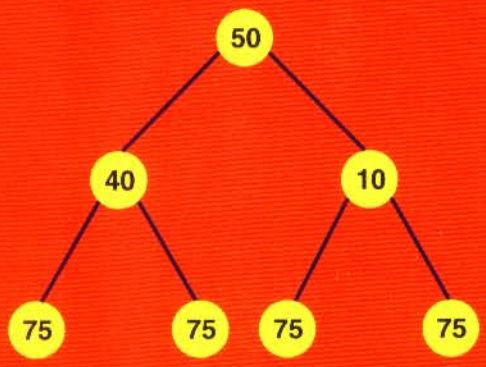
AS PER VTU SYLLABUS REVISED EDITION 2010

ANALYSIS AND DESIGN OF ALGORITHMS

For IV Sem CS/IS



- Includes Lab Programs
- Simple Approach with more examples
- Question & answer type Explanation

Purchase Book with Hologram Only

Other books released for IV Sem. CS/IS Branch

- OBJECT ORIENTED PROGRAMMING USING C++
- INTRODUCTION TO MICROPROCESSORS

- Do not buy the old book - you will not get the latest updated version.
- Helpful for placement.

SRI NA... BANGALO...
005 .1PAD
30100223

A. M. PADMA REDDY

30100223

Analysis & Design of Algorithms

A.M. Padma Reddy B.E., M.Tech., MISTE
Professor and Head, Department of Computer Science
&
Information Science
Sai Vidya Institute of Technology
Rajanukunte, Bangalore - 560 064

NEW HORIZON COLLEGE OF ENGINEERING
NHCE
IN PURSUIT OF EXCELLENCE
LIBRARY
Bangalore - 560 087

SRI NANDI PUBLICATIONS
BANGALORE

Sri Nandi Publications

#256, 2nd main, 10th cross
Padmanabhanagar (B.S.K. 2nd stage)
Bangalore – 560 070

NEW HORIZON COLLEGE OF ENGINEERING
LIBRARY
Acc. No.
Call. No.
Bangalore - 560 087

Phone: 080 – 26698274, 26694533
98450 70827, 99001 70827

E-mail id of author: padmareddy_am@yahoo.com
Mobile : 98450 70827

FIRST EDITION MARCH – 2005
SECOND EDITION JANUARY – 2006
THIRD EDITION FEBRUARY – 2007
FOURTH REVISED EDITION FEBRUARY – 2008
REVISED EDITION FEBRUARY – 2009
REVISED EDITION FEBRUARY – 2010

Copyright © 1999 by Author

Caution No part of this book should be copied, reproduced or adapted, abridged or translated or stored in a retrieval system or transmitted in any form by any device. Resale of the book is strictly prohibited. Breach of the above, will entail in civil and criminal action.

Note Due care and diligence has been taken while editing and printing this book. Neither the author nor the publishers of the book hold any responsibility for any mistakes that may have inadvertently crept in and are not responsible for any consequential damages caused. The book acts as just a reference to move forward and understand the concepts.

PRICE Rs. 200.00

Syllabus

Analysis and Design of Algorithms

IV SEMESTER (Common to CSE/ISE)

Sub Code :
Hours/Week : 4
Total Hours : 52

IA MARKS:25
Exam Hours: 3
Marks : 100

UNIT-I

1. Introduction

What is an Algorithm? Fundamentals of Algorithmic Problem Solving, Important Problem Types, Fundamental Data Structures

- 6 Hours

UNIT-II

2. Fundamentals of the Analysis of Algorithm Efficiency

Analysis Framework, Asymptotic Notations and Basic Efficiency Classes, Mathematical Analysis of Nonrecursive Algorithms, Mathematical analysis of Recursive Algorithms, Example – Fibonacci Numbers

- 6 Hours

UNIT-III

3. Brute Force

Selection Sort and Bubble Sort, Sequential Search and Brute-Force String Matching, Exhaustive Search

4. Divide and Conquer

Mergesort, Quicksort, Binary Search

- 6 Hours

UNIT-IV

Divide and Conquer *contd.*:

Binary tree traversals and related properties, Multiplication of large integers and Stressen's Matrix Multiplication.

5. Decrease and Conquer

Insertion Sort, Depth First Search, Breadth First Search, Topological Sorting, Algorithms for Generating Combinatorial Objects

- 7 Hours

UNIT-V

6. Transform and Conquer

Presorting, Balanced Search Trees, Heaps and Heap sort, Problem Reduction

7. Space and Time Tradeoffs

Sorting by Counting, Input Enhancement in String Matching

- 7 Hours

UNIT-VI

Space and Time Tradeoffs *contd.*:

Hashing

8. Dynamic Programming

Computing a Binomial Coefficient, Warshall's and Floyd's Algorithms, The Knapsack Problem and Memory Functions

- 6 Hours

UNIT-VII

9. Greedy Techniuques

Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, Huffman Trees

10. Limitations of Algorithms Power

Lower-Bound Arguments, Decision Trees

- 7 Hours

UNIT-VIII

Limitations of Algorithms Power *contd.* :

P, NP and NP-Complete Problems

11. Coping with the Limitations of Algorithm Power:

Backtracking, Branch-and-Bound, Approximation Algorithms for NP-Hard Problems

- 7 Hours

Contents

Chapter 1

Introduction

1.1 Notion of algorithm 1.1
1.2 Computing GCD 1.2
1.2.1 Euclid's algorithm (Using modulus) 1.3
1.2.2 Using repetitive subtraction 1.5
1.2.3 Consecutive integer checking 1.7
1.2.4 Middle school method 1.10
1.2.5 Prime numbers using Sieve of Eratosthenes method 1.12
1.3 Fundamentals of Problem solving using Algorithms 1.17
1.4 Important problem types 1.22
1.4.1 Sorting 1.22
1.4.2 Searching 1.23
1.4.3 String processing 1.24
1.4.4 Graph problems 1.24
1.4.5 Combinatorial problems 1.24
1.4.6 Geometric problems 1.25
1.4.7 Numeric problems 1.25
1.5 Introduction to data structures 1.25
1.5.1 Arrays 1.27
1.5.2 Linked lists 1.28
1.5.3 Stacks 1.31
1.5.4 Queues 1.31
1.6 Graphs 1.33
1.7 Trees 1.43
1.8 Sets 1.52
1.9 Other important algorithms and problems 1.54

Chapter 2

Analysis of algorithm efficiency

2.1 Analysis framework 2.1
2.2 Orders of growth 2.4
2.3 Worst case, best case and average case efficiencies 2.6
2.4 Asymptotic notations 2.14
2.4.1 Informal Definitions of asymptotic notations 2.15
2.4.2 Formal Definitions of asymptotic notations 2.17
2.5 Property of asymptotic notation 2.24

2.6 Order of growth using limits 2.25
2.7 Mathematical analysis of Non-recursive algorithms 2.27
2.7.1 Maximum of n elements 2.28
2.7.2 Linear search 2.30
2.7.3 Element uniqueness problem 2.32
2.7.4 Matrix multiplication 2.35
2.8 Mathematical analysis of recursive algorithms 2.37
2.8.1 Factorial of a number 2.37
2.8.2 Tower of Hanoi 2.39
2.8.3 Digits in a binary number (Given a decimal number) 2.43
2.8.4 Fibonacci numbers 2.46
2.8.5 Explicit formula to find n^{th} Fibonacci number 2.48
2.9 Other problems and solutions 2.50
2.10 Problems on asymptotic notations 2.53

Chapter 3

Brute Force

3.1 Introduction 3.1
3.2 Brute force 3.2
3.3 Selection sort (Straight selection sort or pushdown sort) 3.9
3.4 Sequential search 3.13
3.5 Pattern matching (brute-force string matching) 3.14
3.6 Exhaustive search 3.19
3.6.1 Traveling Salesmen Problem (TSP) 3.19
3.6.2 Knapsack problem 3.23
3.6.3 Assignment problem 3.25

Chapter 4

Divide and Conquer

4.1 Introduction 4.1
4.2 Master theorem 4.3
4.3 Sum of n elements of array a 4.3
4.4 Analysis 4.5
4.5 Sorting using divide and conquer 4.7
4.5.1 Merge sort 4.7
4.5.2 Quick sort (Partition Exchange sort) 4.16
4.5.3 Analysis of quick sort 4.22
4.5.4 Binary search 4.29

4.5.5 Binary tree traversals 4.35
4.5.6 Conversion of general tree to binary tree 4.41
4.5.7 Height of a Binary Tree 4.43
4.5.8 Number of nodes in a tree 4.43
4.5.9 Number of leaves in a tree 4.44
4.6 Multiplication of Large Integers 4.45
4.10 Strassen's Matrix Multiplication 4.48

Chapter 5

Decrease and Conquer

5.1 Introduction 5.1
5.1.1 Decrease by a constant 5.2
5.1.2 Decrease by a constant factor 5.4
5.1.3 Variable-size-decrease 5.7
5.2 Insertion Sort 5.7
5.2.1 Analysis(Best case time efficiency) 5.10
5.2.2 Analysis (Worst case time efficiency) 5.10
5.2.3 Analysis (Average case time efficiency) 5.11
5.3 Graph Traversals 5.13
5.3.1 Breadth First Search (BFS) 5.14
5.3.2 Depth First Search (DFS) 5.22
5.3.3 Topological sorting 5.31
5.3.3.1 DFS method 5.31
5.3.3.2 Source Removal method 5.38
5.4 Algorithms for Generating Combinatorial Objects 5.42
5.4.1 Generating Permutations 5.42
5.4.1.1 Bottom up minimal-change algorithm 5.42
5.4.1.2 Johnson Trotter algorithm 5.44
5.4.1.3 Lexicographic algorithm 5.46
5.4.2 Generating subsets 5.48

Chapter 6

Transform and Conquer

6.1 Introduction 6.1
6.2 Presorting 6.2
6.2.1 To check the uniqueness of an element in an array 6.3
6.2.2 Searching 6.6
6.2.3 To find the mode 6.6
6.3 Balanced Search Trees 6.11
6.3.1 AVL Trees 6.16
6.3.1.1 Creating an AVL tree 6.17

6.3.1.2 Left Rotation..... 6.18
6.2.1.3 Right Rotation..... 6.20
6.2.1.4 L-R Rotation 6.21
6.2.1.5 R-L Rotation..... 6.22
6.4. 2-3 Tree..... 6.32
6.4.1 Searching in a 2-3 Tree..... 6.33
6.4.2. Creating a 2-3 Tree..... 6.34
6.4.3 Analysis..... 6.39
6.5 Heap and Heap sort..... 6.41
6.5.1 Create a heap (Bottom-up heap construction)..... 6.45
6.5.2 Analysis for creating a heap using bottom-up approach..... 6.51
6.5.3 Heap sort using bottom up approach..... 6.53
6.5.4 Tracing of heap sort..... 6.54
6.5.5 Analysis of heap sort using bottom up approach..... 6.56
6.6 Create Heap (Top down approach)..... 6.58
6.7 Analysis of heap sort using top-down up approach..... 6.63
6.8 Problem Reduction..... 6.66
6.8.1 To find LCM (Least Common Multiple) of two numbers..... 6.66
6.8.2 To count paths in a graph..... 6.67
6.8.3 Optimization Problem (Maximize/ Minimize)..... 6.68
6.8.4 Optimization Problem (To find extremum points of a function)..... 6.71
6.8.5 Linear Programming..... 6.71
6.8.6 Graph problem..... 6.72

Chapter 7

Space and Time Tradeoffs

7.1 Introduction..... 7.1
7.2 Sorting by counting..... 7.2
7.2.1 Comparison counting..... 7.2
7.2.2 Sorting by Distribution Counting..... 7.8
7.3 Pattern Matching..... 7.13
7.3.1 Horspool's Algorithm..... 7.14
7.3.2 Boyer More algorithm..... 7.23
7.4 Hashing..... 7.23
7.4.1 Open Hashing / Chaining (Separate Chaining)..... 7.24
7.4.2 Closed Hashing (Open Addressing)..... 7.27
7.4.2.1 Linear probing..... 7.27
7.4.2.1 Double hashing..... 7.31

Chapter 8

Dynamic Programming

8.1 Introduction..... 8.1
8.2 Fibonacci numbers..... 8.1
8.3 Computing Binomial Co-efficients..... 8.2
8.4 Warshall's Algorithm (Transitive Closure)..... 8.7
8.5 Floyd's Algorithm (for All Pairs Shortest path problem)..... 8.13
8.6 0/1 Knapsack Problem..... 8.18
8.7 knapsack problem using memory functions..... 8.28

Chapter 9

Greedy Method

9.1 Introduction 9.1
9.2 Dijkstra's algorithm(Single-Source Shortest Path Problem)..... 9.2
9.3 Prim's Algorithm 9.7
9.4 Kruskal's algorithm..... 9.14
9.5 Huffman Trees.....9.23

Chapter 10

Limitations of algorithm power

10.1 Introduction 10.1
10.2 Lower bound arguments 10.1
10.3 Decision trees..... 10.3
10.4 P, NP and NP – Complete problems..... 10.4

Chapter 11

Copying with limitations of algorithm power

11.1 Backtracking..... 11.1
11.2 N-Queens problem 11.1
11.3 Sum of subset problem..... 11.3
11.4 Branch and bound 11.4
11.5 Approximation algorithms for NP-Hard problems..... 11.4

BIBILIOGRAPHY

1. Introduction to Data Structures and applications Tremblay and Sorenson
2. Data Structures and Algorithms in C++ Sahni
3. Data Structures and Algorithms in C++ Drozdek
4. Data Structures and Algorithms in C/C++ Tanenbaum, Langsam, Augenstein
5. Introduction to Algorithms Coremen, Leiserson, Rivest
6. Introduction to The Design and Analysis of Algorithms Levitin
7. Systematic approach to Data Structures using C by Padma Reddy A.M
8. Data Structures and Algorithm Analysis in C Weiss
9. Data Structures and Program Design in C Kruse
10. Design and Analysis of Computer Algorithms Aho/Hopcroft/Ulman
11. Data Structures and algorithms with Object oriented design patterns in C++ by Preiss

Chapter 1: Introduction

What are we studying in this chapter?

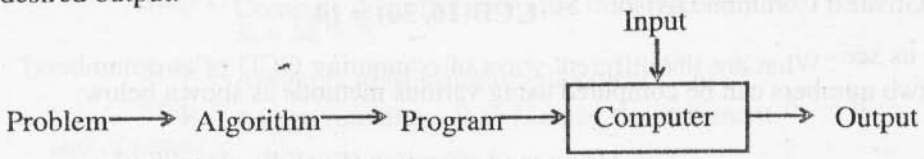
- Notion of Algorithm
- Fundamentals of algorithms
- The various properties of an algorithm
- How to write an algorithm
- Fundamental data structures
  - Arrays
  - Singly linked lists
  - Doubly linked lists
  - Stacks
  - Queues
  - Graphs and their representation
  - Trees
- Algorithms for varieties of problems. **- 6 hours**

1.1 Notion of Algorithm

If we want to be a good computer professional, we should know how to design the algorithms. For this, we first learn, standard algorithms and then learn how to design new algorithms to solve various types of problems. First, we shall see *"What is an algorithm?"*

Definition: An *algorithm* is defined as finite sequence of unambiguous instructions followed to accomplish a given task. It is also defined as unambiguous, step by step procedure (instructions) to solve a given problem in finite number of steps by accepting a set of inputs and producing the desired output. After producing the result, the algorithm should terminate.

The following figure (notion of algorithm) shows how an algorithm is used to get the desired output:



Note: The solution to a given problem is expressed in the form of an algorithm. The algorithm is converted into a program. The program when it is executed, accept the input and produces the desired output.

1.2 Introduction

Now, let us see "What are the criteria that all algorithms must satisfy?" or "What are the properties of an algorithm?" An algorithm must satisfy the following criteria:

- **Input:** Each algorithm should have zero or more inputs. The range of inputs for which algorithm works should be satisfied
- **Output:** The algorithm should produce correct results. At least one output has to be produced.
- **Definiteness:** Each instruction should be clear and unambiguous.
- **Effectiveness:** The instructions should be simple and should transform the given input to the desired output.
- **Finiteness:** The algorithm must terminate after a finite sequence of instructions.

Note: By looking at the algorithm, the programmer can write the program in C or C++ or any of the programming language. Before writing any program, the solution has to be expressed in the form of algorithms.

1.2 Computing GCD

The notion of an algorithm can be explained by computing the GCD of two numbers. Now, let us see "What is GCD of two numbers?"

Definition: The GCD (short form for Greatest Common Divisor) of two numbers *m* and *n* denoted by GCD(m, n) is defined as the largest integer that divides both *m* and *n* such that the remainder is zero. GCD of two numbers is defined only for positive integers but, not defined for negative integers and floating point numbers. For example, GCD(10, 30) can be obtained as shown below:

Step 1: The numbers 1, 2, 5, 10 can divide **10**
Step 2: The numbers 1, 2, 5, 6, 10 and 30 can divide **30**
Step 3: Observe from step 1 and step 2 that the numbers 1, 2, 5 and 10 are common and hence the numbers 1, 2, 5 and 10 are called common divisors of **10** and **30**. Out of these items, 10 is the greatest number which is common and hence it is called Greatest Common Divisor. So, **GCD(10, 30) = 10**.

Now, let us see "What are the different ways of computing GCD of two numbers?" The GCD of two numbers can be computed using various methods as shown below:

Different ways of computing GCD
- → Using mod operation (Euclid's algorithm)
- → Repetitive subtraction (Euclid's algorithm)
- → Consecutive inter checking algorithm
- → Middle school procedure using prime factors

Note: Several algorithms may exist for solving a given problem. It is true not only for computing GCD of two numbers, but for any type of problem. But, the execution speed of each algorithm may be different

1.2.1 Euclid's algorithm (Using mod operator)

The method of finding the GCD of two numbers was named after mathematician Euclid from Alexandria (300 B.C) and this is often called "Euclid's algorithm to find GCD of two numbers". Now, let us "Design Euclid's algorithm using mod operator"

Design: The procedure to obtain GCD(6, 10) using Euclid's algorithm is shown below:

M	N	Remainder (R = M % N)
6	10	6 (6 % 10 = 6)
10	6	4 (10 % 6 = 4)
6	4	2 (6 % 4 = 2)
4	2	0 (4 % 2 = 0)
2	0	stop when n is zero

GCD = 2. Note that when N = 0, M is the GCD of two numbers.

Using the above steps, the algorithm can be written in natural language as shown below:

Example 1.2.1.1: Algorithm (represented in natural language) to compute GCD of two numbers using Euclid's algorithm

Repeat as long as N is not zero

Step 1: Compute the remainder after dividing M by N:
R = M % N

Step 2: Assign N to M using the statement $M \leftarrow N$

Step 3: Assign remainder R to N using the statement $N \leftarrow R$

end of repeat

Output M as GCD of 6 and 10

Using the pseudo code, the algorithm can be expressed as shown below:

Example 1.2.1.2: Iterative algorithm (using pseudo code) to compute GCD of two numbers using Euclid's algorithm

```
ALGORITHM GCD(M, N)
// Description: Computes GCD of M and N
// Input: M and N should be +ve non-zero integers
// Output: Greatest common divisor of M and N
    while N ≠ 0 do
        R ← M % N     // Compute the remainder
        M ← N         // Assign N to M
        N ← R         // Assign remainder to N
    end while
    return M          // Return the GCD
```

```
//Iterative C FUNCTION
int gcd(int m, int n)
{    int r;
     while (n != 0)
     {
          r = m % n;
          m = n;
          n = r;
     }
     return m;
}
```

The recursive definition can be obtained as shown below:

Base case: Now, let us see "How to obtain the base case or terminal condition for the recursive definition?" Observe from the table that GCD (2, 0) = 2. In general,

GCD (M, N) = M if N = 0

General case: Now, let us see "How to compute the general case?" Note the following points from the table:

- GCD(10, 6) = GCD (6, 4). This can be written in general, as shown below:

GCD(10, 6) = GCD (6, 10 % 6)

GCD(M,N) = GCD(N, M % N)

The recursive definition to compute GCD of two numbers M and N using EUCLID's algorithm is shown below:

$$\text{GCD}(M, N) \begin{cases} M & \text{if } N = 0 \\ \text{GCD}(N, M \% N) & \text{otherwise} \end{cases}$$

The recursive algorithm to compute GCD using Euclid's algorithm is shown below:

Example 1.2.1.3: Recursive algorithm to compute GCD of two numbers using Euclid's algorithm

```
ALGORITHM GCD(m, n)
// Description: Computes GCD(m, n)
// Input: M and N should be +ve integers
// Output: Greatest common divisor of M and N

Step 1: [Terminal condition]
        if (N = 0) return M

Step 2: [General case]
        return GCD(N, M % N)
```

```
//RECURSIVE C FUNCTION
int gcd(int m, int n)
{
        if (n == 0) return m;

        return gcd(n, m % n);
}
```

Note: Now, the question is "How do we know that Euclid's algorithm is terminated?" One of the properties of algorithm is that *the algorithm should terminate finally*. Note that in each iteration of the loop, the value of N is decremented (Look at the table to compute GCD of 10 and 6 in the beginning of this section) and finally n becomes zero. It can not be decremented further and hence control comes out of the loop and the algorithm is terminated.

1.2.2 To compute GCD using repetitive subtraction

The GCD of two numbers can also be obtained using Euclid's algorithm by repetitive subtraction. Now, let us "Design Euclid's algorithm by repetitive subtraction" The procedure to obtain GCD using Euclid's algorithm is shown below:

M	N	Description
10	6	M = 10 – 6 = 4 (Since M > N, subtract N from M and store in M)
4	6	N = 6 – 4 = 2 (Since N > M, subtract M from N and store in N)
4	2	M = 4 – 2 = 2 (Since M >N, subtract N from M and store in M)
2	2	Since M and N are same, the GCD will be either M or N which is 2.

GCD = 2.

Note: When M and N are same, GCD is either the final value of M or final value of N. Using the above steps, the algorithm can be written as shown below:

Example 1.2.2.1: Algorithm (represented in natural language) to compute GCD of two numbers using Euclid's algorithm

Repeat as long as M and N are different

Step 1: If M > N, subtract N from M, store result in M

Step 2: If M < N, subtract M from N and store the result in N

end of repeat

Output either M or N

Note: If one of two numbers is a zero, return non-negative number as the GCD.

Using pseudo code, step 1 and step 2 can be written as shown below:

```
if (m > n )
        m ← m - n
else
        n ← n – m
endif
```

and the above statement has to be executed as long as M and N are different. The algorithm using the pseudo code can be expressed as shown below:

Example 1.2.2.2: Iterative algorithm to compute GCD of two numbers using Euclid's algorithm

```
ALGORITHM GCD(m, n)
// Description: Computes GCD(m, n)
// Input: m and n should be +ve integers
// Output: Greatest common divisor of m and n

while m ≠ n do
    if (m > n)
        m ← m – n
    else
        n ← n – m
    end if
end while

return M                    // Return the GCD
```

```
//Iterative C FUNCTION
int gcd(int m, int n)
{
    while (m != n)
    {
        if (m > n)
            m = m – n
        else
            n = n - m
    }

    return m;
}
```

1.2.3 To compute GCD using consecutive integer checking

The alternate method to find the GCD of two numbers is using consecutive integer checking. Now, let us "Design the algorithm to compute GCD of two numbers using consecutive integer checking".

Design By definition, the GCD of two numbers *m* and *n* is the largest integer that divides both *m* and *n* such that the remainder is zero. So, GCD of *m* and *n* cannot be more than the smaller of these two numbers. So, to start with we take smaller of two numbers *m* and *n* as shown below:

small = min(m, n);

General Procedure:

Step 1: Compute small = min(m, n)

Step 2: Divide *m by small.*
If remainder is 0, goto step 3.
Otherwise goto step 4.

Step 3: Divide *n* by *small.*
If remainder is 0, return *small*

Step 4: Decrease *small* by 1
goto step 2

Example 1.2.3.1: Compute GCD(10, 6) using consecutive integer checking method.

Note that initial value of *small* = min(10, 6) = 6

small	m % small	n %small	Remarks
6	10 % 6 = 4		decrement small by 1
5	10 % 5 = 0	6 % 5 = 1	decrement small by 1
4	10 % 4 = 2		decrement small by 1
3	10 % 3 = 1		decrement small by 1
2	**10 % 2 = 0**	**6 % 2 = 0**	**2 is GCD** (remainder is 0)

Note: The division of two numbers *m* and *n* by the number *small* stops when the remainder obtained by dividing both numbers is 0. At this point of time, the value of *small* is 2 which is GCD of 10 and 6. The algorithm to find *GCD* of *m* and *n* using consecutive integer checking is shown in example 1.2.3.2.

Example 1.2.3.2: Algorithm (Using natural language) to compute GCD of 2 numbers (consecutive inter checking)

ALGORITHM GCD(m, n)
//Purpose: To find the GCD of two numbers
//Description: This algorithm computes the GCD of two non-negative and non-zero values accepted as parameters.
//Input: Two non-negative and non-zero integer values *m* and *n*
//Output: GCD of *m* and *n*

Step 1: Find the minimum (m, n) and assign to *small*

Step 2: Divide *m* and *n* by *small*
If remainder is 0 in both cases
return small as the answer
else
goto step 3
end if

Step 3: Decrement *small* by 1 and goto step 2

The above algorithm can be written using pseudo code as shown below:

Example 1.2.3.3: Algorithm (Using pseudo code) to compute GCD of 2 numbers

ALGORITHM GCD(m, n)
//Purpose: To find the GCD of two numbers
//Description: This algorithm computes the GCD of two non-negative and non-zero values accepted as parameters.
//Input: Two non-negative and non-zero integer values *m* and *n*
//Output: GCD of *m* and *n*

```
small ← min(m, n)
while ( 1 )
    if ( m mod small = 0)
        if (n mod small = 0)   // Check if both numbers are divisible
            return small       // If remainder is 0, small is gcd
        end if
    end if
    small ← small -1           // Decrement by 1
end while
```

Note: This algorithm will not work if one of the inputs is zero. So, it is required to specify the range of algorithm's inputs explicitly and carefully.

Example 1.2.1.4 : Find gcd(31415, 14142) by applying Euclid's algorithm. Estimate how many times faster it will be to find gcd(31415, 14142) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from min{m, n} down to gcd(m, n).

Solution (part A) : The GCD(31415, 14142) can be obtained as shown below:

Iteration	m	n	m % n	gcd(m, n) = gcd(n, m % n)
1	31415	14142	3131	gcd(31415, 14142) = gcd(14142, 3131)
2	14142	3131	1618	= gcd (3131, 1618)
3	3131	1618	1513	= gcd (1618, 1513)
4	1618	1513	105	= gcd (1513, 105)
5	1513	105	43	= gcd (105, 43)
6	105	43	19	= gcd (43, 19)
7	43	19	5	= gcd (19, 5)
8	19	5	4	= gcd (5, 4)
9	5	4	1	= gcd (4, 1)
10	4	1	0	= gcd (1, 0)
11	1	0	gcd (1, 0) = 1	

So, gcd(31415, 14142) = gcd(1,0) = 1

Number of iterations = 11

Solution (part B) : Let us compute number of divisions by checking consecutive integers while calculating GCD(31415, 14142). The GCD cannot be more than 14142. So, to start with we take smaller of two numbers *31415* and *14142* as shown below:

small = min(31415, 14142) = 14142

and keep dividing 31415 and 14142 as shown below:

small	m % small	n %small	Remarks
14142	31415%14142 = 3131		decrement small by 1
14141	31415 % 14141 = 3133		decrement small by 1
14140	31415% 14140 = 3135		decrement small by 1
....	....	....	....
1	31415%1 = 0	31451%1 = 0	So, 1 is GCD

Observe that in this scenario, the above procedure is repeated till small is 1. So, number of iterations required = 14142

Minimum number of divisions

The number of divisions in each iteration = 1 (if m % small is not zero)
So, total number of divisions = 1*14142 = 14142

Maximum number of divisons

The number of divisions in each iteration = 2 (if m % small is zero)
So, total number of divisions = 2 *14142

In consecutive integer checking method, number of divisions will be between:
14142 and 2*14142

Number of divisions in Euclid's algorithm = 11 (see solution of part A)

So, Euclid's algorithm is much faster. To see how much faster, divide minimum number of divisions and maximum number of divisions of consecutive integer checking method with number of divisions using Euclids's algorithm:

i.e., 14142/11 = 1285.63 ≈ 1300 (rounding to multiple of 100)
2*14142/11 = 2571.27 ≈ *2600* (rounding to multiple of 100)

So, Euclids' algorithm is approximately 1300 to 2600 times faster than consecutive integer checking method while finding GCD(31415, 14142)

1.2.4 To compute GCD using middle school method

Now, let us see "How to compute GCD of 60 and 120 using middle school method?"

Procedure: The following steps are followed to compute GCD of 60 and 120

Step 1: Compute the prime factors of 60. The prime factors of 60 are obtained as shown below:

$$\begin{array}{r|l} 2 & 60 \\ \hline 2 & 30 \\ \hline 3 & 15 \\ \hline & 5 \end{array}$$

So, the various prime factors of 60 = 2.2.3.5

Step 2: Compute the prime factors of 120. The prime factors of 120 are obtained as shown below:

$$\begin{array}{r|l} 2 & 120 \\ \hline 2 & 60 \\ \hline 2 & 30 \\ \hline 3 & 15 \\ \hline & 5 \end{array}$$

So, the prime factors of 120 = 2.2.2.3.5

Step 3: Identify the common prime factors. The common prime factors of 60 and 120 are shown below:

prime factors of 60 = $\boxed{2.2}$ $\boxed{3.5}$
prime factors of 120 = $\boxed{2.2}$ 2 $\boxed{3.5}$

Common prime factors = 2.2.3.5 = 60

So, GCD(60, 120) = 60

Using the above procedure, the algorithm can be written in natural language as shown below:

Example 1.2.4.1: Algorithm (Using natural language) to compute GCD of 2 numbers using middle school method

ALGORITHM GCD(m, n)
//Purpose: To find the GCD of two numbers
//Description: This algorithm computes the GCD of two non-negative and non-zero values accepted as parameters.
//Input: Two non-negative and non-zero integer values *m* and *n*
//Output: GCD of *m* and *n*

Step 1: Find the prime factors of m

Step 2: Find the prime factors of n

Step 3: Identify the common prime factors obtained in step 1 and step 2

Step 4: Find the product of all common factors and return the result as gcd of two given numbers

Now, let us see "What are the disadvantages of finding the gcd of two numbers using middle-school method?" The various disadvantages are shown below:

- The procedure is more complex and slower when compared to Euclid's algorithm
- This algorithm does not specify how to generate the prime factors and hence it is not a legitimate algorithm
- This algorithm does not specify how to find the common factors and hence it is not possible to write the program

Now, let us see "How to generate prime factors?" To generate prime factors we use the concept of prime numbers known as sieve of Eratosthenes found out in the year 200 B.C in ancient Greece. Now, let us see "Generate prime numbers within 49 using sieve of Eratosthenes method?" The prime numbers can be generated as shown below:

Step 1: Generate all numbers from 2 to 49 as shown below:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,

20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,

35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49

Step 2: Eliminate (strike out) all multiples of 2 except 2 as shown below:

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19,

~~20~~, 21, ~~22~~, 23, ~~24~~, 25, ~~26~~, 27, ~~28~~, 29, ~~30~~, 31, ~~32~~, 33, ~~34~~,

35, ~~36~~, 37, ~~38~~, 39, ~~40~~, 41, ~~42~~, 43, ~~44~~, 45, ~~46~~, 47, ~~48~~, 49

The remaining numbers are shown below: **Note:** square of 2 is eliminated first

2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25,

27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49

Step 3: Eliminate (strike out) all multiples of 3 except 3 as shown below:

2, 3, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, 19, ~~21~~, 23, 25, **Note:** square of 3 is eliminated first

~~27~~, 29, 31, ~~33~~, 35, 37, ~~39~~, 41, 43, ~~45~~, 47, 49

The remaining numbers are shown below:

2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49

Step 4: Eliminate (strike out) all multiples of 5 except 5 as shown below:

2, 3, 5, 7, 11, 13, 17, 19, 23, ~~25~~, 29, 31, ~~35~~, 37, 41, 43, 47, 49

The remaining numbers are shown below: **Note:** square of 5 is eliminated first

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49

Step 5: Eliminate (strike out) all multiples of 7 except 7 as shown below:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, ~~49~~

The remaining numbers are shown below: **Note:** square of 7 is eliminated first

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47

Prime numbers

Design: To design the algorithm, the following approach is used:

Step 1: Generate numbers from 2 to n (assume n = 49) and store these numbers in the array as shown below:

```
Step 1: [Generate the list of integers from 2 to n ]
        for i = 2 to n do
            a[i] = i
        end for
```

By executing the above statement, the array *a* is initialized as shown below:

a[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
2	3	4	5	6	7	8	9	10	11	12	13

a[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]
14	15	16	17	18	19	20	21	22	23	24	25

a[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]
26	27	28	29	30	31	32	33	34	35	36	37

a[38]	[39]	[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[49]
38	39	40	41	42	43	44	45	46	47	48	49

Step 2.a: Now, eliminate multiples of 2 except 2. This is achieved by storing 0 in following index positions:

i = a[4] [6] [8] [10] [12].....a[48]. **Note:** i starts from 2*2 to n in steps of 2
0 0 0 0 0 0

The resulting array is shown below:

a[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
2	**3**	0	**5**	0	**7**	0	**9**	0	**11**	0	**13**

a[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]
0	**15**	0	**17**	0	**19**	0	**21**	0	**23**	0	**25**

a[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]
0	**27**	0	**29**	0	**31**	0	**33**	0	**35**	0	**37**

a[38]	[39]	[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[49]
0	**39**	0	**41**	0	**43**	0	**45**	0	**47**	0	**49**

Step 2.b: Now, eliminate multiples of 3 except 3. This is achieved by storing 0 in following index positions:

i = a[9] [12] [15] [18] [21].....a[48]. **Note:** Here, i starts from 3*3 to n in steps of 3
0 0 0 0 0 0

The resulting array is shown below:

a[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
2	**3**	0	**5**	0	**7**	0	0	0	**11**	0	**13**

a[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]
0	0	0	**17**	0	**19**	0	0	0	**23**	0	**25**

a[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]
0	0	0	**29**	0	**31**	0	0	0	**35**	0	**37**

a[38]	[39]	[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[49]
0	0	0	**41**	0	**43**	0	0	0	**47**	0	**49**

Step 2.c: After 3, the next nonzero item in the array is 5. So, eliminate multiples of 5. The item 4 need not be considered, since its multiples are also multiples of 2 and are already eliminated. Multiples of 5 except 5 can be eliminated by storing 0 in the following index positions:

i = a[25] [30] [35] [40] [45] **Note:** Here, i starts from 5*5 to n in steps of 5
0 0 0 0 0

The resulting array is shown below:

a[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
2	**3**	0	**5**	0	**7**	0	0	0	**11**	0	**13**

a[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]
0	0	0	**17**	0	**19**	0	0	0	**23**	0	0

a[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]
0	0	0	**29**	0	**31**	0	0	0	0	0	**37**

a[38]	[39]	[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[49]
0	0	0	**41**	0	**43**	0	0	0	**47**	0	**49**

Step 2.d: After 5, the next nonzero item in the array is 7. So, eliminate multiples of 7. The item at location 6 need not be considered, since its multiples are also multiples of 2 and are already eliminated. Multiples of 7 except 7 can be eliminated by storing 0 in the following index positions:

i = a[49] **Note:** Here, i starts from 7*7 to n in steps of 7
a[i] = 0

The resulting array is shown below:

a[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]
2	**3**	0	**5**	0	**7**	0	0	0	**11**	0	**13**

a[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]	[24]	[25]
0	0	0	**17**	0	**19**	0	0	0	**23**	0	0

a[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]
0	0	0	**29**	0	**31**	0	0	0	0	0	**37**

a[38]	[39]	[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]	[48]	[49]
0	0	0	**41**	0	**43**	0	0	0	**47**	0	0

Note that the resulting numbers in the table are all prime numbers within 49. Thus, the prime numbers within 49 are shown below:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47

Note: In the above final pass, 7 is the number whose multiples are eliminated. Even though the multiples of 7 are 14, 21, 28, 35, 42 and 49, observe that, except 49 all other numbers are already eliminated.

So, given the number 7, the first item to be eliminated is 49 which is <= n
In other words, given the number 7, the first item to be eliminated is 7*7 which is <= n.

In general, given the number p, the first item to be eliminated is $p*p$ which is <= n.

i.e., $p*p <= n$
i.e., $p^2 <= n$
i.e., $p <= \sqrt{n}$

Note: Since the first prime number is 2, initial value of p is 2. So, p range from 2 to $\sqrt{n}$.

The partial code for this can be written as:

```
for p = 2 to √n
    if (a[p] != 0)                  // p is the next prime number obtained
        i ← p * p                   // Obtain position of multiples of p
        while ( i ≤ n )             // Multiples of p may exist
            a[i] ← 0                // Eliminate a[i] which is multiple of p
            i ← i + p               // Obtain the position of next multiple of p
        end while
    end if
end for
```

Step 3: Copy all the non-zero elements in the locations from 2 to n into another array say B. The array B gives all the prime numbers between 2 to n.

The complete algorithm is shown in example 1.2.4.2.

Example 1.2.4.1: Algorithm to generate prime numbers within n

ALGORITHM SIEVE(n)
//Purpose: To generate prime numbers between 2 to n
//Description: Generates prime numbers using sieve of Eratosthenes method
//Input: A positive integer $n \geq 2$
//Output: Prime numbers $\leq n$

```
Step 1: [Generate the list of integers from 2 to n ]
    for i ← 2 to n do
        a[i] ← i
    end for

Step 2: [Eliminate the multiples of p between 2 to n]
    for p = 2 to √n
        if (a[p] != 0)
            i ← p * p                 // p is the next prime number obtained
            while ( i ≤ n )           // Obtain position of multiples of p
                a[i] ← 0              // Multiples of p may exist
                i ← i + p             // Eliminate a[i] which is multiple of p
            end while                 // Obtain the position of next multiple of p
        end if
    end for

Step 3: [Obtain the prime numbers by copying the non zero elements]
    j ← 0
    for i ← 2 to n do
        if ( a[i] != 0 )
            b[j] ← a[i];
            j ← j + 1
        end if
    end for

Step 4: [Output the prime numbers between 2 to n]
    for i ← 0 to j-1
        write b[i]
    end for

Step 5: [Finished]
    Exit
```

1.3 Fundamentals of Problem solving using Algorithms

Now, let us "Explain the various stages of algorithm design and analysis process with the help of a flowchart" or "Explain algorithm design and analysis process used in problem solving" or let us see "What are the sequence of steps to be performed in designing and analysis of an algorithm?" **[Q:1.a VTU July 2007]**

1.18 Introduction

The algorithm design and analysis process is explained by considering the following flow-chart:

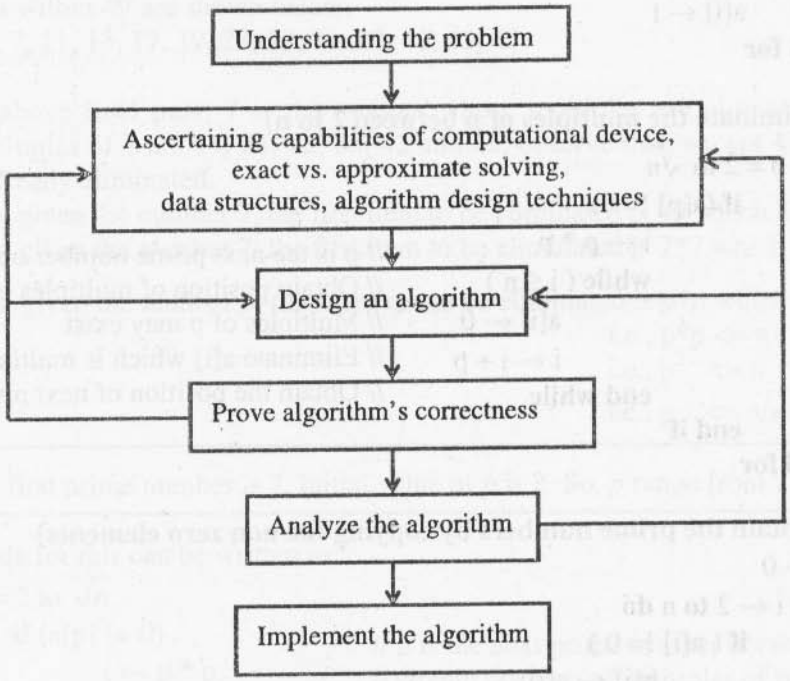


Figure 1.3 Phases in program development

Understanding the problem (Statement of the problem): This is the first and most important step. This step requires us to the following:

- Thoroughly understand the problem
- We need to identify the problem requirements. Requirements can include whether the program requires interaction with the user, whether it produces the output and how to present the output etc.
- In the description of the problem, if we have any doubts we should ask the questions and clarify the doubts.
- We should know what input should be given and what output is obtained along with sample input and output data
- It is very essential to specify the exact range of inputs given to algorithm. If exact range is not specified, the algorithm may work correctly for a majority of inputs but may crash on some other input.

Note: The *correct algorithm is not the one that works most of the time* but *one that works correctly for all legitimate values* (input).

Ascertain the capabilities of a Computational Device: Once we understand the problem clearly, we need to ascertain the capabilities of the computational device based on the following factors:

1. **Architecture of the device:** Based on the architecture of computational device, we may have to write two types of algorithms.
   - For the devices based on Von Neumann architectures in which instructions and data are stored in memory and programs are executed sequentially, we have to design sequential algorithms
   - If a device is capable of executing the instructions in parallel, we may have to design parallel algorithms.
2. **Speed of the device:** For many problems we need not worry about the speed of the device. But, in problems involving military applications, in communication and real time products (such as mobiles) etc., we have to worry about the speed of the device.

Select exact/approximate algorithms: There are two types of algorithms based on the output (result) obtained:

→ Exact algorithm

→ Approximate algorithm

- **Exact algorithm:** The algorithms that solve the problem exactly are called exact algorithms. Algorithms for sorting, searching, string matching, traveling salesman problem, knapsack problem etc., give the exact result and hence are called exact algorithms.

- **Approximate algorithm:** An algorithm that solves the problem approximately is called approximate algorithm. For example, finding square root, solving non-liner equations etc.

  Note: The exact algorithms such as traveling sales person problem are extremely slow and we may require fast approximate result. In such case, we have to choose between the exact and approximate results.

Selection of appropriate data structures: The next step is to choose the proper data structures. Using well defined algorithms and data structures, we can write efficient programs

Algorithms + Data structures = Programs

As shown above, we can use various data structures and write variety or programs. So, to write efficient programs, we have to choose efficient algorithm and efficient data structures.

Design an algorithm: The next step is to *design an algorithm.* Now, let us see "What is an algorithm design technique?" An algorithm design technique is a general method of solving a problem in the form of algorithms. The various algorithms design strategies are shown below:

Algorithm Design Strategies
- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Dynamic programming
- Greedy technique
- Backtracking
- Branch and bound

An algorithm can be specified in various ways. The algorithm can be specified using following three methods:

- Natural language
- Pseudo code
- Flowchart

Natural language: Algorithms can be written in English like statements with very little mathematical expressions. Normally, the algorithms using natural language are not preferred, because of its inherent ambiguity. The instructions specified in natural language should be clear and unambiguous. Each one of us can interpret a given statement written in English in different manner. Hence, natural language is not preferred to represent the solution. So, it is rarely used.

Flowchart: A *flow chart* is a pictorial representation of an algorithm. That is, flowchart consists of sequence of instructions that are carried out in an algorithm. All the steps are drawn in the form of different shapes of boxes, circles and connecting arrows. These shapes represent various operations that are carried out and arrows represent the sequence in which these operations are carried out. Flowcharts are mainly used to help the programmer to understand the logic of the program and are not used for larger algorithms.

Pseudo code: The pseudo code is a method of representing an algorithm using natural language and programming language constructs. Thus, pseudo code is a mixture of natural language and programming language constructs. The pseudo code may be similar in many respects to higher languages such as C, Pascal or Java and hence, if we know any of these languages, we should have little trouble reading the algorithms. The pseudo code to find the area of a rectangle can be written as

```
read (length, breadth)
area ← length * breadth
write "Area of rectangle is", area
exit
```

Proof of algorithm's correctness: After designing the algorithm for a specific problem, we have to prove its correctness.

- It is our responsibility to prove that the algorithm produces the required output for every legitimate input.
- The proof of correctness of an algorithm is normally done using *mathematical induction.*

Analysis of algorithms: The solution to the problem can be obtained using different algorithms. For example, elements can be sorted using bubble sort, selection sort, insertion, quick sort etc. Out of the many sorting techniques we have to choose the most efficient algorithm. So, let us see "What are the two methods using which efficiency of an algorithm can be measured?" The efficiency of an algorithm can be measured using:

- Time efficiency: indicates how fast the algorithm can be executed
- Space efficiency: indicates how much (minimum) extra memory is used by the algorithm during execution

The analysis of algorithms helps us to:

- Select an algorithm which is more efficient in terms of *time* and *space* i.e., knowing the *estimate of time and space* best algorithm can be selected.
- Analysis also helps to find the bottlenecks in a given program. For example, which portion of the program consumes more time? By knowing this, one can re-design the algorithm to solve the same problem much efficiently and easily. The weaker algorithm can be improved.
- It is very important to write the algorithms so as to maintain the *simplicity.* The simple algorithms are easier to understand, easier to debug and easy to program.
- Another characteristic feature of an algorithm is the *generality.* It is required to write general programs always.

Note: Follow **KISS** (**K**eep **I**t **S**imple and **S**tructured) technique: Write simple algorithms and programs for a given problem. Do not believe in lengthy, complicated algorithms and programs

Note: If we are not satisfied with the efficiency, simplicity and generality of an algorithm, it is necessary to re-design the algorithm.

Implementation (Coding an algorithm): Once we have designed the algorithm and verified its correctness, the next step is to convert it into an equivalent programming code. Using the text editor, we can enter the program into computer. We should see that the program follows the syntax of programming language. After successful compilation we can execute the program. The syntax may vary from language to language. The selection of the programming language is also important. The language selected should support the features mentioned in the design phase. For an object oriented design one can select C++ where as C language cannot be selected even though C is subset of C++.

1.4 Important Problem Types

Now, let us see "What are the various important problem types?" along with examples or "List the different problem types and name at least one algorithm of each type" **[Q:1.b VTU July 2007]** The different problem types are shown below:

Important problem types →
- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

1.4.1 Sorting

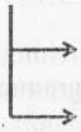
Now, let us see "*What is sorting?*"

Definition: More often programmers will be working with large amount of data and it may be necessary to arrange them in ascending or descending order. This process of rearranging the given elements so that they are in ascending order or descending order is called *sorting*. In real time environment it may be necessary to arrange numbers, strings, record of students, employees etc. Sorting is very essential for searching the dictionaries, telephone directories etc.

The various algorithms that can be used for sorting are bubble sort, selection sort, insertion sort, quick sort, merge sort, heap sort etc.

Normally, we arrange the records based on some *key* information. Now, let us see "What is a key?" We may have a record of students consisting of names, marks, USN etc., or a record of employees consisting of name, salary, id etc. In such cases we may have to arrange the records in increasing order of marks or in alphabetical order, based on employee/student id. Such a specially chosen piece of information based on which the records are sorted is called a *key*.

Now, the next question is "What are the two properties of sorting algorithms?" The two properties of sorting techniques are:



- **Stable**: Now, the question is "When we say that the sorting algorithm is stable?" If the sorting algorithm preserves the relative order of any two equal elements, then the sorting algorithm is stable. That is, if input has two elements a[i] and a[j] at positions i and j respectively where $i < j$, after sorting if the elements a[i] and a[j] are moved from i and j to the positions i' and j' and if $i' < j'$, then the algorithm is called *stable*.

  For example, merge sort, bubble sort etc.

- **In place:** Now, the question is "When we say that the algorithm is in place?" If an algorithm does not require an extra memory space except for few memory units, then the algorithm is said to be *in place*.

  For example, bubble sort, selection sort, insertion sort etc.

1.4.2 Searching

Now, let us see "What is searching?"

Definition: More often we will be working with large amount of data. It may be necessary to determine whether a particular item (search key) is present in the large amount of data. This process of finding a particular item in the large amount of data is called searching.

The various algorithms that are used for searching are:

- Binary search
- Linear search
- Interpolation search
- Hashing etc.

No searching technique fits for all the situations. Some algorithms work faster on sorted items, some require extra space, some are complex to understand, and some are very easy to understand and so on. Based on the situation and whether the items are sorted, the appropriate searching technique can be used.

1.4.3 String processing

Now, let us see "What is string processing?"

Definition: String processing deals with manipulation of characters or strings, string matching, search and replace, deleting a string in a text, inserting a string at any position in the text etc. The string processing algorithms deal with numeric and non-numerical data.

For example, string processing algorithms such as string matching algorithms are used in the design of assemblers, compilers etc. and various programming languages. Now, let us see "What is string matching?"

Definition: The process of searching for a given word in a text is called string matching. The various algorithms that we use for string matching are shown below:

- String/Pattern matching using Brute force method
- String/Pattern matching using Boyer More algorithm
- String/Pattern matching using Horspool's algorithm

1.4.4 Graph problems

Graph algorithms play a very important role in the field of computer science. So, let us see "What is a graph?"

Definition: A graph is defined as G = (V, E) where

- V is set of vertices
- E is set of edges connecting the vertices

A graph can be represented using arrays or linked lists. The array representation of a graph is called adjacency matrix and linked list representation of a graph is called adjacency list. The graphs are used to model real time situations such as *transportation and communication networks, project scheduling, games* etc. The graph algorithms that we deal are:

- Graph traversal algorithms (BFS, DFS)
- Topological sorting algorithm
- Shortest path problem (Dijkstra's algorithm)

1.4.5 Combinatorial problems

Let us see "What are combinatorial problems?"

Definition: The problems that involve permutations, combinations or subset constructions are called *combinatorial problems*. The combinatorial problems satisfy certain conditions by either *maximizing the profit* or *minimizing the cost*. Usually the problems involving combinatorial problems are most difficult problems in computing both from theoretical and practical perspective because of the following reasons:

- Even if the size of the problem increases slightly, the number of combinatorial objects grows extremely fast and reaches unimaginable magnitudes.
- There are no known algorithms to solve these problems in an acceptable amount of time.

For example, *traveling salesman problem* and *graph coloring* problems are some of the examples of combinatorial problems.

1.4.6 Geometric problems

Now, let us see "What are geometric problems?"

Definition: The problems that deal with geometric objects such as points, curves, lines, polygons etc are called *geometric problems*. Variety of geometric algorithms are used in *computer graphics, robotics* etc. For example, the algorithms exist for the following problems:

- Closest-pair problem: Given a set of points in the plane, find the closest pair of points in a plane
- Convex hull problem: Find the smallest convex polygon that include all the points of a given set.

1.4.7 Numerical problems

Now, let us see "What are numerical problems?"

Definition: The problems involving mathematical manipulations such as solving equations, computing differentiations and integrations, evaluating various types of functions etc are called numerical problems. All these problems can not be solved exactly but approximate and nearly exact values can be found.

For example, Gauss elimination methods of solving simultaneous equations, Newton-Raphson method to find the square root, cube root etc are numerical problems.

1.5 Introduction to Data Structures

Now, let us concentrate on the various data structures that are used in the text. First, let us see "What is a data structure?"

Definition: A data structure is a method of storing the data in the computer so that it can be used efficiently. The *data structures* deal with:

- The study of how the data is organized in the memory
- How efficiently the data can be retrieved
- How efficiently the data can be manipulated
- The possible ways in which different data items are logically related

The data structures are classified as shown below:

Types of data structures
- → Primitive data structures (Ex: int, float, char, double etc)
- → Non primitive data structures
  - → Linear data structures (Ex: stacks, queues, linked lists)
  - → Non linear data structures (Ex: Trees, Graphs)

Now, let us see "What are primitive data structures?"

Definition: The Primitive Data structures are the data structures that can be manipulated directly by machine instructions.

For example, the integers, floating point numbers (real), characters, pointers etc., are some of the primitive data structures. In C language, the different primitive data structures are defined using the data types such as **int, float, char** and **double.**

Now, let us see "What are non-primitive data structures?"

Definition: The Non-primitive data structures are the data structures that cannot be manipulated directly by machine instructions.

For example, arrays, structures, stacks, queues, linked lists, files etc., are some of the non- primitive data structures. The non-primitive data structures are classified into linear data structures and non-linear data structures.

Now, let us see "What are linear data structures and non-linear data structures?"

Definition: The data structures that show the relationship of logical adjacency between the elements are called linear data structures.

For example, arrays, stacks, queues, linked lists etc.

Definitions: The data structures that do not show the relationship of logical adjacency between the elements are called *non-linear data structures.* For example, trees, graphs etc are non-linear data structures.

In the subsequent sections, let us discuss various linear and non-linear data structures in brief.

1.5.1 Arrays

Let us see "What is an array?"

Definition An *array* is defined as an ordered collection of similar (homogeneous) data items. All the data items of an array are stored in consecutive memory locations in memory. The elements of an array are of same data type and each item can be accessed using the same name.

To access an item in the array, we specify the name of the array along with position of the item. The position of the item must be written within square brackets ([]). The position of the item enclosed within square brackets is called "*subscript*" or "*index*". For example, consider an array of marks of 5 students as shown below:

80	90	45	99	100
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

The marks of each student can be accessed as shown below:

marks[0] i.e., 80 – represent marks of first student
marks[1] i.e., 90 – represent marks of second student
marks[2] i.e., 45 – represent marks of third student
marks[3] i.e., 99 – represent marks of fourth student
marks[4] i.e., 100 – represent marks of fifth student

Note: Any element in an array, irrespective of its location in an array, can be accessed in the same amount of time and each element in an array occupies the same amount of space. Strings are considered to be an array of characters. Let us see "What is a bit string?" A string consisting of zeros and ones is a binary string or bit string.

Even though arrays are very useful data structures, there are some limitations due to *sequential* and *static allocation techniques*. Now, let us see "What are the limitations of arrays?" The limitations of the array are:

- *The size of the array is fixed*: The memory required for most of the applications often cannot be predicted while writing a program. If more memory is allocated and the application requires less memory, it results in wastage of memory space. If less memory space is allocated and if the applications require more memory space during execution, it is not possible to allocate extra memory for arrays during execution.
- *Array items are stored contiguously*: Sometimes enough contiguous memory locations may not be available. There are situations where a number of chunks of contiguous memory locations are available. Even though the size of these chunks of

contiguous memory locations is same as the size of the array, this memory space cannot be used, since arrays require contiguous storage space.

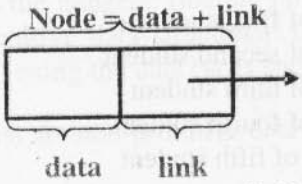
- *Insertion and deletion operations involving arrays is a tedious job*: Consider an array consisting of more than 100 or 200 elements. If we have to insert an item in the ith position, all the elements from $(i+1)^{th}$ position have to be moved to the next subsequent locations to make room for the item to be inserted. Only after this movement, item can be inserted into i^{th} location. Similarly, to delete an ith item, all elements from $(i+1)^{th}$ position, should be moved to their corresponding previous locations. So, if the applications require extensive manipulation of stored data, more time is spent only in the movement of data.

All the above limitations can be overcome using linked list.

1.5.2 Linked lists

Now, let us see "What is a linked list? What are the various types of linked lists?"

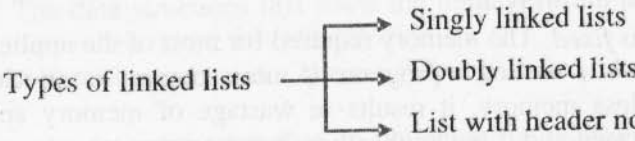
Definition: A linked list is a data structure which is collection of zero or more nodes with each node consisting of two fields *data* and *link*. The pictorial representation of the node is shown below:



Note that a node in a list consists of two fields namely data and link:

- data: This field is used to store the actual data or information to be manipulated
- link: A link basically contains address of the next node. So, given address of a node, we can easily obtain addresses of subsequent nodes. Thus, the adjacency between the nodes is maintained by means of links.

Now, let us see "What are the different types of linked lists?" The linked lists are classified as shown below:



Now, let us see "What is a singly linked list?"

Definition: A singly linked list is a linked list, where each node has designated field called link field which contains address of the next node. If there exists only one link in each node of the list, the linked list is called singly linked list. A singly linked list may have a collection of nodes with various fields. The pictorial representation of a singly linked list consisting of the items 50, 20, 45, 10 and 80 is shown below:

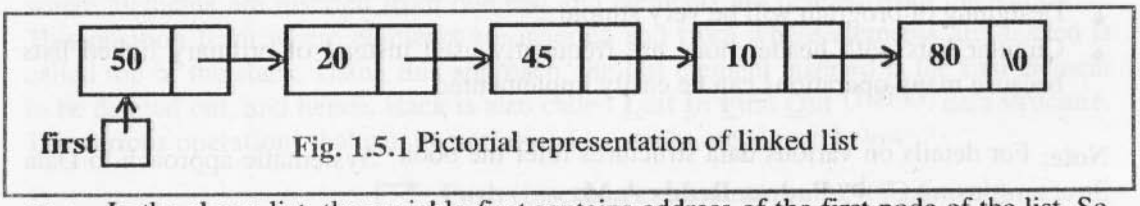


Fig. 1.5.1 Pictorial representation of linked list

Note: In the above list, the variable *first* contains address of the first node of the list. So, the variable *first* is a pointer variable which points to the first node of the list. Using the pointer *first*, we can access any node in the list.

Now, let us see "What is a header node in the linked list? What is its advantage?"

Definition: A header node in a linked list is a special node which may consist of some special information such as:

- The address of the first node of the list
- The address of the last node of the list
- The information about the linked list such as number of nodes in the list
- One or more *links* pointing to different types of linked lists.

For example, the various lists consisting of header nodes is shown below:

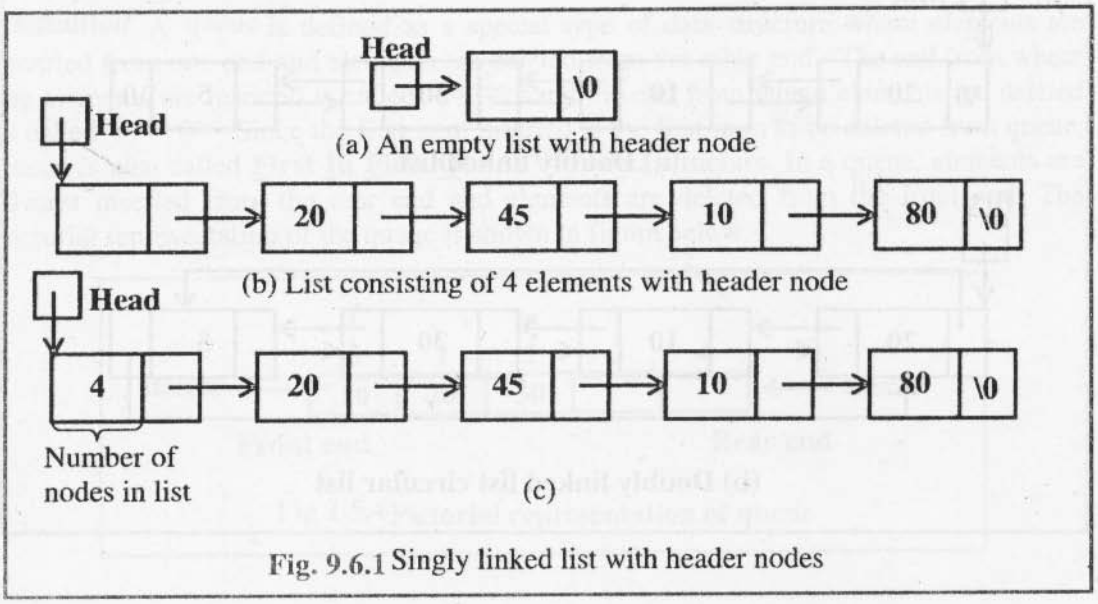


Fig. 9.6.1 Singly linked list with header nodes

Now, let us see "What are the advantages of a list with a header node?" The advantages of a list consisting of header node are shown below:

- Simplifies Insertion and deletion operations
- Avoid the usage of various cases such as "if only one node is present what to do?"
- Designing of program will be very simple
- Circular lists with header node are frequently used instead of ordinary linked lists because many operations can be easily implemented

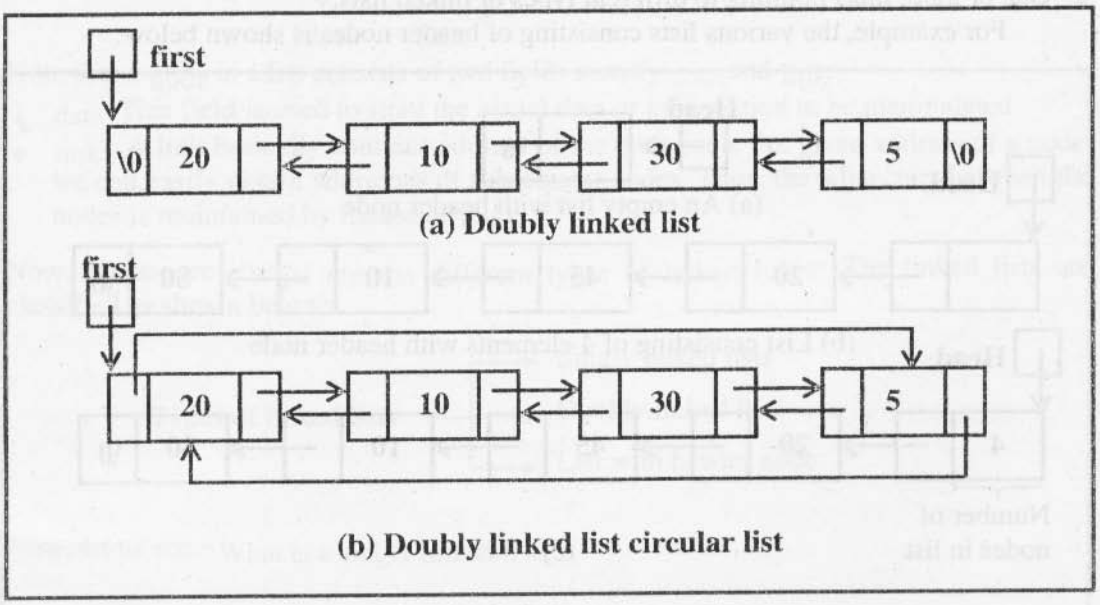
Note: For details on various data structures refer the book "Systematic approach to Data Structures using C" by **Padma Reddy A.M.**

Now, let us see "What is a doubly linked list?"

Definition: One of the most powerful variations of linked lists is the doubly linked list. A doubly-linked list is a linear collection of nodes where each node is divided into 3 parts:

- **info:** a field where the information has to be stored
- **llink:** a pointer field which contains address of the left node of the list
- **rlink:** a pointer field which contains address of the right node of the list

Using such lists, it is possible to traverse the list in forward and backward directions. Such a list where each node has two links is also called a two way list. The pictorial representation of a doubly linked list is shown in figure below:



(a) Doubly linked list

(b) Doubly linked list circular list

1.5.3 Stacks

Let us see another important data structure stack. Let us see "What is a stack? What are the various operations that can be performed on stacks?"

Definition: A stack is a special type of data structure (an ordered collection of items) where elements are inserted from one end and elements are deleted from the same end. This position from where elements are inserted and from where elements are deleted is called top of the stack. Using this approach, the last element inserted is the first element to be deleted out, and hence, stack is also called **L**ast **I**n **F**irst **O**ut **(LIFO)** data structure. The various operations that can be performed on stacks are shown below:

Stack operations →
- **Push** (Inserting an element on the top of stack)
- **Pop** (Deleting an element from the top of stack)
- **display** (Display contents of stack)

Once we know the definition of a stack, let us concentrate on "What are the applications of stack?" The various applications of stacks are:

- Conversion of expressions
- Evaluation of expressions
- Recursion

1.5.4 Queues

Let us see another important data structure *queue*. First, let us see "What is a queue? What are the various operations that can be performed on queues?

Definition: A queue is defined as a special type of data structure where elements are inserted from one end and elements are deleted from the other end. The end from where the elements are inserted is called rear end and the end from where elements are deleted is called front end. Since the first item inserted is the first item to be deleted from queue, queue is also called **F**irst **I**n **F**irst **O**ut **(FIFO)** data structure. In a queue, elements are always inserted from the rear end and elements are deleted from the front end. The pictorial representation of the queue is shown in figure below:

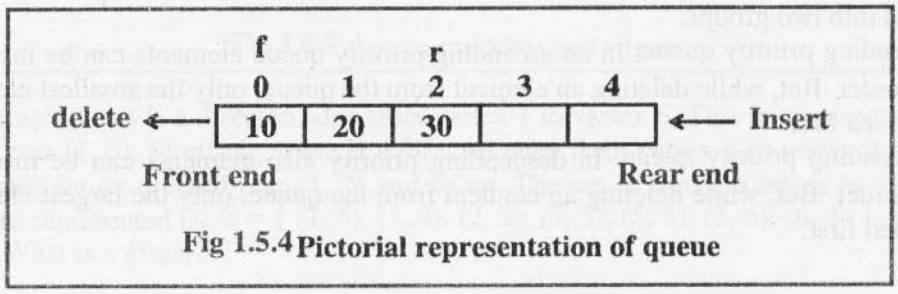


Fig 1.5.4 Pictorial representation of queue

Here, the first item and last item can be accessed using the index variables *f* and *r*. So, the first item inserted into the queue is 10, the second item inserted is 20 and the last item inserted is 30. Any new element to be inserted into this queue has to be inserted towards right of 30 and that item will be the last element in the queue. The first item to be deleted from the queue is the item, which is at the front of the queue i.e., 10. So, it is very clear from the operations performed on queues that **First** item **Inserted** is the **First** item to be deleted **Out** from the queue. So, queue is also called **First In First Out** (FIFO) data structure.

This data structure is useful in time-sharing systems where many user jobs will be waiting in the system queue for processing. These jobs may request the services of CPU, main memory or external devices such as printer etc. All these jobs will be given a fixed time for processing and are allowed to use one after the other. This is the case of an ordinary queue where priority is same for all the jobs and whichever job is submitted first, that job will be processed. Now, let us see "What are the various operations that can be performed on queues?" The various primitive operations that can be performed on queues are

Operations on queues →
- Insert an item into queue
- Delete an item from queue
- Display the contents of queue

Note: Let us see "What is *enqueue and dequeue*?" Inserting an element into queue is called *enqueue* and deleting an element from the queue is called *dequeue*.

In this section, let us see another variation of queue called priority queue. Let us see "What is priority queue? What are the various types of priority queues?"

Definition: The priority queue is a special type of queue in which items can be inserted or deleted based on the priority. Always an element with highest priority is processed before processing any of the lower priority elements. If the elements in the queue are of same priority, then the element, which is inserted first into the queue, is processed. Priority queues are used in job scheduling algorithms in the design of operating system where the jobs with highest priorities have to be processed first. The priority queues are classified into two groups:

- Ascending priority queue: In an ascending priority queue elements can be inserted in any order. But, while deleting an element from the queue, only the smallest element is removed first.
- Descending priority queue: In descending priority also elements can be inserted in any order. But, while deleting an element from the queue, only the largest element is deleted first.

1.6 Graphs

In this section, the various terminologies normally associated with graphs are discussed. First, let us see "What is a vertex?"

Definition: A vertex is a synonym for a node. A vertex is normally represented by a circle. For example, consider the following figure:

Fig 1.6.1 showing vertices

In the above figure, the nodes identified by 1, 2, 3, 4 are also called vertices and normally denoted by a set V = {1, 2, 3, 4}. Now, let us see "What is an edge?"

Definition: If *u* and *v* are vertices, then a *line* joining two vertices *u* and *v* is called an *edge* and it is denoted by an ordered pair (u,v). The edge can be directed or undirected. For example, consider the following graph:

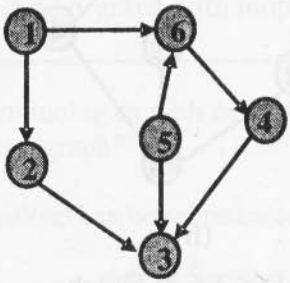


Fig 1.6.2 showing vertices and edges

In this graph, there is a directed edge from vertex 1 to vertex 6. This is represented by an ordered pair (1, 6). Similarly, there is a directed edge from vertex 1 to vertex 2 which is represented by an ordered pair (1, 2) and so on. The set of directed edges in the above figure are represented by E = { (1, 6), (1, 2), (2, 3), (4, 3), (5, 3), (5, 6), (6, 4) }. Now, let us see "What is a graph?"

Definition: A graph G is defined as a pair of two sets V and E denoted by

$$G = (V, E)$$

where

- V is set of vertices
- E is set of edges.

For example, consider the graph shown in figure 1.6.2. Here, G = (V, E) where

$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{ (1, 6), (1, 2), (2, 3), (4, 3), (5, 3), (5, 6), (6, 4) \}$

Note: $|V| = |\{1, 2, 3, 4, 5, 6\}| = 6$ represent the number of vertices in the graph.

$|E| = |\{(1, 6), (1, 2), (2, 3), (4, 3), (5, 3), (5, 6), (6, 4) \}| = 7$ represent the number of edges in the graph.

Now, let us see "What is a directed graph? What is an undirected graph?"

Definition: A graph G = (V, E) in which every edge is directed is called a directed graph. The directed graph is also called digraph. The graph shown in fig.1.6.3.c is a directed graph. A graph G = (V, E) in which every edge is undirected is called an undirected graph. The graph shown in fig.1.6.3.a and 1.6.3.b are undirected graphs

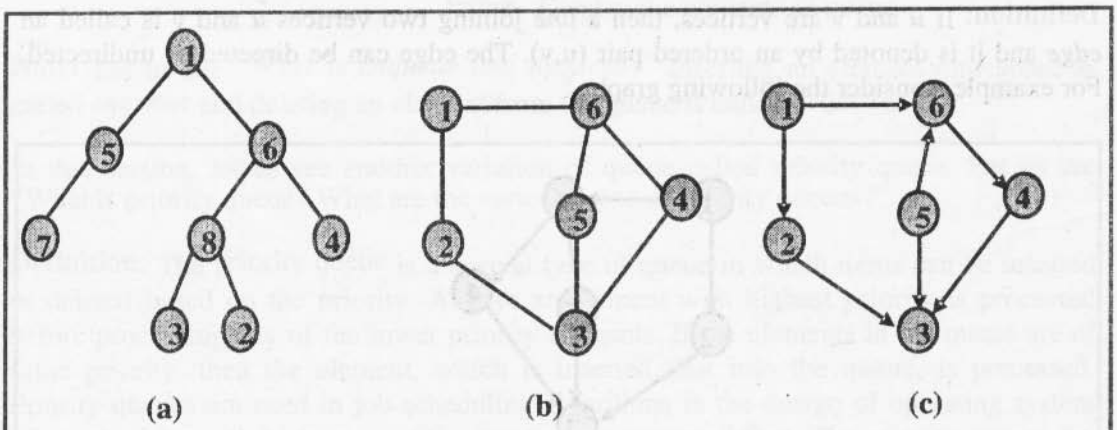


Fig. 1.6.3: Graphs

Now, let us see "What is a complete graph?"

Definition: A graph G = (V, E) is said to be a complete graph, if there exists an edge between every pair of vertices.

For example, the graphs shown in figure 1.6.3 are not complete, whereas the graph shown in fig 1.6.4 is complete.

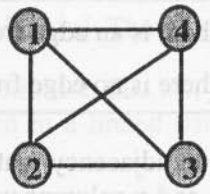


Fig. 1.6.4: Complete graph

Let us see "What is a loop?"

Definition: A loop is an edge which starts and ends on the same vertex. A loop is represented by an ordered pair (i, i). This indicates that the edge originates and ends in the same vertex. A graph with a loop (2,2) is shown below:

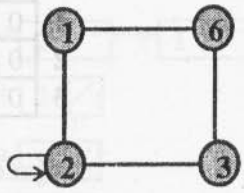


Fig. 1.6.5: A graph with loop at vertex 2

Once we know the various terminologies with respect to graph, let us see "What are the two common ways to represent a graph?"

The graphs for computer algorithms can be represented in two different methods:

Representation of graph → Adjacency matrix
Representation of graph → Adjacency linked list

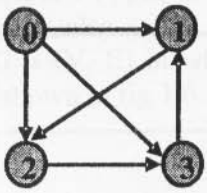
Let us see "What is an adjacency matrix? and explain with example"

Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges. Let N be the number of vertices in graph G. The adjacency matrix A of a graph G is formally defined as shown below:

$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j. \\ 0 & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

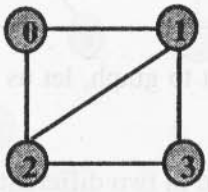
- It is clear from the definition that an adjacency matrix of a graph with n vertices is a boolean square matrix with n rows and n columns with entries 1's and 0's (bit-matrix)
- If there exists an edge (i, j), the element in i^{th} row j^{th} column will be 1.
- If there is no edge from vertex i to vertex j, then the element in i^{th} row j^{th} column will be 0.

Note: The above definition is true both for directed and undirected graph. For example, fig.1.6.6 shows the directed and undirected graphs along with equivalent adjacency matrices:



	0	1	2	3
0	0	1	1	1
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

(a) Directed graph **Adjacency matrix**



	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

(b) Undirected graph **Adjacency matrix**

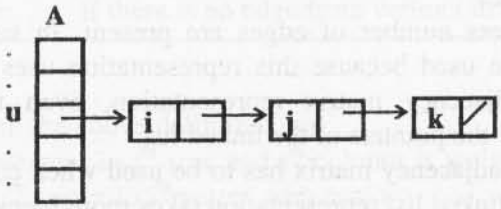
Fig. 1.6.6: Graphs and equivalent adjacency matrices

Now, let us see "What is an adjacency list? and explain with example"

Definition: Let G = (V, E) be a graph. An *adjacency linked list* is an array of n linked lists where n is the number of vertices in graph G. Each location of the array represents a

vertex of the graph. For each vertex $u \in V$, a linked list consisting of all the vertices adjacent to u is created and stored in A[u]. The resulting array A is an adjacency list.

Note: It is clear from the above definition that if i, j and k are the vertices adjacent to the vertex u, then i, j and k are stored in a linked list and starting address of linked list is stored in A[u] as shown below:



For example, fig.1.6.7 shows the directed and undirected graphs along with equivalent adjacency linked list:

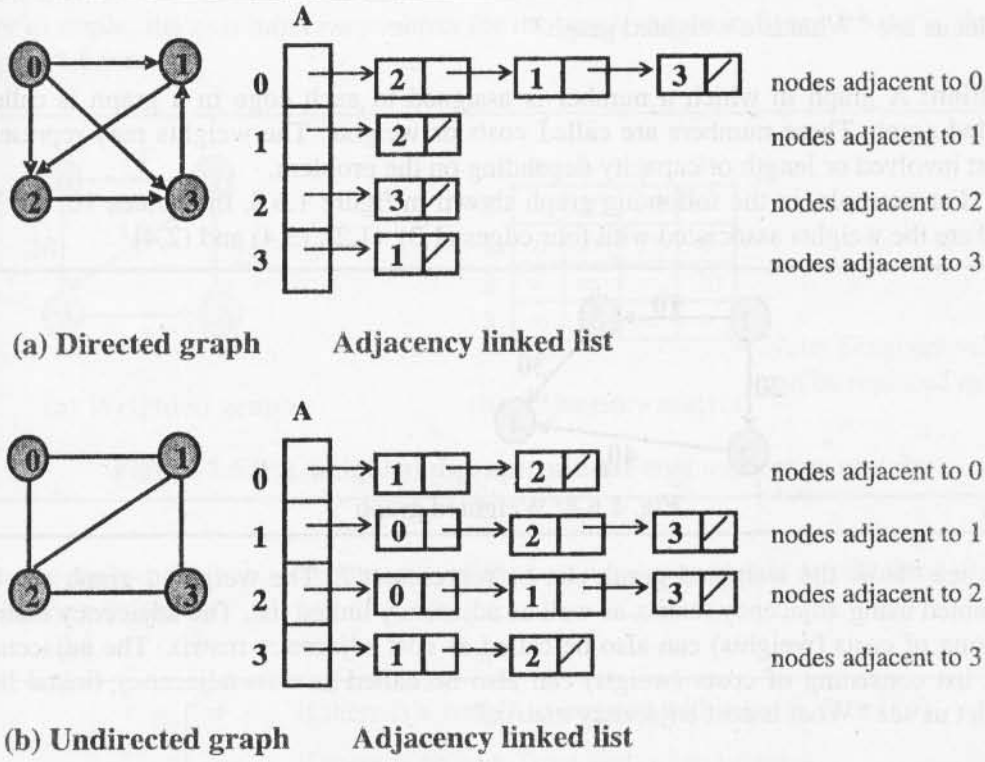


Fig. 1.6.7: Graphs and equivalent adjacency linked lists

Now, let us see "Which graph representation is best?" The graph representation to be used depends on the following factors:

- Nature of the problem
- Algorithm used for solving
- Type of the input.
- Number of vertices and edges:
  - If a graph is *sparse*, less number of edges are present. In such case, the adjacency list has to be used because this representation uses lesser space when compared to adjacency matrix representation, even though extra memory is consumed by the pointers of the linked list.
  - If a graph is *dense*, the adjacency matrix has to be used when compared with adjacency list since the linked list representation takes more memory.

Note: So, based on the nature of the problem and based on whether the graph is *sparse* or *dense*, one of the two representations can be used.

Now, let us see "What is a weighted graph?"

Definition: A graph in which a number is assigned to each edge in a graph is called weighted graph. These numbers are called costs or weights. The weights may represent the cost involved or length or capacity depending on the problem.

For example, in the following graph shown in figure 1.6.8, the values 10, 20, 30 and 40 are the weights associated with four edges (1,3), (1,2), (3,4) and (2,4)

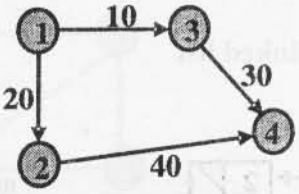


Fig. 1.6.8: Weighted graph

Let us see "How the weighted graph can be represented?" The weighted graph can be represented using adjacency matrix as well as adjacency linked list. The adjacency matrix consisting of costs (weights) can also be called as cost adjacency matrix. The adjacency linked list consisting of costs (weigts) can also be called as cost adjacency linked list. Now, let us see "What is cost adjacency matrix?"

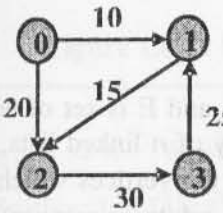
Definition: Let G = (V, E) be the graph where V is set of vertices and E is set of edges with *n* number of vertices. The cost adjacency matrix A of a graph G is formally defined as shown below:

$$A[i][j] = \begin{cases} w & \text{if there is a weight associated with edge from vertex } i \text{ to vertex } j. \\ \infty & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

It is clear from the above definition that

- The element in i^{th} row and j^{th} column is weight *w* provided there exist an edge from i^{th} vertex to j^{th} vertex with cost *w*
- ∞ *if there is no edge* from vertex *i* to vertex *j*.
- The cost from vertex *i* to vertex *i* is ∞ *(assuming there is no* loop).

For example, the *cost adjacency matrix* for the graph shown in figure 1.6.9.a is shown in figure 1.6.9.b.



(a) Weighted graph

	0	1	2	3
0	∞	10	20	∞
1	∞	∞	15	∞
2	∞	∞	∞	30
3	∞	25	∞	∞

Note: Diagonal values can be replaced by 0's

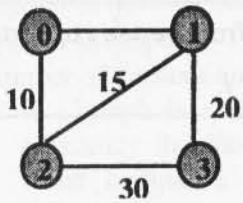
(b) Adjacency matrix

Figure 1.6.9 A weighted digraph and the cost adjacency matrix

For the undirected graph, the elements of the cost adjacency matrix are obtained using the following definition:

$$A[i][j] = \begin{cases} w & \text{if there is a weight associated with edge (i, j) or (j, i)} \\ \infty & \text{if there is no edge from vertex } i \text{ to vertex } j. \end{cases}$$

For the undirected graph shown in figure 1.6.10.a, the adjacency matrix is shown in figure 1.6.10.b.



(a)

	0	1	2	3
0	∞	25	10	∞
1	25	∞	15	20
2	10	15	∞	30
3	∞	20	30	∞

Diagonal values can be replaced by 0's

(b)

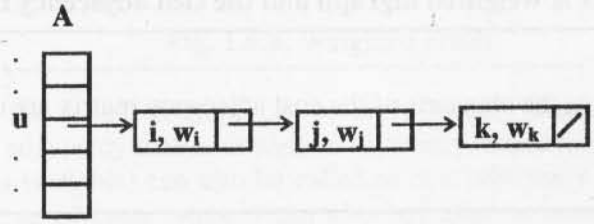
Figure 1.6.10 Weighted undirected graph and the adjacency matrix

Note: The cost adjacency matrix for the undirected graph is symmetric (i.e., a[i, j] is same as a[j, i]) whereas the cost adjacency matrix for a directed graph may not be symmetric.

Note: For some of the problems, it is more convenient to store 0's in the main diagonal of cost adjacency matrix instead of ∞.

Now, let us see "What is cost adjacency linked list?"

Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges with *n* number of vertices. A *cost adjacency linked list* is an array of *n* linked lists. For each vertex u ∈ V, A[u] contains the address of a linked list. All the vertices which are adjacent from vertex *u* are stored in the form of a linked list (in an arbitrary manner) and the starting address of first node is stored in A[u]. If *i, j* and *k* are the vertices adjacent to the vertex u, then *i, j* and *k* are stored in a linked list along with the weights in A[u] as shown below:



For example, the figure 1.6.11 shows the weighted diagraph and undirected graph along with equivalent adjacency list.

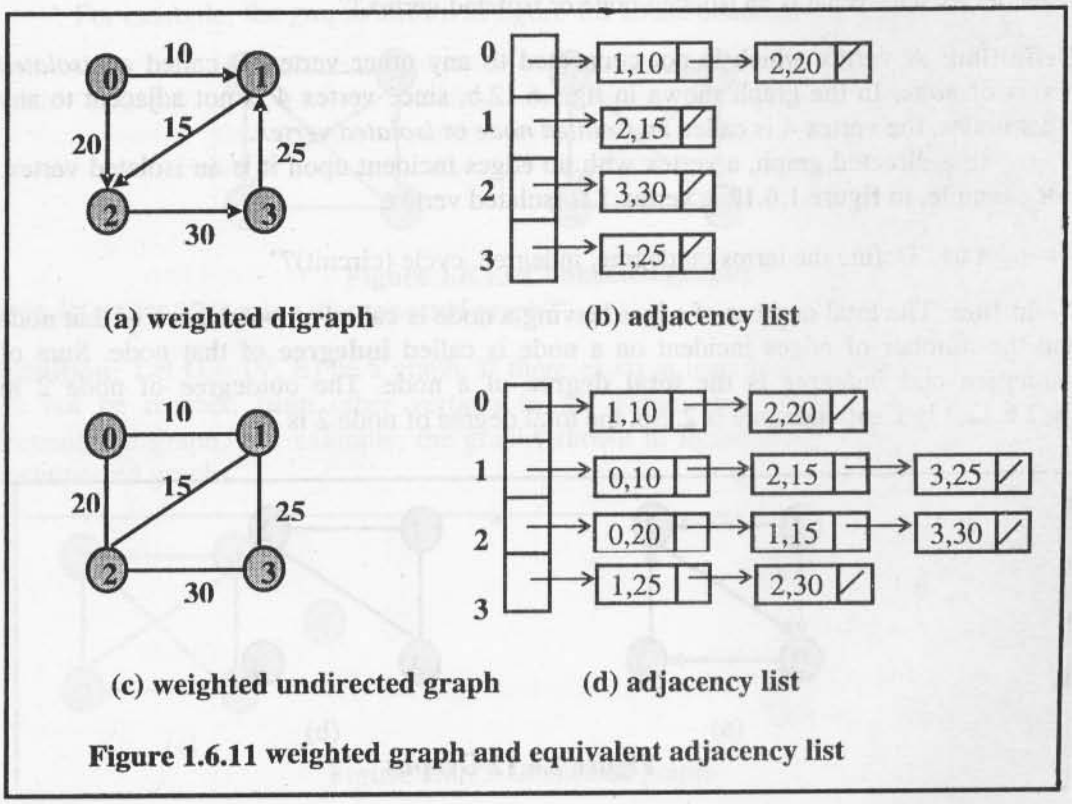


Figure 1.6.11 weighted graph and equivalent adjacency list

Now, let us see "What is a path? What is directed path? What is length of the path?"

Definition: Let G = (V, E) be a graph. A *path* from vertex *u* to vertex *v* in a graph is a sequence of adjacent vertices $<v_0, v_1, v_2, \ldots v_k>$ such that $v_0 = u$ which is the initial vertex and $v_k = v$ which is the terminal vertex. The path contains the vertices $v_0, v_1, v_2, \ldots v_k$ and the edges (v_0, v_1) $(v_1, v_2,), \ldots (v_{k-1}, v_k)$.

If $<v_0, v_1, v_2, \ldots v_k>$ is a sequence of vertices and are distinct and if there is an edge from $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ and all the edges are directed, then the path is called *directed path*.

For example, in the 1.6.11.a, the path from vertex 0 to 3 is given by <0, 1, 2, 3>. Another path from vertex 0 to 3 is given by <0, 2, 3>

Definition: The *length* of the path is the number of edges in the path. If the edges in the path are distinct, then the path is called *simple path*.

For example, the edges appearing in the path from 0 to 3 in the graph shown in figure 1.6.11.a are (0, 1), (1, 2), (2,3). Since 3 edges appear in the path, the length of the path is 3. For the path <0, 2, 3>, the length is 2.

Now, let us see "What is an isolated node or isolated vertex?"

Definition: A vertex which is not connected to any other vertex is called an *isolated vertex* or *node*. In the graph shown in fig.1.6.12.b, since vertex 4 is not adjacent to any other nodes, the vertex 4 is called an *isolated node* or *isolated vertex*.

In a directed graph, a vertex with no edges incident upon it is an isolated vertex. For example, in figure 1.6.12.a, vertex 1 is isolated vertex.

Now, let us "Define the terms outdegree, indegree, cycle (circuit)?"

Definition: The total number of edges leaving a node is called an **outdegree** of that node and the number of edges incident on a node is called **indegree** of that node. Sum of *outdegree* and *indegree* is the **total degree** of a node. The outdegree of node 2 in fig.1.6.12.a is 1 and indegree is 2. So, the total degree of node 2 is 3.

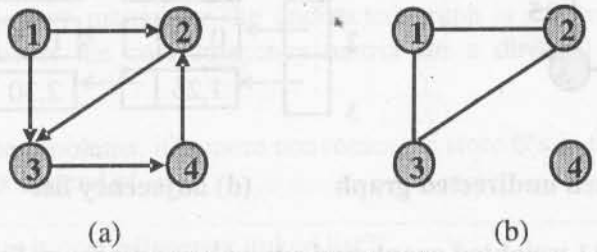


Figure 1.6.12 Graphs

Definition: In a graph if one vertex, say u, is reachable from vertex v and v is reachable from u, then the path from v to v through u or u to u through v is called a **circuit** or a **cycle**. In other words, a cycle is a path in which the first and last vertices are same. There is a cycle

$$C = ((2,3), (3,4) , (4,2))$$

in the graph shown in fig.1.6.12.a.

Note: A graph with at least one cycle is called a cyclic graph and a graph with no cycles is called ***acyclic*** graph. A tree is an acyclic graph. Now, let us see "What is a connected graph?"

Definition: In an undirected graph G, two vertices u and v are said to be connected if there exists a path from u to v. Since G is undirected, there exists a path from v to u also. A graph G (directed or undirected) is said to be connected if and only if there exists a path between every pair of vertices.

For example, the graphs shown in figure 1.6.13 are connected graphs.

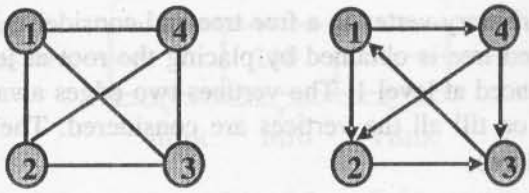


Figure 1.6.13 Connected graphs

Now, let us see "What is a disconnected graph?"

Definition: Let G = (V, E) be a graph. If there exists at least one vertex in a graph that can not be reached from other vertices in the graph, then such a graph is called disconnected graph. For example, the graphs shown in figure 1.6.14.a and 1.6.14.b are disconnected graphs.

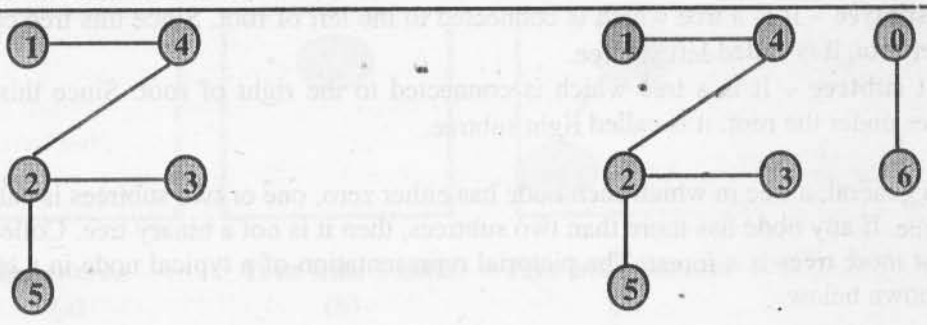
(a) (b)

Figure 1.6.13 Connected graphs

1.7. Trees

This section briefly describes the various definitions associated with respect to trees. First we shall see "What is a tree? What is a forest?"

Definition: A tree which is also called free tree is a connected acyclic graph. A collection of one or more trees is called a forest.



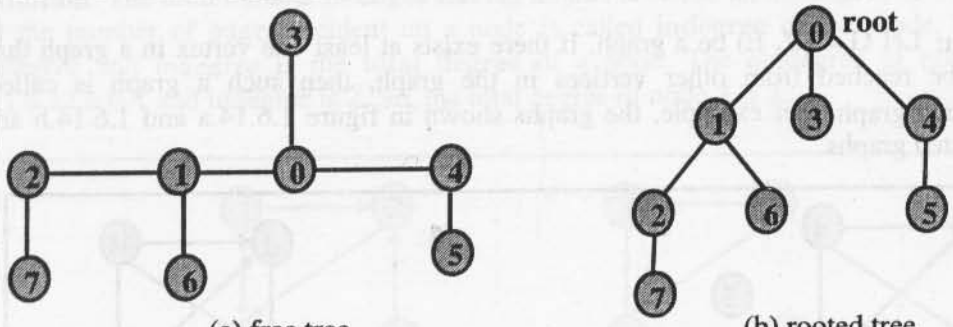
(a) Tree (b) Forest

Fig 1.7.1 Undirected graphs representing tree and forest

Now, let us see "What is a rooted tree?"

Definition: Consider an arbitrary vertex in a free tree and consider this vertex as the root of the rooted tree. A rooted tree is obtained by placing the root at level 0. The vertices adjacent to the root are placed at level 1. The vertices two edges away from the root are placed at level 2 and so on till all the vertices are considered. The resulting tree thus obtained is called a rooted tree.

For example, the figure 1.7.2 shows a free tree and its equivalent rooted tree



(a) free tree (b) rooted tree

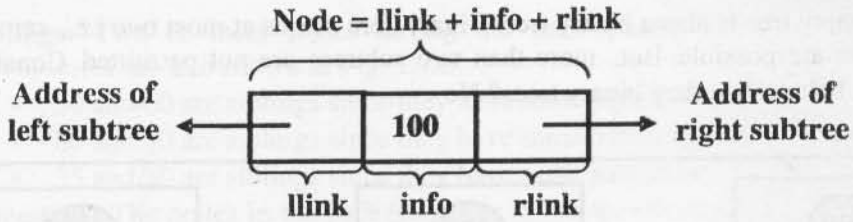
Fig 1.7.2 Free tree to rooted tree

Now, let us see "What is a binary tree?"

Definition: A binary tree is a tree in which each node has at most two children. Each child in the graph is designated as left child or right child. A tree can be empty or partitioned into three subgroups:

- **Root** – If tree is not empty, the first node is called root node.
- **left subtree** – It is a tree which is connected to the left of root. Since this tree comes under root, it is called left subtree.
- **right subtree** – It is a tree which is connected to the right of root. Since this tree comes under the root, it is called right subtree.

Note: In general, a tree in which each node has either zero, one or two subtrees is called a binary tree. If any node has more than two subtrees, then it is not a binary tree. Collection of two or more trees is a forest. The pictorial representation of a typical node in a binary tree is shown below:



Note that a node in a tree consists of three fields namely llink, info and rlink:

- **llink** –This field contains address of left subtree
- **info** – This field is used to store the actual data or information to be manipulated
- **rlink** – This field contains address of right subtree.

The above pictorial representation takes more space. So, to minimize the space, we can represent the above node as shown in figure 1.7.3.

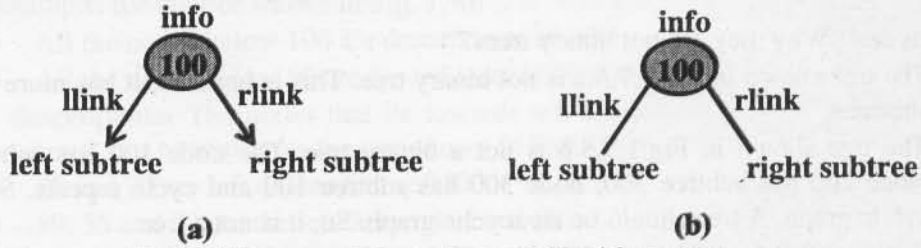


Fig 1.7.3 Representation of a node in a binary tree

The figure 1.7.3.a is shown with directions to left and right subtrees. But, the figure 1.7.3.b shows without directions. In the text book, we show without directions. But, it is implied that directions are present moving away from root. The figure 1.7.4 shows some of the binary trees:

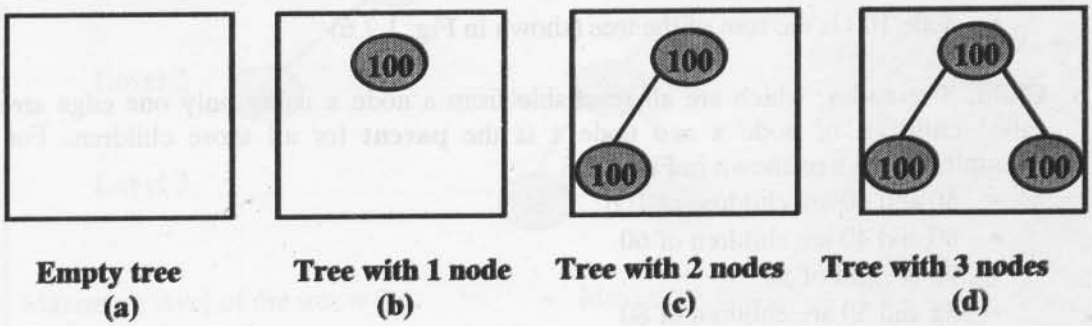


Fig 1.7.4: Binary trees

Note: An empty tree is also a binary tree. Binary here means at most two i.e., zero, one or two subtrees are possible. But, more than two subtrees are not permitted. Consider the trees shown below: Are they binary trees? No.

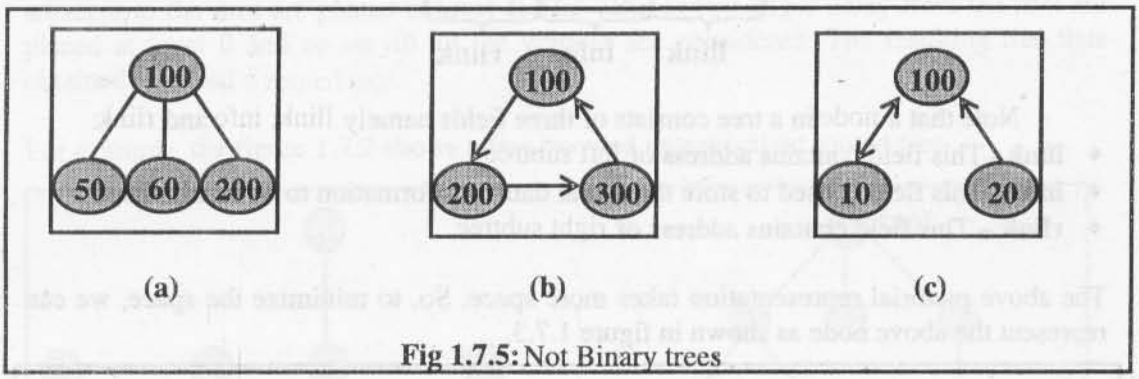


Fig 1.7.5: Not Binary trees

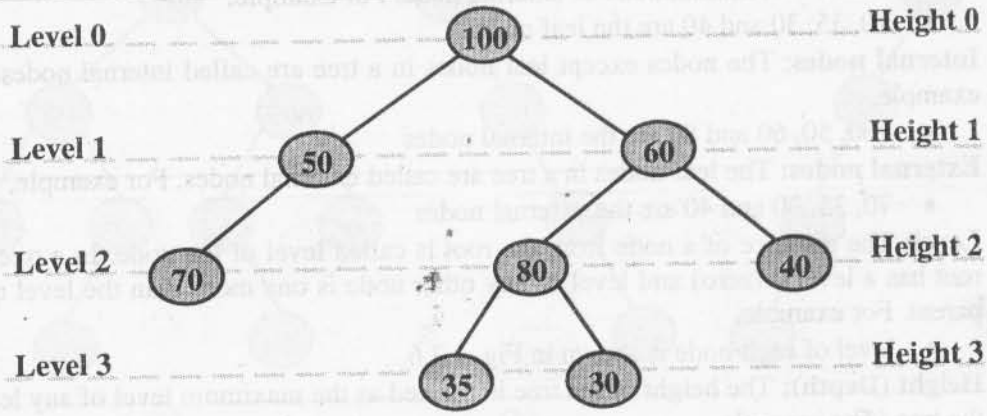
Let us see "Why they are not binary trees?"

- The tree shown in Fig 1.7.5.a is not binary tree. This is because, it has more than two subtrees.
- The tree shown in Fig.1.7.5.b is not a binary tree. The node 100 has subtree 200. Node 200 has subtree 300, node 300 has subtree 100 and cycle repeats. So, it is a cyclic graph. A tree should be an acyclic graph. So, it is not a tree.
- In figure 1.7.5.c, the node 100 has subtree 10 and 10 has the subtree 100. If 10 is subtree of 100, then 100 can not be the subtree of 10. So, it is not a binary tree.

Now; let us see "What are the various terminologies normally associated with trees? Define each of them" The various terminologies that are associated with a tree shown below:

- **Root node:** A node with indegree 0 is called root node. It is the first node in the tree. For example,
  - node 100 is the root of the tree (shown in Fig. 1.7.6)
- **Child:** The nodes, which are all reachable from a node **x** using only one edge are called **children** of node **x** and node **x** is the **parent** for all those children. For example, for the tree shown in Fig. 1.7.6
  - 50 and 60 are children of 100
  - 80 and 40 are children of 60
  - 70 is child of 50
  - 35 and 30 are children of 80

- **Siblings:** Two or more nodes having the same parent are called siblings. For example, for the tree shown in Fig. 1.7.6
  - 50 and 60 are siblings since they have same parent 100
  - 80 and 40 are siblings since they have same parent 60
  - 35 and 30 are siblings since they have same parent 80
- **Ancestors:** The nodes in the path from root to the specified node **x**, are all ancestors of node **x**. For example, for the tree shown in Fig. 1.7.6
  - 100 is the ancestor of 50
  - 60 is the ancestor of 100
  - 50 and 100 are the ancestors of 70
  - 60 and 100 are the ancestors of 80 and 40
  - 80, 60 and 100 are the ancestors of 35 and 30
- **Descendents:** The nodes in the path below the parent are called descendents. In other words, the nodes that are all reachable from a node **x** are all called **descendents** of **x**. For example, for the tree shown in Fig. 1.7.6
  - All the nodes below 100 are descendents of 100
  - All the nodes below 50 are descendents of 50 and so on.
- **Left descendents:** The nodes that lie towards left subtree of node **x** are called left descendents. For example, for the tree shown in Fig. 1.7.6
  - 50 and 70 are left descendents of 100
  - 80, 35 and 30 are the left descendents of 60
  - 35 and 80 are left descendents of 60



Maximum level of the tree = 3

Maximum height/depth of the tree = 3

Fig. 1.7.6: Binary tree

- **Right descendents:** The nodes that lie towards right subtree of node **x** are called right descendents. For example, for the above tree
  - 60, 80, 40, 35 and 30 are right descendents of 100
  - 30 is the right descendent of 80
  - 40 is the right descendent of 60
- **Leftsubtree:** All the nodes that are all left descendents of a node **x** form the **left subtree** of **x**. For example, for the above tree
  - 50 and 70 form the left subtree of 100
  - 80, 35 and 30 form the left subtree of 60
- **Right subtree:** All the nodes that are all right descendents of a node **x** form the **right subtree** of **x**. For example, for the above tree
  - 60, 80, 40, 35 and 30 are right descendents of 100
  - 30 is the right descendent of 80
  - 40 is the right descendent of 60
- **Parent:** A node having left subtree or right subtree or both is said to be a parent node for the left subtree and/or right subtree. For example, for the above tree
  - 100 is the parent for 50 and 60
  - 50 is the parent for 70
  - 60 is the parent for 80 and 40
  - 80 is the parent for 35 and 30
- **Leaf:** A node in a tree that has an outdegree of zero is called a **leaf node.** In other words, a node with an empty left child and an empty right child is called a leaf node. It is also called a terminal node or external node. For example,
  - 70, 35, 30 and 40 are the leaf nodes
- **Internal nodes**: The nodes except leaf nodes in a tree are called internal nodes. For example,
  - 100, 50, 60 and 80 are the internal nodes
- **External nodes:** The leaf nodes in a tree are called external nodes. For example,
  - 70, 35, 30 and 40 are the external nodes
- **Level:** The distance of a node from the root is called level of the node. In a tree, the root has a level 0 (zero) and level of any other node is one more than the level of its parent. For example,
  - level of each node is shown in Fig. 1.7.6
- **Height (Depth):** The height of the tree is defined as the maximum level of any leaf in the tree. For example,
  - height of tree shown in Fig 1.7.6 is 3

Now, let us see "What is an ordered tree?"

Definition: An *ordered tree* is a rooted tree in which the children of each node are ordered from left to right. For example, if a node has *m* children, then this node has first child, second child and so on from left to right. The trees shown in figure 1.7.7 represent different *ordered trees*.

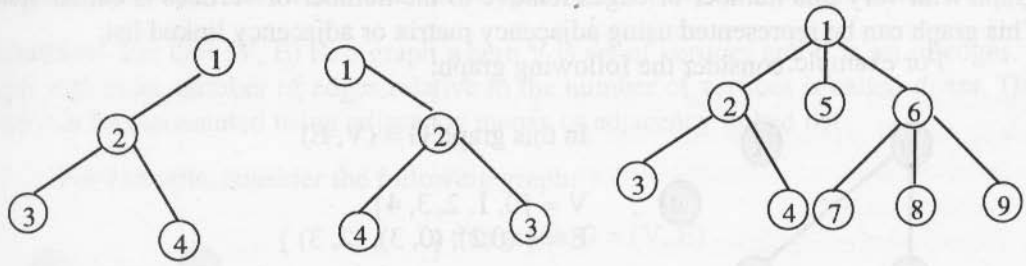


Figure1.7.7 Different ordered trees

Now, let us see "What is a binary search tree?"

Definition: A binary search tree is a binary tree in which for each node say **x** in the tree, elements in the left-subtree are less than **info(x)** and elements in the right subtree are greater or equal to **info(x)**. Every node in the tree should satisfy this condition, if left subtree or right subtree exists. Fig.1.7.6 shows some of the binary search trees.

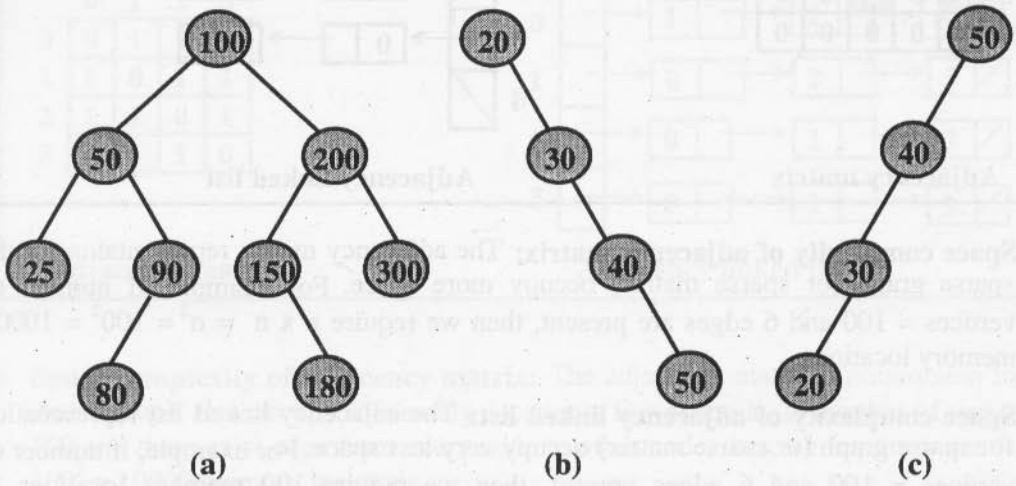
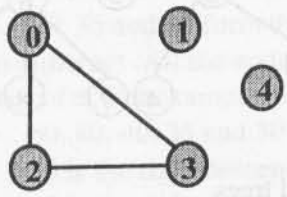


Figure 1.7.6 Binary search trees

Now, let us "Define the term sparse with reference to graph and with example show how this graph can be represented. Comment on space complexity of each representation"

Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges. A graph with very less number of edges relative to the number of vertices is called *sparse*. This graph can be represented using adjacency matrix or adjacency *linked list*.

For example, consider the following graph:



In this graph G = (V, E)

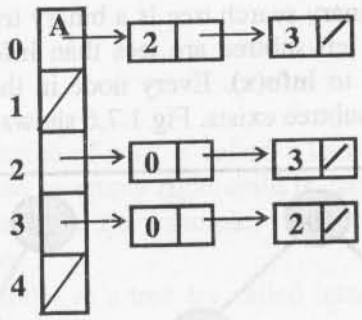
V = {0, 1, 2, 3, 4}
E = { (0,2), (0, 3), (2, 3) }

This graph can be represented using adjacency matrix and adjacency linked list representation as shown below:

A

	0	1	2	3	4
0	0	0	1	1	0
1	0	0	0	0	0
2	1	0	0	1	0
3	1	0	1	0	0
4	0	0	0	0	0

Adjacency matrix



Adjacency linked list

- **Space complexity of adjacency matrix:** The adjacency matrix representaion for the sparse graph (or sparse matrix) occupy more space. For example, if number of vertices = 100 and 6 edges are present, then we require n x n = $n^2 = 100^2 = 10000$ memory locations.

- **Space complexity of adjacency linked list:** The adjacency linked list represenation for sparse graph (or sparse matrix) occupy very less space. For example, if number of vertices = 100 and 6 edges present, then we require 100 memory locations to represent a single dimensional array + memory space for 6 nodes.

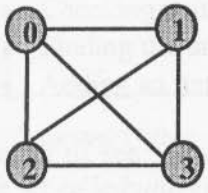
Note: For the sparse graph, if we compare the space with respect to adjacency matrix and adjacency linked list, adjacency linked list representation occupy less space and it is

more suitable whereas adjacency matrix representation occupy more space and it is not suitable.

Now, let us "Define the term dense with reference to graph and with example show how this graph can be represented. Comment on space complexity of each representation"

Definition: Let G = (V, E) be a graph where V is set of vertices and E is set of edges. A graph with more number of edges relative to the number of vertices is called *dense*. This graph can be represented using adjacency matrix or adjacency linked list.

For example, consider the following graph:

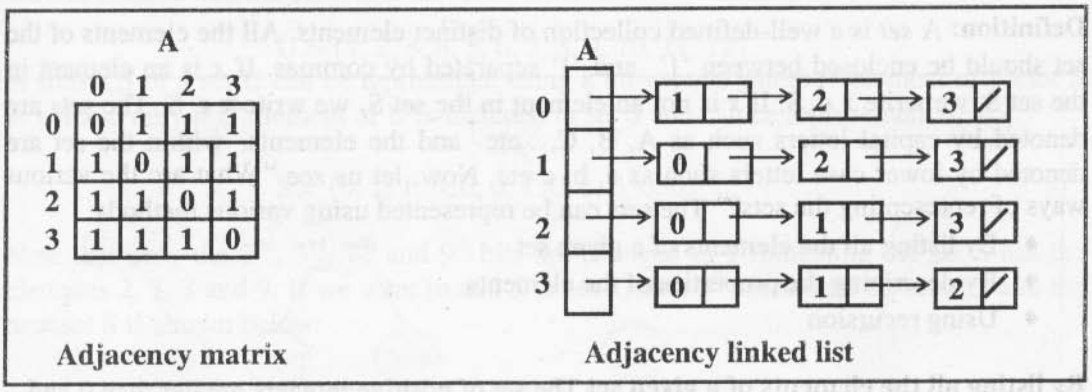


In this graph G = (V, E)

V = {0, 1, 2, 3}

E = { (0,1), (0,2), (0,3), (1,0), (1,2), (1,3), (2,0), (2,1), (2,3), (3,0), (3,1), (3,2) }

This graph can be represented using ajacency matrix and adjacency linked list representation as shown below:



- **Space complexity of adjacency matrix:** The adjacency matrix representaion for the dense graph also occupy n^2 memory locations. For example, if number of vertices = 100 and there exist edge from every node to other node, then we require n x n $= n^2 = 100^2 = 10000$ memory locations.

- **Space complexity of adjacency linked list:** The adjacency linked list represenation for dense graph occupy more space. For example, if number of vertices = 100 and

there exists an edge from each vertex to other vertex, then we will have a linked list consisting of n-1 nodes in each location a[i] for i = 0 to 99.

Note: For the dense graph, we compare the space with respect to adjacency matrix and adjacency linked list, adjacency matrix occupy less space and it is more suitable whereas adjacency linked list representation occupy more space and it is not suitable

Now, let us see "What are the applications of graphs?" Graphs are used for modeling a wide variety of real-applications such as:

- transportation and communication networks (including computer networks and telephone networks)
- Project scheduling and games
- Used to find the shortest distance and path between two points
- Graph coloring, road maps etc.

1.8 Sets

The set notation plays a very important role in mathematics and computer science. Now let us see "What is a set?"

Definition: A *set* is a well-defined collection of distinct elements. All the elements of the set should be enclosed between '{' and '}' separated by commas. If x is an element in the set S, we write $x \in$ S. If x is not an element in the set S, we write $x \notin$ S. The sets are denoted by capital letters such as A, B, C,...etc and the elements within the set are denoted by lower case letters such as a, b, c etc. Now, let us see "What are the various ways of representing the sets?" The sets can be represented using various methods:

- By listing all the elements of a given set
- By describing the properties of the elements
- Using recursion

By listing all the elements of a given set The set of positive integers greater than 0 and less than or equal to 25 which are divisible by 5 can be represented as

S = {5, 10, 15, 20, 25}

The elements of a set can be in any order. The above set can also be written as

S = {10, 20, 5, 25, 15}

Note that no elements are repeated in this set. This representation is useful if the number of elements is less.

By describing the properties of the elements The set of integers greater than 0 and less than or equal to 25 and which are divisible by 5 can be represented as shown below:

$$S = \{5x \mid x \text{ is a positive integer where } 0 < x \leq 5\}$$

Using recursion The set of integers greater than 0 and less than or equal to 25 and which are divisible by 5 can be represented as

$$S = \{a_0 \mid a_i = 5, a_{i+1} = a_i + 5\} \text{ where } a_i <= 20$$

Note: Some of the important set operations are:

- Checking a membership of a given item in a set of items.
- Finding the union of two sets
- Adding an item into a set

Now, let us see "How to represent sets using arrays?" A set which is a subset of a very large set called *universal set*. For example, if we have a set S such that

$$S = \{2, 3, 7, 9\}$$

we can assume that the set S is a subset of set U where

$$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.$$

A subset S of a set U can be represented using a *bit-vector*, where i^{th} element of vector S is 1 if and only if i^{th} element of U is included in set S and can be represented as

$$S = \{0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0\}$$

Note that only the 2nd, 3rd, 7th and 9th bits are changed to 1 indicating the set S has the elements 2, 3, 7 and 9. If we want to add 5 to set S, then 5th bit should be made 1 and the new set S is shown below:

$$S = \{0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0\}$$

Note: We know that a set can be easily represented using a data structure called *array*. The set S can be represented using an array as shown below:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Items	0	1	1	0	1	0	1	0	1	0	0	0

Note: The statement "Add 11 to S" can be implemented using the statement

S[11] = 1

and the resulting array S is shown below:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Items	0	1	1	0	1	0	1	0	1	0	1	0

1.9 Other important algorithms and problems

This section deals with some important algorithms and problems

Example 1.9.1: Which of the following formulas can be considered an algorithm for computing the area of a triangle whose lengths are given positive numbers a, b, and c?

a. S = sqrt(p(p − a)(p − b)(p − c)), where p = (a + b + c)/2
b. S = ½ bc sin A, where A is the angle between sides b and c
c. S = 1/2aha, where ha is the height to base a

Solution: (Part a) The given formula is:

S = sqrt(p(p − a)(p − b)(p − c)), where p = (a + b + c)/2

If we know how to compute the square root of a positive number, the above statement can be considered an algorithm. Otherwise, it cannot be considered an algorithm.

Solution: (Part b) The given formula is:

S = ½ bc sin A

where A is the angle between sides b and c. Note that the values of *b* and *c* are given as specified in the problem. But, angle *A* is not given. So, the above formula cannot be considered as an algorithm.

Solution: (Part c) The given formula is:

$S = 1/2ah_a$

Only the value of *a* is given. But, h_a is not given. So, the above statement cannot be considered as an algorithm.

Example 1.9.2: Write a pseudocode for an algorithm for finding real roots of equation ax2 + bx + c = 0 for arbitrary real coefficients a, b, and c.

Algorithm Quadratic(a, b, c)

Step 1: [Check for zero co-efficients]

if (a = 0 and b $\neq$ *0)*

x = –b / c

print(x)

exit

else if (a = 0 and b = 0)

print ("No solution")

exit

else if (a = 0 and b = 0 and c = 0)

print("No solution")

exit

end if

Step 2: [Find two equal roots]

if ($b^2 - 4ac = 0$) then

x1 = x2 = - b / (2a)

print ("Two equal roots", x1, x2)

stop

end if

Step 3: [Find two distinct roots]

if ($b^2 - 4ac > 0$) then

$$x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

print ("Two distinct roots", x1, x2)

stop

end if

Step 4: [Print complex roots]

print ("roots complex")

Example 1.9.3: Describe the standard algorithm for finding the binary representation of a positive decimal integer

- in English.
- in a pseudocode.

Solution: (Part A): Algorithm to find binary value for a positive decimal integer in English is shown below:

Step 1: Divide the given number n by 2

Step 2: The remainder r_n (0 or 1) will be the next digit of binary representation (from right to left)

Step 3: Replace *n* by the quotient of previous division

Step 4: Repeat through step 1 until n becomes 0

Solution: (Part B): Algorithm to find binary value for a positive decimal integer in pseudocode is shown below:

```
Algorithm Binary(n)
// Purpose : To find binary value for a given positive decimal integer
//Input    : A positive decimal integer n
//Output   : The list b_k b_{k-1}...b_1 b_0 of n's binary digits
    k ← 0
    while (n ≠ 0)
        b_k ← n mod 2
        n ← n/2
        k ← k +1
    end while

    return b
```

Example 1.9.4: Describe the algorithm used by your favorite ATM machine in dispensing cash. (You may give your description in either English or a pseudocode, whichever you find more convenient.)

The following algorithm can be used while dispensing cash in our ATM machine.

Step 1: Prompt the user to insert the code

Step 2: Prompt the user for valid pin number

Step 3: If pin number is invalid terminate the transaction

Step 4: Prompt the user for:

a) Balance checking
b) Deposit
c) Withdrawal

Step 5: If user presses *a* display the balance amount and terminate

Step 6: If user presses *b* insert the envelope into slot provided and terminate

Step 7: If user presses *c* then

prompt the amount to be withdrawn
check for balance
if amount to be withdrawn is less than balance
    get the money
    reduce the balance after withdrawal
else
    prompt "no balance"
end if

Step 8: Terminate

Example 1.9.5: *Can the problem of computing the number π be solved exactly? How* many instances does this problem have?

Solution: Part A: Since π value is 22/7 which is irrational number, its value will not be exact. So, it can be computed only approximately.

Solution: Part B: Note that the value of π is 3.141..... Observe that there are infinite number of digits after decimal point. So, its value can be considered only to a given accuracy. Since any accuracy can be provided, the problem has infinite instances.

Example 1.9.6: Give an example of a problem other than computing the greatest common divisor for which you know more than one algorithm. Which of them is simpler? Which is more efficient?

Solution: Linear search and Binary search.

- **Linear search** is efficient if elements in the list are not arranged in ascending order or descending order and the element is searched less frequently. If frequently an element has to be searched, in such situations, it is better to sort elements using best sorting technique such as quick sort and then search using binary search.
- **Binary search** is efficient if elements in the list are arranged in ascending order

Example 1.9.7: Consider the following algorithm for finding the distance between the two closest elements in an array of numbers.

```
ALGORITHM MinDistance(A[0..n – 1])
//Input: Array A[0..n – 1] of numbers
//Output: Minimum distance between two of its elements
    dmin ← ∞
    for i ← 0 to n – 1 do
        for j ← 0 to n – 1 do
            if (i ≠ j and |A[i] – A[j]| < dmin)
                dmin ← |A[i] – A[j]|
            end if
        end for
    end for
    return dmin
```

Make as many improvements as you can in this algorithmic solution to the problem. (If you need to, you may change the algorithm altogether; if not, improve the implementation given.)

Solution: Note the following points while tracing the above algorithm:

- The value |A[i] – A[j]| is computed in each pass and is compared with *dmin*
- Instead, the value |A[i] – A[j]| can be computed only once, by assuming the elements of array *A* are already sorted (presorted) and still we get the same output.

The following algorithm based on presorting will minimize the number of comparisons and is shown below:

```
ALGORITHM MinDistance(A[0..n – 1])
//Input: Array A[0..n – 1] of numbers
//Output: Minimum distance between two of its elements
    dmin ← ∞
    for i ← 0 to n – 2 do
        for j ← i+1 to n – 1 do
            temp ← |A[i] – A[j]|
            if (temp < dmin)
                dmin ← temp
            end if
        end for
    end for
    return dmin
```

Example 1.9.8: Consider the algorithm for the sorting problem that sorts an array by counting. For each of its elements, the number of smaller elements and then uses this information to put the element in its appropriate position in the sorted array:

```
Algorithm ComparisonCountingSort(A[0..n − 1], S[0..n − 1])
//Purpose: Sorts an array by comparison counting
//Input   : Array A[0..n − 1] of orderable values
//Output  : Array S[0..n − 1] of A's elements sorted in non-decreasing order

for i ← 0 to n − 1 do
    Count[i] ← 0
end for

for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else
            Count[i] ← Count[i] + 1
        end if
    end for
end for

for i ← 0 to n − 1 do
    S[Count[i]] ← A[i]
end for
```

a. Apply this algorithm to sorting the list 60, 35, 81, 98, 14, 47.
b. Is this algorithm stable?
c. Is it in place?

Solution: See section 7.2.1

Example 1.9.9: Design a simple algorithm for the string-matching problem.

Solution: See section 3.5

Example 1.9.10: Describe how one can implement each of the following operations on an array so that the time it takes does not depend on the array's size n.

- Delete the ith element of an array ($1 \le i \le n$).
- Delete the ith element of a sorted array (the remaining array has to stay sorted, of course).

a. Replace the ith element with the last element and decrease the array size by 1 as shown below:

$a[i] \leftarrow a[n]$
$n \leftarrow n - 1$

For example, consider the following array:

a[1]	a[2]	a[3]	a[4]	a[5]	where **n = 5**
10	15	30	5	25	

Assume i = 2. Now, we can replace a[2] by a[5] as shown below:

a[1]	a[2]	a[3]	a[4]	a[5]	where **n = 5**
10	15	30	5	25	

copy

So, the final array, after removing ith element is shown below:

a[1]	a[2]	a[3]	a[4]	a[5]	where **n = 4 (after decrementing)**
10	25	30	5		

Observe that the item 15 is deleted

b. Replace the ith element with a special symbol that cannot be a value of the array's element (e.g., 0 for an array of positive numbers) to mark the ith position as empty. (This method is sometimes called the "lazy deletion".) The equivalent code can be written as:

$A[i] \leftarrow 0$

For example, consider the following array:

a[1]	a[2]	a[3]	a[4]	a[5]	where **n = 5**
10	15	30	5	25	

Assume i = 2. Now, we can replace a[2] by 0 as shown below:

a[1]	a[2]	a[3]	a[4]	a[5]	where n = 5
10	0	30	5	25	

Observe that the item a[2] = 0 indicates that it is deleted

Example 1.9.11: If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for

- Lists represented as arrays.
- Lists represented as linked lists.

Solution: (Part A): Since the elements of the list are arranged in ascending order, we use binary search. The algorithm is shown below:

```
Algorithm binary_search(item, a, n)

//Purpose:  Search for an item in the list identified by a using binary search

//Input:
//        item- the element to be searched
//        a – the list of elements where the searching takes place
//        n – the number of elements in the list

//Output:
//        position is returned if search is successful otherwise, -1 is returned
low ← 0                                  // Points to the first element(page)
high ← n-1                               // Points to the last element(page)

while ( low <= high )
        mid ← ( low + high ) / 2         // Divide the list into two equal parts

        if ( item = a[mid] ) return mid+1   // return the position if search successful

        if ( item < a[mid] )             // Item not found in the middle of the list
                high ← mid – 1           // Search in the left side of the array
        else
                low ← mid + 1            // Search in the right side of the array
        end if
end while

return -1                                // Unsuccessful search
```

Solution: (Part B): When searching in a sorted linked list, stop as soon as search key greater than or equal to the element in the list. If key is equal to the item, we return 1. If key is greater than the element present in the list, we return 0 indicating "Item not found"

```
ALGORITHM search_linked_list(item, first)
// Purpose : To search for the item in the sorted linked list
// Input   : first – contains address of the first node of the list
           : item – element to be searched
// Output  : return 1 if successful
             return 0 if unsuccessful

temp ← first
while (temp ≠ NULL)
        if (item = temp(info)) return 1          // Successful search
        if (item < temp(info)) return 0          // Unsuccessful search
        temp ← temp(link)
end while

return 0                  // Unsuccessful
```

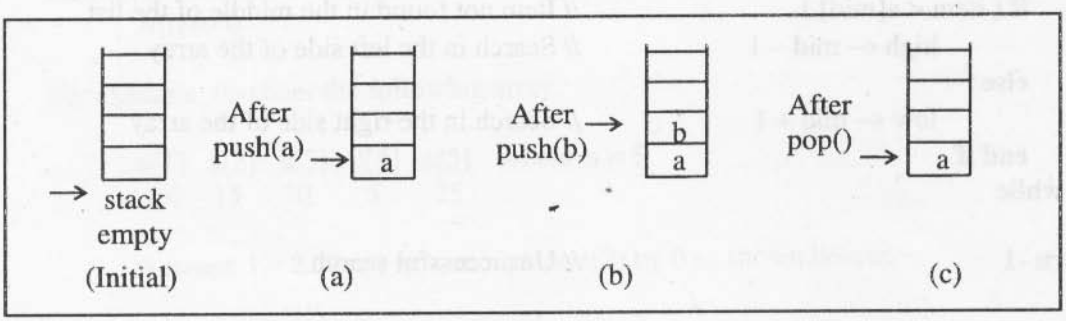
Example 1.9.12: (a) Show the stack after each operation of the following sequence that starts with the empty stack:

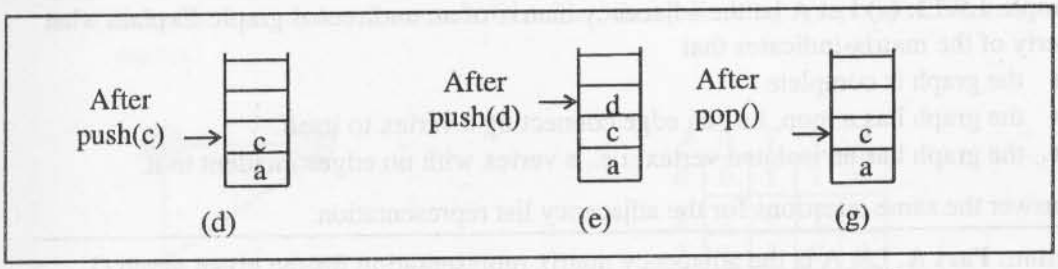
push(a), push(b), pop, push(c), push(d), pop

(b) Show the queue after each operation of the following sequence that starts with the empty queue:

enqueue(a), enqueue(b), dequeue, enqueue(c), enqueue(d), dequeue

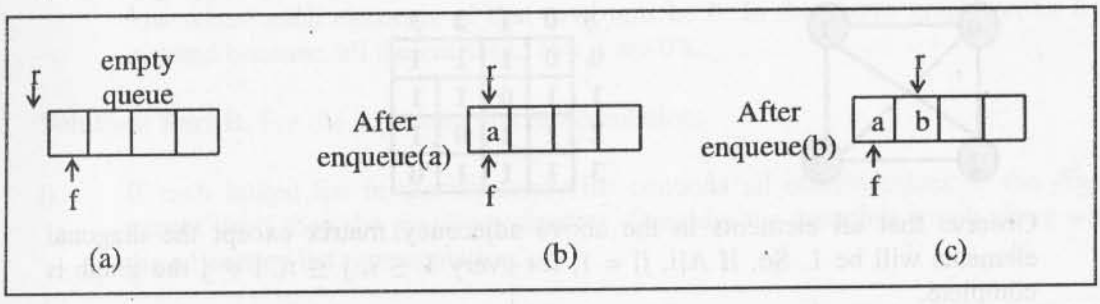
Solution: (Part A) The contents of the stack after each operation with empty stack is shown below:

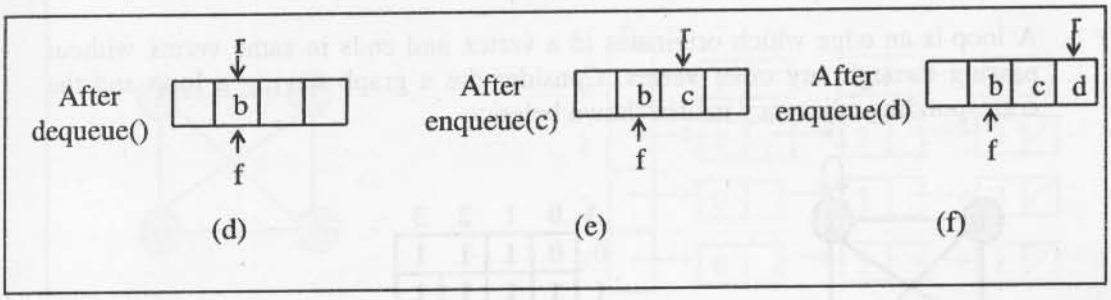


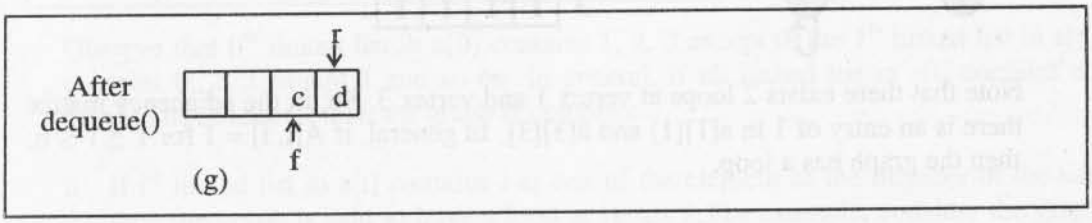


The figure (g) gives the final contents of stack.

Solution: (Part B) The contents of the queue after each operation starting from empty queue is shown below:







Note: The figure (g) gives the final contents of queue.

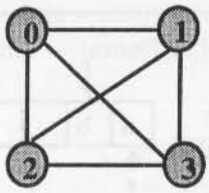
Example 1.9.13: (a) Let A be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that

- the graph is complete.
- the graph has a loop, i.e., an edge connecting a vertex to itself.
- the graph has an isolated vertex, i.e., a vertex with no edges incident to it.

b) Answer the same questions for the adjacency list representation.

Solution: Part A. Let A is the adjacency matrix representation for the given graph G.

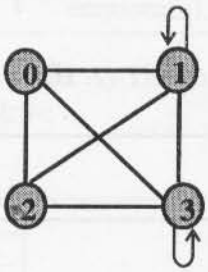
i) A graph is complete if and only if there exists an edge from every vertex to other vertices in the graph. Consider the complete graph along with the adjacency matrix shown below:



A	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

Observe that all elements in the above adjacency matrix except the diagonal elements will be 1. So, if A[i, j] = 1, for every $1 \leq i, j \leq n$, $i \neq j$ the graph is complete.

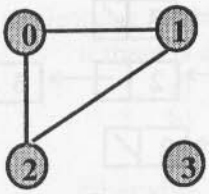
ii) A loop is an edge which originates in a vertex and ends in same vertex without passing through any other vertex. Consider the a graph having a loop and the corresponding adjacency matrix shown below:



A	0	1	2	3
0	0	1	1	1
1	1	1	1	1
2	1	1	0	1
3	1	1	1	1

Note that there exists 2 loops at vertex 1 and vertex 3. So, in the adjacency matrix there is an entry of 1 in a[1][1] and a[3][3]. In general, if A[i, i] = 1 for $1 \leq i \leq n$, then the graph has a loop.

iii) A vertex which does not have any incoming and outgoing edges in a given graph is called an isolated vertex. Consider the following graph and its adjacency matrix:

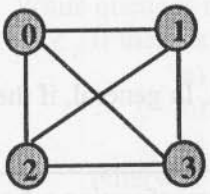


A	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	0
3	0	0	0	0

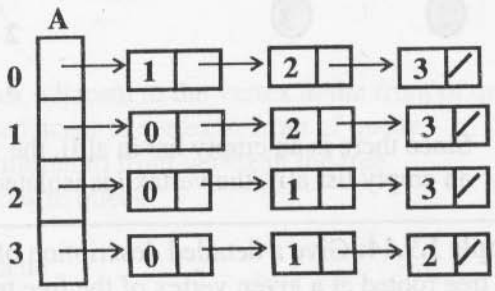
In an adjacency matrix, if row *i* has all 0 entries, then the vertex *i* is isolated i.e., A graph has an isolated vertex if and only if its adjacency matrix has at least one row where each elements of that row must be 0. In the above graph vertex 3 is isolated because, all the entries in row 3 are 0's.

Solution: Part B. For the adjacency list representation:

i) If each linked list in the adjacency list contains all other vertices of the graph except itself then the graph is complete. Consider the complete graph along with the adjacency list representation:



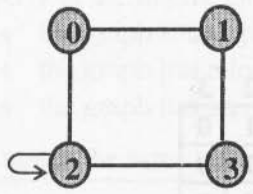
Complete graph



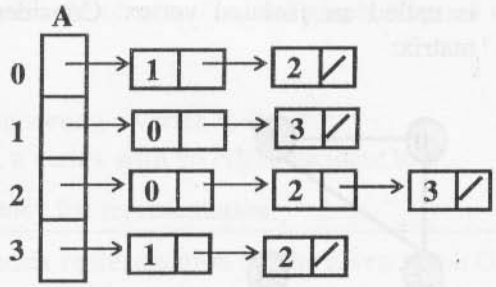
Adjacency linked list

Observe that 0th linked list in a[0] contains 1, 2, 3 except 0, the 1st linked list in a[1] contains 0, 2, 3 except 1 and so on. In general, if ith linked list in a[i] contains all other nodes except *i*, then the graph is complete.

ii) If i^{th} linked list in a[i] contains *i* as one of the element as the member of the list, then the graph is said to have a loop at vertex *i*. For example, consider the graph having a loop along with its linked list representation:



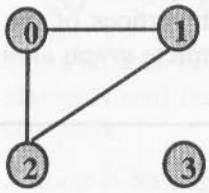
Graph with a loop at vertex 2

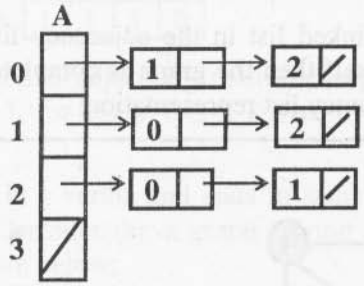


Adjacency linked list

Note that linked list in a[2] corresponding to vertex 2 has 2 as one of the element in the list. This indicates that there is a loop at vertex 2.

iii) If there exists at least one empty list in the adjacency list, then the graph has an isolated vertex. For example, consider a graph with an isolated vertex along with adjacency list representation:





Since there is an empty list in a[3], the vertex 3 is isolated. In general, if there is an empty list a[i], the vertex *i* is isolated.

Example 1.9.14: Give a detailed description of an algorithm for transforming a free tree into a tree rooted at a given vertex of the free tree.

The algorithm for the given problem can be written using one of the two data structures:

- stack
- queue

Algorithm FreeTree_to_RootedTree **using stack**

// Purpose : Given free tree and a vertex of the free tree, convert this free tree into rooted tree using stack

// Input : Free tree and a vertex of free tree

//Output : Rooted tree

Step 1: Make the given vertex as the root of the tree

Step 2: Mark the given vertex

Step 3: Initialize the stack with marked vertex

Step 4: while stack is not empty
if there is an unmarked vertex adjacent to the vertex on the top to the stack
a) mark the former vertex
b) Attach it as a child of the top's vertex in the tree, and push it onto the stack
otherwise
remove a vertex from the stack.
end if
end while

Algorithm FreeTree_to_RootedTree **using queue**
// Purpose : Given free tree and a vertex of the free tree, convert this free tree into rooted tree using queue
// Input : Free tree and a vertex of free tree
//Output : Rooted tree

Step 1: Make the given vertex as the root of the tree

Step 2: Mark the given vertex

Step 3: Initialize queue with marked vertex

Step 4: While queue is not empty
If there is are unmarked vertex adjacent to the vertex at the front of queue
a) mark all vertices adjacent to vertex at front of queue
b) Attach them as children to front vertex in the tree
c) Add all above nodes to queue
otherwise
delete vertex from queue
end if
end while

Example 1.9.15: Prove the inequalities that the height of a binary tree with n vertices:

$$\log_2 n \le h \le n - 1.$$

The height is defined as the length of the longest path from the root of the tree to its leaf. Such path will never include more than n vertices, which is the total number of vertices in the tree. Hence, $h \le n - 1$. .....(1)

The binary tree of height h with the largest number of vertices is the full tree that has all its h + 1 levels filled with the largest number of vertices possible. In such a tree:

Number of nodes at level 0 is $2^0 = 1$
Number of nodes at level 1 is $2^1 = 2$
Number of nodes at level 2 is $2^2 = 4$
.....
.....
In general, number of nodes at level i = 2^i.

But, in an almost complete binary tree, the number of nodes at the last level *i* will be less than 2^i. So, the relationship between the number of nodes and the number of nodes at each level is given by the following inequality:

$$n \le 2^0 + 2^1 + 2^2 + \ldots\ldots 2^i \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

The sum of all the terms in a geometric progression is given by

$$n \le a\left(\frac{r^n - 1}{r - 1}\right)$$ It is a geometric progression with a = 1, r = 2, n = i + 1

$$n \le 1\left(\frac{2^{i+1} - 1}{2 - 1}\right)$$

$$n \le 2^{i+1} - 1$$

$$n \le 2^h - 1$$ since height(T) = Maximum level + 1

$$n + 1 \le 2^h$$

Taking log on both sides we get

$$\log_2 n + 1 \le h * \log_2 2$$

$$\log_2 n + 1 \le h$$

$$h \ge \log_2 n + 1 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

So, using relation (1) and (2) we write $\log_2 n \le h \le n - 1$. Hence, the proof.

Example 1.9.16: Indicate how the ADT priority queue can be implemented as

- an (unsorted) array.
- a sorted array.
- a binary search tree.

The two operations that can be performed in priority queue are: Insertion and Deletion.

Solution: PART A: It is given that array elements are unsorted. In this situation, the priority queue can be implemented using ADT (Abstract Data Type) as shown below:

- Insertion can be implemented by adding the new item at the end of the array
- Deleting can be implemented as shown below:
  - Scan all elements and find the largest.
  - Exchange largest item A[i] with the last element
  - Decrease the array size by 1.

Solution: PART B: It is given that array elements are sorted in ascending order. The priority queue can be implemented using ADT (Abstract Data Type) as shown below:

- Insertion can be implemented as shown below:
  - Scan for the *item* to be inserted from left to right as long as *item* < *A[j]* or the end of the array.
  - If we come across A[j] such that *item* > *A[j]* then
    Move all the elements from A[n-1], A[n-2], ...A[j] to the right by one position
    Insert *item* at position *j*.
    end if
  - If we encounter end of the array, insert the *item* at the end of the array
- Deletion can be implemented as shown below:
  - Largest item is obtained by simply returning value of the last element of sorted array
  - Decrease the size of the array by one.

Solution: PART C: Given a binary search tree, the priority queue can be implemented using ADT (Abstract Data Type) as shown below:

- Insertion can be implemented using standard algorithm (same as inserting an element into binary search tree) as shown below:

```
temp ← getnode()
prev ← NULL
cur ← root
while (cur ≠ NULL)
        prev ← cur
        if item < cur(info)
                cur ← cur(llink)
        else
                cur ← cur(rlink)
        end if
end while
```

```
        if item < prev(info)
                prev(llink) ← temp
        else
                prev(rlink) ←temp
        end if
```

- Deleting can be implemented as shown below:

```
if (root = NULL) tree is empty, return
// Repeatedly move towards right
prev = NULL
cur = root
while (cur(rlink) != NULL)
    prev = cur;
    cur = cur(rlink)
end while

//Right most node has no left child
if cur(llink) = NULL
    prev(rlink) NULL
    return
end if

if (prev = NULL) // If no parent
    root = cur(link)
    free cur
else
    prev(rlink) = cur(llink)
endif
```

Example 1.9.17: How would you implement a dictionary of a reasonably small size n if you knew that all its elements are distinct (e.g., names of 50 states of the United States)? Specify an implementation of each dictionary operation.

Solution: This can be implemented using a *bit vector* i.e., consider an array of *n* bits in which i^{th} bit is 1 if i^{th} element is present in the dictionary and 0 otherwise. Insertion, deletion and search operations can be performed as shown below:

- **Insertion:** To insert an item at i^{th} location, make i^{th} bit as 1.
- **Deletion:** To delete an item at i^{th} location, make i^{th} bit as 0.
- **Searching:** Scan for i^{th} element. If i^{th} bit is 0, item not found else item found

Example 1.9.18: For each of the following applications, indicate the most appropriate data structure:

(a) Answering telephone calls in the order of their known priorities.
(b) Sending backlog orders to customers in the order they have been received.
(c) Implementing a calculator for computing simple arithmetical expressions.

Solution: Part A : Priority queue
Part B : Queue
Part C : Stack

Example 1.9.19: Anagram checking: Design an algorithm for checking whether two given words are anagrams, i.e., whether one word can be obtained by permuting the letters of the other. (For example, the words tea and eat are anagrams.)

ALGORITHM ANAGRAM
// Purpose : To check whether two strings are anagrams or not
// Output : Anagram if one string is obtained from other by rearranging
Not an anagram otherwise

Step 1: Read the first word	// str1 = tea	str1 = ab
Step 2: Read the second word	// str2 = eat	str2 = aba
Step 3: Sort(first word)	// str1 = aet	str1 = ab
Step 4: Sort(second word)	// str2 = aet	str2 = aab
Step 5: if (first word = second word)	// **Output**	**Output**
print "Anagram"	// Anagram	Not anagram
else		
print "Not anagram"		
end if		

Exercises:

1. What is an algorithm? What are the notations used? Explain with an example
2. What are the various properties of an algorithm?
3. Write an algorithm to compute gcd of two numbers using Euclid's algorithm (using mod operation)
4. Write an algorithm to compute gcd of two numbers using Euclid's algorithm (using repetitive subtraction)
5. Explain the procedure of generating prime numbers using the method "Sieve of Eratosthenes" and write the algorithm for the same
6. What are the various steps involved in program development? Explain briefly.

7. With the help of a flow chart, explain the various stages of algorithm design and analysis process. **(Jan/Feb 2005)**
8. What is the difference between pseudo code and algorithm
9. What is an array and what are its limitations and how to overcome these limitations?
10. What is a singly linked list? Explain with an example.
11. What are the disadvantages of singly linked lists?
12. What is a doubly linked list?
13. What is a stack and what are the various operations that can be performed on stacks?
14. What is a queue and what are the various operations that can be performed on queues?
15. What is a graph? What are the various methods used to represent the graphs?
16. Define the following:

    a. Diagraph b. undirected graph c. simple graph d. Complete graph
    e. indegree f. outdegree g. Isolated vertex
17. Explain adjacency matrix and adjacency list with examples.
18. Distinguish between the two common ways to represent a graph. Given the representation of an undirected graph, explain how the following can be ascertained by the representation:

    a. The graph is complete
    b. The graph has a loop
    c. The graph has an insolated vertex

    answer for each of the representation separately **(Jan/Feb 2005)**
19. What is a weighted graph and how it can be represented?
20. Define the following terms:

    a. Path b. length c. reachable d. connected
    e. Acyclic graph f. directed tree g. terminal node h. free tree
    i. Forest j. rooted tree k. external nodes l. internal nodes
    m. level of a tree n. Height of a tree o. Depth of a node p. binary tree
    q. Ordered tree
21. What is a set? How sets can be implemented?
22. With the help of a neat flow diagram, explain a typical sequence of steps in designing and analyzing an algorithm
23. Suggest an algorithm that transforms a free tree into a rooted tree at a given vertex of the free tree

Chapter 2: Analysis of algorithm efficiency

What are we studying in this chapter?

- Analysis Framework
  - Measuring the size of the input
  - Units for measuring running time
  - Orders of growth
  - Worst-case, Best case and average case efficiencies
- Asymptotic Notations and Basic Efficiency Classes
  - Big-Oh(O)
  - Big-Omega(Ω)
  - Big-Theta(θ)
  - Useful property involving the asymptotic notations
  - Using limits for comparing orders of growth
- Mathematical Analysis of Non-recursive Algorithms
- Mathematical analysis of Recursive Algorithms
- Example – Fibonacci Numbers

- 6 hours

2.1 Analysis framework

The main purpose of algorithm analysis is to design most efficient algorithms. Let us see "On what factors efficiency of algorithm depends?" The efficiency of an algorithm depends on two factors:

- → Space efficiency
- → Time efficiency

Now, let us see "What is space efficiency and on what factors space efficiency depends on?"

Definition: The space efficiency of an algorithm is the amount of memory required to run the program completely and efficiently. If the efficiency is measured with respect to the space (memory required), the word space complexity is often used. The space complexity of an algorithm depends on following factors:

Components that affect space efficiency →
- Program space
- Data space
- Stack space

- **Program space:** The space required for storing the machine program generated by the compiler or assembler is called program space.
- **Data space:** The space required to store the constants, variables etc., is called data space
- **Stack space:** The space required to store the return address along with parameters that are passed to the function, local variables etc., is called stack space.

Note: The new technological innovations have improved the computer's speed and memory size by many orders of magnitude. Now a days, space requirement for an algorithm is not a concern and hence, we are not concentrating on space efficiency. Let us concentrate only on time efficiency.

Let us see "What is time efficiency?"

Definition: The time efficiency of an algorithm is measured purely on how fast a given algorithm is executed. Since the efficiency of an algorithm is measured using time, the word time complexity is often associated with an algorithm.

Now, let us see "On what factors the time efficiency of an algorithms depends on?" The time efficiency of the algorithm depends on various factors as shown below:

Components that affect time efficiency
- → Speed of the computer
- → Choice of the programming language
- → Compiler used
- → Choice of the algorithm
- → Number (Size) of inputs/Outputs

Since we do not have any control over speed of the computer, programming language and compiler, let us concentrate only on next two factors such as:

- Choice of an algorithm
- Number (size) of inputs

Note: Many algorithms use n as the parameter to find the order of growth. The parameter n may indicate the number of inputs or size of inputs. Most of the time, the value of n is directly proportional to the size of data to be processed. This is because, almost all algorithms run longer on larger inputs.

For example, it takes longer time to sort larger arrays and longer time to search the larger arrays. So, the time efficiency of an algorithm depends on size of the input n and hence time efficiency is always expressed in terms of n.

Before proceeding further, let us see "What is basic operation?"

Definition: The operation that contributes most towards the running time of the algorithm is called basic operation. A statement that executes maximum number of times in a function is also called basic operation. The number of times basic operation is executed depends on size of the input. The basic operation is the most time consuming operation in the algorithm. For example:

- a statement present in innermost loop in the algorithm
- addition operation while adding two matrices, since it is present in innermost loop
- multiplication operation in matrix multiplication since it is present in innermost loop

Now, let us see "How to compute the running time of an algorithm using basic operation or How time efficiency is analyzed?" The time efficiency is analyzed by determining the number of times the basic operation is executed. The running time T(n) is given by:

$$T(n) \approx b * C(n)$$

where

- T is the running time of the algorithm
- n is the size of the input
- b execution time for basic operation.
- C represent number of times the basic operation is executed

For example, the following table shows the problem, the input size measure and basic operation that is being performed in the corresponding algorithm:

Problem	measure of input size	Basic operation
Search for a key in an array	n-number of elements	Key comparison
Sorting using bubble sort	n – number of elements	Comparison of successive elements
Matrix multiplication	m – Number of rows n – Number of columns	Multiplication operation
Graph problems	number of vertices	Visiting each vertex or traversing an edge

2.2 Orders of growth

Now, let us see "What is order of growth? For what values of n we find the order of growth?"

Definition: We expect the algorithms to work faster for all values of n. Some algorithms execute faster for smaller values of n. But, as the value of n increases, they tend to be very slow. So, the behavior of some algorithm changes with increase in value of n. This change in behavior of the algorithm and algorithm's efficiency can be analyzed by considering the highest order of n. The order of growth is normally determined for larger values of n for the following reasons:

- The behavior of algorithm changes as the value of n increases
- In real time applications we normally encounter large values of n

For example, the order of growth with respect to two running times is shown below:

- Suppose $T(n) \approx c * C(n)$. Observe that T(n) varies linearly with increase or decrease in the value of n. In this case, the order of growth is linear.
- Suppose $T(n) \approx c * C(n^2)$. In this context, the order of growth is quadratic.

Note: If the order of growth of one algorithm is linear and the order of growth of second algorithm to solve the same problem is quadratic, then it clearly indicates that running time of first algorithm is less and it is more efficient. So, while analyzing the time efficiency of an algorithm, the order of growth of n is important. Let us discuss orders of growth in the next section.

The concept of order of growth can be clearly understood by considering the common computing time functions shown in table 2.2.

As N increases order of growth increases

N	log N	N	N log N	N^2	N^3	2^N	N!
1	0	1	0	1	1	2	1
2	1	2	2	4	8	4	2
4	2	4	8	16	64	16	24
8	3	8	24	64	512	256	40320
16	4	16	64	256	4096	65536	high
32	5	32	160	1024	32768	4294967296	very high

Fig 2.2 Values of some of the functions

Note: By comparing N (which is linear) and 2^N(which is exponential) it is observed from the above table that exponential function grows very fast even for small variation of N. So, an algorithm with linear running time is preferred over an algorithm with exponential running time. The basic efficiency asymptotic classes are shown below:

- **1:** Indicates that running time of a program is constant.
- log N: Indicates that running time of a program is logarithmic. This running time occurs in programs that solve larger problems by reducing the problem size by a constant factor at each iteration of the loop (For example, binary search).
- N: Indicates that running time of a program is linear. So, when N is 1000, the running time is 1000 units. When N is doubled, so does the running time. (For example, linear search)
- N log N: Indicates that running time of a program is N log N (For lack of better adjective, it is used as it is instead of linear, quadratic etc). The algorithms to sort elements in ascending order such as quick sort, merge sort and heap sort will have this running time. (These sorting techniques are discussed in later chapters)
- N^2: Indicates that running time of a program is quadratic. The algorithms normally will have two loops. For example, sorting algorithms such as bubble sort, selection sort, addition and subtraction of two matrices have this running time.
- N^3: Indicates that running time of a program is cubic. The algorithms with running time will have three loops. For example, matrix multiplication, algorithm to solve simultaneous equations using gauss-elimination method will have this running time.
- 2^N : Indicates that running time of a algorithm is exponential. The tower of Hanoi problem and algorithms that generate subsets of a given set will have this running time.
- N!: Indicates that running time of an algorithm is factorial. The algorithms that generate all permutations of set will have this running time.

Note: All the above functions can be ordered according to their order of growth (from lowest to highest) as shown below:

$$\mathbf{1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!}$$

Note: Even though running times of 2^N and N! are different, normally both classes are considered as exponential. The algorithms with this running time can be solved practically for smaller values of n.

Note: For a very large value of n, the exponential function 2^n and factorial function n! generate a very high value such that even the fastest computer can take years to execute an algorithm.

For example, a computer which executes 10^{12} instructions per second takes $4*10^{10}$ years to execute 2^{100} operations. The factorial function n! is much more than the value specified for 2^n.

2.3 Worst-case, Best-case and average case efficiencies

Now the question is "Will the algorithm efficiency depends on the size of algorithm's input alone?" For some of the problems, the time complexity will not depend on the number of inputs alone. For example, while searching for a specific item in an array of n elements using linear search, we have following three situations:

- An item we are searching for may be present in the very first location itself. In this case only one item is compared and this is the best case.
- The item may be present some where in the middle which definitely takes some time. Running time is more when compared to the previous case for the same value of n. Since we do not know where the item is we have to consider the average number of cases and hence this situation is an average case.
- The item we are searching for may not be present in the array requiring n number of comparisons and running time is more than the previous two cases. This may be considered as the worst case.

So, knowing n alone itself is not enough to estimate the run time of an algorithm or a function. In such cases, we may have to find the worst-case efficiency, best case efficiency and average case efficiency. Now, let us see "What is worst case efficiency?"

Definition: The efficiency of an algorithm for the input of size n for which the algorithm takes longest time to execute among all possible inputs is called worst case efficiency.

For example, if we use linear search and an item we search for is not present in the array is an example of worst case efficiency. In the worst case, the algorithm runs for the longest duration among all possible inputs for a given size. Here, maximum number of steps are executed for that input.

Now, let us see "What is best case efficiency?"

Definition: The efficiency of an algorithm for the input of size n for which the algorithm takes least time during execution among all possible inputs of that size is called best case efficiency.

In the best case, the algorithm runs fastest for the input specified. For example, in linear search, if the item to be searched is present in the beginning, is an example of the best case efficiency.

Note: In the average case efficiency, the average number of basic operations executed will be considered. This is required only for the randomized input.

Important problems

Example 2.1.1: For each of the following algorithms, indicate (i) a natural size metric for its inputs; (ii) its basic operation; (iii) whether the basic operation count can be different for inputs of the same size:

a) computing the sum of n numbers
b) computing n!
c) finding the largest element in a list of n numbers
d) Euclid's algorithm
e) sieve of Eratosthenes
f) pen-and-pencil algorithm for multiplying two n-digit decimal integers

Solution: For algorithms shown: natural size metric for its inputs, its basic operation and whether the basic operation can be different for inputs of same size is shown below:

Algorithm	Natural size metric for inputs	Basic operation	Whether basic operation count can be different for inputs of same size
Compute sum of n numbers	n	addition of two numbers	no
Computing n!	n	multiplication of two integers	no
Find largest of n numbers	n	comparison of two integers	no
Euclid's algorithm	magnitude of larger/smaller of two inputs or the sum of the magnitudes of two input numbers	modulo division	yes
Sieve of Erathosthenes	magnitude of n	eliminating a number from the list of remaining candidates	no
Multiply two n-digit decimal integers(Pen and pencil algorithm)	n	multiplication of 2 digits	no

Example 2.1.2: a) Consider the definition-based algorithm for adding two n-by-n matrices. What is its basic operation? How many times is it performed as a function of the matrix order n? As a function of the total number of elements in the input matrices?

b) Answer the same questions for the definition-based algorithm for matrix multiplication.

Solution: Part A: Consider the algorithm for adding two nxn matrices:

```
ALGORITHM: Matrix_Addition
for i ← 0 to n-1
    for j ← 0 to n -1
        c[i][j] ← a[i][j] + b[i][j]   ⇨ Basic operation Peformed n^2 times
    end for
end for
```

i) Basic operation: Addition of two numbers

ii) Basic operation is performed: n^2 times

The total number of elements in two given matrices is $N = 2n^2$

From above equation, we get $n^2 = \frac{N}{2}$

So, the total number of additions = $n^2 = \frac{N}{2}$

Solution: Part B: The algorithm for multiplication of two nxn matrices is:

```
ALGORITHM: Matrix_Addition
for i ← 0 to n-1
    for j ← 0 to n -1
        sum ← 0
        for k ← 0 to n-1
            sum ← sum + a[i][k] * b[k][j]   ⇨ Basic operation Performed n^3 times
        end for
        c[i][j] ← sum
    end for
end for
```

i) Basic operation: Multiplication of two numbers

ii) Basic operation is performed: n^3 times

The total number of elements in two given matrices is $N = 2n^2$

$$\text{i.e., } n^2 = \frac{N}{2} \quad \ldots\ldots\ldots\ldots\ldots\ldots(1)$$

$$\text{i.e., } n = \sqrt{\frac{N}{2}} = \left(\frac{N}{2}\right)^{\frac{1}{2}} \quad \ldots\ldots(2)$$

$$\text{Total number of multiplications} = n^3 = n * n^2 = \left(\frac{N}{2}\right)^{\frac{1}{2}} * n^2 \quad \text{(Substituting eq. 2)}$$

$$= \left(\frac{N}{2}\right)^{\frac{1}{2}} * \frac{N}{2} \quad \text{(Substituting eq. 1)}$$

$$= \left(\frac{N}{2}\right)^{\frac{3}{2}}$$

So, the total number of multiplications = $\boxed{n^3 = \left(\frac{N}{2}\right)^{\frac{3}{2}}}$

2.1.3: Consider a variation of sequential search that scans a list to return the number of occurrences of a given search key in the list. Will its efficiency differ from the efficiency of classic sequential search?

Solution: Consider the classic sequential search algorithm

```
ALGORITHM search1
for i ← 0 to n – 1
    if (item = a[i]) return i        (basic operation) best case : once
end for                                                worst case : n times
return - 1
```

2.10 Analysis of algorithm efficiency

Basic operation: comparison statement

Best case: comparison statement is executed once

Worst case: comparison statement is executed n times

So, in classic sequential search : number of comparisons range from 1 to n

Consider a variation of sequential search that scans a list to return the number of occurrences of a given search key in the list

```
ALGORITHM search2
count ← 0
for i ← 0 to n – 1
    if (item = a[i]) count ← count+1
end for
return count
```

(basic operation) best case and worst case : n times

So, in above algorithm : number of comparisons is always n

2.1.4: For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold.

a) $\log_2 n$
b) $\sqrt{n}$
c) n
d) n^2
e) n^3
f) 2^n

Solution: PART A: Given relation: $\log_2 n$ .....(1)

The relation if its argument is increased fourfold : $\log_2 4n$ .....(2)

The function's value that is changed can be obtained by dividing 4n by n and taking the log value as shown below:

$$= \log_2\left(\frac{4n}{n}\right)$$

$$= \log_2 4 = 2$$

So, the functions's value is increased by 2

Solution: PART B: Given relation: $\sqrt{n}$ .....(1)
The relation if its argument is increased fourfold : $\sqrt{4n}$ .....(2)
The function's value that is changed can be obtained by dividing relation 2 by relation 1:

$$= \frac{\sqrt{4n}}{\sqrt{n}}$$

$$= \frac{\sqrt{4} * \sqrt{n}}{\sqrt{n}}$$

$$= \sqrt{4} = 2$$

So, the functions's value is increased by 2

Solution: PART c: Given relation: n.....(1)
The relation if its argument is increased fourfold : 4n.....(2)
The function's value that is changed can be obtained by dividing relation 2 by relation 1:

$$= \frac{4n}{n} = 4$$

So, the functions's value is increased by 4

Solution: PART d: Given relation: n^2.....(1)
The relation if its argument is increased fourfold : $(4n)^2$.....(2)
The function's value that is changed can be obtained by dividing relation 2 by relation 1:

$$= \frac{(4n)^2}{n^2} = 4^2$$

So, the functions's value is increased by 4^2

Solution: PART e: Given relation: n^3.....(1)
The relation if its argument is increased fourfold : $(4n)^3$.....(2)
The function's value that is changed can be obtained by dividing relation 2 by relation 1:

$$= \frac{(4n)^3}{n^3} = 4^3$$

So, the functions's value is increased by 4^3

Solution: PART f: Given relation: 2^n.....(1)
The relation if its argument is increased fourfold : $(2)^{4n}$.....(2)

The function's value that is changed can be obtained by dividing relation 2 by relation 1:

$$= \frac{(2)^{4n}}{2^n} = 2^{4n-n} = 2^{3n} = \left(2^n\right)^3$$

So, the functions's value is increased by $\left(2^n\right)^3$

2.1.5: Indicate whether the first function of each of the following pairs has a smaller, same, or larger order of growth (to within a constant multiple) than the second function.

a) $n(n+1)$ and $2000n^2$
b) $100n^2$ and $0.01n^3$
c) $\log_2 n$ and $\ln n$
d) $\log_2^2 n$ and $\log_2 n^2$
e) 2^{n-1} and 2^n
f) $(n-1)!$ and $n!$

Solution: PART a:

Given: $n(n+1)$ and $2000n^2$

$\Downarrow$ $\Downarrow$

$n^2 + n$ and $2000n^2$

order $\Downarrow$ $\Downarrow$ order

2 2

$\Downarrow$ $\Downarrow$

both have same order

$n(n+1)$ has same order of growth as that of $2000n^2$

Solution: PART b:

Given: $100n^2$ and $0.01n^3$

order $\Downarrow$ $\Downarrow$ order

2 3

$\Downarrow$ $\Downarrow$

2 is less than 3

$100n^2$ has lower order of growth than $2000n^2$

Solution: PART c:

Given: $\log_2 n$ and $\ln n$ **Note:** ln n is natural logarithm (read as lawn n)

We know that:

$$\log_2 n = \frac{\log_e n}{\log_e 2}$$

$$\log_2 n = \frac{\ln n}{\log_e 2} \quad \text{since } \log_e n = \ln n$$

i.e., $\ln n = \log_e 2 * \log_2 n$

So, $\ln n \approx \log_2 n$ (since $\log_e 2$ is a constant multiple)

This shows that all logarithmic functions have the same order of growth within a constant multiple. So,

$\boxed{\log_2 n \text{ has same order of growth as that of } \ln n}$

Solution: PART d:

→ read as log square of n to base 2

→ read as log 'n^2' to base 2

Given: $\log_2^2 n$ and $\log_2 n^2$

⇩ ⇩ can be written as

$\log_2 n * \log_2 n$ $2 \log_2 n$

⇩ ⇩

$\log_2 n * \log_2 n >> 2 \log_2 n$ for very large value of n

So, $\log_2 n * \log_2 n$ has higher order of growth than $2 \log_2 n$

⇩ ⇩

$\log_2^2 n$ has higher order of growth than $\log_2 n^2$

$\boxed{\text{So, } \log_2^2 n \text{ has higher order of growth than } \log_2 n^2}$

Solution: PART e:

Given: 2^{n-1} and 2^n

$\Downarrow$ $\Downarrow$

$\frac{2^n}{2}$ and 2^n

order $\Downarrow$ $\Downarrow$ order

2^n and 2^n

Since 2^n is same as 2^n, we say that

$\boxed{\text{So, } 2^{n-1} \text{ has same order of growth as } 2^n}$

Solution: PART f:

(n – 1)! and n!

Given: (n-1)! and n!

$\Downarrow$ $\Downarrow$

(n-1)! and n*(n-1)!

$\Downarrow$

(n-1)! is less than n*(n-1)!

$\boxed{\text{So, (n-1)! has lower order of growth than n!}}$

2.4 Asymptotic notations

The order of growth can be expressed using two methods:

- Order of growth using asymptotic notations
- Order of growth using limits

Before proceeding further, let us see "What do you mean by asymptotic behavior of a function?"

Definition: The value of the function may increase or decrease as the value of n increases. Based on the order of growth of n, the behavior of the function varies. Asymptotic notations are the notations using which two algorithms can be compared with respect to efficiency based on the order of growth of an algorithm's basic operation.

Now, let us see "What are the different types of asymptotic notations?" The different types of asymptotic notations are shown below:

Asymptotic notations → O(Big Oh)
→ Ω(Big Omega)
→ θ (Big Theta)

Now, let us see the informal definition and formal definition of above asymptotic notations

2.4.1. Informal Definitions of Asymptotic notations

In this section, let us see "Give informal definitions of asymptotic notations" The first asymptotic notation is Big Oh (O). "What is Big Oh (O) notation?"

Definition: Assuming n indicates the size of input and g(n) is a function, informally O(g(n)) is defined as set of functions with a small or same order of growth as g(n)) as n goes to infinity.

Example 1: Let $g(n) = n^3$. Since n and n^2 have smaller order of growth and n^3 has same order of growth when compared to n^3, we say

$$\boxed{\begin{array}{l} n \in O(n^3) \\ n^2 \in O(n^3) \\ n^3 \in O(n^3) \end{array}}$$

Example 2: Let $g(n) = n^3$. Since n^4, $0.001n^4$, n^5+n+3 do not have smaller order or same order as that of n^3, they do not belong to O(g(n)) and can be represented as shown:

$$\boxed{\begin{array}{ll} n^4 & \notin O(n^3) \\ 0.001n^4 & \notin O(n^3) \\ n^5+n+3 & \notin O(n^3) \end{array}}$$

Now, let us see "What is big Omega (Ω) notation?"

Definition: Assuming n indicates the size of input and g(n) is a function, informally Ω(g(n)) is defined as set of functions with a larger or same order of growth as g(n)) as n goes to infinity.

Example 1: Let $g(n) = n^3$. Note that n^3, n^4, $0.001n^4$, n^5+n+3 have larger order or same order as that of n^3. Hence, they belong to $\Omega(g(n))$ and can be represented as shown:

$$\boxed{\begin{array}{ll} n^3 & \in \Omega(n^3) \\ n^4 & \in \Omega(n^3) \\ 0.001n^4 & \in \Omega(n^3) \\ n^5+n+3 & \in \Omega(n^3) \end{array}}$$

Example 2: Let $g(n) = n^3$. Since n and n^2 do not have larger order or same order of growth when compared to n^3, we say

$$\boxed{\begin{array}{l} n \notin \Omega(n^3) \\ n^2 \notin \Omega(n^3) \end{array}}$$

Example 3: Let $g(n) = n^2$. What is the relation between n^3, $\frac{1}{2}(n)(n-1)$ and $100n + 5$ using Ω-notation?

Solution: It is given that $g(n) = n^2$ and we need to find the relation between:

a) n^3 and n^2
b) $\frac{1}{2}(n)(n-1)$ and n^2
c) $100n+5$ and n^2

a) $\underset{n^3}{n^3}$ and $\underset{>}{}$ $\underset{n^2}{n^2}$

$$\boxed{\text{So, } n^3 \in \Omega(n^2)}$$

b) $\frac{1}{2}(n)(n-1)$ and n^2

$\Downarrow$ $\Downarrow$

$\frac{n^2}{2} - \frac{n}{2}$ and n^2

order $\Downarrow$ $\Downarrow$ order

n^2 same as n^2

So, $\frac{1}{2}(n)(n-1)$ has same order of growth as n^2. So, $\boxed{\frac{1}{2}(n)(n-1) \in \Omega(n^2)}$

c) 100n+5 and n^2

order ⇩ ⇩ order

$n < n^2$

Observe that n has lower order of growth than n^2. So, $100n+5 \notin \Omega(n^2)$

Now, let us see "What is big Theta (θ) notation?"

Definition: Assuming n indicates the size of input and g(n) is a function, informally θ(g(n)) is defined as set of functions that have same order of growth as g(n)) as n goes to infinity.

Example 1: Let $g(n) = n^2$. Note that an^2+bn+c for $a > 0$, $n^2+\sin n$, $n^2+\log n$ have same order as that of n^2. Hence, they belong to θ(g(n)) and can be represented as shown:

$$\boxed{\begin{array}{ll} n^3 & \in \theta(n^3) \\ n^4 & \in \theta(n^3) \\ 0.001n^4 & \in \theta(n^3) \\ n^5+n+3 & \in \theta(n^3) \end{array}}$$

2.4.2. Formal Definitions of Asymptotic notations O (Big-Oh)

Let us see "What is Big Oh (O) notation?"

Definition: Let f(n) be the time efficiency of an algorithm. The function f(n) is said to be O(g(n)) [read as big-oh of g(n)], denoted by

$$f(n) \in O(g(n))$$

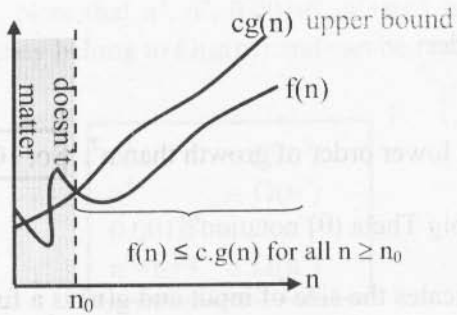
or

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

such that there exists a positive constant c and positive integer n_0 satisfying the constraint

$$f(n) \leq c*g(n) \text{ for all } n \geq n_0.$$

So, if we draw the graph f(n) and c*g(n) verses n, the graph of the function f(n) lies below the graph of c*g(n) for sufficiently large value of n as shown below:



Here, c.g(n) is the upper bound. The upper bound on f(n) indicates that function f(n) will not consume more than the specified time c*g(n) i.e., running time of function f(n) may be equal to c.g(n), but it will never be worse than the upper bound. So, we can say that f(n) is generally faster than g(n)

Note: Big-O is the formal method of expressing the upper bound of an algorithm's running time. It is a measure of the longest amount of time it could possibly take for the algorithm to complete.

Note: The function g(n) is normally expressed using higher order terms of f(n). This is achieved using the following steps:

- Take the lower order term of f(n), replace the constant with next higher order variable. Repeat this step, till we get the higher order term and call it as c.g(n).
- Once the constraint "$f(n) \le c.g(n)$ for $n \ge n_0$" is obtained we say $f(n) \in O(g(n))$

Example 1: Let f(n) = 100n + 5. Express f(n) using big-oh

Solution: It is given that f(n) = 100n + 5. Replacing 5 with n (so that next higher order term is obtained), we get 100n + n and call it c.g(n)

i.e., $\mathbf{c.g(n) = 100n + n}$ for n = 5

$\mathbf{= 101n}$ for n = 5

Now, the following constrain is satisfied:

$$f(n) \le c * g(n) \quad \text{for } n \ge n_0$$

i.e., $100n + 5 \le 101\,n$ for $n \ge 5$

It is clear from the above relations that c = 101, **g(n) = n** and $n_0 = 5$. So, by definition

$f(n) \in O(g(n))$ i.e., $\mathbf{f(n) \in O(n)}$

Example 2: Let $f(n) = 10n^3+8$. Express f(n) using Big-Oh.

Solution: It is given that $f(n) = 10n^3 + 8$. Replacing 8 with n^3 (so that next higher order term is obtained), we get $10n^3 + n^3$ and call it c.g(n).

i.e., $\mathbf{c.g(n)} = 10n^3 + n^3 = 11n^3$ for $n = 8$

Now, the following constraint is satisfied:

$$f(n) \le c * g(n) \quad \text{for } n \ge n_0$$

$$\text{i.e, } 10n^3 + 8 \le 11 * n^3 \quad \text{for } n \ge 8$$

It is clear from the above relations that $c = 11$, $\mathbf{g(n) = n^3}$ and $n_0 = 8$. By definition,

$$f(n) \in O(g(n)) \text{ i.e., } \mathbf{f(n) \in O(n^3)}$$

Example 3: Prove that every polynomial $p(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots\ldots a_1 n^1 + a_0$ with $a_k > 0$ belongs to $O(n^k)$.

Solution: It is given that $f(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots\ldots a_1 n^1 + a_0$

Each term in the summation is of the form $a_i n^i$. Since, n is non-negative, a particular term will be negative only if $a_i < 0$. So,

$$|f(n)| = |\, a_k n^k + a_{k-1}n^{k-1} + \ldots\ldots a_1 n^1 + a_0 \,|$$

$$\le |a_k| n^k + |a_{k-1}| n^{k-1} + \ldots\ldots |a_1|\, n^1 + |a_0|$$

$$= n^k \left[|a_k| + \frac{|a_{k-1}|}{n} + \ldots\ldots \frac{|a_1|}{n^{k-1}} + \frac{|a_o|}{n^k} \right]$$

$$\le n^k \left(|a_k| + |a_{k-1}| + \ldots\ldots |a_1| + |a_o| \right)$$

$$\le c * n^k \text{ where } c = \left(|a_k| + |a_{k-1}| + \ldots\ldots |a_1| + |a_o| \right)$$

$$\text{i.e., } f(n) \le c * n^k \text{ where } c = \left(|a_k| + |a_{k-1}| + \ldots\ldots |a_1| + |a_o| \right) \text{ for } n > 1$$

$$f(n) \le c * g(n) \quad \text{for } n > n_o \text{ where } g(n) = n^k \text{ and } n_0 = 1.$$

So, by definition, we can write

$$f(n) = O(g(n)) \text{ or } f(n) \in O(g(n))$$

$$\text{i.e., } \mathbf{f(n) = O(n^k) \text{ or } f(n) \in O(n^k)}$$

Example 4: $f(n) = 6*2^n + n^2$. Express f(n) using Big-Oh.

Solution: It is given that $f(n) = 6*2^n + n^2$. Replacing n^2 with 2^n (so that next higher order term is obtained), we get $6*2^n + 2^n$ and call it c.g(n).

i.e., $\mathbf{c.g(n)} = 6*2^n + 2^n = 7*2^n$ for $n >= 0$

Since $2^n \geq n^2$. Taking log
i.e., $n \log_2 2 \geq 2\log_2 n$
i.e., $n \geq 2\log_2 n$
which is true for $n \geq 0$

Now, the following constraint is satisfied:

$$f(n) \leq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e, } 6*2^n + n^2 \leq 7 * 2^n \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 7$, $g(n) = 2^n$ and $n_0 = 0$. So, by definition,

$$f(n) \in O(g(n)) \text{ i.e., } \mathbf{f(n) \in O(2^n)}$$

Note: Instead of replacing n^2 by 2^n, we could have replaced it by $2*2^n$, $3*2^n$,.....Only thing is, the value of c and n_0 will change.

Note: Big-O notation is widely used. This is because, we normally take worst case scenario and prepare for the worst and hope for the best. "Is there is any disadvantage?" Yes. The only limitation of Big-O is that there is no lower bound for f(n) for large value of n. The lower bound can be obtained using Big-Omega.

Ω (Big-Omega)

Let us see "What is Big-Ω?"

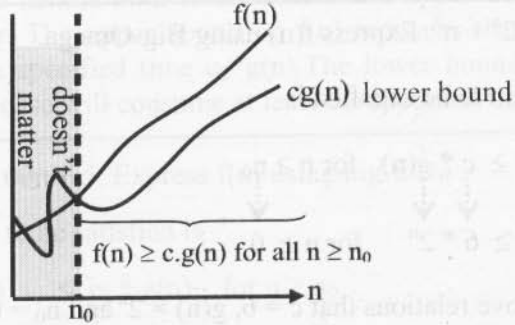
Definition: Let f(n) be the time complexity of an algorithm. The function f(n) is said to be Ω(g(n)) [read as big-omega of g(n)] which is denoted by

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

such that there exists a positive constant c and non-negative integer n_0 satisfying the constraint

$$f(n) \geq c*g(n) \text{ for all } n \geq n_0.$$

So, if we draw the graph f(n) and c*g(n) verses n, the graph of f(n) lies above the graph of g(n) for sufficiently large value of n as shown below:



This notation gives the lower bound on a function f(n) within a constant factor. The lower bound on f(n) indicates that function f(n) will consume at least the specified time c*g(n) i.e., the algorithm has a running time that is always greater than c*g(n). In general, the lower bound implies that below this time the algorithm can not perform better.

Note: $f(n) \geq c*g(n)$ indicates that g(n) is a lower bound and the running time of an algorithm is always greater than g(n). So, big-omega notation is used for finding best case time efficiency.

Example 1: Let $f(n) = 100n + 5$. Express f(n) using big-omega

Solution: The constrain to be satisfied is

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e., } 100n + 5 \geq 100\,n \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 100$, $g(n) = n$ and $n_0 = 0$. So, by definition

$$f(n) \in \Omega(g(n)) \text{ i.e., } f(n) \in \Omega\,(n)$$

Example 2: Let $f(n) = 10n^3+5$. Express f(n) using Big-omega.

Solution: The constrain to be satisfied is

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e, } 10n^3 + 5 \geq 10 * n^3 \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 10$, $g(n) = n^3$ and $n_0 = 0$. By definition,

$$f(n) \in \Omega\,(g(n)) \text{ i.e., } f(n) \in \Omega\,(n^3)$$

Example 3: $f(n) = 6*2^n + n^2$. Express f(n) using Big-Omega.

Solution: The constrain to be satisfied is

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e, } 6*2^n + n^2 \geq 6 * 2^n \quad \text{for } n \geq 0$$

It is clear from the above relations that $c = 6$, $g(n) = 2^n$ and $n_0 = 0$. By definition,

$$f(n) \in \Omega(g(n)) \text{ i.e., } f(n) \in \Omega(2^n)$$

θ (Big-Theta)

Let us see "What is Big-θ?"

Definition: Let f(n) be the time complexity of an algorithm. The function f(n) is said to be big-theta of g(n), denoted

$$f(n) \in \theta(g(n))$$

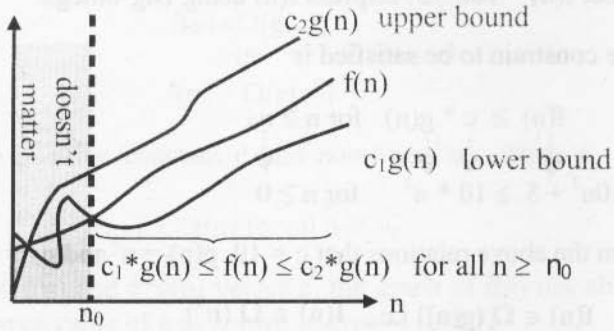
or

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

such that there exists some positive constants c_1, c_2 and non-negative integer n_0 satisfying the constraint

$$c_1*g(n) \leq f(n) \leq c_2*g(n) \text{ for all } n \geq n_0.$$

So, if we draw the graph f(n), $c_1*g(n)$ and $c_2*g(n)$ verses n, the graph of function f(n) lies above the graph of $c_1*g(n)$ and lies below the graph of $c_2*g(n)$ for sufficiently large value of n as shown below:



This notation is used to denote both lower bound and upper bound on a function f(n) within a constant factor. The upper bound on f(n) indicates that function f(n) will not consume more than the specified time $c_2*g(n)$.The lower bound on f(n) indicates that function f(n) in the best case will consume at least the specified time $c_1*g(n)$.

Example 1: Let f(n) = 100n + 5. Express f(n) using big-theta

Solution: The constrain to be satisfied is

$$c_1*g(n) \le f(n) \le c_2*g(n) \quad \text{for } n \ge n_0$$

$$100*n \le 100n + 5 \le 105*n \quad \text{for } n \ge 1$$

It is clear from the above relations that $c_1 = 100$, $c_2 = 105$, $n_0 = 1$, $g(n) = n$.
So, by definition

$$f(n) \in \theta(g(n)) \text{ i.e., } \quad f(n) \in \theta(n)$$

Example 2: Let $f(n) = 10n^3+5$. Express f(n) using Big-omega.

Solution: The constrain to be satisfied is

$$c_1*g(n) \le f(n) \le c_2*g(n) \quad \text{for } n \ge n_0$$

$$10*n^3 \le 10n^3 + 5 \le 11*n^3 \quad \text{for } n \ge 2$$

$$10n^3 + 5 \le 10n^3 + n^3 \quad \text{for } n^3 \ge 5$$

$$10n^3 + 5 \le 11*n^3 \quad \text{for } n^3 \ge 5$$

i.e., $n \ge 2$

It is clear from the above relations that $c_1 = 10$, $c_2 = 11$, $n_0 = 2$, $g(n) = n^3$. By definition,

$$f(n) \in \theta(g(n)) \text{ i.e., } \quad f(n) \in \theta(n^3)$$

Example 3: $f(n) = 6*2^n + n^2$. Express f(n) using Big-Omega.

Solution: The constrain to be satisfied is

$$c_1*g(n) \le f(n) \le c_2*g(n) \quad \text{for } n \ge n_0$$

$$6*2^n \le 6*2^n + n^2 \le 7*2^n \quad \text{for } n \ge 0$$

Since $2^n \ge n^2$. Taking log
i.e., $n \log_2 2 \ge 2\log_2 n$
i.e., $n \ge 2\log_2 n$
which is true for $n \ge 0$

It is clear from the above relations that $c_1 = 6$, $c_2 = 7$, $n_0 = 0$, $g(n) = 2^n$. By definition,

$$f(n) \in \theta(g(n)) \text{ i.e., } \quad f(n) \in \theta(2^n)$$

2.5 Property of asymptotic notation

If an algorithm has two executable parts, the analysis of this algorithm can be obtained using the following theorem.

Theorem: If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then $f_1(n)+f_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

Note: The same assertion is true if we replace O notation by Ω or θ notations.

Note: Let us consider four arbitrary numbers a_1, a_2, b_1 and b_2. If $a_1 \leq b_1$ and $a_2 \leq b_2$, then the following simple relation holds good: $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$. Let us use the above fact in proving this theorem.

Proof: By definition, we know that the function f(n) is said to be big-oh of g(n), denoted by

$$f(n) \in O(g(n))$$

such that there exists a positive constant c and positive integer n_0 satisfying the constraint

$$f(n) \leq c*g(n) \text{ for all } n \geq n_0.$$

It is given that $f_1(n) \in O(g_1(n))$. So, by definition, there exists a relation

$$f_1(n) \leq c_1 * g_1(n) \quad \text{for } n \geq n_1 \text{ ----------(1)}$$

It is given that $f_2(n) \in O(g_2(n))$. So, by definition, there exists a relation

$$f_2(n) \leq c_2 * g_2(n) \quad \text{for } n \geq n_2 \text{ ----------(2)}$$

Let us assume $c_3 = \max\{c_1, c_2\}$ and $n = \max\{n_1, n_2\}$ ----------(3)

By adding equation (1) and equation (2) we have,

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 * g_1(n) + c_2 * g_2(n) \\ &\leq c_3 * g_1(n) + c_3 * g_2(n) \\ &\leq c_3 [g_1(n) + g_2(n)] \\ &\leq c_3\, 2*\max\{g_1(n), g_2(n)\} \end{aligned}$$

Since, $f_1(n) + f_2(n) \leq c_3\, 2*\max\{g_1(n), g_2(n)\}$, by definition we can write

$$\mathbf{f_1(n) + f_2(n) \in O(\max\{ g_1(n), g_2(n) \})}$$

where $c = 2c_3 = 2\max\{c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$. Hence, the proof.

Note: It is clear from the above property that, the overall efficiency of the algorithm is determined by the executable part which has larger order of growth.

Example 1: Find the time complexity of an algorithm which has two parts:

- sorting part requiring ½n(n-1) comparisons
- To check consecutive elements requiring n comparisons

Part 1: The sorting algorithm in the worst case uses ½n(n-1) comparisons. So, the time complexity of sorting part is $O(n^2)$.

Part 2: The algorithm to check whether consecutive elements are present or not, requires no more than (n-1) comparisons. So, the time complexity of the second part is given by O(n).

We know from the property of asymptotic notations that
if $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$ then $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

According to this property, the time complexity of entire algorithm is given by

$$O(\max\{n^2, n\}) = O(n^2)$$

2.6 Order of growth using limits

Instead of using asymptotic notations, the order of growth can be obtained by computing the limit of the ratio of the two functions as shown below:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0 & \text{imply f(n) has smaller order of growth than g(n) i.e., } f(n) \in O(g(n)) \\ c & \text{imply f(n) has same order of growth than g(n) i.e., } f(n) \in \theta(g(n)) \\ \infty & \text{imply f(n) has a larger order of growth than g(n) i.e., } f(n) \in \Omega(g(n)) \end{cases}$$

Note: To compute the order of growth, the limit-based approach is often convenient because, it is more advantage to use powerful calculus techniques such as L'Hospitals's rule defined by

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f'(n)}{g'(n)}$$

and Sterling's' formula which is given by

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for very large values of n.}$$

Example 1: Compare the orders of growth $\frac{1}{2}n(n-1)$ and n^2 using limits

Solution: It is given that $f(n)=\frac{1}{2}n(n-1)$ and $g(n) = n^2$, we can find the limit of ratio of function f(n) and g(n)as shown below:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right)=\frac{1}{2}\left(1-\frac{1}{\infty}\right)=\frac{1}{2}$$

Since the limit is a positive constant, both the functions f(n) and g(n) have the same order of growth (second condition holds good) and it is symbolically represented as:

$$\boxed{\frac{1}{2}n(n-1)\in\theta(n^2)}$$

Example 2: Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$

Solution: It is given that $f(n)=\log_2 n$ and $g(n)=\sqrt{n}$ we can find the limits as shown below:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{\left(\frac{\log_e n}{\log_e 2}\right)}{\sqrt{n}}$$

$$=\lim_{n\to\infty}\frac{\frac{1}{\log_e 2}(\log_e n)}{\sqrt{n}}$$

$$=\lim_{n\to\infty}\frac{\log_2 e(\log_e n)}{\sqrt{n}}$$

$$=\log_2 e\left(\lim_{n\to\infty}\frac{\log_e n}{\sqrt{n}}\right)$$

According to L'Hospital's rule,

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{f'(n)}{g'(n)}=\log_2 e\left(\lim_{n\to\infty}\frac{(\log_e n)'}{(\sqrt{n})'}\right)\text{.....(1)}$$

$(\log_e n)'$ = derivative of $(\log_e n)=\frac{1}{n}$ .....(2)

$(\sqrt{n})'$ = derivative of $\sqrt{n}=n^{\frac{1}{2}}$ can be obtained as shown below:

Note: derivative of $x^n = n.x^{n-1}$

$$= \frac{1}{2}n^{\left(\frac{1}{2}-1\right)} = \frac{1}{2}n^{\left(-\frac{1}{2}\right)} = \frac{1}{2.n^{\frac{1}{2}}} = \frac{1}{2\sqrt{n}} \quad \text{.....(3)}$$

Substituting equations (2) and (3) in equation (1) we have,

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f'(n)}{g'(n)} = \log_2 e\left(\lim_{n\to\infty}\frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}}\right) = \log_2 e\left(\lim_{n\to\infty}\frac{2\sqrt{n}}{n}\right) = 2\log_2 e\left(\lim_{n\to\infty}\frac{\sqrt{n}}{n}\right) = 0$$

Since the limit is equal to 0, the first condition holds good and $f(n) = \log_2 n$ has a smaller order of growth than $g(n) = \sqrt{n}$ and symbolically, it is represented as shown below:

$$\boxed{\log_2 n \in O(\sqrt{n})}$$

Example 3: Compare the orders of growth of $n!$ and 2^n

Solution: Given that $f(n) = n!$ and $g(n) = 2^n$ we can find the limits as shown below:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{n!}{2^n} = \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n^n}{2^n e^n}\right) = \lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty$$

Since the third condition is satisfied, i.e., $\lim_{n\to\infty}\frac{t(n)}{g(n)} = \infty$, symbolically it is represented as

$$\boxed{f(n) = n! \in \Omega(2^n)}$$

Note: It is clear from above example that $n!$ has larger order of growth than 2^n. So, as 2^n grows very fast, $n!$ still grows faster. But, it is not evident from formal definition of Ω.

Now, let us see "What is the general plan of analyzing non-recursive algorithms?" The

2.7 Mathematical analysis of Non-recursive algorithms

Now, let us see "What is the general plan of analyzing non-recursive algorithms?" The general plan is shown below:

- Based on the input size, determine the number of parameters to be considered.
- Identify the basic operation in the algorithm.
- Check whether the number of times the basic operation is executed depends only on the size of the input. If the basic operation to be executed depends on some other conditions, then it is necessary to obtain the worst case, best case and average case separately.
- Obtain the total number of times a basic operation is executed
- Simplify using standard formulas, obtain the order of growth

2.7.1 Maximum of n elements

Let us "Obtain the time efficiency of an algorithm which finds the largest element in a list of n numbers?

Design: Let a[0], a[1],.....etc. be an array of n elements. Let us assume that the variable big contains the first element of the array. The code for this can be of the form:

```
big ← a[0];    // Assume first item is big
```

In the next step, subsequent elements a[1],a[2]....a[n-1] should be compared with big as shown below:

```
if ( a[i] > big )
    big ← a[i];
end if
```

Note: This statement should be executed for each value of i = 1, 2, 3, ...n-1

If the above condition is true, then a[i] is larger and it is stored in variable big. Since the above if-statement has to be executed for each value of i = 1, 2, 3,...n-1, the corresponding pseudo code can be written as shown below

```
for i ← 1 to n-1 do
    if ( a[i] > big )
        big ← a[i];
    end if
end for
```

The complete algorithm to find the largest of N elements is shown below:

Algorithm Maximum(a[], n):

//**Purpose:** Finds the largest element in an array of n elements

```
// Inputs :
//      n – the number of items present in the table
//      a – the table consisting of n elements

//Output:
//      big – contains the largest value of n elements and this value is returned

      big ← a[0]                          // first element is assumed to be big

      for i ← 1 to n-1 do                 // Find the largest in remaining elements
            if ( a[i] > big )
                  big ← a[i]
            end if
      end for

      return big                          // return the largest
```

Analysis: The time efficiency can be calculated as shown below:

- **Step 1**: In this problem, the size of input is n. This is the parameter to be considered.
- **Step 2**: The statement executed in inner most loop is a basic operation. In this algorithm:

  Basic operation is **"if (a[i] > big)"**

- **Step 3**: Each time the control enters into the loop, the basic operation is executed once. So, the total number of times the basic operation is executed is given by the following relation:

```
for i ← 1 to n-1 do
      if ( a[i] > big )
```

$$f(n) = \sum_{i=1}^{n-1} 1$$ **Note:** Upper bound = n-1, Lower bound = 1

$= (n-1)-1+1$ **Note:** Result = [Upper bound – lower bound +1]

$= n-1$

i.e., **f(n) = n-1**

- **Step 4:** Express f(n) using asymptotic notation.

  The total number of times the basic operation is executed remains same both in best case and worst case. So, f(n) is expressed using θ notation. The constraint to be satisfied is:

$$c_1 g(n) \le f(n) \le c_2 g(n) \quad \text{for } n \ge n_0$$

$$1/3*n \le n-1 \le 1*n \quad \text{for } n \ge 1 \text{ where } c_1 = 1/3,\ c_2 = 1,\ g(n) = n.$$

So, by definition,

$$f(n) \in \theta(g(n))$$

i.e., $\mathbf{f(n) \in \theta(n)}$

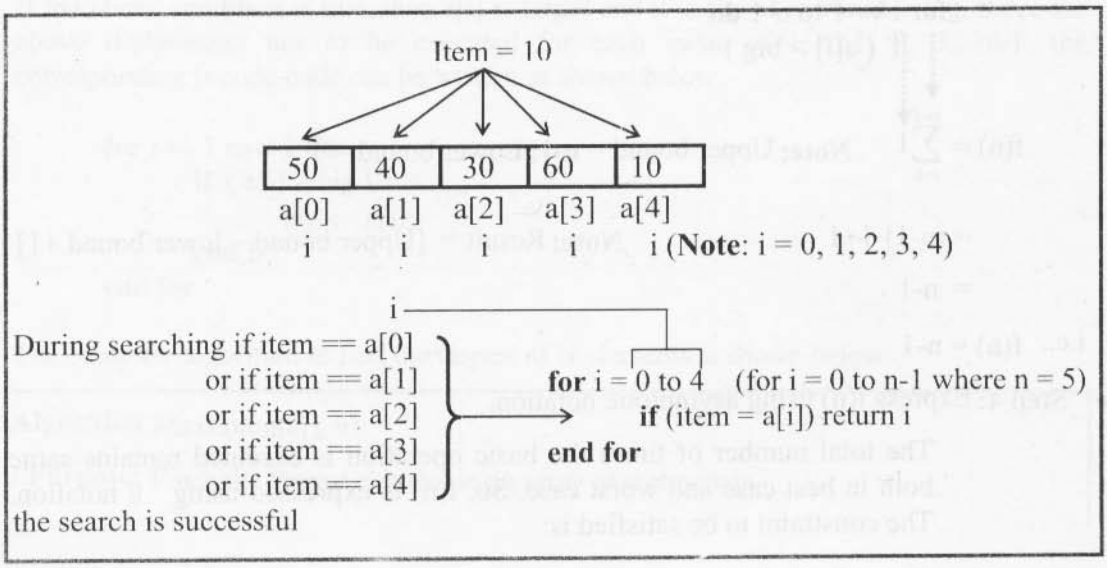
2.7.2 Linear search

Before writing the algorithm or the program it is better to know the answer for "What is linear search?"

Definition: A linear search is a simple searching technique. In this technique we search for a given key item in the list in linear order (sequential order) i.e., one after the other. The item to be searched is often called key item. The linear search is also called sequential search.

For example, if key = 10 and the list is 20, 10, 40, 25 after searching we say that key is present. If key = 100 after searching we say that key is not present. Once we know the concept of linear search, the next question is "How to search for an item in a list of elements?" The procedure is shown below:

Design: Let us take an example. Assume 10 is the item to be searched in the list 50, 40, 30, 60, 10. Observe from following figure that, 10 has to be compared with a[0], a[1], a[2], a[3] and a[4] as shown below:



But, once the value of i is greater than or equal to 5, it is an indication that item is not present and display the message "Unsuccessful search". The complete algorithm is shown below:

```
Algorithm Linear_Search(key, a[], n):
//Purpose: To search for key in an array of n elements
// Inputs :
//        n – the number of items present in the table
//        a – the items to be searched are present in the table
//        key – the item to be searched
//Output:
//        return i as the position if key found
//        return -1  if key not found

Step 1: [Search for the key in the array/table]
        for  i ← 0 to n-1 do
                if ( a[i] = key )
                        return i;        // Key Found at position i
                end if
        end for

Step 2:[Key not found]
        return -1;
```

Note: In the algorithm, the basic operation is "if (a[i] = key)" in the for loop. Here, the number of comparisons purely depends on the position of key in the array. So, the running time does not depend on the input value n. So, we find worst case, best case and average case time efficiencies.

Best case efficiency: In the best case, the algorithm runs fastest among all possible inputs. So, to analyze the best case, we have to identify the kind of inputs for which the basic operation count is smallest among all possible inputs.

- Successful search: In linear search, for successful search the best case occurs if the search key happens to be the first element in the table and number of comparisons required will be 1. So, best case efficiency for successful search is given by

$$F(n) \in \Omega(1)$$

- Un-Successful search: In linear search, for unsuccessful search the best case occurs if the search key is not there in the table and number of comparisons required will be n. So, best case efficiency for unsuccessful search is given by

$$f(n) \in \Omega(n)$$

Worst case efficiency: In the worst case analysis, the algorithm runs for the longest duration among all possible inputs. Here, maximum number of steps are executed for a given input. In the worst case, the maximum number of comparisons are required.

- Successful search: This case occurs if the item to be searched is present in the last position. So, the number of times the basic operation is executed = n. So, the worst case time efficiency for successful search is given by

$$f(n) \in O(n)$$

- Un-Successful search: In linear search, for unsuccessful search the worst case occurs if the search key is not there in the table and number of comparisons required will be n. So, worst case efficiency for unsuccessful search is given by

$$f(n) \in O(n)$$

Note: The worst case analysis of a given algorithm provides very important information about the efficiency of a given algorithm. It guarantees that for any given worst case input of size n, the running time will not exceed the time f(n).

Average case efficiency: The item that we are searching for may be present at any position in the list. In such case, we take the average time efficiency.

- Successful search: The item that we are searching for may be present anywhere in the array from first to last position. So, the total number of comparisons at any of the position is given by:

$$1 + 2 + 3 + \ldots\ldots n$$

The average number of comparisons is given by

$$\frac{1 + 2 + 3 + \ldots\ldots n}{n} = \frac{n(n+1)}{2n} = \frac{(n+1)}{2}$$

So, f(n) = (n+2)/2 whose time efficiency is given by $\theta(n)$

So, average time efficiency is given by $f(n) \in \theta(n)$

Note: The time efficiency for unsuccessful is same as above.

2.7.3 Element uniqueness problem

Let us "Design an algorithm to check whether all the elements in a given array are distinct and find the time efficiency"

Note: If all the elements in the list are distinct, then we say that elements in the list are unique otherwise, the elements in the list are not unique.

Design: Consider an array with 5 elements. For the first time, the item a[0] should be compared with a[1], a[2], a[3] and a[4]. Second time, item a[1] should be compared with a[2], a[3] and a[4]. Third time, item a[2] should be compared with a[3] and a[4] and finally, a[3] should be compared with a[4]. Pictorially, the above procedure is shown below:

1st time	2nd time	3rd time	4th time
a[0] – a[1]	a[1] – a[2]	a[2] – a[3]	a[3] – a[4]
a[0] – a[2]	a[1] – a[3]	a[2] – a[4]	
a[0] – a[3]	a[1] – a[4]		
a[0] – a[4]			
i j	i j	i j	i j

Observe the following facts from the above table:

- The index i variable starts from 0 and goes up to 3.
  i.e., i = 0 to 3
  i.e., i = 0 to 5-2;
  i.e., i = 0 to n – 2 where n is the number of elements in the array

- The value of the index variable j always starts from i+1 and goes up to 4 in this example.
  i.e., j = i + 1 to 4
  i.e., j = i + 1 to 5 – 1
  i.e., j = i + 1 to n – 1

- If a[i] is same as a[j], then duplicate item found and return 0.
  if (a[i] = a[j]) return 0

In general, the algorithm can be written using the following pseudocode as shown below:

```
Algorithm Unique_Elements(a[], n)

//Purpose: Checks whether the elements in a given array are distinct

// Inputs :
//        n: the number of items present in the table
//        a: an array consisting of n elements

// Output:
// The function returns 0, if duplicate items are present
// The function returns 1, if the items are unique
```

```
for i ← 0 to n-2 do
    for j ← i + 1 to n – 1 do
        if (a[i] = a[j])
            return 0;        // Distinct elements are not present
        end if
    end for
end for

return 1;        // Distinct elements are present
```

Analysis The time efficiency of this algorithm can be computed as shown below:

Step 1: The parameter to be considered is n, which represent size of the input

Step 2: The basic operation is the statement "if a[i] = a[j]" in the innermost for loop

Step 3: The number of comparisons not only depends on the value of n but depends on the position of repeated elements if present. The repeated elements if not present, the number of comparisons are more. So, let us find the worst case time efficiency and best case time efficiency.

Worst case time efficiency: The partial code considered is shown below:

```
for i = 0 to n-2
    for j = i+1 to n-1
        if (a[i] = a[j])
```

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \quad \text{Here, upper limit = n-1, lower limit = i + 1}$$

$$= \sum_{i=0}^{n-2} (n-1)-(i+1)+1 \quad \text{(Result = upper limit – lower limit + 1)}$$

$$= \sum_{i=0}^{n-2} n-1-i-1+1 = \sum_{i=0}^{n-2} n-1-i \quad \text{(Replace i by 0, 1, …n-2)}$$

$$= (n-1)+(n-2)+(n-3)+.....3+2+1$$

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

So, the total number of times the basic statement is executed is given by $\frac{(n-1)n}{2}$.

So, if we consider $f(n) = \frac{(n-1)n}{2}$, then

$$\frac{(n-1)n}{2} \le n^2 \text{ for } n \ge 0 \text{ which is of the form}$$

$$f(n) \le cg(n) \text{ for } n \ge n_0 \text{ where } c = 1,\ g(n) = n^2 \text{ and } n_0 = 0$$

By definition, $\mathbf{f(n) \in O(g(n))}$ for the worst case. So, $\mathbf{f(n) \in O(n^2)}$

> So, time complexity to find the uniqueness in the worst case is given by $f(n) \in O(n^2)$

Best case time efficiency: The partial code considered is shown below:

```
for i = 0 to n-2
    for j = i+1 to n-1
        if (a[i] = a[j])
```

The best case occurs if a[0] is same as a[1]. It is observed from the above code that in the best case, the basic operation "if(a[i] = a[j])" will be executed only once. So, the time complexity is given by

$$\boxed{\mathbf{f(n) \in \Omega(1)}}$$

2.7.4 Matrix multiplication

Design: The two matrices a and b can be multiplied and the result can be stored in matrix c for each value of i and j as shown below:

$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j] \text{ for } i = 0 \text{ to n-1 and for } j = 0 \text{ to n-1}$$

The above mathematical formula can be converted into pseudocode as shown below:

Algorithm Multiplication(**a**[], **b**[], **c**[], **n**)

//**Purpose:** Multiply two matrices a and b of size **n** x **n**

// **Inputs :**
// **n:** represent size of the array
// **a:** First matrix of size n x n
// **b:** Second matrix of size n x n
//**Output:**
// **c:** Resultant matrix where the product of two matrices should be stored

```
for i ← 0 to n-1
    for j ← 0 to n-1
        sum ← 0
        for k = 0 to n-1
            sum ← sum + a[i][k]*b[k][j]
        end for

        c[i][j] ← sum
    end for
end for
```

Analysis The time complexity in the best case, worst case remains same.

Step 1: The parameter to be considered is n, which is a measure of input's size

Step 2: The basic operation is the multiplication statement in the innermost for loop

Step 3: The number of multiplications depends on the value of n only and not on any other factors. So, the total number of times the multiplication statement is executed can be obtained as shown below:

```
for i ← 0 to n-1
    for j ← 0 to n-1
        sum ← 0
        for k = 0 to n-1
            sum ← sum + a[i][k]*b[k][j]
```

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 \qquad \text{(Upper limit = n-1 and lower limit = 0)}$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n-1-0+1 \qquad \text{(Result = upper limit – lower limit + 1)}$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= \sum_{i=0}^{n-1} n \sum_{j=0}^{n-1} 1 \qquad \text{(upper limit = n-1, lower limit = 0)}$$

$$= \sum_{i=0}^{n-1} n(n-1-0+1) \qquad \text{(Result = upper limit – lower limit + 1)}$$

$$= \sum_{i=0}^{n-1} n(n) = \sum_{i=0}^{n-1} n^2$$

$$= n^2 \sum_{i=0}^{n-1} 1 \qquad \text{(upper limit = n-1, lower limit = 0)}$$

$$= n^2 (n-1-0+1) \qquad \text{(Result = upper limit – lower limit + 1)}$$

$$= n^3$$

So, time complexity to multiply two matrices of size n x n = $\theta(n^3)$

Note: For the worst case we use O-notation, for the best case we use Ω-notation. When worst case and best case is same, then we use θ-notation.

2.8 Mathematical analysis of recursive algorithms

In this section, let us concentrate on analysis of recursive algorithms. Now, let us see "What is the general plan to analyze the efficiency of recursive algorithms?" The general plan to analyze the recursive algorithms is shown below:

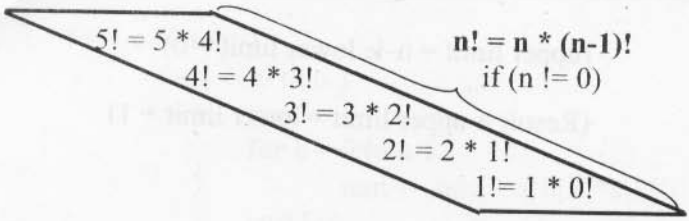
- Based on the input size, decide the various parameters to be considered.
- Identify the basic operation in the algorithm
- Obtain the number of times the basic operation is executed on different inputs of the same size. If it varies, then it is necessary to obtain the worst case, best case and average case separately.
- Obtain a recurrence relation with an appropriate initial condition
- Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

2.8.1 Factorial of a number

Let us see how to write an algorithm to find the factorial of a number. The factorial of a number n can be found using the series:

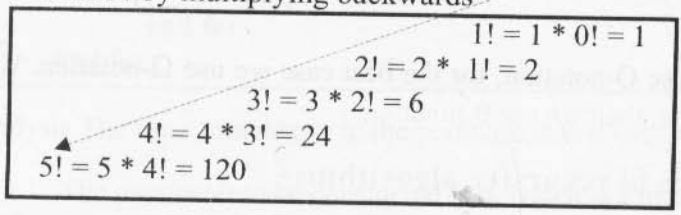
$$n! = 1 * 2 * 3 * \ldots n$$

Using recursion, let us see "How to compute factorial of 5 using recursion?" Factorial of 5 can be computed using recursion as shown below:



0! = 1 ⟶ n! = 1 when n=0

Obtain 5! by multiplying backwards



Note: **n! = 1 if n = 0**
n! = n*(n-1)! otherwise

Formally, the recursive definition to compute n! is shown below:

$$\textbf{fact(n)} = \begin{cases} 1 & \textbf{if n = 0} \\ \textbf{n x fact(n-1)} & \textbf{if n > 0} \end{cases}$$

Using the above recursive definition, the recursive algorithm can be written as shown below:

```
Algorithm fact(n)
// Purpose: This function computes factorial of n
// Input:
//      n: represent a positive integer
//Output:
//      factorial of n
    if ( n == 0 )   return 1            // n! = 1 if n is 0
    return          n*fact(n-1)         // n! = n*(n-1)! otherwise
```

Analysis The time efficiency of the above algorithm can be obtained as shown below:

Step 1: The parameter to be considered is n, which is a measure of input's size

Step 2: The basic operation is the multiplication statement.

Step 3: The total number of multiplications can be obtained using the recurrence relation as shown below:

$$t(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + t(n-1) & \text{otherwise} \end{cases} \quad \text{OR} \quad \begin{aligned} t(0) &= 0 \\ t(n) &= 1 + t(n-1) \end{aligned} \quad \text{----------(1)}$$

The above recurrence relation can be solved using repeated substitution as shown below:

$t(n) = 1 + t(n-1)$	$t(n) = 1 + t(n-1)$ ----------(2)
	$t(n-1) = 1 + t(n-2)$ replacing n by n-1 in equation (2)
$= 1 + 1 + t(n-2)$	
$= 2 + t(n-2)$	$t(n-2) = 1 + t(n-3)$ replacing n by n-2 in equation (2)
$= 2 + 1 + t(n-3)$	
$= 3 + t(n-3)$	
$= 4 + t(n-4)$	**Note:** Looking at the previous expression it is written
.....	
.....	
.....	
$= i + t(n-i)$	
⋮ ..... →	Finally, to get initial condition t(0), **let i = n**
$= n + t(n-n)$	
$= n + t(0)$	**Note:** $t(0) = 0$ from equation (1)
$= n + 0 = n$	

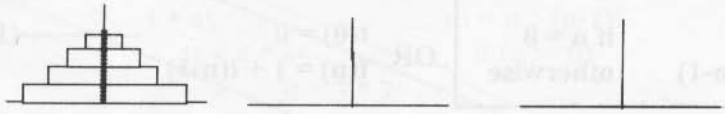
So, the time complexity of factorial of N is given by $t(n) \in \theta(n)$

2.8.2 Tower of Hanoi

Let us see, "What is tower of Hanoi problem?" In this problem, there are three needles say **A**, **B** and **C**. There are **n** discs of different diameters in the needle **A** and are placed one above the other such that always a smaller disc is placed above the larger disc. The two needles B and C are empty. All the discs from needle A are to be transferred to needle C using needle B as temporary storage. The rules to be followed while transferring the discs are

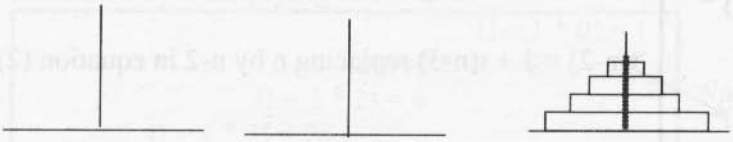
- Only one disc is moved at a time from one needle to another needle
- Smaller disc should be above the larger disc at any time in all the needles.
- Any needle can be used to for storing intermediate disks

The initial set up of the problem is shown below.



A **source** **B** **temp** **C** **destination**

After transferring all discs from **A** to **C** we get the following setup.



source A **B temp** **C destination**

Base Case: Now, let us see "What is the base case?" This case occurs when there is only one disk. In that case move the disk from source to destination. So,

Move disk **n** from **source** to **destination** **if n = 1**

General case: Now, let us see "What is the general case?" This case occurs if there are more disks to be transferred from source to destination. If there are **n** disks, then all **n** disks from source to destination using the specified constraints can be transferred recursively using following three steps:

- Move **n-1** discs recursively from **source** to **temp**.
- Move **n**th from **source** to **destination**.
- Move **n-1** discs from **temp** to **destination.**

The algorithm to implement tower of Hanoi problem is shown below:

```
Algorithm TowerOfHanoi(n, source, temp, destination)

//Purpose: To move n discs from source to destination and see that only one disc is
           moved and always the smaller disc should be placed above the larger disc
           using one of the needle as temporary holder for the disc being moved
// Inputs :
//      n: total number of disks to be moved

//Output:
//      all n disks should be available on the destination needle.

      if ( n = 1 )
            write('Move disk 1 from', source, 'to', destination)
            return
      end if
```

```
// Move n-1 disks from A to B using C as temporary needle
TowerOfHanoi(n-1, source, destination, temp)

write("Move disk", n, "from", source, "to", destination)

// Move n-1 disks from B to C using A as temporary peg
TowerOfHanoi(n-1,temp,source,destination)
```

Analysis The time efficiency can be calculated as shown below:

Step 1: The parameter to be considered is n, which represent number of disks

Step 2: The basic operation is the movement of disk

Step 3: The total number of disk movements can be obtained using the recurrence relation as shown below:

$$t(n) = \begin{cases} 1 & \text{if } n = 1 \\ t(n-1) + 1 + t(n-1) & \text{otherwise} \end{cases}$$

Number of disk movements from source to temp

Number of disk movements from source to destination

Number of disk movements from temp to destination

The above recurrence relation can also be written as shown below:

$$f(1) = 1 \quad \text{----------(1)}$$
$$f(n) = 2f(n-1) + 1$$

The above recurrence relation can be solved using repeated substitution as shown below:

$f(n) = f(n-1) + 1 + f(n-1)$

$f(n) = 2f(n-1) + 1$ ----------(2)

$= 2f(n-1) + 1$

$f(n-1) = 2f(n-2) + 1$ by replacing n by n-1 in eq. (2)

$= 2\,[2f(n-2) + 1\,] + 1$

$= 2^2\, f(n-2) + 2 + 1$

$f(n-2) = 2f(n-3) + 1$ by replacing n by n-2 in eq. (2)

$= 2^2\,[2f(n-3) + 1] + 2 + 1$

$= 2^3\, f(n-3) + 2^2 + 2 + 1$

$= 2^4 f(n-4) + 2^3 + 2^2 + 2 + 1$ **Note:** Written by looking at the previous expression

.....

.....

In general,

$= 2^i f(n-i) + 2^{i-1} + 2^{i-2} \ldots\ldots 2^3 + 2^2 + 2 + 1$

$= 2^i f(n-i) + \underbrace{2^{i-1} + 2^{i-2} \ldots\ldots 2^3 + 2^2 + 2^1 + 2^0}$ ---------- **(3)**

This geometric series can be solved using $S = \frac{a(r^n - 1)}{r-1}$

where a = 1, r = 2, n = i **(since number of terms from 0 to i-1 = i)**

So, $S = \frac{1(2^i - 1)}{2-1} = 2^i - 1$. Substituting this value in equation (3) we have.

$= 2^i f(n-i) + 2^i - 1$

⋮.....>Finally, to get initial condition f(1), **let i = n-1**

$= 2^{n-1} f(n-[n-1]) + 2^{n-1} - 1$

$= 2^{n-1} f(n - n + 1) + 2^{n-1} - 1$

$= 2^{n-1} f(1) + 2^{n-1} - 1$

$= 2^{n-1} * 1 + 2^{n-1} - 1$ **Note:** f(1) = 1 from equation (1)

$= 2^{n-1} + 2^{n-1} - 1$

$= 2*2^{n-1} - 1 = 2*\frac{2^n}{2} - 1 = 2^n - 1$

So, $\mathbf{f(n) = 2^n - 1}$

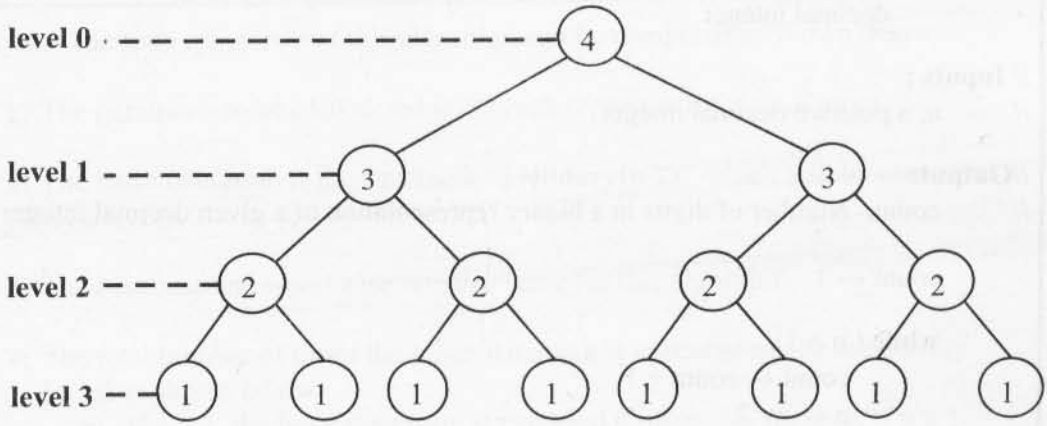
Step 4: Since number of disk movements in best case is same as worst case, we express f(n) using θ-notation as shown below:

So, the time complexity for Tower of Hanoi problem is given by $f(n) \in \theta(2^n - 1) \in \theta(2^n)$

Note: It is clear from the above relation that, the time complexity of this algorithm is exponential in nature and so this algorithm should perform more computations even for a relatively smaller value. This does not mean that the algorithm is poor. In fact, this is the

most efficient algorithm for this problem. The nature of the problem itself is such that it is computationally expensive.

Note: For analysis purpose, a binary tree of recursive calls can be constructed. The nodes in a binary tree correspond to the recursive calls made and are labeled with the value of n which is passed as parameter. So, the tree for tower of Hanoi problem for n = 4 is shown below:



Once the tree is constructed, the number on nodes represent the number of calls made. To find the total number of nodes, it is clear from the above figure that:

- The number of nodes at level 0 = $2^0 = 1$
- The number of nodes at level 1 = $2^1 = 2$
- The number of nodes at level 2 = $2^2 = 4$
- .....
- .....
- Finally, the number of nodes at level n-1 = 2^{n-1}

So, total number of nodes = $f(n) = 2^0 + 2^1 + 2^2 + \ldots\ldots 2^{n-1}$

$$= \frac{a(r^n - 1)}{r - 1} = \frac{1(2^n - 1)}{2 - 1} = 2^n - 1$$

which is same as the answer that we have obtained earlier.

2.8.3 Digits in a binary number (given a decimal number)

In this section, let us see "How to find the number of digits of a binary corresponding to the decimal number?" Here, it is not necessary to obtain the binary number.

Design (Iteration): Let us initial count to 1. As long as the given N is greater than 1, keep dividing it by 2 and increment corresponding count by 1. The algorithm is shown below:

```
Algorithm Binary(n):
//Purpose: To count the number of digits in a binary representation of a given positive
           decimal integer.

// Inputs :
//      n: a positive decimal integer

//Output:
//      count - Number of digits in a binary representation of a given decimal integer.

      count ← 1

      while ( n > 1)
            count ← count + 1
            n ← n / 2
      end while

      return count  // The number of digits
```

Design (recursion): The time complexity of this algorithm can be found very easily, if the recursive version of this algorithm is available. The recursive definition to find the number of binary which is shown below:

$$f(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 1 + f(n/2) & \text{otherwise} \end{cases}$$

The recursive algorithm can be easily written by looking at the above recursive definition. Instead of using f as the name of the function, let us use Binary as the name of the function. The algorithm is shown below:

```
Algorithm Binary(n)

//Purpose: To count the number of digits in a binary given positive decimal integer.

// Inputs :
//      n: a positive decimal integer
```

```
//Output:
//      Number of digits in a binary representation of a given positive decimal integer.

        if ( n = 1) return 1

        return 1 + Binary(n/2)
```

Analysis The time efficiency of this algorithm can be computed as shown below:

Step 1: The parameter to be considered is n which represent the size of input

Step 2: The basic operation is the statement **"1+binary(n/2)"** which will be executed recursively.

Step 3: The best case and worst case remains same for this algorithm.

Step 4: The total number of times the basic statement is executed can be recursively found as shown below:

a) If n = 1, the basic operation is executed 0 times. So, **f(n) = 0, if n ≤ 1**

b) If (n > 1), the basic operation is executed at least once which is given by the relation:

$$\mathbf{f(n) = 1 + f(n/2)}$$

So, the recurrence relation obtained in step 4 can be formally written as shown below:

$$f(n) = \begin{cases} 0 & \text{if } n \le 1 \\ 1 + f(n/2) & \text{Otherwise} \end{cases} \quad \text{----------(1)}$$

Step 5: The recurrence relation can be solved as shown below:

Consider the relation:

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

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

$$f(n) = 1 + f\left(\frac{n}{2}\right) \quad \text{----------(2)}$$

$$f\left(\frac{n}{2}\right) = 1 + f\left(\frac{n}{4}\right) = 1 + f\left(\frac{n}{2^2}\right) \quad \text{replacing n by } \frac{n}{2} \text{ in eq (2)}$$

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

$$f\left(\frac{n}{2^2}\right) = 1 + f\left(\frac{n}{2^3}\right)$$ By replacing n by $\frac{n}{2^2}$ in equation (2)

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

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

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

Note: By looking at the previous expression

.....

.....

In general,

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

.....> Finally, to get initial condition f(1), let $2^i = n$ ----------(3)

$$= i + f(1)$$ **f(1) = 0** from equation (1)

$f(n) = i$ ----------(4)

From equation (3) we have $2^i = n$. Taking log on both sides, we have

$$i * \log_2 2 = \log_2 n$$

$$i = \log_2 n$$

Substituting the value of i in equation (4) we have,

$$f(n) = \log_2 n$$

So, time complexity is given by $f(n) \in \theta(\log_2 n)$

2.8.4 Fibonocci numbers

The Fibonacci series is an infinite sequence of integers

0, 1, 1, 2, 3, 5, 8, 13, ...

The first two integers 0 and 1 are initial values. The next subsequent numbers are obtained by adding the previous two numbers. The definition to find n^{th} Fibonacci number is given by

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

The iterative algorithm to generate fibonocci numbers within n is shown below:

```
Algorithm Fibonocci(n, a)

//Purpose: Generates n Fibonocci numbers

// Inputs :
//      n: the number of terms to be generated

//Output:
//      a – contains n fibonocci numbers

        a[0] ← 0
        a[1] ← 1

        for i ← 2 to n do
                a[i] ← a[i-1] + a[i-2]
        end for
```

The recursive algorithm to find the nth Fibonacci number using recursive definition is shown below:

```
Algorithm Fibonocci(n)

//Purpose: Find nth Fibonacci number

// Inputs :
//      n: a positive integer

//Output:
//      nth Fibonacci number

        if ( n = 0 ) return 0

        if ( n = 1 ) return 1

        return fib(n-1) + fib (n-2)
```

Note: The iterative algorithm is more efficient when compared to the recursiv algorithm. But, even with iterative algorithm, to generate n fibonocci numbers consumes time. So, let us obtain the formula to find n^{th} fibonocci number

2.8.5 Explicit formula to find n^{th} Fibonocci number

We can compute n^{th} fibonocci number using the following relation:

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases} \quad \text{---------(1)}$$

From the above relation, we have

$$fib(n) = fib(n-1) + fib(n-2)$$

i.e., $fib(n) - fib(n-1) - fib(n-2) = 0$ ----------(2)

which is of the form

$$a*x(n) + b*x(n-1) + c*x(n-1) = 0 \text{----------(3)}$$

Equation (3) can be solved using the following formula:

$$x(n) = \alpha x_1^n + \beta x_2^n \text{ ----------(4)}$$

where α and β are arbitrary real constants.

But, to find the value of x_1 and x_2 we need to solve the quadratic equation:

$$ax^2 + bx + c = 0 \text{ ----------(5)}$$

By comparing equations, (2) and (3) we have

$a = 1$, $b = -1$ and $c = -1$.

Substituting these values in equation (5) we get,

$$x^2 - x - 1 = 0$$

The roots of this quadratic equation can be obtained using the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{1 \pm \sqrt{1-4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

So, $x_1 = \frac{1+\sqrt{5}}{2}$ and $x_2 = \frac{1-\sqrt{5}}{2}$

Substituting x_1 and x_2 in equation (4) we have

$$x(n) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^n + \beta\left(\frac{1-\sqrt{5}}{2}\right)^n \text{----------(6)}$$

From equation 1, we have two initial conditions: fib(0) = 0 and fib(1) = 1 which can be written as x(0) = 0 and x(1) = 1. Substituting these initial conditions one by one in equation (6), we have:

$$x(0) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^0 + \beta\left(\frac{1-\sqrt{5}}{2}\right)^0 = \alpha + \beta = 0 \text{----------(7)}$$

$$x(1) = \alpha\left(\frac{1+\sqrt{5}}{2}\right)^1 + \beta\left(\frac{1-\sqrt{5}}{2}\right)^1 = \alpha\left(\frac{1+\sqrt{5}}{2}\right) + \beta\left(\frac{1-\sqrt{5}}{2}\right) = 1 \text{-----(8)}$$

From equation (7),

$$\alpha = -\beta \text{.----------(9)}$$

Substituting this in equation 8, we have:

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right) + \beta\left(\frac{1-\sqrt{5}}{2}\right) = 1$$

$$-\beta\left(\frac{1+\sqrt{5}}{2}\right) + \beta\left(\frac{1-\sqrt{5}}{2}\right) = 1$$

$$\beta\left[\left(\frac{1-\sqrt{5}}{2}\right) - \left(\frac{1+\sqrt{5}}{2}\right)\right] = 1$$

$$\beta\left[\frac{1}{2} - \frac{\sqrt{5}}{2} - \frac{1}{2} - \frac{\sqrt{5}}{2}\right] = 1$$

$$\beta\left[-\sqrt{5}\right]=1$$

$$\beta=-\frac{1}{\sqrt{5}} \text{ ----------(10)}$$

Substituting $\beta=-\frac{1}{\sqrt{5}}$ in equation (9) we have $\alpha=\frac{1}{\sqrt{5}}$

Substituting the value of α and β in equation (6) we have,

$$x(n)=\alpha\left(\frac{1+\sqrt{5}}{2}\right)^n+\beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$x(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n=\frac{1}{\sqrt{5}}\left(\phi^n-\hat{\phi}^n\right) \text{ -----(11)}$$

where

$$\phi=\left(\frac{1+\sqrt{5}}{2}\right) \text{ and } \hat{\phi}=\left(\frac{1-\sqrt{5}}{2}\right)$$

Using the equation (11), we can find nth Fibonacci Number.

2.9 Other problems and solutions

This section provides the various problems and the corresponding solutions for a variety of problems.

Example 1: Show that $5x^2$ is $O(x^3)$

Solution: It is given that $f(x) = 5x^2$.

By definition, if $f(x) \le c*g(x)$ for $x \ge x_o$, then $f(x) \in O(g(x))$.

Here, $5x^2 \le 1 * x^3$ for $x \ge 5$ where $c = 1$, $g(x) = x^3$, $n_o = 5$

So, $f(x) \in O(x^3)$

Example 2: Solve the following recurrence relation:

$$x(n) = x(n-1) + 5 \text{ for } n > 1, x(1) = 0$$

Solution: The above recurrence relation can be written as shown below:

$$x(n) = \begin{cases} x(n-1) + 5 & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

Consider the relation when n > 1.

$x(n) = x(n-1) + 5$ ----------**(1)**

Note: In equation (1), replace n by n-1. We get $x(n-1) = x(n-2) + 5$. Substitute this value in equation (1).

$= x(n-2) + 5 + 5$

$= x(n-2) + 2*5$ ------**(2)**

Note: Replace n by n -2 in equation (1). We get $x(n-2) = x(n-3) + 5$. Substitute this value in equation (2)

$= x(n-3) + 5 + 2*5$

$= x(n-3) + 3*5$

llly, $= x(n-4) + 4*5$

$= x(n-5) + 5*5$

……..

……..

Finally,

$= x[n-(n-1)] + (n-1)5$

$= x(1) + (n-1)5$

$= 0 + (n-1)5$

$= \mathbf{5(n-1)}$

Example 3: Solve the following recurrence relation:

$$x(n) = 3x(n-1) \text{ for } n > 1, x(1) = 4$$

Solution: The above recurrence relation can be written as shown below:

$$x(n) = \begin{cases} 3x(n-1) & \text{if } n > 1 \\ 4 & \text{if } n = 1 \end{cases}$$

Consider the relation when n > 1. i.e.,

$x(n) = 3x(n-1)$ ----------(1)

$= 3[3x(n-2)]$ **Note:** Replacing n by n - 1 in equation (1), $x(n-1) = 3x(n-2)$. Substitute for x(n-1) in equation(1)

$= 3^2x(n-2)$ ----------(2)

$= 3^2[3x(n-3)]$ **Note:** Replacing n by n - 2 in equation (1) $x(n-2) = 3x(n-3)$. Substitute for x(n-2) in equation (2)

$= 3^3x(n-3)$

lly, $= 3^4x(n-4)$

$= 3^5x(n-5)$

.....

.....

Finally,

$= 3^{(n-1)}x[n-(n-1)]$

$= 3^{(n-1)}x(1)$

$= 3^{(n-1)}4$

$= \frac{3^n}{3} * 4$

$= \frac{4}{3} * 3^n$

Example 4: Prove that $3n^3 + 2n^2 = O(n^3)$

Solution: It is given that $f(n) = 3n^3 + 2n^2$. Replacing 2 with n (so that next higher order term is obtained), we get $3n^3 + n*n^2$ and call it c.g(n)

i.e., $\mathbf{c.g(n)} = 3n^3 + n*n^2$ for n = 2

$= 4n^3$ for n = 2

Now, the following constrain is satisfied:

$f(n) \le c * g(n)$ for $n \ge n_0$

i.e., $3n^3 + 2n^2 \le 4 * n^3$ for $n \ge 2$

It is clear from the above relations that c = 4, $\mathbf{g(n) = n^3}$ and $n_0 = 2$. So, by definition

$f(n) \in O(g(n))$ i.e., $\mathbf{f(n) \in O(n^3)}$

Example 5: Prove that $3n \neq O(2^n)$

Solution: Let $f(n) = 3^n$ and $g(n) = 2^n$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{3^n}{2^n} = \lim_{n\to\infty}\left(\frac{3}{2}\right)^n = \infty$$

By definition, ∞ indicates that f(n) has a larger order of growth when compared to g(n) and hence the relation $3^n \neq O(2^n)$ is true.

2.10 Problems on asymptotic notations

Example : Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

a) $n(n + 1)/2 \in O(n^3)$ b) $n(n + 1)/2 \in O(n^2)$

c) $n(n + 1)/2 \in \Theta(n^3)$ d) $n(n + 1)/2 \in \Omega(n)$

Solution: Part A:

a) Given: $n(n + 1)/2 \in O(n^3)$

$n^2/2 + n/2 \in O(n^3)$

order ⇩ ⇩ order

2 3

Since 2 has smaller order of growth than 3, we say that $n(n+1)/2 \in O(n^3)$

So, the relation $\boxed{n(n+1)/2 \in O(n^3) \text{ is true}}$

b) Given: $n(n + 1)/2 \in O(n^2)$

$n^2/2 + n/2 \in O(n^2)$

order ⇩ ⇩ order

2 2

Since n^2 has same order of growth than n^2, we say that $n(n+1)/2 \in O(n^2)$

So, the relation $\boxed{n(n+1)/2 \in O(n^2) \text{ is true}}$

c) Given: $n(n + 1)/2 \in \Theta(n^3)$

$$n^2/2 + n/2 \in \Theta(n^3)$$

order $\Downarrow$ $\qquad$ $\Downarrow$ order

2 $\qquad$ 3

Since n^2 has smaller order of growth than n^3, we say that $n(n+1)/2 \in O(n^3)$. But, it is given that:

$$n(n+1)/2 \in \Theta(n^3)$$

So, the relation $\boxed{n(n+1)/2 \in \Theta(n^3) \text{ is false}}$

d) Given: $n(n+1)/2 \in \Omega(n)$

$$n^2/2 + n/2 \in \Omega(n)$$

order $\Downarrow$ $\qquad$ $\Downarrow$ order

2 $\qquad$ 1

Since 2 has larger order of growth than 1, we say that $n(n+1)/2 \in \Omega(n)$

So, the relation $\boxed{n(n+1)/2 \in \Omega(n) \text{ is true}}$

Example: For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. (Use the simplest g(n) possible in your answers.) Prove your assertions.

a) $(n^2+1)^{10}$

b) $\sqrt{10n^2 + 7n + 3}$

c) $2n\lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$

d) $2^{n+1} + 3^{n-1}$

e) $\log_2 n$

Solution: part A: Given $(n^2+1)^{10}$

Informally, $(n^2+1)^{10} \approx (n^2)^{10} = n^{20} \in \Theta(n^{20})$. So, $\boxed{(n^2+1)^{10} \in \Theta(n^{20})}$

Assertion: Let $f(n) = (n^2+1)^{10}$ and $g(n) = n^{20}$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{(n^2+1)^{10}}{n^{20}} = \lim_{n\to\infty}\frac{(n^2+1)^{10}}{(n^2)^{10}} = \lim_{n\to\infty}\left(\frac{n^2+1}{n^2}\right)^{10} = \lim_{n\to\infty}\left(1+\frac{1}{n^2}\right)^{10} = 1$$

So, $\lim_{n\to\infty}\frac{f(n)}{g(n)} = 1$ i.e., $\lim_{n\to\infty}\frac{f(n)}{g(n)} = c$(constant).

By definition, $f(n) \in \Theta(g(n))$. So, $\boxed{(n^2+1)^{10} \in \Theta(n^{20}) \text{ is proved.}}$

Solution: part B: Given $\sqrt{10n^2 + 7n + 3}$

Informally, $\sqrt{10n^2 + 7n + 3} \approx \sqrt{10n^2} = \sqrt{10}\, n \in \Theta(n)$.

So, $\boxed{\sqrt{10n^2 + 7n + 3} \in \Theta(n)}$

Assertion: Let $f(n) = \sqrt{10n^2 + 7n + 3}$ $\quad g(n) = n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{10n^2 + 7n + 3}}{n}$$

$$= \lim_{n\to\infty} \sqrt{\frac{10n^2 + 7n + 3}{n^2}}$$

$$= \lim_{n\to\infty} \sqrt{10 + \frac{7}{n} + \frac{3}{n^2}} = \sqrt{10}$$

So, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \sqrt{10}$ i.e., $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ (constant).

By definition, $f(n) \in \Theta(g(n))$. So, $\boxed{\sqrt{10n^2 + 7n + 3} \in \Theta(n) \text{ is proved.}}$

Solution: part c: Given: $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$

Informally, $2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2}$

$$= 2n\, 2\lg(n+2) + (n+2)^2 (\lg n - \lg 2)$$

$$= 4n \lg(n+2) + (n+2)^2 (\lg n - 1) \quad \ldots\ldots\ldots\ldots\ldots\ldots(1)$$

$$\approx n \lg(n) + n^2 \lg n \quad \text{(For a constant multiple when n is very large}$$

$$\approx n^2 \lg n$$

So, $\boxed{2n \lg(n+2)^2 + (n+2)^2 \lg \frac{n}{2} \in \Theta(n^2 \lg n)}$

Assertion: Let $f(n) = 2n\lg(n+2)^2 + (n+2)^2 \lg\frac{n}{2}$ and $g(n) = n^2 \lg n$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{2n\lg(n+2)^2 + (n+2)^2\lg\frac{n}{2}}{n^2\lg n}$$

Replacing numerator by eq(1) we have

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{4n\lg(n+2) + (n+2)^2(\lg n - 1)}{n^2\lg n}$$

(simplification is left to the reader)

Solution: part d: Given: $2^{n+1} + 3^{n-1}$

$$\text{Informally, } 2^{n+1} + 3^{n-1} = 2^n * 2 + \frac{3^n}{3} \in \theta(2^n) + \theta(3^n) = \theta(3^n)$$

$$\text{So, } \boxed{2^{n+1} + 3^{n-1} \in \Theta(3^n)}$$

Assertion: Let $f(n) = 2^{n+1} + 3^{n-1}$ and $g(n) = 3^n$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{2^{n+1}+3^{n-1}}{3^n} = \lim_{n\to\infty}\frac{2*2^n}{3^n} + \frac{\frac{3^n}{3}}{3^n}$$

$$= \lim_{n\to\infty}\frac{2*2^n}{3^n} + \frac{3^n}{3*3^n} = \lim_{n\to\infty}\frac{2*2^n}{3^n} + \frac{1}{3} = \frac{1}{3}$$

So, $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \frac{1}{3}$ i.e., $\lim_{n\to\infty}\frac{f(n)}{g(n)} = c$ (constant).

By definition, $f(n) \in \Theta(g(n))$. So, $\boxed{2^{n+1} + 3^{n-1} \in \Theta(3^n) \text{ is proved.}}$

Solution: part e: Given: $\log_2 n$

i.e., $\boxed{\log_2 n \in \Theta(\log n)}$ By neglecting the constant

Assertion: Let $f(n) = \log_2 n$ and $g(n) = \log n$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{\log_2 n}{\log n} \quad \text{(Left as exercise)}$$

Example: a) The values of several functions that often arise in analysis of algorithms such as log n, n, n log n, n^2, n^3, 2^n, n! are listed in increasing order of their order of growth. Do these values prove this fact with mathematical certainty?

b) Prove that the functions are indeed listed in increasing order of their order of growth.

Solution: Part A: The order of growth and the related notations O, Ω, and Θ deal with the asymptotic behavior of functions as n goes to infinity i.e., practically for very large value of n. Therefore no specific values of functions within a finite range of n's values, can establish their orders of growth with mathematical certainty. So, the above facts can not be proved with mathematical certainty for a finite range of n's values.

Solution: Part B: We can prove that the functions:

$$\log n,\ n,\ n \log n,\ n^2,\ n^3,\ 2^n,\ n!$$

are listed in increasing order of their order of growth. This can be done by dividing the previous term by current term and simplify using limits as n tends to infinity and proving that the result is 0.

<u>We find relation between **log n** and **n**</u>

So, consider $\lim_{n\to\infty}\frac{\log n}{n} = \lim_{n\to\infty}\frac{\log_2 n}{n}$ **Note: log n = $\log_2$ n**

$$= \lim_{n\to\infty}\frac{(\log_2 n)'}{(n)'}$$

As per L'Hospital's rule, $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f'(n)}{g'(n)}$

Since $\log_a b = \frac{\log_e b}{\log_e a}$ the above relation can be written as shown below

$$= \lim_{n\to\infty}\frac{\left(\frac{\log_e n}{\log_e 2}\right)'}{(n)'} = \lim_{n\to\infty}\frac{(\log_2 e * \log_e n)'}{(n)'} \qquad \text{since } \frac{1}{\log_e a} = \log_a e$$

$$= \log_2 e \lim_{n\to\infty}\frac{(\log_e n)'}{(n)'} = \log_2 e \lim_{n\to\infty}\frac{\frac{1}{n}}{1}$$

$$= \log_2 e \lim_{n\to\infty}\frac{1}{n} = \log_2 e \lim_{n\to\infty}\frac{1}{\infty} = 0$$

So, log n < n .....(1)

2.58 Analysis of algorithm efficiency

Now, we find relation between **n** and **n log n**

So, consider $\lim_{n\to\infty}\frac{n}{n*\log n}=\lim_{n\to\infty}\frac{n}{n*\log_2 n}=\lim_{n\to\infty}\frac{1}{\log_2 n}=0$

So, n < n log n .....(2)

Now, we find relation between **n log n** and **n²**

So, consider $\lim_{n\to\infty}\frac{n*\log n}{n^2}=\lim_{n\to\infty}\frac{\log n}{n}=0$ see how we get relation (1)

So, n log n < n² .....(3)

Now, we find relation between **n²** and **n³**

So, consider $\lim_{n\to\infty}\frac{n^2}{n^3}=\lim_{n\to\infty}\frac{1}{n}=0$

So, n² < n³ .....(4)

Now, we find relation between **n³** and **2ⁿ**

So, consider $\lim_{n\to\infty}\frac{n^3}{2^n}$

$$=\lim_{n\to\infty}\frac{(n^3)'}{(2^n)'}$$ (According to L'Hospital's rule)

$$=\lim_{n\to\infty}\frac{3n^2}{\lg 2*2^n}$$ since $\frac{d}{dx}(2^n)=\ln 2(2^n)$

$$=\frac{3}{\lg 2}\lim_{n\to\infty}\frac{n^2}{2^n}$$

$$=\frac{3}{\lg 2}\lim_{n\to\infty}\frac{(n^2)'}{(2^n)'}$$ (According to L'Hospital's rule)

$$=\frac{3}{\lg 2}\lim_{n\to\infty}\frac{2n}{\lg 2*2^n}=\frac{3}{\lg 2}*\frac{2}{\lg 2}\lim_{n\to\infty}\frac{n}{2^n}=\frac{6}{(\lg 2)^2}\lim_{n\to\infty}\frac{n}{2^n}$$

$$=\frac{6}{(\lg 2)^2}\lim_{n\to\infty}\frac{n'}{(2^n)'}$$ (According to L'Hospital's rule)

$$=\frac{6}{(\lg 2)^2}\lim_{n\to\infty}\frac{1}{\lg 2*2^n}=\frac{6}{(\lg 2)^3}\lim_{n\to\infty}\frac{1}{2^n}=\frac{6}{(\lg 2)^3}*0=0$$

$$\boxed{\text{So, } n^3 < 2^n \text{ .....(5)}}$$

Now, we find relation between 2^n and **n!**

So, consider $\lim_{n\to\infty}\frac{2^n}{n!}$

$$= \lim_{n\to\infty}\frac{2^n}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}$$

Note: Striling's formula is given by $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$$= \frac{1}{\sqrt{2\pi}}\lim_{n\to\infty}\frac{2^n}{\sqrt{n}\left(\frac{n}{e}\right)^n} = \frac{1}{\sqrt{2\pi}}\lim_{n\to\infty}\frac{(2e)^n}{\sqrt{n}*n^n}$$

$$= \frac{1}{\sqrt{2\pi}}*0$$

Since $(2e)^n$ is far less than $\sqrt{n}*n^n$

$$= 0$$

$$\boxed{\text{So, } 2^n < n! \text{ .....(6)}}$$

Thus, using the relations from (1) to (6) we have proved that

$$\boxed{\log n < n < n\log n < n^2 < n^3 < 2^n < n!}$$

Example: Order the following functions according to their order of growth (from the lowest to the highest):

$$(n-2)!,\ 5\lg(n+100)^{10},\ 2^{2n},\ 0.001n^4+3n^3+1,\ \ln^2 n,\ \sqrt[3]{n},\ 3^n$$

Solution: The following steps are followed while arranging the above functions in increasing order of growth :

$(n-2)! = (n-2)!$

$5\lg(n+100)^{10} = \log n^{10}$

$2^{2n} = 2^{2n}$

$0.001n^4+3n^3+1 = n^4$

$\ln^2 n = \log^2 n$

$\sqrt[3]{n} = n^{3/2}$

$3^n = 3^n$

The above functions now can be arranged in increasing order as shown below:

$(n-2)! \quad = (n-2)!$
$5\lg(n+100)^{10} \quad = \log n^{10}$
$2^{2n} \quad = 2^{2n}$
$0.001n^4+3n^3+1 = n^4$
$\ln^2 n \quad = \log^2 n$
$\sqrt[3]{n} \quad = n^{3/2}$
$3^n \quad = 3^n$

$$\log n^{10} < \log^2 n < n^{3/2} < n^4 < 3^n < 2^{2n} < (n-2)!$$

$$\boxed{5\lg(n+100)^{10},\ \ln^2 n,\ \sqrt[3]{n},\ 0.001n^4+3n^3+1,\ 3^n,\ 2^{2n},\ (n-2)!}$$

Example: Prove that every polynomial of degree k, $p(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots\ldots + a_0$, with $a_k > 0$ belongs to $\Theta(n^k)$.

Solution: It is given that $p(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots\ldots + a_0$, with $a_k > 0$. It is required to prove that f(n) belongs to $\Theta(n^k)$

So, let us compute: $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{p(n)}{n^k}$ .....(1)

$$= \lim_{n\to\infty}\frac{a_k n^k + a_{k-1}n^{k-1} + \ldots\ldots\ldots a_2 n^2 + a_1 n^1 + a_0}{n^k}$$

$$= \lim_{n\to\infty}\left(a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \ldots\ldots \frac{a_2}{n^{k-2}} + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}\right)$$

$= a_k$ (a constant) imply f(n) has same order of growth than g(n)

i.e., $f(n) \in \theta(g(n))$ by asymptotic definition using limits

i.e., $p(n) \in \theta(n^k)$ using relation (1)

So, $\boxed{p(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots\ldots + a_0 \in \Theta(n^k)}$

Example: Prove that exponential functions a^n have different orders of growth for different values of base $a > 0$.

Solution: Consider the two functions a_1^n and a_2^n. To see that they have different orders of growth for different values of base $a > 0$, let us compute:

$$\boxed{\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{a_1^n}{a_2^n}}$$

Case 1: If $a_1 < a_2$, then $\lim_{n\to\infty}\frac{a_1^n}{a_2^n} = 0$. According to asymptotic definition using limits, this imply that $f(n) \in O(g(n))$

i.e., $\boxed{a_1^n \in O(a_2^n)}$ .....(1)

Case 2: If $a_1 = a_2$, then $\lim_{n\to\infty}\frac{a_1^n}{a_2^n} = 1$. According to asymptotic definition using limits, this imply that $f(n) \in \Theta(g(n))$

i.e., $\boxed{a_1^n \in \Theta(a_2^n)}$ .....(2)

Case 3: If $a_1 > a_2$, then $\lim_{n\to\infty}\frac{a_1^n}{a_2^n} = \infty$. According to asymptotic definition using limits, this imply that $f(n) \in \Omega(g(n))$

i.e., $\boxed{a_1^n \in \Omega(a_2^n)}$ .....(2)

By observing relations in case 1, case 2 and case 3 it is proved that exponential functions a^n have different orders of growth for different values of base $a > 0$.

Exercises:

1. What are the various factors that affect the program performance (How the efficiency of an algorithm is measured)?
2. Explain space complexity and time complexity
3. What are the various factors that affect the execution time?
4. Explain time complexity with respect to worst case, best case and average case efficiency.
5. Obtain an algorithm for linear search. Obtain the worst case, best case and average case for successful and unsuccessful search

6. Explain the concept of asymptotic notations, indicating the normally used notations. **(Jan/Feb 2005)**
7. Explain all Asymptotic notations used in the analysis of algorithms
8. What are various asymptotic notations? Explain with examples
9. Define O, θ and Ω notations. **(July/August 2004)**
10. Consider the polynomial form $f(n) = \sum_{i=0}^{m} a_i n^i$ where $a_m > 0$. Show that $f(n) = O(n^m)$.
11. Let f(n) = 100n+5. Express f(n) using Big-Omega.
12. Let $f(n) = 10n^3+5$. Express f(n) using Big-Omega.
13. Let $f(n) = 6*2^n + n^2$. Express f(n) using Big-Oh.
14. Let f(n) = 100n+5. Express f(n) using Big-theta.
15. Prove that $3n^3 + 2n^2 = O(n^3)$ and $3^n \neq O(2^n)$ **(July/Aug 2004)**
16. If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then shown that $T_1(n) + T_2(n) = O(\max(f(n), g(n))$ **(July/Aug 2004)**
17. Order the following functions according to their order of growth (from lowest to highest)

$$(n-2)!,\ 5\log_{10}(n+100)^{10},\ 2^{2n},\ \log^2_e n,\ \sqrt{n},\ 3^n$$

18. Design an algorithm to obtain maximum of N elements and obtain the time complexity
19. Write a recursive algorithm to compute 2^n for any non-negative integer n which is based on the formula $2^n = 2^{n-1} + 2^{n-1}$
20. Estimate how many times faster it will be to find GCD(31415, 14142) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from min{m,n} down to gcd(m,n)
21. Obtain an algorithm to check whether the array elements are unique or not. Obtain the worst case time complexity.
22. Write the general plan for analyzing non-recursive algorithms
23. Suggest a general plan for analyzing the efficiency of non recursive algorithms. Suggest an algorithm to find whether the elements in an array are unique. Analyze it's efficiency using the method suggested by you. **(Jan/Feb 2005)**
24. Write an algorithm to multiply two matrices and obtain the time complexity
25. Write an algorithm to obtain a factorial of a number using recursion and obtain the corresponding recurrence relation to obtain the number of multiplications and then solve the recurrence relation.
26. Solve the following recurrence relation:

$$x(n) = x(n-1) + 5 \text{ for } n > 1,\ x(1) = 0$$

Chapter 3: Brute Force

What are we studying in this chapter?

- Brute Force technique
- Selection sort
- Bubble sort
- Sequential search
- Brute force string matching
- To find the mode
- Exhaustive search
  - Traveling Salesman Problem
  - Knapsack problem
  - Assignment Problem

- 6 hours

3.1 Introduction

There are various approaches to solve a given problem. In a straight forward approach and simple technique such as *brute force,* there is no emphasis on the efficiency of the algorithm. The concept used in this technique is "Just do it". In this chapter let us concentrate on "How the *Brute Force method* is used to design the algorithms?" Along with these algorithms, let us obtain the time efficiency of all the algorithms. Now, let us see "What is brute force method? What are the various problems that can be solved using this method?"

Definition: The straight forward method of solving a given problem based on the problem's statement and definitions is called *Brute Force* technique. This method is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

For example, the algorithms to find GCD of 2 numbers (Consecutive integer checking method, Matrix multiplication, addition etc, Selection sort, Bubble sort, Linear search (sequential search), Brute-force string matching, Traveling sales man problem and Knapsack problem can be solved using brute force technique.

Now, let us see "What are the advantages and disadvantages of algorithms with brute force technique?" The various advantages and disadvantages of this approach are shown below:

Advantages

- This method is applicable for wide variety of problems such as finding the sum of n numbers, Computing power, Computing GCD and so on
- Simple and easy algorithms can be written. For example, bubble sort, selection sort, matrix multiplication etc
- Can be used to judge more efficient alternative approaches to solve a problem

Disadvantages

- Rarely yields efficient algorithms
- Some brute force algorithms are unacceptably slow: For example, bubble sort
- Not as constructive/creative as other design techniques such as divide and conquer

3.2 Bubble sort

This is one of the simplest sorting technique and most straightforward method of sorting. In this brute-force technique, the adjacent elements in the list are compared and exchanged if they are out of order. Now, let us "Design the algorithm for bubble sort and find the time efficiency"

Design: This is the simplest and easiest sorting technique. In this technique, the two successive items A[i] and A[i+1] are exchanged whenever A[i] > A[i+1]. For example, consider the elements shown below:

40, 50, 30, 20, 10

The elements can be sorted as shown below:

	1st Pass				2nd Pass			3rd Pass		4th Pass
A[0] = **40**	40	40	40	**40**	30	30	**30**	20	**20**	**10**
A[1] = **50**	**50**	30	30	**30**	**40**	20	**20**	**30**	**10**	**20**
A[2] = 30	**30**	**50**	20	20	**20**	**40**	10	**10**	**30**	**30**
A[3] = 20	**20**	20	50	**10**	**10**	10	40	**40**	40	**40**
A[4] = 10	10	10	**10**	**50**	50	50	**50**	50	**50**	**50**
Given array	**50** sinks to bottom after pass 1				**40** sinks to bottom after pass 2			**30** sinks to bottom after pass 3		**20** sinks to bottom

Note: Observe that after each pass, the larger values sink to the bottom of the array and hence it is called *sinking sort*.

In the first pass 40 is compared with 50 and they are in order. So, no exchange has been done. Next 50 is compared with 30 and they are exchanged since 50 is greater than 30. If we proceed in the same manner, at the end of the first pass the largest item occupies the last position. On each successive pass, the items with the next largest value will be moved to the bottom and thus elements are arranged in ascending order.

The following figure below shows the output of each pass.

Original Array	Out put of each pass 1st	2nd	3rd	4th
A[0] = 40	40	30	20	10
A[1] = 50	30	20	10	20
A[2] = 30	20	10	30	30
A[3] = 20	**10**	**40**	**40**	**40**
A[4] = 10	50	50	50	50

Note: Observe that at the end of each pass smaller values gradually "bubble" their way upward to the top (like air bubbles moving to surface of water) and hence called ***bubble sort***.

Now we concentrate on the designing aspects of this sorting technique. The comparisons that are performed in each pass are shown below:

Passes →	Pass-1	Pass-2	Pass 3	Pass 4
	0 – 1	0 – 1	0 – 1	0 – 1
	1 – 2	1 – 2	1 – 2	
Comparing	2 – 3	2 – 3		
	3 – 4			
	i=0 to 3	i=0 to 2	i=0 to 1	i=0 to 0
	5-2	5-3	5-4	5-5
	5-(1+1)	5-(2+1)	5-(3+1)	5-(4+1)
	n- (j+1)	n –(j+1)	n-(j+1)	n-(j+1)

Note: The index i = 0 to n-(j+1) or **i = 0 to n-j-1**
The index j = 1 to 4 or j = 1 to 5-1. In general, **j = 1 to n-1**

So, the partial code can be written as:

```
for j = 1 to n-1
    for i = 0 to n-j-1
        if ( A[i] > A[i +1] )
            exchange ( A[i] , A[i+1] )
        end if
    end for
end for
```

So, the complete bubble sort algorithm is shown below:

```
Algorithm BubbleSort(a[], n)

//Purpose: Arrange the numbers in ascending order
// Inputs :
//      n – the number of items present in the table
//      a – the items to be sorted are present in the table
//Output:
//      a: contains the sorted list

for j ← 1 to n – 1 do                   // Perform n-1 passes
    for i ← 0 to n-j-1do                // To compare items in each pass
        if ( a[i] > a[i+1] )            // If out of order exchange adjacent elements
            temp ← a[i]
            a[i] ← a[i+1]
            a[i+1] ←temp
        end if
    end for
end for
```

Advantages of bubble sort

- Very simple and easy to program
- Straight forward approach

Disadvantages of bubble sort

- It runs slowly and hence it is not efficient. More efficient sorting techniques are present
- Even if the elements are sorted, n-1 passes are required to sort.

Analysis The time efficiency of bubble sort algorithm can be obtained as shown below:

Step 1: The parameter to be considered is n, which represent size of the input

Step 2: The basic operation is the statement "if a[i] >= a[i+1]" in the innermost for loop

Step 3: The number of comparisons depends on the value of n and the number of times the two for loops are executed. The total number of times the basic operation is executed can be obtained as shown below:

```
for j ← 1 to n – 1 do                  // Perform n-1 passes
    for i ← 0 to n-j-1 do              // To compare items in each pass
        if ( a[i] > a[i+1] )           // If out of order exchange adjacent elements
```

$$C(n) = \sum_{j=1}^{n-1} \sum_{i=0}^{n-j-1} 1$$

$$= \sum_{j=1}^{n-1} n-j-1-0+1 \text{ (Note: upper bound – lower bound + 1)}$$

$$= \sum_{j=1}^{n-1} n-j$$

$$= (n-1) + (n-2) + \ldots\ldots 3 + 2 + 1 \text{ (After substituting for j from 1 to n-1)}$$

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

Note: Whether the elements are sorted or not, this algorithm requires $\frac{(n-1)n}{2}$ comparisons. So, the best case and worst case time efficiency remains same. So, let us express using θ notation.

Step 4: Let us express f(n) using θ-notation. The constraint to be satisfied is:

$$c_1 * g(n) \le f(n) \le c_2 * g(n) \quad \text{for } n \ge n_0$$

$$\frac{1}{4} * n^2 \le \frac{n^2}{2} - \frac{n}{2} \le 1 * n^2 \quad \text{for } n \ge 2$$

It is clear from the above relations that $c_1 = 1/4$, $c_2 = 1$, $n_0 = 2$, $g(n) = n^2$. So, by definition

$$f(n) \in \theta(g(n)) \text{ i.e., } \quad f(n) \in \theta(n^2)$$

So, time complexity of bubble sort = $\theta(n^2)$

Let us see "What happens when elements are already sorted?" In the given algorithm, even if the elements are sorted, the algorithm has to perform all (n-1) passes requiring total of n(n-1)/2 comparisons.

Note: To improve the efficiency of this function, it is required to find whether the items are already in ascending order in the current pass. In any specific pass say *j*, if the elements are already sorted, there is no need to go for the next pass. This can be achieved by using a *flag* and setting *flag* to 0 just before the jth pass. In the jth pass, if any exchange is made, set *flag* to 1. After j^{th} pass is over, check for *flag*. If *flag* has the value 0, it means that items are already sorted and we terminate the loop. Otherwise, items are not sorted and go for the next pass. The efficient bubble sort algorithm is shown below:

```
Algorithm BubbleSort(a[], n)

//Purpose: To arrange the numbers in ascending order

// Inputs :
//        n – the number of items present in the table
//        a – contains unsorted elements

//Output:
//        a – Contains the sorted elements

for j ← 1 to n – 1 do                  // Perform n-1 passes
      flag ← 0                         // Initially, no exchange has taken place

      for i ← 0 to n-j-1do             // To compare items in each pass
            if ( a[i] > a[i+1] )       // If out of order exchange adjacent elements
                  temp ← a[i]
                  a[i] ← a[i+1]
                  a[i+1] ← temp
                  flag ← 1;            // Exchange has taken place
            end if
      end for

      if (flag = 0) break;             // If no exchange, the items are already sorted
end for
```

Analysis The time efficiency of efficient bubble sort can be obtained as shown below:

Step 1: The parameter to be considered is *n* which represent the size of the input.

Step 2: The basic operation in this algorithm is the statement **"if (a[i] > a[i+1])"**
Step 3: If the items are already sorted, the basic operation is executed only in pass 1. In the worst case, sorting requires all the passes. So, it is required to calculate the best case and worst case time efficiencies.

Best case analysis: The best case occurs when the elements are already in ascending order. At least one pass is required to check whether the items are sorted. So, the total number of times the basic operation is executed can be obtained as shown below:

for i ← 0 **to** n-j-1 **do** **Note: Replace j by 1:** If all the items are sorted
if (a[i] > a[i+1])

$$f(n) = \sum_{i=0}^{n-2} 1$$

$$= n-2-0+1 = n-1$$

Thus, **f(n) = n-1.** In the best case, time efficiency is expressed using Ω-notation. The constraint to be satisfied

$$f(n) \geq c * g(n) \quad \text{for } n \geq n_0$$

$$\text{i.e., } n-1 \geq 1 * n \quad \text{for } n \geq 0$$

It is clear from the above relations that c = 1, g(n) = n and n_0 = 0. So, by definition

$$f(n) \in \Omega(g(n)) \text{ i.e., } \quad f(n) \in \Omega(n)$$

So, time complexity of efficient bubble sort in the best case is given by Ω(n)

Worst case analysis: The total number of times the basic operation is executed is obtained by considering all the passes as shown below:

for j ← 1 **to** n – 1 **do** // Perform n-1 passes
for i ← 0 **to** n-j-1 **do** // To compare items in each pass
if (a[i] > a[i+1]) // If out of order exchange adjacent elements

$$C(n) = \sum_{j=1}^{n-1} \sum_{i=0}^{n-j-1} 1$$

$$= \sum_{j=1}^{n-1} n-j-1-0+1 \text{ (Note: upper bound – lower bound + 1)}$$

$$= \sum_{j=1}^{n-1} n-j$$

$= (n-1) + (n-2) + ...3 + 2 + 1$ (After substituting for j from 1 to n-1)

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

The worst case time efficiency is expressed using O-notation. For this, the constraint to be satisfied is:

$$f(n) \le c * g(n) \quad \text{for } n \ge n_0$$

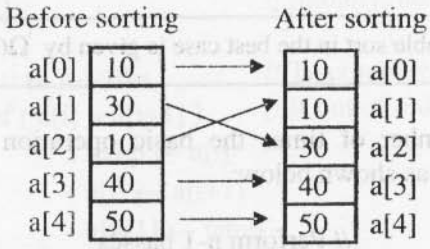
$$\frac{n^2}{2} - \frac{n}{2} \le 1 * n^2 \quad \text{for } n \ge 2$$

It is clear from the above relations that $c = 1$, $n_0 = 2$, $g(n) = n^2$. So, by definition

$$f(n) \in O(g(n)) \text{ i.e., } \quad f(n) \in O(n^2)$$

So, time complexity of efficient bubble sort in the worst case is given by= $O(n^2)$

Now, the question is "Is the bubble sort stable?" The bubble sort algorithm is stable because this algorithm preserves the relative order of any two equal elements. For example, consider the given array and sorted array as shown below:



Observe that position of first 10 is less than position of second 10 before sorting and after sorting also position of first 10 is less than position of second 10 i.e., the relative order of two equal elements has not been changed. So, bubble sort algorithm is stable.

Now, the question is "Is the bubble sort in place?" The bubble sort algorithm uses the variables i, j and temp apart from the two variables:

- n: which represent the number of items to sort
- a: contains the elements to be sorted

Apart from the variables *n* and *a*, the memory units used by the variables *i, j* and *temp* are very less and hence the algorithm is *in place*.

3.3 Selection sort (Straight selection sort or pushdown sort)

Before writing the algorithm, let us know the answer for "*What is the concept used in selection sort?*"

Procedure: As the name indicates, we first find the smallest item in the list and we exchange it with the first item. Obtain the second smallest in the list and exchange it with the second element and so on. Finally, all the items will be arranged in ascending order. Since, the next least item is selected and exchanged appropriately so that elements are finally sorted, this technique is called *Selection sort*. Let us see, how the elements 45 20 40 5 15 can be sorted using selection sort.

Given items	After pass 1	After pass 2	After pass 3	After pass 4
A[0] = **45**	5	5	5	5
A[1] = 20	**20**	15	15	15
A[2] = 40	40	**40**	20	20
A[3] = **5**	45	45	**45**	40
A[4] = 15	**15**	**20**	**40**	45
1st smallest is 5. Exchange it with 1st item	2nd smallest is 15. Exchange it with 2nd item	3rd smallest is 20. Exchange it with 3rd item	4th smallest is 40. Exchange it with 4th item	All elements are sorted

Design: The smallest element from i^{th} position onwards can be obtained using the following code.

```
pos ← i
for j ← i+1 to n-1
    if ( a[j] < a[pos] ) pos ← j
end for
```

After finding the position of the smallest number, it should be exchanged with i^{th} position. The equivalent statements are shown below:

```
temp ← a[pos]
a[pos] ← a[i]
a[i] ← temp
```

3.10 Brute Force

The selection sort algorithm is shown below:

```
Algorithm SelectionSort(a[], n)

//Purpose: Sort the given elements using selection sort

// Inputs :
//      n – the number of items present in the array
//      a – the items to be sorted are present in the array
//Output:
//      a – contains the sorted list

for i ← 0 to n-2 do

    pos ← i;                    //Assume ith element as smallest
    for j ← i +1 to n-1 do      // Find the position of the smallest item
        if ( a[j] < a[pos]) pos ← j;
    end for

    temp ← a[pos]               //Exchange ith item with  smallest elelment
    a[pos] ← a[i]
    a[i] ← temp
end for
```

Analysis The time efficiency of selection sort can be calculated as shown below:

Step 1: The parameter to be considered is *n*, which represent size of the input

Step 2: The basic operation is the comparative statement "if a[j] < a[pos]" in the innermost for loop

Step 3: The number of comparisons depends on the value of *n* and the number of times the two for loops are executed. Here, best case and worst case remains same. It is clear from the two for loops that, the number of basic operations f(n) can be obtained as shown below:

```
for i ← 0 to n-2 do
    for j ← i +1 to n-1 do
        if ( a[j] < a[pos]) pos ← j
```

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$ (**Note**: one comparison a[j] < a[pos] for iteration)

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

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= (n-1)+(n-2)+.....3+2+1$$

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

$$\text{So, } f(n) = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Let us express f(n) using θ-notation. The constraint to be satisfied is:

$$c_1 * g(n) \le f(n) \le c_2 * g(n) \quad \text{for } n \ge n_0$$

$$\frac{1}{4} * n^2 \le \frac{n^2}{2} - \frac{n}{2} \le 1 * n^2 \quad \text{for } n \ge 2$$

It is clear from the above relations that $c_1 = 1/4$, $c_2 = 1$, $n_0 = 2$, $g(n) = n^2$. So, by definition

$$f(n) \in \theta(g(n)) \text{ i.e., } \quad f(n) \in \theta(n^2)$$

So, time complexity of selection sort = $\theta(n^2)$

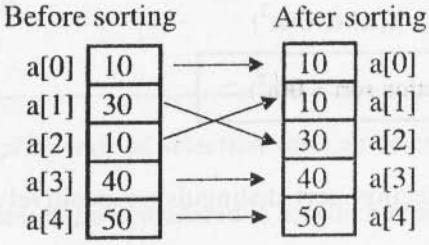
Let us see "What property of selection sort distinguishes positively from many other sorting algorithms?"

The time efficiency of selection sort is $\theta(n^2)$. But, there are so many other sorting techniques which has the same time efficiency or better efficiency. Even then, if we consider the number of swaps (or exchanges) made, the number of swaps in all the passes will be n-1, both in the worst case and best case. That is, time efficiency of selection sort with respect to swaps is $\theta(n)$. This property distinguishes *selection sort* positively from many other sorting algorithms.

Let us "Sort the list **E, X, A, M, P, L, E** in alphabetical order by selection sort" The selection sort algorithm requires n-1 passes where *n* represent number of elements to sort. Since there are 7 items in the given list, the selection sort requires 6 passes. In each pass, find the next smallest item and exchange with the item at the position specified by pass number. The given list can be sorted using selection sort as shown below:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	Range	Pass	Least	Action
E	X	A	M	P	L	E	0 to 6	0	A	swap a[0],a[2]
A	X	E	M	P	L	E	1 to 6	1	E	swap a[1],a[2]
A	E	X	M	P	L	E	2 to 6	2	E	swap a[2],a[6]
A	E	E	M	P	L	X	3 to 6	3	L	swap a[3],a[5]
A	E	E	L	P	M	X	4 to 6	4	M	swap a[4],a[5]
A	E	E	L	M	P	X	5 to 6	5	P	swap a[5],a[5]
A	E	E	L	M	P	X	All elements are sorted			

Now, the question is "Is the selection sort stable?" The selection sort algorithm is stable because this algorithm preserves the relative order of any two equal elements. For example, consider the given array and sorted array as shown below:



Observe that position of first 10 is less than position of second 10 before sorting. After sorting also position of first 10 is less than position of second 10 i.e., the relative order of two equal elements has not been changed. So, selection sort is stable.

Now, the question is "Is the selection sort in place?" The selection sort algorithm uses the variables i, j, pos and temp apart from the two variables:

- n: which represent the number of items to sort
- a: contains the elements to be sorted

Apart from the variables *n* and *a*, the memory units used by the variables *i, j, pos* and *temp* are very less and hence the algorithm is *in place*.

Advantages of selection sort

- Very simple and easy to understand
- Easy to implement the algorithm correctly and hence function can be written easily
- Straight forward approach

Disadvantages of selection sort

- Its time complexity is $O(N^2)$ and hence it is not efficient. More efficient sorting techniques are present whose time efficiency is N log N
- Even if the elements are sorted, n-1 passes are required.

3.4 Sequential search

Let us take an example of *linear search* which is also called *sequential search* to find the *worst case*, *best case* and *average case* efficiencies. In this technique, the search *key* is compared with each item sequentially one after the other. If the search *key* is found, the search is successful. If search *key* is not there we get unsuccessful search. This technique we have already seen in section 2.7.2 (Page 2.20). For easy access, the part of the algorithm is shown below:

```
for i ← 0 to n-1 do
        if ( a[i] = key )
                return i;        // Key Found at position i
        end if
end for
return i;                        // Key not found
```

Now, the question is "Is it possible to increase the efficiency of above algorithm? If yes how?"

The for-loop is executed n-1 times. This loop is used to check the end of the list. We can eliminate a check for the list's end on each iteration of the loop. This is achieved by appending the search key at the end of the list and then search for the key. Thus, the checking for end of list is eliminated and hence efficiency of algorithm is slightly increased. The corresponding algorithm is shown below:

Algorithm LinearSearch(key, a[], n):
//Purpose: This algorithm searches for the *key* in the array *a* which has *n* elements and
// *search key* acts as a delimiter
// Inputs :

```
//      n – the number of items present in the array
//      a – the items in the array where searching takes place
//      key – the item to be searched

//Output:
//      The function returns the position if key found;
//      Otherwise, the function returns -1 indicating search is unsuccessful

        a[n] ← key              // Insert search key at the end of the list

        i ← 0                   // search for the key in the given array
        while (a[i] ≠ key)
                i ← i + 1
        end while
        if (i = n) return -1    // search unsuccessful. Because, item found at n^th position
        return i                // Search is successful
```

Advantages

- Very simple and easy to understand
- Easy to code

Disadvantages

- Time efficiency is O(N) in the worst case. But, there are number of algorithms which work better than linear search

3.5 Pattern matching (brute-force string matching)

Now, let us see "What is string matching?"

Definition: Given a string called *text* with *n* characters and another string called *pattern* with *m* characters where $m \leq n$. It is required to search for the string *pattern* in the string *text*. If search is successful return the position of the first occurrence of *pattern* string in the *text* string. Otherwise, return -1. This process of searching for a *pattern* string in a given *text* string is called *string matching* or *pattern matching*.

Design: Align the *pattern* string against the first m characters of the *text* string and compare the characters of the *pattern* string with the characters of *text* string from left to right. In this process, if all characters are compared and found same, the pattern string is

present in the text string and the algorithm terminates. If there is a mismatch, the *pattern* string is shifted one position to the right and start comparing the *pattern* string with *text* string. **Note:** This procedure can be interpreted as sliding a *template* which contains the *pattern string* and slide over *text string* as long as the string shown in shaded region matches.

The sequence of steps to be followed while searching for a *pattern* string "UNCLE" in the *text* string "FUN-UNCLE" is shown below:

i

	0	1	2	3	4	5	6	7	8
text	F	U	N	-	U	N	C	L	E
pattern	U	N	C	L	E				
	0	1	2	3	4				

j

(a)

Note: The characters 'F' and 'U' are compared and they are not equal. So, slide the pattern string towards right as shown below:

i

	0	1	2	3	4	5	6	7	8
text	F	U	N	-	U	N	C	L	E
pattern		U	N	C	L	E			
		0	1	2	3	4			

j

(b)

Note: The characters 'U' and 'U', 'N' and 'N' are compared and found equal. But, there is a mismatch when '–' and 'C' are compared. So, slide the pattern string towards right as shown below:

i

	0	1	2	3	4	5	6	7	8
text	F	U	N	-	U	N	C	L	E
pattern			U	N	C	L	E		
			0	1	2	3	4		

j

(c)

Note: The characters 'N' and 'U' are compared and they are not equal. So, slide the pattern string towards right as shown below:

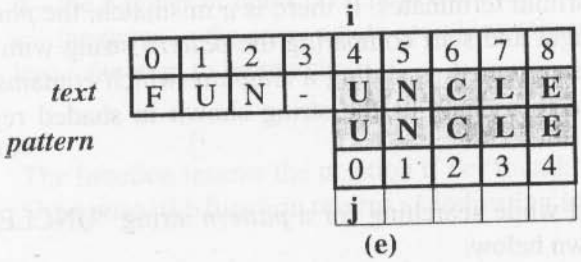
i

	0	1	2	3	4	5	6	7	8
text	F	U	N	-	U	N	C	L	E
pattern				U	N	C	L	E	
				0	1	2	3	4	

j

(d)

Note: The characters '–' and 'U' are compared and they are not equal. So, slide the pattern string towards right as shown below:



Note: The pattern string found and position of *i* has to be returned

Note: Observe from figures (a) through (e) that "sliding the pattern string towards right is nothing but incrementing the index i of the text string by 1"

Design: The entire sequence of operations explained so far can be pictorially represented as shown below:

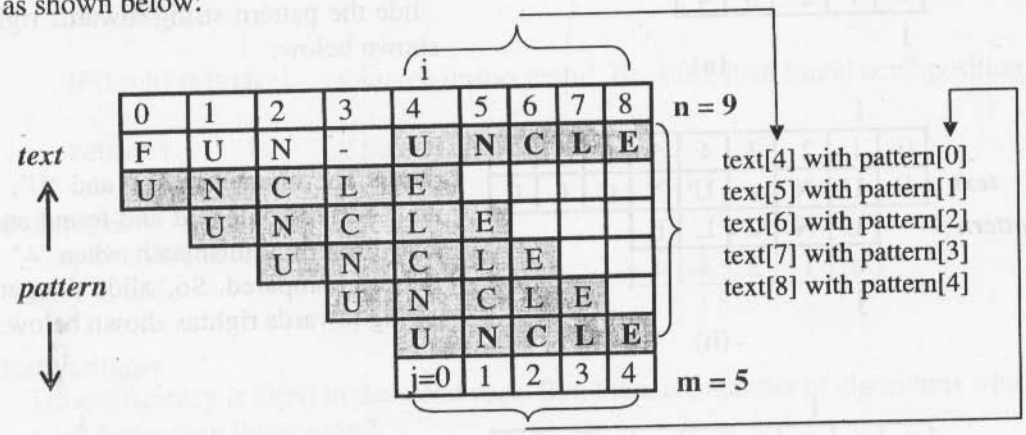


Figure 3.5. Complete trace for string matching

Consider the scenario shown in figure 3.5 with *pattern* string UNCLE being compared with the *text* string from position *i* whose index value is 4 (see above figure). It is clear from the figure that the following characters are compared:

text[4] with pattern[0]
text[5] with pattern[1]
text[6] with pattern[2]
text[7] with pattern[3]
text[8] with pattern[4]

Can be written as

text[4+0] with pattern[0]
text[4+1] with pattern[1]
text[4+2] with pattern[2]
text[4+3] with pattern[3]
text[4+4] with pattern[4]

text[i + j] with pattern[j]

Note: text[i+j] is compared with pattern[j], where j <= 4
j < 5
j < m (m = 5 is length of pattern string)

So, keep comparing **pattern[j] = text[i + j]** as long as j < m as shown in the following pseudocode with initial value of *j* as 0.

```
j ← 0
while j < m and pattern[j] = text[i + j]
        j ← j + 1
end while

if j = m return i;          // pattern string found at position i
```

Note: It is clear from the above statement that if *j* reaches *m*, the *pattern* string is found and the corresponding index *i* in the *text* string can be returned.

Note: Observe from figure 3.5 that the index value *i* = 4 is the last position in *text* string that can match a *pattern* string. Beyond this position, there are not enough number of characters in *text* string to compare with *pattern* string. So, i range from 0 to 4. This can be written as shown below:

i = 0 to 4
i = 0 to 9 – 5 (Expressing 4 in terms of length of text and pattern string)
In general, i = 0 to n – m

So, the modified pseudo code can be written as shown below:

Algorithm String_Match(pattern, text)

//**Purpose:** This algorithm searches for the *pattern* string in *text* string using brute-force string matching

// **Inputs :**

// pattern[]: is a string of *m* characters representing a *pattern* to be searched

// text[]: is a string of *n* characters representing a *text* string where searching has to be carried

// **Output:**

// returns i which is the position of *pattern* string in the *text* string if found

// returns -1 indicating *pattern* string is not present in the *text* string

```
n ← length(text)            // Obtain length of text string
m ← length(pattern)         // Obtain length of pattern string

for i ← 0 to n-m do
        j ← 0
        while j < m and pattern[j] = text[i + j]
                j ← j + 1
        end while
        if j = m return i   // Pattern string found
end for

return -1                   // Pattern string not found
```

Analysis The time efficiency of this algorithm can be computed as shown below:

Best case analysis: The best case occurs if the pattern string to be searched is present in the beginning of the text string. If *m* is the length of the *pattern* string, the number of comparisons required is *m*. Therefore, the best case time complexity is given by $\Omega(m)$

$$f(n) \in \Omega(m)$$

Worst case Analysis: The worst case occurs, if the pattern string is present at the end of the text string or if the pattern string is not present in the text string.

Step 1: The parameter to be considered are *m* and *n*, which represent the length of pattern string and text string.

Step 2: The basic operation is The basic operation is "pattern[j] = text[i+j]"

Step 3: The total number of times the basic operation is executed can be obtained as shown below:

```
for i ← 0 to n – m do
        j ← 0
                while (j < m) and (pattern[j] = text[i + j] )
                        .......
                        .......
```

The above program segment can be written as shown below:

```
for i ← 0 to n – m do
    for j ← 0 to m-1 do
        if (pattern[j] = text[i + j] )
        .....
```

$$f(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1$$

$$= \sum_{i=0}^{n-m} m-1-0+1$$

$$= \sum_{i=0}^{n-m} m$$

$$= m\sum_{i=0}^{n-m} 1$$

$$= m(n-m-0+1) = mn - m^2 + m$$

Note: mn is higher when compared to m^2 because normally n is greater than m

So, time complexity for pattern matching is given by **f(n) ∈ O(mn)**

3.6 Exhaustive search

Now, let us see "What is exhaustive search? What are the problems that can be solved using this technique?"

Definition: The brute-force approach used to solve combinatorial problems is called exhaustive search. The solution to the combinatorial problems consumes too much time. This searching technique generates all the possible solutions by satisfying the constraints given in the problem. Finally the desired solution which maximizes the profit is selected. The goal of the exhaustive search method is to search all possible solutions and obtain an optimal solution. Some of the problems that involve exhaustive search are:

- Traveling Salesman problem
- Knapsack problem
- Assignment problem

3.6.1 Traveling Salesman Problem (TSP)

Now, let us see "What is traveling salesman problem?"

Definition: Given *n* cities, a salesperson starts at a specified city (often called source), visit all *n*-1 cities only once and return to the city from where he has started. The

objective of this problem is to find a route through the cities that minimizes the cost (there by maximizing the profit).

Now, let us see "How to model traveling salesman problem?" This problem can be modeled by a directed weighted graph as shown below:

- The vertices of the graphs represent the various cities
- The weights associated with edges represent the distances between two cities or the cost involved while traveling from one city to other city.
- The cost involved from city *i* to city *j* is represented using a 2-dimensional array and C[i, j] gives the cost from city *i* to city *j*.

Now, the question is "How the traveling salesman problem can be solved?"This can be achieved using brute force approach as shown below:

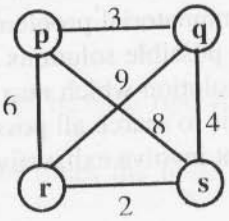
- Get all the routes from one city to another city by taking various permutations
- Compute the route length for each permutation and select the shortest among them.

But, in this technique, the execution time increases very rapidly as the size of the TSP problem increases. For example,

- If the *n*umber of cities =16, the number of possible routes are 1,307,674,368,000
- If number of cities =20, the number of possible routes are 12,164,510,040,883,200.

Note: The easiest way to find an exact solution to the traveling salesman problem is to compute the cost of all tours and determine the tour with the minimum cost. The above method is called *exhaustive sequential search algorithm* and uses *brute force approach.*

Example 1: Solve the following TSP which is represented as a graph:



where the cities are represented as *p, q, r* and *s* and the distance between various cities are represented as numbers in each of the edge

Solution: The graph represented using *cost adjacency matrix* is shown below:

	p	q	r	s
p	0	3	6	8
q	3	0	9	4
r	6	9	0	2
s	8	4	2	0

If we assume that the salesperson starts from city p, the various routes using which he can visit each and every city exactly once and returns back to the start city p along with the cost incurred for each route is shown below:

$$p \xrightarrow{3} q \xrightarrow{9} r \xrightarrow{2} s \xrightarrow{8} p \text{ (Cost = 22)}$$

$$p \xrightarrow{3} q \xrightarrow{4} s \xrightarrow{2} r \xrightarrow{6} p \text{ (Cost = 15)}$$

$$p \xrightarrow{6} r \xrightarrow{9} q \xrightarrow{4} s \xrightarrow{8} p \text{ (Cost = 27)}$$

$$p \xrightarrow{6} r \xrightarrow{2} s \xrightarrow{4} q \xrightarrow{3} p \text{ (Cost = 15)}$$

$$p \xrightarrow{8} s \xrightarrow{4} q \xrightarrow{9} r \xrightarrow{6} p \text{ (Cost = 27)}$$

$$p \xrightarrow{8} s \xrightarrow{2} r \xrightarrow{9} q \xrightarrow{3} p \text{ (Cost = 22)}$$

Note that even though we have variety of routes with varying costs, we have to consider the route with minimum costto get the maximum profit. So, the following routes can be selected by the salesperson:

$$p \xrightarrow{3} q \xrightarrow{4} s \xrightarrow{2} r \xrightarrow{6} p \text{ (Cost = 15)}$$

$$p \xrightarrow{6} r \xrightarrow{2} s \xrightarrow{4} q \xrightarrow{3} p \text{ (Cost = 15)}$$

Analysis Let us see the various routes (permutations) that can be used by a traveling salesperson when he visits each and every city exactly once and returns back to the city from where he has started. Let us find out the possible routes taken by the salesperson by assuming he has started from the city a. Let n denote the number of cities to visit.

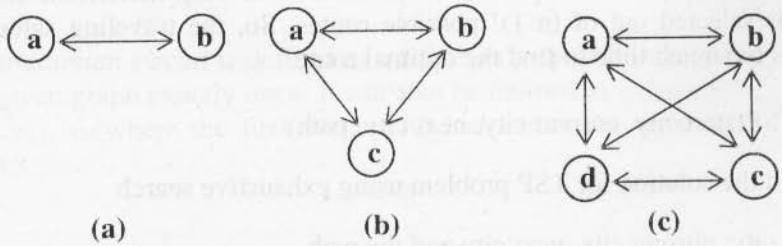


Figure 3.6Cities represented using graphs

When n = 2: It is clear from figure 3.6.a that the various routes are:

$$a \rightarrow b \rightarrow a \text{ i.e, number of routes} = 1 \text{ [i.e., (2-1)!]}$$

When n = 3: It is clear from figure 3.6.b that the various routes are:

$$\left.\begin{array}{l} a \rightarrow b \rightarrow c \rightarrow a \\ a \rightarrow c \rightarrow b \rightarrow a \end{array}\right\} \text{2 routes} = (3\text{-}1)!$$

When n = 4: It is clear from figure 3.6.c that the various routes are:

$$\left.\begin{array}{l} a \rightarrow b \rightarrow c \rightarrow d \rightarrow a \\ a \rightarrow b \rightarrow d \rightarrow c \rightarrow a \\ a \rightarrow c \rightarrow b \rightarrow d \rightarrow a \\ a \rightarrow c \rightarrow d \rightarrow b \rightarrow a \\ a \rightarrow d \rightarrow c \rightarrow b \rightarrow a \\ a \rightarrow d \rightarrow b \rightarrow c \rightarrow a \end{array}\right\} \text{6 routes} = (4\text{-}1)!$$

In general, for n cities number of routes = (n-1)! i.e., f(n) = (n-1)! So, time complexity is given by

$$\mathbf{f(n) \in O(n!)}$$

Note: This method always gives an optimal solution since we have taken all permutations and combinations of routes and selected the route with least cost. It is very straight forward approach and simple technique. But, this method is very inefficient since the optimal solution is selected out of (n-1)! possible routes. So, the traveling salesperson problem consumes too much time to find the optimal route.

ALGORITHM TSP(start city, current city, next city, path)

// **Purpose**: To find the solution for TSP problem using exhaustive search

// **Input**: The start city, current city, next city and the path

// **Output**: The minimum distance covered along with the path

Step 1: [Check for the disconnection between the current city and next city]

```
if there is no connection between current city and next city then
        abandon the path
end if
```

Step 2: [Check whether the traveling sales person has visited all the cities]

```
if current city is the last city in the path then
        Check if the current city has a connection with the start city
        if connection exists then
                return full path along with cost
        else
                abandon the path
        end if
end if
```

Step 3: [Find the next city to be visited]

```
for each next city on this path
        check the path with next city recursively by
                calling TSP(start city, current city, next city, path)
end for
```

Step 4: [Find the solution and terminate]

Compare the results of all cities and get the minimum cost route
Return the optimal path with its cost

Note: The traveling salesperson problem can be stated as the problem of finding the shortest *Hamiltonian circuit* in a given graph. Now, let us see "What is Hamiltonian circuit?"

Definition: Let G = (V, E) is a directed and connected graph where

- V is set of vertices
- E is set of edges.

A *Hamiltonian circuit* is defined as a circuit or a cycle that passes through all the vertices of a given graph exactly once. It can also be defined as a sequence of $n+1$ vertices $v_0, v_1, v_2, \ldots v_{n-1}, v_0$ where the first vertex and last vertex are same and all other vertices are distinct

3.6.2 Knapsack problem

Now, let us see "What is knapsack problem?"

Definition: Given a knapsack (bag or container) of capacity M and n objects of weights $w_1, w_2, w_3, \ldots w_n$ with profits $p_1, p_2, p_3, \ldots p_n$. Let $x_1, x_2, x_3, \ldots x_n$ be the fractions of the objects that are supposed to be added into the knapsack. The main objective is to place the objects into the knapsack so that maximum profit is obtained and the weights of objects chosen should not exceed the capacity of knapsack. This problem can be stated as:

$$\text{Maximize } \sum_{i=1}^{n} P_i X_i$$

$$\text{Subject to the constraint } \sum_{i=1}^{n} W_i X_i \le M$$

Example 1: Solve the following knapsack problem using brute force method given

M = 40, n = 3

$(w_1, w_2, w_3) = \{20, 25, 10\}$ represents weight of 3 objects

$(p_1, p_2, p_3) = \{30, 40, 35\}$ represents profits of 3 objects

Note: Assume that entire object has to be selected

Solution: The various feasible solutions for this problem are shown below:

Objects selected	Total weight of objects selected	Feasible or not	Profit earned
{}	0	Feasible	0
{1}	20	Feasible	30
{2}	25	Feasible	40
{3}	10	Feasible	35
{1, 2}	20 + 25 = 45	Not feasible	
{1,3}	20 + 10 = 30	Feasible	65
{2, 3}	25 + 10 = 35	Feasible	75
{1,2,3}	20 + 25 + 10 = 55	Not feasible	

The combination of objects {1, 2} and {1, 2, 3} are not feasible because, the total weight of the objects selected will exceed the capacity of the knapsack 40 and rest are all feasible solutions. A *feasible solution* is the one that satisfies the given constraint. Out of the feasible solutions, we have to select the subset which leads to maximum profit which in this case is 75 by selecting the objects {2, 3}.

Analysis The total number of subsets obtained for 3 elements is 8. In general, given n objects, the total number of subsets obtained will be 2^n. Even, in the best case and worst case, the total number of subsets generated will be 2^n. So, the time complexity of knapsack is

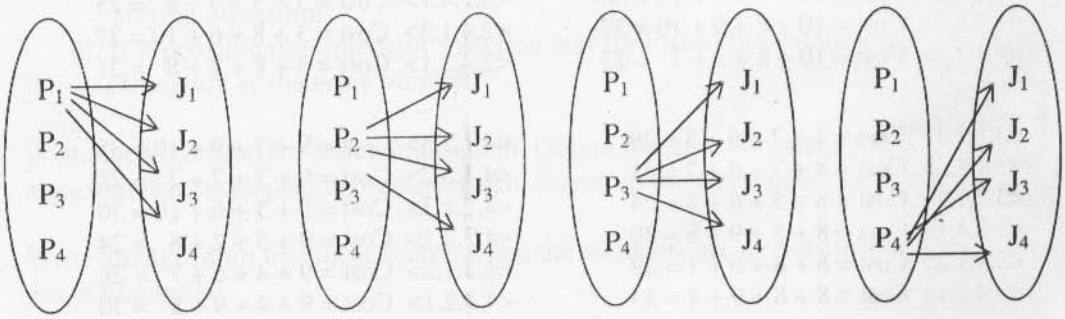
$$f(n) \in \theta(2^n)$$

Note: The exhaustive search for TSP problem and knapsack problems are extremely inefficient on every input.

3.6.3 Assignment problem

The assignment problem which finds many applications in industry and in the field of computer science is discussed in this section. The *assignment problem* can be solved by *exhaustive search* technique. Now, let us see "What is an assignment problem?"

Definition: Given n jobs $<j_1, j_2, j_3, j_4 \ldots\ldots j_n>$ and n persons $<p_1, p_2, p_3, p_4 \ldots\ldots p_n>$, it is required to assign all n jobs to all n persons with the constraint that one job has to be assigned to one person and the cost involved in completing all the jobs should be minimum. The various combinations in which 4 jobs are assigned to 4 people are shown below:



The solution to this problem can be expressed as a n-tuple using a single dimensional array as shown below:

$$<j_1, j_2, j_3, j_4, \ldots j_n>$$

where

- The index represent the person
- The value stored in the index represent the job number

For example, <2, 3, 4, 1> is one possible solution and gives the following information:

- 1st value is 2: For 1st person, job 2 is assigned
- 2nd value is 3: For 2nd person, job 3 is assigned

- 3rd value is 4: For 3rd person, job 4 is assigned
- 4th value is 1: For 4th person, job 1 is assigned

Example 1: Find the optimal solution for the given assignment problem which is represented as a matrix as shown below:

	J1	J2	J3	J4
P1	10	3	8	9
P2	7	5	4	8
P3	6	9	2	9
P4	8	7	10	5

Solution: The cost involved for i^{th} person to complete j^{th} job is represented using *cost adjacency matrix*. The exhaustive approach to the assignment problem requires all permutations of integers 1,2,3,.....n for n jobs and n people. For each permutation, obtain the total cost. The various instances of the problem's *solutions* are shown below:

<1,2,3,4> Cost = 10 + 5 + 2 + 5 = 22
<1,2,4,3> Cost = 10 + 5 + 9 + 10 = 34
<1,3,2,4> Cost = 10 + 4 + 9 + 5 = 28
<1,3,4,2> Cost = 10 + 4 + 9 + 7 = 30
<1,4,2,3> Cost = 10 + 8 + 9 + 10 = 37
<1,4,3,2> Cost = 10 + 8 + 2 + 7 = 27

<2,1,3,4> Cost = 3 + 7 + 2 + 5 = 17
<2,1,4,3> Cost = 3 + 7 + 9 + 10 = 29
<2,3,1,4> Cost = 3 + 5 + 6 + 5 = 19
<2,3,4,1> Cost = 3 + 5 + 9 + 8 = 25
<2,4,1,3> Cost = 3 + 8 + 6 + 10 = 27
<2,4,3,1> Cost = 3 + 8 + 2 + 8 = 21

<3,1,2,4> Cost = 8 + 7 + 9 + 5 = 29
<3,1,4,2> Cost = 8 + 7 + 9 + 7 = 31
<3,2,1,4> Cost = 8 + 5 + 6 + 5 = 24
<3,2,4,1> Cost = 8 + 5 + 9 + 8 = 30
<3,4,1,2> Cost = 8 + 8 + 6 + 7 = 29
<3,4,2,1> Cost = 8 + 8 + 9 + 8 = 33

<4,1,2,3> Cost = 9 + 7 + 9 + 10 = 35
<4,1,3,2> Cost = 9 + 7 + 2 + 7 = 25
<4,2,1,3> Cost = 9 + 5 + 6 + 10 = 30
<4,2,3,1> Cost = 9 + 5 + 2 + 8 = 24
<4,3,1,2> Cost = 9 + 4 + 6 + 7 = 26
<4,3,2,1> Cost = 9 + 4 + 9 + 8 = 30

Out of these solutions, the solution that takes least cost is

<2,1,3,4> Cost = 3 + 7 + 2 + 5 = 17

and this solution is the optimal solution.

Analysis Note that total number of feasible solutions for $n = 4$ will be 24 which is nothing but 4!. In general, if there are n jobs and n people, then the total number of feasible solutions will be n!. So, time complexity of assignment problem in the worst case is given by

$$f(n) \in O(n!)$$

Exercises:

1. What is a brute-force method? Under what conditions does the method become desirable? **(Jan/Feb 2005)**

2. Write the bubble sort algorithm and show that the worst case efficiency is quadratic **(July/Aug 2004)**

3. What changes to be made to bubble sort so as to avoid the time complexity $O(n^2)$ when the elements are arranged in ascending order. Give the algorithm for the same

4. Write the algorithm for selection sort and show that the time complexity of this algorithm is quadratic.

5. Suggest an algorithm for selection sort and explain its operation and analyze the algorithm mathematically

6. Write an algorithm for selection sort
   i. Is selection sort stable?
   ii. Write the property, which distinguishes selection sort positively from other sorting algorithms
   iii. Is it possible to implement selection sort for linked list with the same efficiency as the array version?

7. Write the algorithm for sequential search. Obtain the time complexity of this algorithm for successful and unsuccessful search in the worst case and best case

8. Write an algorithm for linear search when the elements are already arranged in ascending order

9. Explain pattern-matching using brute-force technique.

10. Give the algorithm for brute-force pattern matching and obtain the time complexity in the best case and worst case

11. Suggest an algorithm to find the mode in a set of elements. Analyze its efficiency using the method suggested by you

12. Discuss whether the traveling sales person problem can be solved by exhaustive search methods. **(Jan/Feb 2005)**

3.28 Brute Force

13. Outline an exhaustive search algorithm for traveling salesman problem. What is the efficacy class of this algorithm? Illustrate with an example.
    a. Assuming that each tour can be generated in constant time, what will be the efficiency class of exhaustive search algorithm for TSP?
    b. If this algorithm is programmed on a computer that makes 1 billion additions per second, estimate the maximum number of cities for which the problem can be solved in one hour, 24 hours, one year, and one century.

14. Explain knapsack problem with respect to brute-force technique

15. Explain assignment problem with respect to brute-force technique

Chapter 4: Divide and Conquer

What are we studying in this chapter?

- Concept of Divide and Conquer Technique
- Merge sort
  - Design
  - Analysis
- Quick sort
  - Design
  - Analysis of best case
  - Analysis of worst case
  - Analysis of average case
- Binary search
  - Design
  - Analysis of best case
  - Analysis of worst case
  - Analysis of average case
- Binary tree Traversals and Related Properties
- Multiplication of Large integers
- Strassen's Matrix multiplication along with analysis **- 6 hours**

4.1 Introduction

Divide and conquer method of designing the algorithm is the best known method of solving a problem. Now, let us see "What is divide and conquer technique? What is the general plan using which these algorithms work?"

Definition: Divide and conquer is a top-down technique for designing algorithms that consist of dividing the problem into smaller sub problems hoping that the solutions of the sub problems are easier to find. The solutions of all smaller problems are then combined to get a solution for the original problem.

The divide and conquer technique of solving a problem involves three steps at each level of the recursion:

- **Divide:** The problem is divided into a number of sub problems
- **Conquer:** The sub problems are conquered by solving them recursively. If the sub problems are smaller in size, the problem can be solved using straightforward method
- **Combine:** The solutions of sub problems are combined to get the solution for the larger problem.

Note: The divide and conquer technique is represented as shown in figure 4.1.

A problem is divided into smaller instances of the same type as the original problem. All these smaller instances are of the same size. The various problems that can be solved using this approach are shown below:

- Sum of elements in a given array
- Merge sort
- Quick sort
- Tree traversals
- Binary search
- Matrix multiplication using Strassesn's algorithm

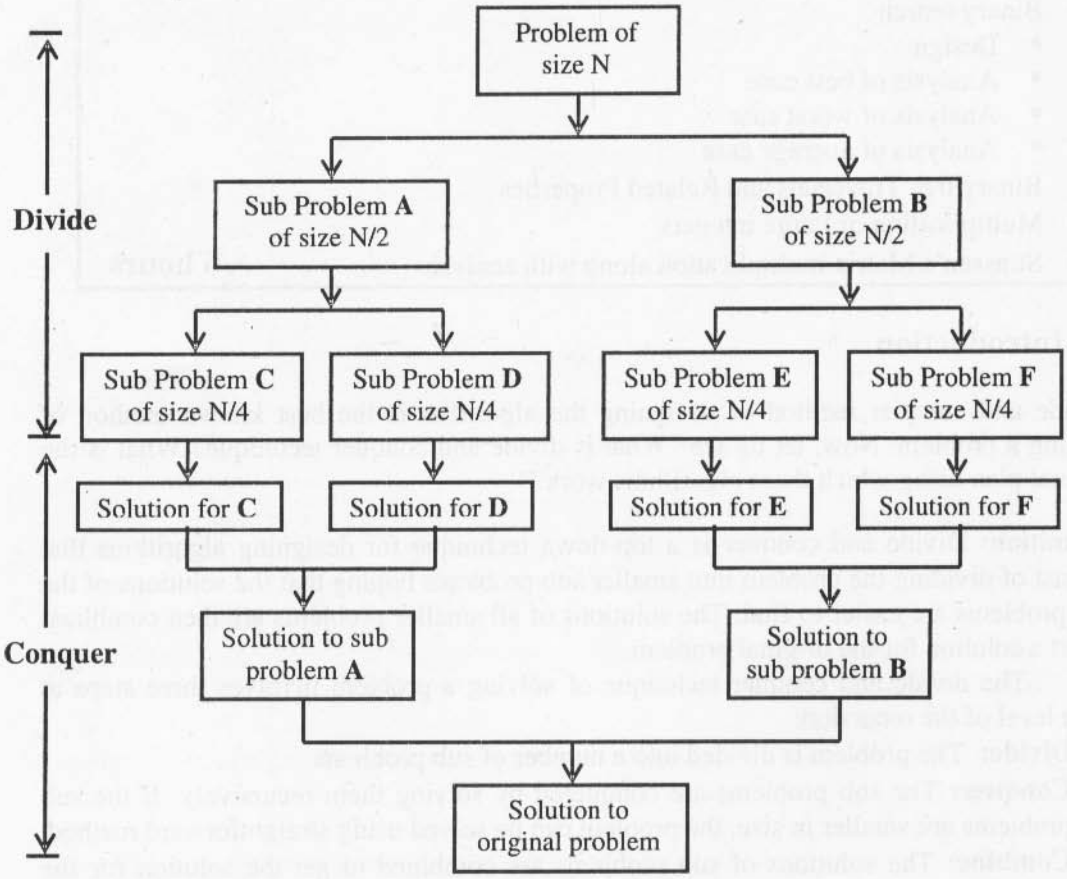


Figure 4.1 Divide and Conquer technique

4.2 Master theorem

The *time complexity* of *divide and conquer* technique can be easily computed using Master theorem. An instance of size *n* can be divided into several instances say *a* of size n/b and the *time complexity* can be obtained using the *general divide and conquer recurrence* relation:

$$T(n) = aT(n/b) + f(n)$$

where

- *a* and *b* are positive constants such that b > 1
- f(n) is a function which is the time spent on dividing the problem into smaller instances and combining them to get a single solution.

For the above equation, the *time efficiency* (time complexity) can be easily calculated using the following relation (which is obtained using MASTER THEORM):

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

Note: Here, *d* is the power of n in f(n).

4.3 Sum of n elements of array a

In this section, let us "Write an algorithm to find the sum of *n* numbers stored in an array using divide and conquer method and obtain the time complexity".

Given the following:

- *a* – list of elements to be added
- *low* – the position of the first element
- *high* – the position of the last element

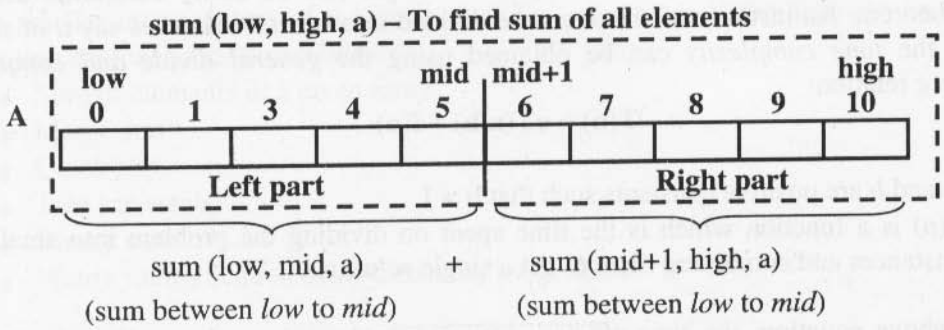
the sum of all the elements stored in array *a* between *low* and *high* can be calculated using the following procedure:

Step 1: If *low* is same as *high*, it indicates that there is only one element in the array and return a[low] as the solution.

Step 2: If *low* is less than *high*, we divide the problem into two parts using the relation:

mid ← (low + high) / 2

Step 3: Recursively obtain the sum of left part ranging from *low* to *mid* and recursively obtain the sum of right part ranging from *mid+1* to *high* and find the sum using the sum. This can be pictorially represented as shown below:



The recursive relation to find sum of all elements can be written as:

$$F(low, high, a) = \begin{cases} a[low] & \text{if } low = high \\ mid \leftarrow (low + high)/2 & \\ F(low, mid, a) + F(mid+1, high, a) & \text{otherwise} \end{cases}$$

By looking at the above recursive relation (by replacing the identifier F by SUM), we can write the algorithm as shown below:

```
Algorithm SUM(low, high, a)
// Purpose: Finds the sum of first n numbers stored in an array using divide and
            conquer technique recursively

// Input: The array a where N elements are stored
//        low and high represent the low index and high index in the array

// Output:  returns sum of all the numbers stored in the array A

    if ( low = high) return a[low]

    mid ← (low + high) / 2

    return SUM(low, mid, a) + SUM(mid+1, high, a)
```

Analysis (using substitution method) The time efficiency to find the sum of n elements using divide and conquer method can be obtained using the following relation:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/2) + T(n/2) + 1 & \text{Otherwise} \end{cases}$$

Time required to add the items in the left part of the array

Time required to add the items in the right part of the array

Time unit for adding left part and right part

where

- T(n/2) is the time required to add the items to the left part of the array
- T(n/2) is the time required to add the items to the right part of the array
- The constant 1 represent the time required to add the result of left part and right part. The relation can be solved using substitution method as shown below:

Consider the relation

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

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

$$= 2\left[2T\left(\frac{n}{4}\right) + 1\right] + 1$$

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

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

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

.....

.....

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

To get the initial condition T(1) = 0, let $2^i = n$

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

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 1 \text{ by replacing n by n/2}$$

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

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

$$= 2^i T(1) + 2^{i-1} + 2^{i-2} + ...2^3 + 2^2 + 2 + 1$$

Substituting $2^i = n$, we have

$$= n*0 + \frac{a(r^n - 1)}{r-1} = \frac{1(2^i - 1)}{2-1} = 2^i - 1 = n - 1$$

i.e., T(n) = n – 1. By expressing T(n) in terms of θ-notation we have, T(n) ∈ θ(n).
∴, Time complexity to find sum of n numbers is given by

$$\boxed{\mathbf{T(n) \in \theta(n)}}$$

Analysis (using Master theorem) Using Master theorem, the time complexity can be obtained very easily. The generalized recurrence relation for divide and conquer method is of the form:

$$T(n) = a\,T(n/b) + f(n) \text{ ----------(1)}$$

and the time complexities can be obtained using following relations:

$$T(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log_b n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases} \text{ ----------(2)}$$

We know that the recurrence relation to find the sum of *n* numbers is given by

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/2) + T(n/2) + 1 & \text{Otherwise} \end{cases}$$

which can also be written as

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + 1 & \text{Otherwise} \end{cases}$$

From the above relation, we have T(n) = 2T(n/2) + 1 ----------(3)

Comparing equations (1) and (3) we have a = 2, b = 2 and $f(n) = 1 = n^0 = n^d$ and so the value of *d* is zero. Here, a = 2 and $b^d = 2^0 = 1$. It is evident that the third relation $a > b^d$ of equation (2) holds good and time complexity is given by

$$T(n) = \theta(n^{\log_b a}) = \theta(n^{\log_2 2}) = \theta(n) \qquad \boxed{\therefore, \text{ Time complexity is given by } T(n) = \theta(n)}$$

4.5 Sorting using divide and conquer

The two sorting techniques that closely follows the divide and conquer method are:

- Merge sort
- Quick sort (Partition exchange sort)

4.5.1 Merge sort

Now, let us see "What are the steps involved in merge sort?" The various steps that are involved while sorting using merge sort are shown below:

- **Divide:** Divide the given array consisting of *n* elements into two parts of n/2 elements each
- **Conquer:** Sort the left part of the array and right part of the array recursively using merge sort
- **Combine:** Merge the sorted left part and sorted right part to get a single sorted array

The key operation in merge sort is combining the sorted left part and sorted right part into a single sorted array. This process of merging of two sorted vectors into a single sorted vector is called *simple merge*. The only necessary condition for this problem is that both arrays should be sorted. Now, let us see "How to design simple merge algorithm?"

Design: Suppose we have two sorted vectors A and B with *m* and *n* elements respectively. The following procedure is used to merge these two sorted vectors:

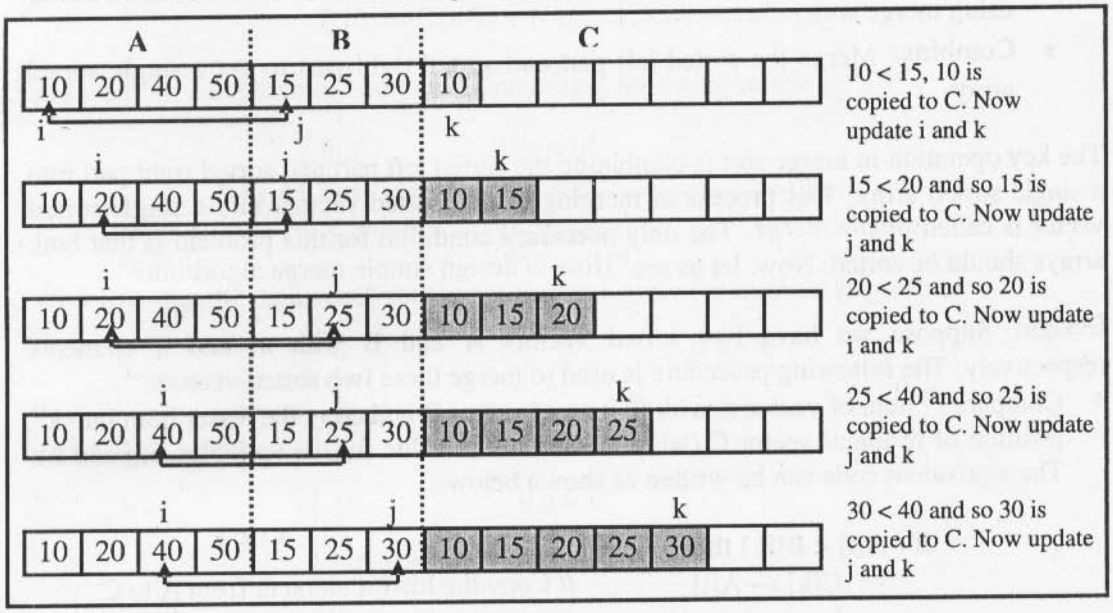
- Compare i^{th} item of vector A with j^{th} item of vector B and copy the lesser item into k^{th} position of resultant vector C (with 0 as the initial value for the variables i, j and k). The equivalent code can be written as shown below:

```
if ( A[i] < B[j] ) then
        C[k] ← A[i]              // Copy the lowest element from A to C
        k← k+1, i ← i + 1        // Point to next item in C and A
else
        C[k] ← B[j]              // Copy the lowest element from B to C
        k←k + 1, j←j + 1         // Point to next item in C and B
end if
```

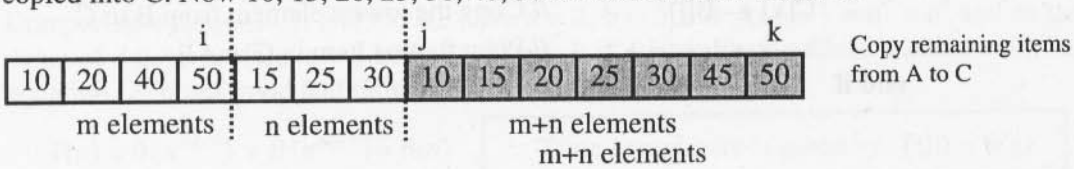
- Note that while updating *i*, the index *i* should not exceed *m* and while updating *j*, the index *j* should not exceed *n*. So, the above statement has to be executed if and only if $i < m$ and $j < n$ and can be modified as shown below:

```
while ( i < m and j < n )
        if ( A[i] < B[j] ) then
                C[k] ← A[i]              // Copy the lowest element from A to C
                k← k+1, i ← i + 1        // Point to next item in C and A
        else
                C[k] ← B[j]              // Copy the lowest element from B to C
                k←k + 1, j←j + 1         // Point to next item in C and B
        end if
end while
```

For example, consider the two sorted vectors A containing 4 elements and B containing 3 elements. The trace of the above algorithm is shown below.



Since, there are no more elements in B to compare, the remaining elements of A can be copied into C. Now 10, 15, 20, 25, 30, 40, 50 are stored in the vector C.



In general, when either *i* or *j* exceeds its limits, the elements remaining in other vector, can be copied into the vector C and the corresponding pseudocode is shown below:

```
while ( i < m )                    // Copy the remaining items from A to C
        C[k] ← A[i]
        k ← k + 1, i ← i + 1
end while
while ( j < n )                    // Copy the remaining items from B to C
        C[k] ← B[j]
        k ← k + 1, j ← j + 1
end while
```

The complete algorithm for simple merge is shown below:

```
Algorithm SimpleMerge(A, B, C, m, n)

//Purpose : Merge two sorted arrays into a single sorted array

//Input:
//      A is a sorted vector with m elements
//      B is a sorted vector with n elements

//Output:
//      C is a sorted vector obtained after merging the elements of A and B.

i ← j ← k ← 0
while ( i < m and j < n )
        if ( A[i] < B[j] ) then
                C[k] ← A[i]             // Copy the lowest element from A to C
                i ← i + 1               // Point to next item in A
                k ← k + 1               // Point to next item in C
        else
                C[k] ← B[j]             // Copy the lowest element from B to C
                j ← j + 1               // Point to next item in B
                k ← k + 1;              // Point to next item in C
        end if
end while
while ( i < m )                         // Copy the remaining items from A to C
        C[k] ← A[i]
        i ← i + 1
        k ← k + 1
end while
```

```
while ( j < n )                    // Copy the remaining items from B to C
    C[k] ← B[j]
    j ← j + 1
    k ← k + 1
end while
```

Modified simple merge: In the above algorithm we have used two sorted arrays A and B with index values of array A and array B starting from 0. Now, consider a situation of a single array which is divided into two sorted parts as shown below:

A	10	20	30	40	50	15	25	35	45
	0	1	2	3	4	5	6	7	8
	↑ low				↑ mid	↑ mid + 1			↑ high

Observe the following facts in the above array:

- The elements from *low* to *mid* are sorted
- The elements from *mid+1* to *high* are also sorted.
- But, if we take the elements from *low* to *high* the elements are not sorted.

Here, instead of considering the array A as single array, we can assume that there are two arrays with single name A:

- The first array starts from *low* to *mid*
- The second array starts from *mid+1* to *high.*

Now, if we want to merge the two sorted vectors into a single sorted vector, we can modify the earlier algorithm by replacing the array B by A and the limits of the first part of the array being *low* and *mid* whereas the limits of second part of the array being *mid+1* and *high.* The modified algorithm to merge two sorted vectors is shown below:

Algorithm SimpleMerge(A, low, mid, high)

//**Purpose :** Merge two sorted arrays where the first array starts from *low* to *mid* and the
// second starts from *mid+1* to *high*

//**Input:** A is a sorted from the index position *low* to *mid*
// A is a sorted from the index position *mid+1* to high

//**Output**: A is a sorted from index *low* to *high.*

```
i ← low, j ← mid+1, k ← low

while ( i <= mid and j <=  high )
    if ( A[i] < A[j] ) then
            C[k] ← A[i]    // Copy the lowest element from first part of A to C
            i ← i + 1      // Point to next item in the left part of A
            k ← k + 1      // Point to next item in C
    else
            C[k] ← A[j]    // Copy the lowest element from second part of A to C
            j ← j + 1      // Point to next item in the second part of A
            k ← k + 1;     // Point to next item in C
    end if
end while

while ( i <= mid )         // Copy the remaining items from left part of A to C
    C[k] ← A[i]
    k ← k + 1, i ← i + 1
end while

while ( j <= high )        // Copy the remaining items from right part of A to C
    C[k] ← A[j]
    k ← k + 1, j ← j + 1
end while

for i = low to high        // Copy the elements from array C to array A
    a[i] ← c[i]
end for

// End of the algorithm
```

Once the merging is over, we can easily arrange the numbers in ascending order using merge sort. If *low* and *high* are lower limit and upper limits in an array, the general procedure to sort the items using merge sort is shown below:

if (low < high)

- Divide the array into equal parts
- Sort the left part of the array recursively
- Sort the right part of the array recursively
- Merge the left part and right part

end if

The complete algorithm to sort the numbers using merge sort is shown below:

```
Algorithm MergeSort(A, low, high)

//Purpose : Sort the elements of the array between the lower bound and upper bound

//Input: A is an unsorted vector with low and high as lower bound and upper bound

//Output: A is a sorted vector

if ( low < high)
        mid ← (low + high)/2                    // Divide the array into equal parts
        MergeSort (a, low, mid)                 // Sort the left part of array
        MergeSort (a, mid + 1, high)            // Sort the right part of the array
        SimpleMerge(a, low, mid, high)          // Merge the left part and right part
end if
```

Analysis It is clear from the algorithm that the problem instance is divided into two equal parts. Let us assume that the number of items in the array *a* are integral multiples of 2 (i.e., $n = 2^k$ for $k \geq 0$). The recurrence relation for this algorithm can be written as:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \quad \text{(sorting not required)} \\ \underbrace{T(n/2)}_{\text{Time required to sort the left part of the array}} + \underbrace{T(n/2)}_{\text{Time required to sort the right part of the array}} + \underbrace{n}_{\text{Time required to divide \& merge one instance of a solution with n elements}} & \text{Otherwise} \end{cases}$$

Note: If n=1, there is only one item and hence can be considered as sorted and no time is required for this. So, T(1) = 0 if n = 1. The above relation can be written as:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Time complexity using MASTER THEOREM

The general recurrence relation for divide and conquer method is

$$T(n) = a\,T(n/b) + f(n) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

whereas the recurrence relation for merge sort is given by

$$T(n) = 2T(n/2) + cn \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

By comparing the relations (1) and (2) we have a = 2, b = 2 and $f(n) = cn = cn^1 = cn^d$ and so the value of *d* is one. Substituting these values in the following recurrence relation

$$T(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log_b n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

we get a = 2 and $b^d = 2^1 = 2$. It is evident that the relation $a = b^d$ holds good and time complexity is given by

$$T(n) = \theta(n^d \log_b n) = \theta(n^1 \log_2 n) = \theta(n \log_2 n)$$

∴, Time complexity of merge sort is given by:

$$\boxed{T(n) = \theta(n \log_2 n)}$$

Time complexity using substitution method

Instead of using MASTER THEOREM, we can also find the time complexity using ordinary substitution method as shown below:

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

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

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

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

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

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

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

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

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

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

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} \quad \text{by replacing n by n/2}$$

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

$$\cdots\cdots\cdots\cdots\cdots\cdots$$

$$\cdots\cdots\cdots\cdots\cdots\cdots$$

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

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

→ To get the initial condition T(1) = 0, **let $2^i = n$** ----------(1)

$$= n.T\left(\frac{n}{n}\right) + n.i$$

$$= n.T(1) + n.i$$

$$= n.0 + n.i = n.i \text{ ----------(2)}$$

From equation (1) we have $2^i = n$

Taking log on both sides we have,

$$i.\log_2^2 = \log_2^n$$

$$i = \log_2^n$$

Substituting this in equation (2) we have,

$$t(n) = n\log_2 n$$

∴, Time complexity of merge sort is given by

$$\boxed{T(n) = \theta\,(n\log_2 n)}$$

Advantages

- It can be used for sorting array elements in main memory as well as to sort elements in external file
- It can be applied to files of any size.
- The time complexity is ***n log n*** which is very efficient when compared to time complexities of other algorithms

Disadvantages

- The algorithm uses extra space proportional to N. So, algorithm is not in-place.
- It uses more memory on the stack because of recursion

Example 1: Sort the following elements using merge sort

60, 50, 25, 10, 35, 25, 75, 30

The above elements can be sorted using merge sort and the trace for the same is given below:

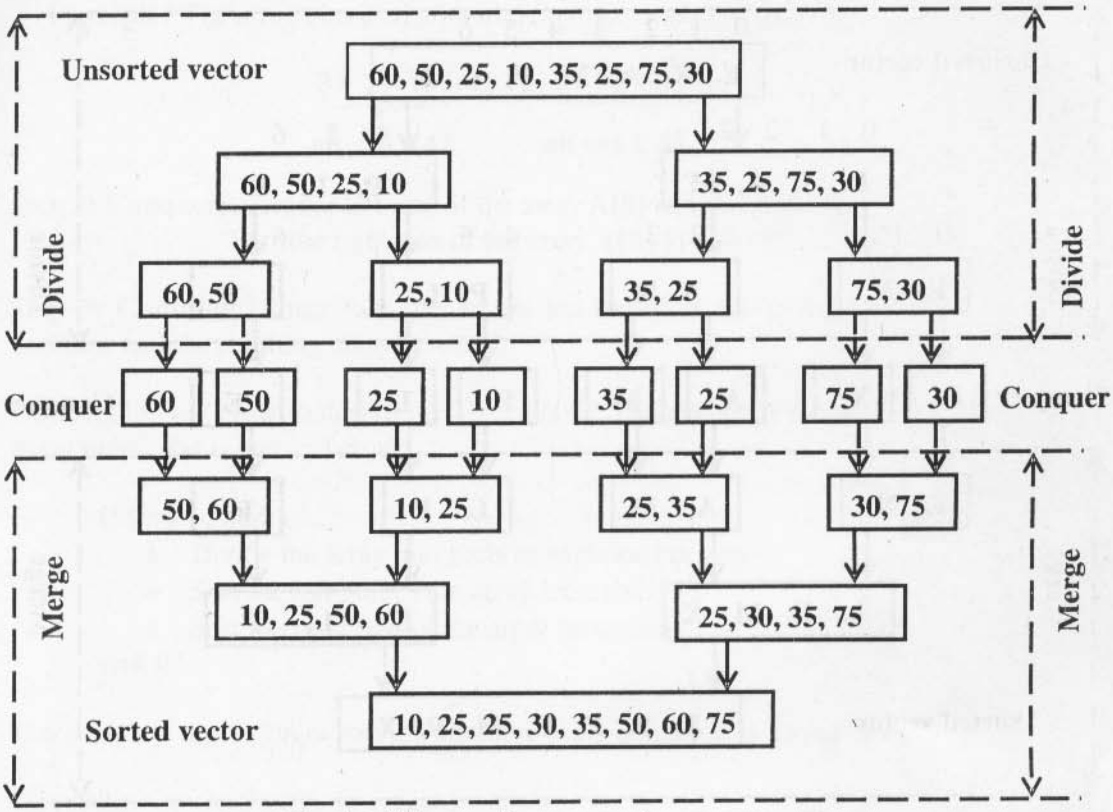


Fig 4.5.1 Trace of merge sort

Now, let us see, "Whether merge sort algorithm is stable? Is it in place?"

STABLE: Since, the relative positions of duplicated elements remain same before sorting and after sorting, merge sort algorithm is stable.

IN PLACE: Since merge sort algorithm uses extra space to store the array elements for sorting which is proportional to N, it consumes more space. So, merge sort algorithm is in place.

Example 2: Sort the following elements using merge sort

E, X, A, M, P, L, E

The above elements can be sorted using merge sort and the trace for the same is given below:

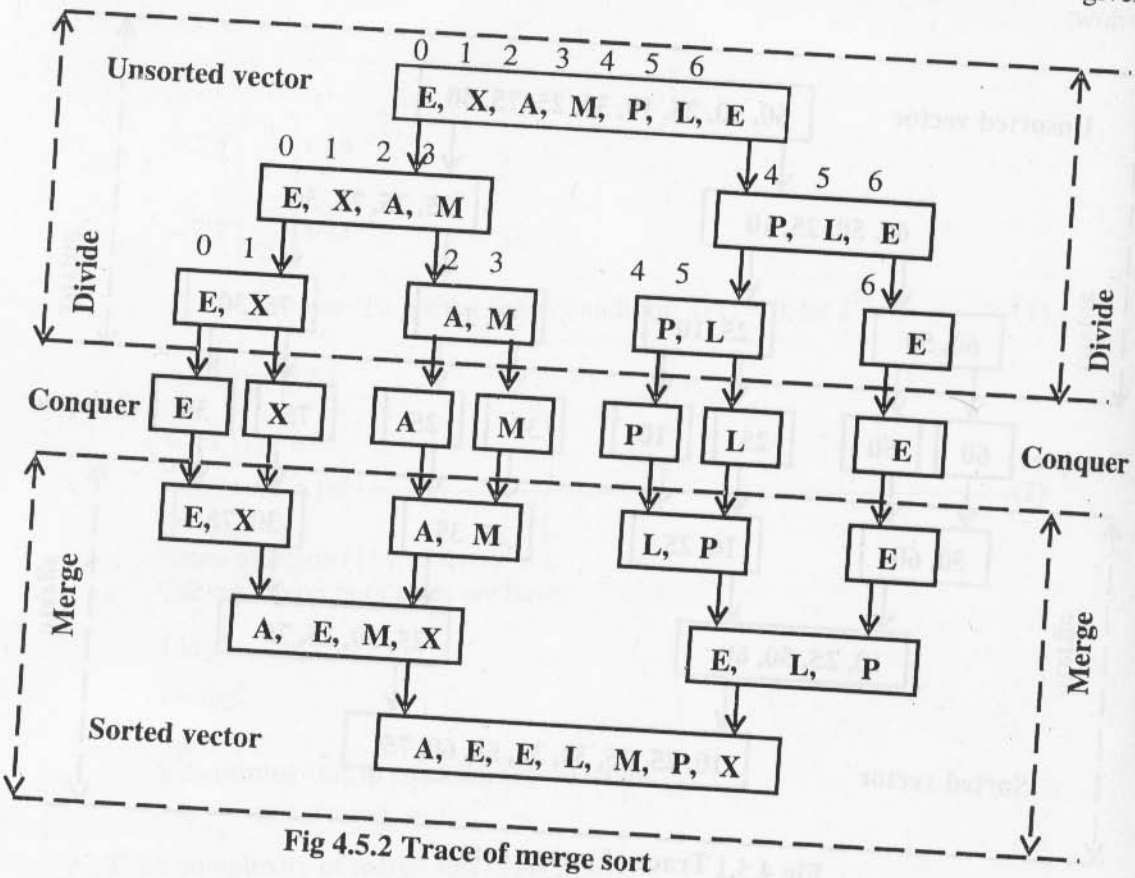
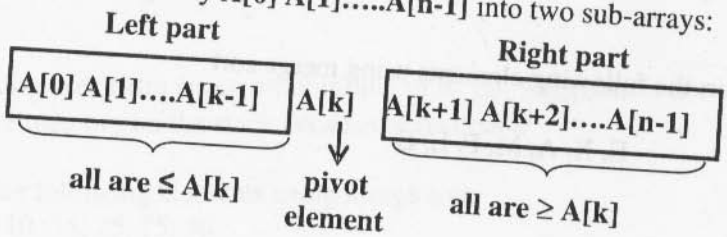


Fig 4.5.2 Trace of merge sort

4.5.2 Quick sort (Partition exchange sort)

The quick sort algorithm is also based on the principle of "Divide and conquer". It works very well on large set of data. Now, let us see "What are the steps involved in quick sort?" The various steps that are involved while sorting using quick sort are shown below:

Step 1: Divide: Divide the array **A[0] A[1].....A[n-1]** into two sub-arrays:



Note: With respect to **A[k]** the elements are divided into two parts. So, A[k] is called *pivot* element.

Example: The array obtained after dividing may look like the way it is shown below:

$$\underbrace{37\ 11\ 10}_{\text{all are} \le 42}\quad 42\ 72\ \underbrace{65\ 98\ 88\ 78}_{\text{all are} \ge 42}$$

Step 2: Conquer: Sort the left part of the array **A[0] A[1].....A[k-1]** recursively
Sort the right part of the array **A[k+1] A[k+2].....A[kn-1]** recursively

Step 3: Combine: Since two sub arrays are sorted in the previous step, no need to combine and the resulting array is sorted.

Now, let us see "How to design the algorithm?" The general procedure to sort the items using quick sort is shown below:

if (low < high)
- Divide the array into parts as explained in step 1.
- Sort the left part of the array recursively
- Sort the right part of the array recursively

end if

The complete algorithm to sort the numbers using quick sort is shown below:

Algorithm Quick_Sort(A, low, high)

//**Purpose** : Sort the elements of the array using quick sort
//**Inputs**: **low** : The position of the first element of array *a*
high : The position of the last element of array *a*
a : It is an array consisting of unsorted elements

//**Output**: ***a***: It is an array consisting of sorted elements

```
if ( low < high)
        k ← partition(a, low, high)        // Divide the array into equal parts
        Quick_Sort (a, low, k-1)           // Sort the left part of array
        Quick_Sort (a, k + 1, high)        // Sort the right part of the array
end if
```

Now, let us see "How to write the algorithm for partitioning the array?"

4.18 Divide and Conquer

Design: To simplify the design, assume that a very large value is stored at the end of the array. This is achieved by storing ∞ in a[n]. Apart from the variables *a, low* and *high* other variables that are used are:

- ***i*** **:** The initial value of index *i* is low i.e., i ← low
- ***j*** **:** The initial value of index *j* is one more than *high* i.e., j ← high+1
- ***pivot*** **:** a[low] is treated as the pivot element.

The general procedure to partition the array is shown below:

- Keep incrementing index *i* as long as *pivot* ≥ a[i]. This is achieved using the statement:

  do i ← i + 1 **while** (pivot >= a[i]);

- Once the above condition fails, keep decrementing index *j* as long as *pivot* < a[j]. This is achieved using the statement:

  do j ← j – 1 **while** (pivot < a[j]);

- Once the above condition fails, if *i* is less than *j* exchange a[i] with a[j] and repeat all the above process as long as i <= j.
- Finally, exchange a[low] with a[j] and final value of j gives the position of the pivot element.

The complete algorithm expressed using partial pseudo code to achieve the above task is shown below:

Algorithm partition (A, low, high)

//**Purpose** : Divide the array into parts such that elements towards left of *pivot* element are ≤ pivot element and elements towards right of key are ≥ pivot element

//**Inputs**: **low :** The position of the first element of array *a*
high: The position of the last element of array *a*
a **:** It is an array consisting of unsorted elements

//**Output**: ***a*****:** Partitioned array such that elements towards left of pivot element are ≤ pivot element and elements towards right of key are ≥ pivot element

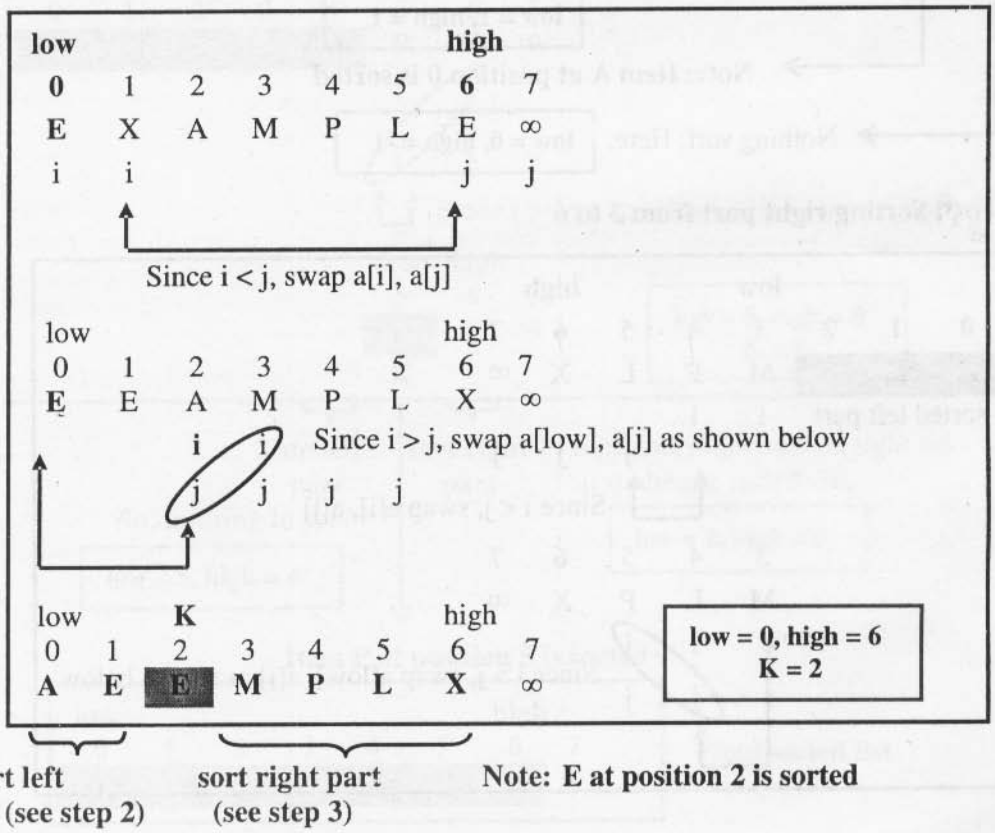
pivot ← a[low]
i ← low
j ← high + 1

```
while (i <= j)
        do i ← i + 1 while (pivot >= a[i])
        do j ← j -1 while ( pivot < a[j] )
        if (i < j) exchange(a[i], a[j])
end while

exchange (a[j], a[low])

return j
```

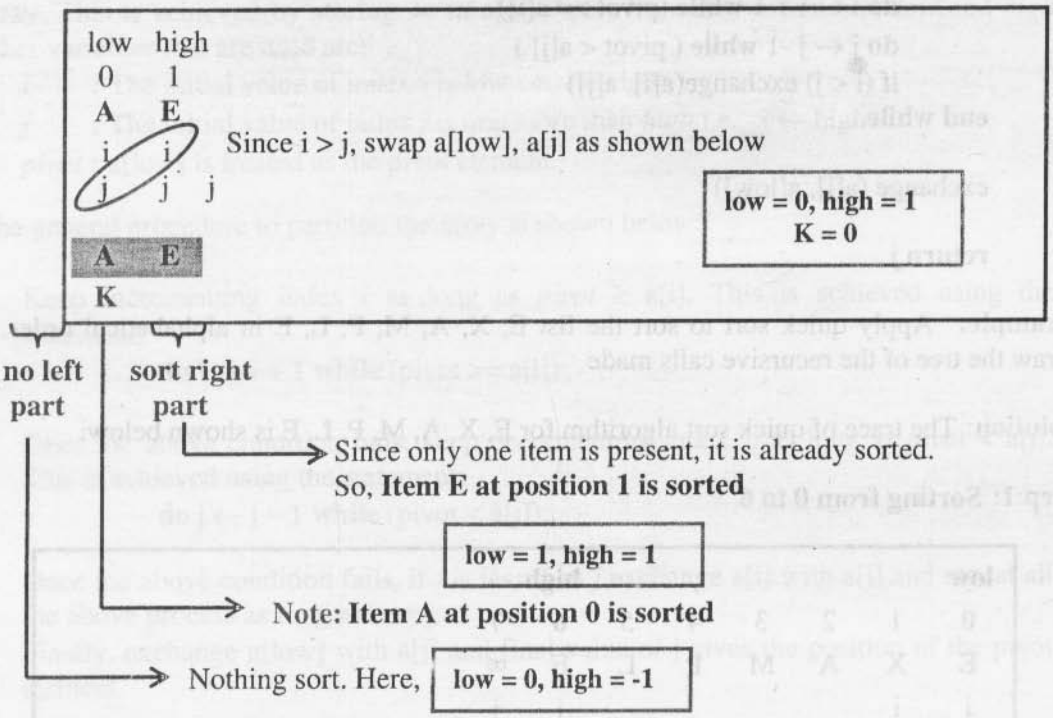
Example: Apply quick sort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made

Solution: The trace of quick sort algorithm for E, X, A, M, P, L, E is shown below:

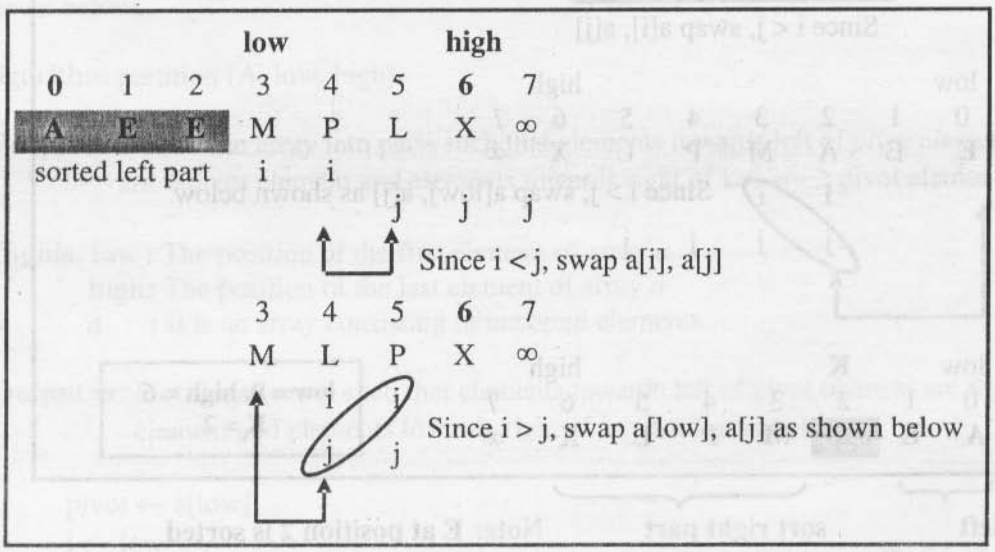
Step 1: Sorting from 0 to 6

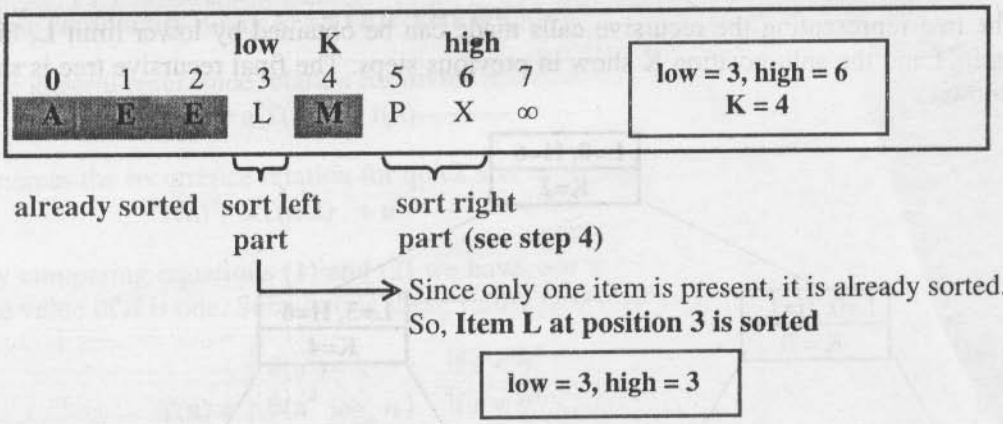


Step 2: Sort from 0 to 1

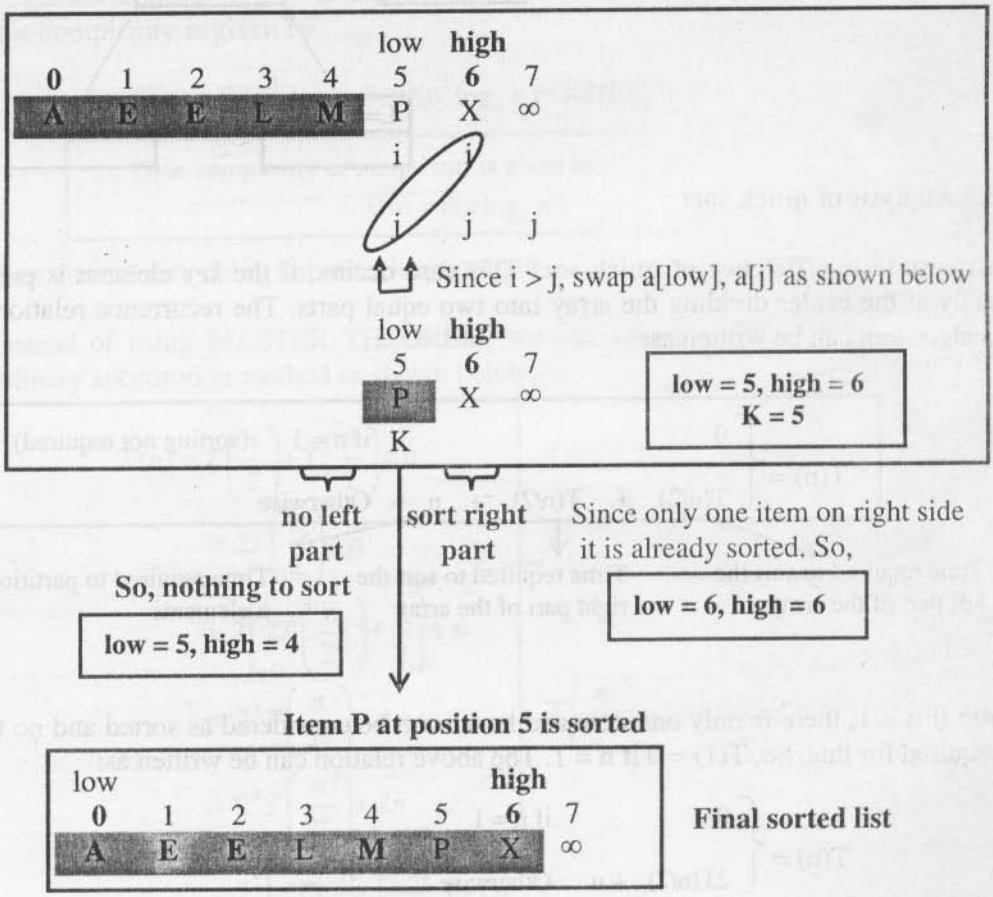


Step 3: Sorting right part from 3 to 6

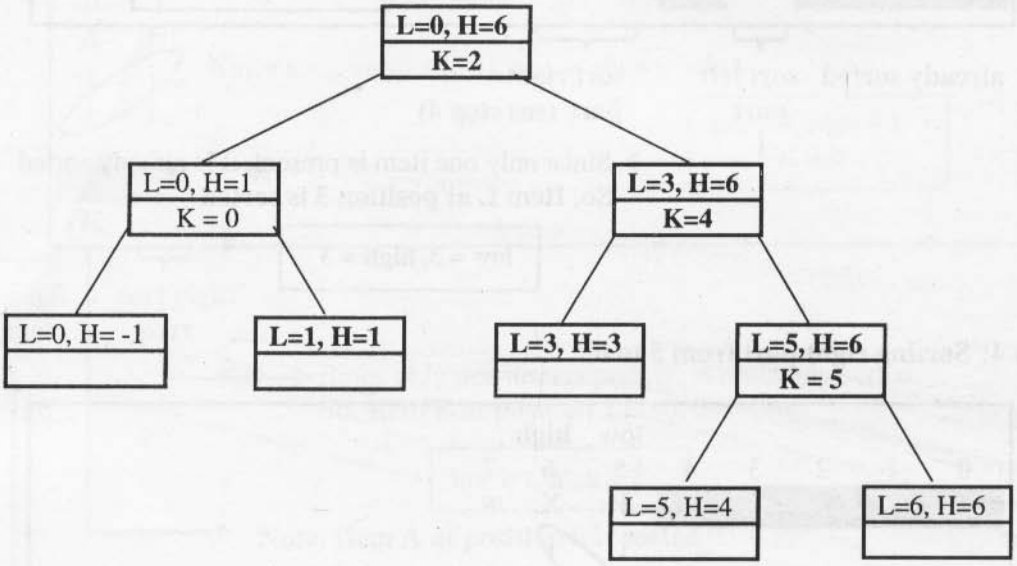




Step 4: Sorting right part from 5 to 6



The tree representing the recursive calls made can be obtained by lower limit L, higher limit H and the split position K show in previous steps. The final recursive tree is shown below:



4.5.3 Analysis of quick sort

Best case time efficiency of quick sort This case occurs, if the *key* element is present exactly at the center dividing the array into two equal parts. The recurrence relation for this algorithm can be written as:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \quad \text{(sorting not required)} \\ \underbrace{T(n/2)}_{\text{Time required to sort the left part of the array}} + \underbrace{T(n/2)}_{\text{Time required to sort the right part of the array}} + \underbrace{n}_{\text{Time required to partition n elements}} & \text{Otherwise} \end{cases}$$

Note: If n = 1, there is only one item and hence can be considered as sorted and no time is required for this. So, T(1) = 0 if n = 1. The above relation can be written as:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{Otherwise} \end{cases}$$

Time complexity using MASTER THEOREM

The general recurrence relation for divide and conquer method is

$$T(n) = a\,T(n/b) + f(n) \text{----------} (1)$$

whereas the recurrence relation for quick sort is given by

$$T(n) = 2T(n/2) + n \text{----------} (2)$$

By comparing equations (1) and (2) we have, a = 2, b = 2 and $f(n) = n = n^1 = n^d$ and so the value of *d* is one. Substituting these values in the following recurrence relation

$$T(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log_b n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

we get a = 2 and $b^d = 2^1 = 2$. It is evident that the second relation $a = b^d$ holds good and time complexity is given by

$$T(n) = \theta(n^d \log_b n) = \theta(n^1 \log_2 n) = \theta(n \log_2 n)$$

$\therefore$, Time complexity of merge sort is given by

$$T(n) = \theta(n \log_2 n)$$

Time complexity using substitution method

Instead of using MASTER THEOREM, we can also find the time complexity using ordinary substitution method as shown below:

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

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

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

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

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

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

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

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2} \quad \text{by replacing n by n/2}$$

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

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

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

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

.....

.....

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

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

→ To get the initial condition T(1) = 0, let $2^i = n$ .....(1)

$$= n.T\left(\frac{n}{n}\right) + n.i$$

$$= n.T(1) + n.i$$

$$= n.0 + n.i = n.i \quad \text{.....(2)}$$

From equation (1) we have $2^i = n$

Taking log on both sides we have,

$$i.\log_2^2 = \log_2^n$$

$$i = \log_2^n$$

Substituting this in equation (2) we have,

$$t(n) = n\log_2^n$$

∴, Time complexity of quick sort is given by

$$T(n) = \theta\,(n\log_2 n)$$

Worst case time efficiency of quick sort

The worst case occurs when, at each invocation of the procedure, the current array is partitioned into two sub arrays with one of them being empty. This situation occurs if all the elements are arranged in ascending order or descending order. For example, the array [22, 33, 44, 55] would yield the following sequence of partitions:

22 [33, 44, 55]
22 33 [44, 55]
22 33 44 [55]

To start with, 22 is the key element. It is clear from the partitioning algorithm that after partitioning, there are no elements towards left of 22 and 3 elements are there towards right of 22. In general, if the array has n elements, after partitioning, n-1 elements will be there towards right and 0 elements will be there towards left of the key element. The recurrence relation for this case is given by:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(0) + T(n-1) + n & \text{Otherwise} \end{cases}$$

where

- T(0) indicates that there are no elements towards left and it can be equated to zero.
- T(n-1) is the time required to sort remaining (n-1) elements
- *n* is the time required to partition the array into two sub arrays.

Now, consider the relation

$$\begin{aligned} T(n) &= T(0) + T(n-1) + n \\ &= T(n-1) + n && \text{By replacing n by n-1 in above eq.} \\ &= T(n-2) + (n-1) + n && \text{By replacing n-1 by n-2 in above eq.} \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\ldots\ldots \\ &\ldots\ldots \\ &\ldots\ldots \\ &= T(n-n) + 1 + 2 + \ldots + (n-2) + (n-1) + n \\ &= 0 + 1 + 2 + \ldots + (n-2) + (n-1) + n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

If $f(n) = \frac{n(n+1)}{2}$, we can have the relation $\frac{n(n+1)}{2} \le n^2$ for $n \ge 0$

which is of the form $f(n) \le c*g(n)$ for $n \ge n_0$

It is clear from the above relations that $c = 1$, $n_0 = 0$, $g(n) = n^2$
By definition, $f(n) = O(g(n))$ i.e., $f(n) = O(n^2)$

$\therefore$, Time complexity of quick sort in the worst case is given by
$T(n) = O(n^2)$

Average case time efficiency of quick sort

In the average case, the given array may not be exactly partitioned into two sub arrays or may not be skewed as in case of worst case. The *key element*(also called pivot element) may be placed at any arbitrary position in the array ranging from 0 to n-1. Let us consider the situation shown below which gives *k* as the position of the partitioning element:

Position of partitioning item

0, 1, 2,.....k-1	k	k+1, k+2,.....n-1
Left part		**Right part**

To compute time to sort left part

lower bound = 0
upper bound = k -1
No. of items on left
= upper – lower bound + 1
= k – 1 – 0 + 1
= k

So, time to sort left part = T(k) -------(1)

To compute time to sort right part

lower bound = k + 1
upper bound = n – 1
No. of items on left
= upper – lower bound + 1
= n – 1 – k – 1 + 1
= n – 1 – k

So, Time to sort right part = T(n – 1 – k) ---(2)

The total number of elements to be sorted is n.
So, the time unit to partition n elements = **n** time units ----------(3)

So, time efficiency of quick sort = time units for partition i.e., n
+ time to sort left part i.e., T(k)
+ time to sort right part

i.e., T(n) = n + T(k) + T(n – 1 – k) from equations (1), (2) and (3)

So, average time efficiency of quick sort is given by the following relation

$$\therefore, T(n) = \frac{1}{n}\sum_{k=0}^{n-1} n + T(k) + T(n-1-k) \text{ for } n > 1$$

Multiplying above equation by n we get

$$nT(n) = \sum_{k=0}^{n-1}\left[n+T(k)+T(n-1-k)\right]$$

$$= \sum_{k=0}^{n-1} n + \sum_{k=0}^{n-1}\left[T(k)+T(n-1-k)\right]$$

$$nT(n) = n\sum_{k=0}^{n-1} 1 + \sum_{k=0}^{n-1}\left[T(k)+T(n-1-k)\right] \quad \text{----------(4)}$$

We know that $n\sum_{k=0}^{n-1} 1 = n(n-1-0+1) = n^2$ ----------(5)

$$\sum_{k=0}^{n-1}\left[T(k)+T(n-1-k)\right] = T(0) + T(n-1) + T(1) + T(n-2) + \ldots\ldots T(n-2) + T(1) + T(n-1) + T(0)$$

$$= 2[T(0) + T(1) + T(2) + \ldots\ldots T(n-1)] \quad \text{----------(6)}$$

Substituting values of equation (5) and (6) in equation (4) we get

$$n.T(n) = n^2 + 2[T(0) + T(1) + T(2) + \ldots\ldots T(n-1)] \quad \text{----------(7)}$$

Replacing n with n-1 in equation (7), we have

$$(n-1)T(n-1) = (n-1)^2 + 2[T(0) + T(1) + T(2) + \ldots\ldots T(n-2) \quad \text{----------(8)}$$

Now, equation (7) – equation(8) gives

$$nT(n) - (n-1)T(n-1) = n^2 - n^2 + 2n - 1 + 2.T(n-1)$$

$$= 2n - 1 + 2T(n-1)$$

i.e., $nT(n) = (n-1)T(n-1) + 2n - 1 + 2T(n-1)$

$$nT(n) = T(n-1)(n-1+2) + 2n - 1$$

$$nT(n) = T(n-1)(n+1) + 2n \quad \text{by neglecting -1}$$

Dividing the above equation by n(n + 1) we have

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

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

Replacing n by n-1 we have

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

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

Replacing n by n-1 we have

$$\frac{T(n-2)}{(n-1)} = \frac{T(n-3)}{n-2} + \frac{2}{n-1}$$

$$= \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

Similarly, next term can be written as shown below:

$$= \frac{T(n-4)}{n-3} + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

.....

.....

$$= \frac{T(0)}{1} + \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{n} + \frac{2}{n+1}$$

$$= 0 + \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{n} + \frac{2}{n+1}$$

$$= 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1}\right)$$

$$= 2\sum_{k=2}^{n+1} \frac{1}{k} = 2\int_{2}^{n+1} \frac{1}{k} dk = 2[\log k]_2^{n+1}$$

$$= 2[\log_2(n+1) - \log_2 2] \approx 2\log_2(n+1) \text{ for very large value of } n$$

i.e, $\dfrac{T(n)}{(n+1)} = 2\log_2(n+1)$

$\therefore$, $T(n) = (n+1)2\log_2(n+1) \approx n\log_2 n$ for very large value of n.

$\therefore$, Time complexity of quick sort in the average case is given by $T(n) = \theta(n\log_2 n)$

Note: Even though it is an approximate value, on an average, quick sort takes around 38% more comparisons than the best case and runs faster than merge sort.

Advantages

- In-Place: The quick sort algorithm is in-place since it uses only a small auxiliary stack.
- Time complexity: Quick sort requires time complexity of O(n log n) in the best case and average case to sort n items.
- It has an extremely short inner loop
- A very precise statement can be made about performance issues.

Disadvantages

- It is not stable
- Since the algorithm is recursive, it is very difficult for the beginners to grasp. So, this algorithm requires a higher level of knowledge on the part of the programmer.
- Time complexity: The time complexity of quick sort is quadratic *i.e.,* $O(n^2)$ time in the worst-case.
- It is fragile i.e., a simple mistake in the implementation can go unnoticed and algorithm may not work

4.5.4 Binary search

To overcome the disadvantages of *linear search,* we generally use *binary search* when the elements are *sorted.* Now, let us see "*What is binary search? What is the concept used in binary search?*"

Definition: A *binary search* is a simple searching technique which can be applied if the items to be compared are either in ascending order or descending order.

The general idea used in binary search is similar to the way we search for the telephone number of a person in the telephone directory. Obviously, we do not use *linear search.* Instead, we open the book in the middle and the *name* is compared with the element at the middle of the book. If the name is found, the corresponding telephone number is retrieved and the searching has to be stopped. Otherwise, we search either the left part of the book or right part of the book. If the *name* to be searched is less than the middle element, search towards left otherwise, search towards right. The procedure is repeated till *key* item is found or the *key* item is not found.

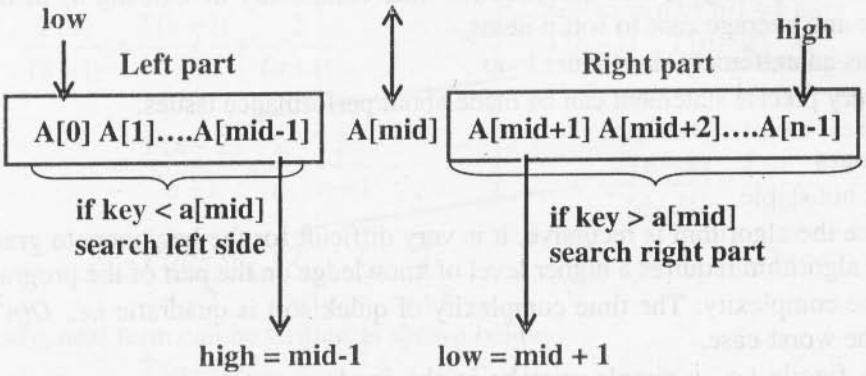
Once we know the concept of *binary search*, the next question is "*How to search for key in a list of elements?*" Now, let us see how to search for an item. The procedure is shown below:

Procedure: The program can be designed as follows. If *low* is considered as the position of the first element and *high* as the position of the last element, the position of the middle element can be obtained using the statement

mid = (low + high) /2;

The *key* to be searched is compared with middle element. This can be done using the statement

if (key = a[mid]) return mid;



Suppose *key* item is not present at the *mid* position. In that case, the *key* item may be on the left side of the array or on the right side of the array as shown in above figure. This can be achieved using the following statement.

```
if ( key < a[mid] )
        high ← mid – 1
else
        low ← mid + 1
```

Note: In general, binary search works by comparing an *item* with middle element a[mid]. If they are same, the algorithm is terminated; otherwise, the same operation is repeated for the left side of the array or for the right side of the array.

The complete algorithm for binary search is shown below:

Algorithm binary_search(item, a, n)

//**Purpose:** Search for an item in the list identified by *a* using binary search

//**Input**:
// item- the element to be searched
// a – the list of elements where the searching takes place
// n – the number of elements in the list

//**Output**:
// position is returned if search is successful otherwise, -1 is returned

```
low ← 0                                  // Points to the first element(page)
high ← n-1                               // Points to the last element(page)

while ( low <= high )
    mid ← ( low + high ) / 2             // Divide the list into two equal parts

    if ( item = a[mid] ) return mid+1    // return the position if search successful

    if ( item < a[mid] )                 // Item not found in the middle of the list
        high ← mid – 1                   // Search in the left side of the array
    else
        low ← mid + 1                    // Search in the right side of the array
    end if
end while

return -1                                // Item is not present in the list
```

Note: The necessary condition for binary search is that the list of elements should be sorted. Otherwise, binary search should not be used. If elements are not sorted and are very less, normally we go for linear search. But, if the elements are more and not sorted, it is better to sort them and use binary search for the following reasons.

Analysis (Best case) The best case occurs when the item to be searched is present in the middle of the array. So, the total number of comparisons required will be 1. Therefore, the time complexity of binary search in the best case is given by $\Omega(1)$.

Analysis (Worst case) The worst case occurs when the *key* to be searched is in either at the first position or at the last position in the array. In such situations, maximum number of element comparisons are required and the time complexity is given by the following recurrence relation:

Number of comparisons if n = 1

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{Otherwise} \end{cases} \quad \text{----------(1)}$$

Time required to compare the middle element

Time required to search either the upper part or the lower part of the array

Now, consider the relation:

$$t(n) = t\left(\frac{n}{2}\right) + 1 \qquad t(n) = 1 + t\left(\frac{n}{2}\right) \text{ ----------(2)}$$

$$t(n) = 1 + t\left(\frac{n}{2}\right) \qquad t\left(\frac{n}{2}\right) = 1 + t\left(\frac{n}{4}\right) = 1 + t\left(\frac{n}{2^2}\right) \text{ replacing n by } \frac{n}{2} \text{ in eq (2)}$$

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

$$= 2 + t\left(\frac{n}{2^2}\right) \qquad t\left(\frac{n}{2^2}\right) = 1 + t\left(\frac{n}{2^3}\right) \text{ By replacing n by } \frac{n}{2^2} \text{ in equation (2)}$$

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

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

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

Note: By looking at the previous expression

.....

.....

In general,

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

.....> Finally, to get initial condition t(1), let $2^i = n$ ----------(3)

$$= i + t(1) \qquad \mathbf{t(1) = 1} \text{ from equation (1)}$$

$$t(n) = i \text{ ----------(4)}$$

From equation (3) we have $2^i = n$. Taking log on both sides, we have

$$i * \log_2 2 = \log_2 n$$

$$i = \log_2 n$$

Substituting the value of i in equation (4) we have,

$$t(n) = \log_2 n$$

So, time complexity is given by $t(n) \in \theta(\log_2 n)$

Note: Before considering the average case, let us find out the number of comparisons required when n is 1, 2, 4, 8 etc. Consider the various examples shown below:

case 1: When n = 1

a[0] = 10. Here, number of elements = 1 and number of comparisons = 1

So, when $n = 1 = 2^0$, the number of comparisons = 1

case 2: When n = 2, item = 20

a[0]	a[1]
10	20

After finding the mid point, one comparison at mid point. If item not equal, we have to search either the left or right part which has one item (which requires 1 comparison as in case 1). So, total number of comparisons = 2

So, when $n = 2 = 2^1$, the number of comparisons = 2

case 3: When n = 4, item = 40

a[0]	a[1]	a[2]	a[3]
10	20	30	40

After finding the mid point, one comparison at mid point. If item not equal, we have to search either the left or right part which has two items (which requires 2 comparison as in case 2). So, total number of comparisons = 3

So, when $n = 4 = 2^2$, the number of comparisons = 3

case 4: When n = 8, item = 80

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
10	20	30	40	50	60	70	80

After finding the mid point, one comparison at mid point. If item not equal, we have to search either the left or right part which has four items (which requires 3 comparison as in case 3). So, total number of comparisons = 4

So, when $n = 8 = 2^3$, the number of comparisons = 4

Analysis (Average case) Observe the following facts from case 1 to case 4:

So, when n = 1 = 2^0, the number of comparisons = 1
So, when n = 2 = 2^1, the number of comparisons = 2
So, when n = 4 = 2^2, the number of comparisons = 3
So, when n = 8 = 2^3, the number of comparisons = 4

So, when n=2^{k-1}, the number of comparisons = k .....(1)

So, average number of comparisons of binary search is given by the following relation:

$$T(n) = \frac{1}{n}\sum_{i=1}^{k} i.2^{i-1} \quad \text{.....(2)}$$

We know that $2^{i-1} = \frac{2^i}{2} = 2^i - \frac{2^i}{2} = 2^i - 2^{i-1}$ .....(3)

Substituting equation (3) in equation (2) we have,

$$= \frac{1}{n}\sum_{i=1}^{k} i\left(2^i - 2^{i-1}\right)$$

$$= \frac{1}{n}\left\{\sum_{i=1}^{k} i2^i - \sum_{i=1}^{k} i2^{i-1}\right\}$$

$$= \frac{1}{n}\left\{\sum_{i=1}^{k} i2^i - \sum_{i=0}^{k-1} (i+1)2^i\right\}$$

$$= \frac{1}{n}\left\{\sum_{i=1}^{k} i2^i - \sum_{i=0}^{k-1} i2^i - \sum_{i=0}^{k-1} 2^i\right\}$$

$$= \frac{1}{n}\left\{\sum_{i=1}^{k} i2^i - \sum_{i=1}^{k-1} i2^i - \sum_{i=0}^{k-1} 2^i\right\}$$

$$= \frac{1}{n}\left\{k2^k - \sum_{i=0}^{k-1} 2^i\right\}$$

$$= \frac{1}{n}\left\{k2^k - \left[1 + 2^1 + 2^2 + \ldots 2^{k-1}\right]\right\}$$

$$= \frac{1}{n}\left\{k2^k - 1\left(\frac{2^k - 1}{2-1}\right)\right\} \text{ using } S = a\left(\frac{r^n - 1}{r-1}\right) \text{ where a = 1, r = 2, n = k}$$

$$= \frac{1}{n}\{kn - 1(n-1)\} = \frac{1}{n}\{kn - n + 1)\} = k - 1 + \frac{1}{n}$$

As the value of *n* increases, it is approximated to k-1

i.e., t(n) = k-1 .....(4)

From, equation (1) we have $n = 2^{k-1}$. Taking log on both sides we have

$$\log_2 n = (k-1)\log_2^2$$

$$\log_2 n = k-1$$

Substituting this value is equation (4) we get,

$$t(n) = \log_2 n$$

$\therefore$, Time complexity of binary search in the average case is given by

$$T(n) = \theta(\log_2 n)$$

Advantages of binary search

- Simple technique
- Very efficient searching technique

Disadvantages of binary search

- The array should be sorted

Now, let us "Compare binary search with linear search"

Binary search	Linear search
♦ Works only on sorted items	♦ Works on sorted as well as unsorted items
♦ Very efficient if the items are sorted	♦ Very efficient if the items are less and present in the beginning of the list
♦ Works well with arrays and not on linked lists	♦ Works with arrays and linked lists
♦ Time complexity is O(log n)	♦ Time complexity is O(n)

4.5.5 Binary tree traversals

Now, let us see "What is a binary tree?"

Definition: A binary tree is a directed tree in which *outdegree* of each node is less than or equal to two i.e., each node in the tree can have 0, 1 or 2 children. In other words, a

binary tree is a tree which can be empty or partitioned into three subgroups namely root, left subtree and right subtree:

- **Root** – The first node in the tree which does not have a parent is called root node.
- **left subtree** – It is a tree which is connected to the left of root. Since this tree comes under root, it is called left subtree.
- **right subtree** – It is a tree which is connected to the right of root. Since this tree comes under the root, it is called right subtree.

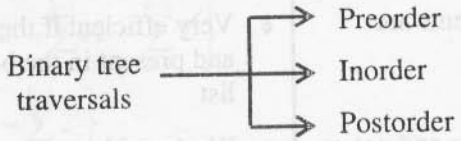
Note: In general, a tree in which each node has either zero, one or two subtrees is called a binary tree. If any node has more than two subtrees, then it is not a binary tree.

Note that the definition of binary tree itself divides a binary tree into two subsets of the same type namely left subtree and right subtree. So, many problems on binary trees can be solved using divide and conquer. For example, the following operations on trees can be solved very efficiently using divide and conquer technique:

- Tree traversals
- Height of the tree
- Number of nodes in a binary tree
- Number of leaf nodes in a binary tree etc.,

Now, let us see "What is the meaning of traversing a tree?"

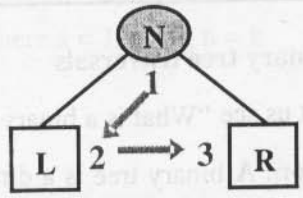
Definition: Traversing is a method of visiting each node of a tree exactly once in a systematic order. During traversal, we may print the **info** field of each node visited. Now, let us see "What are the different traversal techniques of a binary tree?" The various traversal techniques of a binary tree are shown below:



Now, let us see "What is preorder traversing of a binary tree?"

Definition: The preorder traversal of a binary tree can be recursively defined as follows:

1. Process the root Node [N]
2. Traverse the Left subtree in preorder [L]
3. Traverse the Right subtree in preorder [R]



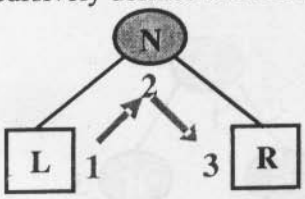
The Algorithm to traverse the tree in preorder is shown below:

```
Algorithm preorder(T)

if ( T = NULL ) return

print (info(T))          // visit the node
preorder(T_L)            //visit left subtree T_L in preorder
preorder(T_R)            // visit right subtree T_R in preorder
```

Now, let us see "What is inorder traversing of a binary tree?"

Definition: The inorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the **Left** subtree in inorder **[L]**
2. Process the root **Node [N]**
3. Traverse the **Right** subtree in inorder **[R]**



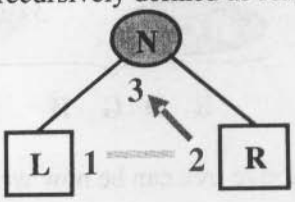
The algorithm to traverse the tree in inorder is shown below:

```
Algorithm inorder(T)

if ( T = NULL ) return;

inorder(T_L);            // visit left-subtree T_L in inorder
print(info(T))           // visit the node
inorder(T_R)             // visit right-subtree T_R in inorder
```

Now, let us see "What is postorder traversing of a binary tree?"

Definition: The postorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the **Left** subtree in postorder **[L]**
2. Traverse the **Right** subtree in postorder **[R]**
3. Process the root **Node [N]**

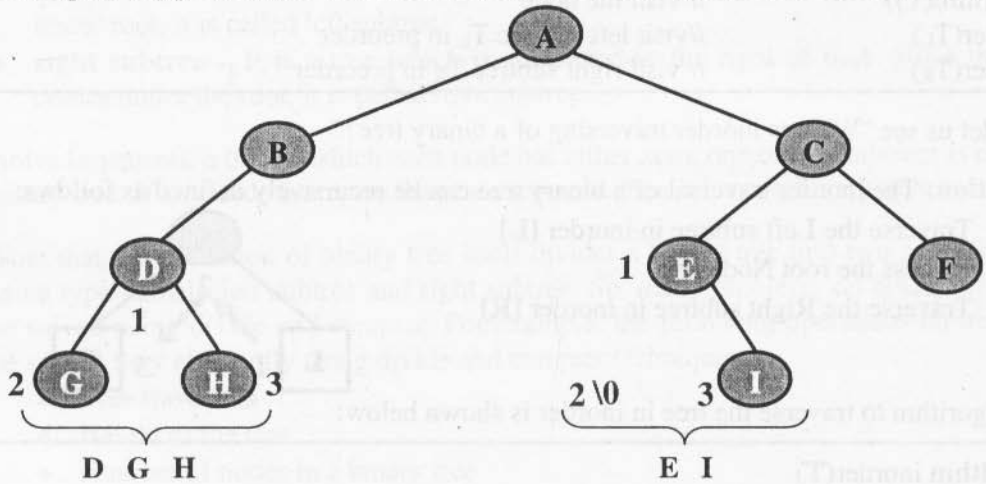


The algorithm to traverse the tree in postorder is shown below:

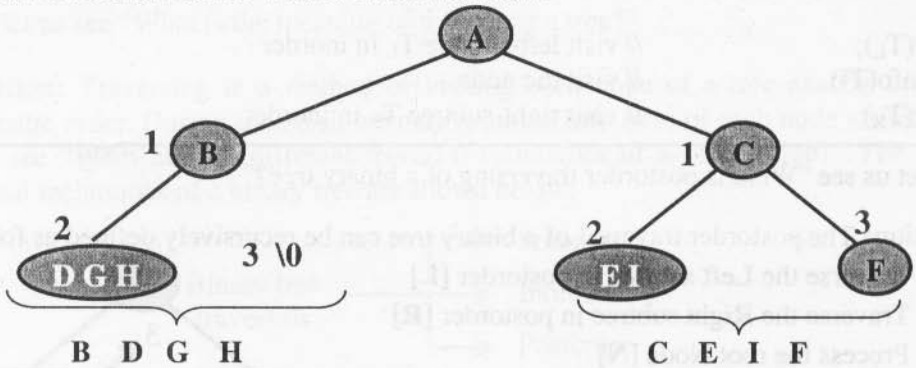
```
Algorithm postorder(T)

if ( T = NULL ) return;

postorder(T_L)           // visit left subtree T_L in postorder
postorder(T_R)           // visit right subtree T_R in postorder
print(info(T))           // visit the node
```

Example 4.5.5.1: Traverse the following tree in preorder

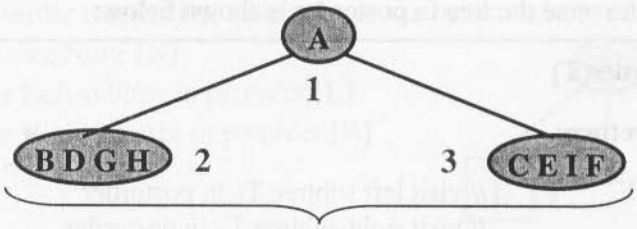
Solution: Easiest method of writing preorder traversal is to consider the leaves and the corresponding parents and visit the nodes in the manner specified for the preorder.



The above tree now can be written as shown below:

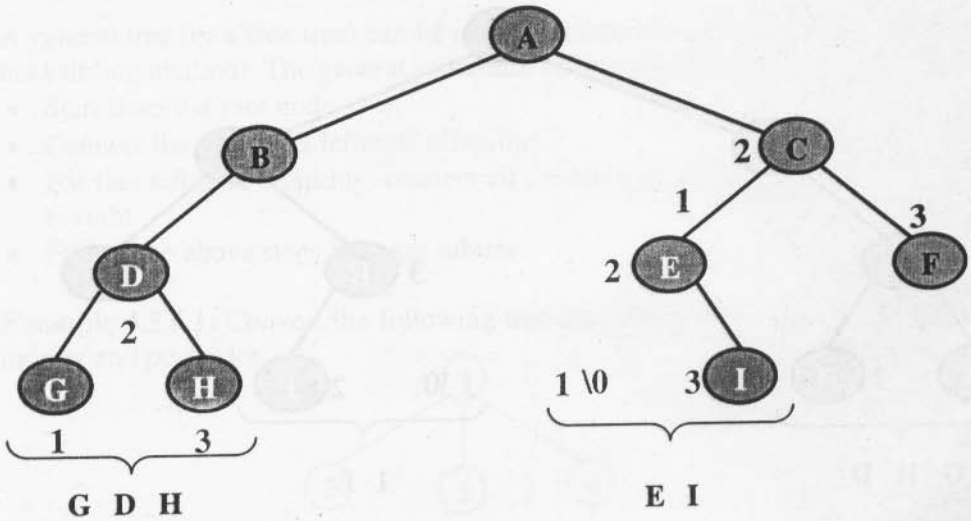


The above tree can be now written as shown below:

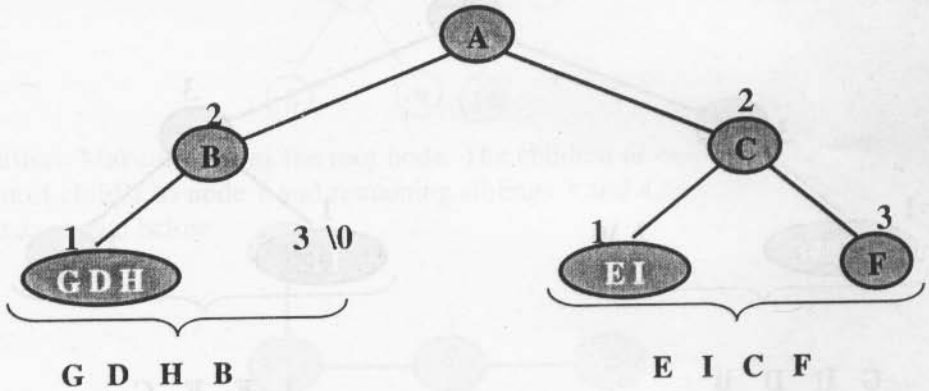


So, preorder traversal is A B D G H C E I F

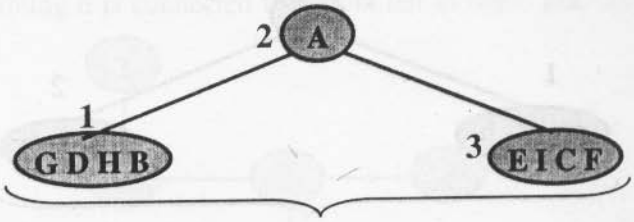
Example 4.5.5.2: Traverse the following tree inorder:



The above tree now can be written as:

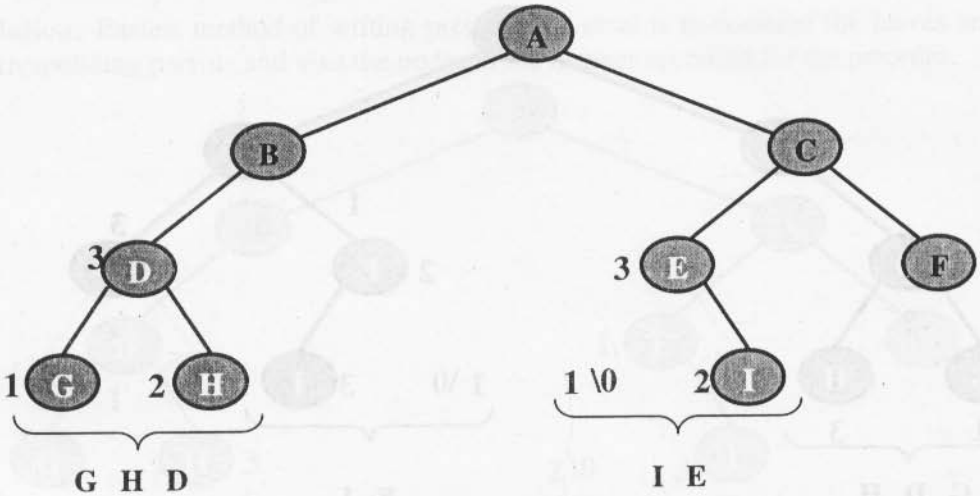


The above tree can be now written as shown below:

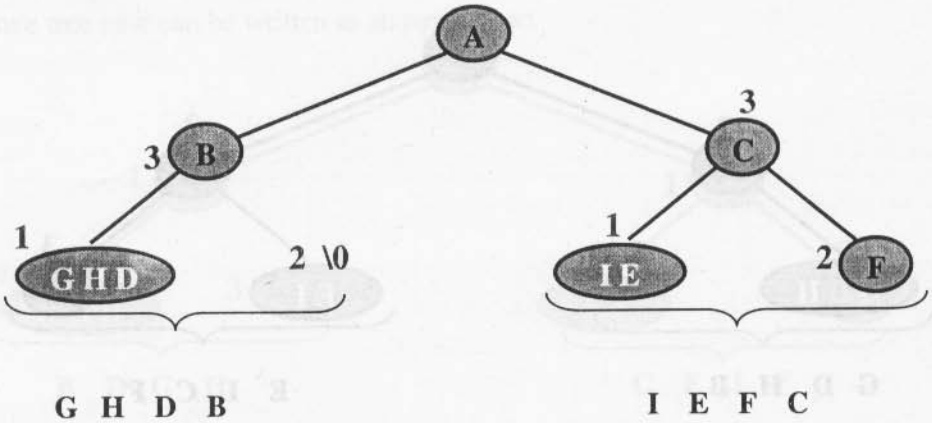


In order traversal is **G D H B A E I C F**

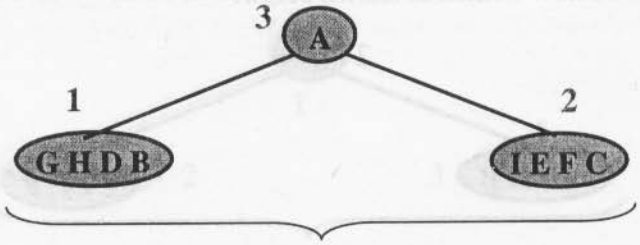
Example 4.5.5.3: Traverse the following tree in postorder



The above tree now can be written as:



The above tree can be now written as shown below:



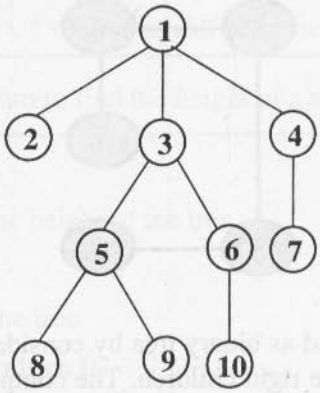
Postorder traversal is **G H D B I E F C A**

4.5.6 Conversion of general tree to a binary tree (first child-next sibling method)

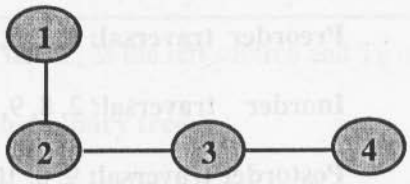
A general tree (or a free tree) can be converted into an equivalent binary tree (first child-next sibling method). The general procedure followed in the algorithm is shown below:

- Start from the root node
- Connect the root to its leftmost offspring.
- For this leftmost offspring, connect all the siblings which are in same level from left to right.
- Follow the above steps for each subtree.

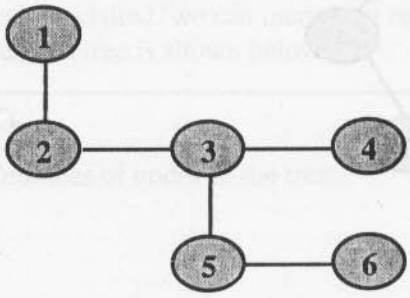
Example 4.5.6.1: Convert the following tree into binary tree and traverse it in preorder, inorder and postorder.



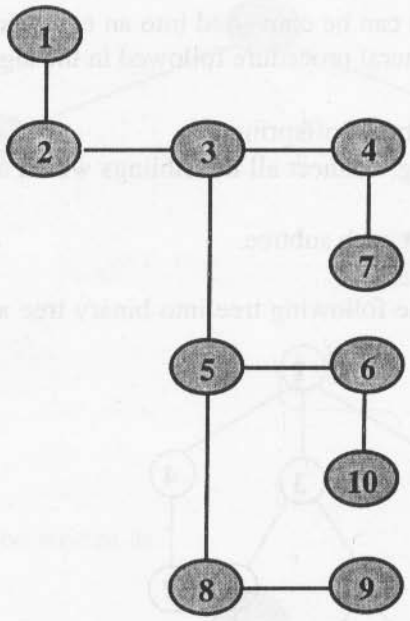
Solution: Make node 1 as the root node. The children of node 1 are 2, 3 and 4. Connect leftmost child 2 to node 1 and remaining siblings 3 and 4 are connected to 2 from left to right as shown below:



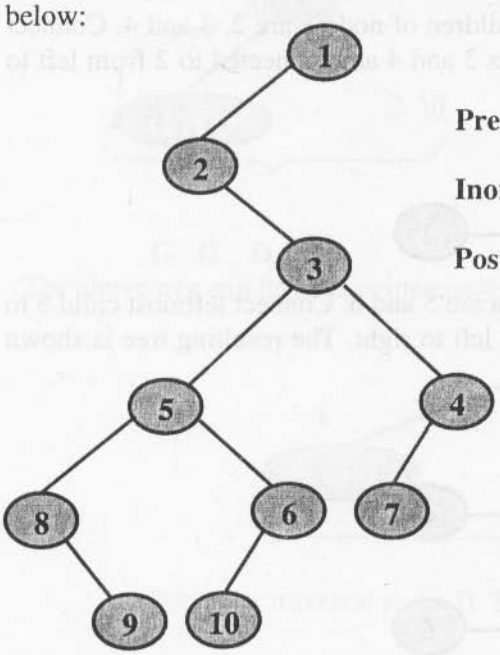
For the subtree whose root node is 3, the children are 5 and 6. Connect leftmost child 5 to the parent. The sibling 6 is connected to 5 from left to right. The resulting tree is shown below:



By repeating the same procedure for all the subtrees, the final tree can be written as shown below:



Now, the above tree can be represented as binary tree by considering the vertical lines as left children and horizontal lines as the right children. The complete binary tree is shown below:



Preorder traversal: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7

Inorder traversal: 2, 8, 9, 5, 10, 6, 3, 7, 4, 1

Postorder traversal: 9, 8, 10, 6, 5, 7, 4, 3, 2, 1

4.5.7 Height of a binary tree

Now, let us "Define the height of the tree"

Definition: The height of a tree is one more than the maximum level of a tree. In other words, it is defined as the length of the longest path from root node to a leaf node. The height of the tree can be obtained by finding the maximum of the heights of left subtree and right subtree and adding one to it. The recursive definition to find the height of the tree is shown below:

$$\text{Height}(T) = \begin{cases} 0 & \text{if } T = \text{NULL} \\ 1 + \max(\text{height}(TL), \text{height}(T_R)) & \text{otherwise} \end{cases}$$

The corresponding algorithm to find the height of a tree is shown below:

```
Algorithm height(T)
// Purpose: To compute the height of the tree
// Input: A binary tree
// Output: The height of the tree
        if ( T == NULL ) return 0
        return 1 + max( height(TL), height(TR) )
// end of algorithm
```

Note: T is the root of the tree T_L is the left subtree and T_R is the right subtree

4.5.8 Number of nodes in a binary tree

Now, let us see "How to find the number of nodes in a tree?" The number of nodes in the tree is obtained by traversing the tree in any of the traversal technique and increment the counter whenever a node is visited. We can use one global variable by name *count* with initial value 0. As each node is visited, we can increment *count* bye one. The algorithm to obtain the number of nodes in a tree is shown below:

```
Algorithm count_node(T)
// Purpose: To count the number of nodes in the tree
// Input : The binary tree
```

```
// Output: Number of nodes in the tree
        if ( T = NULL ) return;
        count_node(T_L)                 // Traverse recursively towards left
        count ← count + 1               // Update the count
        count_node(T_R)                 // Traverse recursively towards right
// end of the algorithm
```

4.5.9 Number of leaves in a binary tree

Now, let us see "How to find the number of leaves in a tree?" The number of leaves in the tree is obtained by traversing the tree in any of the traversal technique and count the number of nodes if and only if the node visited has empty left and empty right child. We can use one global variable by name *count* with initial value 0. As each node is visited, we can increment *count* bye one if the left child and right child of a node is NULL. The algorithm to obtain the number of leaves in a tree is shown below:

```
Algorithm count_leaf(T)
// Purpose: To count the number of leaves in a tree
// Input: A binary tree
// Output: The number of leaves in a tree
if ( T = NULL ) return
count_leaf(T_L);                        // Traverse recursively towards left
if (T_L = NULL && T_R = NULL )          // if the node is a leaf
        count ← count + 1               //      update the count
end if
count_leaf(T_R)                         // Traverse recursively towards right
// End of the algorithm
```

4.6 Multiplication of Large Integers

Modern cryptology and various applications require manipulation of long integers often exceeding more than 100 decimal digits. Note that these numbers can not be stored in a single word of modern computer and requires special data structures to store and manipulate. In "Data Structures" we have seen how to store and add two long positive numbers. In this section, let us see "How very long numbers can be multiplied using divide and conquer method? Explain with example"

Example 4.6.1: Multiply two numbers **a = 1234** and **b = 5678** using divide and conquer method

The given numbers can be split equally into two parts as shown below:

$a_1 = 12$ and $a_0 = 34$ .....(1)
$b_1 = 56$ and $b_0 = 78$

Now the value of *a* can be obtained using a_1 and a_0 as shown below:

$$a = 10^2 * a_1 + 10^0 * a_0 \quad \text{.....(2)}$$

i.e., $a = 10^2 * 12 + 10^0 * 34 = 1200+34 = 1234$

On similar lines *b* = 5678 can be obtained from b_1 and b_0 as shown below:

$$b = 10^2 * b_1 + 10^0 * b_0 \quad \text{.....(3)}$$

i.e., $b = 10^2 * 56 + 10^0 * 78 = 5600 + 78 = 5678$

Now, the two numbers 1234 and 5678 can be multiplied using equation (2) and (3) as shown below:

$$a * b = \underbrace{(10^2 a_1 + 10^0 a_0)}_{\text{(equation 1)}} \underbrace{(10^2 b_1 + 10^0 b_0)}_{\text{(equation 2)}}$$

$$= 10^4 a_1 b_1 + 10^2 (a_1 b_0 + b_1 a_0) + 10^0 a_0 b_0 \quad \text{.....(4)}$$

Substituting the values for a_1, a_0, b_1 and b_0 from equation (1), into equation (4), we have

$$\begin{aligned} 1234*5678 &= 10^4*12*56 + 10^2(12*78 + 56*34) + 10^0*34*78 \\ &= 6720000 + 284000 + 2652 \\ &= 7006652 \end{aligned}$$

Note: In equation (4) we have four multiplications

a_1b_1
a_1b_0
b_1a_0
a_0b_0

Note: 10^4 and 10^2 can be computed by left shifting and hence not considered as multiplication terms.

Now, let us see "How four multiplications can be reduced to three multiplications?" so that number of multiplications are reduced. Let us consider the multiplication of two terms $(\mathbf{a_1 + a_0})$ and $(\mathbf{b_1 + b_0})$ as shown:

$$(a_1 + a_0)\,(b_1 + b_0) = a_1b_1 + \mathbf{a_1b_0} + \mathbf{b_1a_0} + a_0b_0$$

From the above equation, we have

$$a_1b_0 + b_1a_0 = (a_1 + a_0)\,(b_1 + b_0) - a_1b_1 - a_0b_0 \text{.....}\mathbf{(5)}$$

Substituting equation (5) in equation (4) we have:

$$a*b = 10^4a_1b_1 + 10^2[(a_1 + a_0)\,(b_1 + b_0) - a_1b_1 - a_0b_0] + 10^0a_0b_0$$
$$a*b = 10^4a_1b_1 + 10^2[(a_1 + a_0)\,(b_1 + b_0) - a_1b_1 - a_0b_0] + a_0b_0 \text{.....}\mathbf{(6)}$$

Note: In the above equation, there are three multiplications:

a_1b_1
$(a_1 + a_0)\,(b_1 + b_0)$
a_0b_0

Note that in equation (6) even though a_1b_1 and a_0b_0 appears twice, the good compilers will compute a_1b_1 and a_0b_0 only once.

Now, let us see "How a and b are multiplied using equation (6)?" Substituting the values for a_1, a_0, b_1 and b_0 from equation (1), into equation (6), we have:

$$\begin{aligned} a * b = 1234*5678 &= 10^4 * a_1 * b_1 + 10^2[(a_1 + a_0)\,(b_1 + b_0) - a_1*b_1 - a_0*b_0] + a_0*b_0 \\ &= 10^4*12*56 + 10^2\,[(12+34)\,(56+78) - 12*56 - 34*78] + 34*78 \\ &= 6720000 + 100[46*134 - 672 - 2652] + 2652 \\ &= 6720000 + 284000 + 2652 \\ &= \mathbf{7006652} \end{aligned}$$

Note: With just three terms we could able to multiply two numbers. Just by this example, it is not possible to tell the significance of reducing the number of multiplications by one. But, this procedure is very useful and efficient if the numbers to be multiplied have more than 600 digits.

Analysis It is clear from the example that using divide and conquer method, two n-digit numbers requires 3 multiplications of n/2-digit numbers and the recurrence relation is given by:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) & \text{Otherwise} \end{cases} \quad \ldots\ldots\ldots\ldots(1)$$

Consider the relation

$$T(n) = 3.T\left(\frac{n}{2}\right)$$

Replacing n by n/2 in above equation, we have

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

$$T(n) = 3.T\left(\frac{n}{2}\right)$$

$$= 3.3\left(\frac{n}{2^2}\right)$$

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

The next terms can be easily obtained a

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

$$= 3^4\left(\frac{n}{2^4}\right)$$

.....

.....

In general,

$$= 3^i\left(\frac{n}{2^i}\right)$$

To get the terminal condition T(1) = 1 from equation (1)

let $\mathbf{n = 2^i}$ .....(2)

$$= 3^i\left(\frac{n}{n}\right) = 3^i.1 = 3^i$$

So, $T(n) = 3^i$ .....(3)

From equation (2) we have $2^i = n$. So, taking log on both sides we get:

$$i.\log_2^2 = \log_2^n$$

$$i = \log_2^n$$

Substituting this in equation (3) we have,

$$t(n) = 3^{\log_2^n} = n^{\log_2^3} = n^{1.585}$$

Note: $a^{\log_2 b} = b^{\log_2 a}$

∴, Time complexity to multilply two integers using divide and conquer is given by

$$T(n) = \theta(n^{1.585})$$

4.7 Strassen's matrix multiplication

We learnt that the time complexity of an algorithm to multiply two matrices using brute force method is $\theta(n^3)$. But, Strassen V has found out a new method where the number of multiplications can be reduced there by increasing the efficiency. This algorithm is called Strassen's Matrix multiplication algorithm and can be accomplished by using the following formulas:

$$\begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix} = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} * \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{pmatrix}$$

where

$m_1 = (A_1 + A_4) * (B_1 + B_4)$
$m_2 = (A_3 + A_4) * B_1$
$m_3 = A_1*(B_2-B_4)$
$m_4 = A_4*(B_3-B_1)$
$m_5 = (A_1 + A_2) * B_4$
$m_6 = (A_3 - A_1) * (B_1 + B_2)$
$m_7 = (A_2 - A_4) * (B_3 + B_4)$

It is clear from the above relations that, Strassen's matrix multiplication algorithm require 7 multiplications. For example, consider the two matrices A and B shown below and let us multiply using this procedure:

$$A \overset{2 \times 2}{\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}} \times \overset{2 \times 2}{\begin{bmatrix} 8 & 7 \\ 1 & 2 \end{bmatrix}} B$$

where $A_1=1$, $A_2=2$, $A_3=5$, $A_4=6$ $B_1=1$, $B_2=7$, $B_3=1$, $B_4=2$. Substituting these values in following formulas we get:

$$\begin{aligned}
m_1 &= (A_1 + A_4) * (B_1 + B_4) &&= 8\,(1 + 6)\,(2 + 8) &&= 70 \\
m_2 &= (A_3 + A_4) * B_1 &&= (5 + 6)\,(8) &&= 88 \\
m_3 &= A_1*(B_2-B_4) &&= (1)\,(7 - 2) &&= 5 \\
m_4 &= A_4*(B_3-B_1) &&= (6)\,(1 - 8) &&= -42 \\
m_5 &= (A_1 + A_2) * B_4 &&= (1 + 2)\,(2) &&= 6 \\
m_6 &= (A_3 - A_1) * (B_1 + B_2) &&= (5 - 1)\,(8 + 7) &&= 60 \\
m_7 &= (A_2 - A_4) * (B_3 + B_4) &&= (2 - 6)\,(1 + 2) &&= -12
\end{aligned}$$

$$C = \begin{bmatrix} 70 - 42 - 6 - 12 & 5 + 6 \\ 88 - 42 & 70 + 5 - 88 + 60 \end{bmatrix}$$

$$C = \begin{bmatrix} 10 & 11 \\ 46 & 47 \end{bmatrix}$$ **is the product of given two matrices**

Analysis: Note that Strassen's algorithm is more efficient than the conventional matrix multiplication for sufficiently large value of *n*, but not for smaller values of *n*. The recurrence relation for this can be written as:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/2) & \text{Otherwise} \end{cases} \quad \text{.....(1)}$$

Consider the relation

$$T(n) = 7.T\left(\frac{n}{2}\right)$$

$$= 7.7.T\left(\frac{n}{2^2}\right)$$

$$= 7^2.T\left(\frac{n}{2^2}\right)$$

$$T(n) = 7.T\left(\frac{n}{2}\right)$$

Replacing n by n/2 in above equation, we have

$$T\left(\frac{n}{2}\right) = 7.T\left(\frac{n}{2^2}\right)$$

The next terms can be easily obtained a

$$= 7^3\left(\frac{n}{2^3}\right)$$

$$= 7^4\left(\frac{n}{2^4}\right)$$

.....

.....

In general,

$$= 7^i\left(\frac{n}{2^i}\right)$$

To get the terminal condition T(1) = 1 from equation (1) let $n = 2^i$ .....(2)

$$= 7^i\left(\frac{n}{n}\right) = 7^i.1 = 7^i$$

So, $T(n) = 7^i$ .....(3)

From equation (2) we have $2^i = n$. So, taking log on both sides we get:

$$i.\log_2^2 = \log_2^n$$

$$i = \log_2^n$$

Substituting this in equation (3) we have,

$$t(n) = 7^{\log_2^n} = n^{\log_2^7} = n^{2.807}$$

Note: $a^{\log_2 b} = b^{\log_2 a}$

∴, Time complexity to multilply two integers using divide and conquer is given by

$$T(n) = \theta(n^{2.807})$$

So, it is very clear from the time complexities obtained for various methods while multiplying two matrices that Strassen's matrix multiplication algorithm is more efficient compared to that of conventional matrix multiplication for very large value of *n*.

Note: This algorithm works fine if the size of the matrix is always a power of 2. If the size of the matrix is not a multiple of 2, add sufficient zeros both in rows and columns so that the size of the matrices are always a multiple of 2. Also note that size of the matrices to be multiplied should have the same size.

Example 2: Apply Strassen's algorithm to multiply the following two matrices:

0	1	0	1
2	1	0	4
2	0	1	1
1	3	5	0

exiting the recursion when n = 2 i.e., computing the products of 2 by 2 matrices by brute force algorithm

Solution:

Let A =

A1	A2
1 0 / 4 1	2 1 / 1 0
0 1 / 5 0	3 0 / 2 1
A3	A4

$$A1 = \begin{vmatrix} 1 & 0 \\ 4 & 1 \end{vmatrix} \quad A2 = \begin{vmatrix} 2 & 1 \\ 1 & 0 \end{vmatrix}$$

$$A3 = \begin{vmatrix} 0 & 1 \\ 5 & 0 \end{vmatrix} \quad A4 = \begin{vmatrix} 3 & 0 \\ 2 & 1 \end{vmatrix}$$

Let B =

B1	B2
0 1 / 2 1	0 1 / 0 4
2 0 / 1 3	1 1 / 5 0
B3	B4

$$B1 = \begin{vmatrix} 0 & 1 \\ 2 & 1 \end{vmatrix} \quad B2 = \begin{vmatrix} 0 & 1 \\ 0 & 4 \end{vmatrix}$$

$$B3 = \begin{vmatrix} 2 & 0 \\ 1 & 3 \end{vmatrix} \quad B4 = \begin{vmatrix} 1 & 1 \\ 5 & 0 \end{vmatrix}$$

Substituting these values in following formulas we get:

$$m_1 = (A_1 + A_4) * (B_1 + B_4)$$

$$\begin{vmatrix} 4 & 0 \\ 6 & 2 \end{vmatrix} * \begin{vmatrix} 1 & 2 \\ 7 & 1 \end{vmatrix} = \begin{vmatrix} 4 & 8 \\ 20 & 14 \end{vmatrix}$$

$m_2 = (A_3 + A_4) * B_1$

$$\begin{vmatrix} 3 & 1 \\ 7 & 1 \end{vmatrix} * \begin{vmatrix} 0 & 1 \\ 2 & 1 \end{vmatrix} = \begin{vmatrix} 2 & 4 \\ 2 & 8 \end{vmatrix}$$

$m_3 = A_1*(B_2-B_4)$

$$\begin{vmatrix} 1 & 0 \\ 4 & 1 \end{vmatrix} * \begin{vmatrix} -1 & 0 \\ -5 & 4 \end{vmatrix} = \begin{vmatrix} -1 & 0 \\ -9 & 4 \end{vmatrix}$$

$m_4 = A_4*(B_3-B_1)$

$$\begin{vmatrix} 3 & 0 \\ 2 & 1 \end{vmatrix} * \begin{vmatrix} 2 & -1 \\ -1 & 2 \end{vmatrix} = \begin{vmatrix} 6 & -3 \\ 3 & 0 \end{vmatrix}$$

$m_5 = (A_1 + A_2) * B_4$

$$\begin{vmatrix} 3 & 1 \\ 5 & 1 \end{vmatrix} * \begin{vmatrix} 1 & 1 \\ 5 & 0 \end{vmatrix} = \begin{vmatrix} 8 & 3 \\ 10 & 5 \end{vmatrix}$$

$m_6 = (A_3 - A_1) * (B_1 + B_2)$

$$\begin{vmatrix} -1 & 1 \\ 1 & -1 \end{vmatrix} * \begin{vmatrix} 0 & 2 \\ 2 & 5 \end{vmatrix} = \begin{vmatrix} 2 & 3 \\ -2 & -3 \end{vmatrix}$$

$m_7 = (A_2 - A_4) * (B_3 + B_4)$

$$\begin{vmatrix} -1 & 1 \\ -1 & -1 \end{vmatrix} * \begin{vmatrix} 3 & 1 \\ 6 & 3 \end{vmatrix} = \begin{vmatrix} 3 & 2 \\ -9 & -4 \end{vmatrix}$$

$C1 = m_1 + m_4 - m_5 + m_7$

$$\begin{vmatrix} 4 & 8 \\ 20 & 14 \end{vmatrix} + \begin{vmatrix} 6 & -3 \\ 3 & 0 \end{vmatrix} - \begin{vmatrix} 8 & 3 \\ 10 & 5 \end{vmatrix} + \begin{vmatrix} 3 & 2 \\ -9 & -4 \end{vmatrix} = \begin{vmatrix} 5 & 4 \\ 4 & 5 \end{vmatrix}$$

$C2 = m_3 + m_5$

$$\begin{vmatrix} -1 & 0 \\ -9 & 4 \end{vmatrix} * \begin{vmatrix} 8 & 3 \\ 10 & 5 \end{vmatrix} = \begin{vmatrix} 7 & 3 \\ 1 & 9 \end{vmatrix}$$

$C3 = m_2 + m_4$

$$\begin{vmatrix} 2 & 4 \\ 2 & 8 \end{vmatrix} * \begin{vmatrix} 6 & -3 \\ 3 & 0 \end{vmatrix} = \begin{vmatrix} 8 & 1 \\ 5 & 8 \end{vmatrix}$$

$C4 = m_1 + m_3 - m_2 + m_6$

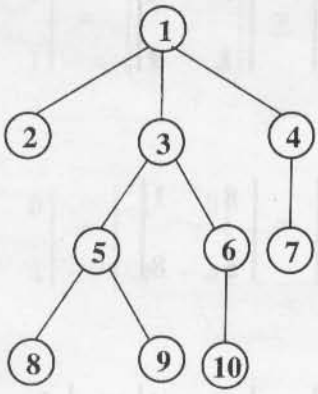
$$\begin{vmatrix} 4 & 8 \\ 20 & 14 \end{vmatrix} + \begin{vmatrix} -1 & 0 \\ -9 & 4 \end{vmatrix} - \begin{vmatrix} 2 & 4 \\ 2 & 0 \end{vmatrix} + \begin{vmatrix} 2 & 3 \\ -2 & -3 \end{vmatrix} = \begin{vmatrix} 3 & 7 \\ 7 & 7 \end{vmatrix}$$

$$A \times B = \begin{vmatrix} C1 & C2 \\ C3 & C4 \end{vmatrix} = \begin{vmatrix} \overset{C1}{\begin{matrix} 5 & 4 \\ 4 & 5 \end{matrix}} & \overset{C2}{\begin{matrix} 7 & 3 \\ 1 & 9 \end{matrix}} \\ \underset{C3}{\begin{matrix} 8 & 1 \\ 5 & 8 \end{matrix}} & \underset{C4}{\begin{matrix} 3 & 7 \\ 7 & 7 \end{matrix}} \end{vmatrix}$$

Exercises:

1. Describe binary search algorithm. Show that the worst case efficiency is $\theta(\log n)$. **(July/Aug 2004)**
2. What is divide and conquer method? Explain divide and conquer method along with the control abstraction of this method
3. Write an algorithm to find the sum of n elements stored in the vector A and find the time complexity
4. State the merge sort algorithm and analyze its complexity **(Jan/Feb 2005)**
5. Write the merge sort algorithm and find the time complexity using Master's theorem

6. Write algorithms for preorder, postorder and inorder traversals of a tree. List the tree shown below using preorder, inorder and postorder **(July/Aug 2004)**



7. Write the merge sort algorithm and find the time complexity using substitution method
8. Trace the merge sort algorithm for the data: 60, 50, 25, 10, 35, 25, 75, 30
9. Explain quick sort and write the algorithm to sort N elements using quick sort
10. Find the time complexity of quick sort for best case, worst case and average case
11. Give an example of an array of *n* elements for which the sentinel is necessary. What should be the value of sentinel? Also explain why a single sentinel is sufficient for any input.
12. Apply quick sort to sort the list EXAMPLE in alphabetical ordering. Draw the trees of recursive calls made.
    a. Write the best case input for the quick sort
    b. Find the best case time efficiency of quick sort
    c. Are arrays made up of all equal elements the worst-case input, the best-case input, or neither?
13. Obtain the average time complexity of binary search
14. Write functions to find the height of the tree, number of nodes in the tree and the number of leaves in a tree
15. Suggest an algorithm based on divide and conquer methodology to multiply two large integers and analyze its performance **(Jan/Feb 2005)**
16. Explain matrix multiplication with respect to divide and conquer technique
17. What is Strassen's matrix multiplication? Briefly explain and obtain its time complexity
18. Outline an algorithm for multiplication of matrices using Strassen's method. Evaluate it's efficiency

Chapter 5: Decrease and Conquer

What are we studying in this chapter?

- Concept of Decrease and Conquer Technique
  - Decrease by constant
  - Decrease by constant factor
  - Variable size decrease
- Insertion sort
  - Design
  - Best case analysis
  - Worst case analysis
  - Average case analysis
- Graph Traversals
  - Depth-First Search (DFS)
  - Analysis of DFS
  - Breadth-First Search (BFS)
  - Analysis of BFS
  - Applications of graph traversals
- Topological Sorting
  - Using DFS algorithm
  - Source removal algorithm
- Algorithms for Generating Combinatorial Objects
  - Generating permutations using bottom up *minimal-change* technique
  - Generating permutations using Johnson Trotter algorithm
  - Generating permutations using lexicographic order
  - Generating subsets

- 6 hours

NEW HORIZON COLLEGE OF ENGINEERING
LIBRARY
Acc. No. 30100223
Call. No.
Bangalore - 560 087.

5.1 Introduction

Let us see "What is decrease and conquer? or What is the concept of decrease and conquer methodology?"

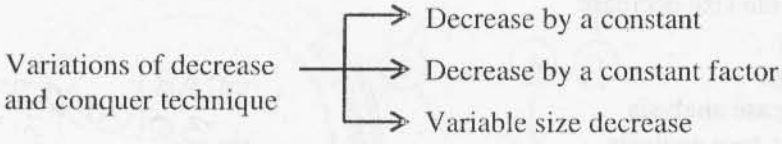
Definition: The decrease-and-conquer is a method of solving a problem by:

- Changing the problem size from *n* to smaller size of n-1, n/2 etc. In other words, change the problem from larger instance into smaller instance.
- Conquer (or solve) the problem of smaller size
- Using the solutions of smaller size problem obtain a solution for larger size problem.

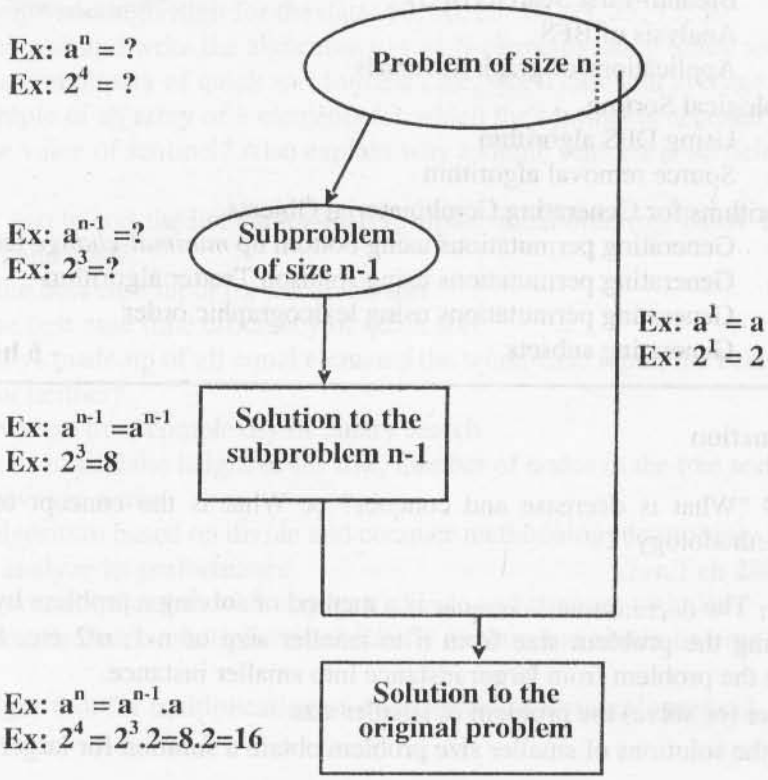
5.2 Decrease and Conquer

Using *decrease-and-conquer* technique we can solve a given problem using *top-down* technique (usually implemented using recursion) or *bottom-up* technique (usually iterative procedure/without using recursion). The *decrease-and-conquer* technique is slight variation of divide and conquer method.

Now, let us see "What are the 3 major variations of decrease and conquer method?" The three variations of decease and conquer method are shown below:



5.1.1 Decrease by a constant: Now, let us see "What is decrease by a constant?" The decrease by a constant is one of the variations of *decrease-and-conquer* technique. Here, the problem size is usually decremented by one in each iteration of the loop. The decrease by a constant is illustrated in following figure:



Examples: computing a^n, Insertion sort algorithm, traversing the graph using BFS and DFS method, topological sorting, generating permutations etc.

For example, Recursively a^n can be computed as shown below:

$$\mathbf{a^n = a^0 = 1} \quad \text{i.e.,} \quad a^n = 1 \quad \text{if } n = 0$$
$$\mathbf{a^n = a^1 = a} \quad \text{i.e.,} \quad a^n = a \quad \text{if } n = 1$$
$$\mathbf{a^n = a^{n-1} * a} \quad \text{i.e.,} \quad a^n = a^{n-1} * a \quad \text{otherwise}$$

$$\Downarrow$$

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a & \text{if } n = 1 \\ a^{n-1} * a & \text{otherwise} \end{cases}$$

In the above relation, note that the larger instance of size n is expressed in terms of smaller instance of size n-1. The above relation can also be written as shown below:

$$f(a, n) = \begin{cases} 1 & \text{if } n = 0 & \text{// Base case 1} \\ a & \text{if } n = 1 & \text{// Base case 2} \\ f(a, n-1) * a & \text{otherwise} & \text{// General case} \end{cases}$$

Now, let us "Design the algorithm to compute a^n using decrease by a constant" The algorithm can be written by replacing identifier f by the identifier *power* for the readability purpose and can be written as shown below:

Algorithm : power(a, n)
//Purpose : To compute a^n
//Input : **a** and **n** are positive inputs used to compute a^n
//Output : a^n is returned

Step1 : [Base case]
if (n = 0) **return** 1 // Return the result with respect to first base case

if (n = 1) **return a** // Return the result with respect to second base case

Step 2: [Recursively compute a^n]
return power(a, n-1) * a // Find the power recursively

Note: In the above algorithm, we start from n and we keep postponing the solution by expressing in terms of itself by decrementing the value of n by one recursively till the terminal condition is reached. Once the base condition is reached, we work backwards and compute a^n. This approach is called top down technique. The above algorithm can be solved using bottom up technique as shown below:

Algorithm : power(a, n)
//Purpose : To compute a^n
//Input : **a** and **n** are inputs used to compute a^n
//Output : a^n is returned

Step 1: [Initialization]
product ← 1

Step 2: [Multiply repeatedly to find the power]
for i ← 1 **to** n **do**
product ← product * a
end for

Step 3: [Return a^n]
return product

// end of the algorithm

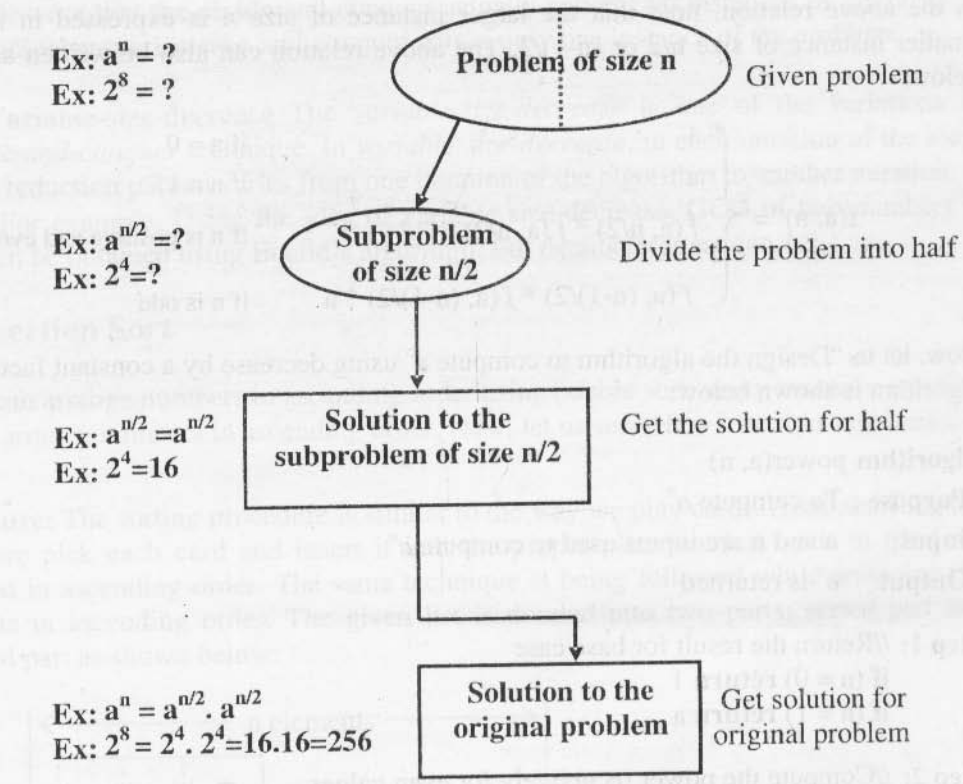
Note: Computing factorial of a number discussed in section 2.8.1 is another example of *decrease and conquer* technique using top down approach. Using bottom up approach, factorial of a number can be obtained using the following pseudocode.

product ← 1

for i ← 1 **to** n **do**
product ← product * i
end for

return product

5.1.2 Decrease by a constant factor: Now, let us see "What is decrease by a constant factor?" The decrease by constant factor is one of the variations of *decrease-and-conquer* technique. Here, the problem size is reduced by same constant factor (usually by 2) on each iteration of the algorithm. The idea of decrease by constant factor is illustrated in following figure:



Now, let us see "How can we compute a^n using decrease by a constant factor?" We know that:

$a^n = 1$	This is true if n is 0.	Ex: $3^0 = 1$
$a^n = a$	This is true if n is 1.	Ex: $3^1 = 3$
$a^n = a^{n/2}.a^{n/2} = \left(a^{n/2}\right)^2$	This is true if n is even	Ex: $3^8 = 3^4.3^4 = \left(3^4\right)^2$
$a^n = a^{(n-1)/2}.a^{(n-1)/2}.a = \left(a^{(n-1)/2}\right)^2.a$	This is true if n is odd	Ex: $3^9 = 3^4.3^4.3 = \left(3^4\right)^2.3$

These relations can be expressed as shown below:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a & \text{if } n = 1 \\ \left(a^{n/2}\right)^2 & \text{if n is positive and even} \\ \left(a^{(n-1)/2}\right)^2 * a & \text{if n is odd} \end{cases}$$

In the above relation, note that the larger instance of size n is expressed in terms of smaller instance of size n/2 or (n-1)/2. The above relation can also be written as shown below:

$$f(a, n) = \begin{cases} 1 & \text{if } n = 0 \\ a & \text{if } n = 1 \\ f(a, n/2) * f(a, n/2) & \text{if n is positive and even} \\ f(a, (n-1)/2) * f(a, (n-1)/2) * a & \text{if n is odd} \end{cases}$$

Now, let us "Design the algorithm to compute a^n using decrease by a constant factor" The algorithm is shown below:

Algorithm power(a, n)
//Purpose : To compute a^n
//Input: **a** and **n** are inputs used to compute a^n
//Output: a^n is returned

Step 1: //Return the result for base case
if (n = 0) **return** 1
if (n = 1) **return** a

Step 2: //Compute the power recursively for even values
if (n > 1 **and** n % 2 = 0) **return** power(a, n/2) * power(a, n/2)

Step 3: //Compute the power recursively for odd values
return power(a, (n-1)/2) * power(a, (n-1)/2) * a

// End of algorithm power

a^n can be calculated as shown below:

- Solve recursively using top down approach:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a & \text{if } n = 1 \\ \left(a^{n/2}\right)^2 & \text{if even value of } n > 1 \\ \left(a^{(n-1)/2}\right)^2 * a & \text{if odd value of } n > 1 \end{cases}$$

Note: Requires time efficiency of O(log n)

Note: Observe that the divide-and-conquer actually solves two instances of the problem of size n/2 whereas decrease-and-conquer solves only one instance of the problem.

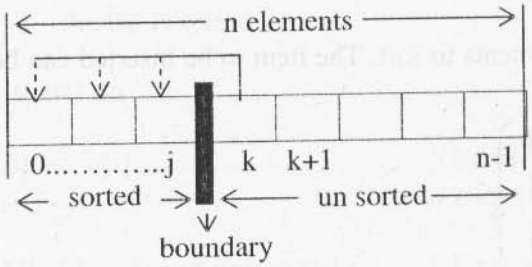
5.1.3 Variable-size-decrease The *variable-size-decrease* is one of the variations of *decrease-and-conquer* technique. In *variable-size-decrease*, in each iteration of the loop, the size reduction pattern varies from one iteration of the algorithm to another iteration.

For example, Using the idea of variable-size-decrease, GCD of two numbers *m* and *n* can be obtained using Euclid's algorithm. For details refer section 1.2.1,

5.2 Insertion Sort

As we can arrange numbers in ascending order using bubble sort, using insertion sort also we can arrange numbers in ascending order. Now, let us see "How insertion sort works?"

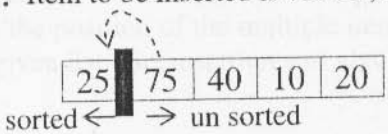
Procedure: The sorting procedure is similar to the way we play cards. After shuffling the cards, we pick each card and insert it into the proper place so that cards in hand are arranged in ascending order. The same technique is being followed while arranging the elements in ascending order. The given list is divided into two parts: sorted part and unsorted part as shown below:



Note that all the elements from 0 to j are sorted and elements from k to n-1 are not sorted. The kth item can be inserted into any of the positions from 0 to j so that elements towards left of boundary are sorted. As each item is inserted towards the sorted left part, the boundary moves to the right decreasing the unsorted list. Finally, once the boundary moves to the right most position, the elements towards the left of boundary represent the sorted list.

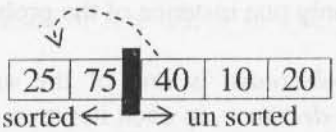
Example : Sort the elements 25, 75, 40, 10, 20 using insertion sort

Step 1: Item to be inserted is 75. i.e., item = a[1]



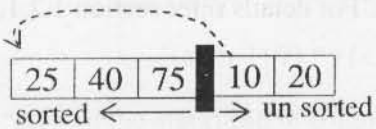
75 is inserted after 25

Step 2: Item to be inserted is 40 i.e., item = a[2]



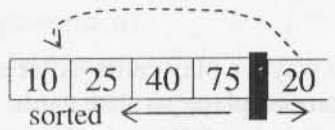
40 is inserted between 25 and 75

Step 3: Item to be inserted is 10 i.e., item = a[3]



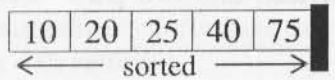
10 is inserted before 25

Step 4: Item to be inserted is 20 i.e., item = a[4]



20 is inserted between 10 and 25

Output



Final sorted list

Design: Consider an array of ***n*** elements to sort. The item to be inserted can be accessed as shown below:

in step 1: item = a[**1**]
in step 2: item = a[**2**]
in step 3: item = a[**3**]
in step 4: item = a[**4**]

i.e., item = a[i] where i = 1 to 4
i.e., i = 1 to 5-1
i.e., i = 1 to n -1

So, in general **item = a[i] for i = 1 to n-1**

Now, the **item** has to compared with **a[j]** as long as **item < a[j]** and j >= 0 with initial value of j = i-1. As long as the above condition is true perform the following activities:

- copy a[j] to a[j+1]
- decrement j by 1.

The equivalent statements can be written as shown below:

```
j = i – 1;
while (item < a[j] && j >= 0)
        a[j+1] = a[j];
        j = j – 1;
end while
```

These statement should be executed for each item = a[i], where i = 1 to n-1

Once control comes out of the above loop, insert the *item* into a[j+1] using the statement:

```
a[j+1] = item
```

The complete algorithm can be written as shown below:

```
Algorithm Insertion Sort(a,n)

//Purpose: Sort the list in ascending order

//Input:    a – the list to be sorted
//          n – the total number of elements in the list to be sorted

//Output:
//          a – the list is sorted

for i ← 1 to n-1 do
    Item ← a[i]                          // Insert the item from unsorted part
    j ← i – 1                            // Initial position of sorted part

    while ( j >= 0 and item < a[j] )     // Find the appropriate place to insert
        a[j+1] ← a[j]
        j ← j – 1
    end while

    a[j+1] ← item                        // Insert at the appropriate place

end for
```

Now the question is "How we say that insertion sort algorithm is stable?" Assume that the given list contains some multiple items with the same value. After sorting, even though the position of the multiple items changes, they appear in the order as they appear in the given list. So, insertion sort algorithm is stable.

Now, the question is "Is insertion sort algorithm is in place?" In the insertion sort algorithm, other than the input array ***a***, no other array or no other extra space is used during sorting. So, insertion sort algorithm is in place.

5.2.1 Analysis (Best case time efficiency)

The best case occurs when the items in the list are partially or nearly sorted. Whenever for loop is executed, the conditional expression "item < a[j]" is executed. Since the loop is executed (n-1) times, the conditional expression "item < a[j]" is also executed (n-1) times. So, the time complexity is given by

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

So, time complexity of insertion sort in the best case is $\Omega(n)$

Note: When the array is already sorted or when the array is almost sorted, insertion sort algorithm is very efficient. The performance of quick sort can be further improved by partially sorting the array and then use the insertion sort.

5.2.2 Analysis (Worst case time efficiency)

The worst case occurs when the conditional expression "item < a[j]" is executed maximum number of times. This situation occurs when the elements of the list are sorted in descending order. The time complexity can be obtained using the following relation:

for i ← 1 **to** n-1 **do**
  Item ← a[i]
  j ← i – 1
  while (j >= 0 **and** item < a[j])

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i-1) - 0 + 1 = \sum_{i=1}^{n-1} i$$

$$= \sum_{i=1}^{n-1} i$$

$$f(n) = 1+2+3+.....(n-2)+(n-1)$$

$$f(n) = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

The constraint to be satisfied is

$$f(n) \le c^* g(n) \text{ for } n \ge n_0 \text{ where } c = 1, n_0 = 0, g(n) = n^2.$$

$$\frac{n^2}{2} - \frac{n}{2} \le 1 * n^2 \quad \text{for } n \ge 0 \text{ where } c = 1, n_0 = 0\ g(n) = n^2. \text{ So, by definition,}$$

$$\mathbf{f(n) \in O(g(n)) \in O(n^2)}$$

So, time complexity of insertion sort in the worst case is $O(n^2)$

5.2.3 Analysis (Average case time efficiency)

Let us assume that two elements are already sorted and we have to insert the 3rd item at the appropriate position. There are 3 possible places where the item can be inserted. Let us take all possible cases one by one.

Case 1: i = 2 (position where the item has to be inserted) and Item = a[2] = 13

A	10	12	13
	0	1	2
		j	i

The item 13 is compared with 12. Since 13 is greater than 12, control comes out of the while loop and the condition in while loop is executed only once.
So, the total number of times the while loop is executed = 1

Case 2: i = 2 (position where the item has to be inserted) and Item = a[2] = 11

A	10	12	11
	0	1	2
		j	i

Note that the item 11 has to be compared with 12 as well as 10. The item should be inserted between 10 and 12 which results in the while loop to be executed 2 times. So, the total number of times the while loop is executed = 2.

Case 3: i = 2 (position where the item has to be inserted) and Item = a[2] = 9

A	10	12	9
	0	1	2
		j	i

In this case, item 9 is compared with 12 as well as 10 and should inserted before 10 which results in while loop to be executed 3 times. So, the total number of times the while loop is executed = 3.

Note that all these three cases have the same probability and the average number of times the while loop is executed is given by

$$\left(\frac{1+2+3}{3}\right)$$

This result is true if we are inserting the 3rd element in to the array. In general, to insert an item X with index *i*, in the correct position, the total number of times the while loop is executed is given by

$$\frac{1+2+3+.....i}{i} = \frac{i(i+1)}{2*i} = \frac{i+1}{2}$$

It is clear from the algorithm that the index variable *i* start from 1 to n-1. So, the average number of times the while loop is executed is given by the following relation:

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

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

$$= \frac{n^2}{4} - \frac{n}{4} + \frac{n}{2} - 1$$

$$= \frac{n^2}{4} - \frac{n}{4} - 1$$

$$\approx \frac{n^2}{4} \text{ for very large value of n}$$

So, time complexity of insertion sort in the average case is $\theta(n^2)$

Advantages

- Very simple to implement
- Efficient on smaller input size
- Efficient when the elements are substantially sorted
- The algorithm is more stable i.e., the relative ordering of items with same value remains same even after sorting
- The algorithm is in-place i.e., no extra memory is required

Disadvantages

- It is not as efficient as quick sort, heap sort or merge sort
- Not suitable for more random elements

5. 3 Graph Traversals

In the graph G = (V, E), V is set of vertices and E is set of edges. |V| gives the number of vertices and |E| gives the number of edges. For details refer section 1.6. Now, we concentrate on a very important topic namely graph traversal techniques. Let us see "What is graph traversal? What are the different ways of traversing the graph?"

Definition: A graph traversal means visiting the nodes of a graph one after the other in a systematic manner. Many graph algorithms require processing vertices or edges of a graph in a systematic manner one after the other. In graphs, we do not have any special vertex designated as *source* vertex. So, traversal can start from any arbitrary vertex. The two important graph traversal techniques are:

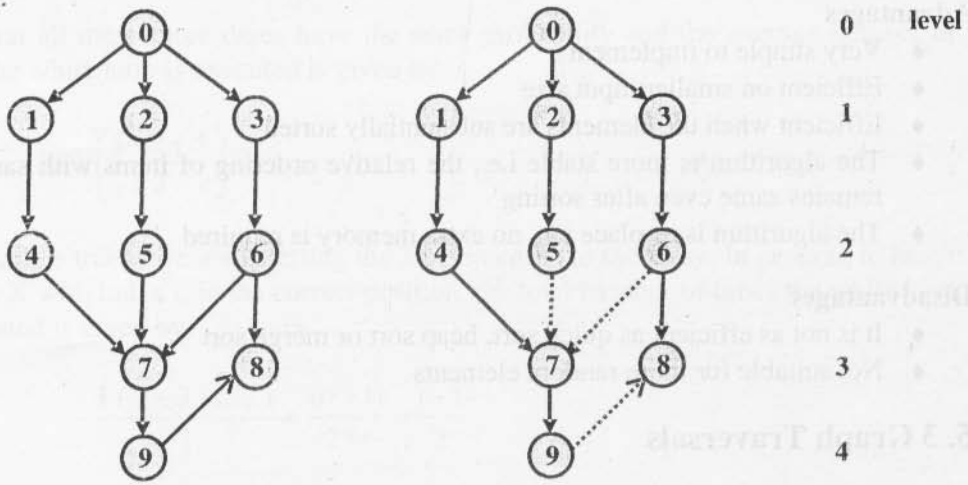
Graph traversals → **B**readth **F**irst **S**earch (**BFS**)
Graph traversals → **D**epth **F**irst **S**earch (**DFS**)

5.3.1 Breadth First Search (BFS)

The graph can be traversed in BFS. Now, let us see "What is breadth first search (BFS)?"

Definition: The breadth first search is a method of traversing the graph by visiting each node of the graph in a systematic order. Assume u is the start vertex. This vertex is considered to be at level 0. BFS is a method of traversing the graph in the order of the level of a vertex. In the first stage, we visit all the vertices at distance 1 (i.e., vertices at a level 1 from u). In the second stage, we visit all the vertices at distance 2 (i.e., vertices at level 2 from u) and so on. At each level the vertices are visited from left to right (in increasing order of the vertex value). In general, BFS discovers all vertices at a distance k from given start vertex before discovering any vertices at distance k+1. The search will terminate when all the vertices have been visited.

For example, the following figures show the graph and its equivalent BFS traversal.



(a) Graph (b) BFS traversal

Figure 5.3.1 Graph and its BFS traversal

Note: The search continues horizontally or breadth wise level by level (as shown in above figure) thus exploring all the nodes at a distance k and hence the name BFS. In BFS, a vertex is fully explored before the exploration of any other vertex begins.

Now, let us see "What are the different types of edges that are encountered during BFS traversal?" During BFS traversal, we encounter following types of edges:

BFS traversal edges → Tree edge
→ Cross edge

Now, let us see "What is a tree edge? What is a cross edge?"

Definition: During traversal, when a new unvisited vertex say v is reached for the first time from a current vertex say u, then the edge (u, v) is called a tree edge. In the tree edge (u, v) the vertex *u* is the parent and vertex *v* is the child. The tree edges are represented using solid lines. A parent vertex may have several children.

For example, the edges with solid lines of figure 5.3.1.b, are all tree edges

Definition: During traversal, when an already visited vertex say *v* is reached from the current vertex *u* and if *v* is not the immediate predecessor of *u*, then the edge (u, v) is called a cross edge. The cross edges are represented using dotted lines. The cross edges connect the vertices that are already visited but to either siblings or cousins on the same or adjacent levels.

For example, the edges with dotted lines of figure 5.3.1.b, are all cross edges

Now, the question is "What is the data structure that is used to traverse the graph in BFS?" The queue which provides first in first out property is very useful and convenient while traversing the graph in BFS. When a vertex is reached for the first time, it is inserted into rear end of queue. When we get a dead end (i.e., the vertex is already explored), we delete a vertex from the front of queue. Thus, the BFS yields only one ordering of vertices i.e., "the order in which the vertices are inserted into queue is same as the order in which the vertices are deleted from queue".

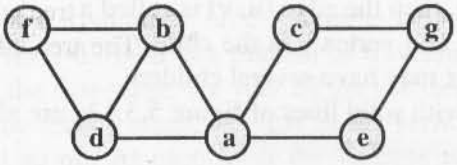
Design methodology Since the vertices are visited from level 0, level 1 and so on one level at a time, we can implement BFS using *queue* data structure. To begin with, insert the source vertex (**Note**: Any arbitrary node can be considered as source) into the queue. The general procedure to traverse the graph in BFS is shown below:

Step 1: Initialize queue with start vertex and mark this vertex as visited

Step 2: while queue is not empty

```
        Delete a vertex u from queue
        Identify all the vertices v adjacent to u
        If the vertices adjacent to u are not visited
                mark them as visited
                Insert all the marked vertices into queue
                Output u, v
        end if
end while
```

Example 1: Traverse the following graph by breadth-first search and construct the corresponding BFS tree. Start the traversal at vertex *a* and resolve ties by the vertex alphabetical order. Show the tree edges and cross edges in the BFS traversal.



Solution: It is given that source vertex is *a*. Perform the following activities:

Initial step: Insert source vertex *a* into Q and add *a* to S as shown in table

Stage 1: The various activities that are performed are shown below:

Step 1: Delete an element *a* from queue

Step 2: The vertices *b, c, d* and *e* are adjacent to *a*

Step 3: Since *b, c, d* and *e* are not visited earlier, they are added to S, inserted into Q and output is:

a – b
a – c
a – d
a – e (Look at the table for details)

Stage 2: The various activities that are performed are shown below:

Step 1: Delete *b* from Q

Step 2: Vertices *a, d* and *f* are adjacent to *b*

Step 3: Since *a* and *d* are already in S, we take only *f* as adjacent, add *f* to S, insert *f* to Q and output is: **b – f**

Stage 3: The various activities that are performed are shown below:

Step 1: Delete *c* from Q

Step 2: Vertices *a* and *g* are adjacent to *c*

Step 3: Since *a* is already in S, we take only g as adjacent, add *g* to S, insert *g* to Q and output is: **c – g**

Stage 4: The various activities that are performed are shown below:

Step 1: Delete *d* from Q

Step 2: Vertices *a, b* and *f* are adjacent to *c*

Step 3: Since *a, b* and *f* str already in S, no vertex is adjacent and hence no output

The remaining stages are shown in the following table:

	Step 1	Step 2	Step 3		
	u = del(Q)	v = adj. to u	Nodes visited S	Q	Output T(u, v)
Initial Step	-	-	a	a	-
Stage 1	a	b, c, d, e	a, b, c, d, e	b, c, d, e	a – b a – c a – d a – e
Stage 2	b	f	a, b, c, d, e, f	c, d, e, f	b – f
Stage 3	c	g	a, b, c, d, e, f, g	d, e, f, g	c – g
Stage 4	d	-	a, b, c, d, e, f, g	e, f, g	-
Stage 5	e	-	a, b, c, d, e, f, g	f, g	-
Stage 6	f	-	a, b, c, d, e, f, g	g	-
Stage 7	g	-	a, b, c, d, e, f, g	empty	-

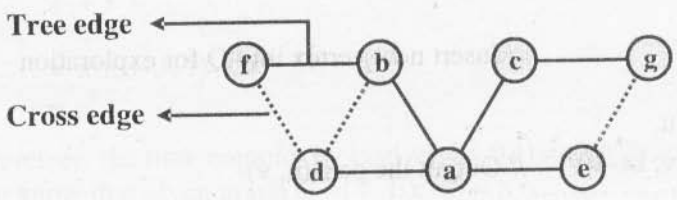
Ordering

BFS traversal

a	b	c	d	e	f	g
1	2	3	4	5	6	7

Queue data structure is used. So, the order in which the items are inserted is same as the order in which the items are deleted. So, BFS requires only one ordering. So, numbering is given in sequence.

The BFS traversal is obtained by looking at the *output* column in the table as shown below:



Note: In the above figure, the solid lines represent tree edges and dotted lines represent the cross edges.

The complete algorithm is shown below:

```
Algorithm BFS(a, n, source, T)

//Purpose: Traverse the graph from the given source node in BFS
//Input: a – adjacency matrix of the given graph
//       n – the number of nodes in the graph
//       source – from where the traversal is initiated

//Output:
//       (u, v) – the nodes v reachable from u are stored in a vector T

for i ← 0 to n-1 do
        s[i] ← 0                          // No node is visited
end for

f ← r ← 0
q[r] ← source                             // Insert source vertex into queue

s[source] ← 1                             // Add source to S (mark source as visited)
k ← 0                                     // Index to store the tree edges
while (f <= r)                            // As long as queue is not empty

        u ← q[f]
        f ← f + 1                         // Delete the next vertex to be explored

        for every v adjacent to u do      // Find the nodes v which are adjacent to u
                if v is not visited
                        s[v] ← 1          // add v to s indicates that v is visited now

                        r ← r + 1
                        q[r] ← v          // Insert new vertex into Q for exploration

                        T[k,1] ← u
                        T[k,2] ← v, k←k+1 // Output the pair (u, v)
                end if
        end for
end while
```

Note: The analysis of BFS depends on the number of vertices and whether the *adjacency matrix* or *adjacency list* representation of a graph is used. So, we find the time complexity for the graph represented using *adjacency matrix* and *adjacency list*.

Analysis of BFS represented using adjacency matrix

The time efficiency of BFS traversal can be obtained as shown below:

Step 1: The parameter to be considered is n, which represent number of vertices in the graph.

Step 2: The basic operation is the "if statement" with in the for loop.

Step 3: The number of times inner most for loop is executed depends on the outer while loop. So, the total number of times the basic operation is executed can be obtained as shown below:

.....

.....

while f <= r (i.e., f = 0 to n-1 where r = n-1 when all vertices are inserted to que)

.....

for every v adjacent to u (i.e, for v = 0 to n-1 do)

if (v is not visited)

$$f(n) = \sum_{f=0}^{n-1} \sum_{v=0}^{n-1} 1$$

$$= \sum_{f=0}^{n-1} n-1-0+1 \text{ (Note: upper bound – lower bound + 1)}$$

$$= \sum_{f=0}^{n-1} n$$

$$= n\sum_{f=0}^{n-1} 1$$

$$= n(n-1-0+1)$$

$$= n^2$$

Therefore, the time complexity is given by $f(n) = O(n^2)$ .....**(1)**

We know that given graph G =(V, E), |V| = n .....**(2)**

Substituting eq.(2) in equation (1) we have:

$$\mathbf{f(n) \in \theta(n^2) = \theta(|V|^2)}$$

So, time complexity of BFS traversal is $\mathbf{f(n) \in \theta(n^2) = \theta(|V|^2)}$

Analysis of BFS represented using adjacency list

Let G = (V, E) be a graph where

- V is set of vertices in the graph.
- E is set of edges in the graph

The adjacency linked lists is an array of linked lists. All the vertices 0, 1, 2,.....n-1 represent the vertices and they become the index into an adjacency linked. So, if *n* represent the number of vertices in the graph, and |V| also represent the number of vertices in the graph, we have n = |V| .....**(1)**

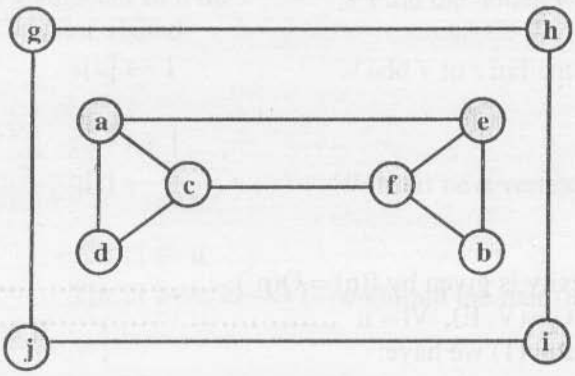
If *i* is a vertex, then a[i] contains the address of the first node of the ith linked list. The number of nodes present in that linked list represent the number of edges adjacent to vertex *i*. Let *m* represent the number of edges in the graph. The number of edges *m* in the graph is nothing but the sum of nodes present in all the linked lists which is given by |E|. So, m = |E| .....**(2)**

In BFS, each vertex is visited and each edge is taken from the list. So, the time efficiency of accessing each vertex and time efficiency to access the edge is obtained by adding equation (1) and (2). So, time efficiency is given by f(n) = n + m

i.e., f(n) = |V| + |E|

The total time complexity of BFS is given by $f(n) \in O(\|V\|+\|E\|)$

Example 2: Traverse the following graph by breadth-first search and construct the corresponding BFS tree. Start the traversal at vertex *a* and resolve ties by the vertex alphabetical order. Show the tree edges and cross edges in the BFS traversal.



Solution: Insert source vertex *a* into Q and add *a* to S as shown in table. The activities during BFS traversal are shown in the table below:

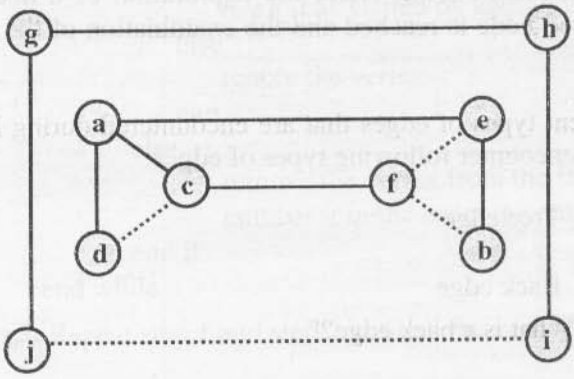
	Step 1	Step 2	Step 3		
	u = del(Q)	v = adj. to u	Nodes visited S	Q	Output T(u, v)
Initial Step	–	–	a	a	–
Stage 1	a	c, d, e	a, c, d, e	c, d, e	a – c a – d a – e
Stage 2	c	f	a, c, d, e, f	d, e, f	c – f
Stage 3	d	-	a, c, d, e, f	e, f	–
Stage 4	e	b	a, b, c, d, e, f	f, b	e – b
Stage 5	f	-	a, b, c, d, e, f	b	-
Stage 6	b	-	a, b, c, d, e, f	-	-

Note: Now, queue is empty. But, there are some vertices that are not visited. So, for the next subgraph, take the next vertex and repeat the above procedure

Initial Step	-	-	a, b, c, d, e, f, g	g	-
Stage 7	g	h, j	a,b,c,d,e,f,g,h,j	h, j	g – h g – j
Stage 8	h	i	a,b,c,d,e,f,g,h,i,j	j, i	h – i
Stage 9	j	-	a,b,c,d,e,f,g,h,i,j	i	-
Stage 10	i	-	a,b,c,d,e,f,g,h,i,j	-	-

Ordering **BFS traversal**

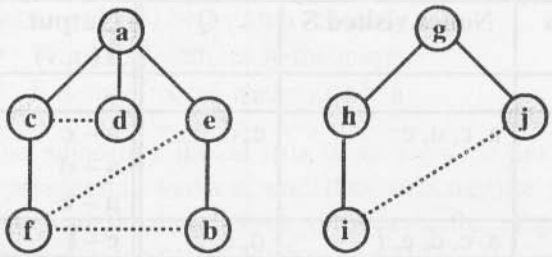
a	c	d	e	f	b	g	h	j	i
1	2	3	4	5	6	7	8	9	10



The BFS traversal is obtained b looking at the *output* column i the table

Note: In the BFS traversal, th solid lines represent tree edge and dotted lines represent th cross edges.

The above BFS traversal can be written with respect to levels as shown below:



Ordering

a	c	d	e	f	b	g	h	j	i
1	2	3	4	5	6	7	8	9	10

Now, let us see "What are the applications of BFS?" The applications of BFS are shown below:

- Used to check connectivity of the graph.
- Used to check whether the graph is acyclic or not
- To find the spanning tree
- Used to find the path with fewest number of edges.

5.3.2 Depth First Search (DFS)

The depth first search is a method of traversing the graph by visiting each node of the graph in a systematic order. As the name implies *depth-first-search* means "to search deeper in the graph". Now, let us see "What is depth first search (DFS)?"

Definition: In DFS, a vertex u is picked as source vertex and is visited. The vertex u at this point is said to be unexplored. The exploration of the vertex u is postponed and a vertex v adjacent to u is picked and is visited. Now, the search begins at the vertex v. There may be still some nodes which are adjacent to u but not visited. When the vertex v is completely examined, then only u is examined. The search will terminate when all the vertices have been examined.

Note: The search continues deeper and deeper in the graph until no vertex is adjacent or all the vertices are visited. Hence, the name DFS. Here, the exploration of a node is postponed as soon as a new unexplored node is reached and the examination of the new node begins immediately.

Now, let us see "What are the different types of edges that are encountered during DFS traversal?" During DFS traversal, we encounter following types of edges:

DFS traversal edges → Tree edge
DFS traversal edges → Back edge

Now, let us see "What is a tree edge? What is a back edge?"

Definition: During traversal, when a new unvisited vertex say v is reached for the first time from a current vertex say u, then the edge (u, v) is called a tree edge. In the tree edge (u, v) the vertex *u* is the parent and vertex *v* is the child. The tree edges are represented using solid lines. A parent vertex may have several children.

Definition: During traversal, when an already visited vertex say *v* is reached from the current vertex *u* and if *v* is not the immediate predecessor of *u*, then the edge (u, v) is called a back edge. The back edge is connected to ancestor other than the parent i.e., in the back edge (u, v) the vertex *u* is the ancestor (except parent) of *v*. The back edges are represented using dotted lines.

Now, the next question is "What is the data structure that is used to traverse the graph in DFS?" The stack which provides last in first out property is very useful and convenient while traversing the graph in DFS. When a vertex is reached for the first time, it is pushed on to the stack. When we get a dead end (i.e., the vertex is already explored), pop the vertex from the stack. Thus, the DFS yields two orderings of vertices:

- The order in which the vertices are reached for the first time. When a vertex is reached for the first time it is pushed onto the stack and each vertex is numbered in the order in which it is pushed onto the stack
- The order in which the vertices become dead ends. When a vertex is dead end (i.e., all adjacent vertices are explored), it is removed from the stack. Each node is numbered in the order in which it is deleted from the stack.

Design methodology The procedure to traverse the graph in DFS is shown below:

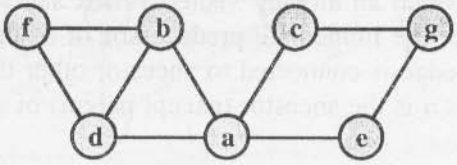
Step 1: Select node *u* as the start vertex (select in alphabetical order), push *u* onto stack and mark it as visited. We add *u* to S for marking

Step 2: While stack is not empty

```
        For vertex u on top of the stack, find the next immediate adjacent vertex.
        if v is adjacent
            If a vertex v not visited then
                push it on to stack and number it in the order it is pushed.
                mark it as visited by adding v to S
            else
                ignore the vertex
            end if
        else
                remove the vertex from the stack
                number it in the order it is popped.
        end if
    end while
```

Step 3: Repeat step 1 and step 2 until all the vertices in the graph are considered

Example 1: "Traverse the following graph using DFS and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices become dead ends (popped off the stack). Also mention, the tree edges and back edges of the graph"



Note: When a node is pushed, number it in sequence as the first subscript. When a node is removed from the stack, number it in sequence as the second subscript.

Solution: Since vertex *a* is the least in alphabetical order, it is selected as the start vertex.

Initial step: Push start vertex *a* onto the stack and number it as the first item to be inserted on the stack i.e., a_1 is placed on the stack. Also add it to S, indicating the vertex is marked or visited.

Stage 1: The various activities that are performed are shown below (Look stage 1 row in the table):

Step 1: Stack not empty. Top of stack is vertex *a*.
Step 2: Obtain the first node which is adjacent to vertex *a* i.e., vertex *b*.
Step 3: Add vertex *b* to S indicating it is visited or marked.
Step 4: Not performed since a vertex is added to S. This step is performed if nothing is added to S
Step 5: Output vertex on top of stack and node adjacent to it i.e., output the edge (a, b) and push vertex *b* on to stack. Since vertex *b* is second vertex being pushed on to the stack, it has prefix 2 on the stack.

Stage 2: The various activities that are performed are shown below:

Step 1: Stack not empty. Top of stack is vertex *b*.
Step 2: Obtain the first node which is adjacent to vertex *b* i.e., vertex *d*.
Step 3: Add vertex *d* to S indicating it is visited or marked.
Step 4: Not performed since a vertex is added to S. This step is performed if nothing is added to S
Step 5: Output vertex on top of stack and node adjacent to it i.e., output the edge (b, d) and push vertex *d* on to stack. Since vertex *d* is third vertex being pushed on to the stack, it has prefix 3 on the stack.

Stage 3: The various activities that are performed are shown below:

Step 1: Stack not empty. Top of stack is vertex *d*.

Step 2: Obtain the first node which is adjacent to vertex *d* i.e., vertex *f*.

Step 3: Add vertex *f* to S indicating it is visited or marked.

Step 4: Not performed since a vertex is added to S. This step is performed if nothing is added to S

Step 5: Output vertex on top of stack and node adjacent to it i.e., output the edge (d, f) and push vertex *f* on to stack. Since vertex *f* is fourth vertex being pushed on to the stack, it has prefix 4 on the stack.

Stage 4: The various activities that are performed are shown below:

Step 1: Stack not empty. Top of stack is vertex *f*.

Step 2: Obtain the first node which is adjacent to vertex *f*. No vertex found since all the nodes are already marked (i.e., they are added to S)

Step 3: Sine no vertex found in step 2, nothing is added. So, no output

Step 4: Delete an item from stack and mark it as the first node to be deleted. This number should be the second subscript i.e., $f_{4,1}$ **(Note:** This indicates that it the fourth item pushed and first item to be popped)

Step 5: No-output since no vertex found in step 2.

Note: Similarly proceed in this manner and fill the table as shown below:

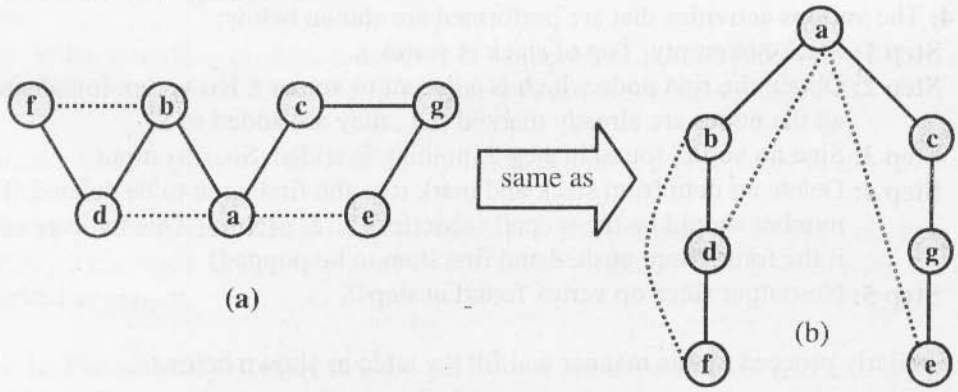
	Step1	Step 2	Step 3	Step 4	Step 5
	Stack	**v=adj(s[top])**	**Nodes visited S**	**pop(stack)**	**Output**
Initial step	a_1	-	a		
Stage 1	a_1	b	a, b	-	a – b
Stage 2	a_1, b_2	d	a, b, d	-	b – d
Stage 3	a_1, b_2, d_3	f	a, b, d, f	-	d – f
Stage 4	$\mathbf{a_1, b_2, d_3, f_4}$	-	a, b, d, f	$f_{4,1}$	-
Stage 5	a_1, b_2, d_3	-	a, b, d, f	$d_{3,2}$	-
Stage 6	a_1, b_2	-	a, b, d, f	$b_{2,3}$	-
Stage 7	a_1	c	a, b, c, d, f	-	a – c
Stage 8	a_1, c_5	g	a, b, c, d, f, g	-	c – g
Stage 9	a_1, c_5, g_6	e	a, b, c, d, f, g, e	-	g – e
Stage 10	$a_1, \mathbf{c_5, g_6, e_7}$	-	a, b, c, d, f, g, e	$e_{7,4}$	-
Stage 11	a_1, c_5, g_6	-	a, b, c, d, f, g, e	$g_{6,5}$	-
Stage 12	a_1, c_5	-	a, b, c, d, f, g, e	$c_{5,6}$	-
Stage 13	a_1	-	a, b, c, d, f, g, e	$a_{1,7}$	-

Ordering **DFS traversal**

Note: The traversal stack is obtained by looking at stack contents in Stage 4 and Stage10 along with column values of step 4 as shown below:

$f_{4,1}$	$e_{7,4}$	
$d_{3,2}$	$g_{6,5}$	**Traversal stack**
$b_{2,3}$	$c_{5,6}$	
$a_{1,7}$		

The DFS traversal of the graph is obtained by looking at the output column of step 5 in above table. Joining all the edges shown in last column of the above table, the DFS traversal is shown below:



DFS traversal

Note: The DFS traversal shown in figure (a) and (b) are same. The dotted lines represent the back edges and solid lines represent tree edges.

Design methodology It is clear from the above example that, the *stack* is the most suitable data structure to implement DFS. Whenever a vertex is visited for the first time, that vertex is pushed on to the stack and the vertex is deleted from the stack when a dead end is reached and the search resumes from the vertex that is deleted most recently. If there are no vertices adjacent to the most recently deleted vertex, the next node is deleted from the stack and the process is repeated till all the vertices are reached or till the stack is empty. The algorithm to implement DFS traversal is shown below:

Algorithm dfs(a, n, u, s, t)
//Purpose: Traverse the graph from the given node source in DFS

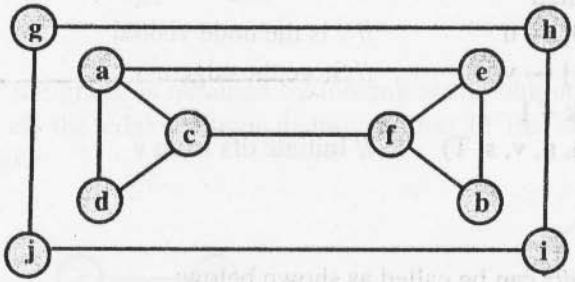
//Input: a – adjacency matrix of the given graph
// n – the number of nodes in the graph
// u – from where the traversal is to be initiated
// s – indicates the vertices that are visited and that are not visited

```
//Output:
//      (u, v) – the nodes v reachable from u are stored in a vector t
s [u] ←1                        // visit the node
for every v adjacent to vertex u
      if v is not visited
            t[k][0] ← u         // v is the node visited
            t[k][1] ← v         // Store the edge u-v
            k ← k + 1
            dfs (a, n, v, s, T)  // Initiate dfs from v
      end if
end for
```

Note: The function *dfs* can be called as shown below:

```
Step 1: [read the number of vertices in the graph]
      read n

Step 2: [Input the adjacency matrix]
      for i ← 0 to n-1 do
            for j ← 0 to n-1 do
                  read a[i][j]
            end for
      end for

Step 3: [Input the vertex from where DFS should start]
      read source

Step 4: [Indicate that no node is visited]
      for i ← 0 to n-1 do s[i] ← 0

      k ← 0

Step 5: [Initiate DFS]
       for each v ∈ V
            if (v is not visited)
                  dfs(a, n, v, s, T)
            end if
      end for
Step 6: [Output the DFS path]
      for i ← 0 to n-2 do
            write T[i][0], T[i][1]
      end for
```

Example 2: "Traverse the following graph using DFS and construct the corresponding depth-first search tree. Give the order in which the vertices were reached for the first time (pushed onto the traversal stack) and the order in which the vertices become dead ends (popped off the stack). Also mention, the tree edges and back edges of the graph"



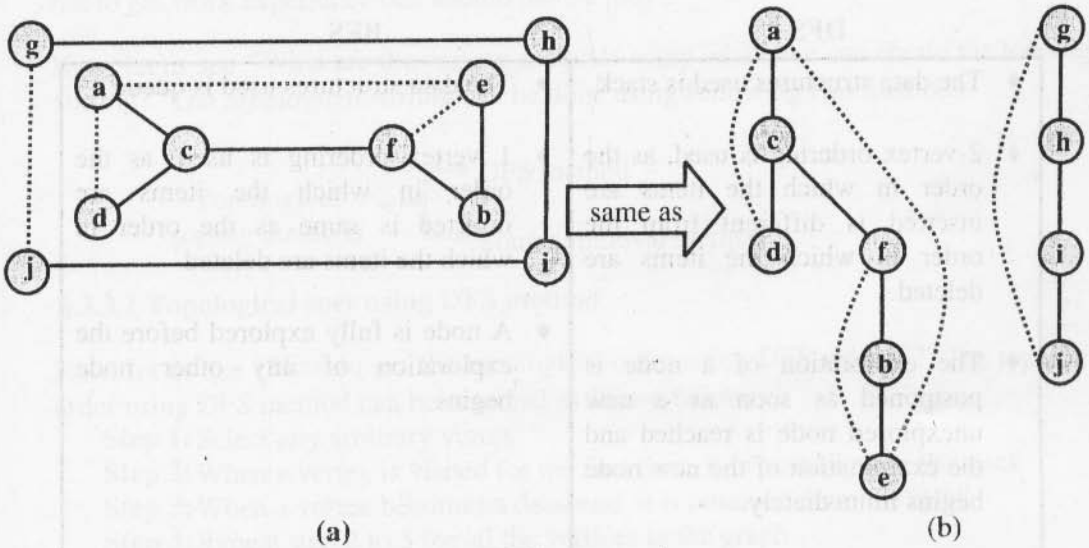
Solution: The solution is similar to the example 1 in this section. The solution is expressed using the table as shown below:

	Step1	Step 2	Step 3	Step 4	Step 5
	Stack	**v=adj(s[top])**	**Nodes visited S**	**pop(stack)**	**Output**
Initial step	a_1	-	a		
Stage 1	a_1	c	a, c	-	a – c
Stage 2	a_1, c_2	d	a, c, d	-	c – d
Stage 3	a_1, c_2, d_3	-	a, c, d	$d_{3,1}$	–
Stage 4	a_1, c_2	f	a, c, d, f	-	c – f
Stage 5	a_1, c_2, f_4	b	a, b, c, d, f	-	f – b
Stage 6	a_1, c_2, f_4, b_5	e	a, b, c, d, e, f	-	b – e
Stage 7	a_1, c_2, f_4, b_5, e_6	-	a, b, c, d, e, f	$e_{6,2}$	–
Stage 8	a_1, c_2, f_4, b_5	-	a, b, c, d, e, f	$b_{5,3}$	–
Stage 9	a_1, c_2, f_4	-	a, b, c, d, e, f	$f_{4,4}$	–
Stage 10	a_1, c_2	-	a, b, c, d, e, f	$c_{2,5}$	–
Stage 11	a_1	-	a, b, c, d, e, f	$a_{1,6}$	–
	Note: Stack empty. So, take the next vertex which is not marked				
Stage 13	g_7	h	a,b,c,d,e,f,g,h	-	g – h
Stage 14	g_7, h_8	i	a,b,c,d,e,f,g,h,i	-	h – i
Stage 15	g_7, h_8, i_9	j	a,b,c,d,e,f,g,h,i,j	-	i – j
Stage 16	g_7, h_8, i_9, j_{10}	-	a,b,c,d,e,f,g,h,i,j	$j_{10,7}$	–
Stage 17	g_7, h_8, i_9	-	a,b,c,d,e,f,g,h,i,j	$i_{9,8}$	–
Stage 18	g_7, h_8	-	a,b,c,d,e,f,g,h,i,j	$h_{8,9}$	–
Stage 19	g_7	-	a,b,c,d,e,f,g,h,i,j	$g_{7,10}$	–
				Ordering	**DFS traversal**

Note: The traversal stack is obtained by looking at stack contents in Stage 4 and Stag along with column values of step 4as shown below:

	$e_{6,2}$	$j_{10,7}$	
$d_{3,1}$	$b_{5,3}$	$i_{9,8}$	**Traversal stack**
$c_{2,5}$	$f_{4,4}$	$h_{8,9}$	
$a_{1,6}$		$g_{7,10}$	

The DFS traversal of the graph is obtained by looking at the output column of step 5 in above table. Joining all the edges shown in last column of the above table, the DFS traversal is shown below:



DFS traversal

Note: The DFS traversal shown in figure (a) and (b) are same. The dotted lines represent the back edgesand solid lines represent tree edges.

Now, let us see "How to check for the presence of a cycle in the graph?" After traversing the graph in DFS, if there are no back edges, then the graph is acyclic. If there is a back edge from some vertex *u* to its ancestor *v*, then the graph has a cycle.

Analysis of DFS: The analysis of DFS depends on the number of vertices and whether the *adjacency matrix* or *adjacency list* representation of a graph is used. The time efficiency of DFS is same as that of BFS.

Ex S

ons of DFS are shown below:

her a given graph is connected or not.

- To check whether the graph is acyclic or not
- To find the spanning tree
- Solving puzzles with only one solution, such as mazes.
- Topological sorting

Now, let us "Compare and contrast BFS traversal and DFS traversal" The main facts about BFS and DFS are shown below:

DFS	BFS
♦ The data structures used is stack	♦ The data structures used is queue
♦ 2 vertex ordering is used, as the order in which the items are inserted is different from the order in which the items are deleted	♦ 1 vertex ordering is used, as the order in which the items are inserted is same as the order in which the items are deleted
♦ The exploration of a node is postponed as soon as a new unexplored node is reached and the examination of the new node begins immediately.	♦ A node is fully explored before the exploration of any other node begins
♦ The *tree edges* and *back edges* are present	♦ The *tree edges* and *cross edges* are present
♦ Used to check for connectivity and acyclicity of a graph	♦ Used to check for connectivity and acyclicity of a graph
♦ Time efficiency of adjacency matrix representation is $\theta(\|V\|^2)$	♦ Time efficiency of adjacency matrix representation is $\theta(\|V\|^2)$
♦ Time efficiency of adjacency linked list representation is given by $\theta(\|V\| + \|E\|)$	♦ Time efficiency of adjacency linked list representation is $\theta(\|V\| + \|E\|)$

5.3.3 Topological Sorting

Let us see "What is topological sorting?"

Definition: The topological sort of a *directed acyclic graph (DAG)* G = (V, E) is a linear ordering of all the vertices such that for every edge (u, v) in graph G, the vertex *u* appears before the vertex *v* in the ordering. A *topological sort* of a graph can be viewed as an ordering of vertices along a horizontal line so that all directed edges go from left to right. For a cyclic graph, no linear ordering is possible.

Note: If A depends on B and B depends on A, then it is cyclic. A graph which is cyclic does not have topological sequence (Ex: To get a job one should have work experience, but to get work experience one should have a job)

Now, let us see "What are the various methods using which we can obtain the topological sorting?" The *topological sorting* can be done using following two methods:

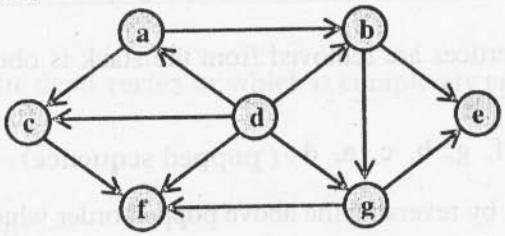
Topological sorting methods → DFS method
Topological sorting methods → Source removal method

5.3.3.1 Topological sort using DFS method

Now, let us see "How to get the topological order using DFS method?" The topological order using DFS method can be obtained as shown below:

Step 1: Select any arbitrary vertex
Step 2: When a vertex is visited for the first time, it is pushed on to the stack
Step 3: When a vertex becomes a dead end, it is removed from the stack
Step 4: Repeat step 2 to 3 for all the vertices in the graph
Step 5: Reverse the order of deleted items to get the topological sequence.

Example 1: Apply the DFS based algorithm to solve the topological sorting problem for the following graph:



Note: The solution is similar to the DFS traversal of the previous section. Since vertex *a* is the least in alphabetical order, it is selected as the start vertex.

Initial step: Push start vertex *a* onto the stack and add it to S, indicating the vertex is marked or visited.

Stage 1: The various activities that are performed are shown below (Look stage 1 row in the table):

Step 1: Stack not empty. Top of stack is vertex *a*.
Step 2: Obtain the first node which is adjacent to vertex *a* i.e., vertex *b*.
Step 3: Add vertex *b* to S indicating it is visited or marked.
Step 4: Not performed since a vertex is added to S. This step is performed if nothing is added to S

On similar lines perform various stages and express the solution as shown in the table below:

	Step1	Step 2	Step 3	Step 4
	Stack	**v=adj(s[top])**	**Nodes visited S**	**pop(stack)**
Initial step	a	-	a	-
Stage 1	a	b	a, b	-
Stage 2	a, b	e	a, b, e	-
Stage 3	a, b, e	-	a, b, e	**e**
Stage 4	a, b	g	a, b, e, g	-
Stage 5	a, b, g	f	a, b, e, f, g	-
Stage 6	a, b, g, f	-	a, b, e, f, g	**f**
Stage 7	a, b, g	-	a, b, e, f, g	**g**
Stage 8	a, b	-	a, b, e, f, g	**b**
Stage 9	a	c	a, b, c, e, f, g	-
Stage 10	a, c	-	a, b, c, e, f, g	**c**
Stage 11	a	-	a, b, c, e, f, g	**a**
	Note: Stack is empty. So, take the next vertex in sequence which is not visited and push it onto stack and add to S.			
Stage 12	d	-	a, b, c, d, e, f, g	-
Stage 13	d	-	a, b, c, d, e, f, g	**d**

Note: The order in which the vertices are removed from the stack is obtained from the last column. The popped order is:

e, f, g, b, c, a, d **(popped sequence)**

The topological order is obtained by reversing the above popped order which is given by:

d → a → c → b → g → f → e **(Topological order)**

Thus, DFS traversal can be used to obtain topological sorting solution for the given graph. The complete algorithm is shown below:

ALGORITHM DFS(u, n, a)

//Purpose: To obtain the sequence of jobs to be executed resulting in topological order

//Input:

// u – From where the DFS traversal start
// n – the number of vertices in the graph
// a – adjacency matrix of the given graph

// Global variables:

// s – to know what are the nodes visited and what are the nodes that are not visited
// j – index variable to store the vertices (only those nodes which are dead ends or
// those nodes whose nodes are completely explored
// res – an array which holds the order in which the vertices are popped

//Output:

// res -indicates the vertices in the reverse order that are to be executed

Step 1: [Visit the vertex u]
s[u] ← 1

Step 2: [Traverse deeper in to the graph till we get dead end or till all vertices are visited]
for v ← 0 **to** n-1 **do**
if (a[u][v] = 1 **and** s[v] = 0) **then**
DFS(v, n, a)
end if
end for

Step 3: [Store the dead vertex or which is completely explored]
j ← j + 1
res[j] ← u;

Step 4: [Finished]
return

ALGORITHM topological_order(n, a)

//**Purpose**: To obtain the sequence of jobs to be executed resulting in topological order

//**Input**: a – adjacency matrix of the given graph
// n – the number of vertices in the graph

// **Global variables**:
// s – to know what are the nodes visited and what are the nodes that are not visited
// j – index variable to store the vertices (only those nodes which are dead ends or
// those nodes whose nodes are completely explored
// res – an array which holds the order in which the vertices are popped

//**Output**:
// res -indicates the vertices in the reverse order that are to be executed

Step 1: [Initialization to indicate that no vertex has been visited]

```
for i ← 0 to n-1 do
        s[i] ← 0
end for

j ← 0; // An index to store the vertices which are dead ends and
                which are completely explored
```

Step 2: [Process each vertex in the graph]

```
for u ← 0 to n-1 do
        if (s[u] = 0) call DFS(u, n, a);
end for
```

Step 3: [Output the topological sequence by printing in the reverse order of popped sequence]

```
for i ← n-1 down to 0
        print res[i]
end for
```

Step 4: [Finished]

```
return;
```

Analysis of Topological sort using DFS method (represented as adjacency matrix)

The running time of function DFS is the time required to execute the statement

```
for v ← 0 to n-1 do
    if ( a[u][v] = 1 and s[v] = 0 ) then
        DFS(v, n, A) // This statement is executed n times
```

So, the time efficiency is given by the following relation:

$$
\begin{aligned}
f(n) &= \sum_{v=0}^{n-1}\sum_{v=0}^{n-1} 1 \\
&= \sum_{v=0}^{n-1} n-1-0+1 \\
&= \sum_{v=0}^{n-1} n \\
&= n\sum_{v=0}^{n-1} 1 \\
&= n(n-1-0+1) \\
&= n^2
\end{aligned}
$$

So, the time complexity of topological sort is given by $\theta(n^2)$. For a given graph G = (V,E) where V is the number of vertices(Here, we are using n as the number of vertices),

$$n = |V|$$

Therefore, time complexity of topological sort is given by $\theta(|V^2|)$.

Analysis of Topological sort represented using adjacency list(DFS)

Let G = (V, E) be a graph where

- V is set of vertices in the graph.
- E is set of edges in the graph

The adjacency linked lists is an array of linked lists. All the vertices 0, 1, 2,.....n-1 represent the vertices and they become the index into an adjacency linked. So, if n represent the number of vertices in the graph, and |V| also represent the number of vertices in the graph, we have n = |V| .....(1)

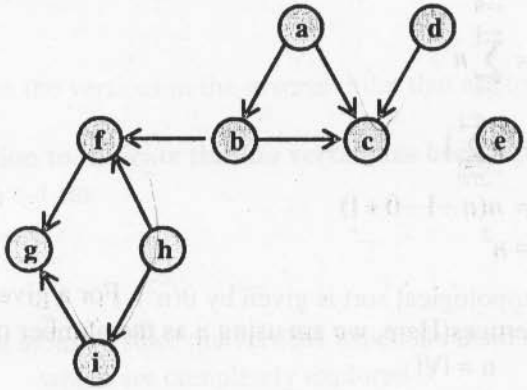
When a vertex u is deleted from stack, it is required to find the vertices adjacent to u. The nodes adjacent to u are represented in the form of an adjacency list. Thus, for every node u deleted from stack, it is necessary to traverse the list to find the nodes adjacent to u which has at most |E| number of edges. (**Note**: |E| indicates the total number of edges). The total number of edges to be accessed = |E|.....(2)

In DFS, each vertex is visited and each edge is taken from the list. So, the time efficiency of accessing each vertex and time efficiency to access the edge is obtained by adding equation (1) and (2). So, time efficiency is given by f(n) = n + m

i.e., f(n) = |V| + |E|

The total time complexity of BFS is given by $f(n) \in O(\|V\|+\|E\|)$

Example 2: Apply the DFS based algorithm to solve the topological sorting problem for the following graph: **(July/Aug 2004)**



Note: The solution is similar to the DFS traversal of the previous section. Since vertex *a* is the least in alphabetical order, it is selected as the start vertex.

Initial step: Push start vertex *a* onto the stack and add it to S, indicating the vertex is marked or visited.

Stage 1: The various activities that are performed are shown below (Look stage 1 row in the table):

Step 1: Stack not empty. Top of stack is vertex *a*.
Step 2: Obtain the first node which is adjacent to vertex *a* i.e., vertex *b*.
Step 3: Add vertex *b* to S indicating it is visited or marked.
Step 4: Not performed since a vertex is added to S. This step is performed if nothing is added to S

Note: When we proceed as we did in stage 1, if we encounter a dead end, the vertex from the stack is popped and above procedure is repeated. During this process, if stack is empty, take a next vertex in sequence and proceed as explained earlier. The various stages and the solution can be obtained as shown in the table below:

	Step1	Step 2	Step 3	Step 4
	Stack	**v=adj(s[top])**	**Nodes visited S**	**pop(stack)**
Initial step	a	-	a	-
Stage 1	a	b	a, b	-
Stage 2	a, b	c	a, b, c	-
Stage 3	a, b, c	-	a, b, c	**c**
Stage 4	a, b	f	a, b, c, f	-
Stage 5	a, b, f	g	a, b, c, f, g	-
Stage 6	a, b, f, g	-	a, b, c, f, g	**g**
Stage 7	a, b, f	-	a, b, c, f, g	**f**
Stage 8	a, b	-	a, b, c, f, g	**b**
Stage 9	a	-	a, b, c, f, g	**a**
	Note: Stack is empty. So, take the next vertex in sequence which is not visited and push it onto stack and add to S.			
Initial step	d	-	a, b, c, d, f, g	
Stage 10	d	-	a, b, c, d, f, g	**d**
	Note: Stack is empty. So, take the next vertex in sequence which is not visited and push it onto stack and add to S.			
Initial step	e	-	a, b, c, d, e, f, g	-
Stage 11	e	-	a, b, c, d, e, f, g	**e**
	Note: Stack is empty. So, take the next vertex in sequence which is not visited and push it onto stack and add to S.			
Initial step	h	-	a,b,c,d,e,f,g,h	-
Stage 12	h	i	a,b,c,d,e,f,g,h,i	-
Stage 13	h, i	-	a,b,c,d,e,f,g,h,i	**i**
Stage 14	h	-	a,b,c,d,e,f,g,h,i	**h**

Note: The order in which the vertices are removed from the stack is obtained from the last column. The popped order is:

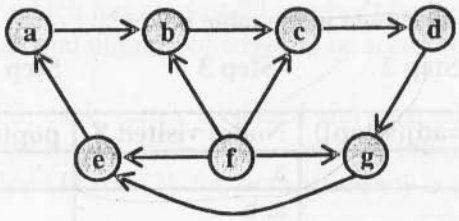
c, g, f, b, a, d, e, i, h **(popped sequence)**

The topological order is obtained by reversing the above popped order which is given by:

h → i → e → d → a → b → f → g → c **(Topological order)**

Note: A topological sorting problem may have several alternative solutions.

Example 3: Apply the DFS based algorithm to solve the topological sorting problem for the following graph:



Solution: Since there is a cycle a-b-c-d-g-e-a, the topological sequence does not exist for this problem.

5.3.3.2 Topological sort using source removal method

Now, let us see "How we can obtain the topological ordering using source removal method?" This method is based on *decrease-and-conquer* technique: In this method, a vertex with no incoming edges (whose indegree is 0) is selected and deleted along with the outgoing edges. If there are several vertices with no incoming edges, arbitrarily a vertex is selected. The order in which the vertices are visited and deleted one by one results in topological sorting.

For example, consider the graph shown in figure below, where 1, 2, 3, 4 and 5 represent the tasks to be completed.

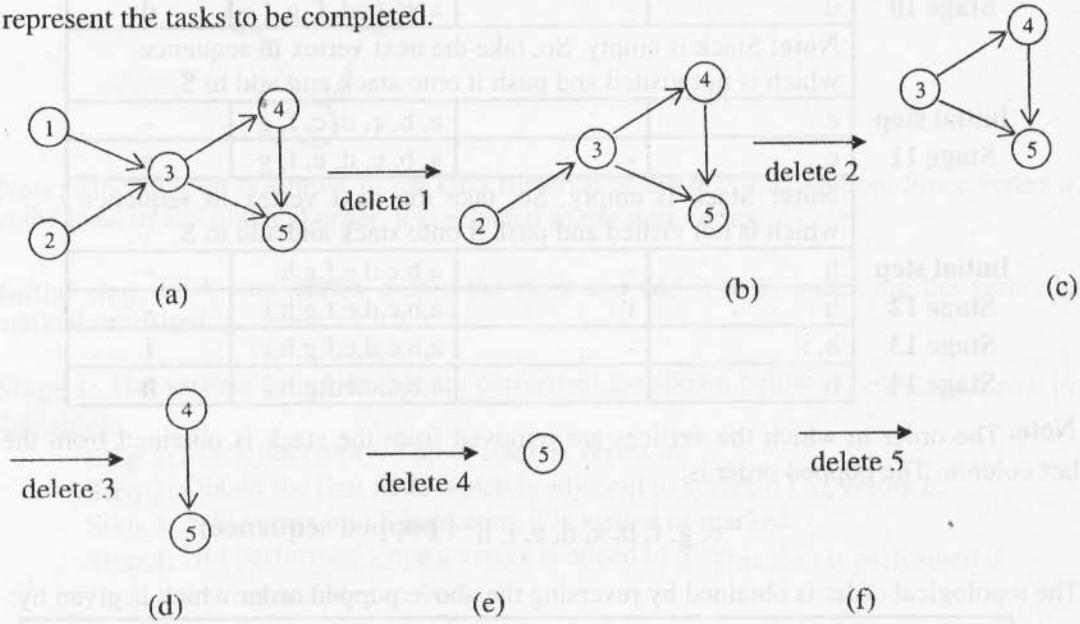
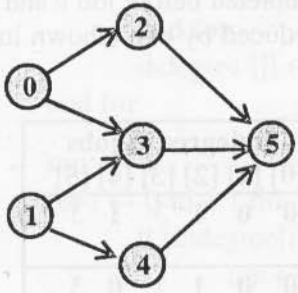


Fig. 5.3.3.2 Graph showing the pre-tasks to be completed

It is clear from figure (a) that tasks 1 and 2 are independent and one of them can be selected and processed and graph shown in figure (b) is obtained. Now, the next independent task is 2 (figure b) and can be deleted. The sequence of vertices that are processed and deleted are shown in figure (a) through (f). Thus, the order in which each task has to be completed is given by 1, 2, 3, 4 and 5. The topological sequence is shown below:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ **(Topological order)**

Design Let us consider the following graph shown in figure (a). The adjacency matrix for this graph is shown in figure (b).



(a)

	0	1	2	3	4	5
0	0	0	1	1	0	0
1	0	0	0	1	1	0
2	0	0	0	1	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	0

Adjacency matrix A

0	0	1	3	1	3

← Indegree of each node

Sum of columns

(b)

Fig. 5.5.3.3 Graph showing the pre-tasks to be completed and adjacency matrix

Note: By adding the columns of adjacency matrix the indegree of each vertex is obtained. It is clear from above figure that indegree[i] gives the number of tasks to be completed before i^{th} job is taken for executing. So, the vertices for which the indegree is 0, are independent task which do not depend on any task and can be computed independently. Observe the following facts:

- Indegree[0] = Indegree[1] = 0. Since, indegree of jobs 0 and 1 is 0, these jobs are considered independent jobs.
- Indegree[2] = 1 indicates that job 2 depends on one job (namely job 0).
- Indegree[3] = 3 indicates that job 3 depends on 3 jobs (namely, 0,1 and 2)
- Indegree[4] = 1 indicates that job 4 depends on only one job (i.e., 1)
- Indegree[5] = 3 which indicates that job 5 depends on three jobs (namely 2, 3 and 4).

The various steps involved in the design have to be performed repeatedly till the stack is empty:

- Find the vertices whose indegree is zero and place them on the stack (shown in first column of table). These vertices denote the jobs which do not depend on any other job and can be performed independently.
- Pop a vertex *u* and it is the task to be done (2nd column of table)
- Add the vertex ***u*** to the solution vector (3rd column of table). This vertex represents the next job to be considered for executing.
- Find the vertices *v* adjacent to vertex ***u***. The vertices *v* represents the jobs which depend on job ***u*** (4th column in the table)
- Decrement indegree[v] by one, thereby reducing the number dependencies on *v* by one. Since, indegree[v] gives the number of jobs to be completed before job *v* and job *u* is completed, the number of dependent jobs of *v* are reduced by one (shown in 5th column of table)

Steps	stack	u = pop( )	Solution T	v = adj (u)	Indegree of jobs [0] [1] [2] [3] [4] [5]
Initial step	Find indegree of each node →				0 0 1 3 1 3
1	0, 1	1	1	3, 4	0 0 1 2 0 3
2	0, 4	4	1 4	5	0 0 1 2 0 2
3	0	0	1 4 0	2 , 3	0 0 0 1 0 2
4	2	2	1 4 0 2	3, 5	0 0 0 0 0 1
5	3	3	1 4 0 2 3	5	0 0 0 0 0 0
6	5	5	1 4 0 2 3 5	-	-

Topological sequence

The detailed activities performed are shown below step by step:

- In initial step, find the indegree of each node (shown in the last column of table)
- The indegree[0] and indegree[1] are zero indicating jobs 0 and 1 are independent jobs and so are pushed on to the stack. Then job 1 is deleted from stack and it is added to solution vector. The nodes adjacent to 1 are taken (for example, jobs 3 and 4). Reduce the dependency of job 3 and job 4 by one (see the last column in step 1). Observe that indegree[3] and indegree[4] are decremented by one. Now, indegree[4] is zero.
- Since indegree[4] is zero, it is pushed on to the stack. Then delete job 4 from the stack and it to solution. Then, find the vertices adjacent to 4. Here, 5 is adjacent to 4 and so decrement indegree[5] by one and the process is repeated till stack is empty

```
Algorithm topological_sort(a, n, T)

//Purpose: To obtain the sequence of jobs to be executed resulting in topological order

//Input: a – adjacency matrix of the given graph
//       n – the number of vertices in the graph

//Output:
//       T – indicates the jobs that are to be executed in the order

    for j ← 0 to n-1 do                    //Obtain indegree of each vertex
        sum ← 0
        for i ← 0 to n-1 do
            sum ← sum + a [i] [j]
        end for
        indegree [j] ← sum
    end for

    top ← -1                               // Place independent jobs on the stack
    for i ← 0 to n-1 do
        if (indegree[i] = 0)
            top ← top + 1
            s[top] ← i
        end if
    end for
    while top ≠ 1                          // Find the topological sequence
        u ← s[top]                         // Delete the job from the stack
        top ← top – 1
        Add u to solution vector T
        for each vertex v adjacent to u
            decrement indegree[v] by one
            if (indegree[v] = 0)
                top ← top + 1
                s[top] ← v
            end if
        end for
    end while

    write T                                //Output the topological sequence
    return
```

5.4 Algorithms for Generating Combinatorial Objects

Now, let us answer the question "What are the different types of combinatorial objects?" There are three types of combinatorial objects are shown below:

- Permutations
- Combinations
- Subsets of a given set

5.4.1 Generating Permutations

The permutations can be generated using various methods such as:

- Bottom up minimal change algorithm
- Johnson Trotter algorithm
- Lexicographic ordering

5.4.1.1 Bottom up minimal-change algorithm

Let us see "How to generate permutations by bottom up minimal-change algorithm?"

In this approach, we first assume that we have a solution to a given smaller problem. For example, assume that we have solution for *n-1* elements. The solution to the larger problem with *n* elements is obtained by inserting *n* in various possible positions of *n-1* elements. It is clear from this concept that all the permutations obtained in this fashion are distinct and the total number obtained will be n*(n-1)! = n!.

Now, let us "Generate permutations of {1, 2, 3, 4} using bottom-up minimal-change algorithm"

Initial : 1

Insert 2: Insert 2 from right to left to 1

12 21

Insert 3: Insert 3 from right to left to 12

123

132

312

Insert 3 from left to right 21

321

231

213

Insert 4: Take each of the above solution, keep inserting **4** from left to right for one instance and right to left for another instance of the solution alternatively as shown below:

(a) Insert 4 from right to left to 123

1234

1243

1423

4123

(b) Insert 4 from left to right to 132

4132

1432

1342

1324

(c) Insert 4 from right to left to 312

3124

3142

3412

4312

(d) Insert 4 from left to right to 321

4321

3421

3241

3214

(e) Insert 4 from right to left to 231

2314

2341

2431

4231

(f) Insert 4 from left to right to 213

4213

2413

2143

2134

Note: It is clear from the above solution that, if we have to generate the permutation for *n* elements, then we will have n! permutations. Also note that inserting from right to left for one set and inserting from left to right for another set, it is easier to generate the permutations. Using this approach, each permutation can be obtained from its immediate predecessor by exchanging only two items and satisfying *minimal-change* requirement.

Advantages

- Each permutation can be obtained from its immediate predecessor by exchanging just two elements
- The minimal-change requirement is beneficial for algorithm's speed and for the application using the permutations

Disadvantages

- To get the permutations for *n* elements we need to know the permutations for (n-1) elements.

5.4.1.2 Johnson Trotter algorithm

Now, the question is "How to over come the disadvantage of bottom up minimal exchange method?" Using Johnson's algorithm we can overcome this disadvantage. Using this algorithm, we can generate permutations of n elements without knowing the permutation of n-1 elements. This is an efficient way of generating permutations. This is achieved by associating a direction with each element in a permutation. The direction can be from left to right indicated by '→' or right to left by '←'. For example,

$$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$$

Note: If an element k points to a smaller number adjacent to it, then the element k is said to be *mobile*. In the above combination:

- The arrow on 3 points to 2 which is a smaller number. So, element 3 is mobile.
- The arrow on 2 points to 3 which is a higher number. So, element 2 is not mobile.
- The arrow on 4 points to 1 which is a smaller number. So, element 4 is mobile.
- The arrow on 1 points to 4 which is a higher number. So, element 1 is not mobile.

The *Johnson-Trotter-algorithm* for generating various permutations is shown below:

Algorithm Johnson_Trotter(n)

// **Purpose**: Generates various permutations of the sequence [1, 2, 3, ...n]

// **Input**: n – a positive integer n

// **Output**: A list of permutations

Step 1: [Obtain the first permutation]

$$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{4}\ldots\ldots\overleftarrow{n}$$

Step 2 [Generate the subsequent permutations]

while mobile integer k exists

find the largest mobile element k

swap the mobile element k with immediate adjacent integer pointing to by arrow mark

reverse the direction of arrows of all integers larger than k.

end while

Step 3:[Finished]

return

Now, let us "Generate permutations of {1, 2, 3, 4} using Johnson Trotter algorithm" The various permutations of {1, 2, 3, 4} and the action taken to generate the permutations using Johnson Trotter algorithm is shown below:

1.	$\overrightarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{4}$	**4 is mobile.** Exchange 4 and 3
2.	$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}\ \overleftarrow{3}$	**4 is mobile.** Exchange 4 and 2
3.	$\overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{3}$	**4 is mobile.** Exchange 4 and 1
4.	$\overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$	**3 is mobile.** Exchange 3 and 2, reverse direction of 4
5.	$\overrightarrow{4}\ \overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$	**4 is mobile.** Exchange 4 and 1
6.	$\overleftarrow{1}\ \overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}$	**4 is mobile.** Exchange 4 and 3
7.	$\overleftarrow{1}\ \overleftarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}$	**4 is mobile.** Exchange 4 and 2
8.	$\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}$	**3 is mobile.** Exchange 3 and 1, reverse direction of 4
9.	$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}$	**4 is mobile.** Exchange 4 and 2
10.	$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}$	**4 is mobile.** Exchange 4 and 1
11.	$\overleftarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}$	**4 is mobile.** Exchange 4 and 3
12.	$\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$	**2 is mobile.** Exchange 2 and 1, reverse direction of 4 and 3
13.	$\overrightarrow{4}\ \overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$	**4 is mobile.** Exchange 4 and 3
14.	$\overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}$	**4 is mobile.** Exchange 4 and 2
15.	$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$	**4 is mobile.** Exchange 4 and 1
16.	$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}$	**3 is mobile.** Exchange 3 and 2, reverse direction of 4
17.	$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}$	**4 is mobile.** Exchange 4 and 1
18.	$\overleftarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{1}$	**4 is mobile.** Exchange 4 and 3
19.	$\overleftarrow{2}\ \overleftarrow{4}\ \overrightarrow{3}\ \overleftarrow{1}$	**4 is mobile.** Exchange 4 and 2
20.	$\overleftarrow{4}\ \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$	**3 is mobile.** Exchange 3 and 1, reverse direction of 4
21.	$\overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$	**4 is mobile.** Exchange 4 and 2
22.	$\overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}\ \overrightarrow{3}$	**4 is mobile.** Exchange 4 and 1
23.	$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}\ \overrightarrow{3}$	**4 is mobile.** Exchange 4 and 3
24.	$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}\ \overrightarrow{4}$	**No mobile integer. Algorithm stops**

Note: It is clear from the above example that the items {1, 2, 3, 4} can be arranged in 24 different ways (4!). In general, *n* items can be arranged in n! ways. Thus, the time complexity of Johnson Trotter algorithm is θ(n!). We know that the algorithms with these time complexities are horribly slow for all values of n (except small values of n). However, it is to be noted that this algorithm is most efficient since it is not the problem of the algorithm. But, the nature of the problem is like that. This problem asks to generate too many items.

5.4.1.3 Lexicographic algorithm

Another easiest way of generating permutations is writing the given integers in *lexicographic* order. The general algorithm is shown below:

Step 1: [Generate the sequence in increasing order]
[1, 2, 3, 4, ...n] which are denoted by $a_1, a_2, a_3, \ldots\ldots a_n$.

Step 2: [Scan the symbols from right to left and generate the lexicographic sequence]

if $a_i < a_{i+1}$ exchange a_i and a_{i+1}

if $a_i > a_{i+1}$, consider a_{i-1}. **if** $a_{i-1} < a_i$, replace a_{i-1} by next higher number on the right hand side and rearrange the remaining elements in the permutation in increasing order.

Step 3: [Repeat through step 2 by considering the item a_{i-2}, a_{i-4} and so on]

Step 4: [Finished]
return

Now, let us "Generate permutations of {1, 2, 3, 4} using lexicographic order" The various permutations of {1, 2, 3, 4} and the action taken to generate the permutations using lexicographic order is shown below:

a_1	a_2	a_3	a_4	**Action taken**
1	2	3	4	Since 3 < 4, exchange 3 and 4
1	2	4	3	Replace 2 with next higher number on RHS and arrange the remaining elements in ascending order
1	3	2	4	Since 2 < 4, exchange 2 and 4

1	3	4	2	Replace 3 with next higher number on RHS and arrange the remaining elements in ascending order
1	4	2	3	Since 2 < 3, exchange 2 and 3
1	4	3	2	Replace 1 with next higher number on RHS and arrange the remaining elements in ascending order
2	1	3	4	Since 3 < 4, exchange 3 and 4
2	1	4	3	Replace 1 with next higher number on RHS and arrange the remaining elements in ascending order
2	3	1	4	Since 1 < 4, exchange 1 and 4
2	3	4	1	Replace 3 with next higher number on RHS and arrange the remaining elements in ascending order
2	4	1	3	Since 1 < 3, exchange 1 and 3
2	4	3	1	Replace 2 with next higher number on RHS and arrange the remaining elements in ascending order
3	1	2	4	Since 2 < 4, exchange 2 and 4
3	1	4	2	Replace 1 with next higher number on RHS and arrange the remaining elements in ascending order
3	2	1	4	Since 1 < 4, exchange 1 and 4
3	2	4	1	Replace 2 with next higher number on RHS and arrange the remaining elements in ascending order
3	4	1	2	Since 1 < 2, exchange 1 and 2
3	4	2	1	Replace 3 with next higher number on RHS and arrange the remaining elements in ascending order
4	1	2	3	Since 2 < 3, exchange 2 and 3
4	1	3	2	Replace 1 with next higher number on RHS and arrange the remaining elements in ascending order
4	2	1	3	Since 1 < 3, exchange 1 and 3
4	2	3	1	Replace 2 with next higher number on RHS and arrange the remaining elements in ascending order
4	3	1	2	Since 1 < 2, exchange 1 and 2
4	3	2	1	This is the final sequence and the algorithm stops

5.4.2 Generating subsets

In exhaustive search in section 3.6.2, we have seen that problems such as knapsack problem need all possible subsets. In this section, let us see "How to generate various subsets of a given set?" The set of all subsets of a set is called *power set*. For example, if set A = $\{a_1, a_2, a_3, \ldots a_n\}$ then total number of subsets is given by 2^n. The easiest and most convenient way of generating all possible subsets is shown below:

There is one to one correspondence between 2^n subsets of set A = $\{a_1, a_2, a_3, \ldots a_n\}$ and 2^n bit-string $b_1, b_2, b_3, \ldots b_n$. In this approach, whenever $a_i \in A$, then corresponding b_i is 1 with rest of the bit-patter having 0's. For example,

{0, 0, 0} is equivalent to {} empty set
{1, 0, 1} is equivalent to set $\{a_1, a_3\}$
{1, 1, 1} is equivalent to set $\{a_1, a_2, a_3\}$

Now, let us "Generate subsets of set A=$\{a_1, a_2, a_3\}$ along with bit-pattern representation".

Solution: All possible subsets of set A ={a1, a2, a3} is shown below with equivalent bit-pattern.

Subsets of set A = { a1, a2, a3}	Bit-string representation of subsets $a_1 a_2 a_3$
{}	0 0 0
{a3}	0 0 1
{a2}	0 1 0
{a2, a3}	0 1 1
{a1}	1 0 0
{a1, a3}	1 0 1
{a1, a2}	1 1 0
{a1, a2, a3}	1 1 1

The algorithm to generate the subsets of a given set is shown below:

Algorithm Subset(n)

// **Purpose**: Generate all subset of a set with bit patter as shown in second column of
//

// **Input** : n represents A = {a1, a2, a3,.....}

// **Output**: Generates the subsets of set A in bit-pattern b

Step 1: [Store the appropriate bit-pattern representing all possible subsets]

for i ← 0 to 2^n-1 **do**

// Convert the numbers generated in sequence into binary and store

b[i] ← convert_to_binary(i)

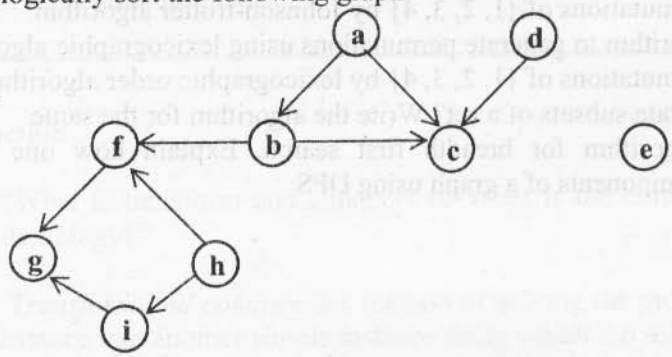
end for

Step 2: [Finished]

return

Exercises:

1. Explain the concept of decrease and conquer methodology, indicating the three major variations of the same.
2. Suggest an algorithm for generating combinatorial objects based on decrease and conquer methodology **(Jan/Feb 2005)**
3. Write DFS algorithm and find its worst case efficiency **(July/Aug 2004)**
4. Write an algorithm to topologically sort a digraph using DFS. Prove the correctness and find the time efficiency **(July/Aug 2004)**
5. Topologically sort the following graph



6. Suggest pseudocodes for depth first search and breadth first search. Illustrate with suitable examples.
7. What is decrease and conquer? Explain the variations of *decrease-and-conquer* technique
8. What is decrease by a constant? Explain with an example
9. What is decrease by a constant factor? Explain with an example
10. What is variable size decrease? Explain with an example
11. Explain insertion sort with example along with the algorithm
12. Obtain the worst case, best case and average case time complexities of insertion sort

13. What are the various methods of graph traversal techniques? Briefly explain
14. Define the following terms
    a. tree edge
    b. cross edge
    c. back edge
15. What is BFS? Write an algorithm to traverse the graph using BFS and obtain its time complexity
16. What is DFS? Write an algorithm to traverse the graph using DFS and obtain its time complexity
17. What are the various applications of BFS and DFS?
18. What are the differences and similarities between BFS and DFS?
19. What is topological sorting? What are the various techniques using which topological sort can be obtained?
20. How do you obtain topological sort using DFS? Explain with an example and write the algorithm for the same
21. How do you obtain topological sort using source removal method? Explain with an example and write the algorithm for the same
22. What is bottom up minimal change algorithm and what is its use?
23. Generate permutations of {1, 2, 3, 4} by bottom-up minimal-change method
24. What is Johnson trotter algorithm? Explain with an example
25. Generate permutations of {1, 2, 3, 4} by Johnson-trotter algorithm
26. Write an algorithm to generate permutations using lexicographic algorithm
27. Generate permutations of {1, 2, 3, 4} by lexicographic order algorithm
28. How to generate subsets of a set? Write the algorithm for the same
29. Write an algorithm for breadth first search. Explain how one can identify connected components of a graph using DFS

Chapter 6: Transform and Conquer

What are we studying in this chapter?

- Concept of Transform and Conquer Technique
- Presorting
- Heap
- Heap Applications
  - Priority queue
  - Heap sort
- Balanced Search Trees
  - AVL trees
  - 2-3 Trees
- Problem reduction
  - Find LCM of two numbers
  - To count paths in a graph
  - Optimization problem
  - Linear programming examples such as knapsack problem etc
  - Graph problems

- 6 hours

6.1 Introduction

Let us see "What is transform and conquer? or What is the concept of transform and conquer methodology?"

Definition: *Transform-and-conquer* is a method of solving the problem by transforming problem's instance into another simple instance using which the solution can be obtained. The transform-and-conquer method as the name indicates, it works in two stages

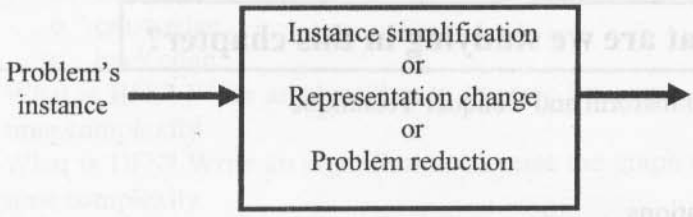
- **Transformation stage:** Here, the problem's instance is changed so that the solution to the problem can be obtained easily. Let us see "How the transformation can be done?" There are three variations using which transformation can be applied. They are:

Variations of transformation → Instance simplification
Variations of transformation → Representation change
Variations of transformation → Problem reduction

- **Conquer stage:** (or solve) the problem

6.2 Transform and Conquer

Using any of the variations of transformation, the problem's instance can be changed using which we can obtain the solution easily as shown below:



Instance simplification: In this strategy, a given problem is transformed into another simple problem for which we can get the solution easily. For example, presorting algorithms, AVL-trees, solving simultaneous equations using Gauss elimination method etc uses instance simplification strategy.

Representation change: In this strategy, one representation of a problem is changed to another representation of the same instance. For example, heap sort, Horner's rule to evaluate polynomials etc., uses representation change.

Problem reduction: In this strategy, a given problem is transformed into another problem for which the solution exists in the form of a known algorithm. For example, finding LCM of two numbers using GCD, finding maximum of n numbers using minimization function, finding minimum of n numbers using maximization function and various other problems uses this strategy.

6.2 Presorting

Now, let us see "What is presorting? Why is it necessary?"

Definition: Arranging the numbers in ascending order or descending order before solving the actual instance of a problem is called pre-sorting. Even though, it is not the solution for a given instance of a problem, using presorting, the given problem can be solved efficiently. The efficiency of some of the algorithms can be increased using pre-sorting. Some of the problems whose efficiency can be improved using presorting are:

- Checking element uniqueness in an array
- Searching problem
- Computing a mode

Now, the question is "Which sorting technique to use to presort?" We have seen very simple and sophisticated algorithms with the following time complexities:

- Bubble sort: with time efficiency of $\theta(N^2)$ (discussed in section 3.2)

- Selection sort: with time efficiency of $O(N^2)$ (discussed in section 3.3)
- Insertion sort: with time efficiency of $O(N^2)$ (discussed in section 5.2)
- Quick sort: with time efficiency of $\Omega(N \log N)$ in best and average case and $O(N^2)$ in the worst case. (discussed in section 4.5.2)
- Merge sort: with time efficiency of $\theta(N \log N)$ both in the worst case and efficiency (discussed in section 4.5.1)

Note: The advanced sorting technique such as *merge sort* has the time complexity of $\theta(n \log n)$ both in worst case and average case. Even though the quick sort is also having the time complexity of $\theta(n \log n)$ in the best case and average case, since its worst case time complexity is $\theta(n^2)$, merge sort can be a better choice. But, if the input is random then quick sort is better than merge sort. If we consider all the cases such as *worst case, best case* and *average case*, then merge sort is a better choice.

Now, we shall see "How the pre-sorting is useful to improve the efficiency of some solutions?"

6.2.1 Checking element uniqueness in an array

The algorithm to check element uniqueness in an array has been discussed in section 2.7.3. The time efficiency of this algorithm is $O(N^2)$. Now, the question is "Is it possible to improve the efficiency further from being $O(N^2)$?" Yes, the efficiency of an algorithm can be improved using pre-sorting as shown below:

Design methodology Let us take an example. Consider the elements shown below:

	0	1	2	3	4	5	6
Items	20	5	50	10	25	50	15

There are two duplicate items 50 in position 2 and 3 respectively. These elements can be arranged in ascending order as shown below:

Index →	0	1	2	3	4	5	6	
Items	5	10	15	20	25	50	50	n = 7

Observe that in the sorted list, the duplicate elements 50 are present side by side in positions 5 and 6 respectively. To check whether items are present or not we may have to do the following comparisons in the worst case:

a[0] = a[1]
a[1] = a[2]
... .
... .
a[5] = a[6]

↓

In general, if (a[i] = a[i+1]) we return -1 indicating elements are not unique
where i = 0, 1, 2, 3, 4, 5
i.e., i = 0 to 5
i = 0 to 7 – 2
i = 0 to n – 2 (Note: In the given problem n = 7)

The pseudocode can take the form shown below:

```
for i ← 0 to n-2 do
        if ( a[i] = a[i+1]) return 0      // Elements are not unique
end for                                   // i.e., duplicate items found

return 1                                  // Elements are unique
                                          // i.e., duplicate items not found
```

The complete algorithm is shown in below

ALGORITHM Unique(n, a[]):

//**Purpose**: This algorithm checks whether the elements given in the list are unique or not

// **Inputs** :
// n – the number of items present in the table
// a – the table (represented using array) consisting of *n* elements

//**Output**:
// return 0 – if elements are not unique
// return 1 – if elmenets are unique.

```
MergeSort(a, 0, n-1)          // Pre-sort using merge sort

for i ← 0 to n-2 do
    if (a[i] = a[i+1])        // Current item and next item are same?
        return 0;             // If yes, elements are not unique i.e, duplicate item found
    endif
end for

return 1;                     // Elements are unique i.e., no duplicate item found
```

Analysis The time efficiency of this algorithm can be computed as shown below:

$$\begin{array}{ccccc} \text{Time efficiency of algorithm} & = & \text{Time efficiency of merge sort} & + & \text{Time efficiency to check for uniqueness} \\ f(n) & = & f_1(n) & + & f_2(n) \end{array} \quad \text{.....(1)}$$

The time efficiency to check for uniqueness can be obtained as shown below:

Step 1: The parameter to be considered is *n*, which represent size of the input

Step 2: The basic operation is the statement "if a[i] = a[i+1]" in the innermost for loop

Step 3: The total number of times the above statement is executed can be obtained as shown below:

```
for i = 0 to n-2
    if (a[i] = a[j])
```

$$f_2(n) = \sum_{i=0}^{n-2} 1$$

$$= n-2-0+1$$

$$= n-1 \approx n$$

i.e., $f_2(n) = n$

So, time complexity is given by $f_2(n) \in \theta(n)$.....(2)

Time efficiency of merge sort is $f_1(n) \in \theta(n \log n)$.....(3)

By substituting equation (2) and (3) in equation we have,

$$f(n) = \theta(n \log n) + \theta(n)$$

$$= \theta(\max\{n \log n, n\}) \text{ [Property of asymptotic notation: section 2.5]}$$

$$= \theta(n \log n)$$

So, time complexity to find the uniqueness in the worst case is given by $f(n) \in \theta(n \log n)$

Note: The time complexity of pre-sorting is much superior when compared with brute-force technique whose time complexity is $O(n^2)$ discussed in section 2.7.3. If the array is not sorted, use an efficient algorithm(quick sort or merge sort or heap sort) to sort first and then use the algorithm shown in this section to check for the uniqueness of elements in the given array.

6.2.2 Searching

The time efficiency of linear search is $f(n) = \theta(n)$

The time efficiency of binary search is $\theta(\log_2 n)$

Comparing the above two time efficiencies binary search is far more superior when compared to linear search. But, the necessary condition for binary search is that the array has to be sorted earlier. If the array is not sorted, we can use merge sort to sort the elements and then use binary search to search for an element. In such a situation, the time complexity of binary search with presorting can be obtained as shown below:

$f(n)$ = time complexity of merge sort + time complexity of binary search

$= \theta(n \log n) + \theta(\log_2 n)$

$= \theta(\max\{n \log n, \log_2 n\})$ [Property of asymptotic notation: section 2.5]

$= \theta(n \log n)$

So, time complexity of binary search algorithm using presorting is $f(n) \in \theta(n \log n)$

Note: The time complexity of binary search using pre-sorting technique is $\theta(n \log n)$ and the time complexity of linear search in the worst case is given by $\theta(n)$. So, linear search is better than binary search with pre-sorting. But, we can justify that binary search using pre-sorting technique is more efficient than linear search, if the list has to be searched very frequently. If items to be searched are less, then we can use the linear search. Otherwise, it is better to go for binary search with presorting.

6.2.3 To find the mode

Let us see "What is a mode?"

Definition: In a given list, an item that occurs maximum number of times is called a *mode*. For example, consider the list of items shown below:

20, 10, 25, 40, 25, 20, 25, 15, 25

In the above list, 20 is duplicated, 25 is duplicated. Out of all the items, the item 25 occurs more frequently than any other item and so *mode* of given elements is 25. The mode can be computed using:

- Brute force technique
- Presorting

Now, let us see "How to design the algorithm using brute-force technique?" The algorithm to find the *mode* using straight forward approach (*brute-force-strategy*) is shown below:

```
Algorithm mode(a[], n):

//Purpose: This algorithm finds the mode in a given list (represented using an array)

// Inputs :
//      n – the number of items present in the list
//      a – the list (represented using array) consisting of n elements

// Output:
//      returns mode

modecount ← 1                                   // Since an item appears at least once
modevalue ← a[0]

for i ← 0 to n-2 do
    count ← 1
    for j ← i + 1 to n – 1 do
        if (a[i] = a[j])                        // Mode is present
            count ← count + 1
        end if
    end for

    if (count > modecount)                      // new mode value found
        modecount ← count
        modevalue ← a[i]
    end if
end for

return modevalue
// end of the algorithm
```

Now, let us see "What is the time complexity of the algorithm to obtain mode using brute force technique?"

Analysis The time efficiency of algorithm to find the mode using brute-force can be computed as shown below:

Step 1: The parameter to be considered is n, which represent size of the input

Step 2: The basic operation is the statement "if a[i] = a[j]" in the innermost for loop

Step 3: The total number of times the above statement is executed can be obtained as shown below:

```
for i ← 0 to n-2 do
    count ← 1
    for j ← i + 1 to n – 1 do
        if (a[i] = a[j])          // Mode is present
```

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

$$= \sum_{i=0}^{n-2} n-1-i-1+1$$

$$= \sum_{i=0}^{n-2} n-1-i$$

$$= (n-1) + (n-2) + (n-3) + \ldots\ldots..+3+2+1$$

$$= \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} \approx n^2$$ by neglecting constants and lower order terms

Therefore, time efficiency is given by $f(n) = O(n^2)$

Mode using presorting We can improve this time complexity by presorting. Consider the following list of elements which are sorted:

10, 15, 20, 20, 25, 25, 25, 25, 40

It is clear from the above sorted array that all equal items will be adjacent and mode can be found very easily by counting the longest run of adjacent equal elements.

Design Let us consider the scenario shown below and find the running length of item 20.

...	20	20	20	20	20	20	30	....
..3	4	5	6	7	8	9	10	11...
	i	j	j	j	j	j	j	

Here, the item a[i] i.e., 20 should be compared with a[5], a[6], a[7], a[8], a[9] and a[10] as shown below:

Keep Comparing			
a[4] = a[5] i.e., 20		a[4] = a[4+1]	i.e., 20
a[4] = a[6] i.e., 20	can be written as	a[4] = a[4+2]	i.e., 20
a[4] = a[7] i.e., 20	⟹	a[4] = a[4+3]	i.e., 20
a[4] = a[8] i.e., 20		a[4] = a[4+4]	i.e., 20
a[4] = a[9] i.e., 20		a[4] = a[4+5]	i.e., 20
a[4] = a[10] i.e., 30		a[4] = a[4+6]	i.e., 30
till a[4] is not equal to R.H.S		**i**	**i + j**

Now, it is evident that, in general, we have to keep comparing a[i] with a[i+j] as long as i+j is less than or equal to *n-1*. Observe that initial value of *j* is 1. Now, the pseudocode can be written as:

```
j ← 1
while ( i + j <= n and a[i]=a[i+j])
        j ← j + 1
end while
```

Once control comes out of the loop, *j* gives the run length of a[i]. If this run length is greater than the previous run length *pj*, then *pj* is set to *j* and set mode value to a[i]. Now, the pseudocode can be written as

```
j ← 1
while (i + j <= n and a[i]=a[i+j])          // or while (i + j < n and a[i] = a[i+j])
        j ← j + 1
end while

//Obtain the largest mode value and its frequency
if (j > pj)
        pj = j
        modevalue = a[i]
end if
```

Now, it is required to update the position of running length *i* to *j* using the statement

```
i ← i + j
```

The above procedure has to be performed as long as *i* is less than *n* and the corresponding algorithm which uses *transform-and-conquer* strategy is shown below:

ALGORITHM mode(n, a[])

//**Purpose**: This algorithm finds the mode in a given table *a* (represented using an array)

// **Inputs** :
// n – the number of items present in the table
// a – the table (represented using array) consisting of *n* elements

//**Output**:
// returns mode of an item which occurs most

```
Mergesort(a, 0, n-1)          // Pre-sort the elements using merge sort

i ← 0                         // Position of current run
frequency ← 0                 // Highest frequency seen so far

while i < n do
    j ← 1;

    while i + j < n and a[i] = a[i+j] do
        j ← j + 1
    end while

    if j > frequency
        frequency ← j;
        modevalue ← a[i]
    end if

    i ← i + j
end while

return modevalue
// end of the algorithm
```

Analysis The time efficiency of the above algorithm can be obtained as shown below:

Time efficiency of algorithm	=	Time efficiency of merge sort	+	Time efficiency to check for consecutive elements

$$f(n) = f_1(n) + f_2(n) \quad \ldots\ldots\ldots(1)$$

The time efficiency of merge sort is $f_1(n) = \theta(n \log n)$ .....(2)
The time efficiency to check for consecutive elements is $f_2(n) = \theta(n)$ .....(3)

Substituting equation (2) and (3) in equation (1) we have,

$$\begin{aligned} f(n) &= \theta(n \log n) + \theta(n) \\ &= \theta(\max\{n \log n, n\}) \text{ [Property of asymptotic notation: section 2.5]} \\ &= \boldsymbol{\theta(n \log n)} \end{aligned}$$

So, time complexity to find the mode using presort is $\mathbf{f(n) = \theta(n \log n)}$

Note: This time complexity is much superior when compared with brute-force technique.

6.3 Balanced Search Trees

Binary search trees are very important data structures. Now, let us see "What is a binary search tree?"

Definition: A binary search tree is a binary tree in which for each node say **x** in the tree, elements in the left-subtree are less than **info(x)** and elements in the right subtree are greater or equal to **info(x)**. Every node in the tree should satisfy this condition, if left subtree or right subtree exists. Fig.6.3.1 shows some of the binary search trees.

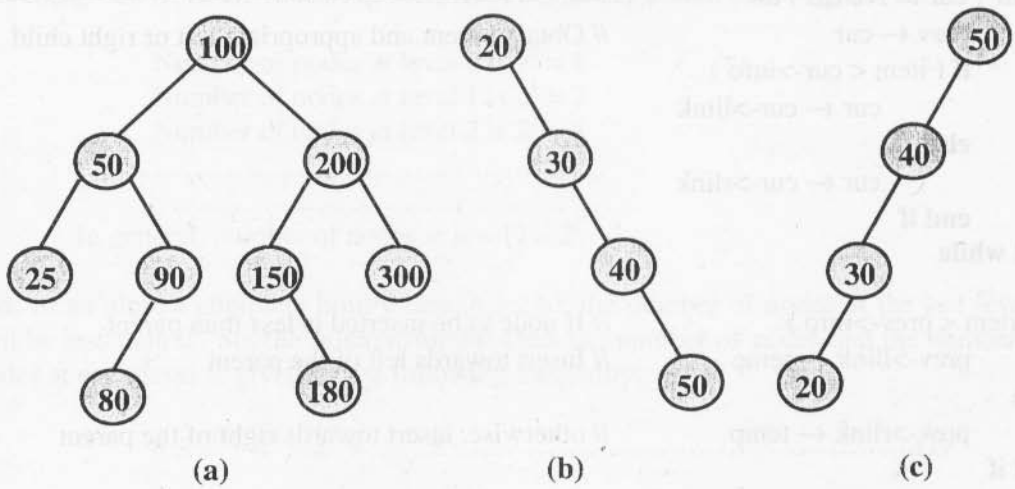


Figure 6.3.1 Binary search trees

6.12 Transform and Conquer

The algorithm to create a binary search tree is shown below:

```
Algorithm insert_item(item, root)
// Purpose: To insert an item into binary search tree. By repeatedly calling this function,
            a binary search tree can be created.

// Input:
//      item – the element to be inserted
//      root – the binary search tree into which the item has to be inserted

// Output:
//      root – points to the binary search tree created

temp ← getnode()                    // Obtain new node from the availability list
temp(info) ← item                   // Copy appropriate data
temp(llink) ← NULL
temp(rlink) ← NULL

if ( root = NULL ) return temp;     // Insert a node for the first time

//find the position to insert
prev ← NULL
cur ← root

while ( cur != NULL ) do
    prev ← cur                      // Obtain parent and appropriate left or right child
    if ( item < cur->info )
        cur ← cur->llink
    else
        cur ← cur->rlink
    end if
end while

if ( item < prev->info )            // If node to be inserted is less than parent
    prev->llink ← temp              // Insert towards left of the parent
else
    prev->rlink ← temp              // otherwise, insert towards right of the parent
end if

return root                         // Return the root of the tree
```

Example 1: Creating a binary search tree by inserting the items:

(a) 50, 35, 70, 25, 40, 60 and 85

(b) 50, 40, 60, 35, 45, 55 , 70, 25 and 36

Solution: The binary search trees for the first input order is shown in figure 6.3.2(a) and binary search tree for the second input order is shown in figure 6.3.2 (b).

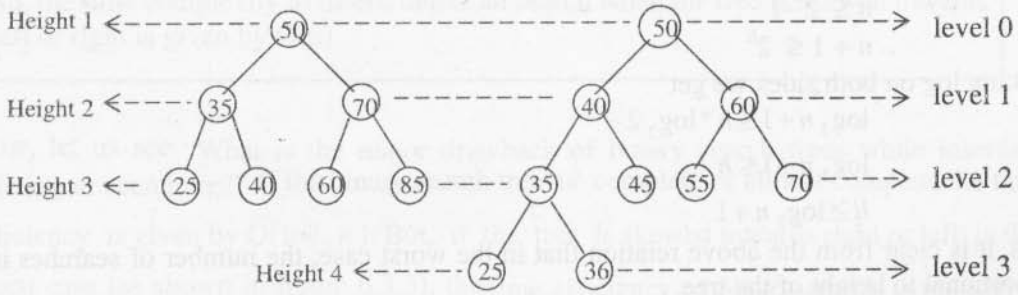


Figure 6.3.2 A tree showing the height and levels

When we want to insert an item or delete an item or to search for an item, the comparison starts from the root node and ends at one of the leaves in the worst case. The search is efficient, if the tree is complete binary tree as shown in figure 6.3.2.a and almost complete binary tree as shown in figure 6.3.2.b. The number of comparisons required in the worst case (if the tree is complete) is equal to the height of the tree as shown below:

Analysis Let us find the time efficiency of binary search tree for inserting, deleting and searching. Observe the following facts from the binary search tee of fig.6.3.2.a

Number of nodes at level 0 is $2^0 = 1$
Number of nodes at level 1 is $2^1 = 2$
Number of nodes at level 2 is $2^2 = 4$
.....
.....
In general, number of nodes at level i = 2^i.

But, in an almost complete binary tree (6.3.2.b), the number of nodes at the last level *i* will be less than 2^i. So, the relationship between the number of nodes and the number of nodes at each level is given by the following inequality:

$$n \le 2^0 + 2^1 + 2^2 + \ldots\ldots 2^i \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

The sum of all the terms in a geometric progression is given by

$$n \le a\left(\frac{r^n - 1}{r - 1}\right)$$ It is a geometric progression with a = 1, r = 2, n = i + 1

$$n \le 1\left(\frac{2^{i+1} - 1}{2 - 1}\right)$$

$$n \le 2^{i+1} - 1$$ Height(T) = Maximum level + 1

$$n \le 2^h - 1$$

$$n + 1 \le 2^h$$

Taking log on both sides we get

$$\log_2 n + 1 \le h * \log_2 2$$

$$\log_2 n + 1 \le h$$

$$h \ge \log_2 n + 1$$

Thus, it is clear from the above relation that in the worst case, the number of searches is proportional to height of the tree.

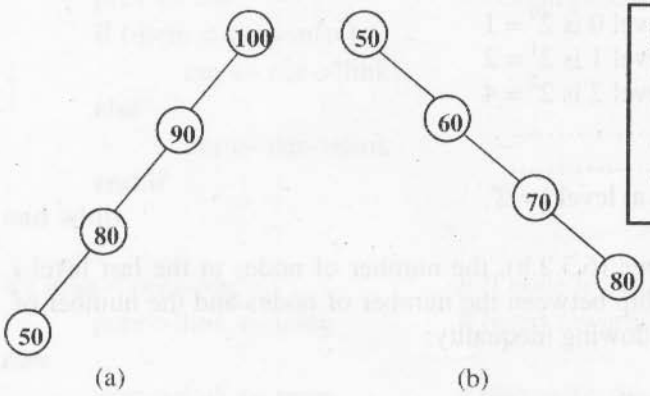
So, the time complexity to insert, delete an search when the tree is complete is given by $O(\log_2 n)$ [Neglecting all lower order terms and constants]

Example 2: Creating a binary search tree by inserting the items:

(a) 100, 90, 80 and 50

(b) 50, 60, 70 and 80

Solution: The binary search trees for the first input order is shown in figure 6.3.3(a) and binary search tree for the second input order is shown in figure 6.3.3 (b).



Note: If each node in the tree is inserted towards left of a node as shown in figure (a) or vice versa as shown in figure (b), such trees are called skewed binary trees

Figure 6.3.3 Skewed binary tees

Analysis Even though above trees are binary search trees, observe that the tree may be skewed either towards left or right. If the binary trees are skewed on one side either towards left or right, the number of comparisons required is again proportional to height of the tree which in this case is n. So, the time complexity in the worst case for binary search tree if skewed to left side or right side is given by O(n).

So, the time complexity to insert, delete an search when the tree is skewed towards left or right is given by O(n)

Now, let us see "What is the major drawback of binary search trees while inserting, deleting or searching?" If the binary search tree is complete or almost complete the time efficiency is given by $O(\log_2 n)$. But, if the tree is skewed towards right or left, in the worst case (as shown in figure 6.3.3), the time efficiency of binary search is reduced to O(n). So, the time efficiency of skewed tree and linear search is same. This is a disadvantage. This disadvantage is overcome using balanced binary search trees.

Now, let us see "What is balancing a binary search tree?" To avoid the worst case time complexity of O(n), each time a node is inserted or deleted, the tree should be re-ordered so that the time complexity is always $\log_2 n$ and there by number of comparisons are reduced. This process is called *balancing a binary search tree.*

Now, we shall see "What are the various approaches used to balance a tree?" The tree can be balanced using two approaches namely:

- Instance simplification
- Representation change

- Instance simplification: Here, the moment the tree is unbalanced, at that instant the tree is transformed so that the tree is balanced. For example, AVL trees, red-black trees etc
- Representation change: The binary search tree has one item in each node. Here, instead of only one item in each node, we allow more than one element in each node. This is representation change. For example, 2-3 trees, B-trees etc., allow more than one item in each node.

In this text, we limit our discussion to AVL trees and 2-3 trees.

6.3.1 AVL Tree

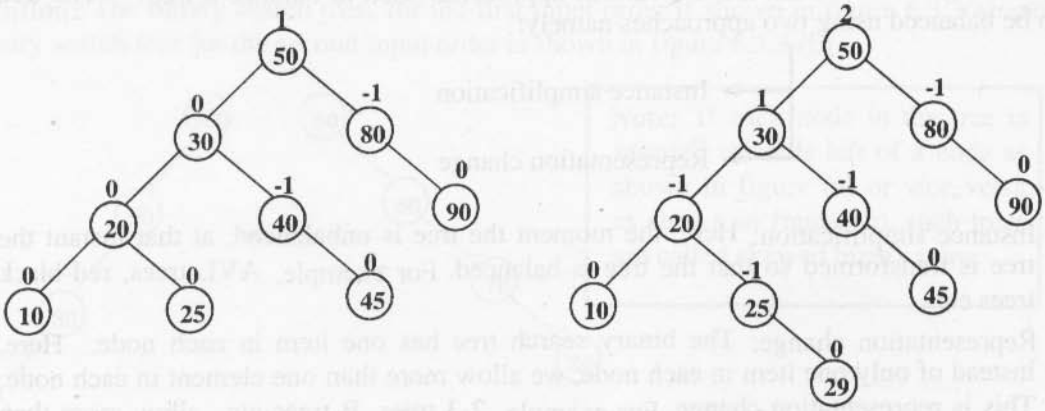
AVL Tree was implemented by two Russian scientists namely: G.M. Adelson-Velsky and E.M. Landis and the first letters from each word of their names are taken to name the balanced tree and we call them AVL trees. Now, let us see "What is an AVL tree?"

Definition: An *AVL* tree is an ordered binary search tree in which the heights of the two subtrees of every node differ maximum by 1 i.e., the height of the left subtree minus the height of the right subtree can be 0 or 1 or -1. If this condition is satisfied by each node in the binary tree, then the tree is called *AVL* tree. In an AVL tree, each node is associated with a *balance factor* which is the height of the left subtree minus the height of the right subtree and is given by

Balance factor = Height(left subtree) – Height (right subtree)

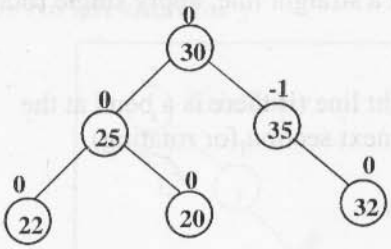
- **Balance factor is 0:** If the height of the left subtree and height of the right subtree are same.
- **Balance factor is 1:** If the height of the left subtree is 1 more than the height of the right subtree
- **Balance factor is -1:** If the height of the left subtree is 1 less than the height of the right subtree

For example, consider the following binary search trees. Some trees are AVL trees and some are not.

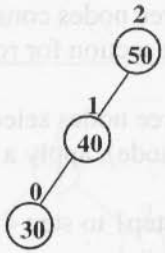


(a) AVL Tree

(b) Not an AVL Tree because it is not balanced (Balance factor of 50 is 2)



(c) Not an AVL Tree (It is not a binary search Tree)
(20 is inserted towards right of 25)



(d) not an AVL tree
(Balance factor of 50 is 2)

6.3.1.1 Creating an AVL tree

Now, the question is "How to create AVL tree?" Creating a tree is nothing but repeated insertion of items into binary search tree. But, after adding each item into the tree, a check must be made to ensure that the balance factor of each node is -1 or 0 or 1. If so, then the tree is balanced and the tree is an AVL tree. If the balance factor of any node is not either -1 or 0 or 1, it is necessary to rearrange the nodes in the tree so that the balance factor of all the nodes will either be -1 or 0 or 1. The algorithm to insert an item into the tree is shown below:

Algorithm AVL_Insert(item, root)

// **Purpose**: To insert an item into AVL tree. The tree remains balanced after insertion

// **Input**:
// item – the item to be inserted into the tree
// root – the root of the AVL tree

// **Output**:
// balanced tree with time efficiency of $O(\log_2 n)$.

Step 1: Insert an item at the leaf level of an AVL tree as we insert an item into binary search tree

Step 2: Trace back wards from the node inserted, towards the root and compute the balance factors along the path. If the balance factor of each node in this path is 0 or 1 or -1, the tree is balanced and insertion is over.

Step 3: If we find a node whose balance factor is other than -1 or 0 or 1, consider this node and the two nodes below it in this path.

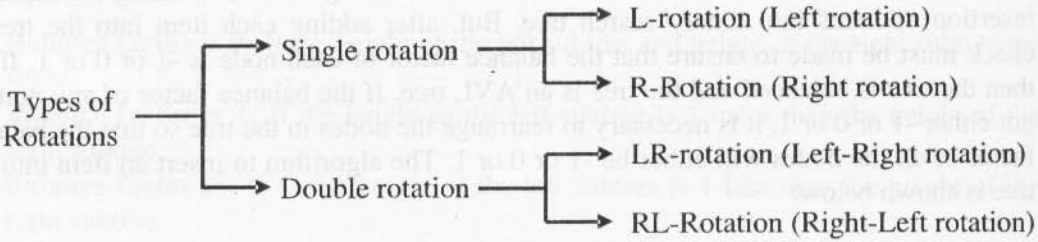
Step 4: If the three nodes considered in step 3 are in a straight line, apply single rotation. (See next section for rotating)

Step 5: If the three nodes selected are not in a straight line (if there is a bend at the second node), apply a double rotation. (See next section for rotating)

Step 6: Repeat step1 to step 4 for all the given items

Step 7: Terminate

Once an item is inserted into the tree, the tree may be unbalanced. In such case, it is necessary to balance the tree. Balancing of AVL tree is done using rotations. Now, let us see "What are the different types of rotations?" The various types of rotations are shown below:



Now, let us see "When rotation is required?" After inserting a node, trace back ward towards the root and compute the balance factors along the path. If we find a node whose balance factor is other than -1 or 0 or 1, then rotation is required. The procedure to identify L-Rotation and R-Rotation is shown below:

Step 1: Identify the first node where balance factor is not 0 or -1 or 1 in the path while moving from node inserted towards root.

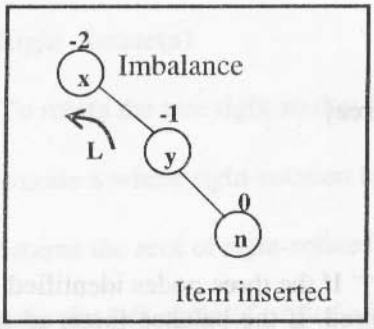
Step 2: Identify two other nodes below the node where imbalance occurs in the path.

After these two steps are over we can easily identify which type of rotation is required.

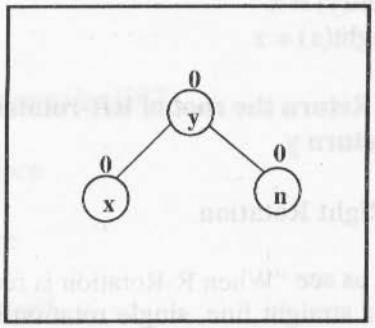
6.3.1.2 Left Rotation

Now, let us see "When L-Rotation is required?" If the three nodes identified in step 1 and 2 are in a straight line, single rotation is required. If the balance factor of a node where imbalance occurs is -2, then the tree is heavy towards right. So, to balance it we rotate left. This is called L-Rotation.

Case 1: No left child for y

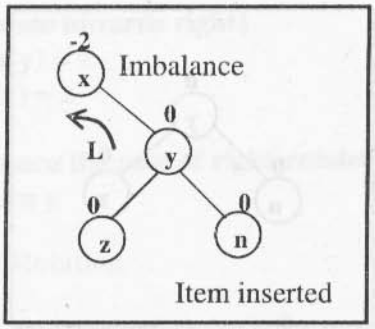


Before left rotation

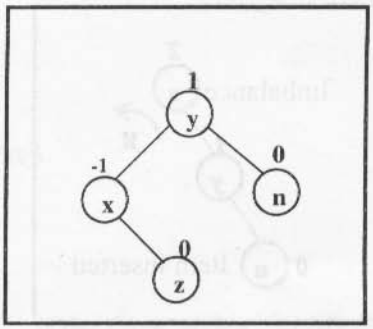


After left rotation

Case 2: There is a left child for y



Before left rotation



After left rotation

The algorithm to left-rotate at the specified node is shown below:

Algorithm Left_Rotate(x)

// **Purpose**: To rotate the tree left so that the tree remains BST

// **Input**: The node x where left-rotation takes place

// **Output**: Returns the root of left-rotated subtree

Step 1: [Obtain the right child of node to be rotated]

y = right(x)

Step 2: [Obtain the left child of y]

z = left(y)

Step 3: [Rotate towards left]

left(y) = x
right(x) = z

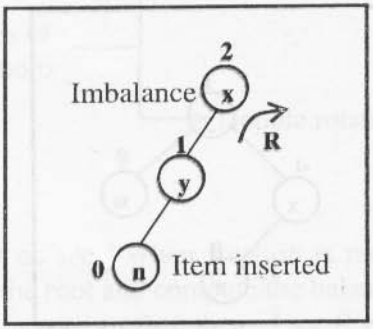
Step 4: [Return the root of left-rotated subtree]

return y

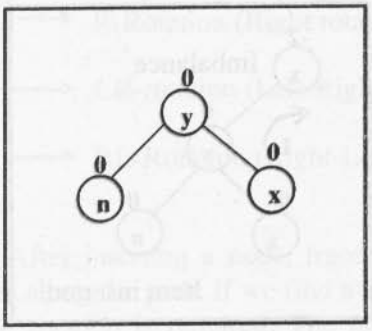
6.3.1.3 Right Rotation

Now, let us see "When R-Rotation is required?" **If the three nodes identified in step 1 and 2 are in a straight line, single rotation is required. If the balance factor of a node where imbalance occurs is 2, then the tree is heavy towards left. So, to balance it we rotate right.** This is called R-Rotation.

Case 1: No right child for y

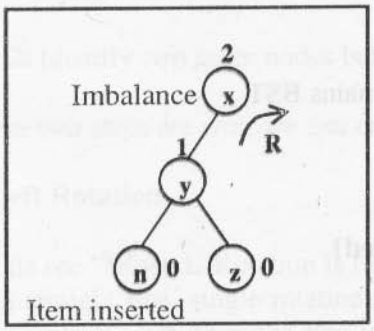


Before right rotation

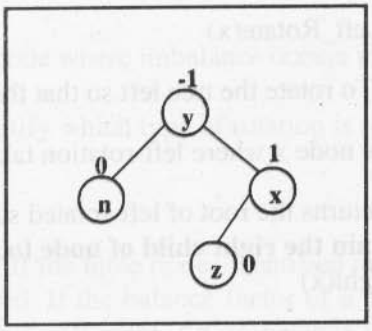


After right rotation

Case 2: There is right child for y



Before right rotation



After right rotation

The algorithm to right-rotate at the specified node is shown below:

Algorithm Right_Rotate(x)

// Purpose: To rotate the tree right so that the tree remains BST

// Input: The node x where right-rotation takes place

// Output: Returns the root of right-rotated subtree

Step 1: [Obtain the left child of node to be rotated]
y = left(x)

Step 2: [Obtain the right child of y]
z = right(y)

Step 3: [Rotate towards right]
right(y) = x
left(x) = z

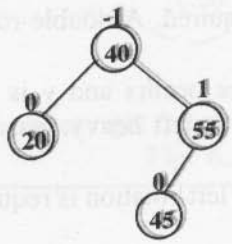
Step 4: [Return the root of right-rotated subtree]
return y

6.3.1.4 L-R Rotation

Now, let us see "When L-R Rotation is required?" If the three nodes identified are not in a straight line, then double rotation is required. A double-rotation is the combination of two single-rotations as shown below:

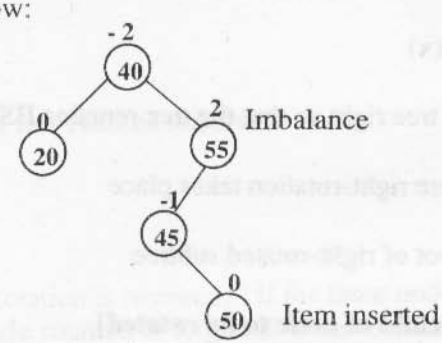
- Assume x is a node where imbalance occurs and y is its child in the path. If the balance factor of y is -1, this subtree is right heavy. So, rotate left at y. Attach the parent of resulting subtree to x
- Second rotation is required at x. Here, right-rotation is required.

Example 1: Consider the following tree. Insert the item 50

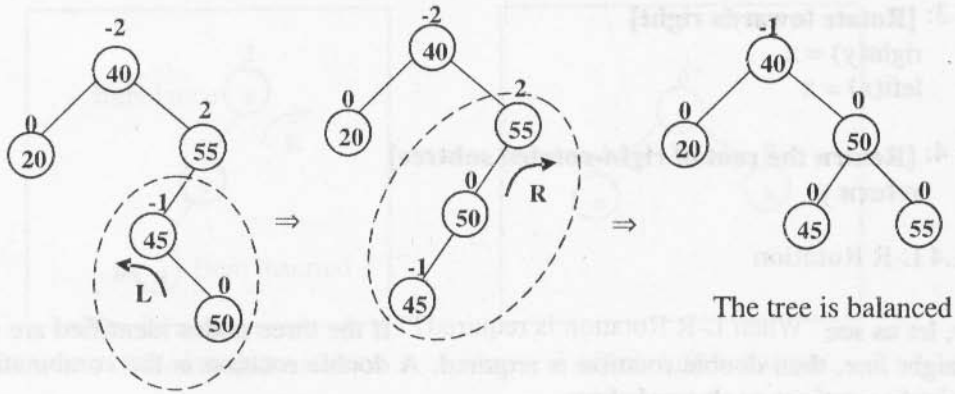


The tree is balanced

Solution: The item 50 is inserted into the tree to the right of 45 and the balance factors are computed as shown below:



Observe that the nodes 55, 45 and 50 are not in a straight line. So, two rotations are required. The first rotation at 45 as shown in figure 6.31.a. Second rotation as shown in figure 6.31.b and the resulting tree is shown figure 6.31.c



(a) Before left rotation (b) After left rotation at 45 (c) After right rotation at 55

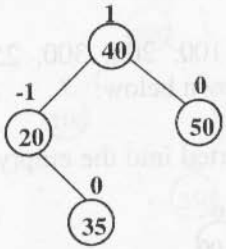
Figure 6.3.1

6.3.1.5 R-L Rotation

Now, let us see "When R-L Rotation is required?" If the three nodes identified are not in a straight line, then double rotation is required. A double-rotation is the combination of two single-rotations as shown below:

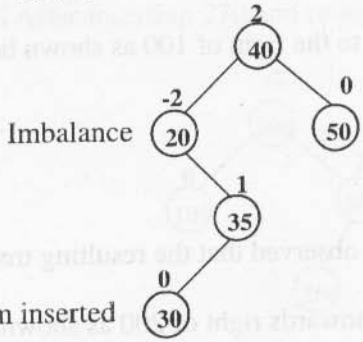
- Assume x is a node where imbalance occurs and y is its child in the path. If the balance factor of y is 1, this subtree is left heavy. So, rotate right at y. Attach the parent of resulting subtree to x
- Second rotation is required at x. Here, left-rotation is required.

Example 1: Consider the following tree. Insert the item 30

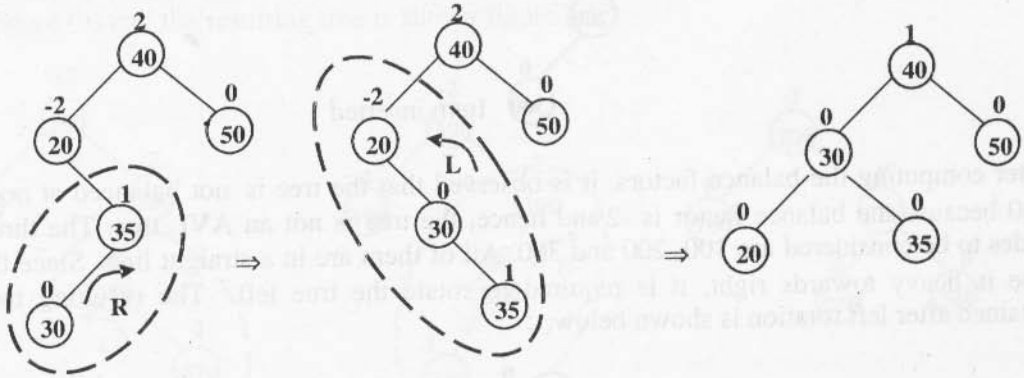


The tree is balanced

Solution: The item 30 is inserted into the tree to the left of 35 and the balance factors are computed as shown below:



Observe that the nodes 20, 35 and 30 are not in a straight line. So, two rotations are required. The first rotation at 35 as shown in figure 6.3.2.a. Second rotation as shown in figure 6.3.2.b and the resulting tree is shown figure 6.3.2.c.



(a) Before right rotation (b) After right rotation at 35 (d) After left rotation at 20

Fig. 6.3.2

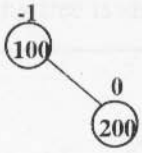
Example 2: Construct an AVL tree by inserting the elements 100, 200 300, 250, 270, 70 and 40 successively starting from an empty tree

Solution: An AVL tree for the items 100, 200, 300, 250, 270, 70 and 40 can be constructed by inserting one by one as shown below:

Step 1: Item = 100. The item 100 is inserted into the empty tree and the resulting tree is shown below:

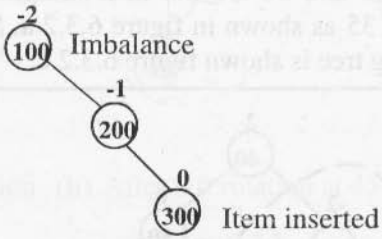
Balance factor of 100 is 0. Hence, the tree is balanced and resulting tree is AVL tree.

Step 2: Item = 200. Item 200 is inserted to the right of 100 as shown below:

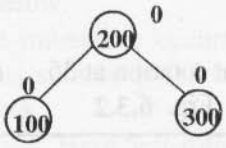


After computing the balance factors, it is observed that the resulting tree is an AVL tree.

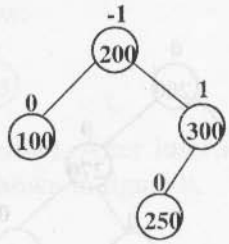
Step 3: Item = 300. Item 300 is inserted towards right of 200 as shown below:



After computing the balance factors, it is observed that the tree is not balanced at node 100 because the balance factor is -2 and hence, the tree is not an AVL tree. The three nodes to be considered are 100, 200 and 300. All of them are in a straight line. Since the tree is heavy towards right, it is required to rotate the tree left. The resulting tree obtained after left rotation is shown below:

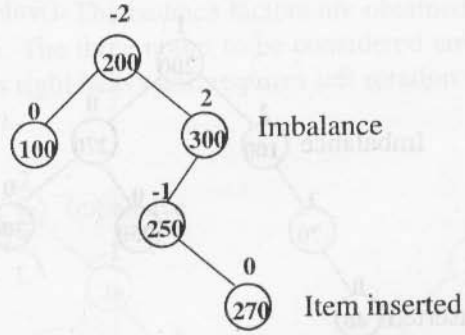


Step 4: Item = 250. After inserting 250 and re-computing the balance factors, the tree is shown below:

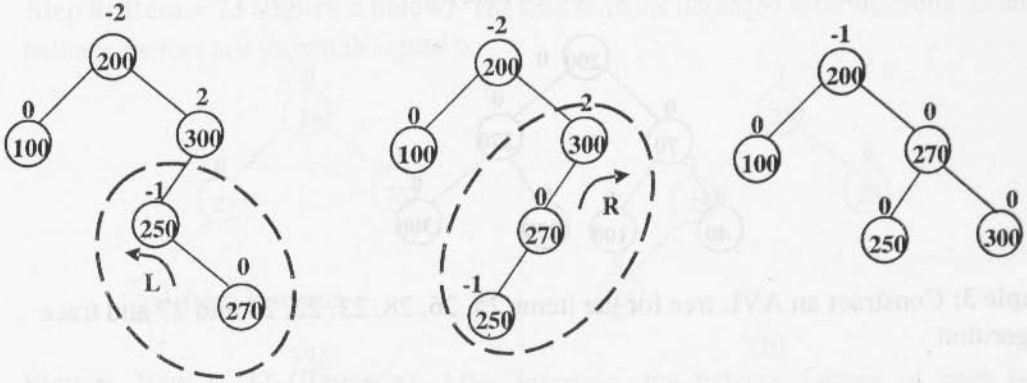


Since the balance factor of each node is either -1 or 0 or 1, the tree is balanced.

Step 5: Item = 270. After inserting 270 and re-computing the balance factors, the tree is shown below:

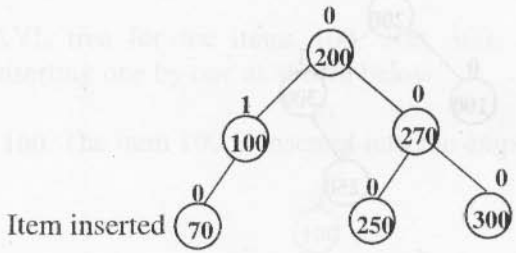


Observe that the nodes 300, 250 and 270 are not in a straight line. So, two rotations are required. The first rotation at 250 as shown in figure (a). Second rotation at 300 as shown in figure (b) and the resulting tree is shown figure (c)



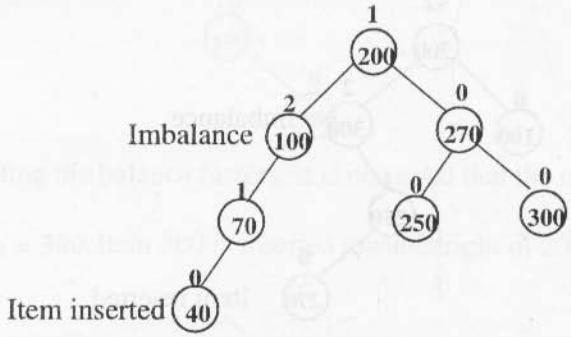
(a) Before left rotation **(b)** After left rotation at 250 **(c)** After right rotation at 300

Step 6: Item = 70. After inserting 70 and re-computing the balance factors, the tree is shown below:

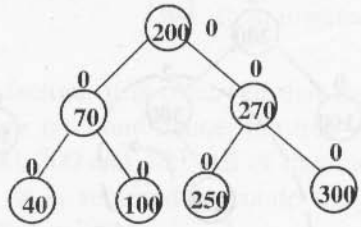


The tree is balanced since the balance factor of each node is either 0 or 1 or -1.

Step 7: Item = 40. After inserting 40 and re-computing the balance factors, the tree is shown below:



The three nodes to be considered are 100, 70 and 40. All of them are in a straight line. Since the tree is heavy towards left, it is required to rotate the tree right. The resulting tree obtained after right rotation at 100 is shown below:

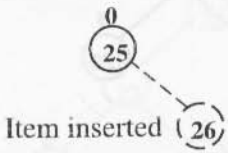


Example 3: Construct an AVL tree for the items 25, 26, 28, 23, 22, 24 and 27 and trace the algorithm

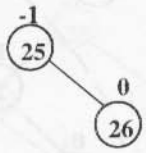
Solution: An AVL tree for the items 25, 26, 28, 23, 22, 24 and 27 can be constructed by inserting one by one as shown below:

Step 1: Item = 25. Since it is inserted into an empty, the tree is balanced with balance factor 0 as shown in figure below:

Step 2: Item = 26 (figure a below). After insertion, the tree remains balanced and the balance factor of each node is shown in figure b.

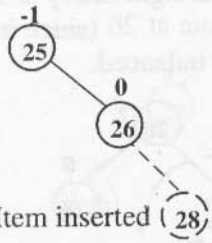


(a)

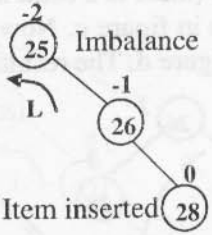


(b)

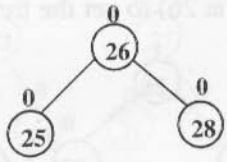
Step 3: **Item = 28 (figure a below).** The balance factors are obtained as shown in figure b. Imbalance occurs at node 25. The three nodes to be considered are 25, 26 and 28. All are in a straight line. The tree is right heavy and requires left rotation. After left rotation, the resulting tree is shown in (c).



(a)

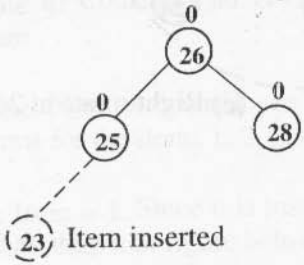


(b) Before left rotate

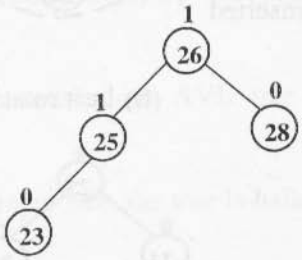


(c) After left rotation

Step 4: Item = 23 (figure a below). The tree remains balanced after inserting 23 and the balance factors are shown in figure b.



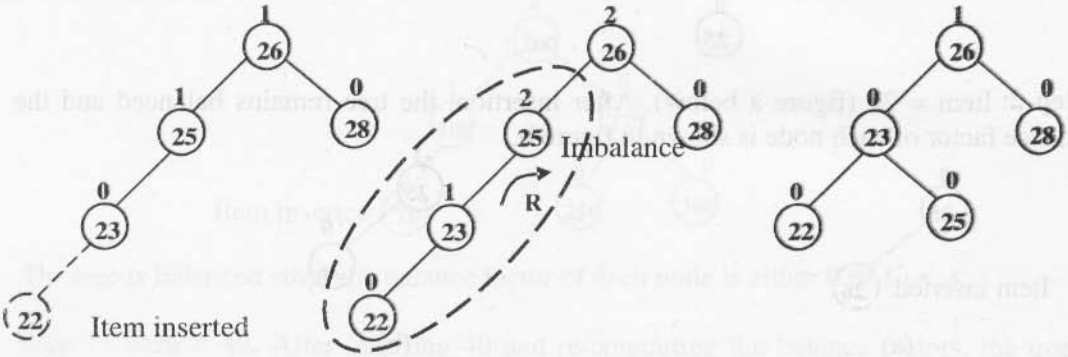
(a)



(b)

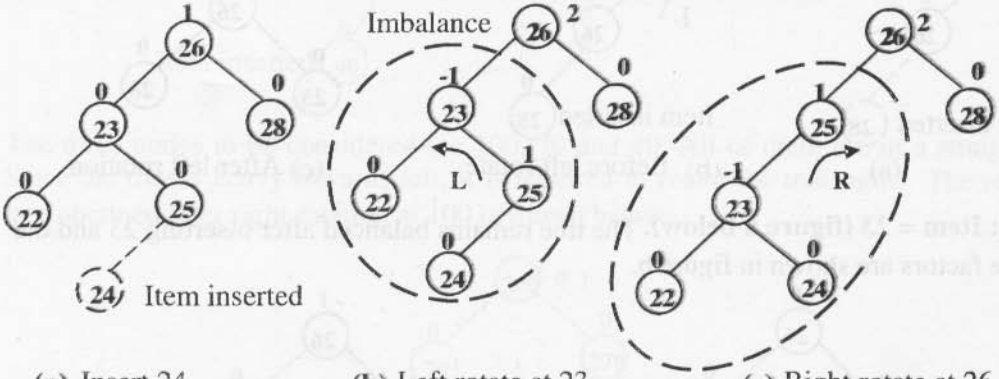
Step 5: **Item = 22 (figure a).** After insertion, the balance factors of each node is computed and the tree is shown in figure b. Since the tree is not balanced at 25 and the three nodes below it are in straight line a single rotation is required. The tree is left heavy

at 25 and so right rotate at 25 to balance it. The tree shown in figure c is obtained and the tree results in an AVL tree.

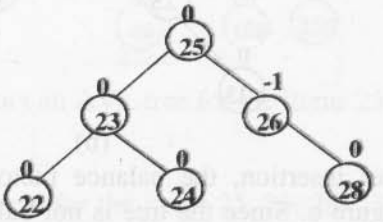


(a) Insert 22 (b) Before right rotate (c) After right rotation

Step 6: Item = 24(figure a). The balance factors of each node are shown in figure b and note that the tree is unbalanced at 26. The three nodes to be considered are 26, 23 and 25. Since these nodes are not in a straight line (there is a bend at 23) and is right heavy at 23, first left rotate at 23 to get the tree shown in figure c. Now, right rotate at 26 (since it is left heavy at 26) to get the tree shown in figure d. The resulting tree is balanced.

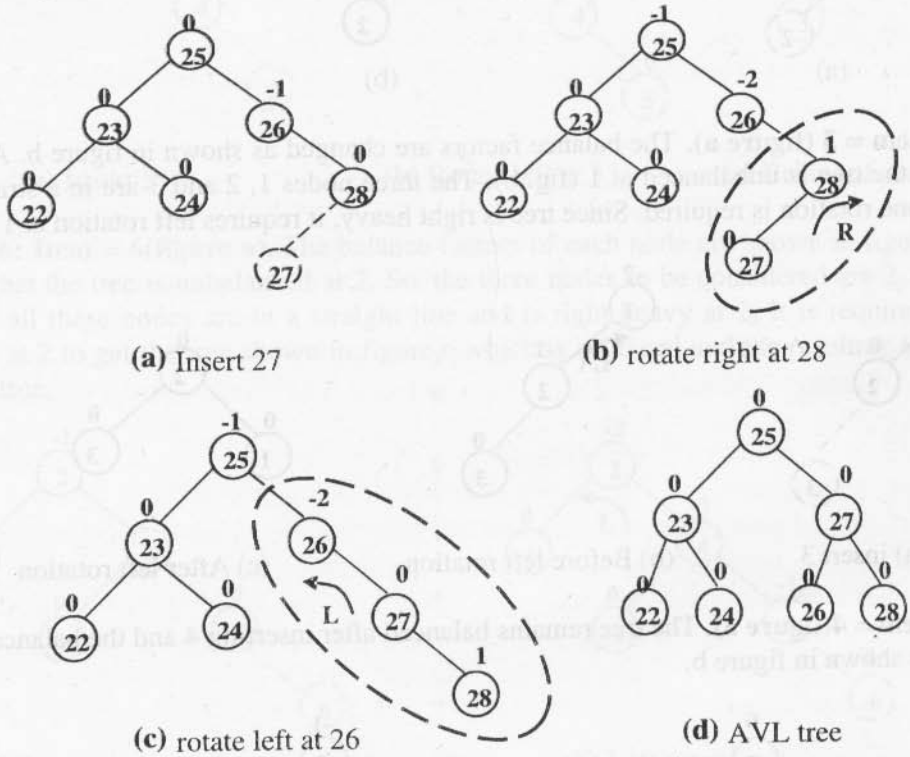


(a) Insert 24 (b) Left rotate at 23 (c) Right rotate at 26



(d) AVL tree (Balanced tree)

Step 7: Item = 27(figure a). The balance factors of each node are shown in figure b. Imbalance occurs at 26. The three nodes to be considered are 26, 28 and 27 and they are not in a straight line. So, a double rotation is required. The first rotation occurs at 28. Since at node 28, the tree is left heavy we do right rotate and the tree shown in figure c is obtained. Now, at node 26 (figure c), the tree is right heavy and needs a left rotation. The resulting tree shown in figure d is an AVL tree.



(a) Insert 27 **(b)** rotate right at 28

(c) rotate left at 26 **(d)** AVL tree

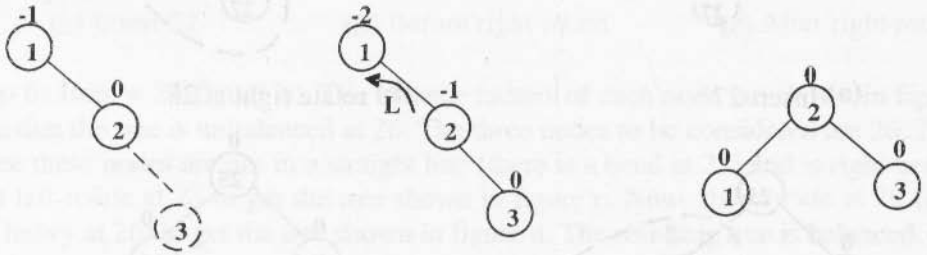
Example 4: Construct an AVL tree for the items 1, 2, 3, 4, 5, and 6 and trace the algorithm

Solution: Now, we shall see the way to construct an AVL tree using the above algorithms for the items 1, 2, 3, 4, 5 and 6.

Step 1: Item = 1. Since it is inserted into an empty tree, the tree is balanced with balance factor 0 as shown in figure below:

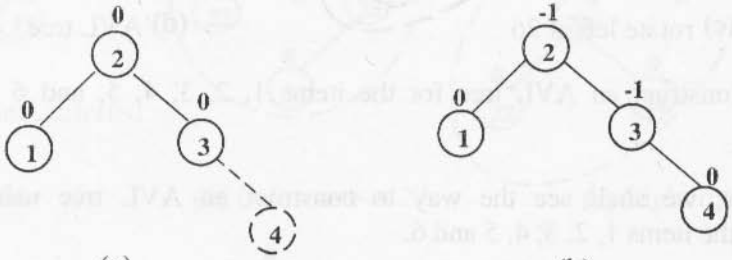
Step 2: Item = 2 (figure a). It has to be inserted towards right of 1. After insertion, the tree remains balanced and the balance factor of each node is shown in figure b.

Step 3: Item = 3 (figure a). The balance factors are changed as shown in figure b. After inserting, the tree is unbalanced at 1 (fig. b). The three nodes 1, 2 and 3 are in a straight line. So, one rotation is required. Since tree is right heavy, it requires left rotation at 1 (b).



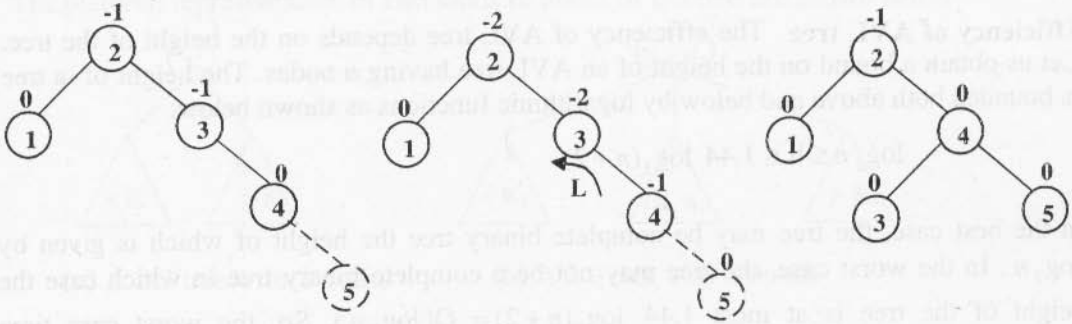
(a) insert 3 (b) Before left rotation (c) After left rotation

Step 4: Item = 4(figure a). The tree remains balanced after inserting 4 and the balance factors are shown in figure b.



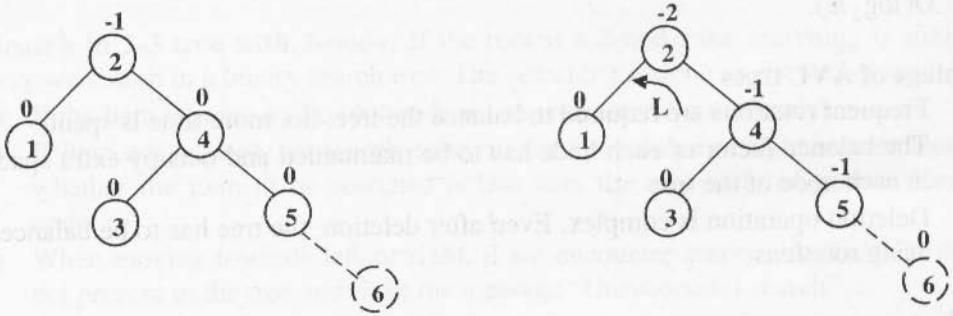
(a) (b)

Step 5: Item to be inserted is 5 (figure a) and has to be inserted towards right of 4. After insertion, the balance factors of each node is computed and the tree is shown in figure b. Since the tree is not balanced at 3 and the three nodes below it are in straight line, a single rotation is required. The tree is right heavy at 3 and so left rotate at 3 to balance it. The tree shown in figure c is obtained and the tree results in an AVL tree.

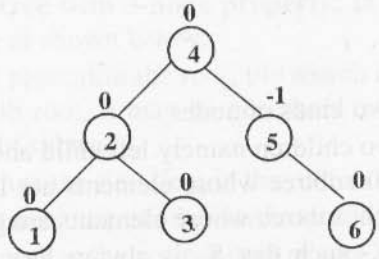


(a) Insert 5 (b) Rotate left at 3 (c) Balanced tree

Step 6: Item = 6(figure a). The balance factors of each node are shown in figure b and note that the tree is unbalanced at 2. So, the three nodes to be considered are 2, 4 and 5. Since all these nodes are in a straight line and is right heavy at 2, it is required to left rotate at 2 to get the tree shown in figure c which is balanced and the resulting tree is an AVL tree.



(a) Insert 6 (b) Left rotate at 2



(c) AVL tree (Balanced tree)

Thus, the AVL tree is obtained by successively inserting into a balanced tree.

Efficiency of AVL tree The efficiency of AVL tree depends on the height of the tree. Let us obtain a bound on the height of an AVL tree having n nodes. The height of a tree is bounded both above and below by logarithmic functions as shown below:

$$\log_2 n \leq h \leq 1.44 \log_2(n+2)$$

In the best case, the tree may be complete binary tree the height of which is given by $\log_2 n$. In the worst case, the tree may not be a complete binary tree in which case the height of the tree is at most $1.44 \log_2(n+2) = O(\log_2 n)$. So, the worst case time complexity of searching an inserting an element into AVL tree is given by $\theta(\log_2 n)$. Observe that the search time required by AVL tree is almost same as search time required by a binary search tree. Note that even though deleting an element from AVL tree is most complex, the time complexity to delete an element is also given by $\theta(\log_2 n)$.

Advantage of AVL trees

- The search time, insertion time and deletion time efficiency will be always $O(\log_2 n)$.

Disadvantage of AVL trees

- Frequent rotations are required to balance the tree. So, more time is spent
- The balance factor of each node has to be maintained and occupy extra space in each node of the tree
- Deletion operation is complex. Even after deletion, the tree has to be balanced using rotations.

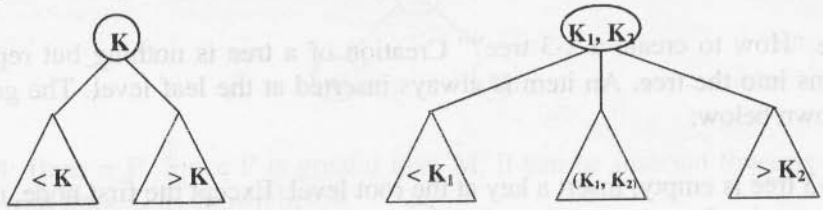
6.4 2-3 Tree

Now, let us see "What is a 2-3 tree?"

Definition: A ***2-3 tree*** is a tree that has two kinds of nodes

- ***2-node***: contains a single key with two children namely left child and right child. The left child serves as the root of the left subtree whose elements are less than key and right child serves as the root of the right subtree whose elements are greater than key
- ***3-node***: contains two keys K_1 and K_2 such that K_1 is always less than K_2 and has three children namely left child, middle child and right child. The left child serves as the root of a subtree whose elements are less than K_1. The middle child serves as the root of a subtree whose elements are between K_1 and K_2. The right child serves as the root of a subtree whose elements are greater than K_2.

The pictorial representation of two kinds of nodes of 2-3 tree are shown below:



(a) A 2-3 tree with 2-nodes (b) A 2-3 tree with 3-nodes

Note: A 2-3 tree is always height balanced i.e., the height of a 2-3 tree with respect to all the leaves in a tree is same. In other words, the length of the path from root to any leaf in the tree is same.

6.4.1 Searching in a 2-3 Tree

Now, let us see "How to search for an item in a 2-3 tree?" Searching for an item always starts from the root.

Search in 2-3 tree with 2-node: If the root is a 2-node, the searching is similar to the way we search in a binary search tree. The searching is done as shown below:

- If the item is same as the root node we say the search is successful.
- If item not present, we search either in the left subtree or right subtree based on the whether the item to be searched is less than the root node or greater than the root node.
- When moving towards left or right, if we encounter empty subtree, then the item is not present in the tree and flash the message "Unsuccessful search".

Search in a 2-3 tree with 3-node property: In a 3-node, each node has two items. The searching is done as shown below:

- If key item is present in the root, the search is successful and return. If key item does not match with root, it may be necessary to search either the left subtree, or middle subtree or right subtree.
- If *key* is less than the first item in the root, continue searching the left subtree.
- If *key* is in between the two items of root, continue searching in the middle subtree.
- If *key* is greater than the second item of root, continue searching in the right subtree.
- When we keep updating towards left or middle or right, if we encounter an empty subtree, it means that the *key* is not present in the 2-3 tree and the message "Unsuccessful search" can be flashed. Thus, searching takes place in a 2-3 tree.

6.4.2 Creating a 2-3 Tree

Now, let us see "How to create a 2-3 tree?" Creation of a tree is nothing but repeated insertion of items into the tree. An item is always inserted at the leaf level. The general procedure is shown below:

Step 1: If the 2-3 tree is empty, insert a key at the root level. Except the first node, rest of the nodes are inserted only at the leaf level.

Step 2: By performing a search for the key to be inserted, appropriate leaf node is found.

Step 3: If the leaf node is a 2-node, compare the *key* item with the element already in the node. If *key* to be inserted is less than the item already present, insert towards left of old item; otherwise, insert towards right of the old item.

Step 4: If the leaf node is a 3-node, split the leaf into two parts: the smallest item in 3-node is placed in the first leaf, the largest item in 3-node is placed in the second leaf, where as the middle element is promoted to the place of the old 3-node's parent. If tree's root itself is the leaf node, in such situation, a new root is created to accept the middle key.

Step 5: If the promotion of a middle item to its parent leads to a 3-node, it may be required to split along the chain of leaf's ancestor. By repeating this procedure, it is possible to create a 2-3 tree.

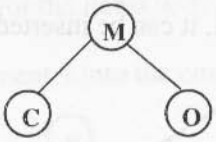
Example 1: Construct 2-3 tree for the list C, O, M, P, U, T, I, N, G

Step 1: **Item = C.** Insert this into the empty tree as shown below:

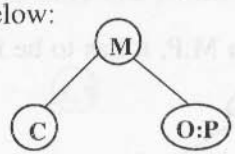
Step 2: **Item = O.** Since there is only one item in the root node, the item O is inserted along with C as shown below:

Step 3: **Item = M.** Since, M is greater than C and less than O, it is inserted between C and O as shown below:

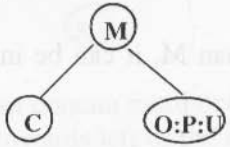
Since a node in a 2-3 tree cannot contain more than 2 items, we move the middle node to the parent position, inserting left item C towards left of the parent and right item O towards right of the parent as shown below:



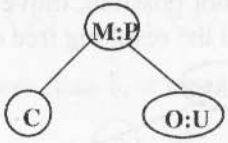
Step 4: **Item = P.** Since P is greater than M, it can be inserted towards right of M, but along with node O as shown below:



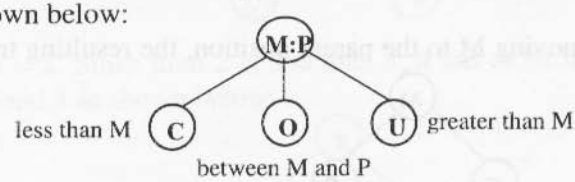
Step 5: **Item = U.** Since P is greater than M, it can be inserted towards right of M, but along with O:P as shown below:



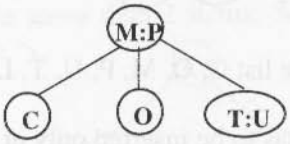
But, a node can not have more than 2 items. So, out of O:P:U, the middle item P is moved to parent position as shown below:



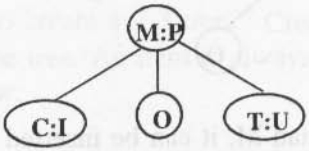
A node with two items except the leaf node should have 3 children. So, the node M:P should have three children namely left child with C, middle child with O and right child with U as shown below:



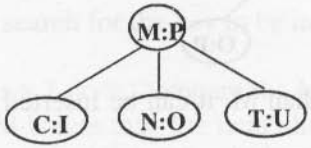
Step 6: **Item = T.** Since it is greater than P, it can be included along with U as shown below:



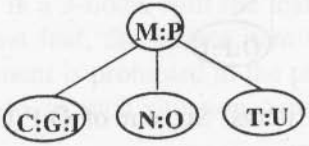
Step 7: Item = I. Since it is less than M, it can be inserted along with C as shown below:



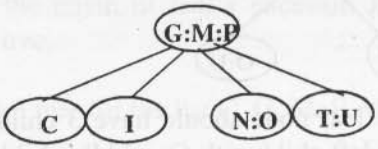
Step 8: Item = N. Since it is between M:P, it has to be inserted along with O as shown below:



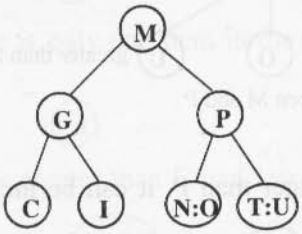
Step 9: Item = G. Since G is less than M, it can be inserted along with C:I as shown below:



Since more than 3 items in a node is not possible, move G to the parent position so that node C and I are the children for G and the resulting tree obtained is shown below:



Again in the node G:M:P, moving M to the parent position, the resulting tree obtained is shown below:



This is the final 2-3 tree obtained for the list C, O, M, P, U, T, I, N, G

Note: Each time an item is inserted, it has to be inserted only at the leaf level and later the tree can be modified so that the tree is always height balanced.

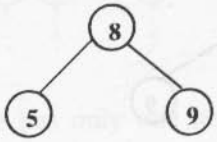
Example 2: Construct 2-3 tree for the items 9, 5, 8, 3, 2, 4 and 7

Step 1: Item = 9. Insert the element 9 into the empty tree as shown below:

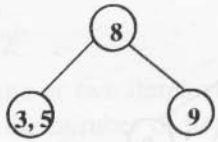
Step 2:Item = 5. Since there is only one item in the root node, the item 5 is inserted along with 9 as shown below:

Step 3: Item = 8. Since, 8 is greater than 5 and less than 9, it is inserted between 5 and 9 as shown below:

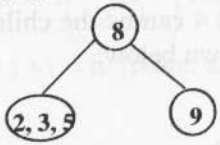
Since a node in a 2-3 tree can not contain more than 2 items, we move the middle node to the parent position, left item 5 towards left of the parent and right item 9 towards right of the parent as shown below:



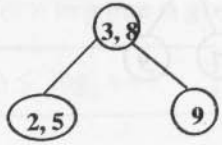
Step 4: Item = 3. Since 3 is less than 8, it can be inserted towards left of 8, but along with node 5 as shown below:



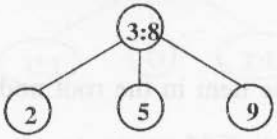
Step 5: Item = 2. Since item 2 is less than 8, it has to be inserted towards left of 8, but along with 3 and 5 as shown below:



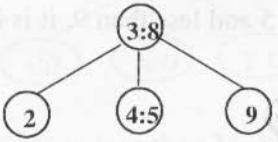
But, a node can not have more than 2 items. So, move the middle item 3 to its parent position as shown below:



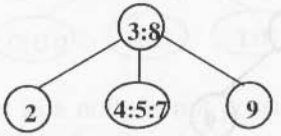
But, if a node has two items i.e., 3, 8 (except the leaf i.e., 2, 5), it should have 3 children: left child containing all the elements less than 3, middle child containing all the elements between 3 and 8, and finally the right child containing the elements greater than 8. So, the above tree can be modified as shown below:



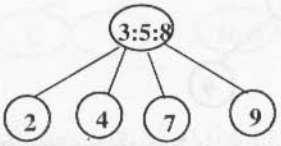
Step 6: Item = 4. Since, it is between 3 and 8, it can be inserted along with 5 as shown below:



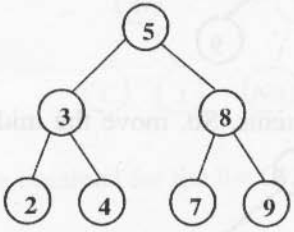
Step 7: Item = 7. Since, it is between 3 and 8, it can be inserted along with 4:5 as shown below:



Since a node can not have more than 2 items, move 5 to the parent position and the resulting tree is shown below:



Since, there are 3 items namely 3:5:8, it is required to move the middle element to the root. In such a situation, the items 2 and 4 can be the children of node 3 and the items 7 and 9 an be the children of node 8 as shown below:



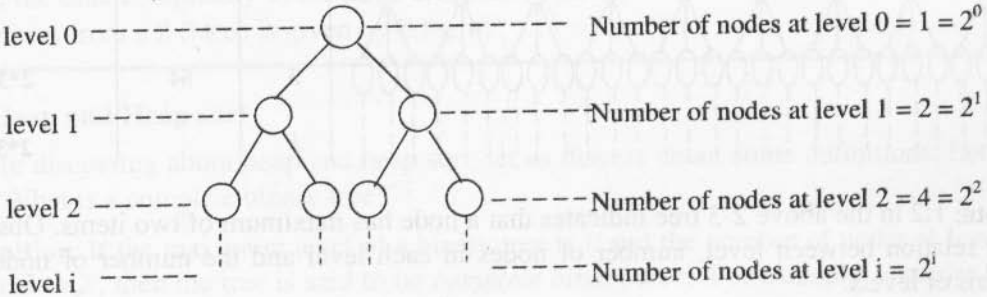
Thus, a 2-3 can be created.

Note: At any point after inserting an element, observe that the length from root node to any leaf in a tree is same and so, we say that a 2-3 tree is height balanced tree.

6.4.3 Analysis

Since a 2-3 tree is always height balanced, the efficiency of any operation depends only on the height of the tree. To find the time complexity, it is required to find the lower bound as well as the upper bound.

Case 1: To find the upper bound for 2-3 tree: Consider a 2-3 tree with each node exhibiting the property of 2-nodes (tree is similar to a binary search tree)



Let us assume that each node has only one item. Now, the relation between the total number of items n and the number of nodes occurring in each level from level 0 to level i is given by the following inequality:

$$n = 2^0 + 2^1 + 2^2 + \ldots\ldots 2^i$$

But, the leaf nodes may have one or two items and the height still remains same. In such situation, the relation between the number of items and the number of nodes is given by the inequality:

$$n \geq 2^0 + 2^1 + 2^2 + \ldots\ldots 2^i = 1\left(\frac{2^{i+1}-1}{2-1}\right) \text{ using the formula } S = a\left(\frac{r^n-1}{r-1}\right)$$

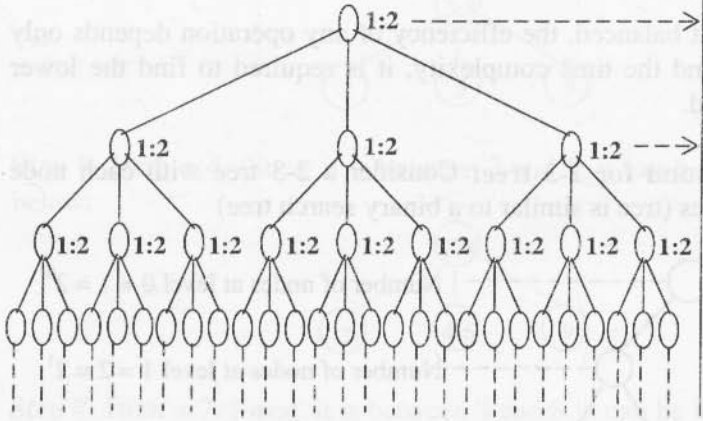
i.e., $n \geq 2^{i+1} - 1 = 2^h - 1$ where $i + 1 = h$ (**Note**: last level + 1 = height of the tree)

i.e., $n + 1 \geq 2^h$

So, taking log on both sides we get $\log_2 n + 1 \geq h$. So, the relation between the height of the tree and the number of nodes n in a tree is given by

$$\boxed{h \leq \log_2 n + 1} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)$$

Case 2: To find the lower bound for 2-3 tree: Consider a 2-3 tree with each node exhibiting the property of 3-nodes with each node having maximum of two items as shown below:



level	number of items	Items in terms of levels
0	2	$2*3^0$
1	6	$2*3^1$
2	18	$2*3^2$
3	54	$2*3^3$
i		$2*3^i$

Note: 1:2 in the above 2-3 tree indicates that a node has maximum of two items. Observe the relation between level, number of nodes in each level and the number of nodes in terms of levels.

The total number of items in each level by assuming each node has 2 items is given by the relation:

$$n = 2*3^0 + 2*3^1 + 2*3^2 + 2*3^3 + \ldots\ldots\ldots\ldots 2*3^i$$

In case of some nodes having less than 2-items, then the relation is given by:

$$n \le 2*3^0 + 2*3^1 + 2*3^2 + 2*3^3 + \ldots\ldots\ldots\ldots 2*3^i$$

$$n \le 2\,(3^0 + 3^1 + 3^2 + 3^3 + \ldots\ldots\ldots\ldots 3^i)$$

$$n \le 2*1\left(\frac{3^{i+1}-1}{3-1}\right) \text{ using the formula } S = a\left(\frac{r^n-1}{r-1}\right)$$

$$n \le 3^{i+1} - 1$$

$$n+1 \le 3^{i+1}$$

$$n+1 \le 3^h \qquad \text{(Because, height of the tree is total number of levels + 1)}$$

Taking *log* on sides we have,

$$\log_3 (n+1) \le h*\log_3 3$$

i.e., $h \ge \log_3 (n+1)$

So, the relation between height of the tree and the number of nodes n in a tree is given by

$$\boxed{h \geq \log_3(n+1)} \quad \text{.....(2)}$$

It is clear from equation (1) and (2) that

$$\boxed{\log_3(n+1) \leq h \leq \log_2(n+1)}$$

By neglecting the lower order terms and constants we have

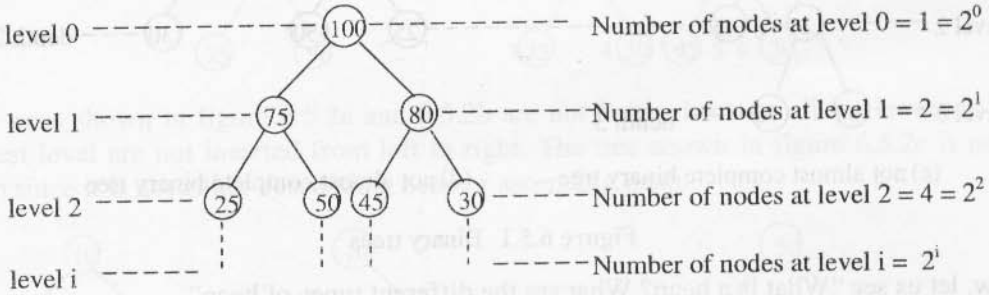
$$h \approx \log n$$

So, the time complexity to insert an element, to search for an element and to delete an element from a 2-3 tree is given by $\theta(\log n)$

6.5 Heap and Heap sort

Before discussing about heap and heap sort, let us discuss about some definitions. Let us see "What is a complete binary tree?"

Definition: If the maximum level of a binary tree is i, and the number of nodes at level i is equal to 2^i, then the tree is said to be ***complete binary tree***. For example, consider the binary tree shown below:



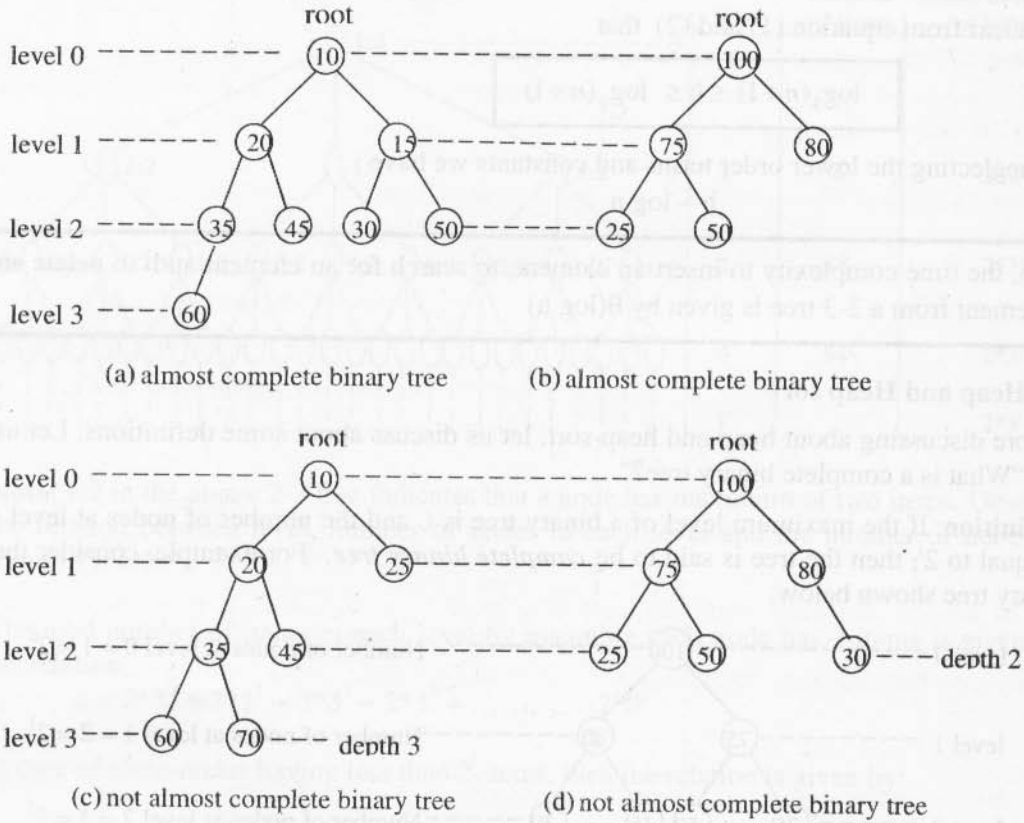
The maximum level of this tree is 2 and the number of nodes at level 2 is 2^2 which is nothing but 4. In general, in a complete binary tree, the number of nodes at the last level i will be 2^i.

Let us see "What is an almost complete binary tree or nearly complete binary tree?"

Definition: A binary tree of level i is called an almost complete binary tree or nearly complete binary tree if and only if the following two conditions are satisfied:

- If level of a tree is i, it should have 2^i nodes in level i.
- If 2^i nodes are not present in level i, in $(i-1)^{th}$ level, the number of nodes should be 2^{i-1} and in level i all the nodes should be filled from left to right.

If one of the above two conditions fails, the binary tree is not an *almost complete binary tree*. For example, the trees shown in 6.5.1a and 6.5.1.b are almost complete binary trees, because both the trees satisfy the given two conditions.



(a) almost complete binary tree (b) almost complete binary tree

(c) not almost complete binary tree (d) not almost complete binary tree

Figure 6.5.1 Binary trees

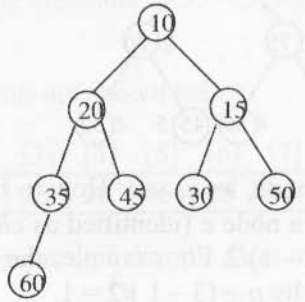
Now, let us see "What is a heap? What are the different types of heap"

Definition: A heap is a complete binary tree or an almost complete binary tree such that the key at each node is greater than or equal to the keys of its children. A heap can be classified as shown below:

- Ascending heap (min heap)
- Descending heap(max heap)

Let us see "What is an ascending heap?" An *ascending heap* often called *min-heap* is an almost complete binary tree such that an item at any given node is less than or equal to the left child and right child. So, in an ascending heap, the root node contains the least

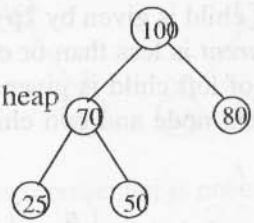
element and the elements in any path from root to a leaf will be in ascending order. The tree shown below an ascending heap.

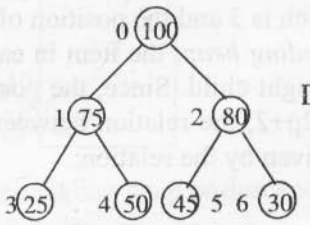


Ascending heap

Now, let us see "What is a descending heap?" A de*scending heap* often called *max-heap* is an almost complete binary tree such that an item at any given node is greater than or equal to the left child and right child. So, in a descending heap, the root node contains the highest element and the elements in any path from root to a leaf will be in descending order. The trees shown below represent descending heaps.

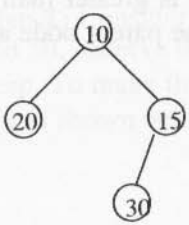
Descending heap



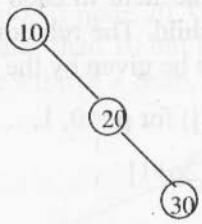


Descending heap

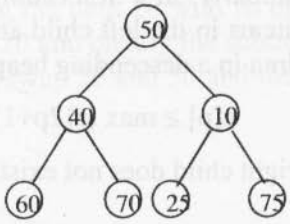
The trees shown in figure 6.5.2a and 6.5.2b are not heaps, because all the leaves in the lowest level are not inserted from left to right. The tree shown in figure 6.5.2c is not a heap since it is not either descending heap or ascending heap.



(a)



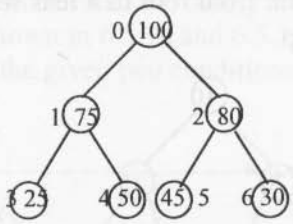
(b)



(c)

Figure 6.5.2 Binary trees

Since a heap is *complete binary tree* or almost complete binary tree, the nodes can be numbered from the root, one level at a time. In each level, the nodes are numbered sequentially from left to right as shown below:



Using the numbers given for each node, let us see "How to find the position of the parent given the child's position?" Given a node *c* (identified as *child* node), the position of the parent node say *p* is obtained by (c – 1)/2. For example, the position of node 25 is 3 (i.e., c = 3). The parent position is given by *p* = (3 – 1)/2 = 1.

Now, let us see "How to find the position of the child given the parent position?" A given node *p* (identified as parent node), the position of left child can be obtained using the relation 2p+1 and that of right child can be obtained using 2p + 2. For example, in the above tree, consider the item 75 whose position is 1. If this node is considered as the parent node, its left child is 25 and the right child is 50. The position of the left child is given by 2p+1 which is 3 and the position of right child is given by 2p+2 which is 4.

In an *ascending heap*, the item in each *parent* is less than or equal to the item in the left child and right child. Since, the position of left child is given by 2p+1 and right child is given by 2p+2, the relation between parent node and two children in ascending heap can also be given by the relation:

$$a[p] \le \min(a[2p+1], a[2p+2]) \text{ for } p = 0, 1, \ldots\ldots \left(\frac{n}{2}-1\right)$$

Note: If right child does not exist then always a[p] ≤ a[2p+1]

Similarly, in a *descending heap*, the item in each parent node is greater than or equal to items in the left child and right child. The relation between the parent node and two children in a descending heap can also be given by the relation:

$$a[p] \ge \max(a[2p+1], a[2p+2]) \text{ for } p = 0, 1, \ldots\ldots \left(\frac{n}{2}-1\right)$$

Note: If right child does not exist a[p] ≥ a[2p+1]

Note: Heaps are represented using arrays and not using linked lists. A tree is represented as an array but not as a linked list. The above tree can be represented using arrays as shown below:

A[0]	[1]	[2]	[3]	[4]	[5]	[6]
100	75	80	25	50	45	30

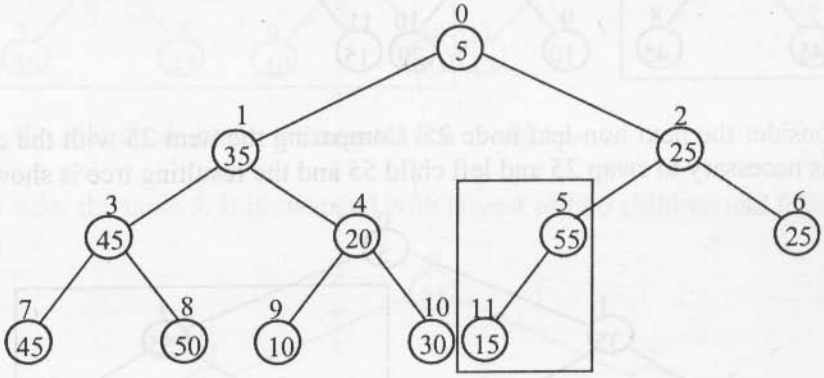
6.5.1 Create a heap (Bottom-up heap construction)

Let us "Create a heap for the elements 5, 35, 25, 45, 20, 55, 25, 45, 50, 10, 30 and 15 using bottom up approach"

Solution: Given array elements are shown below:

a	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
	5	**35**	**25**	**45**	**20**	**55**	**25**	**45**	**50**	**10**	**30**	**15**

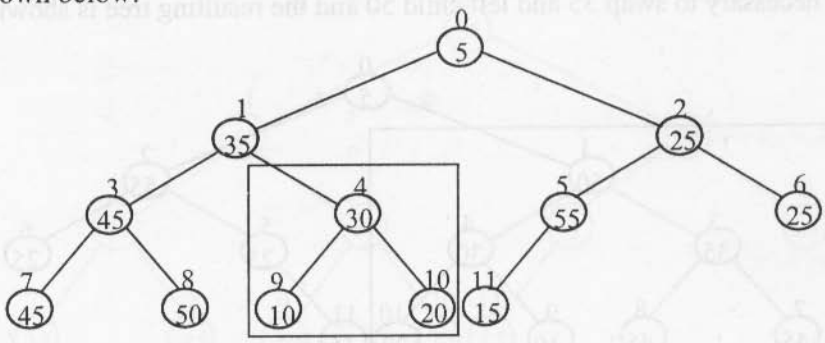
These array elements can be represented in the form of a binary tree as shown below:



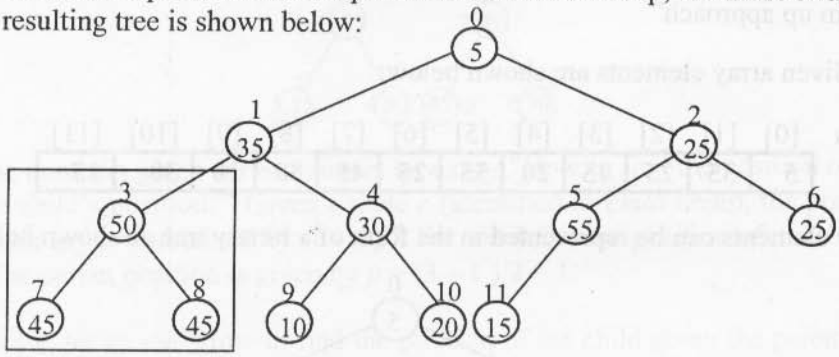
Note: The tree thus represented is not either ascending heap or descending heap.

Step 1: Consider the last non-leaf node 55. Compare 55 with its children. Clearly, 55 is greater than its child 15 and the sub-tree (shown in rectangular box) satisfies the heap condition and no change is required.

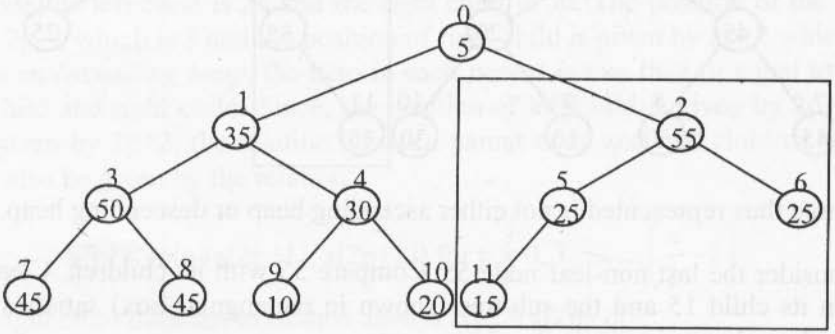
Step 2: Consider the next non-leaf node 20. Comparing the item 20 with its children 10 and 30, observe that 20 is greater than 10 but less than 30 and clearly the subtree is not a heap. To make this part of the subtree a heap, we must swap 20 and 30 and the resulting tree is shown below:



Step 3: Consider the next non-leaf node 45. Comparing the item 45 with its children 45 and 50, observe that 45 is equal to left child but less than the right child 50 and clearly the subtree is not a heap. To make this part of the subtree a heap, we must swap 45 and 50 and the resulting tree is shown below:

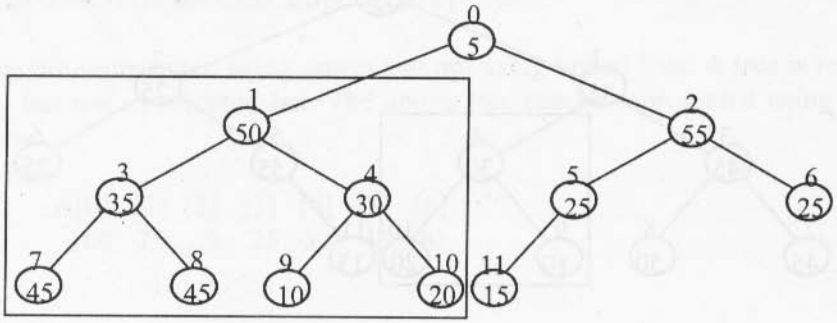


Step 4: Consider the next non-leaf node 25. Comparing the item 25 with the children 55 and 25, it is necessary to swap 25 and left child 55 and the resulting tree is shown below:

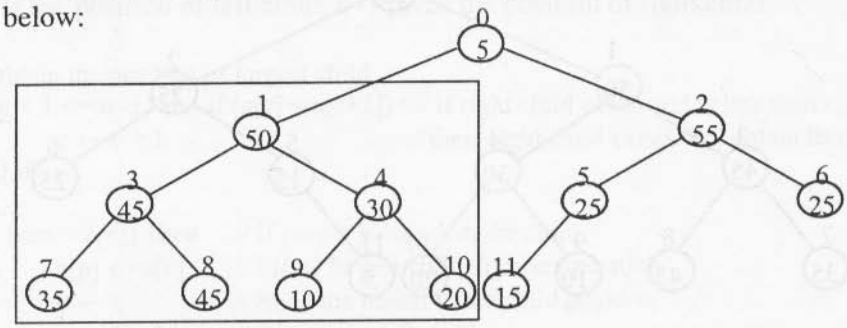


After swapping, compare 25 with its left child. Since 25 is greater than 15, the subtree shown in rectangular box is a heap.

Step 5: Consider the next non-leaf node 35. Comparing the item 35 with the children 50 and 30, it is necessary to swap 35 and left child 50 and the resulting tree is shown below:

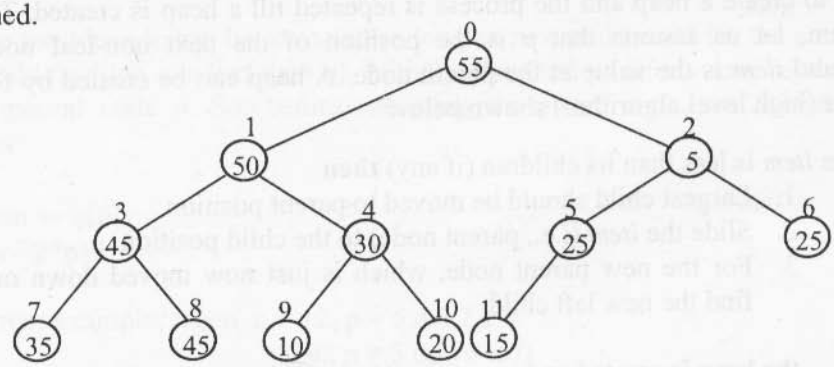


After swapping 35 and 50, check whether subtree identified by root node 35 forms a heap. Comparing 35 with 45 and 45, clearly it is not a heap. So, 35 should be exchanged with largest of two children. Since left child and right child are same, we can exchange either with left child or right child. Let us exchange with left child and the resulting tree is shown below:

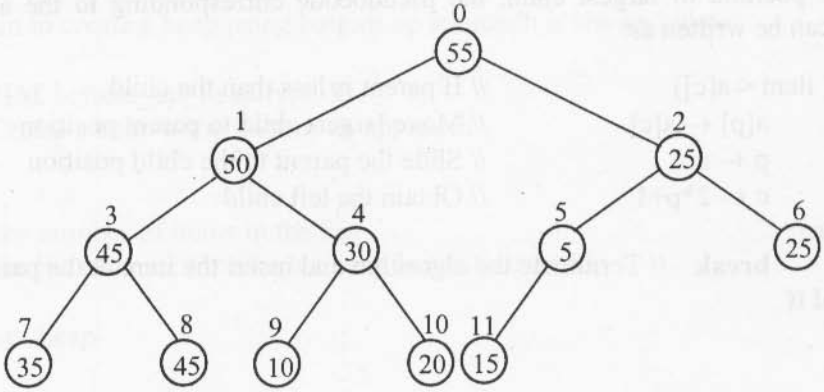


Note: Now, the subtree shown in rectangular box is a heap.

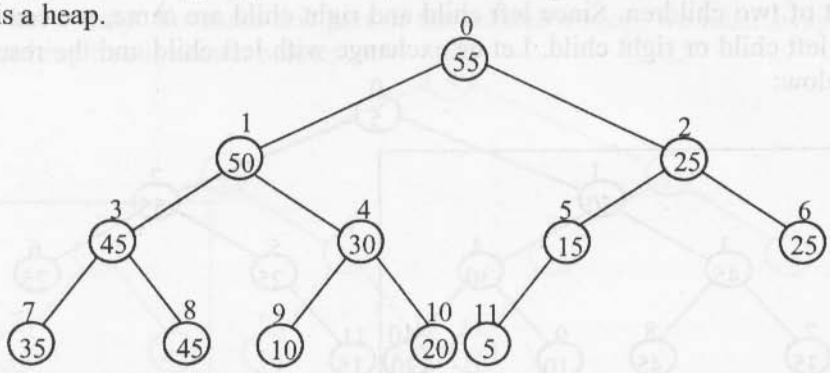
Step 6: Consider the node 5. It is swapped with largest of two children and following tree is obtained.



Now, it is necessary to swap 5 with left child (We could have swapped with right child also since left child and right child are same) and the resulting tree is shown below:



Finally, compare 5 with 15 and we know that 5 is less than its child 15 and so swapping is necessary. After swapping 5 with 15, the following tree is obtained and observe that the tree is a heap.



Design It is clear from the above procedure that the last non-leaf (last node which is a parent) node was considered first and the subtree of that node was altered to create a heap. Moving backwards, the next non-leaf is considered and the subtree from that node was altered to create a heap and the process is repeated till a heap is created. To design the algorithm, let us assume that *p* is the position of the next non-leaf node to be considered and *item* is the value at the parent node. A heap can be created by following the activities (high level algorithm) shown below:

If the *item* is less than its children (if any) **then**

1. Largest child should be moved to parent position
2. Slide the *item* (i.e., parent node) to the child position.
3. For the new parent node, which is just now moved down one level, find the new left child

else

the heap is created and insert the item at the parent position and terminate

end if

If *c* is the position of largest child, the pseudocode corresponding to the above three activities can be written as:

```
if ( item < a[c])          // If parent is less than the child
    a[p] ← a[c]            // Move largest child to parent position
    p ← c                  // Slide the parent to the child position
    c ← 2*p+1              // Obtain the left child
else
    break  // Terminate the algorithm and insert the item at the parent position
end if
```

The above statements should be performed only after finding the position of the largest child. The pseudocode to find the position of the largest child and to create a heap is shown below:

Note: If *c* is the position of left child, c+1 gives the position of right child

```
// Obtain the position of largest child
if (c + 1 <= n -1) and if (a[c] < a[c+1])  // if right child exists and is less than right child
        c ← c + 1                          //then, right child exists. So, obtain its position
end if

if ( item < a[c]) then     // If parent is less than the child
        a[p] ← a[c]        // Move largest child to parent position
        p ← c              // Slide the parent to the child position
        c ← 2*p+1          // Obtain the left child
else
        break  // Terminate the algorithm and insert the item at the parent position
end if
```

The above pseudocode can be executed as long as new child exist for the parent (i.e., as long as child position *c* is less than *n*). Initially, the child position *c* can be obtained only from the parent node *p*. So, before executing the above code, the initialization part constitutes:

```
item ← a[p]
c ← 2*p+1
```

In the current example, where n = 12, p = 5,4,3,2,1,0

i.e., p = 5 down to 0

p = (12-1)/2 down to 0

In general, p = (n-1)/2 down to 0

The algorithm to create a heap using bottom up approach is shown below:

ALGORITHM bottom_up_heapify(n, a)
//**Purpose**: Create a heap using bottom up approach

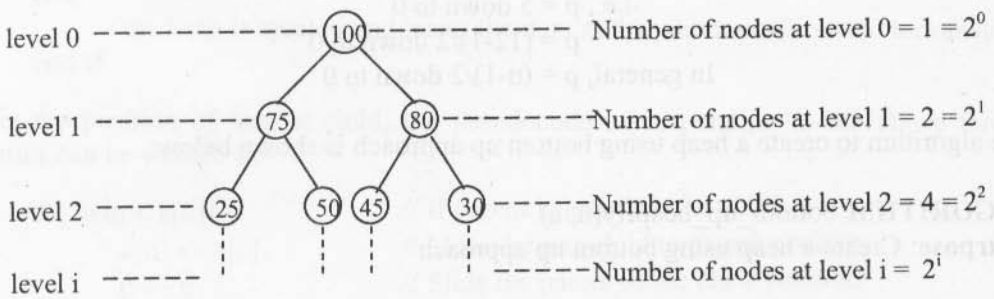
// **Inputs** :
// n – the number of items in the list
// a – the list of items for which a heap has to be created

//**Output**: max-heap

```
for i = (n-1)/2 down to 0
    item ← a[p]          // p is position of next last-node & item is the parent node
    c ← 2*p+1            // obtain the position of left child

    while c <= n-1 do                          // As long as left child exists do
        if ( c + 1 <= n-1 ) then               // If right child also exists?
            if ( a[c] < a[c+1] ) c ← c + 1     // obtain position of largest child
        endif

        if ( item < a[c] ) then  //if item (parent) is less than the largest child
            a[p] ← a[c]          //move the largest child to parent position
            p ← c                //Move the parent to child position
            c ← 2*p+1            //and obtain the position of new left child
        else
            break                // and the algorithm is terminated
        end if
    end while
    a[p] ← item;   // The resulting tree is heap. So, insert at parent position
end for
// End of algorithm
```

Example: Prove that the height of a heap with n nodes is equal to $\lfloor \log_2 n \rfloor$

Solution: Let us take the worst scenario in which there are maximum number of items in a tree for which the heap has to be created. This scenario occurs if and only if the tree is fully complete as shown in figure below:



It is clear from the above tree that, the relation between the total number of items n and the number of nodes occurring in each level from level 0 to level i is given by the following inequality:

$$n = 2^0 + 2^1 + 2^2 + \ldots\ldots 2^i = 1\left(\frac{2^{i+1}-1}{2-1}\right) \text{ using the formula } S = a\left(\frac{r^n-1}{r-1}\right)$$

i.e.,$n = 2^{i+1}-1 = 2^h - 1$ where $i + 1 = h$ (Note: level + 1 = height of the tree)
i.e., $n + 1 = 2^h$

So, taking log on both sides we get $\log_2 n+1 = h$. So, the relation between the height of the tree and the number of nodes n in a tree is given by

$$\boxed{h = \log_2 n+1 \approx \log_2 n}$$

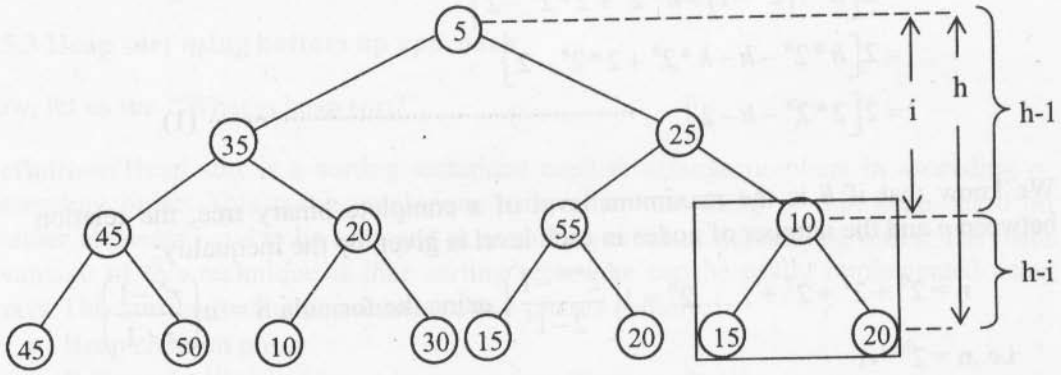
So, we can say that there exists a binary tree with n nodes whose height is equal to $\lfloor \log_2 n \rfloor$.

6.5.2 Analysis of creating a heap using bottom up approach

Observe that while creating a heap, the number of comparisons required to move an item from the parent position to the child position is given by 2 i.e.,

- First comparison to find the largest child (For example, in rectangular box shown 15 and 20 are compared)
- The second comparison to determine whether parent should be exchanged with child node. (For example, in the rectangular box check whether swapping is necessary between 10 and 20)

and thus 2 comparisons are required).



So, the total number of comparisons required to move a single node from level i to leaf level in the worst case is given by

2(h-i)

Since the number of nodes at level i is 2^i (in this example, $2^i = 4$), the total number of comparisons required to move all these nodes from level i to leaf level is given by

$$2(h-i)*2^i$$

Starting from last non-leaf nodes at level i which is at a height of h-1,(see above figure for clarity), we move backwards till we get the root node which is at level 0. So, the total number of key comparisons from level (h-1) to level 0 is given by

$$f(n) = \sum_{i=0}^{h-1} \underbrace{2(h-i)}_{\text{Number of comparisons required to move a single node from level } i \text{ to leaf level}} * \underbrace{2^i}_{\text{Number of nodes at level i}}$$

$$= 2\sum_{i=0}^{h-1}(h-i)*2^i = 2\left[\sum_{i=0}^{h-1} h*2^i - \sum_{i=0}^{h-1} i*2^i\right]$$

$$= 2\left[h\sum_{i=0}^{h-1} 2^i - \sum_{i=1}^{h-1} i*2^i\right]$$

$$= 2\left[h\left(2^0+2^1+2^2+\ldots 2^{h-1}\right)-\left((h-2)2^h+2\right)\right] \quad \textbf{Note: } \sum_{i=1}^{n} i2^i = (n-1)2^{n+1}+2$$

$$= 2\left[h*1\left(\frac{2^h-1}{2-1}\right)-h*2^h+2*2^h-2\right]$$

$$= 2\left[h*1\left(2^h-1\right)-h*2^h+2*2^h-2\right]$$

$$= 2\left[h*2^h-h-h*2^h+2*2^h-2\right]$$

$$= 2\left[2*2^h-h-2\right] \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \textbf{(1)}$$

We know that if h is the maximum level of a complete binary tree, the relation between n and the number of nodes in each level is given by the inequality:

$$n = 2^0+2^1+2^2+\ldots\ldots 2^h = 1\left(\frac{2^{h+1}-1}{2-1}\right) \text{ using the formula } S = a\left(\frac{r^n-1}{r-1}\right)$$

i.e., $n = 2^{h+1}-1$

i.e., $n + 1 = 2^{h+1}$

Taking log on both sides we get,

$\log_2 n+1 = (h+1)\ \log_2 2 = h+1$

$\therefore, h = \log_2(n+1) - 1$

Substituting this value in equation (1) we get

$$
\begin{aligned}
f(n) &= 2\left[2*2^{h}-h-2\right] \\
&= 2\left[2*2^{\log_2(n+1)-1}-(\log_2(n+1)-1)-2\right] \\
&= 2\left[2*\left(\frac{2^{\log_2(n+1)}}{2}\right)-\log_2(n+1)-1\right] \\
&= 2\left[2^{\log_2(n+1)}-\log_2(n+1)-1\right] \\
&= 2\left[(n+1)^{\log_2 2}-\log_2(n+1)-1\right] \\
&= 2\left[n+1-\log_2(n+1)-1\right] \\
&= 2(n-\log_2(n+1))
\end{aligned}
$$

So, it is clear from the above relation that using bottom up algorithm, a heap of size n can be constructed with fewer than $2n$ comparisons.

So, the time complexity to create a heap using bottom up approach is given by O(n).

Example: Prove the following inequality $\sum_{i=h-1}^{0} 2(h-i)*2^{i}$ =2(n-log$_2$(n+1))

where n = 2^{h+1}-1

Solution: See the previous derivation

6.5.3 Heap sort using bottom up approach

Now, let us see "What is heap sort?"

Definition: Heap sort is a sorting technique used to arrange numbers in ascending or descending order. This sorting technique uses either max heap or min heap based on whether elements have to be arranged in ascending order or descending order. The main advantage of this technique is that sorting technique can be easily implemented using arrays. This sorting technique consists of two phases namely:

- Heap creation phase
- Delete root item and insert in appropriate (Sorting phase)

Heap creation phase: A heap can be created using bottom-up technique as discussed earlier. If we want the items to be in ascending order it is required to create max heap. If we want the items in descending order it is required to create min heap.

Sorting phase: In this phase, items are arranged in ascending order (if we use max heap) or descending order(if we use min heap). This is achieved by repeatedly performing the following activities:

- Exchange the root item with last element of the heap
- Decrement the size of the heap by 1 and recreate the heap by using the *bottom_up_heapify*()

The pseudocode corresponding to these two steps is given by:

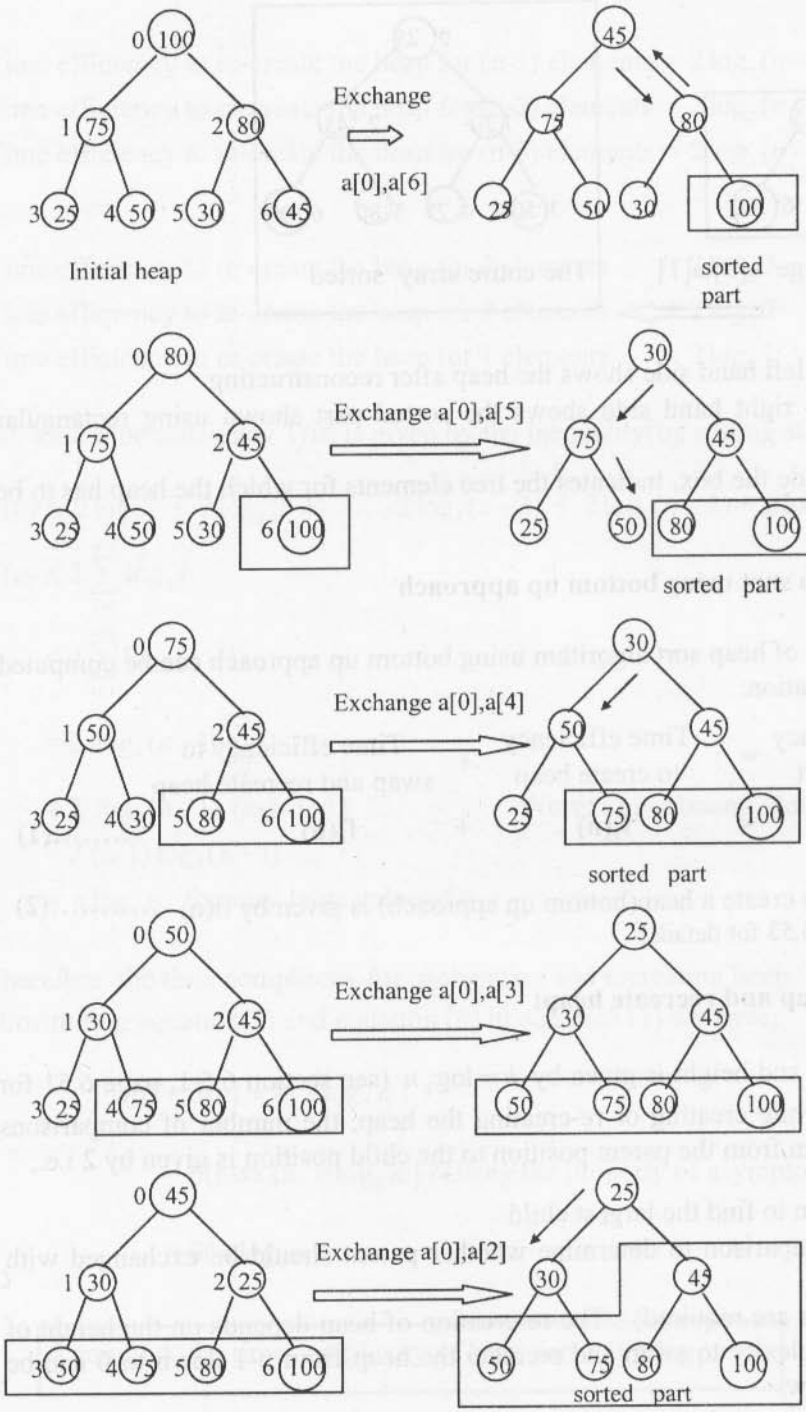
```
Exchange a[0], a[i]          // exchange the root item with the last leaf
bottom_up_heapify(i, a)      // Recreate the heap from the root
```

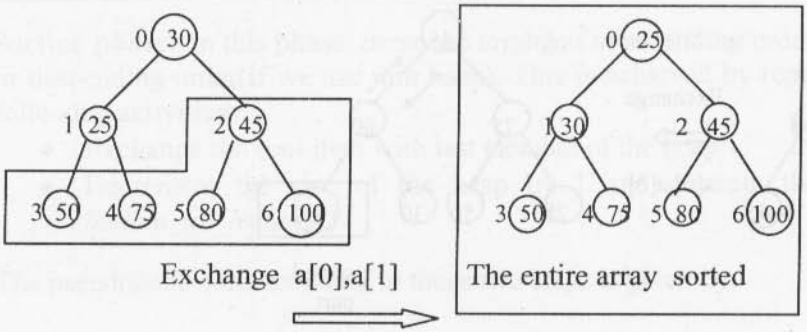
The complete algorithm to arrange numbers in ascending order using bottom-up-approach is shown below:

```
ALGORITHM heap_sort(n, a[])

//Purpose: To sort the items by using heap

// Inputs :
//      1: n – the number of items to be sorted
//      2: a – the items to be arranged using sorting technique

//Output:
//      a – contains the sorted list

Step1: [Create the heap using bottom-up-approach]
       bottom_up_heapify(n, a)          // create a heap

Step 2:[Repeatedly exchange and recreate the heap]
       for i = n-1 down to 0
              Exchange a[0], a[i]       // exchange the root item with the last leaf
              bottom_up_heapify(i, a)   // Recreate the heap from the root
       end for
Step 3:[The items are sorted and terminate the algorithm]
       return
```

6.5.4 Tracing of heap sort

Initially, using the function bottom_up_heapify create the heap as shown in figure below: Exchange a[0] with a[6]. The last item is sorted. Recreate the heap for remaining 6 elements. Exchange a[0] with a[5]. Now last two items are sorted. Then recreate the heap for 5 elements. Repeating this way all the items are arranged in ascending order.


Exchange
a[0],a[6]
Initial heap
sorted part
Exchange a[0],a[5]
sorted part
Exchange a[0],a[4]
sorted part
Exchange a[0],a[3]
sorted part
Exchange a[0],a[2]
sorted part



- The figures on the left hand side shows the heap after reconstructing
- The figure on the right hand side shows the sorted part shown using rectangular region.
- The elements outside the box, indicates the tree elements for which the heap has to be created.

6.5.5 Analysis of heap sort using bottom up approach

So, the time efficiency of heap sort algorithm using bottom up approach can be computed using the following relation:

Time efficiency of heap sort	=	Time efficiency to create heap	+	Time efficiency to swap and recreate heap	
f(n)	=	**$f_1(n)$**	+	**$f_2(n)$**	**.....(1)**

The time complexity to create a heap(bottom up approach) is given by $\theta(n)$ **.....(2)**
(See section 6.5.2, page 6.53 for details)

Time efficiency to swap and recreate heap:

The relation between n and height is given by $h = \log_2 n$ (see section 6.5.1, page 6.51 for details. Observe that while creating or re-creating the heap, the number of comparisons required to move an item from the parent position to the child position is given by 2 i.e.,

- First comparison to find the largest child
- The second comparison to determine whether parent should be exchanged with child node

and thus 2 comparisons are required). The re-creation of heap depends on the height of the tree. So, time complexity to swap and recreate the heap from n-1 down to 0 can be obtained as shown below:

Time efficiency to re-create the heap for (n-1) elements = $2\log_2(n-1)$

Time efficiency to re-create the heap for (n-2) elements = $2\log_2(n-2)$

Time efficiency to re-create the heap for (n-3) elements = $2\log_2(n-3)$

.....

.....

Time efficiency to re-create the heap for 3 elements = $2\log_2 3$

Time efficiency to re-create the heap for 2 elements = $2\log_2 2$

Time efficiency to re-create the heap for 1 elements = $2\log_2 1$

So, total time efficiency T(n) is given by the inequality(by adding all the above terms)

$$T(n) \le 2\log_2 1 + 2\log_2 2 + \ldots\ldots 2\log_2(n-3) + 2\log_2(n-2) + 2\log_2(n-1)$$

$$T(n) \le 2\sum_{i=1}^{n-1}\log_2 i$$

$$\le 2\sum_{i=1}^{n-1}\log_2(n-1)$$

$$= 2\ \log_2(n-1)\sum_{i=1}^{n-1}1$$

$$= 2\ \log_2(n-1)\ (n-1-1+1)$$ **Note:** upper bound –lower bound + 1

$$= 2\ (n-1)\log_2(n-1)$$

$$\approx n\log_2 n$$ for very large value of n

Therefore, the time complexity for exchanging and recreating heap = $n\log_2 n$ .....(3)

Substituting equation(2) and equation (3) in equation (1) we have,

$$T(n) = \theta(n) + \theta(n\log_2 n)$$

$$= \theta(\max\{n,\ n\log_2 n\})$$ Using the property of asymptotic notation

$$= \theta(n\log_2 n)$$

Therefore, the time complexity of heap sort = $\theta(n\log_2 n)$

6.6 Create a heap (Top-down approach)

In top-down approach, a *heap* can be created by successive insertion of a new key into a previously created heap. The new item say *key*, should be inserted always after the last leaf in the existing heap.

For example, consider the descending heap shown in figure (a). The new item say 95 should be inserted after the last leaf i.e., at 5th position as shown in figure (b).

Note: In the heap shown in figure(a), the first leaf is 30, the second leaf is 25 and last leaf is 50.

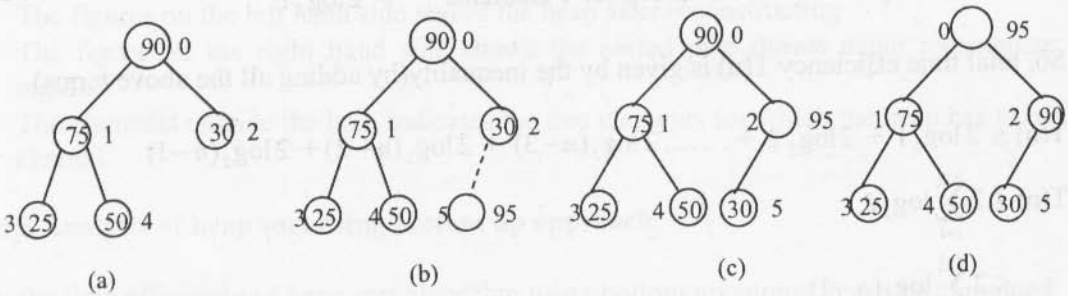


Figure 6.6.1 Inserting an item into a heap

Observe that if the item 95 is inserted at the last position as shown in figure (b), the resulting tree is not a heap. So, we can not simply drop an item at the last leaf (called child position). Instead, the child (the position where the item is dropped) in the heap is moved towards the root upward by moving the items in the path down to child position as shown in figure (c) and (d) and finally, 95 can be inserted at 0th position.

In general, the process of moving the items down the heap there by moving the child upward is terminated either when the child has reached the root (as shown in figure (d), or when the child is moved to a position in which inserting the item at child's position results in a heap. Now, let us see "What is the procedure to be followed to create heap using to-down approach?" Thus, heap can be constructed using top-down approach as shown below:

- Obtain the item to be inserted and its position. This position is called child's position. For example, in the figure (b), item is 95 and child position is 5.
- As long as parent exists and item is greater than parent
  - i. Move the parent down to child
  - ii. Make parent as the child
  - iii. Obtain the position of new parent
- Insert the item into child's position

The corresponding pseudocode to insert 95 at position 5 using the above general procedure can be written as:

```
Step 1: key ← a[5]          // Item to be inserted
        c ← 5;              // Position of the child where item has to be inserted
        p ← (c-1)/2         // Obtain the position of the parent

Step 2: while c ≠ 0 and key > a[p]
            a[c] ← a[p]     // Move the parent down to the position of child
            c ← p           // Make parent as the child
            p ← (c-1)/2     //  Obtain the position of new parent
        end while

Step 3: a[c] ← key (=95)    // Insert key at the child's position
```

The complete algorithm to create a heap is shown below:

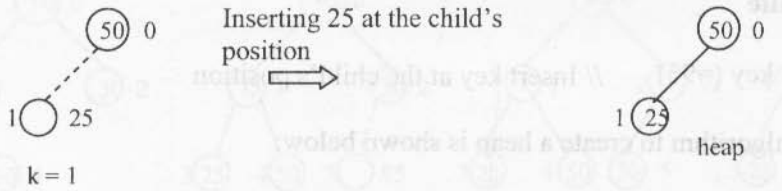
```
ALGORITHM  top_down_heapify(n, a[])

//Purpose: To create a max heap

// Inputs :
//      1: n – the number of items to be inserted into the heap
//      2: a – the items to be inserted into the heap

//Output:
//      a – contains the max heap

for k ← 1 to n-1 do
        item ← a[k]         // Item to be inserted
        c ← k               // Obtain the position child where item has to be inserted
        p ← (c-1) / 2       // Obtain the position of the parent

        // While parent exists and item is greater than parent
        while c > 0 and item > a[p] do
            a[c] ← a[p]     // Move the parent down to the position of child
            c ← p           // Make parent as the child
            p ← (c-1)/2     //Obtain the position of new parent
        end while

        a[c] ← key          // // Insert key at the child's position
end for
```

Note: In top-down approach, a *heap* can be created by successive insertion of a new key into a previously created heap. So, by assuming 0th item is the existing heap, start inserting 1st, 2nd, 3rd and $(n-1)^{th}$ item and create the heap.

Example: Construct a heap for the list 50, 25, 30, 75, 100, 45, 80 by successive insertion using top-down approach

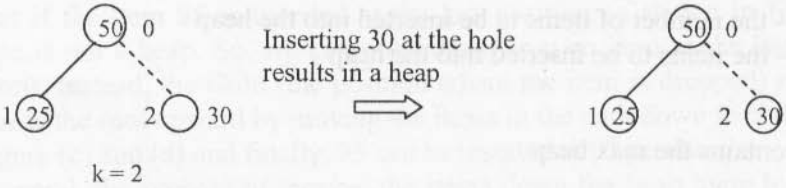
Step 1: Item = 50. Inserting the item into an empty tree, we get the tree as shown below:

Step 2: Item = 25. Inserting the item into above heap at position 1, following heap is obtained



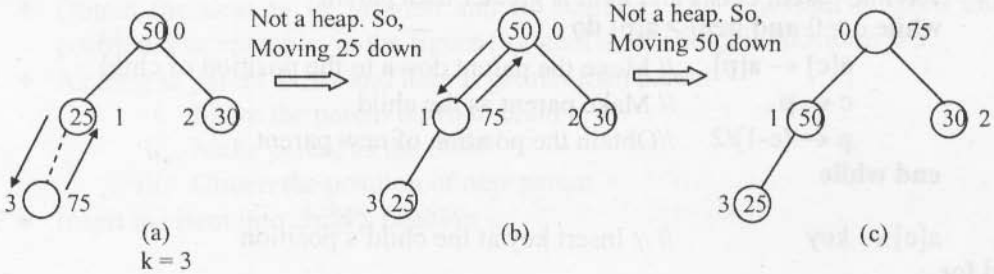
Observe that the parent 50 is greater than the child 25 and so, it can be inserted at the child's position and the resulting tree is a heap.

Step 3: Item = 30. After inserting into above heap at position 2 we get following heap

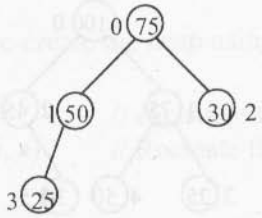


Observe that the parent 50 is greater than the child 30 and so, it can be inserted at the child's position and the resulting tree is a heap.

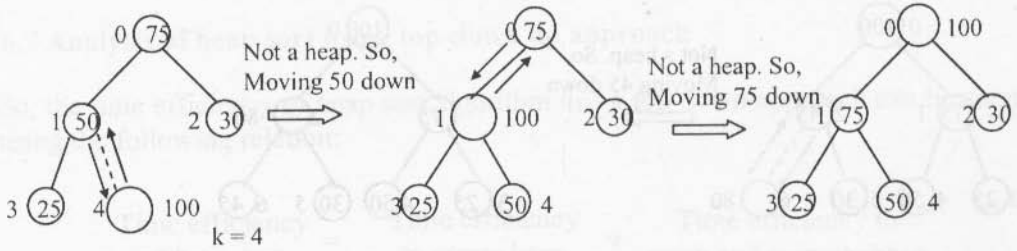
Step 4: Item = 75. Inserting the item into above heap at position 3 we get the following heap:



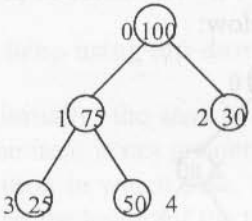
Finally, the key 75 can be inserted into the child at root node and the resulting heap is shown below:



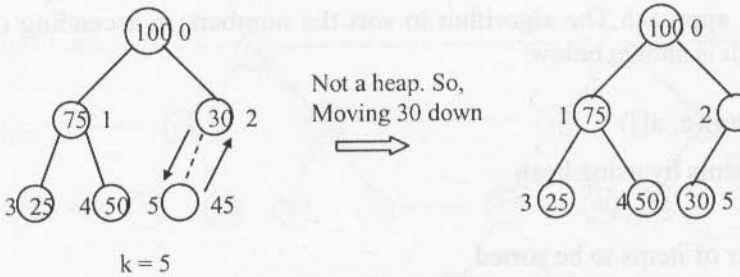
Step 5: Item = 100. Inserting the item into above heap at position 4 we get the following tree:



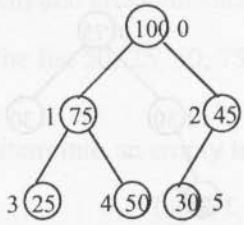
Finally, the key 100 can be inserted into the child's at root node and the resulting heap is shown below:



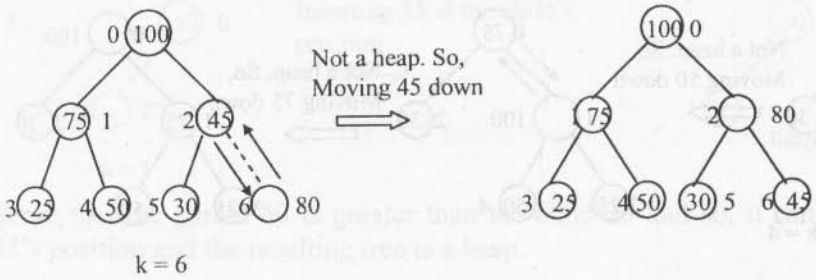
Step 6: Item = 45 Inserting the item into above heap at position 5 we get the following tree:



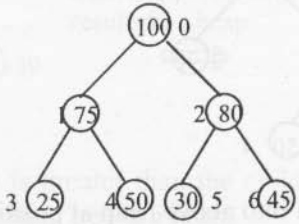
Since the parent 100 is greater the item 45, we can insert 45 at child's position and the resulting tree which is a heap is shown below:



Step 7: Item = 80. Inserting the item into above heap at position 6, we get the following tree:



Since the parent 100 is greater the item 80, we can insert 80 at position of child and the resulting tree which is a heap is shown below:



The resulting tree is a heap. Thus, items can be inserted one by one and a heap can be created using top-down approach.The algorithm to sort the numbers in ascending order using top-down approach is shown below:

ALGORITHM heap_sort(n, a[])

//**Purpose**: To sort the items by using heap

// **Inputs** :

// 1: n – the number of items to be sorted

// 2: a – the items to be arranged using sorting technique

//**Output**: // a – contains the sorted list

Step1: [Create the heap using top-down-approach]
top_down_heapify(n, a)

Step 2: [Repeatedly exchange and re-create the heap using bottom-up-approach]
for i ← n-1 **down to** 0
Exchange a[0], a[i] // exchange the root item with the last leaf
bottom_up_heapify(i, a) // Recreate the heap from the root node
end for

Step 3: [All items are arranged in ascending order and terminate the algorithm]
return

6.7 Analysis of heap sort using top-down up approach

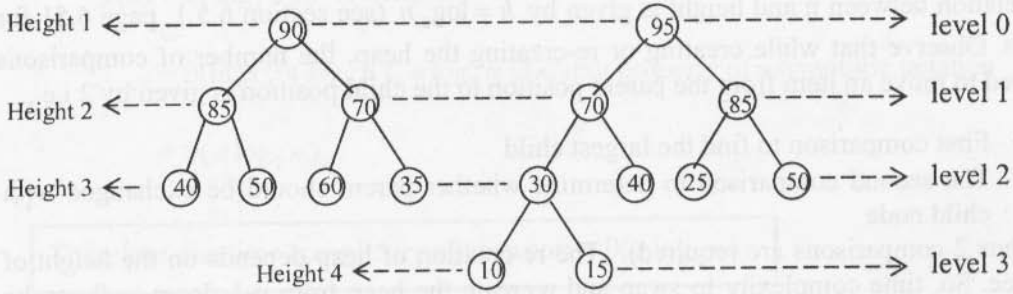
So, the time efficiency of heap sort algorithm using top-down approach can be computed using the following relation:

$$\text{Time efficiency of heap sort} = \text{Time efficiency to create heap} + \text{Time efficiency to swap and recreate heap}$$

$$f(n) = f_1(n) + f_2(n) \quad \text{.....(1)}$$

Time efficiency to create heap using top-down approach:

While creating the heap, initially, the *item* to be inserted is inserted at the leaf level. Later, it is moved up till the item is not greater than its parent. So, in the worst case, the item may occupy *root* position in which case, the total number of movements from leaf level to root level is given by the height of the tree. Now, let us have the relation between height *h* of the tree and number of elements *n*. Consider the complete binary tree shown in figure below:



In a complete binary tree,
Number of nodes at height 1 is $2^{1-1} = 1$
Number of nodes at height 2 is $2^{2-1} = 2$
Number of nodes at height 3 is $2^{3-1} = 4$
.....
.....
Number of nodes at height h is 2^{h-1}.

Some times the number of nodes at the last level will be less than 2^{h-1} (see the second figure). So, the total number of nodes in the complete binary tree is given by the inequality

$$n \le 2^0 + 2^1 + 2^2 + \ldots\ldots 2^{h-1}$$

$$\text{i.e., } n \le 2^{h-1}$$

Taking log on both sides we have

$$\log_2 n \le (h-1)\log_2 2$$

$$i.e., \log_2 n \le h-1$$

$$i.e., h-1 \ge \log_2 n$$

$$i.e., h \ge \log_2 n + 1$$

Note that the heap will be either complete binary tree or almost complete binary tree. In such case if an element is inserted at the leaf level, it may move up to root level in the worst case and thus requiring the number of comparisons to be equal to that of the height of the tree and is proportional to $\log_2 n$.

So, the time complexity to create the heap is given by $\theta(\log_2 n)$ .....(2)

Time efficiency to swap and recreate heap:

The relation between n and height is given by $h = \log_2 n$ (see section 6.5.1, page 6.51 for details. Observe that while creating or re-creating the heap, the number of comparisons required to move an item from the parent position to the child position is given by 2 i.e.,

- First comparison to find the largest child
- The second comparison to determine whether parent should be exchanged with child node

and thus 2 comparisons are required). The re-creation of heap depends on the height of the tree. So, time complexity to swap and recreate the heap from n-1 down to 0 can be obtained as shown below:

Time efficiency to re-create the heap for (n-1) elements = $2\log_2(n-1)$
Time efficiency to re-create the heap for (n-2) elements = $2\log_2(n-2)$
Time efficiency to re-create the heap for (n-3) elements = $2\log_2(n-3)$
.....
.....
Time efficiency to re-create the heap for 3 elements = $2\log_2 3$
Time efficiency to re-create the heap for 2 elements = $2\log_2 2$
Time efficiency to re-create the heap for 1 elements = $2\log_2 1$

So, total time efficiency T(n) is given by the inequality(by adding all the above terms)

$$T(n) \le 2\log_2 1 + 2\log_2 2 + \ldots\ldots 2\log_2(n-3) + 2\log_2(n-2) + 2\log_2(n-1)$$

$$T(n) \le 2\sum_{i=1}^{n-1}\log_2 i$$

$$\le 2\sum_{i=1}^{n-1}\log_2(n-1)$$

$$= 2\ \log_2(n-1)\sum_{i=1}^{n-1}1$$

$$= 2\ \log_2(n-1)\ (n-1-1+1)$$ **Note:** upper bound –lower bound + 1

$$= 2\ (n-1)\log_2(n-1)$$

$$\approx n\log_2 n$$ for very large value of n

Therefore, the time complexity for exchanging and recreating heap = $n\log_2 n$ .....(3)
Substituting equation(2) and equation (3) in equation (1) we have,

$$T(n) = \theta(\log_2 n) + \theta(n\log_2 n)$$

$$= \theta(\max\{\theta(\log_2 n),\ n\log_2 n\})$$ Using property of asymptotic notation

$$= \theta(n\log_2 n)$$

Therefore, the time complexity of heap sort = $\theta(n\log_2 n)$

Note: The various features of heap sort are shown below:

- The time efficiency of heap sort is θ(n log n). Even then, heap sort is the slowest of the θ(n log n) sorting algorithms such as merge sort and quick sort.
- Unlike the merge and quick sorts it doesn't require massive recursion or multiple arrays to work. This makes it the most attractive option for very large data sets of millions of items.
- Heap sort is in-place (i.e., no extra space is used for an array other the given array) and non-recursive, making it a good choice for extremely large data sets.
- Heap psort is not stable. A sorting algorithm is stable, if it the order of equal elements remains unchanged.

6.8 Problem Reduction

Let us see "What is the strategy used in problem reduction?" In this strategy, a given problem X is transformed into another problem Y for which the solution exists in the form of a known algorithm. This can be done easily if we are able to relate X and Y and care must be taken while transforming the problem X into problem Y. The various problems include:

- Find LCM of two numbers
- To count paths in a graph
- Optimization problem
- Linear programming examples such as knapsack problem etc
- Graph problems

6.8.1 To find LCM (Least Common Multiple) of two numbers

Now, let us see "How to compute LCM of two numbers using problem reduction?" The LCM of two numbers say m and n can be calculated by multiplying m and n and then dividing the result with GCD of m and n. For example, if the values of m and n are 10 and 6 respectively, LCM is given by

$$LCM = \frac{10 * 6}{GCD(10,6)} \quad \frac{60}{2} = \quad = 30$$

Note: The GCD(m, n) can be obtained efficiently using Euclid's algorithm

The algorithm to find LCM of two numbers is shown below:

Algorithm LCM. This algorithm finds the LCM of two numbers.

Step 1: [Input the number]
read m, n

Step 2: [Find GCD of m, n. It is pre-defined in example 1.3]
d ← gcd (m, n)

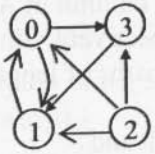
Step 3: [Find the LCM]
lcm ← m * n / d

Step 4: [Output the LCM]
write lcm

Step 5: [Finished]
exit.

6.8.2 To count paths in a graph

Let us consider the problem of counting the number of paths between two vertices using *problem reduction* technique. By using *adjacency matrix* and matrix multiplication procedure, we can easily find the number of paths of fixed length between various vertices. Consider the graph shown in figure 6.8.1 along with its adjacency matrix.



(a)

	0	1	2	3
0	0	1	0	1
1	1	0	0	0
2	1	1	0	1
3	0	1	0	0

(b)

Figure 6.8.1 A digraph and the adjacency matrix

We know that the matrix A is an adjacency matrix and an entry of 1 in i^{th} row and j^{th} column of matrix A shows the existence of an edge (v_i, v_j) which is a path of length 1 from v_i to v_j. Now, A^2 can be computed using

$$a_{ij}^{(2)} = \sum_{k=0}^{n-1} a_{ik} a_{kj}$$

Observe that for any k from 0 to n-1, if a_{ik} is 1 and a_{kj} is 1, then only $a_{ik}a_{kj}$ is 1. Thus, if there is a path from (v_i, v_k) and (v_k, v_j), it implies that there is a path from v_i to v_j of length 2. Therefore, $a_{ij}^{(2)}$ gives the various paths of length 2 from v_i to v_j. Similarly $a_{ij}^{(3)}$ gives the various paths of length 3 from v_i to v_j and $a_{ij}^{(4)}$ gives the various paths of length 4 from v_i to v_j as shown in figure 6.8.2.

$A^2 =$

	0	1	2	3
0	1	1	0	0
1	0	1	0	1
2	1	2	0	1
3	1	0	0	0

$A^3 =$

	0	1	2	3
0	1	1	0	1
1	1	1	0	0
2	2	2	0	1
3	0	1	0	1

$A^4 =$

	0	1	2	3
0	0	2	0	1
1	1	1	0	1
2	2	3	0	2
3	1	1	0	0

Figure 6.8.2 Paths of length 2, 3 and 4 between all nodes

Note: The elements of A, A^2, A^3 and A^4 indicate the number of paths of lengths 1, 2, 3 and 4 respectively.

In general, the value in i^{th} row and j^{th} column of A^n gives a path of length n from i^{th} node to j^{th} node. Observe that the value 2 in 2nd row and 1st column (assuming the index starts from 0) of A^2 indicate that there are two paths of length 2 from v_2 to v_1. The value of 2 in 2nd row and 0th column as well as value of 2 in 2nd row and 1st column in A^3 indicate that there are 2 paths of length 3 from v_2 to v_0 and v_2 to v_1 respectively. The value of 2 in 2nd row 0th column, 0th row 1st column and 2nd row 3rd column in A^4 indicate there are 2 paths of length 4 from v_2 to v_0, v_0 to v_1 and v_2 to v_3 respectively. Similarly the value of 3 in 2nd row and 1st column in A^4 indicate that there are 3 paths of length 4 from v_2 to v_1 namely

$2 \rightarrow 0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ is one path,
$2 \rightarrow 1 \rightarrow 0 \rightarrow 3 \rightarrow 1$ is another path and
$2 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow 1$ is another path.

6.8.3 Optimization Problem (Maximize/minimize)

Let us see "How to find minimum of n numbers using problem reduction?" Assume that we have to find minimum of following numbers:

10, 5, 45, 20, 50, 15, 25

But, we have an existing algorithm to find maximum of n numbers ((for details see section 2.7.1 and page 2.18). Using this maximum function, let us see how to find minimum of n numbers.

This can be achieved by negating all the values as shown below:

-10, -5, -45, -20, -50, -15, -25

and use our maximum function. The function returns the value -5. But, to get the correct answer i.e, to find minimum of all numbers, we are supposed to negate the answer. After negating we get 5 which is minimum of given numbers. The complete algorithm to find minimum of n numbers using the algorithm to find maximum of n numbers is shown below:

Algorithm minimum(n, a)

//Purpose: To find minimum of n numbers

//Input:
// n – number of elements
// a – the list of elements

//Output:
// minimum of n numbers

Step 1: [Negate all the given numbers]
for i ← 0 to n-1
a[i] ← -a[i]
end for

Step 2: [Compute maximum of negated numbers]
big ← maximum(n, a)

Step 3: [Negate big to get small number]
small ← -big

Step 4: [Finished]
return

Note: To find maximum of a function using minimization function, it is required to negate all the values and then find minimum value. To get the correct answer, it is required to negate the result.

Now, let us see "What is maximization problem? What is minimization problem?" Given a problem, if it is required to find a maximum of some function, the problem is called

maximization problem and if it is required to find a minimum of some function, the problem is called *minimization problem*.

Now, let us see "How to obtain the minimum of the function f(x)?" Suppose, we know the algorithm to find maximum of some function f(x) and we need to obtain minimum of the function f(x). This can be achieved using the following relation:

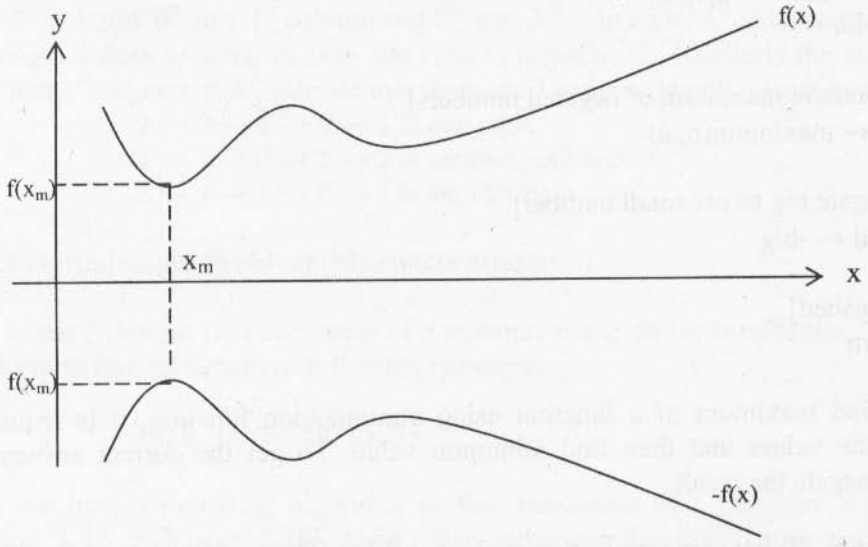
$$\min f(x) = -\max(-f(x))$$

It is clear from the above relation that to obtain a minimized function, it is required to negate all the values of f(x) and find maximum value. To get the minimum value it is required to change the sign of the maximum value.

Now, let us see "How to obtain the maximum of the function f(x)?" Suppose, we know the algorithm to find minimum of some function f(x) and we need to obtain maximum of function f(x). This can be achieved using the following relation:

$$\max f(x) = -\min(-f(x))$$

It is clear from the above relation that to obtain a result for maximum function, it is required to negate all the values of f(x) and find minimum value. Finally, to get the maximum value, it is required to change the sign of minimum value. The relationship between maximization and minimization problem can be graphically as shown below:



Note: Thus, a maximization problem can be reduced to an equivalent minimization problem and a minimization problem can be reduced to an equivalent maximization problem.

6.8.4 Optimization Problem (To find extremum points of a function)

Another problem which is based on problem reduction is the standard calculus procedure to find the extremum points of a function. This can be done very easily by finding the functions derivative $f'(x)$ and then solving the equation $f'(x) = 0$ and thereby obtaining the critical points. In other words, the problem of finding the extremum points is reduced to the problem of solving the equation $f'(x) = 0$, there by obtaining the critical points. Since there is no general method to solve and the procedure to standard calculus is not clearly defined, reducing this problem to find the extremum points and then finding the critical points will solve the problem and is much easier.

6.8.5 Linear Programming

Many optimization problems can be solved using linear programming very easily. This is achieved by optimizing a linear function consisting of several variables (parameters) subject to some constraints. These constraints will be expressed in the form of linear equations and linear inequalities. Solving these equations, it is possible to get the optimized solution. Many applications such as airline crew scheduling, transportation and communication network planning, oil exploration and refining etc., can be modeled using linear programming and optimization solution can be obtained. Using *simplex method* algorithm linear programming problems can be solved very easily.

Knapsack Problem

In this problem, we are given a knapsack (bag or container) of capacity M and n objects of weights $w_1, w_2, w_3, \ldots w_n$ with profits $p_1, p_2, p_3, \ldots p_n$. The main objective is to place the objects into the knapsack so that maximum profit is obtained and the number of objects chosen should not exceed the capacity of knapsack. The knapsack problem can be stated using two ways:

1. Continuous or fractional version
2. Discrete or 0/1 version

Continuous or fractional knapsack problem

Let $x_1, x_2, x_3, \ldots x_n$ be the fractions that are supposed to be added into the knapsack. If $x_i = 1$, it means that entire ith object is placed into the knapsack and the value $x_i = 0$ indicates that i^{th} object has not been selected. It is possible that the fraction of the item

can be selected in which the inequality $0 \le x_i \le 1$ must be satisfied. The main objective is to place the objects into the knapsack so that maximum profit is obtained and the number of objects chosen should not exceed the capacity of knapsack. This problem can be stated in *linear programming* as:

$$\text{maximize} \sum_{i=1}^{n} P_i X_i$$

subject to the constraints

$$\sum_{i=1}^{n} W_i X_i \le M$$

and

$$0 \le x_i \le 1$$

Discrete or 0/1 knapsack problem

The only difference between continuous knapsack problem and discrete knapsack problem is that, in continuous version the fractions of the objects can be selected i.e., $0 \le x_i \le 1$ whereas in discrete version each x_i can take either 0 or 1 i.e., we select the item fully or reject the item fully. This problem can be stated in *linear programming* as:

$$\text{maximize} \sum_{i=1}^{n} P_i X_i$$

subject to the constraints

$$\sum_{i=1}^{n} W_i X_i \le M$$

and

$$x_i = \{0, 1\}$$

6.8.6 Graph Problem

In varieties of graph applications, the finite automaton is very useful to get the solution to varieties of problems. In finite automaton, the vertices of a graph represent the possible states of the given problem while the edges indicate the transitions from one state to another state. One state is designated as the *start state* is identified by an arrow labeled *start* and another state is designated as *final state* is represented by two circles. There may be one or more final states. Such a graph is called *state-space graph* or *transition graph*. Thus, the transformation of a problem to a finite automaton reduces to another problem of finding the path from the *source state* to *final state* or *accepting state*.

Example: A peasant along with wolf, goat and cabbage is on one side of the river bank and he wants to take all three to the other side of the river bank. However, the boat has place for the peasant and one of the three. In the absence of the peasant the goat will eat cabbage and wolf will eat goat. It is required that peasant should take all the three to the other side of the river bank. Find the solution for this problem.

Solution: The *p*easant, *w*olf, goat and *c*abbage are denoted by *pwgc*. Let us assume that *pwgc* are on the left side of the bank and is denoted by

pwgc ||

where || represent the river. Finally all four namely *pwgc* should be on the other side of the bank and is denoted by

|| pwgc

Note: Suppose peasant and goat are on left side of the river bank and wolf and cabbage are on the right side of the river bank. This is denoted by "**pg || wc**".

The peasant should not leave *goat* and *cabbage* or *wolf* and *goat* on the same side in his absence. Also, at any point of time the boat can accommodate maximum of two occupants. The *transition diagram* or *state-space* graph which shows the solution is shown in figure 6.4.4.

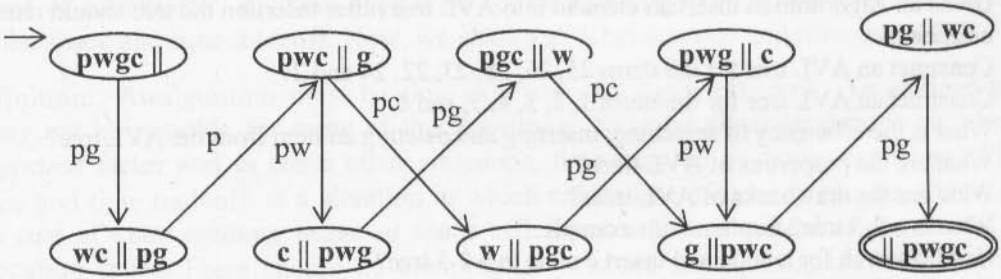


Figure 6.4.4 The peasant, wolf, goat and cabbage puzzle

A state represents who are all present on the left side of the bank and who are all present on the right side of the bank. The edges represent boats occupants while crossing the river. If we start from the start state and reach the final state, we get two paths which represent the two solutions to cross the river.

Exercises:

1. What is a 2-3 tree? Show that the time efficiencies of searching, insertion and deletion are all $\theta(\log n)$ in both worst case and average case
2. What is an AVL tree? Explain the need for rotation of AVL trees. Construct an AVL tree for the list 8. 9, 11, 6, 5, 7, 10 by successive insertions. Clearly show the steps.
3. What is transform-and-conquer? Explain along with example

4. What is instance simplification? What is representation change? What is problem reduction? Give examples for each
5. What is presorting? How presorting is used to improve the efficiency? Explain with an example
6. Using presorting, design the algorithm to find the uniqueness and mode obtain the time complexity.
7. What is the time complexity of binary search if we use pre-sorting. Explain.
8. Write the bottom-up-approach algorithm to sort the elements using heap sort. Obtain the time complexity.
9. Write the top-down-approach algorithm to sort the elements using heap sort. Obtain the time complexity.
10. Create a heap for the items 5, 35, 25, 45, 20, 55, 25, 45, 50, 10, 30 and 15 using bottom-up approach
11. Prove that the height of a heap with n nodes is equal to $\lfloor \log_2 n \rfloor$
12. Obtain the time complexity to create a heap using bottom up approach
13. Construct a heap for the list 50, 25, 30, 75, 100, 45, 80 by successive insertion using top-down approach
14. What is an AVL Tree? Explain with an example
15. Construct an AVL tree by inserting the elements 100, 200 300, 250, 270, 70 and 40 successively starting from an empty tree
16. What are the various steps involved in inserting a node into an AVL tree?
17. Given an Algorithm to insert an element into AVL tree (after insertion the tree should remain balanced)
18. Construct an AVL tree for the items 25, 26, 28, 23, 22, 24 and 27
19. Construct an AVL tree for the items 1, 2, 3, 4, 5, and 6
20. What is the efficiency of searching, inserting and deleting an item from the AVL tree?
21. What are the properties of AVL trees?
22. What are the drawbacks of AVL trees?
23. What is a 2-3 tree? Explain with example
24. How to search for a node and insert a node in a 2-3 tree?
25. Construct 2-3 tree for the list C, O, M, P, U, T, I, N, G
26. Construct 2-3 tree for the items 9, 5, 8, 3, 2, 4 and 7
27. Obtain the time complexity for creating, searching or deleting a node from a 2-3 tree
28. What is an AVL tree? Explain the need for rotation of AVL trees. Construct an AVL tree for the list 8. 9, 11, 6, 5, 7, 10 by successive insertions. Clearly show the steps.
29. What is a 2-3 tree? Explain search, insertion and deletion operations and show that these operations are all $\theta(\log n)$ in both worst and average case. **(July/Aug 2004)**
30. What is heap? Outline an algorithm to construct a heap. Derive the efficiency class of this algorithm **(July/Aug 2004)**
31. Explain the concept of 2-3 tree. How can keys be inserted into it? Comment on the efficiency of search operations on a 2-3 tree. **(Jan/Feb 2005)**
32. With the help of necessary algorithms, explain the bottom up heap sort method of sorting **(Jan/Feb 2005)**

Chapter 7: Space and Time Tradeoffs

What are we studying in this chapter?

- Concept of Transform and Conquer Technique
- Sorting by Counting
  - Comparison counting sort
  - Distribution counting sort
- Horspool's Algorithm for String Matching
- Hashing
  - Open Hashing (Separate Chaining)
  - Closed Hashing (Open addressing)

- 3 hours

7.1 Introduction

When we design algorithms, we must consider both time and space factors. The space and time tradeoff is an issue while computing. To achieve time efficiency "Is it possible to forego the space issues?" or To achieve space efficiency "Is it possible to forego the speed of execution?" or there should be compromise between space and time. This is called space and time tradeoff. Now, we shall see"What is space and time tradeoff?"

Definition: An algorithm must be time efficient and space efficient. To achieve both many not be possible for some of the algorithms. In some situations, space may be an important factor and in some other situations, time may be an important factor. Thus, space and time tradeoff is a situation in which either time efficiency can be achieved at the cost of extra memory usage or space efficiency can be achieved at the cost of the execution speed. There should be compromise between time and space based on the situation.

Normally time factor is preferred over space in most of the situations. Now, the question is"What are the methods using which time efficiency is achieved at the cost of space?" The time efficiency can be achieved using methods shown below:

Methods of achieving time efficiency by foregoing space

- → Input enhancement — Ex: Sorting by counting, Horspool's algorithm
- → Pre-structuring — Ex: Hashing, Indexing with B-Trees
- → Dynamic programming — Ex: Knapsack problem, Floyd's algorithm etc.

Now, let us see"What is input enhancement?"

7.2 Space and Time Tradeoffs

Definition: Given a problem and various inputs, the input is pre-processed to get additional information about the problem. The additional information thus obtained may be stored in the form of a table which may be used by an algorithm to get the required result with less time. This technique is called input enhancement In other words, getting additional information earlier to accelerate the time afterwards by preprocessing the given input is called input enhancement.

For example, sorting by counting algorithms such as Comparison counting sort and Distribution counting sort, pattern matching using Horspoo's algorithm etc uses input enhancement technique

Now, let us see "What is pre-structuring?"

Definition: Pre-structuring is a method of achieving time efficiency that uses extra space to facilitate faster and flexible accessing of data. Here, some processing is done before a problem in question is actually solved, but unlike input enhancement technique, it deals with accessing the data.

For example, hashing and B-trees use this technique to solve the problems

Now, let us see "What is dynamic programming?"

Definition: Dynamic programming is a method solving a problem by recording the solutions of overlapped sub problems of a given problem in a table and use these recorded solutions to get the solution for the given problem. When a sub problem of earlier instance is encountered, instead of recalculating the values, it is simply retrieved from the table thus saving the time. Thus, in dynamic programming we get the solution by retrieving the already computed values and use these values to get the solution for the given problem.

For example, computing binomial co-efficient, finding the transitive closure using Warshall's algorithm, The all pairs shortest path problem using Floyd's algorithm etc uses the concept of dynamic programming.

7.2 Sorting by counting

Sorting by counting is a method of arranging numbers in ascending order using input enhancement. The two sorting methods which use input enhancement strategy are:

- Sorting by comparison. This method is also called comparison counting method
- Sorting by distribution. This method is also called called distribution counting method or linear sorting

7.2.1 Comparison Counting

Now, let us discuss "What is sorting by comparison counting?"

Definition: Comparison counting is a method of sorting which works as shown below:

- **Step 1:** For each element a[i] in the given list, find the total number of elements say c[i] that are less than a[i]
- **Step 2:** The count c[i] obtained in previous step will be the position of a[i] in the final sorted list

Using the above procedure, we will be able to sort the list by simply copying the elements to their appropriate positions in a new, sorted list.

Design methodology Let us take an example. Consider the elements shown below:

	0	1	2	3	4	5
a	25	45	10	20	50	15

Step 1: Compute the number of elements less than each item as shown below:

	0	1	2	3	4	5
a	25	45	10	20	50	15
c	3	4	0	2	5	1

Number of elements less than 15
Number of elements less than 50
Number of elements less than 20
Number of elements less than 10
Number of elements less than 45
Number of elements less than 25

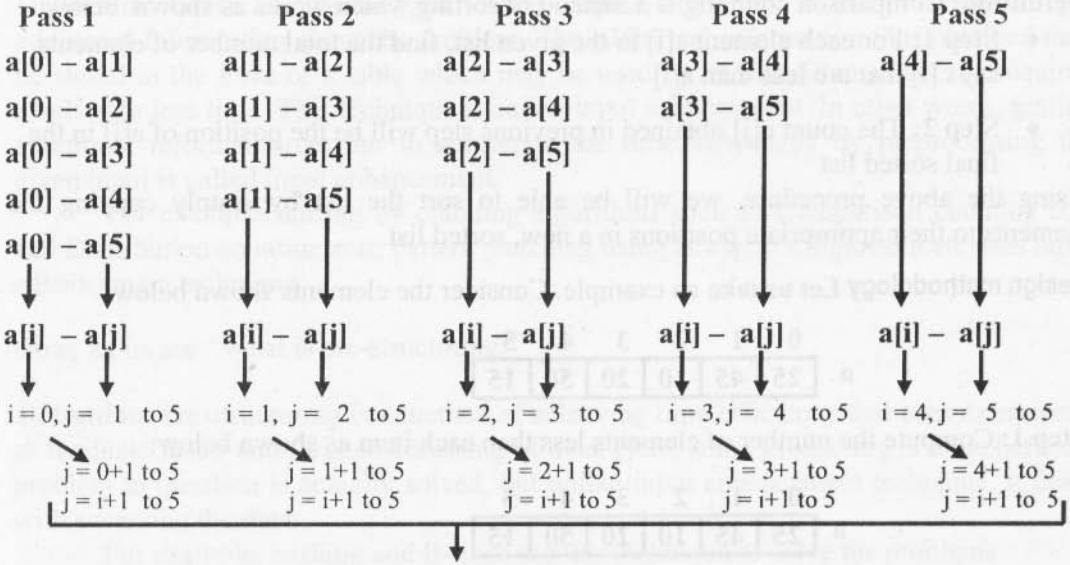
Now, let us see "How to compute the number of elements less than each element?" Suppose a[i] and a[j] are two numbers and c[i] and c[j] represent the number of elements less than a[i] and a[j] respectively. Now, carefully understand the following statement:

Suppose a[i] < a[j]

If above condition is true, there is one element a[i] which is less than a[j]. So, corresponding count c[j] has to be incremented by 1. Otherwise, corresponding count c[i] has to be incremented by 1. The equivalent statement is shown below:

```
if (a[i] < a[j])
        c[j] ← c[j] + 1
else                                    ..........(1)
        c[i] ← c[i] + 1
end for
```

Now, value of *i* and *j* can be obtained as shown below:



In each pass j = i + 1 to 5

j = i + 1 to 6 – 1 where 6 represent number of elements

$\boxed{\text{In general, } j = i + 1 \text{ to } n - 1}$ where n represent number of elements

In the above example, observe that i = 0, 1, 2, 3, 4

i = 0 to 4

i = 0 to 6 – 2 where 6 represent number of elements

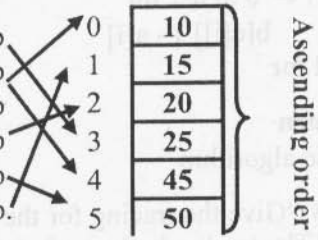
$\boxed{\text{In general, } i = 0 \text{ to } n - 2}$ where n represent number of elements

Note: Observe from above results that the index variable *i* varies less frequently when compared with index variable *j*. So, *i* should be in the outer loop and *j* should be in the inner loop. The statement shown in equation (1) can be written as shown below:

```
for i ← 0 to n – 2
    for j ← i + 1 to n – 1
        if (a[i] < a[j])
            c[j] ← c[j] + 1
        else                                    .....(2)
            c[i] ← c[i] + 1
        end if
    end for
end for
```

Step 2: Insert each item at the corresponding position in array b

Insert a[0] = 25 into position 3 of array b
Insert a[1] = 45 into position 4 of array b
Insert a[2] = 10 into position 0 of array b
Insert a[3] = 20 into position 2 of array b
Insert a[4] = 50 into position 5 of array b
Insert a[5] = 15 into position 1 of array b



In general, insert a[i] into position c[i] in array b which can be written as shown below:

b [c[i]] ← a[i] for i = 0 to 5
b [c[i]] ← a[i] for i = 0 to 6 – 1 where 6 represent number of elements

In general, $\boxed{\text{b [c[i]] ← a[i] for i = 0 to n – 1}}$ where n represent number of elements

The above statement can be written as shown below:

```
for i ← 0 to n – 1
        b[c[i]] ← a[i] ..............................(3)
end for
```

The complete algorithm to sort the elements using *sorting by counting* method can be written using the statements labeled with (2) and (3) as shown below:

```
ALGORITHM Comparison_Countint(a, n, b)
//Purpose:    To sort n elements stored in an array : a
//Input:
//      a- the items to sort
//      n – the number of items to sort
//Output:
//      b – contains the sorted list
        for i ← 0 to n-1 do c[i] ← 0            // Initialize count of each element
        for i ← 0 to n-2 do                     // Compute number of elements
                for j ← i + 1 to n-1 do         // less than each element
                        if ( a[i] < a[j])
                                c[j] ← c[j]+1
                        else
                                c[i] ← c[i] + 1
                        endif
                end for
        end for
```

```
// Insert each item at the appropriate position
for i ← 0 to n-1 do
        b[c[i]] ← a[i]
end for
return
// end of the algorithm
```

Now, let us "Give the tracing for the above algorithm to sort the elements 25, 45, 10, 20, 50 and 15" The tracing is shown below:

	0	1	2	3	4	5	
a	**25**	**45**	**10**	**20**	**50**	**15**	Elements to sort
index	0	1	3	4	5	5	
C	**0**	**0**	**0**	**0**	**0**	**0**	Initial count of each item
When i = 0, i.e, pass 0	**3**	**1**	**0**	**0**	**1**	**0**	Count of each item in each pass
When i = 1, i.e., pass 1		**4**	**0**	**0**	**2**	**0**	
When i = 2, i.e., pass 2			**0**	**1**	**3**	**1**	
When i = 3, i.e., pass 3				**2**	**4**	**1**	
When i = 4, i.e., pass 4					**5**	**1**	
C	**3**	**4**	**0**	**2**	**5**	**1**	Final count of each item

	0	1	2	3	4	5	
B	**10**	**15**	**20**	**25**	**45**	**50**	Sorted list

Analysis The time complexity in the best case, worst case remains same.

Step 1: The parameter to be considered is *n*, which is a measure of input's size

Step 2: The basic operation is the statement "if (a[i] > a[j])"

Step 3: Each time control enters into loop, the above basic operation is executed once. So, the total number of times the basic opoeration is executed can be obtained as shown below:

```
for i ← 0 to n-2 do
    for j ← i +1 to n-1 do
        if ( a[i] < a[j])
```

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

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

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= (n-1)+(n-2)+.....3+2+1$$

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

So, $f(n) = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} \approx n^2$ for very large value of n

So, time complexity of comparison counting algorithm is $\theta(n^2)$

Now, the question is "Will the comparison counting algorithm work correctly for arrays with equal values?" Let us consider the elements 10, 20, 10, 20, 10, 5 and tracing for these elements is shown below:

	0	1	2	3	4	5	
a	10	20	10	20	10	5	Elements to sort
C	0	1	2	3	4	5	
	0	0	0	0	0	0	Initial count of each item
When i = 0, i.e, pass 0	3	1	0	1	0	0	Count of each item in each pass
When i = 1, i.e., pass 1		5	0	1	0	0	
When i = 2, i.e., pass 2			2	2	0	0	
When i = 3, i.e., pass 3				4	0	0	
When i = 4, i.e., pass 4					1	0	
C	3	5	2	4	1	0	Final count of each item

	0	1	2	3	4	5	
B	5	10	10	10	20	20	Sorted list

So, it is evident from the above example that, the comparison counting algorithm in fact works for arrays with equal values.

Now, the question is "Is the algorithm stable?" **No**, since the relative position of duplicate elements is not same before and after sorting, the algorithm is not stable. (See above tracing)

Now, the question is "Is the algorithm in place?" **No.** Since this algorithm uses extra space (array) for the count of each item and another array to hold the sorted list, the algorithm is not in place.

Advantages

- Very simple algorithm

Disadvantages

- Uses more space. Other than the array for given elements, it uses array for the count of each item and another array to hold the sorted list. So, the algorithm is not in place.
- It is not as efficient as quick sort, merge sort or heap sort whose time complexities are n log n
- Its time complexity is $\theta(n^2)$

7.2.2 Sorting by Distribution Counting

Now, let us see "What is sorting by distribution counting?"

Definition: Distribution counting is a method of sorting which works as shown below:

- **Step 1:** The frequency of each item and accumulated frequency of each item is calculated
- **Step 2:** For each item a[i] in the array (access from last item to first item)

  Obtain the distribution value of a[i] say d[j]

  decrement distribution value d[j] by 1

  Insert a[i] into array B at position givey by d[j]

  end for

Now, let us see "What type of elements can be sorted using distribution method?" The input array may contain duplicate elements. But, the elements of the array should be such that all the items within the lower bound (least value) and upper bound (highest value) should be in sequence.

For example, the elements 15, 13, 10, 14, 11, 12 can be sorted. This is because, least value is 10 and highest value is 15 and the numbers within 10 and 15 are present in the list. The elements 15, 13, 10, 12 can not be sorted. This is because, in the above list, least number is 10 and highest number is 15. But, between 10 and 15, the number 11 is missing and hence can not be sorted.

Design methodology Let us take an example. Consider the elements shown below:

a	0	1	2	3	4	5	6	7	8
	12	13	10	12	10	12	11	10	13

Note that the entire given list consists of items from the set {10, 11, 12, 13}. Here, the least element is 10 and highest element is 13. We can call these two elements as lower bound denoted by **LB** and upper bound denoted by **UB.** So, **LB = 10** and **UB = 13.**

Step 1: Let us find the frequency and accumulated frequency of each item of the set **{10, 11, 12, 13}**

Note: LB = 10 and UB = 13

Index	0	1	2	3
Array elements	10	11	12	13
Frequency	3	1	3	2
Accumulated frequency	3	4	7	9

Distribution values (d)

Index	0	1	2	3
D	3	4	7	9

Using the lower bound and upper bound the final index is given by UB – LB. The initial frequency of each item is 0. This can be achived using

```
for i ← 0 to UB-LB do          Note: i ← 0 to 3 i.e., 0 to UB-LB
    d[i] ← 0                   ..........................(1)
end for
```

The index *j* in array d for the item a[i] can be obtained using the relation:

j ← a[i] – LB (**Ex:** If a[i] is 13, index of 13 in array d = 13-10 = 3)

The frequency of each item can be obtained using the code

```
for i ← 0 to n-1 do
    j ← a[i] – LB              ..........................(2)
    d[j] ← d[j] + 1
end for
```

The accumulated frequencies can be obtained by adding the frequency of current item with the accumulated frequency using the code

```
for i ← 1 to ub-lb do
    d[i] ← d[i] + d[i-1]       ..........................(3)
end for
```

Note: In general, the distribution values indicate the positions of last occurrences of their elements in the final sorted list.

Step 2: Now, let us see how the distribution values of each item help us to sort the following elements more efficiently:

12, 13, 10, 12, 10, 12, 11, 10, 13

Let us process the given elements from right to left and insert into B using the distribution values stored in D as shown below:

Item is a[8] = 13. Using 13, the index to its distribution value = 13 – LB
= 13 – 10
= 3

Now, decrement d[3] by 1 i.e., 9 in d[3] is changed to 8. Now, insert item 13 into B[8] as shown in table.

Item is a[7] = 10. Using 10, the index to its distribution value = 10 – LB
= 10 – 10
= 0

Now, decrement d[0] by 1 i.e., 3 in d[0] is changed to 2. Now, insert item 10 into B[2] as shown in table.

Item is a[6] = 11. Using 11, the index to its distribution value = 11 – LB
= 11 – 10
= 1

Now, decrement d[1] by 1 i.e., 4 in d[1] is changed to 3. Now, insert item 11 into B[3] as shown in table.

.....

.....

Item is a[0] = 12. Using 12, the index to its distribution value = 12 – LB
= 12 – 10
= 2

Now, decrement d[2] by 1 i.e., 5 in d[2] is changed to 4. Now, insert item 12 into B[4] as shown in table. Finally, all the elements are arranged in ascending order.

The completing tracing is shown using the following table:

Given list

A[8]	=	13
A[7]	=	10
A[6]	=	11
A[5]	=	12
A[4]	=	10
A[3]	=	12
A[2]	=	10
A[1]	=	13
A[0]	=	12

← D →

0	1	2	3
3	4	7	9
3	4	7	8
2	4	7	8
2	3	7	8
2	3	6	8
1	3	6	8
1	3	5	8
0	3	5	8
0	3	4	7

← B →

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
								13
		10						
			11					
						12		
	10							
					12			
10								
							13	
				12				

10	10	10	11	12	12	12	13	13

Sorted list B

Thus, the given elements can be arranged in ascending order using distribution counting method.

Once the distribution values are known, for each item a[i], the position of distribution value is given by the following relation:

$$j \leftarrow a[i] - LB$$

Now, decrement the distribution value at d[j] using the relation:

$$d[j] \leftarrow d[j] - 1$$

Then, insert the item a[i] in array b at the position d[j] using the statement:

$$b[d[j]] \leftarrow a[i]$$

Note: The above statements should be executed for each i = 8, 7, 6, 5, 4, 3, 2, 1, 0

i.e, $i \leftarrow 8$ down to 0

$i \leftarrow 9 - 1$ down to 0

$i \leftarrow n - 1$ down to 0 where n is number of elements in given array.

The pseudocode to insert an item a[i] at the position d[j] in array b can be written as shown below:

```
for i ← n-1 downto 0 do
    j ← a[i] – LB                    ..........................(4)
    d[j] ← d[j] – 1
    b[d[j]] ← a[i]
end for
```

The complete algorithm can be written by combining the statements labeled (1), (2), (3) and (4) as shown below:

```
ALGORITHM Distribution_Counting_Sort(a, n, b)
//Purpose:    To sort n elements stored in an array using distribution counting

//Input:      a- the items to sort
//            n – the number of items to sort

//Output:     b – contains the sorted list

    // Compute lower bound lb and upper bound ub
    LB = minimum(a, n)
    UB = maximum(a, n)

    for i ← 0 to ub-lb do          // Initialize frequencies of each item
        d[i] ← 0
    end for
    for i ← 0 to n-1 do            //Compute frequencies of each item
        j ← a[i] – lb
        d[j] ← d[j] + 1
    end for

    for i ← 1 to ub-lb do          //Compute sum of accumulated frequencies
        d[i] ← d[i] + d[i-1]
    end for

    // Access item and insert into sorted vector based on the distribution value
    for i ← n-1 downto 0 do
        j ← a[i]-lb                // Obtain the index of distribution value
        d[j] ← d[j] -1             // decrement the distribution value of the item
        B[d[j]] ← a[i]             // Insert into the appropriate position in array b
    end for
// end of algorithm
```

Analysis The time efficiency can be computed as shown below:

Step 1: The parameter to be considered is *n*, which is a measure of input's size

Step 2: The statements within the final *for* loop can be considered as basic operations and each time the control enters into the loop, the statements are executed only once. So, the total number of times the basic operations are executed can be obtained as shown below:

```
for i ← n-1 downto 0 do
    j ← a[i]-lb
    d[j] ← d[j] -1
    B[d[j]] ← a[i]
```

$$f(n) = \sum_{i=0}^{n-1} 1 = n - 1 - 0 + 1 = n$$

So, time complexity of sorting by distribution method is = $\theta(n)$

Note: Now, let us see "Why this algorithm is called linear sorting algorithm?" Since the time complexity is O(n), as the value of *n* increases, the time taken by the algorithm also increases linearly and hence this algorithm is called linear sorting algorithm.

Advantages

- The time efficiency is linear i.e., O(n)
- This is the most efficient sorting method even compared to merge sort, quick sort and heap sort

Disadvantages

- It can not be applied for all the input values
- It requires additional storage space to store the distribution value in array d and sorted elements in array b.

7.3 String Matching

Now, let us see "What is string matching?"

Definition: Given a string called *text* with *n* characters and another string called *pattern* with *m* characters where $m \leq n$. It is required to search for the string *pattern* in the string *text*. If search is successful return the position of the first occurrence of *pattern* string i the *text* string. Otherwise, return -1. This process of searching for a *pattern* string in a given *text* string is called *string matching* or *pattern matching*.

7.14 Space and Time Tradeoffs

The pattern matching technique along with design and time complexity is already discussed in the section 3.5 which uses *brute-force* technique. Note that the time complexity of this algorithm in the worst case is given by $\theta(mn)$ where *m* is the length of the pattern string and *n* is the text string. In the best case and average case, the time complexity is given by $\theta(n)$.

The better algorithms compared to brute-force technique have been discovered. Most of these algorithms uses *input enhancement* technique. The *pattern* string is pre-processed to get some information about the pattern string and the information is stored in a table. The information stored in the table can be used by the algorithm to effectively search for the pattern string in a given *text* string. The various algorithms which uses the *input enhancement* technique for string matching are:

- Knuth-Morris-Pratt algorithm
- Boyer-Moore algorithm
- Horspool's algorithm

7.3.1 Horspool's Algorithm

Now, let us "State Horspool's algorithm for pattern matching" The Horspool's algorithm is very simple and efficient pattern matching algorithm. The technique used here is to first align the *pattern* string to the beginning of the *text* string. But, instead of comparing from left to right, we start comparing from right to left. This results in two cases:

Case 1: If all the characters of the pattern string matched with the corresponding characters of the text string and found same, then search is successful. We can return the position from where pattern string matched.

Case 2: If there is a mismatch, then the pattern string has to be shifted towards right. How many positions the pattern string has to be moved towards right is depends on the shift value present in the shift table for the corresponding character of the text string. Like this, keep shifting the pattern string from left to right as long as there is no match. Finally, after aligning even with end of text string, if there is a mismatch, display the message "pattern string not found".

Now, let us see "How to compute the shift table?" The shift for each and every character can be obtained using the following relation:

$$S[c] = \begin{cases} m = \text{length(pattern)} & \text{If char c is not in first m-1 characters of pattern string} \\ \text{Distance of each right most character in the first m-1 characters of pattern string from the last character of pattern string, otherwise} & \end{cases}$$

Example 1: Compute the shift table for the pattern string BARBER.

Solution: Suppose the search characters in the text string consists of only alphabetical characters from 'A' to 'Z'. Here, pattern string is "BARBER". Let *m* denote the length of the pattern string i.e., m = LENGTH("BARBER") which is 6. To start with fill the entire table with length of the patter string which is 6 as shown below:

Note: 0, 1, 2, 3, ...127 represent the ASCII values of characters

Index	0	1	...	A	B	C	D	E	F	.....	p	q	r	....	x	y	z	.....	127
Length of pattern	6	6	...	6	6	6	6	6	6	.....	6	6	6	.....	6	6	6	.....	6

The pseudocode to fill the above table is shown below:

```
m ← length(p)          // Compute the length of pattern string
for i ← 0 127
        S[i] ← m       // Fill the initial shift table with length of pattern string
end for
```

To compute the number of shifts required during searching, consider the pattern string BARBER and the distance of each character from the right most character can be obtained as shows below:

B A R B E R

- E → is at a distance **1** from right end
- R → is at a distance **2** from right end
- B → is at a distance **3** from right end
- A → is at a distance **4** from right end
- B → B value from right end is already calculated. Use that value

Observe from above figure that value for E is 1, for B it is 2, for R it is 3 and for A it is 4. Copy these values in to above table. The resulting table is shift table which gives the number of shifts required for each character of the text string. The shift table is shown below:

Note: 0, 1, 2, 3, ...127 represent the ASCII values of characters

Index	0	1	...	A	B	C	D	E	F	.....	P	Q	R	....	X	Y	Z	.....	127
Length of pattern	6	6	...	4	2	6	6	1	6	.....	6	6	3	.....	6	6	6	.....	6

index →	0	1	2	3	4	5
Pattern string P →	B	A	R	B	E	R

Access each character of the pattern string and find how much distance is that character from last character. Note the following facts from above figure and verify:

p[0] = B is at a distance of 5 = 5 – 0 from the end
p[1] = A is at a distance of 4 = 5 – 1 from the end
p[2] = R is at a distance of 3 = 5 – 2 from the end
p[3] = B is at a distance of 2 = 5 – 3 from the end
p[4] = E is at a distance of 1 = 5 – 4 from the end

In general, p[i] is at a distance of 5 – i from the end
i.e., p[i] is at a distance of 6 – 1 – i from the end
i.e., p[i] is at a distance of **m – 1 – i** from the end

Note: The index i = 0 to 4
i = 0 to 6 – 2
i = 0 to m – 2 where m is length of pattern string.

Note: m – 1 – i should be copied into table S at an index p[i] where i = 0 to m – 2. This can be achieved using the following pseudocode.

```
for i ← 0 to m – 2
        S[p[i]] ← m – 1 – i
end for
```

The complete algorithm to compute shift table for the pattern string is shown below:

ALGORITHM Shift_Table(p, s)

//Purpose: To fill the shift table, based on the pattern string

//Input: p – pattern string to be searched

//Output: t – shift table is returned through parameter

```
m ← length(p)          // Obtain the length of the pattern string
for i ← 0 to 127 do    // Initially fill the table with length of pattern string
        s[i] ← m
end for

for i ← 0 to m-2 do    // Fill the table with number of shifts required
        s[p[i]] ← m-1-i
end for

return
// end of the algorithm
```

Once the shift table is ready, the input enhancement phase is over and using this table, we can efficiently search for the pattern string using the method given by R. Horspool and hence the name Horspool's algorithm. The general procedure to search for the pattern string using Horspool's algorithm is shown below:

Step 1: Compute the shift table

Step 2: Align the pattern string against the beginning of the text string

Step 3: As long as there is no match or end of text string perform the following steps:

- Start comparing from last character of the pattern string with substring of the text string
- If there is match, return the position of the pattern string in the text string.
- Otherwise, take the last character of substring of text string that is being matched with pattern string which becomes an index into table S. Using this index obtain the value from shift table. This value gives the number of shifts required. Then move the pattern string towards right by the value obtained from the shift table

Design Let us consider the situation shown below. Here, the pattern string "ABOUT" is aligned to text string at position *i* of text string from right.

Position of text string from where searching starts from left to right

i

Index →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Text(t)	W	E	-	K	N	E	W	-	A	B	O	U	T	-	S	H	A	S	T	R	Y

Pattern(p)	A	B	O	U	T
Index →	0	1	2	3	4

In Horspool's algorithm, the comparison starts from right to left. The character of each pattern string is matched with corresponding character of text string as shown below:

t[12] is compared with p[4]		t[12 - 0] is compared with p[4-0]
t[11] is compared with p[3]	Can be	t[12 - 1] is compared with p[4-1]
t[10] is compared with p[2]	written as	t[12 - 2] is compared with p[4-2]
t[09] is compared with p[1]		t[12 - 3] is compared with p[4-3]
t[08] is compared with p[0]		t[12 - 4] is compared with p[4-4]

i.e., t[12 – k] is compared with p[4-k]

i.e., t[12 – k] is compared with p[5-1-k]

In general, t[i – k] is compared with p[m-1-k]

where k = 0 and final value of k = 4

k = 5 – 1

k = m – 1

Hence, initial value of k = 0 and final value of k = m - 1

In general, t[i-k] and p[m-1-k] are compared repeatedly as long as the corresponding characters are same and as long as k <= m -1. If the above condition is true, compare the next character by incrementing k by 1. The pseudocode is shown below:

```
k ← 0
while ( k <= m-1 and t[i-k] = p[m-1-k] ) do
        k ← k + 1
end while
```

If the pattern string is found in the text string the value of *k* will be equal to *m* and control comes out of the **while** statement. Then, we can say that pattern string found. Now, let us see how to find the position of pattern string if search is successful. Observe that, in the given example pattern string starts from position = 8

= 12 – 4 (expressing in terms of i)

= 12 – (5 – 1)

= 12 – 5 + 1

= i – m + 1

So, in general the position of the pattern string if found is given by i – m + 1

The equivalent statement can be written as

if (k = m) **return** i – m + 1

If the above condition fails, it is required to shift the pattern string by the desired positions towards right side . The shift value can be obtained by using the character in the text string at the position *i* i.e., t[i] and then obtain the shift value from the table *s* using t[i] as the index. Thus, s[t[i]] gives the shift value and this value should be added to *i* so that the pattern string is aligned to the text string. This is obtained using the statement:

i ← i + s[t[i]]

So, the complete code can be written as

```
k ← 0
while ( k <= m-1 and t[i-k] = p[m-1-k] ) do
        k ← k + 1
end while

if ( k = m) return i – m + 1

i ← i + s[t[i]]
```

and note that the above statements should be executed as long as the string is found in text string or it reaches the length of the text string. So, the complete algorithm can be written as shown below:

```
ALGORITHM Horspools_Pattern_Matching(p, t)
//Purpose:   To check whether the pattern string is present in the text string

//Input:     p – pattern string to be searched
//           t – text string where searching takes place

//Output:    Position of pattern string p in the text string t if search successful
//           -1 Otherwise

    //Compute the necessary shifts to be taken
    Shift_Table(p, s)        // s is the table which contains the shifts required

    // Align the pattern string with the text string
    n ← length(t)            // Compute length of text string
    m ← length(p)            // Compute length of pattern string
    i ← m – 1                // Position of pattern string at the beginning of text string
```

```
//Search for the pattern string in text string
while ( i <= n-1 ) do
        k ← 0

        while ( k <= m-1 and  t[i – k] = p[m –1– k]) do
                k ← k + 1                    // Matching in progress
        end while

        if ( k = m ) return i – m + 1   // Return the position of the pattern string

        i ← i + s[t[i]]                      // Align pattern string using next farther shift

end while

return -1        // Pattern not found in the text string
// end of algorithm
```

Analysis The time efficiency can be obtained as shown below:

Step 1: The parameter to be considered are *m* and *n*, which are a measure of input's size namely *pattern* string and *text* string.

Step 2: The basic operation is the comparative statement "p[m-1-k] = t[i-k]" in the while loop

Step 3: The total number of times the *basic operation* is executed depends on the number of times the two loops are executed and can be obtained as shown below:

```
……..
……..
i ← m – 1
while ( i <= n-1 ) do
        k ← 0
        while ( k <= m-1 and  t[i – k] = p[m –1– k]) do
```

$$f(n) = \sum_{i=m-1}^{n-1} \sum_{k=0}^{m-1} 1$$

$$= \sum_{i=m-1}^{n-1} m-1-0+1$$

$$= \sum_{i=m-1}^{n-1} m$$

$$= m \sum_{i=m-1}^{n-1} 1$$

$$= m(n - 1 - m + 1 + 1)$$

$$= m(n - m + 1)$$

i.e., $f(m,n) = mn - m^2 + m$

Worst case: This case occurs if the pattern string is found at the end of the text string or it is not present. Using the above relation for very large value of n

$$f(m,n) = mn - m^2 + m \approx mn$$

So, time complexity for pattern matching in the worst case is given by = $\theta(mn)$

Best case: Pattern string found in the beginning. In that case number of comparisons will be m. So, the best case time efficiency is given by $\Omega(m)$

Example 2: "Apply Horspool's algorithm to search for the pattern BARBER in the text JIM-SAW-ME-IN-BARBER-SHOP"

Solution: It is given that pattern string = "BARBER" whose length is 6. To start with fill the entire table with length of the patter string which is 6 as shown below:

Note: 0, 1, 2, 3, ...127 represent the ASCII values of characters

Index	0	1	…	A	B	C	D	E	F	…..	p	q	r	….	x	y	z	…..	127
	6	6	…	6	6	6	6	6	6	…..	6	6	6	…..	6	6	6	…..	6

Length of pattern

For the pattern string BARBER the distance of each character from the right most character can be obtained as shows below:

B A R B E R

- E → is at a distance 1 from right end
- R → is at a distance 2 from right end
- B → is at a distance 3 from right end
- A → is at a distance 4 from right end
- B → B value from right end is already calculated. Use that value

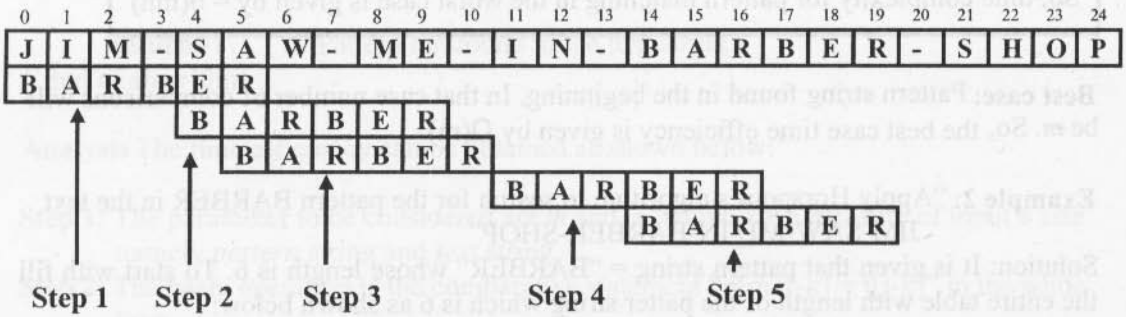
Observe from above figure that value for E is 1, for B it is 2, for R it is 3 and for A it is 4. Copy these values in to above table. The resulting table is shift table which gives the number of shifts required for each character of the text string. The shift table is shown below:

Note: 0, 1, 2, 3, ...127 represent the ASCII values of characters

Index	0	1	...	A	B	C	D	E	F	.....	P	Q	R	....	X	Y	Z	.....	127
	6	6	...	4	2	6	6	1	6	.....	6	6	3	.....	6	6	6	.....	6

Length of pattern

The pattern string can be aligned to the left of the text string and can be compared as shown below:



Step 1: Align BARBER to the beginning of text string. The characters A and R are not equal. So, take character A and use it as index in shift table S i.e.,S[A] is 4. So, shift patter string towards right by 4 positions

Step 2: Compare E and R. They are not equal. So, take character E and use it as index in shift table S i.e., S[E] is 1. So, shift the pattern string towards right by 1 position

Step 3: Compare symbol '–' with R. They are not equal. So, take character '–' and use it as the index in shift table S i.e.,S[–] is 6. So, shift pattern string towards right by 6 positions

Step 4: Compare R and R they are equal. So, move left and compare A and E. They are not equal. Now, take R last character of substring in text string and use it as the index in shift table S i.e., S[R] is 3. So, shift pattern string towards right by 3 positions.

Step 5: After comparing, it is found that all characters of pattern string are found and hence there is a match. So, pattern string found

7.3.2 Boyer-Moore Algorithm

It is similar to Horspool's algorithm, but it is slightly complicated. It is discussed in APPENDIX A

7.4 Hashing

Now, let us see "What is hashing? What are the different types of hashing techniques?"

Definition: Hashing is the process of transforming a string of characters into a short integer value (usually called key). This short integer value is used as index to get the original string. Hashing is used to retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value. It is an efficient way of implementing dictionaries be performing the following operations:

- Searching: To search for an item in the given set. Each entry in the set may may have many fields such as name, id, roll_no etc.
- Insertion: An item can be inserted into a given set
- Deletion: An item can be deleted from the set

Now, let us see "What is hash function? What is hash value? What is hash table?"

Definition: A function that is used to generate a short integer (also called key) can be used as an index into a table using which the data can be retrieved. This short integer is called hash value and the function that is used to generate a hash value is called hash function. The table on which insertion, deletion and retrieve operations takes place after computing the hash value is called *hash table*. A *hash table* can be implemented usually using an ordinary array H[0..m-1]. The size of the hash table is limited and so it is necessary to map the given data into this fairly restricted set of integers.

Note: A good hash function should satisfy two criteria:

- A *hash function* should generate the hash addresses such that all the keys are distributed as evenly as possible in the hash table.
- Computation of a key by hash function should be simple.

Now, let us see "What is collision?"

Definition: If the size of the hash table is fixed and the size of the table is smaller than the total number of keys generated, then two or more keys will have the same hash address. This phenomenon of two or more keys being hashed to the same location of hash table is called collision.

Example 1: Create the hash table for the items 15, 11, 22, 14 and 33 where size of table is 5. Assume the hash function = item % size_of_table.

Solution: Let us compute the hash value for each item.

Item = 15. Hash value = 15 % 5 = 0. So, insert 15 into a[0]
Item = 11. Hash value = 11 % 5 = 1. So, insert 11 into a[1]
Item = 22. Hash value = 22 % 5 = 2. So, insert 22 into a[2]
Item = 14. Hash value = 14 % 5 = 4. So, insert 14 into a[4]
Item = 33. Hash value = 33 % 5 = 3. So, insert 33 into a[3]

Inserting these values at the specified position, the contents of hash table are shown below:

	0	1	2	3	4
a	15	11	22	33	14

Hash table

Now, suppose we want to insert an item 37 into above table. Its hash value = 37 % 5 = 2 i.e., item 37 has to be inserted into a[2]. But, there is an item already in that location. This results in collision. We say that the second item is hashed into value which already exists. This is called collision.

Note: We can expect collision even if the number of keys generated is less than the size of the hash table. Most of the time collision can not be avoided and in the worst case all keys may have the same value.

Now, the question is "How to avoid collision?" The various hashing techniques using which collision can be avoided (collision resolution techniques) are shown below:

Collision resolution techniques → Open hashing (Separate chaining)
Collision resolution techniques → Closed hashing (Open addressing)

7.4.1 Open Hashing / Separate Chaining

Now, let us see "What is the technique used in open hashing to avoid collision?" The open hashing technique which is also called separate chaining is a very simple method using which collisions can be avoided. This is achieved using an array of linked lists. For

a given item, find the hash value using appropriate hash function. This hash value is used to find the index of the list to which the item is to be added. The item can be added at the end of the linked list using insert_rear() function. If more than one key has same hash value, all the keys hashed into same locations will be inserted at the end of the list and thus collision is avoided.

Now, let us see "How to search for the item using open hashing?" The searching using this technique takes less time. When an item has to be searched, we find the hash value of the item using hash function. This hash value corresponds to an index which will be used to obtain the address of the list. If the address of the list is NULL, the item we search for is not present and report unsuccessful search. If the address of the list is not NULL, that list is traversed sequentially to search for the item. If an item is found in the list report successful search and if end of list is encountered report unsuccessful search.

Now, the question is "How to delete an entry from the hash table which is created using open hashing technique?" Deleting a node from the hash table is also simple. The hash function is used to determine the hash value which can act as an index of the item to be deleted. The linked list corresponding to the index is searched and if the item is present, the corresponding node in that list can be deleted. During searching if end of list is encountered, the item not found and report that it is not possible to delete as the item not found.

Note: The only disadvantage of this technique is that linked list requires extra storage space for the pointers because of linked lists.

Example 2: Let the size of hash table is 5. Insert the following words LIKE, A, TREE, YOU, FIRST, FIND, A, PLACE, TO, GROW, AND, THEN, BRANCH, OUT into a hash table using appropriate hash function and show how each item is inserted into the table, how an item can be searched and deleted

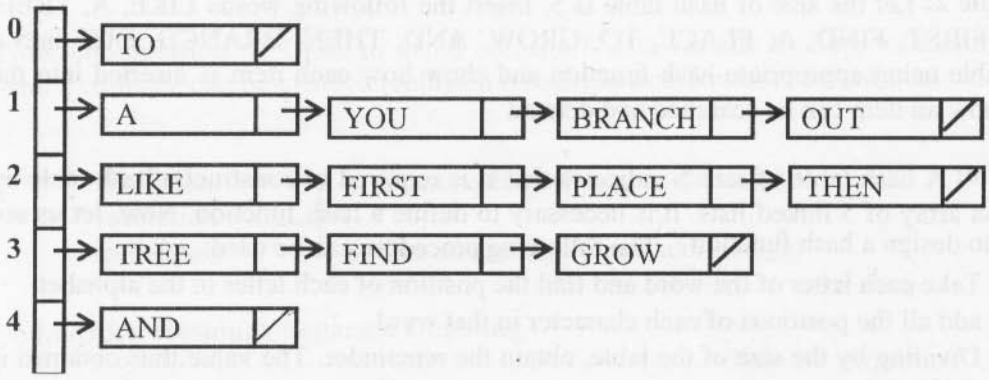
Solution: A hash table of size 5 indicates that it is required to construct a hash table by using an array of 5 linked lists. It is necessary to define a hash function. Now, let us see "How to design a hash function?" The following procedure can be used:

- Take each letter of the word and find the position of each letter in the alphabet
- add all the positions of each character in that word
- Dividing by the size of the table, obtain the remainder. The value thus obtained is the hash value

Using above procedure, we compute the hash value for each word as shown below:

Words	Sum of positions	Hash value sum %5
LIKE	12+9+11+5 =37	2
A	1 =1	1
TREE	20+18+5+5 =48	3
YOU	25+15+21 =61	1
FIRST	6+9+18+19+20 =72	2
FIND	6+9+14+4 =33	3
A	1 =1	1
PLACE	16+12+1+3+5 = 37	2
TO	20+15 =35	0
GROW	7+18+15+23 =63	3
AND	1+14+4 = 19	4
THEN	20+8+5+14 =47	2
BRANCH	2+18+1+14+3+8 =46	1
OUT	15+21+20 =56	1

Once we know the hash value for each word, now the question is "How to construct the hash table?" Since the size of the hash table is 5, we will have an array of 5 rows with each row having a linked list. Obtain the words one by one, find the hash value from the above table and insert at the end of the list in the appropriate location in the array. The hash table obtained is shown below:



Observe that hash value of word 'TO' is 0. So, it is inserted into h[0]. The hash value of the words 'LIKE', 'FIRST', 'PLACE' and 'THEN' is 2 and the words are inserted into h[2] at the end of the list one by one as and when they are scanned. Thus, a hash table can be created.

Now, let us see "How to search for the item in the table?" For example, if the word to be searched is "RAMA", find the hash value which is given by (18+1+13+1) % 5 = 33 % 5 = 3. So, it is required to search for "RAMA" in h[3]. In the list identified by h[3] has the words "TREE", "FIND" and "GROW". The string "RAMA" is compared with all the words in the list. Since there is no matching word, unsuccessful message has to be displayed and if found, display the appropriate message. Thus, string search can be achieved very efficiently using *hashing technique*.

7.4.2 Closed Hashing (Open Addressing)

Now, let us see "What is closed hashing?" In closed hashing or open addressing hashing, the amount of space available for storing various data is fixed at compile time by declaring a fixed array for the hash table. So, all the keys are stored in this fixed hash table itself without the use of linked lists. In such a table, collision can be avoided by finding another, unoccupied location in the array. The various strategies using which collision can be avoided are:

- Linear probing
- Double hashing

7.4.2.1 Linear probing

Now, let us see "What is linear probing?"

In this technique, if an item is to be inserted its hash value is computed. If the position identified by the hash value is empty, an item is inserted at that location. In case, at that position if an item already exists, then a collision occurs and a next empty location has to be searched and the item has to be inserted at the empty location. This way of avoiding the collision is called linear probing.

Now, "How to insert an item into hash table using linear probing?" If collision occurs at *i*th position, the locations from i+1 onwards are verified for empty locations and whenever we get an empty location, item is inserted at that position.

Now, let us see "How to search for an item using linear probing?" To search for an item say *key*, first find the hash value using hash function. If the location identified by hash value is empty, the item is not found in the table and search is unsuccessful. If the location is not empty, it is required to compare the key with item in that location. If they are equal, search is successful. If they are not equal, check the item with the items in the

following locations until the item found (and search is successful) or an empty location is encountered. If empty location is encountered, the search is unsuccessful.

Example 3: Let the size of hash table is 15. Insert the following words LIKE, A, TREE, YOU, FIRST, FIND, A, PLACE, TO, GROW, AND, THEN, BRANCH, OUT into a hash table using appropriate hash function and show how each item is inserted into the table, how an item can be searched and deleted using linear probing

Note: In closed hashing, the size of the hash table must be at least equal to the number of unique words in a given text. For example, in the string "LIKE A TREE YOU MUST FIND A PLACE TO GROW AND THEN BRANCH OUT", even though the number of words are 14, the number of unique words is 13 (The word A appears twice). So, the size of the hash table must be at least 13 and it can be larger than 13. Let us assume the table size is 15.

Insert an item: The words to be inserted into the hash table are: LIKE, A, TREE, YOU, FIRST, FIND, A, PLACE, TO, GROW, AND, THEN, BRANCH, OUT. As usual find the hash address by taking the sum of positions of alphabets in the word and taking the remainder by dividing it by 15. The various hash values computed are shown below:

Words	Sum of positions	hash value sum %15
LIKE	12+9+11+5 =37	7
A	1 =1	1
TREE	20+18+5+5 =48	3
YOU	25+15+21 =61	1
FIRST	6+9+18+19+20 =72	12
FIND	6+9+14+4 =33	3
A	1 =1	1
PLACE	16+12+1+3+5 = 37	7
TO	20+15 =35	5
GROW	7+18+15+23 =63	3
AND	1+14+4 = 19	4
THEN	20+8+5+14 =47	2
BRANCH	2+18+1+14+3+8 =46	1
OUT	15+21+20 =56	11

Item = LIKE. It is inserted into position 7
Item = A. It is inserted into position 1
Item = TREE It is inserted into position 3
Item = YOU Its hash value is 1 and that location is full. So, it is inserted in the \ next available location 2 as shown in table 7.6.b.
Item = FIRST It is inserted into position 12
Item = FIND Its hash value is 3. It is already present and has to be inserted into next subsequent empty location.

Thus, keep inserting all the items into the table. The complete hash table using linear probing is shown below:

0		0		0	
1	A	1	A	1	A
2		2	YOU	2	YOU
3	TREE	3	TREE	3	TREE
4		4	FIND	4	FIND
5		5		5	TO
6		6		6	GROW
7	LIKE	7	LIKE	7	LIKE
8		8	PLACE	8	PLACE
9		9		9	AND
10		10		10	THEN
11		11		11	BRANCH
12		12	FIRST	12	FIRST
13		13		13	OUT
14		14		14	
(a)		(b)		(c)	

Table 7.6 To insert various items into hash table

Search for an item Consider the word "KID" to be searched. The hash value h(KID) = (11 + 9 + 4) % 15 = 9. So, compare the word "KID" with the items at position 9 to 13 and is not available in those locations and location 14 is empty and so, the search stops and we display unsuccessful search. If the word to be searched is "BRANCH" whose hash value is 1, it is required to search for that word from position 1 onwards, till the word found or till an empty location is found. Since, it is available in location 11, repo. that the search is successful.

Note: During searching using this technique, we assume that the array is circular, so that if we search past the end of the hash table, we start again at item 0.

Delete an item: Deleting an element from the hash table using this technique is quite complex. Suppose, it is required to delete the word "TO". Its hash value is 5. So, the location 5 is searched for the word "TO" and since there is a match, we can delete that word from that location. But, we can not delete the word "BRANCH". Because, the hash value of the word "BRANCH" is 1. The algorithm searches for the word "BRANCH" from position 1 onwards. Since the word "TO" at location is already deleted, the algorithm encounters an empty location and report the unsuccessful search result. This problem can be avoided by a technique called "lazy deletion", that is, mark the previously occupied locations by a special symbol to distinguish them from the locations that have not been occupied. So, the complexity of the algorithm increases.

Disadvantages: The disadvantage of linear probing is that its performance deteriorates because of the phenomenon called *clustering*. Clustering is a phenomenon where the data tend to cluster (gather) at certain points in the table, with other part of the table not being used. This phenomenon leads to lengthy sequential searches through the table when attempting to retrieve the data. For example, even though hash value of the word "BRANCH" is 1, since earlier items are already clustered together, there was no empty location nearby and so it was inserted at location 11 and searching time will naturally increase. Thus, once the data are filled in nearby locations, if another item has to be inserted which has the same hash value, a lengthy search has to be made for empty location and then the item has to be inserted. Thus inserting subsequent elements leads to longer and longer clusters. For example, consider the figure 7.4.2 which shows an item being inserted into a hash table using linear probing to resolve collision. Assume four elements are inserted already into a table. Now, suppose a new item to be added has the hash value of 4. Since the locations 4, 5, 6 and 7 are already full, the collision resolution technique eventually places the item in location 8.

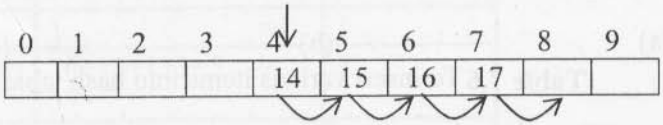


Figure 7.4.2 Clustering effect

Primary Clustering Whenever an insertion is made between two clusters that are separated by an empty location, the two clusters become one, thereby potentially increasing the cluster length. This phenomenon is called primary clustering. For example, there are two clusters: the first cluster has three elements 11, 12 and 13 and the second

cluster has four elements 15, 16, 17 and 18. If an item say 14 is inserted at position 4, the insertion results in a single clustering consisting of elements from 11 to 18.

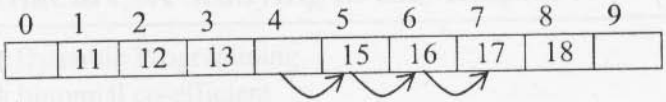


Figure 7.4.3 Primary clustering

7.4.2.2 Double hashing

Using this approach, the primary clustering can be avoided. This technique requires two distinct hash functions h(key) and s(key) both functions accept the same key as the parameter. Whenever the item has to be inserted or retrieved, the primary hashing function

$$i = h(key)$$

is used to determine the position at which the item has to be inserted. The table size *m* is used in primary hashing function. If that position is occupied by some other element, the secondary hash function s(key) is called to get second hash address which can be computed as shown below:

$$i = (i + s(key)) \% m-1$$

where *m* is the size of the hash table. Once this new hash address(or index) is obtained, determine if the location identified by *i* is empty. If empty, insert the item at that position; otherwise, check the subsequent locations obtained by computing the value of *i* repeatedly using the relation:

$$i = (i + s(key)) \% m-1$$

Note that secondary hashing or re-hashing depends not only on the index to be rehashed, but also on the original key. It is required to choose functions *h* and *s* that distribute the hashes and rehashes uniformly over the interval 0 to m-1 and also minimize clustering. But, it is difficult to find such functions. Note that for optimal solution, it is always necessary to choose *m* as prime number.

Note: Some practical results show that with good hashing functions-both primary and secondary- double hashing is more superior when compared with linear probing. But, the performance of this algorithm also deteriorates when the hash table gets close to being full. In such situation, it is required to perform *rehashing* which is a process of reallocating the keys by choosing a larger table.

Applications: Hashing technique is widely used in generating a symbol table during compilation process. With some modifications to hashing function (called extensible hashing), it is also proved that this technique can be used to store very large dictionaries on disks.

Exercises:

1. Is it possible to exchange values of two integer variables *a* and *b* without using extra storage?
2. If equal items are present in an array, will the comparison counting algorithm works?
3. Trace the algorithm for Comparison Counting technique for the items *b, c, d, a, c, b, a, b* and sort in alphabetical order
4. Trace the algorithm for Distribution Counting technique for the items *b, c, d, a, c, b, a, b* and sort in alphabetical order
5. Write the Horspools's algorithm to count the number of occurrences of a pattern string in the given string. Find the shift table for the pattern 'BARBER'
6. State Horspool's algorithm for pattern matching (Pseudocode not necessary). Apply it to search for the pattern BAOBAB in the text BESS-KNEW-ABOUT-BAOBABS
7. Explain the concept of hashing as a method of implementing dictionaries. What are the two main methods of resolving collisions? Briefly explain them. **(Jan/Feb 2005)**

8.2 Dynamic programming

Note: The word programming in dynamic programming has no particular connection to computer programming. Here, programming is the process of obtaining the plan to get the optimal solution.

Now, let us see "What are the differences and similarities between divide and conquer and dynamic programming?

Divide and Conquer	Dynamic programming
♦ This is applicable when the subproblems are independent	♦ This is applicable when the subproblems are not independent
♦ The subproblems are solved separately and combined to get the solution for the original problem	♦ The original problem is solved by using the results of previous subproblems
♦ Every instance of the subproblem is recomputed and is not stored	♦ Only one instance of the subproblem is computed and stored
♦ If same instance of a subproblem appears, the results are recomputed	♦ If the same instance of a subproblem appears, the solution is retrieved from the table and never recomputed
♦ Not efficient because of re-computations	♦ More efficient because re-computations are not done
♦ Uses top-down approach	♦ Uses bottom up approach
♦ **Similarity:** Both techniques solve the problem by dividing the problem into subproblems. Using the solutions of subproblems, the solution for the larger instance of the problem can be obtained	

8.3 Computing Binomial coefficient

We know that a binomial formula is given by

$$(a+b)^n = nc_0a^nb^0 + nc_1a^{n-1}b^1 + nc_2a^{n-2}b^2 + \ldots\ldots nc_ia^{n-i}b^i + \ldots\ldots nc_na^{n-n}b^n$$

Here, nc_0, nc_1, nc_2, ……nc_i,…… nc_n are called binomial coefficients. The two useful properties of binomial coefficients are

$$^nC_k = {}^{n-1}C_{k-1} + {}^{n-1}C_k \quad \text{for } n > k > 0$$

$$^nC_k = 1 \quad \text{if } k = 0 \text{ or } n = k$$

The above recurrence relation can also be written as

$$C(n, k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } n = k \\ C(n-1, k-1) + C(n-1, k) & \text{for } n > k > 0 \end{cases} \quad \ldots\ldots\ldots\ldots\ldots(1)$$

Note: It is clear from the above recurrence relation that c(n, k) can be computed by computing two overlapping problems c(n-1, k-1) and C(n-1, k) and hence this problem can be solved using dynamic programming. Let us solve $^6C_3$ using dynamic programming.

Example 1: Compute $^6C_3$ using dynamic programming

Step 1: We know that whenever k = 0, $^nC_0 = 1$ for any value of *n*. Here, n = 6.

$$\text{So, } {}^0C_0 = {}^1C_0 = {}^2C_0 = {}^3C_0 = {}^4C_0 = {}^5C_0 = {}^6C_0 = 1$$

Step 2: We know that whenever n = k, $^nC_k = 1$ for any value of *n*. Here, k = 3.

$$\text{So, } {}^0C_0 = {}^1C_1 = {}^2C_2 = {}^3C_3 = 1$$

Note: Since we want to compute $^6C_3$, let us store the values computed in step 1 and step2 in the form of the table as shown below:

i \ j → k	0	1	2	3
0	1			
1	1	1		
2	1		1	
3	1			1
4	1			
5	1			
n 6	1			

Note: There is no need to compute the values shown in *black* boxes, in equation (1) the value of k at any point of time should not exceed *n*.

Step 3: Now, using equation (1), it is required to find the unfilled locations shown in white boxes. The unfilled results using equation (1) can be obtained as shown below:

C(2, 1) = C(1, 0) + C(1, 1) = 1 + 1 = 2
C(3, 2) = C(2, 1) + C(2, 2) = 2 + 1 = 3
C(4, 2) = C(3, 1) + C(3, 2) = 3 + 3 = 6
C(5, 1) = C(4, 0) + C(4, 1) = 1 + 4 = 5
C(5, 3) = C(4, 2) + C(4, 3) = 6 + 4 = 10
C(6, 2) = C(5, 1) + C(5, 2) = 5 + 10 = 15

C(3, 1) = C(2, 0) + C(2, 1) = 1 + 2 = 3
C(4, 1) = C(3, 0) + C(3, 1) = 1 + 3 = 4
C(4, 3) = C(3, 2) + C(3, 3) = 3 + 1 = 4
C(5, 2) = C(4, 1) + C(4, 2) = 4 + 6 = 10
C(6, 1) = C(5, 0) + C(5, 1) = 1 + 5 = 6
C(6, 3) = C(5, 2) + C(5, 3) = 10 + 10 = 20

Note: By storing all these values in the above table, following table is obtained. Note that no values are calculated more than once. This is the beauty of dynamic programming.

i \ j →	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10
n 6	1	6	15	20

→ **C[6,3] = $^6C_3$ = 20 (result)**

Note: In general, the problem can be easily solved if we record the values of binomial coefficients in table consisting of n+1 rows ranging from 0 to n and k+1 columns ranging from 0 to k.

Design: Observe that main idea here is to fill the table with n + 1 rows and k + 1 columns using the relation

C[i,j] = 1 if j = 0 or i = j
C[i,j] = C(i-1, j-1) + C(i-1, j) if n > k > 0

The above relation can be written in the form of pseudocode as shown below"

```
if (j = 0 or i = j)
        C[i,j] = 1
else                                    .....(1)
        C[i,j] = C[i-1, j-1] + C[i-1, j]
end if
```

The range for index *i* and index *j* can be obtained by considering the various values of *i* and *j* in the above table. Note the following values of *i* and *j*:

When i = 0, j = 0 to 0
When i = 1, j = 0 to 1
When i = 2, j = 0 to 2
When i = 3, j = 0 to 3

Note: Final value of j is same as
So, in general **j = 0 to i** .....(2)

When i = 4, j = 0 to 3
When i = 5, j = 0 to 3
When i = 6, j = 0 to 3

Note: Final value of j is k
where k = 3
So, in general **j = 0 to k** .....(3)

i = 0 to 6. In general, **i = 0 to n** .....(4)

Using equations (2) and (3) we can say that j = 0 to (i or k) whichever is minimum.
In general, we can say that **j = 0 to min(i, k)** .....(5)

Using statements labeled (1), (4) and (5), the algorithm can be written as shown below:

```
ALGORITHM Binomial_Coefficient(n, k)
//Purpose:    To compute binomial by dynamic programming
//Input:      non negative integers such that n ≥ k ≥ 0
//Output:     Value of ⁿCₖ also designated as C(n, k)
    for i ← 0 to n do
        for j ← 0 to min(i, k) do
            if (j = 0 or i = j)
                c[i, j] ← 1        // Fill the diagonal and first column with 1
            else
                c[i, j] ← c[i-1, j-1] + c[i-1, j]
            endif
        end for
    end for
    return C[n,k]
// end of algorithm
```

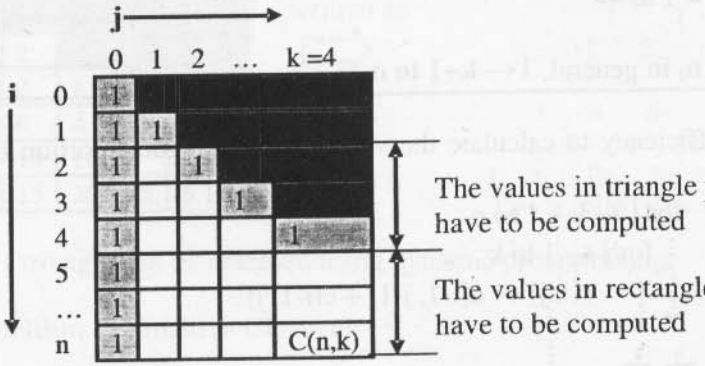
Analysis The time efficiency of binomial coefficient algorithm can be obtained as shown below:

Step 1: The parameter to be considered are n and k which represent the size of input

Step 2: The basic operation in the above algorithm is the statement

$$c[i][j] \leftarrow c[i-1][j-1] + c[i-1][j]$$

Step 3: The total number of times the basic operation is executed to compute any value of ${}^nC_k$ for $n > k > 0$ can be represented in the form of a table as shown below:



Observe from the table that all the values to be computed in first k rows and k columns form a triangle and the values to be computed from k+1 rows onwards till n^{th} row along

with k+1 columns form a rectangle. So, the total number of times the basic operations are executed is given by the following relation:

$$f(n, k) = \underset{\mathbf{f1(n, k)}}{\text{Time efficiency to compute the values for triangle portion}} + \underset{\mathbf{f2(n, k)}}{\text{Time efficiency to compute the values for rectangle portion}} \quad \text{.....(1)}$$

Let us see "How to compute f1(n, k)?" Note from the table that:

when i = 2, j = 1 to 1
when i = 3, j = 1 to 2
when i = 4, j = 1 to 3

Final value of *j* is 1 less than *i*.
So, j ← 1 to i - 1

Here i = 2 to 4. In general, **i ← 2 to k**

Now, time efficiency to calculate the values for triangular portion can be obtained as shown below:

```
for i ← 2 to k
    for j ← 1 to i – 1
        c[i, j] ← c[i-1, j-1] + c[i-1, j]
```

$$f1(n, k) = \sum_{i=2}^{k} \sum_{j=1}^{i-1} 1 \quad \text{.....(2)}$$

Let us see "How to compute f2(n, k)?" Note from the table that:

when i = 5, j = 1 to 4
when i = 6, j = 1 to 4
when i = 7, j = 1 to 4

Final value of *j* is 1 less than *i*.
So, j ← 1 to k

Here i = 5 to n. In general, **i ← k+1 to n**

Now, time efficiency to calculate the values for rectangular portion can be obtained as shown below:

```
for i ← k + 1 to n
    for j ← 1 to k
        c[i, j] ← c[i-1, j-1] + c[i-1, j]
```

$$f1(n, k) = \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1 \quad \text{.....(3)}$$

Substituting equation (2) and (3) in equation (1) we have,

$$T(n, k) = \sum_{i=2}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1$$

$$= \sum_{i=2}^{k} i-1-1+1 + \sum_{i=k+1}^{n} k-1+1$$

$$= \sum_{i=2}^{k} i-1 + \sum_{i=k+1}^{n} k$$

$$= 1+2+3+.....(k-1)+k\sum_{i=k+1}^{n} 1$$

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

$$= \frac{k^2}{2} - \frac{k}{2} + nk - k^2 \approx nk \text{ for very large value of } n.$$

So, time complexity to obtain binomial coefficient using dynamic programming is given by $\theta(nk)$.

Example 2: Compute ${}^6C_6$ using dynamic programming and show contents of the table

Solution: The user is required to compute ${}^6C_6$ using the relations shown earlier and verify that following table is obtained.

j → k

i ↓ n	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

can be written as ⇨

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

This is called Pascal's triangle

Note: Thus, Pascal's triangle can be obtained using dynamic programming.

8.4 Warshal's Algorithm (Transitive Closure)

In this section, let us see "What is transitive closure?"

8.8 Dynamic programming

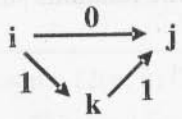
Definition: Let G = (V, E) be a simple graph where V is set of vertices and E is set of edges. Let N be the number of vertices in graph G. The matrix P (of size N x N) whose elements are given by

$$P[i, j] = \begin{cases} 1 & \text{if there is a path from vertex } i \text{ to vertex } j \\ 0 & \text{Otherwise} \end{cases}$$

is called *path matrix (transitive closure)*. It is clear from the above definition that the element in i^{th} row and j^{th} column is 1 provided there exists a path from i^{th} vertex to j^{th} vertex and 0 if there is no path. It is observed that the matrix P is square matrix and the path matrix has 0's or 1's and so the matrix is called *bit matrix* or *Boolean matrix*.

Now, let us see "How to find the transitive closure?" Given a graph G = (V, E), adjacency matrix can be obtained. Using this adjacency matrix, transitive closure can be obtained. This process of transforming adjacency matrix to transitive closure is done by Warshal's algorithm.

Design: If there is a path between i to j then the entry in P[i, j] = 1. Consider the situation where P[i, j] = 0. This indicates that direct path does not exist from vertex *i* to vertex *j*. In such situation, if there exists a path from (i, k) and (k, j) then there exists a path from (i, k). This can be pictorially represented as shown below:



Even though the direct path does not exist between i and j, there exists a path from i to j through k

This equivalent pseudocode can be written as shown below:

```
if (p[i,j] = 0)
        if (p[i,k] = 1 and p[k,j]=1)
                p[i,j] = 1
        end if
end if
```

Note: The above statement has to be executed for each pair(i, j) when k = 0, 1, 2, ...n. So, *k* should be the outer loop, *i* should be within k^{th} loop and *j* should be the innermost loop.

So, the complete algorithm for Warshal's algorithm can be written as shown below:

ALGORITHM Warshall(n, A, P)

//Purpose: To compute Transitive closure(path matrix)

//Input: Adjacency matrix A of size n x n

//Output: Transitive closure(path matrix) of size n x n

Step 1: [Make a copy of the adjacency matrix]

```
for i ← 0 to n-1 do
        for j ← 0 to n-1 do
                P[i, j] = A[i, j]
        end for
end for
```

Step 2: [Find the Transitive closure (path matrix)]

```
for k ← 0 to n-1 do
        for i ← 0 to n-1 do
                for j ← 0 to n-1 do
                        if (P[i, j] = 0 and  (if (P[i, k] = 1 and P[k, j] = 1)) then
                                P[i, j] = 1
                        end if
                end for
        end for
end for
```

Step 3: [Finished]

```
return
```

Analysis The time efficiency of Warshal's algorithm can be obtained as shown below:

Step 1: The parameter to be considered is n which represent the size of input

Step 2: The basic operation is the comparison statement in the innermost loop.

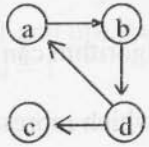
Step 3: The total number of times the basic operation is executed can be obtained as shown below:

for k ← 0 to n-1 **do**
  for i ← 0 to n-1 **do**
    for j ← 0 to n-1 **do**
      if (P[i, j] = 0 and (**if** (P[i, k] = 1 and P[k, j] = 1)) **then**

$$
\begin{aligned}
f(n) &= \sum_{k=0}^{n-1}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1 \\
&= \sum_{k=0}^{n-1}\sum_{i=0}^{n-1} n-1-0+1 \\
&= \sum_{k=0}^{n-1}\sum_{i=0}^{n-1} n \\
&= \sum_{k=0}^{n-1} n \sum_{i=0}^{n-1} 1 \\
&= \sum_{k=0}^{n-1} n(n-1-0+1) \\
&= \sum_{k=0}^{n-1} n(n) = \sum_{k=0}^{n-1} n^2 \\
&= n^2 \sum_{k=0}^{n-1} 1 \\
&= n^2(n-1-0+1) = n^2 * n = n^3
\end{aligned}
$$

So, the time complexity of Warshall's algorithm is given by $\theta(n^3)$

Example 1: Apply Warshall's algorithm to compute transitive closure(path matrix) for the graph shown below:



Solution: The adjacency matrix for the given graph is shown below:

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Step 1: The path through vertex *a*. Since we are finding the path through *a*, ath row and ath column need to be considered. So, let us not worry about all other pairs (shown in white color). Consider only the pairs with entries 1. **Note**: The pair (i, j) is made 1 if and only if (i, k) = 1 and (k, j) = 1. Here, k is nothing but the vertex *a*.

In the above table, since (d, a) = 1 and (a, b) = 1. So, make an entry **(d, b) = 1.** The resulting table is shown below:

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

Step 2: The path through vertex *b*. Since we are finding the path through *b*, bth row and bth column need to be considered. In bth column consider only those vertices with value 1.

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	1	1	0

Since (a, b) = 1 and (b, d) = 1 and we make **(a, d) = 1**
Since (d, b) = 1 and (b, d) = 1 and we make **(d, d) = 1**
The resulting table is shown below:

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Step 3: The path through vertex *c*. Since we are finding the path through *c*, cth row and cth column need to be considered as shown below:

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Note that (d, c) = 1, from vertex *c* there is not path to any other node. So, the values have not been altered and the resulting table is same as the previous one. The table is shown below:

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Step 4: The path through vertex d. Since, we are finding the path through *d*, d^{th} row and d^{th} column need to be considered as shown below:

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Since (a, d) = 1 and (d, a) = 1. So, **(a, a) = 1**
Since (a, d) = 1 and (d, b) = 1. So, **(a, b) = 1**
Since (a, d) = 1 and (d, c) = 1. So, **(a, c) = 1**
Since (a, d) = 1 and (d, d) = 1. So, **(a, d) = 1**

Since (b, d) = 1 and (d, a) = 1. So, **(b, a) = 1**
Since (b, d) = 1 and (d, b) = 1. So, **(b, b) = 1**
Since (b, d) = 1 and (d, c) = 1. So, **(b, c) = 1**
Since (b, d) = 1 and (d, d) = 1. So, **(b, d) = 1**

The resultant table is shown below:

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

The above final matrix or table gives the transition closure (path matrix) obtained from the adjacency matrix.

8.5 Floyd's Algorithm (for All Pairs shortest paths problem)

Now, let us see "What is all pairs shortest path problem?" In all pairs shortest path problem, we find the shortest distance from all nodes to all other nodes. The solution for this problem can be obtained easily using Floyd's algorithm. The Floyd's algorithm is the modified version of Warshal's algorithm. Formally, the problem can be stated as follows:

Let G = (V, E) be a directed graph where V is set of vertices with *n* number of vertices and E is set of edges. Let *cost* be the cost adjacency matrix for the graph G such that

cost(i, i) = 0, for $0 \le i < n$

cost(i, j) is the cost associated with the edge (i, j) if there is an edge from *i* to *j*.

cost(i, j) = ∞ if there is no edge from *i* to *j*.

The *all-pairs-path-problem* is to determine a matrix D such that D{i, j) contains the shortest distance from *i* to *j*.

Note: In Warshal's algorithm, input is adjacency matrix. But, for the Floyd's algorithm, the input is cost adjacency matrix (also called weighted matrix). The Floyd's algorithm is the modified version of Warshal's algorithm. So, consider the following statement of Warshal's algorithm:

```
if (P[i, j] = 0 and (if (P[i, k] = 1 and P[k, j] = 1)) then
        P[i, j] = 1
end if
```

The above statement is changed to the following statement:

D[i, j] = min (D[i, j], D[i, k] + D[k, j])

Rest of the instructions remain same. So, the Floyd's algorithm can be written as shown below:

ALGORITHM FLOYD(n, cost , D)

//Purpose: To implement Floyd's algorithm for all-pairs shortest paths problem

//Input: Cost adjacency matrix cost of size n x n

//Output: Shortest distance matrix of size n x n

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        D[i, j] = cost [i, j]
    end for
end for
```

```
for k ← 0 to n-1 do
    for i ← 0 to n-1 do
        for j ← 0 to n-1 do
            D[i, j] = min(D[i, j], D[i, k] + D[k, j])
        end for
    end for
end for
return
// End of algorithm
```

Analysis The time efficiency of Floyd's algorithm can be obtained as shown below:

Step 1: The parameter to be considered is n which represent the size of input

Step 2: The basic operation is the comparison statement in the innermost loop.

Step 3: The total number of times the basic operation is executed can be obtained as shown below:

```
for k ← 0 to n-1 do
    for i ← 0 to n-1 do
        for j ← 0 to n-1 do
            D[i, j] = min(D[i, j], D[i, k] + D[k, j])
```

$$f(n) = \sum_{k=0}^{n-1}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1$$

$$= \sum_{k=0}^{n-1}\sum_{i=0}^{n-1} n-1-0+1$$

$$= \sum_{k=0}^{n-1}\sum_{i=0}^{n-1} n$$

$$= \sum_{k=0}^{n-1} n \sum_{i=0}^{n-1} 1$$

$$= \sum_{k=0}^{n-1} n(n-1-0+1)$$

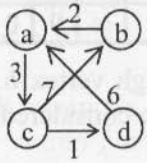
$$= \sum_{k=0}^{n-1} n(n) = \sum_{k=0}^{n-1} n^2$$

$$= n^2 \sum_{k=0}^{n-1} 1$$

$$= n^2(n-1-0+1) = n^2 * n = n^3$$

So, the time complexity of Warshall's algorithm is given by $\theta(n^3)$

Example 1: Solve the all-pairs shortest-path problem for the digraph shown below:



Solution: The cost adjacency matrix for the given graph is shown below:

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Note: The procedure that we follow in Warshall's algorithm is used here. In Warshal's algorithm we used to consider only the column values with value 1. But, in this technique we use non-zero and non-infinity values.

Step 1: Consider the shortest distance through vertex *a*. Since we are finding the shortest distance through *a*, ath row and ath column need to be considered with non-zero and non-infinity numbers shown in first column.

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

(b, a) = 2 and (a, c) = 3. So, (b, c) = min{(b, c) , (b, a) + (a, c)}
= min{∞, 2+ 3} = 5

So, **(b, c) = 5**

(d, a) = 6 and (a, c) = 3. So, (d, c) = min{(d, c) and (d, a) + (a, c)}
= min {∞, 6 + 3} = 9

So, **(d, c) = 9**

The resultant matrix is shown below:

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

Step 2: Consider the shortest distance through vertex *b*. Since we are finding the path through *b*, bth row and bth column need to be considered with non-zero and non-infinity numbers shown in second column.

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

(c, b) = 7 and (b, a) = 2 and so (c, a) = min {(c, a) and (c, b) + (b, a)}
= min {∞, 7 + 2} = 9

So, **(c, a) = 9**

(c, b) = 7 and (b, c) = 5 and so (c, c) = min {(c, c) and (c, b) + (b, c)}
= min {0, 7 + 5} = 0

So, **(c, c) = 0**

The resultant matrix is shown below:

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

Step 3: Consider the shortest distance through vertex *c*. Since we are finding the path through *c*, cth row and cth column need to be with non-zero and non-infinity numbers shown in third column.

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

(a, c) = 3 and (c, a) = 9. So, (a, a) = min{(a, a) and (a, c) + (c, a)}
= min{ 0, 3 + 9} = 0

So, **(a, a) = 0**

(a, c) = 3 and (c, b) = 7. So, (a, b) = min {(a, b) and (a, c) + (c, b)}
= min {∞, 3 + 7} = 10

So, **(a, b) = 10**

(a, c) = 3 and (c, d) = 1. So, (a, d) = min {(a, d) and (a, c) + (c, d)}
= min { ∞ , 3 + 1} = 4

So, **(a, d) = 4**

(b, c) = 5 and (c, a) = 9. So, (b, a) = min{(b, a) and (b, c) + (c, a)}
= min{ 2, 5 + 9} = 2

So, **(b, a) = 2**

(b, c) = 5 and (c, b) = 7. So, (b, b) = min {(b, b) and (b, c) + (c, b)}
= min {0, 5 + 7} = 0

So, **(b, b) = 0**

(b, c) = 5 and (c, d) = 1. So, (b, d) = min {(b, d) and (b, c) + (c, d)}
= min { ∞ , 5 + 1} = 6

So, **(b, d) = 6**

(d, c) = 9 and (c, a) = 9. So, (d, a) = min{(d, a) and (d, c) + (c, a)}
= min{ 6, 9 + 9} = 6

So, **(d, a) = 6**

(d, c) = 9 and (c, b) = 7. So, (d, b) = min {(d, b) and (d, c) + (c, b)}
= min {∞, 9 + 7} = 16

So, **(d, b) = 16**

(d, c) = 9 and (c, d) = 1. So, (d, d) = min {(d, d) and (d, c) + (c, d)}
= min { 0 , 9 + 1} = 10

So, **(d, d) = 0**

So, the resultant table is:

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

Step 4: Consider the shortest distance through vertex *d*. Since we are finding the path through *d*, d^{th} row and d^{th} column need to be considered. So, let us not worry about all other pairs (shown in white color in the figure below) and consider only the pairs with non-zero and non-infinity numbers shown in 4th column.

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

(a, d) = 4 and (d, a) = 6. So, (a, a) = min{(a, a), 4 + 6} = 0
(a, d) = 4 and (d, b) = 16. So, (a, b) = min{(a, b), 4 + 16} = 10
(a, d) = 4 and (d, c) = 9. So, (a, c) = min{(a, c), 4 + 9} = 3

(b, d) = 6 and (d, a) = 6. So, (b, a) = min{(b, a), 6 + 6} = 2
(b, d) = 6 and (d, b) = 16. So, (b, b) = min{(b, b), 6+16} = 0
(b, d) = 6 and (d, c) = 9. So, (b, c) = min{(b, c), 6 + 9} = 5

(c, d) = 1 and (d, a) = 6. So, (c, a) = min{(c, a), 1+6} = 7
(c, d) = 1 and (d, b) = 16 So, (c, b) = min{(c, b), 1+16} = 7
(c, d) = 1 and (d, c) = 9 So, (c, c) = min{(c, c), 1 + 9} = 0

The resultant matrix is:

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

8.6 The 0/1 Knapsack Problem

This problem has been discussed in section 3.6.2, page 3.23 and has been solved using exhaustive search method. Now, let us see the method of solving this problem using dynamic programming. For the sake of convenience, let us "Define knapsack problem"

Definition: Given a knapsack (bag or container) with following:

- M – capacity of the knapsack
- n - number of objects
- *w* – an array consisting of weights $w_1, w_2, w_3, \ldots w_n$
- p – an array consisting of profits $p_1, p_2, p_3, \ldots p_n$.
- x – an array consisting of either 0 or 1. A 0 in x_i represent ith object has not been selected and a 1 in x_i i^{th} object has been selected.

The main objective is to place the objects into the knapsack so that maximum profit is obtained and the weights of objects chosen should not exceed the capacity of knapsack. This problem can be stated as:

$$\text{Maximize } \sum_{i=1}^{n} P_i X_i$$

$$\text{Subject to the constraint } \sum_{i=1}^{n} W_i X_i \le M$$

Design methodology Let i represent the ith object in question to be selected or not to get the maximum profit with weight w_i and profit p_i with remaining capacity j. V[i, j] is the profit obtained by considering all i objects with capacity j. The various cases to be considered are shown below:

Case 1: If there are no objects (i.e., i = 0) or capacity of knapsack itself is 0 (i.e., j = 0) then profit will be 0. The total profit obtained is:

$$V[i, j] = 0 \qquad \text{if } (i = 0 \text{ or } j = 0) \quad \ldots\ldots(1)$$

Case 2: Suppose we have a situation that weight of the ith object w_i is greater than remaining capacity j, in such case ith object can not be selected. So, the profit obtained is:

$$V[i, j] = V[i-1, j] \qquad \text{if } w_i > j \quad \ldots\ldots(2)$$

Note: i^{th} object has been rejected, because it can not fit into the knapsack

Case 3: Suppose we have a situation where w_i is less than remaining capacity j. In such case we have two alternatives:

- We reject ith object. If so, profit obtained is: $V[i, j] = V[i-1, j]$.
- We select ith object. If so, profit obtained is: $V[i, j] = V[i-1, j-w_i] + P_i$

Out of these situations whichever yields maximum profit, that has to be considered. So, we have to select maximum of these two. So, profit obtained if $w_i <= j$ is given by

$$V[i, j] = \max \{ V[i-1, j], V[i-1, j-w_i] + p_i \} \text{ if } w_i <= j \quad \ldots\ldots(3)$$

Thus, the final recurrence relation to get the solution to knapsack problem can be written as shown below:

$$V[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \le j \end{cases}$$

Note: **V[n, M]** gives the optimal profit obtained. Now, let us solve one problem and see how to get the optimal solution using knapsack problem.

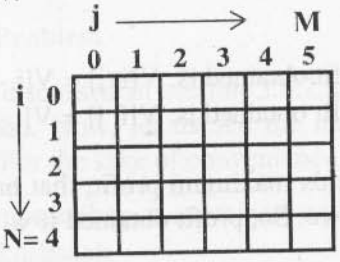
Example 1: Apply bottom-up dynamic programming technique to the following instance of the knapsack problem with capacity M = 5

Item	Weight	Value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

Solution: The following data is provided:

- Number of objects N = 4
- Capacity of the knapsack M = 5
- $w_1 = 2, w_2 = 1, w_3 = 3, w_4 = 2$
- $p_1 = 12, p_2 = 10, p_3 = 20, p_4 = 15$

Step 1: Since N = 4, M = 5 we have a solution matrix consisting of N+1 rows and M+1 columns as shown below:



Step 2: In this bottom up approach, we start from the initial solution with i = 0 or j = 0. Using equation (1) we have

$$V[i, j] = 0 \qquad \text{if } (i = 0 \text{ or } j = 0)$$

So, when i = 0 (i.e., when no objects are there), irrespective of the remaining capacity j = 0, 1, 2, 3, 4, 5 the profit V[i, j] = 0. Similarly, when j = 0 (i.e., when there is no knapsack) irrespective of number of objects i = 0, 1, 2, 3, 4 the profit V[i, j] = 0. These entries are made in above table. The resulting table is shown below:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
N= 4	0					

Profit table

Step 3: When i = 1, W_i = 2 and P_i = 12, we have to compute the results for various values of j = 1, 2, 3, 4, 5 (See row 1 of above table. Those values should be computed). The results of row 1 can be computed as shown below:

$$V[i, j] = V[i-1, j] \quad \text{if } W_i > j \quad \ldots\ldots\ldots\ldots(1)$$

$$W_i \quad j \qquad V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i) \quad \text{if } W_i \le j \ldots\ldots\ldots\ldots(2)$$

2	1	V[1, 1] = V[0, 1] = 0
2	2	V[1, 2] = Max(V[0, 2], V[0, 0] + 12) = Max (0 , 0 + 12) = 12
2	3	V[1, 3] = Max(V[0, 3], V[0, 1] + 12) = Max(0, 0 + 12) = 12
2	4	V[1, 4] = Max(V[0, 4], V[0, 2] + 12) = Max(0, 0 + 12) = 12
2	5	V[1, 5] = Max(V[0, 5], V[0, 3] + 12) = Max)9, 0 + 12) = 12

Note: Only when Wi = 2, = 1, relation (1) is used. For the rest we use the relation (2)

Substituting these computed values in previous profit table, following table is obtained:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
N= 4	0					

Profit table

Step 4: When i = 2, W_i = 1 and P_i = 10, compute the results for various values of j as shown below:

$$V[i, j] = V[i-1, j] \quad \text{if } W_i > j \quad \ldots\ldots\ldots\ldots(1)$$

$$W_i \quad j \quad V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i) \quad \text{if } W_i \le j \ldots\ldots\ldots\ldots(2)$$

1	1	V[2, 1] = Max(V[1, 1], V[1, 0] + 10) = Max (0, 0 + 10) = 10
1	2	V[2, 2] = Max(V[1, 2], V[1, 1] + 10) = Max(12, 0 + 10) = 12
1	3	V[2, 3] = Max(V[1, 3], V[1, 2] + 10) = Max(12, 12+10) = 22
1	4	V[2, 4] = Max(V[1, 4], V[1, 3] + 10) = Max(12, 12 + 10) = 22
1	5	V[2, 5] = Max(V[1, 5], V[1, 4] + 10) = Max(12, 12 + 10) = 22

By storing all these values in the previous profit table, the following table is obtained.

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
N= 4	0					

Profit table

Step 5: When i = 3, W_i = 3 and P_i = 20, the values of row 3 (of above table) are computed as shown below:

$$V[i, j] = V[i-1, j] \quad \text{if } W_i > j \quad \ldots\ldots\ldots\ldots(1)$$

$$W_i \quad j \quad V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i) \quad \text{if } W_i \le j \ldots\ldots\ldots\ldots(2)$$

3	1	V[3, 1] = V[2, 1] = 10
3	2	V[3, 2] = V[2, 2] = 12
3	3	V[3, 3] = Max(V[2, 3], V[2, 0] + 20) = Max(22, 0 + 20) = 22
3	4	V[3, 4] = Max(V[2, 4], V[2, 1] + 20) = Max(22, 10+20) = 30
3	5	V[3, 5] = Max(V[2, 5], V[2, 2] + 20) = Max(22, 12 + 20) = 32

By substituting these values in the previous profit table, we have the following table:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
N= 4	0					

Profit table

Step 6: When $i = 4$, $W_i = 2$ and $P_i = 15$, the values of row 4 (of above table) are computed as shown below:

$$V[i, j] = V[i-1, j] \quad \text{if } W_i > j \quad \ldots\ldots\ldots\ldots(1)$$

$$W_i \quad j \quad V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i) \quad \text{if } W_i \le j \ldots\ldots\ldots\ldots(2)$$

2	1	V[4, 1] = V[3, 1] = 10
2	2	V[4, 2] = Max(V[3, 2], V[3, 0] + 15) = Max(12, 15) = 15
2	3	V[4, 3] = Max(V[3, 3], V[3, 1] + 15) = Max(22, 12 + 15) = 27
2	4	V[4, 4] = Max(V[3, 4], V[3, 2] + 15) = Max(30, 12 + 15) = 27
2	5	V[4, 5] = Max(V[3, 5], V[3, 3] + 15) = Max(32, 22+15) = 37

By substituting these values in the previous profit table, we have the table shown below:

j → M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
N= 4	0	10	15	27	27	37

Profit table

Optimal solution

It is given that $n = 4$ and $M = 5$. Therefore, optimal solution $V[n, M] = V[4, 5] = 37$

Now, the complete algorithm to find the optimal solution for the knapsack can be obtained using the recurrence relation

$$V[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \le j \end{cases}$$

ALGORITHM KNAPSACK(n, m, w, p, v)

//Purpose: To find the optimal solution for the knapsack problem using
// dynamic programming

```
//Input:    n – Number of objects to be selected
//          m – Capacity of the knapsack
//          w – weights of all the objects
//          p – profits of all the objects

//Output:   v – the optimal solution for the number of objects selected with specified
//          remaining capacity

    for i ← 0 to n do
        for j ← 0 to m do
            if ( i = 0 or j = 0)
                V[i, j] = 0
            else if (W[i] > j)
                V[i, j] = V[i-1, j]
            else
                V[i, j] = max( V[i-1, j], V[i-1, j-W[i]] + p[i])
            end if
        end for
    end for

    return
// End of the algorithm
```

Analysis The efficiency for the knapsack algorithm can be obtained as shown below:

Step 1: The parameter to be considered is n and M which represent the inputs

Step 2: The basic operation is "computing V[i,j] = max…." which is executed maximum number of times. The total number of times this basic operation is executed can be obtained as shown below:

```
for i ← 0 to n do
    for j ← 0 to m do
        ……
        V[i, j] = max( V[i-1, j], V[i-1, j-W[i]] + p[i])
```

$$f(n, m) = \sum_{i=0}^{n}\sum_{j=0}^{m} 1$$

$$= \sum_{i=0}^{n} m - 0 + 1$$

$$= \sum_{i=0}^{n} (m+1)$$

$$= (m+1)\sum_{i=0}^{n} 1$$

$$= (m+1)\,(n-0+1)$$

$$= (m+1)\,(n+1)$$

$$= mn + m + n + 1 \approx mn \text{ for very large value of } m \text{ and } n$$

So, the time complexity of knapsack algorithm is given by $\theta(mn)$.

Now, the question is "How to find the objects that are selected to get the optimal solution?" We know that if V[i, j] is same as V[i-1, j] it indicates that ith object has not been selected. if V[i, j] is different from V[i-1, j] it indicates that ith object has been selected and we are supposed to select among i-1 objects and fill the knapsack whose remaining capacity is j-W_i. The code corresponding to this is given by:

```
if (V[i, j] ≠ V[i-1, j] then
        x[i] = 1                    // ith object has been selected
        j = j-W[i]                  // Obtain the remaining capacity of the knapsack
end if
i = i – 1                           // Select the object among i-1 objects
```

The complete algorithm to find the objects that are selected to get the optimal solution is shown below:

ALGORITHM print_optimal_solution(n, m, w, v, x)

```
//Purpose:    To obtain the objects selected to get the optimal solution

//Input:      n – Number of objects to be selected
//            m – Capacity of the knapsack
//            w – weights of all the objects
//            v – the optimal solution for the number of objects selected with specified
//            remaining capacity
//Output:
//            x – The information of objects selected and not selected

    //display the optimum solution
    write 'The optimal solution is', v[n, m]
```

```
// Initialization:  Indicates that no object has been selected
for i ← 0 to n-1 do
        x[i] = 0
end for

i = n;          // Number of objects
j = m           // Capacity of the knapsack

// Obtain the optimum solution
while (i ≠ 0 and j ≠ 0)
    if (v[i][j] ≠ v[i-1][j])
            x[i] = 1        // ith object has been selected
            j = j – W[i]    // Obtain the remaining capacity of the knapsack
    end if
    i = i – 1               // Select the object among i-1 objects
end while

// Output the objects selected
for i ← 1 to n do
    if (x[i] = 1)
            write 'Object selected is',i
    end if
end for
// end of the algorithm
```

Example 2: Find the objects that are selected for the problem specified in example 1

Solution: The profit table obtained from example 1 is shown below

j ⟶ M

i ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
N= 4	0	10	15	27	27	37

Profit table

Optimal solution

It is given that n = 4 and M = 5. Therefore, optimal solution **V[n, M] = V[4, 5] = 37**

Given i = 4 and j = 5. $w_1 = 2$, $w_2 = 1$, $w_3 = 3$, $w_4 = 2$.
To start with (x1, x2, x3, x4) = (0, 0, 0, 0) indicates no object is selected.

Step 1: V[4, 5] V[3,5] Since these two values are different, it indicates that object i=4
37 32
is selected. So, **x(4) = 1.** Hence, solution is (x1, x2, x3, x4) = (0, 0, 0, 1)

Since object 4 is selected, remaining capacity = j = 5-w_4 = 5 – 2 = 3
i = i-1 i.e., i = 3

So, consider V(i, j) i.e., V(3, 3)

Step 2: V[3, 3] V[2, 3]. Since both are same it clearly indicates that object 3 has not
22 22
been considered and solution vector remains same i..e., (x1,x2,x3,x4) = (0,0,0,1)
Since i = 3 has not been selected select i = 2 with earlier remaining capacity.

So, consider V[i, j] i.e., V[2, 3]

Step 3: V[2,3] V[1,3]. Since these two values are different, it indicates that object i=2
22 12
is selected. So, **x(2) = 1.** Hence, solution is (x1, x2, x3, x4) = (0, 1, 0, 1)

So, remaining capacity j = 3 – w_2 = 3 – 1 = 2
i = i -1 = 1

So, consider V[i, j] i.e., V[1, 2]

Step 4: V[1, 2] V[0, 2]. Since these two values are different, it indicates that object i=1
12 0
is selected. So, **x(1) = 1.** Hence, solution is (x1, x2, x3, x4) = (1, 1, 0, 1)

So, remaining capacity j = 2 – w_1 = 2 – 2 = 0.
i = i – 1 = 0.

Since i=0 (no objects are there) and j = 0 (knapsack is full) the algorithm terminates. So, the final solution is (x1, x2, x3, x4) = (1, 1, 0, 1) which indicates objects 1, 2 and 4 are considered to get a maximum profit of 37

8.7 knapsack problem using memory functions

To understand what is the need for memory functions, we should know "What are the drawbacks of both top down and bottom up approach?" Let us compare top-down approach and bottom-up approach:

- In top-down approach, the algorithm solves common subproblems more than once and hence is very inefficient. It is disadvantage and can be overcome using dynamic programming which uses bottom-up approach.
- In dynamic programming which uses bottom-up approach, the common subproblems are solved only once. But, the solution to some of the subproblems is often not necessary to get the solution for the given problem. But, this problem is not there in top-down approach.

So, it is necessary to take the advantages of both top-down approach and bottom-up approach. This is done by memory functions. The goal of memory functions is to solve only subproblems that are necessary and doing it only once. Now, let us "Explain how the drawbacks of top-down and bottom-up approach of dynamic programming are overcome using memory functions" The above said drawbacks of top-down and bottom-up approach can be eliminated using memory function as shown below:

- This method uses top-down approach to solve the problem and maintains a table to store the computed results as is done by bottom-up approach
- Initially, table entries are filled with "-" indicating they have not been computed
- When a new value has to be computed, this method checks the corresponding entry in the table. If this entry is not "-", the value is retrieved from the table. Otherwise, it is computed recursively and is stored in the corresponding location of the table.

Now, let us use the concept of memory functions to solve the knapsack problem. The various inputs to the algorithm before the algorithm is called are:

- n – number of objects
- m – capacity of the knapsack
- w – is an array consisting of weights of all n objects
- p – is an array consisting of profits of all n objects
- V – is a two dimensional array of size n+1 rows and m+1 columns. This matrix is initialized with -ve numbers.
- Call the following recursive function with i = n and j = m

ALGORITHM f(i,j)

//Purpose: To obtain the objects selected to get the optimal solution using memory function

```
//Input:        i – Number of objects to be selected
//              j – Capacity of the knapsack

//Global variable:
//              w – weights of all the objects

//Output:
//              v – the optimal solution for the number of objects selected with specified
//              remaining capacity. It is also a global variable.
    if ( V[i,j] < 0 )
        if (w[i] > j)
            value ← f (i-1, j)
        else
            value ← Max { f(i – 1, j), f(i – 1, j – w[i]) + p[i] }
        end if

        V[i, j] ← value
    end if
```

Example 3: Apply the memory function method to solve the following instance of the knapsack problem with capacity M = 5

Item	Weight	Value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

Solution: The following data is given

- n = 4
- m = 5
- $(w_1, w_2, w_3, w_4) = (2, 1, 3, 2)$
- $(p_1, p_2, p_3, p_4) = (12, 10, 20, 15)$

Note: The following relation is used to solve the problem using memory function:

$$V[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \quad \text{.....(1)} \\ V[i-1, j] & \text{if } w_i > j \quad \text{.....(2)} \\ \max (V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \le j \quad \text{.....(3)} \end{cases}$$

Step 1: Compute **V[4, 5]:** $i = 4$, $p_4 = 15$, $w_4 = 2$, $j = 5$. Since $w_4 < j$, use equation (3)

$V[4, 5] = \max\{ V[3, 5], V[3, 5-2] + 15\}$

$= \max \{ V[3,5], V[3, 3] + 15 \}$. So, compute V[3,5] and V[3,3]

Step 2: Compute **V[3, 5]:** $i = 3$, $p_3 = 20$, $w_3 = 3$, $j = 5$. Since $w_3 \le j$, use equation (3)

$V[3, 5] = \max\{ V[2, 5], V[2, 5-3] + 20\}$

$= \max \{ V[2,5], V[2, 2] + 20 \}$. So, compute V[2,5] and V[2, 2]

Step 3: Compute **V[3,3]** $i = 3$, $p_3 = 20$, $w_3 = 3$, $j = 3$. Since $w_3 \le j$, use equation (3)

$V[3, 3] = \max\{ V[2, 3], V[2, 3-3] + 20\}$

$= \max \{ V[2,3], V[2, 0] + 20 \}$. So, compute V[2,3] and V[2,0]

Step 4: Compute **V[2,5]** $i = 2$, $p_2 = 10$, $w_2 = 1$, $j = 5$. Since $w_2 \le j$, use equation (3)

$V[2, 5] = \max\{ V[1, 5], V[1, 5 - 1] + 10\}$

$= \max \{ V[1,5], V[1, 4] + 10 \}$. So, compute V[1, 5] and V[1,4]

Step 5: Compute **V[2,2]** $i = 2$, $p_2 = 10$, $w_2 = 1$, $j = 2$. Since $w_2 \le j$, use equation (3)

$V[2, 2] = \max\{ V[1, 2], V[1, 2 - 1] + 10\}$

$= \max \{ V[1,2], V[1, 1] + 10 \}$. So, compute V[1, 2] and V[1,1]

Step 6: Compute **V[2,3]** $i = 2$, $p_2 = 10$, $w_2 = 1$, $j = 3$. Since $w_2 \le j$, use equation (3)

$V[2, 3] = \max\{ V[1, 3], V[1, 3 - 1] + 10\}$

$= \max \{ V[1,3], V[1, 2] + 10 \}$. So, compute V[1, 3] and V[1,2]

Step 7: Compute **V[2,0]** But, **V[2, 0] = 0**

Step 8: Compute **V[1,5]** $i = 1$, $p_1 = 12$, $w_1 = 2$, $j = 5$. Since $w_1 \le j$, use equation (3)

$V[1, 5] = \max\{ V[0, 5], V[0, 5 - 2] + 12\}$

$= \max \{ V[0,5], V[0, 3] + 12 \}$

$= \max \{ \quad 0, \quad 0 \quad + 12\}$

$= 12$

Step 9: Compute **V[1,4]** $i = 1$, $p_1 = 12$, $w_1 = 2$, $j = 4$. Since $w_1 \le j$, use equation (3)

$V[1, 4] = \max\{ V[0, 4], V[0, 4 - 2] + 12\}$

$= \max \{ V[0,4], V[0, 2] + 12 \}$

$= \max \{ \quad 0 \quad , \quad 0 + 12\}$

$= 12$

Step 10: Compute **V[1,2]** $i = 1$, $p_1 = 12$, $w_1 = 2$, $j = 2$. Since $w_1 \le j$, use equation (2)

$$V[1, 2] = \max\{ V[0, 2], V[0, 2 - 2] + 12\}$$
$$= \max \{ V[0,2], V[0, 0] + 12 \}$$
$$= \max \{ 0 , 0 + 12\}$$
$$= 12$$

Step 11: Compute **V[1,3]** $i = 1$, $p_1 = 12$, $w_1 = 2$, $j = 3$. Since $w_1 \le j$, use equation (3)

$$V[1, 3] = \max\{ V[0, 3], V[0, 3 - 2] + 12\}$$
$$= \max \{ V[0,3], V[0, 1] + 12 \}$$
$$= \max \{ 0 , 0 + 12\}$$
$$= 12$$

Step 12: Compute **V[1,1]** $i = 1$, $p_1 = 12$, $w_1 = 2$, $j = 1$. Since $w_1 > j$, use equation (2)

$$V[1, 1] = V[0, 1] = 0$$

Now, substituting backwards,

Step 13: Compute **V[2,3]** (taken from step 6)

$$V[2, 3] = \max \{ V[1,3], V[1, 2] + 10 \}$$
$$= \max \{ 12 , 12 + 10\}$$ // Taken from step 10, 11
$$= 22$$

Step 14: Compute **V[2,2]** (taken from step 5)

$$V[2, 2] = \max \{ V[1,2], V[1, 1] + 10 \}$$
$$= \max \{ 12 , 0 + 10\}$$ // Taken from 10, 12
$$= 12$$

Step 15: Compute **V[2,5]** (taken from 4)

$$V[2, 5] = \max \{ V[1,5], V[1, 4] + 10 \}$$
$$= \max \{ 12 , 12 + 10\}$$ // Taken from 8, 9
$$= 22$$

Step 16: Compute **V[3,3]** (taken from step 3)

$$V[3, 3] = \max \{ V[2,3], V[2, 0] + 20 \}$$
$$= \max \{ 22 , 0 + 20\}$$ // Taken from 13, 7
$$= 22$$

Step 17: Compute **V[3, 5]:** (taken from step 2)

$$V[3, 5] = \max \{ V[2,5], V[2, 2] + 20 \}$$
$$= \max \{ 22, 12 + 20\}$$ // Taken from 15, 14
$$= 32$$

Step 18: Compute **V[4, 5]** (taken from step 1)

V[4, 5] = max { V[3,5], V[3, 3] + 15 }
= max { 32 , 22 + 15} // From 17, 16
= 37

Note: The dynamic programming table that are never computed by the memory function and that are computed and retrieved are shown below:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	-	12	22	-	22
3	0	-	-	22	-	32
N= 4	0	-	-	-	-	37

Profit table

Optimal solution

So, the optimal solution = 37

Exercises:

1. With the help of a pseudocode, explain Warshall's algorithm to find the transitive closure of a directed graph. Apply it to the following graph **(Jan/Feb 2005)**

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

2. Describe an algorithm with an example to compute binomial coefficient and derive its time efficiency **(July/Aug 2004)**

3. Prove that the time efficiency of Warshall's algorithm is cubic

4. Apply Warshall's algorithm to compute transitive closure(path matrix) of the digraph defined by the following adjacency matrix

0	1	0	0
0	0	0	1
0	0	0	0
1	0	1	0

5. Design a $\theta(n^2)$ algorithm for finding an optimal binary search tree **(July/Aug 2004)**

6. Explain Dynamic programming and write how it can be used to compute binomial coefficient. Also write the algorithm for the same

7. Solve all pairs shortest path problem for the digraph with the weight matrix

0	2	∞	1	8
6	0	3	2	∞
∞	∞	0	4	∞
∞	∞	2	0	3
3	∞	∞	∞	0

8. What does dynamic programming have in common with divide and conquer? What is a principal difference between the techniques?

9. Describe an algorithm to compute binomial coefficient and derive its time and space complexity

10. Compute C(6,3) by applying this algorithm

11. State Floyd's algorithm. Use it to solve for all pairs of shortest paths for the graph indicated by the following matrix.

0	2	∞	1	8
6	0	3	2	∞
∞	∞	0	4	∞
∞	∞	2	0	3
3	∞	∞	∞	0

12. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

Item	Weight	Value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

with capacity M = 6

13. Write a pseudocode of the algorithm that finds the composition of an optimal subset from the table generat3e by the bottom-up dynamic programming algorithm for the knapsack problem.

14. Apply the memory function method to solve the following instance of the knapsack problem with capacity M = 6 **(July 2007)**

Item	Weight	Value
1	3	\$25
2	2	\$20
3	1	\$15
4	4	\$40
5	5	\$50

Chapter 9: Greedy Technique

What are we studying in this chapter?

- Concept of Greedy method
- Dijkstra's algorithm
- Prim's algorithm
- Kruskal's algorithm
- Huffman coding

- 3 hours

9.1 Introduction

In this section, let us see "What is greedy strategy?"

Definition: A greedy algorithm is an algorithm that always tries to find the best solution for each sub-problem, with the hope that this will yield an optimal solution for the problem as a whole. Greedy algorithms are simple and straightforward and are easy to invent, easy to implement and most of the time quite efficient. Some well constructed greedy algorithms yield *optimal solutions* while others do not (yield only approximate solutions). For example, the various algorithm that use the concept of greedy approach are:

- Dijkstra's algorithm to find the shortest distance from given source to destination
- Prim's algorithm to find the minimum spanning tree
- Kruskal's algorithm to find the minimum spanning tree
- Huffman coding for data compression and encryption

Now, the question is "Will greedy method yield an optimal solution always?" A greedy algorithm always makes the choice that looks best at the moment. They are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future. That is the reason why, greedy algorithms do not always yield optimal solutions. But, for many problems they yield optimal solutions. This method is quite powerful and works well for a wide range of problems. While solving the problems using this technique, at each step the choice made must be:

- Feasible: The choice made must satisfy the constraint. Any solution that satisfy the given constrain is called feasible solution.
- Locally optimal: The current choice made has to be the best local choice among all feasible solutions
- Irrevocable: Once the choice is made, it should not be changed in subsequent steps of the algorithm

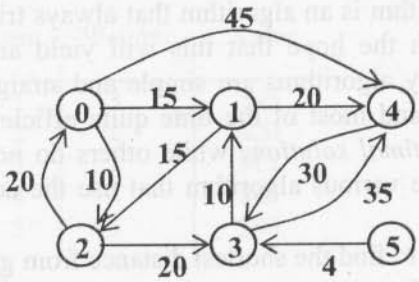
9.2 Dijkstra's Algorithm (Single-Source Shortest Path Problem)

Now, let us see "What is single-source shortest path problem?"

Definition: Given a graph G = (V, E) and a source vertex $v_s \in V$, find the shortest path and shortest distance from v_s to every other vertex v_d in V (v_d is the destination node). Clearly, when we search for the shortest path, we must consider all the vertices in V. If a vertex say v_i is ignored, then we will not consider any of the paths from v_s to v_d through v_i. This problem is not defined for negative numbers.

In short, in single source shortest path problem, we find the shortest distance (may be path also) from a given node to any other node. We can easily find the shortest distance and the path from one node to any other node in the graph using Dijkstra's algorithm. This can be done as shown below:

Example 1: Obtain the shortest distance and shortest path from node 5 to node 1 in the following graph



Solution: Let source = 5 and destination = 0. The cost (weight) adjacency matrix for the given graph is shown below:

W	0	1	2	3	4	5
0	0	15	10	∞	45	∞
1	∞	0	15	∞	20	∞
2	20	∞	0	20	∞	∞
3	∞	10	∞	0	35	∞
4	∞	∞	∞	30	0	∞
5	∞	∞	∞	4	∞	0

Cost (weight) adjacency matrix

D	0	1	2	3	4	5
5	∞	∞	∞	4	∞	0

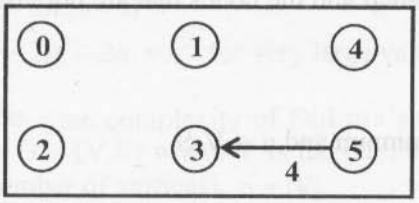
Contains initial distance from 5 to all other nodes

	0	1	2	3	4	5
P	5	5	5	5	5	5

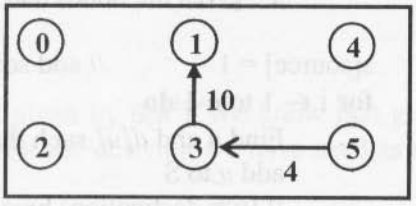
Initial path matrix.

S	v = V- S	d[v] = min(d[v], d[u]+w[u][v])	P[v] = u	find *u*, *d[u]* such that *d[u]* is minimum and *u*∈*V-S*	
5	0,1,2,3,4	—	—	3, 4	1
5, 3	0,1,2,4	d[0] = min(∞, 4 + ∞) = ∞ d[1] = min(∞, 4 + 10) = 14 d[2] = min(∞, 4 + ∞) = ∞ d[4] = min(∞, 4 + 35) = 39	p[1] = 3 p[4] = 3	1, 14	2
5,3,1	0,2,4	d[0] = min(∞, 14 + ∞) = ∞ d[2] = min(∞, 14 + 15) = 29 d[4] = min(∞, 14 + 20) = 34	p[2] = 1 p[4] = 1	2, 29	3
5,3,1,2	0,4	d[0] = min(∞, 29 + 20) = 49 d[4] = min(34,29+∞) = 34	p[0] = 2	4, 34	4
5,3,1,2,4	0	d[0] = min(49, 34 + ∞) = 49	-	0, 49	5
5,3,1,2,4,0	-				

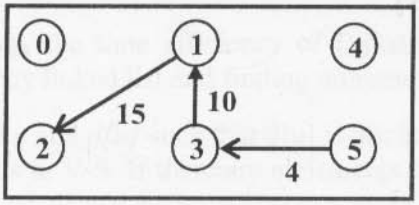
1 5 to 3 is selected with distance 4



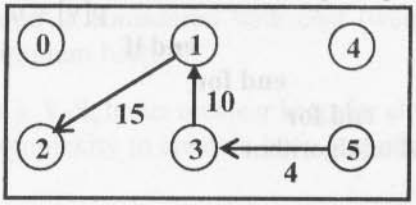
2 Vertex 1 is selected with distance 14
p[1] = 3 ⟹ go to 1 through 3



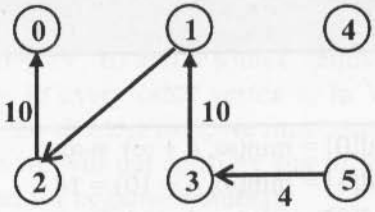
3 Vertex 2 is selected with distance 29
p[2] = 1 ⟹ goto 2 through 1



4 Vertex 4 is selected. Since there is no change in p[4], it is not considered. So, no change in path



5 Vertex 0 is selected with distance 49.
In previous step we have p[0] = 2 ⇨ goto 0 through 2 as shown below:



Note: The shortest distance from 5 to 0 is 49 **using the path 5 → 3 → 1 → 2→ 0**

The algorithm is shown below:

ALGORITHM Dijkstra(n, w, source, destination, d, p)
//Purpose: To compute the shortest distance and shortest path from given *source* to *destination*

//Input: n – Number of vertices in the graph
// w – Cost adjacency matrix with values ≥ 0
// source – source vertex
// destination – destination vertex

//Output: d – shortest distance from source to all other nodes
// p – shortest path from source to destination
// s – gives the nodes that are so far visited and the nodes that are not visited

```
s[source] = 1          // add source to S
for i ← 1 to n-1 do
        Find u and d[u] such that d[u] is minimum and u ∈ V-S
        add u to S
        if (u = destination) break;
        for every v ∈ V-S do  (i.e, for v = 0 to n-1)
                if (d[u] + w[u, v] < d[v])
                        d[v] = d[u] + w[u, v]
                        p[v] = u
                end if
        end for
end for
// end of algorithm
```

Analysis The time efficiency of Dijkstra's algorithm is considered with cost (weight) adjacency matrix and finding minimum in an unordered array.

Step 1: The parameter to be considered is n which represent the size of the input

Step 2: The basic statement is "**if** (d[u] + w[u, v] < d[v])" which is present in the innermost loop. This statement is executed once each time the control enters into the loop.

Step 3: The total number of times the basic statement is executed can be obtained as shown below:

for i ← 1 to n-1 **do**

Find *u* and *d[u]* such that d[u] is minimum and u ∈ V-S (for u = 0 to n-1)

.....

for every v ∈ V-S do (i.e, for v = 0 to n-1)

if (d[u] + w[u, v] < d[v])

$$f(n) = \sum_{i=1}^{n-1}\left(\sum_{j=0}^{n-1}1+\sum_{v=0}^{n-1}1\right)$$

$$= \sum_{i=1}^{n-1}\left[\left((n-1)-0+1\right)+\left((n-1)-0+1\right)\right]$$

$$= \sum_{i=1}^{n-1}2n$$

$$= 2n.\sum_{i=1}^{n-1}1$$

$$= 2n.(n-1-1+1)$$

$$= 2n.(n-1)$$

$$= 2n^2-2n \approx n^2 \text{ for very large value of n}$$

So, the time complexity of Dijkstra's algorithm is given by $\theta(n^2)$. We know that given graph G = (V,E) where V is the number of vertices(while deriving we have used as *n* is the number of vertices), n = |V|

Therefore, time complexity of Dijkstra's algorithm is given by $\theta(\|V^2\|)$

Analysis The time efficiency of Dijkstra's algorithm is considered with cost (weight) adjacency linked list and finding minimum using minimum heap.

To find *u* and *d[u]* such that d[u] is minimum and u ∈ V-S, let us create a heap for all the elements in V-S. If there are *n* elements, the time complexity to create a heap is given by

O(n). See the section 6.5.2, for details. In the given graph G = (V, E), n = |V|. Therefore, the time complexity to create a heap is given by O(|V|). Now, the root will be the least item and can be deleted. This is achieved by exchanging the root with $(n-1)^{th}$ item and recreating the heap for remaining (n-1) elements which depends on the height of the tree. We know that the time complexity to re-create the heap for (n-1) elements with height h is given by

$$2\log_2(n-1) \approx \log_2 n \quad \text{(For details, see page 6.50)}$$

where n is the number of elements. If $n = |V|$, the time complexity is given by

$$O(\log_2 V)$$

This is true for each edge selected. Since all edges E should be considered to find the shortest path and distance, the time complexity is given by $E\log_2 V$.

Therefore, time complexity of Dijkstra's algorithm is given by **θ(E log V)**

The algorithm to print the shortest path from a given source node to destination node is shown below:

ALGORITHM print_shortest_path(n, source, destination, p, d)

//Purpose: To print the shortest path from given *source* to *destination*

//Input: n – Number of vertices in the graph
// source – starting vertex
// destination – destination vertex
// p – shortest path from source to destination
// d – shortest distance from source to destination

//Output: Display the shortest path and distance

```
    i ← destination
    while (i ≠ source)
        write i, '←'
        i ← p[i]
    end while
    write source, '=', d[destination]
// end of the algorithm
```

Note: The Dijkstra's algorithm using cost adjacency matrix is efficient, provided there are more number of edges in a graph. If there are less number of edges, then it is better to implement Dijkstra's algorithm using "adjacency list" and implement "binary heap" to find u and d[u] such that d[u] is minimum and u ∈ V-S.

9.3 Prim's Algorithm

Prim's algorithm is used to find minimum spanning tree. Now, let us see "What is a spanning tree?"

Definition: A spanning tree is a tree in which all nodes are connected without forming a circuit. Formally, a spanning tree of graph G is defined as a sub graph $G' = (V', E')$ with the following properties:

- $V' = V$
- G' is connected
- G' is acyclic

For example, consider the graph shown figure 9.3.1a. The various spanning trees of this graph are shown in figures 9.3.1b through 9.3.1e.

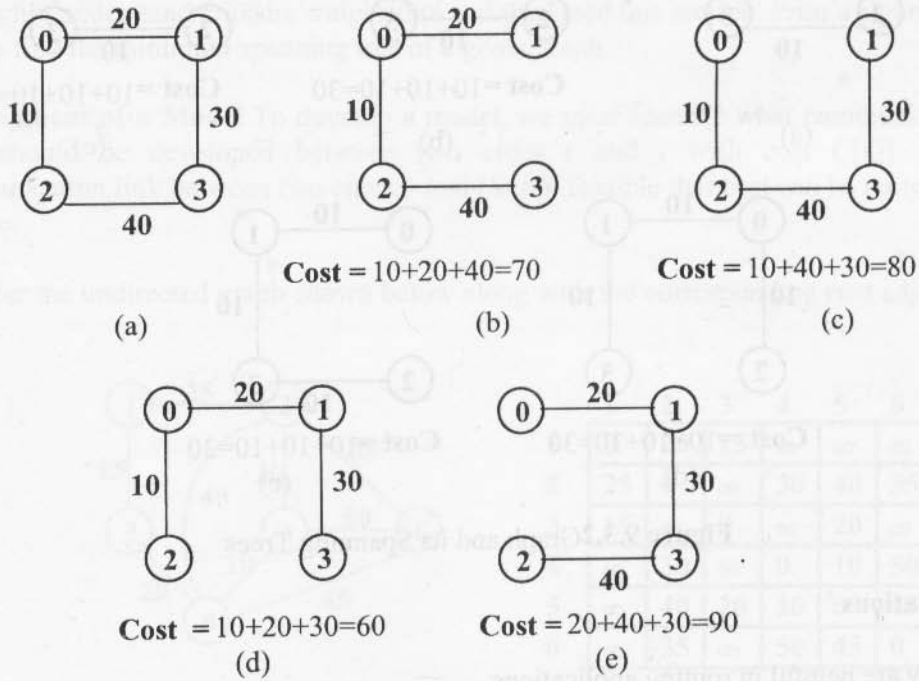


Figure 9.3.1 Graph and its Spanning Trees

Now, let us see "What is a minimum spanning tree?"

Definition: A minimum spanning tree of a given graph G is a spanning tree whose cost is minimum. For the graph shown in figure 9.3.1a, the minimum spanning tree is shown in figure 9.3.1d because the cost of that spanning tree is minimum.

Now, the question is "Given a graph G, will it have only one minimum spanning tree or more than one spanning tree?" A graph will have only one minimum spanning tree if and only if the weights associated will all the edges in the graph are distinct. For example, the graph shown in figure 9.3.1.a, there exists only one minimum spanning as shown in figure 9.3.1.d.

But, if some of the edges have same weights, then the graph will have more than one minimum spanning tree. For example, consider the graph shown in figure 9.3.2.a. It has four minimum spanning trees whose cost remains same. The minimum spanning trees are shown in figures 9.3.2.b to 9.3.2.e.

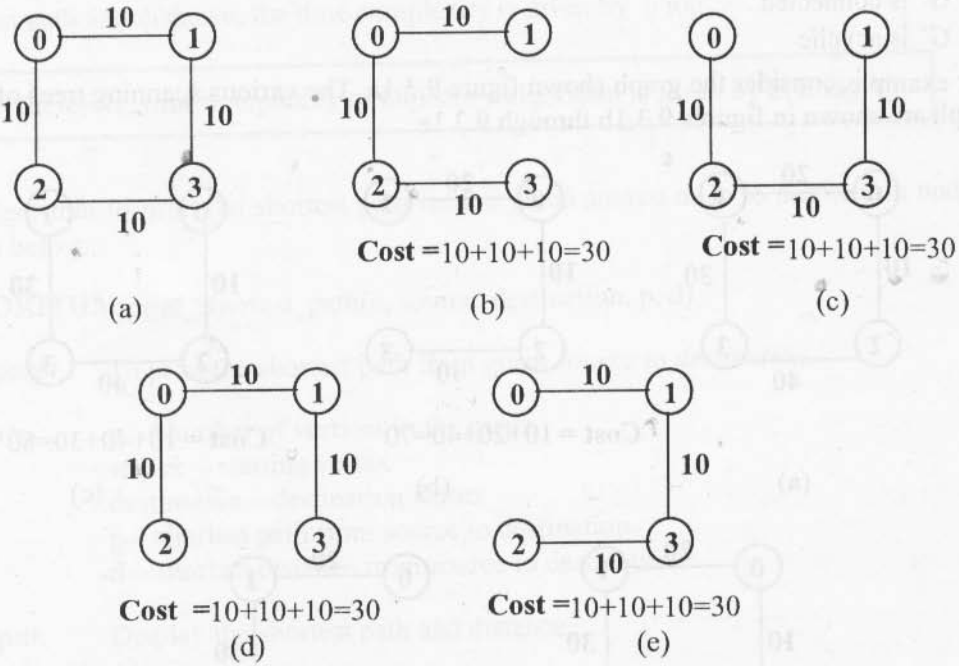


Figure 9.3.2 Graph and its Spanning Trees

Applications

- They are helpful in routing applications
- Communication networks

The Minimum Spanning Tree can be used in many of the applications. Let us take a situation where this algorithm can be applied. Suppose there are 5 cities and wants to establish computer connectivity so that an organization in one city can communicate to any other company located in different city. The various factors to be considered while setting up such computer facilities are:

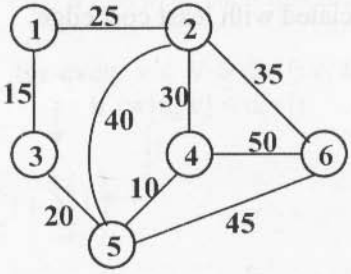
- Various resources available at each site
- Peak demands of the system
- The cost of the proposed network etc.

In this situation if we consider the cost of the proposed network as a key factor, our intention should be to minimize this cost. The cost of the proposed communication network may involve purchasing the equipments, establishing communication links, system maintenance etc. The cost involved for communication links between any two cities can be obtained by consulting a representative. The cost again depends on the distance; transmission rate desired and desired capacity of the line. The cost of such details can be obtained and the approximate cost C[i,j] between two cities *i* and *j* can be calculated. Now, our intention should be to determine the minimum cost of such a communicating network.

Note: Prim's algorithm is the modified version of Dijkstra's algorithm. Hence, thoroughly understand Dijkstra's algorithm and then read this section. Prim's algorithm is used to find the minimum spanning tree of a given graph.

Development of a Model To develop a model, we must identify what communications links should be developed between two cities *i* and *j* with cost C[i,j]. If the communication link between two cities *i* and *j* is not feasible that cost can be represented using ∞.

Consider the undirected graph shown below along with the corresponding cost adjacency matrix.



	1	2	3	4	5	6
1	0	25	15	∞	∞	∞
2	25	0	∞	30	40	35
3	15	∞	0	∞	20	∞
4	∞	30	∞	0	10	50
5	∞	40	20	10	0	45
6	∞	35	∞	50	45	0

The minimum spanning tree can be obtained by modifying the Dijkstra's algorithm discussed in the section 9.2. In Dijkstra's algorithm we used the statement:

```
if (d[u] + w[u][v] < d[v])
        d[v] = d[u] + w[u][v]
        p[v] = u
end if
```

The change required in Prim's algorithm is to remove d[u] in the above *if* statement The resulting statement is:

```
if (w[u][v] < d[v])
        d[v] = w[u][v]
        p[v] = u
end if
```

Another statement to be modified is "After adding *u* to S" add the following statements:

- Store the edge (u, p[u])
- Find the cost of the spanning tree by adding u+p[u] to sum

After considering all edges in the graph, if the cost of spanning tree is greater than or equal to ∞, the spanning tree does not exist; Otherwise, spanning tree exists. The spanning tree thus obtained is the minimum spanning tree. The Prim's algorithm is shown below:

```
ALGORITHM PRIM(n, w)
//Purpose:   To compute the minimum spanning tree
//Input:     n – Number of vertices in the graph
//           w – Cost adjacency matrix with values ≥ 0
//Output:    d – shortest distance from source to all other nodes
//           p – shortest path from source to destination
//           s – gives the nodes that are so far visited and the nodes that are not visited

    // Obtain a source vertex which is associated with least cost edge

    for i ← 0 to n-1 do
        s[i] = 0
        d[i] = w[source, i]
        p[i] = source
    end for

    s[source] = 1          // add source to s
    sum ← 0                // Initial cost of minimum spanning tree
    k ← 0                  // used as index to store the edges selected
```

```
for i ← 1 to n-1 do
        Find u and d[u] such that d[u] is minimum and u ∈ V-S
        Add u to S
        Select an edge with least cost
        Add cost associated with edge to get total cost of minimum spanning tree

        for every v ∈ V-S do
                if (w[u, v] < d[v])
                        d[v] = w[u, v]
                        p[v] = u
                end if
        end for
end for

if (sum >= 9999)
        write 'Spanning tree does not exist'
else
        write 'Spanning tree exists and print the minimum spanning tree is'
end if
// end of algoirthm
```

Analysis The time efficiency of Prim's algorithm is considered with cost (weight) adjacency matrix and finding minimum in an unordered array.

Step 1: The parameter to be considered is n which represent the size of the input

Step 2: The basic statement is "**if** (w[u, v] < d[v])" which is present in the innermost loop. This statement is executed once each time the control enters into the loop.

Step 3: The total number of times the basic statement is executed can be obtained as shown below:

```
for i ← 1 to n-1 do
        Find u and d[u] such that d[u] is minimum and u ∈ V-S (for u = 0 to n-1)
        .....
        for every v ∈ V-S do  (i.e, for v = 0 to n-1)
                if (w[u, v] < d[v])
```

$$f(n) = \sum_{i=1}^{n-1}\left(\sum_{j=0}^{n-1}1+\sum_{v=0}^{n-1}1\right)$$

$$= \sum_{i=1}^{n-1}\left[((n-1)-0+1)+((n-1)-0+1)\right]$$

$$= \sum_{i=1}^{n-1} 2n$$
$$= 2n.\sum_{i=1}^{n-1} 1$$
$$= 2n.(n-1-1+1)$$
$$= 2n.(n-1)$$
$$= 2n^2 - 2n \approx n^2 \text{ for very large value of n}$$

So, the time complexity of Prims's algorithm is given by $\theta(n^2)$. We know that given graph G = (V,E) where V is the number of vertices(while deriving we have used as *n* is the number of vertices), n = |V|

Therefore, time complexity of Prims's algorithm is given by $\theta(\|V^2\|)$

Analysis The time efficiency of Prims's algorithm is considered with cost (weight) adjacency linked list and finding minimum using minimum heap.

To find *u* and *d[u]* such that d[u] is minimum and u ∈ V-S, let us create a heap for all the elements in V-S. If there are *n* elements, the time complexity to create a heap is given by O(n). See the section 6.5.2, for details. In the given graph G = (V, E), n = |V|. Therefore, the time complexity to create a heap is given by O(|V|). Now, the root will be the least item and can be deleted. This is achieved by exchanging the root with $(n-1)^{th}$ item and recreating the heap for remaining (n-1) elements which depends on the height of the tree. We know that the time complexity to re-create the heap for (n-1) elements with height h is given by

$$2\log_2(n-1) \approx \log_2 n \quad \text{(For details, see page 6.50)}$$

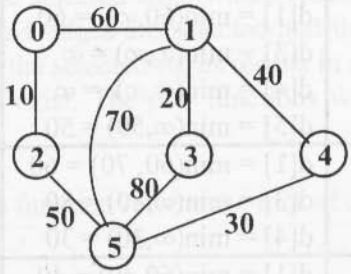
where *n* is the number of elements. If *n* = |V|, the time complexity is given by

$$O(\log_2 V)$$

This is true for each edge selected. Since all edges E should be considered to find the shortest path and distance, the time complexity is given by $E\log_2 V$.

Therefore, time complexity of Prim's algorithm is given by **θ(E log V)**

Example 1: Find the minimum spanning tree for the following graph by giving the tracing:



The cost adjacency matrix for the above graph is shown below:

	0	1	2	3	4	5
0	0	60	10	∞	∞	∞
1	60	0	∞	20	40	70
2	10	∞	0	∞	∞	50
3	∞	20	∞	0	∞	80
4	∞	40	∞	∞	0	30
5	∞	70	50	80	30	0

The least cost is 10 and hence the edge (0, 2) is considered and source = 0 (We can take vertex 2 also as the source). The initial matrix *d* and *p* are shown below:

	d
0	0
1	60
2	10
3	∞
4	∞
5	∞

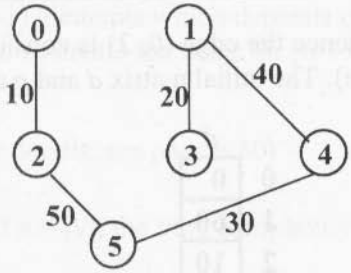
	p
0	0
1	0
2	0
3	0
4	0
5	0

The various activities and computations to get the minimum spanning tree using Prim's algorithm is shown in table 9.2.

S	v = V-S	d[v] = min(d[v], cost[u][v])	P[v] = u	*u, d[u]*	Output u, p[u]
0	1,2,3,4,5	—	—	2, 10	2, 0
0, 2	1, 3, 4, 5	d[1] = min(60,∞) = 60 d[3] = min(∞,∞) = ∞ d[4] = min(∞,∞) = ∞ d[5] = min(∞,50) = 50	p[5] = 2	5, 50	5, 2
0, 2, 5	1, 3, 4	d[1] = min(60, 70) = 60 d[3] = min(∞,80) = 80 d[4] = min(∞,30) = 30	p[3] = 5 p[4] = 5	4, 30	4, 5
0, 2, 5, 4	1, 3	d[1] = min(60,40) = 40 d[3] = min(80, ∞) = 80	p[1] = 4	1, 40	1, 4
0, 1, 2, 5, 4	3	d[3] = min(80, 20) = 20	p[3] = 1	3, 20	3, 1
0, 1, 2, 3, 4, 5	-				

Table 9.2 To find the minimum spanning tree using Prim's algorithm

The minimum spanning tree obtained (last column of the table in 9.2) is given by the following edges: (2, 0), (5, 2), (4, 5), (1, 4) and (3, 1) and the minimum spanning tree obtained is shown below:



The cost of the minimum spanning tree = 10 + 50 + 30 + 20 + 40 = 150

9.4 Kruskal's algorithm

Now, let us see "What is the purpose of Kruskal's algorithm?" Every spanning tree of *n* vertices will have n-1 edges. The problem here is to select n-1 edges such that the cost of the spanning tree is minimum. Like Prim's algorithm, using Kruskal's algorithm also we can find the minimum spanning tree (MST).

General procedure: Let G = (V, E) be a directed graph. This algorithm selects n-1 edges one at a time. The algorithm selects the least cost edge (u, v) from E and delete the edge (u, v) from E. Then, it checks whether the selected edge (u, v) results in a cycle when added to the list of already selected edges to form a *minimum spanning tree*(MST). If no cycles are formed, the selected edge is added to the list of edges selected to form a *minimum spanning tree*(MST). The selected edge which results in a cycle is neglected. The procedure is repeated till all n-1 edges are selected and there are no more edges to consider. If n-1 edges are selected, the selected edges results in a *minimum spanning tree;* otherwise, spanning tree does not exist. The two functions which are essential for the design of Kruskal's algorithm are:

find(u): If u is a node in a tree, this function returns the root of the corresponding tree. For example, consider the sets

$$\{1, 4, 8, 6\} \text{ and } \{2, 3, 7, 5\}$$

These two sets represent two different trees. The first item in each set acts as the root of the tree. The above two sets can be represented as trees as shown below:

Now, the root node for the nodes 4, 8 and 6 is 1 which is indicated by

find(4) = find (8) = find (6) = 1.

Similarly, the root node for the nodes 3, 7 and 5 is 2 which is indicated by

find(3) = find(7) = find(5) = 2

union(i, j): Let C_1 and C_2 be two trees. If *i* is a vertex in the tree C_1 and *j* is a vertex in the tree C_2, this function merges the two trees C_1 and C_2 into a single tree. For example, consider the sets:

$$S_1 = \{1\}\ S_2 = \{2\}\ S_3 = \{3\}\ S_4 = \{4\}\ S_5 = \{5\}\ S_6 = \{6\}\ S_7 = \{7\}\ S_8 = \{8\}.$$

All these sets can be represented using array as shown below:

S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
1	2	3	4	5	6	7	8

The above array indicates that there are 8 trees. Now, $S_2 \cup S_5$ is represented by storing 2 in S_5(i.e., S[5]) as shown below:

S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
1	2	3	4	2	6	7	8

The above array can be represented using set notation as shown below:

$$S_1 = \{1\}\ S_2 = \{2, 5\}\ S_3 = \{3\}\ S_4 = \{4\}\ S_6 = \{6\}\ S_7 = \{7\}\ S_8 = \{8\}$$

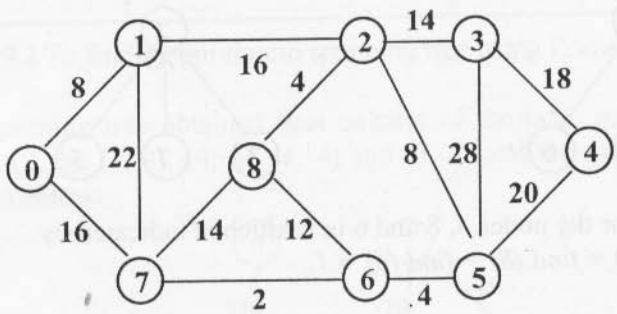
Now, $S_6 \cup S_7$ can be represented using an array by storing 6 in S[7] as shown below:

S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]
1	2	3	4	2	6	6	8

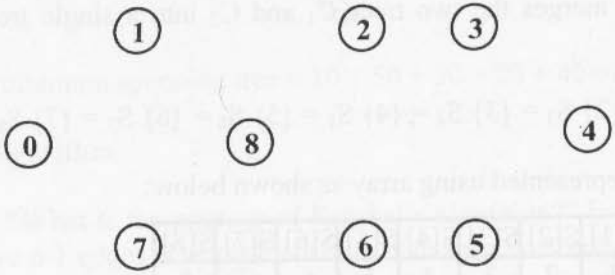
The above array can be represented using set notation as shown below:

$$S_1 = \{1\}\ S_2 = \{2, 5\}\ S_3 = \{3\}\ S_4 = \{4\}\ S_6 = \{6, 7\}\ S_8 = \{8\}$$

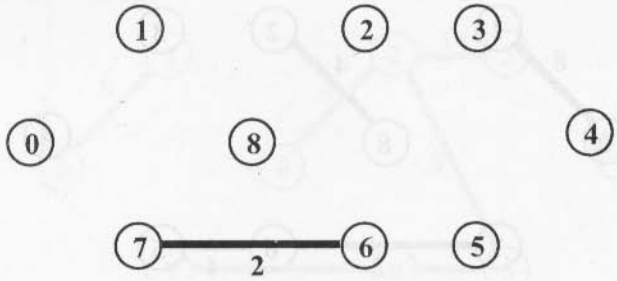
Example 9.4.1: Find the minimum spanning tree using Kruskal's algorithm



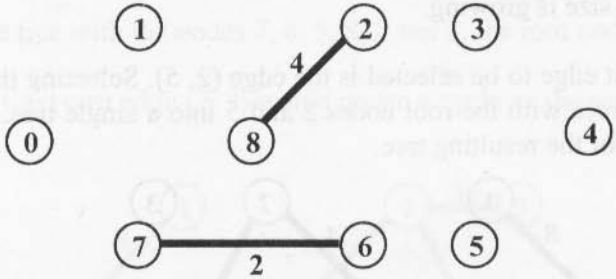
Solution: The minimum spanning tree can be obtained as shown below: In the beginning, no edge is selected thus forming a forest of 9 trees as shown below:



Step 1: The cost of the edge(6, 7) is least and hence, it is selected as shown in figure resulting in 8 trees. Now, for the tree consisting of nodes 6 and 7, node 6 is the root.

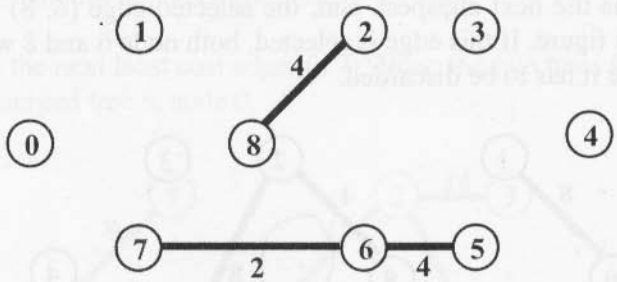


Step 2: The next least cost edge (2, 8) is selected next as shown in figure below:



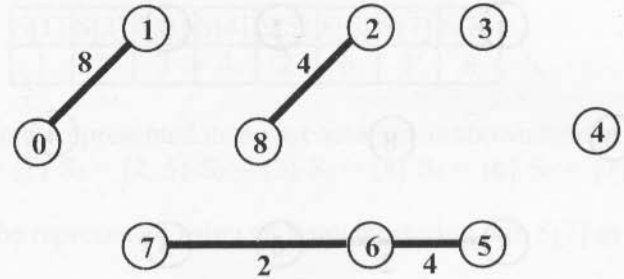
Now the forest has 7 trees. In the tree consisting of the nodes 2 and 8, node 2 is the root node.

Step 3: Edge (5, 6) is the next least cost edge. Add this edge and choose vertex 5 as the root node.



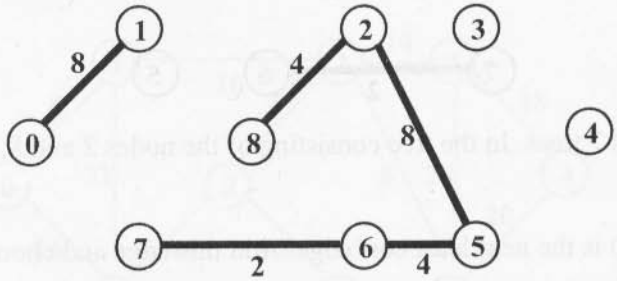
Now, the number of trees in the forest is reduces to 6. The tree with nodes 5, 6 and 7 has 5 as the root node.

Step 4: Edge (0, 1) with next least edge is selected and node 0 will be the root node for that tree.

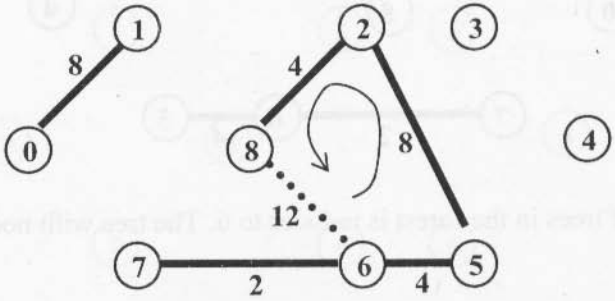


Now, there are 5 trees. Observe that at each step, the numbers of trees in the forest are reducing and the tree size is growing.

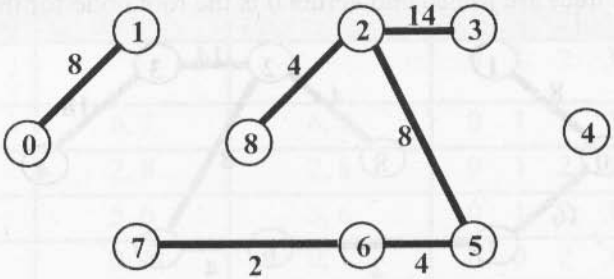
Step 5: The next least edge to be selected is the edge (2, 5). Selecting that edge results in merging of the two trees with the root nodes 2 and 5 into a single tree. Now, the node 2 will be the root node of the resulting tree.



Step 6: Edge (6, 8) is the next cheapest. But, the selected edge (6, 8) would result in a cycle as shown in the figure. If this edge is selected, both node 6 and 8 will have the same root node 2 and hence it has to be discarded.

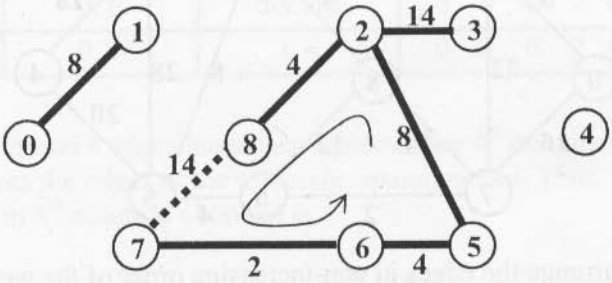


Step 7: So, the edge (6, 8) is discarded and the next edge (2, 3) is selected.



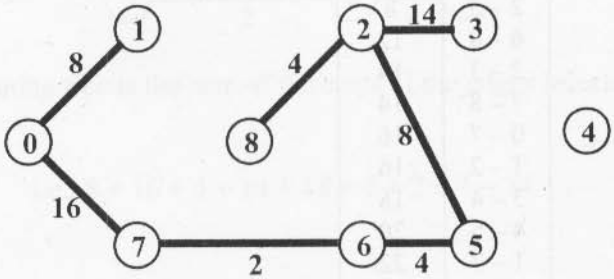
Note: Now, in the tree with the nodes 7, 6, 5, 8, 2 and 3, the root node is 2.

Step 8: The next least cost edge (7, 8) would create a cycle as shown below:

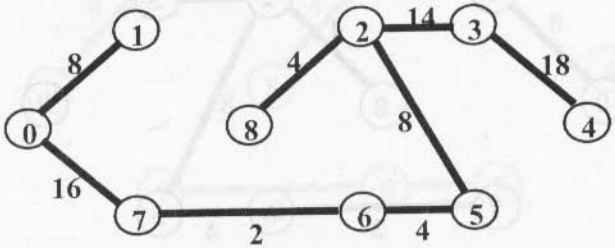


and hence the edge should not be considered.

Step 9: So, select the next least cost edge (0, 7). Now, the two trees are merged and the root node of the merged tree is node 0.

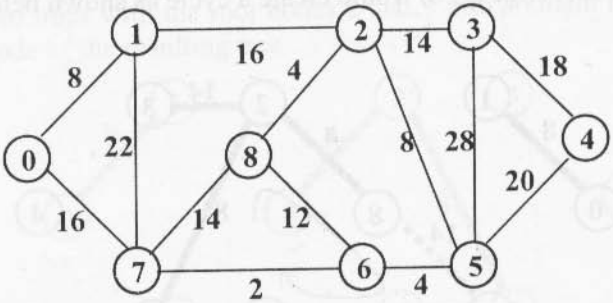


Step 10: The next edge to be selected is (1, 2). Since, addition of this edge results in a cycle, discard it and select the next least cost edge (3, 4) to complete the spanning tree. In this spanning tree all trees are joined and vertex 0 is the root node for the entire tree.



The cost of this minimum spanning tree is given by the sum of the weights of the selected edges i.e., 8 + 16 + 4 + 14 + 18 + 8 + 2 + 4 = 74

Example 9.4.2: Find the minimum spanning tree using Kruskal's algorithm (using tabular method)



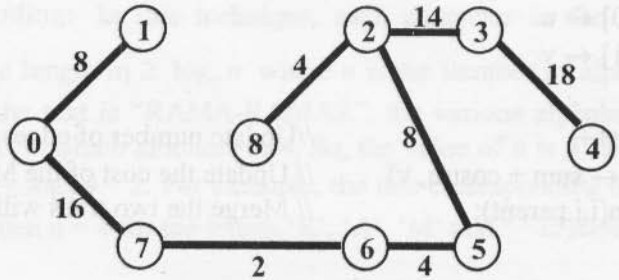
Note: First, let us arrange the edges in non-increasing order of the weights as shown below:

Edge	Cost
6 – 7	2
2 – 8	4
5 – 6	4
0 – 1	8
2 – 5	8
6 – 8	12
2 – 3	14
7 – 8	14
0 – 7	16
1 – 2	16
3 – 4	18
4 – 5	20
1 – 7	22
3 – 5	28

The solution to the above problem can be represented in the form of table as shown below:

Steps	Deleted edge (u, v)	i = find(u), j = find(v)	Output (u ,v)	UNION(i, j) S[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	-	-	-	0	1	2	3	4	5	6	7	8
1	6, 7	6, 7	6, 7	0	1	2	3	4	5	6	6	8
2	2, 8	2, 8	2, 8	0	1	2	3	4	5	6	6	2
3	5, 6	5, 6	5, 6	0	1	2	3	4	5	5	6	2
4	0, 1	0, 1	0, 1	0	0	2	3	4	5	5	6	2
5	2, 5	2, 5	2, 5	0	0	2	3	4	2	5	6	2
6	6, 8	2, 2	discard	0	0	2	3	4	2	5	6	2
7	2, 3	2, 3	2, 3	0	0	2	2	4	2	5	6	2
8	7, 8	2, 2	discard	0	0	2	2	4	2	5	6	2
9	0, 7	0, 2	0, 7	0	0	0	2	4	2	5	6	2
10	1, 2	0, 0	discard	0	0	0	2	4	2	5	6	2
11	3, 4	0, 4	3, 4	0	0	0	2	0	2	5	6	2

Since there are 9 nodes and 8 edges have been selected (see 4th column in the above table), 4th column gives the edges of the minimum spanning tree. Thus, the minimum spanning tree (given in 4th column) obtained is



The cost of the spanning tree is the sum of the costs of the edges selected which is given by:

i.e., 8 + 16 + 4 + 14 + 18 + 8 + 2 + 4 = 74

The algorithm to find the minimum spanning tree (MST) using Kruskal's algorithm is shown below:

```
ALGORITHMKRUSKAL(n, m, E)
//Purpose:    To compute the minimum spanning tree using Kruskal's algorithm
//Input:      n – Number of vertices in the graph
//            m – Number of edges in the graph
//            E –edge list consisting of set of edges along with equivalent weights
//Output:     The minimum spanning tree along
count ← 0      // Initial number of edges
k ← 0          // Points to the first selected edge of MST
sum ← 0        // Initial cost of MST
// Create a forest with n vertices
for i ← 0 to n do
    parent[i] ← i
end for
while (count ≠ n-1 and E ≠ ϕ)
    Select an edge (u, v) with least cost
    i ← find(u,parent)      // Find the root for the vertex u
    j ← find(v,parent)      // Find the root for the vertex v
    if ( i != j )           // If the roots of vertex u and v are different
        //Select the edge (u, v) as the edge of MST
        t[k][0] ← u
        t[k][1] ← v
        k++
        count++                      //Update number of edges selected for MST
        sum ← sum + cost(u, v)       // Update the cost of the MST
        union(i,j,parent);           // Merge the two trees with the roots i and j
    end if
    // Delete the edge (u,v) from the list
end while

if ( count != n-1 )
    write ("Spanning tree does not exists")
    return
end if
```

```
write ("The spanning tree is shown below")
for i ← 0 to n-2 do
        write(t[i][0],t[i][1])
end for

write ("Cost of spanning tree is", sum )
// end of the algorithm
```

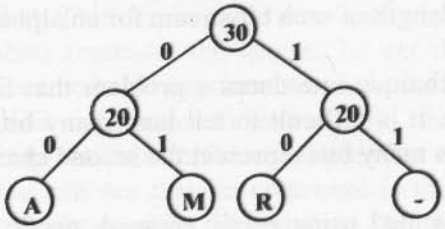
9.5 Huffman Trees

This section deals with how a string or text can be en-coded(compressed) and de-coded (de-compressed) so as to save the space. When a huge text or message has to be transmitted, it is encoded first and then the encoded message is transmitted. On the other hand, the encoded message can be decoded to get the original text or message. This is where Huffman coding scheme comes into picture.

Huffman code is widely used because it is an effective technique for compressing and decompressing the data. Huffman greedy algorithm scans the text and finds the frequency of each character and stores in the form of a table. This table is used to generate a unique code for each character in the given text. This code is called *binary character code* or *codeword*. The *codeword* thus obtained is unique for each character. The two types of encoding schemes are:

- Fixed-length encoding
- Variable-length encoding

Fixed-length encoding: In this technique, each character in the text is assigned a bit string of the same length $m \geq \log_2 n$ where n is the number of alphabets in the text. For example, if the text is "RAMA-RAMAR", the various alphabets in the text are 'R', 'A', 'M' and '-' which amounts to 4. So, the value of n is 4 and bit string length m is given by $m = \log_2 4 = 2$. For example, the tree corresponding to the fixed-length coding scheme when n = 4 for the letters 'R', 'A', 'M' and '-' is shown below:



Note: If the number of characters are n = 4, 5, 6, 7 and 8, then minimum 3-bits are required for generating the codeword. When n = 9, minimum 4-bits are required for representing the codeword.

Assume that the text "RAMA-RAMAR" is repeated 100 times in a file. Now, the file size is 10*100 = 1000 bytes (8000bits). The characters in the text and the frequency of each character in the file is shown in table below:

Characters in the file	A	M	R	-
Frequency	400	200	300	100
Fixed-length	00	01	10	11

It is clear from the above table that the letter 'A' occurs 400 times, the letter 'M' appears 200 times, the letter 'R' appears 300 times and the symbol '-' appears 100 times. The corresponding bit string is also shown for each character. Thus, the string

RAMA-RAMAR

can be encoded using bit string as shown below:

10000100111000010010
R A M A - R A M A R

Note that the length of bit string is only 20 bits. Since, the text is repeated 100 times, the bit-pattern is also repeated 100 times and so the total size of the encoded file will be 100*20 (2000 bits). Note that original file size is 8000 bits. Thus, a given text can be compressed to a very large extent. During decoding, replace every 2-bits by the corresponding alphabet to get the given text.

Variable-length encoding Another way of obtaining the bit stream is to use shorter *codewords* for frequently appearing characters and longer *codewords* for less frequently occurring characters. Thus, the length of each bit-stream for an alphabet is different.

Note: During decoding this technique introduces a problem that fixed-length encoding does not have. This is because, it is difficult to tell how many bits of an encoded text represent the first character, how many bits represent the second character and so on.

This problem can be easily avoided using *prefix code*. A *prefix code* is a *codeword* wherein no code word is the prefix of another character. So, during decoding, we can

scan a bit string until we get the first group of bits which corresponds to a codeword for some character, replace the bit stream by a corresponding character and the process is repeated till the end. For example, consider the text "RAMA-RAMAR" and assume the string is repeated 100 times in a file as shown in fixed-length scheme. The characters in the text and the frequency of each character and along with variable-length prefix code is shown in table below:

Characters in the file	A	M	R	-
Frequency	400	200	300	100
variable-length	1	000	01	001

Note that the code of any alphabet in the table is not the prefix of any other code. Now, the string

RAMA-RAMAR

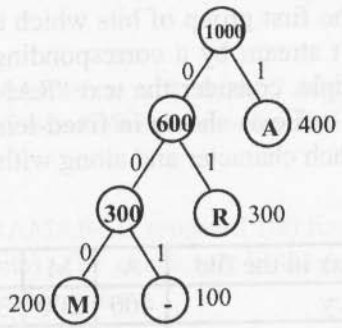
can be encoded using bit string as shown below:

01 1 000 1 001 01 1 000 1 01
R A M A - R A M A R

Note that the length of bit string is 19 bits. Since, the text is repeated 100 times, the bit-pattern is also repeated 100 times and so the total size of the encoded file will be 100*19 (1900 bits). Note that original file size is 8000 bits. Thus, a given text can be compressed to a very large extent using variable-length scheme. During decoding, obtain the first bit 0, it is not the code for any alphabet. Obtain the next bit 1 to get the codeword 01 which corresponds to R. Next bit is 1 which corresponds to A and repeat this process to finally get the decoded string

RAMA-RAMAR

Thus, encoding (compression) and decoding(de-compression) can be done very easily and efficiently using variable-length encoding. Now, the problem is how to find codeword for an alphabet. This can be done very easily by constructing a binary tree of codewords. The alphabets represent the leaves. To get the codeword for a character, select a path from the root node till the character at the leaf level is reached. In this path, left move indicates 0 and right move indicates 1. So, the final bit string obtained by moving left or right from the root to the corresponding character gives the codeword. Using this approach, we can see that no codeword is the prefix of another code. For example, consider the tree obtained for the alphabets 'A', 'R', '-' and 'M'.



Note that if we start from root, with the label 1 we reach A. So, codeword for A is 1. To reach the character 'M' from root, the labels in that path will be 000 which is the codeword for alphabet M and so on. The resulting codewords for each character are shown below:

Characters in the file	A	M	R	-
variable-length	1	000	01	001

The tree using which codewords are generated such that no codeword is the prefix of another codeword is called *Huffman* tree. Now, let us concentrate on how to get the Huffman tree. This tree is named after David Huffman who invented this method of encoding and decoding as a part of class assignment when he was a student. The algorithm is shown below:

Algorithm Huffman_Tree(text, T)
// **Purpose:** To create a tree T so as to generate the codeword for alphabets so that no
// codeword is prefix of another codeword
// **Input:** Text containing various alphabets
// **Output:** A tree T using which the codewords can be obtained for a given character

Step 1: Create n trees where n is the number of alphabets such that each node contains the alphabet and the frequency of alphabet.

Step 2: Identify the two trees with smallest frequencies. Make one of them as the left child and the other as the right child of a new tree. The sum of frequencies of left child and right child is stored in the root node of the new tree. Repeat step 2 until a single tree is obtained.

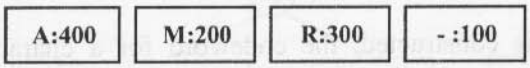
Step 3: To get a codeword for any alphabet which is at the leaf level, start from root till the leaf node is obtained. Whenever we move towards left read as 0 and whenever we move towards right read as 1. The strings of 0's and 1's thus obtained while moving from a root down to a leaf is the codeword of the corresponding alphabet.

Example 9.5.2: Obtain the Huffman code for the following data:

Character	A	M	R	-
Frequency	400	200	300	100

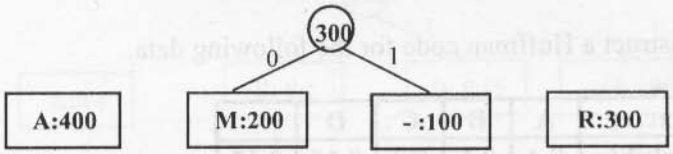
Solution: The Huffman code can be easily obtained by constructing the Huffman tree as shown below:

Step 1: Four trees are created for the characters 'A', 'M', 'R' and '-', and are considered as leaf nodes. Each character in the leaf is followed by the frequency of that character in a given text as shown below:

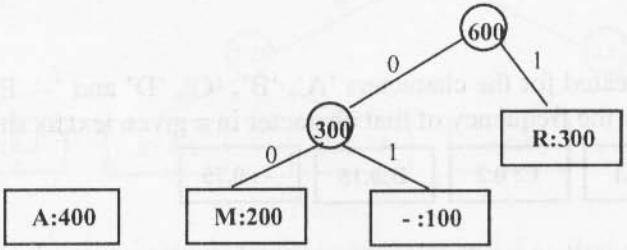


Step 2: Identify the two trees with smallest frequencies and make one of them as the left child and the other as the right child of a new tree. The sum of frequencies of left child and right child is stored in the root node of the new tree.

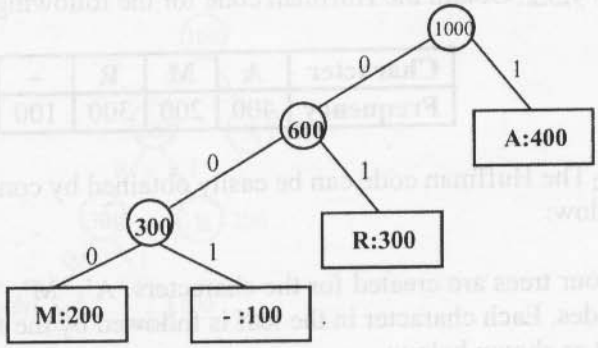
- The two trees with lowest frequencies 100 and 200 are merged into a single tree whose root node is the sum of the frequency of left child and right child. The tree along with other trees who have not considered as shown below:



- Out of the above 3 trees, again select two trees with least weights (consider root node information). Note that the second and third trees in the above figure can be merged into two trees with new root node whose value is the sum of the roots of the two trees as shown below:



- Now, the two trees with root node values 400 and 600 can be merged into a tree having a root node whose value is the sum of 400 and 600 as shown below:



Once the Huffman tree is constructed, the codeword for a character is obtained by following the path from root node till the character is obtained and by concatenating all the bits obtained in that path. The codeword for 'A' is 1, codeword for 'R' is 01, codeword for '-' is 001 and that of 'M' is 200 and is shown in the table below:

Character	A	M	R	-
Frequency	400	200	300	100
Codeword	1	000	01	001

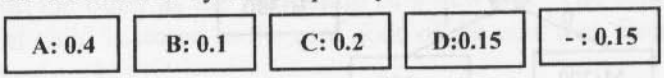
Example 9.5.3: Construct a Huffman code for the following data:

Character	A	B	C	D	-
Probability	0.4	0.1	0.2	0.15	0.15

Encode the text ABACABAD and decode the encoded text 100010111001010

Solution: The encoding and decoding can be done by first obtaining the Huffman code. The Huffman code can be easily obtained by constructing the Huffman tree as shown below:

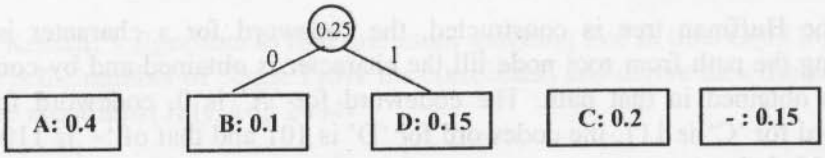
Step 1: Five trees are created for the characters 'A', 'B', 'C', 'D' and '-'. Each character in the tree is followed by the frequency of that character in a given text as shown below:



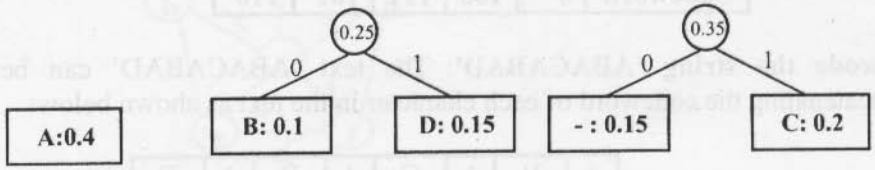
Note that all these trees can be considered as the trees with single node.

Step 2: Identify the two trees with smallest frequencies and make one of them as the left child and the other as the right child of a new tree. The sum of frequencies of left child and right child is stored in the root node of the new tree.

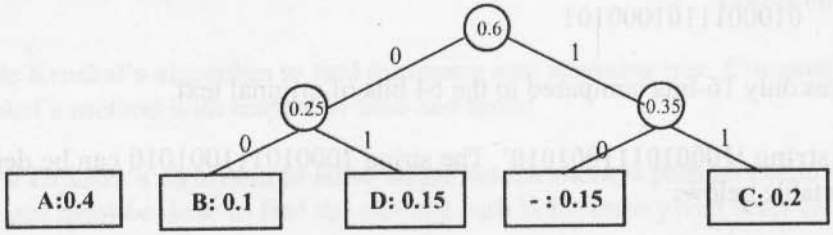
- The two trees with lowest frequencies 0.1 and 0.15 are merged into a single tree whose root node is the sum of the frequency of left child and right child. The tree along with other trees who have not considered is as shown below:



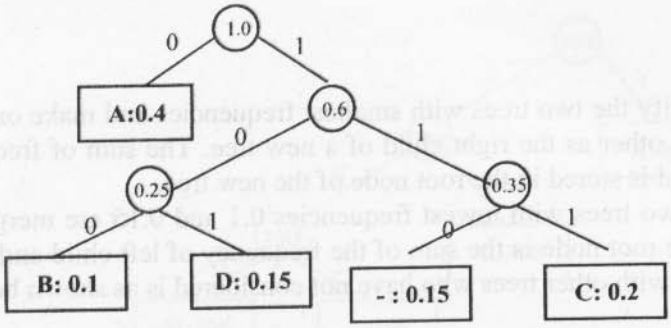
- Out of the above 4 trees, select two trees with least weights (consider root node information). Note that the third and fourth trees can be merged into a single tree with new root node whose value is the sum of the roots of the two trees as shown below:



- Now, the two trees with root node values 0.25 and 0.35 (second and third trees) can be merged into a tree having a root node whose value is the sum of 0.25 and 0.35 as shown below:



Now, the two trees with root node values 0.4 and 0.6 (first and second trees) can be merged into a single tree as shown below:



Once the Huffman tree is constructed, the codeword for a character is obtained by following the path from root node till the character is obtained and by concatenating all the bits obtained in that path. The codeword for 'A' is 0, codeword for 'B' is 100, codeword for 'C' is 111, the codeword for 'D' is 101 and that of '-' is 110 and is shown in the table below:

Character	**A**	**B**	**C**	**D**	**-**
Frequency	**0.4**	**0.1**	**0.2**	**0.15**	**0.15**
Codeword	**0**	**100**	**111**	**101**	**110**

Encode the string 'ABACABAD': The text 'ABACABAD' can be encoded by concatenating the codeword of each character in the text as shown below:

A	B	A	C	A	B	A	D
0	100	0	111	0	100	0	101

and encoded string is

0100011101000101

which requires only 16-bits compared to the 64 bits of original text.

Decode the string '100010111001010': The string 100010111001010 can be decoded as shown in the table below:

String to be decoded	Extract code word	Equivalent character
100010111001010	**100**	B
010111001010	**0**	A

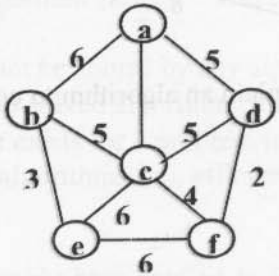
10111001010	101	D
11001010	110	-
01010	0	A
1010	101	D
0	0	A

Thus, the decoded string is given by "BAD-ADA"

Exercises:

1. Write the Kruskal's algorithm to find minimum spanning tree in time O(|E| log |E|), where, |E| is the number of edges. Prove the correctness and derive the efficiency class of the algorithm (July/Aug 2004)

   Apply Kruskal's algorithm to find minimum spanning tree of the graph shown below: (July/Aug 2004)



3. State and explain Dijkstra's algorithm to find single source shortest paths (Jan/Feb 2005)

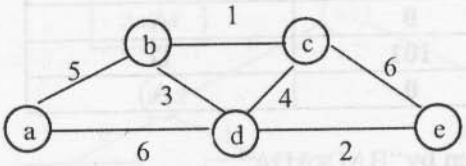
4. What is a Huffman tree? Explain an algorithm to construct the Huffman tree (Jan/Feb 2005)

5. Write Kruskal's algorithm to find minimum cost spanning tree. Compare Prim's and kruskal's method with respect to time and space

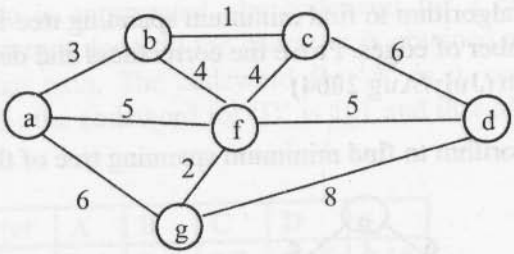
6. Write Dijkstra's algorithm to solve single source shortest path problem. What changes must be done to find the shortest path between w given vertices of a weighted graph.

7. Write Kruskal's algorithm to find a minimum spanning tree in time O(|E| log |E|) and prove the correctness of the algorithm

8. Apply Kruskal's algorithm to find a minimum spanning tree of the graph shown in figure below:



9. Solve the following graph for its minimum spanning tree using Prims's algorithm and Kruskal's algorithm



10. What is a Huffman tree? State and explain an algorithm to construct a Huffman tree.

Chapter 10: Limitations of Algorithm Power

What are we studying in this chapter?

- Lower-Bound Arguments
- Decision Trees
- P, NP and NP-Complete Problems

- 3 hours

10.1 Introduction

Algorithms are very powerful problem solving tools when they are executed by computers. Even then there are some limitations of algorithm power. Let us see "What are the limitations of algorithm power?" The power of algorithms is limited because of the following reasons:

- Some problems can not be solved by any algorithm.
- Some problems can be solved algorithmically but not in polynomial time.
- Even though solution exists for a problem in polynomial time, there are lower bounds on the efficiency of algorithms i.e., efficiency of the algorithm can not be improved further.

Majority of the algorithms we have studied so far are *polynomial-time algorithms* i.e., for all inputs of size *n*, the worst case time complexity is $O(n^k)$ for some constant k. For example, all sorting and searching techniques, BFS and DFS traversal etc.

There are problems that can be solved, but not in time $O(n^k)$ for any constant k. For example, the algorithms such as Traveling sales man problem whose time complexity is O(n-1!), tower of Hanoi whose time complexity is $\theta(2^n)$ can not be solved in *polynomial-time*.

10.2 Lower bound arguments

The efficiency of an algorithm can be looked into using two different ways:

- By establishing the asymptotic efficiency class i.e., obtaining the upper bound which is expressed using Big-Oh notation and compare the upper bound with other class of algorithms. For example, compare bubble sort algorithm whose time efficiency is $O(N^2)$ and tower of Hanoi problem whose time efficiency is $O(2^n)$. Comparing these two time efficiencies we know that bubble sort algorithm is efficient when compared to towre of Hanoi problem. This comparison has to be avoided because these two

problems solve different problems. Note: It is as good as comparing mango with pineapple and say which is best.

- By establishing the efficiency class of solving the same problem with different methods. For example, time efficiency of bubble sort is $O(N^2)$ whereas the time efficiency of quick sort is N log N. By comparing these two class of efficiencies we know that quick sort works faster.

To ascertain the efficiency of an algorithm with respect to other algorithms for the same problem, we should know the best efficiency or the lower bound. Now, let us see "What is lower bound? What are the different methods of obtaining the lower bound?"

Definition: Lower bound is defined as estimating the minimum amount of time needed to solve a given problem. If we have an algorithm with the lower bound, we can look for other improvements in the algorithm to improve the efficiency further. The various methods of obtaining the lower bound are shown below:

Methods to obtain lower bounds
- → Trivial lower bounds
- → Information-Theoretic arguments
- → Adversary arguments
- → Problem reduction

Trivial lower bounds: This is the simplest method of obtaining the lower bound. This is normally obtained by looking into input and output parameters i.e.

- The number of items in the input that must be processed
- The number of outputs that have to be produced

For example, while generating the permutations, the input size is n and the number of distinct outputs will be n!. So, the time efficiency is $\Omega(n!)$. Let us see "What are limitations of trivial bound approach?" The various limitations using trivial bound approach are shown below:

- Trivial lower bounds are often not useful. For example, in a traveling salesman problem, given n cities and n(n-1)/2 distances, the output will be a list of n+1 cities, with time efficiency of $\Omega(n^2)$. There is no other algorithm to compare with this time efficiency.
- Another problem is which part of the input must be processed. For example, in binary search problem, the elements are arranged in ascending order. We have seen that there is no need to search for the entire using this technique.

Information-Theoretic arguments: In this approach, the lower bound is based on the amount of information it has to produce. This method is very useful to find the lower bounds for many problems involving searching, sorting etc. The lower bound using this method can be obtained very easily and precisely using decision trees.

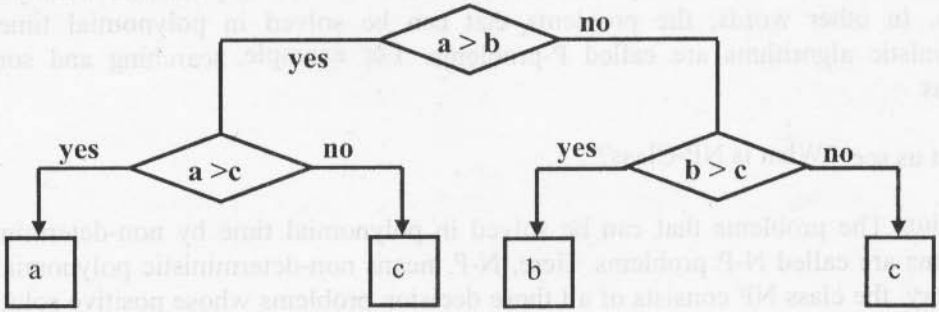
Adversary argument: The adversary method is another way of obtaining the lower bound. In this method, the lower bound is obtained by measuring the amount of work required to reduce a set of potential inputs to a single-input along the most time-consuming path. For example, in the game of guessing a number, the number of questions asked by the adversary must be at least $\log_2 n$.

Problem reduction: In this strategy, we already know that a given problem is transformed into another problem for which the solution exists in the form of a known algorithm. The same idea can be used to find the lower bound also. Suppose, there is a problem P whose lower bound has to be computed. Assume, we have another problem Q whose lower bound is already known. If an instance of a problem P can be transformed into an instance of Q for which solution exists, then lower bound of Q is the lower bound of P. In this way, the lower bound of one instance of a problem can be obtained using the lower bound of another instance of a problem.

10.3 Decision Trees

The various algorithms such as sorting and searching work only by comparing the given elements. The performance of these algorithms that involve comparison can be obtained using decision trees. Now, let us see "What are decision trees?"

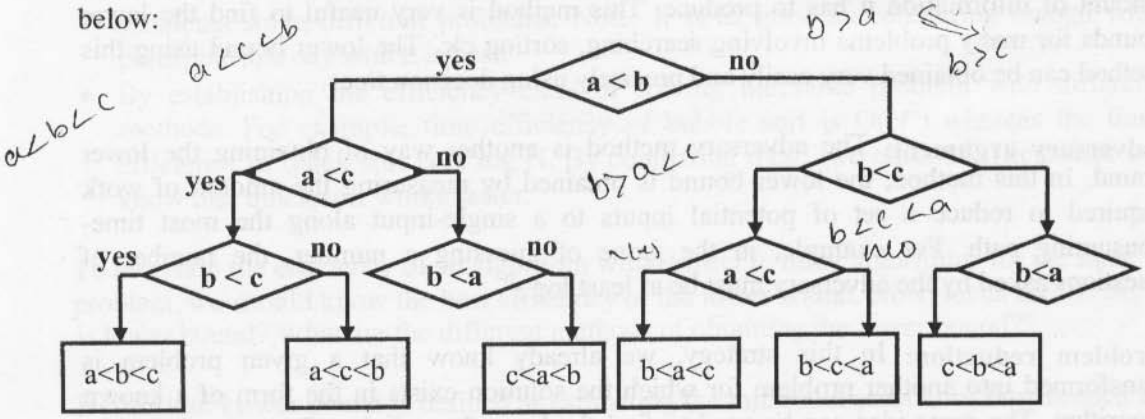
Definition: A decision tree is a binary tree that represents only the comparisons of given elements in the array while sorting or while searching. In the algorithm, the control transfer, data movement and all other aspects of the algorithm are ignored. In the decision tree, internal nodes represent the comparisons made between the items and leaf nodes represent the result of the algorithm. For example, the decision tree to find minimum of three numbers is shown below:



(bac) b<c a c<b

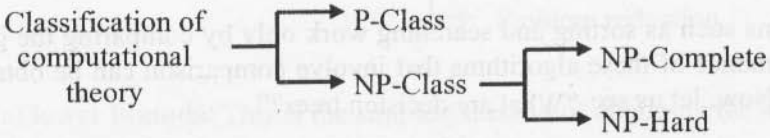
10.4 Limitations of algorithm power

The decision tree for arranging three elements a, b and c in ascending order is shown below:



10.4 P, NP and NP-Complete Problems

The computational complexity theory deals with the resources required during computation to solve a given problem. The most common resources are time taken to solve a problem and space or memory requuired to solve a problem. The computational theory is classfied into two groups as shown below:



Now, let us see "What is P class?"

Definition: In this theory, the class P problem consists of decision problems that can be solved in polynomial time by deterministic algorithms. This class of problems is called *polynomial* (P). P includes only decision problems which are problems with yes/no answers. In other words, the problems that can be solved in polynomial time by deteriministic algorithms are called P-problems. For example, searching and sorting problems

Now, let us see "What is NP-Class?"

Definition: The problems that can be solved in polynomial time by non-deterministic algorithms are called N-P problems. Here, N-P means non-deterministic polynomial. In this theory, the class NP consists of all those decision problems whose positive solutions

can be verified in polynomial time given the right information. The solution should be obtained in a polynomial time on a non-determinsitic machine. The NP-Class problems are divided as NP-Complete and NP-Hard

Now, let us see "What are NP-Complete problems?"

Definition: The NP-problems (non-deterministic polynomial time) are NP-Complete if their solutions can be verified quickly for correctness and if the same solving alogirthm used can solve all other NP problems. For example, sum of subset problem.

Now, let us see "What are NP-hard problem?

Definition: NP-hard is a class of problems that are at least as hards as the hardest problem in NP. A problem H is said to be NP-hard if there is an NP-complete problem that can be solved in a polynomial time. For example, decision problems, search problems and optimiazation problems.

Chapter 11: Coping with the Limitations of Algorithm Power

What are we studying in this chapter?

- **Backtracking**
- **Branch and bound**
- **Approximate algorithms for NP-Hard**

11.1 Backtracking

Now, let us see "What is backtracking?"

Definition: Backtracking is the process of solving the problem by returning to a previous state in a computation on order to try an alternative strategy. It is modification of brute-force search method. In backtracking, multiple solutions can be eliminated without being explicitly examined. In this method, the solution can be expressed as n-tuple (x_1, x_2, x_3,.....x_n). Problems such as n-quuens, sum of subset etc can be solved using back tracking.

11.2 N-Queens problem

Now, let us see "What is N-QEENS problem?" **Given NxN chess board and N-queens, it is required to place all N-queens on the chess board such that no two queens attach. The function to check whether kth queen can be placed successfully or not is shown below:**

```
ALGORITHM place(x[], k)
    for i = 1 to k
        // Check whether two queens attach vertically or diagonally
        if ( x[i] == x[k] || i-x[i]==k-x[k] || i + x[i] == k + x[k])
            return FALSE;        // Queen can not be placed in the kth column
        end if
    end for

    // kth queen can be successfully placed
    return TRUE;
// end of algorithm
```

11. 2 Coping with limitations of algorithm power

The algorithm to place all N-queens on the chess board is shown below:

```
ALGORITHM nqueens(n)
k = 1;                  // Select the first queen
x[k] = 0;               // But, not placed on the chess board

while ( k != 0 )        // A queen exists?
    x[k] += 1;          // Place the kth queen in next column

    // Satisfy both explicit and implicit constraint
    while ( (x[k] <= n) && ( !place(x,k) ) )
        x[k] += 1;      // Place queen in next column
    end while

    // if queen successully placed?
    if (x[k] <= n )
        if ( k == n )                   // If all queens are placed
            print(x1, x2, …xn)
        else
            k++;                        // select the next queen
            x[k] = 0;                   // but do not place
        end if
    else
        k--;                            //Backtrack and select previous queen

    end if
end while

// No more solution exists
return;
```

For example, 4 queens can be placed on the chess board as shown below:

	Q		
			Q
Q			
		Q	

11.3 Sum of subset problem

Example: Find a subset of a given set S = {sl,s2,....s_n} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1,2,6} and {l,8}. A suitable message is to be displayed if the given problem instance doesn't have a solution. (C Program)

```
#include<stdio.h>

int count;                          /* Initial number of solutions */
int w[10];                          /* Holds the elements */
int d;                              /* The total sum required */
int x[10];                          /* Holds the items whose sum is equal to d  */


void subset(int cs,int k,int r)
{                                   /* r Required remaining sum */
        int i;
        x[k]=1;                     /* include kth element */

        if(cs+w[k] == d)            /* If current sum+kth element = d */
        {
                printf("\nSubset solution = %d\n", ++count);
                for(i=0;i<=k;i++)
                {
                        if(x[i] == 1) printf("%d ",w[i]);
                }
        }
        else  if(cs+w[k]+w[k+1] <= d)
                        subset(cs+w[k],k+1,r-w[k]);

        if((cs+r-w[k]>=d) && (cs+w[k+1])<=d)
        {
                x[k]=0;
                subset(cs,k+1,r-w[k]);
        }
}

void main()
{
        int sum=0,i, n;

        printf("Enter no of elements\n");
        scanf("%d",&n);
```

```
        printf("Entre the elements in ascending order\n");
        for(i=0;i<n;i++) scanf("%d",&w[i]);

        printf("Enter the required sum\n");
        scanf("%d",&d);
        for(i=0;i<n;i++) sum += w[i];
        if(sum < d)
        {
                printf("No solution exists\n");
                return;
        }
        printf("The solution is\n");
        count = 0;
        subset(0,0,sum);
}
```

11.4 Branch and bound

Now, let us see "What is branch and bound?"

Definition: Branch and bound is a general algorithm which can be used to find the optimal solutions. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded by using upper and lower estimated bounds of the quantity being optimized.

For example, traveling salesman problem can be solved using this method.

11.5 Approximation algorithms for NP-Hard Problems

Note: The reader is adviced to solve the knapsack problem, assignemnt problem and traveling sales man problem using brach and bound technique.

BIBILIOGRAPHY

1. Introduction to Data Structures and applications Tremblay and Sorenson
2. Data Structures and Algorithms in C++ Sahni
3. Data Structures and Algorithms in C++ Drozdek
4. Data Structures and Algorithms in C/C++ Tanenbaum, Langsam, Augenstein
5. Introduction to Algorithms Coremen, Leiserson, Rivest
6. Introduction to The Design and Analysis of Algorithms Levitin
7. Systematic approach to Data Structures using C by Padma Reddy A.M
8. Data Structures and Algorithm Analysis in C Weiss
9. Data Structures and Program Design in C Kruse
10. Design and Analysis of Computer Algorithms Aho/Hopcroft/Ulman
11. Data Structures and algorithms with Object oriented design patterns in C++ by Preiss

APPENDIX

Program No. 1.a: Perform recursive Binary search. Hence find the time required to search an element.

```
# include <iostream.h>
# include <conio.h>
# include <time.h>
# include <stdlib.h>

int binary_search(int key,int a[],int low,int high)
{
        if ( low > high )    return -1;                    // Unsuccessful search

        int mid = ( low + high ) /2;                       // Find the mid point

        if ( key == a[mid] ) return mid;  // Search is successful

        if ( key < a[mid] )
                return binary_search ( key,a,low,mid-1 );          // Search the left part of table

        return binary_search ( key,a,mid+1,high );                 // Search the right part of table
}

void main()
{
        int a[1000],n,i,key,pos;
        clock_t st,end;

        st = clock();

        cout << "Enter the number of elements\n";
        cin >> n;

        cout << "Enter the elements in ascending order " << endl;
        for(i = 0; i < n; i++ )
        {
                cin >> a[i];
        }

        cout << "Enter the element to be searched\n";
        cin >> key;

        pos = binary_search(key,a,0,n-1);
```

```
        end = clock();

        if ( pos == -1 )
                cout << "Key not found\n";
        else
                cout << "Key is found at" << pos+1 << "position\n";

        cout << "Time taken by binary search is" << ( end - st ) / CLK_TCK << endl;
}
```

Output 1

```
Enter the number of elements
5
Enter the elements in ascending order
10
20
30
40
50
Enter the element to be searched
50
Key is found at5position
Time taken by binary search is12
```

Output 2

```
Enter the number of elements
5
Enter the elements in ascending order
10
20
30
40
50
Enter the element to be searched
5
Key not found
Time taken by binary search is7
```

Program No. 1.b: Perform recursive Linear search. Hence find the time required to search an element.

```
#include <iostream.h>
#include <time.h>

int linear_search( int a[],int key,int n )
{
        if ( n < 0 )            return -1;              // Unsuccessful search

        if ( key == a[n] )      return n;               // Successful search

        return  linear_search(a,key,n-1);               // Search the remaining table
}

void main()
{
        int a[1000],n,key,pos;
        clock_t st,end;

        cout << "Enter the number of elements\n";
        cin >> n;

        cout << "Enter the elements " << endl;
        for( int i = 0; i < n; i++ )
        {
                cin >> a[i];
        }

        cout << "Enter the element to be searched\n";
        cin >> key;

        st = clock();

        pos = linear_search(a,key,n);

        end = clock();

        if ( pos == -1 )
                cout << "Key not found\n";
        else
                cout << "Key is found at " << pos+1 << "position\n";

        cout << "Time taken by linear search is " << ( end - st ) / CLK_TCK << endl;
}
```

Output 1

Enter the number of elements
5
Enter the elements
10
20
30
40
50
Enter the element to be searched
40
Key is found at 4 position
Time taken by linear search is 0

Output 2

Enter the number of elements
5
Enter the elements
10
20
30
40
50
Enter the element to be searched
60
Key not found
Time taken by linear search is 0

Program No. 2: Sort a given set of elements using the Heap sort method.

```
# include<iostream.h>
# include<conio.h>

class heap
{
        private:
                int n;
                int a[20];
        public:
                void get_data();                    // To read the data to sort
                void write_data();                  // Write the sorted data
                void heapify();                     // Create a heap
                void adjust(int n, int a[], int p); // Recreate the heap
                void heapsort();                    // Sort the given elements
};

void heap::get_data()
{
        cout << "Enter the number of elements" << endl;
        cin  >> n;

        cout << "Enter the elements to sort" << endl;
        for (int i = 0; i < n; i++) cin >> a[i];
}

void heap::write_data()
{
        cout << "The sorted vector is " << endl;
        for (int i = 0; i < n; i++) cout << a[i] << endl;
}

void heap::heapify()
{
        int     key;        // Item to be inserted
        int     k;          // Initial position of the hole i.e, next to last leaf in a heap
        int     c;          // Position of the hole
        int     p;          // Position of the parent of the hole

        for ( k = 1; k < n; k++)
        {
                key = a[k];                 // item to be inserted
                c = k;                      // at the child's position
                p = (c-1) / 2;              // obtain the position of the parent
```

```
            while ( c > 0 && key > a[p] )   // while parent exists and item >parent
            {
                    a[c] = a[p];            // Move parent down to child
                    c = p;                  // Make parent as child
                    p = (c-1) / 2;          // Obtain new parent for child
            }

            a[c] = key;                     // Insert item into child's position
    }
}

void heap::adjust(int n, int a[], int p)
{
    int item;
    int c;

    item = a[p];    // p is the position of the next last-node & item is the parent node
    c = 2*p+1;      // obtain the position of left child

    while(c < n)                            // As long as left child exists do
    {
            if(c + 1 < n )                  // If right child also exists?
                    if ( a[c] < a[c+1] ) c++; // obtain position of largest child

            if(item < a[c] )                //if item (parent) is less than the largest child
            {
                    a[p] = a[c];            //move the largest child to parent position
                    p = c;                  //Move the parent to child position
                    c = 2*p+1;              //and obtain the position of new left child
            }
            else
                    break;
    }
    a[p] = item;            // The resulting tree is heap. So, insert at parent position
                            // and the algorithm is terminated
}


void heap::heapsort()
{
    int i, temp;

    heapify();
```

```
        for(i = n-1; i > 0; i--)
        {
                temp = a[0];            // exchange the root item with the last leaf
                a[0] = a[i];
                a[i] = temp;
                adjust(i, a, 0);        // Recreate the heap starting from root node
        }
}
void main()
{
        heap a;
        a.get_data();
        a.heapsort();
        a.write_data();
}
```

Output

```
Enter the number of elements 12
Enter the elements to sort
5
35
25
45
20
55
25
45
50
10
30
15
The sorted vector is
5
10
15
20
25
25
30
35
45
45
50
55
```

Program No. 3.a: Sort a given set of elements using the Merge sort method.

```
#include <iostream.h>
#include <conio.h>

const int MAX = 1000;

void simple_merge(int a[], int low, int mid, int high)
{
        int i = low;            // Points to first element of left part of the array
        int j = mid+1;          // Points to first element of second part of the array
        int k = low;            // Points to first element of resulting array
        int c[MAX];             // Temporarily holds the sorted elements

        while ( i <= mid && j <= high )
        {
                if ( a[i] < a[j] )
                {
                        c[k] = a[i];    // Copy the lowest element from first part of A to C
                        i = i + 1;      // Point to next item in the left part of A
                        k = k + 1;      // Point to next item in C
                }
                else
                {
                        c[k] = a[j];    // Copy the lowest element from second part of A to C
                        j = j + 1;      // Point to next item in the second part of A
                        k = k + 1;      // Point to next item in C
                }
        }

        while ( i <= mid )              // Copy the remaining items from left part of A to C
        {
                c[k++] = a[i++];
        }

        while ( j <= high )             // Copy the remaining items from right part of A to C
        {
                c[k++] = a[j++];
        }

        // Copy the elements from vector C to vector A
        for (i = low; i <= high; i++)
        {
                a[i] = c[i];
        }
}
```

```
void merge_sort(int a[], int low, int high)
{
        int mid;

        if ( low < high)
        {
                mid = (low + high)/2;                   // Divide the array into equal parts

                merge_sort (a, low, mid);               // Sort the left part of array

                merge_sort (a, mid + 1, high);   // Sort the right part of the array

                simple_merge(a, low, mid, high);// Merge the left part and right part
        }
}

void main()
{
        int  a[MAX];    // Elements to be sorted
        int  n;                 // Number of elements to sort
        int  i;                 // Used as an index to access the elements

        cout << "Enter the number of elements to sort" << endl;
        cin  >> n;

        cout << "Enter the elements to sort" << endl;
        for ( i = 0; i < n; i++)
                cin >> a[i];

        merge_sort(a, 0 , n-1);

        cout << "The sorted vector is" << endl;
        for ( i = 0; i < n; i++)
                cout << a[i] << endl;
}
```

Output

Enter the number of elements 12
Enter the elements to sort 5 35 25 45 20 55 25 45 50 10 30 15
The sorted vector is 5 10 15 20 25 25 30 35 45 45 50 55

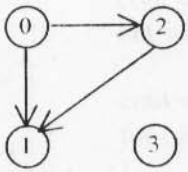
Program No. 3.b: Check whether a given graph is connected or not using DFS method.

```
#include <iostream.h>

class GRAPH
{
        private:
                int     n;                      // Number of vertices in the graph
                int     cost[10][10];           // Adjacency matrix
        public:
                void    read_matrix();          // Reads the adjacency matrix
                void    dfs(int u, int s[]);    // Traverse the graph in DFS
                void    reachability();         // Finds the nodes reachable from source
                int     connectivity();         // Check for the connectivity of the graph
};
void GRAPH :: read_matrix()
{
        int     i, j;

        cout << "Enter the no. of nodes" << endl;
        cin >> n;

        cout << "Enter the cost adjacency matrix" << endl;
        for ( i = 0; i < n; i++)
        {
                for ( j = 0; j < n; j++)
                {
                        cin >> cost[i][j];
                }
        }
}
void GRAPH ::dfs(int u, int s[])
{
    int v;

    s[u] = 1;

    for ( v = 0; v < n; v++)
    {
        if ( cost[u][v] == 1 && s[v] == 0 )
        {
            dfs(v, s);
        }
    }
}
```

```
void GRAPH :: reachability()
{
        int     i, source, s[10];

        for( i = 0; i < n; i++) s[i] = 0;

        cout << "Enter the source" << endl;
        cin >> source;

        dfs(source, s);

        for ( i = 0; i < n; i++)
        {
                if ( s[i] == 0)
                        cout << "Node " << i << "is not reachable" << endl;
                else
                        cout << "Node " << i << "is reachable" << endl;
        }
}

int GRAPH :: connectivity()
{
        int     i, j, flag, s[10];

        for (j = 0; j < n; j++)
        {
                for ( i = 0; i < n; i++)  s[i] = 0;

                dfs(j, s);

                flag = 0;

                for ( i = 0; i < n; i++)
                {
                        if ( s[i] == 0 ) flag = 1;
                }

                if (flag ==0) return 1;
        }

        return 0;
}
```

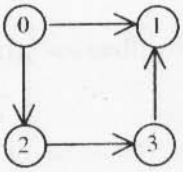
```
void main()
{
        GRAPH g;

        g.read_matrix();            // Input the cost adjacency matrix

        g.reachability();           // Find the nodes that are reachable and that are not reachable

        int flag = g.connectivity();

        if (flag == 1)
                cout << "The graph is connected" << endl;
        else
                cout << "The graph is not connected" << endl;
}
```

Output 1




```
Enter the no. of nodes
4
Enter the cost adjacency matrix
0 0 1 0
1 0 1 0
0 0 0 0
0 0 0 0
The graph is not connected
```

Output 2




```
Enter the no. of nodes
4
Enter the cost adjacency matrix
0 1 1 0
0 0 0 0
0 0 0 1
0 1 0 0
The graph is connected
```

Program No. 4: Sort a given set of elements using Selection sort and hence find the time required to sort elements

```
# include<iostream.h>
# include<time.h>


class SORT
{
   private:
       int n;                  // Number of elements to sort
       int a[20];              // Holds the elements to sort and sorted vector

   public:
       void read_data();       // Reads the number of elements and elements to sort
       void write_data();      // Displays the elements
       void sort_data();       // Sort using selection sort
};

void SORT::read_data()
{
   cout << "Enter the number of elements\n";
   cin >> n;

   cout << "Enter the elements to sort " << endl;
   for ( int i = 0; i < n; i++ )
   {
       cin >> a[i];
   }
}

void SORT::write_data()
{
   cout << "The sorted array is\n";

   for ( int i = 0; i < n; i++ )
       cout << a[i] << endl;
}
```

```
void SORT:: sort_data ()
{
        for ( int i = 0; i < n; i++ )
        {
                int pos = i;

                for ( int j = i+1; j < n; j++ )         // Find the minimum position in each pass
                {
                        if ( a[j] < a[pos] )
                        {
                                pos = j;
                        }
                }

                int temp = a[pos];                      // Exchange a[pos] with a[i]
                a[pos] = a[i];
                a[i] = temp;
        }
}

void main()
{
        SORT a;
        clock_t st.end;


        st = clock();

        a.read_data();

        a.sort_data();

        a.write_data();

        end = clock();

        cout << "The time taken by selection sort is " << (end - st ) / CLK_TCK << endl;
}
```

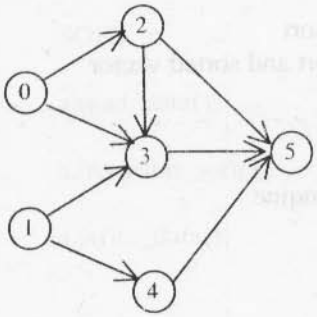
Program No. 5.a: Obtain the Topological ordering of vertices in a given digraph.

```
# include <iostream.h>
# include <conio.h>

class topo
{
        private:
                int n;                          // Number of nodes in the graph
                int a[10][10];                  // Cost adjacency matrix
                int indegre[10];                // Indegree of all nodes
        public:
                void read_data();               // Reads the cost adjacency matrix
                void find_indegre();            // find indegree of each node
                void topological_sort ();       // Obtain the topological ordering
};
// To read the adjacency matrix
void topo :: read_data()
{
        cout<<"Enter the number of jobs\n";
        cin>>n;

        cout<<"Enter the adjacency matrix";
        for (int i = 0; i < n; i++)
        {
                for (int j = 0; j < n; j++)
                {
                        cin>>a[i][j];
                }
        }
}
//find the indegree of all vertices
void topo :: find_indegre ()
{
        for (int j = 0; j < n; j++)
        {
                int sum = 0;

                for (int i = 0; i < n; i++)  // add the column values
                {
                        sum += a[i][j];
                }
                indegre[j] = sum;
        }
}
```

```
// function to find solution (Topological sort)
void topo :: topological_sort ()
{
        int    u, v, t[10], s[10];

        find_indegre ();

        int top = -1;
        int k = 0;

        for (int i = 0; i < n; i++)
        {
                if (indegre[i] == 0)
                {
                        s [++top] = i;
                }
        }

        while (top != -1)
        {
                u = s[top--];
                t[k++] = u;

                for (v = 0; v < n; v++)
                {
                        if (a[u][v] == 1)
                        {
                                indegre[v]--;
                                if (indegre[v] == 0)
                                {
                                        s[++top] = v;
                                }
                        }
                }
        }

        cout<<"The topological sequence is:\n";
        for (i = 0; i < n; i++)
                cout<<t[i]<<" ";
}
```

```
void main ()
{
    topo t;

    t.read_data();

    t.topological_sort();
}
```

Output: If the various tasks to be performed are represented using the graph as shown below:



Number of vertices = 6

Adjacency matrix of the graph is shown below:

	0	1	2	3	4	5
0	0	0	1	1	0	0
1	0	0	0	1	1	0
2	0	0	0	1	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	0

```
Enter the number of jobs
6
Enter the adjacency matrix
0 0 1 1 0 0
0 0 0 1 1 0
0 0 0 1 0 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 0
The topological sequence
1 4 0 2 3 5
```

Program No. 5.b: Sort a given set of elements using Insertion sort method.

```
#include <iostream.h>
#include <conio.h>

class SORT
{
        private:
                int n;                  // Number of elements to sort
                int a[20];              // Holds the elements to sort and sorted vector
        public:
                void read_data();       // Input the data to sor
                void write_data();      // Display the elements
                void insertion_sort();  // Sort using insertion technqiue
};

void SORT::insertion_sort()
{
        for ( int i = 0; i < n; i++ )
        {
                int item = a[i];
                int j = i-1;

                while ( j >= 0 && item <= a[j] )
                {
                        a[j+1] = a[j];
                        j--;
                }
                a[j+1] = item;
        }
}

void SORT::read_data()
{
        cout << "Enter the number of elements\n";
        cin >> n;

        cout << "Enter the elements\n";
        for ( int i = 0; i < n; i++ )
                cin >> a[i];
}
```

```
void SORT::write_data()
{
        cout << "The sorted array is \n";

        for ( int i = 0; i < n; i++ )
                cout << a[i] << endl;
}

void main()
{
        SORT a;

        a.read_data();

        a.insertion_sort();

        a.write_data();
}
```

Program No. 6: Implement 0/I Knapsack problem using dynamic programming.

```
#include<iostream.h>
#include<conio.h>

class KNAPSACK
{
        private:
                int n;                          // Number of objects
                int m;                          // Capacity of the knapsack
                int v[10][10];    // Optimal solution in the form of table
                int p[10];                      // Profits obtained
                int w[10];                      // Weights of all the objects
        public:
                void read_data();                               // Read the appropriate data
                void optimal_solution();// Find the optimal solution
                void write_data();                              // Print the optimal solution
};


void KNAPSACK::read_data()
{
        cout <<"Enter the number of objects " << endl;
        cin >> n;

        cout << "Enter the weights of the objects " << endl;
        for ( int i = 1; i <= n; i++ )
                cin >> w[i];

        cout << "Enter the profits of the objects " << endl;
        for ( i = 1; i <= n; i++ )
                cin >> p[i];

        cout << "Enter the capacity of the knapsack " << endl;
        cin >> m;
}
```

```
void KNAPSACK::write_data()
{
        int i, j,x[10];

        cout << "the output is " << endl;
        for (i = 0; i <= n; i++)
        {
                for (j = 0; j <= m; j++)   cout << v[i][j] << " ";

                cout << endl;
        }

        cout << "The optimal solution is " << v[n][m] << endl;

        // Initially, no object has been selected
        for(i = 0; i < n; i++) x[i] = 0;

        i = n;          // Number of objects
        j = m;          // Capacity of the knapsack

        while (i != 0 && j!= 0)
        {
                if (v[i][j] != v[i-1][j])
                {
                        x[i] = 1;               //ith object has been selected
                        j = j - w[i];           //Obtain the remaining capacity of the knapsack
                }

                i = i - 1;                      // Select the object among i-1 objects
        }

        // Output the objects selected
        for (i = 1; i <= n; i++)
        {
                cout << "X[" << i << "] ";
        }

        cout << "=";

        for (i = 1; i <= n; i++)
        {
                cout << "  " << x[i] << " ";
        }
}
```

```
int max(int a,int b)
{
        return a > b ? a : b;
}

void KNAPSACK::optimal_solution()
{
        for ( int i = 0; i <= n; i++ )
        {
                for ( int j = 0; j <= m; j++ )
                {
                        if ( i== 0 || j == 0 )
                                v[i][j] = 0;
                        else if ( j < w[i] )
                                v[i][j] = v[i-1][j];
                        else
                                v[i][j] = max( v[i-1][j], v[i-1][j-w[i]] + p[i]);
                }
        }
}

void main()
{
        KNAPSACK a;

        a.read_data();

        a.optimal_solution();

        a.write_data();
}
```

Output 1

```
Enter the number of objects
4
Enter the weights of n objects
2
1
3
2
Enter the profits of n objects
12
10
20
15
```

Enter the capacity of knapsack
5
the output is
0 0 0 0 0 0
0 0 12 12 12 12
0 10 12 22 22 22
0 10 12 22 30 32
0 10 15 25 30 37
The optimal solution is 37
X[1] X[2] X[3] X[4] =1 1 0 1

Output 2

Enter the number of objects
5
Enter the weights of n objects
3
2
1
4
5
Enter the profits of n objects
25
20
15
40
50
Enter the capacity of knapsack
6
the output is
0 0 0 0 0 0 0
0 0 0 25 25 25 25
0 0 20 25 25 45 45
0 15 20 35 40 45 60
0 15 20 35 40 55 60
0 15 20 35 40 55 65
The optimal solution is 65
X[1] X[2] X[3] X[4] X[5] =0 0 1 0 1

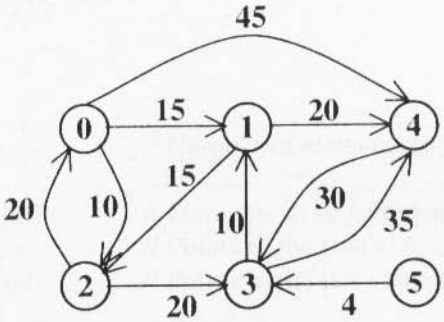
Program No. 7: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

```
#include <iostream.h>
#include <process.h>
#include <conio.h>

class DIJKSTRA
{
        private:
                int n;              / Number of nodes in the graph
                int a[10][10];      // Cost adjacency matrix
                int d[10];          // Holds the shortest distance
                int p[10];          // Holds the shortest path from a source to destination
        public:
                void read_data();
                void print_path();
                void optimal_solution(int source, int dest);
};

void DIJKSTRA::optimal_solution(int source, int dest)
{
    int i,j,u,v,min;
    int s[10];                      // Holds the information of reachablity

    for (i=0; i < n; i++)
    {
        d[i] = a[source][i];
        s[i] = 0;
        p[i] = source;
    }

    s[source] = 1;          /* Add source to S */

    for(i = 1; i < n; i++)
    {
        /* fund u and d[u] such that d[u] is minimum
          and u in V-S
         */
        min = 9999;
        u = -1;
```

```
        for (j=0; j < n; j++)
        {
            if ( s[j] == 0 )
            {
                if ( d[j] < min )
                {
                    min = d[j];
                    u = j;
                }
            }
        }

        if (u == -1) return;

        /* add u to s */
        s[u] = 1;

        if ( u == dest) return;

        /* find d[v] and p[v] for every v in V-S */
        for ( v = 0; v < n; v++ )
        {
            if ( s[v] == 0 )
            {
                if ( d[u] + a[u][v] < d[v] )
                {
                    d[v] = d[u] + a[u][v];
                    p[v] = u;
                }
            }
        }
    }
}

void DIJKSTRA::read_data()
{
    int i,j;

    cout <<"Enter the no. of nodes " << endl;
    cin >> n;
```

```
        cout << "Enter the cost adjacency matrix " << endl;
        for(i = 0; i < n ; i++)
        {
                for (j = 0; j < n; j++)
                {
                        cin >> a[i][j];
                }
        }
}
void DIJKSTRA::print_path()
{
        for (int source = 0; source < n; source++)
        {
                for (int destination = 0; destination < n; destination++)
                {
                        optimal_solution(source, destination);

                        if (d[destination] == 9999)
                                cout << destination << " is not reachable from " << source << endl;
                        else
                        {
                                cout << "The shortest path is shown below" << endl;

                                int i = destination;
                                while (i != source)
                                {
                                        cout << i << " <--";
                                        i = p[i];
                                }

                                cout << i << " = " << d[destination] << endl;
                        }
                }
        }
}
void main()
{
        DIJKSTRA a;

        a.read_data();

        a.print_path();
}
```

Note: For the following graph



Enter the data as shown
Enter the no. of nodes
6
Enter the cost adjacency matrix
0 15 10 9999 45 9999
9999 0 15 9999 20 9999
20 9999 0 20 9999 9999
9999 10 9999 0 35 9999
9999 9999 9999 30 0 9999
9999 9999 9999 4 9999 0
The shortest path is shown below
0 = 0
The shortest path is shown below
1 <- 0 = 15
The shortest path is shown below
2 <- 0 = 10
The shortest path is shown below
3 <- 2 <- 0 = 30
The shortest path is shown below
4 <- 1 <- 0 = 35
5 is not reachable from 0
The shortest path is shown below
0 <- 2 <- 1 = 35
The shortest path is shown below
1 = 0
The shortest path is shown below
2 <- 1 = 15
The shortest path is shown below
3 <- 2 <- 1 = 35
The shortest path is shown below
4 <- 1 = 20
5 is not reachable from 1

The shortest path is shown below
0 <- 2 = 20
The shortest path is shown below
1 <- 3 <- 2 = 30
The shortest path is shown below
2 = 0
The shortest path is shown below
3 <- 2 = 20
The shortest path is shown below
4 <- 1 <- 3 <- 2 = 50
5 is not reachable from 2
The shortest path is shown below
0 <- 2 <- 1 <- 3 = 45
The shortest path is shown below
1 <- 3 = 10
The shortest path is shown below
2 <- 1 <- 3 = 25
The shortest path is shown below
3 = 0
The shortest path is shown below
4 <- 1 <- 3 = 30
5 is not reachable from 3
The shortest path is shown below
0 <- 2 <- 1 <- 3 <- 4 = 75
The shortest path is shown below
1 <- 3 <- 4 = 40
The shortest path is shown below
2 <- 1 <- 3 <- 4 = 55
The shortest path is shown below
3 <- 4 = 30
The shortest path is shown below
4 = 0
5 is not reachable from 4
The shortest path is shown below
0 <- 2 <- 1 <- 3 <- 5 = 49
The shortest path is shown below
1 <- 3 <- 5 = 14
The shortest path is shown below
2 <- 1 <- 3 <- 5 = 29
The shortest path is shown below
3 <- 5 = 4
The shortest path is shown below
4 <- 1 <- 3 <- 5 = 34
The shortest path is shown below
5 = 0

Program No. 8: Sort a given set of elements using Quick sort method.

```
#include <iostream.h>

class quick
{
        public:
                int n;                  // Number of elements to sort
        private:
                int a[10];              // Elements to be sorted and the sorted vector
                int low;                // Points to the first element
                int high;               // Points to the last element
        public:
                int  partition (int low, int high);
                void quick_sort (int low, int high);
                void read_data();
                void write_data();
};
int quick :: partition (int low, int high)
{
        int i,j,temp,key;

        key = a[low];           // First element in the table is the pivot element
        i = low;
        j = high+1;

        // Partition the array into two parts
        while(i <= j)
        {
                /* The code is with sentinel. Without sentinel, the code should
                 be changed to "do i++; while (i <= high && key >= a[i]);"*/
                do i++; while (key >= a[i]);

                do j--; while (key < a[j]);

                // Partition not  completed. Exchange a[i], a[j] and repeat the process
                if ( i < j ) temp = a[i], a[i] = a[j], a[j] = temp;
        }

        // Partition completed and return the position of pivot element
        temp = a[low];
        a[low] = a[j];
        a[j] = temp;

        return j;
}
```

```
void quick :: quick_sort (int low, int high)
{
        int     mid;

        if ( low < high)
        {
                mid = partition(low,high);          /* Partition the array into parts */
                quick_sort (low, mid-1);            /* Sort the left part of array */
                quick_sort (mid + 1, high);         /* Sort the right part of the array */
        }
}

void quick:: read_data ()
{
        cout<<"Enter the value for n\n";
        cin>>n;

        cout<<"Enter the numbers to be sorted\n";
        for (int i = 0; i < n; i++) cin >> a[i];

        a[n] = 9999;
}

void quick :: write_data ()
{
        cout<<"The sorted array is\n";
        for(int i = 0; i < n; i++) cout << a[i] << endl;
}

void main()
{
        quick q;

        q.read_data();
        q.quick_sort(0,q.n-1);
        q.write_data();
}
```

```
Enter the value for n
10
Enter the numbers to be sorted
42 37 11 98 36 72 65 10 88 78
The sorted array is 10 11 36 37 42 65 72 78 88 98
```

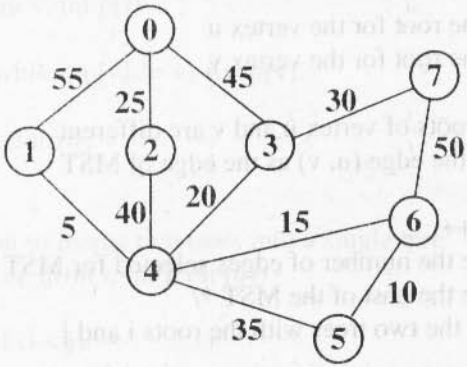
Program No. 9: Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

```
#include <iostream.h>

class KRUSKAL
{
        public:
                int     n;      // Number of nodes in the graph
                int     m;      // Number of edges in the graph
        private:
                int c[10][10];  // The edges along with cost is stored
        public:
                void obtain_cost_adjacency_matrix();
                void minimum_spanning_tree();
};

// Member function to read the cost adjacency matrix
void KRUSKAL::obtain_cost_adjacency_matrix()
{
        int i,j,u,v;

        /* Initialize cost matrix */
        for(i = 0; i < n; i++)
        {
                for(j = 0; j < n; j++)
                {
                        c[i][j]=999;
                }
        }

        /* Enter the specified number of edges */
        for(i = 0; i < m; i++)
        {
                cout << "Enter the next edge(u v)" << endl;
                cin >> u >> v;
                cout << "Enter the cost" << endl;
                cin >> c[u][v];
                c[v][u] = c[u][v];
        }

}
```

```
// function to find the root of a given vertex
int find(int v, int p[])
{
        while ( p[v] != v) v = p[v];

        return v;
}

// function to merge two trees into a single tree
void union_ij(int i, int j, int p[])
{
        if (i < j)
                p[j] = i;
        else
                p[i] = j;
}

void KRUSKAL::minimum_spanning_tree()
{
        int count, i, p[10], min, j, u, v, k, t[10][2], sum;

        count = 0;          /* Initialize the number of edges selected to zero */
        k = 0;              /* Points to the first selected edge of MST */
        sum = 0;            /* Initial cost of Minimum spanning tree */

        for(i = 0;i < n; i++)  p[i] = i;      /* Create forest with n vertices */

        /* Select n-1 edges for minimum spanning tree */
        while(count < n)
        {
                min=999;
                for(i=0;i<n;i++)
                {
                        for(j=0;j<n;j++)
                        {
                                if(c[i][j]<min)
                                {
                                        min=c[i][j];
                                        u=i;
                                        v=j;
                                }
                        }
                }
```

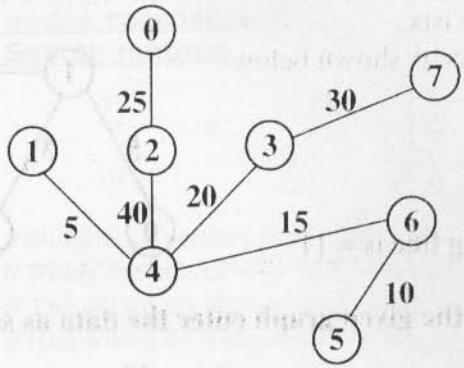
```
            if (min == 999) break;          // No more edges to select

            i = find (u.p);                 // Find the root for the vertex u
            j = find (v.p);                 // Find the root for the vertex v

            if(i != j)                      // If the roots of vertex u and v are different
            {                               // Select the edge (u, v) as the edge of MST

                    t[k][0] = u, t[k][1] = v, k++;
                    count++;        // Update the number of edges selected for MST
                    sum += min;     // Update the cost of the MST */
                    union_ij(i, j, p); // Merge the two trees with the roots i and j

            }

            c[u][v] = c[v][u] = 999; // Delete the edge (u,v) by storing a big value
    }

    if (count == n-1)
    {
            cout << "Cost of spanning tree = " << sum << endl;
            cout <<"Spanning Tree  is shown below" << endl;
            for (k = 0; k < n-1; k++)
                    cout << t[k][0] <<","<< t[k][1] << endl;
            return;
    }

    cout <<"Spanning tree do not exist" << endl;
}

void main()
{
    KRUSKAL k;          // Edge list along with the cost associated with each edge

    cout << "Enter the number of nodes" << endl;
    cin >> k.n;

    cout << "Enter the number of edges" << endl;
    cin >> k.m;

    cout << "Enter the edge list" << endl;
    k.obtain_cost_adjacency_matrix();

    /* Find the MST along with the cost */
    k.minimum_spanning_tree();
}
```

Output 1: For the given graph enter the data as shown below:

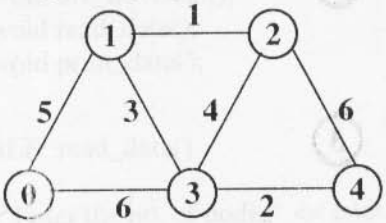


Enter the number of nodes
8
Enter the number of edges
11
Enter the edge list
Enter the edge and cost in the form u v w
1 4 5
Enter the edge and cost in the form u v w
5 6 10
Enter the edge and cost in the form u v w
4 6 15
Enter the edge and cost in the form u v w
3 4 20
Enter the edge and cost in the form u v w
0 2 25
Enter the edge and cost in the form u v w
3 7 30
Enter the edge and cost in the form u v w
4 5 35
Enter the edge and cost in the form u v w
2 4 40
Enter the edge and cost in the form u v w
0 3 45
Enter the edge and cost in the form u v w
6 7 50
Enter the edge and cost in the form u v w
0 1 55

Spanning tree exists
The spanning tree is shown below
1 4
5 6
4 6
3 4
0 2
3 7
2 4
Cost of spanning tree is = 145

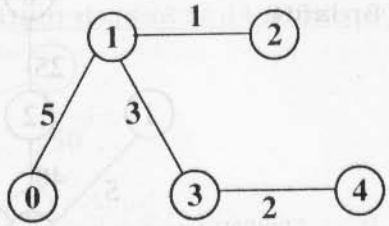


Output 2: For the given graph enter the data as shown below:

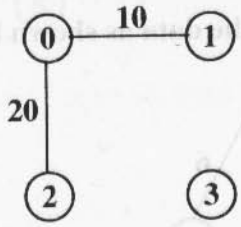


Enter the number of nodes
5
Enter the number of edges
7
Enter the edge list
Enter the edge and cost in the form u v w
1 2 1
Enter the edge and cost in the form u v w
3 4 2
Enter the edge and cost in the form u v w
1 3 3
Enter the edge and cost in the form u v w
2 3 4
Enter the edge and cost in the form u v w
0 1 5
Enter the edge and cost in the form u v w
0 3 6
Enter the edge and cost in the form u v w
2 4 6

Spanning tree exists
The spanning tree is shown below
1 2
3 4
1 3
0 1
Cost of spanning tree is = 11



Output 3: For the given graph enter the data as shown below:



Enter the number of nodes
4
Enter the number of edges
2
Enter the edge list
Enter the edge and cost in the form u v w
0 1 10
Enter the edge and cost in the form u v w
0 2 20
Spanning tree does not exist

Program No. 10.a: Print all the nodes reachable from a given starting node in a digraph using Breadth First Search method.

```
#include <iostream.h>

class REACHABLE
{
        private:
                int n;                  // Number of vertices in the graph
                int s[20];              // Holds the info of what nodes are rechable
                int a[20][20];          // The adjacency matrix of given graph
                int source;             // rom where BFS has to take place
        public:
                void bfs_traversal();
                void read_data();
                void print_data();
};
void REACHABLE::read_data()
{
        cout << "Enter the no. of nodes" << endl;
        cin >> n;

        cout << "Enter the  adjacency matrix" << endl;
        for ( int i = 0; i < n; i++)
        {
                for ( int j = 0; j < n; j++)
                {
                        cin >> a[i][j];
                }
        }

        cout << "Enter the source" << endl;
        cin >> source;
}

void REACHABLE::print_data()
{
        // Display the nodes that are reachable and that are not reachable
        for( int i = 0; i < n; i++)
        {
                if ( s[i] == 0 )
                        cout << "vertex " << i << "is not reachable" << endl;
                else
                        cout << "vertex " << i << "is reachable" << endl;
        }
}
```

```
void REACHABLE::bfs_traversal()
{
        int f,r,q[20];      // for queue operations
        int u,v;            // represent two vertices
        int i;              // index variable

        for (i = 1; i <= n; i++) s[i] = 0;          // No node is visited in the beginning

        f = r = 0;                                  // Queue is empty
        q[r] = source;                              // Insert source vertex into queue

        s[source] = 1;                              // Add source to S (indicates source is visited)

        while ( f <= r )                            // As long as queue is not empty
        {
                u = q[f++];                         // Delete the next vertex to be explored from q

                for ( v = 1; v <= n; v++)
                {
                        // Find the nodes v which are adjacent to u

                        if ( a[u][v] == 1 && s[v] == 0 )
                        {
                                s[v] = 1;           // add v to s indicates that v is visited now

                                q[++r] = v;         // Insert new vertex into Q for exploration
                        }
                }
        }
}

void main()
{
        REACHABLE a;

        a.read_data();
        a.bfs_traversal();
        a.print_data();
}
```

Input: adjacency matrix for the graph shown in figure 5.3.1.a.

A	0	1	2	3	4	5	6	7	8	9
0	0	1	1	1	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	1	0	0	0	0
3	0	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	0	1	1	0
7	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Output:

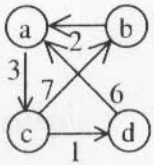
```
Enter the source
0
vertex 0 is reachable
vertex 1 is reachable
vertex 2 is reachable
vertex 3 is reachable
vertex 4 is reachable
vertex 5 is reachable
vertex 6 is reachable
vertex 7 is reachable
vertex 8 is reachable
vertex 9 is reachable
```

Program No. 10.b:Implement All Pair Shortest paths problem using Floyd's algorithm.

```
# include<iostream.h>

class floyd
{
        private:
                int n;                  // number of nodes in the graph
                int cost[10][10];       // cost adjacency matrix
                int d[10][10];          // Holds the shortest distance
        public:
                void dist();            // Finds the shortest distance
                void read_data();       // Read the costadjacency matrix
                void write_data();      // Display all pairs shortest distance
};

/* Function to find minimum of two numbers */
int min(int a, int b)
{
        return a < b ? a : b;
}

void floyd :: write_data()
{
        cout << "The distance matrix is shown below\n";
        for ( int i = 0; i < n ; i++ )
        {
                for ( int j = 0; j < n; j++ )
                        cout << d[i][j] << " ";
                cout <<endl;
        }
}

void floyd :: read_data()
{
        cout << "Enter the number of nodes\n";
        cin >> n;

        cout << "Enter the cost adjacency matrix\n";
        for ( int i = 0; i < n; i++ )
        {
                for ( int j = 0; j < n; j++ )
                        cin >> cost[i][j];
        }
}
```

```
void floyd :: dist()
{
        int i,j,k;

        /* Make a copy of the cost adjacency matrix */
        for ( i = 0; i < n; i++)
        {
                for ( j = 0; j < n; j++)
                {
                        d[i][j] = cost[i][j];
                }
        }

        /* Find the solution to all pairs shortest paths problem */
        for ( k = 0; k < n; k++ )
        {
                for ( i = 0; i < n; i++ )
                {
                        for ( j = 0; j < n; j++ )
                        {
                                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
                        }
                }
        }
}

void main()
{
        floyd f;

        f.read_data();          // Read the cost adjacency matrix

        f.dist();               // Find all pairs shortest path

        f.write_data();         // Display the result
}
```

Output 1
If the given graph is



Enter the data as shown below:

Enter the no. of nodes
4
Enter the cost adjacency matrix

0	9999	3	9999
2	0	9999	9999
9999	7	0	1
6	9999	9999	0

Solution to all pairs shortest path problem is

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

Output 2

Enter the no. of nodes
5
Enter the cost adjacency matrix

0	2	9999	1	8
6	0	3	2	9999
9999	9999	0	4	9999
9999	9999	2	0	3
3	9999	9999	9999	0

Solution to all pairs shortest path problem is

0	2	3	1	4
6	0	3	2	5
10	12	0	4	7
6	8	2	0	3
3	5	6	4	0

Program No. 11: Find a subset of a given set S = {sl,s2,....s_n} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1,2,6} and {l,8}. A suitable message is to be displayed if the given problem instance doesn't have a solution. (C Program)

Note: The recursive program is shown below:

```
#include<stdio.h>

int count;                                  /* Initial number of solutions */
int w[10];                                  /* Holds the elements */
int d;                                      /* The total sum required */
int x[10];                                  /* Holds the items whose sum is equal to d */

void subset(int cs,int k,int r)
{                                           /* r Required remaining sum */

        int i;

        x[k]=1;                             /* include kth element */

        if(cs+w[k] == d)                    /* If current sum+kth element = d */
        {
                printf("\nSubset solution = %d\n", ++count);
                for(i=0;i<=k;i++)
                {
                        if(x[i] == 1) printf("%d  ",w[i]);
                }
        }
        else  if(cs+w[k]+w[k+1] <= d)
                        subset(cs+w[k],k+1,r-w[k]);

        if((cs+r-w[k]>=d) && (cs+w[k+1])<=d)
        {
                x[k]=0;
                subset(cs,k+1,r-w[k]);
        }
}

void main()
{
        int sum=0,i, n;

        printf("Enter no of elements\n");
        scanf("%d",&n);
```

```
    printf("Enter the elements in ascending order\n");
    for(i=0;i<n;i++) scanf("%d",&w[i]);

    printf("Enter the required sum\n");
    scanf("%d",&d);

    for(i=0;i<n;i++) sum += w[i];

    if(sum < d)
    {
            printf("No solution exists\n");
            return;
    }

    printf("The solution is\n");

    count = 0;

    subset(0,0,sum);
}
```

Output

Enter the value of n 5
Enter the set in increasing order
1 2 5 6 8
Enter the maximum subset value of D
9
Solution is
1 2 6
Solution is
1 8

Note: The iterative program is shown below:

```
#include <stdio.h>

void subset(int n, int d, int w[])
{
        int s,k,i,x[10];

        for (i=1; i <= n; i++) x[i] = 0;

        s = 0;
        k = 1;
        x[k] = 1;

        while (1)
        {
                if( k <= n && x[k] == 1)
                {
                        if ( s+w[k] == d)
                        {
                                printf("Solution is \n");
                                for (i = 1; i <= n; i++)
                                {
                                        if (x[i] == 1)
                                                printf("%d ",w[i]);
                                }
                                printf("\n");
                                x[k] = 0;
                        }
                        else if (s + w[k] < d )
                        {
                                s += w[k];
                        }
                        else
                        {
                                x[k] = 0;
                        }
                }

                else
                {
                        k--;
                        while ( k >0 && x[k] == 0)
                        {
                                k--;
                        }
```

```
                if (k == 0) break;

                x[k] = 0;
                s = s-w[k];
            }
            k = k + 1;
            x[k] = 1;
        }
}

void main()
{
        int n,i,d,w[10];

        printf("Enter the value of n");
        scanf("%d",&n);

        printf("Enter the set in increasing order\n");
        for (i= 1; i <= n; i++)
                scanf("%d",&w[i]);

        printf("Enter the maximum subset value of D\n");
        scanf("%d",&d);

        subset(n,d,w);
}
```

Output

Enter the value of n 5
Enter the set in increasing order
1 2 5 6 8
Enter the maximum subset value of D
9
Solution is
1 2 6
Solution is
1 8

Program No. 12.a: Implement Horspool algorithm for String Matching

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
class HORSPOOL
{
        private:
                char p[40];             // Pattern to be matched
                char t[40];             // Text string
                int   m;                // Length of pattern string
                int   n;                // Length of text string
                int   s[256];           // Holds the shift size required

        public:
                void shift_table();     // Computes the shift table
                int pattern_match();    // Search for the pattern in the subject string
                void read_data();       // Reads text and pattern string
};

void HORSPOOL::read_data()
{
        cout << "Enter the text string" << endl;
        gets(t);

        cout << "Enter the pattern string" << endl;
        gets(p);
}

// Function to obtain the shift table for the characters in text string
void HORSPOOL::shift_table()
{
        int i;

        // Length of pattern string
        m = strlen(p);

        // Initialize the table with pattern length
        for (i = 0; i < 128; i++) s[i] = m;

        //Find the distance of first m-1 characters of string from the last character
        for (i = 0; i < m-1; i++)
        {
                s[p[i]]= m - 1 - i;
        }
}
```

```
// Function for pattern matching
int HORSPOOL::pattern_match()
{
        int i, k;

        shift_table();                  // Obtain the shifs required when mismatch occur

        m = strlen(p);                          // Compute the length of pattern string
        n = strlen(t);                          // Compute the length of text string
        i = m - 1;                              // Position of pattern string's right end

        while ( i < n )                         // Repeat till the end of text string
        {
                k = 0;
                while ( k < m  && t[i-k] == p[m-1-k] )
                {
                        k++;                    // Matching of pattern and text string in progress
                }

                // Pattern found ?
                if ( k == m ) return i - m + 1;

                // No match, so shift the pattern towards right
                i = i + s[t[i]];
        }

        return -1;
}

void main()
{
        HORSPOOL a;

        int  pos;

        a.read_data();                          //Read text string and pattern string

        pos = a.pattern_match();//Search for pattern in the text string

        if ( pos == -1)
                cout << "Pattern string not found\n";
        else
                cout << "Pattern string found at " << pos + 1 << " position";
}
```

Program No. 12.b: Find the Binomial Co-efficient using Dynamic Programming.

```
#include<iostream.h>

int binomial_coeff(int n,int k)
{
        int     i, j;           /* Index variables such that 0 <= i <= n and 0 <= j <= k */
        int     c[10][10];      /* To store the binomial coefficients*/

        for ( i = 0; i <= n; i++)
        {
                for ( j = 0; j <= k; j++)
                {
                        if ( j == 0 || i == j)
                                c[i][j] = 1;        /* Fill first column and diagonal items with 1*/
                        else
                                c[i][j] = c[i-1][j-1] + c[i-1][j];
                }
        }
        return c[n][k];                     /* Return nCk */
}

void main()
{
        int n,k;

        cout << "Enter the value of n and k(n>k)" << endl;
        cin >> n >> k;

        int res = binomial_coeff (n,k);

        cout << "C(" << n << "," << k << ") = "<< res;
}
```

Output

Enter the value of n and k(n>k)
6 3
C(6,3) = 20

Program No. 13: Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

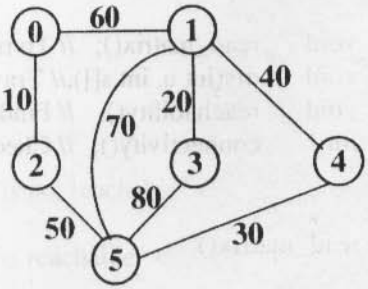
```
#include <iostream.h>

class PRIMS
{
    private:
        int n;              // Number of nodes in the graph
        int a[10][10];      // Cost adjacency matrix

    public:
        void read_data();
        void minimum_spanning_tree();
};

void PRIMS::minimum_spanning_tree()
{
    int i,j,u,v,min;
    int sum;                /* Holds the cost of the minimum spanning  tree */
    int k;                  /* Index for storing the edges w.r.t minimum spanning tree */
    int t[10][2];           /* Holds the minimum spanning tree */
    int p[10];              /* Holds the vertices selected */
    int d[10];              /* Holds the weights for the selected edges */
    int s[10];              /* Has the information of what nodes are visited
                                                and what nodes are not visited */
    int source;             /* Contains a vertex from which least edge starts */


    /* Find the source vertex from which least edge starts */
    min = 9999;
    source = 0;

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
                if (a[i][j] != 0 && a[i][j] <= min)
                {
                        min = a[i][j];
                        source = i;
                }
        }
    }
```

```
/* Initialization */
for (i=0; i < n; i++)
{
    d[i] = a[source][i];
    s[i] = 0;
    p[i] = source;
}

s[source] = 1;          /* Add source to S */

sum = 0;                /* Initial cost of minimum spanning tree */
k = 0;                  /* used as index to store the edges selected */

for(i=1; i < n; i++)
{
    /* fund u and d[u] such that d[u] is minimum
       and u in V-S
     */
    min = 9999;
    u = -1;
    for (j=0; j < n; j++)
    {
            if ( s[j] == 0 )
            {
                    if ( d[j] <= min )
                    {
                            min = d[j];
                            u = j;
                    }
            }
    }

    /* Select an edge with least cost */
    t[k][0] = u;
    t[k][1] = p[u];
    k++;

    /* Add the cost associated with edge to get total cost of
            minimum spanning tree */
    sum += a[u][p[u]];

    /* add u to s */
    s[u] = 1;
```

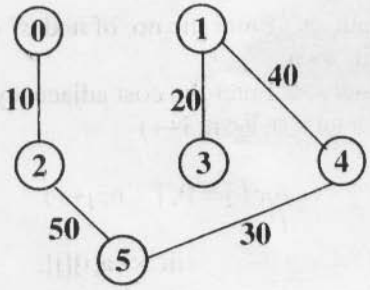
```
        /* find d[v] and p[v] for every v in V-S */
        for ( v = 0; v < n; v++ )
        {
                if ( s[v] == 0 && a[u][v] < d[v] )
                {
                        d[v] = a[u][v];
                        p[v] = u;
                }
        }
    }

    /*Check for the spanning tree */
    if (sum >= 9999)
        cout << "Spanning tree does not exist" << endl;
    else
    {
        cout << "Spanning tree exists and minimum spanning tree is" << endl;
        for (i = 0; i < n-1; i++)
        {
                cout << t[i][0] << "  " << t[i][1] << endl;
        }

        cout << "The cost of the spanning tree = " << sum << endl;
    }
}

void PRIMS::read_data()
{
    int i,j;

    cout << "Enter the no. of nodes" << endl;
    cin >> n;

    cout << "Enter the cost adjacency matrix" << endl;
    for(i=0; i < n ; i++)
    {
        for (j=0; j < n; j++)
        {
                cin >> a[i][j];
        }
    }
}
```

```
void main()
{
    PRIMS a;

    a.read_data();

    a.minimum_spanning_tree();
}
```

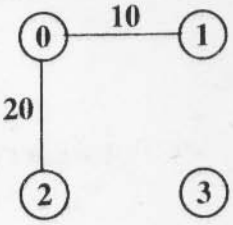
Output 1 for the graph shown below:




```
Enter the no. of nodes
6
Enter the cost adjacency matrix
  0    60   10   9999 9999 9999
 60     0   9999  20   40   70
 10   9999   0   9999 9999  50
9999  20   9999   0   9999  80
9999  40   9999 9999   0    30
9999  70    50    80   30    0
Spanning tree exists and minimum spanning tree is
0 2
5 2
4 5
1 4
3 1
The cost of the spanning tree = 150
```



Output 2 for the graph shown below:




```
Enter the no. of nodes
4
Enter the cost adjacency matrix
0      10     20   9999
10      0   9999  9999
20    9999   0    9999
9999 9999 9999     0
Spanning tree does not exist
```

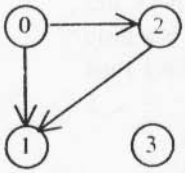
Program No. 14.a: Print all the nodes reachable from a given starting node in a given digraph using Depth First Search method.

```
#include <iostream.h>
class GRAPH
{
        private:
                int     n;              // Number of vertices in the graph
                int     a[10][10];      // Holds adjacency matrix
        public:
                void    read_matrix();  // To read the adjacency matrix
                void    dfs(int u, int s[]);// Travaerse the graph in DFS
                void    reachability();  // Find the nodes that are reachable and that are not
                int     connectivity();  // Check whether the graph is connected or not
};

void GRAPH :: read_matrix()
{
        int     i, j;

        cout << "Enter the no. of nodes" << endl;
        cin >> n;
        cout << "Enter the cost adjacency matrix" << endl;
        for ( i = 0; i < n; i++)
        {
                for ( j = 0; j < n; j++)
                {
                        cin >> a[i][j];
                }
        }
}

void GRAPH ::dfs(int u, int s[])
{
    int v;

    s[u] = 1;

    for ( v = 0; v < n; v++)
    {
        if ( a[u][v] == 1 && s[v] == 0 )
        {
            dfs(v, s);
        }
    }
}
```

```
void GRAPH :: reachability()
{
        int     i, source, s[10];

        for( i = 0; i < n; i++ ) s[i] = 0;

        cout << "Enter the source" << endl;
        cin >> source;

        dfs(source, s);

        for ( i = 0; i < n; i++)
        {
                if ( s[i] == 0)
                        cout << "Node " << i << "is not reachable" << endl;
                else
                        cout << "Node " << i << "is reachable" << endl;
        }
}

int GRAPH :: connectivity()
{
        int     i, j, flag, s[10];

        for (j = 0; j < n; j++)
        {
                for ( i = 0; i < n; i++) s[i] = 0;

                dfs(j, s);

                flag = 0;

                for ( i = 0; i < n; i++)
                {
                        if ( s[i] == 0 ) flag = 1;
                }

                if (flag ==0) return 1;
        }

        return 0;
}
```

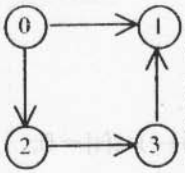
```
void main()
{
        GRAPH g;

        g.read_matrix();            // Input the cost adjacency matrix

        g.reachability();           // Find the nodes that are reachable and that are not reachable

        int flag = g.connectivity();

        if (flag == 1)
                cout << "The graph is connected" << endl;
        else
                cout << "The graph is not connected" << endl;
}
```

Output 1




```
Enter the no. of nodes
4
Enter the cost adjacency
matrix
0 0 1 0
1 0 1 0
0 0 0 0
0 0 0 0
The graph is not connected
```

Output 2



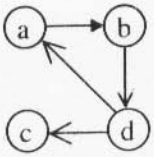

```
Enter the no. of nodes
4
Enter the cost adjacency
matrix
0 1 1 0
0 0 0 0
0 0 0 1
0 1 0 0
The graph is connected
```

Program No. 14.b: Compute the transitive closure of a given directed graph using Warshall's algorithm.

```
# include<iostream.h>

class warshall
{
        private:
                int n;                          // number of nodes in the graph
                int a[10][10];                  // cost adjacency matrix
                int p[10][10];                  // Holds the shortest distance
        public:
                void path_matrix();             // Finds the shortest distance
                void read_data();               // Read the adjacency matrix
                void write_data();              // Display transitive closure (path matrix
};

void warshall :: write_data()
{
        cout << "The path matrix is shown below\n";
        for ( int i = 0; i < n ; i++ )
        {
                for ( int j = 0; j < n; j++ )
                {
                        cout << p[i][j] << " ";
                }
                cout <<endl;
        }
}

void warshall::read_data()
{
        cout << "Enter the number of nodes\n";
        cin >> n;

        cout << "Enter the adjacency matrix\n";
        for ( int i = 0; i < n; i++ )
        {
                for ( int j = 0; j < n; j++ )
                {
                        cin >> a[i][j];
                }
        }
}
```

```
void warshall::path_matrix()
{
        int i,j,k;

        /* Make a copy of the adjacency matrix */
        for ( i = 0; i < n; i++)
        {
                for ( j = 0; j < n; j++) p[i][j] = a[i][j];
        }
        /* Find the transitive closure(path matrix) */
        for ( k = 0; k < n; k++ )
        {
                for ( i = 0; i < n; i++ )
                {
                        for ( j = 0; j < n; j++ )
                        {
                                if ( (p[i][k] ==1 && p[k][j] ==1) ) p[i][j] = 1;
                        }
                }
        }
}

void main()
{
        warshall w;

        w.read_data();          // Read the adjacency matrix

        w.path_matrix();        // Find the pathmatrix

        w.write_data();         // Display the path matrix
}
```

Output
If the given graph is



Enter the data as shown below:

Enter the no. of nodes
4

Enter the adjacency matrix
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0
The transitive closure(path matrix) is shown below

1	1	1	1
1	1	1	1
0	0	0	0
1	1	1	1

Output 2

Enter the no. of nodes
4
Enter the cost adjacency matrix
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
The transitive closure(path matrix) is shown below

0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

Program No. 15: Implement N Queen's problem using Back Tracking.

```
#include <iostream.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0

void print_solution(int n,int x[])
{
        char c[10][10];

        // No queen has placed initially
        for (int i = 1; i <= n; i++)
        {
                for (int j = 1; j <= n; j++)
                {
                        c[i][j] = 'X';
                }
        }

        // Place the queens on the chess board
        for (i = 1; i <= n; i++)
        {
                c[i][x[i]] = 'Q';
        }

        // Print where the queens have been placed on chess board
        for (i = 1; i <= n; i++)
        {
                for ( int j = 1; j <= n; j++)
                {
                        cout<<c[i][j];
                }
                cout<<"\n";
        }
}
```

```
// Function to check whether the queen can be placed successfully or not
int place(int x[],int k)
{
        for ( int i = 1; i < k; i++ )
        {
                // Check whether two queens attach vertically or diagonally
                if ( x[i] == x[k] || i-x[i]==k-x[k] || i + x[i] == k + x[k])
                {
                        return FALSE;  // Queen can not be placed in the kth column
                }
        }

        // kth queen can be successfully placed
        return TRUE;
}

void nqueens(int n)
{
        int x[10];
        int count = 0;     // Number of solutions
        int k = 1;         // Select the first quuen

        x[k] = 0;          // But, not placed on the chess board

        while ( k != 0 )   // A qeen exists?
        {
                x[k] += 1;     // Place the kth queen in next column

                // Satisfy both explicit and implicit constraint
                while ( (x[k] <= n) && ( !place(x,k) ) )
                {
                        x[k] += 1;     // Place queen in next column
                }

                // if queen successully placed?
                if (x[k] <= n )
                {
                        // If all queens are placed
                        if ( k == n )
                        {
                                count++;
                                cout <<"solution " << count <<"is\n";
                                print_solution(n,x);
                        }
```

```
                else
                {
                        k++;            // select the next queen
                        x[k] = 0;       // but do not place
                }
            }
            else
            {
                //Backtrack and select previous queen
                k--;
            }
        }

        // No more solution exists
        return;
}

void main()
{

        int n;

        cout << "Enter the no. of queens\n";
        cin >> n;
        nqueens(n);
}
```

Output

```
Enter the no. of queens
4
solution 1is
XQXX
XXXQ
QXXX
XXQX

solution 2is
XXQX
QXXX
XXXQ
XQXX
```

SAI VIDYA INSTITUTE OF TECHNOLOGY

Rajanukunte, Bangalore - 560 064.
Phone No. : 28468191,93,96,97 email : principal@svit.co.in
(Approved by AICTE New Delhi, Recognised by Govt. of Karnataka, Affiliated to VTU)
CET Code No. : 173 COMED-K No. : E - 122
www.svit.co.in

Promoted by Eminent Professors :

Prof. A.M. Padma Reddy
Head of the Department, Computer Science & Engineering
Information Science & Engineering

Prof. Y. Jayasimha (YJ)
Head of the Department, Electronics & Communication

Prof. R.C. Shanmukha Swamy (RCS)
Head of the Department, Electrical & Electronics

Under the leadership :

Prof. M.R. Holla, President
Sri Sai Vidya Vikas Shikshana Samithi

SRI NANDI PUBLICATIONS

No. 256, 2nd Main, 10th Cross, Padmanabhanagar,
(Banashankari 2nd Stage) Bangalore - 560 070.
Ph: 080-26698274, 26694533 Email: padmareddy_am@yah

ANALYSIS & DESIGN
RS 200.00
EBK 299426