 | 7 | 3 | 7 |
|       | B | 8 | 2 | 5 | 5 |
|       | C | 4 | 9 | 6 | 9 |
|       | D | 7 | 5 | 4 | 8 |
|       | E | 6 | 3 | 5 | 4 |
|       | F | 6 | 8 | 7 | 3 |

## 11.8 Unit End Questions (MCQ and Descriptive)

### A. Descriptive Types Questions

1. How to solve assignment problem with group restrictions?
2. How to choose variables in case of more than one fractional values?
3. How can we use branch and bound method in integer linear programming problem for three or more variables?
4. Which algorithm is fastest in finding the exact solution set of subset sum problem?
5. What exactly do you mean by a partial solution in branch and bound terminology?
6. How can we solve the 8 puzzle problem using branch and bound approach?

### B. Multiple Choice/Objective Type Questions

1. Branch and bound is a \_\_\_\_\_.  
  - (a) Problem-solving technique
  - (b) Data structure
  - (c) Sorting algorithm
  - (d) Type of tree
2. Which data structure is used for implementing a LIFO branch and bound strategy?  
  - (a) Stack
  - (b) Queue
  - (c) Array
  - (d) Linked list
3. Which data structure is used for implementing a FIFO branch and bound strategy?  
  - (a) Stack
  - (b) Queue
  - (c) Array
  - (d) Linked list
4. Which data structure is most suitable for implementing best first branch and bound strategy?  
  - (a) Stack
  - (b) Queue
  - (c) Priority queue
  - (d) Linked list
5. Both FIFO branch and bound strategy and backtracking leads to depth-first search.  
  - (a) True
  - (b) False

6. Choose the correct statement from the following:
  - (a) Branch and bound is more efficient than backtracking
  - (b) Branch and bound is not suitable where a greedy algorithm is not applicable
  - (c) Branch and bound divides a problem into at least two new restricted subproblems
  - (d) Backtracking divides a problem into at least two new restricted sub problems
7. Which of the following can traverse the state space tree only in DFS manner?
  - (a) Branch and bound
  - (b) Dynamic programming
  - (c) Greedy algorithm
  - (d) Backtracking

**Answers:**

1. (a), 2. (a), 3. (b) 4. (c), 5. (b), 6. (c), 7. (d)

---

**11.9 References**

---

References of this unit have been given at the end of the book.

---

# PRACTICAL      DAA BASICS

## UNIT 1

---

### Structure:

#### 1.0 Learning Objectives

- Practical 1.1 Determine whether a particular element is available in a list of elements entered by the user at runtime.
- Practical 1.2 Obtain the topological ordering of vertices in a given digraph.
- 1.3 References

---

### 1.0 Learning Objectives

---

After studying this unit, you will be able to:

- Develop programs for basic DAA concepts
- Define algorithm specifications
- Implement a one-dimensional array; take number from users to search for in the array using binary search.
- Illustrate C program using recursion to search an element in array
- Write a C program to implement linear search algorithm.
- Obtain the topological ordering of vertices in a given digraph

---

**Practical 1.1:** Determine whether a particular element is available in a list of elements entered by the user at runtime

---

### Problem Description

This program will implement a one-dimentional array, take number from users to search for in the array using binary search.

### Problem Solution

1. Create an array of some certain size and define its elements in a sorted fashion.
2. Now take an input from the users which you want to search for.
3. Take two variables pointing to the first and last index of the array (namely low and high).
4. Run start a while and run it until low equals high.
5. Now, take a mid of low and high value, check whether value at mid equals the user input.
6. In case it matches the user input, it means we found the number, after which we must break out from the loop.
7. And if user input is greater than value at mid, then low is assigned the value of mid. Similarly, if user input is smaller than value at mid, then high is assigned the value of mid.
8. In this way, the region of finding the user input becomes half.

### Program/Source Code

Here is a source code of the C program to read an array and search for an element. The program is successfully compiled and tested using Turbo C compiler in windows environment. The program output is also shown below.

```
1.
2. /*
3. * C program accept an array of N elements and a key to search.
4. * If the search is successful, it displays "SUCCESSFUL SEARCH".
5. * Otherwise, a message "UNSUCCESSFUL SEARCH" is displayed.
6. */
```

```
7.
8. #include <stdio.h>
9. void main()
10. {
11.
12. int array[20];
13. int i, low, mid, high, key, size;
14.
15. printf("Enter the size of an array\n");
16. scanf("%d", &size);
17.
18. printf("Enter the array elements\n");
19. for (i = 0; i < size; i++)
20. {
21. scanf("%d", &array[i]);
22. }
23.
24. printf("Enter the key\n");
25. scanf("%d", &key);
26.
27. /* search begins */
28.
29. low = 0;
30. high = (size - 1);
31.
32. while (low <= high)
33. {
34. mid = (low + high) / 2;
35.
36. if (key == array[mid])
37. {
38. printf("SUCCESSFUL SEARCH\n");
```

```
39. return;
40. }
41.
42. if(key < array[mid])
43. high = mid - 1;
44.
45. else
46. low = mid + 1;
47.
48. }
49.
50. printf("UNSUCCESSFUL SEARCH\n");
51.
52. }
```

### Program Explanation

1. Declare an array of capacity 20, taking size from users, define all the elements of the array but in sorted fashion.
2. Now, take three variables, low pointing to the first index of array, i.e., 0, last index of array, i.e., size-1, and mid.
3. Run a loop till the low become equal to high.
4. Inside this loop, first calculate mid using  $(\text{high} + \text{low}) / 2 = \text{mid}$ .
5. Check if the value at mid matches the user input, if it does, that means we found the element and now we can return by breaking out from the loop.
6. If user input is greater than value at mid, then low is shifted to mid; similarly, if user input is smaller than mid, then high is shifted to mid.
7. In this way, each time, we halved the region where we are finding the element.

### Runtime Test Cases

Enter the size of an array

4

Enter the array elements

90

560

300

390

Enter the key

90

SUCCESSFUL SEARCH

\$ a.out

Enter the size of an array

4

Enter the array elements

100

500

580

470

Enter the key

300

UNSUCCESSFUL SEARCH

---

### Practical 1.2: Obtain the Topological Ordering of Vertices in a Given Digraph

---

#### Problem Description

Topological sort is an ordering of the vertices in a directed acyclic graph, such that, if there is a path from u to v, then v appears after u in the ordering.

The graphs should be directed, otherwise for any edge (u,v) there would be a path from u to v, and also from v to u, and hence they cannot be ordered.

The graphs should be acyclic, otherwise for any two vertices u and v on a cycle u would precede v and v would precede u.



### Problem Solution

#### Using Source Removal Algorithm:

1. Compute the indegrees of all vertices.
2. Find a vertex U with indegree 0 and print it (store it in the ordering).
3. If there is no such vertex, then there is a cycle and the vertices cannot be ordered. Stop.
4. Remove U and all its edges (U,V) from the graph.
5. Update the indegrees of the remaining vertices.
6. Repeat steps 2 through 4 while there are vertices to be processed.

#### Solution:

```
#include<stdio.h>
int temp[10],k=0;

void topo(int n,int indegree[10],int a[10][10])
{
 int i,j;
 for(i=1;i<=n;j++)
 {
 if(indegree[i]==0)
 {
 indegree[i]=1;
 temp[++k]=i;
 for(j=1;j<=n;j++)
 {
 if(a[i][j]==1&&indegree[j]!=-1)
 indegree[j]--;
 }
 i=0;
 }
 }
}
```

```
}

void main0
{
 int i,j,n,indegree[10],a[10][10];
 printf("enter the number of vertices:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 indegree[i]=0;

 printf("\n enter the adjacency matrix\n");
 for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 {
 scanf("%d",&a[i][j]);
 if(a[i][j]==1)
 indegree[j]++;
 }

 topo(n,indegree,a);

 if(k!=n)
 printf("topological ordering is not possible\n");

 else
 {
 printf("\n topological ordering is :\n");
 for(i=1;i<=k;i++)
 printf("%d",temp[i]);
 }
}
```

**Using DFS Algorithm:**

1. Run DFS(G), computing finish time for each vertex.
2. As each vertex is finished, insert it onto the front of a list.

**Solution:**

```
#include<stdio.h>
int i,visit[20],n,adj[20][20],s,topo_order[10];

void dfs(int v)
{
 int w;
 visit[v]=1;

 for(w=1;w<=n;w++)
 if((adj[v][w]==1) && (visit[w]==0))
 dfs(w);

 topo_order[i--]=v;
}

void main()
{
 int v,w;
 printf("Enter the number of vertices:\n");
 scanf("%d",&n);
 printf("Enter the adjacency matrix:\n");
 for(v=1,v<=n;v++)
 for(w=1,w<=n,w++)
 scanf("%d",&adj[v][w]);
 for(v=1,v<=n,v++)
 visit[v]=0;
 i=n;
 for(v=1,v<=n,v++)
 {
 if(visit[v]==0)
```

```
 dfs(v);
 }
 printf("\nTopological sorting is:");
 for(v=1;v<=n;v++)
 printf(" %d",topo_order[v]);
}
```

**OUTPUT 1:**

Enter the number of vertices: 5

Enter the adjacency matrix

```
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
```

Topological ordering is v1 v2 v3 v4 v5

**OUTPUT 2:**

Enter the number of vertices: 3

Enter the adjacency matrix

```
0 1 0
0 0 1
1 0 0
```

Topological ordering is not possible.

---

**1.3 References**

References of this unit have been given at the end of the book.

---

## PRACTICAL      DIVIDE AND CONQUER AND UNIT 2        GREEDY METHOD

---

### Structure:

#### 2.0 Learning Objectives

- Practical 2.1 Sort a given set of elements using the Quick Sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
- Practical 2.2 Using OpenMP, implement a parallelised Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.
- Practical 2.3 Find Minimum Cost Spanning Tree of a given undirected graph using Prim's Algorithm
- 2.4 References

---

### 2.0 Learning Objectives

After studying this unit, you will be able to:

- Develop programs using divide and conquer techniques
- Create programs using greedy methods



- Develop programs for quick sort method and determine the time required to sort the elements
- Define algorithm specifications
- Implement merge sort algorithm to sort a given set of elements and determine the time required to sort the elements.
- Find minimum cost spanning tree of a given undirected graph using Prim's algorithm.

**Practical 2.1:** Sort a given set of elements using the Quick Sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Solution:**

```
include <stdio.h>
include <conio.h>
include <time.h>
voidExch(int * p, int * q)
{
 int temp = * p;
 * p = * q;
 * q = temp;
}

voidQuickSort(int a[], int low, int high)
{
 int i, j, key, k;
 if((low>=high))
 return;
 key=low; i=low+1; j=high;
 while(i<=j)
```

```

{
 while (a[i] <= a[key]) i=i+1;
 while (a[j] > a[key]) j=j-1;
 if(i<j) Exch(&a[i], &a[j]);
}

Exch(&a[i], &a[key]);

QuickSort(a, low, j-1);
QuickSort(a, j+1, high);

void main()
{
 int n, a[1000],k
 clock_t st,et;
 doublets;
 clrscr();
 printf("\n Enter How many Numbers: ");
 scanf("%d", &n);
 printf("\n The Random Numbers are:\n");
 for(k=1; k<=n; k++)
 {
 a[k]=rand();
 printf("%d\t",a[k]);
 }
 st=clock();
 QuickSort(a, 1, n);
 et=clock();
 ts=(double)(et-st)/CLOCKS_PER_SEC;
 printf("\nSorted Numbers are: \n");
 for(k=1; k<=n; k++)
 printf("%d\t", a[k]);
 printf("\nThe time taken is %e",ts);
 getch();
}

```

**Output:**

```

Enter How many Numbers : 90
The Random Numbers are :
346 130 16982 1090 11656 7117 17595 6415 22948 31126
9094 14558 3571 22879 18492 31214 27569 5412 26721 22463 25047
27119 31441 7190 13985 3734 13310 3979 21995 14779 19816
21681 19651 17995 23593 5892 13863 22766 5364 17639 21151
18489 11288 28466 8664 8812 15108 12666 12347 19042 19774
26427 160 25795 10941 13390 7878 13565 11636 13268 6460 9169
5589 26383 9666 10941 8333 31937 1779 16190 32233
53 13429 2285 2422 29142 29667 24115 15116 17418 1156 4279

Sorted Numbers are :
53 160 130 346 1090 1156 1360 1779 2285 2422
3571 3734 3979 4279 5364 5412 5589 5892 6415 6458
6460 6936 7117 7190 7878 8160 8333 8664 8812 9004
9169 9666 10941 1288 11636 11636 11636 12347 12666 13268
13310 13390 13429 13565 13863 13985 14558 14779 15168 15116
15561 16692 16190 17418 17595 17639 17995 18489 18492 19042
19651 19774 19816 20427 21151 21681 21995 22463 22766 22879
22948 23593 24115 24842 25047 25795 26383 26571 26721 27119
27569 28466 29142 29667 30252 31126 31214 31441 31937 32233

The time taken is 0.000000e+00

```

**Practical 2.2:** Using OpenMP, implement a parallelised Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Solution:**

```

#include <stdio.h>
#include <conio.h>
#include <time.h>
void Merge(int a[], int low, int mid, int high)
{
 int i, j, k, b[20];
 i=low; j=mid+1; k=low;

```



```

while (i<=mid && j<=high)
{
 if(a[i] <= a[j])
 b[k++] = a[i++];
 else
 b[k++] = a[j++];

 while (i<=mid) b[k++]=a[i++];
 while (j<=high) b[k++]=a[j++];
 for(k=low; k<=high; k++)
 a[k] = b[k];
}

void MergeSort(int a[], int low, int high)
{
 int mid;
 if(low >= high)
 return;
 mid = (low+high)/2 ;
 MergeSort(a, low, mid);
 MergeSort(a, mid+1, high);
 Merge(a, low, mid, high);
}

void main()
{
 int n, a[2000],k;
 clock_tst,et;
 doublets;
 clrscr();
 printf("\n Enter How many Numbers:");
 scanf("%d", &n);
 printf("\nThe Random Numbers are:\n");
 for(k=1; k<=n; k++)
 {
}

```

```
a[k]=rand();
printf("%d", a[k]);
}

st=clock();
MergeSort(a, 1, n);
et=clock();
ts=(double)(et-st)/CLOCKS_PER_SEC;
printf("\n Sorted Numbers are : \n ");
for(k=1; k<=n; k++)
printf("%d", a[k]);
printf("\nThe time taken is %e",ts);
getch();
}
```

**Output:**

```
Enter How many Numbers :15
The Random Numbers are:
346 130 10982 1090 11656 7117 17595 6415 22948 31126
9004 14558 3571 22879 18492
Sorted Numbers are :
130 346 1090 3571 6415 7117 9004 10982 11656 14558
17595 18492 22879 22948 31126
The time taken is 0.000000e+00_
```

**Practical 2.3:** Find Minimum Cost Spanning Tree of a given undirected graph using Prim's Algorithm

**Solution:**

```
#include<stdio.h>
#include<conio.h>
int a,b,u,v,n,i,j,mincost=0,cost[10][10];
int visited[10]={0},min;
void main()
{
 clrscr();
```



```

printf("\n Enter the number of nodes:");
scanf("%d",&n);
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
 scanf("%d",&cost[i][j]);
 if(cost[i][j]==0)
 cost[i][j]=999;
}
visited[1]=1;
printf("\n");
while(ne<n)
{
 for(i=1,min=999;i<=n;i++)
 for(j=1;j<=n;j++)
 if(cost[i][j]<min)
 if(visited[i]==0)
 {
 min=cost[i][j];
 a=u=i;
 b=v=j;
 }
 if(visited[u]==0 || visited[v]==0)
 {
 printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
 mincost+=min;
 visited[b]=1;
 }
 cost[a][b]=cost[b][a]=999;
}
printf("\n Minimum cost=%d",mincost);
getch();
}

```

**Output:**

```
Enter the number of nodes:4
Enter the adjacency matrix:
 0 20 10 50
 20 0 60 999
 10 60 0 40
 50 999 40 0

Edge 1:(1 3) cost:10
Edge 2:(1 2) cost:20
Edge 3:(3 4) cost:40
Minimum cost=70
```

## 2.4 References

References of this unit have been given at the end of the book.



---

## PRACTICAL DYNAMIC PROGRAMMING AND UNIT 3 BACKTRACKING

---

### Structure:

#### 3.0 Learning Objectives

- Practical 3.1 Compute the transitive closure of a given directed graph using Warshall's Algorithm
- Practical 3.2 Implement 0/1 Knapsack Problem using Dynamic Programming
- Practical 3.3 From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's Algorithm
- Practical 3.4 Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's Algorithm
- Practical 3.5 (a) Print all the nodes reachable from a given starting node in a digraph using BFS method  
(b) Check whether a given graph is connected or not using DFS method
- 3.6 References

---

### 3.0 Learning Objectives

---

After studying this unit, you will be able to:

- Develop programs using dynamic programming techniques
- Create programs using backtracking techniques
- Compute the transitive closure of a given directed graph using Warshall's algorithm
- Define algorithm specifications



- Implement 0/1 Knapsack problem using dynamic programming.
- Find, from a given vertex in a weighted connected graph, shortest paths to other vertices using algorithm
- Find minimum cost spanning tree of a given undirected graph using Kruskal's algorithm

---

**Practical 3.1:** Compute the transitive closure of a given directed graph using Warshall's Algorithm

---

**Solution:**

```
include <stdio.h>
include <conio.h>

int,a[10][10],p[10][10];

void path()
{
 int i,j,k;
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 for(k=0;k<n;k++)
 p[i][j]=a[i][j];
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 for(k=0;k<n;k++)
 if(p[i][k]==1&&p[k][j]==1) p[i][j]=1;
}

void main()
{
 int i,j;
 clrscr();
 printf("Enter the number of nodes:");
}
```

```
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&a[i][j]);
path();
printf("\nThe path matrix is shown below\n");
for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
 printf("%d ",p[i][j]);
 printf("\n");
}
getch();
}
```

**Output:**

```
Enter the number of nodes:4
Enter the adjacency matrix:
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
The path matrix is shown below
0 1 1 1
0 0 1 1
0 0 0 1
0 0 0 0
```



---

### Practical 3.2: Implement 0/1 Knapsack Problem using Dynamic Programming

---

#### Solution:

```
#include<stdio.h>
#include<conio.h>
int w[10],p[10],v[10][10],n,i,j,cap,x[10]={0};
int max(int i,int j)
{
 return ((i>j)?i:j);
}
int knap(int i,int j)
{
 int value;
 if(v[i][j]<0)
 {
 if(j<=w[i])
 value=knap(i-1,j);
 else
 value=max(knap(i-1,j),p[i]+knap(i-1,j-w[i]));
 v[i][j]=value;
 }
 return(v[i][j]);
}
void main()
{
 clrscr();
 int profit,count=0;
 printf("\nEnter the number of elements\n");
 scanf("%d",&n);
 printf("Enter the profit and weights of the elements\n");
 for(i=1;i<=n;i++)
 {
```

```

printf("For item no %d\n",i);
scanf("%d%d",&p[i],&w[i]);
}
printf("\nEnter the capacity \n");
scanf("%d",&cap);
for(i=0;i<=n;i++)
for(j=0;j<=cap;j++)
if((i==0)||(j==0))
v[i][j]=0;
else
v[i][j]=-1;
profit=knap(n, cap);
i=n;
j=cap;
while(j!=0&&i!=0)
{
if(v[i][j]!=v[i-1][j])
{
x[i]=1;
j=j-w[i];
i--;
}
else
i--;
}
printf("Items included are\n");
printf("SI.no\tweight\tprofit\n");
for(i=1;i<=n;i++)
if(x[i])
printf("%d\t%d\t%d\n",++count,w[i],p[i]);
printf("Total profit = %d\n",profit);
getch();
}

```