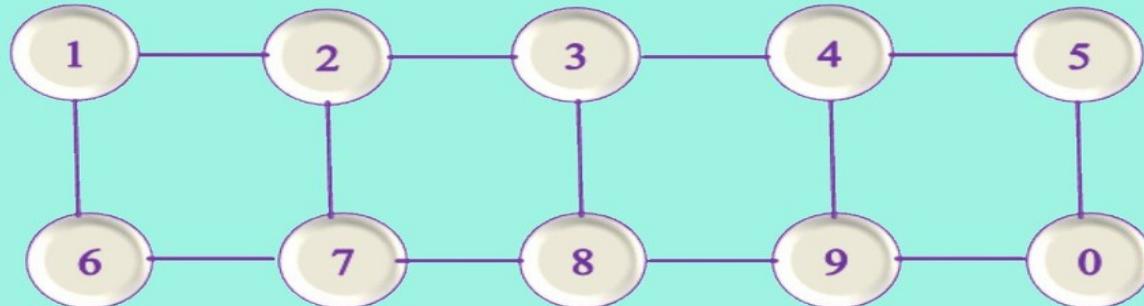
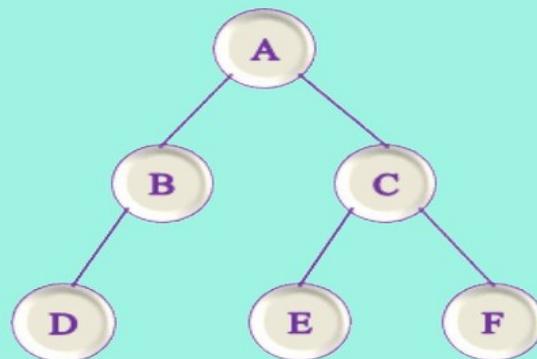
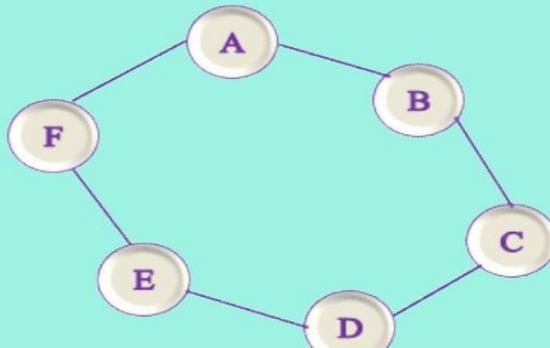


# **DESIGN AND ANALYSIS OF ALGORITHMS**

**The Learners Approach**



**BY: RASHMI V**

# **DESIGN AND ANALYSIS OF ALGORITHMS**

## **The Learners Approach**

Published by Rashmi V

Version 1.0

Copyright © 2020

Author: Rashmi V

# Preface

The Author is in teaching field for Computer Science and Engineering colleges. The author taught this subject to the students.

The design and analysis of algorithms is a slightly complex subject as students felt. The author made detailed examples to be understood by the learners through this book.

The author organized the chapters as below.

*Chapter 1* gives the overview of algorithms and how to write the pseudocode. It also describes the performance analysis in terms of time complexity and space complexity along with priori and posteriori analysis. This chapter covers the asymptotic notations including Big-Oh, Omega, Theta and Little-Oh notations. It gives instructions about probabilistic and amortized analysis. It is also elaborated on how to solve recurrence relations.

*Chapter 2* acquaints the reader with the concepts on representation of sets, types of sets, operations on sets, laws of set theory along with disjoint sets and disjoint set operations. It also includes union and find algorithms with their operations. It covers tree traversal techniques, graph traversal, spanning trees and connected components.

*Chapter 3* introduces to the divide and conquer techniques along with their advantages. It covers applications of divide and conquer approach like binary search, finding minimum and maximum, quick sort, merge sort, Stressen's matrix multiplication and examples of each application.

*Chapter 4* illustrates the greedy method properties along with the comparison with divide and conquer method. It gives the control abstraction of greedy method. Applications of greedy method like job sequencing with deadlines, Knapsack problem, minimum cost spanning trees, single source shortest problem with solved problem.

*Chapter 5* deals with dynamic programming. The key elements of dynamic programming, principle of optimality and differences between divide and conquer and greedy method. It discusses about differences of dynamic programming with divide and conquer and greedy method. General characteristics of dynamic programming along with its applications like matrix chain multiplication, optimal binary search trees, 0/1 Knapsack

problem, all pairs shortest problem, travelling sales person problem with solved problems are explained.

*Chapter 6* briefly covers about the constraints for backtracking. The comparisions between backtracking and brute force approach are explained. The efficiency of backtracking is estimated. The applications of backtracking like n-quuens problem, sum of subsets problem, graph coloring and Hamiltonian cycles are explained with solved problems.

*Chapter 7* discusses branch and bound method. The backtracking method and branch and bound method differences are given. It covers the topics about least cost search, bounding, FIFO branch and bound, LIFO branch and bound. The applications of branch and bound method including travelling sales person problem and 0/1 Knapsack problem are explained with the solved problem.

*Chapter 8* explains NP-hard and NP-complete problems concepts. The non-deterministic algorithms for sorting and searching are discussed. It also includes decision problem and optimization problem. It discusses about satisfiability, NP-hard and NP- complete classes along with cook's theorem.

I sincerely hope that the readers will be able to make most out of this book.

Good luck!

# Table of Contents

## Preface

### 1 OVERVIEW OF ALGORITHMS

#### 1.1 WHAT IS ALGORITHM?

1.1.1 The Definition to Algorithm

1.1.2 Outline of Algorithm Design

1.1.3 Notation of Algorithm

1.1.4 Properties of Algorithm

1.1.5 Process for Design and Analysis of Algorithm

    1.1.5.1 Understanding the Problem

    1.1.5.2 Develop Problem Solving Technique

    1.1.5.3 Design of an Algorithm

    1.1.5.4 Validation of an Algorithm

    1.1.5.5 Analyze the Algorithm

    1.1.5.6 Coding of an Algorithm

    1.1.5.7 Testing of an Algorithm

#### 1.2 WHAT IS PSEUDOCODE WHILE EXPRESSING ALGORITHM?

#### 1.3 WHAT IS PERFORMANCE ANALYSIS?

1.3.1 Space Complexity

1.3.2 Time Complexity

    1.3.2.1 Unit of Algorithm's Run-Time

    1.3.2.2 Order of Growth

    1.3.2.3 Best Case, Average Case and Worst Case Efficiencies

    1.3.2.4 Solved Examples

### 1.3.3 Priori Analysis and Posteriori Analysis Comparisons

## 1.4 WHAT IS ASYMPTOTIC NOTATION?

### 1.4.1 Big-Oh Notation (O)

#### 1.4.1.1 Big-Oh Ratio Theorem

#### 1.4.1.2 Basic Efficiency Classes

### 1.4.2 Omega Notation ( $\Omega$ )

#### 1.4.2.1 Big Omega Ration Theorem

#### 1.4.2.2 Little Omega Notation ( $\omega$ )

### 1.4.3 Theta Notation ( $\Theta$ )

#### 1.4.3.1 Theta Ratio Theorem

### 1.4.4 Little-Oh Notation (o)

### 1.4.5 Order of Growth Using Limits

## 1.5 WHAT IS PROBABILISTIC ANALYSIS?

### 1.5.1 PROBABILISTIC ALGORITHMS

#### 1.5.1.1 Scope of Probabilistic Algorithms

#### 1.5.1.2 Advantages and Disadvantages of Probabilistic Algorithms

## 1.6 WHAT IS AMORTIZED ANALYSIS?

## 1.7 HOW TO SOLVE RECURRENCE RELATIONS?

### 1.7.1 Solving Recurrence Equations

#### 1.7.1.1 Substitution Method

#### 1.7.1.2 Master's Method

## 2 DISJOINT SETS, SPANNING TREES, CONNECTED AND BICONNECTED COMPONENTS

### 2.1 INTRODUCTION TO SETS

#### 2.1.1 Standard Representation of Sets

#### 2.1.2 Cardinality of Sets

- 2.1.3 Types of Sets
- 2.1.4 Operation on Sets
  - 2.1.4.1 Union of Two Sets
  - 2.1.4.1.1 Union of More Than Two Sets
  - 2.1.4.2 Intersection of Two Sets
  - 2.1.4.2.1 Introduction of More Than Two Sets
  - 2.1.4.3 Complement of a Set
  - 2.1.4.3.1 Difference or Relative Complement of Sets
  - 2.1.4.3.2 Symmetric Difference of Sets
- 2.1.5 Laws of Set Theory
- 2.2 DISJOINT SETS
- 2.3 DISJOINT SET OPERATIONS
- 2.4 UNION AND FIND ALGORITHMS
  - 2.4.1 UNION Operation
  - 2.4.2 FIND Operation
  - 2.4.3 Waiting Rule for UNION (i, j)
  - 2.4.4 Collapsing Rule for FIND (i)
- 2.5 TREES
  - 2.5.1 Tree Traversal
    - 2.5.1.1 Preorder Traversal
    - 2.5.1.2 Inorder Traversal
    - 2.5.1.3 Postorder Traversal
- 2.6 GRAPHS AND ITS REPRESENTATION
  - 2.6.1 Basic Definitions
  - 2.6.2 Graph Traversals
    - 2.6.2.1 Breadth First Search (BFS)
    - 2.6.2.2 Depth First Search (DFS) and Depth First

## Traversal (DFT)

### 2.6.2.3 DFS and BFS Comparisions

## 2.7 SPANNING TREES

## 2.8 CONNECTED COMPONENTS

# 3 DIVIDE AND CONQUER TECHNIQUE

## 3.1 INTRODUCTION TO DIVIDE AND CONQUER

## 3.2 ADVANTAGES OF DIVIDE AND CONQUER APPROACH

## 3.3 GENERAL METHOD

### 3.3.1 Control abstraction of Divide and Conquer

## 3.4 APPLICATIONS OF DIVIDE AND CONQUER

### 3.4.1 Binary Search

#### 3.4.1.1 Binary Search Algorithm

#### 3.4.1.2 Time Complexity of a Binary Search Algorithm

#### 3.4.1.3 Modified Binary Search Algorithm

### 3.4.2 Finding Maximum and Minimum

#### 3.4.2.1 Conventional Method for Maximum and Minimum

#### 3.4.2.2 Divide and Conquer Method for Maximum and Minimum

### 3.4.3 Quick Sort

#### 3.4.3.1 Quick Sort Algorithm

#### 3.4.3.2 Time Complexity of Quick Sort Algorithm

#### 3.4.3.3 Randomized Quick Sort

### 3.4.4 Merge Sort

#### 3.4.4.1 Time Complexity of Merge Sort Algorithm

#### 3.4.4.2 Space Complexity of Merge Sort Algorithm

### 3.4.5 Strassen's Matrix Multiplication

### 3.4.5.1 Strassen's Algorithm

## 4 GREEDY METHOD

### 4.1 INTRODUCTION TO GREEDY METHOD

#### 4.1.1 Properties of Greedy Algorithm

#### 4.1.2 Comparison between Divide and Conquer Approach Greedy Approach

### 4.2 GENERAL METHOD

#### 4.2.1 Control Abstraction of Greedy Method

### 4.3 APPLICATIONS OF GREEDY METHOD

#### 4.3.1 Job Sequencing with Deadlines

##### 4.3.1.1 Time Complexity of Job Sequencing with Deadlines Algorithm

##### 4.3.1.2 Solved Problems

#### 4.3.2 Knapsack Problem

##### 4.3.2.1 Time Complexity of Knapsack Algorithm

##### 4.3.2.2 Solved Problems

#### 4.3.3 Minimum Cost Spanning Trees

##### 4.3.3.1 Kruskal's Algorithm

##### 4.3.3.2 Prim's Algorithm

##### 4.3.3.3 Solved Problems

#### 4.3.4 Single Source Shortest Path Problem

##### 4.3.4.1 Solved Problems

## 5 DYNAMIC PROGRAMMING

### 5.1 INTRODUCTION TO DYNAMIC PROGRAMMING

#### 5.1.1 Key Elements of Dynamic Programming

#### 5.1.2 Principle of Optimality

#### 5.1.3 Comparison between Dynamic Programming and Divide and

## Conquer

### 5.1.4 Comparison between Dynamic Programming and Greedy Method

## 5.2 GENERAL METHOD

### 5.2.1 General Characteristics of Dynamic Programming

## 5.3 APPLICATIONS OF DYNAMIC PROGRAMMING

### 5.3.1 Matrix Chain Multiplication

#### 5.3.1.1 Time Complexity of Matrix Chain Multiplication

### 5.3.2 Optimal Binary Search Trees (OBST)

#### 5.3.2.1 Time Complexity of Optimal Binary Search Trees Algorithm

#### 5.3.2.2 Solved Problems

### 5.3.3 0/1 Knapsack Problem

#### 5.3.3.1 Time Complexity of 0/1 Knapsack Problem

#### 5.3.3.2 Solved Problems

### 5.3.4 All Pairs Shortest Problem

#### 5.3.4.1 Time Complexity of All Pairs Shortest Path Problem Algorithm

#### 5.3.4.2 Solved Problems

### 5.3.5 Travelling Sales Person Problem

#### 5.3.5.1 Solved Problem

## 6 BACKTRACKING

## 6.1 INTRODUCTION TO BACKTRACKING

### 6.1.1 Constraints for Backtracking

### 6.1.2 Comparison between Backtracking and Brute Force Approach

## 6.2 GENERAL METHOD

### 6.2.1 Estimating the Efficiency of Backtracking

## 6.3 APPLICATIONS OF BACKTRACKING

### 6.3.1 N Queens Problem

#### 6.3.1.1 4-Queens Problem

#### 6.3.1.2 8-Queens Problem

### 6.3.2 Sum of Subsets Problem

#### 6.3.2.1 Solved Problem

### 6.3.3 Graph Coloring Problem

#### 6.3.3.1 Solved Problem

### 6.3.4 Hamiltonian Cycles

## 7 BRANCH AND BOUND

### 7.1 INTRODUCTION TO BRANCH AND BOUND METHOD

#### 7.1.1 Comparison between Backtracking and Branch and Bound

### 7.2 GENERAL METHOD

#### 7.2.1 LC (Least Cost) Search

##### 7.2.1.1 Control Abstraction for LC Search

##### 7.2.1.2 Properties of LC Search

##### 7.2.1.3 15-Puzzle Problem Example

#### 7.2.2 Bounding

#### 7.2.3 FIFO Branch and Bound

#### 7.2.4 LIFO Branch and Bound

### 7.3 APPLICATIONS OF BRANCH AND BOUND METHOD

#### 7.3.1 Travelling Sales Person Problem

##### 7.3.1.1 Solved Problem

#### 7.3.2 0/1 Knapsack Problem

##### 7.3.2.1 Using LC Branch and Bound Solution

###### 7.3.2.1.1 Solved Problem

### 7.3.2.2 FIFO Branch and Bound Solution

#### 7.3.2.2.1 Solved Problems

## 8 NP-HARD AND NP-COMPLETE PROBLEMS

### 8.1 BASIC CONCEPTS

### 8.2 NON-DETERMINISTIC ALGORITHMS

#### 8.2.1 Non-Deterministic Algorithm for Searching

#### 8.2.2 Non-Deterministic Algorithm for Sorting

### 8.3 DECISION PROBLEM AND OPTIMIZATION PROBLEM

### 8.4 SATISFIABILITY

### 8.5 NP-HARD AND NP-COMPLETE CLASSES

#### 8.5.1 The Complexities of NP-Hard and NP-Complete

#### 8.5.2 Halting P

### 8.6 COOK'S THEOREM

# **1 OVERVIEW OF ALGORITHMS**

## **1.1 WHAT IS ALGORITHM?**

The word algorithm comes from the name of Persian author named Abu Jafar Mohammed Ibn Musa al Khowarizmi who was a mathematician. In mathematics, computer science and related subjects an algorithm is an effective method for solving a problem which is expressed as a finite sequence of steps.

### **1.1.1 The Definition to Algorithm**

An algorithm is typically refers to a set of instructions that can be executed by a computer to produce the desired result.

We can also define algorithm as “An algorithm is clearly specified step-by-step process of solving particular problem.” An algorithm is a set of rules that must be followed when solving a specific problem.

An algorithm is an efficient method that can be expressed within finite amount of time and space. An algorithm is the best way to represent the solution of a specific problem in a very simple and efficient way. If an algorithm is considered for a specific problem, we can implement it in any programming language, which means that the algorithm is independent from any programming language.

### **1.1.2 Outline of Algorithm Design**

The important aspect of algorithm design include creating an efficient algorithm to solve a particular problem in an efficient way using minimum time and space.

Different approaches can be used to solve a particular problem. Some algorithms can be efficient with respect with time consumption, where as other approaches may be memory efficient. However an algorithm cannot optimize both time consumption and memory usage simultaneously. If we require an algorithm to run in lesser time, more memory must be invested. If we require an algorithm to run with lesser memory, we need to have more time.

### **1.1.3 Notation of Algorithm**

Each algorithm is a module which is designed to handle specific problem relative to particular data structure. The following figure shows the notation of algorithm.

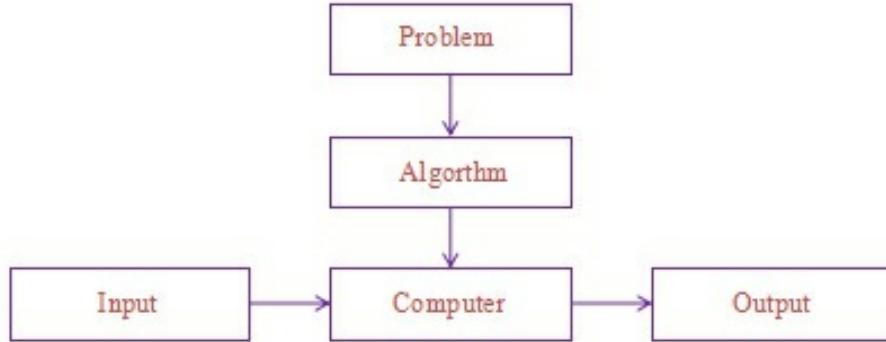


Fig. 1.1.1: Notation of Algorithm

The diagram for notation of algorithm illustrates some of the important points. They are:

- 1) The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- 2) The range of inputs for which an algorithm works has to be specified carefully.
- 3) The same algorithm can be represented in several different ways.
- 4) Several algorithms for solving the same problem may exist.
- 5) Algorithms for the same problem can be based on very different ideas and can solve the problem with different speeds.

#### 1.1.4 Properties of Algorithm

According to D. E. Knuth, the algorithm must have the following five properties.

- 1) **Input:** An algorithm has zero or more input quantities that are given to it initially before the algorithm begins or dynamically as the algorithm runs. These inputs are taken from specified set of objects. The input refers to the data (facts, figures and numbers) provided to the algorithm on which the computation is performed.
- 2) **Output:** An algorithm has one or more outputs that have a specified relation to the inputs.

- 3) **Definiteness:** Each step of an algorithm must be precisely defined which means that the actions to be carried out must be rigorously and unambiguously specified for each case. Operations such as “compute  $7/0$ ” or “add 5 or 3 to  $x$ ” are not permitted because it is not clear what the result is or which of the two possibilities should be done.
- 4) **Finiteness:** An algorithm must always terminate after a finite number of steps, each of which may require one or more operations. By tracing the instructions of an algorithm and for all cases the algorithm terminates after a finite number of steps.
- 5) **Effectiveness:** An algorithm is also generally expected to be effective, in the sense that its operations must all be sufficiently basic and have finite length of time which must be solved by someone using pencil and paper. Performing arithmetic on integers is an example of an effective operation, but arithmetic on real numbers is not because some values may be expressible by infinitely long decimal expansion.

Algorithms which are defined and effective are also called as computational procedures. An example of computational procedure is the digital computer’s operating system.

### **1.1.5 Process for Design and Analysis of Algorithm**

The process and design of algorithm has the sequence of steps one typically goes through the algorithm is shown in the following figure.

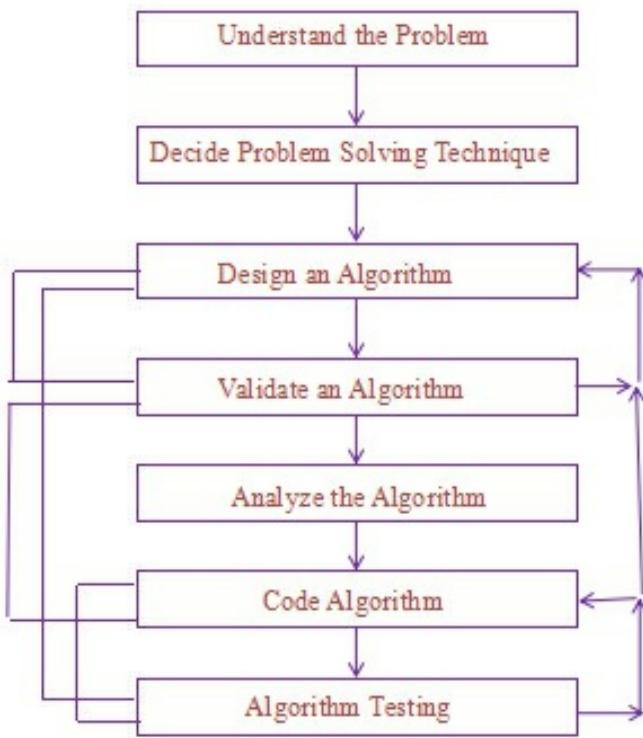


Fig. 1.1.2: Algorithm Development Process

The sequence of steps that typically goes through the algorithm is briefly discussed.

#### **1.1.5.1 Understanding the Problem**

Initially to develop an algorithm, we have to understand the problem complexity. For better understanding we must be able to read the problem carefully and ask the right questions to clarify the doubts about the problem.

#### **1.1.5.2 Develop Problem Solving Technique**

After clear understanding of the problem, it is the time to formulate the problem solving technique which is the very important step to overall solution process. Most of the problems need to give individual attention and follow useful guidelines.

#### **1.1.5.3 Design of an Algorithm**

The technique for designing an algorithm is a general approach to solve the problems with algorithms that is applicable to a variety of problems from different areas of computing. The main goal of this book is to study various

design techniques which are useful to yield good algorithms. By mastering these design strategies, it will be easier to devise new and useful algorithms. The choice of design technique can influence the effectiveness of a solution of an algorithm. Even if two different algorithms are correct it may differ simultaneously in their effectiveness. Measures of their effectiveness are discussed in later sections of this chapter. There are number of standard methods for designing of algorithms. Most important of them are Divide and Conquer Method, Greedy Method, Dynamic Programming Method, Backtracking Method and Branch and Bound Method.

#### **1.1.5.4 Validation of an Algorithm**

The algorithm validation checks the algorithm result for all legal set of input. After designing of algorithm it is necessary to check the algorithm whether the algorithm computes the desired results or not for all possible legal set of inputs. In this phase the algorithm is not converted into any programming language. But after showing the validity of method, a program can be written. This phase is known as program proving or program verification. Here, the checking of program output for all possible sets of inputs is done. Each statement should be precisely defined and all the basic operations must be correctly proved.

#### **1.1.5.5 Analyze the Algorithm**

A data structure can be represented in many ways and there exists number of algorithms to implement to that data structure. In such cases, the comparison of those algorithms which are implemented on that data structure and later on the better one is chosen. The analysis of algorithm mainly focuses on time complexity and space complexity.

When compared to the time analysis, the space requirement analysis for an algorithm is easier. Whenever necessary both the time complexity and space complexity are used. The amount of memory the program needs to run to completion is referred as space complexity. In an algorithm, the time complexity depends on the size of input, thus it is a function of input size “n”. The amount of time the algorithm (program) needed to run to completion is referred as time complexity. It should be noted that different time can arise for the same algorithm. We usually deal with best case time, average case time and worst case time for an algorithm.

The minimum amount of time requires for an algorithm of size ‘n’ is referred as “best case time complexity”. In “average case time complexity” the execution of an algorithm having typical input data of size ‘n’. Similarly, the maximum amount of time needed by an algorithm for an input of size ‘n’ is referred as “worst case time complexity”. These are the three factors influences the time required by an algorithm.

#### **1.1.5.6 Coding of an Algorithm**

Coding of an algorithm is also important task in the implementation process of the algorithm. Before going to write the code of an algorithm, the designing of entire system of computer data structures, i.e., the variables required, types of variables, number of arrays needed, size of each array, linked lists needed and subroutines needed etc. should be considered.

#### **1.1.5.7 Testing of an Algorithm**

To ensure that program works accurately and efficiently before the live operation starts testing is the stage to be implemented. Testing consists of two phases. They are debugging and profiting (performance measurement).

- 1) **Debugging:** The process of executing program on sample data sets to determine whether faulty results occur and correcting them is called as debugging. As per E. Dijkstra, “debugging can only point to the presence of errors but not to their absence”. In cases in which we cannot verify the correctness of output on sample data, the following strategy can be employed.

Consider one or more than one programmer develops the programs for the same problem and compare the output produced by these programs. If the output match, then there are good chances that they are correct. A proof of correctness is much more valuable than a thousand tests (if that proof is correct) since it guarantees that the program will work correctly for all possible inputs. In case, if an error is detected, the process halts and will proceed to the next line only when the error of the previous line has been corrected.

- 2) **Profiting:** Profiling or Performance measurement refers to an execution process performed on the data sets. The main process followed here is to evaluate both time complexity and space complexity required for implementing a particular algorithm. There

are two performance factors for this evaluation process. They are:

- i. Compilers and their options
- ii. Type of computer in which the algorithm is being executed.

In order to perform necessary optimization these complexities helps in conforming and identifying the logical places where those programs which are designed, coded and whose correctness is verified.

## **1.2 WHAT IS PSEUDOCODE WHILE EXPRESSING ALGORITHM?**

In computer science, pseudocode is an informal high-level description of the operating principle of a computer program or another algorithm. Pseudocode is not a programming language, so no compiler exists for any pseudocode.

The general procedure for representing the pseudocode is presented is shown below.

- 1) Comments must begin with // and continue till the end of line.
- 2) The blocks are indicated with matching braces: { and }. A compound statement i.e., collection of simple statements can be represented as a block. The body of procedure also forms a block. The statements are separated by ;.

Representation of a block with n statements is shown below.

```
Procedure Name  
{  
    Statement_1;  
    Statement_2;  
    ...  
    Statement_n;  
}
```

- 3) An identifier begins with a letter. Data types of variables are not explicitly declared. The types will be clear from the context. Weather the variable is local or global to a procedure will also be evident from

the context. We assume simple data types such as integer, float, char, boolean and so on. Compound data types can be formed with **records**. Example for compound for data types is shown below.

```
node = record
{
    datatype_1 data_1;
    datatype_2 data_2;
    ...
    datatype_n data_n;
    node *link;
}
```

In this example, link is a pointer to the record type *node*. Individual data items of a record can be accessed with → and period. If *p* points to a record of type *node*, *p* → *data\_1*, stands for the value of the first field in the record. If *q* is a record of type *node*, *q.data\_1* will denote its first field.

- 4) Assignment of values of variables is done using assignment operator := or ← .
- 5) There are two Boolean values, *true* or *false*. For producing the values of the logical operators (and, or, not) and the relational operators (<, >, ≥, ≤, ≠) the Boolean values are provided.
- 6) Elements of multidimensional arrays are accessed using [ and ]. For example, if *A* is a two dimensional array, the [i, j]<sup>th</sup> element of the array is denoted as *A*[i, j]. Indices of the array starts at zero.
- 7) There are three types of looping statements which can be considered. They are **for**, **while**, **repeat-until**.
  - i. The general form of a **for** loop is shown below.

```
for variable := value1 to value2 step step do
{
    Statement_1;
```

```
Statement_2;
```

```
...
```

```
Statement_n;
```

```
}
```

Here, value1, value2 and step arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of arithmetic expression. The clause “**step** step” is optional and taken as +1 if it does not occur. The step could be either positive or negative. The variable is tested for termination at the start of each iteration.

- ii. The general form of a **while** loop is shown below.

```
while (condition) do
```

```
{
```

```
Statement_1;
```

```
Statement_2;
```

```
...
```

```
Statement_n;
```

```
}
```

As long as *(condition)* is **true**, the statement gets executed. When the *(condition)* is evaluated as **false** the loop is exited. The value of *(condition)* is evaluated at the top of the loop.

- iii. The general form of a **repeat-until** loop is shown below.

```
repeat
```

```
{
```

```
Statement_1;
```

```
Statement_2;
```

```
...
```

```
Statement_n;
```

## **} until (condition)**

Here, the statements are executed as long as the (*condition*) is **false**. The value of condition is computed after executing the statements.

The instruction **break;** can be used with in any of the above looping instructions to force exit.

- 8) A conditional statement has the following forms:

**if (condition) then (statement)**

**if (condition) then (statement1) else (statement2)**

Here, (*condition*) is a boolean expression and (*statement*), (*statement1*) and (*statement2*) are arbitrary statement i.e., simple or compound.

We also employ the following **case** statements:

**case**

**{**

**(condition\_1): (statement\_1)**

**...**

**(condition\_n): (statement\_n)**

**else: (statement\_n+1)**

**}**

Here *statement\_1*, *statement\_2*, *statement\_n*, *statement\_n+1* should be either simple statements or compound statements. A case statement is interpreted as follows.

If (*condition 1*) is **true**, (*statement 1*) gets executed and the case statement is exited. If (*statement 1*) is **false**, (*condition 2*) is evaluated. If (*condition 2*) is **true**, (*statement 2*) gets executed and **case** statement exited and so on.

- 9) Input and output are done using the instructions **read** and **write**. No format is used to specify the input and output quantities.

- 10) Here, there is one type of procedure: **Algorithm**. An algorithm

consists of heading and a body. The heading takes the form,

**Algorithm Name (parameter list)**

where, *Name* is the name of procedure and (*parameter list*) is a listing of procedure parameters. The body has one or more simple or compound statements enclosed with the braces { and }. An algorithm may or may not return any values. Simple values to procedures are passed by value. Arrays and records are passed by reference. An array name or pointer name is treated as pointer to the respective data type.

## **EXAMPLE PROBLEM 1**

### **Write an algorithm for Sequential Search.**

#### Solution:

Algorithm for sequential search is shown below.

**Algorithm** SequSearch (a, x, n)

```
{  
//search for x in a[1:n] where a[0] is used as additional space  
i ← n;  
a[0] ← x;  
while (a[i]≠x) do i ← i-1;  
return i;  
}
```

## **EXAMPLE PROBLEM 2**

### **Write an algorithm for Selection Sort.**

#### Solution:

Algorithm for Selection Sort is shown below.

**Algorithm** SelectionSort (a,n)

```
//sort the array a[1:n] into nondecreasing order  
{  
for i:=1 to n do
```

```

{
j:=i;
for k:=i+1 to n do
    if (a[k]<a[j]) then j:=k;
t:=a[i]; a[i]:=a[j]; a[j]:=t;
}
}

```

### **EXAMPLE PROBLEM 3**

**Write an algorithm to find the roots of a quadratic equation for all the cases.**

Solution:

Algorithm to find the roots of quadratic equation is shown below.

**Algorithm** QuadraticRoots (a,b,c)

//a, b, c are coefficients of quadratic equation  $ax^2+bx+c=0$ .

```

{
d ← b*b-4*a*c;
if (d>0) then
{
x1 ← (-b+sqrt(d))/(2*a);
x2 ← (-b-sqrt(d))/(2*a);
print “The two real roots are”, x1, x2;
}
else if (d=0) then
{
x ← -b/(2*a);
print “Roots are equal and the root is”, x;
}
```

```
else
print "Roots are Imaginary";
}
```

## 1.3 WHAT IS PERFORMANCE ANALYSIS?

There are some criteria to judge an algorithm. Some of them are:

1. Does the algorithm do what it wants to do?
2. Does the algorithm work according to the original specifications of the task?
3. Is there any documentation that describes how it works and how to use it?
4. Are the procedures in the algorithm created in such a way that they perform logical sub-functions?
5. Is the code readable in the algorithm?

All these criteria are important and required for the writing the software especially for large systems.

Analysis of algorithm refers to a task of determining the computation time and storage space requirements of an algorithm. This task is also known as performance analysis. An efficient algorithm can be written by doing the performance analysis of the algorithm. An idea of particular algorithm is taken and have to determine its quantitative behavior, whether the algorithm is optimal and has some sense.

To analyze an algorithm, first task is to determine which operations are employed and what are their relative costs. These operations may include four basic operations on integer addition, subtraction, multiplication and division. Other basic operations are arithmetic on floating point numbers, comparisons, assigning values to variables and executing procedure calls. A fixed amount of time will be taken by all of these operations otherwise we can say that their time is bounded by a constant which is not true for all operations of a computer. Consider an example, comparison of two strings will be depended upon their individual length, while time for each character to compare is bounded by a constant.

The second task is to determine the sufficient amount of data sets which are required for the algorithm to exhibit its possible behaviour.

For the algorithm analysis, we have to determine the number of operations of the algorithm and evaluate the cost. A range of data sets should be set for correct operation. The amount of time and space are important measures for the efficiency of an algorithm.

We can analyse the algorithm by two ways.

1. By checking the correctness of the algorithm.
2. By measuring the time and space complexity of an algorithm.

Performance evaluation can be divided into two major phases. They are:

1. Priori Analysis
2. Posteriori Analysis

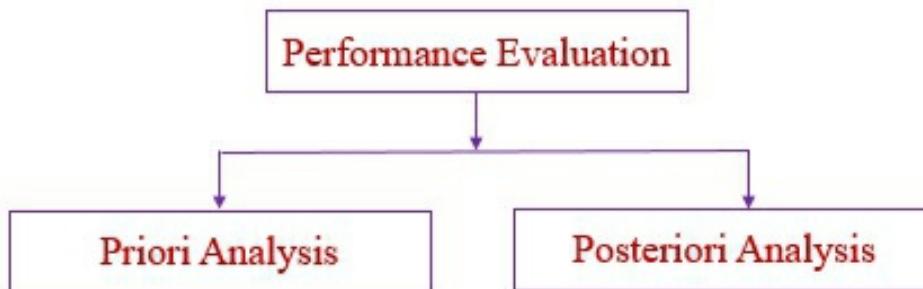


Fig. 1.3.1: Classification of Performance Evaluation

- 1) **Priori Analysis:** This is one of the most creative analysis of algorithms. Here we obtain a function which bounds the algorithm computing time. Suppose if we wish to determine the total amount of time will be spent for execution in some statements in the middle of the program, given some initial input data. This requires two items of information. They are:

- i. The statement frequency count i.e., the number of times the statement will be executed.
- ii. The amount of time taken for one execution.

The product of these two numbers is the total time. Since, the

programming languages, compiler and the machine being used are important in the time for execution. Priori analysis limits itself to determine the frequency count of each statement.

The order of magnitude determination is mainly concerned in the priori analysis, which is important for producing the algorithm in the order of magnitude.

The notations used in the priori analysis are Big-Oh ( $O$ ), Omega ( $\Omega$ ), Theta ( $\Theta$ ) and Small-Oh ( $o$ ).

The computing time in the priori analysis ignores all the factors which are machine or programming languages independent and only concentrates on determining the order of magnitude in the frequency of execution of statements.

- 2) **Posteriori Analysis:** Here, in the posteriori analysis the collection of actual statistics about the algorithm like time and space while executing should be considered. Once the algorithm is written testing should be done. Testing a program consists of two major phases namely, debugging and profiling.

The schema for producing a program performance profile is,

- i. Read (data)
- ii. Time ( $t_1$ )
- iii. Process (data)
- iv. Time ( $t_2$ )
- v. Write (time= $t_2-t_1$ )

The comparison between Analysis and profiling is as follows,

S. NO.	ANALYSIS	PROFILING
1	Analysis is the process of determining how much computing time and storage an algorithm will require.	Profiling is the process of executing the correct program on data sets. It measures the time and space it takes to compute the results.

2	Analysis is independent of machine programming language and it will not involve the execution of program.	Profiling depends on the machine, programming languages and the compiler used.
3	Analysis allows to make the qualitative judgements about the value of one algorithm over another algorithm.	Profiling helps in finding the logical places to perform optimization and consumes the previously done analysis.

Table 1.3.1: Comparison between Analysis and Profiling.

The main purpose of algorithm analysis is to design most efficient algorithms. The efficiency of algorithms mainly depends on two factors. They are:

1. Space Complexity
2. Time complexity

### 1.3.1 Space Complexity

The space complexity of an algorithm indicates the amount of temporary storage required for running an algorithm i.e., it is the amount of memory needed by the algorithm to run to completion.

In most of the cases, we do not count the storage required for the input or for the output as part of the space complexity. This is because the space complexity is used to compare different algorithms for the same problem. In such a case output/input are fixed. Without the input or output we cannot count only the storage that may be served. Since the size of input required is independent of the program itself. Space complexity refers to the worst case and usually denoted as an asymptotic expression in the size of the input. Thus,  $O(n)$  in a space algorithm requires a constant amount of space independent of the size of the input.  $O(1)$  in a space algorithm requires a constant amount of space independent of the size of input.

So, we can say that the space complexity of an algorithm is defined as the amount of memory it needs to run to completion.

The space required by an algorithm consists of the sum of following components.

- 1) **A fixed static part** that is independent of the characters of inputs and outputs. This part typically includes the instruction space for code, space for the simple variable, fixed-sized component variables, space for constants etc.
- 2) **A variable dynamic part**, that consists of space needed by component variable whose size is depended on that particular problem instance at run time being solved, the space needed by reference variables and the recursion stack space which depends on instance characteristics.

The space requirements  $S(P)$  of an algorithm  $P$  is  $S(P)=c+Sp$  (instance characteristics)

Where, ‘c’ is a constant.

Since first part is static we have to concentrate on estimating  $Sp$  (instance characteristics), because the first part is static.

### **EXAMPLE PROBLEM 1**

**Find the amount of space required by the algorithm.**

Algorithm XYZ(X, Y, Z)

{

return  $(X + Y + Y*Z) + (X + Y - Z)/(X + Y) + 44;$

}

Solution:

In the above algorithm, there are no instance characteristics and the space needed by X, Y, Z is independent by the instance characteristics, therefore we can write as

$Sp$  (instance characteristics) = 0

$S(XYZ) = 3 + 0 = 3$

Since one space is required for X, Y and Z.

◻ the space complexity is  $O(1)$ .

### **EXAMPLE PROBLEM 2**

**Find the amount of space required by the algorithm.**

```
Algorithm ADD(Y, n)
```

```
{
```

```
total=0.0;
```

```
for i = 1 to n do
```

```
    total = total + Y[i];
```

```
return total;
```

```
}
```

Solution:

For the above problem, we can make the assumption that one word of space needed for n and the space needed for Y is the space needed by the variable of type array of floating point numbers. This is at least n words since Y must be large enough to hold the n elements to be summed. Here, the problem instance is characterised by n, the number of elements to be summed. So we can write,

$$S(\text{ADD}) = (n + 3)$$

where,

n is for array Y[ ],

3 is one each for n, i and total.

□ space complexity is O(n).

### EXAMPLE PROBLEM 3

**Find the amount of space required by the algorithm.**

```
Algorithm AddR(Y, n)
```

```
{
```

```
if(n ≤ 0) then return 0.0;
```

```
else return (AddR(Y, n - 1) + X[n]);
```

```
}
```

Solution:

Since the above problem uses recursion, stack space includes the space

for formal parameters, local variables and the return address.

Here, the problem instances are characterised by ‘n’.

We can assume that return address requires one word of memory. Each call to ‘AddR’ requires at least 3 words, this includes space for the values on n, the return address and a pointer a[ ]. Depth of recursion is  $n + 1$ . Hence the space needed for this algorithm is,

$$S(\text{AddR}) = 3(n + 1) = 3n + 3$$

□ space complexity is  $O(n)$ .

### 1.3.2 Time Complexity

Time complexity is the Amount of time the algorithm or program takes for execution. It considers how fast the algorithm runs. Here, the time taken by the program for the compilation is not included in the calculation.

There are different topics related to time efficiency of algorithms. Generally researchers give more attention to time efficiency because handling memory problems is easier than time. A number of mathematical formulations will be developed to measure the algorithm efficiency. Almost all the problems have the input size which is popularly known as n. Problems like searching and sorting uses n. The travelling salesmen problem uses n for number of cities. Obviously, when n increases the time taken for the algorithm also increases. Apart from n there are number of parameters that could be used in the algorithm. However, those parameters may not grow as fast as n.

In the time complexity, all the analysis will be centred on n. For example, recall the algorithm finds the maximum element in an array of size n. It is easy to conclude the time taken for the algorithm is n, because until all the elements in the array are compared we cannot find which the maximum is.

The time  $T(P)$  taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we can concern ourselves with just the run time of the program. This run time is denoted by  $t_p$  (instance characteristics).

Since, many of the factors  $t_p$  depends on or not known at the time a program is considered. It is responsible to attempt only to estimate  $t_p$ . If we know the characteristics of the compiler to be used, we can proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on that would be made by the code for program P.

So we could obtain an expression for  $t_p(n)$  of the form as

$$t_p(n) = c_a \text{ADD}(n) + c_s \text{SUB}(n) + c_m \text{MUL}(n) + c_d \text{DIV}(n) + \dots$$

where n denotes the intense characteristics and  $c_a$ ,  $c_s$ ,  $c_m$ ,  $c_d$ , and so on respectively denote the time required for addition, subtraction, multiplication, division and so on. Here, ADD, SUB, MUL, DIV, and so on are functions whose values are the numbers of additions, subtractions, multiplications, divisions and so on that are performed in the code for P is used on an instance with characteristics n.

### 1.3.2.1 Unit of Algorithm's Run-Time

For the time taken by an algorithm to be calculated we have to know how the unit of time is considered. The standard methods of computing the time efficiency of algorithms are operation counts, step counts, asymptotic notations (mathematical analysis) and practical method (precise calculation).

One of the simplest way to analyze an algorithm is by counting the basic operations in an algorithm would give the time efficiency. Following are the two examples to illustrate the basic operations.

#### EXAMPLE:

1) for i = 1 to n do

{

....

....

}

2) for i = 1 to n do

    for j = 1 to n do

```

{
    ....
    ....
}
```

In general running time is,

$$T(n) = t * C(n)$$

Let  $C(n)$  = Number of times the code need to be executed.

$t$  = The time taken for the basic operations (code in the body of for loop)

Consider for example, if we assume  $t$  as 1, then case (1) would take  $n$  amount of time and case (2)  $n^2$  amount of time.

Whenever the size of  $n$  is increased or decreased, the time analysis may not be always be linearly proportional because it depends upon the basic operation we have in the algorithm. Therefore we analyze the algorithm efficiency by the order of growth of  $n$  is also important.

### 1.3.2.2 Order of Growth

If the algorithm are faster for smaller values of  $n$  and slower when  $n$  is larger then we cannot say these algorithms are good. This is called the order of growth of  $n$ . For understanding the order of growth some of the computing functions are shown in the table below.

S. NO.	FUNCTION	NAME
1	1	Constant
2	$\log n$	Logarithmic
3	$n$	linear
4	$n \log n$	$n \log n$
5	$n^2$	Quadratic
6	$n^3$	Cubic
7	$2^n$	Exponential
8	$n!$	Factorial

Table 1.3.2: Common Time Functions

The function shown in the above table are quite common in the analysis of algorithms. These functions have a strong significance in terms of their relative values. For example, comparing  $n$  and  $n!$ , the later one grows very fast for small variations in  $n$ . In fact the functions are written for small to large order of growth.

To understand further about the growth of these common functions typical values for  $n$  can be considered in the following table.

$n$	$\log n$	$n \log n$	$n^2$	$n^3$	$2^n$	$n!$
1	0	0	1	1	2	1
2	1	2	8	8	4	2
4	2	8	64	64	16	24
8	3	24	512	512	256	40320
16	4	64	256	4096	65536	20922789888000
32	5	160	1024	32768	4.2 x 10 <sup>9</sup>	2.6 x 10 <sup>15</sup>

Table 1.3.3: Growth of Common Time Functions

With the above table you get a better comparison among all the functions. Generally the exponential growth of algorithms can be considered as

$$O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

### 1.3.2.3 Best Case, Average Case and Worst Case Efficiencies

For some of the problems, the time complexity will not depend on the number of inputs alone. For example for searching for a specific item in an array of  $n$  elements using linear search, we have following three situations.

- 1) **Best Case:** It is the minimum number of steps that can be executed for a given parameter.

An item we are searching for may be present in the very first location itself. In this case of linear search only one item is compared and this is the best case.

- 2) **Average Case:** It is the average number of steps executed for a given parameter.

In the linear search, when the search item is present somewhere in the middle which definitely takes some time. Here, the running time is more when compared with the previous case for some value of  $n$ . Since we do not know where the item or where we have to consider the average number of cases and hence the situation is an average case.

- 3) Worst Case:** It is the maximum number of steps that can be executed for a given parameter.

Consider, the item we are searching for is not present in the array requiring  $n$  number of comparisons and running time is more than the previous two cases which may be considered as worst case.

Each of these complexities defines a numerical function time Vs. size as shown in figure below.

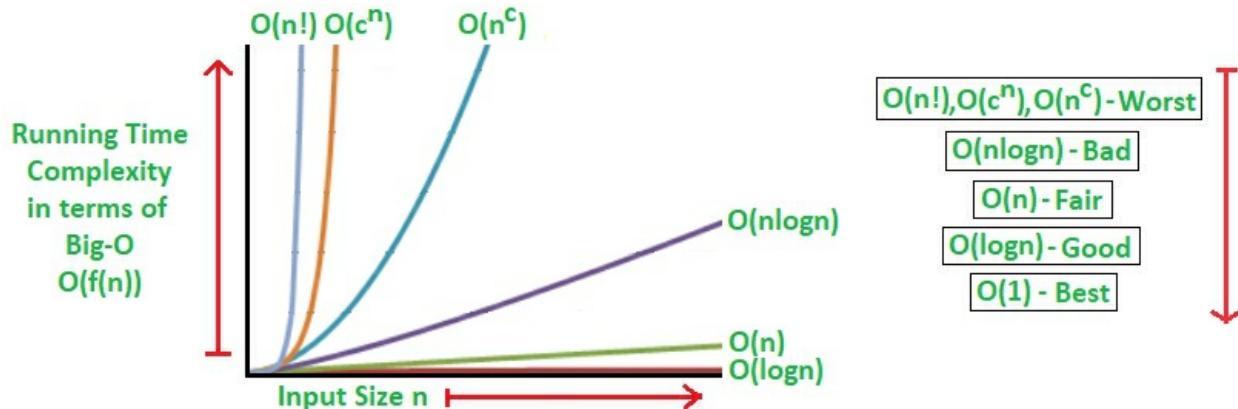


Fig. 1.3.2: Comparison of Best, Average and Worst Case Complexities

#### 1.3.2.4 Solved Examples

##### EXAMPLE PROBLEM 1

Consider an algorithm to sum (Iterative Approach) of  $n$  elements contained in an array. Find its time complexity (total steps).

Solution:

Let us consider  $y$  is an array with  $n$  elements.

Statement	Steps per Execution	Frequency	Total Steps
Algorithm Sum ( $y$ , $n$ )	0	0	0
{	0	0	0

<b>total = 0.0</b>	1	1	1
<b>for i = 1 to n do</b>	1	$n + 1$	$n + 1$
<b>    total = total + y[i];</b>	1	$n$	$n$
<b>    return total;</b>	1	1	1
<b>}</b>	0	0	0
<b>Total Steps</b>			$(2n + 3)$

The total number of steps required by the algorithm is  $(2n + 3)$ .

Where, the frequency count of the ‘for’ statement is  $(n+1)$ , since ‘i’ is to be incremented to  $(n + 1)$  before the ‘for’ loop can terminate.

### EXAMPLE PROBLEM 2

**Consider an algorithm to sum (Recursive Approach) of n elements contained in an array. Find its time complexity (total steps).**

Solution:

Let us consider, a is an array with n elements.

Statement	Steps for Execution	Frequency		Total Steps	
		$n =$	$n >$	$n =$	$n >$
<b>Algorithm sumR (a, n)</b>	0	-	-	0	0
{	0	-	-	-	-
<b>if (<math>n \leq 0</math>) then</b>	1	1	1	1	1
<b>return 0.0;</b>	1	0	0	1	0
<b>else</b>					
<b>return sumR (a, n-1) + a[n];</b>	$1 + x$	1	1	0	$1 + x$
}	0	-	-	0	0
<b>Total Steps</b>				2	$2 + x$

### EXAMPLE PROBLEM 3

**Consider an algorithm to add two matrices. Find its time complexity (total steps).**

Solution:

Let A and B are two matrices of order  $m \times n$ . The resulted matrix C is  $m \times n$

after the addition of two matrices A and B.

Statement	Steps per Execution	Frequency	Total Steps
<b>Algorithm Add (A, B, C, m, n)</b>	0	0	0
{	0	0	0
<b>for i=1 to m do</b>	1	$m + 1$	$m + 1$
<b>for j = 1 to n do</b>	1	$m(n + 1)$	$mn + m$
<b>C[i, j] = A[i, j] + B[i, j]</b>	0	$mn$	$mn$
}		-	0
<b>Total Steps</b>			$(2mn + 2m + 1)$

$$\text{The total number of steps} = (m + 1) + m(n + 1) + (m * n)$$

$$= m + 1 + mn + m + mn$$

$$= 2mn + 2m + 1$$

In the algorithm, first for loop executes  $(m + 1)$  times. The second for loop executes  $m(n + 1)$  times.

### 1.3.3 Priori Analysis and Posteriori Analysis Comparisons

The comparisons for priori and posteriori analysis are shown below.

S. NO.	PRIORI ANALYSIS	POSTERIORI ANALYSIS
1	The time taken for executing the algorithm is analysed prior to the execution of the algorithm.	The execution time taken for the algorithm is evaluated while the algorithm is being executed in the posteriori analysis.
2	Priori analysis is also known as performance analysis that evaluates whether the code is readable or not if it performs the desired functions.	Posteriori analysis is also known as performance measurement that measures the accuracy of the algorithm.
3	Priori analysis focuses on determining the order of	Posteriori analysis focuses on determining the space and

	execution of statement.	time complexity of a particular algorithm.
4	Priori analysis provides approximate values.	Posteriori analysis provides accurate values.
5	Priori analysis is less expensive	Posteriori analysis is very expensive analysis.

Table 1.3.4: Comparison between Priori Analysis and Posteriori Analysis

## 1.4 WHAT IS ASYMPTOTIC NOTATION?

Asymptotic means the study of function containing parameter ‘n’. ‘n’ can become larger and larger without bound. In asymptotic notation we are mainly concerned about the running time of an algorithm increases with the size of input.

Asymptotic notations is an another method for finding the time complexity for an algorithm. There are different kinds of mathematical notations used to represent time complexity.

### 1.4.1 Big-Oh Notation (O)

Big-Oh notation (O) will give the upper bound of the function  $f(n)$ . The upper bound of  $f(n)$  indicates that the function  $f(n)$  will be the worst case that it dose not consume more than its computing time.

**Definition:**

A function  $f(n)$  is said to be in  $\Omega(g(n))$  if  $f(n)$  is bounded below by some positive constant multiple of  $g(n)$  such that

$$f(n) \geq c * g(n) \quad \forall n, n \geq n_0$$

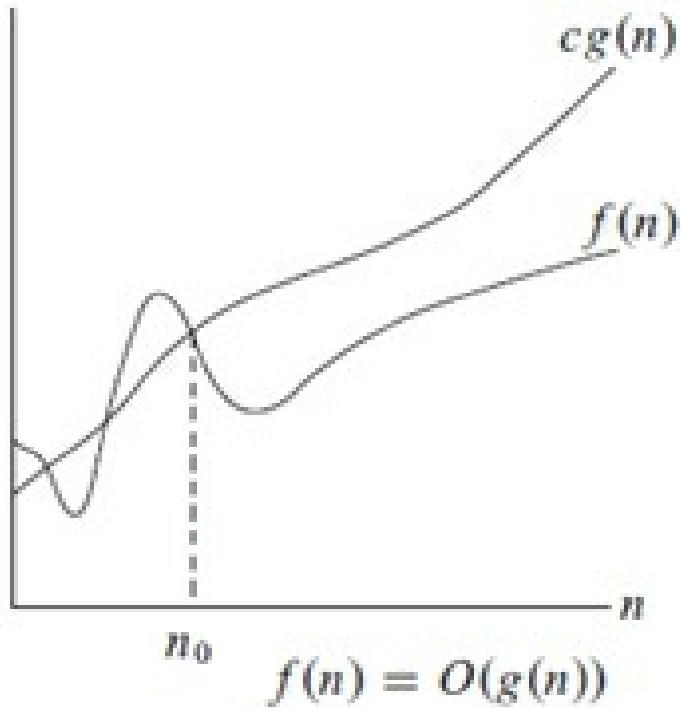


Fig. 1.4.1: Big-Oh Notation

### EXAMPLE PROBLEM 1

**Given  $f(n) = 3n + 2$ , then prove that  $f(n) = O(n)$ .**

Solution:

Given  $f(n) = 3n + 2$

$3n + 2 \leq 5n, \quad \square n \geq 1$

Here, the above inequality can be satisfied by the definition of Big-Oh notation by setting,

$c = 5, g(n) = n$ , and  $n_0 = 1$

$\square f(n) = O(n)$ .

### EXAMPLE PROBLEM 2

**Given  $f(n) = 10n^2 + 4n + 2$ , then prove that  $f(n) = O(n^2)$ .**

Solution:

Given  $f(n) = 10n^2 + 4n + 3$

$$f(n) = \begin{cases} 10n^2 + 4n + 3 \leq 10n^2 + 5n & \forall n \geq 3 \\ 5n \leq n^2 & \forall n \geq 5 \end{cases}$$

$\square 10n^2 + 4n + 3 \leq 10n^2 + n^2 \quad \square n \geq 5$

$\square 10n^2 + 4n + 3 \leq 11n^2 \quad \square n \geq 5$

The above inequality can be satisfied using the definition of Big-Oh notation by setting,

$c = 11, g(n) = n^2$  and  $n_0 = 5$

$\square f(n) = O(n^2)$

### EXAMPLE PROBLEM 3

**Given  $f(n) = 20n^3 - 3$ , then prove that  $f(n) = O(n^3)$ .**

Solution:

Given  $f(n) = 20n^3 - 3$

$20n^3 - 3 \geq 20n^3 \quad \square n \leq 1$

The above inequality can be satisfied using the definition of Big-Oh notation by setting,

$c = 20, g(n) = n^3$  and  $n_0 = 1$

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

$\square f(n) = O(n^3)$

#### 1.4.1.1 Big-Oh Ratio Theorem

**THEOREM:**

Let  $f(n)$  and  $g(n)$  be such that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  exists, then a function  $f \in O(g)$ . If  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \alpha$ , and also includeing the case in which limit is 0.

PROOF:

If  $f(n) = O(g(n))$  then,

for all  $n, n \geq n_0$ , positive 'c' and ' $n_0$ ' exist such  $\frac{f(n)}{g(n)} \leq c$ .

$$\text{Hence, } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c.$$

Now, suppose  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c$ , then that means an ' $n_0$ ' exists such that  $f(n) \geq \max(1, c) * g(n)$  for all  $n, n \geq n_0$ .

### EXAMPLE PROBLEM 1

$f(n) = 3n + 2$ , then prove that  $f(n) = O(n)$ .

Solution:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{3n + 2}{n}$$

$$= \lim_{n \rightarrow \infty} \frac{n(3 + \frac{2}{n})}{n}$$

$$= \lim_{n \rightarrow \infty} 3 + \frac{2}{n}$$
$$= 3$$

Since 3 is a constant.

$$\square f(n) = O(n).$$

### EXAMPLE PROBLEM 2

Consider the function  $f(n) = 2n + 2$  and  $g(n) = n^2$ . Then we have to find a constant  $c$  so that  $f(n) \leq c * g(n)$ .

Solution:

As  $f(n) = 2n + 2$  and  $g(n) = n^2$  then, we find  $c$  for  $n = 1$

$$f(n) = 2n + 2$$

$$= 2(1) + 2$$

$$f(n) = 4$$

and  $g(n) = n^2$

$$= (1)^2$$

$$g(n) = 1$$

i.e.,  $f(n) > g(n)$

If  $n = 2$  then,

$$f(n) = 2n + 2$$

$$= 2(2) + 2$$

$$f(n) = 6$$

and  $g(n) = n^2$

$$= (2)^2$$

$$g(n) = 4$$

i.e.,  $f(n) > g(n)$

If  $n = 3$  then,

$$f(n) = 2n + 2$$

$$= 2(3) + 2$$

$$f(n) = 8$$

and  $g(n) = n^2$

$$= (3)^2$$

$$g(n) = 9$$

i.e.,  $f(n) < g(n)$  is true.

Hence we conclude that for  $n > 2$  we obtain

$$f(n) < g(n)$$

Thus always the **upper bound** of existing time is obtained by **Big-Oh Notation**.

### 1.4.1.2 Basic Efficiency Classes

1. Frequency count is refers to the number of times a statement gets executed.
2. We have to take the frequency count of each operation to calculate the time complexity.
3. Among of all frequency count we have to take the maximum frequency count of any operation which will be considered as the order of the algorithm.
4. The time complexity of linear search in best case is  $O(1)$ .

The time complexity of linear search in worst case is  $O(n)$ .

There are various meaning regarding Big-Oh are shown in below table.

CLASS	NAME	COMMENTS
<b>1</b>	Constant	It is one of the Best Case efficiencies, very few examples can be given since the algorithm running time goes to infinity when the input grows very larger.
<b><math>\log n</math></b>	Logarithmic	A result of cutting a problem's size by a constant factor on each iteration of the algorithm.
<b>n</b>	Linear	An algorithm that scans the list of $n$ elements belongs to this class. Example sequential search.
<b><math>n \log n</math></b>	$n$ -log- $n$	Many divide-and-conquer algorithms including quick sort, merge sort fall under this category.
<b><math>n^2</math></b>	Quadratic	Typically characterize the efficiency of algorithms with two embedded loops. $n$ -by- $n$ matrices are the standard examples.
<b><math>n^3</math></b>	Cubic	Typically characterize the efficiency of algorithms with three embedded loops. Several non trivial algorithms of linear algebra fall under this category.
<b><math>2^n</math></b>	Exponential	For the algorithms that generate all subsets of an $n$ -elements set often the term "exponential" is used to include this and faster orders of

growth as well.

Table 1.4.1: Basic Asymptotic Efficiency Classes

The relationship among these computing time is,

$$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

### EXAMPLE PROBLEM 1

**Find the time complexity of  $f(n) = n^2\log n + n\log n$ .**

Solution:

Here among all frequency counts, i.e.,  $n^2\log n$  and  $n\log n$ . The frequency count  $n^2\log n$  is maximum, so it is the order of algorithm.

□ Time Complexity is  $O(n^2\log n)$ .

### EXAMPLE PROBLEM 3

**Find the time complexity of the following program**

```
for(i=0; i<n; i++)
```

```
{
```

```
    for (k=0; k<n; k++)
```

```
{
```

```
    printf("Rashmi");
```

```
    k = k+1;
```

```
    i = i + 1;
```

```
}
```

```
}
```

```
for(j=0; j<n;j++)
```

```
{
```

```
    Printf("Sai");
```

```
}
```

Solution:

Here, the complexity =  $n^2 + 2n + n$

$$= n^2 + 3n$$

□ Time Complexity is  $O(n^2)$ .

### 1.4.2 Omega Notation ( $\Omega$ )

The Omega Notation ( $\Omega$ ) is used to represent the **lower bound** of algorithm's run time. Using omega notation we can find shortest amount of time taken by the algorithm.

#### Definition:

A function  $f(n)$  is said to be in  $\Omega(g(n))$  if  $f(n)$  is bounded below by some positive constant multiple of  $g(n)$  such that

$$f(n) \geq c * g(n) \quad \forall n, n \geq n_0$$

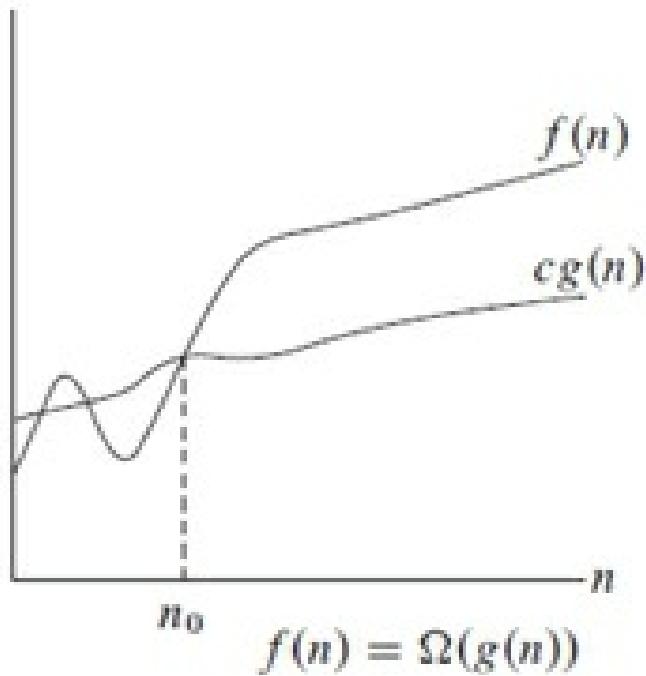


Fig. 1.4.2: Omega Notation

### EXAMPLE PROBLEM 1

Given  $f(n) = 3n + 2$ , then prove that  $f(n) = \Omega(n)$ .

Solution:

$$3n + 2 \geq 3n \quad \forall n \geq 1$$

The above inequality is satisfied according to  $\Omega$  definition by setting,

$$c = 3 \text{ and}$$

$$n_0 = 1.$$

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

### EXAMPLE PROBLEM 2

**Given  $f(n) = 50n^2 + 10n - 5$ , then prove that  $f(n) = \Omega(n^2)$ .**

Solution:

$$50n^2 + 10n - 5 \geq 50n^2 \quad \forall n \geq 1$$

The above inequality is satisfied according to  $\Omega$  definition by setting,

$$c = 50 \text{ and}$$

$$n_0 = 1.$$

$$\square f(n) = \Omega(n^2)$$

### EXAMPLE PROBLEM 3

**Consider  $f(n) = 2n^2 + 5$ , then prove that  $g(n) = 7n$ .**

Solution:

Given  $f(n) = 2n^2 + 5$  and  $g(n) = 7n$

Then if  $n = 0$

$$f(n) = 2(0)^2 + 5 = 5$$

$$g(n) = 7(0) = 0$$

i.e.,  $f(n) > g(n)$

But if  $n = 1$

$$f(n) = 2(1)^2 + 5 = 2 + 5 = 7$$

$$g(n) = 7(1) = 7$$

i.e.,  $f(n) = g(n)$

if  $n = 3$

$$f(n) = 2(3)^2 + 5 = 18 + 5 = 23$$

$$g(n) = 7(3) = 21$$

i.e.,  $f(n) > g(n)$

Thus for  $n > 3$  we get  $f(n) > c * g(n)$ .

It can be represented as

$$2n^2 + 5 \in \Omega(n)$$

Similarly any

$$n^3 \in \Omega(n)^2$$

#### 1.4.2.1 Big Omega Ration Theorem

**THEOREM:**

Let  $f(n)$  and  $g(n)$  be such that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  exist, then function  $f \in \Omega(g)$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$ , including the case in which the limit  $\infty$ .

Proof:

If  $f(n) = \Omega(g(n))$  then,

For all  $n$ ,  $n \geq n_0$ , positive 'c' and ' $n_0$ ' exists such that  $c \leq \frac{f(n)}{g(n)}$

Hence,  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > c$ .

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

Now, suppose  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > c$ , then that means an ' $n_0$ ' exists such that,

$$\max\{1, c\} * g(n) \leq f(n) \quad \forall n, n \geq n_0.$$

#### 1.4.2.2 Little Omega Notation ( $\omega$ )

Little Omega Notation ( $\omega$ ) is used to provide a lower bound which is not asymptotically tight i.e., it provides a strict lower bound. Little omega notation means that  $f(n)$  can never be lesser than  $g(n)$ . That is,

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$$

f(n) is  $\omega(g(n))$  if

We can also say that for all positive constants  $c$  there exists a constant  $n_0 > 0$  such that following is true for  $n \geq n_0$ .

$$0 \leq c * g(n) < f(n)$$

The basic difference is that in big-omega this bound holds for some constant  $c$ , while in small-omega this holds for every constant  $c$ .

### 1.4.3 Theta Notation ( $\Theta$ )

Theta Notation ( $\Theta$ ) denoted as special for functions having the same time complexity for lower and upper bounds.

#### Definition:

Let  $f(n)$  and  $g(n)$  be two non negative functions. There are two positive constants namely  $c_1$  and  $c_2$  such that

$$c_2 g(n) \leq f(n) \leq c_1 g(n)$$

Then we can say that  $f(n) \in \Theta(g(n))$ .

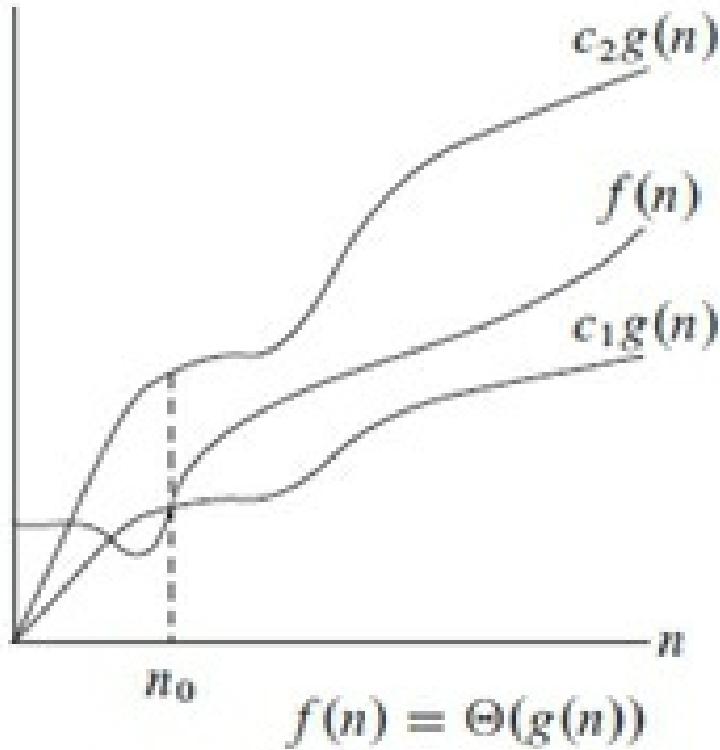


Fig. 1.4.3: Theta Notation

#### 1.4.3.1 Theta Ratio Theorem

Let  $f(n)$  and  $g(n)$  be such that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  exists, then function  $f \in \Theta(g)$  if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ . For some constant  $c$  such that  $0 < c < \infty$ .

#### EXAMPLE PROBLEM 1

**Prove that  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ .**

Solution:

To prove that statement we have to determine the positive constants  $c_1, c_2$  and

$n_0$  such that,  $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2, \forall n \geq n_0$ .

Divide by  $n^2$  throughout,

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

When  $n \geq 1$  and  $c_2 \geq \frac{1}{2}$ , the inequality  $\frac{1}{2} - \frac{3}{n} \leq c_2$ , will hold good.

Similarly, when  $n \geq 7$  and  $c_1 \leq \frac{1}{14}$ , then  $c_1 \leq \frac{1}{2} - \frac{3}{n}$  will hold good.

Here, we have,

$$c_1 = \frac{1}{14}, c_2 = \frac{1}{2} \text{ and } n_0 = 7.$$

$$\therefore f(n) = \Theta(n^2)$$

#### 1.4.4 Little-Oh Notation (o)

As we know,  $O(g(n))$  is the set of functions with

$$f(n) \leq c * g(n).$$

Then  $f(n)$  is big oh if  $g(n)$ .

**Definition:** The  $f$  is little oh of  $g$  as  $n$  approaches to  $n_0$ . We can write it as

$$f(n) = o(g(n)) \text{ where } n \rightarrow n_0$$

That means,

$$\lim_{n \rightarrow n_0} \frac{f(n)}{g(n)} = 0$$

This also implies that  $f(n)$  is much smaller than  $g(n)$  for  $n$  near  $n_0$ .

#### 1.4.5 Order of Growth Using Limits

Let  $f(n)$  and  $g(n)$  be non-negative function on the set of positive integers. We can define asymptotic notations  $O$ ,  $\Omega$ ,  $\Theta$ ,  $\omega$ ,  $o$  using limit theory. Here are the definitions.

1. If  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$ , then  $f(n) = O(g(n))$ , but  $f(n) \neq \Omega(g(n))$ . We say

that “ $f(n)$  grows asymptotically slower than  $g(n)$ ” and write “ $f(n) = o(g(n))$ ”.

2. If  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty$ , then  $f(n) = \Omega(g(n))$ , but  $f(n) \neq O(g(n))$ . We say that “ $f(n)$  grows asymptotically faster than  $g(n)$ ” and write “ $f(n) = \omega(g(n))$ ”.
3. If  $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = c$ , for some constant  $c > 0$ , then  $f(n) = \Theta(g(n))$ , but  $f(n) \neq \Omega(g(n))$ . We say that “ $f(n)$  grows asymptotically at the same rate”.

To compute the order of growth, the limit based approach is often convenient because it is more advantageous to use powerful calculation techniques such as L Hospital’s rule defined by,

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{g'(n)}{f'(n)}$$

and Steriling’s formula which is given by,

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ for very large values of } n.$$

### EXAMPLE PROBLEM 1

**Consider  $f(n) = \log n$ , then prove  $f(n) = o(\sqrt{n})$ .**

Solution:

$$\begin{aligned} f(n) &= \lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} \\ &= \lim_{n \rightarrow \infty} \frac{(\log n)'}{(\sqrt{n})'} \quad \text{by taking derivative} \end{aligned}$$

By applying L Hospital rule,

$$\begin{aligned} &\lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} \\ &= \lim_{n \rightarrow \infty} \frac{1}{\frac{1}{2\sqrt{n}}} \end{aligned}$$

$$= \frac{12\sqrt{n}}{n-1} = \frac{2}{\sqrt{n}}$$

$$= 0$$

Hence it is proved.

### EXAMPLE PROBLEM 2

**Prove that  $f(n) = a_n n^m + \dots + a_1 n + a_0$ , then  $f(n) = O(n^m)$ .**

Solution:

Let,

$$f(n) = a_n n^m + a_{n-1} n^{m-1} + a_{n-2} n^{m-2} \dots + a_1 n + a_0 \quad \dots \quad (1)$$

If we treat  $a_m^i$  as a constant then equation (1) becomes

$$f(n) = A \sum_{i=0}^m a^m \quad \dots \quad (2)$$

where,  $A$  is  $a_0^0 + a_1^1 + a_2^2 + \dots$

If equation (2) is

$$\begin{aligned} f(n) &= A \left[ \sum_{i=0}^n 1 \right] \\ &= A [1 + 1 + 1 + 1 + \dots + 1] \\ &= A(n^1) \end{aligned}$$

If equation (2) is

$$\begin{aligned} f(n) &= A \left[ \sum_{i=0}^n i \right] \\ &= A [1 + 2 + 3 + 4 + \dots + n] \\ &= A(n^2) \\ &= A O(n^2) \end{aligned}$$

If equation (2) is

$$f(n) = A \sum_{i=0}^n n^2 = A O(n^2)$$

$$\text{Thus, } f(n) = A \sum_{i=0}^m n^2 = A O(n^m)$$

Finally after neglecting constant term we will have

$$f(n) = O(n^m)$$

### EXAMPLE PROBLEM 3

**Consider  $f(n) = 2n$ , then prove that  $f(n) = O(n^2)$ .**

Solution:

$$f(n) = \lim_{n \rightarrow \infty} \frac{2n}{n^2}$$

$$= \lim_{y \rightarrow \infty} \frac{2^{\frac{1}{y}}}{\frac{1}{y^2}}$$

$$\begin{aligned} & \lim_{y \rightarrow \infty} 2y \\ &= 0 \end{aligned}$$

$\therefore f(n) = O(n^2)$  as per definition.

### EXAMPLE PROBLEM 4

**Compare the order of growth of  $n!$  and  $2^n$ .**

Solution:

Given that  $f(n) = n!$  and  $g(n) = 2^n$  we can find limit as shown below,

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \frac{n^n}{e^n}}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \frac{n^n}{(2^n e^n)} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \frac{n}{(2e)^n} = \infty$$

As per second definition,  $\frac{f(n)}{g(n)}$ , symbolically it is represented as  $f(n) = n! \in \Omega(2^n)$ .

## **EXAMPLE PROBLEM 5**

**Compare the order of growth of  $\log_2(n)$  and  $\sqrt{n}$ .**

Solution:

For comparison, we will consider various values of n.

If  $n = 2$

$$\log_2(n) = \log_2(2) = 1$$

$$\sqrt{n} = \sqrt{2} = 1.414$$

If  $n = 64$

$$\log_2(64) = \log_2(64) = 6$$

$$\sqrt{64} = 8$$

If  $n = 256$

$$\log_2(256) = \log_2(256) = 8$$

$$\sqrt{256} = 16$$

All these computations shows that

$$\log_2(n) < \sqrt{n}$$

$\therefore$  Hence Proved

## **EXAMPLE PROBLEM 6**

**Compare the order of growth of  $\frac{1}{2}n(n-1)$  and  $n^2$ .**

Solution:

If  $n=2$ ,

$$\frac{1}{2}n(n-1) = \frac{1}{2}*2(2-1) = 1$$

$$n^2 = (2)^2 = 4$$

If  $n=4$ ,

$$\frac{1}{2}n(n-1) = \frac{1}{2}*4(4-1) = 6$$

$$n^2 = (4)^2 = 16$$

If  $n=8$ ,

$$\frac{1}{2}n(n-1) = \frac{1}{2}*8(8-1) = 28$$

$$n^2 = (8)^2 = 64$$

All the above comparisons indicate that

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

## 1.5 WHAT IS PROBABILISTIC ANALYSIS?

Probabilistic analysis of algorithms is an approach to estimate the computational complexity of an algorithm or computational problem. It starts from the assumption about the set of all possible inputs.

For non-probabilistic, more especially, for the deterministic algorithms the most common types of complexity estimates are,

1. The average case complexity (expected time complexity), in which gives an input distribution and the expected time of an algorithm is evaluated.
2. Most of the complexity estimates, for the given input distribution, it is evaluated that the algorithm admits a given complexity estimate that almost surely holds.

In the probabilistic analysis of (probabilistic) randomized algorithms, the distributions or averaging of all the possible choices in randomized steps are also taken into account, in addition to the input distributions.

### 1.5.1 PROBABILISTIC ALGORITHMS

A randomized algorithm or probabilistic algorithm is an algorithm which employs a degree of randomness as a part of logic.

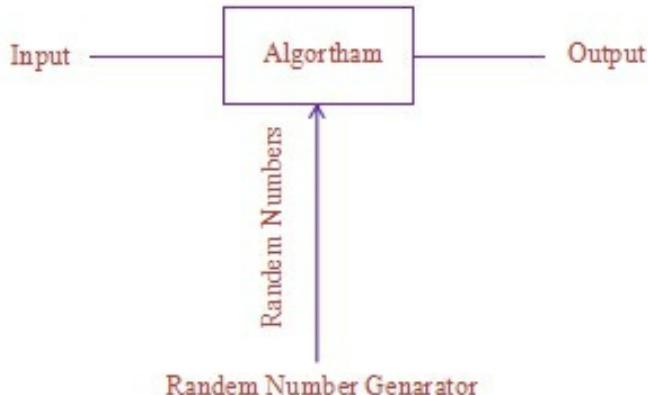


Fig. 1.5.1: Overview of Randomized Algorithms

In probabilistic (randomized) algorithms,

- 1) In addition to input, the algorithm takes a source of random numbers and makes random choices during execution.
- 2) On the fixed input the behaviors of algorithm can vary.

Randomized algorithms make use of a random number generator and some of the decisions made in the algorithm depends on the random number generator. Generally the output of random number generator may vary from run to run. Similarly, the output of the randomized algorithm could also differ from run to run for the same input considered. So, the execution time of the randomized algorithm could also vary from run-to-run for the same input.

Randomized are classified into two classes. They are:

- 1) **Las Vegas Algorithm:** Las Vegas Algorithm always produce the correct answer. The running time is a random variable whose expectation is bounded (say by a polynomial). Execution time of a Las Vegas Algorithm depends on the output of the randomizer. It may terminate fast or run for a long time.
- 2) **Monte Carlo Algorithm:** Monte Carlo Algorithm's output depends upon run-to-run. For a fixed input, a Monte Carlo Algorithm does not display much variation time between runs. Monte Carlo Algorithm runs for a fixed number of steps and produces an answer that is correct with probability  $\geq 1/3$ .

### 1.5.1.1 Scope of Probabilistic Algorithms

The scope of probabilistic algorithms is mentioned below.

- 1) **Data Structures:** Searching, sorting, order statistics, computational geometry.
- 2) **Mathematical Programming:** Faster algorithms for linear programming, Rounding linear programming solutions to integer program solutions.
- 3) **Algebraic Identities:** Interactive proof systems, Polynomial matrix identity verification.
- 4) **Graph Algorithms:** Shortest paths, Minimum spanning trees, minimum cuts.
- 5) **Counting and Enumeration:** Matrix permanent counting combinatorial structures.
- 6) **Parallel and Distributed Computing:** Deadlock avoidance etc.
- 7) **Probabilistic Existence Proofs:** Combinatorial object arises with non-zero probability among objects drawn for suitable probability space.
- 8) **De-randomization:** First after considering randomized algorithms, it can be de-randomized to yield a deterministic algorithm.

#### **1.5.1.2 Advantages and Disadvantages of Probabilistic Algorithms**

Advantages of probabilistic algorithm are shown below.

- 1) Implementation is correct and efficient.
- 2) Many of the randomized algorithms are simpler.
- 3) The probabilistic algorithm provides better complexity bounds.
- 4) The time complexity required for solving elements, which are repeated in identification problems using randomized algorithms is  $O(\log n)$  but the deterministic algorithm is  $\Omega(n)$ .
- 5) The algorithm provide super asymptotic run-time bounds.
- 6) These algorithms have provided to be practically competitive.

Disadvantages of probabilistic algorithms are as follows.

- 1) It is not always possible to yield better performance results even if a randomizer is used with in an algorithm.
- 2) When a faster algorithm is designed with lesser error probability, when compared to the probability of hardware failure, the probability of system malfunctioning is greater than the failure probability of the algorithm.

## **1.6 WHAT IS AMORTIZED ANALYSIS?**

In the analysis of algorithms in computer science, **amortized analysis** finds the average running time per operation over the worst case sequence of operations. Amortized analysis guarantees the time per operation over worst case performance. Here we have to know the cost amortization also.

**Cost Amortization** is an accounting scheme in which separate operations are assigned to different charges (amount) in such a way that the same operation is charged more or less than actual cost of actual operations. It is necessary to ensure that the sum of the amortized cost is greater than or equal to the sum of actual cost.

$$\sum_{1 \leq i \leq n} \text{amortized}(i) \geq \sum_{1 \leq i \leq n} \text{actual}(i)$$

where, *amortized(i)* and *actual(i)*, respectively denote the amortized and actual complexities of the  $i^{\text{th}}$  operation in the sequence of  $n$  operations.

If in case the value of amortized cost is greater than actual cost is greater than its actual cost, then the difference between them is assigned to particular object called as credit. This object can be used when the amortized cost is less than the actual cost.

#### Example:

Consider, the following sequence consisting of insert (I) and delete (D) operations.

I1, I1, I3, D1, I4, I5, D2, I6, I7, D3, I8

The actual cost of eight insert operations is 2 individually and that of D1, D2, D3 is 8, 7, 7 respectively.

∴ The total cost of the sequence =  $2 + 2 + 2 + 8 + 2 + 2 + 7 + 2 + 2 + 7 + 2 = 38$

If the charge of three units is deducted from the D1 operation and transferred to every insert operation executed before D1, then the amortized cost of I1, I2, I3 becomes 3, 3, 3 respectively. Similarly two units are removed from D2, D3 operations then the actual cost of I4, I5, I6, I7 increases by one. Therefore the amortized cost of I4, I5, I6, I7 becomes three and the cost of I8 remains same (i.e., two) and the actual cost of D1, D2, D3 becomes 5 each.

- ∴ The sum of amortized cost =  $3 + 3 + 3 + 5 + 3 + 3 + 5 + 3 + 3 + 5 + 2 = 38$
- ∴ Sum of amortized cost = Sum of actual cost = 38.

Irrespective of operations sequence, if the cost is changed such that the amortized cost if insert is  $\leq 2$  and that of delete is  $\leq 5$ .

$$\Rightarrow \text{Actual cost of interest/delete} \leq (3*i + 5*d) \quad \dots \quad (1)$$

where, i signifies the total number of insert operations and d signifies the total number of delete operations.

If we assume that the actual cost of insert and delete is not greater than 5 and 15 respectively then

$$\text{Actual cost of sequence} \leq (5*i + 15*d) \quad \dots \quad (2)$$

By considering both the above equations (1) and (2), we get,

$$\text{Min}\{(3*i + 5*d), (5*i + 15*d)\}$$

The result generated specifies a higher bounds on the complexity associated with the operations sequence. The amortized time complexity of performing n-operations is  $O(\log n)$ .

The average case analysis and probabilistic analysis are not the same thing as amortized analysis.

- 1) In **average case analysis**, we average the over all possible inputs.
- 2) In **probabilistic analysis**, we average over all possible random choices.
- 3) In **amortized analysis**, we average overall sequence of operations.

There are three most common techniques used in amortized analysis. There is a main difference in the cost assigned to it.

- 1) **Aggregate Method:** The aggregate method has the following characteristics.
  - i. It computes the worst case time  $T(n)$  for a sequence of n operations.
  - ii. The amortized cost is  $T(n)/n$  per operation.
  - iii. It gives the average performance of each operation in the

worst case.

- iv. This method is less precise than other methods , since all operations are assigned the same cost.

**2) Accounting Method:** In this accounting method, we assign charges to different operations, with some operations charged more or less than they actually cost. In other words,we assign the artificial charges to different operations.

- i. Any over charge for an operation on an item is stored (in a bank account) reserved for that item.
- ii. Later, a different operation on that item can pay for its cost with that credit for that item.
- iii. The balance in the (bank) account is not allowed to become negative.
- iv. The sum of the authorized cost for any sequence of operations is an upper bound for the actual total cost of these operations.
- v. The amortized cost of each operaition must be choosen wisely in order to pay for each operation at or before the cost is incurred.

**3) Potential Method:** The method stores for payments as potential or potential energy that can be released to pay for future operations. The stored potential is associated with the entire data structure rather than specific objects with in the data structure.

**Dynamic Table:** If in case the allocated space is not enough, we have to copy the table into larger sized table. Similarly, if larger number of members released from the table it is a good idea to relocate the table with a smaller size. Using amortized analysis we shall show the amortized cost of the insertion and deletion is constant. The unused space in a dynamic table never exceeds a constant fraction of the total space.

Assume that the dynamic table supports the following two operations.

- a) **TABLE\_INSERT:** This operation adds an item into the table by copying it into the unused single slot. The cost of insertion is 1.
- b) **TABLE\_DELETE:** This operation removes an item from table by

freeing a slot. The cost of deletion is 1.

## 1.7 HOW TO SOLVE RECURRENCE RELATIONS?

The recurrence equation is an equation that defines a sequence recursively. It is normally in following form.

$$T(n) = T(n-1) + n \text{ for } n > 0 \quad \dots \quad (1)$$

$$T(0) = 0 \quad \dots \quad (2)$$

Here, equation (1) is called as recurrence relation and equation (2) is called as initial condition. The recurrence equation can have infinite number of sequences. The **general solution** to the recursive function specifies some formula.

For example:

Consider a recurrence relation,

$$f(n) = 2f(n-1) + 1 \text{ for } n > 1$$

$$f(1) = 1$$

Then by solving this recurrence relation we get  $f(n) = 2^n - 1$ . When  $n = 1, 2, 3$  and 4.

### 1.7.1 Solving Recurrence Equations

The recurrence relation can be solved by the following methods. They are:

- 1) Substitution Method
- 2) Master's Method

Let us discuss these methods with some of the suitable examples.

#### 1.7.1.1 Substitution Method

The substitution method is a kind of method in which a guess for the solution is made in it.

There are two methods of substitutions. They are:

1. Forward Substitution
2. Backward Substitution

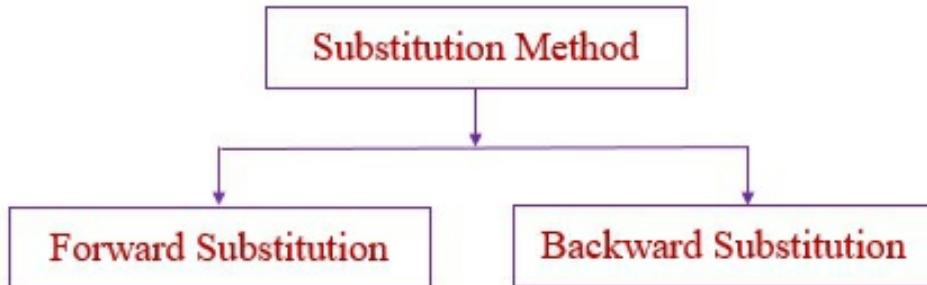


Fig. 1.7.1: Classification of Substitution Method

- 1) **Forward Substitution Method:** This method makes use of an initial condition in the initial term and value for the next term is generated. This process is continued until some formula is guessed. Thus in this kind of substitution method we use recurrence equations to generate the few terms from the recurrence relation.

### EXAMPLE PROBLEM 1

**Consider a recurrence relation**

$$T(n) = T(n-1) + n$$

**With initial condition  $T(0) = 0$ . Solve it using forward substitution method.**

Solution:

Let,

$$T(n) = T(n-1) + n \quad \dots \quad (1)$$

If  $n = 1$  then,

$$\begin{aligned} T(1) &= T(0) + 1 \\ &= 0 + 1 \quad \because \text{Initial condition} \end{aligned}$$

$$\begin{aligned} T(1) &= 1 \quad \dots \\ (2) \end{aligned}$$

If  $n = 2$  then,

$$\begin{aligned} T(2) &= T(1) + 2 \\ &= 1 + 2 \quad \because \text{equation (2)} \end{aligned}$$

$$T(1) = 3$$

...

(3)

If  $n = 3$  then,

$$\begin{aligned} T(3) &= T(3) + 3 \\ &= 3 + 3 \quad \because \text{ equation (3)} \end{aligned}$$

$$\begin{aligned} T(1) &= 6 \quad \dots \\ & \qquad \qquad \qquad (4) \end{aligned}$$

By observing the above generated equation we can derive a formula,

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

We can also denote  $T(n)$  in terms of Big-Oh notation as follows –

$$T(n) = O(n^2)$$

But, it is sometimes difficult to guess the pattern from the forward substitution method. Hence this method is very often used.

- 2) **Backward Substitution Method:** In this method backward values are substituted recursively in order to derive some formula.
- 3) recurrence relation.

## EXAMPLE PROBLEM 1

Solve  $T(n) = T(n-1) + n$  with  $T(0) = 0$

Solution:

Consider the recurrence relation,

$$T(n) = T(n-1) + n \quad \dots \quad (1)$$

With initial condition  $T(0) = 0$

$$T(n-1) = T(n-1-1) + (n-1) + n \quad \dots \quad (2)$$

Putting equation (2) in equation (1) we get,

$$T(n) = T(n-2-1) + (n-2)$$

Putting equation (4) in equation (3) we get,

$$T(n) = T(n-3) + (n-2) + (n-1) + n$$

....

$$= T(n-k) + (n-k-1) + (n-k-2) + \dots + n$$

If  $k = n$  then,

$$T(n) = T(0) + 1 + 2 + 3 + \dots + n$$

$$T(n) = 0 + 1 + 2 + 3 + \dots + n \quad \therefore T(0) = 0$$

$$T(n) =$$

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

Again we can denote  $T(n)$  in terms of Big-Oh notation as,

$$T(n) \in O(n^2)$$

## EXAMPLE PROBLEM 2

$$f(n) = \begin{cases} f(n-1) + 1 & n > 0 \\ 0 & n = 0 \end{cases}$$

**Solve the above recurrence relation.**

Solution:

Let,

$$T(n) = T(n-1) + 1$$

By backward substitution,

$$T(n-1) = T(n-2) + 1$$

$$T(n) = \underbrace{T(n-1)}_{\text{from above}} + 1$$

$$= (T((n-2) + 1)) + 1$$

$$T(n) = T(n-2) + 2$$

$$\text{Again, } T(n-2) = T(n-2-1) + 1$$

$$= T(n-3) + 1$$

$$\therefore T(n) = \underbrace{T(n-2)}_{} + 2$$

$$= (T(n-3)+1) + 2$$

$$T(n) = T(n-3) + 3$$

.....

$$T(n) = T(n-k) + k$$

...  
(1)

If  $k = n$  the equation (1) becomes

$$T(n) = T(0) + n$$

$$= 0 + n \quad \because T(0) = 0$$

$$T(n) = n$$

∴ We can denote  $T(n)$  in terms of Big-Oh notation as

$$T(n) = O(n)$$

### EXAMPLE PROBLEM 3

**Solve the following recurrence relations.**

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

ii.  $T(n) = T\left(\frac{n}{3}\right) + C \quad T(1) = 1$

Solution:

i. Let

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + C \\ &= 2\left(2T\left(\frac{n}{4}\right) + C\right) + C \\ &= 4T\left(\frac{n}{4}\right) + 3C \\ &= 4\left(2T\left(\frac{n}{8}\right) + C\right) + 3C \end{aligned}$$

$$\begin{aligned}
&= 8T(\overline{\frac{n}{8}}) + 7C \\
&= 2^3 T(\overline{\frac{n}{2^3}}) + (2^3 - 1)C \\
&\dots \\
T(n) &= 2^k T(\overline{\frac{n}{2^k}}) + (2^k - 1)C
\end{aligned}$$

If we put  $2^k = n$  then

$$\begin{aligned}
T(n) &= nT(\overline{\frac{n}{n}}) + (n-1)C \\
&= nT(1) + (n-1)C \\
T(n) &= n + (n-1)C \quad \therefore T(1) = 1
\end{aligned}$$

ii. Let,

$$\begin{aligned}
T(n) &= T(\overline{\frac{n}{3}}) + C \\
&= (T(\overline{\frac{n}{3}}) + C) + C \\
&= (T(\overline{\frac{n}{9}}) + C) + C \\
&= T(\overline{\frac{n}{9}}) + 2C \\
&= [(T(\overline{\frac{n}{27}}) + C) + 2C] \\
&= T(\overline{\frac{n}{27}}) + 3C \\
&\dots \\
&= T(\overline{\frac{n}{3^k}}) + kC
\end{aligned}$$

If we put  $3^k = n$  then

$$= T(1) + \log_3 n \cdot C$$

$$\text{So, } T(n) = C \cdot \log_3 n + 1 \quad \because T(1) = 1$$

#### EXAMPLE PROBLEM 4

Find the upper bound of recurrences given below by substitution method.

- i.  $2T(\frac{n}{2}) + n$
- ii.  $T(\frac{n}{2}) + n$

Solution:

i. Let,

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n \\
 &= 2\left(2T\left(\frac{n}{4}\right) + n\right) + n \\
 &= 4T\left(\frac{n}{4}\right) + 2n + n \\
 &= 4T\left(\frac{n}{4}\right) + 3n \\
 &= 4\left(2T\left(\frac{n}{8}\right) + n\right) + 3n \\
 &= 8T\left(\frac{n}{8}\right) + n + 4n + 3n \\
 &= 8T\left(\frac{n}{8}\right) + 7n \\
 &= 2^3 T\left(\frac{n}{2^3}\right) + (2^3 - 1)n \\
 &\dots \\
 &= 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)n \quad \dots \quad (1)
 \end{aligned}$$

If we have  $2^k = n$  then the equation (1) is

$$\begin{aligned} T(n) &= nT\left(\frac{n}{2}\right) + (n-1)n \\ \therefore T(n) &= O(n^2) \text{ Assume } T(1) = 1. \end{aligned}$$

ii. Let,

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \\ &= (2T\left(\frac{n}{4}\right) + 1) + 1 \\ &= T\left(\frac{n}{4}\right) + 2 \\ &= (T\left(\frac{n}{8}\right) + 1) + 2 \\ &= T\left(\frac{n}{8}\right) + 3 \\ &= T\left(\frac{n}{16}\right) + 3 \\ &\quad \dots \\ &= T\left(\frac{n}{2^k}\right) + k \end{aligned} \tag{1}$$

If we assume  $2^k = n$  then equation (2) becomes

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + \log n \quad \because 2^k = n \text{ and } \therefore k = \log n \\ \therefore T(n) &= T(1) + \log n \\ \therefore T(n) &= O(\log n) \end{aligned}$$

### 1.7.1.2 Master's Method

We can solve recurrence relation by using a formula by Master's method.

$$T(n) = aT(n/b) + F(n)$$

Where,  $n \geq d$  and  $d$  is some constant.

Then the Master's theorem can be stated for the efficiency analysis as shown below.

If  $F(n)$  is  $\Theta(n^d)$  where  $d \geq 0$  in the recurrence relation then,

1.  $T(n) = \Theta(n^d)$  if  $a < b^d$
2.  $T(n) = \Theta(n^d \log n)$  if  $a = b$
3.  $T(n) = \Theta(n^{\log_b a})$  if  $a > b^d$

### EXAMPLE PROBLEM 1:

**Solve The following recurrence relation  $T(n) = 4T(n/2) + n$**

Solution:

Consider the mapping this equation with

$$T(n) = aT(n/b) + f(n)$$

Now,  $f(n)$  is  $n$  if  $n^1$ . Hence  $d = 1$ .

$a = 4$  and  $b = 2$  and

$a > b^d$  i.e.,  $4 > 2^1$

By applying Master's Theorem,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) \\ &= \Theta(n^2) \quad \because \log_2 4 = 2 \end{aligned}$$

Hence, the time complexity is  $\Theta(n^2)$ .

For the quick and easy way to calculate the logarithmic values to base 2 the following table can be memorized.

<b>m</b>	<b>k</b>
1	0
2	1

4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10

Table 1.7.1: Logarithmic Values Calculation

$$\log_2 m = k$$

### EXAMPLE PROBLEM 2:

**Solve The following recurrence relation  $T(n) = 9T(n/3) + n$**

Solution:

Consider the mapping this equation with

$$T(n) = aT(n/b) + f(n)$$

Now,  $f(n)$  is  $n$  if  $n^1$ . Hence  $d = 1$ .

$a = 9$  and  $b = 3$  and

$$a > b^d \text{ i.e., } 9 > 3^1$$

Applying Master's Theorem third equality,

$$\begin{aligned} T(n) &= \Theta(n^{\frac{\log_b a}{d}}) = \Theta(n^{\frac{\log_3 9}{1}}) \\ &= \Theta(n^2) \quad \because \log_3 9 = 2 \end{aligned}$$

Hence, the time complexity is  $\Theta(n^2)$ .

### EXAMPLE PROBLEM 3:

**Solve The following recurrence relation  $T(n) = 2T(n/2) + 1$**

Solution:

Consider the mapping this equation with

$$T(n) = aT(n/b) + f(n)$$

Now,  $f(n)$  is 1 if  $n^0$ . Hence  $d = 0$ .

$a = 2$  and  $b = 2$  and

$$a > b^d \text{ i.e., } 2 > 2^0$$

By applying Master's Theorem,

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) \\ &= \Theta(n) \quad \because \log_2 2 = 1 \end{aligned}$$

Hence, the time complexity is  $\Theta(n)$ .

Thus, by using Masters theorem it becomes easy to compute the efficiencies of recurrences.

## 2 DISJOINT SETS, SPANNING TREES, CONNECTED AND BICONNECTED COMPONENTS

### 2.1 INTRODUCTION TO SETS

“A collection of well defined and well distinguished objects” is a formal definition of set. The objects that make up a set are called as the number of elements of the set.

There are standard mathematical symbols used to represent sets are capital letters, A, B, C, P, X, Y etc. The elements of members of sets are denoted by small letters such as a, b, c, x, y etc.

To denote  $x_1$  is an element of the set A, we write  $x_1 \in A$  which we can read it as  $x_1$  belongs to the set A. To denote that  $x_1$  is not an element of set A then we write  $x_1 \notin A$  this is  $x_1$  does not belong to the set A.

#### 2.1.1 Standard Representation of Sets

The most common way of representing of sets are as follows,

1. Tabular Method
2. Set Builder Method

**1) Tabular Method:** In the tabular method, we list all the elements of the set.

**Example:**

- i. A is set of vowels, then we write  $A = \{a, e, i, o, u\}$ .
- ii. The B is set of all positive integers less than 5 as,

$$B = \{1, 2, 3, 4\}.$$

Here, the elements are separated by commas and enclosed by brackets { }. Tabular method is used when the number of set is finite. The tabular method is also referred as Rooster method (It is used when the number of elements in the set is finite) or Enumeration method.

**2) Set Builder Method:** In this set builder method, the elements are not

listed but are indicated by description of their characteristics.

**Example:**

- i.  $A = \{x/x \text{ is a vowel in English alphabet}\}.$
- ii.  $B = \{x/x \text{ is a natural number and } x < 5\}.$

The set builder method is also called as property builder method or rule method.

### 2.1.2 Cardinality of Sets

The cardinality of set A is the number of elements in A and it is denoted by  $|A|$ .

**Example:**

- 1) If  $A = \{a, b, c, d\}$  then  $|A| = 4$ .
- 2) If  $B = \{\text{Red, Blue, Green, Yellow, Pink}\}$  then  $|B| = 5$ .
- 3) If  $C = \{x/x \text{ lies between 0 and 21 and divisible by 4}\}$  then  $|C| = 5$ .

### 2.1.3 Types of Sets

- 1) **Null Set (or) Empty Set:** A set having no element is called as empty set or null set. It is denoted by  $\{\}$  or  $\emptyset$ .

**Example:**

- i.  $A = \{\}$
- ii.  $\{x/x \text{ is an even number not divisible by 2}\}.$
- iii.  $\{x/x \in \mathbb{R} \text{ and } x^4 < 0\}.$

- 2) **Unit Set (or) Singleton Set:** A set consisting of only one element as its member is known as singleton set or unit set.

**Example:**

- i.  $A = \{5\}.$  (since here set A has only one element which is 5).
- ii.  $B = \emptyset.$  (since  $\emptyset$  is an element of set B).

- 3) **Finite (or) Infinite Set:** A set with finite number of elements is called as finite set otherwise called as infinite set.

**Example:**

- i.  $A = \{x/x \text{ is an even positive integer } \leq 50\}$ . (since A is a finite set).
- ii.  $B = \{\emptyset\}$ , (since B is also finite set because it has zero number of elements).
- iii.  $C = \{x/x \text{ is a natural number}\}$  (since C is a finite set).
- iv.  $D = \{x/x \text{ is a positive integer divisible by 6}\}$  (since D is a finite set).

**4) Equal Sets:** If A and B sets such that every element of A is an element of B, and every element of B is also an element of A, then A and B are said to be equal sets.

If A and B are equal then we write ' $A = B$ ', if they are not equal we write ' $A \neq B$ '.

**Example:**

- i.  $A = \{1, 3, 5, 7, 9\}$ ,  $B = \{x/x \in \mathbb{Z}, 1 \leq x \leq 9, x \text{ is odd}\}$  (since  $A = B$ )
- ii.  $C = \{2, 4, 6, 8\}$ ,  $D = \{x/x \in \mathbb{Z}, 1 \leq x \leq 9, x \text{ is odd}\}$  (since  $C \neq D$ )

**5) Equivalent Sets:** Two sets A and B are said to be two equivalent sets (or) similar sets if they have same cardinality.

**Example:**

- i.  $A = \{1, 2, 3, 4\}$ ,  $B = \{a, b, c, d\} \Rightarrow A = B$  as  $|A| = |B|$ .

Where as two infinite sets have the same cardinality if the elements of one can be put in one to one correspondence with elements of the other set.

- ii.  $A = \{a, b, c, \dots\}$ ,  $B = \{b, d, e, \dots\}$ .

**6) Subsets and Supersets:** A set A is called as subset ( $\subseteq$ ) of B if every element of A is also an element of B and then B is called as ( $\supseteq$ ) of A.

If A contains an element which is not in B, then A is not a subset of

B. The statement “A is a subset of B” is symbolically written as “ $A \subseteq B$ ”. Here, the symbol  $\subseteq$  stands for “is a subset of” or “is contained in”.

Similarly, the statement “A is not a subset of B” is symbolically written as  $A \not\subseteq B$ , the symbol  $\not\subseteq$  denoting “is not a subset of” or “not contained in”.

**Example:** Let  $A = \{1, 2, 3\}$ ,  $B = \{1, 2, 3, 4, 5\}$  and  $C = \{2, 3, 5, 6\}$ .

We observe that every element of A is in B, and A contains an element namely 1, which is not in C. Therefore  $A \subseteq B$  and  $A \not\subseteq C$ .

**7) Proper Subset:** Set A is called as proper subset of B, if,

- i. A is a subset of B, and
- ii. B is not a subset of A.

That is, A is to be a proper subset of B, if every element of A belongs to the set B, but there is atleast one element of B, which is not in A. If A is a proper subset of B, then we denote it by  $A \subset B$ .

**Example:** Let  $A = \{1, 2, 3\}$ ,  $B = \{1, 2, 3, 4, 5\}$  and  $C = \{1, 2, 3\}$

We observe that every element of A belongs to the set B, and here (4, 5) are elements of B, but there is atleast one element of B, which is not in A. Hence, A is proper subset of B, then we denote it by  $A \subset B$ .

**8) Comparable Sets:** Two non empty sets A and B are called comparable sets if either  $A \subseteq B$  or  $B \supseteq A$ .

**Example:**

- i.  $A = \{1, 2, 3\}$  and  $B = \{1, 2, 3, 4\}$  are comparable.
- ii.  $A = \{1, 2, 3\}$  and  $B = \{2, 3, 6, 7\}$  are incomparable.

**9) Set of Sets:** Set A is called as set of sets if members present in A are themselves sets.

**Example:**

$$A = \{\emptyset, \{1, 2\}, \{5\}, \{3, 4, 6, 5\}\}.$$

**10) Universal Sets:** Any set which is superset of all the sets under construction is known as the universal set and is denoted by S (or) X (or) U (or) E (or)  $\mu$ .

**Example:** Let  $A = \{1, 2, 3\}$ ,  $B = \{3, 4, 6, 9\}$  and  $C = \{0, 1\}$ .

We can take  $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  as universal set.

**11) Power Set:** The set of all the subsets under consideration is called as power set of A and denoted by  $P(A)$ .

**Example:**  $A = \{1, 2, 3\}$  then its power set is  $P(A) = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \emptyset\}$ .

**12) Multiple Set:** If some elements in a set repeat, it is only one of them which is understood as belonging to the set. Thus if  $A = \{1, 2, 3\}$ ,  $B = \{2, 3, 1, 1, 3\}$  then  $|A| = |B| = 3$ .

It contains certain problems where repetition of an element is of some consequence, we do not make such repetition and call that set as a "Multiple Set".

**Example:** Set of all even integers  $Z_e = \{2x/x \in z\}$ .

## 2.1.4 Operation on Sets

There are some operation performed on sets.

### 2.1.4.1 Union of Two Sets

The union of sets A and B is a set that contains those elements that are in either in A or B or both.

The union of the sets A and B is denoted by  $A \cup B$  and is read as "A union B".

$$A \cup B = \{x/x \in A \text{ or } x \in B\}$$

(or)

$$A \cup B = \{x/x \in A \vee x \in B\}$$

#### 2.1.4.1.1 Union of More Than Two Sets

If  $A_1, A_2, A_3, \dots, A_n$  denotes sets, then the union of these sets denoted by

$\cup_{i=1}^n A_i$  is defined as the set  $\cup_{i=1}^n A_i = \{x/x \in A_i \text{ for atleast one } i\}$ .

#### **2.1.4.2 Intersection of Two Sets**

Intersection of the sets A and B is the set whose elements are all of the elements common to both A and B. The intersection of the sets A and B is denoted by  $A \cap B$ . It is read as A intersection B.

$$A \cap B = \{x/x \in A \text{ and } x \in B\}$$

(or)

$$A \cap B = \{x/x \in A \wedge x \in B\}$$

#### **2.1.4.2.1 Introduction of More Than Two Sets**

If  $A_1, A_2, A_3, \dots, A_n$  denotes sets, then the intersection of these sets denoted by  $\bigcap_{i=1}^n A_i$  is defined as follows

$$\bigcap_{i=1}^n A_i = \{x/x \in A_i: \text{for every } i(i=1, 2, 3, \dots, n)\} = \{x/x \in A_i \forall i=1, 2, 3, \dots, n\}.$$

#### **2.1.4.3 Complement of a Set**

The complement of a set is the set of all those elements which do not belong to the set. In other words, if  $U$  be the universal set and A be any set then the complement of set A is  $U - A$  and is denoted as  $A^c$ ,  $A^c$  or  $\bar{A}$ .

$$\bar{A} = U - A = \{x/x \in U \text{ and } x \notin A\}$$

**Example:**

If  $U = \{1, 2, 6, 9, 10, 18\}$  and  $A = \{2, 6, 18\}$  then  $A^c = \{1, 9, 10\}$ .

#### **2.1.4.3.1 Difference or Relative Complement of Sets**

Given two sets A and B, the set of all elements that belong to B but not in A is called as complement of A relative to B (or relative complement of A in B) and is denoted by  $B - A$ .

**Example:**

If  $A = \{1, 2, 3, 4\}$  and  $B = \{3, 4, 5, 6\}$ , then  $B - A = \{5, 6\}$  and  $A - B = \{1, 2\}$ .

### 2.1.4.3.2 Symmetric Difference of Sets

For two sets A and B, the relative complement of  $A \cap B$  in  $A \cup B$  is called as symmetric difference of A and B and is denoted by  $A \Delta B^{**}$ . Thus,

$$\begin{aligned}A \Delta B &= (A \cup B) - (A \cap B) \\&= \{x/x \in A \cup B \text{ and } x \notin A \cap B\}\end{aligned}$$

**Example:**

If  $A = \{1, 2, 3, 4, 5\}$  and  $B = \{4, 5, 8, 9\}$ , then  $A \Delta B = \{1, 2, 3, 8, 9\}$ .

## 2.1.5 Laws of Set Theory

There are some operations on sets which satisfy certain laws. There following are these laws where A, B, C are subsets of a universal set  $\cup$ .

### 1) Commutative Laws:

- i.  $A \cup B = B \cup A$
- ii.  $A \cap B = B \cap A$

### 2) Associative Laws:

- i.  $A \cup (B \cup C) = (A \cup B) \cup C$
- ii.  $A \cap (B \cap C) = (A \cap B) \cap C$

### 3) Distributive Laws:

- i.  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
- ii.  $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

### 4) Idempotent Laws:

- i.  $A \cup A = A$
- ii.  $A \cap A = A$

### 5) Identity Laws:

- i.  $A \cup \emptyset = A$ .
- ii.  $A \cap U = A$ .

### **6) Double Complement Laws:**

i.  $(A^\complement)^\complement = A$

### **7) Inverse Laws:**

- i.  $A \cup \bar{A} = U$
- ii.  $A \cap \bar{A} = \emptyset$

### **8) Demorgan's Laws:**

- i.  $(A \cup B)^\complement = A^\complement \cap B^\complement$
- ii.  $(A \cap B)^\complement = A^\complement \cup B^\complement$

### **9) Domination Laws:**

- i.  $A \cup U = U$
- ii.  $A^\complement \cap \emptyset = \emptyset$

### **10) Absorption Laws:**

- i.  $A \cup (A \cap B) = A$
- ii.  $A \cap (A \cup B) = A$

## **2.2 DISJOINT SETS**

A disjoint set is a kind of data structure that contains partitioned sets. These partitioned sets are separate non-overlapping sets.

Two sets A and B are disjoint if they have no elements in common, i.e.,  $A \cap B = \emptyset$ . A collection  $S = \{S_i\}$  of non-empty sets forms a partition of a set S if the sets are pair wise disjoint i.e.,  $S_i, S_j \in S$  and  $i \neq j$  imply  $S_i \cap S_j = \emptyset$

$\emptyset$  and their union is S that is,

$$S = \bigcup_{i=1}^n S_i$$

And forms a partition of S if each element of S appears in exactly one  $S_i \in S$ .

### Example:

Consider a set  $S = \{1, 2, 3, 4, \dots, 10\}$ . These elements can be partitioned into three disjoint sets.

$$A = \{1, 2, 3, 6\}, B = \{4, 5, 7\}, C = \{8, 9, 10\}.$$

Each of these sets can be represented as a tree, hence these trees corresponding to sets are,

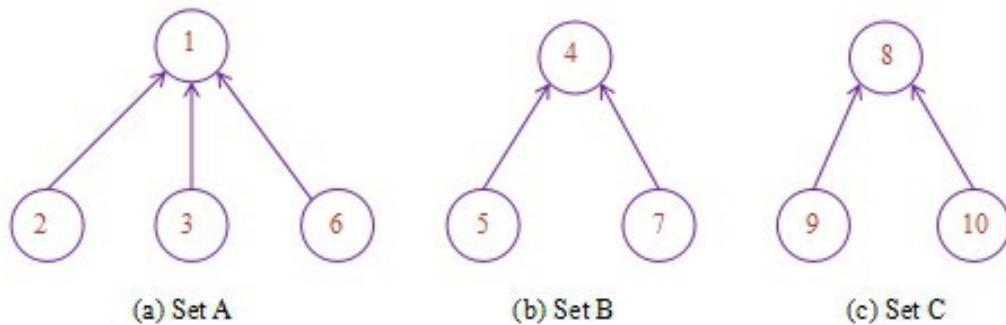


Fig. 2.2.1: Representation of Sets A, B and C

These sets can be represented by storing every element of set in the same array.

The  $i^{\text{th}}$  element of this array represented the tree node that contains element  $i$ . This array element gives the parent pointer of the corresponding tree node.

The figure shown below shows this representation of the sets A, B and C. The root nodes have the parent of -1.

i	1	2	3	4	5	6	7	8	9	10
parent	-1	1	1	-1	4	1	4	-1	8	8

Fig. 2.2.2: Array Representation of the Sets A, B and C

## 2.3 DISJOINT SET OPERATIONS

The operations that can be performed on disjoint sets are as shown below.

1. MIN
2. DELETE
3. FIND
4. UNION
5. INTERSECT

**1) MIN:** Min operation is used to determine minimum set of elements in the set that contains the minimum number of elements.

**Example:**

**2) DELETE:** If  $P = \{1, 2, 3, 4, 5\}$  and  $Q = \{4, 5\}$  are two disjoint sets then  $\min\{P, Q\} = 2$ , since  $P$  has 5 elements and  $Q$  has 2 elements.

**Example:** If  $P = \{2, 5, 6, 7\}$  and  $Q = \{10, 11, 12\}$  are two disjoint sets then to delete the element 6, first we find the set that contains element 6.

$$\text{find}(6) = P$$

Now, delete the element 6 from set  $P$  i.e.,

$$\text{Delete}(6)$$

$\therefore$  The resulting set  $P = \{4, 5, 7\}$

**3) FIND:** Find operation i.e.,  $\text{find}(i)$  is used to find the set to which the element ‘ $i$ ’ belongs to.

**Example:** If  $P = \{1, 2, 3\}$  and  $Q = \{4, 5\}$  are two disjoint sets then,  $\text{find}(4) = Q$ .

**4) UNION:** Union operation is used to combine all element of given sets.

It is represented by ‘ $\cup$ ’ operation.

**Example:**

If  $P = \{1, 2, 3, 4\}$  and  $Q = \{5, 6\}$  are two disjoint sets then,

$$P \cup Q = \{1, 2, 3, 4, 5, 6\}$$

**5) INTERSECT:** Intersection operation is used to determine the common elements of the given two sets. It is represented by ‘ $\cap$ ’ operation.

**Example:**

- i. If  $P = \{2, 3, 4, 5, 6\}$  and  $Q = \{4, 6, 7, 8\}$  then the intersection of  $P$  and  $Q$  i.e.,

$$P \cap Q = \{4, 6\}$$

But if the sets P and Q are two disjoint sets with unique elements then the intersection of P and Q results in an empty set.

ii. If  $P = \{3, 4, 5, 6\}$  and  $Q = \{6, 7\}$  then,

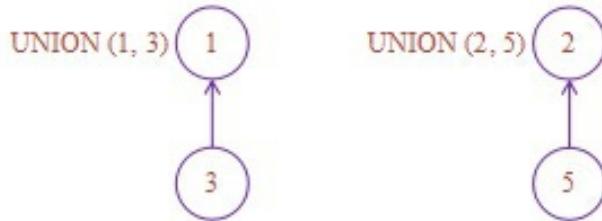
$$P \cap Q = \emptyset$$

## 2.4 UNION AND FIND ALGORITHMS

### 2.4.1 UNION Operation

$\text{UNION}(i, j)$  means the elements of set I and the elements of set j are combined. If we want to represent the UNION operation in the form of a tree, then  $\text{UNION}(i, j)$  where i is the parent and j is the child represented as,

**Example:**



Algorithm for UNION operation:

Algorithm  $\text{UNION}(i, j)$

{

//replace the disjoint sets with roots i and j,  $i \neq j$ , by their union

Integer i, j;

$\text{PARENT}(j) \leftarrow i;$

}

**Example:**

Implement the following sequence of operations.

**UNION(1, 3), UNION(2, 5), UNION(1, 2).**

Step 1: Initially the parent array contains zeros.

**PARENT**

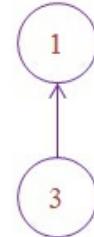
0	0	0	0	0	0
1	2	3	4	5	6

Step 2: After performing UNION(1, 3) operation,

**PARENT**

**PARENT(3)→1**

0	0	1	0	0	0
1	2	3	4	5	6

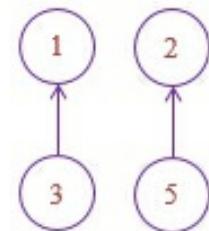


Step 3: After performing UNION(2,5) operation,

**PARENT**

**PARENT(5)→2**

0	1	1	0	2	0
1	2	3	4	5	6

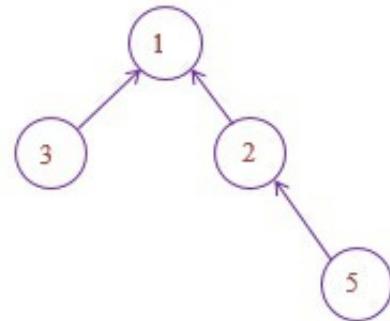


Step 4: After performing UNION(1,3) operation,

**PARENT**

**PARENT(5)→2**

0	1	1	0	2	0
1	2	3	4	5	6



**Time Complexity:** Since the time taken for a UNION is constant, all n- 1 unions can be processed in time O(n).

#### 2.4.2 FIND Operation

FIND( $i^{\text{th}}$ ) implies that it finds the root node if  $i^{\text{th}}$  node, in other words, it returns the name of the set  $i$ .

##### Example:

Consider the UNION(1, 3)



$\text{FIND}(1) = 1$

$\text{FIND}(3) = 1$  since its parent is 1 (i.e., its root node is 1)

Algorithm for FIND operation:

Algorithm FIND(i)

{

//find the root of the tree containing elements

Integer i, j;

$j \leftarrow i$

while  $\text{PARENT}(i) > 0$  do

$j \leftarrow \text{PARENT}(j)$

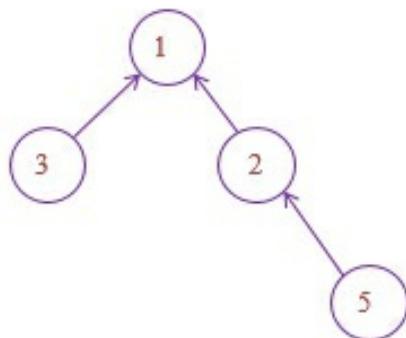
repeat

return (j)

}

Let us explore this algorithm with the previous example.

**Example:**



PARENT						
0	1	1	0	2	0	
1	2	3	4	5	6	
Array Representation						

$\text{FIND}(5), i = 5, j \leftarrow i, j = 5.$

While( $P(j) > 0$ ) do condition is true  
 $j \leftarrow P(j)$  i.e.,  $j = 2$   
again ( $P(2) > 0$ ) do condition is true ( $1 > 0$ )  
So  $j \leftarrow P(2)$  i.e.,  $j = 1$   
while( $P(1) > 0$ ) do condition is false  
return( $j$ ), so 1 is the root node of node 5.

We can observe the same thing from the above FIND.

### Time Complexity:

Each FIND requires following a chain of PARENT links from node 1 to the root. The time required for processing a FIND for an element at level 1 of a tree is  $O(1)$ . Hence, the total time needed to process the  $n-2$  finds is  $O(n^2)$ .

#### 2.4.3 Waiting Rule for UNION (i, j)

If the number of nodes in the tree  $i$  is less than the number in tree  $j$ , then make  $j$  as the parent of  $i$ , as the parent of  $j$ . Both the arguments of UNION must be roots.

A more sophisticated UNION algorithm using weighting rule is shown below.

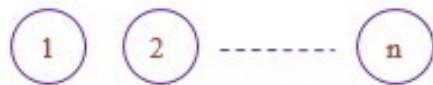
```
Algorithm UNION (i, j)
{
    //Union sets with roots i and j,  $i \neq j$ , using the weighting rule
    //PARENT(i) = -COUNT(i) and PARENT(j) = -COUNT(j)
    Integer i, j, x;
    x  $\leftarrow$  PARENT(i) + PARENT(j), then
    if(PARENT(i) > PARENT(j) then
    {
        PARENT(i)  $\leftarrow$  j      //i has fewer nodes
        PARENT(j)  $\leftarrow$  x
    }
}
```

```

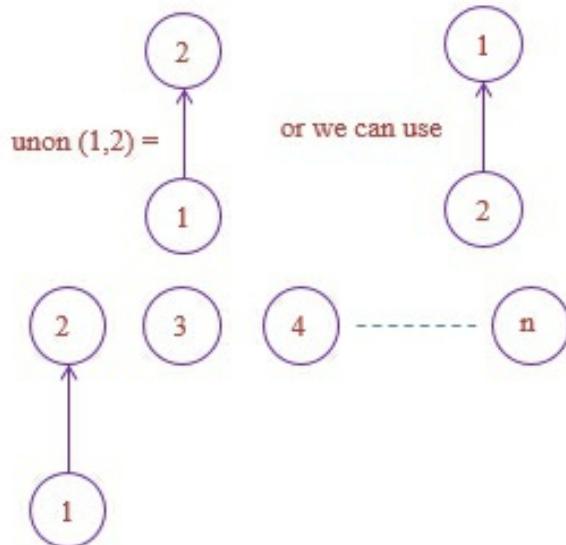
else
{
    PARENT(j) ← i      //j has fewer nodes
    PARENT(i) ← x
}
}

```

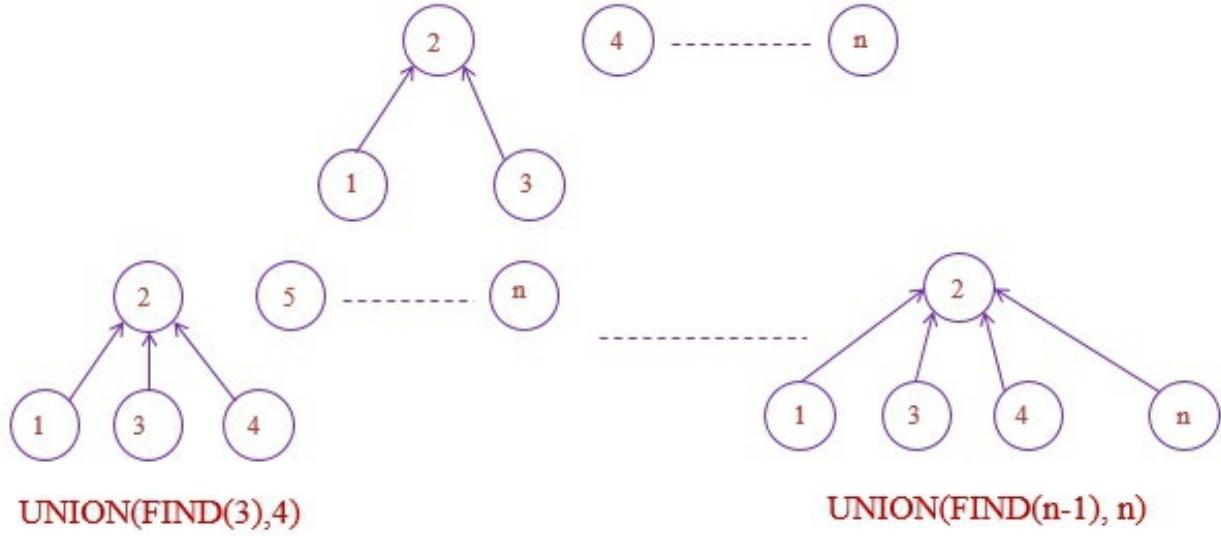
When we use the weighting rule to perform the sequence of set unions given before we obtain the tree as,



$\text{UNION}(1, 2)$ , the number of nodes in tree 1 equal to the number of nodes in tree 2.



$\text{UNION}(2, 3)$ , the number of nodes in tree 2 is 2, number of nodes in tree 3 is 1. So,  $2 > 1$  make parent as 2 and 3 as child.



#### 2.4.4 Collapsing Rule for FIND (i)

If *j* is a node on the path from *i* to its root then set  $\text{PARENT}(j) \leftarrow \text{root}(i)$ .

A more sophisticated algorithm of FIND using collapsing rule is given below,

Algorithm CollapsedFIND(*i*)

{

//Find the root of the tree containing element *i*.

//Use the collapsing rule to collapse all nodes from *i* to the root *j*

*j*  $\leftarrow$  *i*

while( $\text{PARENT}(j) > 0$ ) do //find root

{

*j*  $\leftarrow$   $\text{PARENT}(j)$

}

*k*  $\leftarrow$  *i*

    while( $k \neq j$ ) //collapse nodes from *i* to root *j*

{

*j*  $\leftarrow \text{PARENT}(k)$

$\text{PARENT}(k) \leftarrow j$

```

k ← 1
}
return(j)
}

```

**Example 1:**

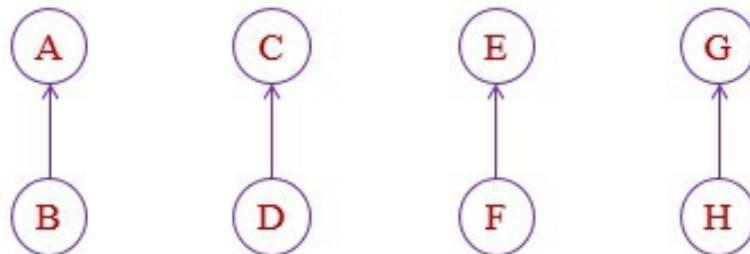
Implement the behavior of weighted unions on the following sequence of unions starting from the height -1.

**UNION(1, 2), UNION(3, 4), UNION(5, 6), UNION(7, 8), UNION(1, 3), UNION(5, 7), UNION(1, 5).**

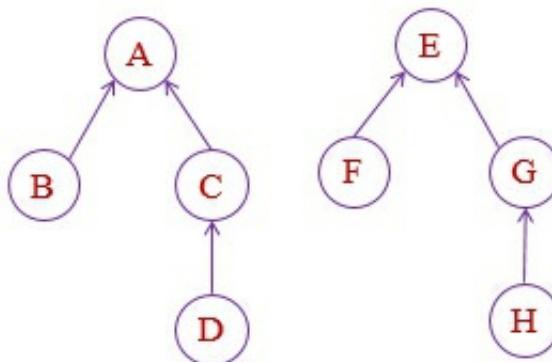
The initial height -1 trees are, shown below.



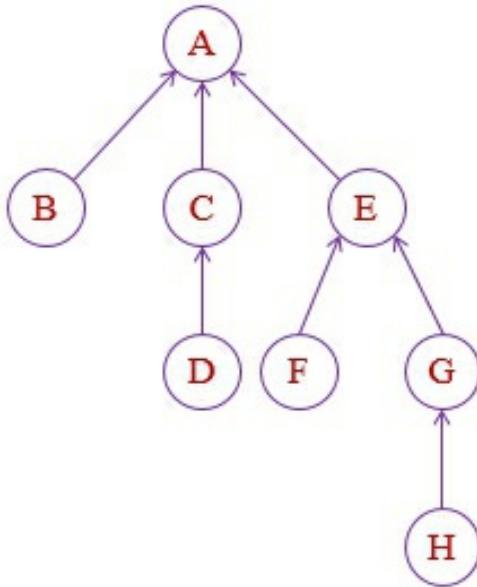
For UNION(A, B), UNION(C, D), UNION(E, F), UNION(G, H), UNION(A, C), UNION(E, G) and UNION(A, E) trees height is -2.



For UNION(A, C) and UNION(E, G) trees have height -3.



For UNION(A, E) tree have height -4.



### Example 2:

Consider the tree which is created as shown below. Now process the following 8 FIND operations.

**FIND(8), FIND(8), FIND(8), FIND(8), FIND(8), FIND(8), FIND(8), FIND(8).**

Find(8) = root node i.e., 1. By using the FIND algorithm, it will require 3 moves upward to reach the root node. Similarly for 8 FIND operations, it will require  $8 \times 3 = 24$  moves. If we use FIND algorithm with collapsing rule, the first FIND(8) requires going up 3 link fields (moves) and then resetting 3 links. Each of remaining 7 fields requires going up only 1 move (link field), the total cost is now only 13 moves.

## 2.5 TREES

A tree is defined as a collection of nodes (elements) which can be empty (or) has a node designed as root (representing element at the top of the hierarchy) from which zero or more subtrees and branches.

Basic tree data structure specifies the data information and have links to other data items. Each data item present in the tree is called as node. The height of the node  $n_i$ , is the longest path from the  $n_i$  to the leaf. In a given tree, the degree of a node is the number of subtrees of a node.

The root is a node at the top most of the tree or node that has no parent is called as the root. Leaf is the node that has no child. Leaf node is also called

as terminal node.

### 2.5.1 Tree Traversal

Traversal is one of the most important operation performed in a tree. Traversing is the process of visiting each vertex of a tree in a specific order exactly once.

We can use tree to store information in a computer. Therefore, some procedures are required for accessing the information easily.

DFS and BFS provides way to walk a tree, that is traverse a tree in specific way, so that each vertex is visited exactly once. There are three traversal methods. They are:

- i. Preorder Traversal
- ii. Inorder Traversal
- iii. Postorder Traversal

#### 2.5.1.1 Preorder Traversal

If  $T$  is the ordered rooted tree with root  $R$  and if  $T$  consists of only  $R$  id the preorder traversal of  $T$ . Suppose,  $T_1, T_2, T_3, \dots, T_n$  are subtrees of  $R$  from left to right. The preorder traversal begins by visiting  $R$ , continues by traversing  $T_1$  in preorder, then  $T_2$  and so on until  $T_n$  is traversed.

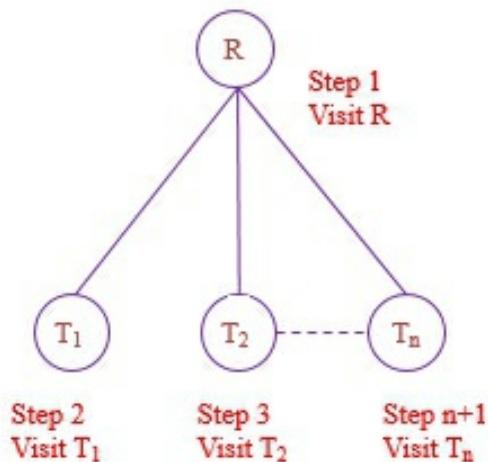


Fig. 2.5.1: Preorder Traversal

Algorithm for preorder traversal:

```

Algorithm Preorder(T)
{
if(T ≠ NULL)
{
    Visit(T);
    Preorder(T → left);
    Preorder(T → right);
}
}

```

In preorder traversal, we visit root, visit subtrees left to right in preorder.

### **Examples:**

Consider the following tree and find the preorder traversal.

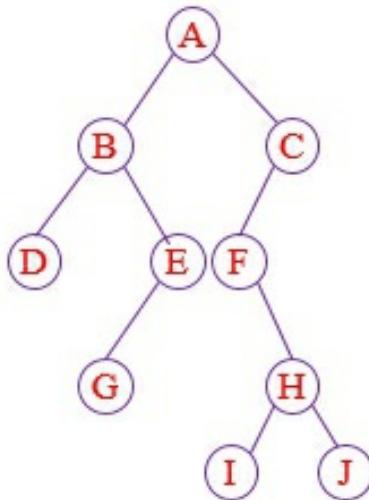


Fig. 2.5.2: Preorder Tree Traversal Tree

**Preorder:** A-B-D-E-G-C-F-H-I-J.

### **2.5.1.2 Inorder Traversal**

Let R be the root of the tree T. If T consists of only R, then R is the inorder traversal of T. Suppose  $T_1, T_2, T_3, \dots, T_n$  are subtrees of R from left to right.

Inorder traversal begins by traversing  $T_1$  in inorder, then visit R. It continues by traversing  $T_2$  in inorder and finally  $T_n$  in order.

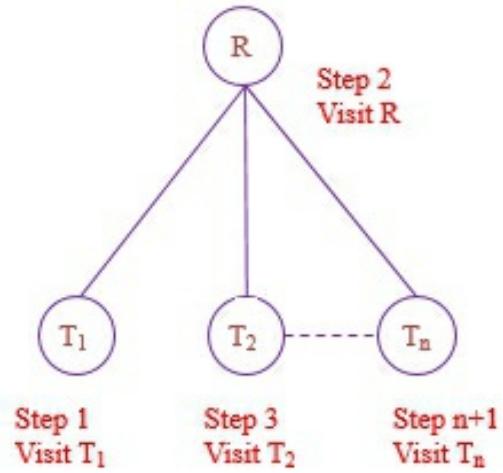


Fig. 2.5.3: Inorder Traversal

Algorithm for inorder traversal:

```
Algorithm Inorder(T)
{
    if(T ≠ NULL)
    {
        Inorder(T → left);
        Visit(T);
        Inorder(T → right);
    }
}
```

**Example:**

Consider the following tree and find the inorder traversal.

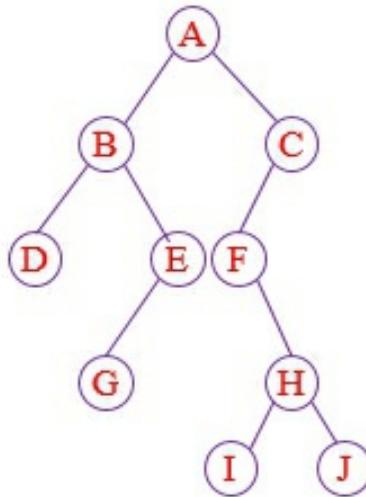


Fig. 2.5.4: Inorder Traversal Tree

**Inorder:** D-B-G-E-A-F-I-H-J-C.

### 2.5.1.3 Postorder Traversal

Let T be the ordered rooted tree with root R. If T contains only R, then R is the postorder traversal of T. Suppose,  $T_1, T_2, T_3, \dots, T_n$  are the sub trees of root R from left to right. The postorder traversal begins by traversing  $T_1$  in postorder then  $T_2$  and finally  $T_n$  in postorder and ends by visiting R.

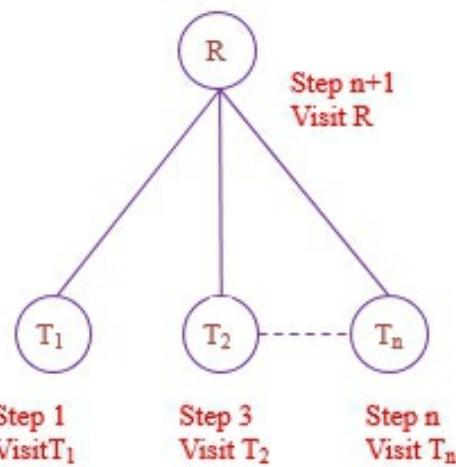


Fig. 2.5.5: Postorder Traversasal

Algorithm for postorder traversal:

Algorithm Postorder(T)

```

{
if(T ≠ NULL)
{
Postorder(T → left);
Postorder(T → right);
Visit(T);
}
}

```

**Example:**

Consider the following tree and find the postorder traversal.

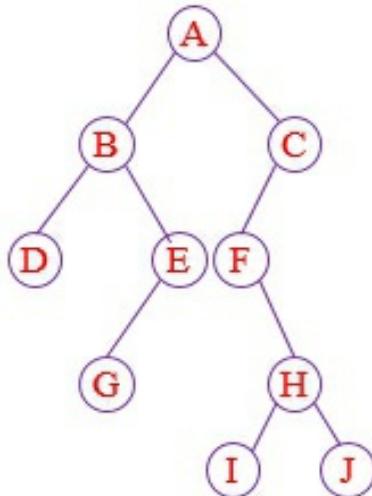


Fig. 2.5.6: Postorder Traversal Tree

**Postorder:** D-G-E-B-I-J-H-F-C-A.

## 2.6 GRAPHS AND ITS REPRESENTATION

A graph  $G$  is a pair of two sets  $V$  and  $E$  where  $V$  is the set of vertices called as nodes and  $E$  is the collection of edges, where edge is an arc which connects two nodes. Each edge is a pair  $(v, w)$  where  $v, w$  belongs to  $V$ .

A graph  $G = (V, E)$  consists of  $V(G) = (V_0, V_1, V_2, \dots, V_n)$  are set of vertices,  $E(G) = (E_1, E_2, E_3, \dots, E_n)$  are set of edges.

### 2.6.1 Basic Definitions

**1) Undirected Graph:** A graph  $G = (V, E)$ , which has unordered pair of vertices called as undirected graph. Consider there is an edge between  $V_1$  and  $V_2$ , then it can be represented as  $(V_1, V_2)$  or  $(V_2, V_1)$  also. The graph shown below is an undirected graph which has 4 nodes and 6 edges.

$$V(G) = \{A, B, C, D\}$$

$$E(G) = \{(A, B), (A, C), (A, D), (B, C), (B, D), (C, D)\}$$

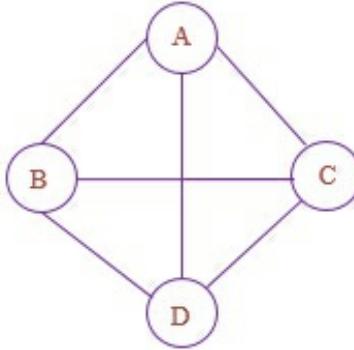


Fig. 2.6.1: Undirected Graph

**2) Directed Graph:** a directed graph is a graph which has ordered pair of vertices that is denoted as  $(V_0, V_1)$  where  $V_0$  is the tail and  $V_1$  is the head of the edge.

In this directed graph, each edge has directions which means,  $(V_0, V_1)$  and  $(V_1, V_0)$  will represent different edges. Hence if there is an edge, a direction will be associated with that edge. Directed graph is also known as a diagraph. The below shown graph is a directed graph which has 5 nodes and 6 edges and they are shown below.

$$V(G) = \{A, B, C, D, E\}$$

$$E(G) = \{(A, B), (B, C), (C, D), (D, A), (D, E), (E, A)\}$$

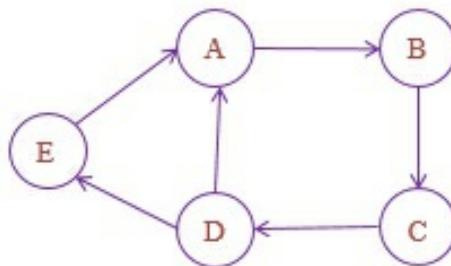


Fig. 2.6.2: Directed Graph

- 3) **Weighted Graph:** A graph  $(V, E)$  is said to be a weighted graph if its edges have been assigned some non-negative values as weight. It is also known as network. The example for a weighted graph is shown below. It has 4 nodes and 6 edges.

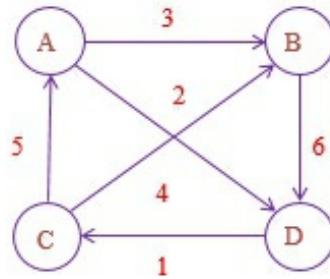


Fig. 2.6.3: Weighted Graph

- 4) **Adjacent Nodes:** A node is adjacent to another node  $V$  if there is an edge from node  $U$  to node  $V$ . In undirected graph, if  $(V_1, V_2)$  is an edge, then it is adjacent to  $V_2$  and is adjacent to  $V_1$ . In a diagram, if  $(V_1, V_2)$  is an edge then, in a directed graph  $V_1$  is adjacent to  $V_2$  and  $V_2$  is adjacent to  $V_1$ .
- 5) **Degree:** In an undirected graph, the degree of node is the number of edges connected to that node.

In directed graph, there are two types of degrees for every node. They are:

- i. **Indegree:** The number of edges coming to a node is the indegree of that node.
- ii. **Outdegree:** The number of edges going outside from a node is the outdegree of that node.

Consider the following directed graph,

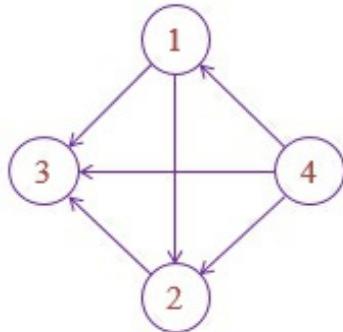


Fig. 2.6.4: Graph

- 6) **Source:** The indegree of a source is zero or it has no incoming edges, but has outgoing edges is referred as source. In the above graph D is source.
- 7) **Successor and Predecessor:** In a directed graph, if a node  $V_1$  is adjacent to node  $V_2$ , then  $V_1$  is the predecessor of  $V_2$  is the successor of  $V_1$ .

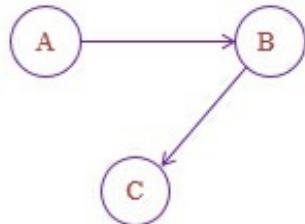


Fig. 2.6.5: Graph G

From the above graph,

- i. Node A is the predecessor of node B.
- ii. Node B is the predecessor of node C.
- iii. Node B is the successor of node A.
- iv. Node C is the successor of node B.

- 8) **Acyclic Graph:** If the edges in the graph does not forms a cycle, then such types of graph is called as acyclic graph.



Fig. 2.6.6: Acyclic Graph

- 9) **Directed Acyclic Graph:** If a directed graph does not form cycles, then such types of graph is called as directed acyclic graph.

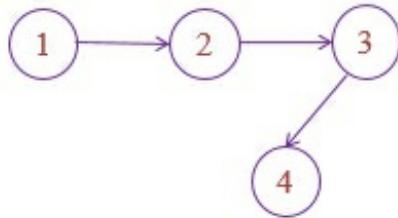


Fig. 2.6.7 Directed Acyclic Graph

**10) Cyclic Graph:** If the edges in the graph does not forms a cycle, then such types of graph is called as cyclic graph.

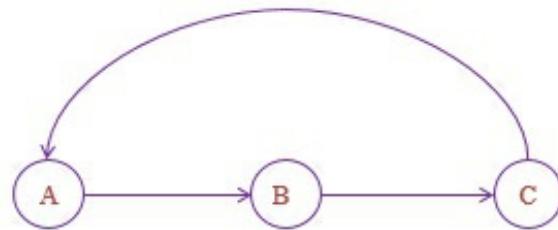


Fig. 2.6.8: Cyclic Graph

In the above graph, the cycle path is (A – B – C – A)

## 2.6.2 Graph Traversals

Traversing and searching every edge and vertex in the graph is one of the most fundamental graph problem. The process of traversing or moving through all vertices in the graph is called as Graph Traversal. Traversing a graph means visiting all the nodes in the graph.

The following are the applications of graph traversal.

- 1) Counting the number of edges.
- 2) Identifying the connected components of a graph.
- 3) Printing the content of each vertex and edge.

There are two standard graph search and graph traversal techniques namely, Breadth First Search (BFS) and Depth First Search (DFS).

### 2.6.2.1 Breadth First Search (BFS)

Breadth First Search (BFS) algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of breadth first search. BFS visits the nodes level-by-level. i.e., root node is visited first, in next step all the adjacent nodes are visited next and then further levels are visited in level order.

Example:

Let us consider a graph G.

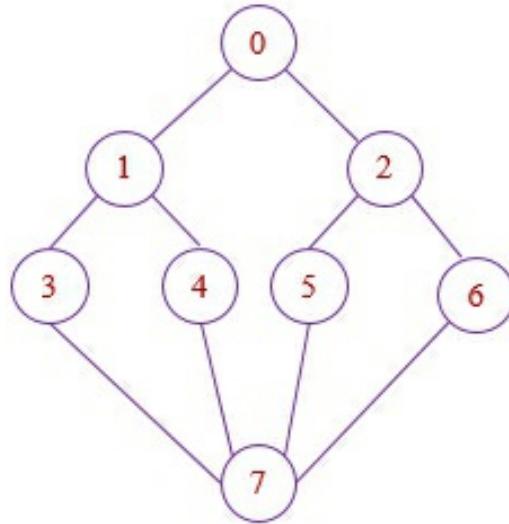


Fig. 2.6.9: Graph G

Let us consider '0' as starting vertex.

Step 1:

	0	1	2	3	4	5	6	7
Visited	FALSE							
<hr/>								
Queue	<hr/>							

Visit vertex '0' first, then move the vertex into the queue and check whether queue is empty or not. You can delete element in queue if queue is not empty. Then, insert all adjacent vertex to delete vertex which are not visited. Make their visit value to true if a vertex is visited.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
<hr/>								
Queue	1   2   <hr/>							

Output : 0

Step 2:

Since the queue is not empty, delete the front element. Insert all the adjacent vertex that are not visited, into queue and make their visit to ‘TRUE’.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
Queue	2	3	4					
						Output : 0,1		

### Step 3:

Now check weather queue is empty or not. If queue is not empty delete the front element and output it i.e., 2. Then insert all adjacent vertex to the deleted node and make their visits to true.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
Queue	3	4	5	6				
						Output : 0,1,2		

### Step 4:

Delete the front element i.e., 3 and output the same. Insert all non-visited adjacent vertex to ‘3’ into queue and make it visited i.e., TRUE.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Queue	4	5	6	7				
						Output : 0,1,2,3		

### Step 5:

Delete the front element i.e., 4 and output it. Here the no adjacent node of ‘4’ with status not visited. Here, nothing is inserted into the queue.



Step 6:

Delete the front element i.e., 5 and output it.



Step 7:

Delete the front element i.e., 6 and output it.



Step 8:

Delete the front element i.e., 7 and output it.



Step 9:

Now, queue is empty and stop the process at this step.

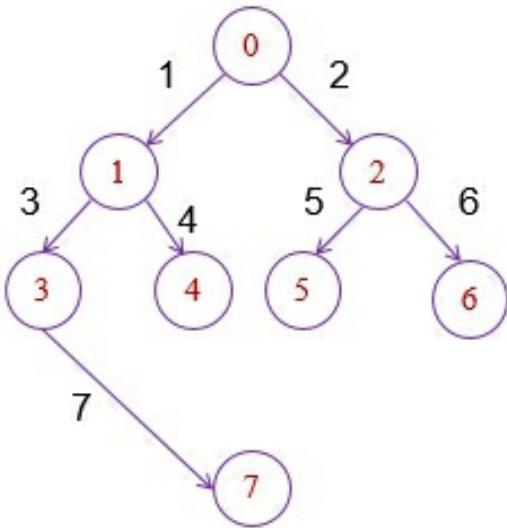


Fig. 2.6.10: BFS Order

### 2.6.2.2 Depth First Search (DFS) and Depth First Traversal (DFT)

Depth first search (DFS) is useful for performing a number of computations on graphs, including finding a path from one vertex to another. It also helps in determining or not a graph is connected and computing a spanning tree of a connected graph.

DFS is used for traversing a graph in such a way that it tries to go far from the root node. DFS implementation uses the stack. DFS starts traversing from one vertex i.e., start vertex  $v$ . Next, an unvisited vertex  $w$  adjacent to  $v$  is selected and depth first search from  $w$  is initiated. When a vertex  $u$  is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex  $w$  adjacent to it and initiated a depth first search from  $w$ .

Example:

Let us consider the graph  $G$ .

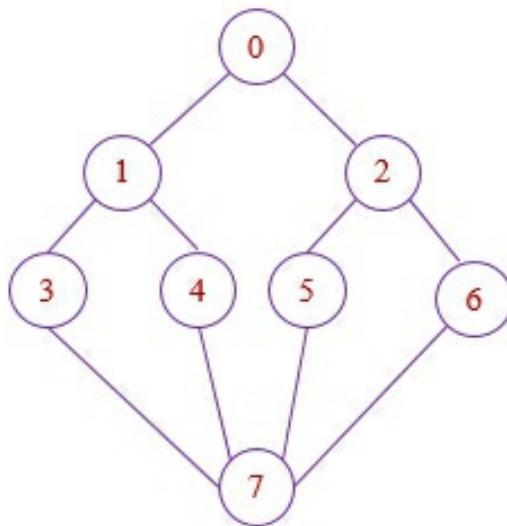


Fig. 2.6.11: Graph G

Step 1:

Let us consider '0' as the starting vertex. Initially, make visited of all vertices as 'FALSE' and consider an empty stack.

	0	1	2	3	4	5	6	7
Visited	FALSE							
Stack								

Step 2:

Now,  $\text{visited}[0] = \text{TRUE}$  and push all the adjacent vertices of '0' into the stack.

	0	1	2	3	4	5	6	7
Visited	TRUE	FALSE						
Stack	2   1							

↑  
top

**OUTPUT : 0**

### Step 3:

Now, pop the top element and call DFS(1) and make visited[1] as TRUE and also output 1. Then, push all the adjacent vertex of 1 into the stack.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
Stack	2	4	3					

↑  
top

**OUTPUT : 0 ,1**

### Step 4:

Now, pop the top element from the stack i.e., 3 and make visited[3] to TRUE and output the same. Then, push all adjacent vertex of '3' that are not visited into the stack.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
Stack	2	4	7					

↑  
top

**OUTPUT : 0, 1, 3**

### Step 5:

Now, pop the top element from the stack i.e., '7' and make visited[7] to TRUE and output the same. Then, push all adjacent vertex of '7' that are not visited into the stack.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE
Stack	2	4	6	5	4			
top								

OUTPUT : 0, 1, 3, 7

### Step 6:

Pop the element from stack i.e., 4 and make visited[4] as TRUE and output the vertex and insert the adjacent vertex of '4' into the stack. Here, adjacent non-visited vertex is none, so nothing was pushed into the stack.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
Stack	2	4	6	5				
top								

OUTPUT : 0, 1, 3, 7, 4

### Step 7:

Pop the element from the stack i.e., 5 and make visited[5] as TRUE and output the vertex.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE
Stack	2	4	6	2				
top								

OUTPUT : 0, 1, 3, 7, 4, 5

### Step 8:

Pop the element from the stack i.e., 2 and make visited[2] as TRUE and output the vertex. Then, insert all adjacent vertices of '2' which are not

visited.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE
Stack	2	4	6	6				

↑  
top

OUTPUT : 0, 1, 3, 7, 4, 5

### Step 9:

Pop the element from the top of the stack i.e., 6 which is not visited and make it as TRUE and output it. Finally there are no more non visited vertex was there in the graph. Hence, stop the process here.

	0	1	2	3	4	5	6	7
Visited	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE
Stack	2	4	6					

↑  
top

OUTPUT : 0, 1, 3, 7, 4, 5, 2, 6

Elements present in the stack are already visited so pop them from the top of the stack.

The final DFS traversal is shown below.

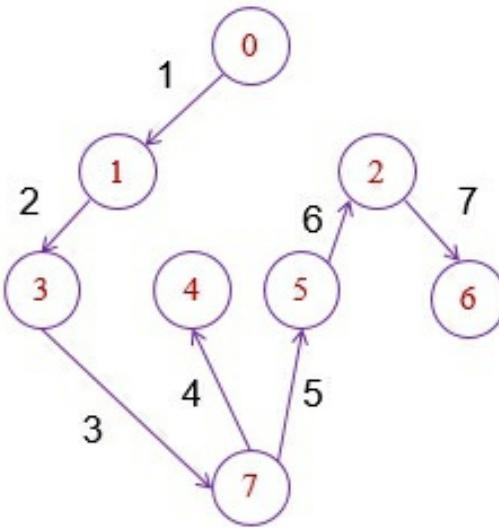


Fig. 2.6.12: DFS Traversal

### 2.6.2.3 DFS and BFS Comparisons

The comparision between Depth First Search (DFS) and Breadth First Search (BFS) are tabulated as follows,

	<b>DFS</b>	<b>BFS</b>
<b>Method</b>	The DFS algorithm explores each possible path to its conclusion before another path tried.	BFS is exactly opposite of DFS. In this method, each node on the same level is checked before the search proceeds to the next level.
<b>Data Structures Used</b>	DFS uses Stack (FIFO) to perform its operations.	BFS uses Queues (FIFO) to perform its operations.
<b>Types of Edges</b>	Trees and back edges.	Tree and cross edges.
<b>Effeciency of Adjacency Matrix</b>	$\Theta( V ^2)$	$\Theta( V ^2)$
<b>Effeciency of Adjacency Linked Lists</b>	$\Theta( V  +  E )$	$\Theta( V  +  E )$
<b>Applications</b>	1) Connectivity	1) Connectivity

2) Acyclicity

3) Articulation Points

2) Acyclicity

3) Minimum Edge Paths

Table 2.6.1: Comparision between BFS and DFS

## 2.7 SPANNING TREES

A spanning tree for a connected for a connected, undirected graph  $G = (V, E)$  is a subgroup of  $G$  that is an undirected tree and contains all the vertices of ‘ $G$ ’. We can also define a spanning tree of a graph as, “A subgraph tree of a graph should include all the vertices and a subset of edges ( $E$ )”.

The BFS and the DFS algorithms are applied to test weather a graph  $G$  is connected or not and to obtain the connected components of  $G$ . Consider the set of all edges  $(u, w)$ , where all vertices  $w$  are adjacent to  $u$  and are not visited.

Consider the graph as shown in the below figure.

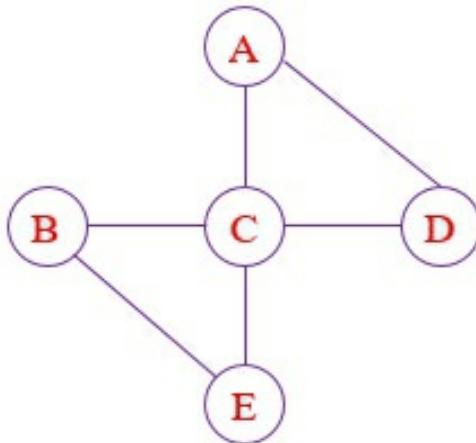


Fig. 2.7.1: A Graph

There are two types of spanning trees. The spanning trees obtained using depth first search are called as depth first spanning trees. The spanning trees obtained using breadth first search are called as breadth first spanning trees.

The following are DFS and BFS spanning trees of the above figure.

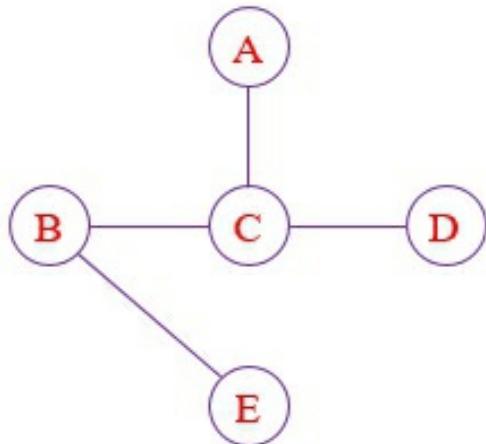


Fig. 2.7.2: DFS Spanning Tree

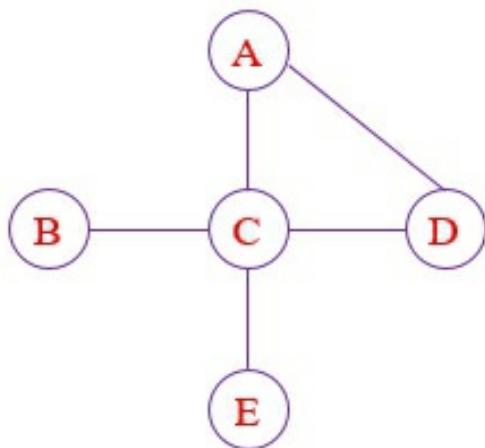


Fig. 2.7.3: BFS Spanning Tree

## 2.8 CONNECTED COMPONENTS

For every pair of its vertices  $u$  and  $v$  if there is a path from  $u$  to  $v$ , then the graph is said to be connected. If a graph is not connected such a model will consist of several connected pieces that are called connected components of the graph i.e; a connected component is the maximum graph of a given graph.

In depth first search , whenever a new unvisited vertex is reached for the first time it is attached as a child to the vertex from which it is being reached. Then, such an edge is called a tree edge.

The DFS algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e, its parent in the tree). Such an edge is called a back edge. A vertex of a connected graph is said to

be its articulation point if its removal with all edges incident to it breaks the graph into disjoint pieces.

Elementary applications of DFS are checking connectivity and checking acyclicity of a graph. Since a DFS halts after visiting all the vertices connected by a path to the starting vertex, checking graph connectivity can be done as follows,

- i. Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts whether all the graph vertices will have been visited.
- ii. If they have, the graph is connected, otherwise, it is not connected.

BFS can be used to check connectivity and acyclicity of a graph, essentially in the same manner as DFS can. If G is a connected undirected graph, then all vertices of G will get visited on the first call to BFS. If G is not connected, then atleast two calls to BFS will be needed. Hence BFS can be used to determine whether G is Connected. The connected component of a graph can be obtained using BST.

For this BFS can be modified so that all newly visited vertices are put onto a list. Then, the sub graph formed by the vertices on the list make up a connected component.

**Example:** Consider the following graph,

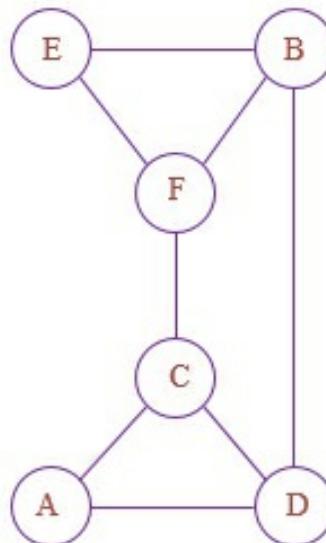


Fig. 2.8.1: Grpah G

The DFS for the graph is as shown below.

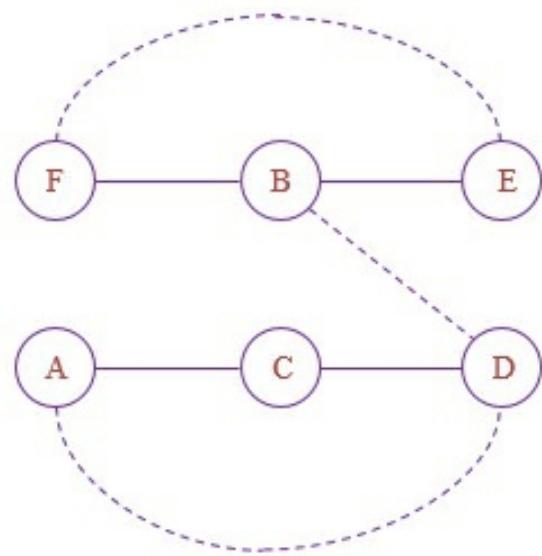


Fig. 2.8.2: DFS for a Graph G

The connected components are,

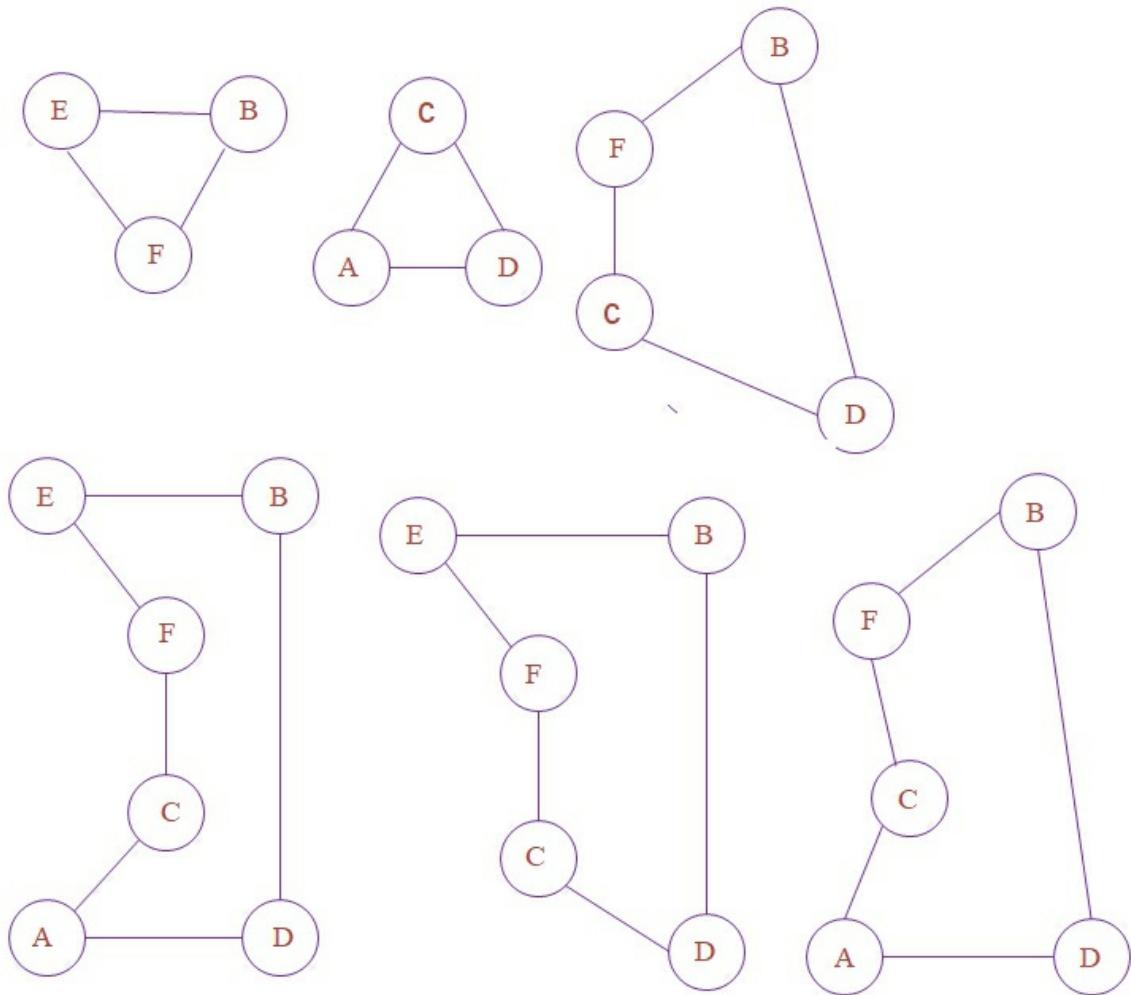


Fig. 2.8.3: Connected Components of a Graph G

Fig. :

### **3 DIVIDE AND CONQUER TECHNIQUE**

#### **3.1 INTRODUCTION TO DIVIDE AND CONQUER**

Divide and Conquer (D and C) is an important algorithm design paradigm based on multi branched recursion. This divide and conquer algorithm works by recursively breaking down the problem into two or more subproblems of the related type, until it becomes simple enough to solve it directly. To give the solution to the original problem, the solutions to the subproblems are combined.

The various problems that can be solved in this approach are shown below.

1. Binary Search
2. Max and Min problem
3. Quick Sort
4. Merge Sort
5. Stressens Matrix Multiplication

Sometimes, this Divide and Conquer approach is also applied to algorithms that reduce each problem to only one subproblem. In binary search algorithm for finding a record in the sorted list we use this approach. These algorithms can be implemented more efficiently than general Divide and Conquer algorithms. In particular, it is named as Decrease and Conquer.

The Divide and Conquer algorithm correctness is usually proved by mathematical induction and its computational cost is often determined by solving the recurrence relation.

#### **3.2 ADVANTAGES OF DIVIDE AND CONQUER APPROACH**

The main advantages of Divide and Conquer approach are:

- 1) Solving Problems:** Divide and Conquer is a powerful tool for solving the conceptually difficult problems like classical Tower of Hanoi problem in which it requires a way of breaking the problems into sub problems, of trivial cases and finally combine the sub problems to the subproblem.
- 2) Efficient Algorithms:** Divide and Conquer paradigm often helps in

discovering the efficient algorithms. The divide and conquer approach led to a improvement of the asymptotic cost of the solution. For example, if in a problem, if the base case have constant bounded size, the work required for the splitting the problem and combining the partial solutions is proportional size ‘n’ and there are a bounded number p of subproblems of size  $(n/p)$  at each stage, then the cost of dividing the Divide and Conquer Algorithm will be of  $O(n \log n)$ .

- 3) **Concurrency Control:** The Divide and Conquer technique is naturally adopted in the execution of multiprocessor machines, especially shared memory systems where the communication of data between processors need to be planned in advance because of distinct sub problems can be executed on different processors.
- 4) **Memory Access:** Divide and Conquer naturally makes use of efficient memory caches. The reason is that the subproblem is small enough. The subproblem and all its subproblems can be solved within the cache, without accessing the slower main memory.
- 5) **Roundoff Control:** The Divide and Conquer technique, in computations with rounding arithmetic, for example, a Divide and Conquer algorithm may yield more accurate results than a superficially equivalent iterative method.

### 3.3 GENERAL METHOD

The principle of Divide and Conquer strategy is that it is easier to solve several smaller instances of problem when compared to one larger problem.

The Divide and Conquer paradigm involves three steps at the each level of recursion. They are:

- 1) **Divide:** Divide the problem into number of subproblems.
- 2) **Conquer:** Conquer the subproblems by recursively solving the subproblems. If in case the subproblem size is small enough, solve the subproblems in a straight forward manner.
- 3) **Combine:** Combine the subproblem into the problem of the original size.

The diagrammatic representation of Divide and Conquer Strategy is shown in the below diagram.

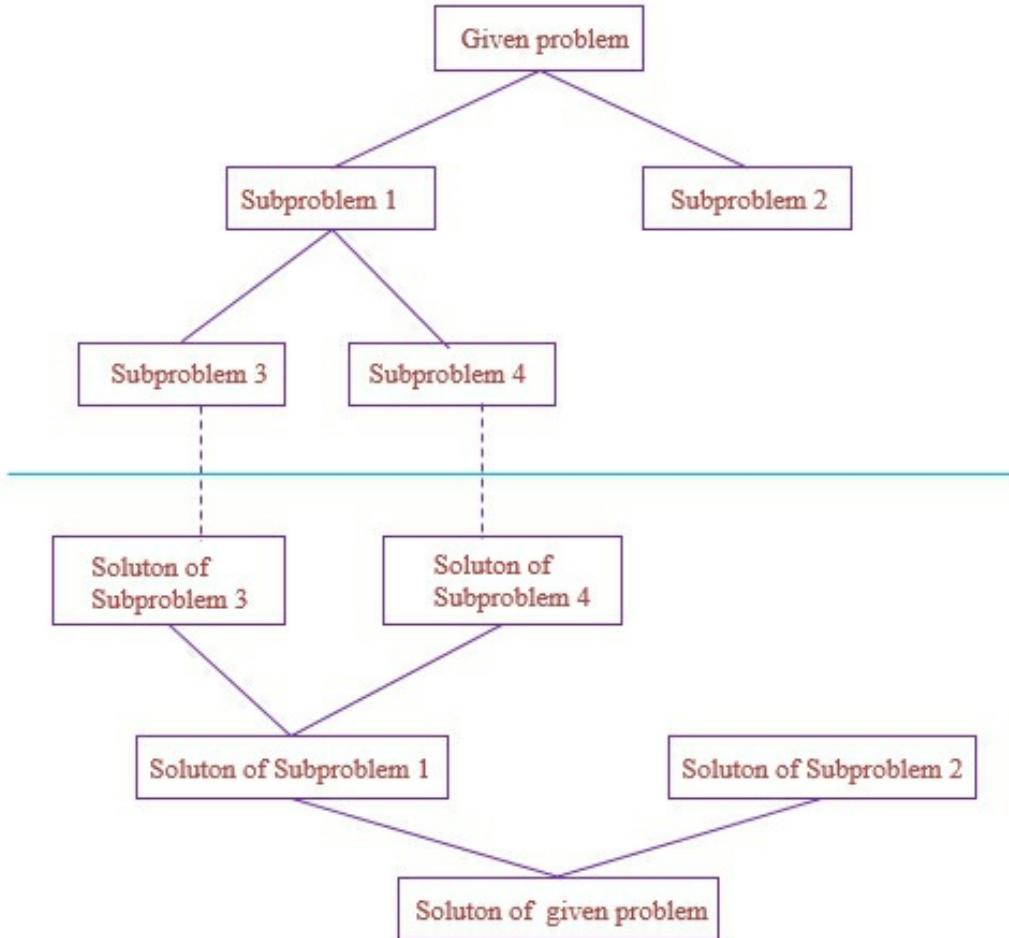


Fig. 3.3.1: Diagrammatical Representation of Divide and Conquer Strategy

### 3.3.1 Control abstraction of Divide and Conquer

A control abstraction is a procedure that reflects the way an actual program based on Divide and Conquer technique will look like. The control abstraction clearly shows the flow control but the primary operations are specified by other procedures. It can be written recursively or iteratively. However, the basic functionality of either method is one and same.

The general strategy for divide and conquer algorithm is shown below.

```

Algorithm DC(P)
{
    if Small(P) then return S(P);
    else
    {
        Divide P into smaller instances P1, P2, P3, ..., Pk, k ≥ 1;
        Apply DC to each of these subproblems;
        return Combine(DC(P1), DC(P2), ..., DC(Pk));
    }
}

```

Algorithm 3.3.1: Control Abstraction for Divide and Conquer

Where, Small( $P$ ) is a Boolean valued function that determine wheather the input size is small enough that the answer can be computed without splitting. Then, the function  $S$  will be invoked. Otherwise, the problem  $P$  is divided into smaller subproblems. The smaller subproblems  $P_1, P_2, P_3, \dots, P_k$  are solved by recursive applications of Divide and Conquer. Combine is a function that determine the solution to  $P$  using the solution to  $k$  subproblems. If the size pf  $P$  is  $n$  and the sizes of  $k$  subproblems are  $n_1, n_2, n_3, \dots, n_k$  respectively, then computing time of Divide and Conquer is described by the recurrence relation.

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

Where,  $T(n)$  is the time for DC on any input of size  $n$  and  $g(n)$  is the time to compute the answer directly for small inputs. The function  $f(n)$  is the time for dividing  $P$  and combining the solutions to subproblems. For Divide and Conquer based algorithms that produce subproblems of the same type as the original problem. This algorithm can be described by using recursion.

The time complexity of many Divide and Conquer algorithms can be given by recurrence relation.

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

Where, a and b are known constants. We assume that  $T(1)$  is known and  $n$  is a power of  $b$  i.e.,  $n = b^k$ .

## 3.4 APPLICATIONS OF DIVIDE AND CONQUER

Various applications of Divide and Conquer strategy are as follows.

### 3.4.1 Binary Search

Let  $A[1:n]$  be an array of elements that are sorted in non-decreasing order. Consider the problem of determining whether a given element  $X$  is present in the array or not. If  $X$  is present, we are to determine its position. If  $X$  is not present in the array, the value 0 is returned.

Let  $P = (n, A_i \dots, A_i, X)$  denote an arbitrary instance of the problem, where  $n$  is the number of elements in the array and  $A_i \dots, A_i$  is the array of elements and  $X$  is the element to be searched.

Divide and Conquer strategy is used to solve the problem. If problem ( $P$ ) size is small i.e., if the array consisting of one element, it takes one unit of time to determine the element  $X$  is present or not. If the problem size not small it can be divided into two equal subarrays. Pick an index in the range  $(i, \frac{i+1}{2})$  i.e.,  $q \leftarrow \frac{i+1}{2}$  and compare the element  $X$  with  $A[q]$ .

There are three possibilities. They are:

- 1) If  $(A[q] = X)$ , then the problem is solved and return  $q$ .
- 2) If  $(X < A[q])$  then  $X$  is the element has to be searched in the subarray  $A[i : q-1]$ . Therefore, the problem reduces its size.
- 3) If  $(X > A[q])$  then  $X$  is the element has to be searched in the subarray  $A[q+1 : l]$  and the process is repeated until the element is found.

#### 3.4.1.1 Binary Search Algorithm

The recursive algorithm for binary search is as follows,

Algorithm BinSearch( $A, i, l, X$ ).

```

{
//Given an array A[1:n] of elements in non-decreasing order.
//X is the element to be searched.

{
if(l = i) then
{
    if(X = A[i]) then return i;
    else return 0;
}
else
{
    //Divide the problems into small problems
    mid := [  $\frac{(i + 1)}{2}$  ];
    if (X = A[mid]) then
        return BinSearch(A, i, mid-1, X);
    else return BinSearch(A, mid+1, l, X);
}
}

```

Iterative algorithm for binary search is as follows.

Algorithm BinSearch(A, n, X)

```

{
    low := 1, high := n;
    while(low ≤ high) do
    {
        mid := [  $\frac{(low + high)}{2}$  ];

```

```

if(X < A[mid]) then
    high := mid - 1;
else if(X > A[mid]) then
    low := mid + 1;
else return mid;
}
return 0;
}

```

### **EXAMPLE PROBLEM 1**

Consider an array consisting of following elements, 1, 3, 5, 6, 8, 10, 24, 56, 85, 100, 116, 120, 131, 160.

#### **1) Let X = 160**

Here, total number of elements, n = 14.

Low	High	Mid	A[mid]	X ? A[mid]
1	14	7	24	160 > 24
8	14	11	116	160 > 116
12	14	13	131	131 > 160
14	14	14	160	found

#### **2) Let X = 9**

Low	High	Mid	A[mid]	X ? A[mid]
1	14	7	24	9 < 24
1	6	3	5	9 > 5
4	6	5	7	9 > 7
6	6	6	10	10 > 9
7	6	-	-	-

Since, low > high, return 0 i.e., not found.

#### **3.4.1.2 Time Complexity of a Binary Search Algorithm**

The time complexity for binary search technique is generally based on

the height of the binary tree. There are three cases for the time complexity of binary search. They are:

- 1) **Best Case:** Best case occurs when we are searching for the middle element itself. In that case, the total number of comparisions required is only one. Hence, in best case time complexity of binary search is  $O(1)$ .
- 2) **Average Case:** Average case occurs some where in between recursive calls, but not till the end of recursive call. Hence, the average case time complexity of binary search is  $O(\log n)$ .
- 3) **Worst Case:** Worst case occurs when the key to be searched is in either at the first position or at the last position of an array. Hence, in this case the maximum number of element comaprision are required. Each time we make one comparision and divide given numbers  $n$  into  $n/2$ . The recurrence relation is shown below.

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= [T(n/4) + 1] + 1 \\ &= [T(n/8) + 1] + 2 \\ &= T(n/2^3) + 3 \\ &\dots \\ &= T(n/2^k) + k \\ &= T(1) + \log_2 n \quad (\text{Let } 2^k = n \implies k = \log_2 n) \\ &= 1 + \log_2 n \\ T(n) &= O(\log_2 n) \end{aligned}$$

In successful searches, the computing time for the binary search in the best, average and worst cases are  $O(1)$ ,  $O(\log n)$  and  $O(\log n)$  respectively. In unsuccessful searches the best, average and worst cases is  $O(\log n)$ .

### 3.4.1.3 Modified Binary Search Algorithm

The binary search algorithm can be modified in such a way that it makes one comparision per each interaction of the while loop. This algorithm is given below,

Algorithm ModBSearch(A, n, X)

{

```

low := high := n+1;
while(low <(high - 1)) do
{
    mid := [ (low + high) / 2 ];
    if( X < A[mid]) then
        high := mid;
    else
        low := mid;
}
if(X = A[low]) then
    return low; // x is found
else
    return 0; //x is not found
}

```

The best, average, worst cases times for ModBSearch algorithm is same for both successful and unsuccessful searches and it is  $O(\log n)$ .

### **3.4.2 Finding Maximum and Minimum**

For finding maximum and minimum item in the set of  $n$  elements is one of the simplest problem to implement directly by using divide and conquer strategy. Finding maximum and minimum problem finds the maximum and minimum elements. They are:

- 1) Conventional method for finding maximum and minimum.
- 2) Divide and conquer method for finding maximum and minimum.

#### **3.4.2.1 Conventional Method for Maximum and Minimum**

In the conventional method we assume that the first elements as minimum (or maximum) and the remaining elements are compared with the first and updating the maximum (or minimum) elements is found. The algorithm for conventional method is as follows,

```

Algorithm ConventionalMinMax(A, n, X)
{
    max = min = A[1];
    for i = 2 to n do
    {
        if(A[i] > max) then max = A[i];
        else if(A[i] < min) then min = A[i];
    }
}

```

Where,

A is an array of elements from 0 to (n-1).

Max used to store the maximum element.

Min used to store the minimum element.

The analysis of best, average and worst case is given below.

- 1) **Best case:** In the best case when all the elements are in the increasing order. In this case we never go to the else part of the code. Therefore the number of comparisions is  $(n - 1)$ .
- 2) **Worst Case:** This occurs when the elements are in the decreasing order of elements. Worst case occurs when the first if condition is always false, so it is necessary to go to the else part. Therefore, the number of comparision for the first if is  $(n - 1)$  and second if is  $(n - 1)$ . Hence, the time complexity is  $2(n - 1)$ .
- 3) **Average Case:** Average case occurs when the elements of the array are grater than the half of the time. So, the number of comparisions is  $\frac{(n - 1) + 2(n - 1)}{2} = \frac{3(n - 1)}{2}$ .

### 3.4.2.2 Divide and Conquer Method for Maximum and Minimum

The Divide and Conquer method can be applied only when the number of elements is grater than 2. A Divide and Conquer algorithm for this problem is obtained by dividing any problem P = (n, A[1], A[2], A[3], ...,

$A[n]$ ) into subproblems.

For example, we can divide  $P$  into two subproblems  $P_1$  and  $P_2$ .

If  $\text{Max}(P)$  and  $\text{Min}(P)$  are the maximum and minimum of the elements in  $P$ , then

$\text{MAX}(P) = \text{The larger of } \text{MAX}(P_1) \text{ and } \text{MAX}(P_2).$

$\text{MIN}(P) = \text{The smaller of } \text{MIN}(P_1) \text{ and } \text{MIN}(P_2).$

For solving the problem consider the following 3 cases.

- 1) If there is only one element ( $n = 1$ ), then  $\min = \max = A[1]$ .
- 2) If there are only two elements ( $n = 2$ ), then the problem is solved using one comparision.
- 3) If more than 2 elements ( $n > 2$ ), then the problem  $P$  is divided into two instances  $P_1$  and  $P_2$ . After dividing into subproblems, apply Divide and Conquer method recursively to get the smallest subproblem.

For the problem, Divide and Conquer algorithm is as follows,

Algorithm DCMinMax( $A, I, j, \max, \min$ )

```
{  
if(i=j) then  
    max = min = A[i];  
else if (i = (j - 1))  
{  
    if(A[i] > A[j][i])  
    {  
        max = A[i];  
        min = A[j];  
    }  
    else  
    {  
        max = A[j];  
    }  
}
```

```

min = A[i];
}
}
else
{
//split the problem into two equal subproblems.
mid = [(i+j)/2];
DCMinMax(A, I, mid, max, min);
DCMinMax(A, mid+1, max1, min1);
//combine the solutions
if(max < max1) then max = max1;
if(min > min1) then min = min1;
}
}

```

Where,

A is an array of elements indexing 0 to n – 1.

i indicates the index of first element in the array (i = 0).

j indicates the elemnt of the last index of the array (j = (n – 1)).

Max and min are the two variables to store maximum and minimum elements.

In the Analysis of algorithm, T(n) represent the number of element comparisions needed to be obtained using the following recurrence relation,

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ T(n/2) + T(n/2) + 2 & \text{if } n > 2 \end{cases}$$

Let  $n = 2^k$ , for some  $k > 0$ ,  $T(n) = T(n/2) + T(n/2) + 2$

$$T(n) = 2T(n/2) + 2$$

$$\begin{aligned}
&= 2(2T(n/4) + 2) + 2 \\
&= 2^2(2T(n/2^2) + 2^2 + 2 \\
&= 2^3T(n/2^3) + 2^3 + 2^2 + 2 \\
&= 2^{k-1} T(n/2^{k-1}) \quad (\because n = 2^k)
\end{aligned}$$

$$\begin{aligned}
&= \frac{n}{2} T(2) + 2^{k-2} \\
&= \frac{n}{2} + n - 2 \\
\therefore T(n) &= \frac{3n}{2} - 2
\end{aligned}$$

Hence,  $\frac{3n}{2} - 2$  is the best, average and worst case number of comparisons when  $n$  is the power of 2.

In conventional method, it takes  $2(n-1)$  comparisons for worst case. So, the Divide and Conquer method saves 25% of number of comparisons compared to the conventional method. Thus, the recursive method is more efficient than the conventional method.

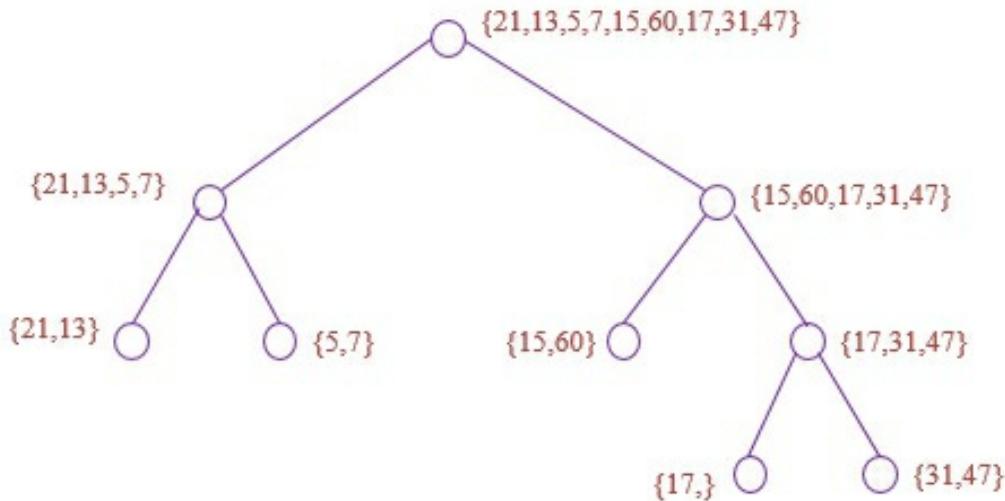
### **EXAMPLE PROBLEM 1**

**Find the maximum and minimum elements from the following list of elements using Divide and Conquer method. The list is {21, 13, 5, 7, 15, 60, 17, 31, 47}.**

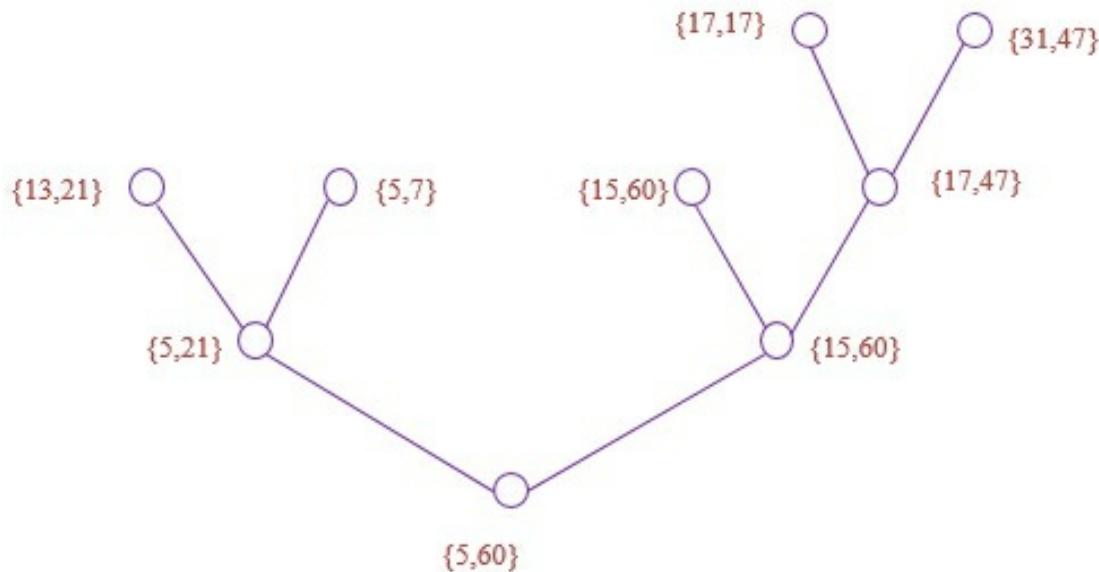
Solution:

Given list is {21, 13, 5, 7, 15, 60, 17, 31, 47}

Dividing it into subproblems until they are small enough to find the solution.



Solve the subproblems and combine as {Min, Max}.



The maximum and minimum elements of the problem are 60 and 5.

### 3.4.3 Quick Sort

Quick Sort Technique is based on Divide and Conquer design technique. In this technique at every step each element is placed in its proper position. Quick sort performs very well on longer lists. It works recursively, by first selecting the random “pivot value” from the list (array). Then it partitions the list into elements that are less than the pivot and the list greater than the pivot. The problem of solving the given list is to reduce to the problem of sorting two sublists and process continues until the list is sorted. This sorting technique is considered as an in-place sorting technique since it uses no other

array storage. The working of quick sort is illustrated in the following figure.

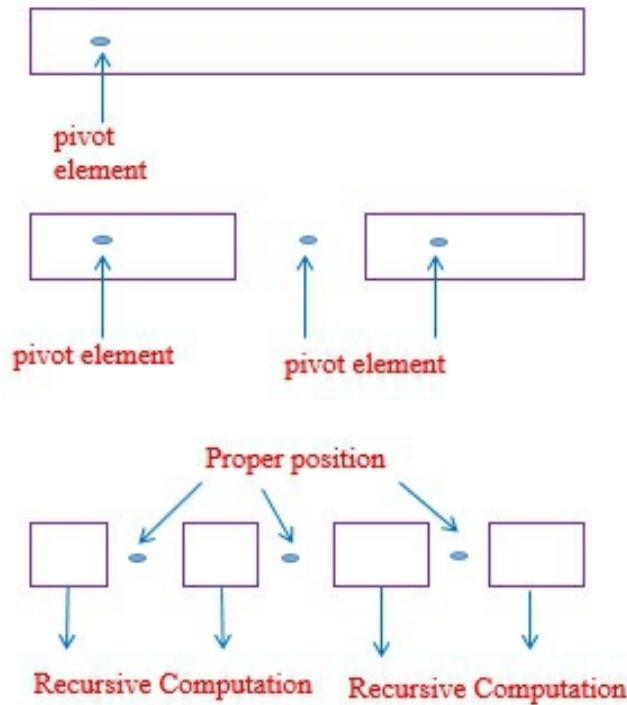
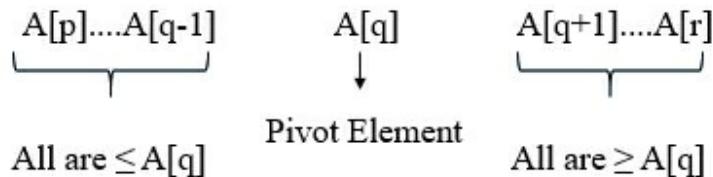


Fig. 3.4.1: Working of Quick Sort

Here is the Divide and Conquer procedure for sorting the subarray  $A[p \dots r]$ .

**Divide:**  $A[p \dots r]$  is divided into two halves  $A[p \dots q-1]$  and  $A[q+1 \dots r]$  to be solved recursively. Here,  $A[q]$  is the pivot element as shown below,



**Conquer:**  $A[p \dots q-1]$  and  $A[q+1 \dots r]$  are sorted recursively.

**Combine:** No need of this steps, as all this leave sorted array in-place. The algorithm for quick sort is as follows,

### 3.4.3.1 Quick Sort Algorithm

The algorithm for quick sort is as follows,

Algorithm QuickSort(first, last)

//sorts the sub-array  $A[first] \dots A[last]$  of a globally

```

//defined array A[0...n] in ascending order
//input: indexes of the element to be sorted
//output: A[first]...A[last] is a sorted rearrangement of the same elements
{
if(first < last)
{
//select a pivot element and position the array into two subarrays
Pivot = Partition(A, first, last + 1);
//solve the subarrays recursively
QuickSort(first, pivot - 1);
QuickSort(pivot + 1, last);
//there is no need for combining solutions
}
else
{
print "Array cannot be partitioned further"
}
}

```

The following accomplishes an in-place partitioning of the element of A[low : high – 1].

```

Algorithm Partition(first, last)
//The subarray A[first..last] is partitioned by taking first element as pivot.
{
pivotEle = A[first];
low = first;
high = last + 1;
while(low < high)

```

```

{
while(A[low] < pivotEle)
{
    low = low + 1;
}
while(A[high] > pivotEle)
{
    high = high -1;
}
if(low < high)
{
    swap(A[low], A[high]);
}
}

return high; //returns the position of pivot element
}

Algorithm swap(a, b)
{
    temp = a;
    a = b;
    a = temp;
}

```

### **EXAMPLE PROBLEM 1**

**Trace the quick sort algorithm to sort the list C, O, L, L, E, G, E in alphabetical order.**

Solution:

The given list is (C, O, L, L, E, G, E)

Consider the elements initially as

0	1	2	3	4	5	6
C	O	L	L	E	G	E

We have to sort it using Quick Sort

0	1	2	3	4	5	6
C	O	L	L	E	G	E

Pivot Element

Scanning from right to left, there is no alphabet that comes before C (pivot element). Hence, C is in the correct position.

Correct Position      Sub list, all are  $\geq C$

0	1	2	3	4	5	6
C	O	L	L	E	G	E

Pivot Element

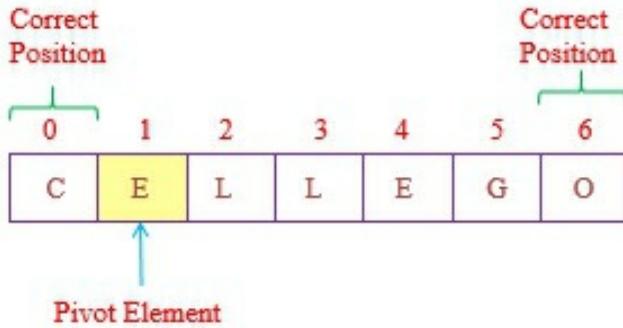
Scanning from right to left, the alphabet that comes before O (pivot element) is E. Hence, exchange both of them.

Correct Position      Sub list, all are  $\leq O$       Pivot Element

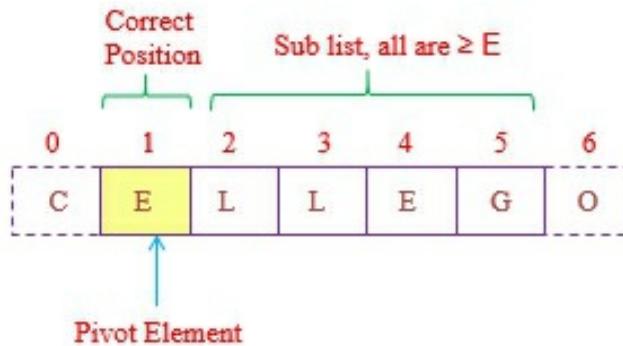
0	1	2	3	4	5	6
C	E	L	L	E	G	O

Pivot Element

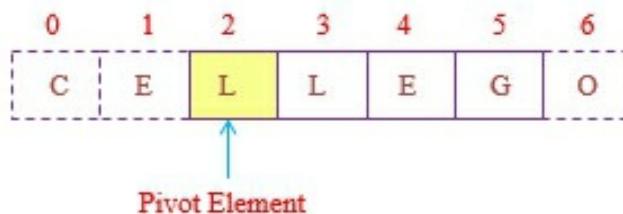
Scanning from left to right, there is no alphabet which comes after O (pivot element). Hence, O is in the correct position.



Now, we have to consider it as sublists as shown below.



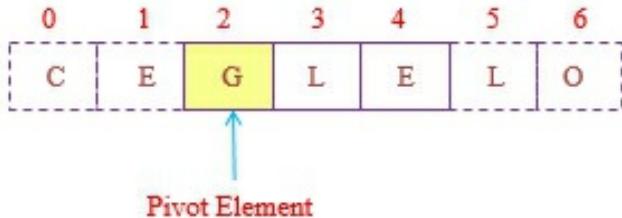
Scanning from right to left, there is no alphabet which comes before E. Hence, E is also in the correct position (i.e., position 2).



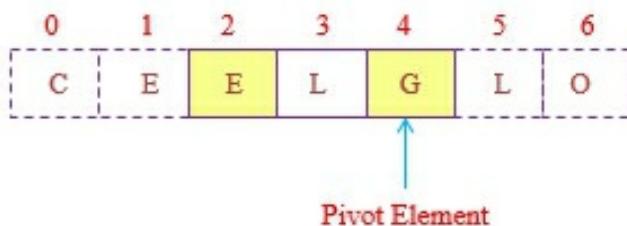
Scanning from right to left, the alphabet which comes before L (the pivot element) is G. Hence, change both of them.



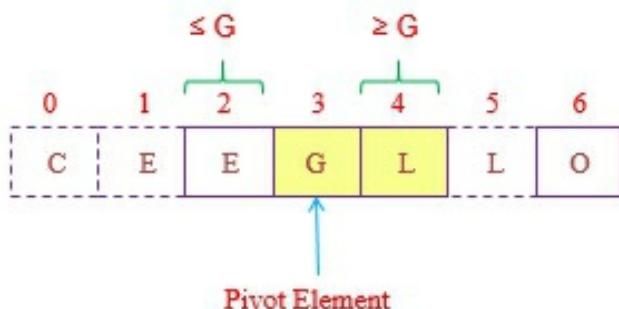
Now, considering the pivot element as shown below.



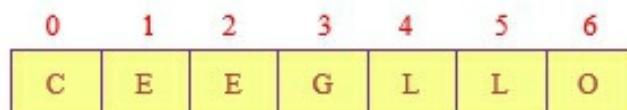
Scanning from right to left, the alphabet which comes before G (pivot element) is E. Hence change both of them.



Scanning from left to right, the alphabet which comes after G (pivot element) is L. Hence, change both of them.



Thus, the sorted list can be obtained as shown below.



## EXAMPLE PROBLEM 2

**Apply quick sort to sort the list E, X, A, M, P, L, E in alphabetical order.**

Solution:

Consider the given list in the form of an array  $a[i]$  as shown below.

0	1	2	3	4	5	6
E	X	A	M	P	L	E

Consider the first element as the pivot element (pivot).

i.e.,  $A[0] = \text{pivot}$  and set the pointers  $i$  and  $j$  as shown below.

0	1	2	3	4	5	6
E	X	A	M	P	L	E
low pivot	i					high j

Now, we perform quick sort on the above array to obtain the list in alphabetical order.

Step 1: Since  $A[i] > \text{pivot}$  (i.e.,  $X > E$ ),  $i$  is not incremented. Whereas  $A[j] \geq \text{pivot}$ . (i.e.,  $E \geq E$ )  $j$  is decremented.

0	1	2	3	4	5	6
E	X	A	M	P	L	E
low	i			j	high	
pivot						

Step 2: Since  $A[j] > \text{pivot}$  (i.e.,  $L > E$ ),  $j$  is decremented.

0	1	2	3	4	5	6
E	X	A	M	P	L	E
low	i		j		high	

pivot

Step 3: Since,  $a[j] > \text{pivot}$  (i.e.,  $P > E$ ),  $j$  is decremented

0	1	2	3	4	5	6
E	X	A	M	P	L	E
low	i	j				high
pivot						

Step 4: Since,  $A[j] > \text{pivot}$  (i.e.,  $M > E$ ),  $j$  is decremented.

0	1	2	3	4	5	6
E	X	A	M	P	L	E
low	i	j				high
pivot						

Step 5: Since,  $A[j] < \text{pivot}$  (i.e.,  $A < E$ ) and  $A[i] > \text{pivot}$  (i.e.,  $X > E$ ). Therefore, i and j are not moved. But, they are exchanged (or swapped).

0	1	2	3	4	5	6
E	A	X	M	P	L	E
pivot	i	j				

Step 6: Since,  $A[i] < \text{pivot}$  (i.e.,  $A < E$ ) and  $A[j] > \text{pivot}$  (i.e.,  $X > E$ ), i and j are incremented and decremented respectively.

0	1	2	3	4	5	6
E	A	X	M	P	L	E
pivot	i	j				

Step 7: As there is no alphabet that comes before A. A is said to be in correct position. Since E comes after A, E is said to be in correct position. Hence, leaving these two elements, consider the remaining elements as the right sublist.

Correct position						
0	1	2	3	4	5	6
A	E	X	M	P	L	E
Left sublist			Right sublist			

Since,  $A[j] > A[i]$ , therefore pivot element and the element j are exchanged.

Step 8: Now, consider the right sublist and set the first element as the pivot element. And also set the I and j pointers as follows,

0	1	2	3	4	5	6
A	E	X	M	P	L	E

pivot      i      j

Step 9: Since,  $A[i] < \text{pivot}$ ,  $i$  is incremented. This is continued until the condition is satisfied.

0	1	2	3	4	5	6
A	E	X	M	P	L	E

pivot      i      j

Step 10: Since,  $A[i]$  and  $A[j]$  are less than pivot element. Therefore, the pivot element and the elements  $i$  and  $j$  are exchanged as shown below,

0	1	2	3	4	5	6
A	E	E	M	P	L	X

Correct position  
Left sublist

As there is no alphabet that comes after X, hence it is said to be in correct position. And consider the remaining elements as the right sublist.

Step 11: Consider the left sublist and set the first element as the pivot element. And also set the  $I$  and  $j$  pointers as follows,

0	1	2	3	4	5	6
A	E	E	M	P	L	X

low      high  
pivot      i      j

Step 12: Since  $A[j] > \text{pivot}$  (i.e.,  $M > E$ ),  $i$  is not incremented. But as  $A[j] > \text{pivot}$  (i.e.,  $L > E$ ) is decremented.

0	1	2	3	4	5	6
A	E	E	M	P	L	X

pivot      i      j

Step 13: Since,  $A[j] > \text{pivot}$  (i.e.,  $P > E$ ),  $j$  is decremented. This is continued until the condition is satisfied.

0	1	2	low	4	high	6
A	E	E	M	P	L	X

pivot      i  
j            Right sublist

Since,  $j = \text{pivot}$  and there is no element that comes before  $E$ . Therefore,  $E$  is said to be in the correct position. Whereas the remaining elements forms the right sublist.

Step 14: Now, consider the right sublist and set the first element as the pivot element. And also set  $i$  and  $j$  pointers as shown below.

0	1	2	3	4	5	6
A	E	E	M	P	L	X

pivot      i      j

Step 15: Since,  $A[i] > \text{pivot}$  ( $P > M$ ),  $i$  is not incremented. And  $A[j] < \text{pivot}$  (i.e.,  $L < M$ ),  $j$  is not decremented. But  $A[i] > A[j]$  (i.e.,  $P < L$ ), therefore  $A[i]$  and  $A[j]$  are swapped.

0	1	2	3	4	5	6
A	E	E	M	L	P	X

pivot      j      i

Step 16: Since,  $A[i] > \text{pivot}$  and  $A[j] < \text{pivot}$ , therefore  $i$  and  $j$  are not incremented and decremented respectively.

0	1	2	3	4	5	6
A	E	E	L	M	P	X

Since, each left and right sublist contains only one element in them, therefore it is impossible to divide the list further. Thus, the sorted list after performing quick sort is shown below.

0	1	2	3	4	5	6
A	E	E	L	M	P	X

### 3.4.3.2 Time Complexity of Quick Sort Algorithm

The best case, average case and worst case time complexities of quick sort is shown below.

- 1) **Best Case:** Quick sort works best if each array is divided into two equal subarrays of size  $n/2$ . This generates  $\log n$  levels in the recursion tree. The recurrence relation is given by,

$$T(n) = 2T(n/2) + O(n)$$

Applying the master theorem, we get,

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

- 2) **Worst Case:** In the worst case, the chosen pivot is either the smallest or largest element in the array. In this case, one part is empty and the other part contains the remaining elements this generates  $n-1$  levels in the recursion tree.

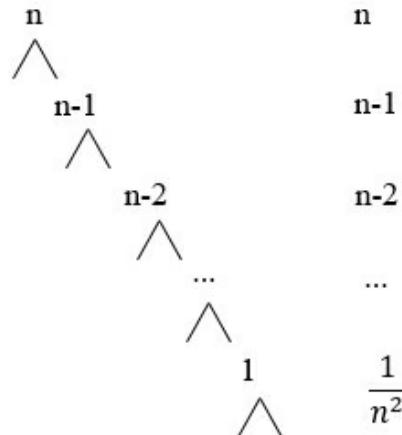


Fig. 3.4.2: Working of Quick Sort

When quick sort is first called partition takes time  $n-1$ . Quick sort is then called twice ,once with an empty array and again with an array of size  $n-1$ . The next call to partition takes time  $n-2$  .Quick sort is again called twice, once with an empty array and again with an array of size

$n-2$ . The next call to partition takes time  $n-3$ . This process continues. The total time for all the calls to partition is,

$$\text{The time complexity } T(n) = T(n-1) + C_n$$

$$= T(n-2) + C(n-1) + C_n$$

$$= T(n-3) + C(n-2) + C(n-1) + C_n$$

.....

$$= T(1) + C(1 + 1 + 3 + 4 + \dots + n)$$

$$= b + \frac{C_n(n+1)}{2}$$

$$= O(n^2)$$

∴ The time complexity of quick sort in worst case is  $T(n) = O(n^2)$ .

- 3) **Average Case:** In average case, the array is partitioned by choosing any random number. In this case at each level some of the partitions are well balanced while some are finally unbalanced. Let us assume that the partition of array to be 9 : 10, then the recurrence so obtained is,

$$T(n) = T(9n/10) + T(n/10) + C_n$$

The recurrence tree is shown in the following figure.

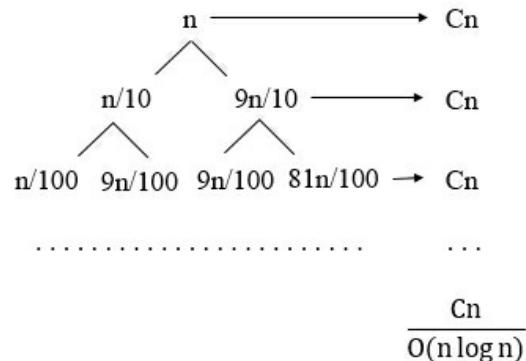


Fig. 3.4.3: Working of Quick Sort

Therefore, the total cost of quick sort is  $O(n \log n)$ .

In analyzing the quick sort, only the number of elements that are needed to be compared are counted i.e.,  $T(n)$ . The frequency count of the other operations is of the same order as the count of element comparison  $T(n)$ .

In this average case, the following points are to be considered,

- i. The elements which are to be sorted are different.
- ii. The partitioning element 'v' in the procedure PARTITION is chosen using a random selection process.

Let the average case value be  $T_A(n)$ . The partitioned element ‘v’ in procedure partition is chosen using a random selection process and it has the same probability of being the  $i^{\text{th}}$  smallest element,  $1 \leq i \leq P - M$  in  $A[M : P-1]$ . Therefore, the two subfiles that are to be sorted are

$A(M : J)$  and  $A(J + 1 : P - 1)$  with a probability equal to  $\frac{1}{P - M}$  where  $M \leq J \leq P$ .

Now, make a note that  $T_A(0) = 0$  and  $T_A(1) = 0$ .

Let,  $(n+1)$  is the number of element comparision required by procedure PARTITION on its first call. Therefore, the recurrence relation can be obtained as,

$$T_A(n) = n + 1 + \frac{1}{n} \sum_{1 \leq i \leq n} [T_A(i-1) + T_A(n-i)] \quad \dots (1)$$

Multiplying both sides of equation (1) by ‘n’, we get,

$$T_A(n) = n(n+1) + 2[T_A(0) + T_A(1) + \dots + T_A(n-1)] \quad \dots (2)$$

Replacing  $n$  by  $(n-1)$  in equation (2), we get,

$$\begin{aligned} (n-1)T_A(n-1) &= (n-1)(n-1+1) + 2[T_A(0) + T_A(1) + \dots + T_A(n-2)] \\ (n-1)T_A(n-1) &= (n)(n-1) + 2[T_A(0) + T_A(1) + \dots + T_A(n-2)] \end{aligned} \quad \dots (3)$$

Subtracting equation (#) from equation (2), we get,

$$\begin{aligned} nT_A(n) - (n-1)T_A(n-1) &= n(n+1) + 2[T_A(0) + T_A(1) + \dots + T_A(n-1)] - n(n-1) - \\ &\quad 2[T_A(0) + T_A(1) + \dots + T_A(n-2)] \\ nT_A(n) - (n-1)T_A(n-1) &= n^2 + n + 2[T_A(0) + T_A(1) + \dots + T_A(n-2)] - \\ &\quad n^2 + n - 2[T_A(0) + T_A(1) + \dots + T_A(n-2)] \\ nT_A(n) - (n-1)T_A(n-1) &= 2n + 2T_A(n-1) \\ nT_A(n) &= (n-1)T_A(n-1) + 2n + 2T_A(n-1) \\ nT_A(n) &= (n-1+2)T_A(n-1) + 2n \end{aligned}$$

$$nT_A(n) = (n+1)T_A(n-1) + 2n \quad \dots \\ (4)$$

Dividing both sides of equation (4) by  $n(n+1)$ , we get,

$$\frac{nT_A(n)}{n(n+1)} = \frac{(n+1)T_A(n-1)}{n(n+1)} + \frac{2n}{n(n+1)}$$

$$\frac{T_A(n)}{(n+1)} = \frac{T_A(n-1)}{n} + \frac{2}{(n+1)}$$

Repeatedly, we substitute ‘n’ with  $(n-1)$ , using the above equation for  $T_A(n-1)$ ,  $T_A(n-2)$  ... we get,

$$\begin{aligned} \frac{T_A(n)}{(n+1)} &= \frac{T_A(n-1)}{n} + \frac{2}{(n+1)} = \frac{T_A(n)}{(n+1)} = \frac{T_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{(n+1)} \\ &= \frac{T_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{(n+1)} \\ &\quad \dots \dots \dots \\ &= \frac{T_A(1)}{2} + 2 \sum_{3 \leq l \leq n+1} \left[ \frac{1}{l} \right] \\ &= 0 + 2 \sum_{3 \leq l \leq n+1} \left[ \frac{1}{l} \right] \quad (\because T_A(1) = 0) \\ &= 2 \sum_{3 \leq l \leq n+1} \left[ \frac{1}{l} \right] \end{aligned}$$

$$\text{But, } \sum_{3 \leq l \leq n+1} \frac{1}{l} \leq \int_2^n \frac{1}{y} dy = \log_e(n+1) - \log_e 2$$

$$\therefore T_A(n) \leq 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Therefore, the average case time complexity of a quick sort algorithm is less than the time complexity which is  $O(n^2)$ .

### 3.4.3.3 Randomized Quick Sort

Quick sort algorithm is modified in such a way that it works good on any input. This modification is necessary mainly for the worst case of quick sort in which the elements are already in sorted order. In this instead of selecting

the first element, a random element is picked as the pivot. The algorithm is given below,

```
Algorithm RandQuickSort(first, last)
{
    //Sorts the element in the array
    A[1 : n]
    if(first < last) then
    {
        if(last - first) > 5 then
            swap (a, Random() mod(last - first + 1) + first, first);
        K = Partition(a, first, last+1);
        //K is the partitioning element
        RandQuickSort(first, K-1);
        RandQuickSort(K+1, last);
    }
}
```

The randomized quick sort algorithm runs in  $O(n \log n)$  time.

#### 3.4.4 Merge Sort

Merge sort is a sorting technique which is generally applied between two sorted arrays. Hence, a third array is created and the elements of both the sorted arrays are matched, correspondingly, the third array is filled. Hence the final third array obtained will be consisting of elements belonging to both the arrays which will be in sorted order.

The algorithm for recursive merge sort is shown below,

```
Algorithm MergeSort(a, first, last)
//sorts array a[first] ... a[last] by merge sort recursively
//input parameters: a, first, last (a[first...last])
//output parameters: array a[first...last] sorted in non-decreasing order
```

```

{
if(first == last)
return;
else
{
    mid = (first + last)/2;
    MergeSort(a, first, last);
    MergeSort(a, mid+1, last);
    Merge(a, first, mid, last);
}
//end merge sort
}

```

Algorithm for iterative merge sort is shown below.

```

Algorithm MergeS(a, first, mid, last)
//Merge two sorted arrays into one sorted array
//Input parameters: a, first, mid, last
//Output parameters: Sorted array a[first...last]
{
p = first; //index in first subarray a[first : mid]
q = mid + 1; // index in second subarray a[mid+1 : last]
r = first; //index in a local array b[ ]
while((p ≤ mid) && (q ≤ last)) do
{
if(a[p] ≤ a[q])
{
    b[r] = a[p];
    r++;
    p++;
}
}
for(i = first; i ≤ last; i++)
{
    a[i] = b[i];
}

```

```

    p = p + 1;
}
else
{
    b[r] = a[q];
    q = q + 1;
}
r = r + 1
}

while(p < mid) do //copy remainder if any of first subarray to b
{
    b[r] = a[p];
    p = p + 1;
    r = r + 1;
}

while(q < last) do //copy remainder if any of second subarray to b
{
    b[r] = a[q];
    q = q + 1;
    r = r + 1;
}

for r = first to last do //copy b[ ] back to a[ ]
    a[r] = b[r];
} //end merge

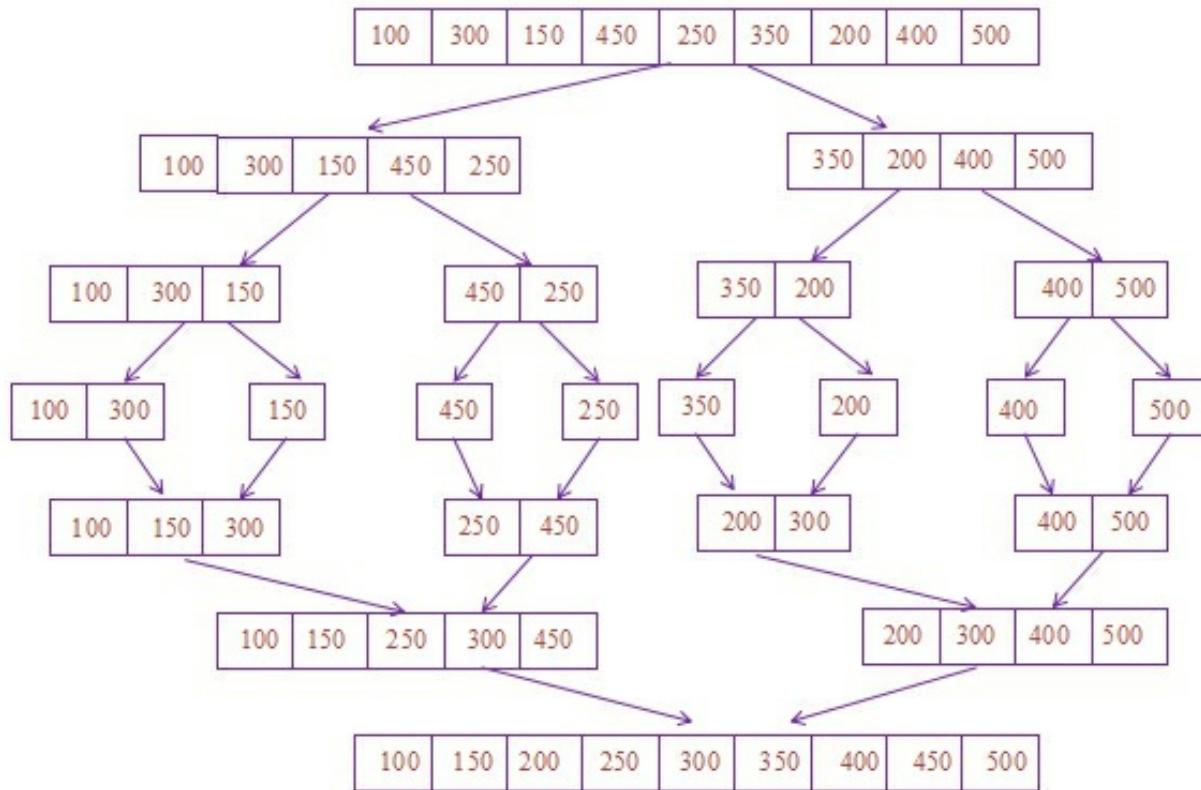
```

### **EXAMPLE PROBLEM 1**

**Show how merge sort algorithm works on the data set keys 100, 300, 150, 450, 250, 350, 200, 400, 500.**

### Solution:

Given, the element set 100, 300, 150, 450, 250, 350, 200, 400, 500.



#### 3.4.4.1 Time Complexity of Merge Sort Algorithm

The best case time of an algorithm is the time required for the best case input of size  $n$ , and the best case input is the input of size  $n$  in which the algorithm runs faster for all possible inputs of that size. The input of size  $n$  in the best case instead of the smallest input for which the algorithm runs fastest.

The best case input for merge sort is the already sorted input. Hence, we assume that the algorithm does not require the step to compare the elements. The best case efficiency of Merge sort is  $O(n \log n)$ .

Merge sort, on one element it takes a constant time. When the elements are greater than 1, i.e.,  $n > 1$  we break down the running time as follows.

**Divide:** In this step, the middle of the subarray will be computed, which takes constant time. Thus, the amount taken for dividing the problem into subproblems is  $O(n = O(1))$ .

**Conquer:** the two problems are solved recursively, each of the size  $n/2$ , which will provide  $2T(n/2)$  to the running time.

**Combine:** The Merge sort takes  $O(n)$  time in merging. Thus, the time required to combining all the subproblems into one solution for the main problem is  $C(n) = O(n)$ .

We have,

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

Addition of  $O(n)$  and  $O(1)$  results in the linear function of ‘n’, i.e.,  $O(n)$ . If the constant time ‘C’ is used for solving the problem of size 1, then we have,

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

Solving the above recurrence relation using change of variable method. Replacing  $n$  by  $2^k$  and

$$T(2^k) = t_k, \text{ we get,}$$

$$T(2^k) = 2T(2^{k-1}) + C2^k$$

$$t_k = 2t_{k-1} + C2^k \quad \dots \quad (1)$$

Replacing  $k$  by  $k-1$ , we get,

$$t_{k-1} = 2t_{k-2} + C2^{k-1} \quad \dots \quad (2)$$

Multiplying the equation (1) by equation (2) and subtracting the result from the equation (1) we get,

$$t_k = 2t_{k-1} + C2^k$$

$$t_{k-1} = 2t_{k-2} + C2^{k-1}$$

---


$$( - ) \quad ( - ) \quad ( - )$$

$$t_k - 2t_{k-1} = 2t_{k-1} - 4t_{k-2}$$

$$t_k = 4t_{k-1} - 4t_{k-2} = 0$$

Put  $t_k = x^k$ ,

We get the following characteristic equation,

$$x^k - 4x^{k-1} + 4x^{k-2} = 0$$

$$x^2 - 4x + 4 = 0$$

$$x^2 - 2x - 2x + 4 = 0$$

$$(x - 2)(x - 2) = 0$$

$$(x - 2)^2 = 0$$

The general equation is,

$$t_k = (C_1 + C_2 k) 2^k$$

Putting back n, we get,

$$T(n) = C_1 n + C_2 n \log n \quad (k = \log_2 n)$$

$$T(n) \in O(n \log n).$$

The worst case of merge sort algorithm is also  $O(n \log n)$ .

#### 3.4.4.2 Space Complexity of Merge Sort Algorithm

Stack is required for implementing recursion in the merge sort. Since the merge sort splits each set into two approximately sized subsets, the maximum depth of the stack is proportional to  $\log_2 n$ . So the space complexity is  $O(\log_2 n)$ .

#### 3.4.5 Strassen's Matrix Multiplication

Strassen's matrix method is a classic example of Divide and Conquer approach in which it achieves efficiency in multiplication of square matrices. Initially, we must consider the problem of matrix multiplication in square matrices.

Let A and B are two matrices of size  $n \times n$  each. Obtain gtheir product matrix C, as  $C = AB$  which is also an  $n \times n$  matrix whose  $(i, j)^{th}$  element can be found as follows,

$$C(i, j) = \sum_{k=1}^n A(i, k)B(k, j)$$

Therefore, multiplication of two  $n \times n$  matrices require  $O(n^3)$  operations because the product matrix consists of  $n^2$  elements and for computing each element of the product matrix requires  $O(n)$  operations.

Suppose the matrices A and B are each partitioned into four sequence matrices, each submatrix having dimensions  $\frac{n}{2} \times \frac{n}{2}$ .

Then,

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

That is we get,

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{21}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

In order to compute AB using the above decomposition, used to perform 8 multiplications of  $\frac{n}{2} \times \frac{n}{2}$  matrices and 4 additions of  $\frac{n}{2}$  matrices.

Since two  $\frac{n}{2} \times \frac{n}{2}$  may be added in time  $Cn^2$  for some constant C, the overall computing time, T(n) of the resulting Divide and Conquer algorithm is given by the recurrence as,

$$T(n) = \begin{cases} b & \text{if } n \leq 2 \\ 8T\left(\frac{n}{2}\right) + an^2 & \text{if } n > 2 \end{cases}$$

Where, a and b are constants.

For recurrence  $T(n) = 8T(\frac{n}{2}) + an^2$ , if we find running time then it is calculated as,

Compare above recurrence with  $T(n) = aT(\frac{n}{b}) + f(n)$  we get,

$$a = 8$$

$$b = 2$$

$$f(n) = an^2$$

$$n^{\log_b a} = n^{\log_2 8} = n^{\log_b 2^3} = n^{3\log_2 2} = n^3$$

$$T(n) = O(n^3)$$

The time complexity for conventional matrix multiplication is  $O(n^3)$ .

### 3.4.5.1 Strassen's Algorithm

Therefore, the above recurrence has solution  $T(n) = O(n^3)$ , so there is no improvement. Since the matrix multiplication ( $O(n^3)$  operations) is more expensive than matrix addition ( $O(n^2)$  operations). To reduce the fewer multiplications and have a way to compute  $C_{ij}$ 's using only 7 multiplications and 18 additions and subtractions.

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{11})(B_{11} + B_{12})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + S - Q + U$$

Then the resulting recurrence relation for  $T(n)$  may be written as,

$$T(n) = \begin{cases} b & \text{if } n \leq 2 \\ 7T\left(\frac{n}{2}\right) + an^2 & \text{if } n > 2 \end{cases}$$

Where,  $a$  and  $b$  are constants.

$$T(n) = 7T\left(\frac{n}{2}\right) + an^2$$

Compare this recurrence with  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ .

We get,

$$a = 7$$

$$b = 2$$

$$f(n) = an^2$$

and

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81} \quad [\log_2 7 = \frac{\log_{10} 7}{\log_{10} 2}]$$

$$T(n) = O(n^{2.81})$$

Thus, the time complexity of Strassen's matrix multiplication is  $O(n^{2.81})$ .

### EXAMPLE PROBLEM 1

Consider the matrices  $A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ ,  $B = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$  use ordinary method and Strassen's Method to compute the matrix product.

**In Ordinary Method:**

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \times 0 + 1 \times 1 & 1 \times 0 + 1 \times 1 \\ 1 \times 0 + 1 \times 1 & 1 \times 0 + 1 \times 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

The above result is obtained by ordinary method.

### In Stressen's Method:

By using Stressen's formulas, we get the following same matrix.

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$P = (A_{11} + A_{22})(B_{11} + B_{22}) = (1 + 1)(0 + 1) = 2$$

$$Q = (A_{21} + A_{22})B_{11} = (1 + 1)(0) = 0$$

$$R = A_{11}(B_{12} - B_{22}) = (1)(0 - 1) = -1$$

$$S = A_{22}(B_{21} - B_{11}) = (1)(1 - 0) = 1$$

$$T = (A_{11} + A_{12})B_{22} = (1 + 1)(1) = 2$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12}) = (1 - 1)(0 + 0) = 0$$

$$V = (A_{12} - A_{11})(B_{11} + B_{12}) = (1 - 1)(1 + 1) = 0$$

$$C_{11} = P + S - T + V = 2 + 1 - 2 + 0 = 1$$

$$C_{12} = R + T = -1 + 2 = 1$$

$$C_{21} = Q + S = 0 + 1 = 1$$

$$C_{22} = P + S - Q + U = 2 - 1 - 0 + 0 = 1$$

Thus, the resultant matrix,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

## **4 GREEDY METHOD**

### **4.1 INTRODUCTION TO GREEDY METHOD**

The greedy method is considered as a most powerful and straight forward technique in designing an algorithm. They are shortlisted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect of these decisions may have in the future. Greedy method algorithms are easy to invent, easy to implement and most of the time quite efficient. Greedy algorithms are used to solve optimization problems.

#### **Functions of Greedy Algorithm:**

To construct the solution in an optimal way, algorithm maintains two sets.

1. One contains chosen items.
2. The other one contains rejected items.

The following four functions are available in Greedy Method. They are:

1. A function that checks whether chosen set of items provide a solution.
2. A function that checks that the feasibility of a set.
3. The selection function tells which of the candidates is the most promising.
4. An objective function, which does not appear explicitly gives the value of the solution.

#### **Structure of Greedy Algorithm:**

- 1) Initially the set of chosen items is empty, i.e., solution set.
- 2) At each step,
  - i. Item will be added in a solution set by using a selection function.
  - ii. IF the set would be no longer be feasible THEN
    - Reject items under consideration. (is never considered)

again)

- iii. ELSE IF set is still feasible THEN
  - Add the current item.

#### **4.1.1 Properties of Greedy Algorithm**

- 1) **Feasibility:** A feasible set is promising if it can be extended to produce not merely a solution but an optimal solution.
- 2) **Greedy Choice Property:** It says that a globally optimal solution can be arrived at by making a locally optimal choice.

Consider some of the problems like Knapsack, Job Sequencing with Deadlines and Minimum Cost Spanning Trees are based on subset paradigm.

For the problem that make decision by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of Greedy Method is ordering paradigm.

Some of the problems like optimal storage on tapes like optimal merge patterns and single storage shortest path are based on ordering paradigm.

#### **4.1.2 Comparison between Divide and Conquer Approach Greedy Approach**

In Divide and Conquer approach, to get the solution of the original problem, a problem is divided into subproblems of the same kind of the original problem, until they are small enough to be solved and finally the solutions of the subproblems are combined. Therefore we will get the solution to the original problem.

In Greedy Approach, a problem is solved by determining a subset to satisfy some constraint. If that subset satisfies the given constraints, then it is called as feasible solution, which maximizes or minimizes a given objective function. A feasible solution that either maximizes or minimizes an objective function is called as optimal solution.

Based on these above definitions of both the Divide and Conquer approach and Greedy approach, we can list out certain relative merits and demerits of divide and conquer approach when compared to the greedy approach.

S. NO.	Divide and Conquer Approach	Greedy Approach
1	Divide and Conquer approach is the result oriented approach.	By Greedy Method, there are some changes of getting an optimal solution to a specific problem.
2	The time taken by this Divide and Conquer algorithm is efficient when compared to Greedy approach.	The time taken by this algorithm is not that much efficient when compared to Divide and Conquer approach.
3	Divide and Conquer approach does not depend on constraints to solve a specific problem.	Greedy approach cannot make further move, if the subset chosen dose not satisfy the specified constraints.
4	Divide and Conquer approach is not efficient for larger problems.	Greedy approach is applicable and as well as efficient for a wide variety of problems.
5	As the problem is divided into a larger number of subproblems, the space requirement is very high in Divide and Conquer approach.	Space requirement is less when compared to divide and conquer approach in Greedy method.
6	Divide and Conquer approach is not applicable to problems which are not divisible. Example Knapsack problem.	Greedy Approach is also applicable to problems which are not divisible. Example Knapsack problem.

Table 4.1.1: Comparisions Between Divide and Conquer Approach and Greedy Approach

## 4.2 GENERAL METHOD

Greedy method works in different stages and it is a straight forward method. It takes only one input at a time and produces number of solutions. This is called as feasible solution which satisfies our constraints. A feasible solution which maximizes or minimizes a given objective function is called as optimal solution.

The important concepts of Greedy method are defined as follows.

**1) Feasible Solution:** Any subset of original input that satisfies a given set of constraints is called as a feasible solution. A solution in which the values of decision variables that satisfy all the constraints of a linear programming (LP) problem is known as feasible solution.

Usually, feasible solutions are present in the feasible region and a linear combination of one or more basic feasible solutions may result in an optimal solution.

**2) Objective Function:** Objective function is an input for which a feasible solution is to be obtained that either maximizes or minimizes. This step must find locally optimal choices. (Once the local choice is optimal, the solution is globally optimal).

The equation  $f = ax + by + cz + \dots$  with linear constraints is the objective function. The maximum or minimum value of the objective function is the optimal value. The optimal values collectively generate an optimal solution.

**3) Optimal Solution:** An optimal solution is one, which minimizes (maximizes) the given objective function. An optimal solution is a feasible solution that maximizes and minimizes the given objective function. In general, every problem will have a unique optimal solution.

### **Properties of Optimal Solution:**

- i. Optimal solution lies on the boundary of the feasible region, which indicates that the interior points of the feasible solution region are neglected while searching for an optimal solution.
- ii. Optimal solution lies at the end of the feasible region, which specifies that the work of the search procedure is reduced while searching for an optimal solution.

**4) Irrevocable:** Once the choices are made in a particular step, it cannot be changed in the subsequent steps.

#### **4.2.1 Control Abstraction of Greedy Method**

All the Greedy method problems have  $n$  inputs. Greedy method is a straight forward method, it works in stages, by using selection procedure, we arrange the input in an order and by considering one input at a time. At each stage a decision is made regarding whether the particular input is in optimal

solution. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then the input is not added to the partial solution otherwise it is added.

The general pseudocode for Greedy strategy is,

```

Algorithm Greedy(A, n)

//A[1:n] is an input array of size n.

{
    solution = 0 //initializes the solution to empty

    for l = 1 to n do

    {
        Y = Select(A);

        if feasible(solution, Y) then

        {
            solution = union(solution, Y);

        }

        else

        reject();

    }

    return solution;
}

```

Algorithm 4.2.1: Control abstraction of Greedy Method for the subset paradigm

### 4.3 APPLICATIONS OF GREEDY METHOD

In this section we will discuss about various applications that can be solved using Greedy approach.

### **4.3.1 Job Sequencing with Deadlines**

On a single processor, the arrangement of jobs with deadline constraints is referred as the job sequencing with deadlines. Job sequencing deals with a set of ‘n’ jobs, associated with each job ‘i’ is an integer deadline  $d_i \geq 0$  and a profit  $p_i \geq 0$ . For any job ‘i’ the profit ‘ $p_i$ ’ is earned if the job is completed by its deadline. To complete a job, one have to process a job on a machine for one unit of time. Only one processor is available for processing all jobs. Not all jobs have to be scheduled.

A feasible solution for this problem is a subset ‘J’ of jobs such that each job in this subset can be completed by its deadline. The value of the feasible solution ‘J’ is the sum of the profits of the jobs in ‘J’, i.e.,  $\sum_{i \in J} P_i$ . An optimal solution is a feasible solution with maximum value.

To formulate a greedy algorithm, to obtain an optimal solution, it is necessary to formulate an optimaization measure to determine how the next job is choosen. The objective function  $\sum_{i \in J} P_i$  is choosen as an optimization measure.

Using this measure, the next job to be included is the one that increases  $\sum_{i \in J} P_i$  the most, subject to the constraint that the resulting ‘J’ is a feasible solution. This requires the jobs to be considered in non-increasing order of this  $p_i$ ’s.

Initially,  $J = \emptyset$  and  $\sum_{i \in J} P_i = 0$ .

Next, the job that has the largest profit is added. It forms feasible solution. The next job, to join the set (J), has to satisfy the following conditions,

- 1) The job should have maximum profit so as to form an optimal solution.
- 2) The job to be included should be completed by its deadline to form a feasible solution.

#### **4.3.1.1 Time Complexity of Job Sequencing with Deadlines Algorithm**

The algorithm for job sequencing with deadlines is shown below.

```

Algorithm JS(D, J, n, K)

//d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1.

//The jobs are ordered such that p[1] ≥ p[2] ≥ ... ≥ p[n]. 

//J[i] is the ith job in the optimal solution, 1 ≤ i ≤ K.

//Also, at termination d[J[i]] ≤ d[J[i+1]], 1 ≤ i < K.

{

D[0] := J[0] := 0; //Initialize

J[1] := 1; K := 1 //Include job 1

for i := 2 to n do //Consider jobs in decreasing order of Pi

{

S := K;

while(D(J[S] > D[i]) and D(J[S] ≠ S)) do

S := S - 1;

}

if(D(J[S] ≤ D[i]) and (D[i] > r)) then //Insert job i into j

{

for l := K to S + 1 step -1 do

{

J[l + 1] := J[l];

}

J[S + 1] := i; K := K + 1;

}

return K;

}//end JS Algorithm

```

Algorithm 4.3.1: Greedy algorithm for Sequencing Unit Time Jobs with Deadlines and Profits

The computing time of JS Algorithm is  $O(n^2)$ , but it can be reduced by

$O(n)$  by using disjoint set union and find algorithms.

#### 4.3.1.2 Solved Problems

##### EXAMPLE PROBLEM 1

Solve the following problem of Job Sequencing with the deadlines specified using the Greedy strategy.

$$N = 4, (P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$$

$$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$$

Solution:

Since the maximum deadlines time = 2 units, maximum of two jobs will form the feasible solution. According to the Greedy strategy, arranging the jobs in decreasing order of their profits.

$$(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$$

$$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$$

S. NO.	FEASIBLE SOLUTION	PROCESSING SOLUTION	PROFIT $\sum_{i \in J} P_i$
1	$J = \emptyset$	-	0
2	$J = \{1\}$	1	100
3	$J = \{1, 4\}$	4, 1	$100 + 27 = 127$
4	$J = \{1, 2\}$	2, 1	$100 + 10 = 110$
5	$J = \{2, 3\}$	2, 3	$10 + 15 = 25$
6	$J = \{3, 4\}$	4, 3	$27 + 15 = 42$
7	$J = \{1\}$	1	100
8	$J = \{2\}$	2	10
9	$J = \{3\}$	3	15
10	$J = \{4\}$	4	27

Here, we begin with  $J = \emptyset$  and  $\sum_{i \in J} P_i = 0$ .

Job 1 is added as it has the largest profit and  $\{2\}$  is a feasible solution. Next,

job 4 is considered,  $\{1, 4\}$  is also feasible. Next, job 3 is considered and discarded as  $\{1, 3, 4\}$  is not a feasible solution. Next, job 2 is considered and this is also discarded as  $\{1, 2, 4\}$  is not a feasible solution.

Similarly, in the same way, we select the job and include them in  $J$ , such that the job included in  $J$  must finish within the deadlines and such ' $J$ ' is said to be the feasible solution. A feasible solution with the maximum profit earned i.e.,  $\sum_{i \in J} P_i$  is said to be optimal solution.

In this problem  $\{1, 4\}$  is the feasible solution with maximum  $\sum_{i \in J} P_i$ . Therefore,  $J = \{1, 4\}$  is the optimal solution in the order 4, 1 and the optimal profit is 127 for the given problem instance.

## **EXAMPLE PROBLEM 2**

**Find the solution generated by the Job Sequencing with Deadlines algorithm when**

$$N = 4, (P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (3, 5, 20, 18, 1, 6, 30)$$

$$(d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2)$$

Solution:

Here  $n = 7$ ,

$$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (3, 5, 20, 18, 1, 6, 30)$$

$$(d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2)$$

According to Greedy strategy, arranging the jobs in decreasing order of their profits = (30, 20, 18, 6, 5, 3, 1).

Deadlines = (2, 4, 3, 1, 3, 1, 1)

S. NO.	FEASIBLE SOLUTION	PROCESSING SOLUTION	PROFIT $\sum_{i \in J} P_i$
1	$J = \emptyset$	-	0
2	$J = \{7\}$	7	30
3	$J = \{3, 7\}$	7, 3	$30 + 20 = 50$
4	$J = \{3, 4, 7\}$	7, 4, 3	$30 + 20 + 18 = 68$

5	$J = \{3, 4, 6, 7\}$	6, 7, 4, 3	$6 + 30 + 20 + 18 = 74$
6	$J = \{1\}$	1	3
7	$J = \{2\}$	2	5
8	$J = \{3\}$	3	20
9	$J = \{4\}$	4	18
10	$J = \{5\}$	5	1
11	$J = \{6\}$	6	6
12	$J = \{7\}$	7	30

The feasible solution  $J = \{3, 4, 6, 7\}$  is the optimal solution yielding a total profit of 74. The optimal job sequence is (6, 7, 4, 3).

### 4.3.2 Knapsack Problem

Now comparing Knapsack problem with general problem. In olden days there was a store, which contains different types of gold powders. Let these powders be  $n_1, n_2, n_3$  types, weights of these gold powders be  $w_1, w_2, w_3$  and cost  $c_1, c_2, c_3$  dollars respectively. Consider that a thief wants to rob the store, for that he bought an empty bag (Knapsack means an empty bag) of size M. Now, his problem was, in what way he has to place the gold powder in the bag such that he would get maximum profit? This problem is called as Knapsack problem.

Knapsack is an empty bag. Consider Knapsack whose size is M. It contains n items with weights  $w_1, w_2, w_3, \dots, w_n$  respectively. The profits of each weight is  $p_1, p_2, p_3, \dots, p_n$ .

Let  $x_1, x_2, x_3, \dots, x_n$  be the fractions of items.

The objective is to fill the Knapsack that maximizes the total profit earned. Since the Knapsack capacity is 'M', we require the total weights of all choosen objects to be atmost 'M'.

The Knapsack problem has two versions. They are:

- 1) **Fractional Version:** In this method the list of items are divisible, which means any fraction of item can be considered.

Let  $x_j$ ,  $j = 1, 2, 3, \dots, n$  be a variable representing a fraction of item  $j$  taken into Knapsack. It is necessary that  $x_j$  must satisfy the inequality  $0 \leq x_j \leq 1$ . Then the total weight of the item selected can be expressed

by the sum  $\sum_{j=1}^n w_j x_j$  and the total value of the item selected can be

expressed by the sum  $\sum_{j=1}^n v_j x_j$ . Thus, the continuous version of the Knapsack problem can be posted as the following linear programming problem.

$$\text{Maximize } \sum_{j=1}^n v_j x_j$$

Subject to constraints,

$$\sum_{j=1}^n w_j x_j \leq W \text{ (Knapsack capacity)} \quad 0 \leq x_j \leq 1 \text{ for } j = 1, 2, 3, \dots, n.$$

- 2) **0/1 Version:** In this method, the list of items are indivisible, which means the item is either accepted or discarded. Hence, the following integer linear programming problem is encountered.

$$\text{Maximize } \sum_{j=1}^n v_j x_j$$

Subjected to constraints,

$$\sum_{j=1}^n w_j x_j \leq W \text{ and}$$

$$x_j \in \{0, 1\} \text{ for } j = 1, 2, 3, \dots, n.$$

#### 4.3.2.1 Time Complexity of Knapsack Algorithm

The general pseudocode for Knapsack problem is,

```

Algorithm GreedyKnapsack(M, n)

//W[1 : n] is the array of weights, P[1 : n] is array of profits

//The objects are arranged in the decreasing order of their p/w.

//M be the capacity of Knapsack

//X[1 : n] is the solution vector

{

    for i := 1 to n do X[i] = 0;

    CU := m;

    for i:=1 to n do

    {

        if(w[i] > CU) then break;

        X[i] := 1;

        CU := CU - W[i];

    }

    if(i ≤ n) then X[i] := CU / W[i];

}

```

#### Algorithm 4.3.2: Greedy Knapsack Algorithm

In the analysis of Knapsack algorithm, if the objects are already sorted into the decreasing order of profit per unit weight ( $P_i/\omega_i$ ) then the time complexity of the algorithm is  $O(n)$ .

#### 4.3.2.2 Solved Problems

##### SOLVED PROBLEM 1

Consider the following instance of Knapsack problem.

$$n = 3, M = 20, (p_1, p_2, p_3) = (25, 24, 15)$$

$$(w_1, w_2, w_3) = (18, 15, 10).$$

Find the optimal solution for,

- i. Maximum Profit
- ii. Minimum Weight
- iii. Maximum Profit per unit Weight.

Solution:

##### CASE I:

In this case, we will place the item in the bag whose profit is maximum.

$x_1 = 1$  complete item  $w_1$  is kept in the bag i.e., 18 only 2 units place in left ( $20 - 18 = 2$ ).

$$x_2 = \text{left out space / weight of item to be placed} = \frac{2}{15}.$$

When  $w_2$  is placed no space is left in bag. So,  $x_3 = 0$  i.e., we cannot place 3<sup>rd</sup> item.

$$\sum_{1 \leq i \leq n} p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

$$\begin{aligned} &= 25 \times 1 + 24 \times \frac{2}{15} + 15 \times 0 \\ &= 25 + 3.2 + 0 \\ &= 28.2 \end{aligned}$$

##### CASE II:

In this case we will place an item in the bag, whose profit per unit weight ratio is maximum.

$$x_3 = 1$$

$$x_2 = \frac{10}{15} = \frac{2}{3}$$

$$x_1 = 0$$

$$\sum_{1 \leq i \leq n} p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3$$

$$\begin{aligned} &= 25 \times 0 + 24 \times \frac{2}{3} + 15 \times 1 \\ &= 0 + 16 + 15 \\ &= 31 \end{aligned}$$

### **CASE III:**

We will place an item in the bag, whose profit per unit weight ratio is maximum.

$$\frac{p_1}{w_1} = \frac{25}{18} = 1.4$$

$$\frac{p_2}{w_2} = \frac{24}{15} = 1.6$$

$$\frac{p_3}{w_3} = \frac{15}{10} = 1.5$$

$$\frac{p_2}{w_2}$$

Here,  $w_2$  is maximum so,  $x_2 = 1$ . Now, remaining weight of the bag is  $20 - 15$

$\frac{p_3}{w_3}$  = 5. Next maximum is  $w_3$ . Now we have to place the third item into the bag but the space is not sufficient. So we have to place  $\frac{1}{2}$  (i.e., 5/10) of third item.

Therefore,  $x_3 = \frac{1}{2}$ . The bag is already filled. So,

$$x_1 = 0$$

$$x_2 = 1$$

$$x_3 = \frac{5}{10} = 0.5$$

$$\begin{aligned}
 \sum_{i=1}^n p_i x_i &= p_1 x_1 + p_2 x_2 + p_3 x_3 \\
 &= 0 + 24 \times 1 + 15 \times 0.5 \\
 &= 24 + 7.5 \\
 &= 31.5
 \end{aligned}$$

## SOLVED PROBLEM 2

Consider the following instance of Knapsack problem.

$$n = 7, M = 15, (p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (10, 5, 15, 7, 6, 18, 3)$$

$$(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1).$$

Find the optimal solution for,

- i. Maximum Profit
- ii. Minimum Weight
- iii. Maximum Profit per unit Weight.

Solution:

CASE I:

$$x_2 = 1$$

$$x_3 = 1$$

$$x_1 = 1$$

$$x_4 = \frac{4}{7}$$

$$\sum p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3 + p_4 x_4$$

$$\begin{aligned}
 &= 18 \times 1 + 15 \times 1 + 10 \times 1 + 7 \times \frac{4}{7} \\
 &= 18 + 15 + 10 + 4 \\
 &= 47
 \end{aligned}$$

CASE II:

$$x_7 = 1$$

$$x_5 = 1$$

$$x_1 = 1$$

$$x_2 = 1$$

$$x_6 = 1$$

$$x_3 = \frac{4}{5}$$

$$\sum p_i x_i = p_1 x_1 + p_2 x_2 + p_3 x_3 + p_5 x_5 + p_6 x_6 + p_7 x_7$$

$$\begin{aligned} &= 10 \times 1 + 5 \times 1 + 15 \times \frac{4}{5} + 6 \times 1 + 18 \times 1 + 3 \times 1 \\ &= 10 + 5 + 12 + 6 + 18 + 3 \\ &= 54 \end{aligned}$$

### **CASE III:**

$$\frac{p_1}{w_1} = \frac{10}{2} = 5$$

$$\frac{p_2}{w_2} = \frac{5}{3} = 1.6$$

$$\frac{p_3}{w_3} = \frac{15}{5} = 3$$

$$\frac{p_4}{w_4} = \frac{7}{7} = 1$$

$$\frac{p_5}{w_5} = \frac{6}{1} = 6$$

$$\frac{p_6}{w_6} = \frac{18}{4} = 4.5$$

$$\frac{p_7}{w_7} = \frac{3}{1} = 3$$

$$\sum p_i x_i = p_5 x_5 + p_1 x_1 + p_6 x_6 + p_3 x_3 + p_7 x_7 + p_2 x_2$$

$$\begin{aligned} &= 6 \times 1 + 10 \times 1 + 18 \times 1 + 15 \times 1 + 3 \times 1 + 5 \times \frac{2}{3} \\ &= 6 + 10 + 18 + 15 + 3 + 3.3 \\ &= 55.3 \end{aligned}$$

A Greedy strategy is used to find the optimal solution. According to this strategy, the objects are arranged according to some criteria. In general, we arrange the objects based on the P/W values in decreasing order and then select the objects one by one and add the object into Knapsack if it fits.

#### 4.3.3 Minimum Cost Spanning Trees

A tree is defined to be an undirected, undirected and connected graph (or) a graph in which there is only one path connecting each pair of vertices. Assume that there is an undirected, connected graph G. A spanning tree is a subgraph of G, is a tree and contains all the vertices of G. A graph can have more than one spanning tree.

A minimum spanning tree is a spanning tree, that has weights with the edges and the total weight of the tree (the sum of weights of its edges).

The problem here is to find a fixed connected subgraph containing all the vertices such that the sum of the costs in the subgraph is minimum. If in case the tree contains any cycle, then we could have broken it by deleting one of its edges, the graph still be connected, but the cost will be smaller. This subgraph is called as minimum cost spanning tree.

The algorithm for minimum cost spanning tree is as follows,

Algorithm GenericMST(G, w)

{

$A \leftarrow \emptyset$

while A does not form a spanning tree do

find an edge (U, V) that does not form a cycle in A;

```

A ← AU{(U, V)};
return A;
}

```

The applications of minimum cost spanning trees are as follows,

- 1) Minimum spanning trees are useful in construction of networks.
- 2) Another application of minimum spanning tree would be to find the airline routes.
- 3) A less obvious application is that the minimum spanning tree can be used to approximately solve the travelling salesmen problem.

### **EXAMPLE PROBLEM 1**

**Consider a connected graph G with n=3 edges, as shown in the below figure. Find the possible spanning tree and minimum spanning tree.**

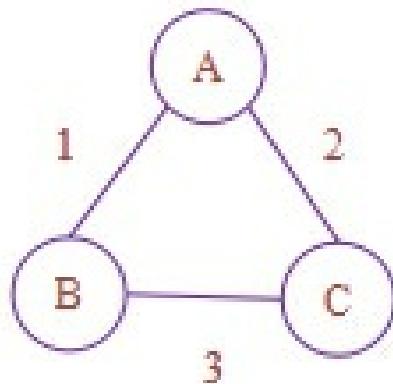


Fig. 4.3.1: Weighted Graph

#### Solution:

If the graph consists of n vertices then the possible spanning trees are  $n^{n-2}$ , for above example n=3, i.e.,  $3^{3-2} = 3^1 = 3$  spanning trees. These spanning trees can be shown in the figures shown below.

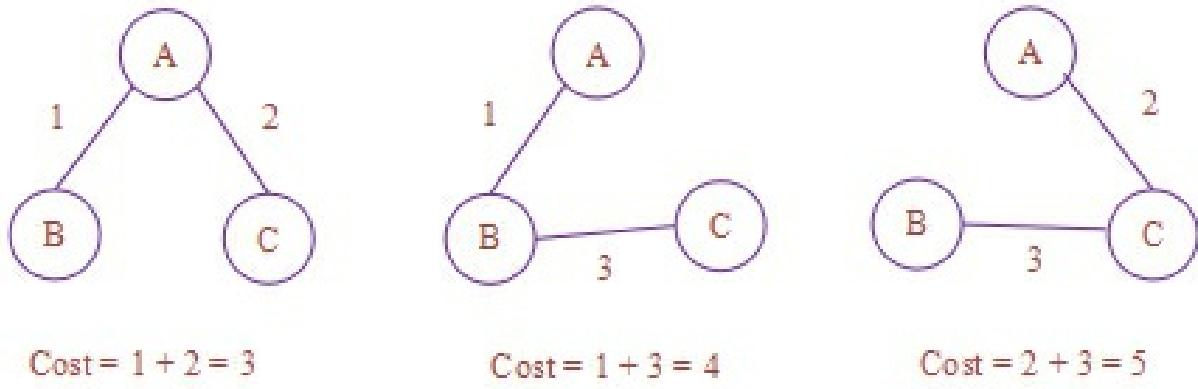


Fig. 4.3.2: Spanning Trees

Among these spanning trees, The figure (c) has the maximum cost, so it is called as minimum spanning tree for this graph.

A tree which includes all vertices of  $G$  with minimum cost is called as minimum spanning tree.

Similarly for  $n = 6$ ,  $6^{6-2} = 6^4 = 1296$  possible spanning trees, draw and find the cost of all spanning trees. It is a time consuming and a difficult process as  $n$  value increases. So, to avoid this difficulty, we will use two standard algorithms. They are Prim's and Kruskal's algorithm.

#### 4.3.3.1 Kruskal's Algorithm

Let  $G$  be a connected graph consisting of set of vertices and set of edges and each edge is associated with non-negative values. To construct minimum spanning tree, we use the following procedure,

- 1) Arrange all edges in increasing order of weight.
- 2) Select an edge with minimum weight. This is the first edge of spanning tree  $T$  to be constructed.
- 3) Select the next edge with minimum weight that does not form a cycle with the edges already included in  $T$ .
- 4) Continue step 3 until  $T$  contains  $(n - 1)$  number of edges, where  $n$  is the number of vertices of  $G$ .

The generic algorithm for Kruskal's algorithm is as follows.

```

Algorithm Kruskal(E, Cost, n, T, mincost)

//E is the set of edges in graph G

//T is the set of edges in the minimum cost spanning tree

//COST(U,V) is the cost of edge (u, v).

{

    construct a Heap out of edge cost using Heapify;

    for i ← 1 to n do

        PARENT[i] ← -1;

        i ← 0; mincost ← 0;

        while((i < n-1) and (Heap not empty)) do

        {

            Delete a minimum cost edge (u, v) from the heap and heapify using Adjust;

            j ← Find(u);

            k ← Find(v);

            if(j ≠ k) then

            {

                i ← i+1;

                T[i, 1] ← u; T[i, 2] ← v;

                mincost ← mincost + Cost[u,v];

                call Union(j, k);

            }

        }

        if(i ≠ n-1) then

        write("No spanning tree");

        else

        return mincost;

}

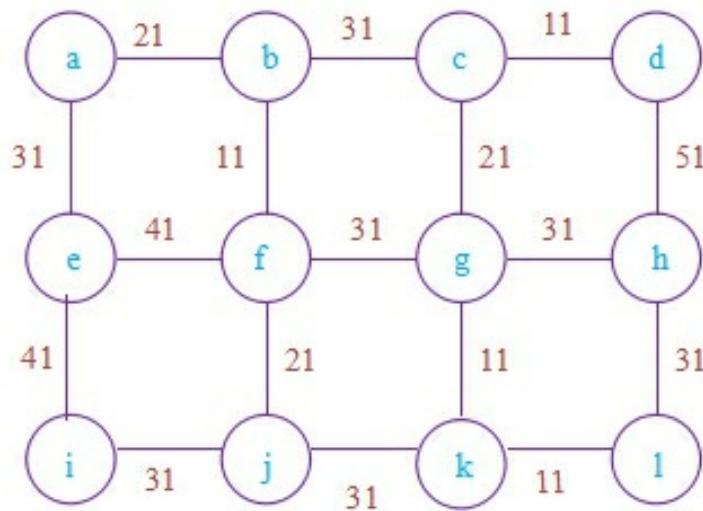
```

### Algorithm 4.3.3: Kruskal's Algorithm

The computing time of Kruskal's algorithm is  $O(E \log n)$  where  $E$  is the number of edges.

#### EXAMPLE PROBLEM 1:

Consider the graph given below. Find the cost of minimum spanning tree using Kruskal's algorithm.

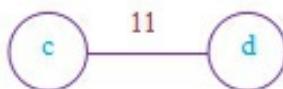


Solution:

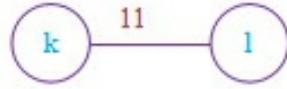
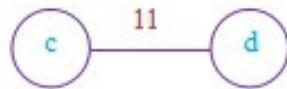
**Step 1:** The edge with minimum weight is selected edge =  $\langle c, d \rangle$ .

Weight of the selected edge = 11.

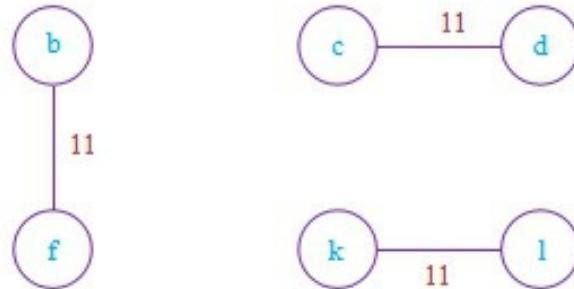
As the addition of edge to the existing tree does not form a cycle, an edge is added.



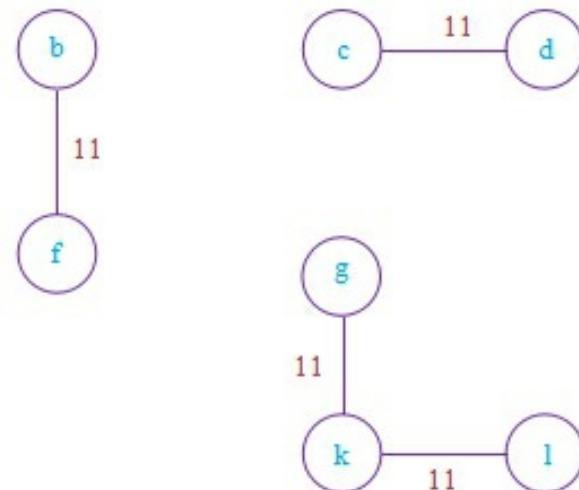
**Step 2:** Selected edge  $\langle k, l \rangle$  with weight 11.



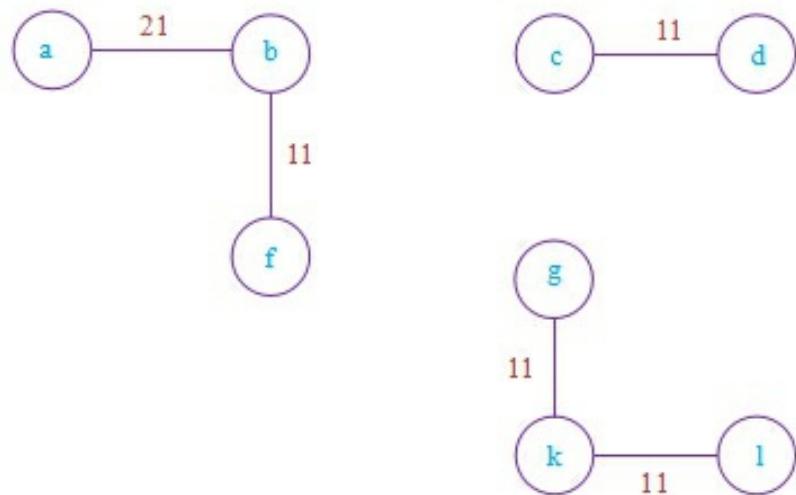
**Step 3:** Selected edge  $\langle b, f \rangle$  with weight 11.



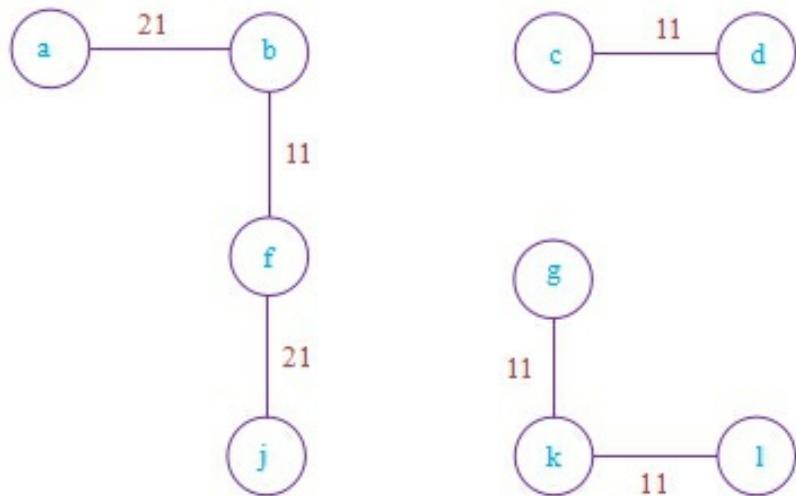
**Step 4:** Selected edge  $\langle g, k \rangle$  with weight 21.



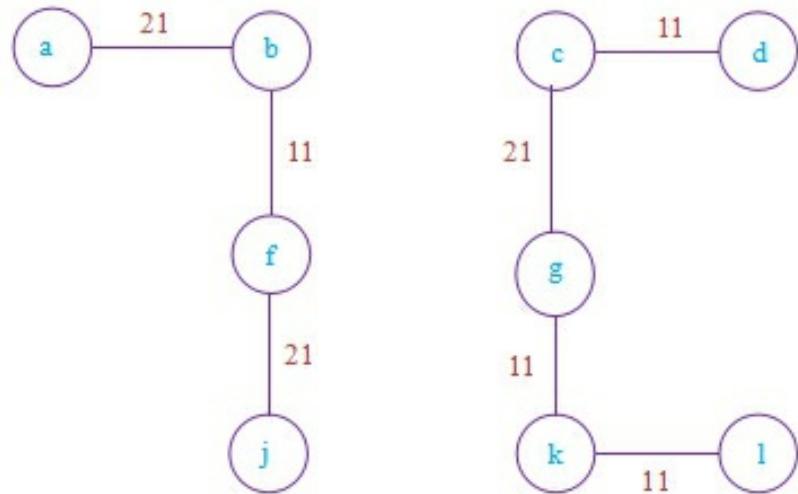
**Step 5:** Selected edge  $\langle a, b \rangle$  with weight 21.



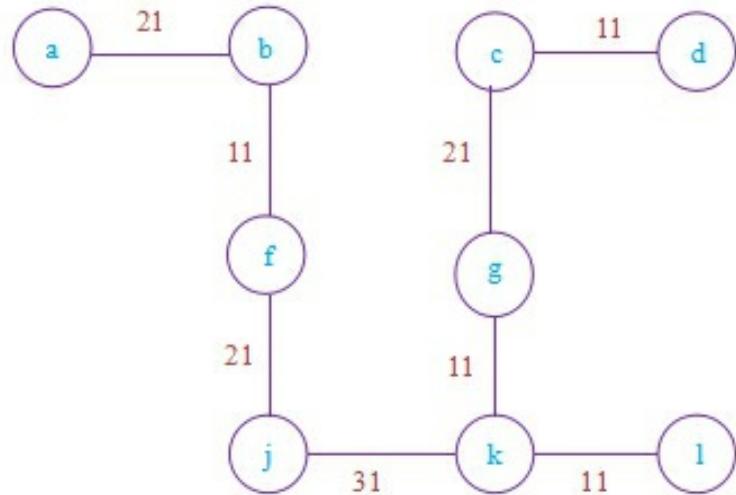
**Step 6:** Selected edge  $\langle f, j \rangle$  with weight 21.



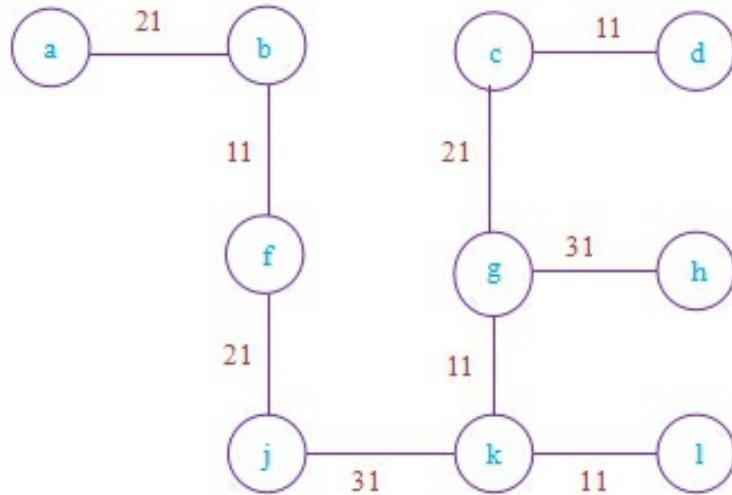
**Step 7:** Selected edge  $\langle c, g \rangle$  with weight 31.



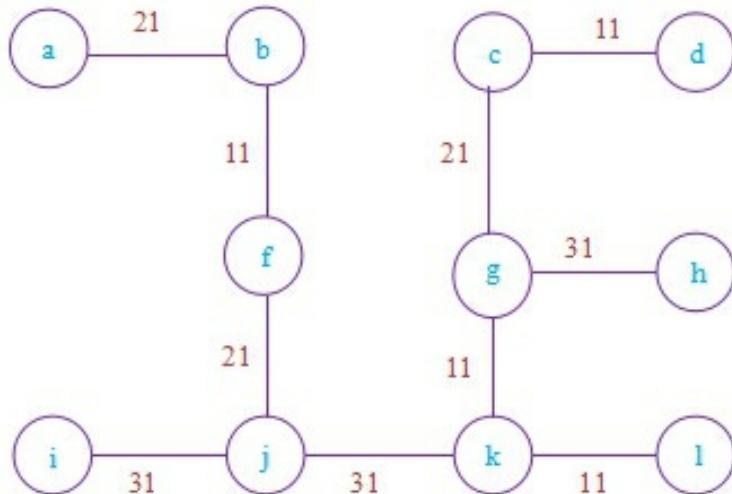
**Step 8:** Selected edge  $\langle j, k \rangle$  with weight 31.



**Step 9:** Selected edge  $\langle g, h \rangle$  with weight 31.



**Step 10:** Selected edge  $\langle i, j \rangle$  with weight 31.



Thus, by applying Kruskal's algorithm the spanning tree is obtained with minimum cost =  $11 + 11 + 11 + 11 + 21 + 21 + 21 + 31 + 31 + 31 = 231$

#### 4.3.3.2 Prim's Algorithm

Let  $G = (V, E)$  be a graph. Where  $V$  is the set of vertices and  $E$  is the set of edges. Each edge is associated with a weight. Let  $S$  be the solution of the minimum spanning tree which is initially empty. Start with some arbitrary vertex  $i$ . Select a vertex  $j$  such that the cost of the edge  $(i, j)$  is minimum among the edges that satisfy condition that  $i$  is a vertex already included in  $S$  and  $j$  is a vertex not included in  $S$ . Include  $j$  into the solution such that  $(i, j)$  is the edge of minimum cost spanning tree. At each step, edge with the smallest weight which connects a vertex in  $S$  to vertex in  $V - S$  is added to the tree. It

adds only the edges that does not create a cycle.

When the algorithm terminates, the edge in  $S$  forms a minimum spanning tree. This strategy is a Greedy strategy since the tree is augmented at each step with an edge that contributes the minimum amount possible to tree weight.

The algorithm for Prim's method is as follows.

```

Algorithm Prim(E, cost, n, t)
//E is the set of edges in G. cost[1:n,1:n] is the cost
//adjacency matrix of an n vertex graph such that cost[i,j] is
//either a positive real number or  $\infty$  if no edge (i,j) exists.
//A minimum spanning tree is computed and stored as a set of edges in the array
//t[1:n-1,1:2]. (t[i,1],t[i,2]) is an edge in the minimum cost spanning tree.
//The final cost is returned.

{
    Let (k,l) be an edge of the minimum cost in E;
    mincost := cost[k,l];
    t[1,1] := k; t[1,2] := l;
    for i := 1 to n do //initialize near
        if(cost[i,l] < cost[l,k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n-1 do
        { //find n-2 additional edges for t
            Let j be an index such that near[j] ≠ 0 and
            cost[j, near[j]] is minimum;
            t[i,1] := j; t[i,2] := near[j];
            mincost := mincost + cost[j, near[j]];
            near[j] := 0;
            for k := 1 to n do //update near[]
                if((near[k] ≠ 0) and (cost[k,near[k]] > cost[k,j]))
                    then near[k] := j;
        }
    return mincost;
}

```

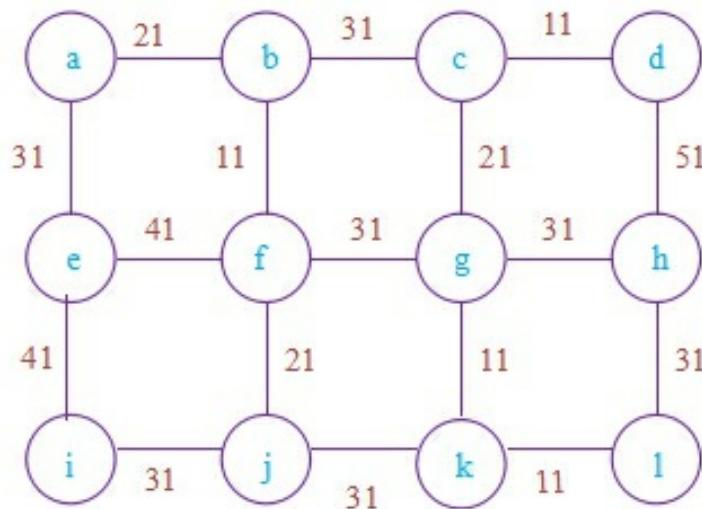
#### Algorithm 4.3.4: Prim's Algorithm

Time complexity of Prim's algorithm is  $O(n^2)$ .

The algorithm spends most of the time in finding the smallest edge. So, the time of the algorithm basically depends on how we search this edge. Therefore, Prim's algorithm runs in  $O(n^2)$  time.

#### EXAMPLE PROBLEM 1

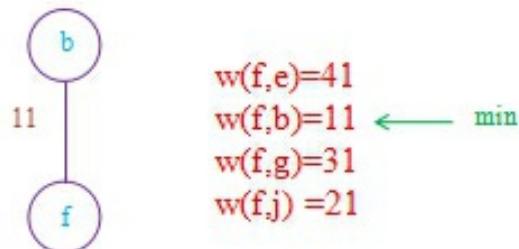
Consider the following graph given below. Find the cost of minimum spanning tree using Prim's algorithm.



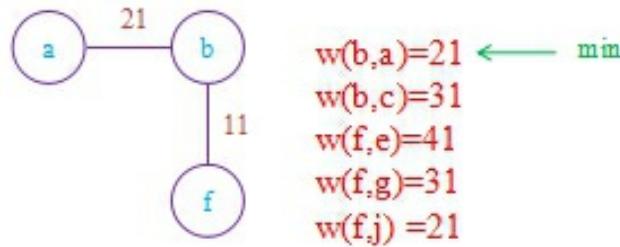
Solution:

Using Prim's algorithm, we get the spanning tree for this graph in the following steps.

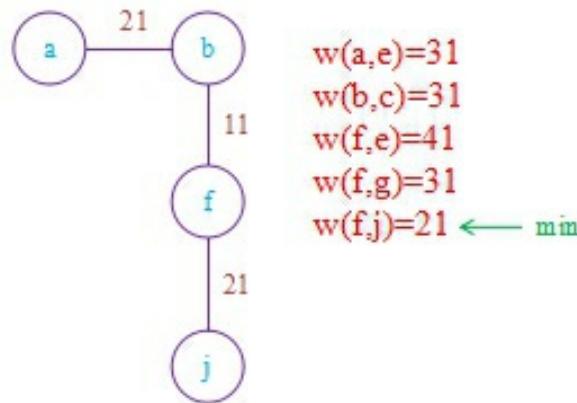
**Step 1:** Let  $f$  be the start vertex. Among the vertices  $e, b, g, j$ , the vertex is nearest one with the edge  $\langle f, b \rangle$  and weight 11.



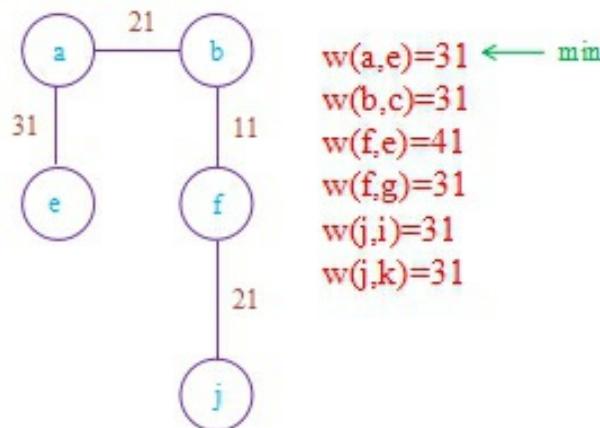
**Step 2:** Among the vertices adjacent to  $b$  and  $f$ , the vertex  $a$  is nearest one with edge  $\langle b, a \rangle$  and weight 21.



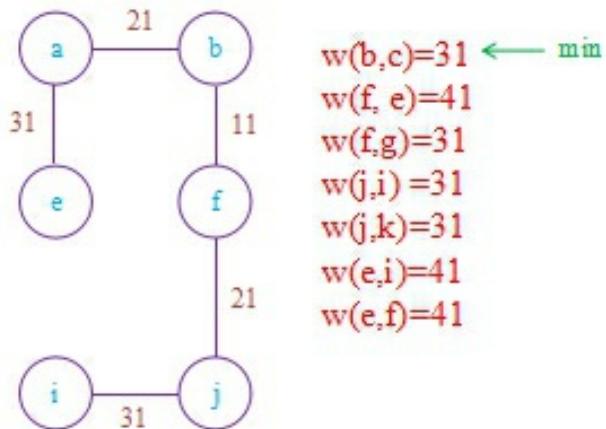
**Step 3:** Similarly the nearest vertex adjacent to one of a, b and f is j with the edge  $\langle f, j \rangle$  and weight 21.



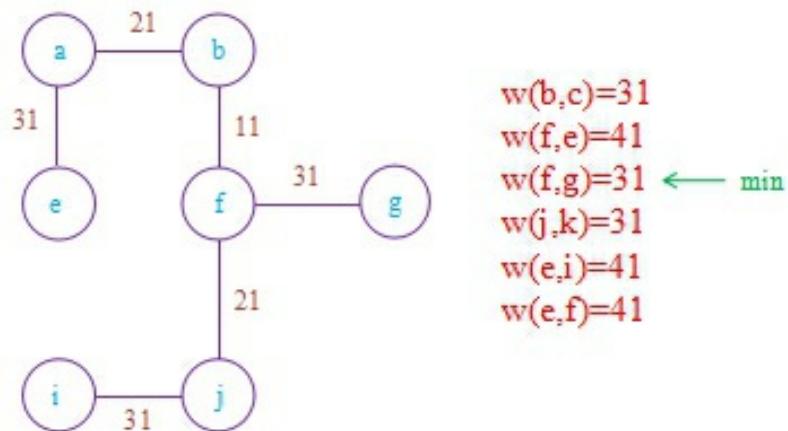
**Step 4:** Similarly, the next edge added is  $\langle a, e \rangle$  with weight 31.



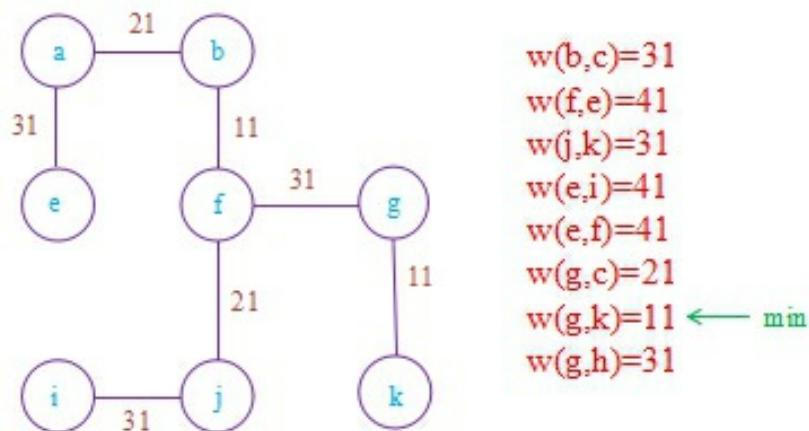
**Step 5:** Edge selected =  $\langle j, i \rangle$  with weight 31.



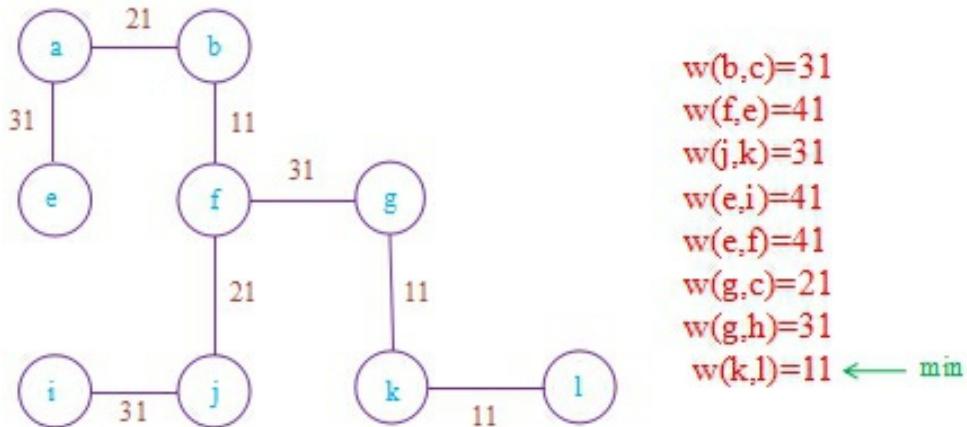
**Step 6:** Edge selected =  $\langle f, g \rangle$  with weight 31.



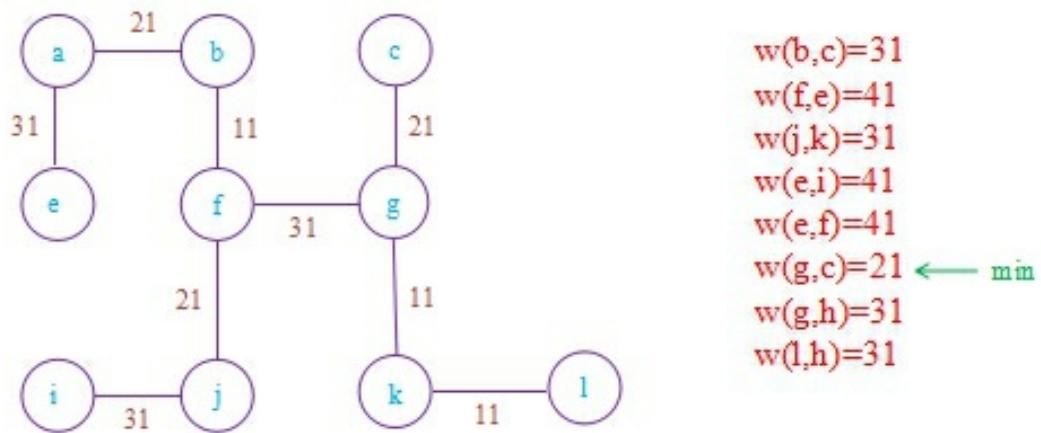
**Step 7:** Edge selected =  $\langle g, k \rangle$  with weight 11.



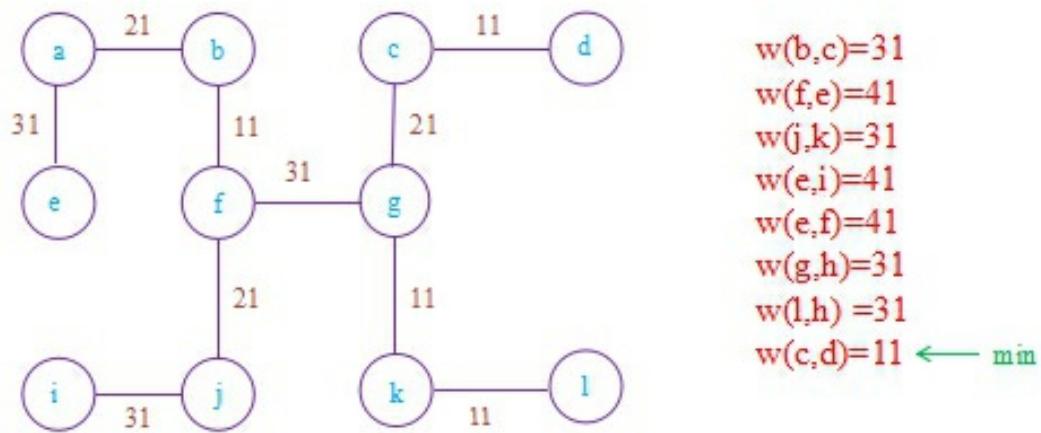
**Step 8:** Edge selected =  $\langle k, l \rangle$  with weight 11.



**Step 9:** Edge selected =  $\langle g, c \rangle$  with weight 21.

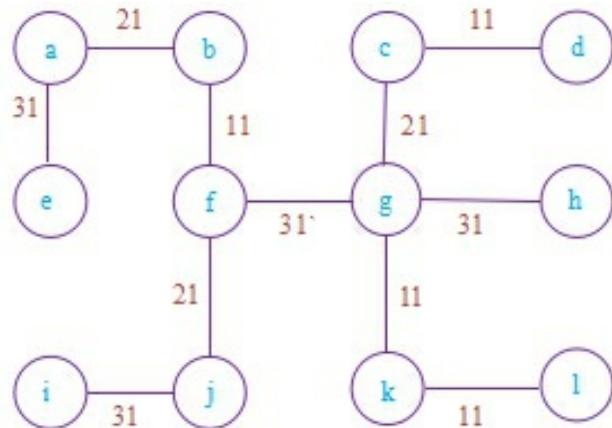


**Step 10:** Edge selected =  $\langle c, d \rangle$  with weight 11.



**Step 11:** Finally the edge selected =  $\langle g, k \rangle$  with weight 31.

As all the vertices are added, the algorithm will end. The resultant spanning tree is shown in the below figure.



The minimum cost spanning tree for the given graph is shown above.

The minimum cost of spanning tree is  $= 11 + 21 + 21 + 31 + 31 + 31 + 11 + 11 + 21 + 11 + 31 = 231$ .

#### 4.3.3.3 Solved Problems

##### PROBLEM 1

Show that in a complete graph of  $n$  vertices, the number of spanning trees generated cannot be greater than  $n^{n-2}$ .

Solution:

Let us consider a complete graph with 3 vertices.

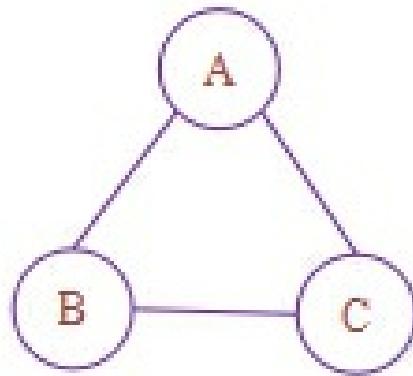


Fig. 4.3.3: Graph ( $K_3$ )

Spanning tree with  $K_3$  trees are,

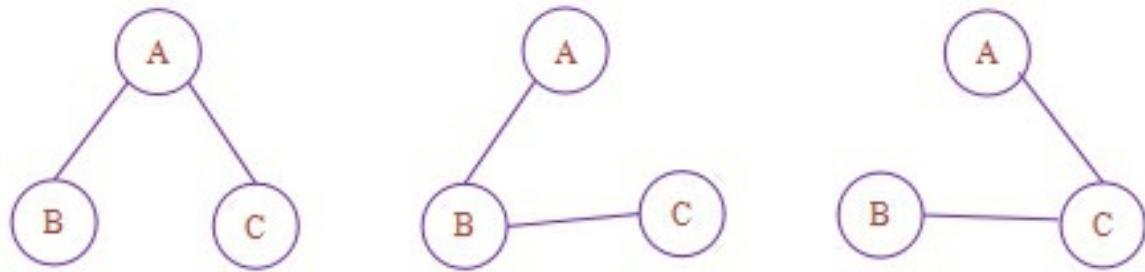


Fig. 4.3.4: Spanning Tree (K<sub>3</sub>)

Since,  $n^{n-2} = 3^{3-2} = 3^1 = 3$ .

Consider one more example.

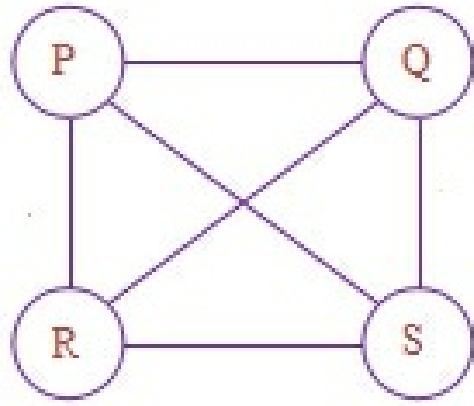


Fig. 4.3.5: Graph (K<sub>4</sub>)

Spanning trees of k<sub>4</sub> are,

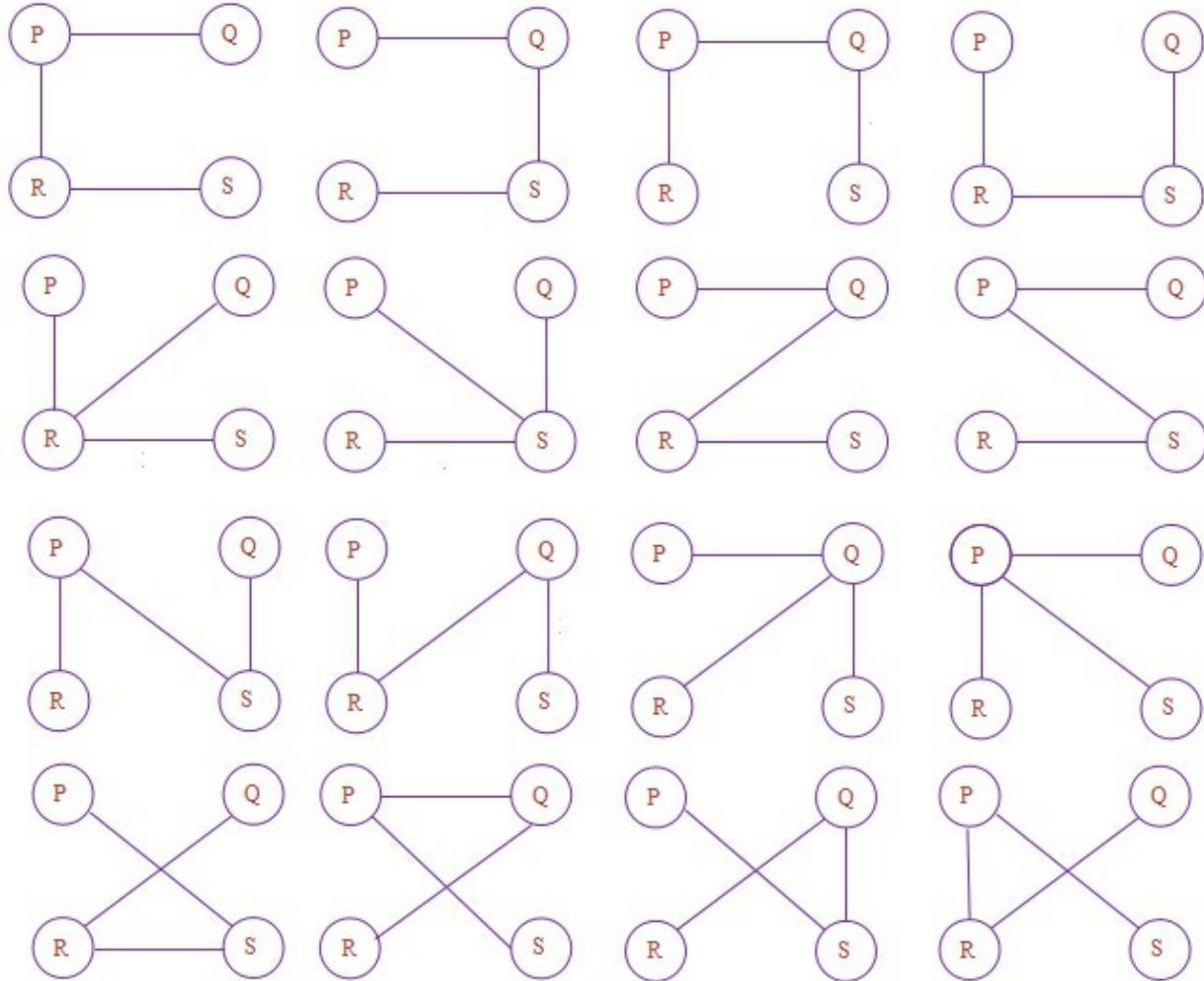


Fig. 4.3.6: Spanning Trees ( $K_4$ )

Since,  $n^{n-2} = 4^{4-2} = 4^2 = 16$ .

Thus, we conclude that a complete graph with  $n$  vertices have  $n^{n-2}$  spanning trees.

### EXAMPLE PROBLEM 2

**Prove that any weighted connected graph with distinct weights has exactly one minimum spanning tree.**

Solution:

Generally this theorem proves two things. They are:

- i. Atleast one minimum spanning tree exists for any weighted connected graph.

- ii. If the weights in a weighted connected graph are all distinct, then the minimum spanning tree is unique.

In first part, since, we know that a finite number of spanning trees exist for any connected graph  $G$  and if some or all weights are distinct, then obviously one minimum spanning tree can be generated.

**Example:**

Consider the following graph.

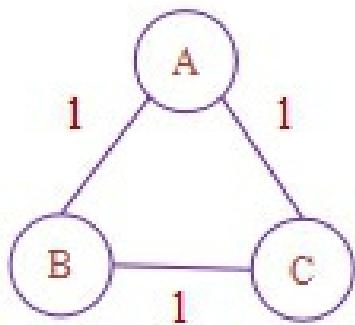


Fig. 4.3.7: Weighted Graph 1

Minimum spanning trees of the above graph are shown below.

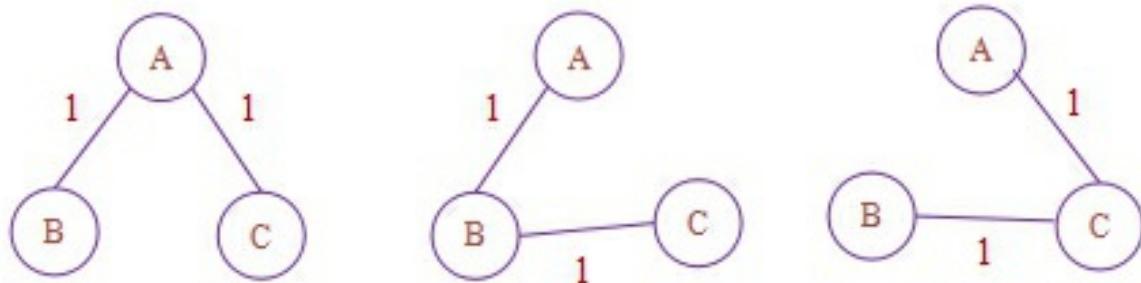


Fig. 4.3.8: Spanning Tree of Graph 1

And if some weights are distinct.

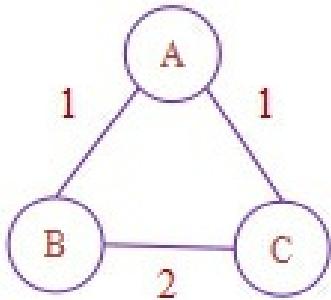


Fig. 4.3.9: Weighted Graph 2

Then there may be atleast one minimum spanning tree as shown in the figure below.

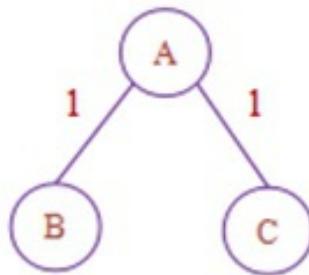


Fig. 4.3.10: Spanning Tree of Weighted Graph 2

Second part also can be proved easily, assume  $T$  is the minimum spanning tree of  $G$ . Suppose there is some edge  $uv$  creates a cycle in which some other edge  $xy$  has weight  $W(xy) > W(uv)$ . Then replacing  $xy$  creates a new spanning tree with total weight less than  $T$ , contradicting the assumption that  $T$  was of minimum weight.

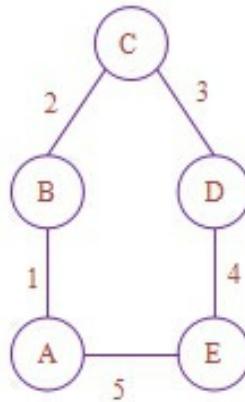
### EXAMPLE PROBLEM 3

**Prove that the edge with the smallest weight will be part of every minimum spanning tree.**

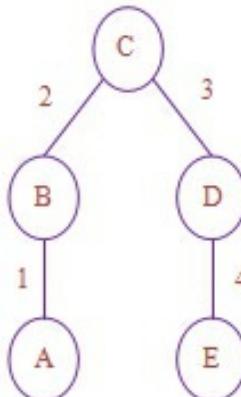
Solution:

**Minimum spanning tree:** A spanning tree is said to be minimum or minimal spanning tree if and only if the total weight of the edge in the tree is small.

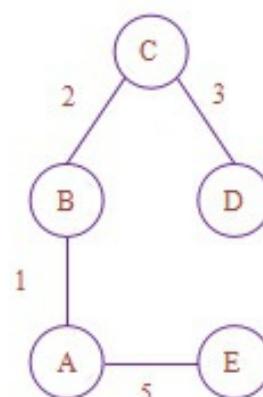
Consider a graph shown below with weight associated with each edge.



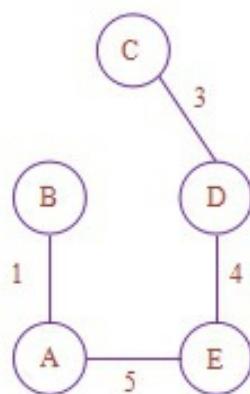
The edge with minimum cost is AB. So we are required to prove that the edge AB is included in the minimum spanning tree. Some of the spanning trees of the above graph are given below.



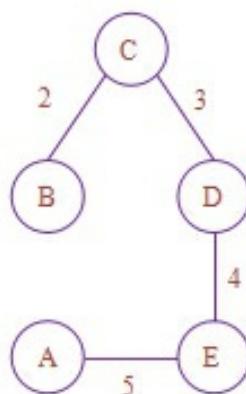
(a) weight = 10



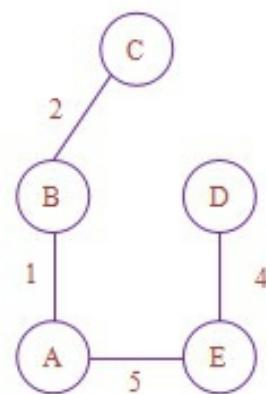
(b) weight = 11



(c) weight = 13



(d) weight = 14



(e) weight = 12

Among all the spanning trees, the tree with minimum edge is shown in the

figure (a), which include the edge AB. Thus, we have proved that the edge with minimum weight among all the edges is included in the minimum spanning tree.

#### **4.3.4 Single Source Shortest Path Problem**

Let  $G = (V, E)$  be a directed graph with weighted function  $w$  for the edges of  $G$ . The starting vertex is called as source and the last vertex is called as the destination vertex. Let  $v$  be any other vertices which belong to the set of vertices  $V$ . The problem to determine a shortest path to given destination vertex  $v$  from the source vertex is called as single source shortest path problem.

To formulate the greedy method algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure. One possibility is to construct the shortest paths one by one. As an optimization measure, we can use the sum of lengths of all paths so far generated. This measure is to be minimized.

The following shown below is the single source shortest path algorithm.

```

Algorithm ShortestPath(v, cost, dist, n)
//dist[j], 1 ≤ j ≤ n, is set to the length of the shortest path from vertex v to vertex j
//in a digraph G with n vertices. Dist[v] is said to zero. G is represented by its
//cost adjacency matrix cost[1:n, 1:n].
{
    for i := 1 to n
        //Initialize S
        S[i] := false; dist [i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0.0; //put v in S.
    for num := 2 to n do
    {
        //Determine n-1 paths from v.
        Choose u from among those vertices not in S such that
        Dist[u] is minimum;
        S[u] := true; //put u in S.
        for (each w adjacent to u with S[w] = false) do
            //Update distances.
            if(dist[w] > dist[u] + cost[u, w])) then
                dist[w] := dist[u] + cost[u, w];
    }
}

```

#### Algorithm 4.3.5: Single Source Shortest Path Algorithm

The first for loop takes  $O(n)$  time. Each execution of second for loop requires  $O(n)$  time to select the next vertex and again at the for loop to update dist so that the total time for this loop is  $O(n^2)$ . Therefore the time complexity for this algorithm is  $O(n^2)$ .

**Example:** Consider the directed graph shown in the below figure.

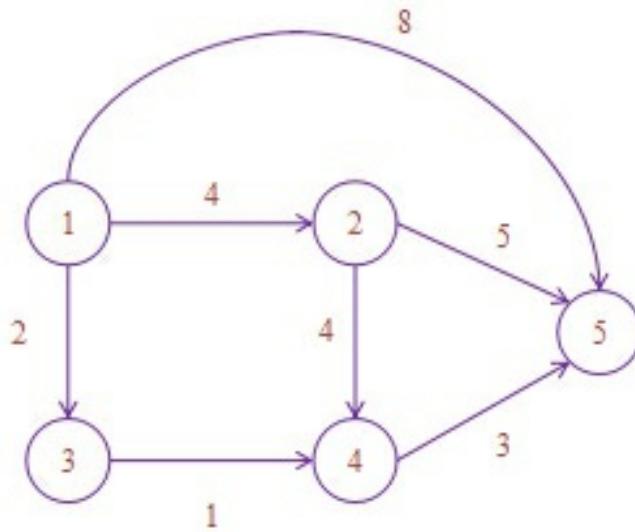


Fig. 4.3.11: Directed Graph

If 1 is the source vertex, then the shortest path from 1 to 5 is 6.

The other shortest paths from vertex 1 to other vertices are shown below.

S. No.	Path	Length
1	1-3	2
2	1-3-4	3
3	1-2	4
4	1-3-4-5	6

The Greedy Method here is to generate shortest path from source vertex to remaining vertices is to generate these path in increasing order of path length.

According to Dijkstra's Algorithm, first we select the source vertex and include that vertex in the set S. To generate the shortest paths from source to the remaining vertices a shortest path to the nearest vertex is generated first and it is included in S. Then a shortest path to the second nearest vertex is generated and so on. To generate these shortest paths we need to determine,

- 1) The next vertex to which a shortest path must be generated and
- 2) A shortest path to this vertex.

#### 4.3.4.1 Solved Problems

##### PROBLEM 1)

Using shortest path algorithm, obtain in non-decreasing order of the lengths of the shortest path from node 1 to all the remaining nodes of given following diagraph.

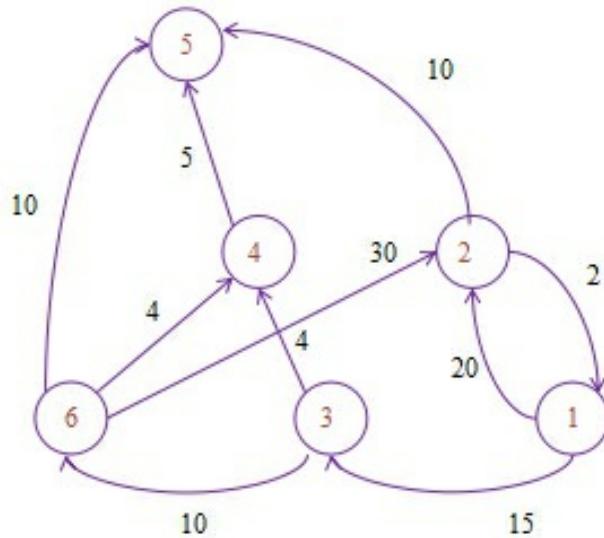


Fig. 4.3.12: Diagraph

Solution:

Set S	Vertex Selected	1	2	3	4	5	6
-	-	0	20	15	$\alpha$	$\alpha$	$\alpha$
{1}	3	0	20	15	19	$\alpha$	25
{1, 3}	4	0	20	15	19	$\alpha$	25
{1, 3, 4}	2	0	20	15	19	30	25
{1, 3, 4, 2}	6	0	20	15	19	30	25
{1, 3, 4, 2, 6}	5	0	20	15	19	30	25

Table 4.3.1: Shortest path from vertex 1

# 5 DYNAMIC PROGRAMMING

## 5.1 INTRODUCTION TO DYNAMIC PROGRAMMING

Dynamic programming is a stage wise search method which is suitable for optimization problem whose solution may be viewed as the result of a sequence of decisions. One of the most attractive property of this strategy is during the search for the solution, it avoids full enumeration by pruning early partial decisions solutions that cannot possibly lead to the optimal solution. In many partial solutions this strategy hits the optimal solution in a polynomial number of decision steps. Therefore, in the worst case in such a strategy may end up performing a full enumeration.

Dynamic programming takes advantage of the duplication and arrange to solve each subproblem only once, saving the solution for later use.

The underlining idea of the dynamic programming is, “Avoid calculating the same stuff twice”. Usually by keeping the table of known results of subproblems. Unlike, divide and conquer which solves the subproblems top down, a dynamic programming is a bottom-up technique. Here, bottom-up approach means

- 1) Start with the smallest subproblems.
- 2) Combine their solutions to obtain the solutions of increasing size.
- 3) Until arrive at the solution of the original problem.

Thus dynamic program is similar to divide and conquer but avoids duplicate work when the problems are identical. Dynamic programming approach is used for optimization problems like travelling salesmen problem, matrix chain multiplication, 0/1 Knapsack problems and all pair shortest problems etc.

### 5.1.1 Key Elements of Dynamic Programming

Dynamic Programming is applied to optimization problems. In order for dynamic programming the two key elements that an optimization problem must have to be applicable.

- 1) **Optimal Substructure:** By dynamic programming, the first in solving an optimization problem is to characterize the structure of optimal solution. Recall that a problem exhibits optimal structure if an optimal

solution to the problem contains within its optimal solution to subproblems.

When any time a problem exhibits an optimal structure, it is a good clue that a dynamic programming might apply. In dynamic programming we build an optimal solution to the problem and from optimal solutions to subproblems. Here, we must take care to ensure that the range of subproblems we consider to includes those used in an optimal solution.

The optimal structure varies across problem domain in two ways. They are:

- a) Number of problems are used in an optimal solution to the original problem.
- b) Number of choices we have in determining which subproblem to use in an optimal solution.

Bottom up fashion is used in dynamic programming. Here, we first find the optimal solutions to subproblems and have to solve the subproblems, we find the optimal solution to the problem. Generally, the cost of the problem solution is the subproblem cost plus a cost that is directly attributable to the choice itself.

## 2) Overlapping Subproblems:

In the dynamic programming, the second element that an optimization problem must have to be applicable is that a space of subproblems must be small. Here, small in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. The total number of distinct subproblems is a polynomial in the input size. We say that the optimization problem has overlapping subproblems. When a recursive algorithm revisits the same problem over and over again, dynamic programming algorithms has the advantage of overlapping subproblems. Typically takes by solving each subproblems once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

Here, the dynamic programming drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal, an optimal sequence of decisions is obtained by using the principle of optimality.

### 5.1.2 Principle of Optimality

According to the principle of optimality, the decisions resulting from the first decisions has a property that they are in the optimal sequence irrespective of the initial state and the decisions. The significance of this principle is that it always give the optimal solution from many feasible solutions. Also optimal solution will be achieved from the initial state itself. Here, we can say that principle of optimality is satisfied. When optimal solution is found for a problem, then optimal solutions are also found for its subproblems as well.

The problem for which principle of optimality does not hold is longest simple path problem. The reason why it does not hold is explained with the help of an example.

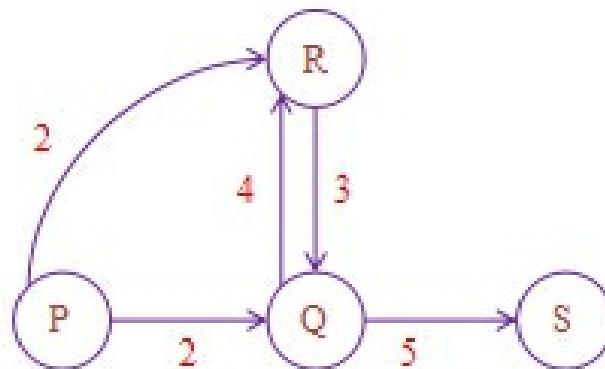


Fig. 5.1.1: A Simple Path

The path should not contain a cycle here. Suppose, we have to find the longest simple path from P to S, there are two paths that goes from P to S.

- 1)  $P \rightarrow Q \rightarrow S$  i.e.,  $2 + 5 = 7$ .
- 2)  $P \rightarrow R \rightarrow Q \rightarrow S$  i.e.,  $2 + 3 + 5 = 10$ .

### 5.1.3 Comparison between Dynamic Programming and Divide and Conquer

S. NO.	Dynamic Programming	Divide and Conquer
1	Dynamic Programming is a method in which the solution to the problem can be viewed	Divide and Conquer is a method in which solution to a problem can be obtained by dividing it into several

	as the result of sequence of decisions.	subproblems until it cannot be divided further and solve it recursively.
2	A dynamic programming algorithm solves every sub-problem once and saves the result in the form of table, from this solution to the problem is obtained.	In Divide and Conquer, every subproblem is solved and their results are combined to get the original solution.
3	Dynamic programming works best when all sub-problems are independent, we don't know where to partition the problem.	Divide and Conquer works best when all the sub-problems are independent.
4	Dynamic programming is best suited for solving problems for overlapping sub-problems.	Divide and Conquer is best suited for the case when no overlapping sub-problems are encountered.
5	Dynamic programming splits its input at every possible split. After trying all possible splits, it determines which split point is optimal.	Divide and Conquer always splits the input into pre-specified deterministic points. Example, always at the middle.

Table 5.1.1: Comparison between Dynamic Programming and Divide and Conquer

#### 5.1.4 Comparison between Dynamic Programming and Greedy Method

S. NO.	Dynamic Programming	Greedy Method
1	Dynamic programming is a method in which the solution to a problem can be viewed as a result of sequence of decisions.	Greedy method is the forward technique for constructing the solution to an optimum problem through a sequence of steps.
2	In this method, more than one decision is made at a time.	In the Greedy method only one decision is made at a time.

	time.	
3	Dynamic programming considers all possible sequences in order to obtain the optimum solution.	Greedy method will consider an optimum solution without considering the previous solutions.
4	Principle of optimality holds in dynamic programming and thus the solution obtained is guaranteed.	Solution obtained is not guaranteed in Greedy Method.
5	Dynamic programming solves the sub-problems bottom up. The problem cannot be solved until we find the solution to the sub-problems.	Greedy method solves the sub-problem top down. We first need to find the Greedy choice for a problem, then reduce it into smaller one.
6	Dynamic programming is expensive than greedy method, because we have to try every possibility before solving a problem.	Greedy method is comparatively less expensive.
7	In this method we can solve any problem.	There are some problems that Greedy method cannot solve while dynamic programming can solve. Therefore first we try Greedy method, if it fails then we need to try dynamic programming.

Table 5.1.2: Comparison between Dynamic Programming and Greedy Method

## 5.2 GENERAL METHOD

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as a result of sequence of decisions.

The fundamental dynamic programming model may be written as

...(1)

$$F_n(R) = \max_{0 \leq R_n \leq R} \{P_n\{R\} + F_{n-1}\{R - R_n\}\}$$

where  $n = 2, 3, 4, \dots$

$$F_n(0) = 0$$

$$F_n(R) = P_1(R)$$

Once  $F_1(R)$  is known eq(1) provides a relation for evaluation of  $F_2(R)$ ,  $F_3(R), \dots$ . This recursive process ultimately leads to the value of  $F_{n-1}(R)$  and finally  $F_n(R)$  at which the process stops.

A dynamic programming problem can be divided into a number of stages, when an optimal decision must be made at each stage. The decision made at each stage influences the decision to be taken next. Here, the decision made at each stage must take into account, its effects not only on the next stage, but also on the entire subsequent stages. Dynamic programming provides a systematic procedure, where it starts with the last stage of the problem and works backwards one makes an optimal decision for each stage of the problem. The information for the last stage is the information derived from the previous stage.

Dynamic programming design involves four major steps, they are shown below.

- 1) Characterize the structure of optimal solution.
- 2) Recursively define the value of the optimal solution.
- 3) Compute the value of an optimal solution in the bottom-up fashion.
- 4) Construct an optimum solution from computed information.

The dynamic programming technique was developed by Bellman based upon his principles known as principle of optimality. This principle states that, “an optimal policy has the property that, whatever the initial decisions are the remaining decisions must continue an optimal policy with regard to the state resulting from the first decision”.

### **5.2.1 General Characteristics of Dynamic Programming**

The general characteristics of dynamic programming are

- 1) The problem can be divided into stages with a policy decision required at each stage.
- 2) Each stage has number of states associated with it.
- 3) Given the current stage an optimal policy for the remaining stages is independent of the policy adopted.
- 4) The solution procedure begins by finding the optimal policy for each state of the last stage.
- 5) A recursive relation is available which identifies the optimal policy for each stage with n-stages. Remaining given the optimal policy for each stage with n-1 stages remaining.

### **5.3 APPLICATIONS OF DYNAMIC PROGRAMMING**

Various applications that can be implemented using dynamic programming are as follows.

#### **5.3.1 Matrix Chain Multiplication**

For the multiplication of n matrices a series of operations are to be performed. A dynamic programming approach gives the optimal sequence or order of operations of this problem.

Here, we have to multiply ‘n’ matrices,  $A_1, A_2, A_3, \dots, A_n$ , which can be accomplished by a series of matrix multiplications.

$$A_1 \times A_2 \times A_3 \times \dots \times A_n$$

We cannot change the above order of multiplication since the matrix multiplication is associative but we can parenthesize the above multiplication.

For example consider the three matrices ( $A_{5 \times 4}, B_{4 \times 3}, C_{3 \times 5}$ )

Multiplication cost of  $[(AB)C] = (5 \times 4 \times 3) + (5 \times 3 \times 5) = 135$

Multiplication cost of  $[(A(BC)] = (4 \times 3 \times 5) + (5 \times 4 \times 5) = 160$

Therefore, we can see from the above illustrations that different evaluation sequence implies different cost of operation.

Here, the problem is to determine the parenthesizing of the expression, defining  $A_1 \times A_2 \times A_3 \times \dots \times A_n$  that minimizes the total number of scalar multiplications performed. In the dynamic programming approach, we divide the problem into several subproblems and then the solutions of these

subproblems are combined to get the solution of the original problem. As we cannot change the order of multiplication, we parenthesize the expression to achieve optimum order of multiplying ‘n’ matrices.

Let  $A_{i-j}$  denote the multiplication of matrices through i to j and the order of  $A_{i-j}$  is  $r_{i-1} \times R_j$ .

We make a number of decisions, our job is to break the problem into several number of subproblems of same kind as the original problem and start parenthesizing the sequences as the situation needs.

Let us consider the highest level of parenthesizing where two matrix multiplications are considered i.e.,  $A_{1...n} = A_{1...k} \cdot A_{k+1...n}$ .

Now, the questions of interest are

- i. How can we split the order of matrices?
- ii. How can we parenthesize the subsequences  $A_{1...k}$  and  $A_{k+1...n}$ .

The answer to the first question is to try out all possible choices of k and pickout the best among them, but it does not lead you to the exponential growth.

The answer to the second question is to try out repeated by parenthesizing depending upon your need.

Another important thing to note about matrix chain multiplication problem is that, it is possible to characterize an optimal solution to a particular problem in terms of optimal solutions to its sub-problems. It does mean that “principle of optimality” applies to this problem.

Dynamic programming approach computes the solutions of the sub-problems and stores them into a table there by avoding the work of recomputing the solution wherever that the sub-problem is encountered.

In order to solve the problem using dynamic programming approach, we need to follow the following steps,

- 1) Let  $N_{ij}$  denotes the minimum number of multiplications needed to compute  $A_{i-j}$  where,  $1 \leq i \leq j \leq n$ .

If the sequence contains only one matrix i.e., if  $i = j$ , then the cost of

operation is 0 i.e.,  $N[i,j]$ .

- 2) If  $i < j$ , then the multiplication for  $A_{i,j}$  is computed by considering each  $k$ .

$$A_{i...j} = A_{i...k} \cdot A_{k+1...j} \text{ where } i \leq k \leq j.$$

$N[i, k]$  and  $N[k+1, j]$  are the optimum multiplications needed to compute  $A_{i...k}$  and  $A_{k+1...j}$ . Since,  $r_{i-1} \times r_k$  is the order of  $k+1...j$ , the total number of operations needed to multiply  $A_{i...k}$  and  $A_{k+1...j}$  is  $r_{i-1} \cdot r_k \cdot r_j$ .

- 3)  $\therefore N[I, j]$  can be computed as,

$$N[i, j] = \min N[i, k] + N[k+1, j] + r_{i-1} \cdot r_k \cdot r_j$$

Where,  $i < j$  and  $i \leq k < j$ .

#### 5.3.1.1 Time Complexity of Matrix Chain Multiplication

The algorithm for chain matrix multiplication using dynamic programming approach is given below,

```

Algorithm MatrixMul(A)

{
    integer i, j, k, b;
    for i := 1 to n do
    {
        N[i, j] := 0 //if i = j i.e., only one matrix.
        for b := 2 to n do
        {
            for i := 1 to n-b+1 do
            {
                j := i + b - 1
                N[i, j] := ∞
                for k := i to j-1 do
                    N[i, j] := Min[N[i, k] + N[k+1, j] + r_{i-1}r_kr_j]
            }
        }
    }
}

```

### Algorithm 5.3.1: Matrix Chain Multiplication

In the above algorithm, there are three nested loops and each loop can iterate atmost ‘n’ times which implies the above algorithm runs in  $O(n^3)$  times.

### 5.3.2 Optimal Binary Search Trees (OBST)

A binary search tree T is a binary tree, either it is empty or each node in the tree contains an identifier and it should satisfy the following conditions.

- 1) All identifiers in the left subtree of tree T are less than (numerically or alphabetically) the identifier in the root node T.
- 2) All identifiers in the right subtree are greater than the identifiers in the root node T.
- 3) The left and right subtree of T are also binary search trees.

If we want to search an element in the binary search tree, first that element is compared with the root node. If element is less than the root node then the search continues in the left subtree. If element is greater than the root node then the search continues in the right subtree. If element is equal to the root node then print the successful search (element found) and terminate search procedure.

To give a set of identifiers, different binary search trees can be created with different performance characteristics. An optimal binary search tree offers the most economical binary search trees with the lowest average cost of searching. This involves consideration that the different identifiers are searched with different probabilities and there may be unsuccessful searches.

Let  $p(i)$  be the probability with which the identifier  $a_i$  is searched in the set of linearly ordered elements in a set i.e.,  $a_1 < a_2 < \dots < a_n$ . Let  $q_i$  be the probability of the identifier X being searched for such that  $a_i < X < a_{i+1}$ ,  $0 \leq i \leq n$ .

Assume that  $a_0 = -\alpha$  and  $a_{n+1} = +\alpha$ .

Then,  $\sum_{0 \leq i \leq n} q(i)$  is said to be the probability of an unsuccessful search.

Then,  $\sum_{1 \leq i \leq n} p(i) + \sum_{0 \leq i \leq n} q(i) = 1$

For the optimal binary search trees in obtaining a cost function the average cost of successful searches is added to the cost of unsuccessful searches. The former is expressed as a function of level of the internal node (the expected cost contribution from the internal node for  $a_i$  is  $p(i) * \text{level}(a_i)$  at which the search stops successful. While for the later, the cost function is defined in terms of the external nodes (the expected cost contribution from the external node for  $a_i$  is  $q(i) * (\text{level}(E_i) - 1)$  where  $E_i$  is level of external node

at level i) in place of every subtree. An external node represents a point where an unsuccessful search may terminate.

$$\begin{aligned} \sum_{1 \leq i \leq n} p(i) * \text{level}(a_i) + & \dots \\ \sum_{0 \leq i \leq n} q(i) * (\text{level}(E_i) - 1) \end{aligned} \quad (1)$$

An optimal binary search tree for the identifier set  $\{a_1, a_2, a_3, \dots, a_n\}$  is a binary search tree for which the equation (1) is minimum. The problem of obtaining optimal binary search tree as the result of sequence of decisions and then observe that the principle of optimality holds when applied to the problem state resulting from the decision.

A possible approach to this would be to make a decision as to which of the  $a_i$  will be assigned to the root node of binary search T. If  $a_k$  is chosen as the root then all the internal nodes  $a_0, a_1, a_2, a_3, \dots, a_{k-1}$  and all the external nodes will lie on the right subtree.

$$\begin{aligned} \text{cost}(l) = & \sum_{1 \leq i \leq k} p(i) * \text{level}(a_i) + \\ & \sum_{0 \leq k \leq n} q(i) * (\text{level}(E_i) - 1) \\ \text{cost}(r) = & \sum_{k \leq i \leq n} p(i) * \text{level}(a_i) + \\ & \sum_{k \leq i \leq n} q(i) * (\text{level}(E_i) - 1) \end{aligned}$$

To represent the sum used  $\omega(i, j) = Q(i) + \sum_{i+1}^j q(1) + p(i)$ . Then the expected cost of the search tree T is obtained as

$$p(k) + \text{cost}(l) + \text{cost}(r) + \omega(0, k-1) + \omega(k, n) \quad \dots \quad (2)$$

If T is optimal then equation (2) must be minimum. If  $C(i, j)$  is used to represent the cost of the optimal binary search tree  $T(i, j)$  containing  $a_{i+1}, \dots, a_j$ ; and  $E_i \dots E_j$  for T to be optimal.

Q is necessary to have,

$$\text{cost}(l) = C(0, k-1) \text{ and } \text{cost}(r) = C(k, n)$$

In addition, k must be chosen that,

$p(k) + \text{cost}(l) + \text{cost}(r) + \omega(0, k-1) + \omega(k, n)$  is minimum.

Hence,

$$C(0, n) = \min_{1 \leq k \leq n} \{ C(0, k-1) + C(k, n) + P(k) + \omega(0, k-1) + \omega(k, n) \} \quad (3)$$

From the equation (3), we can generalize any  $C(i, j)$ .

$$C(i, j) = \min_{1 \leq k \leq j} \{ C(i, k-1) + C(k, j) + \omega(i, j) \}$$

The equation  $C(0, n)$  may be solved by first computing all  $C(i, 0)$  for which  $j-i=1$ . Then  $C(i, j)$  for all  $j-i=2$  and so on.

Note that  $\omega(i, i) = Q(i)$ ,  $C(i, i) = 0$ .

If during the computation, the root  $R(i, j)$  is recorded, for each tree  $T_{ij}$  then an optimal binary search tree may be constructed from the  $C(i, j)$ .

### 5.3.2.1 Time Complexity of Optimal Binary Search Trees Algorithm

Algorithm for optimal binary search tree is as follows,

```

Algorithm OBST(p, q, n)

{
    for i := 0 to n-1 do
    {
        //initiallize
        w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;

        //Optimal trees with one node
        w[i, i+1] := q[i] + q[i+1] + p[i+1];
        r[i, i+1] := i+1;
        c[i, i+1] := q[i] + q[i+1] + p[i+1];
    }

    w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;

    for m := 2 to n do //optimal trees with m nodes.

        for i := 0 to n-m do
        {
            j := i + m;
            w[i, j] := w[i, j-1] + p[j] + q[j];
            k := Find(c, r, i, j);

            //A value of l in the range r[i, j-1] ≤ l ≤ r[i+1, j] that minimizes c[i, l-1] + c[i, j];
            c[i, j] := w[i, j] + c[i, k-1] + c[k, j];
            r[i, j] := k;
        }

        write (c[0, n], w[0, n], r[0, n]);
    }
}

```

Algorithm 5.3.2: Optimal Bnary Search Trees

### 5.3.2.2 Solved Problems

#### Problem 1:

Using algorithm OBST compute  $w(i, j)$ ,  $r(i, j)$  and  $c(i, j)$ ,  $0 \leq i \leq j \leq 4$  for the identifier set  $(a_1, a_2, a_3, a_4) = (\text{and}, \text{goto}, \text{print}, \text{stop})$  with  $p(1) = 3, p(2) = 3, p(3) = 1, p(4) = 1, q(0) = 2, q(1) = 3, q(2) = 1, q(3) = 1, q(4) = 1$  using  $r(i, j)$  construct the optimal binary search tree.

#### Solution:

Initially,  $c(i, j) = 0, r(i, j) = 0, 0 \leq i \leq 4$

$$w(i, j) = q(i)$$

$$w(0, 0) = q(0) = 2$$

$$w(1, 1) = q(1) = 3$$

$$w(2, 2) = q(2) = 1$$

$$w(3, 3) = q(3) = 1$$

$$w(4, 4) = q(4) = 1$$

Now,

$$C(i, j) = \min_{i < k \leq j} \{C(i, k-1) + C(k, j) + \omega(i, j)\}$$

$$w(i, j) = p(j) + q(j) + w(i, j-1)$$

$$r_{ij} = k, i < k \leq j \quad (k \text{ is chosen such that where cost is minimum})$$

#### For $j-i = 1$

$$w(0, 1) = p(1) + q(1) + w(0, 0) = 3 + 3 + 2 = 8$$

$$c(0, 1) = \min_{0 < k \leq 1} \{c(0, 0) + c(1, 1)\} + w(0, 1) = 0 + 0 + 8 = 8$$

$$r(0, 1) = 1$$

$$w(1, 2) = p(2) + q(2) + w(1, 1) = 3 + 1 + 3 = 7$$

$$c(1, 2) = \min_{1 < k \leq 2} \{c(1, 1) + c(2, 2)\} + w(1, 2) = 0 + 0 + 7 = 7$$

$$r(1, 2) = 2$$

$$w(2, 3) = p(3) + q(3) + w(2, 2) = 1 + 1 + 1 = 3$$

$$c(2, 3) = \min_{2 < k \leq 3} \{c(2, 2) + c(3, 3)\} + w(2, 3) = 0 + 0 + 3 = 3$$

$$r(2, 3) = 3$$

$$w(3, 4) = p(4) + q(4) + w(3, 3) = 1 + 1 + 1 = 3$$

$$c(3, 4) = \min_{3 < k \leq 4} \{c(3, 3) + c(4, 4)\} + w(3, 4) = 0 + 0 + 3 = 3$$

$$r(3, 4) = 4$$

**For j-i = 2**

$$w(0, 2) = p(2) + q(2) + w(0, 1) = 3 + 1 + 8 = 12$$

$$\begin{aligned} c(0, 2) &= \min_{0 < k \leq 2} \{c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)\} + w(0, 2) \\ &= \min\{0 + 7, 8 + 0\} + 12 = 7 + 12 = 19 \end{aligned}$$

$$r(0, 2) = 1$$

$$w(1, 3) = p(3) + q(3) + w(1, 2) = 1 + 1 + 7 = 9$$

$$\begin{aligned} c(1, 3) &= \min_{1 < k \leq 3} \{c(1, 1) + c(3, 3), c(1, 2) + c(3, 3)\} + w(1, 3) \\ &= \min\{3, 7\} + 9 = 3 + 9 = 12 \end{aligned}$$

$$r(1, 3) = 2$$

$$w(2, 4) = p(4) + q(4) + w(2, 3) = 1 + 1 + 3 = 5$$

$$\begin{aligned} c(2, 4) &= \min_{2 < k \leq 4} \{c(2, 2) + c(3, 4), c(2, 3) + c(4, 4)\} + w(2, 4) \\ &= \min\{0 + 3, 3 + 0\} + 5 = 3 + 5 = 8 \end{aligned}$$

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

**For j-i = 3**

$$w(0, 3) = p(3) + q(3) + w(0, 2) = 1 + 1 + 12 = 14$$

$$\begin{aligned} c(0, 3) &= \min_{0 < k \leq 3} \{c(0, 0) + c(1, 3), c(0, 1) + c(2, 3), c(0, 2) + c(3, 3)\} + \\ &\quad w(0, 3) \\ &= \min\{12, 11, 19\} + 14 = 11 + 14 = 25 \end{aligned}$$

$$r(0, 3) = 2$$

$$w(1, 4) = p(4) + q(4) + w(1, 3) = 1 + 1 + 9 = 11$$

$$\begin{aligned}
c(1, 4) &= \min_{1 < k \leq 4} \{c(1, 1) + c(2, 4), c(1, 2) + c(3, 4), c(1, 3) + c(4, 4)\} + w(1, 4) \\
&= \min\{0 + 8, 7 + 3, 12 + 0\} + 11 = 8 + 11 = 19
\end{aligned}$$

$$r(1, 4) = 2$$

**For j-i = 4**

$$w(0, 4) = p(4) + q(4) + w(0, 3) = 1 + 1 + 14 = 16$$

$$\begin{aligned}
c(0, 4) &= \min_{0 < k \leq 4} \{c(0, 0) + c(1, 4), c(0, 1) + c(2, 4), c(0, 2) + c(3, 4), \\
&\quad c(0, 3) + c(4, 4)\} + w(0, 4) \\
&= \min\{19, 16, 22, 25\} + 16 = 16 + 16 = 32
\end{aligned}$$

$$r(0, 4) = 2$$

		w(0,0)=2 c(0,0)=0 r(0,0)=0	w(1,1)=3 c(1,1)=0 r(1,1)=0	w(2,2)=1 c(2,2)=0 r(022)=0	w(3,3)=1 c(3,3)=0 r(3,3)=0	w(4,4)=1 c(4,4)=0 r(4,4)=0
		w(0,1)=8 c(0,1)=8 r(0,1)=1	w(1,2)=7 c(1,2)=7 r(1,2)=2	w(2,3)=3 c(2,3)=3 r(2,3)=3	w(3,4)=3 c(3,4)=3 r(3,4)=4	X
		w(0,2)=12 c(0,2)=19 r(0,2)=1	w(1,3)=9 c(1,3)=12 r(1,3)=2	w(2,4)=5 c(2,4)=8 r(2,4)=3	X	X
		w(0,3)=14 c(0,3)=25 r(0,3)=2	w(1,4)=11 c(1,4)=19 r(1,4)=2	X	X	X
		w(0,4)=16 c(0,4)=32 r(0,4)=2	X	X	X	X

Table 5.3.1: Computation of  $c(0, 4)$ ,  $w(0, 4)$  and  $r(0, 4)$

To build OBST,  $r(0, 4) = 2 \Rightarrow k = 2$

Hence  $a_2$  becomes the root node.

Let  $T$  be OBST,  $T_{i,j} = T_{i,k-1}, T_{k,j}$

$T_{0,4}$  is divided into two parts  $T_{0,1}$  and  $T_{2,4}$

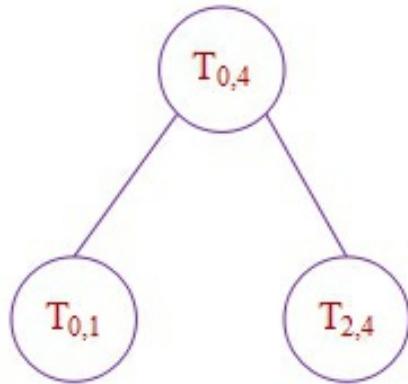


Fig. 5.3.2:  $T_{0,4}$  is divided into two parts  $T_{0,1}$  and  $T_{2,4}$

$$T_{0,1} = r(0, 1) = 1 \implies k = 1$$

$$T_{2,4} = r(2, 4) = 3 \implies k = 3$$

$T_{0,1}$  is divided into two parts as  $T_{0,0}$  and  $T_{1,1}$ . ( $\because k = 1$ )

Again  $T_{3,4}$  is divided into  $T_{3,3}$  and  $T_{4,4}$ . ( $\because k = 4$ )

Since  $r_{00}, r_{11}, r_{22}, r_{33}, r_{44}$  is 0, these are external nodes and can be neglected.

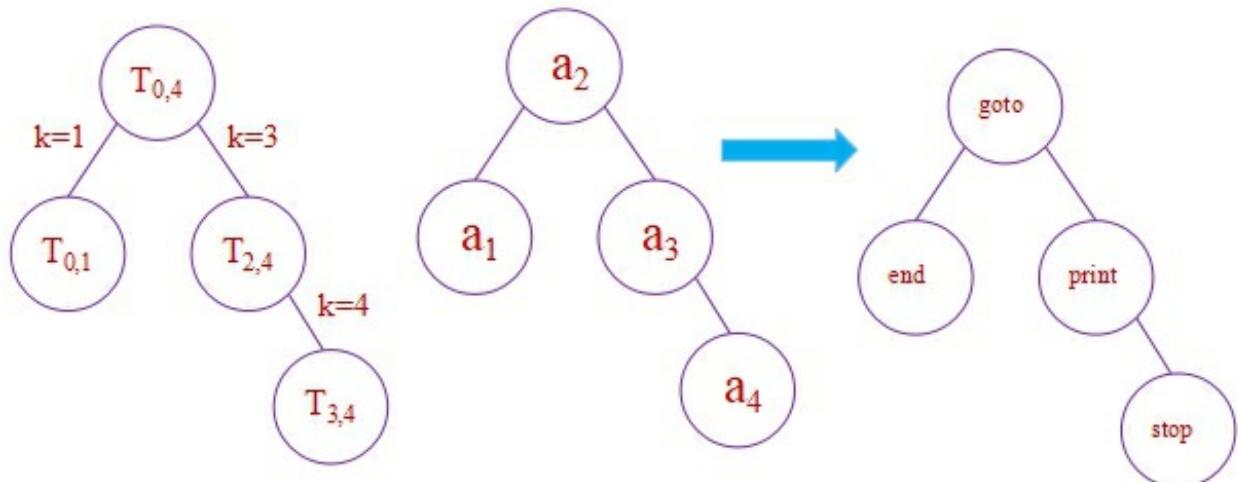


Fig. 5.3.3: Optimal Binary Search Tree

The cost of optimal binary search tree is 32 and the root is  $a_2$ .

### 5.3.3 0/1 Knapsack Problem

Initially, we will compare Knapsack problem with general problem. In olden days, there was a store, assume that, which contains different types of ornaments, which are made up of gold. Let  $n_1, n_2, n_3$  be ornaments, cost and weight of these ornaments are  $c_1, c_2, c_3$  dollars and  $w_1, w_2, w_3$  having weights respectively. Now, a thief wants to rob the ornaments such that he should get maximum profit. In this the thief cannot place the fraction of ornaments in the bag i.e., either he can place ornament completely in the bag or he cannot place ornament. So,  $x_i = 0$  or  $1$ .

- 1) If  $x_i = 0$ , means we cannot place ornament in the bag.
- 2) If  $x_i = 1$  means we can place ornament completely in the bag.

This problem contains either 0 or 1, that way this problem is called as 0/1 Knapsack problem. In this we cannot place fractional weights in the Knapsack. Here, either no item is to be put or the complete item should be in the bag. The fraction of item cannot be placed in the bag. Consider weights  $w_1, w_2, w_3, \dots, w_n$  and fraction of weights to be put in the bag should be  $x_1, x_2, \dots, x_n$ ,  $x_i = 0$  or  $1$ . The dynamic programming solution for 0/1 Knapsack is

$$F_n(M) = \text{Max}[F_{n-1}(M), F_{n-1}(M-w_n) + P_n]$$

$\downarrow$   
 where,  $x_n = 1$

$\downarrow$   
 when  $x_n = 0$

Where,  $M$  = Size of Knapsack.

When  $x_n = 1$ , then the size of the bag is reduced by  $w_n$  which is the weight of the  $n^{\text{th}}$  item. As we are placing the  $n^{\text{th}}$  item we should add the profit for the last item.

Similarly, we find for  $F_{n-1}(M)$  and so on upto  $F_1(M)$

$$S_1^i = \{(P_i, w_i) / (P - P_i, W - w_i) \in S^{i-1}\}$$

Here,  $S_1^i$  is the set of all pairs of  $F_1$  including  $(0,0)$  and  $F_i$  is completely defined by the pairs  $(p_i, w_i)$ , where  $p_i$  is the profit earned on the object  $i$  and  $w_i$  is the weight of the object  $i$ .  $S^i$  is obtained by merging together  $S^{i-1}$  and  $S_1^i$ . This merge corresponds to taking the maximum of two functions  $F_{i-1}(x)$  and  $F_{i-1}(X-w_i) + P_i$  in the objective function of 0/1 Knapsack problem.

Thus, if one of  $S^{i-1}$  and  $S_1^i$  has a pair  $(p_j, w_j)$  and the other has a pair  $(p_k, w_k)$  and  $p_j \leq p_k$ . While  $w_j \geq w_k$ . Then the pair  $(p_j, w_j)$  is discarded. This rule is called as purging rule. When generating  $S^i$  all the pairs  $(p_j, w)$  with  $w > m$  may also be purged.

### 5.3.3.1 Time Complexity of 0/1 Knapsack Problem

Algorithm for 0/1 Knapsack problem is as follows.

```

Algorithm DKP(p, w, n, M)
{
    S0 := {(0, 0)};
    for i := 1 to n-1 do
    {
        S1i = {(Pi, wi) / (P-Pi, W-wi) ∈ Si-1 and W ≤ m};
        S1i = MergePurge(Si-1, S1i);
    }
    (PX, WX) := last pair in Sn-1;
    (PY, WY) := (Pi + pn, Wi + wn) where Wi is the largest W in any pair in Sn-1
    such that W + wn ≤ m;
    //Trace back for xn, xn-1, ..., x1.
    if(PX > PY) then xn := 0;
    else xn := 1;
    TraceBackFor(xn-1, ..., x1);
}

```

### Algorithm 5.3.3: Informal Knapsack Algorithm

The timecomplexity of 0/1 knapsack algorithm using dynamic programming is  $O(n^2)$ . In worst case, time complexity of 0/1 Knapsack problem is  $O(2^{n/2})$ .

#### 5.3.3.2 Solved Problems

##### Problem:

Solve the following 0/1 Knapsack problem using dynamic programming

$$n = 3, M = 6, (P_1, P_2, P_3) = (1, 2, 5), (w_1, w_2, w_3) = (2, 3, 4).$$

Solution:

Consider  $n = 3, M = 6$ ,

$$(w_1, w_2, w_3) = (2, 3, 4) \text{ and } (P_1, P_2, P_3) = (1, 2, 5)$$

$$S_0^0 = (0, 0)$$

$$S_1^i = S^{i-1} + (P_i, w_i)$$

$$S_1^1 = S^0 + (P_1, w_1)$$

$$\begin{aligned} &= \{(0, 0)\} + \{(1, 2)\} \\ &= \{(1, 2)\} \end{aligned}$$

$$S^1 = S^0 + S_1^1$$

$$\begin{aligned} &= \{(0, 0)\} + \{(1, 2)\} \\ &= \{(0, 0), (1, 2)\} \end{aligned}$$

$$S_1^2 = S^1 + (P_2, w_2)$$

$$\begin{aligned} &= \{(0, 0), (1, 2)\} + (2, 3) \\ &= \{(0, 0), (1, 2)\} + \{(2, 3)\} \\ &= \{(2, 3), (3, 5)\} \end{aligned}$$

$$S^2 = S^1 + S_1^2$$

$$\begin{aligned} &= \{(0, 0), (1, 2)\} + \{(2, 3), (3, 5)\} \\ &= \{(0, 0), (1, 2), (2, 3), (3, 5)\} \end{aligned}$$

$$S_1^3 = S^2 + (P_3, w_3)$$

$$= \{(0, 0), (1, 2), (2, 3), (3, 5)\} + (5, 4)$$

$$= \{(5, 4), (6, 6), (7, 7), (8, 9)\}$$

$$S^3 = S^2 + S_1^3$$

$$= \{(0, 0), (1, 2), (2, 3), (3, 5)\} + \{(5, 4), (6, 6), (7, 7), (8, 9)\}$$

$$= \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)\}$$

**Purging Rule (Dominance Rule):** If one of  $S^{i-1}$  and  $S_1^i$  has a pair  $(P_j, w_j)$  and other has a pair  $(P_k, w_k)$  and  $P_j \leq P_k$  while  $w_j \geq w_k$  then the pair  $(P_j, w_j)$  is discarded.

$$S^3 = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)\}$$

After applying Purging rule, we will check the following condition in order to find solution.

If  $(P_i, w_i) \in S^n$  and  $(P_i, w_i) \notin S^{n-1}$  then,

$$x_n = 1$$

Otherwise,  $x_n = 0$

$$(6, 6) \in S^3$$

$$(6, 6) \notin S^2$$

$$x_3 = 1$$

$$(6, 6) - (5, 4) = (1, 2) \in S^2$$

$$(1, 2) \notin S^1 \rightarrow \text{False}$$

$$x_2 = 0$$

$$(1, 2) \in S^1$$

$$(1, 2) \notin S^0 \rightarrow \text{True}$$

$$x_1 = 1$$

$$\therefore x_1 = 1, x_2 = 0, x_3 = 1$$

Maximum Profit is,

$$\begin{aligned}\sum P_i x_i &= P_1 x_1 + P_2 x_2 + P_3 x_3 \\ &= 1 \times 1 + 2 \times 0 + 5 \times 1 \\ &= 1 + 5 \\ &= 6\end{aligned}$$

Here,  $(P_i, w_i) = (6, 6)$  in the above problem so,  $(P_i, w_i) \notin S^2$ . Condition becomes true. So,  $x_3 = 1$  i.e., 3<sup>rd</sup> item is placed entirely in bag. To get next item to be placed in bag, subtract  $(P_3, w_3)$  from  $(6, 6)$  because  $w_3$  is placed in bag. If  $(1, 2)$  belong to  $S^{2-1} = S^1$ . Then, here the condition becomes false so,  $x_2 = 0$ . No quantity of second item can be placed in bag. In order to get next item to be placed in bag, check  $(P_1, w_1) \in (1, 2), (1, 2) \notin S^0$ , here the condition becomes true. So,  $x_1 = 1$ .

The optimal solution is  $(x_1, x_2, x_3) = (1, 0, 1)$

Maximum Profit is,

$$\sum P_i x_i = 6.$$

### 5.3.4 All Pairs Shortest Problem

Let  $G = (V, E)$  be the directed graph consists of  $n$  vertices and each edge is associated with a weight. The problem of finding the shortest path between all pairs of vertices in a graph is called as shortest path problem. This problem is solved by using dynamic programming technique. The all pairs shortest path problem is to determine a matrix  $A$  such that  $A(i, j)$  is the length of the shortest path from vertex  $i$  to vertex  $j$ . Assume that the path contains no cycles. If  $k$  is the intermediate vertex on this path, then the subpaths from  $i$  to  $k$  and from  $k$  to  $j$  is not shortest path. If  $k$  is the intermediate vertex on this path, then the subpaths from  $i$  to  $k$  is the shortest path going through no vertex with index grater than  $k-1$ . Similarly, the path  $k$  to  $j$  is the shortest path which goes through no vertex with index grater than  $k-1$ .

The shortest path can be computed using following recursive method.

$$A^k(i, j) = W(i, j), \text{ if } k = 0$$

$$A^k(i, j) = \min\{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}, \text{ if } k \geq 1$$

#### **5.3.4.1 Time Complexity of All Pairs Shortest Path Problem Algorithm**

The Floyd's algorithm for all pairs shortest path problem is as follows.

```

Algorithm ShortestPath(W, A, n)
//W is the weighted array matrix, n is the number of vertices
//A is the cost of shortest path from vertex i to j.
{
    for i := 1 to n do
    {
        for j := 1 to n do
        {
            A[i, j] := W(i, j);
        }
    }
    for k := 1 to n do
    {
        for i := 1 to n do
        {
            for j := 1 to n do
            {
                A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
            }
        }
    }
}

```

Algorithm 5.3.4: All Pairs Shortest Path Problem

The time complexity for this method is  $O(n^3)$ .

### 5.3.4.2 Solved Problems

#### Problem:

Solve the all pairs shortest path problem for the following graph.

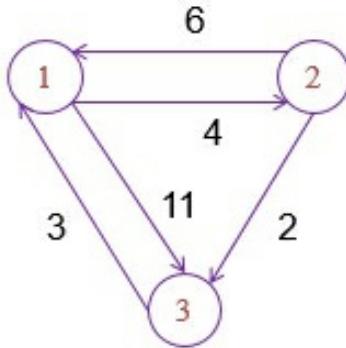


Fig. 5.3.4: Graph G

#### Solution:

Cost adjacency matrix,

$$A^0(i, j) = W(i, j) = \begin{matrix} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{matrix}$$

**Step 1:** For  $k = 1$ , i.e., going from  $i$  to  $j$  with intermediate vertex ‘1’.

$i = 1$ ,

$$A^1(1, 1) = \min\{A^0(1, 1), A^0(1, 1) + A^0(1, 1)\} = 0$$

$$A^1(1, 2) = \min\{A^0(1, 2), A^0(1, 1) + A^0(1, 2)\} = \min\{4, 0 + 4\} = 4$$

$$A^1(1, 3) = \min\{A^0(1, 3), A^0(1, 1) + A^0(1, 3)\} = \min\{11, 0 + 11\} = 11$$

$i = 2$ ,

$$A^1(2, 1) = \min\{A^0(2, 1), A^0(2, 1) + A^0(1, 1)\} = \min\{6, 6 + 0\} = 6$$

$$A^1(2, 2) = \min\{A^0(2, 2), A^0(2, 1) + A^0(1, 2)\} = \min\{0, 6 + 4\} = 0$$

$$A^1(2, 3) = \min\{A^0(2, 3), A^0(2, 1) + A^0(1, 3)\} = \min\{2, 6 + 11\} = 2$$

$i = 3$ ,

$$A^1(3, 1) = \min\{A^0(3, 1), A^0(3, 1) + A^0(1, 1)\} = \min\{3, 3 + 0\} = 3$$

$$A^1(3, 2) = \min\{A^0(3, 2), A^0(3, 1) + A^0(1, 2)\} = \min\{\alpha, 3 + 4\} = 7$$

$$A^1(3, 3) = \min\{A^0(3, 3), A^0(3, 1) + A^0(1, 3)\} = \min\{0, 3 + 11\} = 0$$

**Step 2:** For  $k = 2$ , i.e., going from  $i$  to  $j$  with intermediate vertex ‘2’.

$i=1$

$$A^2(1, 1) = \min\{A^1(1, 1), A^1(1, 2) + A^1(2, 1)\} = \min\{0, 4 + 6\} = 0$$

$$A^2(1, 2) = \min\{A^1(1, 2), A^1(1, 2) + A^1(2, 2)\} = \min\{4, 4 + 0\} = 4$$

$$A^2(1, 3) = \min\{A^1(1, 3), A^1(1, 2) + A^1(2, 3)\} = \min\{11, 4 + 2\} = 6$$

$i=2$

$$A^2(2, 1) = \min\{A^1(2, 1), A^1(2, 2) + A^1(1, 1)\} = \min\{6, 0 + 6\} = 6$$

$$A^2(2, 2) = \min\{A^1(2, 2), A^1(2, 2) + A^1(2, 2)\} = \min\{0, 0 + 0\} = 0$$

$$A^2(2, 3) = \min\{A^1(2, 3), A^1(2, 2) + A^1(2, 3)\} = \min\{2, 0 + 2\} = 2$$

$i=3$

$$A^2(3, 1) = \min\{A^1(3, 1), A^1(3, 2) + A^1(2, 1)\} = \min\{3, \alpha + 6\} = 3$$

$$A^2(3, 2) = \min\{A^1(3, 2), A^1(3, 2) + A^1(2, 2)\} = \min\{7, 7 + 0\} = 7$$

$$A^2(3, 3) = \min\{A^1(3, 3), A^1(3, 2) + A^1(2, 3)\} = \min\{0, 7 + 2\} = 0$$

$$\therefore A^2(i, j) = \begin{matrix} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \left[ \begin{matrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{matrix} \right] \end{matrix}$$

**Step 3:** For  $k = 3$ , i.e., going from  $i$  to  $j$  with intermediate vertex ‘3’.

$i=1$

$$A^3(1, 1) = \min\{A^2(1, 1), A^2(1, 3) + A^2(3, 1)\} = \min\{0, 6 + 3\} = 0$$

$$A^3(1, 2) = \min\{A^2(1, 2), A^2(1, 3) + A^2(3, 2)\} = \min\{4, 6 + 7\} = 4$$

$$A^3(1, 3) = \min\{A^2(1, 3), A^2(1, 3) + A^2(3, 3)\} = \min\{6, 6 + 0\} = 6$$

i=2

$$A^3(2, 1) = \min\{A^2(2, 1), A^2(2, 3) + A^2(3, 1)\} = \min\{6, 2 + 3\} = 5$$

$$A^3(2, 2) = \min\{A^2(2, 2), A^2(2, 3) + A^2(3, 2)\} = \min\{0, 2 + 7\} = 0$$

$$A^3(2, 3) = \min\{A^2(2, 3), A^2(2, 3) + A^2(3, 3)\} = \min\{2, 2 + 0\} = 2$$

i=3

$$A^3(3, 1) = \min\{A^2(3, 1), A^2(3, 3) + A^2(3, 1)\} = \min\{3, 0 + 3\} = 3$$

$$A^3(3, 2) = \min\{A^2(3, 2), A^2(3, 3) + A^2(3, 2)\} = \min\{7, 0 + 7\} = 7$$

$$A^3(3, 3) = \min\{A^2(3, 3), A^2(3, 3) + A^2(3, 3)\} = \min\{0, 0 + 0\} = 0$$

$$\therefore A^3(i, j) = \begin{matrix} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \left[ \begin{matrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{matrix} \right] \end{matrix}$$

### Alternative Method:

$$A^0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \left[ \begin{matrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{matrix} \right] \end{matrix}$$

**Step 1:** Through vertex 1

$$\therefore A^1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \left[ \begin{matrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{matrix} \right] \end{matrix}$$

$$\text{Here, } 3 \rightarrow 2 = (3 \rightarrow 1) + (1 \rightarrow 2) = 3 + 4 = 7$$

$$\min(\infty, 7) = 7$$

### Step 2: Through vertex 2

$$\therefore A^2 = \begin{matrix} & 1 & 2 & 3 \\ 1 & 0 & 4 & \textcircled{6} \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{matrix}$$

Here,  $1 \rightarrow 3 = (1 \rightarrow 2) + (2 \rightarrow 3) = 4 + 2 = 6$

$$\min(6, 11) = 6$$

### Step 3: Through vertex 3

$$\therefore A^3 = \begin{matrix} & 1 & 2 & 3 \\ 1 & 0 & 4 & 6 \\ 2 & \textcircled{5} & 0 & 2 \\ 3 & 3 & 7 & 0 \end{matrix}$$

Here,  $2 \rightarrow 1 = (2 \rightarrow 3) + (3 \rightarrow 2) = 2 + 3 = 5$

$$\min(5, 6) = 5$$

### 5.3.5 Travelling Sales Person Problem

The travelling salesman should start at a point and travels all the places and comes back to the starting point. The problem is to minimize the travelling cost. The main requirement here is there should be communication between nodes.

Suppose we have to route a postal van to pick up mail from mail boxes located at  $n$  different sites. An  $n + 1$  vertex graph may be used to represent this situation. One vertex represent the post office from which the postal van starts and to which it must return. The route taken by the postal van is a tour and it should have minimum length (minimum cost).

- 1)  $g(i, \emptyset) = C_{i,1}, 1 \leq i \leq n$
- 2)  $g(i, S) = \min_{j \in S} \{C_{ij} + g(j, S - \{j\})\}$

Here,  $g(i, S)$  is a graph in which  $i$  means a starting node and the node in  $S$  are to be traversed.  $j \in S$  is considered as the intermediate node.  $g(j, S - \{j\})$  means  $j$  is already traversed. So, the next vertex we have to traverse  $S - \{j\}$  with  $j$  as starting point.

Travelling sales person problem can be applied in the situation where in which we want to tighten the nuts of a machinery by making use of robot arm (i.e., using robots for the purpose of fixing nuts of the machinery during the time of assembling). In this situation, the robot arm rotates from a particular nut and traverse among the remaining nuts of that machine. The scenario can clearly expressed in terms of graph tour where nuts represent the vertices and the path followed represents the edges. If TSP is applied then, we can determine the minimal cost tour by measuring the time taken by all the nut fixing and subsequently choosing the optimal one (i.e., Path).

**Example:** Consider the example to find the optimal (minimum) cost tour for the following diagraph.

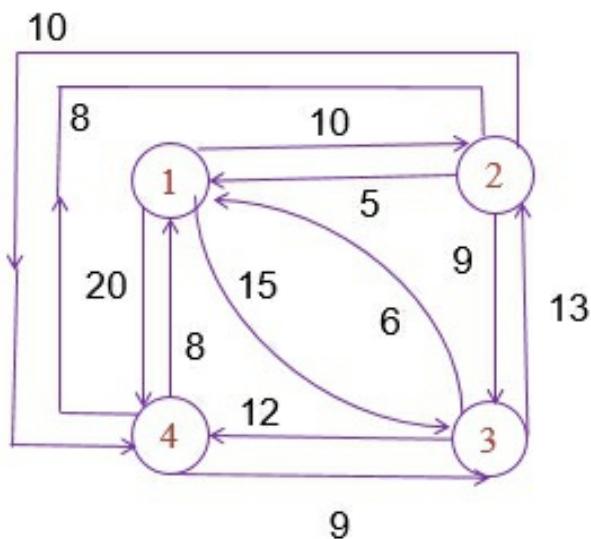


Fig. 5.6.5: Graph G

Its cost adjacent matrix is shown below.

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>1</b>	0	10	15	20
<b>2</b>	5	0	9	10
<b>3</b>	6	13	0	12
<b>4</b>	8	8	9	0

The formula for solving the problem is,

$$g(i, S) = \min_{j \in S} \{C_{ij} + g(j, S - \{j\})\}$$

$$g(i, \emptyset) = C_{i1}, 1 \leq i \leq n$$

$$g(1, \emptyset) = C_{11} = 0$$

$$g(2, \emptyset) = C_{21} = 5$$

$$g(3, \emptyset) = C_{31} = 6$$

$$g(4, \emptyset) = C_{41} = 8$$

$$|S| = 1$$

It means the set contains only one element. Before solving this problem, we make an assumption that the salesperson starts at vertex 1, from that he can move to vertex 2. If he visits vertex 2, from that vertex, he can next visit either vertex 3 or vertex 4.

Since,

$$|S| = 1$$

$$\begin{aligned} g(2, 3) &= \min_{j \in S} \{C_{23} + g(3, \emptyset)\} \\ &= 9 + 6 = 15 \end{aligned}$$

$$\begin{aligned} \min \{C_{24} + g(4, \emptyset)\} \\ g(2, 4) = j \in S &= 10 + 8 = 18 \end{aligned}$$

From vertex 1, next he can visit vertex 3 instead of vertex 2, in this case from vertex 3 next he can visit either vertex 2 or vertex 4.

$$\begin{aligned} \min \{C_{32} + g(2, \emptyset)\} \\ g(3, 2) = j \in S &= 13 + 5 = 18 \end{aligned}$$

$$\begin{aligned} \min \{C_{34} + g(4, \emptyset)\} \\ g(3, 4) = j \in S &= 12 + 8 = 20 \end{aligned}$$

From vertex 1, he can visit vertex 4 instead of vertex 3, in this case from vertex 4, next he can visit either vertex 2 or vertex 3.

$$\begin{aligned} \min \{C_{42} + g(2, \emptyset)\} \\ g(4, 2) = j \in S &= 8 + 5 = 13 \end{aligned}$$

$$\begin{aligned} \min \{C_{43} + g(3, \emptyset)\} \\ g(4, 3) = j \in S &= 9 + 6 = 15 \end{aligned}$$

$|S| = 2$ , here set contains two values, so we can place two vertices in S. From starting vertex 1 next he can visit either vertex 2 or vertex 3 or vertex 4.

If he visits vertex 2, then from that vertex he can visit either vertex 3 or vertex 4. So, determine  $g(2, \{3, 4\})$ , ( $S=\{3, 4\}$  since  $|S|=2$ ),  $g(3, \{2, 4\})$  and  $g(4, \{2, 3\})$ .

$$\begin{aligned} \min \{C_{23} + g(3, 4) + C_{24} + g(4, 3)\} \\ g(2, \{3, 4\}) = j \in S \end{aligned}$$

$$\begin{aligned} \min \{9 + 20, 10 + 15\} \\ = j \in S &= 25 \end{aligned}$$

$$\begin{aligned} \min \{C_{32} + g(2, 4) + C_{34} + g(4, 2)\} \\ g(3, \{2, 4\}) = j \in S \end{aligned}$$

$$\begin{aligned} \min \{13 + 18, 12 + 13\} \\ = j \in S &= 25 \end{aligned}$$

$$\begin{aligned} \min \{C_{42} + g(2, 4) + C_{43} + g(3, 2)\} \\ g(4, \{2, 3\}) = j \in S \end{aligned}$$

$$\begin{aligned} \min \{8 + 15, 9 + 18\} \\ = j \in S &= 23 \end{aligned}$$

$|S| = 3$ , here  $S$  contains 3 elements. After starting from vertex 1, next he can visit vertex 2 or vertex 3 or vertex 4. So, determine  $g(1, \{2, 3, 4\})$ .

$$g(1, \min_{j \in S} \{C_{1j} + g(j, \{3, 4\}), C_{13} + g(3, \{2, 4\}), C_{14} + g(4, \{2, 3\})\}) =$$

$$\min_{j \in S} \{10 + 25, 15 + 25, 20 + 23\}$$

$$\min_{j \in S} \{35, 40, 43\} = 35$$

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$$

$$10 + 10 + 9 + 6 = 35$$

$J(1, \{2, 3, 4\}) = 2$ , thus the tour starts from 1 and goes to 2, then the remaining tour may be obtained from  $g(2, \{3, 4\})$  i.e.,  $J(2, \{3, 4\}) = 4$ , thus, then from 2 to 4 the optimal tour is 1, 2, 4, 3, 1 and the minimum cost is  $10 + 10 + 9 + 6 = 35$ .

### Time Complexity:

$$N = (n-1) \sum_{k=0}^{n-2} (C_{n-k}^n) (\text{ } (C_k^n) \text{ is same as } {}_n C_k)$$

$$= (n-1)({}^{n-2}C_0 + {}^{n-2}C_1 + {}^{n-2}C_2 + \dots + {}^{n-2}C_{n-2})$$

$$= (n-1)(2^{n-2}) \quad (\because {}^n C_0 + {}^n C_1 + {}^n C_2 + \dots + {}^n C_n = 2^n)$$

$$n.N = n(n-1)\left(\frac{2^n}{4}\right) \text{ for } n\text{-stages}$$

$$= n^2 \frac{2^n}{4} - \frac{2^n}{4}$$

$$\Rightarrow O(n^2 \cdot 2^n)$$

The time complexity of travelling salesperson problem is  $O(n^2 \cdot 2^n)$ .

#### 5.3.5.1 Solved Problem

##### Problem:

Find the optimal tour-path and its associated lengths for the following paths. The edges length are given by the matrix.

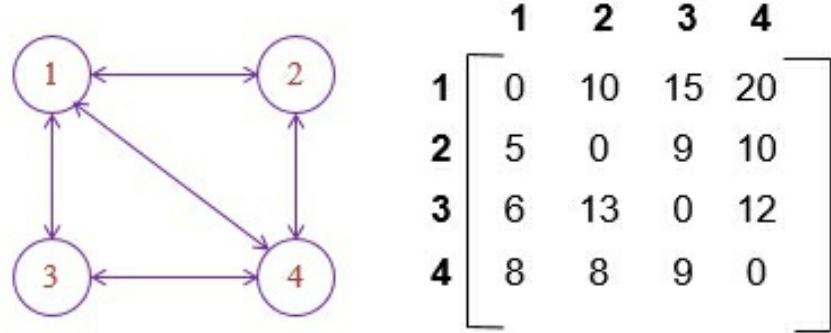


Fig. 5.3.6: Directed Graph and its Edge Length Matrix

Solution:

$$G(1, \{2, 3, 4\}) = ?$$

**Step 1:** For  $|S| = \emptyset$ ,  $g(i, \emptyset) = C(i, 1)$

$$g(1, \emptyset) = C(1, 1) = 5$$

$$g(2, \emptyset) = C(2, 1) = 5$$

$$g(3, \emptyset) = C(3, 1) = 6$$

$$g(4, \emptyset) = C(4, 1) = 8$$

**Step 2:** For  $|S| = 1$ , i.e., the number of elements in S is 1.

$$g(i, S) = \min_{j \in S} \{C(i, j) + g(j, S - \{j\})\}$$

$$g(2, \{3\}) = c(2, 3) + g(3, S - \{3\}) = c(2, 3) + g(3, 5) = 9 + 6 = 15$$

$$g(2, \{4\}) = c(2, 4) + g(4, S - \{4\}) = c(2, 4) + g(4, 5) = 10 + 8 = 18$$

$$g(3, \{2\}) = c(3, 2) + g(2, S - \{2\}) = c(3, 2) + g(2, 5) = 13 + 5 = 18$$

$$g(3, \{4\}) = c(3, 4) + g(4, S - \{4\}) = c(3, 4) + g(4, 5) = 12 + 8 = 20$$

$$g(4, \{2\}) = c(4, 2) + g(2, S - \{2\}) = c(4, 2) + g(2, 5) = 8 + 5 = 13$$

$$g(4, \{3\}) = c(4, 3) + g(3, S - \{3\}) = c(4, 3) + g(3, 5) = 9 + 6 = 15$$

**Step 3:** For  $|S| = 2$ , i.e., the number of elements in S is 2.

$$\begin{aligned} g(2, \{3, 4\}) &= \min\{c(2, 3) + g(3, \{4\}), c(2, 4) + g(4, \{3\})\} \\ &= \min\{9 + 20, 10 + 15\} = \min\{29, 25\} = 25 \end{aligned}$$

$$\begin{aligned} g(3, \{2, 4\}) &= \min\{c(3, 2) + g(2, \{4\}), c(3, 4) + g(4, \{2\})\} \\ &= \min\{13 + 18, 12 + 13\} = \min\{31, 25\} = 25 \end{aligned}$$

$$\begin{aligned} g(4, \{2, 3\}) &= \min\{c(4, 2) + g(2, \{3\}), c(4, 3) + g(3, \{2\})\} \\ &= \min\{8 + 15, 9 + 18\} = \min\{23, 27\} = 23 \end{aligned}$$

**Step 4:** For  $|S| = 3$ , i.e., the number of elements in S is 3.

$$\begin{aligned} g(1, \{2, 3, 4\}) &= \min\{c(1, 2) + g(2, \{3, 4\}), c(1, 3) + g(3, \{2, 4\}), c(1, 4) + \\ &\quad g(4, \{2, 3\})\} \\ &= \min\{10 + 25, 15 + 25, 20 + 23\} \\ &= \min\{35, 40, 43\} \\ &= 35 \end{aligned}$$

$$\begin{aligned} g(1, \{2, 3, 4\}) &= c(1, 2) + g(2, \{3, 4\}) \\ &= c(1, 2) + c(2, 4) + g(4, \{3\}) \\ &= c(1, 2) + c(2, 4) + c(4, 3) + g(3, \emptyset) \\ &= c(1, 2) + c(2, 4) + c(4, 3) + c(3, 1) \end{aligned}$$

$\therefore$  The path is  $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ .

# 6 BACKTRACKING

## 6.1 INTRODUCTION TO BACKTRACKING

In the cases of Greedy Method and Dynamic programming techniques, we will use bruteforce approach. It means, we will evaluate all possible solutions, among which we select one solution of the optimal solution. In back tracking technique we will get same optimal solution with a less number of steps. So by using backtracking technique, we will solve problems in an efficient way. When compared to other methods, Greedy method and Dynamic programming the Backtracking proves to be an efficient approach.

In this Backtracking method, we will use bounding functions (Criterion function), implicit and explicit constraints. While explaining the general method of the backtracking technique, there we will see the implicit and explicit constraints. The major advantages of backtracking method is if a partial solution  $(x_1, x_2, x_3, \dots, x_i)$  cannot lead to an optimal solution then  $(x_{i+1}, \dots, x_n)$  solution may be ignored entirely.

Let us see some technology used in this method.

- 1) **Criterion Function:** It is a function that needs to be maximized or minimized for a given problem.
- 2) **Solution Space:** All the tuples that satisfy the explicit constraints define a possible solution space for a particular instance 'I' of the problem.

**Example:**

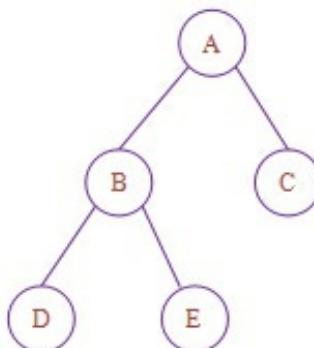


Fig. 6.1.1: Tree organization of a solution space

- 3) **Problem State:** Each node in the tree organization defines a problem state.

**Example:**

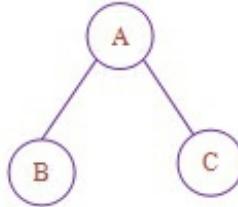


Fig. 6.1.2: Tree (Problem State)

So, A, B, C nodes are problem state.

- 4) **Solution States:** These are those problem states S for which the path from the root to S define a tuple in the solution space.

**Example:**

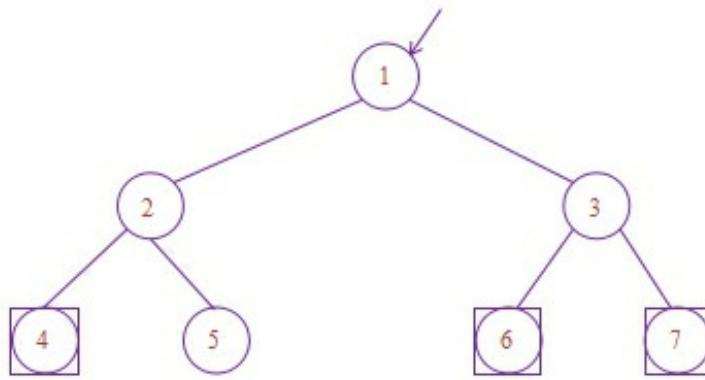


Fig. 6.1.3: Tree (Solution States)

Here, square nodes indicates solution. For the above solution space, there exists 3 solution states. These solution states represented in the form of tuples i.e., (1, 2, 4), (1, 3, 6) and (1, 3, 7) are the solution states.

- 5) **State Space Tree:** If we represent the solution space in the form of tree then the tree is referred as the state space tree.

**Example:**

- i) Statespace tree of the 4-queens problem.

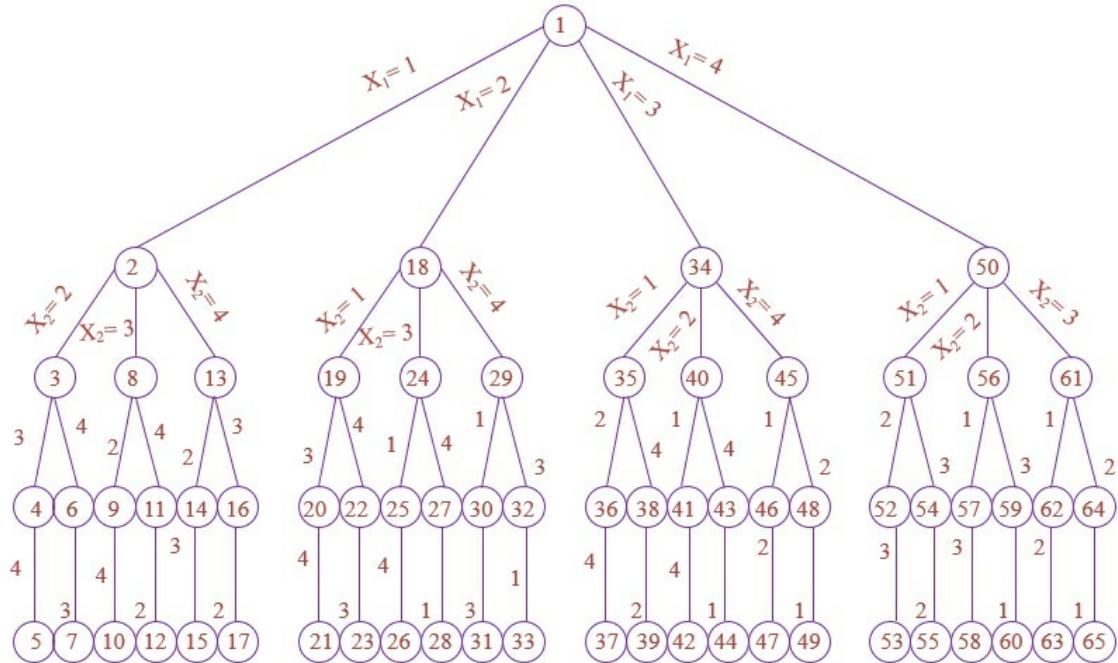


Fig. 6.1.4: Tree organization of the 4 Queens Solution Space

In the figure above the nodes are numbered as in Depth First Search (DFS).

Initially  $x_1 = 1$  or  $2$  or  $3$  or  $4$ . It means we can place first Queen in either first, second, third or fourth column. If  $x_1 = 1$ , then  $x_2$  can be placed in either  $2^{\text{nd}}$ ,  $3^{\text{rd}}$  or  $4^{\text{th}}$  column. If  $x_2 = 2$ , then  $x_3$  can be placed in either in  $3^{\text{rd}}$  or  $4^{\text{th}}$  column. If  $x_3 = 3$ , then  $x_4 = 4$ . So, nodes 1-2-3-4-5 is one solution in the solution space. It may or may not be a feasible solution. Similarly we can observe the remaining solutions in the figure shown above.

- ii) State space tree of travelling sales person problem.

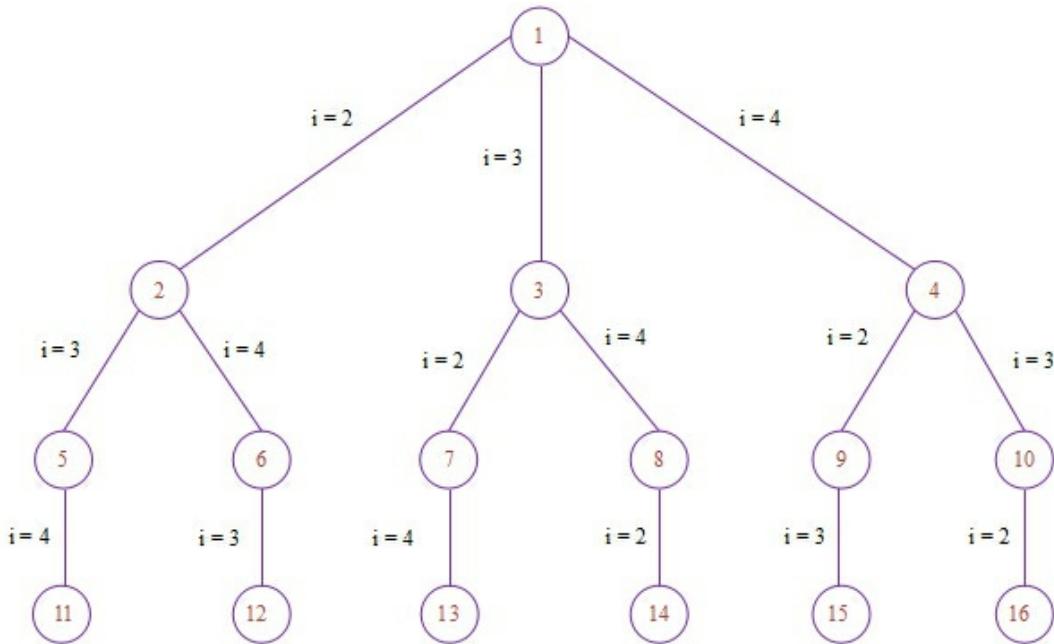


Fig. 6.1.5: State Space Tree for the travelling salesperson problem with  $n = 4$

- 6) **Answer States:** These solution states  $S$  for which the path from the root to  $S$  defines a tuple which is a member of the set of solutions. (i.e., it satisfies the implicit constraints) of the problem.

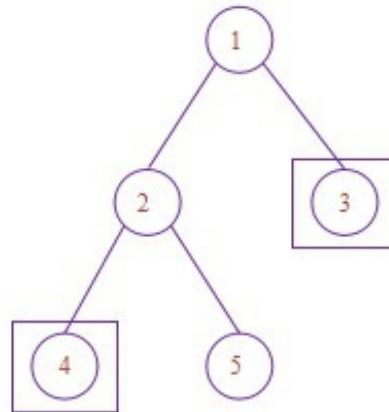


Fig. 6.1.6: Tree (Answer States)

- Here 3, 4 are the answer states. (1, 3) and (1, 2, 4) are solution states.
- 7) **Live Node:** A node which has been generated and all of whose children have not yet been generated is live node.  
**Example:**

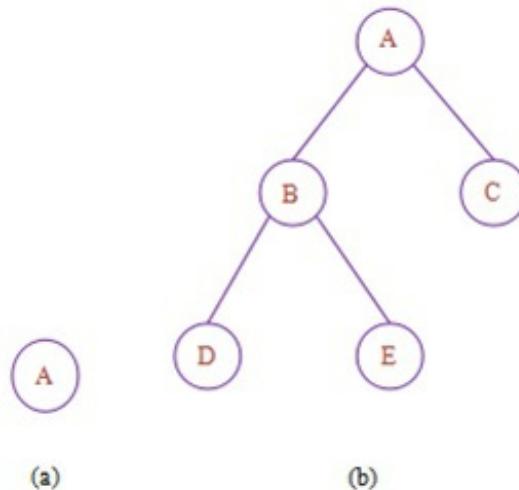


Fig. 6.1.7: Tree (Live Nodes)

In the above figure (a), node A is called as live node since the children of the node A have not yet been generated.

In the figure (b), nodes D, E, C are live nodes because the children of these nodes are not yet been generated.

- 8) E-Node:** The live nodes whose children are currently being generated is called the E-node (the node being expanded).

## Example:



Fig. 6.1.8: Live Node

Node A is live node and its children are currently being generated (expanded). This can be shown in the below figure.

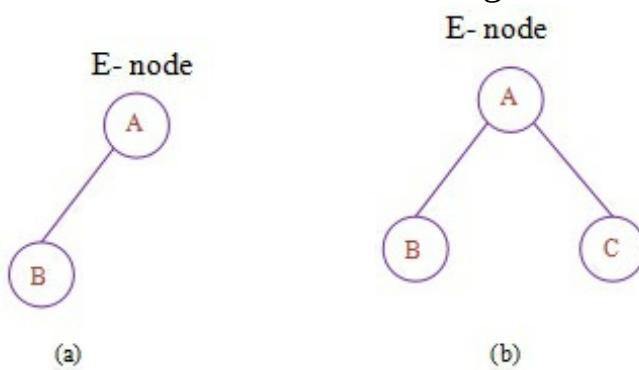


Fig. 6.1.9: E-Nodes

Here, node A is E-node.

- 9) Dead Node:** The dead node is a generated node that fails either not to be expanded further or one for which all of its children have been

generated.

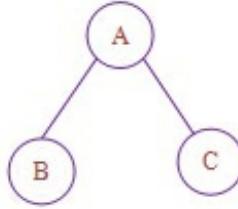


Fig. (a): Dead Nodes

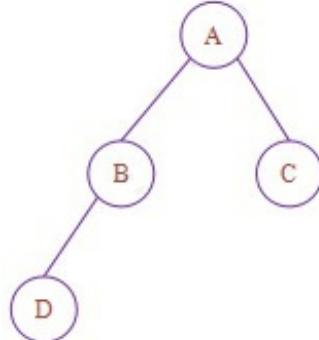


Fig. (b): Tree with One Live Node

In the above figure (a), nodes A, B, C are dead nodes. Since node A's children generated and node B, C are not expanded.

In the above figure (b), node B generates one more node, so, A, C, D are the dead nodes.

**10) Bounding Function:** Bounding function is a function used to kill live nodes without generating all their children.

Various problem uses the concept of backtracking N-Queens problems. Sum of subsets problems, graph coloring and Hamiltanion cycles.

### 6.1.1 Constraints for Backtracking

The problem solved using backtracking require that all the solutions satisfy a complex set of constraints. They are as follows.

**Implicit Constraints:** These are the rules which determine which of the tuples in the solution space satisfy the criterion function. Thus implicit constraints describe the way in which the  $x_i$  must be related to each other.

**Example:** In 4-Queens problem, the implicit constraints are no two queens can be on the same column or on the same row or on the same diagonal.

**Explicit Constraints:** These are the rules for which restrict each  $x_i$  to take the values only from the given set.

**Example 1:** In knapsack problem the explicit constraints are,

- i.  $x_i = 0$  or  $1$ .
- ii.  $0 \leq x_i \leq 1$ .

**Example 2:** In 4-Queens problem, 4 queens can be placed in  $4 \times 4$  chessboard in  $4^4$  ways.

### 6.1.2 Comparison between Backtracking and Brute Force Approach

S. NO.	Backtracking	Brute Force Approach
1	Backtracking is an algorithm design technique for solving the large instances of combinatorial problems.	Brute Force Approach is a straight forward approach for solving a problem, usually based on the problem constraints.
2	In this Backtracking Method the solutions are constructed one component at a time and evaluate the partially constructed solution as if no increasing values of the remaining components can lead to a solution, the remaining components are not generated at all and backtracks to replace the last component of the partially constructed solutions with its next option.	In executive search, a Brute Force approach, all the solutions are generated and then identifies the one with a desired property.
3	Backtracking method is efficient than bruteforce approach.	Brute Force approach is less efficient.
4	When backtracking method is applied to knapsack problem, using a dynamic state space tree formulation leads to an efficient	Executive search, a Brute Force approach when applied to a Knapsack Problem leads to an algorithm i.e., inefficient on every input.

	algorithm for every input.	
5	Backtracking method is applied to combinatorial problems to solve the large instances of the problem, here also we face the difficulty of exponential explosion.	An executive search approach can also be applied to combinatorial problems which suggests generating each and every element of the problems domain, we face the difficulty of exponential explosion.
6	Examples of the backtracking method includes n-queens problems, sum of subsets problem.	Examples of the Brute Force Approach includes selection sort, sequential search.
7	Backtracking method is not as simple as Bruteforce approach.	Brute Force Approach is very simple.

Table 6.1.1: Comparision between Backtracking and Brute Force Approach

## 6.2 GENERAL METHOD

The control abstraction is also called as general scheme for backtracking is as follows. Let  $(X_1, X_2, \dots, X_K)$  be a path from the root to node in a state space tree. Let  $T(X_1, X_2, \dots, X_{K+1})$  be the set of all possible values for  $X_{K+1}$  such that  $(X_1, X_2, \dots, X_{K+1})$  is also path to a problem state. We shall assume the existence of bounding functions  $B_{i+1}$  is (expressed as predicates) such that  $B_{i+1}(X_1, X_2, \dots, X_{K+1})$  is false for a path  $(X_1, X_2, \dots, X_{K+1})$  from the root node to a problem state only if the path cannot be extended to reach an answer node. Thus the candidates for the position  $i+1$  of the solution vector  $X(1:n)$  are those values that are generated by  $T$  and satisfy  $B_{K+1}$ .

The algorithm for the recursive backtracking is as follows.

```

Algorithm RBacktracking(k)

//On entering the first k-1 values x[1], x[2], ..., x[k-1] of the solution vector
//x[1:n] have been assigned.

//x[] and n are global.

{
    for (each x[k] ∈ T(x[1], ..., x[k-1]) do
    {
        if(Bk(x[1], x[2], ..., x[k]) ≠ 0) then
        {
            if(x[1], x[2], ..., x[k] is a path to the answer node)
                then write(x[1:k]);
            if(k < n) then Backtrack(k+1);
        }
    }
}

```

#### Algorithm 6.2.1: Recursive Backtracking Algorithm

The above algorithm represents a recursive formulation of backtracking algorithm. Backtracking is essentially a post order traversal of a tree. This recursive algorithm is initially invoked by call RBacktrack(1).

The solution vector ( $x_1, \dots, x_n$ ) is a global array  $x[1:n]$ . All the possible elements for the  $k^{\text{th}}$  position of the tuple which satisfy  $B_k$  are generated, one by one and adjoin to the current vector  $(x(1), \dots, x(k-1))$ . Each time  $x(k)$  is attached a check is made to determine if a solution has been found. Then the logarithm is recursively invoked. When the for loop is existed, no more values for  $x(k)$  exist and the current copy of the RBacktrack ends. The last unresolved call now resumes, namely the one which continues to examine the

remaining elements assuming only  $k-1$  values have been set.

The iterative approach of the backtracking is shown below.

```
Algorithm IBacktrack(n)
//All the solution are generated in x[1:n] and
//printed as soon as they are determined
{
    k := 1;
    while(k≠0) do
    {
        if(there remains an untired x[k] ∈ T(x[1], x[2], ..., x[k-1]) and Bk(x[1], ..., x[k]) is true) then
        {
            if(x[1], ..., x[k] is a path to an answer node) then
                write(x[1:k]);
            k := k+1; //Consider the next set.
        }
        else k := k-1; //Backtrack to the previous set.
    }
}
```

Algorithm 6.2.2: General Iterative Backtracking Method

### 6.2.1 Estimating the Efficiency of Backtracking

While estimating the efficiency of backtracking,  $T( )$  will yield the set of all possible values that can be placed as the first component  $x_1$  of the solution vector. The component  $x_1$  will take values for which the bounding function  $B_1(x_1)$  is true. The elements are generated in the depth first manner. The variable  $k$  is continually incremented and a solution vector is grown until

either a solution is found or no untired value of  $x_k$  remains. When  $k$  is decremented, the algorithm must resume the generation of possible elements for the  $k^{\text{th}}$  position that have not yet been tried. Therefore one must develop a procedure that generates these values in some order. If only the solution is desired, replacing `write(x[1:k])`; with { `write(x[1:k])`; `return;` } suffices.

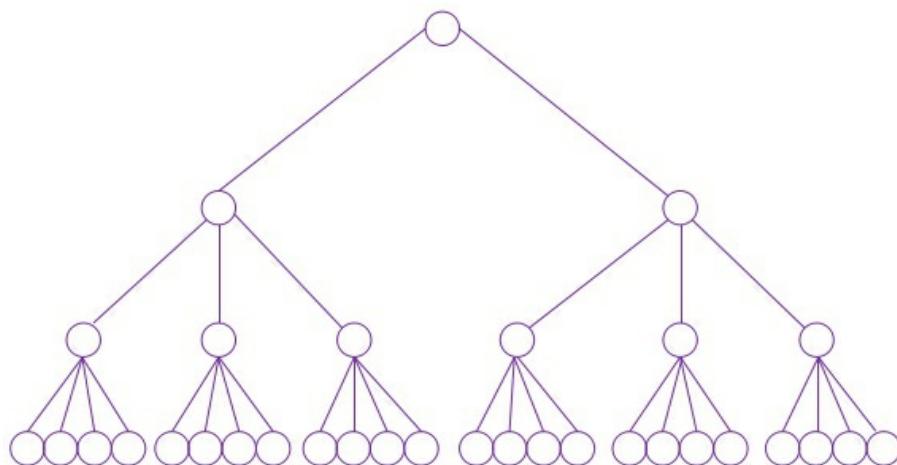
The efficiency of the backtracking alorihms we have just seen depends very much on four factors. They are as follows.

- 1) The time to generate the next possible states  $x[k]$  in the search tree.
- 2) The number of next possible states  $x[k]$  satisfying the explicit constraints.
- 3) The time to compute the bounding functions  $B_x$  and
- 4) The proportion of next possible states  $x[k]$  satisfying the bounding  $B_k$ .

The bounding functions are considered as good if they reduce the nodes that are generated.

But the bounding functions take more time to evaluate for many problems such as n-queens, no good bounding function is known. For these problems rearrangement is one of the best solution.

Rearrangement is the principle of selecting the set  $S_i$ , with fewest elements each time. Since these sets can be taken in any order, smaller branching at the higher levels create langer subtrees. The below figure shows a smaller branch tree.



Removal of rarely nodes cut off larger subtrees. If we remove one node at second level of above figure, then effectively removing 12 nodes from consideration.

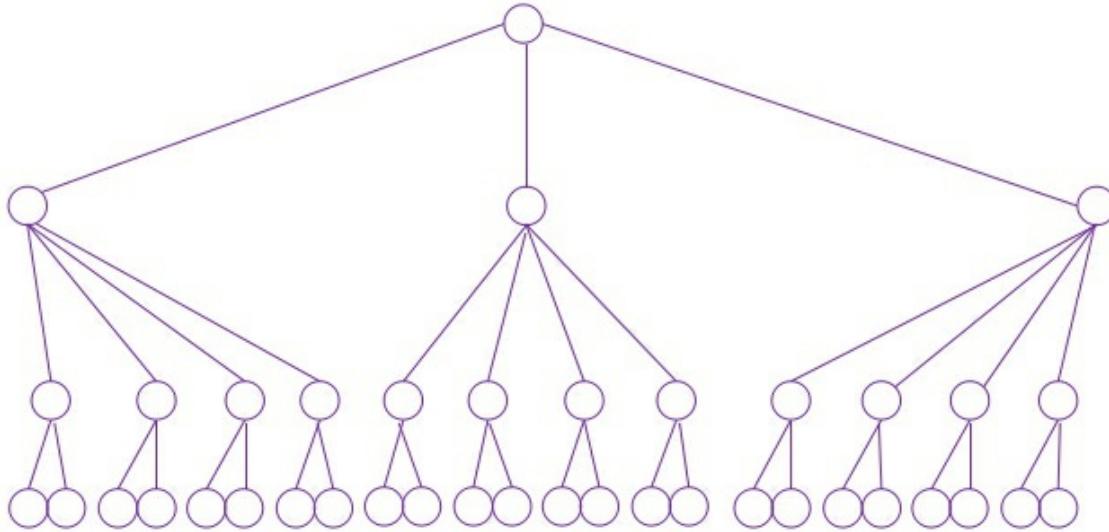


Fig. 6.2.2: A Tree

Where as one node removal at second level of the above figure removes only 8 nodes.

The first three factor, that effect the time required for backtracking depend primarily on the state space tree organization selected. Only the forth factor may vary widely depending on the problem instances.

Worst case predication for backtracking algorithms are,

- 1) If the number of points in the solution space is  $2^n$  or  $n!$  the worst case timing timing is usually either  $O(p(n)2^n)$  or  $O(p(n)n!)$ , where  $p(n)$  is a polynomial.
- 2) Backtracking can often solve some problem instances with large  $n$  in very small amounts of time. However, it may be difficult behavior of algorithm for particular problem instances.

The number of nodes generated in a particular instance can be estimated using Monte Carlo methods. If set  $X$  be a node on this path at level  $I$  of the state space tree the boundary function  $B_i$  is used to determine the number  $m_i$  of its children which will be generated, one child is randomly selected, and the process continue

until the path ends. Then  $m_1 + m_1m_2 + m_1m_2m_3 + \dots$  is an estimate of the nodes that will be generated.

The algorithm for estimating the efficiency of backtracking is as follows.

```
Algorithm EstimateBacktracking()
//This algorithm follows random path in a state space tree
//and produces an estimate of number of nodes in the tree.

{
    k := 1; m := 1; r := 1;
    repeat
    {
        Tk := {x[k] | x[k] ∈ T(x[1], x[2], ..., x[k-1]) and Bk(x[1], ..., x[k]) is true};
        if(size(Tk) = 0) then return m;
        r := r * Size(Tk);
        m := m + r;
        x[k] := Choose(Tk);
        k := k + 1;
    } until(false);
}
```

### Algorithm 6.2.3: Estimating the effency of back tracking

Where, the algorithm follows a random path in a state space tree and produces an estimate of number of nodes in the tree.

The above algorithm determines the value m. Here, it selects the random path from the root of the state space tree. The function size returns the size of the set  $T_k$ . The function choose makes a random choice of an element in  $T_k$ . The desired sum inbuilt using the variable m and r.

## 6.3 APPLICATIONS OF BACKTRACKING

There are various problems that can be solved using backtracking approach.

### 6.3.1 N Queens Problem

Consider an  $n \times n$  chessboard. Let there are  $n$  number of queens. These  $n$ -queens are to be placed in the  $n \times n$  chessboard so that no two queens are on the same column, same row or on the same diagonal.

The algorithm for the  $n$ -queens problems is as follows.

```
Algorithm NQueens(k, n)
//Using backtracking, this procedure prints all
//possible placements of n queens on an n x n chessboard
//so that they are nonattacking
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if(k = n) then write(x[1:n]);
            else NQueens(k+1, n);
        }
    }
}
```

Algorithm 6.3.1: All solutions to the  $n$ -queens problem

This algorithm prints all possible placements of  $n$ -queens on an  $n \times n$  chessboard so that they are non-attacking.

The algorithm for placing n-queens is shown by Place(k, i) function is as follows.

```
Algorithm Place(k, i)
//Return true if a queen can be placed in kth row and ith column.

//Otherwise it returns false. X[ ] is a global array

//whose first (k-1) values have been set.

//ABS(r) return the absolute value of r.

{
    for j := 1 to k-1 do
        if((x[j] = i) or (ABS(x[j]-i) = ABS(j-k)))
            //two in the same column or in the same diagonal
            then return false;
    return true;
}
```

Algorithm 6.3.2: Algorithm to find a new queen can be placed or not

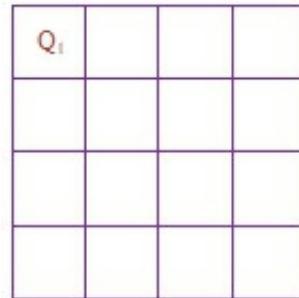
The Place(k, i) function returns true if a queen can be placed in  $k^{\text{th}}$  row and  $i^{\text{th}}$  column, otherwise it returns false. ABS() function returns the absolute value.

### 6.3.1.1 4-Queens Problem

Consider a  $4 \times 4$  chessboard. Let there are 4 queens. The objective here is to place the 4 queens in the  $4 \times 4$  chess board. So that no two queens should be placed in the same row, same column or the same diagonal position. The explicit constraints are 4 queens are to be placed on  $4 \times 4$  chessboard in  $4^4$  ways. The implicit constraints are no two queens are in the same row, or in the same column or in the same diagonal.

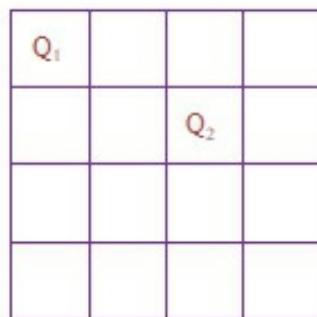
Let  $\{x_1, x_2, x_3, x_4\}$  be the solution vector where  $x_i$ , column number on which  $i$  is placed.

First queen  $Q_1$  is placed in first row and first column.

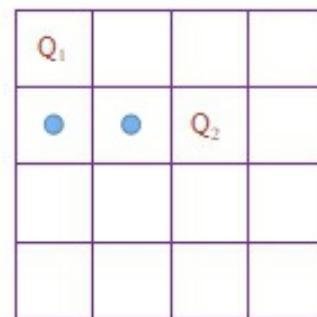


(a)

The second queen should not be placed in the first row and second column. It should not be placed in the second row and in the second, third or in the fourth column. If we place in the second column, both will be in the same diagonal, so place it in third column.

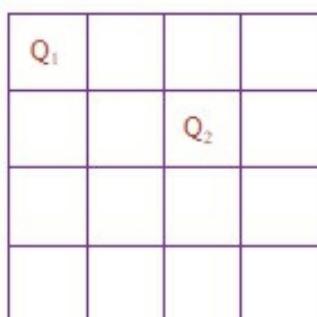


(b)

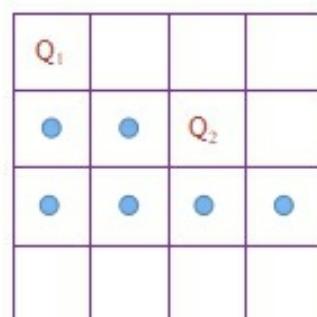


(c)

We are unable to place  $Q_3$  in third row so go back to  $Q_2$  and place it some where else and then place  $Q_3$  in 3<sup>rd</sup> row, 2<sup>nd</sup> column.



(b)



(c)

Now, the fourth queen should be placed in the 4<sup>th</sup> row and 3<sup>rd</sup> column but there will be a diagonal track from Q<sub>3</sub>. So go back, remove Q<sub>3</sub> and place it in the next column. But it is not possible, so move back to Q<sub>2</sub> and remove it to next column but it is not possible. So go back to Q<sub>1</sub> and move it to next column.

It can be shown as follows.

Q <sub>1</sub>			
			Q <sub>2</sub>

(f)

Q <sub>1</sub>			
			Q <sub>2</sub>

(g)

Q <sub>1</sub>			
			Q <sub>2</sub>
Q <sub>3</sub>			

(h)

Q <sub>1</sub>			
			Q <sub>2</sub>
Q <sub>3</sub>			
		Q <sub>4</sub>	

(i)

Fig. 6.3.1: Example of Backtrack solution to the 4-Queens problem

Hence, the solution to 4-queens problem is obtained as x<sub>1</sub>=2, x<sub>2</sub>=4, x<sub>3</sub>=1, x<sub>4</sub>=3. i.e., first queen is placed in 2<sup>nd</sup> column, second queen should be placed in 4<sup>th</sup> column, third queen is placed in 1<sup>st</sup> column and fourth

queen is placed in 3<sup>rd</sup> column.

The state space tree for the 4-queens problem is shown below.

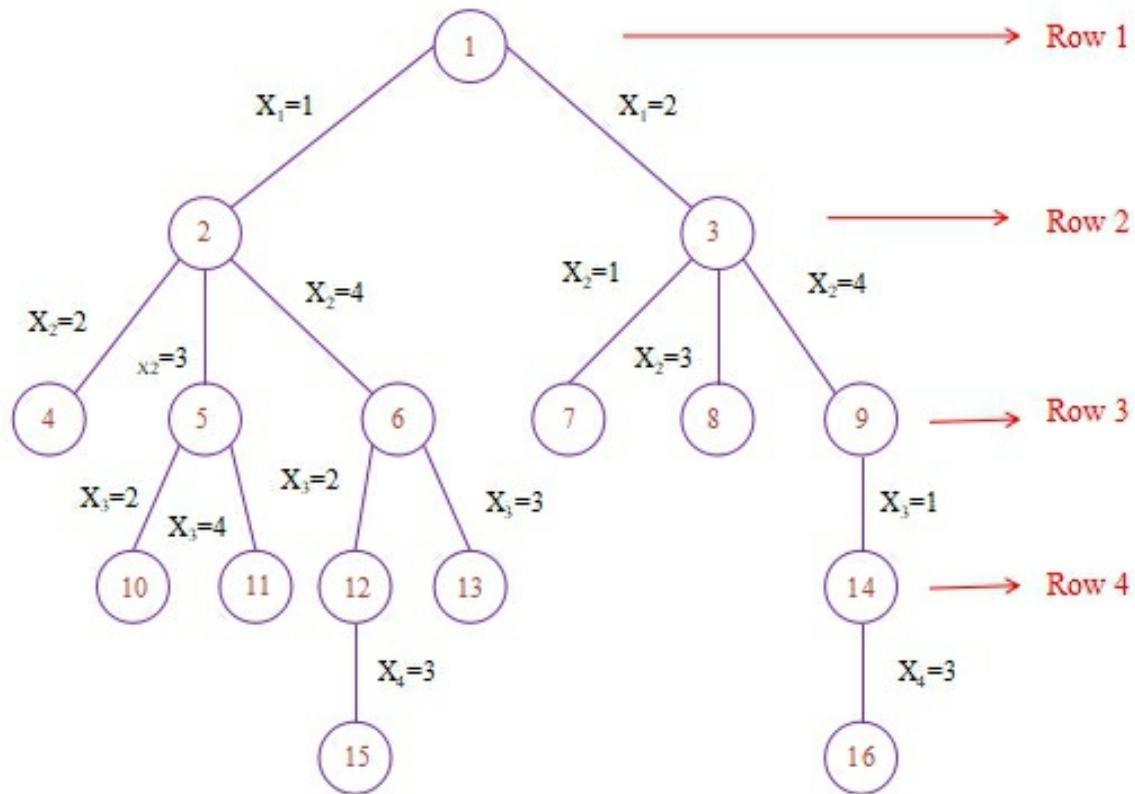


Fig. 6.3.2:Portion of the tree that is generated during backtracking

### 6.3.1.2 8-Queens Problem

Consider a 8 x 8 chessboard. Let there are 8 queens. The objective is to place these 8 queens on the board so that no two queens are in the same row, same column or on the same diagonal.

Let  $\{x_1, x_2, x_3, \dots, x_8\}$  be the solution vector where  $x_i$ , column number on which i is placed. Let  $A[1:8, 1:8]$  be the two dimensional array representing the squares of chessboard. The queens are numbered 1 through 8. Each queen must be on a different row.

The explicit constraints are 8 queens are to be placed on 8 x 8 chessboard in  $8^8$  ways. We reduce the solution space of explicit constraints by applying the implicit constraints. The implicit constraints are no two queens are in the same row, or in the same column or in the same diagonal.

Suppose,  $(i, j)$  and  $(k, l)$  are two positions for two queens. They are on the same diagonal if  $|j-l| = |i-k|$ .

Typical solution to 8-queens problem is shown below.

				$Q_1$			
					$Q_2$		
						$Q_3$	
		$Q_4$					
						$Q_5$	
	$Q_6$						
			$Q_7$				
				$Q_8$			

Fig. 6.3.3: One solution to the 8-queens problem

The solution is  $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (4, 6, 8, 2, 7, 1, 3, 5)$ .

**Time Complexity:** The solution space tree of 8-queens problem contain  $8^8$  tuple. After imposing implicit constraints, the size of solution space is reduced to  $8!$  tuples. Hence the time complexity is  $O(8!)$ . For  $n$ -queens problem it is  $O(n!)$ .

### 6.3.2 Sum of Subsets Problem

#### 6.3.2.1 Solved Problem

#### 6.3.3 Graph Coloring Problem

Let  $G$  be the graph consisting of set of vertices and set of edges and let  $m$  be the positive integer. Graph coloring is the set of coloring each vertex in the graph in such a way that no two adjacent vertices have same color and  $m$  colors are used. This problem is called as  $m$ -coloring problem. If the degree of the graph is  $d$  then we can color it with  $d + 1$  colors. The minimum number of colors required to color the graph is called its chromatic number. The maximum chromatic number of any planar graph is 4.

**Example:** Consider the graph,

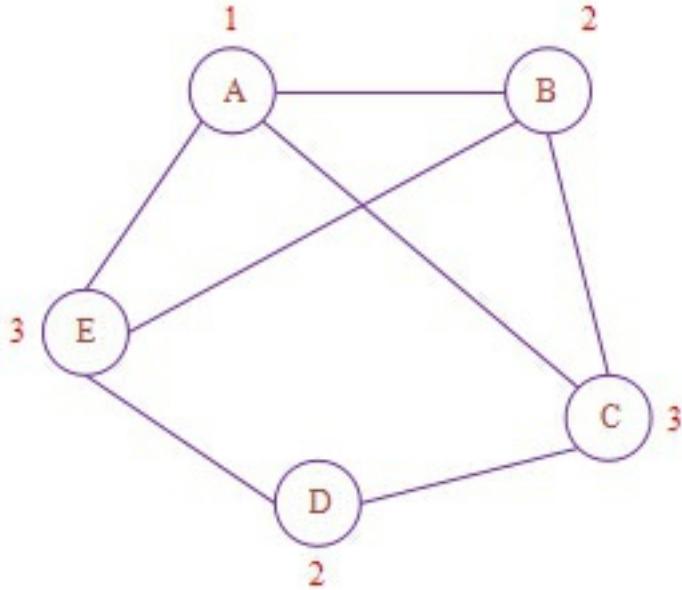


Fig. : Example Graph and its Coloring (Color of each Node is Indicated Next to each Node)

We require three colors to paint this graph. Hence, the chromatic number of given graph is 3.

We can use backtracking problem to solve the graph coloring problem as follows,

Let graph G consisting of n vertices with adjacency matrix A.

i.e.,  $A = [a_{ij}]_{n \times n}$ , where  $a_{ij} = 1$ , if  $(i, j) \in E(G) = 0$ , otherwise

Let the colors represented by the integers (1, 2, 3, ..., m) and let  $(x_1, x_2, \dots, x_n)$  be the solution, where  $x_i$  is the color of vertex i.

The algorithm for finding all m-coloring of a graph using the recursive backtracking formulation is given as follows.

```

Algorithm MColoring(k)

//This algorithm follows recursive backtracking schema.

//The graph is represented by its Boolean adjacency matrix G[1:n, 1:n]. 

//All assignments of 1, 2, 3, ..., m to the vertices of the graph such that

//adjacent vertices are assigned distinct integers are printed.

//k is the index of the next vertex to color.

{

    repeat

        { //Generate all legal assignments of x[k]. 

            NextValue(k); //Assign x[k] a legal color.

            if(x[k] = 0) then return; //no new color possible

            if(k = n) then //atmost m colors have been used to color the n vertices.

                write (x[1:n]);

                else mColoring(k+1);

        }until(false);

}

```

### Algorithm 6.3.3: Finding all m-colorings of a graph

NextValue() function produces the possible colors for  $x_k$  after  $x_1$ , through  $x_{k-1}$  have been defined the main loop of MColoring() repeatedly picks the element from the set of possibilities, assigns it to  $x_k$ , and the n calls MColoring() recursively.

Algorithm for the function NextValue() is given as follows.

```

Algorithm NextValue(k)

//x[1], ..., x[k-1] have been assigned integer values in the range [1, m]
//such that adjacent vertices have distinct integers.

//The value of x[k] is determined in the range [0, m]. x[k] is assigned the next
//highest numbered color while maintaining distinctness from the
//adjacent vertices of vertex k. If no such color exists, then x[k] is 0.

{
    repeat
    {
        x[k] = (x[k] + 1) mod (m + 1); //next highest color.

        if(x[k] = 0) then return; //All colors have been used.

        for j := 1 to n do
        {
            //Check this color is distinct from adjacent colors.

            if((G[k, j] ≠ 0) and (x[k] = x[j]))
                //If (k, j) is an edge and if adj vertices have the same color.

                then break;

        }

        if(j = n + 1) then return; //new color found

    }until(false); //Otherwise try to find another color.
}

```

#### Algorithm 6.3.4: Generating a next color

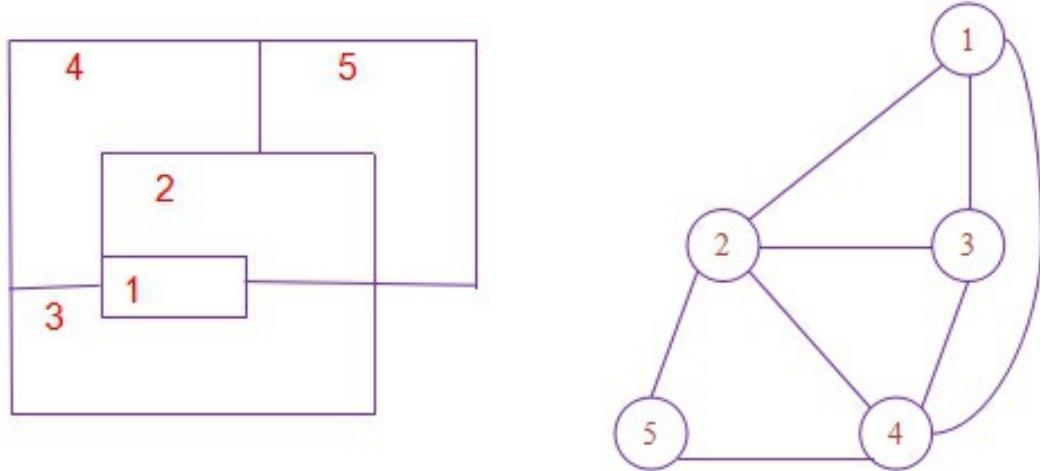
**Time Complexity:** At each internal mode  $O(mn)$  time is spent by next value to determine the children corresponding to legal colorings. Hence, the total time is bounded by

$$\sum_{i=1}^n m^i n = n(m + m^2 + \dots + m^n) = n \cdot m^n$$

$$= O(n \cdot m^n)$$

**Example:**

The planar graph representation using graph coloring is shown below.



Consider state space tree for graph coloring when  $n=3$ ,  $m=3$ .

It means there are 3 vertices in graph and we have to use atmost 3 colors, to color the graph.

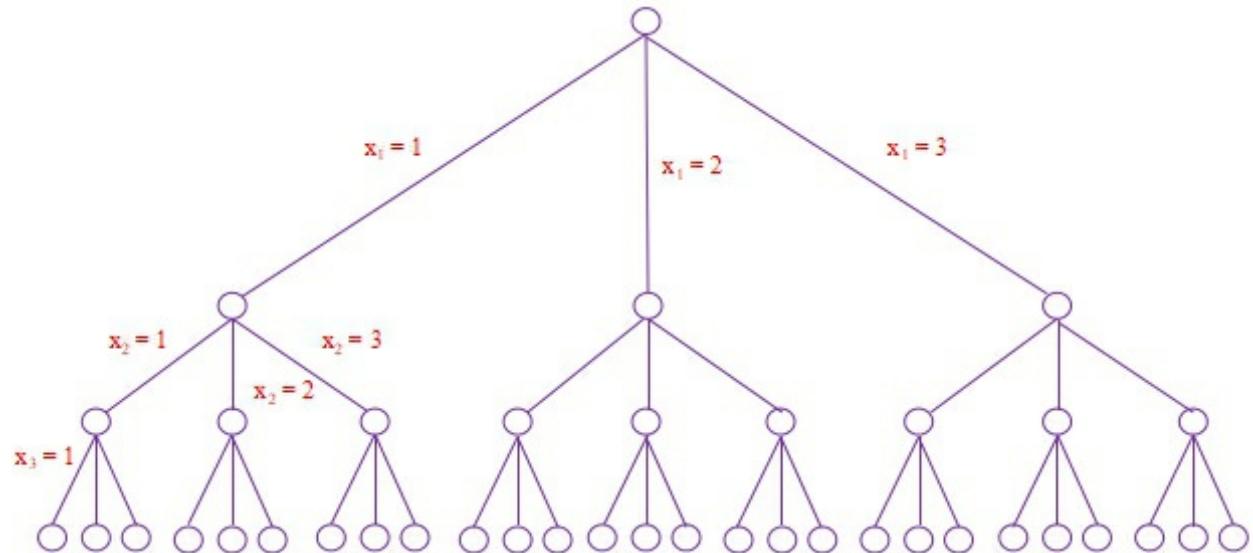


Fig. 6.3.4: State Space Tree for Graph Coloring when  $m=3$  and  $n=3$

State space tree contains all possible moves (some moves leads to

solution or may not lead to solution).

The following tree contains only possible solutions.

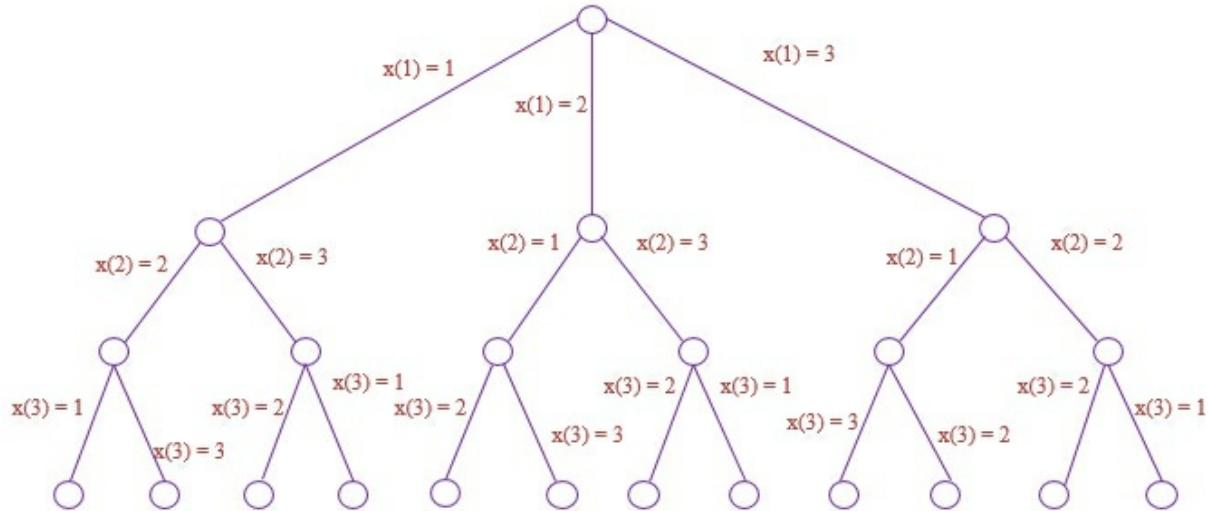


Fig. 6.3.5: A State Space Tree contains Only Possible Solutions

The possible solutions are 12, within that 6 solutions are exist with exactly with 2 colors. In the above tree, when  $x[1] = 1$ , then  $x[2]$  contain either 2 or 3 but not 1, because two adjacent nodes should not have the same color, when  $x[2] = 2$ , then  $x[3]$  contains either 1 or 3. Similarly, we can process remaining nodes.

### 6.3.3.1 Solved Problem

#### Problem:

Generate all possible 3 coloring for the following graph with 4-nodes using a state space tree.

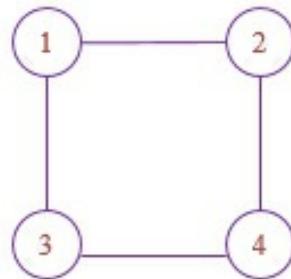


Fig. 6.3.6: Graph

### Solution:

The graph contains 4-nodes. That means  $n=4$  and  $m=3$ . Then the adjacency matrix is,

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

According to the algorithm, initially we start MColoring(1), when  $k=1$  and the array  $x[ ]$  is  $x[1] = x[2] = x[3] = x[4] = 0$ .

The state space tree generated by the algorithm is

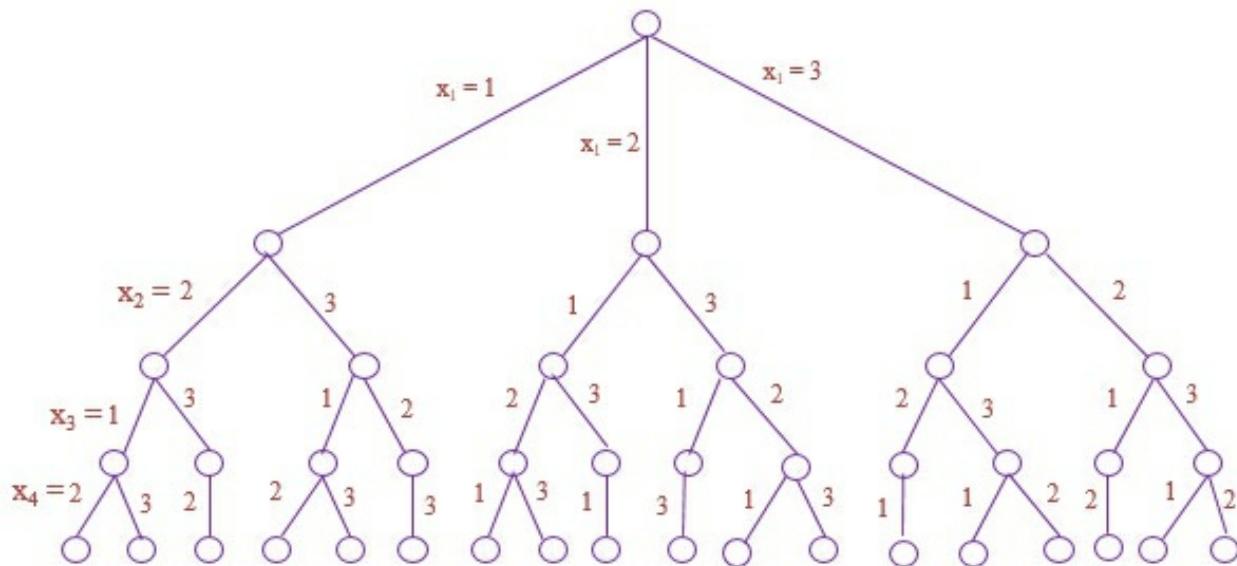


Fig. 6.3.7: A 4 node Graph and all possible 3 coloring

Each path to a leaf represents a coloring using atmost 3 colors with this only 12 solutions are existed.

### 6.3.4 Hamiltonian Cycles

We know that a path  $x_0, x_1, x_2, \dots, x_{n-1}, x_n$  in the graph  $G = (V, E)$  is called a Hamiltonian path, if  $V = \{x_0, x_1, x_2, \dots, x_{n-1}, x_n\}$  and  $x_i \neq x_j$  for  $0 \leq i \leq j \leq n$ . A circuit  $x_0, x_1, x_2, \dots, x_{n-1}, x_n, x_0$  (with  $n < 1$ ). In a graph  $G = (V, E)$  is called a

Hamiltonian circuit. If  $x_0, x_1, x_2, \dots, x_{n-1}, x_n$  is a Hamiltonian path.

Given a graph  $G = (V, E)$  we have to find the Hamiltonian circuit using backtracking approach, we start our search from any arbitrary vertex, say A. This vertex 'A' becomes the root of our implicit tree. Adjacent vertex is selected on the basis of alphabetical/numerical order. If yet any stage an arbitrary vertex, say 'x' makes a cycle with any vertex other than vertex '0' then we say that deadend is reached. In this case we backtrack one step and again the search must begin by selecting another vertex. It should be noted that, after backtracking the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

**Example:**

Consider a graph  $G = (V, E)$  shown in the below figure, we have to find a Hamiltonian circuit using backtracking method.

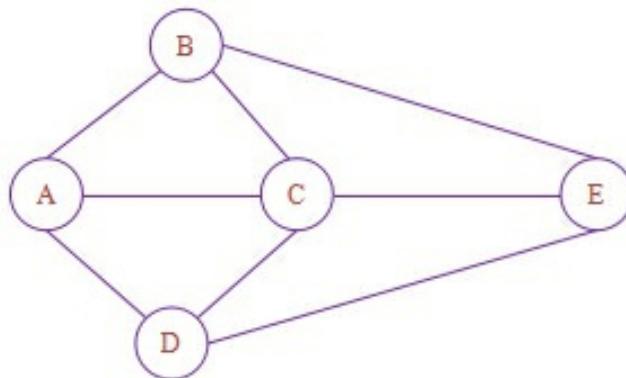
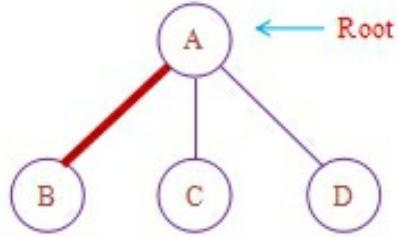


Fig. 6.3.8: Graph G

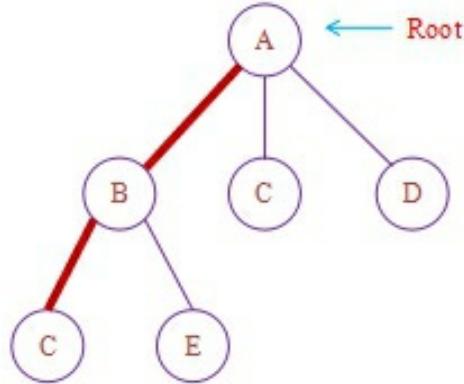
Initially we start our search with vertex 'A', the vertex 'A' becomes the root of our implicit tree.



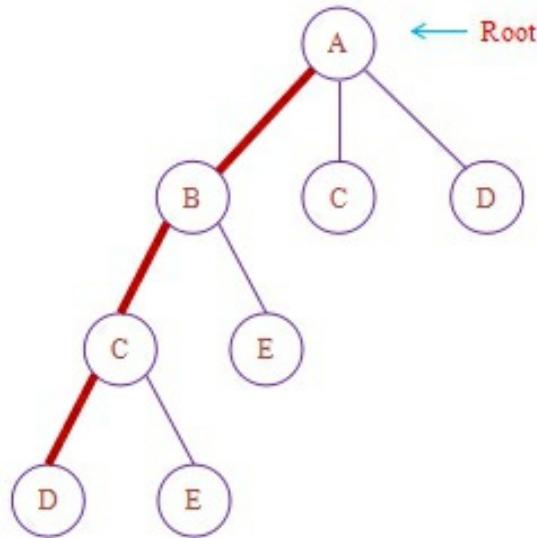
Next we choose vertex 'B' adjacent to 'A', as it comes first in lexicographical order (B, C, D).



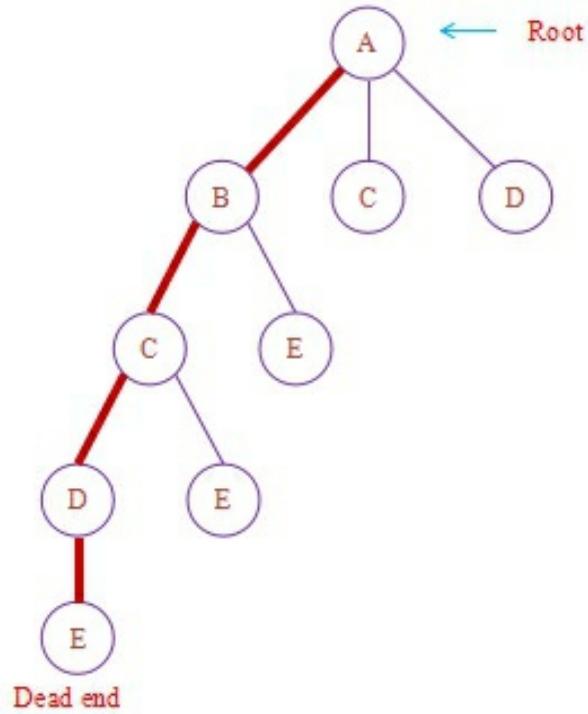
Next vertex 'C' is selected which is adjacent to 'B' and which comes first in lexicographical order (C, E).



Next we select vertex 'D' adjacent to 'C' which comes first in lexicographical order (D, E).

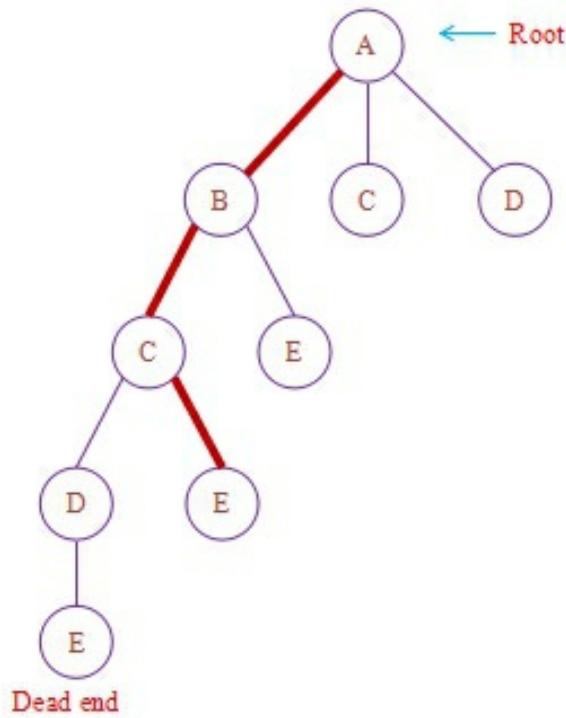


Next vertex 'E' is selected. If we choose vertex 'A' then we do not get the Hamiltonian cycle.

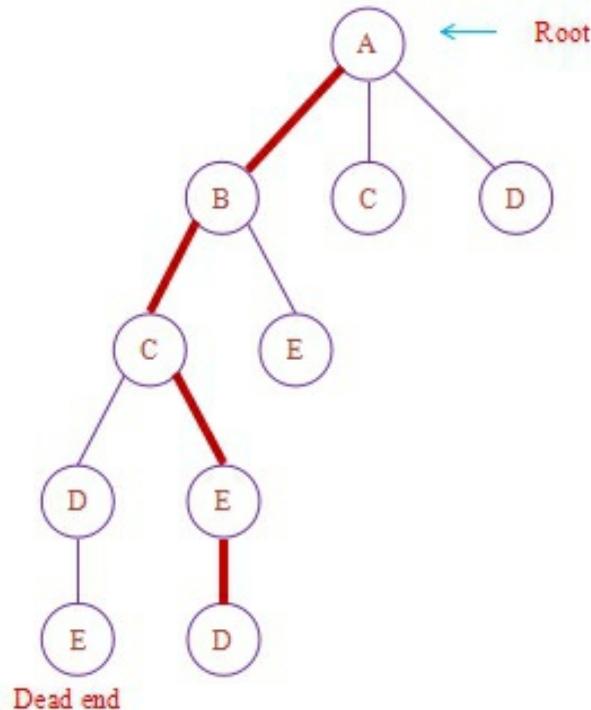


The vertex adjacent to 'E' are B, C, D. But they are already visited. Thus we get the deadend. So we backtrack one step and remove the vertex 'E' from our partial solution. The vertex adjacent to 'D' are E, C, A from which vertex 'E' has already been checked and we are left with vertex 'A' but by choosing vertex 'A' we do not get the Hamiltonian cycle. So, we again backtrack one step.

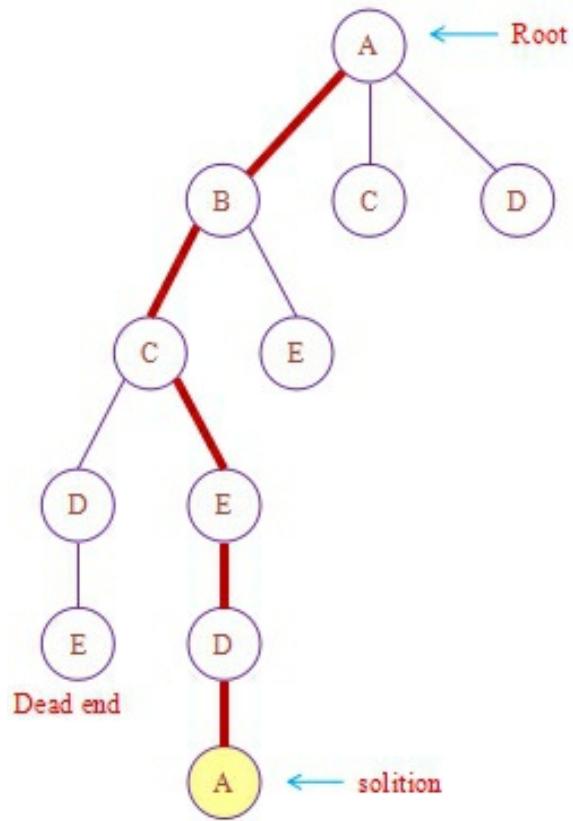
Hence we select the vertex 'E' adjacent to 'C'.



The vertex adjacent to 'E' are (B, C, D). So, vertex D is deleted.



The vertex adjacent to 'D' are (A, C, E). So, vertex 'A' is selected. Hence, we get the Hamiltonian cycle as all the vertex other than the start vertex 'A' is visited only once, A-B-C-E-D-A.



The final implicit tree for the Hamiltonian circuit is shown in the below figure. The number above each node indicates the order in which these nodes are visited.

Fig. 6.3.9: Construction of Hamiltanian cycle using backtracking

The recursive algorithm to find all Hamiltonian cycles of a graph is given as follows.

```

Algorithm HamiltonianCycle(k)

//The graph G[1:n, 1:n] is the adjacency matrix used for the graph G

//the cycle begins from the vertex 1.

{
    repeat
    {
        NextVertex(k);

        if(x[k] == 0) then
            return;
        if (k = n) then
            write(x[1:n])
        else
            HamiltonianCycle(k+1);
    }until(false);
}

```

#### Algorithm 6.3.5: Find all Hamiltonian Cycles

The NextVertex() function determines the next possible vertex for the proposed cycle. The algorithm for the NextVertex() function is as follows.

```

Algorithm NextVertex(k)
//Each time x[k] is assigned to a next highest number vertex
//which is not visited earlier. If x[k] = 0 then no vertex is assigned to x[k]
{
    repeat
    {
        x[k] := (x[k] + 1)mod(n+1); //obtain next vertex
        if(x[k] = 0) then
            return;
        if(G[x[k-1], x[k] ≠ 0) then //if there is any edge between k-1 and k then
        {
            for j := 1 to k-1 do //for every adjacent vertex
            if(x[j] = x[k]) then
                break; //not a distinct vertex
            if(j = k) then
            {
                if((k < n) or (k = n) and G(x[n], x[1] ≠ 0))
                    then return; //return a distinct vertex
            }
        }
    }
}

```

Algorithm 6.3.6: Generating a Next Vertex

## 7 BRANCH AND BOUND

### 7.1 INTRODUCTION TO BRANCH AND BOUND METHOD

The backtracking algorithm is effective for decision problems but it has been not designed for optimization problems. This drawback is rectified in case of branch and bound technique. In this branch and bound technique we will also use bounding function i.e., similar to backtracking. The essential difference between back tracking and branch and bound is, in case if we get a solution then we will terminate the search procedure in backtracking where as the branch and bound we will continue the process (search) until we get an optimal solution. Branch and bound technique is applicable for only minimization problems.

The general idea of branch and bound is a BFS, like search for the optimal solution but not all the nodes gets expanded (i.e., their children generated). A carefully selected criterion is rather determines which node to expand and when and another criterion tells when algorithm when an optimal solution has been found.

Branch and bound technique needs two additional values when compared to backtracking. They are,

- 1) A bound value of the objective function (should be lower bound for minimization problems and upper bound for maximization problems) for every node of the state space tree.
- 2) The value of the best solution so far is compared with a node's bound and if the nodes bound value is not better than the value of the best solution said so far then that node is terminated.

This is the key concept of branch and bound technique.

Some of problems, which uses the concept of branch and bound are,

- 1) Travelling salesperson problem
- 2) 0/1 Knapsack problem

#### 7.1.1 Comparison between Backtracking and Branch and Bound

S. NO.	Backtracking Method	Branch and Bound Method
1	In this backtracking	In Branch and bound technique,

	technique, the solution is obtained by depth first search method.	any of the search methods among depth first search, breadth first search or the best first search can be used to obtain the solution.
2	Backtracking approach provides the solutions to decision problems.	Branch and bound technique is used to solve the optimization problems.
3	There is a possibility of obtaining a bad solution in backtracking method.	No bad solutions are obtained here in the branch and bound method.
4	A state space tree is not searched completely instead, the process of searching terminates as soon as the solution is obtained in the backtracking method.	State space tree generated using branch and bound method is searched completely, since there is a possibility of obtaining an optimum solution at any point in the state space tree.
5	The backtracking technique is used on the problems like graph coloring, sum of subsets, n-queens etc.	Branch and bound method is applied to the problems like Travelling Sales Person (TSP), Job sequencing etc.

Table 7.1.1: Comparisons between Branch and Bound Method and Backtracking Method

Unlike the dynamic programming, branch and bound method can be applied to proximity matrices. In other words, branch and bound is more efficient than dynamic programming of proximity matrices. This is due to the branch and bound consumes less amount of memory when compared to dynamic programming.

## 7.2 GENERAL METHOD

In the branch and bound method, searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.

We assume that each answer node  $x$  has a cost  $C(x)$  associated with it and that is a minimum. Cost answer node is to be found.

Branch and bound is the generalization of both BFS and D-search. In this technique BFS like state space search will be called FIFO search as the list of

live nodes in a First In First Out list (queue). Here, the D-search like state space search is called as LIFO search as the list of live nodes in a Last In First Out (stack).

In this branch and bound method a space tree of possible solutions is generated. Then partitioning (called as branching) is done at each node of the tree. Lower bound and upper bound at each node is computed. This computation leads to selection of answer node. In this bounding functions are to avoid the generation of subtrees that do not contain an answer node.

The three types of search strategies in branch and bound are as follows,

- 1) FIFO search
- 2) LIFO search
- 3) Least Cost (LC) search.

### 7.2.1 LC (Least Cost) Search

In both LIFO and FIFO the branch and bound, the selection rule for the next E-node is blind and rigid. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. The search for an answer node can be speeded by using an intelligent ranking function  $\hat{c}(.)$  for live nodes. The node X is assigned a rank, using,

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

Where,

$\hat{c}(x)$  → It is the cost of x.

$h(x)$  → It is the cost of reaching x from the root.

$f(.)$  → Is any non-decreasing function and

$\hat{g}(x)$  → It is an estimate of the additional effort needed to reach an answer node from x.

A search strategy that uses a cost function,  $\hat{c}(x) = f(h(x)) + \hat{g}(x)$  to select the next E-node would always choose for its next E-node at live node with least  $\hat{c}(.)$  is called as an LC-search (least cost search).

- 1) If  $g = 0$ ,  $f = 1$ , and  $h(x)$  is the level of node  $x$ , this LC-search is a BFS algorithms which generates nodes by levels.
- 2) If  $f = 0$  and  $g(y) < g(x)$  when  $y$  is the child of  $x$ , this LC-search is a D-search.

The cost function  $c(\cdot)$  is defined as follows,

- 1) If  $x$  is an answer node, then  $c(x)$  is the cost of reaching  $x$  from root to the state space tree.
- 2) If  $x$  is not an answer node, but the subtree of  $x$  contains an answer node, then  $c(x)$  is the minimal cost of an answer node in subtree  $x$ .
- 3) Otherwise,  $c(x) = \infty$ .

Then,  $\hat{c}(\cdot)$  with  $f = 1$ , that is  $\hat{c}(x) = h(x) + g(x)$ , is an estimate of  $c(\cdot)$ . It will normally have the additional property that if  $x$  is an answer node or leaf node, then  $\hat{c}(x) = c(x)$ .

Here, the ranking function ( $\hat{c}(\cdot)$ ) or cost function which depends on the problem.

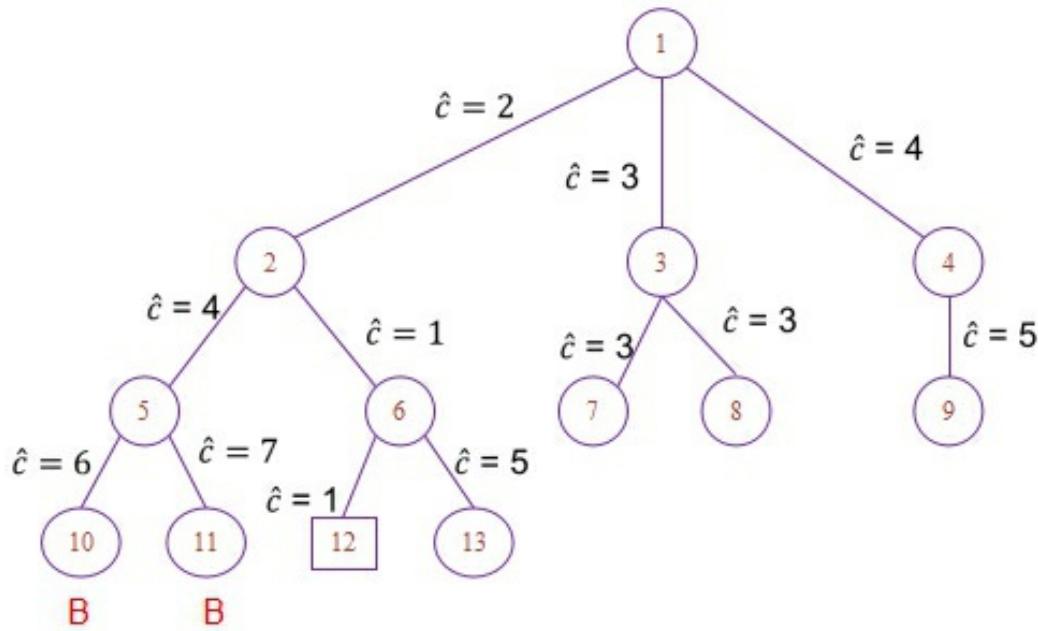


Fig. 7.2.1: State Space Tree

Initially, we will take the node 1 as the E-node. Generate children of node 1, the children are 2, 3, 4. By using ranking function we will calculate

the function of 2, 3, 4 nodes is  $\hat{c} = 2$ ,  $\hat{c} = 3$ ,  $\hat{c} = 4$  respectively. Now we will select a node which has minimum cost, i.e., node 2. For node 2 the children are 5, 6. Between 5 and 6 we will select 6 because its cost is minimum. Generate children of node 6 i.e., 12 and 13. Now, we will select node 12 since its cost ( $\hat{c} = 1$ ) is minimum. Moreover its answer is node 12. So, we terminate search process.

### 7.2.1.1 Control Abstraction for LC Search

Let  $t$  be a state space tree and  $c(\cdot)$  a cost function for the nodes in  $t$ . If  $x$  is a node in  $t$ , then  $c(x)$  is the minimum cost of any answer node in the subtree with the root  $x$ .  $c(t)$  is the cost of a minimum cost answer node in  $t$ . Usually it is not possible to find easily computable function  $c(\cdot)$ . An estimate  $\hat{c}$  of  $c(\cdot)$  is used. It is easy to compute  $\hat{c}$  and has the property that if  $x$  is either an answer node or leaf node then  $c(x) = \hat{c}(x)$ .

The algorithm for LC Search is shown below.

```
Algorithm LCSearch(t)
{
    if(t is an answer node) then
    {
        write(t);
        return;
    }
    E=t;
    repeat
    {
        for(each child x of E) do
        {
            if(x is answer node) then
            {
                output the path from x to t;
                return;
            }
            Add(x);
            x → parent = E; //Pointing the path from x to root
        }
        if(no more live nodes) then
        {
            write("No answer Node");
            return;
        }
        E = Least();
    }until(false);
}
```

### Algorithm 7.2.1: Least Cost Search

A LCBB search of tree uses two functions Least() and Add(). Begin with an upper bound with  $\infty$  and node 1 as the first node. When node 1 is expanded, nodes 2, 3, 4, ... are generated in the order. Add a live node to the list of five nodes if the Least() finds least cost node and the node is deleted from the list of live nodes and returned. The output of least cost search is tracing the path from the answer node to the root node of the tree.

**Example:** We will see this algorithm with an example.

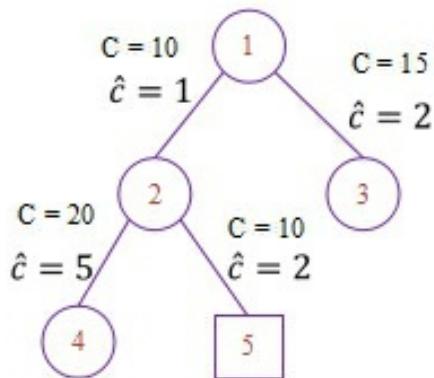


Fig. 7.2.2: LC Branch and Bound Tree

Here, assume 5 is an answer node.

At 4<sup>th</sup> line for loop starts execution i.e., the childrens of node 1 are generated. Every time the child is checked by condition, i.e., child is answer node or not. If it is answer node terminate the search, otherwise we select a node which has minimum cost (ranking function value).

For the nodes 2, 3 the condition is false, so we select 2 as the next E-node since its cost is minimum  $\hat{c}$ . Generate children of nodes 2, i.e., 4, 5 are children of 2, again we will check condition, here condition is true for node 5 since it is an answer node. So we terminate the search process. Above LC Search algorithm gives answer node but it may not give the optimal answer node.

#### 7.2.1.2 Properties of LC Search

The properties of LC Search is as follows,

- 1) LC Search is desirable to find an answer node that has minimum cost among all the answer nodes. But LC Search cannot guarantee to find an answer node G with minimum cost  $c(G)$ .
- 2) When there exists two nodes in a graph such that  $c(x) > c(y)$  in one branch while  $c(x)$  being less than  $c(y)$ .  $c(x)$  is less than  $(c(y).c(x) < c(y))$  in other branch, LC Search cannot find minimum cost answer node.
- 3) Even if  $c(x) < c(y)$  for every pair of nodes x, y such that  $c(x) < c(y)$ , LC may not find a minimum cost answer node.
- 4) When the estimate  $\hat{c}(.)$  for a node is less than the cost  $c(.)$  then a slight modification to the LC search results in search algorithm that terminates when a minimum cost answer node is reached. In this modification, the search continues until an answer node becomes an E-node.
- 5) At the time E-node is an answer node  $\hat{c}(E) \leq \hat{c}(L)$  for every node on a graph L on the list of two live nodes. Hence,  $c(E) \leq c(L)$  and so E is the minimum cost answer node.

### 7.2.1.3 15-Puzzle Problem Example

The 15-puzzle problem was invented by Sam Loyd in 1878. It consists of 15 numbered tiles on a square frame with a capacity of 16 tiles as shown in the below figure.

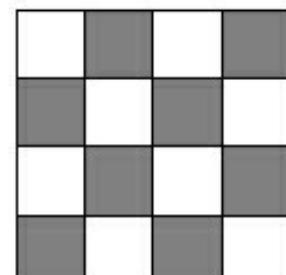
We are given an initial arrangement (a) of the tiles, and the objective is to transform this arrangement into the goal arrangement of the below figure (b) through a series of legal moves. The only legal moves are ones in which a tile adjacent of the Empty Spot (ES) moved to ES.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(a) An arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(a) Goal arrangement



(c)

Fig. 7.2.3: 15-puzzle problem

Thus, from the intital arrangement of Fig (a), four moves are possible. We can move any one of the tiles numbered 2, 3, 5 or 6 to the empty spot. Following this move, the other moves can be made. Each move creates new arrangement of the tiles. These arrangement s are called as states of the puzzle. The initial and goal arrangements are called as the initial and goal states. A state is reachable from the initial state if there is a sequence of legal moves from the initial to this state. The state space of an initial state consists of all states that can be reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the path from the initial state to the goal state as the answer. It is easy to see that there are  $16!$  ( $16! = 20.9 \times 10^{12}$ ) different arrangements of the titles on the frame.

Out of these only one-half are reachable from any given initial state,. Indeed, the state space for the problem is very large. Before the attempt to search this state space for the goal state, it would be worthwhile to determine weather the goal state is reachable from the initial state. There is avery simple way to do this. Let us number the frame positions 1 to 16 position i is the same frame position containing tile numbered i in the goal numbered in the above figure (b). Position 16 is the empty spot. Let position(i) be the position number in the initial state of the tile numbered i. The position of 16 will denote the position of the empty spot.

For any state let less(i) be the number of tiles j, such that  $j < i$  and  $\text{position}(j) > \text{position}(i)$  for the state of above figure (a). We have, for example,  $\text{less}(1) = 0$ ,  $\text{less}(4) = 1$  and  $\text{less}(12) = 6$ . Let  $x = 1$  if in the initial state the empty spot is at one of the shaded positions of the above figure (c) and  $x = 0$  if it is at one of the remaining positions.

### 7.2.2 Bounding

A branch and bound method, searching a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost  $c(x)$  associated with it that a minimum cost answer node is to be found. Three common branch and bound search strategies are three common branch and bound strategies are FIFO, LIFO and LC. A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.

- 1) A cost function  $\hat{c}(.)$  such that  $\hat{c}(x) \leq c(x)$  is used to provide lower bounds on solutions obtainable from any node  $x$ .
- 2) If  $\text{upper}$  is an upper bound on the cost of minimum cost solution, then all the live nodes  $x$  with  $\hat{c}(x) > \text{upper}$  may be killed as all answer nodes reachable from  $x$  have cost  $c(x) \geq \hat{c}(x) > \text{upper}$ .
- 3) The string value of the  $\text{upper}$  can be obtained by some heuristic or can be said to  $\infty$ .
- 4) As long as initial value for  $\text{upper}$  is not less than the cost of a minimum cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum cost answer node.
- 5) Each time a new answer node is found, the value of  $\text{upper}$  can be updated.

Branch and bound algorithms are used for optimization problem is easily converted to a minimization problem by changing the sign of the objective function.

To formulate the search for an optimal ssolution for a least cost answer node in a state space tree, it is necessary to define the cost function  $c(.)$ , such that  $c(x)$  is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for  $c(.)$ .

- 1) For the nodes which represents the feasible solution  $c(x)$  is the value of the objective function for that feasible solution.
- 2) For the nodes repsenting in feasible solutions,  $c(x) = \infty$ .

- 3) For the nodes representing the partial solutions  $c(x)$  is the cost of minimum cost node in the subtree with root  $x$ .
- 4) Since,  $c(x)$  is generally hard to compute, the branch-and-bound algorithm will use an estimate  $\hat{c}(x)$  such that  $\hat{c}(x) \leq c(x)$  for all  $x$ .

**Example:**

Consider instance of job sequencing with deadlines problem,  $n = 4$ ,  $(P_1, P_2, P_3, P_4) = (5, 10, 6, 3)$ ,  $(d_1, d_2, d_3, d_4) = (1, 3, 2, 1)$ ,  $(t_1, t_2, t_3, t_4) = (1, 2, 1, 1)$ .

The function  $\hat{c}(x)$  can be computed for each node  $x$  as,

$$\hat{c}(x) = \sum_{i < m, j \notin S_x} P_i$$

where,  $m = \max\{i / i \notin S_x\}$  and

$S_x$  be the subset of jobs selected for  $j$  at node  $x$ .

The upper bound  $u(x)$  on the cost of minimum cost answer node in the subtree  $x$  is  $u(x) = \sum_{j \notin S_x} P_i$ .

Here,  $u(x)$  is the cost of the solution  $S_x$  corresponding to node  $x$ .

The solution space for this instance consists of all possible subsets of the job index set. The solution space can be organized into a tree by means of either of the two formulations. Here, we consider variable tuple size formulation.

For the node 1 (root) there are 4 children. These children are for selection of either 1 or 2 or 3 or for job 4. Hence,  $x_1 = 2$  or  $x_1 = 3$  or  $x_1 = 4$ .

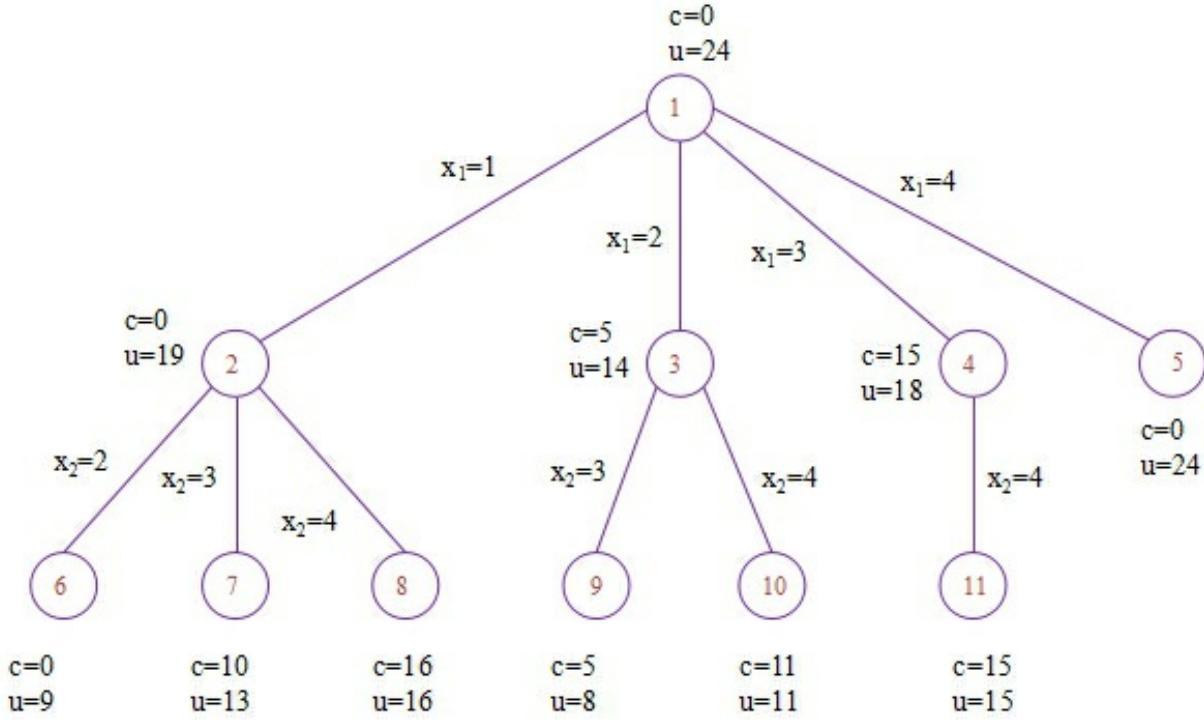


Fig. 7.2.4: State Space Tree With Variable Tuple Size Formulation

For the node 2 we have  $x_1 = 1$  and therefore we can select either 2, 3 or 4. Hence  $x_2 = 2$  or  $x_2 = 3$  or  $x_2 = 4$  corresponds to node 6, 7 or 8. Continuing in this fashion, we have drawn the state space tree.

**At Node 1:**

$$\hat{c} = 0$$

$$u = 24 \quad \because \sum_{i=1}^n P_i = 5 + 10 + 6 + 3$$

**At Node 2:**

$$\hat{c} = 0 \quad \because \sum P_i \text{ where } i < 1 = 0$$

$$u = 19 \quad \because \sum P_i \text{ excluding } x_1 = 1, \text{ i.e., } p_1 = 5$$

**At Node 3:**

$$\hat{c} = 5 \quad \because \sum P_i \text{ where } i < 2$$

$$u = 14 \quad \because \sum P_i \text{ excluding } i = 2 \text{ i.e., } 21 - 10 = 14$$

**At Node 4:**

$$\hat{c} = 15 \quad \because \sum P_i \text{ where } i < 3$$

$$u = 18 \quad \because \sum P_i \text{ excluding } i = 3 \text{ i.e., } 24 - 6 = 18$$

**At Node 5:**

$$\hat{c} = 21 \quad \because \sum P_i \text{ where } i < 4$$

$$u = 21 \quad \because \sum P_i \text{ excluding } i = 4 \text{ i.e., } 24 - 3 = 21$$

**At Node 6:**

$$\hat{c} = 0 \quad \because \sum_{i=2}^4 P_i \text{ excluding } i = 1 \text{ i.e., } P_1$$

$$u = 9 \quad \because \sum_{i=1}^4 P_i \text{ excluding } i = 2 \text{ and } i = 1. \text{ Hence } 6 + 3 = 9$$

**At Node 7:**

$$\hat{c} = 10 \quad \because \sum_{i=1}^3 P_i \text{ excluding } i = 1 \text{ and } i = 3. \text{ Hence } P_2 = 10$$

$$u = 13 \quad \because \sum_{i=1}^4 P_i \text{ excluding } i = 1 \text{ and } i = 4. \text{ Hence } P_2 + P_3 = 16$$

**At Node 8:**

$$\hat{c} = 16 \quad \because \sum_{i=1}^4 P_i \text{ excluding } i = 1 \text{ and } i = 4. \text{ Hence } P_2 + P_3 = 16$$

$$u = 16 \quad \because \sum_{i=1}^4 P_i \text{ excluding } i = 1 \text{ and } i = 4. \text{ Hence } P_2 + P_4 = 16$$

**At Node 9:**

$$\hat{c} = 5 \quad \because \sum_{i=1}^3 P_i \text{ excluding } i = 2 \text{ and } i = 3. \text{ Hence } P_1 = 5$$

$$u = 8 \quad \because \sum_{i=1}^4 P_i \text{ excluding } i = 2 \text{ and } i = 3. \text{ That is } P_1 + P_4 = 8$$

**At Node 10:**

$\hat{c} = 11 \quad \because \sum_{i=1}^n P_i < 4 P_i$  excluding  $i = 2$  and  $i = 4$ . Hence  $P_1 + P_3 = 5 + 6 = 11$

$u = 11 \quad \because \sum_{i=1}^n P_i$  excluding  $i = 2$  and  $i = 4$ . That is  $P_1 + P_3 = 5 + 6 = 11$

### At Node 11:

$\hat{c} = 15 \quad \because \sum_{i=1}^n P_i$  excluding  $i = 3$  and  $i = 4$ . Hence  $P_1 + P_2 = 5 + 10 = 15$

$u = 15 \quad \because \sum_{i=1}^n P_i$  excluding  $i = 3$  and  $i = 4$ . Hence  $P_1 + P_2 = 5 + 10 = 15$

### 7.2.3 FIFO Branch and Bound

A FIFO branch and bound algorithm for the job sequencing problem can begin with upper ( $u$ ) = 24 (since  $\sum P_i = 5 + 10 + 6 + 3 = 24$ ).

Starting with node 1 as E-node, node 2, 3, 4, 5 are generated,  $u(2) = 19$ ,  $u(3) = 14$ ,  $u(4) = 18$ ,  $u(5) = 21$ .

Maximum value of  $u(x)$  becomes upper. Hence upper will be updated to 14 when node 3 is generated. Since  $\hat{c}(4) > \text{upper}$ , i.e.,  $15 > 14$  and  $\hat{c}(5) > \text{upper}$ , i.e.,  $21 > 14$ , nodes 4 and 5 are killed. Nodes 2 and 3 becomes live nodes. Node 2 becomes the next E-node. Nodes 6, 7 and 8 are generated when it is expanded

Since,  $u(6) = 9$ ,  $u(7) = 13$ ,  $u(8) = 16$  and minimum of these is 9.

Hence, upper is updated to 9. The cost  $\hat{c}(7) > \text{upper}$ , i.e.,  $10 > 9$  and  $\hat{c}(8) > \text{upper}$ . Hence nodes 7 and 8 are killed. The node 3 becomes the next E-node. The nodes 9 and 10 are generated.

Since,  $u(9) = 8$  and  $u(10) = 11$ . Hence upper is updated to 8. The cost  $\hat{c}(10) > \text{upper}$  i.e.,  $11 > 8$  and this node is killed. The next E-node is 6. Both its children are feasible. The only remaining live nodes is 9. The only child of 9 is infeasible. The minimum cost answer node is node 9. It has a cost of 8.

### 7.2.4 LIFO Branch and Bound

An LCBB search of the tree begin with upper = 24 and node 1 as the first E-node. Nodes 2, 3, 4 and 5 are generated when node 1 is expanded. The upper is updated to 14 when node 3 is generated and since  $\hat{c}(4) > \text{upper}$  and  $\hat{c}(5) > \text{upper}$ , nodes 4 and 5 are killed. The next E-node is node 2 as  $\hat{c}(2) = 0$  and  $\hat{c}(3) = 5$ . Node 6, 7, and 8 are generated and upper is updated to 9. When node 6 is generated, node 7 is killed as  $\hat{c}(7) = 10 > \text{upper}$ .

Node 8 is infeasible and so killed. The only live nodes will be the nodes 3 and 6. Node 6 is the next E-node as  $\hat{c}(6) = 0 < \hat{c}(3)$ . Both its children are infeasible. Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as  $u(9) = 8$  since  $\hat{c}(10) > \text{upper}$  we will kill node 10. Now only remaining node is 9. It becomes the next E-node. Its only child is infeasible. As there are no live nodes, the search will terminate with node 9 representing with minimum cost answer node.

### 7.3 APPLICATIONS OF BRANCH AND BOUND METHOD

Travelling sales person and 0/1 Knapsack problems are some of the problems of branch and bound method.

#### 7.3.1 Travelling Sales Person Problem

If there are  $n$  cities and cost of traveling from one city to other city is given. A salesman has to start from any one of the city and has to vist all the cities exactly once and has to return to the starting place with shortest distance or minimum cost.

Let  $G = (V, E)$  be a directed graph defining an instance of the travelling salesperson problem. Let  $c_{ij}$  be the cost of the edge  $(i, j)$  and  $c_{ij} = \infty$  if  $(i, j) \notin E(G)$  and let  $|V| = n$ .

Assume that every tour starts and ends at vertex 1.

To use least cost branch and bound to search the traveling salesperson state space tree, we must define a cost function  $c(x)$  and two other funtions  $\hat{c}(x)$  and  $\hat{u}(x)$  such that  $\hat{c}(x) \leq c(x) \leq \hat{u}(x)$ .

**Reduced cost matrix:** A row or column is said to be reduced if it contains

atleast one zero and all remaining entries are non- negative. A matrix is reduced if every row and column is reduced.

- 1) If a constant  $t$  is chosen to be minimum entry in row  $i$  or coloumn  $j$  then subtracting it from all entries in row  $i$ (column  $j$ ) will introduce a zero into a row  $i$  (column  $j$ ).
- 2) The total amount subtracted from the columns row is lower bound on the length of a minimum cost tour and can be used as the  $\hat{c}(x)$  value for the root of state space tree.

With every node in stae space tree, we associate a reduced cost matrix.

Let  $A$  be the reduced cost matrix for node  $R$ . Let  $S$  be the child of  $R$  such that the edge  $(R, S)$  corresponds to including edge  $(i, j)$  in the tour. If  $S$  is not a leaf node then the reduced cost matrix for node  $S$  can be obtained as follows,

- 1) Change all entries in row  $i$  coloumn  $j$  of  $A$  to  $\infty$  (To prevent the use of any more edges leaving vertex  $I$  or entering vertex  $i$ ).
- 2) Set  $A(j, 1)$  to  $\infty$ . To prevent the use of edge  $(j, 1)$ .
- 3) Apply row reduction and column reduction except for rows and columns containing  $\infty$ .
- 4) The total cost for node  $S$  can be calclulated as

$$\hat{c}(S) = \hat{c}(R) + A(i, j) + r$$

Where  $r$  is the total amount subtracted in step 3.

### 7.3.1.1     **Solved Problem**

**Problem:**

**Solve the following instance of travelling salesperson problem using LCBB.**

	1	2	3	4	5
1	$\infty$	20	30	10	11
2	15	$\infty$	16	4	2
3	3	5	$\infty$	2	4
4	19	6	18	$\infty$	3
5	16	4	7	16	$\infty$

Solution:

**Row Reduction:**

$$\begin{array}{c}
 \left[ \begin{array}{cccc|c} \infty & 20 & 30 & 10 & 11 & 10 \\ 15 & \infty & 16 & 4 & 2 & 2 \\ 3 & 5 & \infty & 2 & 4 & 2 \\ 19 & 6 & 18 & \infty & 3 & 3 \\ 16 & 4 & 7 & 16 & \infty & 4 \end{array} \right] \\
 \text{21 (total)}
 \end{array} \quad \rightarrow \quad
 \begin{array}{c}
 \left[ \begin{array}{ccccc} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{array} \right]
 \end{array}$$

**Column Reduction:**

$$\begin{array}{c}
 \left[ \begin{array}{ccccc} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{array} \right] \\
 \begin{array}{ccccc} 1 & - & 3 & - & - \end{array} = 4
 \end{array} \quad \rightarrow \quad
 \begin{array}{c}
 \left[ \begin{array}{ccccc} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{array} \right]
 \end{array}$$

The total amount subtracted,  $r = 21 + 4 = 25$ .

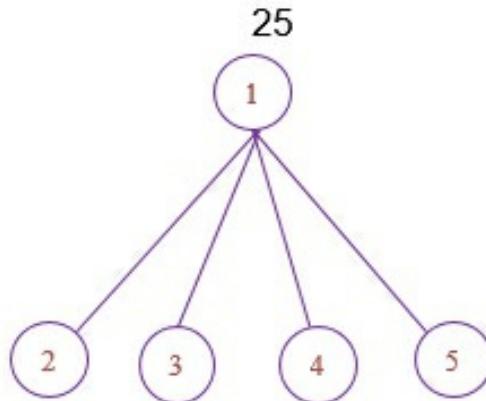


Fig. 7.3.1: Part of a State Space Tree

**Consider the path (1, 2) :** Change all entries of first row and second column of reduced matrix to  $\infty$  and set  $A(2, 1)$  to  $\infty$ .

$$\left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{array} \right]$$

Apply row reduction and column reduction,

$$\implies r = 0.$$

$$\begin{aligned} \hat{c}(2) &= \hat{c}(1) + A(1, 2) + r \\ &= 25 + 10 + 0 \\ &= 35 \end{aligned}$$

**Consider the path (1, 3) :** Change all entries of first row and third column of reduced matrix to  $\infty$  and set  $A(3, 1)$  to  $\infty$ .

$$\begin{array}{cc}
 \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{array} \right] & \xrightarrow{\hspace{1cm}} \\
 \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{array} \right]
 \end{array}$$

11

Applying row reduction and column reduction,

$$\implies r = 11$$

$$\begin{aligned}
 \hat{c}(3) &= \hat{c}(1) + A(1, 3) + r \\
 &= 25 + 17 + 11 \\
 &= 53
 \end{aligned}$$

**Consider the path (1, 4) :** Change all entries of first row and fourth column to  $\infty$  and set  $A(4, 1)$  to  $\infty$ .

$$\left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{array} \right]$$

Applying row reduction and column reduction,

$$\implies r = 0$$

$$\begin{aligned}
 \hat{c}(4) &= \hat{c}(1) + A(1, 4) + r \\
 &= 25 + 0 + 0 \\
 &= 25
 \end{aligned}$$

**Consider the path (1, 5) :** Change all entries of first row and fifth column to

$\infty$  and set  $A(5, 1)$  to  $\infty$ .

$$\begin{array}{c}
 \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{array} \right] \begin{matrix} 2 \\ 3 \\ 5 \end{matrix} \rightarrow \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{array} \right] \\
 \hline
 \end{array}$$

Applying row reduction and column reduction,

$$\Rightarrow r = 5$$

$$\begin{aligned}
 \hat{c}(5) &= \hat{c}(1) + A(1, 5) + r \\
 &= 25 + 1 + 5 \\
 &= 31
 \end{aligned}$$

Since the minimum cost is 25, select node 4.

The matrix obtained for path (1, 4) is considered as the reduced cost matrix.

$$A = \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{array} \right]$$

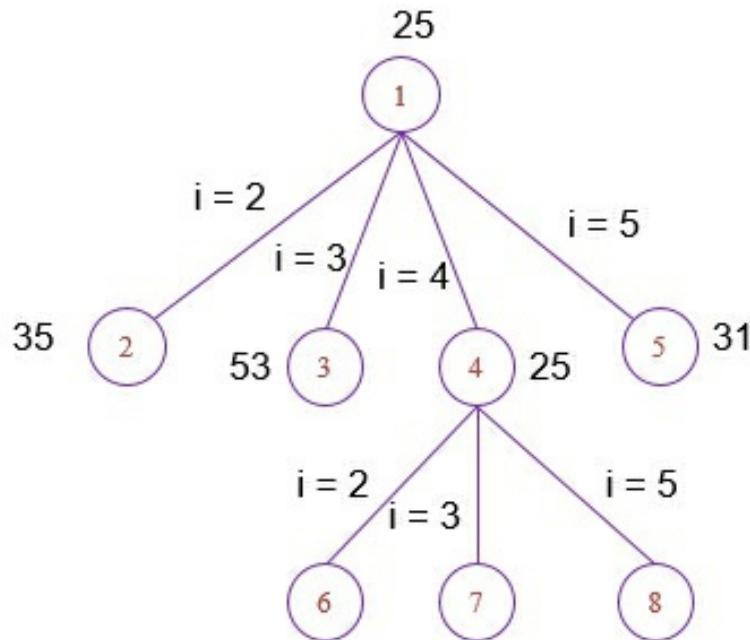


Fig. 7.3.2: Part of a State Space Tree

**Consider the path (4, 2) :** Change all entries of fourth row and second column reduced matrix of A to  $\infty$  and set  $A(2, 1)$  to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Applying row reduction and column reduction,

$$\implies r = 0$$

$$\begin{aligned} \hat{c}(2) &= \hat{c}(4) + A(4, 2) + r \\ &= 25 + 3 + 0 \\ &= 28 \end{aligned}$$

**Consider the path (4, 3) :** Change all entries of fourth row and third column

of A to  $\infty$  and set A(3, 1) to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \xrightarrow[11]{2} \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Applying row reduction and column reduction,

$$\Rightarrow r = 2 + 11 = 13$$

$$\begin{aligned} \hat{c}(3) &= \hat{c}(4) + A(4, 3) + r \\ &= 25 + 12 + 13 \\ &= 50 \end{aligned}$$

**Consider the path (4, 5) :** Change all entries of fourth row and fifth column to  $\infty$  and set A(5, 1) to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \xrightarrow[11]{}$$

Applying row reduction and column reduction.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

$$\implies r = 11 + 0 = 11$$

$$\begin{aligned}\hat{c}(5) &= \hat{c}(4) + A(4, 5) + r \\ &= 25 + 0 + 11 \\ &= 36\end{aligned}$$

Since the minimum cost is 28 select node 2.

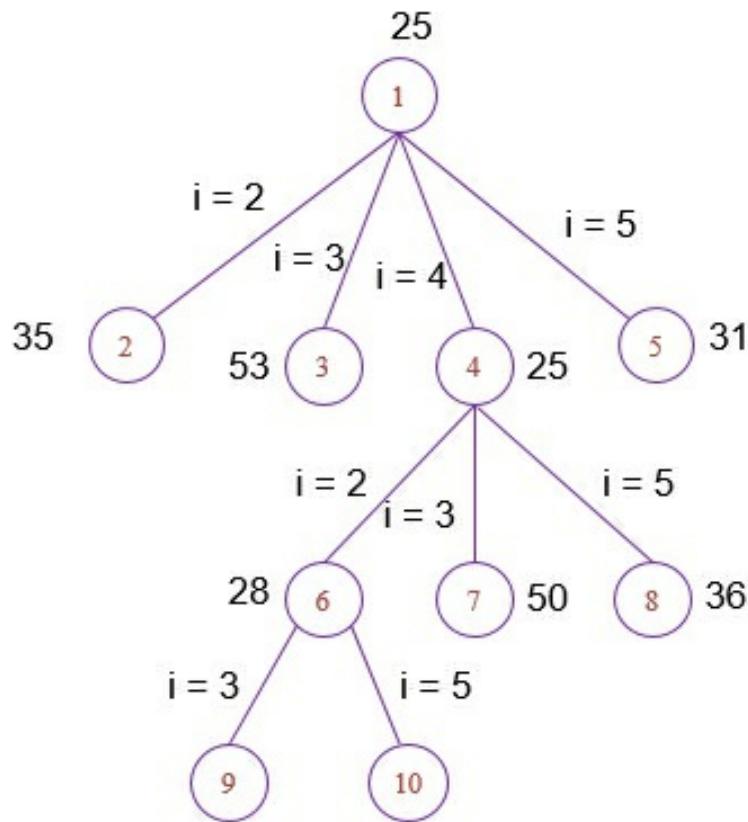


Fig. 7.3.3: Part of a State Space Tree

The matrix obtained for path (4, 2) is considered as the reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

**Consider the path (2, 3) :** Change all entries of second row and third column to  $\infty$  and set  $A(3, 1)$  to  $\infty$ .

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} \begin{matrix} \\ \\ 2 \\ \\ 11 \end{matrix}$$

Apply row reduction and column reduction,

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$\implies r = 2 + 11 = 13$$

$$\begin{aligned} \hat{c}(3) &= \hat{c}(2) + A(2, 3) + r \\ &= 28 + 11 + 13 \end{aligned}$$

$$= 52$$

**Consider the path (2, 5) :** Change all entries of second row and fifth column to  $\infty$  and set  $A(5, 1)$  to  $\infty$ .

$$\left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{array} \right]$$

Apply row reduction and column reduction,

$$\implies r = 0$$

$$\hat{c}(3) = \hat{c}(5) + A(5, 3) + r$$

$$= 28 + 0 + 0$$

$$= 28$$

Since the minimum cost is 28, select node 5.

The matrix obtained for path (2, 5) is considered as the reduced cost matrix.

**Consider the path (5, 3) :** Change all entries in fifth row and third column to  $\infty$  and set  $A(3, 1)$  to  $\infty$ .

$$\left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{array} \right]$$

Apply row reduction and column reduction,

$$\implies r = 0$$

$$\begin{aligned}
 \hat{c}(3) &= \hat{c}(5) + A(5, 3) + r \\
 &= 28 + 0 + 0 \\
 &= 28
 \end{aligned}$$

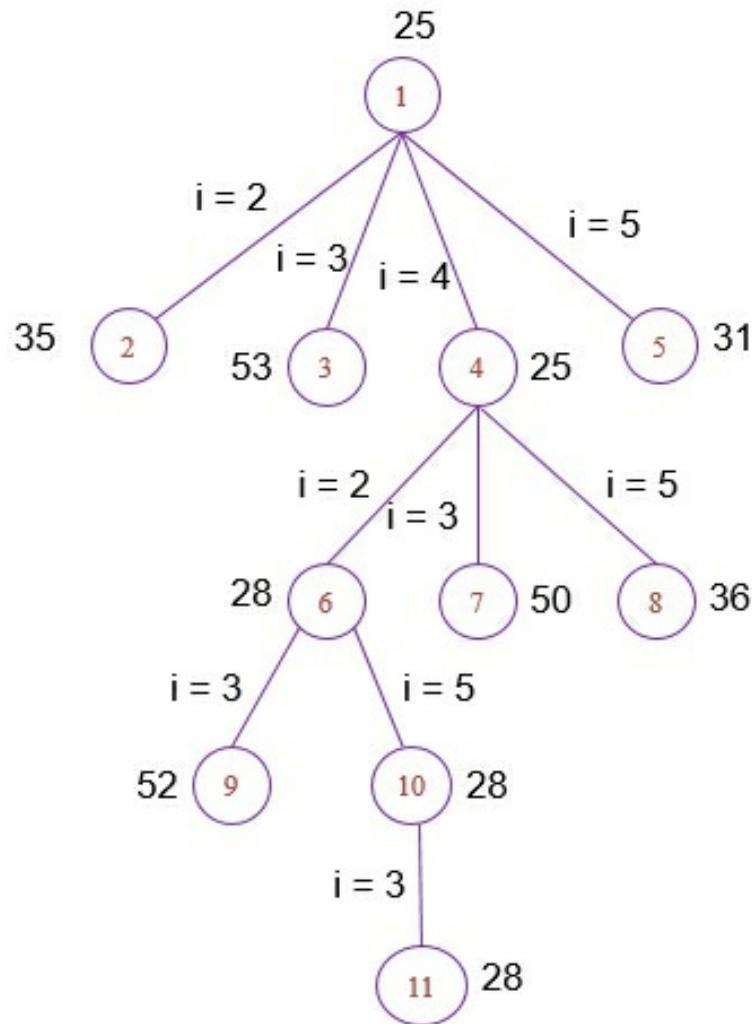


Fig. 7.3.4: State Space Tree

The path is  $1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 1$  and

Minimum cost  $= 10 + 6 + 2 + 7 + 3 = 28$ .

### 7.3.2 0/1 Knapsack Problem

#### 7.3.2.1 Using LC Branch and Bound Solution

The 0/1 Knapsack problem states that, there are  $n$  objects given and capacity of knapsack is  $m$ . Then, select some objects to fill the knapsack in

such a way that it should not exceed the capacity of knapsack and maximum profit can be earned.

The 0/1 knapsack problem can be stated as,

$$\text{Max } z = p_1x_1 + p_2x_2 + p_3x_3 + \dots + p_nx_n$$

Subject to  $w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_mx_m \leq m$ ,  $x_i = 0$  or  $1$ .

A branch and bound technique is used to find solution to the knapsack problem. But we cannot directly apply the branch and bound technique to the knapsack problem. Because the branch and bound deals only with the minimization problems. We modify the knapsack problem to the minimization problem. The modified problem is,

$$\text{Min } z = -p_1x_1 - p_2x_2 - p_3x_3 - \dots - p_nx_n$$

Subject to  $w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_mx_m \leq m$ ,  $x_i = 0$  or  $1$ .

Let  $\hat{c}(x)$  and  $\hat{u}(x)$  are the two cost function such that  $\hat{c}(x) \leq c(x) \leq \hat{u}(x)$ , satisfying the requirements where  $c(x) = -\sum p_i x_i$ . The  $c(x)$  is the cost function for answer node  $x$ , which lies between two functions called lower and upper bounds for the cost function  $c(x)$ . The search begins at the root node. Initially we compute the lower and upper bounds at root node called  $\hat{c}(1)$  and  $\hat{u}(1)$ . Consider the first variable  $x_1$  to take a decision. The  $x_1$  takes values 0 or 1. Compute the lower and upper bounds in each case of the variable. These are the nodes at the first level. Select the node whose cost is minimum i.e.,

$$c(x) = \min\{c(l\text{child}(x)), c(r\text{child}(x))\}$$

$$c(1) = \min\{\hat{c}(2), \hat{c}(3)\}$$

The problem can be solved by making a sequence of decisions on the variable  $x_1, x_2, x_3, \dots, x_n$  level wise. A decision on the variable  $x_1$  involves determining which of the values 0 or 1 is to be assigned, to it by defining  $c(x)$  recursively.

The path from root to the leaf node whose height is maximum is selected is the solution space for the knapsack problem. The algorithm for the upper bound of the Knapsack problem is as follows,

### 7.3.2.1.1 Solved Problem

**Problem 1: Draw a portion of state space tree generated by LCBB by the following Knapsack Problem.  $n = 5$ ,  $(p_1, p_2, p_3, p_4, p_5) = (10, 15, 6, 8, 4)$ ,  $(w_1, w_2, w_3, w_4, w_5) = (4, 6, 3, 4, 2)$  and  $m = 12$ .**

Solution:

Convert the profit to negative,  $(p_1, p_2, p_3, p_4, p_5) = (-10, -15, -6, -8, -4)$ .

Calculate the lower bound and upper bound for each node.

Place the first item in bag, i.e., 4, remaining weight is  $12 - 4 = 8$ .

Place second item in bag, i.e., 6, remaining weight is  $8 - 6 = 2$ .

Since fractions are not allowed in calculation of upper bound, so we cannot place the third and fourth items. Place fifth item.

$\therefore$  Profit earned =  $-10 - 15 - 4 = -29$  = upper bound.

To calculate the lower bound, place third item in a bag since fractions are allowed.

$$\therefore \text{Lower bound} = -10 - 15 - \frac{2}{3} \times 6 = -29$$

$$\therefore \hat{u}(1) = -29, \hat{c}(1) = -29$$

**For the node 2, ( $x_1 = 1$ )** means, we should place first item in the bag.

$$\hat{c}(2) = -10 - 15 - \frac{2}{3} \times 6 = -29$$

$$\hat{u}(2) = -10 - 15 - 4 = -29$$

**For the node 3, ( $x_1 = 0$ )**

$$\hat{c}(3) = -15 - 6 - \frac{3}{4} \times 8 = -15 - 6 - 6 = -27$$

$$\hat{u}(3) = -15 - 6 - 4 = -25$$

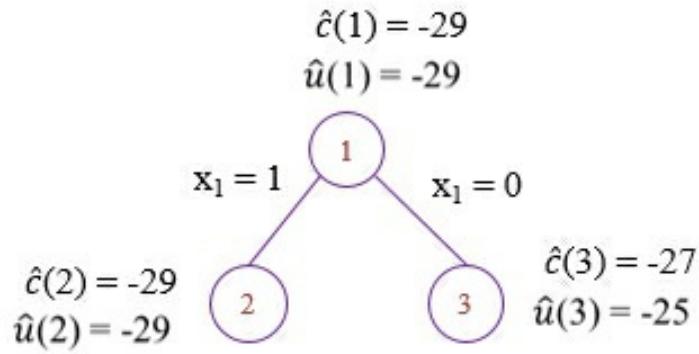


Fig. 7.3.5: Part of a LCBB Tree

Select the minimum of lower bounds, i.e.,

$$\min\{\hat{c}(2), \hat{c}(3)\} = \min\{-29, -27\}$$

$$= -29$$

$$= \hat{c}(2)$$

∴ Choose the node 2.

∴ First object is selected i.e.,  $x_1 = 1$ .

Consider the second variable to take the decision at second level.

**For the node 4, ( $x_2 = 1$ )**

$$\hat{c}(4) = -10 - 15 - \frac{2}{3} \times 6 = -29$$

$$\hat{u}(4) = -10 - 15 - 4 = -29$$

**For the node 5, ( $x_2 = 0$ )**

$$\hat{c}(5) = -10 - 6 - 8 - \frac{1}{2} \times 4 = -10 - 6 - 8 - 2 = -26$$

$$\hat{u}(5) = -10 - 6 - 8 = -24$$

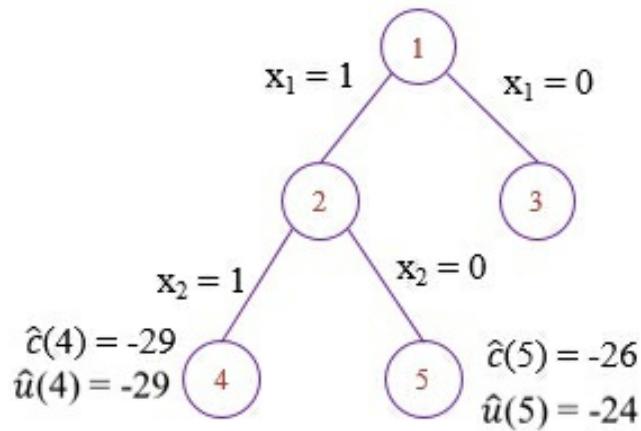


Fig. 7.3.6: Part of LCBB Tree

- ∴ For node 5  $\min\{\hat{c}(4), \hat{c}(5)\} = \min\{-29, -26\} = -29$
- ∴ Node 4 is selected.
- ∴ Second object is selected i.e.,  $x_2 = 1$ .

**For the node 6, ( $x_3 = 1$ )**

$$\hat{c}(6) = -10 - 15 - \frac{2}{3} \times 6 = -29$$

$$\hat{u}(6) = -15 - 6 - 4 = -25$$

**For the node 7, ( $x_3 = 0$ )**

$$\hat{c}(7) = -10 - 15 - \frac{2}{3} \times 6 = -29$$

$$\hat{u}(7) = -10 - 15 - 4 = -29$$

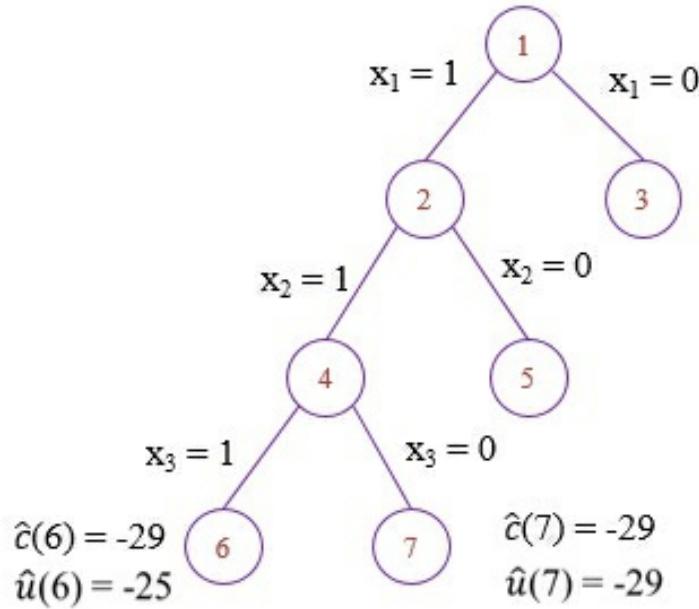


Fig. 7.3.7: Part of LCBB Tree

Since the lower bound are same, select the minimum of upper bound.

$$\therefore \min\{\hat{u}(6), \hat{u}(7)\} = \min\{-25, -29\} = -29$$

$\therefore$  Node 7 is selected.

$\therefore$  Third object is not selected i.e.,  $x_3 = 0$ .

**For the node 8, ( $x_4 = 1$ )**

$$\hat{c}(8) = -10 - 15 - \frac{2}{4} \times 8 = -29$$

$$\hat{u}(8) = -15 - 8 - 4 = -27$$

**For the node 9, ( $x_4 = 0$ )**

$$\hat{c}(9) = -10 - 15 - \frac{2}{3} \times 6 = -29$$

$$\hat{u}(9) = -10 - 15 - 4 = -29$$

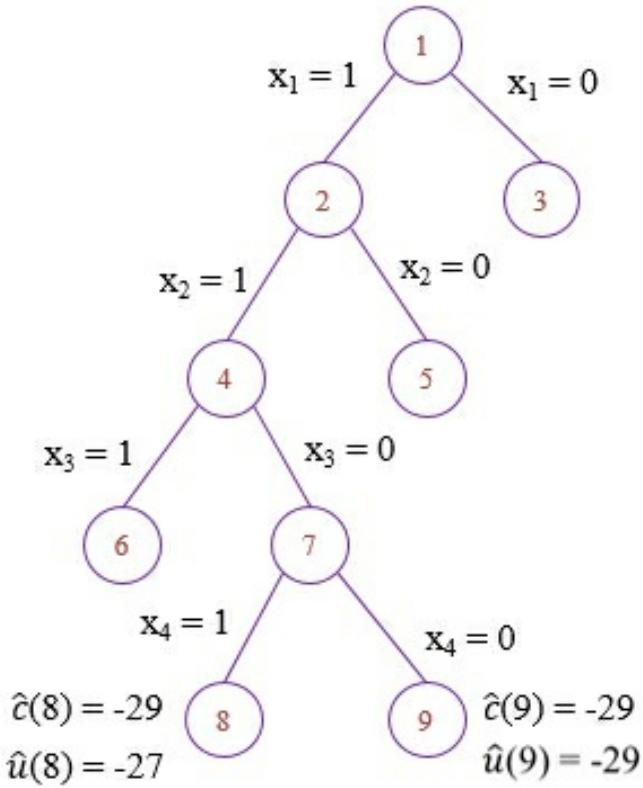


Fig. 7.3.8: Part of LCBB Tree

$$\therefore \min\{\hat{u}(8), \hat{u}(9)\} = \min\{-27, -29\} = -29$$

$\therefore$  Node 9 is selected.

$\therefore$  Fourth object is not selected i.e.,  $x_4 = 0$ .

**For the node 10, ( $x_5 = 1$ )**

$$\hat{c}(10) = -10 - 15 - 4 = -29$$

$$\hat{u}(10) = -10 - 15 - 4 = -29$$

**For the node 11, ( $x_5 = 0$ )**

$$\hat{c}(11) = -10 - 15 = -25$$

$$\hat{u}(11) = -10 - 15 = -25$$

$$\therefore \min\{\hat{u}(10), \hat{u}(11)\} = \min\{-29, -25\} = -29$$

- ∴ Node 10 is selected.
- ∴ Fifth object is selected  $x_5 = 1$ .

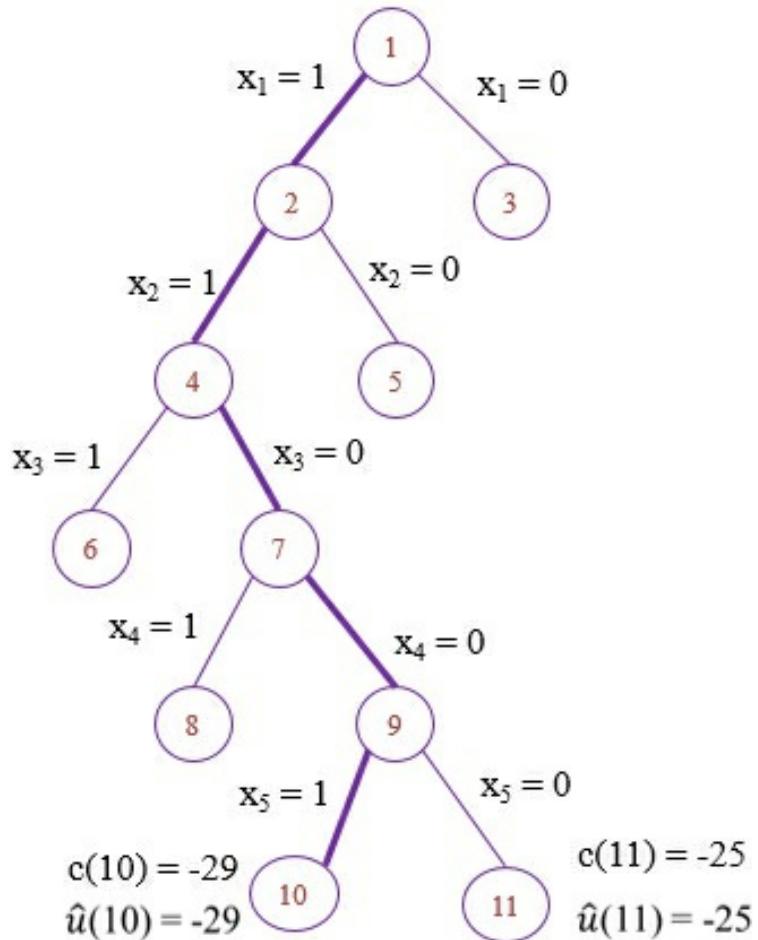


Fig. 7.3.9: LCBB Tree

∴ The path is 1 – 2 – 4 – 7 – 9 – 10.

The solution for 0/1 Knapsack problem is

$$(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 0, 1)$$

$$\text{Maximum profit} = 10 + 15 + 4 = 29.$$

### 7.3.2.2 FIFO Branch and Bound Solution

The problem here in FIFO branch and bound is to find the most valuable subset of the items that fit in the Knapsack where the given (n) items are of

known weights  $w_i$  and values  $v_i$ .

Considering the Knapsack example with,

$$n = 4, (v_1, v_2, v_3, v_4) = (10, 10, 12, 18)$$

$$(w_1, w_2, w_3, w_4) = (2, 4, 6, 9) \text{ and } w = 15$$

The FIFOBB algorithm proceeds with node 1 as the root node and makes it E-node. FIFOBB initializes  $\hat{u}(1)$  to -32 and hence nodes 2 and 3 are produced and therefore 2 and 3 are send to queue.

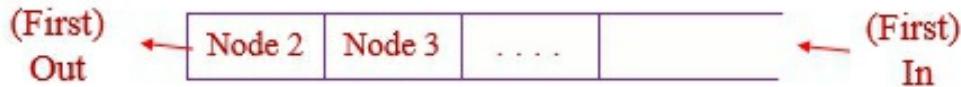


Fig (a): Queue

As node 2 is staying first in the queue, it becomes E-node and it produces node 4 and 5 as its children. Therefore, node 4 and 5 are queued and hence the queue becomes as shown below.

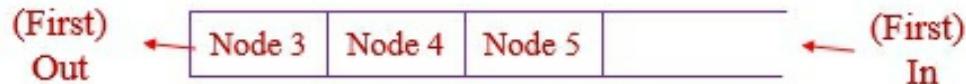


Fig (b): Queue

Node 3 is in the next E-node and then node 6 and 7 are generated. Then the node 7 is immediately killed because  $\hat{c}(7) >$  upper bound.

Then the nodes 8 and 9 are produced from the current E-node, i.e., node 4.

Also the upper bound is updated to -38,  $\hat{u}(9) = -38$

Now, the queue is as shown below.

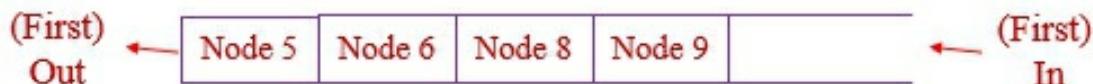


Fig (c): FIFO Search of Branch and Bound Method

Neither node 5 nor 6 are expanded because each  $\hat{c}() >$  upper bound and

hence node 8 is expanded. Their children nodes of node 8, node 10 and 11 are both killed. Since node 10 is infeasible, so it is killed.

The next from the queue is node 9 and it is expanded to node 12 and node 13. Node 13 is killed pushing node 12 into the queue. Therefore, the only live node in queue is node 12 and search terminates. Thus, FIFOBB solves a Knapsack problem.

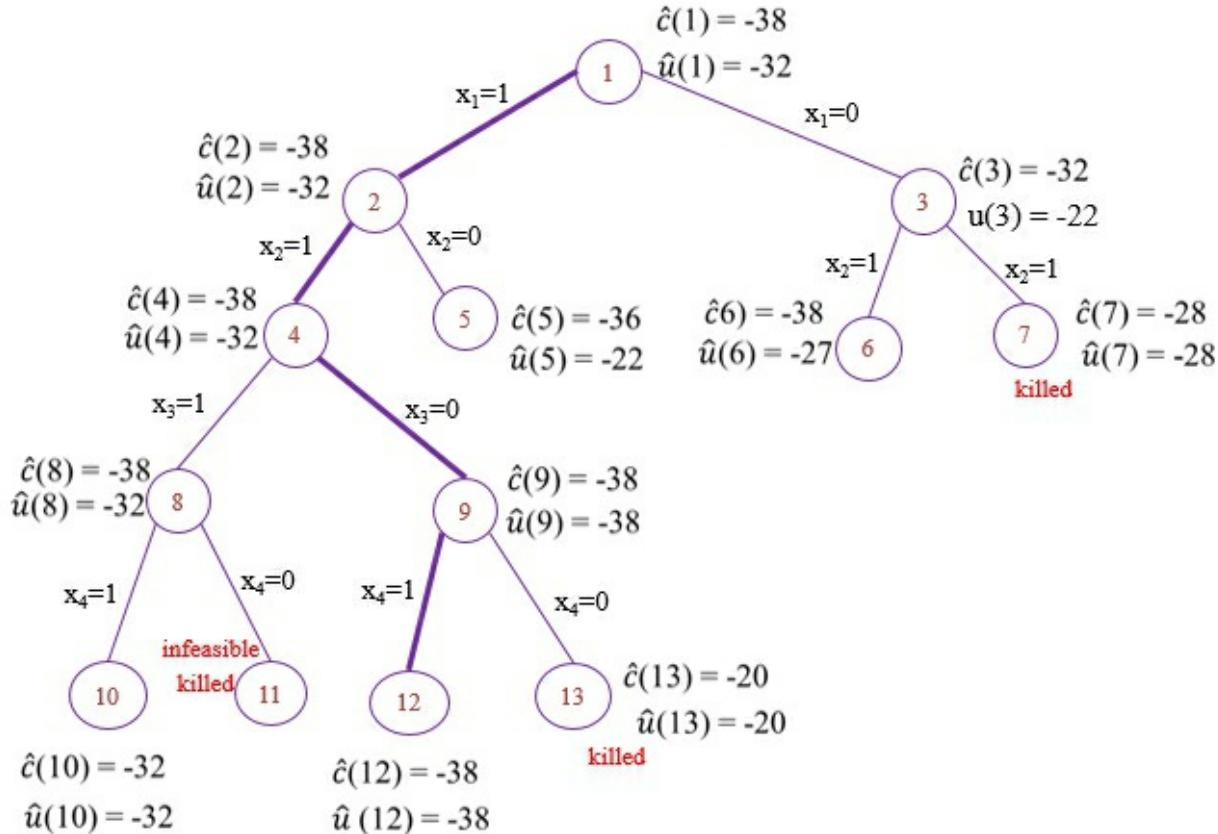


Fig. 7.3.10: FIFOBB Tree

### 7.3.2.2.1 Solved Problems

**Problem 1:** Draw the portion of the state space tree generated by LC Knap and FIFO Knap for the Knapsack instance.  $n = 4$ ,  $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$ ,  $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$  and  $m = 15$ .

Solution:

Knapsack is maximization problem, but the branch and bound technique is applied to the minimization problem only.

In order to convert it into minimization, take negative sign for profits.

$$(p_1, p_2, p_3, p_4) = (-10, -10, -12, -18)$$

In branch and bound technique, we calculate the lower bound and upper bound for each technique.

Place first item in the bag, i.e., remaining weight is  $15 - 2 = 13$ .

Place second and third items in the bag, remaining weight is  $13 - (4 + 6) = 3$ .

Since, fractions are not allowed in the caliculations of upper bound, so we cannot place fourth item in the bag.

$$\therefore \text{Profit earned} = -10 - 10 - 12 = -32 \text{ (upper bound)}$$

To calculate the lower bound, place fourth item in a bag, since fractions are allowed.

$$\text{Lower bound} = -10 - 10 - 12 - \left(\frac{3}{9} \times 18\right) = -10 - 10 - 12 - 6 = -38$$

$$\therefore \hat{u}(1) = -32 \text{ and } \hat{c}(1) = -38$$

After initialization,  $\hat{u}(1) = -32$ , the nodes 2 and 3 are produced and therefore nodes 2 and 3 are sent into the queue.



Fig (a) Queue

**For node 2 ( $x_1 = 1$ ),** that means we should place first item in the bag.

$$\hat{c}(2) = -10 - 10 - 12 - \left(\frac{3}{9} \times 18\right) = -10 - 10 - 12 - 6 = -38$$

$$\hat{u}(2) = -10 - 10 - 12 = -32$$

**For node 3 ( $x_1 = 0$ ),** that means we should not place first item in the bag.

$$\hat{c}(3) = -10 - 12 - \left(\frac{5}{9} \times 18\right) = -10 - 12 - 10 = -32$$

$$\hat{u}(3) = -10 - 12 = -22$$

Since node 2 is in the first of the queue, it becomes E-node and produces node 4 and 5 as children.

Therefore, node 4 and 5 are queued and the queue becomes as

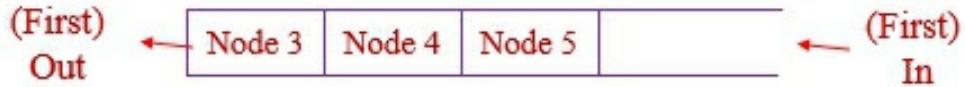


Fig (b) Queue

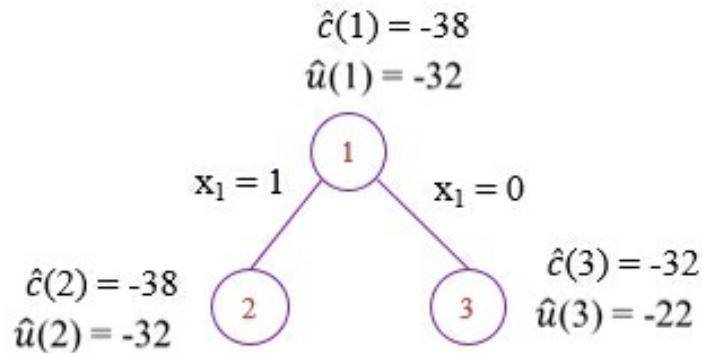


Fig. 7.3.11: Part of a FIFO Tree

Select the minimum of the lower bounds, i.e.,

$$\min\{\hat{c}(2), \hat{c}(3)\} = \min\{-38, -32\} = -38 = \hat{c}(2)$$

∴ Choose the node 2

∴ First object is selected i.e.,  $x_1 = 1$ .

Consider the second variable to take the decision at second level.

**For node 4 ( $x_2 = 1$ )**

$$\hat{c}(4) = -10 - 10 - 12 - \left(\frac{9}{3} \times 18\right) = -10 - 10 - 12 - 6 = -38$$

$$\hat{u}(4) = -10 - 10 - 12 = -32$$

**For node 5 ( $x_2 = 0$ )**

$$\hat{c}(5) = -10 - 12 - \left(\frac{7}{9} \times 18\right) = -10 - 12 - 14 = -36$$

$$\hat{u}(5) = -10 - 12 = -22$$

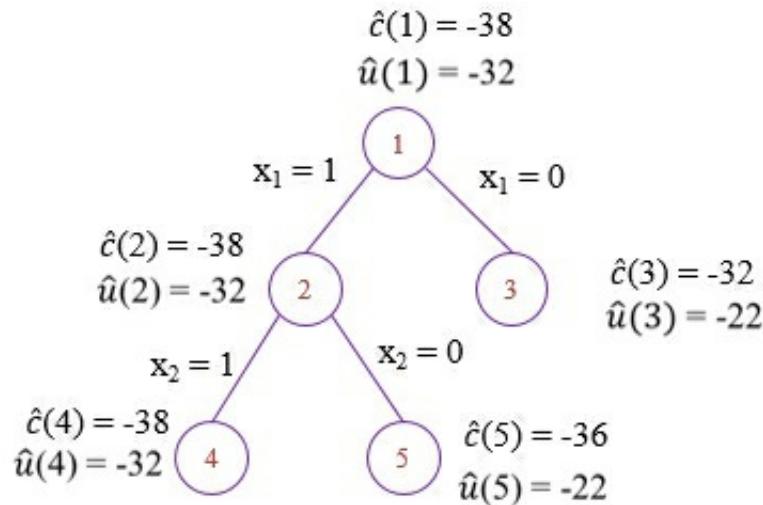


Fig. 7.3.12: Part of a FIFO Tree

Select the minimum of the lower bounds, i.e.,

$$\min\{\hat{c}(4), \hat{c}(5)\} = \min\{-38, -36\} = -38 = \hat{c}(4)$$

∴ Choose the node 4.

∴ Second object is selected i.e.,  $x_2 = 1$ .

Consider the third variable to take the decision at third level.

**For node 6 ( $x_3 = 1$ )**

$$\hat{c}(6) = -10 - 10 - 12 - \left(\frac{3}{9} \times 18\right) = -10 - 10 - 12 - 6 = -38$$

$$\hat{u}(6) = -10 - 10 - 12 = -32$$

**For node 7 ( $x_3 = 0$ )**

$$\hat{c}(7) = -10 - 10 - 18 = -38$$

$$\hat{u}(7) = -10 - 10 - 18 = -38$$

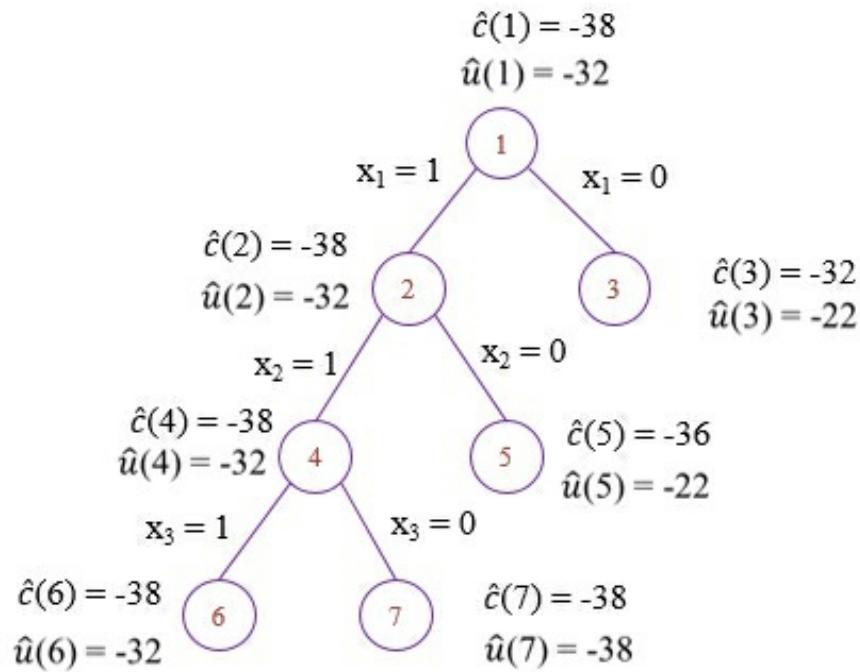


Fig. 7.3.13: Part of a FIFO Tree

Since the lower bounds are same, select the minimum of upper bounds.

$$\min\{\hat{u}(6), \hat{u}(7)\} = \min\{-32, -38\} = -38 = \hat{u}(7)$$

∴ Choose the node 7, i.e.,  $x_3 = 0$ .

Consider the fourth variable to take the decision at fourth level.

**For node 8 ( $x_4 = 1$ )**

$$\hat{c}(8) = -10 - 10 - 18 = -38$$

$$\hat{u}(8) = -10 - 10 - 18 = -38$$

**For node 9 ( $x_4 = 0$ )**

$$\hat{c}(9) = -10 - 10 = -20$$

$$\hat{u}(9) = -10 - 10 = -20$$

Now the queue is



Fig (c) Queue

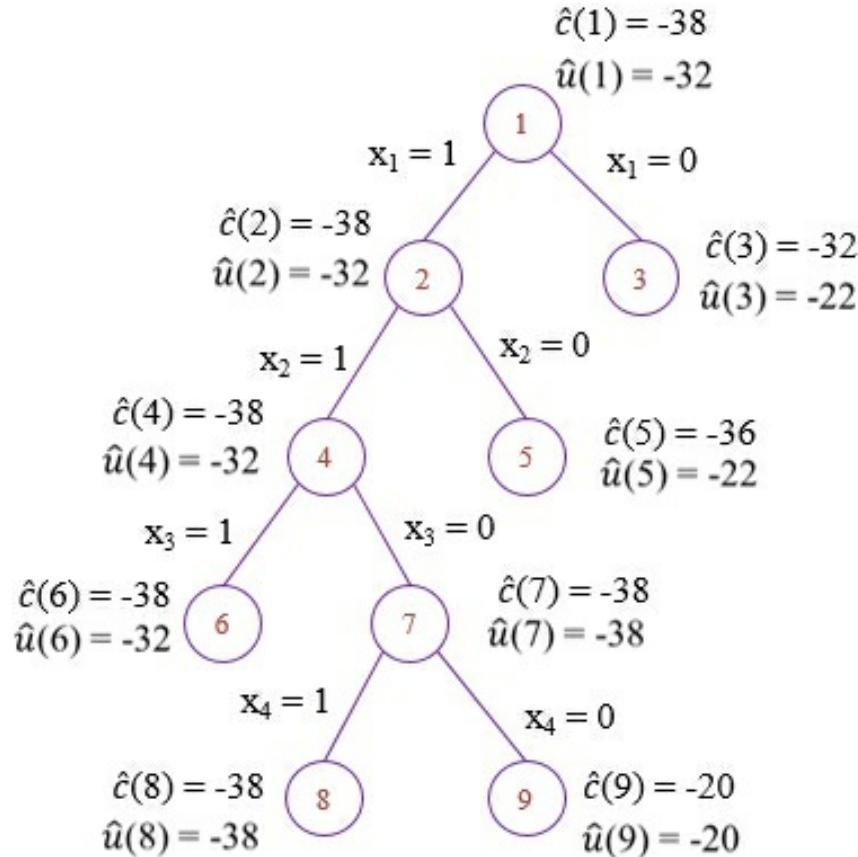


Fig. 7.3.14: Part of a FIFO Tree

Neither node 5 or 6 is expanded because  $\hat{c}() > \text{upper bound}$ . Hence, node 8 is expanded since it becomes the next E-node and its children nodes 10 and 11 are both killed (node 10 is infeasible, so killed). Because  $\hat{c}(11) > \text{upper}$ , kill node 11. Node 9 becomes the next E-node and  $\text{upper} = -38$ .

Next in the queue is node 9 and is expanded to 12 and 13.

Node 13 is killed pushing node 12 into the queue.

$\therefore$  Only live node in the queue is node 12 and search terminates.

Since, **for node 12 ( $x_4 = 1$ )**, place fourth item in the bag.

$$\hat{c}(12) = -10 - 10 - 18 = -38$$

$$\hat{u}(12) = -10 - 10 - 18 = -38$$

**For node 13 ( $x_4 = 0$ )**

$$\hat{c}(13) = -10 - 10 = -20$$

$$\hat{u}(13) = -10 - 10 = -20$$

Children 12 and 13 are generated. But  $\hat{c}(13) >$  upper, so kill node 13. Finally node 13 becomes an answer node.

Therefore the solution is  $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1$ .

FIFO BB solves the Knapsack problem.

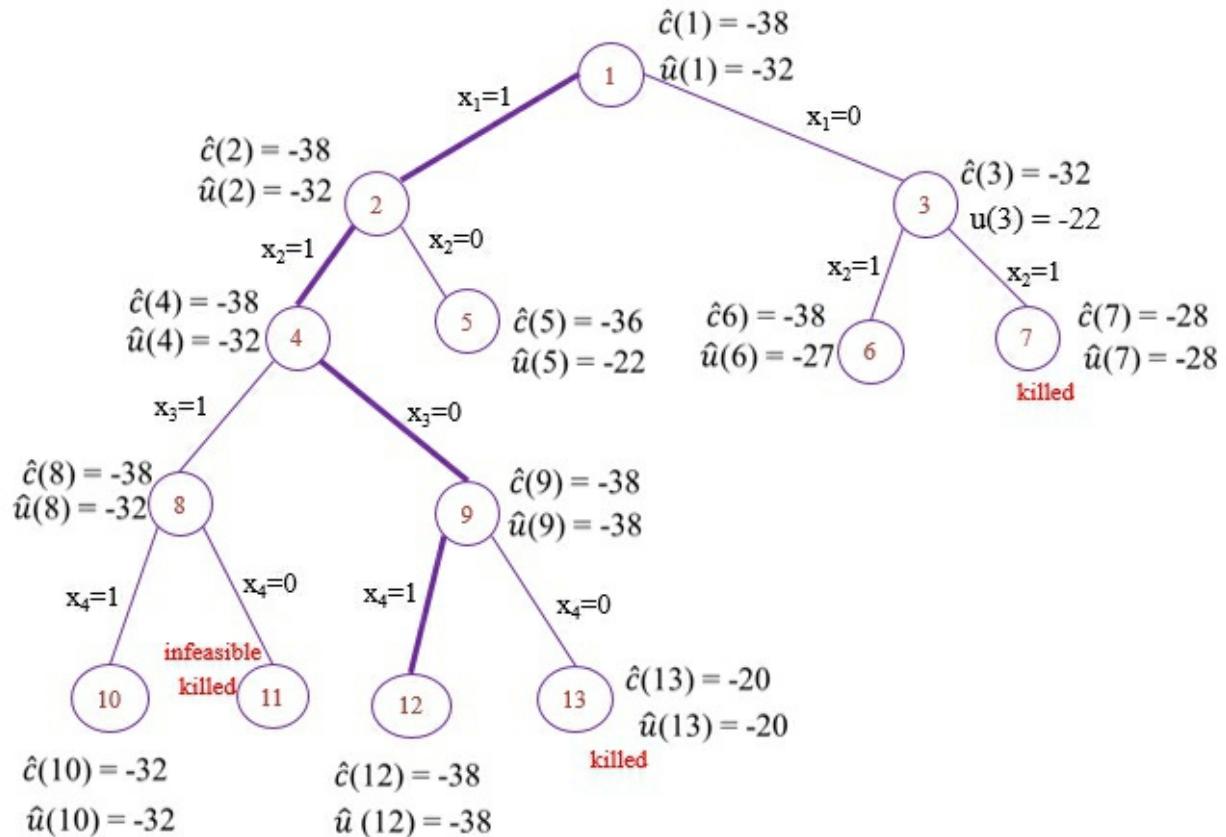


Fig. 7.3.15: FIFOBB Tree

# 8 NP-HARD AND NP-COMPLETE PROBLEMS

## 8.1 BASIC CONCEPTS

Some computational problems are complex and difficult to find efficient algorithms for solving them and it can be solved in future also and have no polynomial time algorithm and some problems are there, that can be solved by a time algorithms, ( $O(n)$ ,  $O(\log n)$ ,  $O(n \log n)$  and  $O(n^2)$ ). Polynomial time algorithms are being tractable (searching of an element from the list  $O(\log n)$ , sorting of elements  $O(n \log n)$  and problems that require non-polynomial time are being intractable (Knapsack problem  $O(2^{n/2})$  and Travelling salesperson problem  $(O(n^2 2^n))$ ).

Any problem that involves the identification of optimal cost (minimum or maximum) is called as optimization problem. The algorithm for optimization problem is called as optimization algorithm.

**Definition of P-Class:** *The problem that can be solved in polynomial time by deterministic algorithm are called P-problems. “P” stands for polynomial. P-class includes only decision problems. Any problem for which answer is either yes or no is called as decision problem. The algorithm for decision problem is called as decision algorithm.*

**Example:** Searching of key element, Sorting of elements, All pairs shortest path.

**Definition of NP-Class:** *The problems which can be solved in polynomial time by non-deterministic algorithms are called as NP-problems. “NP” stands for non-deterministic polynomial time, not for non-polynomial time.*

**Example:** The travelling salesperson problem, Graph coloring problem, Knapsack problem, Hamiltonian circuit problems.

The NP-class problems can be further classified into NP-complete and NP-hard problems.

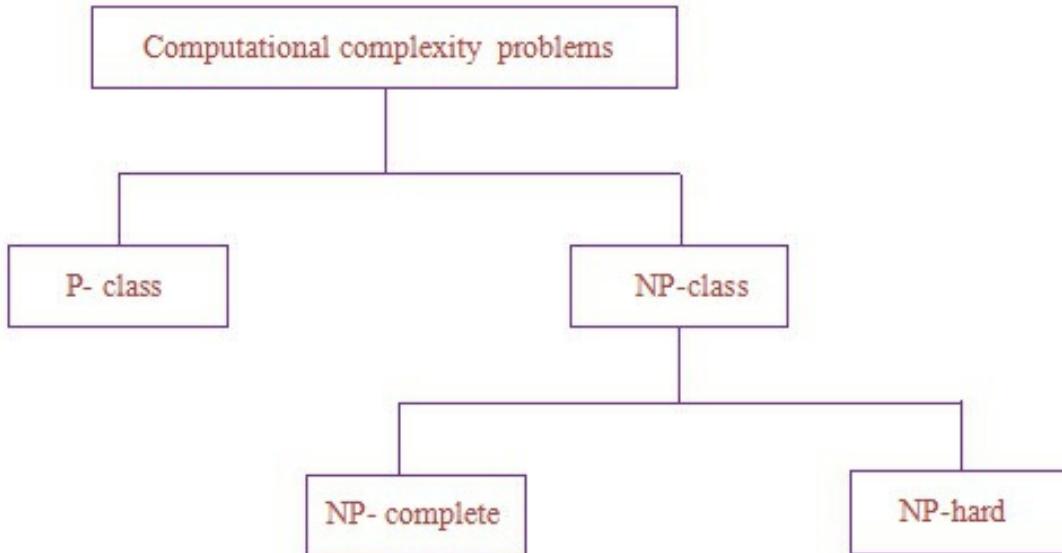


Fig. 8.1.1: Classification of Complexity Problems

It has not been able to develop polynomial algorithms for the problems of second category. Also, it has not been proved yet that the problems in the second category do not have polynomial-time-algorithms. The theory of NP-completeness simply tells that many of the problems which do not have known polynomial-time-algorithms yet are computationally related and hence classified into two groups. They are namely NP-hard and NP-complete.

A problem that is NP-complete has the property that can be solved in polynomial time if all other NP-complete problems can also be solved in the polynomial time, then all NP-complete problems can be solved in the polynomial time. For example, decision problems, search problems, optimization problems. All NP-complete problems are NP-hard, some NP-hard problems are not known to be NP-complete.

## 8.2 NON-DETERMINISTIC ALGORITHMS

Deterministic algorithms are those algorithms in which the result of every operation is uniquely defined. These algorithms agree with the way the programs are executed on a computer.

The problems whose solutions are provided by the non-deterministic polynomial time algorithms fall under the class of NP problems. Those algorithms containing the operations whose results are not defined uniquely, restricting them only to specify their possibility and hence this causes the operation to make a choice to terminate instead of producing the result. Of

course, the process is in the according to the termination condition it selects. And these algorithms are said to be non-deterministic algorithms.

To understand these non-deterministic algorithms, three functions can be used. They are,

- 1) **Choice(A)** : This function is used to select an element from the given set A and this selection is done randomly and assigns the selected value to a variable.
- 2) **Failure()**: this function is used to indicate an unsuccessful completion.
- 3) **Success()**: this function is used to indicate a successful completion.

The failure() and success() function are used only to outline the operating scheme in the algorithm. An non-deterministic algorithm terminates unsuccessfully if there exist no set of choices leading to a success signal. A machine capable of executing a non-deterministic algorithm is called a non-deterministic machine.

A non deterministic algorithm are as follows,

- 1) The non-deterministic “guessing” phase, some completely arbitrary string of characters, is written beginning at some deterministic place in memory. Each time the algorithm is run, the string written may differ. (This string is the certificate, it may be thought of as a guess at a solution for the problem, so this phase may be called the guessing phase, but it could just as well be gibberish).
- 2) The deterministic “verifying” phase. A deterministic(ie, ordinary) subroutine begins execution. In addition to the decision problem’s input, the subroutine may use s or it may ignore s. Eventually it returns a value true or false or it may get into infinite loop or it may never halt.
- 3) The output step, if the verifying phase returned true, the algorithm outputs yes. Otherwise, there is no output.

The number of steps carried out during one execution of a non deterministic algorithm is defined as the sum of steps in two phases, that is number of steps taken to write s (simply the number of characters in s) plus the number of steps executed by the non deterministic second phase.

### **8.2.1 Non-Deterministic Algorithm for Searching**

Considering the problem to search a specified element e in a set S[1:x], x ≥ 1 and give the output, the index (i) of the searching element.

A non-deterministic search algorithm for the problem is as follows,

```
Algorithm NDSearch(S, x, e)
{
    i = Choice(1, x);
    if(S[i] = e)
        then
    {
        output(i);
        Success();
    }
    else
    {
        Output(0);
        Failure();
    }
}
```

Algorithm 8.2.1: Non-deterministic Searching Algorithm

If the set of choice, which in turns leads to complete algorithm successfully exists, then the algorithm ends successfully.

Insetad of this, if there are no set of choices to call Success() function, then the algorithm is leads to unsuccessful termination of the algorithm and hence such algorithms are termed as non-deterministic algorithms.

### 8.2.2 Non-Deterministic Algorithm for Sorting

Let A(i), 1 ≤ i ≤ n be an unsorted set of positive integers. The non-deterministic algorithm NSORT(A, n) sorts the number into non-decreasing

order. An additional array B(1 : n) is used for consequence.

```
Algorithm NSORT(A, n)
//sort n positive integers
integer A(n), B(n), n, i, j;
B := 0           //initializes B to zero
for i := 1 to n do
    j := choice(1:n)
    if B(j) ≠ 0 then failure
    endif
    B(j) := A(i)
    repeat
        for i := 1 to n-1 do //verify order
            if B(i) > B(i+1) then failure
            endif
        repeat
    print(B)
    success
end NSORT
```

### Algorithm 8.2.2: Non-deterministic Sorting

Line 3 initializes B to zero through any value different from all the B to zero through any value different from all the A(i) will do. In the loop of lines 4-8 each A(i) is assigned to a position in B. Line 5 non-deterministically determines this position. Line 5 ascertains that B(j) has not been already used. Lines 9 to 10 verify that B is sorted in non-decreasing order. A successful completion is achieved if the numbers are output in non-decreasing order. Its complexity is O(n). All deterministic sorting algorithms must have a complexity of  $\Omega(n \log n)$ .

This algorithm is based on a two-step non-deterministic process.

- 1) Display the position of each element of a list in a non-deterministic manner.
- 2) Check whether  $M[i] < M[i+1]$  for  $i = 1$  to  $x-1$ , where  $m$  is a auxiliary array.

Both these steps are completed in polynomial time each requires  $x$  and  $x-1$  steps respectively.

### **8.3 DECISION PROBLEM AND OPTIMIZATION PROBLEM**

Any problem for which the answer os 0 or 1 is called as a decision problem. An algorithm defined with the decision problem is referred as the decision algorithm. For example, each of the following are decision problem,

- 1) Given a string  $S_1$  and string  $S_2$ , does  $S_2$  appear as a substring of  $S_1$ ?
- 2) Given two sets  $S_1$  and  $S_2$ , do  $S_1$  and  $S_2$  contain the same set of elements?
- 3) Given a graph  $G$  with integer weights on its edges, and integer  $K$ , does  $G$  give a minimum spanning tree of weight atmost  $K$ ?

Any problem, that involves the identification of an optimal (maximum or minimum) value of a given cost function is known as optimization problem. The last example shown above can be illustrated with the optimization problem. Also optimization problems can be considered as NP-hard problems.

The NP-completeness has been studied in the framework of the decision problems. Most of the problems are not decision problems, but optimization problems. In order to apply the theory of NP-completeness to optimization problems, we must convert them to decision problems.

**Example:** We consider an example (maximum clique) of how an optimization problem can be converted to a decision problem.

A maximum complete subgraph of a graph  $G = (V, E)$  is a clique, which is isomorphic to  $K_n$ , where  $K_n$  is complete graph on  $n$  vertices. The size of the clique is the number of vertices in it. The maximum clique problem in an optimization problem that has that has to determine the size of a largest clique in  $G$ . We make use of decision problems to determine  $G$  has a clique

of size atleast K for some given K.

Let us consider D clique ( $G, k$ ) be the deterministic decision algorithm for the clique decision problem. Let  $n$  be the number of vertices in  $G$ . The size of max clique in  $G$  can be found by making several applications of D clique. The clique is used once for each  $K$ ,  $K = n, n-1, n-2, n-3, \dots$ , until the output from D clique is 1.

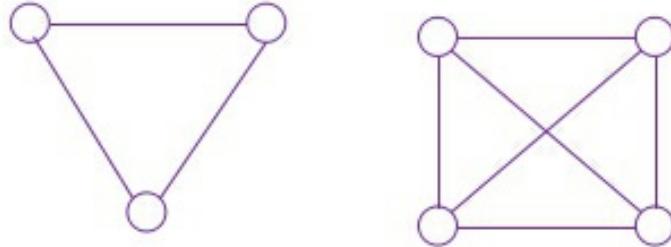


Fig. 8.3.1: Complete Graph

If the time complexity of D clique is  $f(n)$  then the size of most clique can be found in time  $\leq f(n)$ . If in case the size of maximum clique can be determined in time  $g(n)$ , then the decision problem can be solved in time  $g(n)$ . Hence the maximum clique problem is solved in polynomial time if the clique decision problem can be solved in polynomial time.

The input to the maximum clique decision roblem can be provided as the sequence of edges and an integer  $k$ . Each edge is  $E(G)$  is a pair of numbers  $(i, j)$  the size of the input for each edge  $(i, j)$  is  $\log_2 i + \log_2 j + 2$  if a binary representation is assumed. The input size of any instance is,

$$n = \sum_{\substack{(i, j) \in E(G) \\ i < j}} ([\log_2 i] + [\log_2 j] + 2) \log_2 k + 1$$

If  $G$  has only one connected component, then  $n \geq |V|$ . Thus if this decision problem cannot be solved by any algorithm of complexity  $P(|V|)$ . Algorithm DCK is a non-deterministic algorithm for the clique decision problem. Algorithm DCK begins to form set of  $k$  distinct vertices. Then it tests to see weather these vertices forms a complete subgraph.

```

Algorithm DCK(G, n, k)

{
    s = ∅;
    for i = 1 to k do
    {
        t = choice(1, n);
        if t ∈ s then failure();
        s = s ∪ {t}; //combine t to set S
    }
    //Here s contains k distinct vertex indices
    for all pairs(i, j)
        such that i ∉ s, j ∉ s and i ≠ j do if (i, j) is not the edge of G
        then failure();
        success();
}

```

Algorithm 8.3.1: Non-deterministic Clique Pseudocode

If  $G$  is given by its adjacency matrix and  $|V| = n$ , the input length  $m$  is  $n^2 + \log_2 k + n + 2$ . The for loop of lines 4 to 9 can be easily implemented to run in nondeterministic time  $O(n)$ . The time for the loop 11 and 12 is  $O(k^2)$ . Hence, the overall nondeterministic time is  $O(n + k^2) = O(n^2) = O(m)$ . There is no known polynomial time deterministic algorithm for this problem.

## 8.4 SATISFIABILITY

We formulate satisfiability problem to determine whether a formula is true for some assignment of truth values for the variables. The Boolean formula  $\emptyset$  is composed of,

- 1) Boolean variables  $x_1, x_2, \dots$

- 2) Boolean connectives or any Boolean functions with one or two inputs and one output such as  $\wedge$  (AND),  $\vee$  (OR),  $\sim$  (NOT),  $\rightarrow$  (implication),  $\leftrightarrow$  (if and only if and)
- 3) Paranthesis

A truth assignment for a Boolean formula  $\emptyset$  is a set of values for variables of  $\emptyset$  and a satisfying assignment is a truth assignment that causes it to evaluate to 1. A formula with satisfying assignment is a satisfiable formula.

## 8.5 NP-HARD AND NP-COMPLETE CLASSES

An algorithm is said to be “polynomially bounded” if its worst case complexity is bounded by a polynomial function of the input size (i.e., if there is a polynomial P such that for each input size ‘n’, then the algorithm terminates atmost P(n) steps).

“P” is the class of decision problems that are polynomially bounded. “P” defined only for decision problems.

“NP” is the class of decision problems for which there is polynomially bounded non-deterministic algorithm (the name “NP” comes from “Non-deterministic Polynomial Bounded”).

**NP-Complete Problems:** NP-complete is the term used to describe the decision problems that are harder ones. NP in the sence that if there were polynomially bounded algorithm for an NP-complete problem, then there would be polynomially bounded algorithm for each problem in NP.

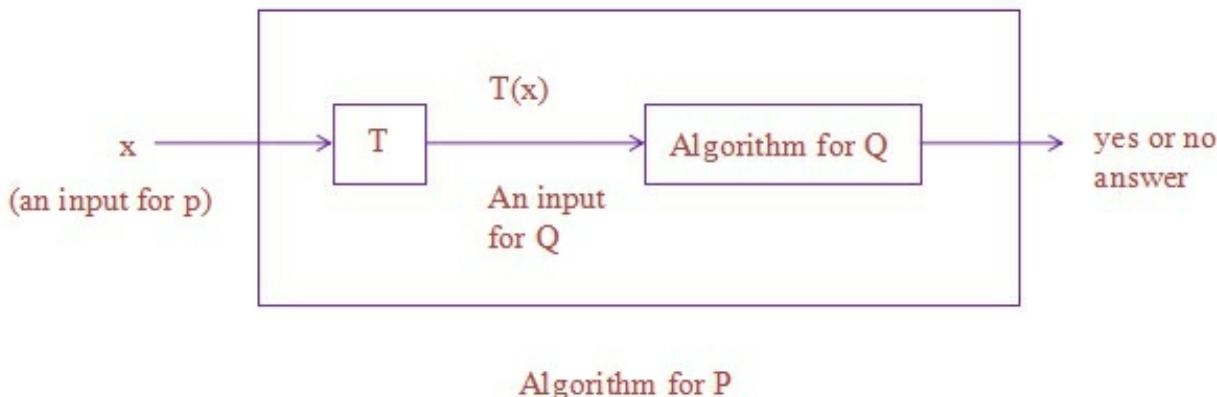


Fig. 8.5.1: Reduction of a Problem P to a problem Q : Problem to Q's answer

for  $T(x)$  must be the same as P's answer for  $x$

The formal identification of "NP-complete" uses reduction or transformations of one problem to another. Suppose we want to solve a problem T and we already have an algorithm for another problem Q.

Suppose we have a function T that takes an input x for P and produce  $T(x)$ , an input for Q such that the correct answer for P on x is yes if and only if the correct answer for Q on  $T(x)$  is yes. Then their composition T and the algorithm Q, we have an algorithm for P.

**Example:** A simple reduction.

Let the problem P be a given sequence of Boolean values, does atleast one of them have the value true? (In other words, this is a decision problem version of computing the n-way Boolean or when the string has n values.

Let Q be given a sequence of integers is the maximum of the integers positive? Let the transformation T be defined by,

where,  $y_i = 1$  if  $x_i = \text{true}$  and  $y_i = 0$  if  $x_i = \text{false}$ .

Clearly an algorithm to solve Q, when applied to  $y_1, y_2, \dots, y_n$  solves for  $x_1, x_2, \dots, x_n$ .

Let  $L_1$  and  $L_2$  be problems. Problem L reduced to  $L_2$  (Also written  $L_1 \leq L_2$ ) if and only if there is a way to solve  $L_1$  by a deterministic polynomial time algorithm using deterministic polynomial algorithm that solves  $L_2$  in polynomial time. This implies that if we have a polynomial time algorithm for  $L_2$ , then we can solve  $L_1$  in polynomial time.

A problem L is NP-hard and only its satisfiability reduce to L (satisfiability  $\alpha L$ ). We say that a problem X is NP-hard if there an NP-

complete problem Y that can be polynomially turning reduced to it  $y \leq^p t_X$ . Definition of polynomial reductions and polynomial time algorithm for X would translate into one for Y. Since Y is NP-complete, this would imply that  $P = NP$ , contrary to the generally accepted belief. Therefore no NP-hard problem can be solved in polynomial time in the worst case under the assumptions that  $P \neq NP$ .

There are several reasons to study NP-hardness rather than NP-completeness. In particular NP-hard problems do not have to be decision problems. Consider for an example, the graph coloring problems.

It is obvious that any efficient algorithm to find an optimal graph coloring (COLC) or to determine the chromatic number of a graph (COLO) can be used to determine efficiently whether a graph can be painted with three colors (3 COL).

In symbols,  $3\text{COL} \leq^p t\text{COLO} \leq^p t\text{COLC}$ . It follows from the NP-completeness of 3 COL that both COLO and COLC are NP-hard even though they are not NP-complete because they are not decision problems.

The notation of NP-hardness is interesting also for decision problems. There are decision problems that are known to be NP-hard but believed not to be in NP and thus not NP-complete.

Consider for example, the problem COLE for exact coloring, given a graph G and an integer k, can G be painted with k colors but no less?

Again it is obvious that  $3\text{COL} \leq^p t\text{COLE}$  because a graph is 3-colorable if and only if its chromatic number is either 0, 1, 2, or 3. From the NP-completeness of 3COL we conclude that the exact graph coloring problem is NP-hard. However, this decision problem does not seem to be in NP. Although any valid coloring of G with K colors can be used as a succinct certificate that G can be painted with K colors. It is hard to imagine what a succinct certificate that G cannot be painted with fewer colors would look like, and there are strong theoretical reasons to believe that such certificates do not exist in general.

### 8.5.1 The Complexities of NP-Hard and NP-Complete

Complexity of an algorithm is measured by using the input length as the parameter. An algorithm X is of polynomial complexity if there exists a polynomial time P( ) such that the computing time of A is O(P(n)) for every input of size n.

The time which is required by a non-deterministic algorithm for performing on any given input is the minimum number of steps to reach a successful completion if in case there exists a sequence of choices leading to

such completion. In case successful completion is not possible then the time required is  $O(1)$ . A non-deterministic algorithm is having the complexity of  $O(f(n))$ . If for all inputs of size  $n$ ,  $n^3n_0$ , that results in a successful completion. The time required is atmost  $Cf(n)$  for some constants  $C$  and  $n_0$ .

$P$  is the set of allg decision problems solvable by deterministic algorithms in polynomial time.  $NP$  is the set of all decision problems solvable by non-deterministic algorithms in polynomial time. Since, deterministic algorithms are a special case of non-deterministic ones, and we can conclude  $P$  is less than or equal to  $NP$  but one question arrises like weather  $P = NP$  or  $P \neq NP$  because of the unsolved problems in computer science.

Generally, there are some  $NP$ -hard problems that are not  $NP$ -complete. Only a decision problem can be  $NP$ -complete. However, an optimization problem may be  $NP$ -hard further if  $L$  is decision problem and  $L_2$  an optimization problem, it is quite possible that  $L_1 \alpha L_2$  one can trivially show that clique decision problem reduces to the clique optimization problem. Optimization problems cannot be  $NP$ -complete. Where as decision problems can be  $NP$ -complete. There also exists  $NP$ -hard decision problems that are not  $NP$ -complete.

$NP$ -complete problems arise in diverse areas like Boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrivial, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, arthematic and language theory. Program optimization and many more clique program already we have solved coming under  $NP$ -complete problems group.

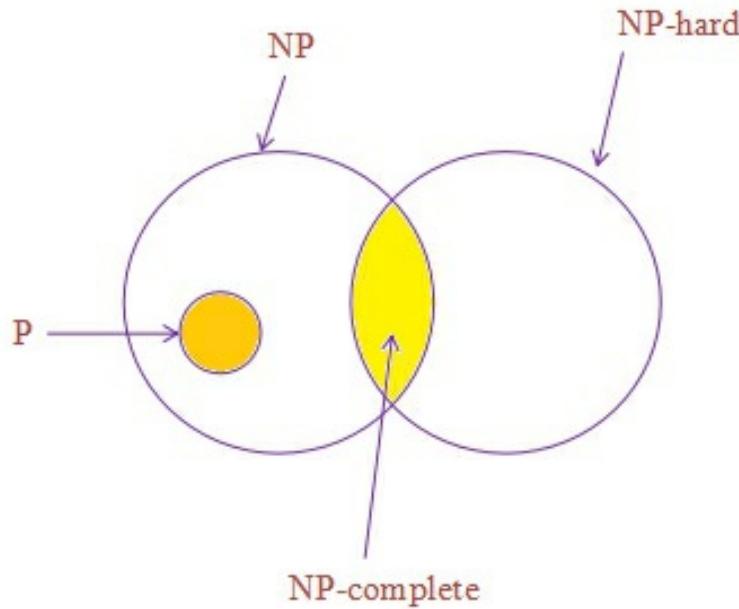


Fig. 8.5.2: Relationship between P, NP, NP-hard and NP-complete

### 8.5.2 Halting P

The halting problem is an example of an NP-hard decision problem that is not in NP-complete. The halting problem is to determine an arbitrary deterministic algorithm A and an input I whether algorithm A with input I ever terminates (or enters an infinite loop). This problem is undecidable i.e., there exists no algorithm (of any complexity) to solve this problem. So, it clearly cannot be in NP. To show satisfiability a halting problem simply construct an algorithm A whose input is a propositional formula X. If X has 'n' variables then A tries out all  $2^n$  possible truth values and verifies if X is satisfiable. If on input X if X is satisfiable. If we had a polynomial time algorithm for the halting problem then we should solve the satisfiability problem in polynomial time using A and X as input to the algorithm for the halting problem. Hence, the halting problem is NP-hard problem which is not in NP.

## 8.6 COOK'S THEOREM

Cook's theorem states that the satisfiability is in P if P = NP. We have already known that the satisfiability is in NP, then the satisfiability is in P.

Then it remains to be shown that if satisfiability is in P then P = NP. For proving this statement, we show how to obtain any polynomial time non-deterministic decision algorithm A and input I, a formula Q(A, I) such that Q

is satisfiable if A has successful termination with input I. If the length of I is ‘n’ and the time complexity of A is  $p(n)$  for some polynomial  $P(\cdot)$  then the length of Q will be  $O(P^3(n) \log n) = O(P^4(n))$ . The time needed for constructing Q will also be same.

A deterministic algorithm Z to determine the outcome of A on any input I may be obtained as shown below.

If (Z) computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether or not Q is satisfiable. If  $O(q(m))$  is the time needed to determine if a formula of length ‘m’ is satisfiable then the complexity of Z is  $O(p^3(n) \log n + q(P^3(n) \log n))$ . If satisfiability is in P then  $q(m)$  is polynomial function of m and the complexity of Z becomes  $O(r(n))$  for some polynomial  $r(\cdot)$ . Hence, if satisfiability is in P then for every non deterministic algorithm A in NP we can obtain a deterministic Z in P.