

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/326477402>

# DESIGN AND ANALYSIS OF ALGORITHMS- (1st Edition)- S. R. Jena, S. Patro

Book · July 2018

---

CITATIONS

4

READS

33,597

2 authors:



Soumya Jena  
Vel Tech - Technical University

54 PUBLICATIONS 63 CITATIONS

[SEE PROFILE](#)



Satyabrata Patro  
Raghu Engineering College

7 PUBLICATIONS 7 CITATIONS

[SEE PROFILE](#)

# DESIGN AND ANALYSIS OF ALGORITHMS

S. R. Jena | S. Patro

## ABOUT THE BOOK

This book is intended for the students of B.Tech & BE (CSE/IT), M.Tech & ME (CSE/IT), MCA, M.Sc (CS/IT).

This book includes:

- Fundamental Concepts on Algorithms
- Framework for Algorithm Analysis
- Asymptotic Notations
- Sorting Algorithms
- Recurrences
- Divide and Conquer Approach
- Dynamic Programming Approach
- Greedy Algorithms
- Graph Algorithms
- Backtracking
- String Matching
- NP-Completeness
- NP-Complete Problems (with proofs)
- Approximation Algorithms

## Salient Features:

- ✓ Complete and Focused Coverage of Syllabus in Very Simple Language.
- ✓ Expanded Coverage on Sorting Algorithms in Chapter-4.
- ✓ Expanded Coverage on NP-Completeness and Approximation Algorithms in Chapter-17.
- ✓ More than 250 Solved Examples.
- ✓ A Quick Reference Table for Time Complexity of Algorithms in Appendix-II.
- ✓ Last 3 Years Solved University Question Papers in Appendix-III.
- ✓ Chapter-wise Short Type Questions with Answers in Appendix-IV.
- ✓ Solved GATE Question Papers in Appendix-V.

## ABOUT THE AUTHORS

**Soumya Ranjan Jena** is working as an Assistant Professor in the department of Computer Science and Engineering at K L E F (Koneru Lakshmaiah Education Foundation), Deemed to be University, Guntur, Andhra Pradesh, India. He has awarded M.Tech in IT, B.Tech in CSE and CCNA. He has got immense experience in teaching to graduate as well as post-graduate students. He is the author of Theory of Computation and Application book published by University Science Press, Laxmi Publications, New Delhi.

**Satyabrata Patro** is working as an Assistant Professor in the department of Computer Science and Engineering at K L E F (Koneru Lakshmaiah Education Foundation), Deemed to be University, Guntur, Andhra Pradesh, India. He has obtained M.Tech in CSE, MCA, and M.Sc (Mathematics). He has 12 years of teaching experience at UG and PG level.

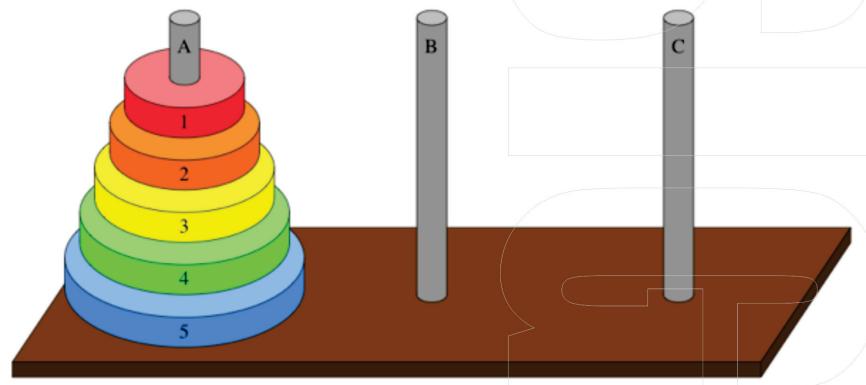
DESIGN AND ANALYSIS OF ALGORITHMS

S. R. Jena | S. Patro

 UNIVERSITY SCIENCE PRESS  
(An Imprint of Laxmi Publications Pvt. Ltd.)  
An ISO 9001:2015 Company



# DESIGN AND ANALYSIS OF ALGORITHMS



S. R. Jena | S. Patro

# **DESIGN AND ANALYSIS OF ALGORITHMS**

# **DESIGN AND ANALYSIS OF ALGORITHMS**

*By*

**S.R. JENA**

*Assistant Professor*

*Computer Science and Engineering  
K L University, Guntur,  
Andhra Pradesh, India*

**S. PATRO**

*Assistant Professor*

*Computer Science and Engineering  
K L University, Guntur,  
Andhra Pradesh, India*

## **UNIVERSITY SCIENCE PRESS**

(An Imprint of Laxmi Publications Pvt. Ltd.)

BANGALORE • CHENNAI • COCHIN • GUWAHATI • HYDERABAD  
JALANDHAR • KOLKATA • LUCKNOW • MUMBAI • RANCHI  
NEW DELHI • BOSTON, USA



# **PREFACE**

In this vast changing complex world dependence of IT on Algorithms has been raised up to maximum level. To meet the need we have the endeavour to design this text with the objectives to facilitate the learners of Computer Science and Engineering, MCA and IT.

Programming is a very complex task and there are a number of aspects of programming that make it so complex. The first is that most programming projects are very large, requiring the coordinated efforts of many people. The next is that many programming projects involve storing and accessing large quantities of data efficiently. The last is that many programming projects involve solving complex computational problems, for which simplistic or brute force solutions may not be efficient enough. The complex problems may involve numerical data, but often they involve discrete data. This is where the topic of algorithm design and analysis is important.

Although the algorithms discussed in this book will often represent only a tiny fraction of the code that is generated in a large software system, this small fraction may be very important for the success of overall project. An unfortunately common approach to this problem is to first design an inefficient algorithm and data structure to solve the problem, and then take this poor design and attempt to fine-tune its performance. The problem is that if the underlying design is bad, then often no amount of fine-tuning is going to make a substantial difference.

The focus of this book is on how to design good algorithms, and how to analyze their efficiency. This is among the most basic aspects of good programming.

Contribution of many people is a must for successful completion of any attempt. Therefore we thank Er. Swapna Kumar Naik, Programmer, Dept of CSA, Utkal University, Odisha for his unending support all the way. Mr. Abhaya Kumar Jena, PGT English rendered his praiseworthy help in the progress; as such we are grateful to him. Last but not least, we thank Laxmi Publications Pvt Ltd, New Delhi for their interest in publication of the book on time.

—Authors



# CONTENTS

<i>Chapters</i>	<i>Page No.</i>
Preface .....	(v)
<b>0. READER'S MANUAL.....</b>	<b>1–6</b>
0.1 Objectives of Learning Algorithms .....	2
0.2 Why to Choose the Book .....	3
0.3 Organization of the Book .....	3
0.4 List of Important Notations .....	5
<b>1. OVERVIEW .....</b>	<b>7–11</b>
1.1 Introduction .....	8
1.2 Beginning Designing Algorithm .....	8
<i>Chapter Notes</i> .....	11
<i>Exercises</i> .....	11
<b>2. FRAMEWORK OF ALGORITHM ANALYSIS .....</b>	<b>12–21</b>
2.1 Introduction.....	13
2.2 The Basics .....	13
2.3 Definitions of Preliminary Terms .....	14
2.4 Phases of Algorithm Construction .....	15
2.5 Mathematical Model of a Computer: RAM Model .....	17
2.6 EM Model .....	19
2.7 PRAM Model .....	20
<i>Chapter Notes</i> .....	20
<i>Exercises</i> .....	21

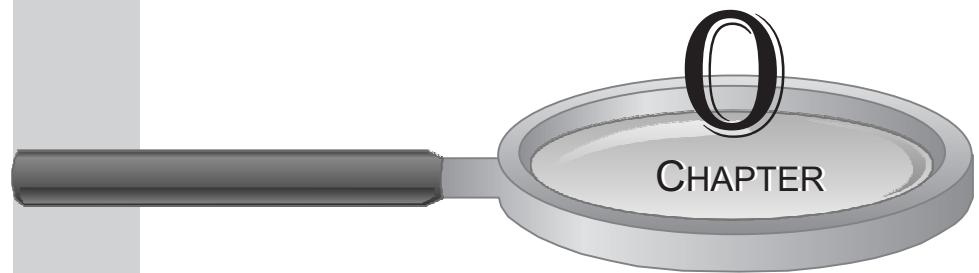
<i>Chapters</i>	<i>Page No.</i>
<b>3. ASYMPTOTIC NOTATION .....</b>	<b>22–39</b>
3.1    Introduction.....	23
3.2    Types of Notations .....	24
3.3    Relational Properties of Asymptotic Notations .....	26
3.4    Some Standard Notations and Common Functions .....	26
<i>Chapter Notes</i> .....	39
<i>Exercises</i> .....	39
<b>4. SORTING ALGORITHMS .....</b>	<b>40–58</b>
4.1    Introduction.....	41
4.2    Insertion Sort .....	42
4.3    Bubble Sort .....	49
4.4    Selection Sort .....	51
4.5    Counting Sort .....	52
4.6    Radix Sort.....	54
4.7    Bucket Sort .....	56
<i>Chapter Notes</i> .....	57
<i>Exercises</i> .....	58
<b>5. DIVIDE AND CONQUER APPROACH .....</b>	<b>59–71</b>
5.1    Introduction.....	60
5.2    Merge Sort .....	60
5.3    An $n \log n$ Lower Bound for Sorting .....	63
5.4    Pros and Cons of Merge Sort.....	64
5.5    Strassen's Matrix Multiplication.....	67
<i>Chapter Notes</i> .....	71
<i>Exercises</i> .....	71
<b>6. RECURRENCES .....</b>	<b>72–107</b>
6.1    Introduction.....	73
6.2    Substitution Method .....	73
6.3    Recursion-Tree Method.....	83
6.4    Master Theorem .....	100
<i>Chapter Notes</i> .....	106
<i>Exercises</i> .....	107
<b>7. BINARY SEARCH .....</b>	<b>108–113</b>
7.1    Introduction.....	109
7.2    Algorithm Steps .....	110

<i>Chapters</i>	<i>Page No.</i>
7.3 Pros and Cons of Binary Search.....	111
<i>Chapter Notes</i> .....	113
<i>Exercises</i> .....	113
<b>8.   QUICKSORT .....</b>	<b>114–125</b>
8.1 Introduction.....	115
8.2 Quicksort Algorithm.....	116
8.3 Pros and Cons of Quicksort.....	117
8.4 Random Quicksort .....	120
<i>Chapter Notes</i> .....	124
<i>Exercises</i> .....	124
<b>9.   ORDER STATISTICS .....</b>	<b>126–134</b>
9.1 Introduction.....	127
9.2 The Selection Problem .....	128
9.3 Algorithm for Finding Maximum and Minimum .....	128
9.4 Algorithm for Finding <i>i</i> th Smallest Element .....	129
<i>Chapter Notes</i> .....	134
<i>Exercises</i> .....	134
<b>10.   HEAPSORT .....</b>	<b>135–158</b>
10.1 Introduction.....	136
10.2 Heaps .....	137
10.3 Types of Heaps .....	139
10.4 Algorithm for Max-Heapify .....	140
10.5 Algorithm for Building Max-Heap .....	144
10.6 Heapsort-Algorithms.....	149
10.7 Pros and Cons of Heapsort .....	157
<i>Chapter Notes</i> .....	158
<i>Exercises</i> .....	158
<b>11.   PRIORITY QUEUES .....</b>	<b>159–169</b>
11.1 Introduction.....	160
11.2 Types of Priority Queues .....	160
11.3 Algorithm for Max-Priority Queue Operations .....	161
11.4 Different Applications of Priority Queue .....	162
<i>Chapter Notes</i> .....	168
<i>Exercises</i> .....	169

<i>Chapters</i>	<i>Page No.</i>
<b>12. DYNAMIC PROGRAMMING .....</b>	<b>170–189</b>
12.1 Introduction .....	171
12.2 Longest Common Subsequence (LCS) .....	172
12.3 Chain Matrix Multiplication .....	177
<i>Chapter Notes</i> .....	188
<i>Exercises</i> .....	189
<b>13. GREEDY ALGORITHMS .....</b>	<b>190–206</b>
13.1 Introduction .....	191
13.2 Activity Selection Problem/Activity Scheduling .....	192
13.3 Elements of Greedy Strategy .....	194
13.4 Knapsack Problem (Rucksack Problem) .....	195
13.5 Huffman Coding .....	197
<i>Chapter Notes</i> .....	205
<i>Exercises</i> .....	206
<b>14. ELEMENTARY GRAPH ALGORITHMS .....</b>	<b>207–277</b>
14.1 Introduction .....	208
14.2 Representation of Graphs and Digraphs .....	210
14.3 Graph Traversal Techniques .....	212
14.4 Breadth-First Search (BFS) .....	213
14.5 Depth-First Search (DFS) .....	226
14.6 Applications of DFS .....	240
14.7 Minimum Spanning Tree .....	245
14.8 Kruskal's Algorithm .....	248
14.9 Prim's Algorithm .....	252
14.10 Single Source Shortest Paths .....	260
14.11 Dijkstra's Algorithm .....	261
14.12 The Bellman-Ford Algorithm .....	264
14.13 All-Pairs Shortest Paths .....	268
14.14 Floyd-Warshall Algorithm .....	269
<i>Chapter Notes</i> .....	276
<i>Exercises</i> .....	276
<b>15. BACKTRACKING .....</b>	<b>278–284</b>
15.1 Introduction .....	279
15.2 N-Queens Problem .....	279
15.3 Subset Sum Problem .....	282

<i>Chapters</i>	<i>Page No.</i>
15.4 Graph Coloring Problem .....	283
<i>Chapter Notes</i> .....	284
<i>Exercises</i> .....	284
<b>16. STRING MATCHING .....</b>	<b>285–298</b>
16.1 Introduction .....	286
16.2 Brute-Force String Matching Algorithm .....	288
16.3 Rabin-Karp String Matching Algorithm .....	292
<i>Chapter Notes</i> .....	298
<i>Exercises</i> .....	298
<b>17. NP-COMPLETENESS AND APPROXIMATION ALGORITHMS .....</b>	<b>299–333</b>
17.1 Introduction .....	300
17.2 Matching Problem .....	302
17.3 Reductions and its Applications .....	303
17.4 The Classes : P and NP .....	306
17.5 NP-hard and NP-complete .....	309
17.6 Satisfiability (SAT) .....	310
17.7 Cook's Theorem .....	311
17.8 Common NP-complete Problems with Proofs .....	312
17.9 Other Useful NP-complete Problems .....	321
17.10 Other Important Complexity Classes .....	322
17.11 Approximation Algorithms .....	323
17.12 Different Approximation Schemes .....	331
17.13 History of NP-completeness .....	332
<i>Chapter Notes</i> .....	332
<i>Exercises</i> .....	333
<b>APPENDIX-I : Mathematical Background.....</b>	<b>334–338</b>
<b>APPENDIX-II : Table for Time Complexity of Algorithms .....</b>	<b>339–341</b>
<b>APPENDIX-III : Solved University Question Papers .....</b>	<b>342–378</b>
<b>APPENDIX-IV : Chapterwise Short Type Questions with Answers .....</b>	<b>379–410</b>
<b>APPENDIX-V : Brain Teasers from GATE (Solved) .....</b>	<b>411–426</b>
<b>APPENDIX-VI : Model Question Papers for Practice .....</b>	<b>427–432</b>
<b>FURTHER READINGS .....</b>	<b>433</b>
<b>INDEX .....</b>	<b>433–437</b>

# **READER'S MANUAL**



# Chapter 0 READER'S MANUAL

## INSIDE THIS CHAPTER

- 0.1 Objectives of Learning Algorithms
- 0.2 Why to Choose the Book
- 0.3 Organization of the Book
- 0.4 List of Important Notations

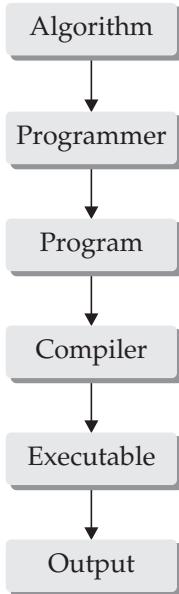
## 0.1 OBJECTIVES OF LEARNING ALGORITHMS

Life is meaningful; without objective life is vague. Similarly while learning any subject we should have certain objectives. At this moment our goal is to learn *Algorithms*. Basically algorithm is the step by step procedure written in pseudo code for solving any given problem. Then *what is the need to study this subject? What we gain from this subject? Does this subject helpful to us in our day-to-day life? If yes, then in which way?* etc.

Here are some possible answers which might be murmuring in your mind at this moment !

- Algorithms are *fundamental* to computer science.
- It is a *compulsory subject* in our graduate and postgraduate level courses in Computer Science.
- The real world performance of any software or hardware system depends on the algorithm chosen and the *suitability* and *efficiency* of various layers of implementation. Therefore, *efficient algorithms* is a pervasive theme through this area.
- Efficient algorithms lead to *efficient programs*.
- Efficient programs *sell better*.
- Efficient programs make *better use of hardware*.
- Programmers who write efficient programs are *more marketable* than those who don't !

An important part of this subject is *implementation*. Here we find how the implementation of an algorithm takes place. In Fig. 0.1 we have sketched a flowchart which tells us regarding this matter.



**Fig. 0.1.** Implementation of algorithm.

## 0.2 WHY TO CHOOSE THE BOOK

---

Now-a-days in the market there is abundant number of books available on this subject. Then question is “*Why to choose this book?*” *Does it have the uniqueness?*”

When the authors started writing this book they have aim towards their enthusiastic readers who are the students in the field of Computer Science and Engineering and Information Technology. To fulfill their basic needs, authors have segmented this book into small chapters. Some of the key features of this book include:

- Each chapter contains *minimum theory* which fulfills the basic need.
- More number of *solved examples* at each section which helps a student for the preparation of semester examinations as well as different competitive examinations like *GATE, NET* etc.
- *Reduce study time* for this subject.

## 0.3 ORGANIZATION OF THE BOOK

---

The book contains a number of major sections excluding this chapter.

**Chapter 1 and 2** represent the preliminary concepts on designing an algorithm, phases of construction and different mathematical models through which we can analyze algorithms.

**Chapter 3** precisely defines several asymptotic notations, which we use for bounding algorithm running times from above and/or below.

**Chapter 4** highlights different sorting algorithms like Insertion Sort, Bubble Sort, Selection Sort, Counting Sort, Radix Sort and Bucket Sort and their performance analysis.

**Chapter 5** presents the techniques and concepts behind divide and conquer approach, including Merge Sort and Strassen's algorithm for matrix multiplication.

**Chapter 6** represents different methods for solving recurrences like Substitution method, Recursion-tree method, and Master theorem.

**Chapter 7** gives the idea about Binary search.

**Chapter 8** discusses about Quicksort and its expected running time.

In **Chapter 9** we show that we can find the  $i^{\text{th}}$  smallest element in  $O(n)$  time, even when the elements are arbitrary real numbers. We present randomized algorithm that run in  $O(n^2)$  worst case, but its expected running time is  $O(n)$ .

Heap Sort presented in **Chapter 10** sorts  $n$  numbers in  $O(n \log n)$  time. It uses an important data structure called heap, with which we can also implement a priority queue which has discussed in **Chapter 11**.

**Chapter 12** describes dynamic programming paradigm. We also see its different applications like LCS finding and chain matrix multiplication in this chapter.

**Chapter 13** focuses on Greedy algorithms including activity selection problem, Knapsack problem (0-1 Knapsack and fractional Knapsack) and Huffman problem.

The next major **Chapter 14** focuses on graph algorithms. This will cover breadth-first and depth-first search and their application in various problems related to connectivity of graphs.

**Chapter 15** introduces the recursive algorithm strategy called backtracking and its various applications like N-Queens problem, subset sum problem and graph coloring problem.

**Chapter 16** studies the problem of finding all occurrences of a given pattern string in a given text string. After examining the brute-force approach, we will see an efficient string matching called Rabin-Karp algorithm.

**Chapter 17** introduces to different complexity classes like **P**, **NP**, **NP-hard**, and **NP-complete**. We also discuss different **NP**-complete problems such as clique problem, independent set problem, vertex cover problem,  $\leq 3$  vertex cover problem, and exact cover problem with their proofs. Finally we demonstrate different approximation algorithms for travelling salesman problem, vertex cover problem and 0-1 knapsack problem.

**Appendix-I** includes the preliminary mathematics that are needed to understand this subject.

**Appendix-II** gives a suitable table for time complexity of different algorithms that are discussed throughout the book for quick reference.

**Appendix-III** discusses the solutions of last 3 years university question papers.

**Appendix-IV** contains short type questions with answers.

**Appendix-V** provides solved questions with answers from GATE examinations.

Last but not the list, **Appendix-VI** gives some important and selected model question papers for practice.

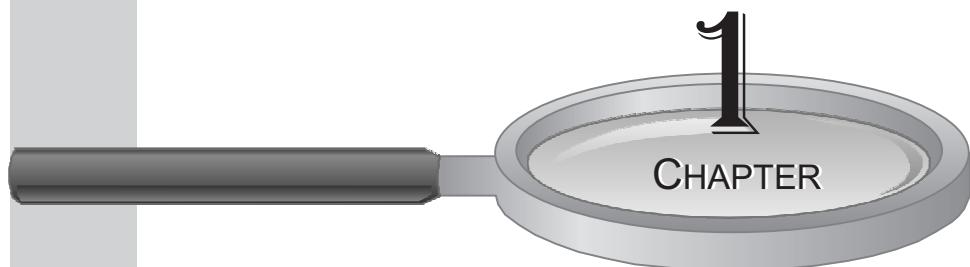
## 0.4 LIST OF IMPORTANT NOTATIONS

---

Symbol	Meaning
$\theta$	Theta notation
$O$	Big Oh notation
$\Omega$	Big Omega notation
$o$	Little oh notation
$\omega$	Little omega notation
$\exists$	There exists
$\forall$	For all
$\lceil \rceil$	Ceil
$\lfloor \rfloor$	Floor
$\sum_{i=0}^n a_i$	Summation of $a_0, a_1, \dots, a_n$ terms = $a_0 + a_1 + \dots + a_n$
$x!$	Factorial of $x$
$ x $	+ve values of $x$
$f(n) \in \theta(n)$	The function $f(n)$ is belongs to $\theta(n)$
$\theta(g(n)) \subset O(g(n))$	$\theta(g(n))$ is subset of $O(g(n))$
$a \neq b$	$a$ not equals to $b$
$A \cup B$	The union of sets A and B
$A \cap B$	The intersection of sets A and B
$[s_p, t_p)$	Half open interval
$d_T(x)$	Length of the codeword in expected encoding length
$G^T$	Transpose of graph G
$W : E \rightarrow \mathbb{R}^+$	Positive weighted graph
$v. \pi$	Predecessor of vertex $v$
$\pi[v]$	Predecessor of vertex $v$
$W(u, v)$	Weight of vertex $u$ to vertex $v$
$d_{ij}^{(k)}$	Shortest path between vertex $i$ to vertex $j$ going through $k$
$\leq_{\text{poly}}$	Polynomial reduction

$\leq_{\log}$	Log space reduction
P and NP	Classes
NP-HARD	Classes
NP-COMPLETE	Classes
T	Truth value
F	False value
$\wedge$	Logical AND
$\vee$	Logical OR
$\wedge_{i=1}^k c_i$	Representation of CNF; where $c_i$ is a clause
$\vee_{i=1}^k c_i$	Representation of DNF; where $c_i$ is a clause
$\overline{x_i}$	Complement of literal $x_i$
$\bigcup_{i=1}^n A_i$	The union of the sets $A_1, A_2, \dots, A_n$
$d[i, j]$	Distance between vertex $i$ to vertex $j$
$\Leftrightarrow$	If and only if
$\emptyset$	Null set

# OVERVIEW



## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- Our approach towards algorithm study
- Design fast algorithm
- Euclid's algorithm for GCD problem

**INSIDE THIS CHAPTER**

- 1.1 Introduction
- 1.2 Beginning Designing Algorithm

---

**1.1 INTRODUCTION**

---

Let us start with a basic question: given a certain problem how will you solve this through computer? For many problems, it is relatively easy to design algorithms that will some how solve them. However the requisite cleverness is needed to design algorithms which are fast *i.e.*, algorithms that give answers very quickly. This will be the major challenge in this subject.

Our ultimate objective is to design algorithms which are *fast*. As we may realize design anything such as computers, cars, cloths is an art. In some sense we have to be *creative* but it canot be taught, in other sense there are very well defined *design techniques* which are used for these purposes. Our goal is to study these techniques and to apply later in our life. Therefore, we need some prerequisites to study these techniques. These are given as follows.

- Familiar with *basic programming languages* (such as C, C++, Basic etc)
- Knowledge on *data structure*
- Some amount of knowledge on *discrete mathematics*

At this point one can think how to approach towards this subject. So never the less our approach should be analytical. In this concern, we build a mathematical model of a computer known as *random access machine* (RAM), study properties of algorithms on this model. The whole point will be *reason* about algorithms as well as *prove* facts about time taken.

**Note:** RAM his discussed in Chapter-2

---

**1.2 BEGINNING DESIGNING ALGORITHM**

---

In this section, we discuss one of the simplest mathematical problems called *greatest common divisor* (GCD) and derive two different algorithms to solve this problem and finally compare

them. Basically the GCD of two or more integers, when at least one of them is not zero, is the largest positive integer that divides the numbers without a remainder. For example, the GCD of 6 and 9 is 3.

**Problem Statement :** Given two numbers  $m$  and  $n$ ; find their GCD.

Input : Two integers i.e.,  $m$  and  $n$

Output : Find largest integer that divides both

**Algorithm 1 (School Level Method):** This algorithm is simple which taught at school level. The steps of the algorithm are given as follows:

1. Factorize  $m$  : Find primes  $m_1, m_2 \dots m_k$  such that  $m = m_1 \times m_2 \times \dots \times m_k$
2. Factorize  $n$  : Write  $n = n_1 \times n_2 \times \dots \times n_j$
3. Identify common factors and then multiply and return the result

**Algorithm 2 (Euclid's Algorithm):** Euclid's algorithm also known as Euclidean algorithm is an efficient method for computing the GCD of two numbers. It is named after the ancient Greek mathematician Euclid. It is an example of an *algorithm*, a step-by-step procedure for performing a calculation according to well-defined steps, and is one of the oldest numerical algorithms in common use.

```
Euclid (m, n) // Assuming n > m
{
    while m does not divide n
    {
        r = n mod m
        n = m
        m = r
    }
    return m
}
```

Let's now check whether the two algorithms work perfectly or not.

Checking Algorithm 1:

Suppose  $m = 36, n = 48$

Factors of  $m$  are 2, 2, 3, 3

Factors of  $n$  are: 2, 2, 2, 2, 3

Common factors are 2, 2 and 3

So our GCD =  $2 \times 2 \times 3 = 12$

Checking Algorithm 2:

36 does not divide 48

Then we enter the while loop.

$r = 48 \text{ mod } 36 = 12$

$n = 36$

$m = 12$

After the first iteration of Euclid's algorithm is over we are left with  $n = 36$  and  $m = 12$ . At the start of the second iteration we check whether 12 divides 36. Answer is undoubtedly yes. Now loop terminates. The final value that we get is 12.

Comparing the above two algorithms we conclude that the simple algorithm has to do factorization, and then it has to collect common factors and finally multiply the factors together and return the result. Euclid's algorithm on the other hand does have one division i.e., it checks that whether 36 divides 48. When it finds that the division is not possible than it takes the remainder, exchange the numbers and again does one more division. So we may conclude that *Euclid's algorithm does have two divisions where as school level algorithm does have nine divisions. Therefore, Euclid's algorithm is faster than school level algorithm.*

### 1.2.1 Correctness of Euclid's Algorithm

In this section, we don't learn the detailed proof as we learn the basic concepts behind the proof which are outlined as follows.

If  $m$  divides  $n$ , then undoubtedly  $\text{GCD}(m, n)$  is  $m$ , otherwise  $\text{GCD}(m, n)$  is  $\text{GCD}(n \bmod m, m)$ . Suppose  $g$  is the GCD then we can write  $m = ag$  and  $n = bg$  (where  $a$  and  $b$  are relatively prime). As we go through the iterations we have always maintain integers  $m$  and  $n$  and who is GCD, will be the GCD of  $m$  and  $n$ . The specific values of  $m$  and  $n$  might change but their GCD does not. As a result if we ever get out of the loop that will be because  $m$  divides  $n$  but even at this point GCD will be same as that of old GCD of  $m$  and  $n$ . But if  $m$  divides  $n$  then GCD will be  $n$ . Therefore the loop terminates and we will be returning the correct value.

**Theorem 1.1** If Euclid is called with values  $p$  and  $q$  i.e., Euclid  $(p, q)$  and  $m$  and  $n$  are their variables respectively then if  $p < q$ , then in each iteration the sum of the values of variables  $m$  and  $n$  will decrease at least by a factor 3/2.

**Proof:** Our goal is to estimate what happens to the sum of the values taken by  $m$  and  $n$  in each iteration. Recall that at the beginning of the iteration we have  $m = p$  and  $n = q$ .

After first iteration:  $m = p'$  and  $n = q'$ . In order to proof the ratio  $(p + q)/(p' + q')$  is at most 3/2 we need to express  $p'$  and  $q'$  in terms of  $p$  and  $q$ . Call the Euclid's procedure again. If loop does not terminate, it computes  $r = n \bmod m$  and sets  $n = m$ . At this stage the new value of  $n$  is the old value on  $n$ . But the new value on  $n$  must be the old value of  $m$ . Therefore, we get  $n = q' = p$ . Again the new value of  $m$  is same as the value of  $r$  which is nothing but (old value of  $n$ ) mod (old value  $m$ ). Hence  $m = p' = q \bmod p$ . Now take the case of  $p' + q'$ . Here  $p'$  is the remainder when  $q$  is divided by  $p$  and  $q'$  is  $p$  itself. So  $p' + q' = \text{remainder} + \text{divisor}$ . We know that divisor < dividend which implies  $p < q$ . Then  $p' + q' \leq \text{dividend} = q$ . Therefore we conclude  $p'$  has to be less than  $p$ . Combining the above results we get

$$\begin{aligned} p' + q' + 2(p' + q') &< p + p + 2q \\ \Rightarrow 3(p' + q') &< 2(p + q) \\ \Rightarrow (p' + q') &< 2(p + q)/3 \end{aligned}$$

This concludes the analysis of Euclid's algorithm.

## CHAPTER NOTES

---

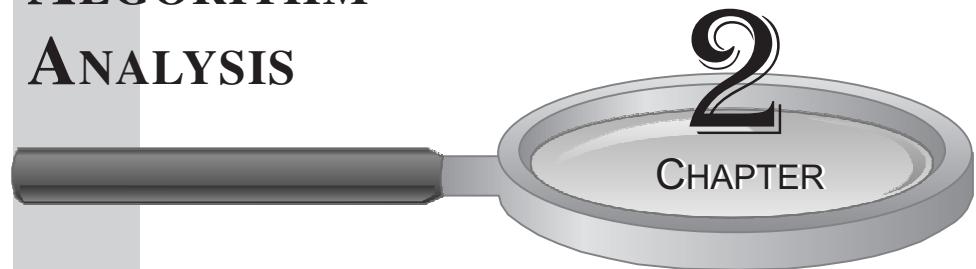
- We have studied various properties of designing fast algorithms.
- Here we have seen two general methods for the calculation of greatest common divisor. One is old fashion school level method which taught at primary classes and another is the Euclid's algorithm. Finally we get to the conclusion that the Euclid's method is faster than the school level method.
- It is easier to design algorithms by counting number of iterations.

## EXERCISES

---

1. How Euclid's algorithm for GCD calculation differs from general school level algorithm?
2. Show that Euclid's algorithm is correct.
3. How can you design fast algorithms?
4. If Euclid's algorithm for GCD problem called with values  $p$  and  $q$ , then prove that the number of iterations executed by this algorithm is  $\leq \log 3/2 (p + q)$ .

# **FRAMEWORK OF ALGORITHM ANALYSIS**



## **OBJECTIVES OF LEARNING**

**After going through this chapter, the reader would be able to understand:**

- Formal/Informal definitions of few basic terms
- Different forms of algorithm
- Random access machine
- External memory model and PRAM model

**INSIDE THIS CHAPTER**

- 2.1 Introduction
- 2.2 The Basics
- 2.3 Definitions of Preliminary Terms
- 2.4 Phases of Algorithm Construction
- 2.5 Mathematical Model of a Computer : RAM Model
- 2.6 EM Model
- 2.7 PRAM Model

---

**2 . 1 INTRODUCTION**

---

The framework that we design can be used not only for comparing the execution time of algorithms which is that we primarily mean algorithm analysis but it also can be used for comparing other resources that an algorithm might use. For example, an algorithm might use varying amounts of memory. So we could use essentially the same framework that we are going to discuss very soon and use that framework to formally compare the memory requirements of different programs or different algorithms.

---

**2 . 2 THE BASICS**

---

Anyone of us who wants to analyse any algorithm (as we have analysed one in Chapter-1), but had till now failed to do so, here is a good news for us! Essentially we are going to make a *mathematical model of a computer* and then we mentally execute the algorithm on that model and through this execution we will be able to say how much time the algorithm takes to execute. This is what we call it *analysis*.

So our basic idea is: Build a mathematical model of a computer. Mentally execute algorithm on that model and determine the execution time. In order to develop it we need to answer several questions as follows:

- What is the mathematical going to be? Which essentially asking that what is the time required on the model for every operation that an algorithm might perform?
- What should be the input data? When we want to estimate the time taken by an algorithm; then we must have a very clear idea about what input is being given to that algorithm; as the time of execution generally depends upon the input.
- How does our model relate to real computers? If our model is terribly different then our conclusions for the model might not be too useful for real computers as we want our conclusions eventually apply to real computers.

Before looking at mathematical model of a computer let us observe some important definitions and different forms of algorithm.

### **2.3 DEFINITIONS OF PRELIMINARY TERMS**

---

#### **Problem**

A *problem* is a specification of what valid inputs are and what acceptable outputs for each valid input. Examples of different types of problems are:

- Locating GCD of two numbers.

Suppose the inputs to this problem are 36 and 48, then the GCD is 12. So 12 is the output.

- Finding the shortest path on a map.

For this problem the names of two cities in a map and name of the map which constitute valid inputs and acceptable output would be the description of the path.

- Searching the actual meaning of a word in dictionary.

Here the inputs are the word and the name of the dictionary which we have to use for searching the word. Output is the meaning of the word which has been given in the dictionary.

- In given X-ray whether the disease is found or not.

For this problem we have to supply an actual X-ray as input. The output is either there is disease or not. So the output is of the form of yes or no.

#### **Input Instance**

A value  $x$  is an *input instance* for problem P, if  $x$  is a valid input as per the specification.

#### **Size of an Instance**

It denotes the number of bits needed to represent the input instance. The specific input instance will have a certain specific size. If we look at 36 and 48 which constitute the input instance for

the GCD problem, then how many bits are required to represent 36 and 48? Suppose we say the numbers are going to be represented in binary then 36 and 48 will be represented as 100100 (i.e., 6 bits) and 110000 (i.e., 6 bits). So in this case the *size of the input instance* is  $6 + 6 = 12$ .

For second problem a map can be represented in graph. A map could be represented in terms of metrics and the metrics could be represented as an array of bits.

### **Algorithm**

A solution for a given problem may be in the form of an *algorithm* or *an algorithm* is a step by step procedure for solving the given problem. It takes some value or values as input and produces a value or values as output.

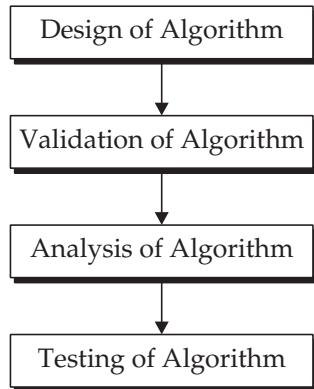
### **Program**

A *program* is a language specific implementation of the algorithm.

## **2.4 PHASES OF ALGORITHM CONSTRUCTION**

---

For constructing a new algorithm for any problem we have to pass through four different phases. These phases are shown in Fig. 2.1.



**Fig. 2.1.** Phases of algorithm construction.

### **Design of Algorithm**

Algorithm is written in step by step procedure by using pseudo code in English. No programming language is used to design the algorithm. An algorithm is written in specific purpose. There are different ways or techniques that we follow. They are:

- Brute-force approach or Naive approach
- Divide and conquer approach
- Dynamic programming approach

- Greedy approach
- Branch and bound approach

We will come across these techniques as we proceed further. But for the moment we should be remain silent.

### **Validation of Algorithm**

After design the algorithm the validity of it is testing. For validation checking all possible input sequence is supplied to the algorithm and expected the desired correct output. If for every input sequence the algorithm can map a correct output then it is valid. The input sequence given to the algorithm is turned as instance problem or problem characteristics.

### **Analysis of Algorithm**

Analysis of algorithm can be made thoroughly by considering each and every statement and by considering some domain statement. The former form of analysis is *micro analysis* and the later form is *macro analysis*. Throughout the book we analyze different algorithms by using RAM model that will be discussed soon. But note that if we use any other model like PRAM model for our analysis then the running time of the algorithm will depend upon the number of processors, computation time, communication delay time in distributed model provided all the processors should run concurrently.

The analysis of algorithm is subject to complex studies. They are of mainly two kinds. One is *time complexity* and other is *space complexity*.

### **Time Complexity**

Time complexity is the amount of time taken by the algorithm to successfully complete its execution. There are two methods:

(i) *Priori Analysis*: It is based on determining the order of the magnitude of the statement.

This method is independent of machine, programming language and operating system. If algorithm has to be analyzed further in detail then each operation (arithmetic, relational and logical) is considered to take 1 unit of time.

(ii) *Posteriori Analysis*: Posteriori analysis refers to the technique of coding a given solution and then measuring its efficiency which provides the actual time taken by the program.

This is useful in practice. The drawback of posteriori analysis is that it depends upon the programming language, the processor and quite a lot of other external parameters.

### **Space Complexity**

The better the time complexity of an algorithm, the faster the algorithm will carry out in practice. Apart from time complexity, its space complexity is important. This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible too. There is often a *time-space-tradeoff* involving a problem, i.e., it cannot be solved with few computing time and low memory consumption.

The space needed by each of the algorithm is the sum of the following components:

- (i) A *fixed part* denotes the space of inputs and outputs. It is also dependent upon space taken by instructions, identifiers and variables.
- (ii) A *variable part* that consists of the space needed by the component variable whose size is dependent upon the particular problem instance being solved.

So the space requirement  $S(A)$  of any algorithm  $A$  can be given as:

$$S(A) = c + SA$$

where  $c$  is a constant *i.e.*, fixed part and  $SA$  is space dependent instance characteristics which denotes variable part.

The *behaviour of algorithm* (which is parameterized by either time or space complexity) can be measured in three ways given as follows.

- (i) *Worst case analysis*: It is the Maximum time required by the algorithm to successfully complete execution while performing time complexity algorithm. It gives us an upper bound on the running time for any input.
- (ii) *Best case analysis*: It is the minimum time required by the algorithm to successfully complete execution while performing time complexity analysis. It gives us the lower bound on the running time for any input.
- (iii) *Average case analysis*: It is the average time taken on the average input size while performing time complexity.

### **Testing of Algorithm**

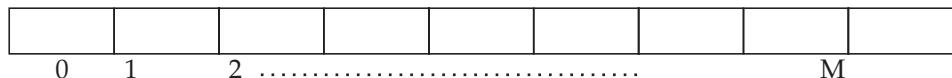
Testing is accomplished by two steps:

- (i) *Debugging*: Debugging is the process of finding error and fixing it. Debugging phase may not take place always if the algorithm contain no error then debugging is not done.
- (ii) *Performance measurement*: When there are two or more algorithms written for a problem it is necessary to measure their *performance*. There are so many factors that affect the performance. The factors are mainly input size, memory available, machine speed etc.

---

### **2.5 MATHEMATICAL MODEL OF A COMPUTER: RAM MODEL**

We will now analyze the algorithm through our abstract mathematical model which is called **RAM**. RAM is essentially a *random access machine* and it is of unbounded capacity. In RAM there is a processor which will execute our algorithms in the form of computer programs and a memory which is a collection of locations (or cells), addressed 0, 1, 2 .... M. The memory is represented in an array as shown in Fig. 2.2.



**Fig. 2.2.** Memory of RAM.

To make things simple we assume that each cell can hold a number, say an integer, of arbitrary size. A further assumption is that an arithmetic operation (+, -,  $\times$ ,  $\div$ ) takes unit time, regardless of the size of the numbers involved. This assumption is unrealistic in a computation where numbers may grow very large, but is often useful. In RAM instructions are executed one after the other subject to no simultaneous operations is allowed. As is the case with all models, the responsibility for using them properly lies with the user.

### Design Issues of RAM Model

- Variable names allowed.
- We will also allow array and structures.
- The data types which will be used in RAM model are integer and floating point.

### Instruction Set of Processor

The processor is going to have a number of instructions which can be divided into three groups.

- One of them is *arithmetic and logical operation*. So in this context we will allow our program to take two locations from memory, add their content and deposit them in 3<sup>rd</sup> location. Let B and C are two operands. According to this operation add them and put them in one location i.e.,  $A = B + C$ .
- The second type of instruction is *jumps and conditional jumps*. This will also execute in one step. As part of our program we are allowed to use the keyword “Go to” or we will be allowed to write “If  $A > B$  then Go to”.
- Third group of instruction is called *pointer instructions*. Suppose we write  $B = *C$ , where C as pointer or C itself contains the address and we are going to fetch the location whose address contain in C and B will get that value. We can make store on that pointer as  $*C = B$ . As pointer and array are related to each other so our random machine will treat pointers and arrays in similar manner. So in this group we put array operation. Here we consider one dimensional array. So  $A[I] = B$ ,  $B = C[I]$ .

### Limitations of RAM

There is clear trade-off between the simplicity and the fidelity achieved by an abstract model. Few serious limitations of RAM model are listed below:

- One of the obvious drawbacks of RAM model is the *assumption of unbounded capacity* the memory access cost is uniform. In RAM there is single processor and single memory whereas real computers are much more complicated architecture. In real

computers there is a memory hierarchy comprising of registers, several levels of cache, main memory and finally disks.

- In real computers “*pipelining*” concept is used through which several instructions are executed simultaneously where as in RAM we do not have that kind of facility.
- As no *mathematical tool* is available in RAM model for calculation of complex mathematics like probability theory, permutation, combination etc, it is difficult to analyze even a simple algorithm but whereas in case of real computers it is quite so easy.
- Another big issue is that RAM model somewhat suspect for *larger input sizes*. This has been readdressed by external memory model.

## 2.6 EM MODEL

---

EM stands for *external memory* model. In this model, the primary concern is the number of disk accesses. Given the rather high cost of a disk access compared to any CPU operation, this model actually ignores all other costs and counts only the number of disk accesses. The disk is accessed as contiguous memory locations called *blocks*. The blocks have fixed size of  $B$  unit as shown in Fig. 2.3.

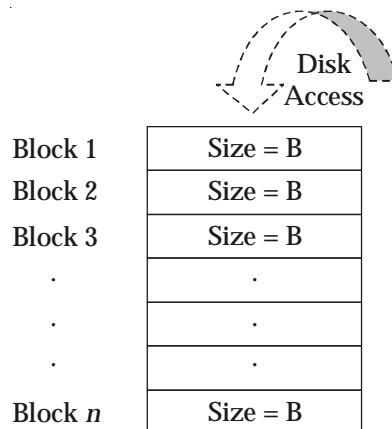


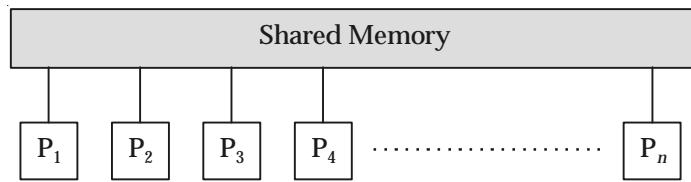
Fig. 2.3. Accessing a disk.

In this two level model, the algorithms are only charged for transferring a block between the internal and external memory and all other computations are free. There are further refinements to this model that parameterizes multiple levels and also accounts for internal computation. As the model becomes more complicated, designing algorithms also becomes more challenging and often more laborious.

## 2.7 PRAM MODEL

---

PRAM stands for *parallel random access machine* which is the analogue of RAM. Here  $n$  numbers of processors are connected to a shared memory as shown in Fig. 2.4 and the communication happens through reading and writing in a globally shared memory. It is left to the algorithm designer to avoid read and write conflicts. It is further assumed that all operations are synchronized globally and there is no cost of synchronization. In this model, there is no extra overhead of communication as it charged in the same way as a local memory access. Even in this model, it has been found that it is not always possible to obtain ideal speedup.



**Fig. 2.4.** PRAM model.

A more realistic parallel model is the *interconnection network* model that has an underlying communication network, usually a regular topology like a two-dimensional mesh, hypercube etc. These can be embedded into VLSI chips and can be scaled according to our needs. To implement any parallel algorithm, we have to design efficient schemes for data routing.

A very common model of parallel computation is a *hardware circuit comprising of basic logic gates*. The signals are transmitted in parallel through different paths and the output is a function of the input. The size of the circuit is the number of gates and the (parallel) time is usually measured in terms of the maximum path length from any input gate to the output gate (each gate contributes to a unit delay). Those familiar with circuits for addition, comparison can analyze them in this framework.

One of the most fascinating developments is the *quantum model* which is inherently parallel but it is also fundamentally different from the previous models. A breakthrough result in recent years is a polynomial time algorithm for factorization which forms the basis of many cryptographic protocols in the conventional model.

*Biological computing* models are a very active area of research where scientists are trying to assemble a machine out of DNA strands. It has potentially many advantages over silicon based devices and is inherently parallel.

## CHAPTER NOTES

---

- Before constructing an algorithm for any given problem we have to pass through various phases. They are: design of algorithm, validation of algorithm, analysis of algorithm and testing of algorithm.
- Designing an algorithm is not a tedious task. There are so many tools and techniques available to make our task easier.

- Analysis of algorithm consists of two major parts: time complexity and space complexity.
- Testing of an algorithm is performed by two steps: debugging and performance measurement.
- We construct an abstract mathematical model called RAM model and it is the first step taken by us towards analysis of algorithms through any model. We find out that RAM model is terribly different than real computers.
- Apart from RAM model there are also different advanced models present. They are external memory model (EM model) and parallel random access machine model (PRAM model).
- EM model works on the principle of number of disk accesses whereas the popular PRAM model works on number of processors connected to a shared memory.

## EXERCISES

---

1. Discuss various phases of algorithm construction.
2. What is RAM model? Discuss its design issues and instruction set of the processor?
3. What are the limitations of RAM model and how these limitations are overcome by External memory model?
4. How real computers differ from the RAM model?
5. Define the following terms: algorithm, problem, input instance, and program.
6. Discuss PRAM model with its various applications.
7. Explain how external memory model works?
8. Calculate the number of steps needed to execute for *loop*.

```
for i = 1 to n  
    C[i] = A[i] + B[i]  
end for
```

# ASYMPTOTIC NOTATION

3

CHAPTER

## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- Asymptotic analysis
- Different notations for rate of growth of relations
- Relational properties of asymptotic notations
- Ceil and floor
- Logarithmic functions and exponential functions

# Chapter 3

# ASYMPTOTIC NOTATION

## INSIDE THIS CHAPTER

- 3.1 Introduction
- 3.2 Types of Notations
- 3.3 Relational Properties of Asymptotic Notations
- 3.4 Some Standard Notations and Common Functions

## 3.1 INTRODUCTION

In this chapter, we are going to develop notations which allows us to talk nicely about classes of functions.

**Asymptotic Notations:** *Asymptotic notation* is a formal way notation to speak about functions and classify them.

**Asymptotic Analysis:** *Asymptotic analysis* refers to classify the behaviour of functions but in this not too precisely but by putting them in classes.

Now we will see what do we need from the classes that we are going to define. So we want two kinds of features like

- We would like to put functions such as  $10n^3 + 5n^2 + 17$  and  $2n^3 + 3n + 79$  should belong into the same class. Because we are classifying the above functions as cubic and we want them to get together. Again constant multipliers should be ignored. In  $10n^3 + 5n^2 + 17$  the constant multiplier is 10 and in case of  $2n^3 + 3n + 79$  the constant multiplier is 2; we should ignore these. So we want a class notation which allows us to nicely ignore constant multipliers.
- Our class notation should also worry about what happens when  $n \rightarrow \infty$ . As  $n \rightarrow \infty$ ,  $5n^2 + 17$  and  $3n + 79$  parts of the above functions will go out and therefore we are really be worry about  $10n^3$  vs  $2n^3$  and then the first feature that we want in class definition will take over and it will say  $10n^3$  and  $2n^3$  are really the same thing. So the above said functions are in same class.

### 3.2 TYPES OF NOTATIONS

---

There are mainly five kinds of notations are present. They are:

- (i) Theta ( $\Theta$ ) Notation
- (ii) Big ( $O$ ) Notation
- (iii) Big ( $\Omega$ ) Notation
- (iv) Little ( $\circ$ ) Notation
- (v) Little ( $\omega$ ) Notation

The above notations will define function classes and they will do exactly what we discussed so far. Let us study in detail about these notations one by one.

**(i) Theta ( $\Theta$ ) Notation:** Let  $f(n)$   $g(n)$  be two non-negative functions of non-negative arguments, then  $\Theta(g(n))$  is defined as

$\Theta(g(n)) : \{f(n) \mid f(n) \text{ is a non-negative function such that } \exists \text{ constants } c_1, c_2, n_0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ this is true not necessarily for all values of } n \text{ but certainly for } n \geq n_0\}$

Recall the properties that we have previously discussed. Whenever what the class structure we define, we should give importance to  $n \rightarrow \infty$ . Here in the definition of  $\Theta(g(n))$ , “for  $n \geq n_0$ ” is the part of our requirement. We are only bothered about what happens when  $n \geq n_0$ ; we are not at all worry about smaller values of  $n$ , as well as not worry about constant multipliers.

In the relation  $c_1g(n) \leq f(x) \leq c_2g(n)$ , we want  $f(n)$  to be sandwiched between  $c_1g(n)$  and  $c_2g(n)$ .

Looking at Fig. 3.1 our claim is if the function  $f$  occupies the dotted region entirely and does not go beyond this region; then  $f$  is sandwiched between  $c_1g(n)$  and  $c_2g(n)$ , then we will put  $f$  in the class  $\Theta(g(n))$ . Here we are not worry for  $n < n_0$ .

**(ii) Big-O Notation:** If  $f(n)$  and  $g(n)$  be two non-negative functions of non-negative arguments, then  $O(g(n))$  is defined as

$O(g(n)) : \{f(n) \mid f(n) \text{ is a non-negative function such that } \exists \text{ constants } c_2 \text{ and } n_0 \text{ such that } f(n) \leq c_2g(n) \text{ for } n \geq n_0\}$

So O-notation gives the asymptotic upper bound for a function to within a constant factor. The Fig. 3.2 represents the representation for O-notation.

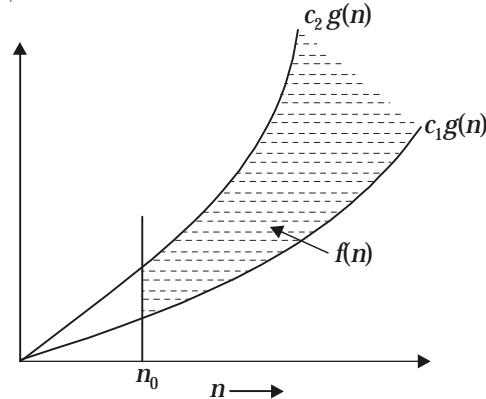


Fig. 3.1. Theta notation representation.

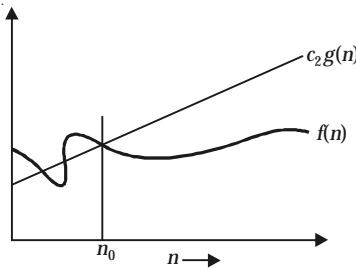


Fig. 3.2. Big-O notation representation.

**(iii) Big- $\Omega$  Notation:** If  $f(n)$  and  $g(n)$  be two non-negative functions of non-negative arguments, then  $\Omega(g(n))$  is defined as

$$\Omega(g(n)) : \{f(n) \mid f(n) \text{ is a non-negative function such that } \exists \text{ constants } c_1 \text{ and } n_0 \text{ such that } c_1 g(n) \leq f(n) \text{ for } n \geq n_0\}$$

The  $\Omega$ -notation provides us asymptotic lower bound for a function within a constant factor. The Fig. 3.3 represents  $\Omega$ -notation.

**(iv) Little-o Notation:** If  $f(n)$  and  $g(n)$  be two non-negative functions of non-negative arguments, then  $o(g(n))$  is defined as

$$o(g(n)) : \{f(n) \mid f(n) \text{ is a non-negative function such that } \exists \text{ constants } c_2 \text{ and } n_0 \text{ such that } f(n) < c_2 g(n) \text{ for } n \geq n_0\}$$

The little-o notation is used to denote an upper bound that is not asymptotically tight. Fig. 3.4 represents  $o$ -notation.

In  $o$ -notation, the function  $f(n)$  becomes insignificant relative to  $g(n)$  as  $n$  approaches to infinity;

$$\text{i.e., } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

**(v) Little- $\omega$  Notation:** Let  $f(n)$  and  $g(n)$  be two non-negative functions of non-negative arguments, then  $\omega(g(n))$  is defined as

$$\omega(g(n)) : \{f(n) \mid f(n) \text{ is a non-negative function such that } \exists \text{ constants } c_1 \text{ and } n_0 \text{ such that } c_1 g(n) < f(n) \text{ for } n \geq n_0\}$$

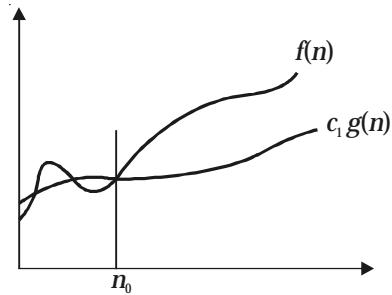
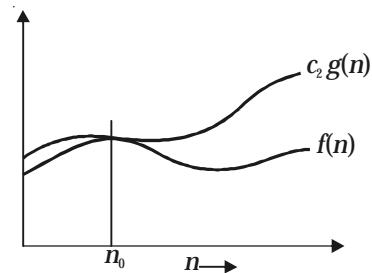
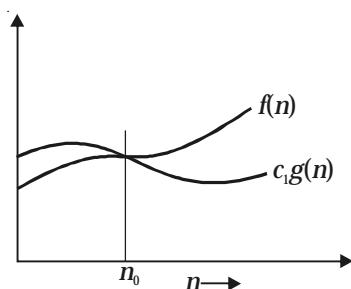
Fig. 3.3. Big- $\Omega$  notation representation.

Fig. 3.4. Little-o notation representation.

Fig. 3.5. Little- $\omega$  notation representation.

The little- $\omega$  notation use to denote a lower bound *i.e.*, not asymptotically tight. The Fig. 3.5 represents  $\omega$ -notation.

$$\text{The relation } f(n) = \omega(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

### **3.3 RELATIONAL PROPERTIES OF ASYMPTOTIC NOTATIONS**

---

If  $f(n)$  and  $g(n)$  be two non-negative functions having non-negative arguments, then many of the relational properties of required numbers are also applicable to it. They are:

**(i) Transitive property**

- (a) If  $f(n) = \Theta(g(n))$  and  $g(n) = \Theta(h(n))$   
 $\Rightarrow f(n) = \Theta(h(n))$
- (b) If  $f(n) = O(g(n))$  and  $g(n) = O(h(n))$   
 $\Rightarrow f(n) = O(h(n))$
- (c) If  $f(n) = \Omega(g(n))$  and  $g(n) = \Omega(h(n))$   
 $\Rightarrow f(n) = \Omega(h(n))$
- (d) If  $f(n) = o(g(n))$  and  $g(n) = o(h(n))$   
 $\Rightarrow f(n) = o(h(n))$
- (e) If  $f(n) = \omega(g(n))$  and  $g(n) = \omega(h(n))$   
 $\Rightarrow f(n) = \omega(h(n))$

**(ii) Reflexive property**

- (a)  $f(n) = \Theta(f(n))$
- (b)  $f(n) = O(f(n))$
- (c)  $f(n) = \Omega(f(n))$

**(iii) Symmetric property**

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

**(iv) Transpose property**

$$\begin{aligned} f(n) = O(g(n)) &\text{ iff } g(n) = \Omega(f(n)) \\ f(n) = O(g(n)) &\text{ iff } g(n) = \omega(f(n)) \end{aligned}$$

### **3.4 SOME STANDARD NOTATIONS AND COMMON FUNCTIONS**

---

**(i) Monotonicity**

- A function  $f(n)$  is monotonically increasing if  $m \leq n \Rightarrow f(m) \leq f(n)$ .
- A function  $f(n)$  is monotonically decreasing if  $m \leq n \Rightarrow f(m) \geq f(n)$ .

➤ A function  $f(n)$  is strictly increasing if  $m < n \Rightarrow f(m) < f(n)$ .

➤ A function  $f(n)$  is strictly decreasing if  $m < n \Rightarrow f(m) > f(n)$ .

**(ii) Ceil and Floor:** The ceil of any real number  $n$  is represented as  $\lceil n \rceil$  and is defined as the least integer greater than or equal to  $n$ .

**Example:**  $\lceil 4.7 \rceil = 5, \lceil 9.3 \rceil = 10$

The floor of any real number  $n$  is represented as  $\lfloor n \rfloor$  and is defined as the greatest integer less than or equal to  $n$ .

**Example:**  $\lfloor 4.7 \rfloor = 4, \lfloor 9.3 \rfloor = 9$

For any real integer  $n$   $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

For any real number  $n \geq 0$  and integers  $x, y > 0$

$$\begin{aligned} \left\lceil \frac{\lceil n/x \rceil}{y} \right\rceil &= \lceil n/xy \rceil & \left\lfloor \frac{\lfloor n/x \rfloor}{y} \right\rfloor &= \lfloor n/xy \rfloor \\ \lceil x/y \rceil &\leq (x + (y - 1))/y & \lfloor x/y \rfloor &\geq (x - (y - 1))/y \end{aligned}$$

**(iii) Modular Arithmetic:** For any integer  $a$  and any positive integer  $n$ , the value of  $a \bmod n$  is the remainder of the quotient  $a/n$ :

$$a \bmod n = a - n \lfloor a/n \rfloor$$

If  $a \bmod n = b \bmod n$ , we write  $a \equiv b \pmod{n}$  and say that  $a$  is equivalent to  $b$  modulo  $n$ .

**(iv) Polynomials:** For a positive integer  $n$ , a polynomial in  $p$  of degree  $d$  is a function  $f(n)$  of the form.

$$f(n) = \sum_{i=0}^d a_i p^i = a_0 p^0 + a_1 p^1 + a_2 p^2 + \dots + a_d p^d$$

where, the constants  $a_0, a_1, a_2 \dots a_d$  are the coefficients of the polynomial and  $a_d \neq 0$ .

#### (v) Logarithmic Functions

$$\text{Binary logarithm : } \lg n = \log_2 n$$

$$\text{Natural logarithm : } \ln n = \log_e n$$

$$\text{Exponential logarithm : } \log^k n = (\log n)^k$$

$$\text{Composite logarithm : } \log \log n = \log(\log n)$$

$$\text{For any real numbers } a > 0, b > 0, c > 0 \text{ and } n$$

$$a = b^{\log_b a}$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b \left( \frac{1}{a} \right) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Series expansion for  $\log(1 + x)$  when  $|x| > 1$  is

$$\log(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

**(vi) Exponential Functions:** For all real nos  $a > 0$ ,  $x$  and  $y$

$$a^0 = 1 \quad a^1 = a$$

$$a^{-1} = \frac{1}{a} \quad (a^x)^y = a^{xy}$$

$$(a^x)^y = (a^y)^x \quad a^m \cdot a^n = a^{m+n}$$

$$\frac{a^m}{a^n} = a^{m-n}$$

For all real  $x$        $e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3} + \dots = \sum_{i=1}^{\infty} \frac{x^i}{i}$

For all real  $x = 0$        $e^x \geq 1 + x$

When  $|x| \leq 1$  we have the approximation  $1 + x \leq e^x \leq 1 + x + x^2$

$$\text{For all } x, \lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n = e^x$$

**Example 3.1** Prove that if  $f(n) = 10n^3 + 5n^2 + 17$  then it belongs to  $\Theta(n^3)$ .

**Solution:** Clearly       $10n^3 \leq f(n) \leq (10 + 5 + 17)n^3$ .

$$\Rightarrow \quad 10n^3 \leq f(n) \leq 32n^3. \text{ This is true for all } n.$$

$$\text{Here,} \quad c_1 = 10, c_2 = 32$$

$$\text{So, } c_1 n^3 \leq f(n) \leq c_2 n^3; \text{ for all } n \geq 1 = n_0$$

$$\text{So, } f(n) = 10x^3 + 5x^2 + 17 \in \Theta(x^3).$$

**Example 3.2** Prove that if  $f(n) = 10n^3 + n\log n$  then it belongs to  $\Theta(n^3)$ .

**Solution:** We have  $f(n) = 10n^3 + n\log n$ , As  $n$  increases to  $\infty$  then

$$f(n) = 10n^3 \quad (\text{$n\log n$ can be neglected for large } n)$$

$$\text{Then} \quad 10n^3 \leq f(n) \leq 11n^3$$

$$\text{So,} \quad c_1 = 10, c_2 = 11 \quad \text{and} \quad n_0 = 1$$

$$f(n) = 10n^3 + n\log n \in \Theta(n^3).$$

**Example 3.3** Prove that if  $f(n) = 27$  then it belongs to  $O(1)$ .

**Solution:** We have       $f(n) = 27 = 27 * 1$

$$\leq c_2 g(n)$$

$$\begin{aligned} \text{Here} \quad & c_2 = 27, g(n) = 1 \\ \text{So,} \quad & f(n) = O(g(n)) \\ \Rightarrow \quad & 27 = O(1). \end{aligned}$$

**Example 3.4** Prove that if  $f(n) = 5n^2 + 4n + 2$  then it belongs to  $O(n^2)$ .

**Solution:** We have  $f(n) = 5n^2 + 4n + 2$

As  $n$  dominates 2,

$$\begin{aligned} 5n^2 + 4n + 2 &\leq 5n^2 + 4n + n \\ &\leq 5n^2 + 5n \end{aligned}$$

As  $n^2$  deminates  $5n$ ,

$$\begin{aligned} 5n^2 + 5n &\leq 5n^2 + n^2 \\ &\leq 6n^2 \end{aligned}$$

$$\text{So, } 5n^2 + 4n + 2 \leq 5n^2 + 5n \leq 6n^2$$

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

$$\text{Here, } c_2 = 6, g(n) = n^2, n_0 = 5$$

$$\text{So, } f(n) = 5n^2 + 4n + 2 \in O(n^2).$$

**Example 3.5** Prove that if  $f(n) = 2 + 1/n$  then it is belongs to  $\Theta(1)$ . Given  $g(n) = 1$

$$\text{Solution: } 2 \leq 2 + \frac{1}{n} \leq 3$$

$$\text{So, } c_1 = 2, c_2 = 3, n_0 = 1$$

$$\text{So, } f(n) = 2 + 1/n \in \Theta(1)$$

**Example 3.6** Let  $f(n)$  and  $g(n)$  asymptotically non-negative functions. Using the basic definition of  $\Theta$ -notation, prove that  $\max(f(n), g(n)) = \Theta(f(n) + g(n))$

**Solution:**

$f(n)$  and  $g(n)$  are +ve for large value of  $n$ ,  $\exists c_1, c_2, n_1 > 0$

Such that  $c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n))$

[This is true according to  $\Theta$ -notation definition.]

$$\text{So, } c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \quad \dots(i)$$

$$\max(f(n) + g(n)) \leq c_2(f(n) + g(n)) \quad \dots(ii)$$

If we can prove the above two equations, then we can show that

$$\max(f(n), g(n)) = \Theta(f(n) + g(n))$$

$$\text{We can write } f(n) + g(n) = \max(f(n), g(n)) + \min(f(n), g(n)) \quad \dots(iii)$$

$$\text{Then } \max(f(n), g(n)) \geq \min(f(n), g(n))$$

$$\text{Again, } f(n) + g(n) \leq 2 \max(f(n), g(n))$$

$$\Rightarrow 1/2(f(n) + g(n)) \leq \max(f(n), g(n)) \quad \dots(iv)$$

From (i) and (iv)  $c_1 = 1/2$ . So equation (i) holds good.

$$\text{Again, } f(n) + g(n) \geq \max(f(n), g(n)) \quad \dots(v)$$

Comparing equation (ii) and (v),  $c_2 = 1$ , So equation (i) holds good.

$$\text{Hence, } \max(f(n), g(n)) = \theta(f(n) + g(n))$$

**Example 3.7** Let  $f(n)$  and  $g(n)$  be asymptotically non-negative functions. Using the basic definition of  $\theta$  prove that  $(f(n) + g(n)) = \theta(\max(f(n), g(n)))$ .

**Solution:** According to the basic defintion of  $\theta$ ; if  $f(n)$  and  $g(n)$  are +ve for large value of  $n$ ,  $\exists c_1, c_2$  and  $n_1 > 0$  such that

$$c_1 \max(f(n), g(n)) \leq f(n) + g(n) \leq c_2 \max(f(n), g(n))$$

$$\text{So, } c_1 \max(f(n), g(n)) \leq f(n) + g(n) \quad \dots(i)$$

$$f(n) + g(n) \leq c_2 \max(f(n), g(n)) \quad \dots(ii)$$

If we can prove the above two equations then we can show

$$f(n) + g(n) = \theta(\max(f(n), g(n)))$$

$$\text{We can write } f(n) + g(n) = \max(f(n), g(n)) + \min(f(n), g(n))$$

$$\text{and } \max(f(n), g(n)) \geq \min(f(n), g(n))$$

$$\text{Again, } f(n) + g(n) \leq 2 \max(f(n), g(n)) \quad \dots(iii)$$

Equating eqation (ii) and (iii),  $c_2 = 2$ . So equation (ii) holds good.

$$\text{Again, } f(n) + g(n) \geq \max(f(n), g(n)) \quad \dots(iv)$$

So, from equation (i) and (iv),  $c_1 = 1$ . So equation (i) holds good.

$$\text{Hence, } f(n), g(n) = \theta(\max(f(n), g(n))).$$

**Example 3.8** Show that for any real constants  $a$  and  $b$  where  $b > 0$   $(n + a)^b = \theta(n^b)$

**Solution:** To show that  $(n + a)^b = \theta(n^b)$ , we want to find constants  $c_1, c_2, n_0 > 0$  such that  $c_1 n^b \leq (n + a)^b \leq c_2 n^b$  for all  $n \geq n_0$ .

$$\text{Note that, } n + a \leq n + |a|$$

$$\leq 2n \quad \text{when } |a| \leq n,$$

$$\text{and } n + a \geq n - |a|$$

$$\geq (1/2)n \quad \text{when } |a| \leq (1/2)n$$

Thus, when  $n \geq 2|a|$ ,  $(1/2)n \leq n + a \leq 2n$

Since  $b > 0$ , the inequality still holds, when all parts are raised to the power  $b$ :

$$((1/2)n)^b \leq (n + a)^b \leq (2n)^b$$

$$(1/2)^b n^b \leq (n + a)^b \leq 2^b \cdot n^b$$

Thus,  $c_1 = (1/2)^b$ ,  $c_2 = 2^b$  and  $n_0 = 2|a|$  satisfy the definition.

**Example 3.9** Is  $2^{n+1} = O(2^n)$ ? Is  $2^{2n} = O(2^n)$ ? Justify your answers.

**Solution:**  $2^{n+1} = O(2^n)$ , but  $2^{2n} \neq O(2^n)$

To show that  $2^{n+1} = O(2^n)$ , we must find constants  $c, n_0 > 0$  such that  $2^{n+1} \leq c \cdot 2^n$  for all  $n \geq n_0$ .

Since,  $2^{n+1} = 2 \cdot 2^n$  for all  $n$ , we can satisfy the definition with  $c = 2$  and  $n_0 = 1$ .

To show that,  $2^{2n} \neq O(2^n)$ , assume there exist constants  $c, n_0 > 0$  such that  $2^{2n} \leq c \cdot 2^n$  for all  $n \geq n_0$ .

Then  $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$ . But no constant is greater than all  $2^n$  and so the assumption leads to a contradiction.

**Example 3.10** Prove that  $o(g(n)) \cap \omega(g(n))$  is the empty set.

**Solution:** Suppose not. Let  $f(n) \in o(g(n)) \cap \omega(g(n))$ . Now  $f(n) = \omega(g(n))$  if and only if  $g(n) = o(f(n))$  and  $f(n) = o(g(n))$  by assumption. By transitive property  $f(n) = o(f(n))$  i.e., for all constants  $c > 0$ ,  $f(n) < cf(n)$ .

Choose  $c < 1$  and we have the desired contradiction from the asymptotic non-negativity of  $f(n)$ .

**Example 3.11** Show that if  $f(n)$  and  $g(n)$  are monotonically increasing functions, then so are  $f(n) + g(n)$  and  $f(g(n))$  and if  $f(n)$  and  $g(n)$  are in addition non-negative, then  $f(n) \cdot g(n)$  is monotonically increasing.

**Solution:**

(a) We must show that  $f(n)$  and  $g(n)$  are monotonically increasing functions, then so is  $f(n) + g(n)$ .

Suppose not. Let  $n_1 < n_2$  and  $f(n_1) + g(n_1) > f(n_2) + g(n_2)$ .

Now  $f(n_1) \leq f(n_2)$  and  $g(n_1) \leq g(n_2)$ .

$$f(n_1) \leq f(n_2)$$

$$f(n_1) + g(n_1) \leq f(n_2) + g(n_1)$$

$$f(n_1) + g(n_1) \leq f(n_2) + g(n_2)$$

This contradicts our assumption

(b) We must show that if  $f(n)$  and  $g(n)$  are monotonically increasing functions, then so is  $f(g(n))$ . Suppose not. Let  $n_1 < n_2$  and  $f(g(n_1)) > f(g(n_2))$

Let  $m_1 = g(n_1), m_2 = g(n_2)$  and  $m_1 \leq m_2$ .

Clearly  $f(m_1) \leq f(m_2)$ , which contradicts our assumption.

(c) We must show that if  $f(n)$  and  $g(n)$  are monotonically increasing functions, then so is  $f(n) \cdot g(n)$  if  $f$  and  $g$  are non-negative. Let  $n_1 < n_2$  and  $f(n_1) + g(n_1) > f(n_2) + g(n_2)$ .

$$f(n) \leq f(n_2) \text{ and } g(n_1) \leq g(n_2)$$

Now,  $f(n_1) \leq f(n_2)$

$$f(n_1) g(n_1) \leq f(n_2) g(n_1)$$

$$f(n_1) g(n_1) \leq f(n_2) g(n_2)$$

This contradicts our assumption.

**Example 3.12** Is the function  $\lceil \log n \rceil!$  polynomially bounded? Is the function  $\lceil \log \log n \rceil!$  polynomially bounded? Justify your answers.

**Solution:**  $\lceil \log n \rceil!$  is not polynomially bounded, but  $\lceil \log \log n \rceil!$  is.

Proving that a function  $f(n)$  is polynomially bounded is equivalent to prove that  $\log(f(n)) = O(\log n)$  for the following reasons.

- If  $f$  is polynomially bounded, then there exist constants  $c, k, n_0$  such that for all  $n \geq n_0$ ,  $f(n) \leq cn^k$ . Hence,  $\log(f(n)) \leq kc \log n$ , which, since  $c$  and  $k$  are constants, means that  $\log(f(n)) = O(\log n)$ .
- Similarly, if  $\log(f(n)) = O(\log n)$ , then  $f$  is polynomially bounded.

In the following proofs, we will make use of the following two facts:

- (i)  $\log(n!) = \Theta(n \log n)$
- (ii)  $\lceil \log n \rceil = \Theta(\log n)$  because

1.  $\lceil \log n \rceil \geq \log n$
2.  $\lceil \log n \rceil < \log n + 1 \leq 2 \log n$  for all  $n \geq 2$

$$\log(\lceil \log n \rceil!) = \Theta(\lceil \log n \rceil \log \lceil \log n \rceil) = \Theta(\log n \log \log n) = \omega(\log n)$$

$\therefore \log(\lceil \log n \rceil!) \neq O(\log n)$  and so  $\lceil \log n \rceil!$  is not polynomially bounded.

$$\begin{aligned} \therefore \log(\lceil \log \log n \rceil!) &= \Theta(\lceil \log \log n \rceil \log \lceil \log \log n \rceil) \\ &= \Theta(\log \log n \log \log \log n) \\ &= O((\log \log n)^2) \\ &= O(\log^2(\log n)) \\ &= O(\log n). \end{aligned}$$

The last step above follows from the property that any polylogarithmic function grows more slowly than any positive polynomial function, i.e., for constants  $a, b > 0$ , we have  $\log^b n = o(n^a)$ . Substitute  $\log n$  for  $n$ , 2 for  $b$ , and 1 for  $a$ , giving  $\log^2(\log n) = o(\log n)$ .

$\therefore \log(\lceil \log \log n \rceil!) = O(\log n)$  and so  $\lceil \log \log n \rceil!$  is polynomially bounded.

**Example 3.13** Prove by induction that the  $k$ th Fibonacci number satisfies the equality  $F_k = \frac{\phi^k - \hat{\phi}^k}{\sqrt{5}}$ .

Where  $\phi$  is the golden ratio  $\left( \text{i.e. } \phi = \frac{1+\sqrt{5}}{2} \right)$  and  $\hat{\phi}$  is its conjugate  $\left( \text{i.e., } \hat{\phi} = \frac{1-\sqrt{5}}{2} \right)$ .

**Solution:** We have

$$F_k = \left\{ \left( \frac{1+\sqrt{5}}{2} \right)^k - \left( \frac{1-\sqrt{5}}{2} \right)^k \right\} / \sqrt{5}$$

$$\text{When } k = 1 : \quad F_1 = \frac{1+\sqrt{5}}{2\sqrt{5}} - \frac{1-\sqrt{5}}{2\sqrt{5}} = \frac{1+\sqrt{5}-1+\sqrt{5}}{2\sqrt{5}} = \frac{2\sqrt{5}}{2\sqrt{5}} = 1$$

We have to prove  $F(k)$  is true assuming  $F(0), F(1) \dots F(k-1)$  all are true.

$$\text{So, } F_{k-1} = \frac{1}{\sqrt{5}} \phi^{k-1} - \frac{1}{\sqrt{5}} \hat{\phi}^{k-1} \text{ (assumption)}$$

$$F_{k-2} = \frac{1}{\sqrt{5}} \phi^{k-2} - \frac{1}{\sqrt{5}} \hat{\phi}^{k-2} \text{ (assumption)}$$

Then we have to prove,

$$F_k = \frac{1}{\sqrt{5}} \phi^k - \frac{1}{\sqrt{5}} \hat{\phi}^k$$

$$\text{We know that, } F_k = F_{k-1} + F_{k-2}$$

$$\begin{aligned} &= \frac{1}{\sqrt{5}} \phi^{k-1} - \frac{1}{\sqrt{5}} \hat{\phi}^{k-1} + \frac{1}{\sqrt{5}} \phi^{k-2} - \frac{1}{\sqrt{5}} \hat{\phi}^{k-2} \\ &= \frac{1}{\sqrt{5}} (\phi^{k-1} + \phi^{k-2}) - \frac{1}{\sqrt{5}} (\hat{\phi}^{k-1} + \hat{\phi}^{k-2}) \\ &= \frac{\phi^{k-2}}{\sqrt{5}} (\phi + 1) - \frac{\hat{\phi}^{k-2}}{\sqrt{5}} (\hat{\phi} + 1) \end{aligned}$$

$$\text{But } 1 + \phi = 1 + \frac{1+\sqrt{5}}{2} = \frac{3+\sqrt{5}}{2}$$

$$\text{Again, } \phi^2 = \left( \frac{1+\sqrt{5}}{2} \right)^2 = \frac{1+5+2\sqrt{5}}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2}$$

$$1 + \hat{\phi} = 1 + \frac{1-\sqrt{5}}{2} = \frac{3-\sqrt{5}}{2}$$

$$\hat{\phi}^2 = \frac{3-\sqrt{5}}{2}$$

$$\text{Hence, } F_k = \frac{\phi^{k-2}}{\sqrt{5}} (\phi^2) - \frac{\hat{\phi}^{k-2}}{\sqrt{5}} (\hat{\phi}^2)$$

$$\Rightarrow F_k = \frac{\phi^k}{\sqrt{5}} - \frac{\hat{\phi}^k}{\sqrt{5}} \quad \text{So, it is proved}$$

**Note:** There are various types of induction methods present. They are:

- ⇒ Structural induction
- ⇒ Mutual induction
- ⇒ Quick induction
- ⇒ Strong induction.

We generally apply “Quick induction” to solve our problems. But in the above example we have applied “Strong induction.”

In quick induction:

$$n = n_0$$

$$s(k) = s(k + 1) \text{ for } k \geq n_0$$

$s(k + 1)$  depends on only one previous term.

In strong induction: We assume that all previous terms before  $s(k + 1)$  are true.

**Example 3.14** Rank the following functions by order of growth : that is, find an arrangement  $g_1, g_2, \dots, g_{30}$  of the functions satisfying  $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \dots, g_{29} = \Omega(g_{30})$ . Partition your list into equivalence classes such that  $f(n)$  and  $g(n)$  are in the same class if and only if  $f(n) = \theta(g(n))$ .

$\log(\log^* n)$	$2^{\log^* n}$	$(\sqrt{2})^{\log n}$	$n^2$	$n!$	$(\log n)!$
$\left(\frac{3}{2}\right)^n$	$n^3$	$\log^2 n$	$\log(n!)$	$2^{2n}$	$n^{1/\log n}$
$\ln \ln n$	$\log^* n$	$n \cdot 2^n$	$n^{\log \log n}$	$\ln n$	$1$
$2^{\log n}$	$(\log n)^{\log n}$	$e^n$	$4^{\log n}$	$(n+1)!$	$\sqrt{\log n}$
$\log^*(\log n)$	$2^{\sqrt{2 \log n}}$	$n$	$2^n$	$n \log n$	$2^{2n+1}$

**Solution:** Most of the ranking is fairly straightforward. Several identities are helpful.

$$n^{\log \log n} = (\log n)^{\log n}$$

$$n^2 = 4^{\log n}$$

$$n = 2^{\log n}$$

$$2^{\sqrt{2 \log n}} = n^{\sqrt{2/\log n}}$$

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

$$\log^*(\log n) = \log^* n - 1 \text{ for } n > 1.$$

In addition, asymptotic bounds for Stirling's formula are helpful in ranking the expressions with factorials:

$$n! = \Theta\left(n^{\frac{(n+1)}{2}} e^{-n}\right)$$

$$\log(n!) = \Theta(n \log n)$$

$$(\log n)! = \Theta\left((\log n)^{\log(n+1)/2} e^{-\log n}\right)$$

Each term gives a different equivalence class, where the  $>$  symbol means  $\omega$ .

$$\begin{aligned}
 2^{2n+1} &> 2^{2n} &> (n+1)! &> n! &> e^n &> \\
 n \cdot 2^n &> 2^n &> \left(\frac{3}{2}\right)^n &> \frac{n^{\log \log n}}{(\log n)^{\log n}} &> (\log n)! &> \\
 n^3 &> \frac{n^2}{4^{\log n}} &> \frac{n \log n}{\log(n!)} &> \frac{n}{2^{\log n}} &> (\sqrt{2})^{\log n} &> \\
 2\sqrt{2 \log n} &> \log^2 n &> \ln n &> \sqrt{\log n} &> \ln \ln n &> \\
 2^{\log^* n} &> \frac{\log^*(\log n)}{\log^* n} &> \log(\log^* n) &> \frac{1}{n^{1/\log n}}
 \end{aligned}$$

**Example 3.15** If  $s(n) = \sum_{i=1}^n i$ . Then prove the  $s(n) \in \Theta(n^2)$ . While proving avoid to use  $\sum_{i=1}^n i = \frac{n(n+1)}{2}$

**Solution:**  $s(n) = \sum_{i=1}^n i \leq \sum_{i=1}^n i = n^2$   
 $\Rightarrow s(n) \in O(n^2)$

Again,  $s(n) = \sum_{i=1}^n i$   
 $\geq \sum_{i=\frac{n}{2}+1}^n i \quad (\text{ignoring the first two terms}) \geq \sum_{i=\frac{n}{2}+1}^n n = \frac{n}{2} \cdot \frac{n}{2}$   
 $= \frac{n^2}{4}$

So,  $s(n) \in \Omega(n^2)$

Now  $s(n) \in \Theta(n^2)$ .

**Example 3.16** Prove the equation  $\log(n!) = \Theta(n \log n)$ , also prove that  $n! = \omega(2^n)$  and  $n! = o(n^n)$ .

**Solution:**  $\log(n!) = \log [n(n-1)(n-2) \dots 1] \leq \log [n \times n \times \dots \times n]$   
 $= \log n^n = n \log n$

Thus,  $\log(n!) \leq c_1 n \log n$  for  $c_1 \geq 1$  and  $n \geq 0$

This proves  $\log(n!) = O(n \log n)$

To prove  $\log(n!) = \Omega(n \log n)$ , we have to use Stirling's approximation:

$$n! = \sqrt{2\pi n} \left[ \frac{n}{e} \right]^n \left[ 1 + \theta\left(\frac{1}{n}\right) \right]$$

Where  $e$  is the base of the natural logarithm which gives us a tighter upper bound and a lower bound as well.

Now prove  $n! \geq cn^n$ .

$$\begin{aligned} n! &\geq \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \left( 1 + \frac{c}{n} \right) = \sqrt{2\pi} \left( \frac{n^{n+0.5}}{e^n} \right) \left( 1 + \frac{c}{n} \right) \\ &= \sqrt{2\pi} \left( \frac{n^{n+0.5}}{e^n} \right) + \sqrt{2\pi} \left( \frac{cn^{n-0.5}}{e^n} \right) \\ &= 2\pi n^n \text{ for } c > 0, n \geq 1. \end{aligned}$$

Hence  $n! \geq n^n$  for  $c > 0, n = 1$ .

If we take log of both sides, the inequality will still be true because log is a monotonic function.

$$\log(n!) \geq \log(n^n) = n \log n \text{ for } c > 0, n = 1$$

Hence,  $\log(n!) = \Omega(n \log n)$

Since we proved  $\log(n!) = \Omega(n \log n)$  and  $\log(n!) = O(n \log n)$   
 $\log(n!) = \Theta(n \log n)$

Now prove  $n! = \omega(2^n)$

As,  $n \times (n-1) \times (n-2) \times \dots \times 1 > c \times 2 \times \dots \times 2$

Divide both sides by  $2^n$

$$\frac{n}{2} \times \frac{(n-1)}{2} \times \frac{(n-2)}{2} \times \dots \times \frac{1}{2} > \frac{c \times 2 \times 2 \times \dots \times 2}{2 \times 2 \times \dots \times 2}$$

Each element in the left side except  $\frac{1}{2}$  is greater than 1.

Hence if  $c > \frac{1}{2}$ , the inequality holds for  $n > 0$  therefore

$$n! = \omega(2^n)$$

Now prove  $n! = o(n^n)$

As  $n \times n \times (n-1) \times (n-2) \times \dots \times 2 \times 1 < c \times n \times n \times \dots \times n$

Inequality holds for  $n > 1$ , and  $c > 1$ .

**Example 3.17** If  $f(n) = 3 \times 2^n + 4n^2 + 5n + 3$  then prove that  $f(n) \in \theta(n^2)$ .

**Solution:** We have  $f(n) = 3 \times 2^n + 4n^2 + 5n + 3 \quad \dots(i)$

For finding  $\theta$ -notation first of all we find lower bound for  $f(n)$  as  $c_1 g(n) \leq f(n) \quad \dots(ii)$

$$\Rightarrow 3 \times 2^n \leq 3 \times 2^n + 4n^2 + 5n + 3 \text{ (for all } n \geq n_0 = 5)$$

$$\Rightarrow c_1 = 3 \text{ (By comparing with equation (ii) and } g(n) = 2^n.$$

Now, we find upper bound for the given function  $f(n)$  as:

$$\begin{aligned} f(n) &\leq c_2 g(n) \quad \dots(iii) \\ \Rightarrow 3 \times 2^n + 4n^2 + 5n + 3 &\leq 8 \times 2^n \text{ (for all } n \geq n_0 = 5) \\ \Rightarrow c_2 &= 8 \quad \text{and} \quad g(n) = 2^n \text{ (By comparing with equation (iii))} \\ \therefore \text{For } \theta\text{-notation a function must be upper and lower bounded as} \end{aligned}$$

$$\begin{aligned} c_1 g(n) &\leq f(n) \leq c_2 g(n) \\ \Rightarrow 3 \times 2^n &\leq 3 \times 2^n + 4n^2 + 5n + 3 \leq 8 \times 2^n \\ \Rightarrow f(n) &= \Theta(g(n)) \\ \Rightarrow f(n) &= \Theta(n^2) \end{aligned}$$

**Example 3.18** If  $f(n) = a_0 + a_1 n + a_2 n^2 + \dots + a_m n^m$  is any polynomial of degree  $m$  or less, then prove that  $f(n) \in \Theta(n^m)$ .

**Solution:** We have:  $f(n) = a_0 + a_1 n + a_2 n^2 + \dots + a_m n^m$

Then  $|f(n)| \leq |a_0| + |a_1| n + |a_2| n^2 + \dots + |a_m| n^m$

$$\begin{aligned} &= \left( \frac{|a_0|}{n^m} + \frac{|a_1|}{n^{m-1}} + \frac{|a_2|}{n^{m-2}} + \dots + |a_m| \right) n^m \\ &\leq (|a_0| + |a_1| + |a_2| + \dots + |a_m|) n^m \end{aligned}$$

When  $n = 1$

$$\begin{aligned} c_2 &= |a_0| + |a_1| + |a_2| + \dots + |a_m| \\ \Rightarrow f &\leq c_2 g(n) \text{ (where } g(n) = n^m) \end{aligned}$$

Similarly, we can check for lower bound.

i.e.,  $c_1 g(n) \leq f(n)$

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

This proves  $f(n) \in \Theta(n^m)$

**Example 3.19** If  $5n^3 + 4n + 6 = O(n^3)$  then prove or disprove  $5n^3 + 4n + 6 = o(n^3)$ .

**Solution:** We have given  $5n^3 + 4n + 6 = O(n^3)$

Here  $f(n) = 5n^3 + 4n + 6$  and  $g(n) = n^3$

For little-o notation;  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

So,  $\lim_{n \rightarrow \infty} \left( \frac{5n^3 + 4n + 6}{n^3} \right) = \lim_{n \rightarrow \infty} \left( \frac{5n^3}{n^3} + \frac{4n}{n^3} + \frac{6}{n^3} \right) = \lim_{n \rightarrow \infty} \left( 5 + \frac{4}{n^2} + \frac{6}{n^3} \right) = 5 \neq 0$

Thus the condition for little-o notation is not satisfied.

Therefore  $5n^3 + 4n + 6 \neq o(n^3)$

**Example 3.20** Prove that for any two functions  $f(n)$  and  $g(n)$  we have  $f(n) = \Theta(g(n))$  if and only if  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n))$ .

**Solution:** We need to prove  $f(n) = O(g(n))$  and  $f(n) = \Omega(g(n)) \leftrightarrow f(n) = \Theta(g(n))$ .

To prove one implication

$$f(n) = O(g(n)) \text{ i.e., } f(n) \leq c_0 g(n), n \geq n_0.$$

$$f(n) = \Omega(g(n)) \text{ i.e., } f(n) \geq c_1 g(n), n \geq n_1.$$

Which means  $c_1 g(n) \leq f(n) \leq c_0 g(n)$  ( $n \geq \max(n_0, n_1)$ )

Now to prove the other side of implication

$$f(n) = \Theta(g(n)) \text{ means } c_0 g(n) \leq f(n) \leq c_1 g(n), n \geq n_0$$

Hence  $f(n) \leq c_0 g(n), n \geq n_0$  means  $f(n) = O(g(n))$

$$f(n) \geq c_1 g(n), n \geq n_0 \text{ means } f(n) = \Omega(g(n))$$

Both sides of implication were proved true. Hence

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \leftrightarrow f(n) = \Theta(g(n)).$$

**Example 3.21** Let  $p(n) = \sum_{i=0}^d a_i n^i$  where  $a_d > 0$ , be a degree-d polynomial in  $n$  and let  $k$  be a constant.

Use the definitions of the asymptotic notation to prove the following properties:

$$(i) \text{ If } k \geq d, \text{ then } p(n) = O(n^k) \quad (ii) \text{ If } k = d, \text{ then } p(n) = \Theta(n^k)$$

**Solution:**  $p(n) = \sum_{i=0}^d a_i n^i$ , where  $a_d > 0$

(i)  $p(n) = O(n^k)$  if  $\sum_{i=0}^d a_i n^i \leq c n^k$  for all  $n > n_0$  and some constant  $c$ . Dividing both sides

by  $n^k$  we get  $\sum_{i=0}^d a_i n^{i-k} \leq c$ . Since  $k \geq d$ , all  $i - k \leq 0$ , i.e., all  $n^{i-k}$  will be less than 1. So,

choose  $c = \sum_{i=0}^d a_i$  the statement is true.

$$(ii) p(n) = \Theta(n^k) \text{ if } c_1 n^k \leq \sum_{i=0}^d a_i n^i \leq c_2 n^k.$$

Divide  $c$  by  $n^k$  we get,

$$c_1 \leq \sum_{i=0}^d a_i n^{i-k} \leq c_2.$$

Since  $k = d$ ,

$$\text{Thus } c_1 \leq a_0 n^{0-k} + a_1 n^{1-k} + a_2 n^{2-k} + \dots + a_k n^{k-k} \leq c_2$$

$$\Rightarrow c_1 \leq a_0 n^{-k} + a_1 n^{1-k} + a_2 n^{2-k} + \dots + a_k n^0 \leq c_2$$

Here, we choose  $c_1 = a_d$  and  $c_2 = \sum_{i=0}^d a_i$ , the statement is true.

## CHAPTER NOTES

---

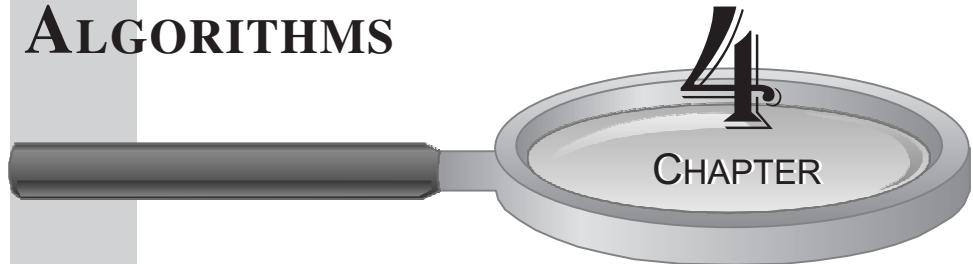
- Asymptotic complexity is the running time of an algorithm as a function of input size for large  $n$ . It is expressed using only the highest-order term in the expression for the exact running time. It describes behaviour of the function in the limit.
- Asymptotic notation describe different rate of growth relations between the defining function and the defined set of functions, in other words time estimates for algorithms expressed using above.
- In compare orders of growth, always look at the leading term.
- In  $O$ -notation  $g(n)$  is an asymptotic upper bound for  $f(n)$ .  
Then  $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$ .  
 $\Theta(g(n)) \subset O(g(n))$ .
- In  $\Omega$ -notation  $g(n)$  is an asymptotic lower bound for  $f(n)$ .  
Then  $f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$ .  
 $\Theta(g(n)) \subset \Omega(g(n))$ .
- The relation between  $\Theta$ ,  $\Omega$ ,  $O$  notations is  $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

## EXERCISES

---

1. Any linear function  $an + b$  is in  $O(n^2)$ . How?
2. Show that  $3n^3 = O(n^4)$  for appropriate  $c$  and  $n_0$ .
3. Is  $3n^3 \in \Theta(n^4)$ ?
4. Does  $2^{2n} \in \Theta(2^n)$ ? Why or why not.
5. Prove that  $\frac{n^2}{2} - 3n = \Theta(n^2)$ .
6. Prove that if  $f(n) = 5n \log n + 10$  then  $f(n) \in \Theta(n \log n)$ .
7. Let  $A(n) = \sum_{i=0}^k a_i n^i$ . Assume  $a_k > 0$ . Then prove that any polynomial of  $k$ th degree i.e.,  $A(n) = \Theta(n^k)$ .
8. If  $T(n) = \sum_{i=1}^k i^2$ . Then prove that  $T(n) \in \Theta(n^3)$
9. Fibonacci series is divide by  $F(n) = F(n-1) + F(n-2)$ .  
Given  $F(1) = F(0) = 1$ . Then prove that  $F(n) \geq 2^{n/2}$  and also prove  $F(n) = \Omega((\sqrt{2})^n)$ .
10. Prove that  $k \ln k = \Theta(n)$  implies  $k = \Theta(n/\ln n)$ .

# **SORTING ALGORITHMS**



## **OBJECTIVES OF LEARNING**

**After going through this chapter, the reader would be able to understand:**

- What is sorting?
- Comparison based sorting
- Non-comparison based sorting
- Performance analysis

# Chapter 4 SORTING ALGORITHMS

## INSIDE THIS CHAPTER

- 4.1 Introduction
- 4.2 Insertion Sort
- 4.3 Bubble Sort
- 4.4 Selection Sort
- 4.5 Counting Sort
- 4.6 Radix Sort
- 4.7 Bucket Sort

## 4.1 INTRODUCTION

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms.

We categorize different sorting algorithms based on following criteria:

- **Comparison based Sorting Algorithm:** A comparison sort examines the data only by comparing two elements with a comparison operator. Some of the well known comparison sorts include insertion sort, bubble sort, quick sort, merge sort, selection sort, etc. Non-comparison based sorting algorithms include radix sort, counting sort and bucket sort.
- **In-place:** In particular, some sorting algorithm are in-place (memory usage *i.e.*, use of computer resources). An in-place sort needs only  $O(1)$  memory beyond the items being sorted, sometime  $O(\log n)$  additional memory is considered in-place.
- **Recursion:** Some algorithm are either recursive or non-recursive, while others may be both such as merge sort.

- **Stable:** A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that items that are equal on the second key, remain sorted on the first key.

## 4.2 INSERTION SORT

Insertion sort is a sorting algorithm that builds the sorted array one entry at a time. Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm.

Sorting is typically performed in place. The resulting array after  $k$  iterations has the property where the first  $k + 1$  entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, the extending the result with each element greater than  $m$  copied to the right as it is compared against  $m$ . The operation of insertion sort on arrays can be described as follows.

Suppose there exists a function called `insert` designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element.

To perform insertion sort, begin at the left most element of the array and invoke `insert` to insert each element encountered into its correct position. The ordered sequence into which each element is sorted at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value; the value being inserted.

### 4.2.1 Algorithm Steps

```

INSERTION SORT (A, n) // n is the length of array A
1. for j ← 2 to n do
2.   key ← A[j]
3.   i ← j - 1
4.   // all elements in array A[1 ..... i] is in sorted order
5.   while i > 0 and A [i] > key
6.     do A [i + 1] = A[i]
7.     i = i - 1
8.   A[i + 1] = key

```

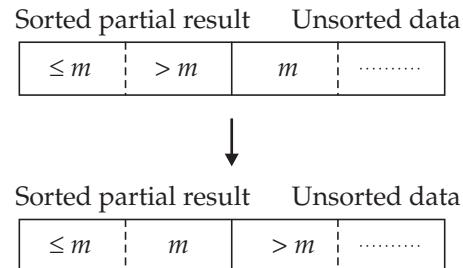


Fig. 4.1. Sorting through insertion sort on array.

#### 4.2.2 Performance Analysis

INSERTION SORT (A, n)	Cost	Time
1. for $j \leftarrow 2$ to $n$ do	$c_1$	$n$
2. key $\leftarrow A[j]$	$c_1$	$n - 1$
3. $i \leftarrow j - 1$	$c_3$	$n - 1$
4. // all elements in array A [1 ..... i] is in sorted order	0	$n - 1$
5. while $i > 0$ and $A[i] > \text{key}$	$c_5$	$\sum_{j=2}^n t_j$
6. do $A[i + 1] = A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7. $i = i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8. $A[i + 1] = \text{key}$	$c_8$	$n - 1$

Note:  $t_j$  is the number of times the while loop test is executed for  $j$ .

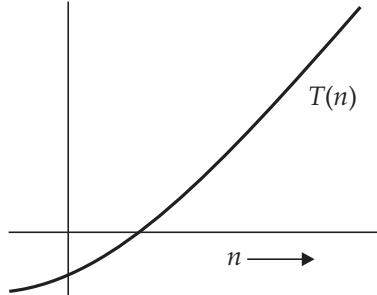
$$\begin{aligned}
 \text{So overall } T(n) &= c_1n + c_2(n-1) + c_3(n-1) + 0 \cdot (n-1) + c_5 \sum_{j=2}^n t_j \\
 &\quad + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1) \\
 &= c_1n + c_2(n-1) + c_3(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\
 &\quad + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)
 \end{aligned}$$

#### Best Case Running Time

Best case arises when all the elements is in sorted order. In this case step 6 and 7 of insertion sort algorithm never execute. Here  $t_j = 1$ .

$$\begin{aligned}
 \text{Then } T(n) &= c_1(n) + c_2(n-1) + c_3(n-1) + c_5(n-1) + c_8(n-1) \\
 &= [c_1 + c_2 + c_3 + c_5 + c_8] + n[-c_2 - c_3 - c_5 - c_8] \\
 &= an + b \\
 &\quad (\text{where } a = [c_1 + c_2 + c_3 + c_5 + c_8] \text{ and } b = [-c_2 - c_3 - c_5 - c_8]) \\
 &= \Theta(n)
 \end{aligned}$$

Therefore the running time is a linear function of  $n$ . Here,  $a$  and  $b$  are constants. The function  $T(n) = \theta(n)$  can be represented as shown in Fig. 4.2.



**Fig. 4.2.** Best case running time.

### Worst Case Running Time

In case of worst case running time  $t_j = j$ . This is because we must compare each element  $A[j]$  with each element in the entire sorted array  $A[1] \dots j - 1$ .

$$\text{So } T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_5 \sum_{j=2}^n t + c_6 \sum_{j=2}^n (j-1) + c_7 \sum_{j=2}^n (j-1) + c_8 (n-1) \dots(i)$$

$$\sum_{j=2}^n j = 2 + 3 + 4 + 5 + \dots + n = \frac{n(n+1)}{2} - 1 \quad \dots(ii)$$

$$\sum_{j=2}^n j-1 = 2 + 3 + 4 + \dots + (n-1) = \frac{n(n-1)}{2} \quad \dots(iii)$$

Putting equation (ii) and (iii) in equation (i) we get

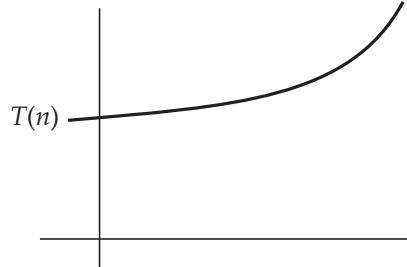
$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_5 \left[ \frac{n(n+1)}{2} - 1 \right] + c_6 \left[ \frac{n(n-1)}{2} \right] + c_7 \left[ \frac{n(n-1)}{2} \right] + c_8(n-1) \\ &= \left[ \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right] n^2 + \left[ c_1 + c_2 + c_3 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right] n + [-c_2 - c_3 - c_5 - c_8] \end{aligned}$$

Now the equation is of the form  $T(n) = an^2 + bn + c$ , but  $a \neq 0$ . Moreover, it is a quadratic equation of degree 2.

It can be represented as shown in Fig. 4.3. Hence,  $T(n) = \theta(n^2)$ .

### Average Case Running Time

Before analyzing the average case running time of insertion sort, we have to assume half of the given



**Fig. 4.3.** Worst case running time.

elements is in sorted order. So, for loop runs for  $j/2$  times. Then the average case running time is calculated as follows.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_5 \sum_{j=2}^n \frac{j}{2} + c_6 \sum_{j=2}^n \left( \frac{j}{2} - 1 \right) + c_7 \sum_{j=2}^n \left( \frac{j}{2} - 1 \right) + c_8(n-1) \\ &= c_1 n + c_2(n-1) + c_3(n-1) + c_5 \left[ \frac{n(n+1)}{2} - 1 \right] + c_6 \left[ \sum_{j=2}^n \left( \frac{j}{2} - 1 \right) \right] \\ &\quad + c_7 \left[ \sum_{j=2}^n \left( \frac{j}{2} - 1 \right) \right] + c_8(n-1) \end{aligned}$$

Here  $\sum_{j=2}^n \left( \frac{j}{2} - 1 \right)$  can be expanded as  $\sum_{j=2}^n \frac{j}{2} - \sum_{j=2}^n 1$  which is equals to  $\frac{1}{2} \left[ \frac{n(n+1)}{2} - 1 \right] - (n-1)$ . Therefore  $T(n)$  becomes

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_5 \left[ \frac{n(n+1)}{2} - 1 \right] \\ &\quad + c_6 \left[ \frac{1}{2} \left( \frac{n(n+1)}{2} - 1 \right) - (n-1) \right] + c_7 \left[ \frac{1}{2} \left( \frac{n(n+1)}{2} - 1 \right) - (n-1) \right] + c_8(n-1) \\ &= \left( \frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4} \right) n^2 + \left( c_1 + c_2 + c_3 + \frac{c_5}{4} - \frac{3c_6}{4} + \frac{3c_7}{4} + c_8 \right) n \\ &\quad + \left( -c_2 - c_3 - c_5 + \frac{c_6}{2} + \frac{c_7}{2} - c_8 \right) \end{aligned}$$

Now the equation for running time of  $T(n)$  is of the form  $T(n) = an^2 + bn + c$ , where  $a$ ,  $b$  and  $c$  are constants. The equation is a quadratic equation of degree 2. Hence, the running time for the average case is same as in case of worst case. So,  $T(n)$  is  $\theta(n^2)$ .

**Example 4.1** Implement insertion sort on the array A given as

$$A = \langle 2, 8, 5, 7, 6, 1, 4, 3 \rangle$$

What is the total running time required for this operation to perform?

**Solution:** The given array A can be written as

	1	2	3	4	5	6	7	8
A:	2	8	5	7	6	1	4	3

Run the insertion sort algorithm for  $j \rightarrow 2$  to 8

**Step 1:** For  $j = 2$

key  $\rightarrow A[2]$  i.e., key = 8

$$i = 2 - 1 = 1$$

	1	2	3	4	5	6	7	8
	2	8	5	7	6	1	4	3

Here while condition doesn't satisfy as  $1 > 0$  is true but  $2 < 8$  is false. So algorithm steps 6 and 7 doesn't execute. Now  $A[i + 1] = A[2] = \text{key}$ .

	1	2	3	4	5	6	7	8
A:	2	8	5	7	6	1	4	3

**Step 2:** For  $j = 3$

key  $\rightarrow A[3]$  i.e., key = 5

$$i = 3 - 1 = 2$$

	1	2	3	4	5	6	7	8
A:	2	8	5	7	6	1	4	3

Here while condition satisfies as  $2 > 0$  and  $8 > 5$  are true. So we interchange  $A[3]$  with  $A[2]$ . Now  $i$  become 1. Again,  $A[i + 1] = A[2] = \text{key}$ . For  $i = 1$  while loop doesn't satisfy, so no change takes place.

	1	2	3	4	5	6	7	8
A:	2	5	8	7	6	1	4	3

**Step 3:** For  $j = 4$

Key  $\leftarrow A[4]$  i.e., key = 7

$$i = 4 - 1 = 3$$

	1	2	3	4	5	6	7	8
A:	2	5	8	7	6	1	4	3

In this case while condition satisfies as  $3 > 0$  and  $8 > 7$  both are true. So interchange  $A[4]$  with  $A[3]$ . Now  $i$  becomes 2. So  $A[i + 1] = A[3] = \text{key}$ . For  $i = 2$  and  $i = 1$ , the while loop doesn't satisfy. Hence, no change takes place in array A.

	1	2	3	4	5	6	7	8
A:	2	5	7	8	6	1	4	3

**Step 4:** For  $j = 5$

key  $\leftarrow A[5]$  i.e., key = 6

$$i = 4$$

	1	2	3	4	5	6	7	8
A:	2	5	7	8	6	1	4	3

In this case while condition satisfies as  $4 > 0$  and  $8 > 6$  both are true. So, interchange  $A[5]$  with  $A[4]$ . Now  $i$  becomes 3. Therefore,  $A[i + 1] = A[4] = \text{key}$ .

A:	1	2	3	4	5	6	7	8
	2	5	7	6	8	1	4	3

For  $i = 3$  the while condition satisfies as  $3 > 0$  and  $7 > 6$ . So, interchange  $A[4]$  and  $A[3]$  as shown in array A.

A:	1	2	3	4	5	6	7	8
	2	5	6	7	8	1	4	3

For  $i = 2$  and  $i = 1$ , the while loop don't satisfy. Hence, there is no change takes place in array A.

**Step 5:** For  $j = 6$

$\text{key} \leftarrow A[6]$  i.e.,  $\text{key} = 1$

$i = 5$

A:	1	2	3	4	5	6	7	8
	2	5	6	7	8	1	4	3

In this case the while condition satisfies as  $5 > 0$  and  $8 > 1$ . So, interchange  $A[6]$  and  $A[5]$ . Now  $i$  becomes 4.  $A[i + 1] = A[5] = \text{key}$ .

A:	1	2	3	4	5	6	7	8
	2	5	6	7	1	8	4	3

Repeat the above operation for  $i = 4, 3, 2$  and 1. Finally, array A becomes

A:	1	2	3	4	5	6	7	8
	1	2	5	6	7	8	4	3

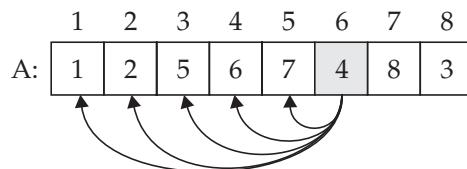
**Step 6:** For  $j = 7$

$\text{key} \leftarrow A[7]$  i.e.,  $\text{key} = 4$

$i = j - 1 = 6$

A:	1	2	5	6	7	8	4	3
	1	2	5	6	7	8	4	3

In this case while condition satisfies as  $6 > 0$  and  $8 > 4$ . So interchange  $A[7]$  and  $A[6]$ . Now, the value of  $i$  becomes 5. Again  $A[i + 1] = A[6] = \text{key}$ .



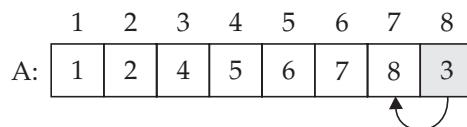
For  $i = 5, 4$  and  $3$  the while loop satisfies. The changes are shown in array A.



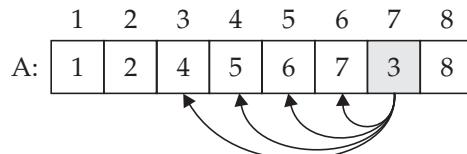
**Step 7:** For  $j = 8$

key  $\rightarrow A[8]$  i.e., key = 3

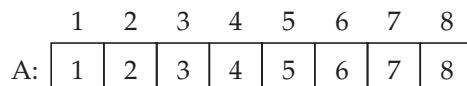
$$i = j - 1 = 7$$



In this case while condition satisfies as  $7 > 0$  and  $8 > 3$ . Therefore, interchange  $A[8]$  with  $A[7]$  values. The value of  $i$  becomes 6. Now,  $A[i + 1] = A[7] = \text{key}$ .



Repeat the above operation for  $i = 6, 5, 4$  and  $3$ . But for  $i = 2$  and  $1$  the while condition doesn't satisfy. At last the sorted array A is given as shown.



**Example 4.2** Rewrite the INSERTION-SORT procedure to sort the elements into decreasing order.

**Solution:**

```
INSERTION-SORT (A, n) // n is the length of the array A
1. for j ← 2 to n do
2.   key ← A[j]
3.   i ← j - 1
4.   // all elements in array A [1 ..... i] is in sorted order
5.   while i > 0 and A [i] < key
6.     do A [i + 1] = A [i]
7.     i = i - 1
8.   A [i + 1] = key
```

**Example 4.3** Write a pseudocode for linear search for the given searching problem.

*Input:* A sequence of  $n$  number  $A = \langle a_1, a_2, \dots, a_n \rangle$  and a value  $V$ .

*Output:* An index  $i$  such that  $V = A[i]$  or special value NIL if  $V$  doesn't appear in  $A$ .

Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

**Solution:**

```
LINEAR-SEARCH (A, V)
Input: A = <a1, a2, ..., an> and a value V.
Output: An index i such that V = A[i] or special value NIL if V doesn't
appear in A.
1. for i ← 1 to n do
2.   if A [i] = V then
3.     return i
4.   end if
5. end for
6. return nil
```

As a loop invariant implies that not a single element at index  $A[1 \dots i-1]$  are equal to  $V$ . Here all properties are satisfied by the loop invariant.

### 4.3 BUBBLE SORT

---

The bubble sort is the oldest and simplest sort in use. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items. This causes larger values to “bubble” to the end of the list while smaller values “sink” towards the beginning of the list. As it only uses comparisons to operate on elements, it is a **comparison sort** and as well as it is in-place and stable.

Now we give the pseudocode of bubble sort algorithm.

#### 4.3.1 Algorithm Steps

```
Bubble Sort (A, n) //n is the length of array A
1. repeat
2.   swapped = FALSE // Repeat until not swapped
3.   for i ← 2 to n
4.     if A [i - 1] > A [i]
5.       swap A [i - 1], A [i]
```

```

6. Swapped = TRUE
7. end if
8. end for

```

**Example 4.4** Sort the following elements of array A using bubble sort.

$$A = \langle 6, 1, 4, 2, 7 \rangle$$

**Solution:** We require three iterations to sort the elements of array A. Mark that elements written within square box are being compared.

First Iteration:

6	1	4	2	7	(Algorithm checks first two elements)
1	6	4	2	7	(Swap since 6 > 1)
1	4	6	2	7	(Swap since 6 > 4)
1	4	2	6	7	(Swap since 6 > 2)
1	4	2	6	7	(As 7 > 6, the elements are already sorted)

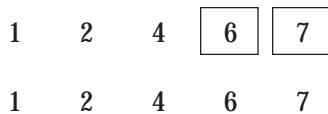
Second Iteration:

1	4	2	6	7	
1	4	2	6	7	
1	2	4	6	7	(Swap since 4 > 2)
1	2	4	6	7	
1	2	4	6	7	

At the end of second iteration we find that the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs another iteration without any swap to know it is sorted.

Third Iteration:

1	2	4	6	7
1	2	4	6	7
1	2	4	6	7



#### 4.3.2 Performance Analysis

The running time of bubble sort can be bound as the total number of comparison being made in the entire execution of the algorithm. Thus, the worst case (input is descending order) comparisons in the respective iterations are as follows:

1<sup>st</sup> →  $n - 1$

2<sup>nd</sup> →  $n - 2$

.....

( $n - 1$ )<sup>th</sup> iteration → 1

Therefore the total number of comparisons =  $n(n - 1)/2$ ; which implies  $O(n^2)$  time complexity (where  $n$  is the number of elements being sorted). This is also same as the average case complexity. Performance of bubble sort over an already – sorted list (best case) is  $O(n)$ .

There exist many sorting algorithm with substantially better worst case or average case complexity of  $O(n \log n)$ . Even other  $O(n^2)$  sorting algorithms such as previously discussed insertion sort, tend to have better performance than bubble sort. Therefore for larger value of  $n$  bubble sort is not useful.

---

#### 4.4 SELECTION SORT

Selection sort is another comparison based sorting algorithm similar to insertion sort. Like insertion sort it is in-place i.e., it requires a constant amount  $O(1)$  of additional memory space. The algorithm divides the input list into two parts: the sublist of item already sorted, which is build up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

At the first iteration the algorithm selects the smallest element in the array and swaps it with the first element. In the second iteration it selects the second smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second largest element and swaps it with the second last index, leaving the largest element in the last index.

The pseudocode of selection sort algorithm is given as follows:

#### 4.4.1 Algorithm Steps

```

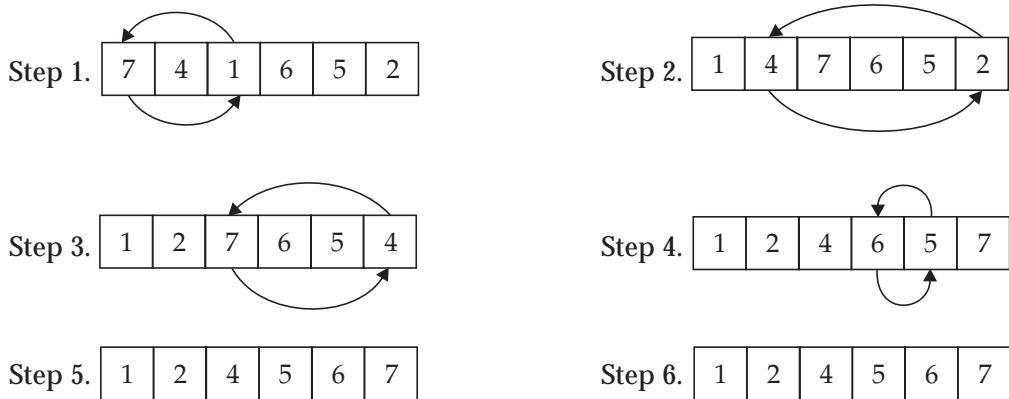
SELECTION SORT (A, n) //n is the length of array A
1. for j ← 1 to n
2. min ← A[j]
3. for i ← j + 1 to n
4. do if min > A [i]
5. then min ← A [i]
6. swap (A[j], A[i])

```

**Example 4.5** Sort the following elements of array A using selection sort.

$$A = \langle 7, 4, 1, 6, 5, 2 \rangle$$

**Solution:** The selection sort sorts the array A in the following way.



#### 4.4.2 Performance Analysis

Like other sorting algorithms none of the loops depend in the selection sort depend on data in the array. The running time can be bound as the total number of comparison being made in the entire execution of this algorithm. From the previous example, it is clear that to select the lowest element in the array of  $n$  elements we require  $n - 1$  comparisons. Similarly, to select the second lowest element we require at most  $n - 2$  comparisons and so on. So the total running time of selection sort becomes

$$(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1)/2 \in \Theta(n^2).$$

## 4.5 COUNTING SORT

Counting sort is a simplest non-comparison based integer sorting algorithm. This algorithm is used to sort data (*i.e.*, elements) while range is pre-specified and multiple occurrences of data

are encountered. The essential requirement is that the range of the data set from which the elements to be sorted are drawn, is small compared to the size of the data set.

For example, suppose that we are sorting elements drawn from  $\{0, 1, \dots, m-1\}$  i.e., the set of integers in the interval  $[0, m-1]$ . Then the sort uses  $m$  counters. The  $i$ th counter keeps track of the number of occurrences of the  $i$ th element of the universe.

#### 4.5.1 Algorithm Steps

```

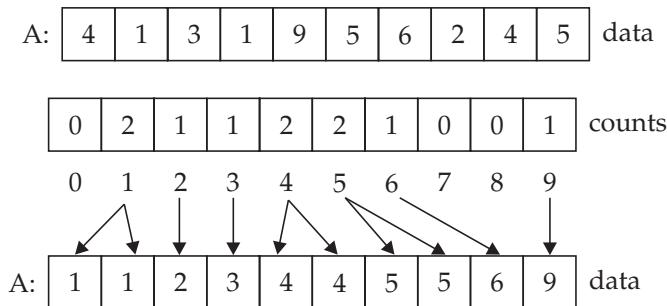
Counting Sort (A, n) //n is the length of array A
1. Elements count [n];
2. for (i = 0; i < n; i++)
3.   count [i] = 0
4. for (j = 0; j < n; j++)
5.   ++count [array [j]]
6. for (i = 0, j = 0; i < n; i++)
7.   for ( ; count [i] > 0; .... count [i])
8.     array [j++] = i

```

**Example 4.6** Sort the following elements of array A using counting sort.

$$A = \langle 4, 1, 3, 1, 9, 5, 6, 2, 4, 5 \rangle$$

**Solution:** We sort the elements in the following way



#### 4.5.2 Performance Analysis

The counting sort runs in  $O(n + m)$  time complexity; where  $n$  is the number of elements in the given array and  $m$  is the range of the elements, which implies that the most efficient use will be when  $m \ll n$  (i.e.,  $m$  is strictly less than  $n$ ). In that case the time complexity will turn out to be linear. Moreover, the sort works optimally in the case when the data is uniformly distributed.

If the range  $m \gg n$ , the complexity will not be linear in  $n$  and thus this sort doesn't remain useful anymore. This is because chances of introduction of gaps, that is counters for these elements which don't exist in the list, will cause a higher space complexity.

## 4.6 RADIX SORT

---

Radix sort is a non-comparison based algorithm that can rearrange integer representations based on processing of individual digits in such a way that the integer representations are eventually in either ascending or descending order. Integer representations can be used to represent things such as characters (name of people, words, places, characters, dates etc.) and floating point numbers as well as integers. So anything which can be represented as an ordered sequence of integer representations can be rearranged to be in order by radix sort.

Most digital computers internally represent all of their data as electronic representations of binary numbers, so processing the digits of integer representations by groups of binary digit representations is most convenient. Two classifications of radix sorts are:

1. Least Significant Digit (LSD) radix sorts and
2. Most Significant Digit (MSD) radix sorts

LSD radix sorts process the integer representations starting from the least significant digit and move the processing towards the most significant digit.

MSD radix sorts process the integer representations starting from the most significant digit and move the processing towards the least significant digit.

The integer representations that are processed by sorting algorithms are often called “keys” which can exist all by themselves or be associated with other data. LSD radix sorts typically use the following sorting order: short keys come before longer keys, and keys of the same length are sorted lexicographically. This coincides with the normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

MSD radix sorts use lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations. A sequence such as *b, c, d, e, f, g, h, i, j, ba* would be lexicographically sorted as *b, ba, c, d, e, f, g, h, i, j*.

If lexicographic ordering is used to sort variable-length integer representations, the representation of numbers from 1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9 as if the shorter keys were left-adjusted and padded on the right with blank characters to make the shorter keys as long as the longer key for the purpose of determining sorted order.

### 4.6.1 Algorithm Steps

```
Radix Sort (A, d)
1. for i ← 1 to d
2. do use any stable sort to sort array A on digit i. (counting sort will
   do the job)
```

The running time of the radix sort depends on the stable sort used as an intermediate stage of the sorting algorithm. When each digit is in the range 1 to *k*, and *k* is not too large, counting sort can be taken into account. In case of counting sort each pass over *n* *d*-digit

numbers take  $O(n + k)$  time. There are  $d$  iterations, so the total time complexity of radix sort becomes  $\theta(d(n + k))$ . When  $d$  is a constant and  $k = \theta(n)$ , the radix sort runs in linear time.

**Example 4.7** Consider the list of 4-bit binary numbers as given: 1001, 0010, 1101, 0001, 1110. Sort these binary numbers using LSD radix sort.

**Solution:** We perform the following steps to sort the binary numbers as follows:

**Step 1.** First arrange the list of numbers according to the least significant bit. The sorted list is given by:

0010, 1110, 1001, 1101, 0001

**Step 2.** Then arrange the list of numbers according to the next significant bit. The sorted list is given by:

1001, 1101, 0001, 1010, 1110

**Step 3.** Then arrange the list of numbers according to the next significant bit. The sorted list is given by:

1001, 0001, 0010, 1101, 1110

**Step 4.** Then arrange the list of numbers according to the most significant bit. The final sorted list is given by:

0001, 0010, 1001, 1101, 1110

The above operations can be observed by Fig. 4.4.

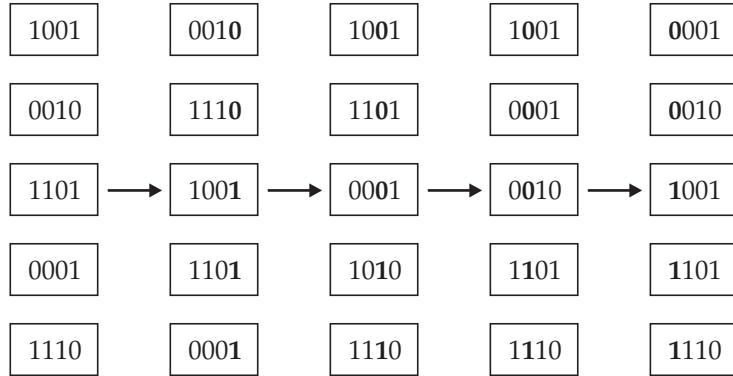


Fig. 4.4. Sorting the sequence of 4-bit integers.

**Example 4.8** Considering the most significant bit 1st, sort the following list of number using MSD radix sort.

178, 45, 76, 97, 2, 24, 809, 66

**Solution:** We perform the following steps to sort the given list of numbers.

**Step 1.** Sort the list by most significant digit. It gives

045, 076, 097, 002, 024, 066, 178, 809

**Step 2.** Then sort the list by the next significant digit.

002, 802, 024, 045, 066, 076, 170, 090

**Step 3.** At last sort the list by least significant digit.

002, 024, 045, 066, 076, 097, 178, 809

#### 4.6.2 Limitations of Radix Sort

Speed of radix sort largely depends on the inner basic operations and if operations are not efficient enough then radix sort can be slower than some other algorithms such as quick sort or merge sort. These operations include the insert, delete function of the sublists and the process of isolating the digit we want.

Radix sort can also take up more space than other sorting algorithms, since in addition to the array that will be sorted; we need to have a sublist for each of the possible digits or letters. If pure English words are sorted then atleast 26 sublist are needed and if alpha numeric words are sorted then probably more than 40 sublists are required. Therefore, radix sort is less flexible as compared to the any other sorts.

## 4.7 BUCKET SORT

---

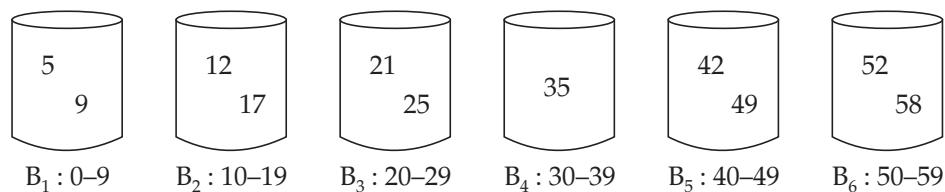
Bucket sort (also called bin sort) is a non-comparison based sorting algorithms that works by partitioning array into a number of buckets. Each bucket is then sorted individually using a insertion sort (or any other sort). The pseudocode of bucket sort is given as follows.

#### 4.7.1 Algorithm Steps

```

Bucket Sort (A, n) // n is the length of array A
1. for i = 1 to n do
2.   insert A[i] into bucket B[A[i]/b] // b is the bucket size
3. for i = 1 to n do
4.   sort list B with insertion sort
5. concatenate the lists B[1], B[2], . . . . . . , B[n]
    
```

**Example 4.9** A simple illustration of bucket sort can be observed in the following way. The given array of elements  $A = <5, 25, 35, 21, 49, 42, 9, 52, 58>$ . The bucket  $B_1$  carries the elements in the range 0-9, where as bucket  $B_2$  carries 10-19,  $B_3$  carries 20-29 and so on. We sort the elements within each bucket in the following. (Fig. 4.5)



**Fig. 4.5.** Elements are sorted within each bucket.

#### 4.7.2 Complexity Analysis

In bucket sort algorithm we assume three things.

- The data lies in the range R with  $d$  digits which is of the order of  $n$  or lesser.
- The data is more or less uniformly distributed i.e., empty buckets are rare.
- The data is discrete for integers, a mixture of integers and numbers with decimals would again not be feasible as it would increase the number of digits.
- The time complexity of bucket sort is  $O(nd)$ . This is because the comparisons are being done  $n$  times at each of the  $d$  levels that the data has to pass before getting sorted.

---

#### CHAPTER NOTES

- In a comparison based sorting algorithm we sort data by comparing two elements with a comparison operator where as in case of a non-comparison based sorting we don't perform this.
- Comparison based sorting includes insertion sort, bubble sort, quick sort, merge sort, selection sort etc.
- Non-comparison based sorting includes counting sort, radix sort, bucket sort etc.
- In insertion sort we insert each item into its proper place in the final list. The simplest implementation of this requires two list structures—the source list and the list into which sorted items are inserted. Its worst case, best case and average case running times are  $\Theta(n^2)$ ,  $\Theta(n)$  and  $\Theta(n^2)$  respectively.
- In bubble sort we compare each item in the list with the item next to it and swapping them if required. This results larger values to bubble to the end of the list while smaller values sink towards the end of the list. Generally it takes  $\Theta(n^2)$  sorting time.
- The selection sort takes  $\Theta(n^2)$  time complexity to sort the array. It works by selecting the smallest element in the array and swap it with the first element, then it selects the second smallest and swap it with the second element and so on.
- Counting sort is used to sort data where multiple occurrences of data are encountered and the range ( $m$ ) is pre-specified. It runs in  $O(n + m)$  time complexity.
- Radix sort is based on processing individual digits of integer representation either by least significant order or by most significant order.
- Bucket sort is applied by partitioning array into a number of bucket and then each bucket is then sorted individually using a insertion sort.

## EXERCISES

---

1. Sort the array A = < 38, 45, 79, 13, 74, 36 > using insertion sort. What is the running time of doing this operation?
2. What is the insertion sort algorithm ? Analyze its worst case, best case and average case running time.
3. Sort the elements of array A = < 5, 9, 11, 6, 12 > using bubble sort.
4. Write the bubble sort algorithm. Find its time complexity.
5. Write the selection sort algorithm. Sort the elements < 15, 19, 6, 20, 11, 25 > using selection sort.
6. Give an example to explain counting sort. Analyze its performance.
7. Write the counting sort algorithm. Sort the elements < 2, 5, 9, 5, 6, 10, 7, 5, 4, 2, 7, 8 > using counting sort.
8. What are LSD and MSD in radix sort? Give examples to explain each.
9. Apply radix sort by considering least significant bit on the following decimal numbers.  
523, 153, 088, 554, 235
10. Write the bucket sort algorithm. Sort the array A having elements < 27, 37, 17, 07, 47, 9, 10, 20, 50 > among five buckets  
B<sub>1</sub>: 0–10, B<sub>2</sub>: 11–20, B<sub>3</sub>: 21–30, B<sub>4</sub>: 31–40 and B<sub>5</sub>: 41–50.

# DIVIDE AND CONQUER APPROACH

5

CHAPTER

## OBJECTIVES OF LEARNING

After going through this chapter, the reader would be able to understand:

- The technique of “divide and conquer” in the context of merge sort
- Advantages and disadvantages of merge sort
- Simple algorithm for matrix multiplication
- Strassen’s method for matrix multiplication

# Chapter 5 DIVIDE AND CONQUER APPROACH

## INSIDE THIS CHAPTER

- |   |                                 |
|---|---------------------------------|
| 5.1 Introduction                          | 5.2 Merge Sort                  |
| 5.3 An $n \log n$ Lower Bound for Sorting | 5.4 Pros and Cons of Merge Sort |
| 5.5 Strassen's Matrix Multiplication      |                                 |

## 5.1 INTRODUCTION

Many useful algorithms are recursive in structure to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide and conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively and then combine these solutions to create a solution to the original problem.

The divide and conquer paradigm involves three steps at each level of the recursion.

**Divide** the problem into a number of subproblems.

**Conquer** the subproblems by solving them recursively through any one of the three methods *i.e.*, substitution methods, recursion tree method and master method. In chapter 6, we will study these techniques in detail.

**Combine** the solutions to the subproblems into the solution for original problem.

## 5.2 MERGE SORT

The merge sort algorithm closely follows the divide and conquer paradigm. Intuitively it operates as follows:

**Divide:** Divide the  $n$  element sequence to be sorted into two subsequences of  $n/2$  elements each.

**Conquer:** Sort the two sequences recursively using merge sort.

**Combine:** Merge the two sorted subsequences to produce the sorted answer.

We note that the recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. To perform the merging we use an auxiliary procedure MERGE ( $A, p, q, r$ ) where  $A$  is an array and  $p, q$  and  $r$  are indices numbering elements of the array such that  $p \leq q \leq r$ . The procedure assumes that the subarrays  $A[p \dots q]$  and  $A[q + 1 \dots r]$  are in the current subarray  $A[p \dots r]$ .

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT ( $A, p, r$ ) sorts the elements in the subarray  $A[p \dots r]$ . If  $p \geq r$ , the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index  $q$  that partitions  $A[p \dots r]$  into two subarrays:  $A[p \dots q]$ , containing  $\lceil n/2 \rceil$  elements, and  $A[q + 1 \dots r]$ , containing  $\lfloor n/2 \rfloor$  elements.

### 5.2.1 Merge-sort Algorithm Steps

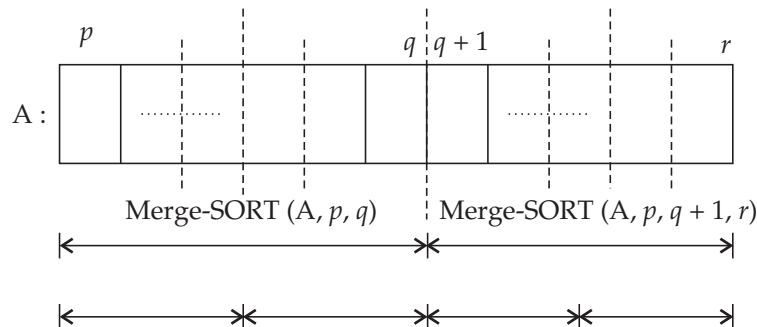


Fig. 5.1. Merge-sort operation.

```

Input: An array A and indices p and r.
Output: An sorted array A.
MERGE-SORT (A, p, r)
1. if p < r
2. then q = ⌊(p + r)/2⌋
3. MERGE-SORT (A, p, q)
4. MERGE-SORT (A, q + 1, r )
5. MERGE (A, p, q, r)

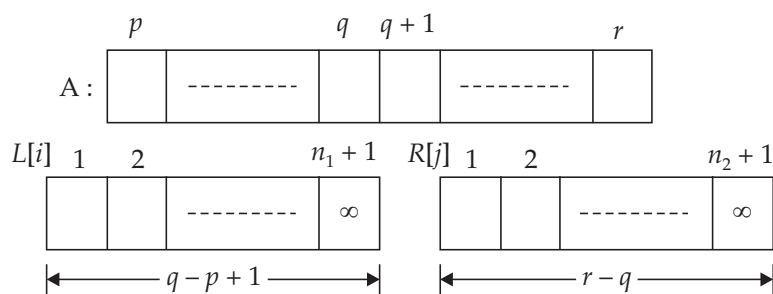
```

**MERGE (A, p, q, r)**

```

1.  $n_1 = q - p + 1$  //  $n_1$  is the length of 1st subproblem.
2.  $n_2 = r - q$  //  $n_2$  is the length of the 2nd subproblem.
3. Construct two new array i.e., L[1 ...  $n_1 + 1$ ] and R[1 ...  $n_2 + 1$ ]
4. for  $i = 1$  to  $n_1$ 
5. L[i] = A[p + i - 1] // Inserting elements from array A to L[i]
6. for  $j = 1$  to  $n_2$ 
7. R[j] = A[q + j] // Inserting elements from array A to R[j].
8. L[ $n_1 + 1$ ] =  $\infty$  //  $\infty$  is a very large value called sentinel value, which
   acts as a boundary denoting all elements before it in array are
   smallest.
9. R[ $n_2 + 1$ ] =  $\infty$ 
10. i = 1
11. j = 1
12. for  $k = p$  to  $r$ 
13. if L[i]  $\leq$  R[j]
14. A[k] = L[j]
15. i = i + 1
16. else A[k] = R[j]
17. j = j + 1

```

**Fig. 5.2.** Merging two arrays of elements.**5.2.2 Algorithm Analysis**

We are now going to analyze the worst case running time of merge sort through the use of Master theorem.

**Note:** Master theorem has discussed in chapter 6 in detail.

If there are only one element (i.e.,  $n = 1$ ) in the array then the merge sort takes constant time i.e.,  $\theta(1)$ . When we have more than one element in the array (i.e.,  $n > 1$ ), we break the running time as follows.

**Divide:** The divide step just computes the middle of the subarray, which takes constant time. So  $T(n) = \theta(1)$

**Conquer:** We recursively solve two subproblems, each having the size  $n/2$ , which gives  $2T(n/2)$  to the running time.

**Combine:** Merge-sort on  $n$ -element subarray takes  $\theta(n)$  time i.e.,  $T(n) = \theta(n)$

Now combining the above 3 steps we will define the worst case running time of merge-sort as follows

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

We can rewrite the above recurrence as

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

Where  $c$  is the constant which represents to solve problems of size 1 as well as the time per array element of the divide and combine steps.

$$\text{So } T(n) = 2T(n/2) + cn$$

It is of the form of  $T(n) = aT(n/b) + f(n)$ ; Where  $a = 2$ ,  $b = 2$ ,  $f(n) = cn \equiv cn (\log n)^0$  (So,  $k = 0$ )

$$\text{Hence } n^{\log_b a} = n^{\log_2 2} = n$$

By applying case-II of Master theorem we get

$$T(n) = (n^{\log_b a} \log n) = \theta(n \log n)$$

So from the above analysis we get that the worst case running time of merge-sort is  $\theta(n \log n)$ .

We can also get the worst case running time by using recursion tree method, which has discussed in chapter – 6 under the section recursion tree method (solved examples).

### 5.3 AN $n \log n$ LOWER BOUND FOR SORTING

---

In this section, we discuss the notion of lower bounds for problem of sorting. We show any comparison based sorting algorithm must take  $\Omega(n \log n)$  time sort an array of  $n$  elements in the worst case. But before that here a question might be murmuring in your mind: What is lower bound? What does this signify?

Lower bounds help us to understand how close we are to the best possible solution to some problem: e.g., if we have an algorithm that runs in time  $O(n \log^2 n)$  and a lower bound of  $\Omega(n \log n)$ , the we have a  $\log(n)$  gap: the maximum possible saving we could hope to achieve by improving our algorithm. Therefore any algorithm that takes  $\Omega(g(n))$  time to solve any problem  $P$  then our goal should be  $g(n)$  as large as possible. Now we discuss the following theorem.

**Theorem 5.1:** Any comparison based sorting algorithm must perform  $\Omega(n \log n)$  comparisons to sort  $n$  elements in the worst case.

**Proof:** The sorting algorithm must output a permutation of the input  $[a_1 a_2 \dots a_n]$ . The key to the argument is that (a) there are  $n!$  different possible permutations the algorithm might output, and (b) for each of these permutations, there exists an input for which that permutation is the only correct answer.

Given (a) and (b) above, this means we can fix some set of  $n!$  inputs (e.g., all ordering of  $\{1, 2, \dots, n\}$ ), one for each of the  $n!$  output permutations.

Let  $S$  be the set of these inputs that are consistent with the answers to all comparisons made so far (so, initially,  $|S| = n!$ ). We can think of a new comparison as splitting  $S$  into two groups: those inputs for which the answer would be YES and those for which the answer would be NO.

Now, suppose an adversary always gives the answer to each comparison corresponding to the larger group. The each comparison will cut down the size of  $S$  by at most a factor of 2. Since  $S$  initially has size  $n!$  and by construction, the algorithm at the end must have reduced  $|S|$  down to 1 in order to know which output to produce, the algorithm must take  $\log_2(n!)$  comparisons before it can half. We can then solve

$$\log_2(n!) = \log_2(n) + \log_2(n-1) + \dots + \log_2(2) = \Omega(n \log n).$$

Lets take an example with  $n = 3$  and  $S$  as initially consisting of the 6 possible orderings of  $\{1, 2, 3\}$ :

(123), (132), (213), (231), (312), (321)

Suppose the sorting algorithm initially compares the first two elements  $a_1$  and  $a_2$ . Half of the possibilities have  $a_1 > a_2$  and half have  $a_2 > a_1$ . So the adversary can answer either way and let's say it answers that  $a_2 > a_1$ . This produce the three possibilities.

(123), (132), (231)

Suppose the next comparison is between  $a_2$  and  $a_3$ . In this case, the most popular answer is that  $a_2 > a_3$ , so the adversary returns that answer which remove just one ordering, leaving the answer with:

(132), (231)

It takes one more comparison to produce the output in correct order.

## 5.4 PROS AND CONS OF MERGE SORT

---

### Pros (Advantages)

- ⇒ Marginally faster than heap sort for larger database.
- ⇒ Merge sort always perform lesser number of comparisons than quick sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.

⇒ Merge sort is often the best choice for sorting a linked list because the slow random access performance of a linked list makes some other algorithms such as quick sort perform poorly and others (such as heap sort) completely impossible.

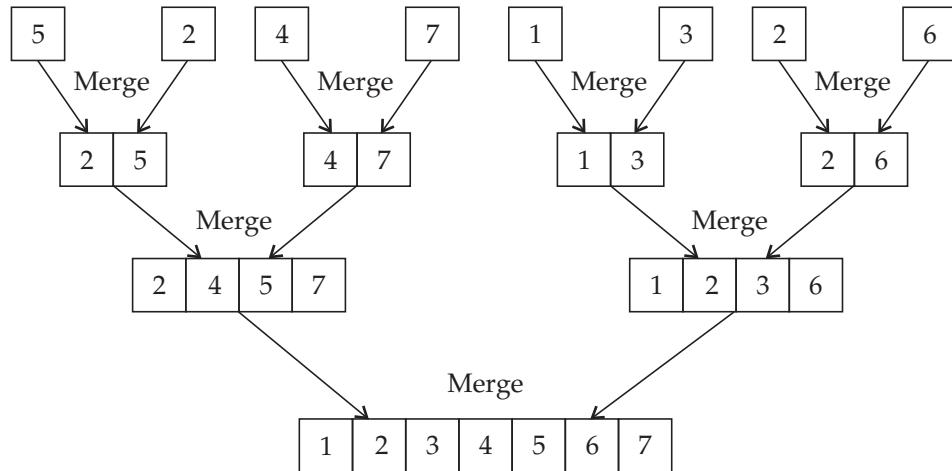
### Cons (Disadvantages)

- ⇒ At least twice the memory requirements of the other sorts because it is recursive. This is the BIGGEST cause for concern as its space complexity is very high. It requires about  $\Theta(n)$  auxiliary space for its working.
- ⇒ Function overhead calls ( $2n - 1$ ) are much more than those for quick sort. This causes it to take more time marginally to sort the input data.

**Example 5.1** Illustrate the operation of merge-sort on the array  $A = <5, 2, 4, 7, 1, 3, 2, 6>$

Find the time complexity of the entire operation to perform and number of comparisons needed.

**Solution:**



$$\text{Time complexity } T(n) = \Theta(n \log n)$$

Number of comparisons = 7

**Example 5.2** Illustrate the procedure MERGE ( $A, p, q, r$ ) on array  $A$  given as  $A : <10, 20, 30, 40, 5, 11, 25, 32>$  where  $p = 1, q = 4$  and  $r = 8$ . Find the time complexity of this operation.

**Solution:**

	1	2	3	4	5	6	7	8
<b>Step 1.</b> A :	10	20	30	40	5	11	25	32

$p$                      $q$      $q + 1$                      $r$

$$n_1 = q - p + 1 = 4 - 1 + 1 = 4$$

$$n_2 = r - q = 8 - 4 = 4$$

**Step 2.** Construct two new arrays i.e., L and R

	1	2	3	4	5
L :	10	20	30	40	$\infty$
R :	5	11	25	32	$\infty$

**Step 3.**

	1	2	3	4	5				
L :	10	20	30	40	$\infty$				
R :	5	11	25	32	$\infty$				
A[k] :	5								

**Step 4.**

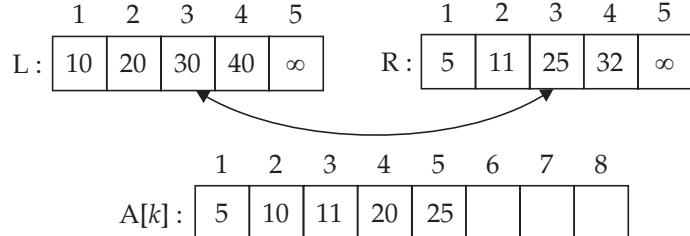
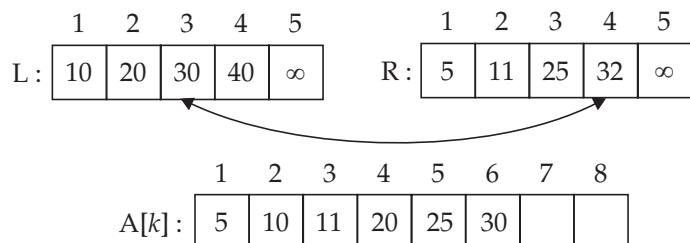
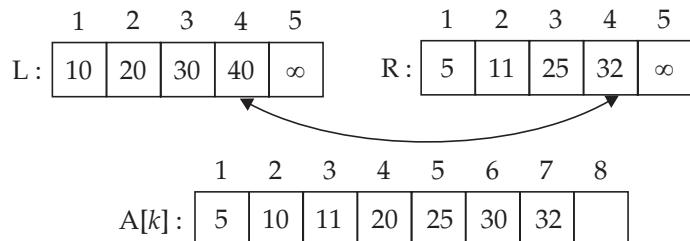
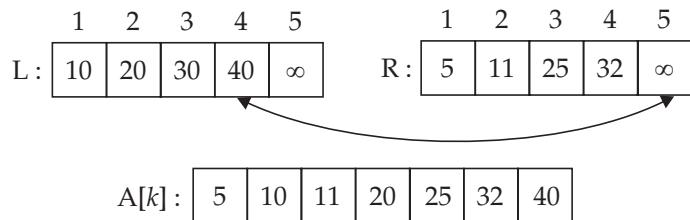
	1	2	3	4	5				
L :	10	20	30	40	$\infty$				
R :	5	11	25	32	$\infty$				
A[k] :	5	10							

**Step 5.**

	1	2	3	4	5				
L :	10	20	30	40	$\infty$				
R :	5	11	25	32	$\infty$				
A[k] :	5	10	11						

**Step 6.**

	1	2	3	4	5				
L :	10	20	30	40	$\infty$				
R :	5	11	25	32	$\infty$				
A[k] :	5	10	11	20					

**Step 7.****Step 8.****Step 9.****Step 10.**

Time complexity  $T(n) = \theta(n \log n)$ , where  $n$  represents the number of elements.

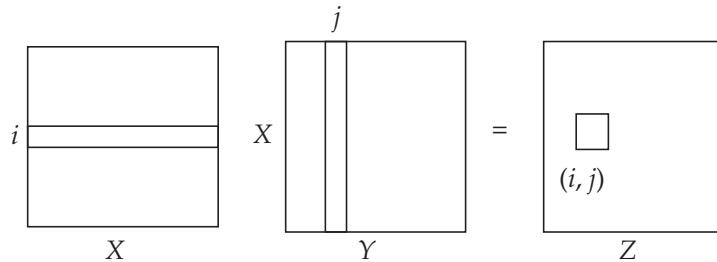
## 5.5 STRASSEN'S MATRIX MULTIPLICATION

---

The product of two  $n \times n$  matrices X and Y is a third  $n \times n$  matrix Z = XY, with  $(i, j)$ th entry

$$Z_{ij} = \sum_{k=1}^n X_{ik} Y_{kj}$$

To make it more visual,  $Z_{ij}$  is the dot product of the  $i$ th row of  $X$  with the  $j$ th column of  $Y$



**Fig. 5.4.** Simple matrix multiplication.

In general,  $XY$  is not the same as  $YX$ ; matrix multiplication is not commutative.

The preceding formula implies an  $O(n^3)$  algorithm for matrix multiplication: there are  $n^2$  entries to be computed and each takes  $O(n)$  time. For quite a while, this was widely believed to be the best running time possible, and it was even proved that in certain models of computation no algorithm could do better. It was therefore a source of great excitement when in 1969, the German mathematician Volker Strassen announced a significantly more efficient algorithm, based on divide and conquer.

### 5.5.1 A Simple Algorithm for Matrix Multiplication

```
SQUARE-MATRIX-MULTIPLY (X, Y)
1. n = rows
2. let Z be a new n × n matrix
3. for i = 1 to n
4.   for j = 1 to n
5.     Zij = 0
6.     for k = 1 to n
7.       Zij = Zij + Xik · Ykj
8. return Z.
```

Square-matrix-Multiply algorithm takes  $O(n^3)$  time.

### 5.5.2 Strassen's Procedure for Matrix Multiplication

Matrix multiplication is particularly easy to break into subproblems, because it can be performed blockwise. To see what this means, carve  $X$  into four  $n/2 \times n/2$  blocks and also  $Y$ .

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Then their product can be expressed in terms of these blocks and is exactly as if the blocks were single elements.

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

We now have a divide and conquer strategy; to compute the size- $n$  product  $XY$ , respectively compute eight size  $-n/2$  products  $AE, BG, AF, BH, CE, DG, CF, DH$  and then do a few  $O(n^2)$  time additions. The total running time is described by the recurrence relation  $T(n) = 8T(n/2) + O(n^2)$

This comes out to an unimpressive  $O(n^3)$ , the same for the default algorithm. But the efficiency can be further improved, and as with integer multiplication the key is some clever algebra.

It turns out  $XY$  can be computed from just seven  $n/2 \times n/2$  subproblems, via a decomposition so tricky and intricate that one wonders how Strassen was ever able to discover it!

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

The new running time is  $T(n) = 7T(n/2) + O(n^2)$ , which by the Master theorem works out to  $O(n^{\log_2 7}) \approx O(n^{2.81})$ . So the running time reduced than which we got from Square-Matrix-Multiply algorithm.

**Example 5.3** Show how the Strassen's algorithm computes the following multiplication?

$$\begin{bmatrix} 3 & 1 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} 2 & -5 \\ 6 & -3 \end{bmatrix}$$

**Solution:** Let  $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 4 & -1 \end{bmatrix}$

$$Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} 2 & -5 \\ 6 & -3 \end{bmatrix}$$

According to Strassen's method

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where  $P_1 = A(F - H) = 3(-5 - (-3)) = 3(-5 + 3) = -6$

$P_2 = (A + B)H = (3 + 1)(-3) = -12$

$$P_3 = (C + D)E = (4 - 1)2 = 6$$

$$P_4 = D(G - E) = (-1)(6 - 2) = -4$$

$$P_5 = (A + D)(E + H) = (3 - 1)(2 - 3) = -2$$

$$P_6 = (B - D)(G + H) = (1 + 1)(6 - 3) = 6$$

$$P_7 = (A - C)(E + F) = (3 - 4)(2 - 5) = 3$$

So,  $XY = \begin{bmatrix} -2 - 4 + 12 + 6 & -6 - 12 \\ 6 - 4 & -6 - 2 - 6 - 3 \end{bmatrix} = \begin{bmatrix} 12 & -18 \\ 2 & -17 \end{bmatrix}$

**Example 5.4** Use Strassen's algorithm to compute the matrix product  $\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$ . Find the

running time of this operation to perform. Compare the running time with Square Matrix-Multiplication operation.

**Solution:** Let  $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix}$

and  $Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 4 & 2 \end{bmatrix}$

According to Strassen's method

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Where,

$$P_1 = A(F - H) = 1(8 - 2) = 6$$

$$P_2 = (A + B)H = (1 + 3)2 = 8$$

$$P_3 = (C + D)E = (7 + 5)6 = 72$$

$$P_4 = D(G - E) = 5(4 - 6) = -10$$

$$P_5 = (A + D)(E + H) = (1 + 5)(6 + 2) = 48$$

$$P_6 = (B - D)(G + H) = (3 - 5)(4 + 2) = -12$$

$$P_7 = (A - C)(E + F) = (1 - 7)(6 + 8) = -84$$

So,  $XY = \begin{bmatrix} 48 - 10 - 8 - 12 & 6 + 8 \\ 72 - 10 & 6 + 48 - 72 + 84 \end{bmatrix} = \begin{bmatrix} 18 & 14 \\ 62 & 66 \end{bmatrix}$

Running time of Strasson's matrix multiplication =  $O(n^{2.81})$

But the running time of Square-Matrix-Multiply =  $O(n^3)$ .

Clearly,  $O(n^3) > O(n^{2.81})$ . So Strassen's algorithm asymptotically better than Square-Matrix-Multiply.

## CHAPTER NOTES

---

- Divide-and-conquer as a technique for designing algorithms dated back to at least 1962 in an article by Karatsuba and Ofman. However C.F. Gauss devised the fast Fourier transform algorithm in 1805 and Gauss's formulation breaks the problem into smaller subproblems whose solutions are combined.
- Mergesort works recursively. It is a classical divide-and-conquer algorithm. The array is split into two subarrays of roughly equal size. They are sorted recursively. Then the two sorted subarrays are merged together in  $\Theta(n)$  time.
- Mergesort is a stable sorting algorithm. The downside of the mergesort is that it requires additional array storage. This is because the merging process merges the two arrays into a third array. Although it is possible to merge arrays in place, it cannot be done in  $\Theta(n)$  time.
- Strassen's algorithm asymptotically better than Square-Matrix-Multiply.
- From a practical point of view. Strassen's algorithm is often not the method of choice for matrix multiplication for four reasons.
  - (1) The constant factor hidden in  $O(n^{2.81})$  running time of Strassen's algorithm is larger than constant factor in the  $O(n^3)$  time Square-Matrix-Multiply procedure.
  - (2) When the matrices are sparse.
  - (3) Strassen's algorithm is not quite as numerically stable as Square-Matrix-Multiply.
  - (4) The submatrices formed at the levels of recursion consume space.

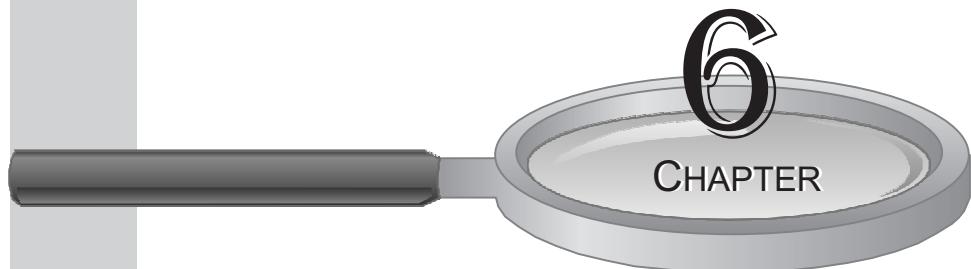
## EXERCISES

---

1. Briefly explain how divide, conquer and combine are applied to Merge-sort.
2. Write a recursive algorithm for merge-sort. Show the running time is  $O(n \log n)$ . Can we say the time for merge-sort is  $\Theta(\log n)$  ?
3. Prove how the worst case running time of merge-sort algorithm is  $O(n \log n)$ .
4. Prove that merge-sort is optimal.
5. Provide an algorithm for multiplication of  $n \times n$  matrix.
6. Write the pseudocode for Strassen's algorithm. Show that how strassen's algorithm asymptotically better than Square-Matrix-Multiply.
7. Show how the Strassen's algorithm computes the following multiplication.

$$\begin{bmatrix} 4 & -2 \\ 3 & -5 \end{bmatrix} \begin{bmatrix} 7 & -1 \\ -4 & 3 \end{bmatrix}$$

# RECURRENCES



## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- Method for calculating asymptotic efficiency of recurrences
  - Substitution method
  - Recursion tree method
  - Master method
- Changing variables
- Limitations (gaps) of master method
- Proof of master theorem

# Chapter 6 RECURRENCES

## INSIDE THIS CHAPTER

- |                             |                         |
|-----------------------------|-------------------------|
| 6.1 Introduction            | 6.2 Substitution Method |
| 6.3 Recursion - Tree Method | 6.4 Master Theorem      |

## 6.1 INTRODUCTION

When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A *recurrence* is an equation or inequality that describes a function in terms of its value on smaller inputs. This can be achieved by using any one of the three methods. These methods are:

**Substitution Method:** Here we guess a bound and then use mathematical induction to prove our guess is correct.

**Recursion Tree Method:** In this method, we convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion.

**Master Theorem:** This method provides bounds for recurrences of the form

$$T(n) = a T(n/b) - f(n)$$

where  $a \geq 1$ ,  $b > 1$  and  $f(n)$  is a given function.

Lets now discuss these methods one by one.

## 6.2 SUBSTITUTION METHOD

This substitution method is the ultimate method of guess and confirm. That means we first guess the solution and then prove it using mathematical induction that the solution works. This method is powerful, but it obviously can be applied only in case when it is easy to guess the form of the solution.

Later parts of these section describe techniques to generate guesses that are guaranteed to be correct, provided you use them correctly. Even if you guess slightly wrong but the guess

is very much close to the solution then you try to verify your guess inductively, usually either the proof will succeed, in which case you are done or the proof will fail, in which case the failure will help you refine your guess.

### 6.2.1 Pitfalls of Substitution Method

The two pitfalls of substitution method include:

1. Guess the solution: This can be achieved by experience and creativity or by using some sort of heuristics.
2. Use mathematical induction: We use this to find the constants and show the solution works.

Let us consider the following recurrence

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

For this recurrence we guess the solution may be  $n \log n + n$ .

Now we apply the method of induction to check whether our solution works correctly or not.

**Basis step:**  $n = 1 \Rightarrow n \log n + n = 1 = T(n)$

**Inductive step:** Inductive hypothesis is that  $T(k) = k \log k + k$  for all  $k < n$ . We will use this induction hypothesis for  $T(n/2)$ .

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2 \log n/2 + n/2) + n && (\text{by using induction hypothesis}) \\ &= n \log n/2 + n + n \\ &= n(\log n - \log 2) + n + n \\ &= n \log n - n + n + n \\ &= n \log n + n \end{aligned}$$

Therefore the solution  $T(n) = n \log n + n$  is correct. Note that the recurrence that we have taken with an exact function rather than asymptotic, and the solution is also exact rather than asymptotic.

But generally we use asymptotic notation for our purpose. Then, we represent the recurrence as  $T(n) = 2T(n/2) + \theta(n)$  and assume  $T(n) = O(1)$  for sufficiently smaller value of  $n$ . On the other hand, we express the solution by asymptotic notation i.e.,  $T(n) = \theta(n \log n)$ . At this stage we don't worry about boundary cases, nor do we show base cases in the substitution proof.

Most importantly, for the substitution method:

- We name the constants in the additive term.
- We also show the upper ( $O$ ) and lower ( $\Omega$ ) bounds separately. To show this we might need to use different constants for each.

For example,  $T(n) = 2T(n/2) + \theta(n)$ . If we want to show an upper bound of  $T(n) = 2T(n/2) + O(n)$ , we write  $T(n) \leq 2T(n/2) + cn$  for some positive constant  $c$ . Then

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn = 2\left(d\frac{n}{2} \log \frac{n}{2}\right) + cn = dn \log \frac{n}{2} + cn \\ &= dn \log n - dn + cn \\ &\leq dn \log n \quad \text{if } -dn + cn \leq 0 \\ d &\geq c \end{aligned}$$

Therefore the upper bound is given as  $T(n) = O(n \log n)$ . For lower bound we write  $T(n) \geq 2T(n/2) + cn$  for positive constant  $c$ . Applying substitution

$$\begin{aligned} T(n) &\geq 2T(n/2) + cn = 2\left(d\frac{n}{2} \log \frac{n}{2}\right) + cn = dn \log \frac{n}{2} + cn = dn \log n - dn + cn \\ &\geq dn \log n \quad \text{if } -dn + cn \geq 0 \\ d &\leq c \end{aligned}$$

Hence  $T(n) = \Omega(n \log n)$ . We have previously seen  $T(n) = O(n \log n)$ .

This implies  $T(n) = \theta(n \log n)$

### 6.2.2 Techniques to Handle Pitfalls

Below the discuss several useful points that will undoubtedly help us to solve recurrences through the method of substitution.

- **Similar Recurrence:** If a recurrence is similar to one that you have previously seen like:  $T(n) = 2T((n/2) + 5) + n$  which is similar to  $T(n) = 2T(n/2) + n$  then we guess the solution as  $O(n \log n)$ .
- **Floor, Ceil and Lower Order Terms:** Floor, ceil and lower order terms don't affect the result of the recurrence. Such as if the recurrence is  $T(n) = 2T(\lceil n/2 \rceil) + n$  then the solution is  $O(n \log n)$ .
- **Avoid Wrong Guess:** Suppose the recurrence is given as  $T(n) = 2T(\lfloor n/2 \rfloor) + n$  and you guess  $T(n) = O(n)$  by guessing  $T(n) \leq cn$  and then you argue  

$$T(n) \leq 2(c\lfloor n/2 \rfloor) + n \leq cn + n = O(n) \quad \dots \text{Wrong!!}$$

Here the error is that you have not prove the exact form of  $T(n) \leq cn$  (i.e., your inductive hypothesis is wrong).

- **Changing Variables:** At times we need to change any complex term present in recurrence by any variable. To perform this we first select a variable in place of complex term and put the variable as substitute in the recurrence equation. Then for finding the actual result replace the chosen variable by the complex term. For example, we are given with the following recurrence  $T(n) = 2T(\sqrt{n}) + \log n$

Then, we rename  $m = \log n$ . This gives

$$T(2^m) = 2T(2^{m/2}) + m$$

Again we can rename  $S(m) = T(2^m)$  to produce the new recurrence  $S(m) = 2S(m/2) + m$  and this recurrence is very much similar to  $T(n) = 2T(n/2) + n$ . Therefore the solution becomes  $S(m) = O(m \log m)$ .

Now changing  $S(m)$  to  $T(n)$  we get

$$T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n) \text{ (as } m = \log n).$$

**Example 6.1** If the recurrence relation is given by  $T(n) = 4T(n/2) + n$ . Solve it through substitution method.

### Solution: 1st Guess

Let

$$T(n) = O(n^3)$$

$$T(n) \leq cn^3 \forall n \geq n_0$$

$$k < n$$

$$T(k) \leq ck^3 \forall k < n$$

For

$$k = \frac{n}{2}$$

$$T(n/2) \leq c(n/2)^3$$

$$T(n) = 4T(n/2) + n$$

$$\leq 4c(n/2)^3 + n$$

$$= \frac{cn^3}{2} + n = cn^3 - \frac{cn^3}{2} + n$$

$$\leq cn^3 - n\left(\frac{cn^3}{2} - 1\right), c > 2 \text{ (Wrong guess!!)}$$

### 2nd Guess

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

$$T(n) \leq cn^3 \forall n \geq n_0$$

$$k < n$$

$$T(k) \leq ck^2 \forall k < n$$

For

$$k = n/2$$

$$T(n/2) \leq c(n/2)^2 \text{ for } k = n$$

$$T(n) = 4T(n/2) + n \leq 4c(n/2)^2 + n = cn^2 + n$$

$$T(n) \leq cn^2 + n$$

$$T(n) \leq cn^2 - (-n)$$

Here we can not neglect  $(-n)$ .

### 3rd Guess

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

$$T(n) \leq c_1 n^2 - c_2 n \forall n \geq n_0$$

$$T(k) \leq c_1 k^2 - c_2 k \quad \forall k < n$$

$$T(n/2) \leq c_1 (n/2)^2 - c_2 (n/2)^2 \text{ for } k = \left(\frac{n}{2}\right)^2$$

$$\begin{aligned} T(n) &= 4T(n/2)^2 + n \\ &\leq 4(c_1 (n/2)^2 - c_2 (n/2)^2) + n \\ &= c_1 n^2 - 2c_2 n + n \\ &\leq \underset{\text{Required}}{c_1 n^2} - \underset{\text{Residue}}{2c_2 n + n} \quad (\text{for } c_2 > 1) \end{aligned}$$

So the solution is  $T(n) = O(n^2)$ .

**Example 6.2** If  $T(n) = 2T(n/2) + n$ . Then prove that  $T(n) = O(n^2)$  by substitution method.

$$\begin{array}{ll} \text{Solution:} & T(n) \leq cn^2 \quad \forall n \geq n_0 \\ & T(k) \leq ck^2 \quad \forall k < n \\ \text{For} & k = n/2 \\ & T(n/2) \leq c(n/2)^2 \\ \text{For} & k = n \\ & T(n) = 2T(n/2) + n \leq 2c(n/2)^2 + n = c(n^2/2) + n \leq cn^2 - cn^2/2 + n \\ & \leq cn^2 - n \left(\frac{cn}{2} - 1\right) \quad (\text{for } c > 2) \end{array}$$

If  $\left(\frac{cn}{2} - 1\right)$  is neglected then,

$$T(n) \leq cn^2$$

Here,  $T(n) = O(n^2)$ .

**Example 6.3** If  $T(n) = 2T(n/2) + n$ . Then prove that  $T(n) = O(n \log n)$  by substitution method.

$$\begin{array}{ll} \text{Solution:} & T(n) \leq c_1 n \log n - c_2 n \quad \forall n \geq n_0 \\ & \quad \downarrow \\ & \quad \text{Lower order term} \\ & T(k) \leq c_1 k \log k - c_2 k \quad (\text{for } k < n) \\ \text{For} & k = n/2, T(n/2) \leq c_1 (n/2) \log (n/2) - c_2 (n/2) \\ \text{For} & k = n, T(n) = 2T(n/2) + n \\ & \leq 2[c_1(n/2) \log (n/2) - c_2(n/2)] + n \\ & \leq c_1 n \log \left(\frac{n}{2}\right) - c_2 n + n \\ & \leq c_1 n \log n - c_1 n - c_2 n + n \end{array}$$

$$\begin{aligned}
 &\leq c_1 n \log n - c_2 n - n(c_1 - 1) \quad (\text{for } c_1 > 1) \\
 &\quad \text{Required} \qquad \qquad \text{Residue} \\
 &= n(c_1 \log n - c_2) \\
 \text{Hence} \qquad T(n) = O(n \log n)
 \end{aligned}$$

**Example 6.4** Solve the problem through substitutions method (change of variable)

$$T(n) = 2T(\sqrt{n}) + \log n.$$

**Solution:** Let  $n = 2^k$

$$\begin{aligned}
 \Rightarrow \qquad \log n &= \log 2^k = k \log 2 = k \quad (\text{As } \log 2 = 1) \\
 \text{As} \qquad \qquad \qquad n = 2^k &\Rightarrow \sqrt{n} = 2^{\frac{k}{2}} \\
 \text{Let} \qquad \qquad \qquad T(2^k) &= S(k) \\
 \text{Then} \qquad \qquad \qquad T(2^k) &= 2T(2^{k/2}) + k \\
 \Rightarrow \qquad \qquad \qquad S(k) &= 2S(k/2) + k \\
 \text{Here} \qquad \qquad \qquad a = 2, b = 2, f(k) &= k \\
 \text{Then} \qquad \qquad \qquad k^{\log_b a} = k^1 &= k
 \end{aligned}$$

Using Master's theorem (Case - II) if  $f(k) = \Theta(k^{\log_b a} \log^p k)$  (where  $p \geq 0$ )

$$\begin{aligned}
 S(k) &= \Theta(k^{\log_b a} \log^{p+1} k) \\
 \Rightarrow \qquad S(k) &= \Theta(k \log k) \quad (p=0) \\
 \text{So,} \qquad \qquad \qquad T(2^k) &= \Theta(k \log k) \\
 \Rightarrow \qquad \qquad \qquad T(n) &= \Theta(\log(\log(\log n)))
 \end{aligned}$$

**Example 6.5** Solve the problem through substitution method (change of variable).

$$T(n) = 2T(\sqrt{n}) + 1.$$

**Solution:** Let  $n = 2^k$

$$\begin{aligned}
 \Rightarrow \qquad \log n &= k \log 2 = k \quad (\text{As } \log 2 = 1) \\
 \text{As} \qquad \qquad \qquad n = 2^k &\Rightarrow \sqrt{n} = 2^{\frac{k}{2}}
 \end{aligned}$$

Now the recurrence equation becomes

$$\begin{aligned}
 T(2^k) &= 2T\left(2^{\frac{k}{2}}\right) + 1 \\
 \text{Let,} \qquad \qquad \qquad T(2^k) &= S(k) \\
 \text{Then} \qquad \qquad \qquad S(k) &= 2S(k/2) + 1 \\
 \text{Here,} \qquad \qquad \qquad a = 2, b = 2, f(k) &= 1 \\
 k^{\log_b a} = k^{\log_2 2} &= k
 \end{aligned}$$

Using Case – 1 of Master's theorem

$$S(k) = \Theta(k)$$

$$\text{Then } T(2^k) = \Theta(k)$$

$$\Rightarrow T(n) = \Theta(\log n)$$

**Example 6.6** Solve  $T(n) = T(\sqrt{n}) + 1$  by substitution method.

**Solution:** Let

$$n = 2^k$$

$$\Rightarrow \log n = k \log 2$$

$$\Rightarrow \log n = k$$

$$\text{Again } \sqrt{n} = 2^{k/2}$$

The recurrence equation becomes

$$T(2^k) = T(2^{k/2}) + 1 \quad \dots(i)$$

$$\text{Let } T(2^k) = S(k)$$

$$\text{Then equation (i) becomes } S(k) = S(k/2) + 1 \quad \dots(ii)$$

$$\text{Here, } a = 1, b = 2, f(k) = 1$$

$$\text{Now } k^{\log_b a} = k^{\log_2 1} = k^0 = 1$$

Applying Case -II of Master theorem on equation (ii)

$$S(k) = \Theta(\log k)$$

$$T(2^k) = \Theta(\log k)$$

$$\Rightarrow T(n) = \Theta(\log(\log n))$$

**Example 6.7** Solve the recurrence  $T(n) = 2T(\sqrt{n}) + 1$  by making change of variables. Your solution should be asymptotically tight.

**Solution:** Given recurrence:

$$T(n) = 2T(\sqrt{n}) + 1 \quad \dots(i)$$

$$\text{Let } n = 2^k$$

$$\Rightarrow \log n = \log 2^k$$

$$\Rightarrow \log n = k \log 2 = k \text{ (As } \log 2 = 1\text{)}$$

$$\text{Again } n = 2^k \Rightarrow \sqrt{n} = 2^{k/2}$$

Now the given recurrence equation becomes

$$T(2^k) = 2T\left(2^{\frac{k}{2}}\right) + 1 \quad \dots(ii)$$

$$\text{Let } T(2^k) = S(k)$$

Then equation (ii) becomes  $S(k) = 2S(k/2) + 1$  ... (iii)

From equation (iv)  $a = 2, b = 2, f(k) = 1$

Now  $k^{\log_b a} = k^{\log_2 2} = k^1 = k$

Applying Master's theorem (Case - I) to solve the recurrence equation (iii)

$$S(k) = O(k^{\log_b a}) = O(k^{\log_2 2}) = O(k)$$

(Important: We have taken 'O' notation instead of  $\theta$  notation as the question asked for asymptotically tight bound)

$$\text{Now } T(2^k) = O(k)$$

$$\Rightarrow T(n) = O(\log n)$$

**Example 6.8** Show that the solution to  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$  is  $O(n \log n)$ .

**Solution:** Here the given recurrence equation is  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$

As the solution to the recurrence equation is given in the question so start guessing from  $O(n \log n)$ . Then, we have to prove

$$\begin{aligned} T(n) &\leq cn \log n \\ \Rightarrow T(n) &\leq 2\{cn/2 \log(n/2) + 17\} + n \\ \Rightarrow T(n) &= cn \log(n/2) + 34 + n = cn\{\log n - \log 2\} + 34 + n \\ &= cn[\log n - 1] + 34 + n = cn \log n - cn + 34 + n \\ &= cn \log n - (c-1)n + 34 \quad (\text{for } c > 1) \\ &\qquad \qquad \qquad \text{Required} \qquad \qquad \text{Residue} \end{aligned}$$

So neglecting residue part  $T(n) = cn \log n$

$$\text{Hence } T(n) = O(n \log n)$$

**Example 6.9** Show the recurrence  $T(n) = T(n - 1) + 1/n$ .

**Solution:** Solving this problem by algebraic substitution

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

$$\text{So, } T(n) = \theta(\log n)$$

**Example 6.10** Solve the recurrence  $T(n) = \sqrt{n}T(\sqrt{n}) + n$

**Solution:** Given recurrence  $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Solving this recurrence by algebraic substitution

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

When  $n^{\frac{1}{2^k}}$  falls under 2, we have  $k > \log \log n$ . We then have

$$\begin{aligned} T(n) &= n^{1-1/\log n} T(2) + n \log \log n \\ \Rightarrow T(n) &= \Theta(n \log \log n) \end{aligned}$$

**Example 6.11** Show that the solution of  $T(n) = T(\lceil n/2 \rceil) + 1$  is  $O(\log n)$  by using substitution method.

**Solution:** The given recurrence equation is  $T(\lceil n/2 \rceil) + 1$

Let guess  $T(n) \in O(n \log n)$ . Then we have to prove

$$T(n) \leq c \log(n)$$

$$\begin{aligned} \Rightarrow T(n) &\leq c \log\left(\frac{n}{2}\right) + 1 = c(\log n - \log 2) + 1 = c(\log n - 1) + 1 \\ &= c \log n - c + 1 = c \log n - (c-1) \end{aligned}$$

Required Residue

$$\text{So, } T(n) \leq c \log n \text{ (for } c \geq 1\text{)}$$

$$\text{Hence } T(n) = O(\log n)$$

**Example 6.12** Show that the solution of  $T(n) = T(\lceil n/2 \rceil) + 1$  is  $\Omega(n \log n)$ . Also conclude that the solution is  $\Theta(n \log n)$ .

**Solution:** The given recurrence equation is  $T(n) = T(\lceil n/2 \rceil) + 1$

$$\text{Let } T(n) \in \Omega(n \log n)$$

$$\text{Then } T(n) \geq 2 \left( c \frac{n}{2} \log \frac{n}{2} \right) + n$$

$$\Rightarrow T(n) \geq cn \log \frac{n}{2} + n$$

$$\begin{aligned} \Rightarrow T(n) &\geq cn (\log n - \log 2) + n \\ &= cn (\log n - 1) + n \\ &= cn \log n - cn + n \end{aligned}$$

$$= cn \log n - n(c-1)$$

Required      Residue

Neglecting residue part  $T(n) \geq cn \log n$  (for  $c = 1$ )

Hence,  $T(n) = \Omega(n \log n)$  (for  $c = 1$ )

$T(n) = \Theta(n \log n)$  only when  $T(n) \leq cn \log n$  (for large value of  $n$  and  $c$ ).

**Example 6.13** Show that the solution of the recurrence equation  $T(n) = T(n-1) + \log n$  is  $\Theta(n \log n)$ .

**Solution:** The given recurrence equation  $T(n) = T(n-1) + \log n$

Let us guess that  $T(n) = \Theta(n \log n)$  as well as  $T(n) \in \Omega(n \log n)$  then  $T(n) \in \Theta(n \log n)$ .

**Prove for  $T(n) \in O(n \log n)$**

Our induction hypothesis is  $T(n) \leq cn \log n$  (where  $c$  is a constant)

Then,

$$\begin{aligned} T(n) &= T(n-1) + \log n \\ &\leq c(n-1) \log(n-1) + \log n = cn \log(n-1) - c \log(n-1) + \log n \\ &\leq cn \log(n-1) - c \log\left(\frac{n}{2}\right) + \log n \\ &= cn \log(n-1) - c \log n + c + \log n \\ &\leq cn \log n - c \log n + c + \log n \\ &\leq cn \log n \end{aligned}$$

If  $-c \log n + c + \log n \leq 0$ ,

then  $-c \log n + c + \log n \leq 0$

$$c \leq (c-1) \log n$$

$$\log n \geq c/(c-1)$$

This works for  $c = 2$  and  $n \geq 4$ .

Hence  $T(n) \in O(n \log n)$

**Prove for  $T \in \Omega(n \log n)$**

Our inductive hypothesis is  $T(n) \geq c_1 n \log n + c_2 n$  for constant  $c_1$  and  $c_2$ .

Then

$$\begin{aligned} T(n) &= T(n-1) + \log n \\ &\geq c_1(n-1) \log(n-1) + c_2(n-1) + \log n \\ &= c_1 n \log(n-1) - c_1 \log(n-1) + c_2 n - c_2 + \log n \\ &\geq c_1 n \log\left(\frac{n}{2}\right) - c_1 \log(n-1) + c_2 n - c_2 + \log n \\ &= c_1 n \log n - c_1 n - c_1 \log(n-1) + c_2 n - c_2 + \log n \\ &\geq c_1 n \log n \end{aligned}$$

If  $-c_1 n - c_1 \log(n-1) + c_2 n - c_2 + \log n \geq 0$ , then

$$-c_1 n - c_1 \log(n-1) + c_2 n - c_2 + \log n > -c_1 n - c_1 \log(n-1) + c_2 n - c_2 + \log(n-1),$$

It suffices to find conditions in which  $-c_1 n - c_1 \log(n-1) + c_2 n - c_2 + \log(n-1) \geq 0$ .

Equivalently,  $-c_1n - c_1 \log(n-1) + c_2n - c_2 + \log(n-1) \geq 0$

$$(c_2 - c_1)n \geq (c_1 - 1)\log(n-1) + c_2$$

This will be true for  $c_1 = 1$ ,  $c_2 = 2$  and for all  $n \geq 2$ .

Hence  $T(n) \in \Omega(n \log n)$ . We have seen before that  $T(n) \in O(n \log n)$

This concludes  $T(n) \in \Theta(n \log n)$ .

### 6.3 RECURSION - TREE METHOD

Recursion tree method converts the recurrence into a tree whose nodes represent costs incurred at various levels of the recursion.

So the basic idea is:

- Each node represents the cost of a single subproblem.
- Sum up the costs with each level to get level cost.
- Sum of all the level costs to get total cost.

Recursion tree method is particularly suitable for divide and conquer recurrence. It is best used to generate a good guess, tolerating "sloppiness".

#### 6.3.1 How to Find Height and Level of Recursion Tree

The general form of recurrence relation is given by  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  where  $a \geq 1$ ,  $b > 1$  and  $f(n)$  is a given function. Then its recursion tree is given by

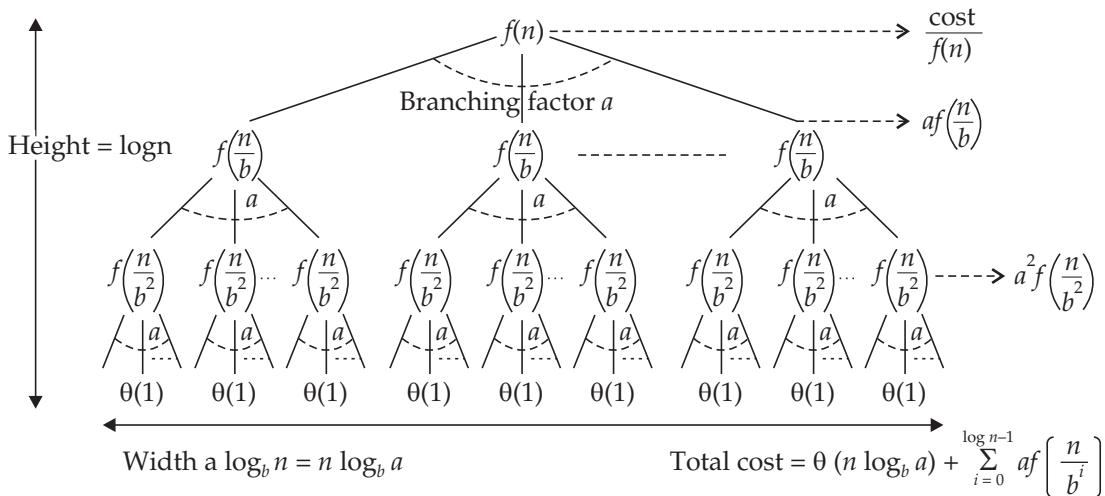


Fig. 6.1. Recursion tree method.

The above tree is a complete array tree with  $n^{\log_b a}$  leaves and height  $\log_b n$ .

$$\left( \text{as } \frac{n}{b^i} = 1 \Rightarrow n = b^i \Rightarrow i = \log_b n \right)$$

**Example 6.14** Find the solution to the recurrence  $T(n) = T(n/2) + T(n/4) + n^2$  by using a recursion tree.

**Solution:** The given recurrence equation is

$$T(n) = T(n/2) + T(n/4) + n^2 \quad \dots(i)$$

When

$$n = n/2$$

$$T(n/2) = T(n/4) + T(n/8) + n^2 \quad \dots(ii)$$

$$\text{Similarly, } T(n/4) = T(n/8) + T(n/16) + n^2 \quad \dots(iii)$$

Now lets draw the recursion tree

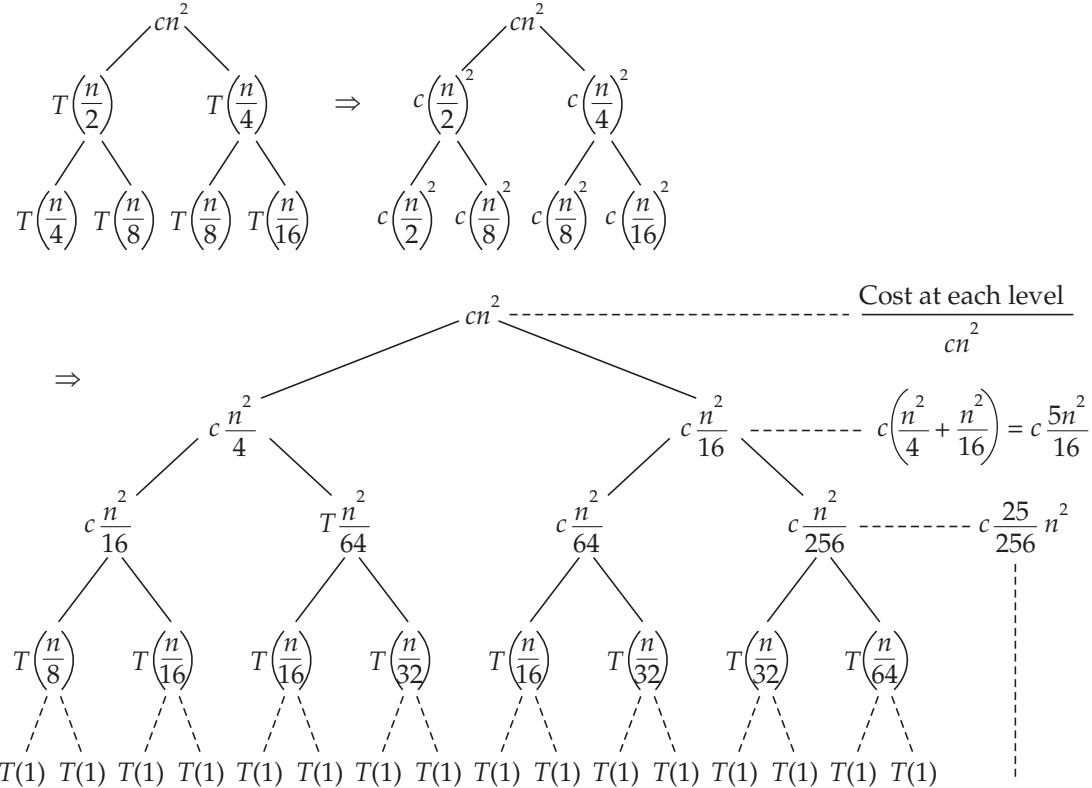
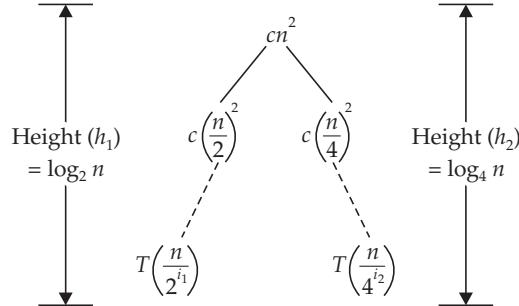


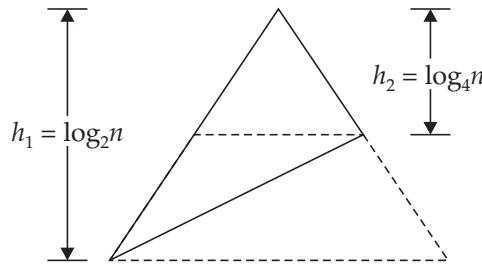
Fig. 6.2. Recursion tree for Example 6.14.

So cost at each level =  $c(5/16)^i n^2$



Here  $h_1 > h_2$  as  $\log_2 n > \log_4 n$ .

Now the tree is like



$$\text{So, } T(n) \leq cn^2 \sum_{i=0}^{\log_2 n} (5/16)^i \quad \dots(iv)$$

$$\text{and } T(n) \geq cn^2 \sum_{i=0}^{\log_4 n} (5/16)^i \quad \dots(v)$$

From equation (iv),

$$\begin{aligned} \sum_{i=0}^{\log_2 n} (5/16)^i &= 1 + (5/16) + (5/16)^2 + (5/16)^3 + \dots + (5/16)^{\log_2 n} \\ &= 1 + 0.312 + 0.0973 + 0.03037 + \dots \\ &\cong 1.4 \leq 2 \end{aligned}$$

Hence,

$$T(n) \leq 2cn^2 = c_1 n^2 \quad (\text{Where } c_1 = 2c)$$

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

From equation (v),

$$\begin{aligned} \sum_{i=0}^{\log_4 n} (5/16)^i &= 1 + (5/16) + (5/16)^2 + (5/16)^3 + \dots + (5/16)^{\log_4 n} \\ &= 1 + 0.312 + 0.0973 + \dots + 0.312 \geq 2 \end{aligned}$$

Hence

$$T(n) \geq c_2 n^2 \Rightarrow T(n) = \Omega(n^2).$$

As  $T(n) \in O(n^2)$  and it is also belongs to  $\Omega(n^2)$  this implies  $T(n) = \Theta(n^2)$ .

**Example 6.15** If  $T(n) = T(n/3) + T(2n/3) + cn$ , then find the solution to the recurrence using recursion tree method.

**Solution:** The given recurrence equation is

$$T(n) = T(n/3) + T(2n/3) + cn \quad \dots(i)$$

When  $n = \frac{n}{3}$ ,

$$T(n/3) = T(n/9) + T(2n/9) + cn \quad \dots(ii)$$

When  $n = \frac{2n}{3}$ , in equation (i),

$$T(2n/3) = T(2n/9) + T(4n/9) + cn \quad \dots(iii)$$

When  $n = \frac{n}{9}$  in equation (i)

$$T(n/9) = T(n/27) + T(n/27) + cn \quad \dots(iv)$$

and so on.

Now lets draw the recursion tree

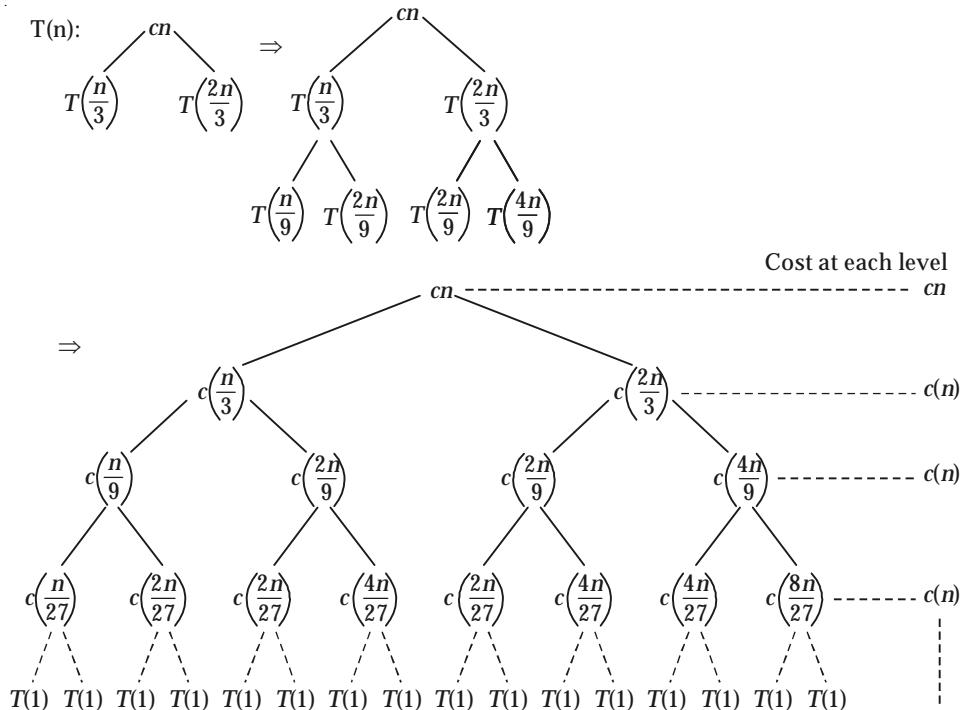
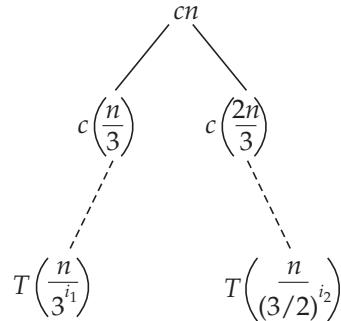


Fig. 6.3. Recursion tree for Example 6.15.

So, the cost at each level is  $cn$ . At level 0 cost is  $cn$ .  
level 1 cost is  $cn$ .

level 2                    cost is  $cn$ .

.....  
level  $i_2$                     cost is  $cn$ .

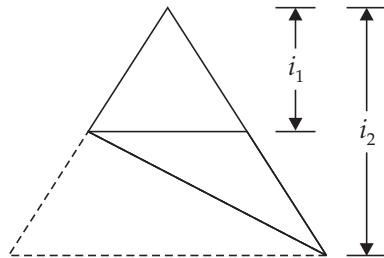


Here                          $n/3^{i_1} = 1 \Rightarrow n = 3^{i_1} \Rightarrow i_1 = \log_3 n$

Again                          $\frac{n}{(3/2)^{i_2}} = 1 \Rightarrow n = (3/2)^{i_2} \Rightarrow i_2 = \log_{3/2} n$

Here                          $i_2 > i_1$  as  $\log_{3/2} n > \log_3 n$

Now the tree is like



It is clear that,

$$T(n) \leq (i_2 + 1) cn \quad \dots(v)$$

$$\text{Again, } (1 + i_1) cn \leq T(n) \quad \dots(vi)$$

$$\text{Now, } (1 + i_1) cn \leq T(n) \leq (1 + i_2) cn \text{ (combining equation (v) and (vi))}$$

$$\Rightarrow cn + i_1 cn \leq T(n) \leq cn + i_2 cn$$

$$\Rightarrow i_1 cn \leq T(n) \leq i_2 cn$$

$$\Rightarrow c(\log_3 n) n \leq T(n) \leq c(\log_{3/2} n) n \quad \dots(vii)$$

(Substituting the values of  $i_1$  and  $i_2$ )

$$\log_3 n = \frac{\log_2 n}{\log_2 3} = \frac{\log n}{\log 3} \quad \dots(viii)$$

$$\log_{\frac{3}{2}} n = \frac{\log_2 n}{\log_2 \frac{3}{2}} = \frac{\log n}{\log \frac{3}{2}} \quad \dots(ix)$$

Substituting the values of equation (viii) and (ix) in equation (vi)

$$c \frac{\log n}{\log 3} n \leq T(n) \leq c \frac{\log n}{\log \frac{3}{2}} n$$

$$\Rightarrow c_1 n \log n \leq T(n) \leq c_2 n \log n$$

where  $c_1 = \frac{c}{\log 3}$  and  $c_2 = \frac{c}{\log \frac{3}{2}}$

So,  $T(n) = \Theta(n \log n)$

**Example 6.16** Solve the recurrence relation given by  $T(n) = 2T(n - 1) + 1$ . Also verify your answer through the substitution method.

**Solution:** The given recurrence equation is that

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

When  $n = n - 1$ , the equation (i) becomes

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

When  $n = n - 2$ , the equation (i) becomes

$$T(n - 2) = 2T(n - 3) + 1 \quad \dots(iii)$$

and so on.

Now lets draw the recursion tree

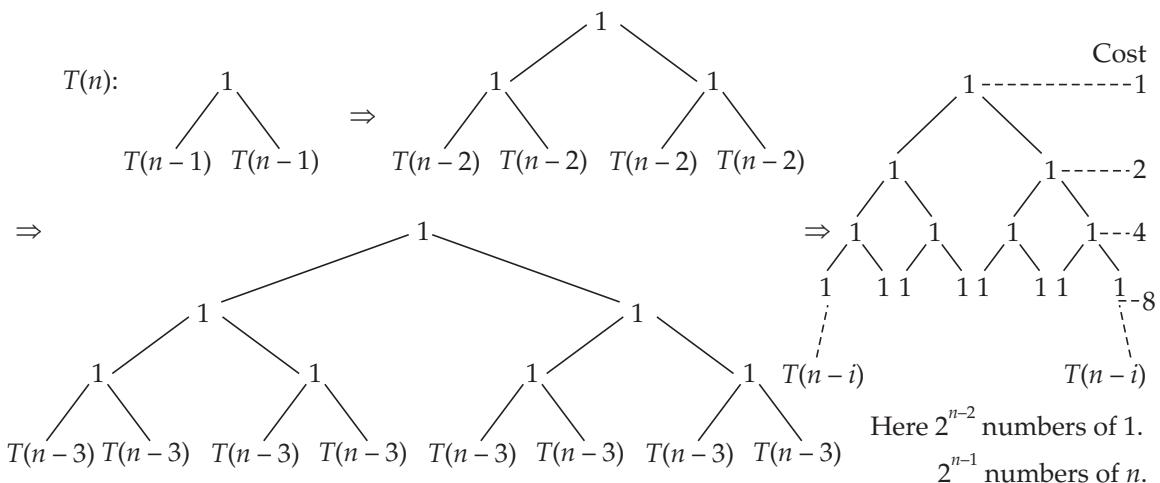


Fig. 6.4. Recursion tree for Example 6.16.

$$\text{Total sum} = \sum_{i=0}^{n-2} 2^i + 2^{n-1} T(1) = T(n)$$

$$\sum_{i=0}^{n-2} 2^i = \frac{2^{n-2+1}-1}{2-1} = 2^{n-1} - 1$$

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

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

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

But  $\frac{1}{2} + \frac{T(1)}{2}$  is a constant represented as  $c$ .

So  $T(n) = 2^n c - 1$

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

#### Verification through substitution method

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

Let us assume  $T(n) = O(2^n)$

Then  $T(n) \leq c \cdot 2^n (\forall n \leq n_0)$

For  $n = k$ ;  $T(k) \leq c \cdot 2^k (\forall k < n)$

For  $k = n - 1$ ;  $T(n - 1) \leq c \cdot 2^{n-1}$

Now  $T(n) = 2T(n - 1) + 1 \leq 2c \cdot 2^{n-1} + 1$

$\Rightarrow T(n) \leq c_1 2^n - c_2 (\forall n \geq n_0)$

$\Rightarrow T(k) \leq c_1 2^k - c_2 (k < n)$

$\Rightarrow T(n) = 2T(n - 1) + 1 \leq 2(c_1 2^{n-1} - c_2) + 1 = c_1 2^n - 2c_2 + 1$

$\Rightarrow T(n) \leq c_1 2^n - c_2 - (c_2 - 1)$

Required      Residue

So,  $T(n) \leq c_1 2^n - c_2$

Here,  $T(n) = O(2^n)$  is correct.

#### Example 6.17 Solve the Towers of Hanoi problem using recurrences.

**Solution:** Let X, Y and Z be three pegs. Our goal is to move  $n$  discs from X to Z using rod Y. The arrangement of disc on X in such a way that the largest disc (Disc  $n$ ) is placed at the bottom of the rod and the smallest disc (Disc 1) is placed at the top, thus making a canonical shape as shown in Fig. 6.5.

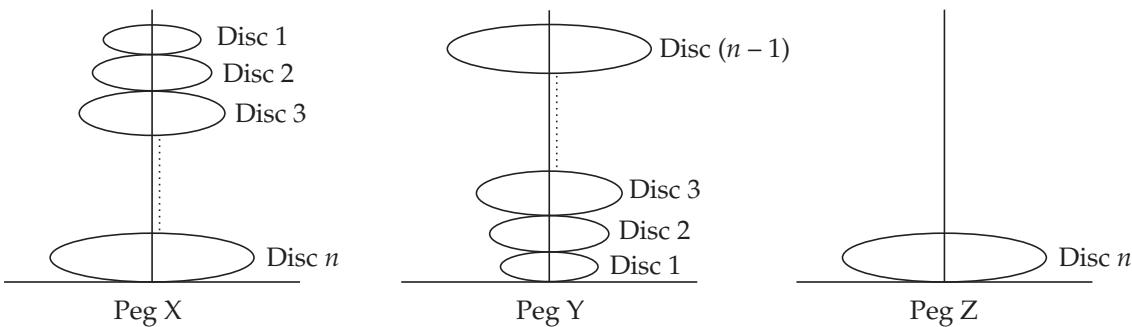


Fig. 6.5. Formulating Towers of Hanoi Problem.

To solve the problem we obey the following rules:

- Only one disc must be moved at a time.
- Each move consisting of taking the upper disc from one of the rod and sliding it into another rod, on top of the other discs that may already be present on that rod.
- No disc can be placed on top of smaller disc.

The following sequence of steps are executed recursively.

**Step 1** Move  $(n - 1)$  discs from X to Y. This leaves disc  $n$  alone on peg X. So the cost of this move is  $n - 1$ .

**Step 2** Move disc  $n$  from X to Z which costs 1.

**Step 3** Move  $n - 1$  discs from Y to Z so they sit on disc  $n$ . This costs  $n - 1$ .

So the recurrence for the example becomes

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

This is the same recurrence that we solved previously in example 6.16.

This results  $T(n) = O(2^n)$ .

**Example 6.18** Solve the recurrence relation given by  $T(n) = 3T(\lfloor n/4 \rfloor) + \theta(n^2)$  by using recursion tree method.

**Solution:** As we know the floors and ceilings usually doesn't matter in recurrence solving. The given recurrence relation becomes

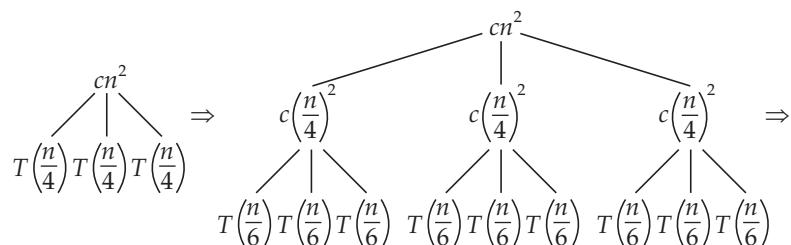
$$T(n) = 3T(n/4) + cn^2 \quad \dots(i)$$

When  $n = (n/4)$  equation (i) becomes  $T(n/4) = 3T(n/16) + cn^2 \quad \dots(ii)$

When  $n = (n/16)$  equation (ii) becomes  $T(n/16) = 3T(n/256) + cn^2 \quad \dots(iii)$

Now lets draw the recurrence tree

$T(n)$ :



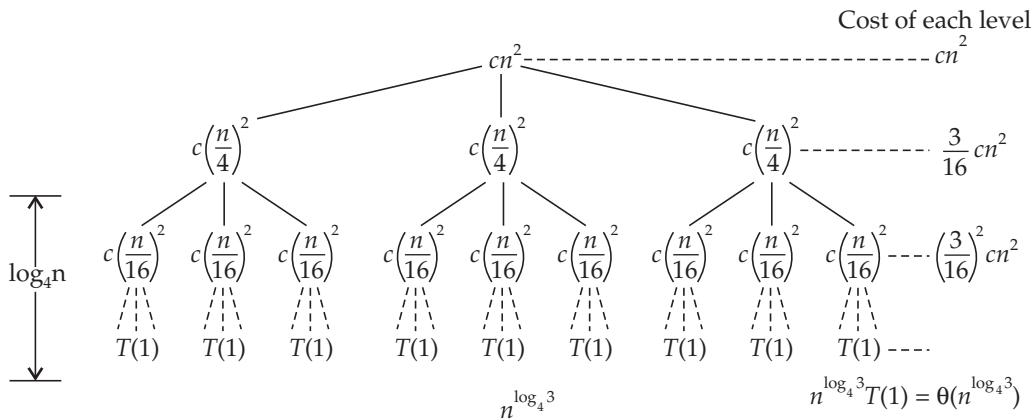


Fig. 6.6. Recursion tree for Example 6.18.

Here,  $\frac{n}{4^i} = 1 \Rightarrow n = 4^i \Rightarrow i = \log_4 n$  (height of the tree)

Thus the levels =  $\log_4 n + 1$

$$\begin{aligned}
 \text{Total cost} &= T(n) = cn^2 + (3/16) cn^2 + (3/16)^2 cn^2 + \dots \\
 &\quad + (3/16)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \sum_{i=0}^{\log_4(n-1)} (3/16)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &< \sum_{i=0}^{\infty} (3/16)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2)
 \end{aligned}$$

**Example 6.19** Use a recursion tree to determine a good asymptotic upper bound on the recurrence  $T(n) = T(n/2) + n^2$ . Use the substitution method of verify your answer.

**Solution:** The given recursion equation becomes

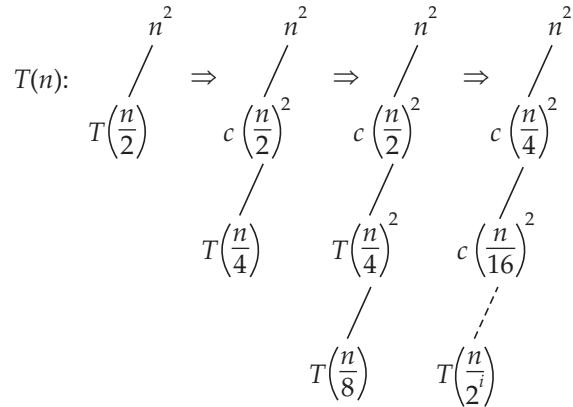
$$T(n) = T(n/2) + n^2 \quad \dots(i)$$

$$\text{When } n = n/2 \text{ equation (i) becomes } T(n/2) = T(n/4) + n^2 \quad \dots(ii)$$

$$\text{When } n = n/4 \text{ equation (i) becomes } T(n/4) = T(n/8) + n^2 \quad \dots(iii)$$

and so on.

Now lets draw the recurrence tree



**Fig. 6.7.** Recursion tree for Example 6.19.

Here  $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log n = \text{height of the tree}$

$$\text{Hence total cost } T(n) = n^2 + \frac{n^2}{4} + \frac{n^2}{16} + \dots + \frac{n^2}{4^i} = n^2 \sum_{i=0}^{\log n} 1/4^i$$

Here,  $1/4^i$  is a constant, which is  $\leq 2$ . So,  $T(n) = O(n^2)$

#### Verification through substitution method

$$\text{Given } T(n) = T(n/2) + 1$$

$$\text{Let us assume } T(n) \in O(n^2)$$

$$\text{Then } T(n) \leq c.n^2$$

$$T(k) \leq c.k^2 \quad (\text{for } n = k)$$

$$T(n/2) \leq c(n/2)^2 \quad \left( \text{for } k = \frac{n}{2} \right)$$

$$\text{Again, } T(n) = T(n/2) + n^2$$

$$\Rightarrow T(n) \leq c(n/2)^2 + n^2$$

$$\Rightarrow T(n) \leq c \frac{n^2}{4} + n^2$$

$$\Rightarrow T(n) \leq n^2 \left( \frac{c}{4} + 1 \right)$$

Here  $\left( \frac{c}{4} + 1 \right)$  is a constant. Let denote it as  $c'$ .

$$\begin{aligned} \text{So, } & T(n) \leq c'n^2 \\ \Rightarrow & T(n) = O(n^2) \text{ is correct.} \end{aligned}$$

**Example 6.20** Use the recursion tree to determine the solution of the recurrence equation.

$$T(n) = T(n/2) + 3. \text{ Given } T(1) = 4.$$

**Solution:** The given recurrence equation  $T(n) = T(n/2) + 3$  ... (i)

$$\text{When } n = (n/2), \quad T(n/2) = T(n/4) + 3 \quad \dots (ii)$$

Substituting the value of  $T(n/2)$  from (ii) in (i)

$$T(n) = T(n/4) + 3 + 3 \quad \dots (iii)$$

$$\text{From (i) when } n = n/4, T(n/4) = T(n/8) + 3 \quad \dots (iv)$$

Substituting the value of  $T(n/4)$  from (iv) in (iii)

$$T(n) = T(n/8) + 3 + 3 + 3$$

$$\text{Similarly, } T(n) = T(n/16) + 3 + 3 + 3 + 3$$

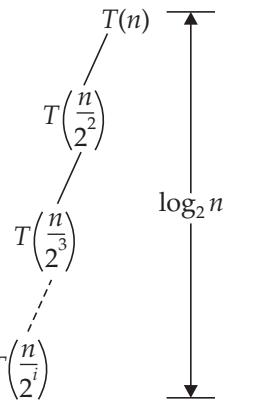
and so on.

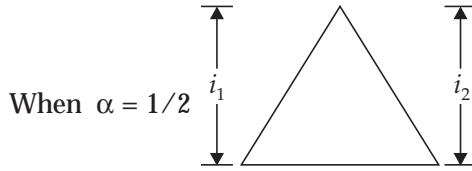
$$\begin{aligned} \text{So we get } & T(n) = T(n/2^i) + 3 + 3 + 3 + \dots + 3 \\ & \quad i \text{ times } 3 \end{aligned}$$

$$\text{Here } (n/2^i) = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$$

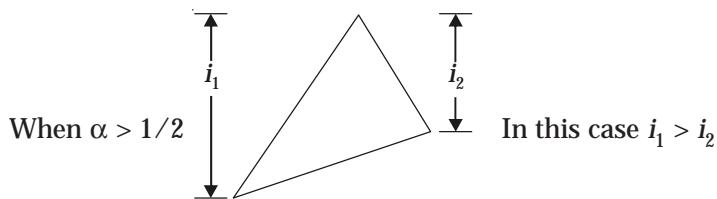
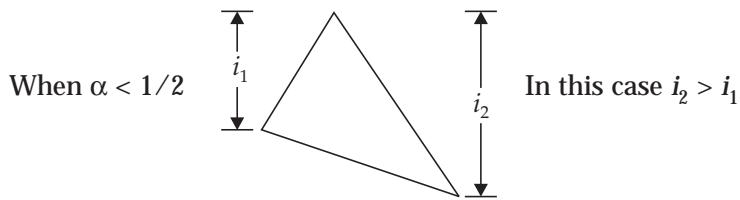
$$\text{Now } T(n) = T(1) + 3(\log_2 n)$$

$$\Rightarrow T(n) = 3\log_2 n + 4 \text{ (As } T(1) = 4 \text{ given)}$$





In this case  $i_1 = i_2$  i.e., height of leftside of the tree = height of right side



So,

$$i_1 = \log_{1/\alpha} n = \frac{\log n}{\log_2 1/\alpha} \quad (\text{Applying } \log_a b = \log_c b / \log_c a)$$

and

$$i_2 = \log \frac{1}{1-\alpha} n$$

Now the recurrence relation becomes

$$\begin{aligned} (1 + i_1)cn &\leq T(n) \leq (1 + i_2)cn \\ \Rightarrow c_1 n \log n &\leq T(n) \leq c_2 n \log n \\ \Rightarrow T(n) &= \theta(n \log n) \end{aligned}$$

#### Remark:

- ⇒ If the previous question asks to find the asymptotic tight solution then  
 $T(n) \leq (1 + i_2)cn = O(n \log n)$
- ⇒ If  $T(n)$  contains one part having 99% of the subproblem and the other part contains only 1% of the subproblem then in every case (like 6.20)  $T(n)$  will be  $\theta(n \log n)$ .
- ⇒ Other examples of problems of this kind are

$$T(n) = T(n/100) + T(99n/100) + cn, \text{ where } T(n) \text{ gives } \theta(n \log n)$$

$$T(n) = T(n/10) + T(9n/10) + cn, \text{ where } T(n) \text{ gives } \theta(n \log n) \text{ etc.}$$

**Example 6.22** Use a recursion tree to give an asymptotically tight solution to the recurrence  $T(n) = T(n - a) + T(a) + cn$ , where  $a \leq 1$  and  $c > 0$  are constants. Verify your answer through substitution method.

**Solution:** The given recurrence equation is  $T(n) = T(n - a) + T(a) + cn$  ... (i)

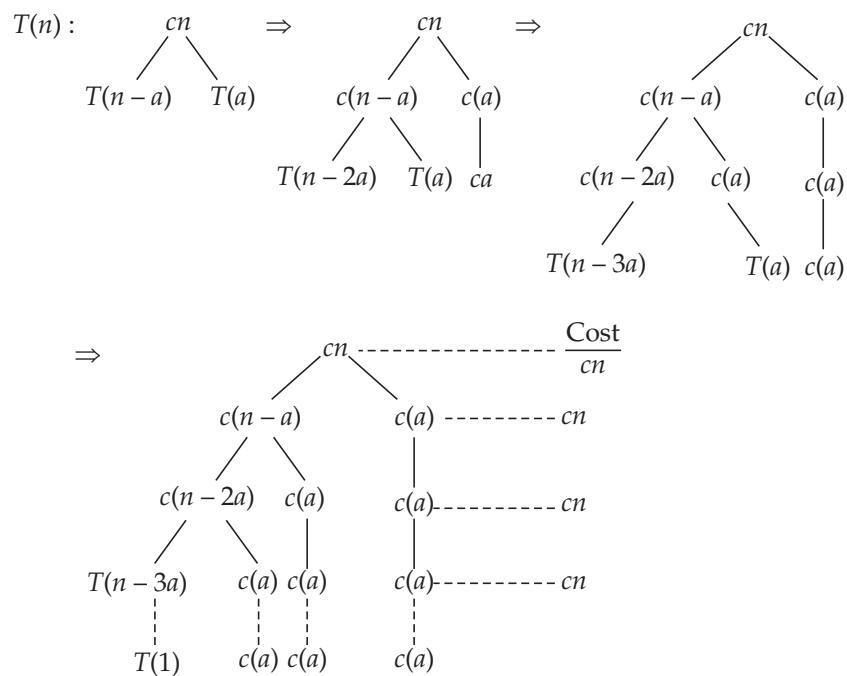
When  $n = n - a$ , the recurrence (i) becomes  $T(n - a) = T(n - 2a) + T(a) + cn$  ... (ii)

When  $n = n - 2a$ , the recurrence (i) becomes  $T(n - 2a) = T(n - 3a) + T(a) + cn$  ... (iii)  
and so on.

Now drawing the recursion tree

Let the tree ends at  $(n - ia)$ . (where ' $i$ ' is the level)

Then  $(n - ia) = 1 \Rightarrow i = \frac{n-1}{a} \leq \frac{n}{a}$  number of levels



**Fig. 6.10.** Recursion tree for Example 6.22.

$$\text{So } T(n) = cn + cn + cn + \dots + cn = \left(\frac{n}{a} + 1\right)cn = \frac{cn^2}{a} + cn = O(n^2)$$

$\left(\frac{n}{a} + 1\right)$  times

## Verification by substitution method

$$\text{Given } T(n) = T(n - a) + T(a) + cn$$

Let us assume  $T(n) \in O(n^2)$

Then  $T(n) \leq c'n^2$

$$T(k) \leq c' k^2 \text{ (for } n = k)$$

$$\text{Now, } T(n) \equiv T(n-a) + T(a) + cn \leq c'(n-a)^2 + c'(a)^2 + cn$$

$$\equiv c'n^2 - 2ac'n + 2c'a^2 + cn \equiv c'n^2 - (2ac'n - 2c'a^2 - cn)$$

For  $T(n) \leq c'n^2$

$$2ac'n - 2c'n^2 - cn > 0$$

So,  $c' = \left( \frac{c+1}{2a} \right)$

Then  $T(n) \leq c'n^2 - (n - ac - a)$

$$\Rightarrow T(n) = O(n^2) \text{ (for large value of } n\text{)}$$

This proves that our guess is correct.

**Example 6.23** Find the running time of merge sort using recursion tree method.

**Solution:** The (worst case) running time of the merge sort is given by

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

We can rewrite the above recurrence as

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

Here  $c$  is a constant.

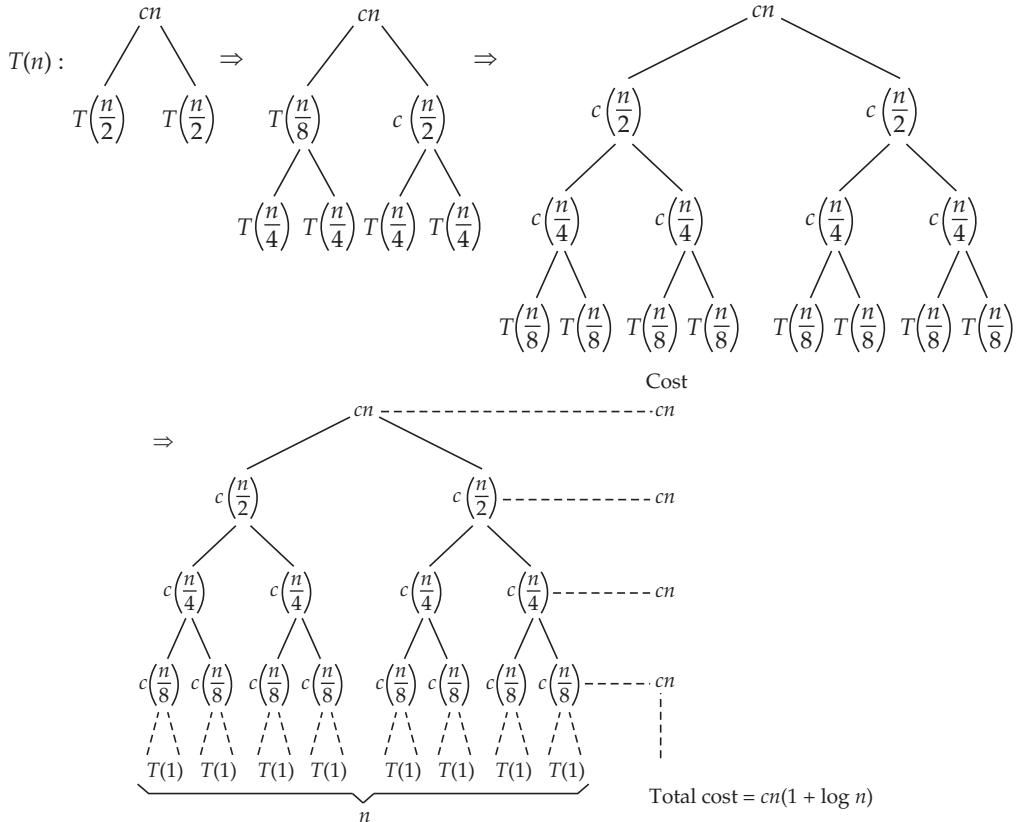
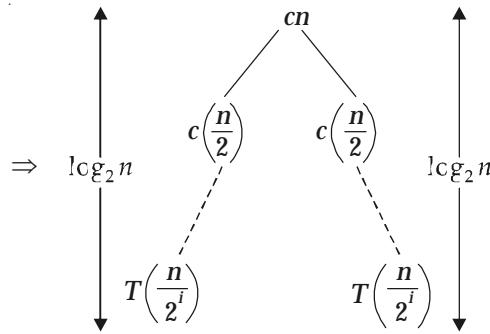


Fig. 6.11. Recursion tree for Example 6.23.



As,  $n/2^i = 1 \Rightarrow i = \log_2 n = \text{height of the tree}$

So, number of levels =  $\log_2 n = 1$

Hence total cost = cost of each level  $\times$  total number of levels

$$\begin{aligned} &= cn \times (\log_2 n + 1) \\ &= cn \log_2 n + cn \end{aligned}$$

Running time of the merge-sort =  $\Theta(n \log n)$

**Example 6.24** Using the recursion tree method find the asymptotic solution to the recurrence

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n.$$

**Solution:** The given recurrence equation is

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n.$$

Now,

$$T(n/2) = T(n/4) + T(n/8) + T(n/16) + n$$

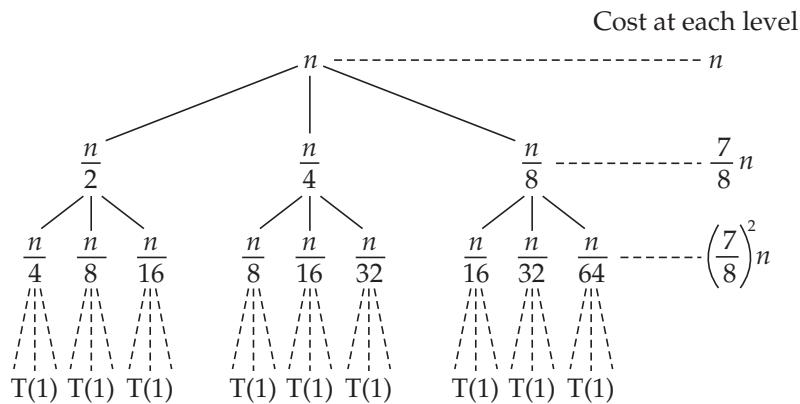
Similarly,

$$T(n/4) = T(n/8) + T(n/16) + T(n/32) + n$$

and

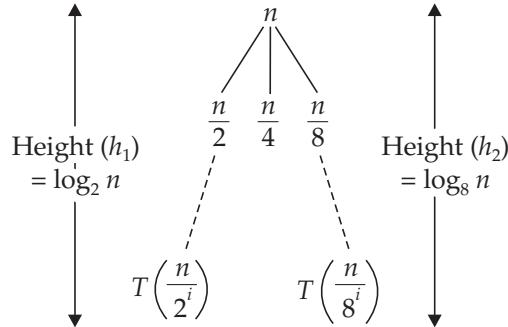
$$T(n/8) = T(n/16) + T(n/32) + T(n/64) + n \text{ and so on.}$$

Now the recursion tree becomes



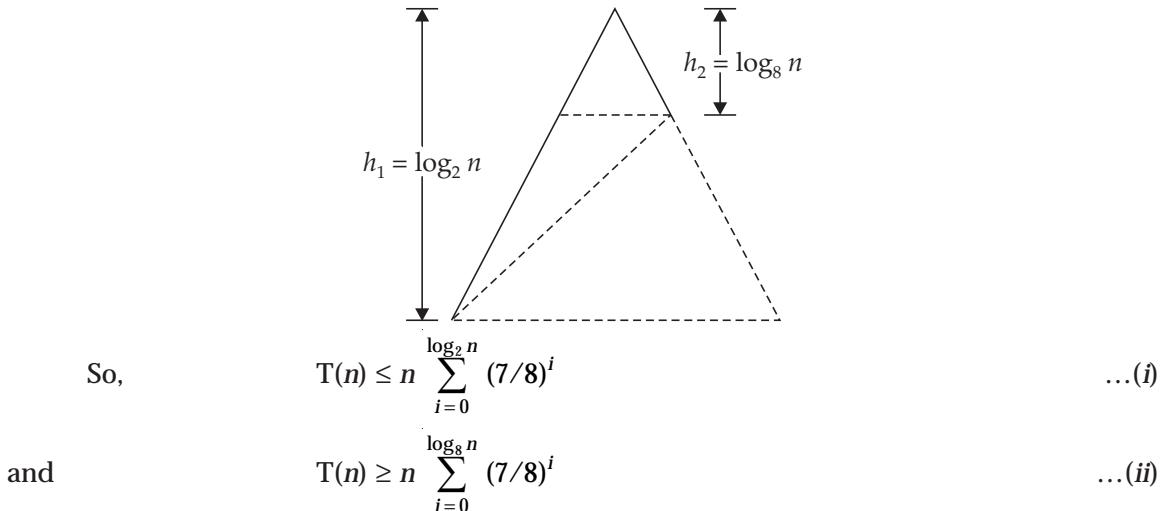
**Fig. 6.12.** Recursion tree for Example 6.24.

So cost at each level is expressed as  $\left(\frac{7}{8}\right)^i n$  (where  $i$  represents the level number)



Here  $h_1 > h_2$  as  $\log_2 n > \log_8 n$

Now the tree is like



$$\text{From equation (i), } \sum_{i=0}^{\log_2 n} (7/8)^i = 1 + (7/8) + (7/8)^2 + (7/8)^3 + \dots + (7/8)^{\log_2 n}$$

$$= 1 + 0.875 + 0.765 + 0.669 + \dots$$

$$\cong 5.7 \leq 6$$

$$\text{Hence, } T(n) \leq 6.n = c_1 n \text{ (where } c_1 = 6\text{)}$$

$$\Rightarrow T(n) = O(n)$$

$$\text{Again from equation (ii), } \sum_{i=0}^{\log_8 n} (7/8)^i = 1 + (7/8) + (7/8)^2 + (7/8)^3 + \dots + (7/8)^{\log_8 n}$$

$$= 1 + 0.875 + 0.765 + 0.669 + \dots \geq 6$$

$$\text{Hence, } T(n) \geq 6n \Rightarrow T(n) = \Omega(n)$$

We have seen  $T(n) \in O(n)$  as well as  $T(n) \in \Omega(n)$ . It concludes  $T(n) \in \Theta(n)$ .

**Example 6.25** Show that  $T(n) = 4T(n/3) + n$  is  $\theta(n^{\log_3 4})$  by using recursion tree method.

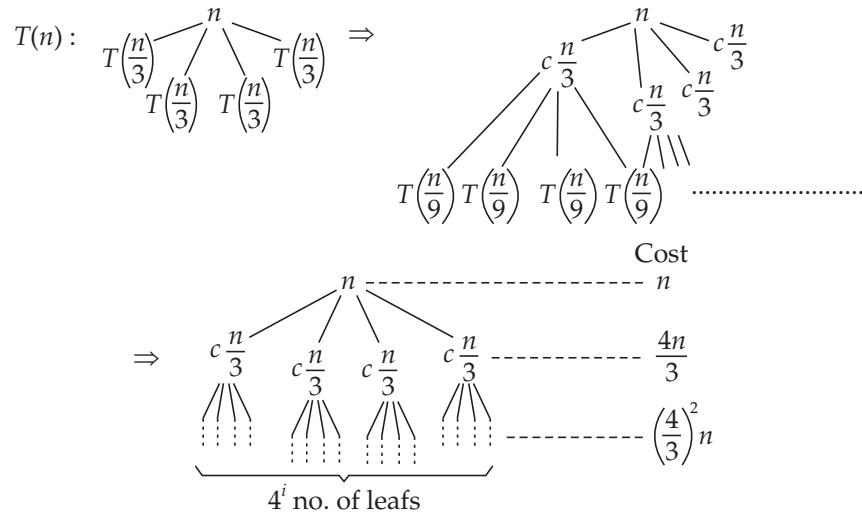
**Solution:** The given recurrence equation is:

$$T(n) = 4T(n/3) + n$$

$$\text{So, } T(n/3) = 4T(n/9) + n$$

and so on.

Lets draw the recursion tree.



**Fig. 6.13.** Recursion tree for Example 6.25.

$$\text{Here, } n/3^i = 1 \Rightarrow n = 3^i \Rightarrow i = \log_3 n$$

$$\text{Number of leaf} = 4^i = 4^{\log_3 n} = n^{\log_3 4}$$

$$\text{Now, } T(n) = n + (4/3)n + (4/3)^2 n + (4/3)^3 n + \dots + (4/3)^i n + 4^i T(1)$$

$$\begin{aligned} &= n\{1 + 4/3 + (4/3)^2 + \dots + (4/3)^i\} + cn^{\log_3 4} \\ &= n \frac{(4/3)^{i+1} - 1}{(4/3) - 1} + cn^{\log_3 4} = n \frac{(4/3)^{i+1} - 1}{(1/3)} + cn^{\log_3 4} = 3n((4/3)^{i+1} - 1) + cn^{\log_3 4} \\ &= 3n \left\{ \frac{4}{3} \left( \frac{4}{3} \right)^i - 1 \right\} + cn^{\log_3 4} = 4n \frac{4^i}{3^i} - 3n + cn^{\log_3 4} = 4 \times 4^{\log_3 4} - 3n + cn^{\log_3 4} \\ &= (4 + c) 4^{\log_3 4} - 3n \\ &= c'n^{\log_3 4} - 3n \end{aligned}$$

$$\text{Hence } T(n) = \theta(n^{\log_3 4})$$

## 6.4 MASTER THEOREM

If the given recurrence relation is of the form of  $T(n) = aT(n/b) + f(n)$ ; where  $a \geq 1$ ,  $b > 1$  are positive integers,  $f(n)$  be a non-negative function and  $n/b$  may be  $\lceil n/b \rceil$  or  $\lfloor n/b \rfloor$  then we can apply Master theorem to find the running time by using below three cases.

### Case – I

If  $f(n) = O(n^{\log_b a - \epsilon})$  for some  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$

### Case – II

If  $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$$

### Case – III

If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some  $\epsilon > 0$  and if  $af\left(\frac{n}{b}\right) \leq cf(n)$  for some  $c < 1$  and all sufficiently large  $n$ , then  $T(n) = \Theta(f(n))$

### Tricks to Remember Master Theorem

From recurrence relation of form  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  determine,  $a$ ,  $b$ ,  $f(n)$  and  $n^{\log_b a}$

### Case – I

If  $n^{\log_b a}$  is growing faster than  $f(n)$  i.e.,  $n^{\log_b a} >> f(n)$  then case – I is applicable.

### Case – II

If  $n^{\log_b a} = f(n)$  then case – II is applicable

### Case – III

If you found  $f(n)$  is growing faster than  $n^{\log_b a}$  then case - III is applicable. Here  $f(n) >> n^{\log_b a}$

### 6.4.1 Implications of Master Theorem

There are different implications of Master theorem. They are given as follows:

- ⇒ Comparison between  $f(n)$  and  $n^{\log_b a}$  ( $<$ ,  $=$ ,  $>$ )
- ⇒ Must be asymptotically smaller (or larger) by a polynomial, i.e.,  $n^\epsilon$  for some constant  $\epsilon > 0$ .
- ⇒ In Case - III, the “regularity” must be satisfied i.e.,  $af\left(\frac{n}{b}\right) \leq cf(n)$  for some  $c < 1$ .

$\Rightarrow$  There are gaps

- between Case - I and II:  $f(n)$  is smaller than  $n^{\log_b a}$ , but not polynomially smaller.
- between Case - II and III:  $f(n)$  is smaller than  $n^{\log_b a}$ , but not polynomially larger.
- In Case - III, if the “regularity” fails to hold.

$\Rightarrow$  These gaps are shown below:

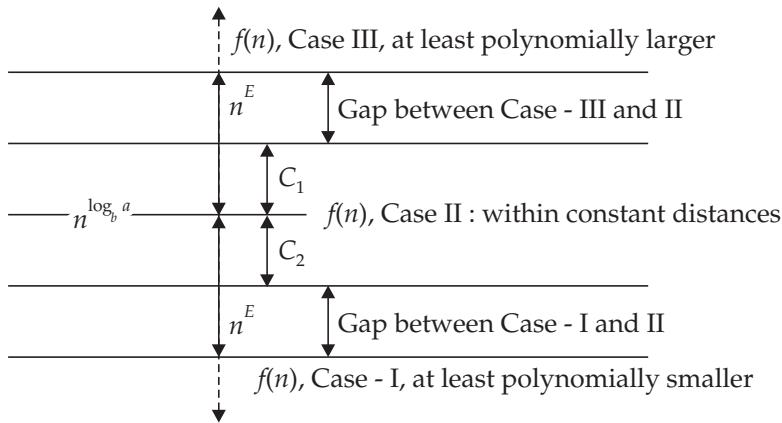


Fig. 6.14. Gaps in Master Theorem.

**Note:**

1. For Case - III, the regularity also must hold i.e., of  $\left(\frac{n}{b}\right) \leq c f(n)$  for some constant  $c < 1$ .
2. If  $f(n)$  is  $\log n$  smaller, then fall in gap between Case - I and II.
3. If  $f(n)$  is  $\log n$  larger, then fall in gap between Case - III and II.
4. If  $f(n) = \theta(n^{\log_b a} \log^k n)$ , then  $T(n) = \theta(n^{\log_b a} \log^{k+1} n)$ .

#### 6.4.2 Proof of Master Theorem

From section 6.3.1, the recursion tree of  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  gives  $g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \cdot g(n)$

can be bounded for exact power of  $b$  as:

##### Case - I

If  $f(n) = O(n^{\log_b a - \varepsilon})$  for some  $\varepsilon > 0$  then  $g(n) = O(n^{\log_b a})$

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \varepsilon}\right)$$

$$\begin{aligned}
&= O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} \frac{a^j}{(b^{\log_b a - \varepsilon})^j}\right) = O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} \frac{a^j}{(a^j(b^{-\varepsilon})^j)}\right) \\
&= O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{\log_b n - 1} (b^\varepsilon)^j\right) = O(n^{\log_b a - \varepsilon} ((b^\varepsilon)^{\log_b n} - 1)/(b^\varepsilon - 1))) \\
&= O(n^{\log_b a - \varepsilon} (((b^{\log_b n})^\varepsilon - 1)/(b^\varepsilon - 1))) = O(n^{\log_b a} n^{-\varepsilon} (n^\varepsilon - 1)/(b^\varepsilon - 1)) = O(n^{\log_b a})
\end{aligned}$$

**Case – II**

If  $f(n) = \Theta(n^{\log_b a})$  then  $g(n) = \Theta(n^{\log_b a} \log n)$

**Proof:**  $f(n) = \Theta(n^{\log_b a})$  implies  $f\left(\frac{n}{b^j}\right) = \Theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)$ , so

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) \\
&= \Theta\left(n^{\log_b a} \sum_{j=0}^{\log_b n - 1} a^j / (b^{\log_b a})^j\right) = \Theta\left(n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1\right) \\
&= \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \log n)
\end{aligned}$$

**Case – III**

If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some  $\varepsilon > 0$  and if  $af\left(\frac{n}{b}\right) \leq cf(n)$  for some  $c < 1$  and all sufficiently large  $n \geq b$  then  $g(n) = \Theta(f(n))$

**Proof:** Since  $g(n)$  contains  $f(n)$ ,  $g(n) = \Omega(f(n))$

and

$$af\left(\frac{n}{b}\right) \leq cf(n), a^j f\left(\frac{n}{b^j}\right) \leq c^j f(n)$$

$$\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \leq \sum_{j=0}^{\log_b n - 1} c^j f(n) \leq f(n) \sum_{j=0}^{\infty} c^j \\
&= f(n) \left(\frac{1}{(1-c)}\right) = O(f(n))
\end{aligned}$$

Thus

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

**Example 6.26** Find the running time of the Merge sort from its recurrence relation  $T(n) = 2T(n/2) + cn$  by using Master theorem.

**Solution:** From merge-sort

$$T(n) = 2T(n/2) + cn$$

It is of the form of

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

Where  $a = 2, b = 2, f(n) = cn \approx cn' \cdot \log^0 n$  (where  $k = 0$ )

$$\text{Now } n^{\log_b a} = n^{\log_2 2} = n$$

By applying Case - II of Master theorem.

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

**Example 6.27** Find the running time for  $T(n) = 4T(n/2) + n$  by using Master theorem.

**Solution:** Given  $T(n) = 4T(n/2) + n$

$$\text{Here } a = 4, b = 2, f(n) = n$$

$$\text{Again } n^{\log_b a} = n^{\log_2 4} = n^2$$

We can see  $n^2$  grow faster than  $n$  i.e.,  $n^2 >> n$ .

Applying Case - I of Master theorem we get

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

**Example 6.28** If  $T(n) = 4T(n/2) + n^2$ . Then find its running time through Master theorem.

**Solution:** Given  $T(n) = 4T(n/2) + n^2$

$$\text{Here } a = 4, b = 2, f(n) = n^2 \approx n^{\log_4 2} (\log n)^0 \text{ (where } k = 0\text{)}$$

$$\text{Again, } n^{\log_b a} = n^{\log_2 4} = n^2$$

So, applying Case - II of Master theorem for finding the running time

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^2 \log n)$$

**Example 6.29** If  $T(n) = 4T(n/2) + n^3$ . Then solve this recurrence equation by using Master theorem.

**Solution:** Given  $T(n) = 4T(n/2) + n^3$

The above recurrence equation is of the form of  $T(n) = aT(n/b) + f(n)$

$$\text{Where } a = 4, b = 2, f(n) = n^3$$

$$\text{Again } n^{\log_b a} = n^{\log_2 4} = n^2$$

As  $n^3$  grows faster than  $n^2$  i.e.,  $n^3 >> n^2$ , it is a clear case of applying Case - III of Master theorem to find its solution.

$$\text{Then } T(n) = \Theta(f(n)) = \Theta(n^3).$$

**Example 6.30** Use the master method to show that the solution to the binary search recurrence  $T(n) = T(n/2) + \Theta(1)$  is  $T(n) = \Theta(n \log n)$ .

**Solution:** Given recurrence is  $T(n) = T(n/2) + \Theta(1)$

$$\text{Here } a = 1, b = 2, f(n) = \Theta(1) = 1$$

$$\text{Again } n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

So applying Case - II of Master theorem to solve the recurrence

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a} \log^{k+1} n) \\ &= \Theta(n^0 \cdot \log n) = \Theta(\log n) \end{aligned} \quad (\text{Here } k = 0)$$

**Example 6.31** Solve the recurrence relation  $T(n) = 3T(n/4) + n \log n$  by using Master theorem.

**Solution:** The given recurrence  $T(n) = 3T(n/4) + n \log n$

$$\text{Here } a = 3, b = 4, f(n) = n \log n$$

$$\text{Again } n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$\text{Since, } f(n) = \Omega(n^{\log_b a + \varepsilon}) \quad (\text{where } \varepsilon \approx 0.2)$$

Clearly it is the Case - II of Master theorem.

$$\text{So, } T(n) = \Theta(n \log n)$$

**Example 6.32** Can the master method be applied to the recurrence  $T(n) = 4T(n/2) + n^2 \log n$ ? Why or why not? Give an asymptotic upper bound for this recurrence?

**Solution:**  $T(n) = 4T(n/2) + n^2 \log n$

$$\text{Here } a = 4, b = 2, f(n) = n^2 \log n$$

$$\text{Again } n^{\log_b a} = n^{\log_2 4} = n^2$$

We can see that  $f(n)$  is asymptotically larger than  $n^{\log_b a}$  but not polynomially larger because  $\frac{f(n)}{n^{\log_b a}} = \frac{n^2 \log n}{n^2} = \log n$  which is asymptotically less than  $n^\varepsilon$  for any  $\varepsilon > 0$ .

So the recurrence falls into the gap between Case - II and Case - III of Master theorem. Hence, the Master theorem can not be applied to the above recurrence.

**Example 6.33** Solve the recurrence relation  $T(n) = 7T(n/2) + n^2$  by using Master theorem method.

**Solution:**  $T(n) = 7T(n/2) + n^2$

$$\text{Here } a = 7, b = 2, f(n) = n^2$$

$$\text{Again } n^{\log_b a} = n^{\log_2 7} \geq n^{2.8}$$

Here  $n^{2.8} >> n^2$ . Hence Case - I of Master theorem is applicable here.

$$\text{So, } T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.8})$$

**Example 6.34** Solve the recurrence  $T(n) = 7T(n/3) + n^2$  by using Master theorem.

**Solution:** Given  $T(n) = 7T(n/3) + n^2$

$$\text{Here } a = 7, b = 3, f(n) = n^2$$

$$\text{Again } n^{\log_b a} = n^{\log_3 7} \text{ and } 1 < \log_3 7 < 2$$

$$f(n) = \Omega(n^{\log_3 7 + \varepsilon}) \text{ where } \varepsilon = 2 - \log_3 7$$

$$\text{So, } T(n) = \Theta(n^2)$$

**Example 6.35** Solve the recurrence  $T(n) = 3T(n/2) + n \log n$  by using Master theorem.

**Solution:** Given  $T(n) = 3T(n/2) + n \log n$

$$\text{We have } a = 3, b = 2, f(n) = n \log n.$$

$$\text{Again } n^{\log_b a} = n^{\log_2 3} \leq n^{1.58}$$

Since  $f(n) = O(n^{\log_b a - \varepsilon})$  (where  $\varepsilon \leq 0.58$ )

Clearly Case - I of Master theorem is applicable here.

So,  $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58})$

**Example 6.36** Solve the recurrence relation  $T(n) = 4T(n/2) + n^2\sqrt{n}$  by using Master method.

**Solution:** Given  $T(n) = 4T(n/2) + n^2\sqrt{n}$

$$\text{Here } a = 4, b = 2, f(n) = n^2\sqrt{n} = n^2 \cdot n^{\frac{1}{2}} = n^{\left(2 + \frac{1}{2}\right)} = n^{\frac{5}{2}} = n^{2.5}$$

$$\text{Again } n^{\log_b a} = n^{\log_2 4} = n^2$$

We can see clearly  $f(n) >> n^{\log_b a}$ . So applying Case - III of Master theorem to solve the recurrence, we get

$$T(n) = \Theta(f(n)) = \Theta(n^2\sqrt{n}) = \Theta(n^{2.5})$$

**Example 6.37** Solve the recurrence relation  $T(n) = 5T(n/5) + (n/\log n)$

**Solution:** Given  $T(n) = 5T(n/5) + (n/\log n)$

$$\text{Here } a = 5, b = 5, f(n) = n/\log n$$

$$\text{Again } n^{\log_b a} = n^{\log_5 5} = n^1 = n$$

None of the Master theorem cases may be applied here, since  $f(n)$  is neither polynomially bigger or smaller than  $n$ , and is not equal to  $\Theta(n \log_k n)$  for any  $k \geq 0$ . Therefore, we will solve this problem through algebraic substitution.

$$\begin{aligned} T(n) &= 5T(n/5) + (n/\log n) \\ &= 5\left(5T(n/25) + (n/5)\right) + \frac{n}{\log n} \\ &= 25T(n/25) + \frac{n}{\log(n/5)} + \frac{n}{\log n} \\ &\dots \\ &= 5^i T(n/5^i) + \sum_{j=1}^{i-1} \frac{n}{\log(n/5^j)} \end{aligned}$$

When  $i = \log_5 n$  the first term reduces to  $5^{\log_5 n} T(1)$ .

$$\text{So we have } T(n) = n\theta(1) + \sum_{j=1}^{\log_5 n - 1} \left( \frac{n}{\log(n/5^{j-1})} \right)$$

$$\begin{aligned}
&= \theta(n) + n \sum_{j=1}^{\log_5 n - 1} (1/(\log n - (j-1) \log_2 5)) \\
&= \theta(n) + n(1/\log_2 5) \sum_{j=1}^{\log_5 n - 1} (1/\log_5 n - (j-1)) \\
&= \theta(n) + n \log_5 2 \sum_{j=2}^{\log_5 n} \left( \frac{1}{j} \right)
\end{aligned}$$

This is the harmonic sum (refer Appendix-I for Harmonic sum), so we have

$$T(n) = \theta(n) + c_2 n \log (\log_5 n) + \theta(1) = \theta(n \log \log n)$$

## CHAPTER NOTES

---

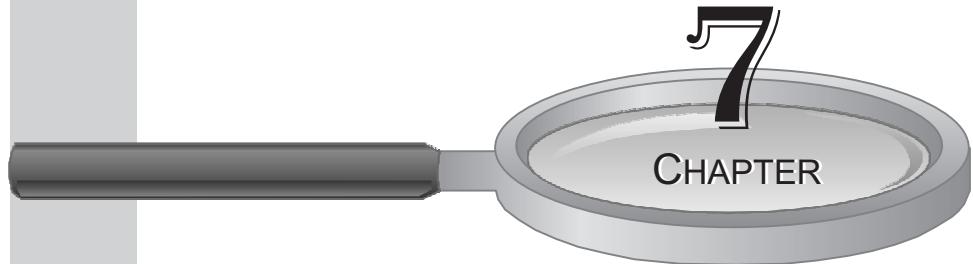
- Recurrences are a major tool for analysis of algorithms.
  - Divide and conquer algorithms are analyzable by recurrences.
  - Substitution method is the most general method for solving the recurrences by Guess the form of solution, Verify by induction, and Solve for constants.
  - A recursion tree models the costs of a recursive execution of an algorithm.
  - The recursion tree method is good for generating guesses for the substitution method.
  - The recursion tree method can be unreliable, just like any method that uses ellipses.
  - The recursion tree method promotes intuition however.
  - The Master method applies to recurrences of the form  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
- Where  $a \geq 1$ ,  $b > 1$  and  $f$  is asymptotically positive.
- Idea of Master method (through recursion tree):
    - Case - I: The weight increases geometrically from the root to the leaves.  
The leaves hold a constant fraction of the total weight.
    - Case - II: ( $k = 0$ ) The weight is approximately the same on each of the  $\log_b n$  levels.
    - Case - III: The weight decreases geometrically from the root to the leaves. The root holds a constant fraction of the total weight.

## EXERCISES

---

1. Solve the recurrence  $T(n) = 3T(\sqrt{n}) + \log n$  by using change of variables. Your solution should be asymptotically tight.
2. Give asymptotic upper and lower bounds for  $T(n)$  for each of the following recurrences. Assume that  $T(n)$  is a constant for  $n \leq 2$ . Make your bounds as tight as possible and justify your answers.
  - (i)  $T(n) = 2T(n/2) + n^4$
  - (ii)  $T(n) = T(7n/10) + n$
  - (iii)  $T(n) = T(n - 1) + (1/n)$
  - (iv)  $T(n) = T(n - 2) + 2 \log n$
3. Use a recursion tree to determine a good asymptotic upper bound on the recurrence  $T(n) = 4T\left(\frac{n}{2} + 2\right) + n$ . Verify your answer by using substitution method.
4. Find the asymptotic solution of recurrence relation  $T(n) = 3T(n/4) + cn^2$  using recursion tree.
5. Use a recursion tree to determine the asymptotic upper bound on the recurrence  $T(n) = T(n - 1) + T(n/2) + n$ . Use the substitution method to verify your answer.
6. Find the asymptotic solution of the recurrence relation  $T(n) = T\left(\frac{n}{100}\right) + T\left(\frac{99n}{100}\right) + cn$  (where  $c > 0$ ) using recursion tree method.
7. Use the master method to show that the solution to the binary search recurrence  $T(n) = T(n/2) + \theta(1)$  is  $T(n) = \theta(\log n)$ .
8. Can the Master method be applicable to the recurrence  $T(n) = 2T(n/2) + n (\log n)$ ? Why or why not?
9. Solve the below recurrences.
  - (i)  $T(n) = 5T(n/5) + \frac{n}{\log n}$
  - (ii)  $T(n) = \sqrt{n}T(\sqrt{n}) + n$
10. Solve the following recurrence relations and give a  $\theta$  bound for each of them
  - (i)  $T(n) = T(n - 1) + n^c$ , where  $c \geq 1$  is a constant.
  - (ii)  $T(n) = T(n - 1) + c^n$ , where  $c > 1$  is a constant.
11. You now notice that  $ad + bc$  can be computed as  $(a + b)(c + d) - ac - bd$ . Why is this advantageous? Write and solve a recurrence for the running time of the modified algorithm.  
**(Hints :  $T(n) = 3T(n/2) + \theta(n) = \theta(n^{\log_2 3})$ ) .**

# BINARY SEARCH



## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- Practical implementation of binary search
- Binary search algorithm and its running time
- Limitations of binary search

# Chapter 7 *BINARY SEARCH*

## INSIDE THIS CHAPTER

- 7.1 Introduction
- 7.2 Algorithm Steps
- 7.3 Pros and Cons of Binary Search

### **7.1 INTRODUCTION**

---

Suppose DATA is an array is sorted in increasing numerical order or, equivalent, alphabetically. Then there is an extremely efficient searching algorithm, called binary search, which can be used to find the location LOC of the given ITEM of information in DATA. Before formally discussing the algorithm, we indicate the general idea of this algorithm by means of an idealized version of a familiar everyday example.

Suppose one wants to find the location of some name in a telephone directory (or some word in a dictionary). Obviously, one does not perform a linear search. Rather, one opens the directory in the middle to determine which half contains the name being sought. Then one opens that half in the middle to determine which quarter of the directory contains the name. Then one opens that quarter in the middle to determine which eighth of the directory contains the name. And so on. Eventually, one finds the location of the name, since one is reducing (very quickly) the number of possible locations for it in the directory.

The binary search algorithm applied to our array DATA works as follows. During each stage of our algorithm, our search for ITEM is reduced to a segment of elements of DATA:

DATA[BEG], DATA[BEG + 1], DATA[BEG + 2] ,..., DATA [END].

Here the variable BEG and END denote, respectively, the beginning and end locations of the segment under consideration. The algorithm compares ITEM with the middle element DATA [MID] of the segment where MID is obtained.

$$MID = \left[ \frac{BEG + END}{2} \right]$$

If  $\text{DATA}[\text{MID}] = \text{ITEM}$ , then the search is successful and we get  $\text{LOC} = \text{MID}$ . Otherwise the new segment of DATA is obtained as follows.

(a) If  $\text{ITEM} < \text{DATA}[\text{MID}]$ , then ITEM can be present in the left half of the segment:

$\text{DATA}[\text{BEG}], \text{DATA}[\text{BEG} + 1], \dots, \text{DATA}[\text{MID} - 1]$

So reset  $\text{END} = \text{MID} - 1$  and begin searching again.

(b) If  $\text{ITEM} > \text{DATA}[\text{MID}]$ , then ITEM can be present in the right half of the segment:

$\text{DATA}[\text{MID} + 1], \text{DATA}[\text{MID} + 2], \dots, \text{DATA}[\text{END}]$

So reset  $\text{BEG} = \text{MID} + 1$  and begin searching again.

Initially, we begin with the entire array DATA; i.e., we begin with  $\text{BEG} = 1$  and  $\text{END} = n$ , or, more generally, with  $\text{BEG} = \text{LB}$  and  $\text{END} = \text{UB}$ . ITEM is not DATA, then eventually we get  $\text{END} < \text{BEG}$ .

This condition signals that the search is unsuccessful, and in such a case we assign  $\text{LOC} = \text{NULL}$ . Here NULL is a value that lies outside the set of indices of DATA.

## 7.2 ALGORITHM STEPS

---

Here we take DATA is a sorted array with lower bound LB and upper bound UB and ITEM is a given item of information. The variables BEG, END and MID denote the beginning, end and middle locations respectively of the segment of elements of DATA. The algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

```
BINARY (DATA, LB, UB, ITEM, LOC)
1. BEG = LB, END = UB and MID = ⌊(BEG + MID)/2⌋ // Initialization
2. Repeat steps 3 and 6 while BEG ≤ END and DATA [MID] ≠ ITEM
3. If ITEM < DATA [MID], then:
4.   Set END = MID - 1
5. else set BEG = MID + 1
6. Set MID = ⌊(BEG + END)/2⌋
7. If DATA [MID] = ITEM then :
8.   Set LOC = MID
9. else set LOC = NULL
10. Exit
```

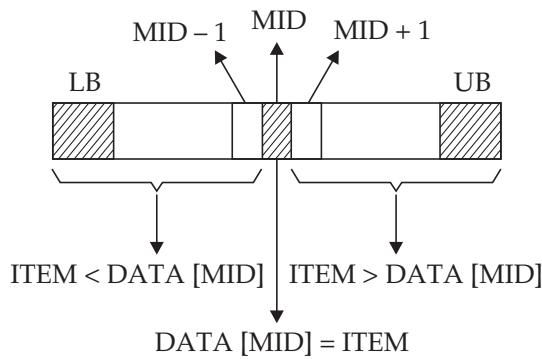


Fig. 7.1. Binary search.

### Running Time

The complexity is measured by the number  $f(n)$  of comparisons to locate ITEM in DATA where DATA contains  $n$  elements. Observe that each comparison reduces the sample size to half. Hence, we require at most  $f(n)$  comparisons to locate ITEM where  $2^{f(n)} > n$ .

$$\text{So } f(n) = \lfloor \log_2 n \rfloor + 1.$$

Hence the running time for the worst case is  $\theta(\log n)$ . This is also approximately equal to the average case running time.

## 7.3 PROS AND CONS OF BINARY SEARCH

---

### Pros (Advantages)

- Binary search is very efficient (e.g., it requires only about 20 comparisons with an initial list of 1000000 elements. As  $2^{10} = 1024 > 1000$ , hence  $2^{20} > 1000^2 = 1000000$ .)

### Cons (Limitations)

- Algorithm has two conditions:
  - (1) the list must sorted and
  - (2) one must have direct access to the middle element in any sublist.

This implies that one must essentially use a sorted array to hold the data, which is very expensive when there are many insertions and deletions.

**Example 7.1** Search the Item = 41 by using Binary search from the sorted array A. Find the time complexity of this operation to perform.

	1	2	3	4	5	6	7	8	9	10	11
A :	12	23	31	34	41	45	56	61	65	75	80

**Solution:**

**Step 1.** Initially BEG = 1 and END = 11.

$$\text{Hence } \text{MID} = \left\lfloor \frac{(1+11)/2}{2} \right\rfloor = 6 \text{ and } A[\text{MID}] = 15$$

1	2	3	4	5	6	7	8	9	10	11
A : 12	23	31	34	41	45	56	61	65	75	80

**Step 2.** Since  $41 < 45$ , END has its value changed by  $\text{END} = \text{MID} - 1 = 5$ .

$$\text{Hence, } \text{MID} = \left\lfloor \frac{1+5}{2} \right\rfloor = 3 \text{ and so } A[\text{MID}] = 31$$

1	2	3	4	5	6	7	8	9	10	11
A : 12	23	31	34	41	45	56	61	65	75	80

**Step 3.** Since  $41 > 31$ , BEG has its value changed by  $\text{BEG} = \text{MID} + 1 = 4$ .

$$\text{Hence, } \text{MID} = \left\lfloor \frac{4+5}{2} \right\rfloor = 4 \text{ and so } A[\text{MID}] = 34$$

1	2	3	4	5	6	7	8	9	10	11
A : 12	23	31	34	41	45	56	61	65	75	80

**Step 4.** Since,  $41 > 34$ , BEG has its value changed by  $\text{BEG} = \text{MID} + 1 = 5$

$$\text{Hence, } \text{MID} = \left\lfloor \frac{5+5}{2} \right\rfloor = 5 \text{ and so } A[\text{MID}] = 41$$

We found the ITEM in location LOC = MID = 5

Time complexity of the Binary search on array A =  $\Theta(\log n)$ .

**Example 7.2** Search the ITEM = 85 by using Binary search from the sorted array A. Find the time complexity of this operation to perform.

1	2	3	4	5	6	7	8	9	10	11	12	13
A : 11	22	30	33	40	44	55	60	66	77	80	88	99

**Solution:**

**Step 1.** Initially BEG = 1 and END = 13, MID = 7 and A[MID] = 55

**Step 2.** Since  $85 > 55$ , BEG has its value changed by  $\text{BEG} = \text{MID} + 1 = 8$ .

$$\text{Hence } \text{MID} = \left\lfloor \frac{(8+13)}{2} \right\rfloor = 10 \text{ and so } A[\text{MID}] = 77$$

**Step 3.** Since,  $85 > 77$ , BEG has its value changed by  $BEG = MID + 1 = 11$ .

$$\text{Hence } MID = \left\lfloor \frac{(11+13)}{2} \right\rfloor = 12 \text{ and so } A[MID] = 88$$

**Step 4.** Since  $85 < 88$ , END has its value changed by  $END = MID - 1 = 11$ .

$$\text{Hence, } MID = \left\lfloor \frac{(11+11)}{2} \right\rfloor = 11 \text{ and so } A[MID] = 80$$

As  $85 > 80$ , BEG has its value changed by  $BEG = MID + 1 = 12$ .

But now  $BEG > END$ . So we conclude that ITEM does not belong to array A.

Running time of this operation =  $\theta(\log n)$ .

## CHAPTER NOTES

---

- Binary search is a divide and conquer technique.
- Binary search technique is always applied on the sorted list and the running time of the sorted list depends on the algorithm which is nothing but  $\theta(\log n)$ .

## EXERCISES

---

1. Write a divide and conquer algorithm for Binary search and find its time complexity.
2. Is Binary search is an efficient searching technique? Justify your answer. State the limitations of Binary search.
3. Search the ITEM (or ELEMENT) = 47 by using Binary search from the given array A.

$$A = \langle 16, 23, 27, 40, 47, 49, 72, 75, 80, 89, 95 \rangle$$

State the time complexity of the entire operation.

4. Search the ITEM = 25 by using Binary search from the given array A.

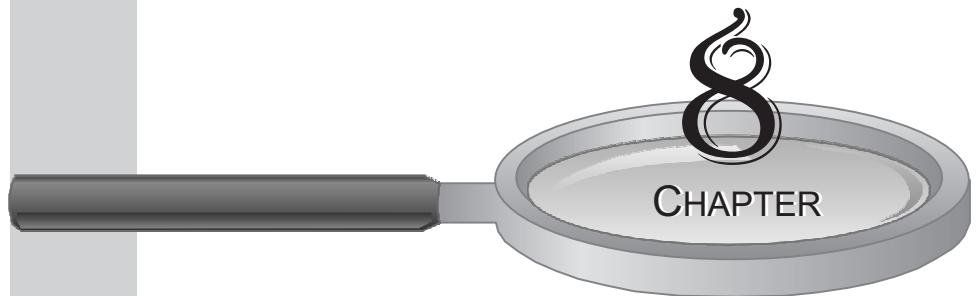
$$A = \langle 35, 16, 20, 25, 31, 9, 45, 65 \rangle$$

State the time complexity of the entire operation.

(**Hints:** 1st sort the array then search. running time = sorting + searching)

5. Explain why the complexity of testing the binary search algorithm in  $2n + 1$  for each  $n$ , independent of the actual array elements?
6. Write a recursive version of Binary search algorithm and find its time complexity.

# QUICKSORT



## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- Quicksort algorithm working procedure
- Advantages and disadvantages of quicksort
- Random quicksort
- Analysis of quicksort

# Chapter 8      *QUICKSORT*

## INSIDE THIS CHAPTER

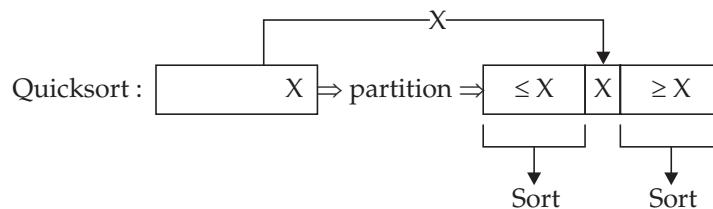
- |                                |                         |
|--------------------------------|-------------------------|
| 8.1 Introduction               | 8.2 Quicksort Algorithm |
| 8.3 Pros and Cons of Quicksort | 8.4 Random Quicksort    |

## **8.1 INTRODUCTION**

Quicksort is another sorting algorithm based on divide-and-conquer paradigm. As opposed to mergesort and heap sort, quicksort has a relatively bad worst case running time of  $\theta(n^2)$ . Nevertheless, quicksort has proved to be very fast in practice, hence the name. Theoretical evidence for this behaviour can be provided by an average case analysis.

Basically, quicksort works as follows:

1. If the input array has less than two elements, there is nothing to do. Otherwise do the following subroutine : pick a particular key called the pivot and divide the array into two subarrays, of which the first only contains items whose key is smaller than or equal to the pivot and the second only items whose key is greater than or equal to the pivot.



**Fig. 8.1.** Choosing pivot in quicksort.

2. Sort the two subarrays recursively.

Note that, quicksort does most of the work in “divide” step (i.e., in the partitioning routine), whereas in mergesort the dividing is trivial, but the “conquer” step requires most of the work when reassembling the recursively sorted subarrays in the merge subroutine. This is not necessary in quicksort, because after the first step all elements in

the first subarray are smaller than those in the second subarray. A problem, which is responsible for the bad worst-case running time of quicksort, is that the partitioning step is not guaranteed to divide the array into two subarrays of the same size. Indeed, if we implement partitioning in a naive way, it may happen that all items end up in one of the two subarrays, and we have gained nothing.

## 8.2 QUICKSORT ALGORITHM

Here are the three steps of divide-and-conquer process for sorting a typical subarray  $A[p \dots r]$ .

**Divide:** Partition the array  $A[p \dots r]$  into two subarrays  $A[p \dots q-1]$  and  $A[q+1 \dots r]$  such that each element of  $A[p \dots q-1]$  is less than or equal to  $A[q]$ , which in turn, is less than or equal to each of the elements of  $A[q+1 \dots r]$ .

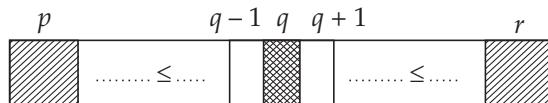


Fig. 8.2. Divide step in quicksort.

**Conquer:** Sort the two subarrays  $A[p \dots q-1]$  and  $A[q+1 \dots r]$  by recursive calls to quicksort.

**Combine:** Because the subarrays are already sorted, no work is needed to combine them; the entire array  $A[p \dots r]$  is now sorted.

### Quicksort Algorithm Steps

```

Quicksort (A, p, r)
1. if (p < r)
2. then q = PARTITION (A, p, r) // break up A w.r.t. pivot element
3. QUICKSORT (A, p, q - 1)
4. QUICKSORT (A, q + 1, r)
    
```

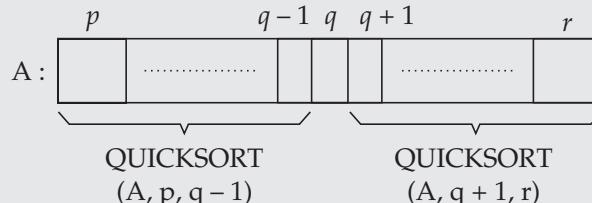


Fig. 8.3. Quicksort.

```

int PARTITION (A, p, r)
1. pivot = A[r]      // pivot is last element in A[p, r]
2. i = p - 1
    
```

```

3. for j = p to r - 1
4.     if (A[j] ≤ pivot)
5.         i = i + 1
6.     swap (A[i], A [j])
7. swap (A[i + 1], A[r]) // swap pivot to the middle
8. return (i + 1) // return final index of pivot

```

### Running Time

**(i) Worst case:** The worst case for quicksort arises when the array is always partitioned into subarrays of sizes 1 and  $n - 1$  because in this case we get a recurrence  $T(n) = T(n - 1) + T(1) + \theta(n) = T(n - 1) + \theta(n)$ , which implies  $T(n) \in \Theta(n^2)$ . Unfortunately this may happen if the input array A is already sorted, which is not uncommon for some applications of sorting.

**(ii) Best case:** The best arises when the array is always split in the middle. Then, we get the same recurrence as for merge sort.

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \theta(n)$$

which implies  $T(n) \in \Theta(n \log n)$

**(iii) Average case:** Fortunately, the typical or “average” case is much closer to the best case than to the worst case. A mathematically complicated analysis shows that for random arrays with each permutation of the keys being equally likely, the average running time of quicksort is in  $\Theta(n \log n)$ .

## 8.3 PROS AND CONS OF QUICKSORT

---

### Pros (Advantages)

- One advantage of parallel quicksort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sub list is available for it to work on and it does not communicate with other threads. When all threads are complete, sort is done.
- All comparisons are being done with a single pivot value, which can be stored in a register.
- The list is being traversed sequentially, which produces very good locality of reference and cache behaviour of arrays.

### Cons (Disadvantages)

- Auxiliary space used in the average case for implementing recursive function calls in  $\Theta(\log n)$  and hence provide to be a bit space costly, especially when it comes to large data sets.
- Its worst case has a time complexity of  $\Theta(n^2)$  which can prove very fatal for large data sets.

- It achieves its worst case time complexity on data sets that are common in practice: sequences that are already sorted (or mostly sorted).

**Example 8.1** Sort the array  $A = \langle 6, 12, 11, 5, 7, 9, 10, 8 \rangle$  using quicksort.

**Solution:** The array A becomes

1    2    3    4    5    6    7    8	
A : [6   12   11   5   7   9   10   8]	$p = 1, r = 8$

**Step 1:** pivot =  $A[8] = 8, i = 0$

When  $j = 1$ :

1    2    3    4    5    6    7    8	
A : [6   12   11   5   7   9   10   8]	

$i$        $j$        $p$        $r$       pivot

**Step 2:** When  $j = 2$ :

1    2    3    4    5    6    7    8	
A : [6   12   11   5   7   9   10   8]	

$i$        $j$        $p$        $r$       pivot

**Step 3:** When  $j = 3$ :

1    2    3    4    5    6    7    8	
A : [6   12   11   5   7   9   10   8]	

$i$        $j$        $p$        $r$       pivot

**Step 4:** When  $j = 4$ :

1    2    3    4    5    6    7    8	
A : [6   12   11   5   7   9   10   8]	

$i$        $j$        $p$        $r$       pivot

**Step 5:** When  $j = 5$ :

1    2    3    4    5    6    7    8	
A : [6   5   11   12   7   9   10   8]	

$p$        $i$        $j$        $r$       pivot

**Step 6:** When  $j = 6$ :

A :	1	2	3	4	5	6	7	8
	6	5	7	12	11	9	10	8
	$p$	$i$		$j$			$r$	pivot

**Step 7:** When  $j = 7$ :

A :	1	2	3	4	5	6	7	8
	6	5	7	12	11	9	10	8
	$p$	$i$		$j$		$r$		pivot

**Step 8:** Exchange  $A[i + 1]$  with  $A[r]$  i.e.,  $A[8]$

1	2	3	4	5	6	7	8
6	5	7	8	11	9	10	12
$p$	$i$	pivot			$r$		

**Example 8.2** What value of  $q$  does PARTITION returns when all element in the array  $A[p \dots r]$  have the same value? Modify Partition so that  $q = \left\lfloor \frac{(p+r)}{2} \right\rfloor$  when all elements in the array  $A[p \dots r]$  have the same value?

**Solution:** Given: the array A contains the elements having same value.

Take an example:	$p$	$p + 1$	$q - 1$	$q$	$q + 1$	$r$
	5	5	.....	5	5	5

By applying PARTITION, it will return  $q = \left\lfloor \frac{(p+r)}{2} \right\rfloor$

### QUICKSORT (A, p , r)

1. if ( $p < r$ )
2.  $q = \text{PARTITION } (A, p, r)$
3.  $\text{QUICKSORT } (A, p, q - 1)$
4.  $\text{QUICKSORT } (A, q + 1, r)$

### int PARTITION (A, p, r)

1. counter 1 = 1, counter 2 = 1,  $n = r - p + 1$
2. pivot =  $A[p]$
3.  $i = p$
4. for  $j = p + 1$  to  $r$
5. do if ( $A[j] < \text{pivot}$ )
6. then counter 2 = counter 2 + 1

```

7.  i = i + 1
8.  swap (A[i], A[j])
9.  if (A[j] = pivot)
10. then counter 1 = counter 1 + 1
11. i = i + 1
12. swap (A[i], A[j])
13. if (counter 1 = n)
14. then return ⌊  $\frac{(p+r)}{2}$  ⌋
15. else swap (A[i], A[p])
16. return i

```

**Example 8.3** How would you modify QUICKSORT to sort into nonincreasing order?

**Solution:** QUICKSORT ( $A, p, r$ )

```

1.  if (p < r)
2.  do q = PARTITION (A, p, r)
3.  QUICKSORT (A, p, q - 1)
4.  QUICKSORT (A, p, q + 1, r)

int PARTITION (A, p, r)
1.  pivot = A[r]
2.  i = p - 1
3.  for j = p to r - 1
4.  if (A[j] ≥ pivot)
5.  i = i + 1
6.  swap (A[i], A[j])
7.  else swap (A[i + 1], A[r])
8.  return (i + 1)

```

## 8.4 RANDOM QUICKSORT

---

In random quicksort, instead of using  $A[r]$  as the pivot, we will select a randomly chosen element from the subarray  $A[p \dots r]$ . We will do so by first exchanging (swap) element  $A[r]$  with an element chosen at random from  $A[p \dots r]$ . By applying randomization technique (called **random sampling**) the range  $p, \dots, r$ , we ensure that the pivot element =  $A[r]$  is equally likely to be any one of  $r - p + 1$  elements in the subarray.

#### 8.4.1 Algorithm for Random-Quicksort

```
RANDOM-QUICKSORT (A, p, r)
1. if (p < r)
2. do q = RANDOM-PARTITION (A, p, r)
3. RANDOM-QUICKSORT (A, p, q - 1)
4. RANDOM-QUICKSORT (A, q + 1, r)

int RANDOM-PARTITION (A, p, r)
1. i = RANDOM (p, r) // midsquare algorithm
2. swap (A[r], A[i])
3. return PARTITION (A, p, r)
```

#### 8.4.2 Algorithm for (Random) Quicksort

##### Worst-case Analysis

Let us begin by considering the worst case performance, as it is easier than the average case (expected running time). Suppose that, we are sorting an array of size  $n$ ,  $A[1 \dots n]$ , and further suppose that the pivot that we select is of rank  $q$ , for some  $q$  in the range 1 to  $n$ . It takes  $\Theta(n)$  time to do partitioning and other overhead, we make two recursive calls. The first is to the subarray  $A[p \dots q - 1]$  which has  $q - 1$  elements, and the other is to subarray  $A[q + 1 \dots n]$  which has  $r - (q + 1) + 1 = r - q$  elements. So if we ignore  $\Theta$ (as usual) we get the recurrence:

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

This depends on the value of  $q$ . To get the worst case, we maximize over all possible values of  $q$ . As a basis we have that  $T(0) = T(1) = \Theta(1)$ . Putting this together we have

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max_{1 \leq q \leq n} (T(q - 1) + T(n - q) + n) & \text{otherwise} \end{cases}$$

Recurrences that have max's and min's embedded in them are very messy to solve. The key is to determine which value of  $q$  gives the maximum. In this case, the worst case happens at either of the extremes. If we expand the recurrence in the case  $q = 1$  we got:

$$\begin{aligned} T(n) &\leq T(0) + T(n - 1) + n = 1 + T(n - 1) + n \\ &= T(n - 1) + (n + 1) \\ &= T(n - 2) + n + (n + 1) \\ &= T(n - 3) + (n - 1) + n + (n + 1) \\ &= \dots\dots \\ &= T(n - k) + \sum_{i=-1}^{k-2} (n - i) \end{aligned}$$

For the basis,  $T(1)$ , we set  $k = n - 1$  and get

$$\begin{aligned} T(n) &\leq T(1) + \sum_{i=-1}^{n-3} (n-i) \\ &= 1 + (3 + 4 + 5 + \dots + (n-1) + n + (n+1)) \\ &\leq \sum_{i=-1}^{n+1} i = \frac{(n+1)(n+2)}{2} \in \Theta(n^2) \end{aligned}$$

In fact a more careful analysis reveals that it is  $\Theta(n^2)$  in this case.

### Average Case Analysis/Expected Running Time

Ideal partition around the random element

- Running time is independent of input order.
- No specific assumptions need to be made about the input distribution.
- No specific input elicits the worst case behaviour.
- The worst case is determined only by of a random number generator.

Let  $T(n)$  = the random variable for the running time of randomized quicksort of an input of size  $n$  assuming random numbers are independent.

For  $k = 0, 1, 2, \dots, n-1$  define the indicator random.

$$x_k = \begin{cases} 1, & \text{if partition generates } k, k-1 \text{ split} \\ 0, & \text{otherwise} \end{cases}$$

$E[x_k] = 1/n$  (E stands for expected)

$$\begin{aligned} T(n) &= \begin{cases} T(0) + T(n-1) + \theta(n) & \text{if } 0 \ n-1 \text{ split} \\ T(1) + T(n-2) + \theta(n) & \text{if } 1 \ n-2 \text{ split} \\ T(n-1) + T(0) + \theta(n) & \text{if } n-1 \ 0 \text{ split} \end{cases} \\ T(n) &= \sum_{k=0}^{n-1} x_k (T_k + T(n-k-1) + \theta(n)) \quad \dots(i) \end{aligned}$$

$$\begin{aligned} \text{Expected } (T(n)) &= E\left[ \sum_{k=0}^{n-1} x_k (T(k) + T(n-k-1) + \theta(n)) \right] \\ &= E(x_k) * E\left[ \sum_{k=0}^{n-1} T(k) + T(n-1-k) + \theta(n) \right] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E(T(k)) + \frac{1}{n} \sum_{k=0}^{n-1} T(n-1-k) + \frac{1}{n} \sum_{k=0}^{n-1} \theta(n) \\ &= \frac{2}{n} \sum_{k=0}^{n-1} E(T(k)) + \theta(n) \quad \dots(ii) \end{aligned}$$

$$\text{We can rewrite this as, } E(T(n)) = \frac{2}{n} \sum_{k=2}^{n-1} E(T(k)) \theta(n) \quad \dots(iii)$$

By substitution method, we have to solve this recurrence. So let us assume

$$E(T(n)) = O(n \log n) \text{ (asymptotic upper bound)}$$

So,  $E(T(n)) \leq an \log n$  for constant  $a > 0$ .

$$\begin{aligned} \Rightarrow \quad & \sum_{k=2}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \\ \Rightarrow \quad & \sum_{k=2}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \end{aligned} \quad \dots(iv)$$

$$\text{Again from recurrence (iii), } E(T(n)) = \frac{2}{n} \sum_{k=2}^{n-1} E(T(k)) + \theta(n)$$

$$\begin{aligned} & \leq \frac{2}{n} \sum_{k=2}^{n-1} ak \log k + \theta(n) \\ & \leq \frac{2}{n} \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \theta(n) \quad \text{from recurrence (iv))} \\ & \leq \frac{2a}{n} \cdot \frac{1}{2} n^2 \log n - \frac{2a}{n} \cdot \frac{n^2}{8} + \theta(n) \\ & \leq an \log n - \frac{an}{4} + \theta(n) \\ & \leq an \log n - \left( \frac{an}{4} - \theta(n) \right) \end{aligned} \quad \dots(v)$$

Required                      Residual

This implies  $E(T(n)) \leq an \log n$ . So we conclude that in the average case Quicksort runs in  $\theta(n \log n)$ .

### Best Case Analysis

Intuition: Perfectly balanced splits – each partition gives at most equal split that is  $\lfloor n/2 \rfloor$  elements in one,  $\lceil n/2 \rceil - 1$  elements in the other, and the pivot

$$\begin{aligned} \text{So} \quad T(n) & \leq T(n/2) + T(n/2) + \theta(n) \\ & = 2T(n/2) + \theta(n) \end{aligned}$$

Using Master theorem:

$$a = 2, b = 2, f(n) = n$$

$$\log_b a = 1$$

$$n^{\log_b a} = n^1 = n$$

Comparing  $f(n)$  and  $n^{\log_b a}$ :  $n = \theta(n)$  or  $f(n) = \theta(n^{\log_b a})$

So, we are in Case-II of Master theorem, and  $T(n) = \theta(n^{\log_b a} \log n) = \theta(n \log n)$

Therefore, best cas running time is  $\theta(n \log n)$ .

## CHAPTER NOTES

---

- Quicksort is a great general purpose sorting algorithm.
- Quicksort is popular algorithm for implementation because its actual performance is so good.
- Quicksort (like Mergesort) is not formally an in-place algorithm, because it does make use of a recursion stack. In Mergesort and in the expected case of Quicksort, the size of the stack is  $O(\log n)$ , so this is not really a problem.
- Quicksort is not stable, since it exchanges non-adjacent elements.
- If stability is not required, quicksort provides a very attractive alternative to mergesort.
  - Quicksort is likely to run a bit faster than mergesort-perhaps 1.2 to 1.4 times as fast.
  - Quicksort requires less memory than mergesort.
  - A good implementation of quicksort is probably easier to code than a good implementaion of mergesort.
- Quicksort can benifit substantially form code tuning.
- Quicksort behaves well even with the code and virtual memory.
- With randomized version of Quicksort (pivot element randomly), the standard deviation in the number of comparisons is also small.
- Worst case running time of Quicksort is  $\theta(n^2)$ , the average case running time is  $\theta(n \log n)$  and the best case is also  $\theta(n \log n)$ .

## EXERCISES

---

1. Illustrate the operation of PARTITION of the array  $A = \langle 7, 9, 12, 5, 11, 8, 4, 3, 15, 1 \rangle$ .
2. Analyze the worst case time complexity of Quicksort in detail.
3. Explain the working procedure of Quicksort.
4. Rewrite the Quicksort algorithm taking the first element in the array as pivot element. Does this hamper to the time complexity of Quicksort? What are the worst case, best case and average case running times in this cases?

5. Consider the enhanced version print Quicksort of quicksort displayed as Algorithm below. Line 1 simply prints the keys of  $A[i], \dots, A[j]$  on a separate line of the standard output.

Algorithm print Quicksort ( $A, i, j$ )

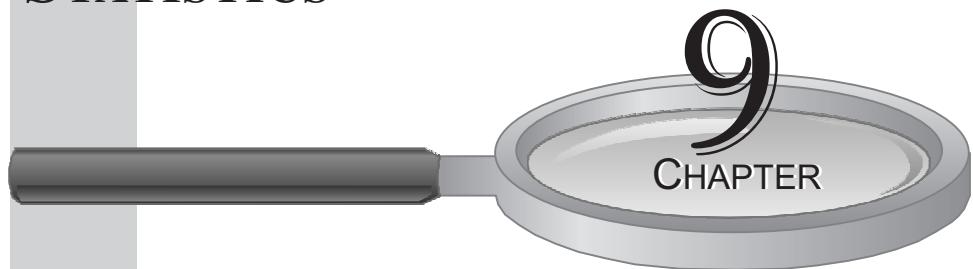
1. print  $A[i \dots j]$
2. if  $i < j$  then
3. split  $\leftarrow$  partition ( $A, i, j$ )
4. quicksort ( $A, i, \text{split}$ )
5. quicksort ( $A, \text{split} + 1 \dots j$ )

Let  $A = \langle 5, 0, 3, 11, 9, 3, 4, 6 \rangle$

What does print Quicksort ( $A, 0, 7$ ) print?

6. Derive the expected running time of the quicksort.
7. Our recursive implementation of Quicksort does not sort in place. Why? How much memory space does it use? Describe a non-recursive in place version of quicksort.

# ORDER STATISTICS



## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- $i^{\text{th}}$  order statistic
- Selection problem and how to solve it
- Maximum and minimum
- $i^{\text{th}}$  smallest element algorithm and its analysis
- Random-select and random-partition

# Chapter 9 ORDER STATISTICS

## INSIDE THIS CHAPTER

- 9.1 Introduction
- 9.2 The Selection Problem
- 9.3 Algorithm for Finding Maximum and Minimum
- 9.4 Algorithm for Finding ith Smallest Element

## 9.1 INTRODUCTION

This chapter addresses the problem of selecting the  $i$ th order statistic from a set of  $n$  distinct numbers. We assume for convenience that the set contains distinct number, although virtually everything that we do extends to the situation in which a set contains repeated values.

### Few Basic Terms

- **ith order statics** is the  $i$ th smallest element of a set of  $n$  elements.
- The **minimum** is the first order statistic ( $i = 1$ ).
- The **maximum** is the  $n$ th order statistic ( $i = n$ ).
- A **median** is the “halfway point” of the set.
- When  $n$  is odd, the median is unique, at  $i = (n + 1)/2$
- When  $n$  is even, there are two medians
- The **lower median**, at  $i = n/2$  and
- The **upper median**, at  $i = \frac{n}{2} + 1$
- We mean lower median when we use the phrase “the median”.

## 9.2 THE SELECTION PROBLEM

---

We formally define the selection problem as follows:

**Input:** A set A of  $n$  distinct numbers and a number  $i$ , with  $1 \leq i \leq n$ .

**Output:** The element  $x \in A$  that is larger than exactly  $i - 1$  other elements in A. In other words, the  $i$ th smallest element of A.

However, the selection problem can be solved in  $O(n \log n)$  time. Method is:

- Sort the numbers using an  $O(n \log n)$  time algorithm, such as heapsort or mergesort.
- Then return the  $i$ th element in the sorted array. But there are faster algorithms, however.
- First, we will look at the problem of selecting the minimum and maximum of a set of elements.
- Then, we will look at a simple general selection algorithm with a time bound of  $O(n)$  in the average case.
- Finally, we will look at a more complicated general selection algorithm with a time bound of  $O(n)$  in the worst case.

## 9.3 ALGORITHM FOR FINDING MAXIMUM AND MINIMUM

---

We can easily obtain an upper bound of  $n - 1$  comparisons for finding the minimum of a set of  $n$  elements.

- Examine each element in turn and keep track of the smallest one.
- This is the best we can do, because each element, except the minimum, must be compared to a smaller element at least once.

The following pseudocode finds the minimum element in array  $A[1 \dots n]$ .

```
Minimum (A, n)
1. min ← A[1]
2. for i ← 2 to n
3. do if min > A[i]
4. then min ← A[i]
5. return min
```

Algorithm for MAXIMUM (A, n) can be found in exactly the same way by replacing the  $>$  with  $<$  in the above algorithm.

### 9.3.1 Finding Simultaneous Maximum and Minimum

Some applications need both the minimum and maximum of a set of elements. For example, a graphics program may need to scale a set of  $(x, y)$  data to fit onto a rectangular display. To do so, the program must first find the minimum and maximum of each coordinate.

A simple algorithm to find the minimum and maximum is to find each one independently. There will be  $n - 1$  comparisons for the minimum and  $n - 1$  comparisons for the maximum, for a total of  $2n - 2$  comparisons. This will result in  $\Theta(n)$  time.

**In fact, at most  $3\lfloor n/2 \rfloor$  comparisons are needed to find both the minimum and maximum.**

- Maintain the minimum and maximum of elements seen so far.
- Don't compare each element to the minimum and maximum separately.
- Process elements in pairs.
- Compare the elements of a pair to each other.
- Then compare the larger element to the maximum so far, and compare the smaller element to the minimum so far.

This leads to only 3 comparisons for every 2 elements. Setting up the initial values for the minimum and maximum depends on whether  $n$  is odd or even.

- If  $n$  is even, compare the first two elements and assign the larger to maximum and the smaller to minimum. Then process the rest of the elements in pairs.
- If  $n$  is odd, set both minimum and maximum to the first element. Then process the rest of the elements in pairs.

#### Analysis of the total number of comparisons

- If  $n$  is even, we do 1 initial comparisons and then  $3(n - 2)/2$  more comparisons.  
Total number of comparisons =  $3(n - 2)/2 + 1 = 3n/2 - 3 + 1 = 3n/2 - 2$
- If  $n$  is odd, we do  $3(n - 1)/2 = 3\lfloor n/2 \rfloor$  comparisons.

In either case, the maximum number of comparisons is  $\leq 3\lfloor n/2 \rfloor$ .

## 9.4 ALGORITHM FOR FINDING $i$ th SMALLEST ELEMENT

---

Selection of the  $i$ th smallest element of the array A can be done in  $\Theta(n)$  time. The function RANDOM-SELECT uses RANDOM-PARTITION from the quicksort algorithm in Chapter-8. RANDOM-SELECT differs from quicksort because it recurses on one side of the partition only.

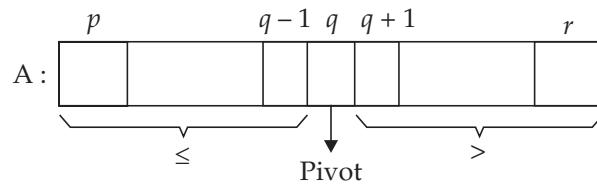
```
RANDOM-SELECT (A, p, r, i) // A[p . . . . . r]
1. if (p = r)
2. then return A[p]
3. q ← RANDOM - PARTITION (A, p, r)
4. k ← q - p + 1
5. if (i = k) // pivot value is the answer.
```

```

6. then return A[q]
7. else if (i < k)
8. then return RANDOM-SELECT (A, p, q-1, i)
9. else return RANDOM-SELECT (A, q+1, r, i-k)

```

After the call to RANDOM-PARTITION, the array is partitioned into two subarray  $A[p \dots q - 1]$  and  $A[q + 1 \dots r]$  along with the pivot element  $A[q]$ .



**Fig. 9.1.** Random-partition.

- The elements of subarray  $A[p \dots q - 1]$  are all  $\leq A[q]$
- The elements of subarray  $A[q + 1 \dots r]$  are all  $> A[q]$ .
- The pivot element is the  $k$ th element of the subarray  $A[p \dots r]$  where  $k = q - p + 1$ .
- If the pivot element is the  $i$ th smallest element (i.e.,  $i = k$ ), simply returns  $A[q]$ . Otherwise, it recurse the subarray containing the  $i$ th smallest element. If  $i < k$ , this subarray is  $A[p \dots q - 1]$ , and we want the  $i$ th smallest element. If  $i > k$ , this subarray is  $A[q + 1 \dots r]$  and since, there are  $k$  elements in  $A[p \dots r]$  that proceed  $A[q + 1 \dots r]$ , we want the  $(i - k)$ th smallest element of this subarray.

#### 9.4.1 Algorithm Analysis

**(i) Worst Case Running Time:**  $\theta(n^2)$ , because, we could be extremely unlucky and always recurse on a subarray that is only 1 element smaller than the previous subarray.

**(ii) Expected Running Time:** RANDOM-SELECT works well on average. Because it is random, no particular input brings out the worst-case behaviour consistently. The running time of RANDOM-SELECT is a random variable that we denote by  $T(n)$ . We obtain an upper bound on  $E[T(n)]$  as follows:

- RANDOM-PARTITION is equally likely to return any element of A as the pivot.
- For each  $k$  such that  $1 \leq k \leq n$ , the subarray  $A[p \dots q]$  has  $k$  elements (all  $\leq$  pivot) with probability  $\frac{1}{n}$ .
- For  $k = 1, 2, \dots, n$ , define indicator random variable.  
 $x_k = 1$  {subarray  $A[p \dots q]$  has exactly  $k$  elements}
- Since  $\text{pr} \{ \text{subarray } A[p \dots q] \text{ has exactly } k \text{ elements} \} = \frac{1}{n}$ , we have  $E[x_k] = \frac{1}{n}$

- When we call RANDOM-SELECT, we don't know if it will terminate immediately with correct answer, recurse on  $A[p \dots q - 1]$ , or recurse on  $A[q + 1 \dots r]$ . It depends on whether the  $i$ th smallest element is less than, equal to, or greater than the pivot element  $A[q]$ .
- To obtain an upper bound, we assume that  $T(n)$  is monotonically increasing and that the  $i$ th smallest element is always in large subarray.
- For a given call of RANDOM-SELECT,  $x_k = 1$  for exactly one value of  $k$ , and  $x_k = 0$  for all other  $k$ .
- When  $x_k = 1$ , the two subarrays have sizes  $k - 1$  and  $n - k$ .
- For a subproblem of size  $n$ , RANDOM-PARTITION takes  $O(n)$  time. Therefore, we have the recurrence.

$$\begin{aligned} T(n) &\leq \sum_{k=1}^n x_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^n x_k \cdot (T(\max(k-1, n-k)) + O(n)) \end{aligned}$$

Taking expected values gives

$$\begin{aligned} E[T(n)] &\leq E \left[ \sum_{k=1}^n x_k \cdot T(\max(k-1, n-k)) + O(n) \right] \\ &= \sum_{k=1}^n E[x_k \cdot T(\max(k-1, n-k))] + O(n) \\ &\quad \text{(linearity of expression)} \\ &= \sum_{k=1}^n E[x_k] \cdot E[T(\max(k-1, n-k))] + O(n) \quad \dots(i) \\ &= \sum_{k=1}^n \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n) \end{aligned}$$

We rely on  $x_k$  and  $T(\max(k-1, n-k))$  being independent of random variables in order to apply equation (i). Looking at the expression  $\max(k-1, n-k)$  we have

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \\ n-k & \text{if } k \leq \lceil n/2 \rceil \end{cases}$$

- If  $n$  is even, each term from  $T(\lceil n/2 \rceil)$  upto  $T(n-1)$  appears exactly twice in the summation.

- If  $n$  is odd, these term appear twice and  $T(\lfloor n/2 \rfloor)$  appears once. Either way,

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k) + O(n)]$$

Solving this recurrence by the method of substitution, we have to follow following steps:

- Guess that  $T(n) \leq cn$  for some constant  $c$  that satisfies the initial condition of the recurrence.
- Assume that  $T(n) = O(1)$  for  $n <$  some constant. We will pick this constant later.
- Also pick a constant  $a$  such that the function described by the  $O(n)$  term is bounded from the above by  $an$  for all  $n > 0$ .
- Using this guess and constants  $c$  and  $a$ , we have

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} ck + an \\ &= \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\left\lfloor \frac{n}{2} \right\rfloor - 1} k \right) + an = \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{\left( \left\lfloor \frac{n}{2} \right\rfloor - 1 \right) \left\lfloor \frac{n}{2} \right\rfloor}{2} \right) + an \\ &\leq \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{\left( \frac{n}{2} - 2 \right) \left( \frac{n}{2} - 1 \right)}{2} \right) + an \\ &= \frac{2c}{n} \left( \frac{n^2 - n}{2} - \frac{\frac{n^2}{4} - \frac{3n}{2} + 2}{2} \right) + an = \frac{c}{n} \left( \frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\ &= c \left( \frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \leq \frac{3cn}{4} + \frac{c}{2} + an \\ &= cn - \left( \frac{cn}{4} - \frac{c}{2} - an \right) \end{aligned}$$

To complete this proof, we choose  $c$  such that  $\frac{cn}{4} - \frac{c}{2} - an \geq 0$

$$\begin{aligned} \frac{cn}{4} - an &\geq \frac{c}{2} \\ n \left( \frac{c}{4} - a \right) &\geq \frac{c}{2} \end{aligned}$$

$$\begin{aligned} n &\geq \frac{\frac{c}{2}}{\frac{c}{4} - a} \\ n &\geq \frac{2c}{c - 4a} \end{aligned}$$

Thus, as long as we assume that  $T(n) = O(1)$  for  $n < \frac{2c}{c - 4a}$ , we have  $E[T(n)] = O(n)$ .

Therefore, we can determine any order statistic in linear time on average.

**Example 9.1** Show that the second smallest of  $n$  elements can be found with  $n + \lceil \log n \rceil - 2$  comparisons in the worst case.

**Solution:** The smallest of  $n$ -numbers can be found with  $n - 1$  comparisons by conducting a tournament as follows: Compare all the numbers in pairs. Only the smaller of each pair could possibly be the smallest of  $n$ , so the problem has been reduced to that of finding the smallest of  $\lceil n/2 \rceil$  numbers. Compare those numbers in pairs, and so on, until there is just one number left, which is the answer.

To see that, this algorithm does exactly  $n - 1$  comparisons, remember that each number except the smallest loses exactly once. To show this more formally, draw a binary tree of the comparisons the algorithm does. The  $n$  numbers are the leaves, and each number that came out smaller in the comparison is parent of the two numbers that were compared. Each non-leaf node of the tree represents a comparison, and there are  $n - 1$  internal nodes in an  $n$ -leaf full binary tree, so exactly  $n - 1$  comparisons are made.

In the search for the smaller number, the second smallest number must have come out smallest in every comparison made with it until it was eventually compared with the smallest. So the second smallest is among the elements that were compared with the smallest during the tournament. To find it, conduct another tournament (as above) to find the smallest of these numbers.

At most  $\lceil \log n \rceil$  (the height of the tree of comparisons) elements were compared with the smallest, so finding the smallest of these takes  $\lceil \log n \rceil - 1$  comparisons in the worst case.

Hence, the total number of comparisons made in the two tournaments was:

$$n - 1 + \lceil \log n \rceil - 1 = n + \lceil \log n \rceil - 2 \text{(in worst case)}$$

**Example 9.2** Show how quicksort can be made in run in  $O(n \log n)$  time in the worst case, assuming that all elements are distinct.

**Solution:** A modification to quicksort that allows it to run in  $O(n \log n)$  time in the worst case uses the deterministic PARTITION algorithm that was modified to take an element to partition around as input parameter.

SELECT takes as array A, the bounds  $p$  and  $r$  of the subarray in A, and the rank  $i$  of an order static, and in time linear in the size of the subarray  $A[p \dots r]$  it returns the  $i$ th smallest element in  $A[p \dots r]$ .

```

BEST-CASE-QUICKSORT (A, p, r)
1. if (p < r)
2. then i ← ⌊(r - p + 1)/2⌋
3. x ← SELECT (A, p, r, i)
4. q ← PARTITION (x)
5. BEST-CASE-QUICKSORT (A, p, q - 1)
6. BEST-CASE-QUICKSORT (A, q + 1, r)

```

For an  $n$ -element array, the largest subarray that BEST-CASE-QUICKSORT recurses on

has  $\frac{n}{2}$  elements. This situation occurs when  $n = r - p + 1$  is even; then the subarray  $A[q + 1 \dots r]$

has  $\frac{n}{2}$  elements, and the subarray  $A[p \dots q - 1]$  has  $\frac{n}{2} - 1$  elements.

Because BEST-CASE-QUICKSORT always recurses on subarrays that are most half the size of the original array, the recurrence for the worst case running time is

$$T(n) \leq 2T(n/2) + \theta(n) = O(n \log n)$$

## CHAPTER NOTES

---

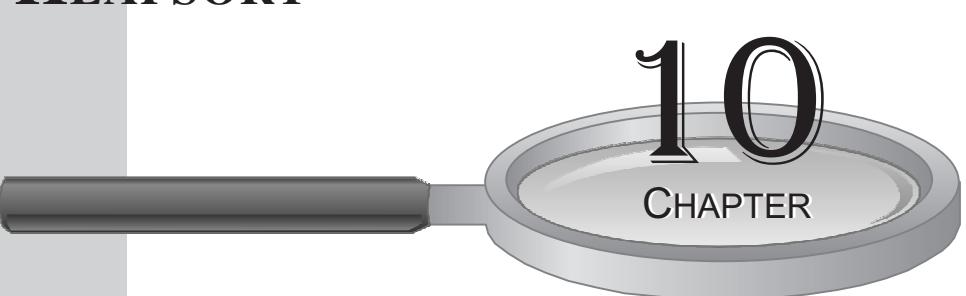
- The  $i$ th order statistic is the  $i$ th smallest element e.g.
  - minimum = 1st order static
  - maximum =  $n$ th order static
  - median ( $s$ ) =  $\lfloor (n + 1)/2 \rfloor$  and  $\lceil (n + 1)/2 \rceil$
- For finding running time for minimum (or maximum) just scan input array and exactly  $n - 1$  comparisons mode.
- At most  $3 \lfloor n/2 \rfloor$  comparisons are needed to find both minimum and maximum.
- The algorithm for finding  $i$ th smallest element has the worst case running time of  $\Theta(n^2)$  and expected running time of  $O(n)$ .

## EXERCISES

---

1. Define the selection problem of order statistics.
2. Write the algorithm for finding maximum element in the given array of  $n$ -elements.
3. Design a divide and conquer algorithm for finding the minimum and the maximum element of  $n$  number using no more than  $3n/2$  comparisons.
4. Write a non-deterministic algorithm to find the  $k^{\text{th}}$  smallest element in the list of  $n$  elements. The  $k^{\text{th}}$  smallest element is the one that would be in position  $k$  if the array were sorted.

# HEAPSORT



10

CHAPTER

## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand :**

- Basic concepts of heapsort
- Heap property
- Max-heap and min-heap
- Max-heapify procedure
- Running time of build-max-heap
- Pros and cons of heapsort

# 10 HEAPSORT

## INSIDE THIS CHAPTER

- |                                      |                                |
|--------------------------------------|--------------------------------|
| 10.1 Introduction                    | 10.2 Heaps                     |
| 10.3 Types of Heaps                  | 10.4 Algorithm for Max-Heapify |
| 10.5 Algorithm for Building Max-Heap | 10.6 Heapsort-Algorithms       |
| 10.7 Pros and Cons of Heapsort       |                                |

## 10.1 INTRODUCTION

Heapsort is a comparison based sorting algorithm to create a sorted array (or list), and is a part of selection sort family. Although somewhat slower in practice on most machines than a well implemented quicksort, it has the advantage of a more favourable worst-case  $O(n \log n)$  runtime.

Heapsort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the partially sorted array. After removing the largest item, it reconstructs the heap, removes the largest remaining item, and places it in the next open position from the end of the partially sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementation requires two arrays—one to hold the heap and the other to hold the sorted elements.

Heapsort inserts the input list elements into a binary heap data structure. The largest value (in a max-heap) or the smallest value (in a min-heap) are extracted until non-remain, the values having been extracted in sorted order. During extraction, the only space required is that needed to store the heap. To achieve constant space overhead, the heap is stored in the part of the input array not yet stored.

Heapsort uses two heap operations : insertion and root deletion. Each extraction places an element in the last empty location of the array. The remaining prefix of the array stored the unsorted elements.

## 10.2 HEAPS

---

Conceptually, a heap is a kind of binary tree. That is a collection of nodes with a “root node” and in which each node can potentially have a “left child node” and a “right child node”. A node is said to be the parent node of its child node. Every node in a binary tree, except for the root node, has exactly one parent node. Figure 10.1 represents different examples of binary tree.

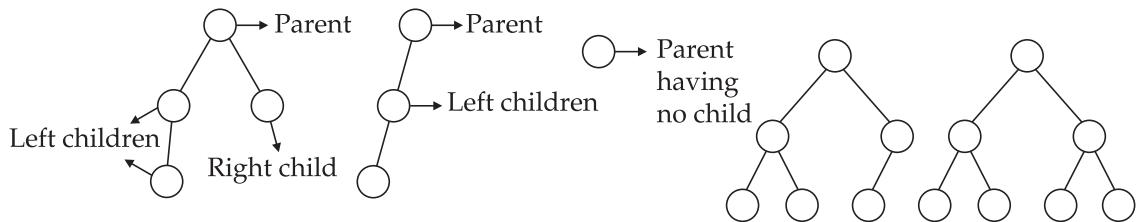


Fig. 10.1. Examples of binary tree.

A heap must satisfy the “**heap property**”, which say that the priority of every node is greater than or equal to the priority of any child nodes of that node. Another way of saying this is that the priority of each non-root node must be less than or equal to the priority of its parent node. Note that the heap property implies that the priority of any node is greater than or equal to the priorities of all its descendant nodes, not just the nodes immediately below it. In particular, the priority of the root node is greater than or equal to the priority of every other node in the heap.

A heap has one more important property: It is a full binary tree. In a full binary tree, there are no missing nodes in the interior of the tree. In other words a binary tree is full (some books refer it as complete binary tree) if at every level of the tree contains  $2^i$  no of nodes, where  $i$  is the level number. Figure 10.2 represents a full binary tree.

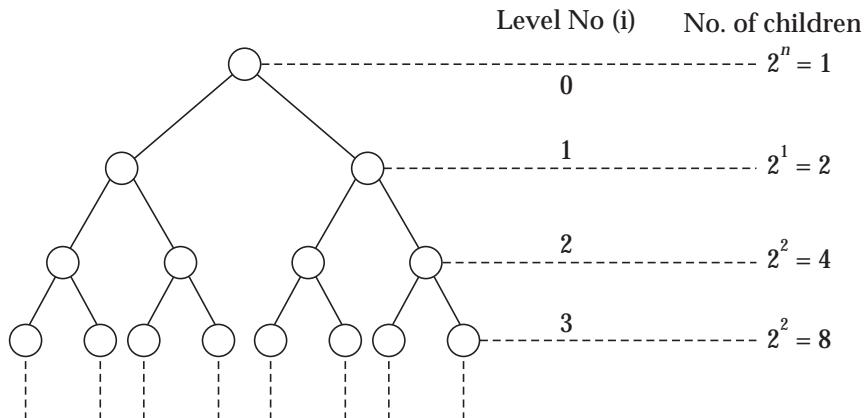


Fig. 10.2. Full binary tree.

If you think of the tree being build up by adding nodes one at a time, then nodes are added level-by-level from top to bottom, and within a level they are added from left to right.

Here, for example, a heap is constructed with twelve items. Only the properties of the items are shown in the nodes in Fig. 10.3:

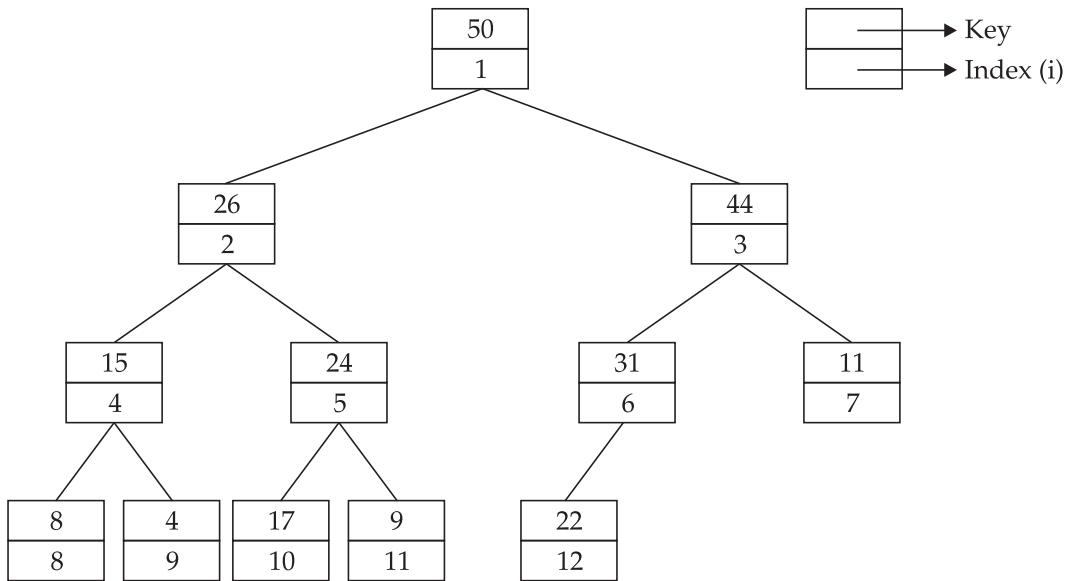


Fig. 10.3. Building a heap.

Now, although a heap is conceptually a binary tree, the fact that it is a full binary tree makes it possible to physically represent the heap as a simple data structure. In fact, heap (or any full binary tree) can be represented as an array. All you have to do is line up the nodes in the array, starting with the root node, then the children of the root, then the grandchildren, and so on. If we do with the above heap, we get the following array as shown in Fig. 10.4.

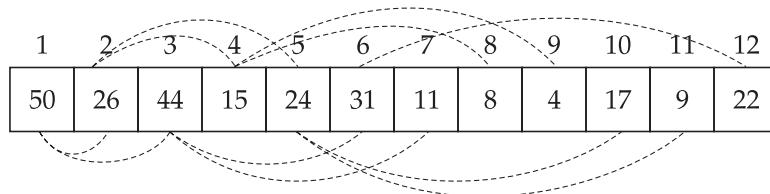


Fig. 10.4. Array representation of heap given in Fig. 10.3.

The important thing about this array is that the structure of the binary tree can be described completely in terms of array: The children of the node at index  $i$  in the array are located in the array at index  $2i$  and  $2i + 1$ , provided these numbers are within the range of indices of the array. For example, the root node is at index 1, and its children are at indices  $2 * 1$  and  $2 * 1 + 1$ , that is at indices 2 and 3. The children of node 2 are node 6 ( $2 * 3$ ) and node 7 ( $2 * 3 + 1$ ). Node 6 has only one child, at position 12 ( $2 * 6$ ); its other potential child, at position 13, lies outside the range of indices in the array.

**Note:** If  $i$  is the index of a non-root node in the array, then the parent of that node in index  $\lceil (i - 1)/2 \rceil$ .

### 10.3 TYPES OF HEAPS

---

There are two kinds of (binary) heaps. They are:

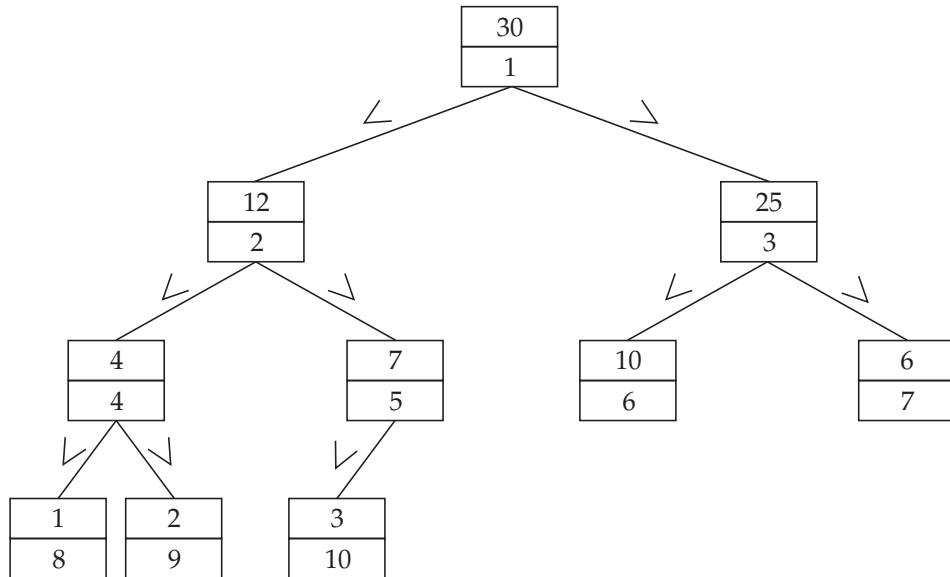
- (i) Maximum Heap (in short “Max-Heap”)
- (ii) Minimum Heap (in short “Min-Heap”)

**(i) Max-Heap:** In a max-heap, the max-heap property is that for every node  $i$  other than the root,

$$A[\text{Parent}(i)] \geq A[i]$$

that is, the value of a node is at most the value of the parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains value no larger than that contains at the node itself.

**Example 10.1** This example shows a max-heap of size 10.



**Fig. 10.5.** Example of max-heap.

To built a max-heap every node of allmost full binary tree must satisfy the rule of max-heap. Applying the rule to a node is a procedure for **MAX-HEAPIFY**. When a max-heapify procedure is applied on every node except the leaf node, the almost complete binary tree is a max-heap.

**(ii) Min-Heap:** A min-heap is organized in the opposite way; the min-heap property is that for every node  $i$  other than the root,

$$A[\text{Parent}(i)] \leq A[i]$$

The smallest element in a min-heap is at the root.

**Example 10.2** In this example we show a min-heap of size 9.

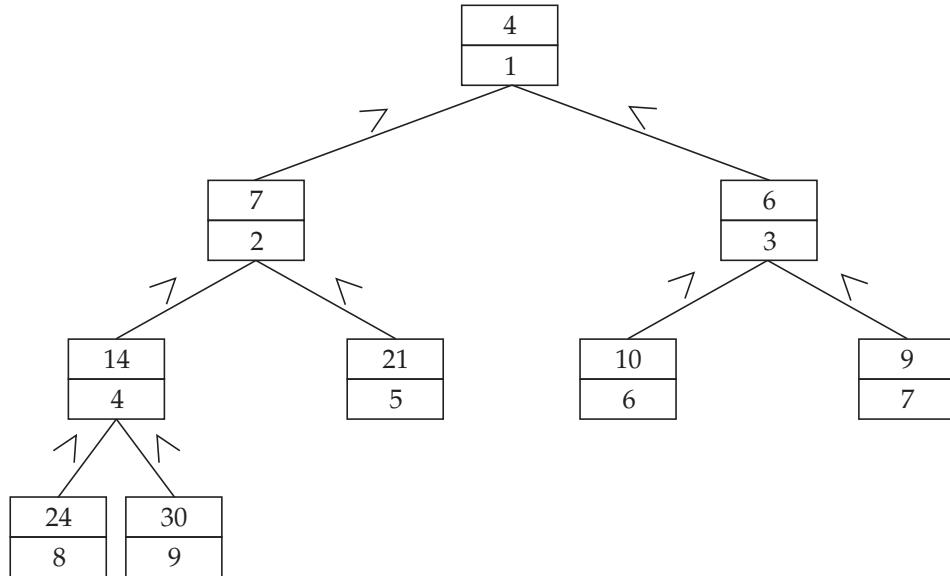


Fig. 10.6. Example of min-heap.

#### 10.4 ALGORITHM FOR MAX-HEAPIFY

**Input :** an array A and index i.

**Output :** Finding the maximum between root, left child and right child and placing maximum at root.

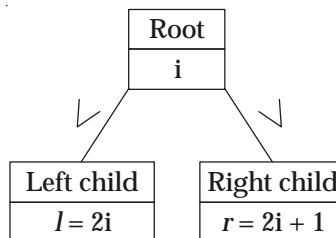


Fig. 10.7. Max-heapify.

```

MAX-HEAPIFY (A, i)
1. n = heapsize (A)
2. l = 2i, r = 2i + 1 // l is the left child and r is the right child
  
```

```

3. if ( $l \leq n$  and  $A[i] < A[l]$ )
4. then maximum =  $l$ 
5. else maximum =  $i$ 
6. if ( $r \leq n$  and  $A[maximum] < A[r]$ )
7. then maximum =  $r$ 
8. if (maximum  $\neq i$ )
9. Then interchange ( $A[i]$ ,  $A[maximum]$ )
10. MAX-HEAPIFY ( $A$ , maximum)

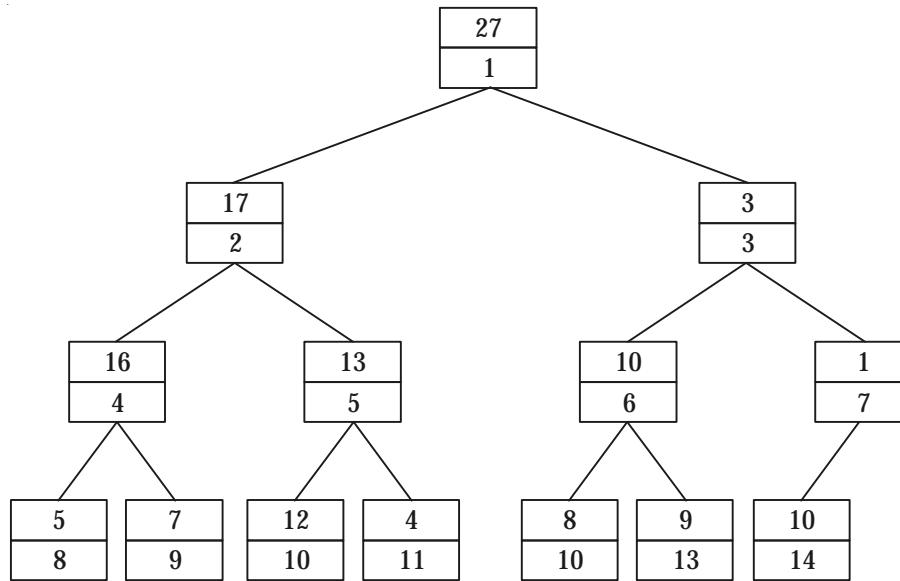
```

### Running Time

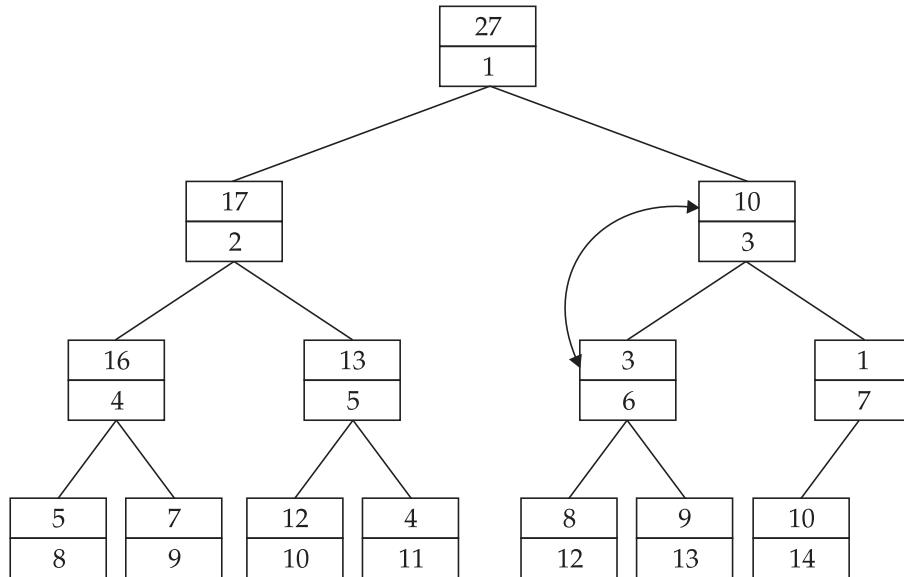
The running time of the MAX-HEAPIFY procedure is  $O(\log n)$  (which is nothing but the height of tree at worst case). Alternatively we can say the running time of MAX-HEAPIFY on a node of height  $h$  is  $O(h)$ .

**Example 10.3** Illustrate the operation of MAX-HEAPIFY ( $A, 3$ ) on the array  $A$  given as  $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 10 \rangle$ .

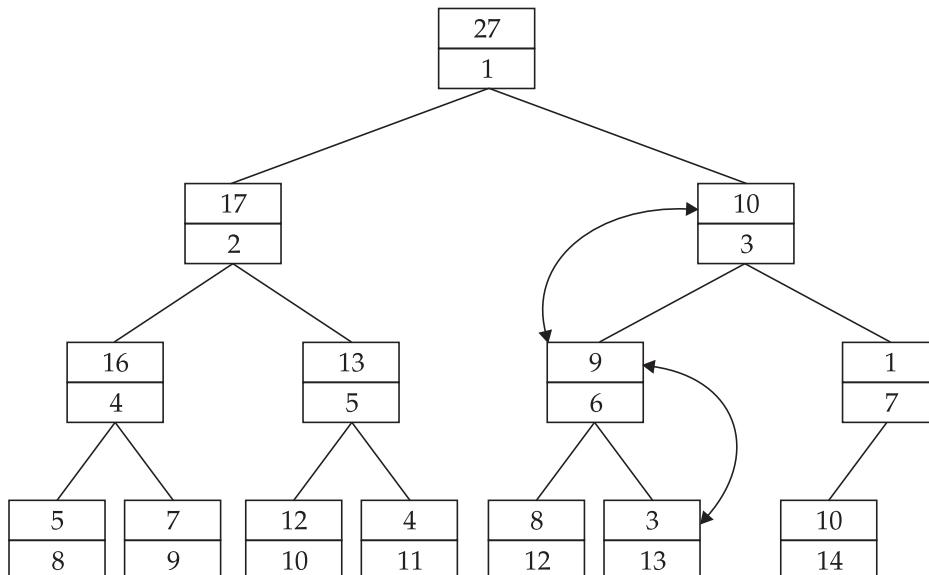
**Solution:** The heap for array  $A$  is given as follows:



Applying MAX-HEAPIFY (A, 3)



Applying MAX-HEAPIFY (A, 6)



Applying MAX-HEAPIFY (A, 3) no change take place as no children left.

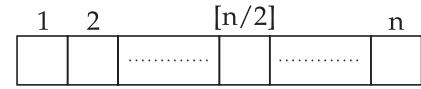
**Example 10.4** The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

**Solution:** MAX-HEAPIFY (A, i)

```

1. n = Heapsize (A)
2. l = 2 * i
3. r = 2 * i + 1
4. if (l < n and A [l] > A [i])
5. then maximum = l
6. else maximum = i
7. if (r <= n and A [r] > A [maximum])
8. then maximum = r
9. if (maximum ≠ i)
10. then interchange (A [i], A [maximum])

```



MAX-HEAPIFY (A, MAXIMUM)

(Removed by question)

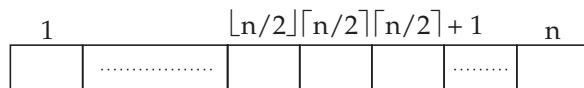
```

11. while (maximum ≤ ⌊n/2⌋)
12. do i = maximum
13. l = 2 * i
14. r = 2 * i + 1
15. if (l ≤ n and A [l] > A[i])
16.     then maximum = l
17.     else maximum = i
18. if (r ≤ n and A [r] > A [maximum])
19.     then maximum = r
20. if (maximum ≠ i)
21. then interchange (A[i], A[maximum])
22. else exit.

```

**Example 10.5** Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ .

**Solution:**



We know that the element at index  $n$  has the parent at position  $\lfloor n/2 \rfloor$ . After that all the elements in the array are leaf nodes which are represented as.

$$\lceil n/2 \rceil = \lfloor n/2 \rfloor + 1, \lceil n/2 \rceil + 1 = \lfloor n/2 \rfloor + 1 + 1 = \lfloor n/2 \rfloor + 2 \text{ and so on}$$

So we conclude that, leaf nodes are indexed by  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ .

## 10.5 ALGORITHM FOR BUILDING MAX-HEAP

```
BUILD - MAX - HEAP (A)
1. n = heapsize (A)
2. For i =  $\lfloor n/2 \rfloor$  down to 1
3. do MAX-HEAPIFY (A, i)
```

### 10.5.1 Analyzing the Running Time of BUILD-MAX-HEAP

A  $n$ -element heap has height  $\lfloor \log n \rfloor$  and it contains at most  $\lceil n/2^{h+1} \rceil$  nodes of any height  $h$ . We know that the time required by MAX-HEAPIFY when called on a node of height  $h$  is  $O(h)$ . So, we can express the total cost of BUILD-MAX-HEAP as being bounded from the above by

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} (h/2^h)\right)$$

Again  $\sum_{h=1}^{\infty} (h/2^h) = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots + \frac{7}{128} + \dots = 1.1111 \leq 2$

So,

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

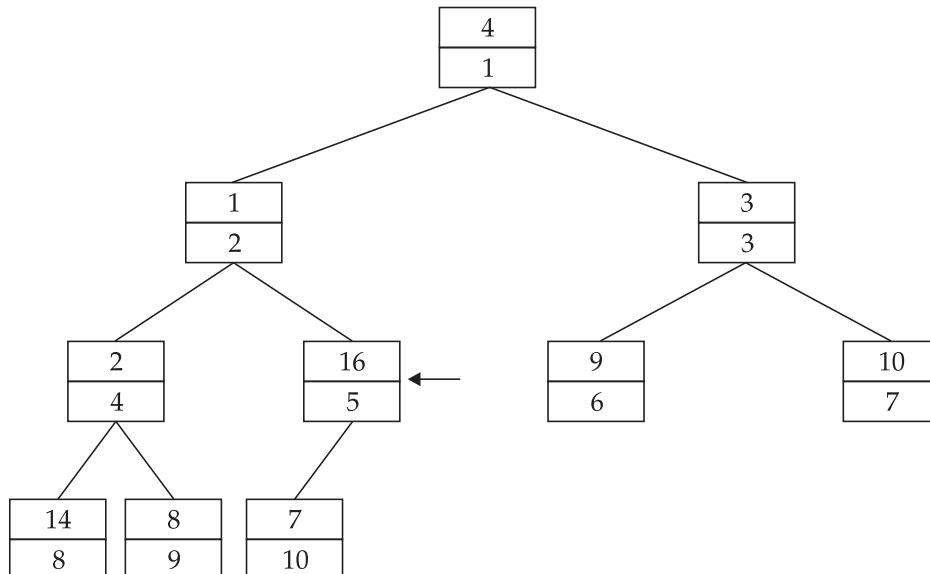
Hence, we can build a max-heap from an unordered array in linear time.

**Example 10.6** Build a MAX-HEAP from the given array  $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$

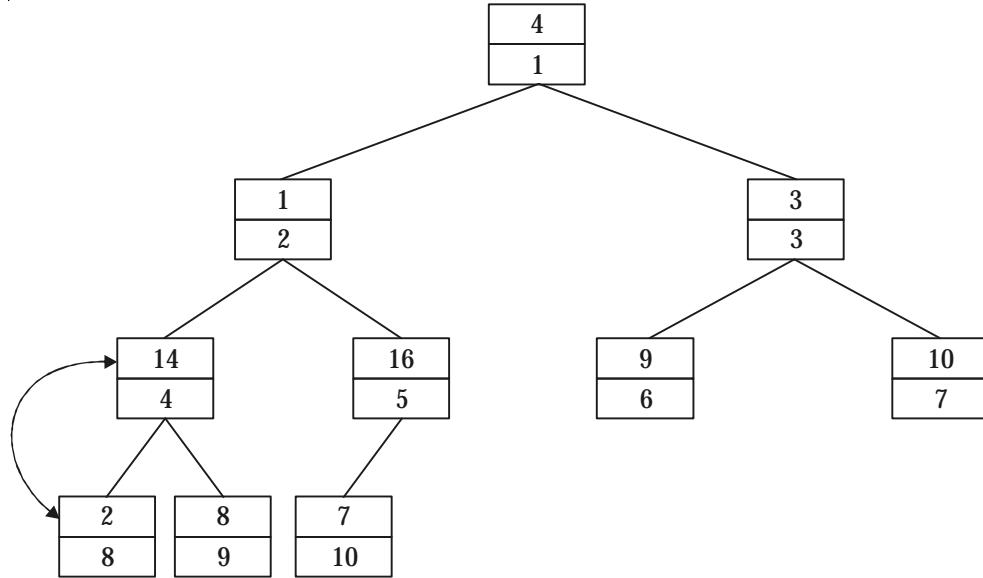
**Solution:** Heapsize of  $A = n = 10$

for  $i = \lfloor 10/2 \rfloor$  down to 1 = 5 to 1

**Step 1:** MAX-HEAPIFY (A, 5)

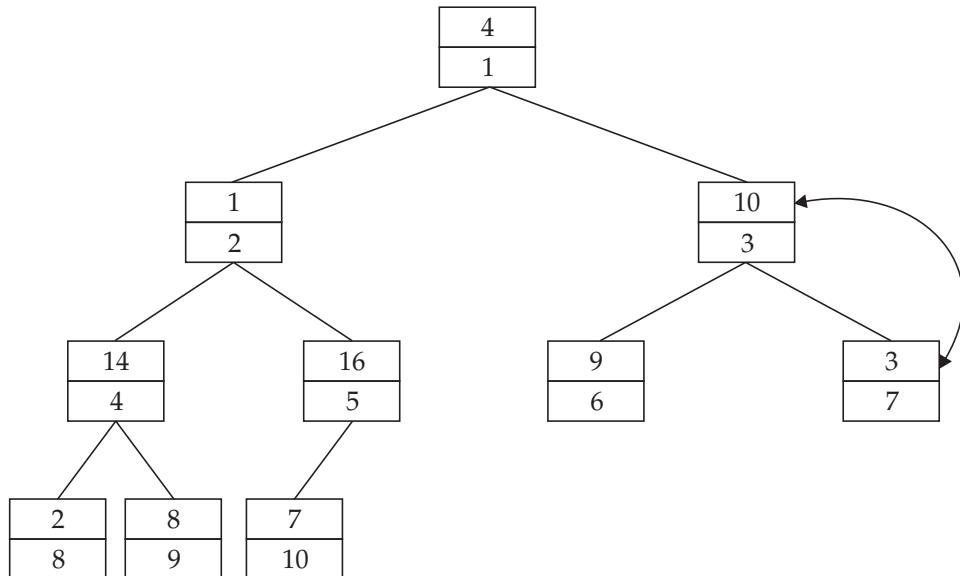


**Step 2:** MAX-HEAPITY (A, 4)



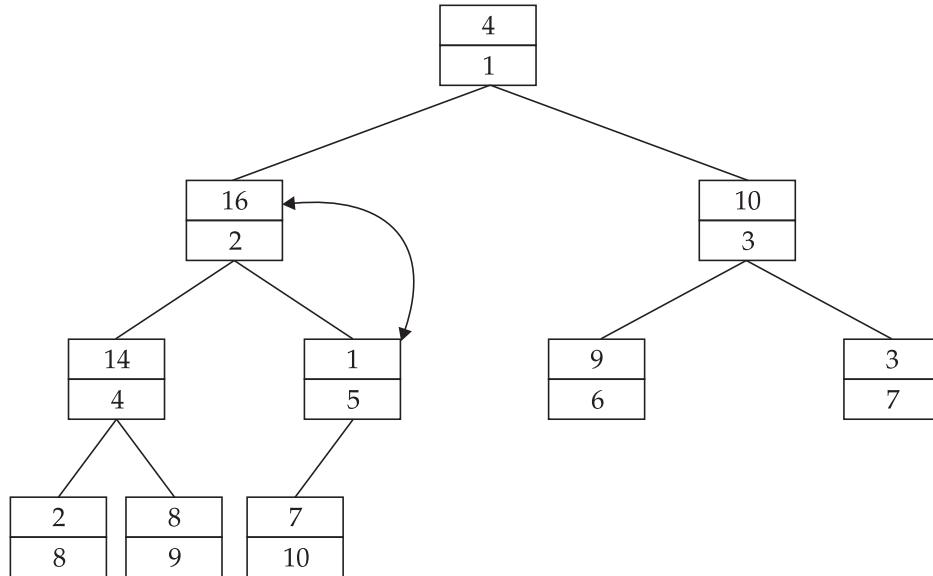
Now calling MAX-HEAPIFY (A, 8) no change takes place.

**Step 3:** MAX-HEAPIFY (A, 3)

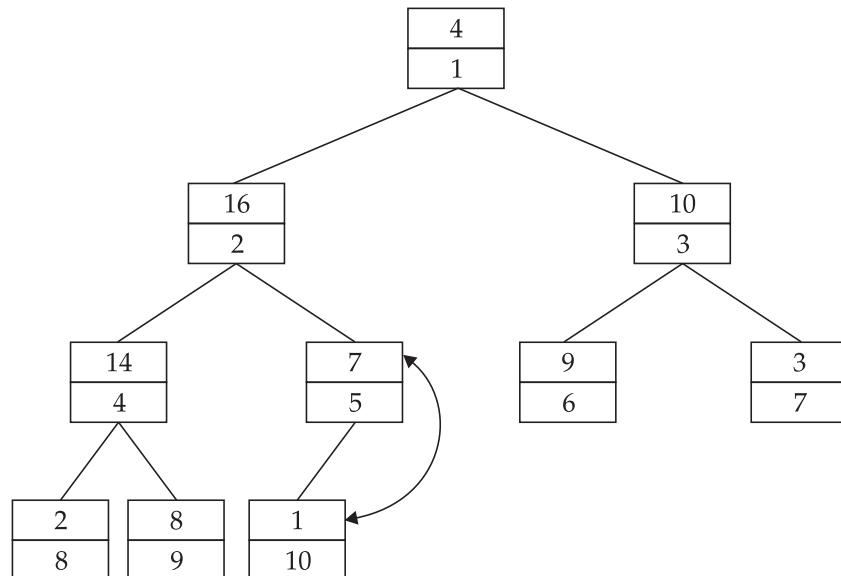


Calling MAX-HEAPIFY (A, 7) no changes takes place.

**Step 4:** MAX-HEAPIFY (A, 2)

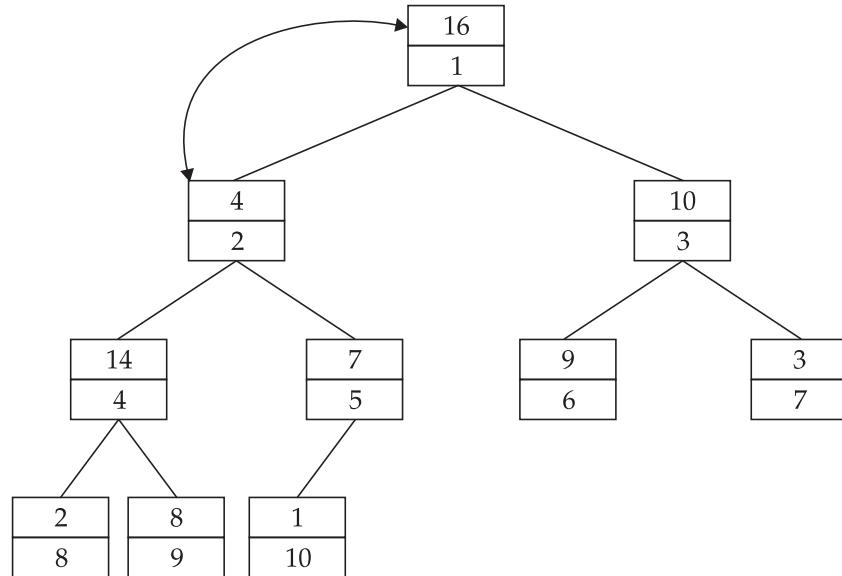


Again calling MAX-HEAPIFY (A, 5)

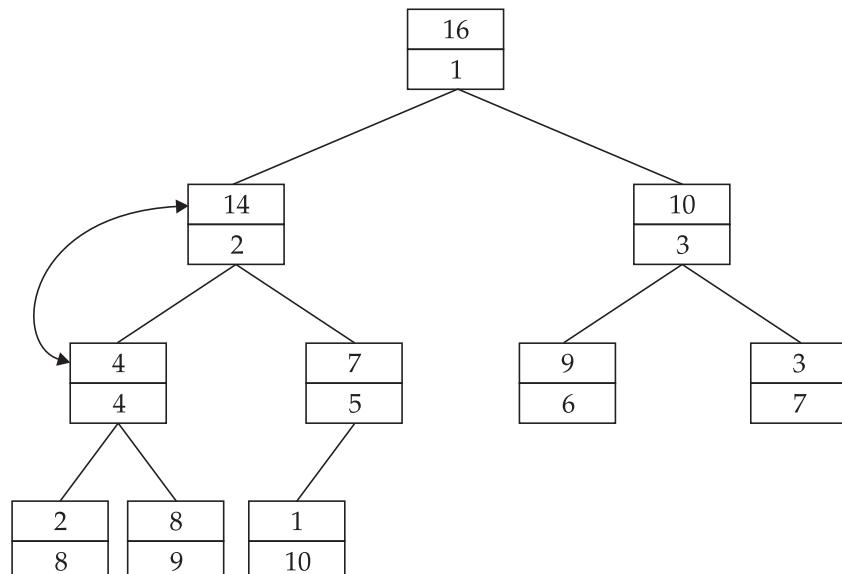


Now calling MAX-HEAPIFY (A, 10) no change takes place.

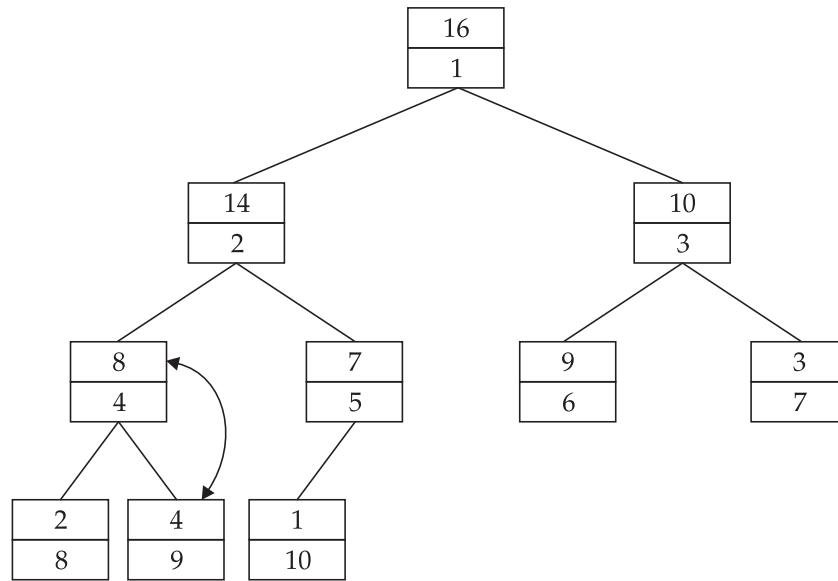
**Step 5:** MAX-HEAPIFY (A, 1)



Calling MAX-HEAPIFY (A, 2)



Calling MAX-HEAPIFY (A, 4)



Now calling MAX-HEAPIFY (A, 9) no change takes place.

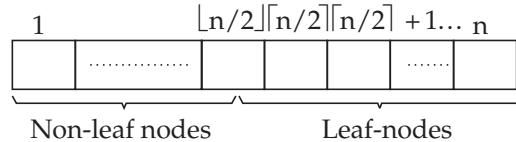
**Example 10.7** Show that there are at most  $\lceil n/2^{h+1} \rceil$  nodes of height  $h$  in any  $n$ -element heap.

**Solution:** Our claim is  $\rightarrow$  nodes at height  $h \leq \lceil n/2^{h+1} \rceil$

Let us prove this by method of induction.

**Step 1:** When  $h = 0$ , the number of nodes at height ( $h$ ) =  $0 \leq \lceil n/2^1 \rceil$ .

$$\begin{aligned} \text{Non-leaf nodes} &= 1, \dots, \lceil n/2 \rceil \\ &= \lfloor n/2 \rfloor \text{ numbers of nodes} \end{aligned}$$



$$\text{Leaf nodes} = \lceil n/2 \rceil, \lceil n/2 \rceil + 1, \dots, n = \lceil n/2 \rceil \text{ numbers of nodes}$$

So maximum nodes at height ( $h$ ) = 0 are  $\lceil n/2 \rceil$

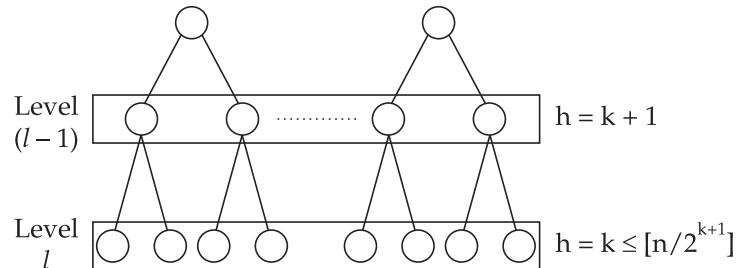
Hence Step - 1 is proved.

**Step 2:** At height  $k$  at most nodes  $\leq \lceil n/2^{k+1} \rceil$  nodes present.

**Step 3:** We have to prove at height  $k + 1$  at most  $\leq \lceil n/2^{k+2} \rceil$  node present.

At level ' $l$ ' number of nodes at height  $h \leq \lceil n/2^{k+1} \rceil$ .

So at level ' $l-1$ ' number of nodes present = Half of the nodes present at level ' $l \leq \lceil n/2^{k+1} \rceil / 2 = \lceil n/2^{k+2} \rceil$ '. This concludes our proof.



## 10.6 HEAPSORT-ALGORITHMS

```
HEAPSORT (A)
1. BUILD-MAX-HEAP (A)
2. n = heapsize (A)
3. for i = n down to 2
4. do interchange (A [i], A[1])
5. heapsize (A) = heapsize (A) - 1
6. MAX-HEAPIFY (A, 1)
```

### Running Time

Line 1 of the algorithm *i.e.*, BUILD-MAX-HEAP procedure takes  $O(n)$  time. Each of the  $n - 1$  calls to MAX-HEAPIFY takes  $O(\log n)$  time.

In total, the running time of Heap-sort algorithm is  $O(n \log n)$ .

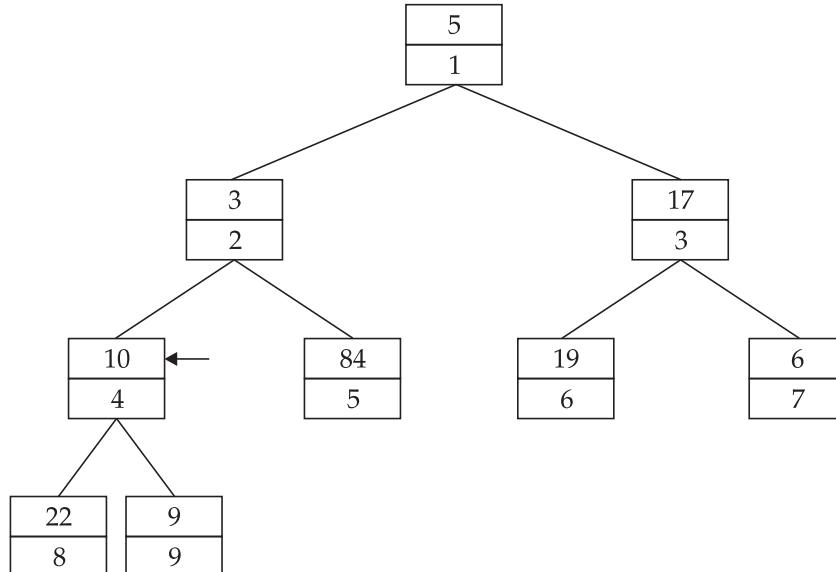
**Example 10.8** Build a MAX-HEAP and then do HEAPSORT array A given as

$A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$ . What is the time complexity of the whole process?

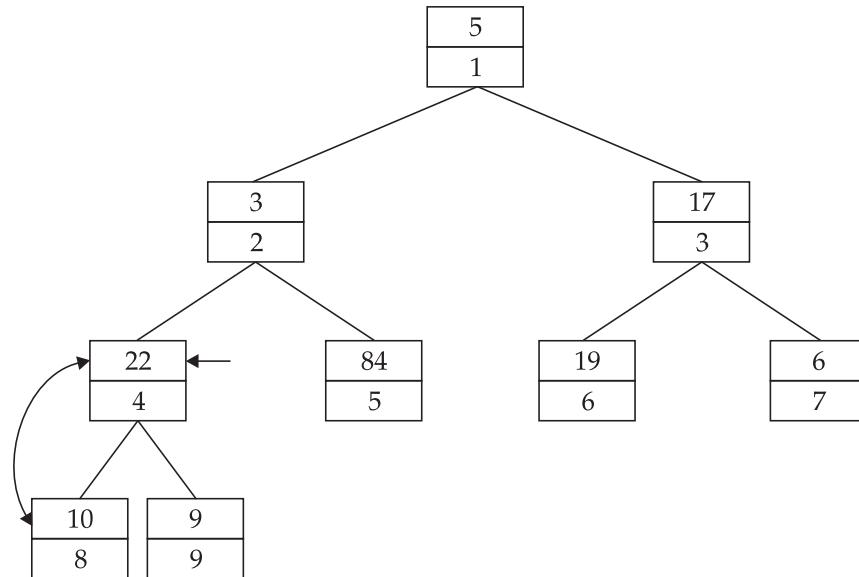
**Solution:** Building MAX-HEAP

Heapsize of A =  $n = 9$

For  $i = \lfloor 9/2 \rfloor$  down to 1 = 4 to 1

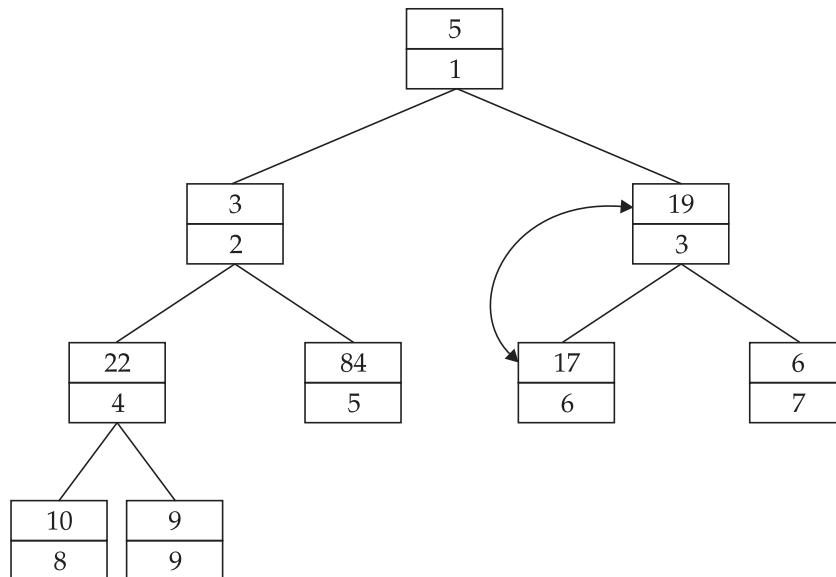


**Step 1:** MAX-HEAPIFY (A, 4)



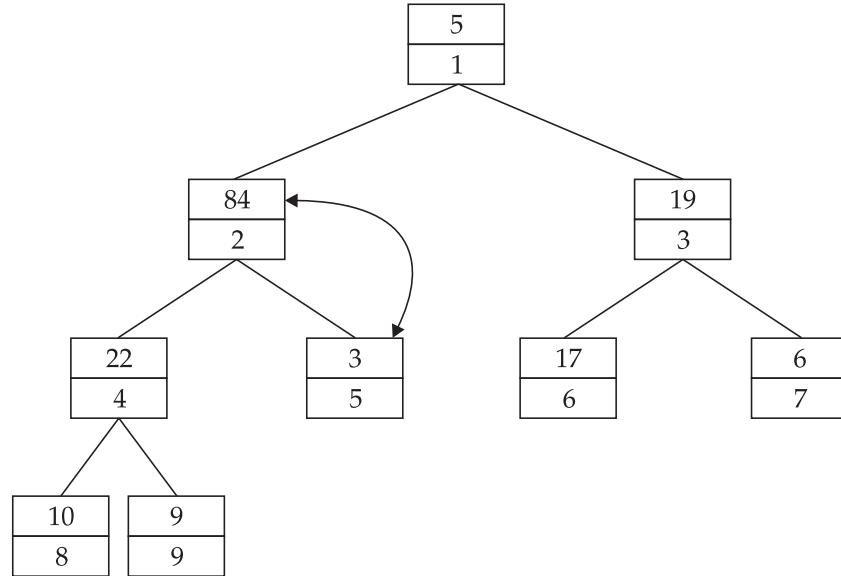
Calling MAX-HEAPIFY (A, 8) no change takes place.

**Step 2:** MAX-HEAPIFY (A, 3)



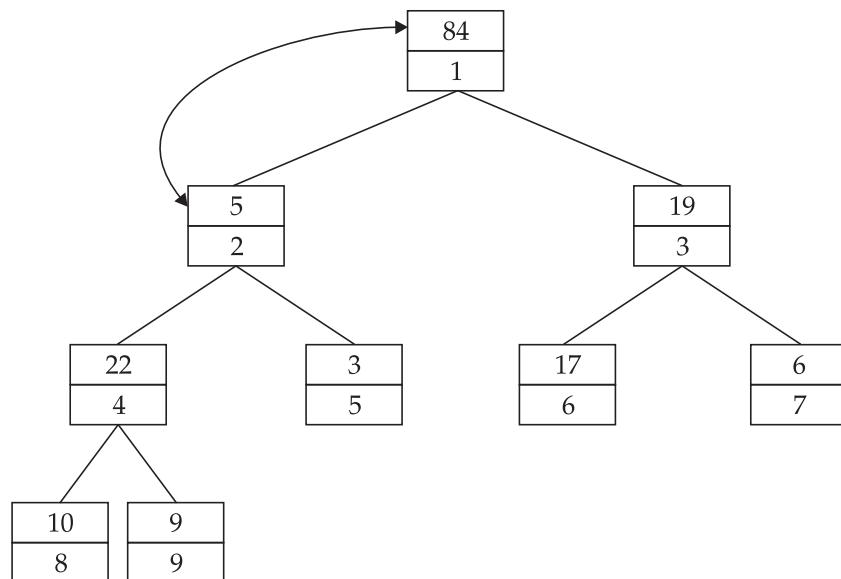
Calling MAX-HEAPIFY (A, 6) no change takes place.

**Step 3:** MAX-HEAPIFY (A, 2)

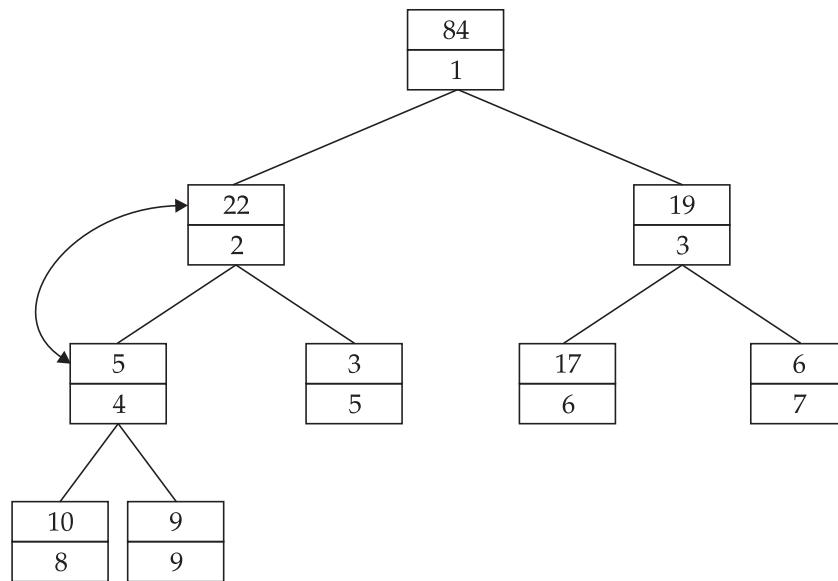


Calling MAX-HEAPIFY (A, 5) no change takes place.

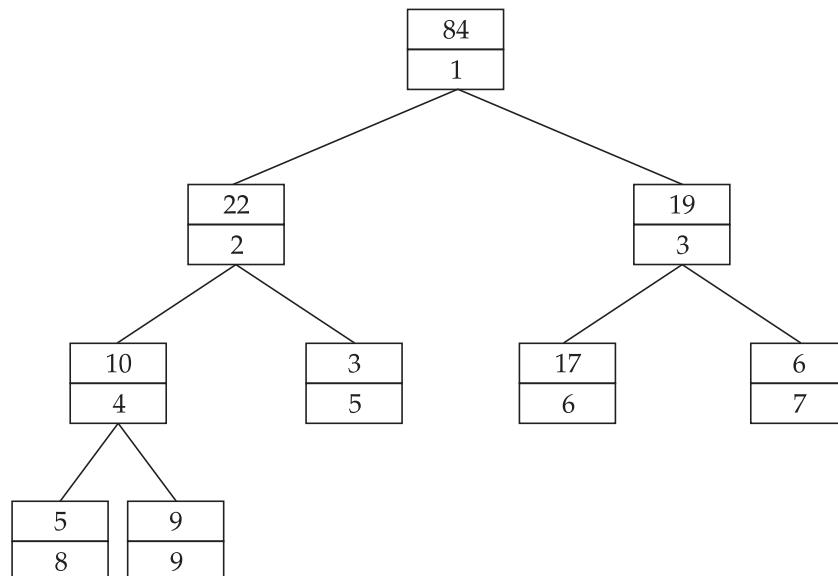
**Step 4:** MAX-HEAPIFY (A, 1)



Do MAX-HEAPIFY (A, 2)



Do MAX-HEAPIFY (A, 4)



Calling MAX-HEAPIFY (A, 8) no change takes place. So the above tree is the Building HEAPSORT

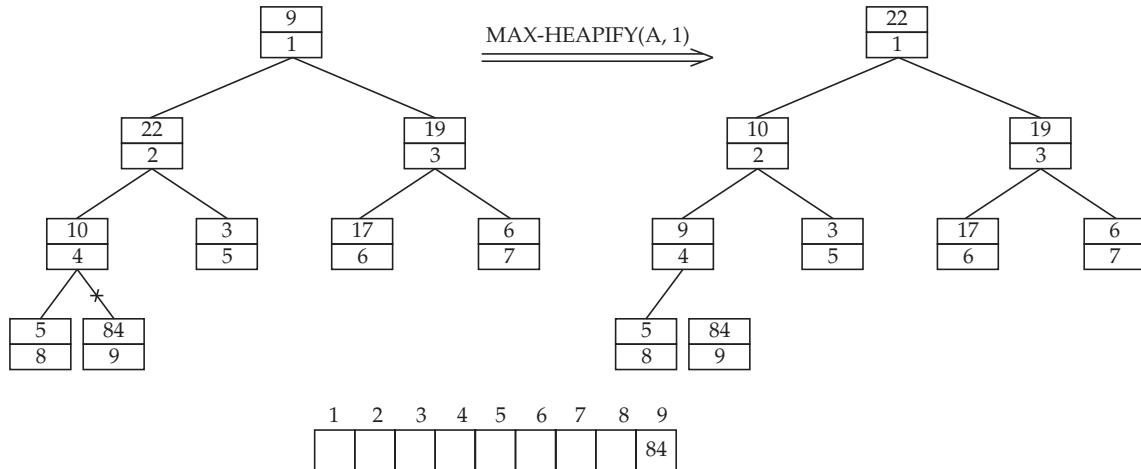
Heapsize ( $n$ ) = 9

For  $i = 9$  down to 2.

**Step 1:** For  $n = 9$

Interchange A[9]  $\leftrightarrow$  A [1]

Heapsize = 8  
Do MAX-HEAPIFY (A, 1)

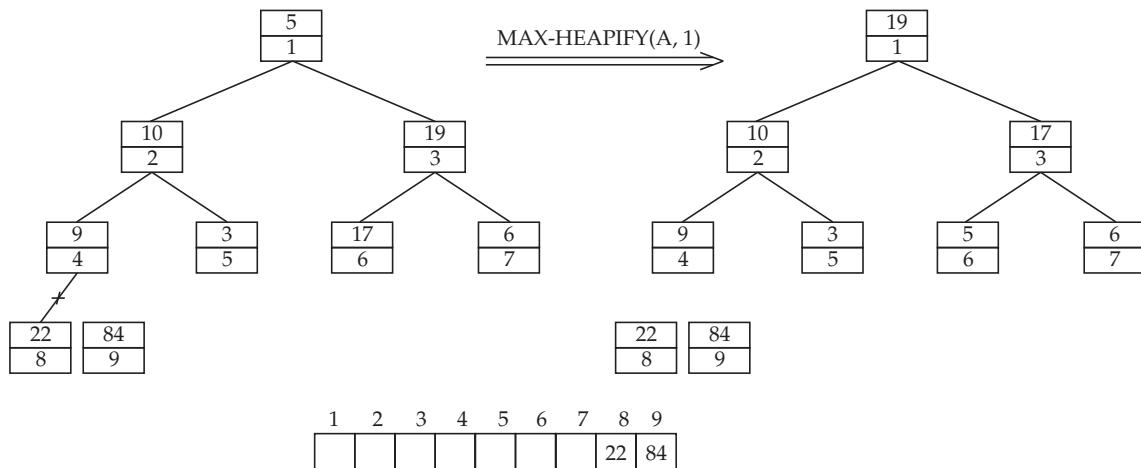


**Step 2: For  $n = 8$**

Interchange A [8]  $\leftrightarrow$  A[1]

Heapsize = 7

Do MAX-HEAPIFY (A, 1)

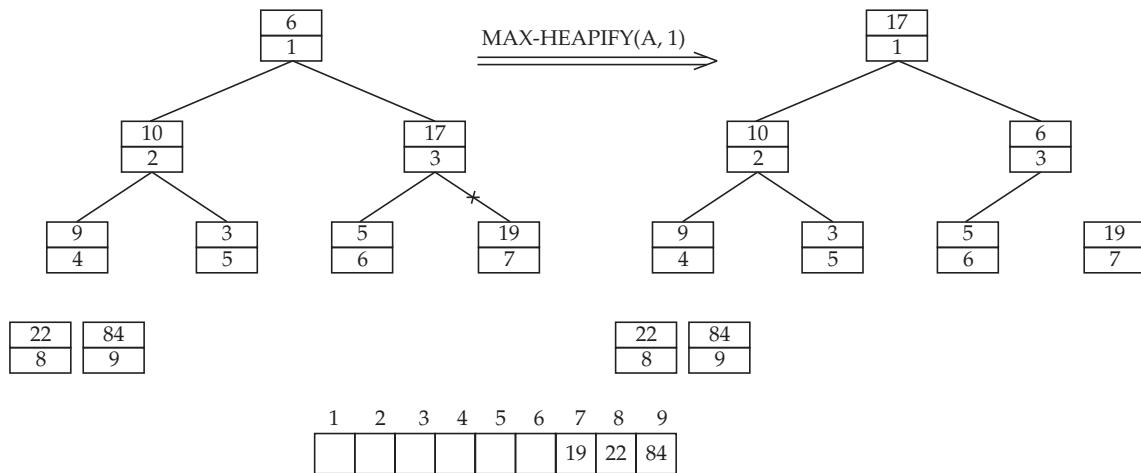


**Step 3: For  $n = 7$**

Interchange A [7]  $\leftrightarrow$  A [1]

Heapsize = 6

Do MAX-HEAPIFY (A, 1)

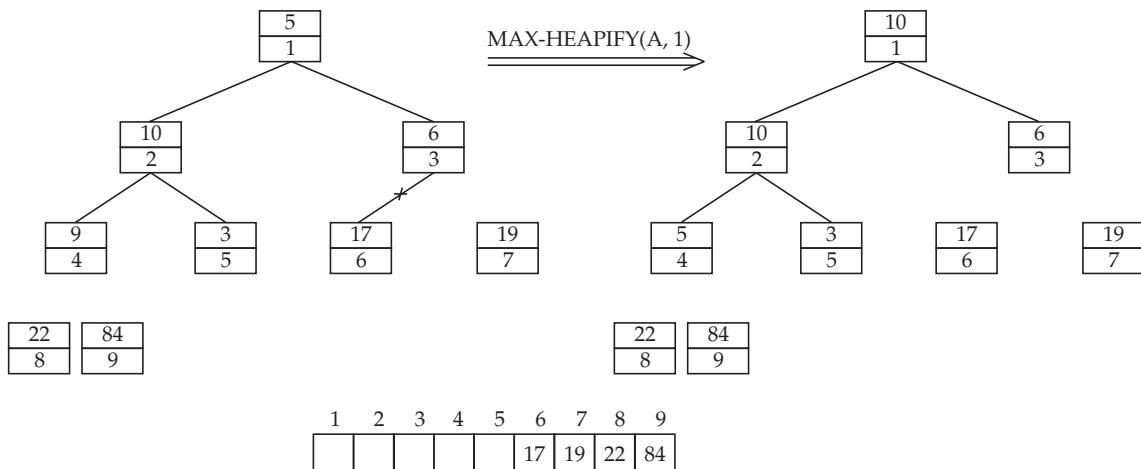


**Step 4: For  $n = 6$**

Interchange  $A[6] \leftrightarrow A[1]$

Heapsize = 5

MAX-HEAPIFY (A, 1)

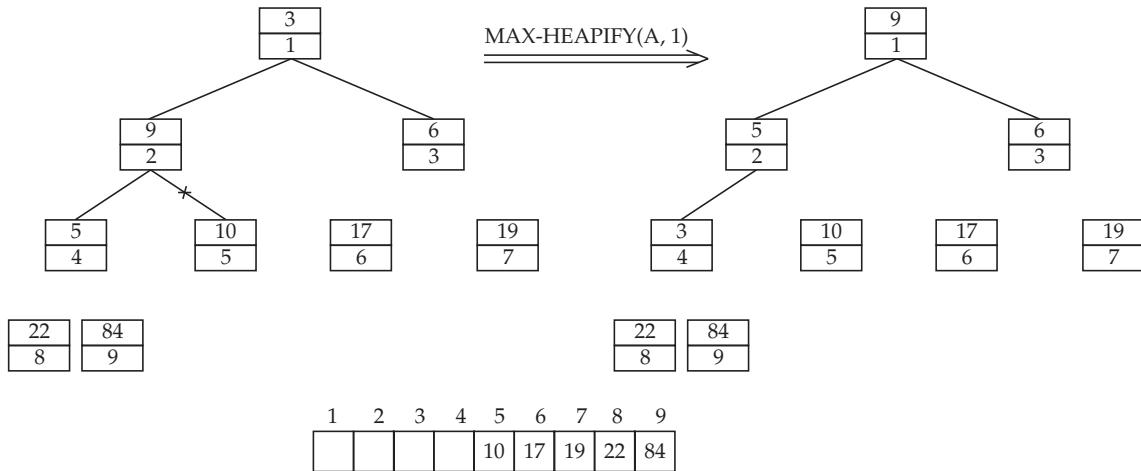


**Step 5: For  $n = 5$**

Interchange  $A[5] \leftrightarrow A[1]$

Heapsize = 4

MAX-HEAPIFY (A, 1)

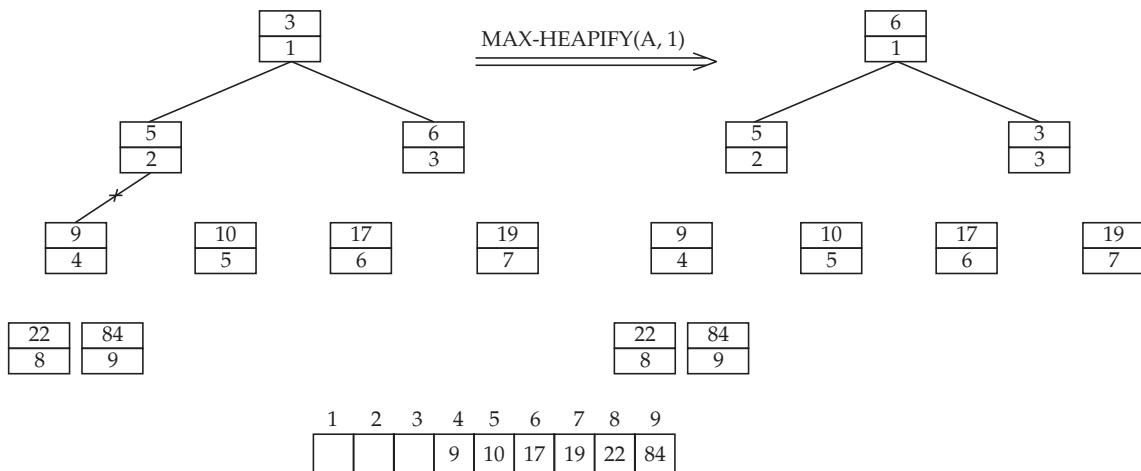


**Step 6: For  $n = 4$**

Interchange  $A[4] \leftrightarrow A[1]$

Heapsize = 3

Do MAX-HEAPIFY (A, 1)

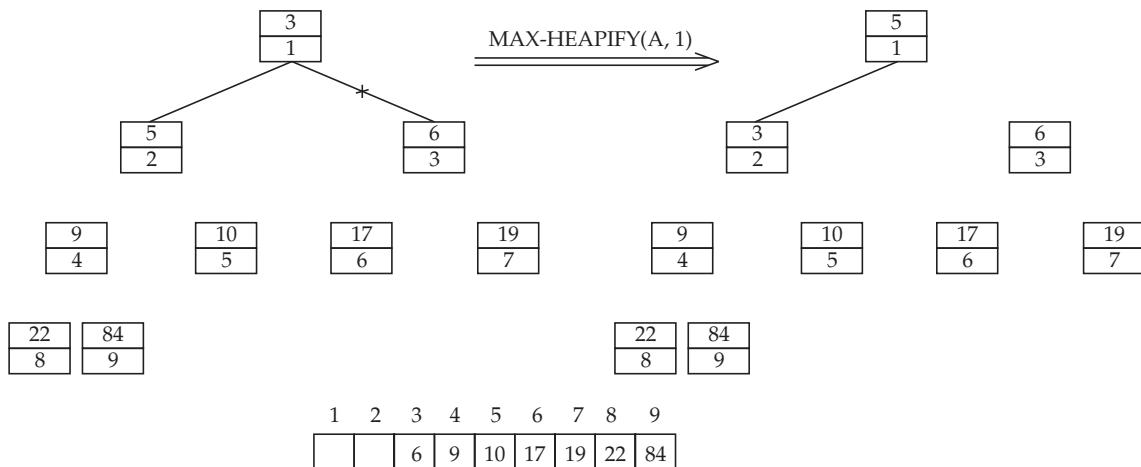


**Step 7: For  $n = 3$**

Interchange  $A[3] \leftrightarrow A[1]$

Heapsize = 2

Do MAX-HEAPIFY (A, 1)

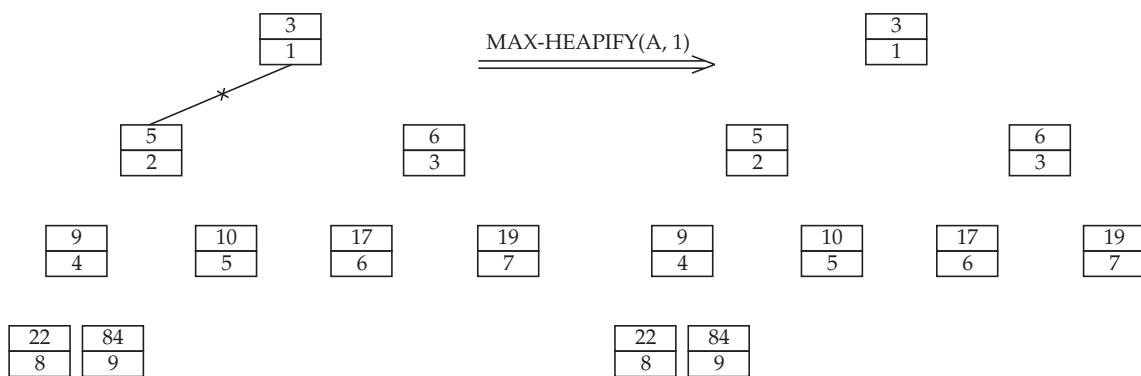


**Step 8: For  $n = 2$**

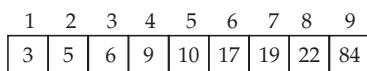
Interchange  $A[2] \leftrightarrow A[1]$

Heapsize = 1

Do MAX-HEAPIFY (A, 1)



For loop ends here. Heap becomes.



### Time complexity

Build MAX-HEAP operation takes  $O(n)$

Procedure HEAPSORT takes  $O(n \log n)$

Each MAX-HEAPIFY operation takes  $O(\log n)$  ( $n$  may vary from 1 to 9)

**Example 10.9** Write an algorithm for MIN-HEAPIFY, BUILD MIN-HEAP. Using these algorithms design another for HEAPSORT. Write the time complexity of entire process.

**Solution:** (i) Algorithm for MIN-HEAPIFY (A,i) // A is the given array of elements and i is the index.

1.  $n = \text{heapsize} (A)$
2.  $l = 2i, r = 2i + 1$  // l is the left child and r is the right child.

```

3. if ( $l \leq n$  and  $A[l] < A[i]$ )
4. then smallest =  $l$ 
5. else smallest =  $i$ 
6. if ( $r \leq n$  and  $A[r] < A[smallest]$ )
7. then smallest =  $r$ 
8. if (smallest  $\neq i$ )
9. then interchange ( $A[i], A[smallest]$ )
10. MIN-HEAPIFY ( $A, Smallest$ )

```

The running time of MIN-HEAPIFY procedure is  $O(\log n)$ .

### (ii) Algorithm for BUILD MIN-HEAP

BUILD MIN-HEAP ( $A$ )

```

1.  $n = \text{heapsize } (A)$ 
2. for  $i = \lfloor n/2 \rfloor$  down to 1
3. do MIN-HEAPIFY ( $A, i$ )

```

The running time of BUILD MIN-HEAP procedure is  $O(n)$ .

### (iii) Algorithm to construct HEAPSORT

HEAPSORT ( $A$ )

```

1. BUILD-MIN-HEAP ( $A$ )
2.  $n = \text{heapsize}(A)$ 
3. for  $i = n$  down to 2
4. do interchange ( $A[i], A[1]$ )
5.  $\text{heapsize}(A) = \text{heapsize}(A) - 1$ 
6. MIN-HEAPIFY ( $A, 1$ )

```

So the total running time of heapsort algorithm is  $O(n \log n)$  using MIN-HEAP.

---

## 10.7 PROS AND CONS OF HEAPSORT

### Pros : (Advantages)

- Time complexity of the algorithm is  $O(n \log n)$ .
- Auxiliary space required for the algorithm is  $O(1)$ .
- In-place and non-recursive makes it a good choice for large data sets.

### Cons : (Disadvantages)

- Works slower than other such DIVIDE-AND-CONQUER sorts that have the same  $O(n \log n)$  time complexity due to cache behaviour and other factors.

- Unable to work when dealing linked lists due to non-convertibility of linked list to heap structure.

## CHAPTER NOTES

---

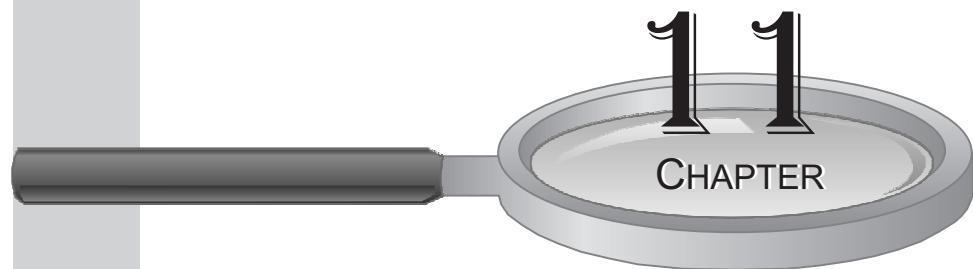
- A heap is simply an array of times, rather than an array of items with priorities.
- The MAX-HEAPIFY procedure, which runs in  $O(\log n)$  time, is the key to maintain the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The heapsort procedure, which runs in  $O(n \log n)$  time, sorts an array in place.
- Removing an item from the heap means moving the last item into first position and then applying heapify operation at first position.
- The worst case running time of HEAPSORT is  $\Omega(n \log n)$ .
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY and HEAP-MAXIMUM procedures which runs in  $O(\log n)$  time, allow the heap data structure to implement priority queue which will be discuss in Chapter – 11.

## EXERCISES

---

1. Explain the process of Heapsort. Write an algorithm to construct a min-heap. Construct max-heap from the following list. Sketch the heap after deleting 77 and 75. What is the time complexity of the whole process?  
 $\{21, 64, 56, 63, 44, 7, 9, 77, 75, 32, 34, 14, 49\}$
2. Show that the worst-case running time of MAX-HEAPIFY on a heap of size  $n$  is  $\Omega(\log n)$ .
3. Illustrate the operation BUILD-MAX-HEAP on the array  $A = \{5, 3, 17, 10, 84, 19, 6, 22, 9\}$ .
4. Show that the worst-case running time of HEAPSORT is  $\Omega(n \log n)$ .
5. Find the running time of BUILD-MAX-HEAP procedure.

# PRIORITY QUEUES



## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand :**

- Priority queue: an abstract data type
- Maximum priority queue and minimum priority queue
- Different operations on priority queue
- Different applications of priority queue

# 11 PRIORITY QUEUES

## INSIDE THIS CHAPTER

- 11.1 Introduction
- 11.2 Types of Priority Queues
- 11.3 Algorithms for Max-Priority Queue Operations
- 11.4 Different Applications of Priority Queue

## 11.1 INTRODUCTION

In this chapter, we look at an abstract data type known as **Priority queue**. Like a (normal) queue, a priority queue contains a collection of items that are waiting to be processed. In a queue, items are removed for processing in the same order in which they were added to the queue. In a priority queue, however, each item has a priority, and when it's time to select an item, the item with the highest priority is the one that is chosen. The priority might have nothing to do with the time at which the item was added to the data structure. Note that, the priority queue is not, strictly speaking, a queue at all, since a queue is a first-in, first out data structure and a priority queue is not.

**Definition 11.1 (Priority queue):** For any data type, base type, we define an abstract data type priority queue of base type. A possible value of this type is a collection of items of type base type, where each item has an associated number, which is called its priority.

## 11.2 TYPES OF PRIORITY QUEUES

Priority queues have two types:

- (i) Maximum Priority Queue
- (ii) Minimum Priority Queue.

**(i) Maximum Priority Queue:** In maximum priority queue the maximum value element is present at the first position in the array.

Example:	1	2	3	4	5	6	7	8
	23	15	12	17	14	12	10	4

### Operations Supported by MAX - Priority Queue Having “n” Elements

1. Delete a key value =  $O(\log n)$  (at worst case in case of extracting maximum element)  
This is called “EXTRACT MAX”.
2. Insert a key value =  $O(\log n)$  (at worst case, always equals to the height of the tree).  
This operation is called “INSERT”.
3. Find MAX =  $O(1)$  (as the 1st position of the array consists of the maximum element).  
This is called “MAXIMUM”.
4. Increase the key value =  $O(\log n - \text{height } i)$  (height  $i$  refers to the height of the element in heapsort). This is called “INCREASE – KEY”.

**(ii) Minimum Priority Queue:** In the priority queue if the first position consists the minimum element, then it referred as the min-priority queue.

Example:	1	2	3	4	5	6	7	8
	4	10	12	14	17	12	15	23

### Operations Supported by MIN - Priority Queue Having “n” Elements

1. Delete a key value =  $O(\log n)$  This operation is called “EXTRACT-MIN”.
2. Insert a key value =  $O(\log n)$  This operation is called “INSERT”.
3. Find MIN =  $O(1)$  This is referred as “MINIMUM”.
4. Decrease the key value =  $O(\log n - \text{height } i)$ . This is called “DECREASE KEY”.

## 11.3 ALGORITHMS FOR MAX - PRIORITY QUEUE OPERATIONS

---

### 11.3.1 Algorithm for HEAP - MAXIMUM (A)// where A is the given array

```
1. return A[1]
Running time = O(1)
```

### 11.3.2 Algorithm for HEAP - EXTRACT-MAX (A)// A is the given array

```
1. n = Heapsiz (A)
2. if n < 1
3. then "error: heap underflow".
4. maximum = A[1]
5. A[1] = A[n]
6. n = n - 1
```

```

7. MAX-HEAPIFY (A, 1)
8. return maximum

```

Running time of Heap-Extract-Max (A) is  $O(\log n)$

#### 11.3.3 Algorithm for HEAP-INCREASE-KEY (A, i, key)

```

1. if key < A[i]
2. then "error: new key is smaller than current key"
3. A[i] = key
4. while i > 1 and A[PARENT (i)] < A[i]
5. exchange A[i] with A[PARENT (i)]
6. i = PARENT (i)

```

The running time of Heap-Increase-key on an  $n$ -element heap is  $O(\log n)$ .

#### 11.3.4 Algorithm for MAX-HEAP-INSERT (A, key)

```

1. n = Heapsize (A)
2. A[n] = ∞.
3. HEAP-INCREASE-KEY (A, n, key)

```

The running time of MAX-HEAP-INSERT on an  $n$ -element heap is  $O(\log n)$ .

---

## 11.4 DIFFERENT APPLICATIONS OF PRIORITY QUEUE

There are many applications of priority queues. They are:

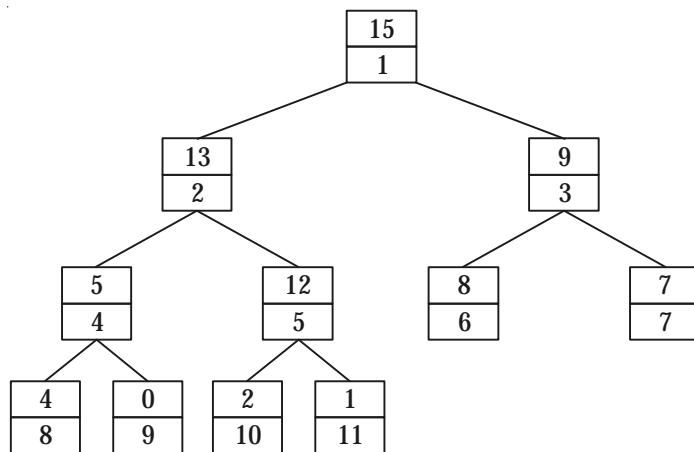
- A priority queue could be used to hold packets of data waiting to be transmitted over a network. Packets that contain important data or data that must arrive in a timely way would be given higher priority and would therefore be transmitted ahead of lower priority data.
- In an operating system, a priority queue might hold “jobs”, that is, programs that are waiting for execution. When processing time becomes available, a job would be removed from the priority queue for processing. Because of the way priority queues work, high priority jobs would be executed before low priority jobs, even if the low priority jobs have been waiting longer.
- A computer helpdesk, for example, might use a priority queue to hold requests for help until tasks become available to process them.

**Example 11.1** Illustrate the operation of HEAP-EXTRACT-MAX on the heap  $A = < 15, 13, 9, 5, 12, 8, 7, 4, 0, 2, 1 >$ .

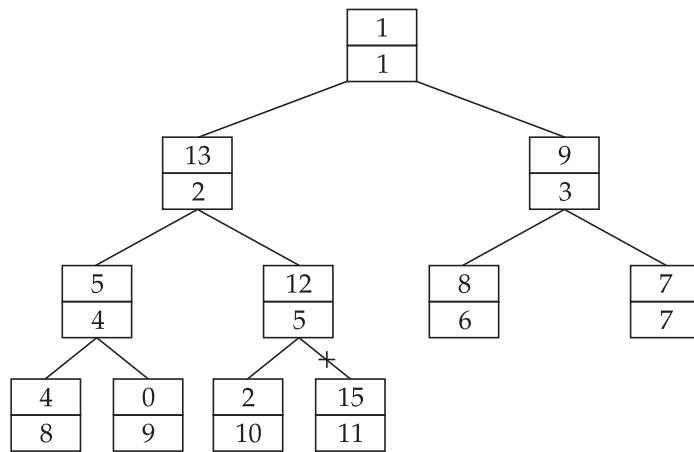
**Solution:** Heapsize of  $A = n = 11$ .

Maximum =  $A[1] = 15$

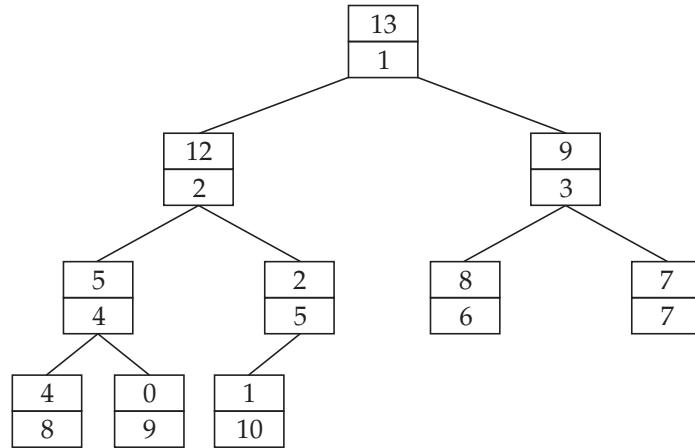
Heap becomes:



Interchange  $A[1] \leftrightarrow A[11]$  then, reduce the heapsize to 10.



Applying MAX-HEAPIFY ( $A, 1$ ), heap becomes

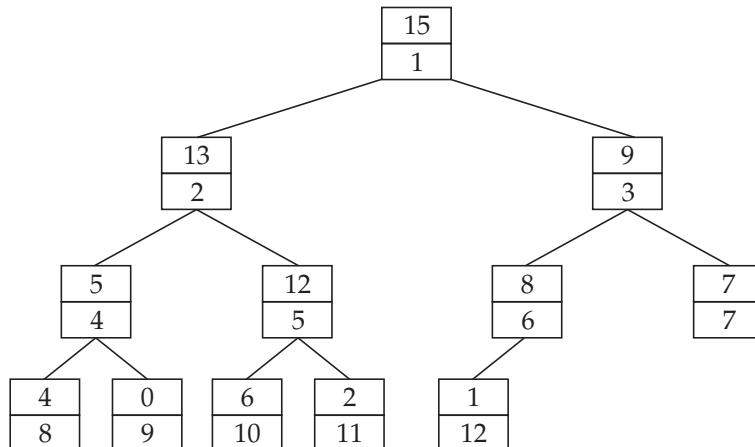


So the operation HEAP-EXTRACT-MAX returns 15.

**Example 11.2** Illustrate the operation of MAX-HEAP-INSERT ( $A, 10$ ) on the heap  $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ . What is the running time required to do this operation?

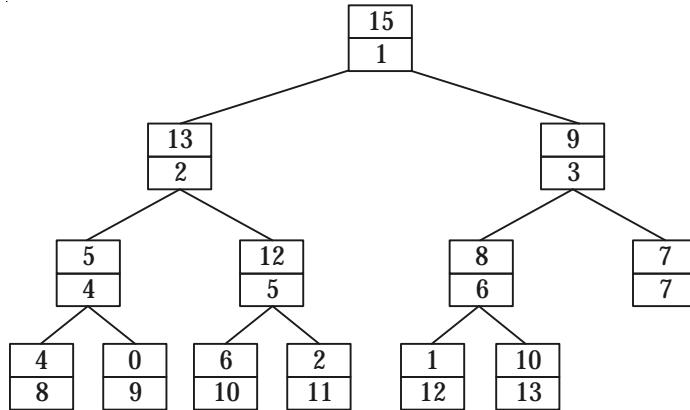
**Solution:** The given heap  $A$  is  $\langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ .

So drawing the MAX-HEAP

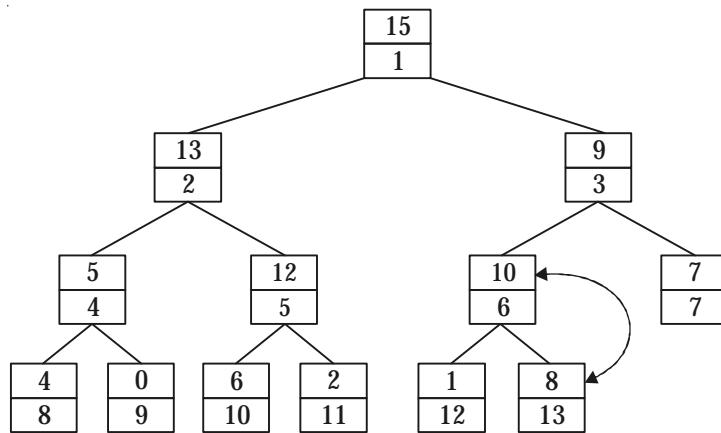


To insert the element into the Max-heap 1st increase the heapsize from 12 to 13. Then put the key value as 10 in 13<sup>th</sup> position.

Build Max-Heap: for  $\lfloor 13/2 \rfloor$  to 1 i.e., 6 to 1.



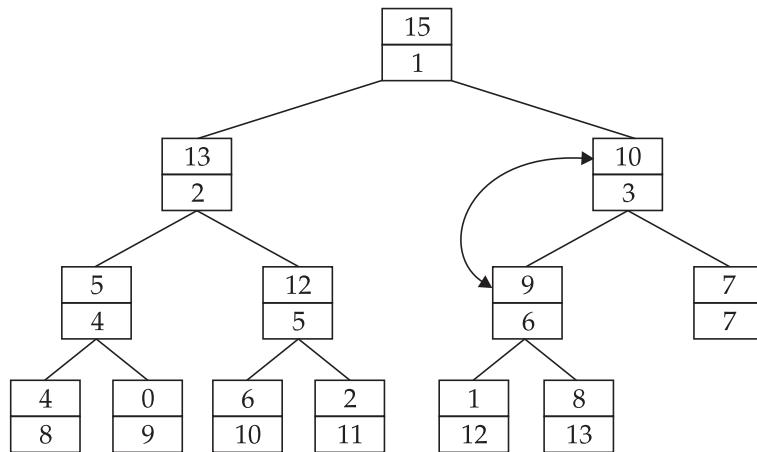
**MAX-HEAPIFY (A, 6)**



**MAX-HEAPIFY (A, 5):** No change takes place.

**MAX-HEAPIFY (A, 4):** No change takes place.

**MAX-HEAPIFY (A, 3)**



**MAX-HEAPIFY (A, 2):** No change takes place.

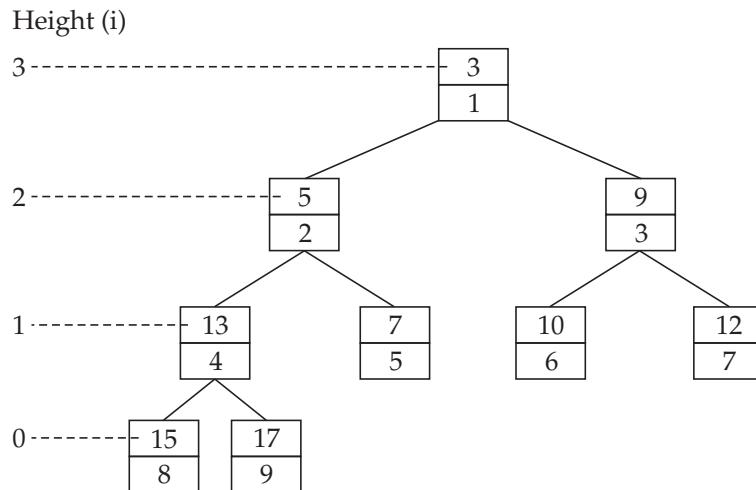
**MAX-HEAPIFY (A, 1):** No change takes place.

So the required running time =  $O(\log n)$ .

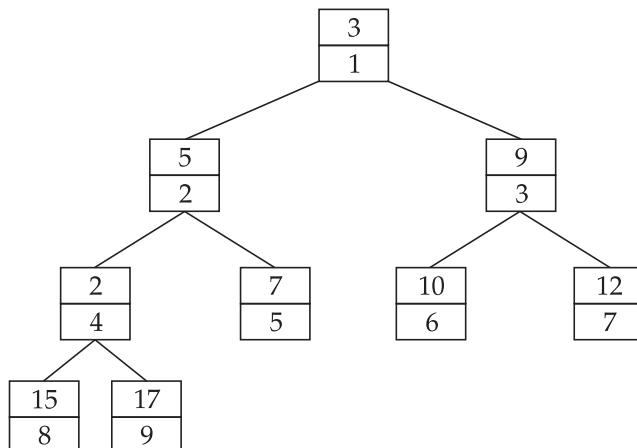
**Example 11.3** Illustrate the operation HEAP-DECREASE-KEY (A, 4, 2) on the heap  $A = \langle 3, 5, 9, 13, 7, 10, 12, 15, 17 \rangle$ . What is the time complexity taken by this operation to perform.

**Solution:** The given heap  $A = \langle 3, 5, 9, 13, 7, 10, 12, 15, 17 \rangle$ .

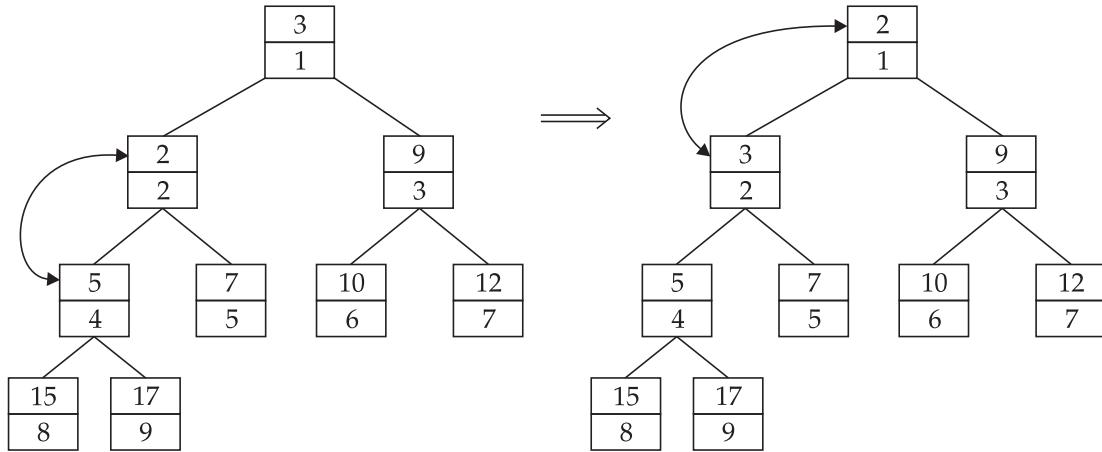
Constructing the min-heap



According to the question we need to decrease the key value at 4<sup>th</sup> position i.e., 13 to 2. Replacing 13 by 2 and drawing the heap.



By applying MIN-HEAPIFY procedure we get the following result.



The complexity of the entire process =  $O(\log n - \text{height } i) = O(\log n - 1)$

**Note:** height  $i$  means the  $i$ th index height where the element is replaced.

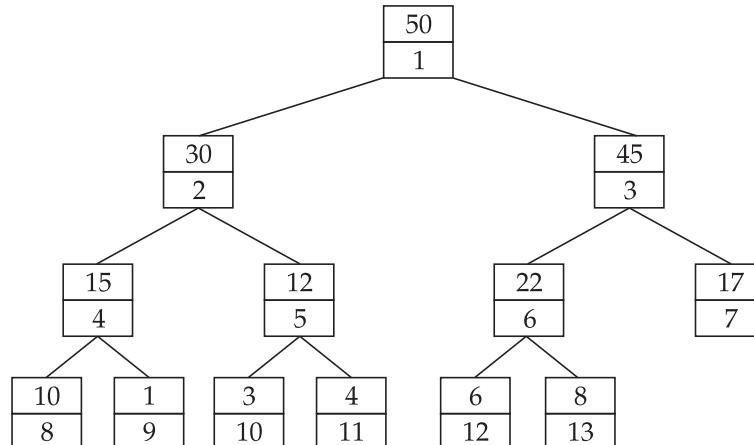
**Example 11.4** The operation **HEAP-DELETE** ( $A, i$ ) deletes the item in node  $i$  from heap  $A$ . Give an implementation of **HEAP-DELETE** that runs in  $O(\log n)$  time for an  $n$ -element max-heap.

**Solution:** Algorithm for **HEAP-DELETE** ( $A, i$ )

1.  $n = \text{Heapsize } (A)$
2. interchange  $(A[i], A[n])$
3.  $\text{Heapsize } (A) = \text{Heapsize } (A) - 1$
4. **MAX-HEAPIFY** ( $A, i$ )
5. return  $A[n]$ .

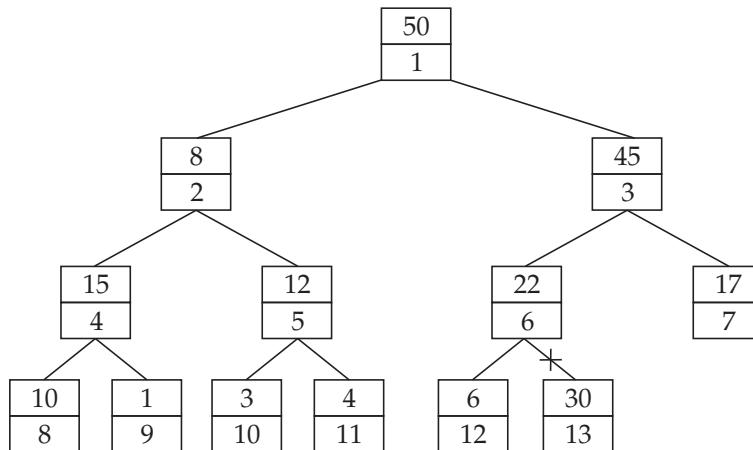
Lets take an example. The max-heap  $A$  is given by

$$A = \langle 50, 30, 45, 15, 12, 22, 17, 10, 1, 3, 4, 6, 8 \rangle$$

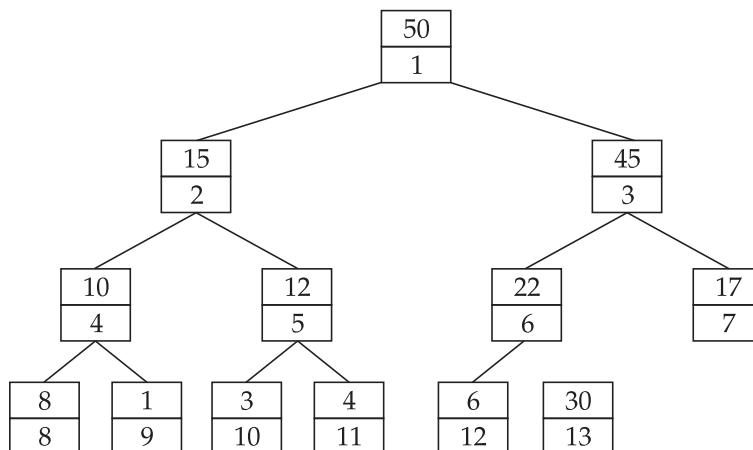


If we want to delete 30, then interchange  $A[2] \leftrightarrow A[13]$ . Reduce the heapsize to 12.

Then do **MAX-HEAPIFY** ( $A, 2$ )



Applying MAX-HEAPIFY (A, 2)



## CHAPTER NOTES

---

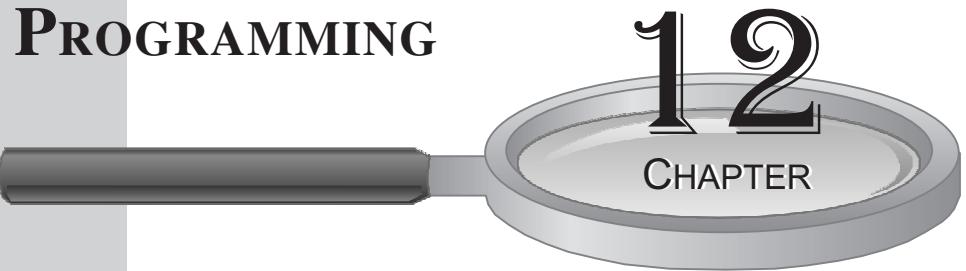
- Priority queue is an Abstract Data make Structure for maintaining a set S of elements, each with an associated key value.
- A sequence of EXTRACT-MIN operation is monotone, that is values returned by successive EXTRACT-MIN operations are monotonically increasing over time. This case arises in Dijkstra's single source shortest path algorithm.
- A heap can support any priority queue operation on a set of size  $n$  in  $O(\log n)$  time.

## EXERCISES

---

1. Illustrate the operation of HEAP-INCREASE-KEY ( $A, 5, 25$ ) on the heap  $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$ . Compute the time complexity of this operation.
2. Write the pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY AND MIN-HEAP-INSERT that implement a min-priority queue with a min-heap. Calculate the time complexity at each case.
3. Implement the HEAP-EXTRACT-MIN ( $A$ ) operation on the heap  
 $A = \langle 4, 7, 9, 15, 8, 12, 16, 32, 26, 14 \rangle$ . Compute the running time of this operation to perform.

# DYNAMIC PROGRAMMING



12

CHAPTER

## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand :**

- Basic elements of dynamic programming
- Principle of optimality
- Divide and conquer approach vs dynamic programming
- Dynamic programming formulation of LCS
- Chain matrix multiplication
- Dynamic programming formulation of chain matrix multiplication

# Chapter 12 DYNAMIC PROGRAMMING

## INSIDE THIS CHAPTER

- 12.1 Introduction
- 12.3 Chain Matrix Multiplication

- 12.2 Longest Common Subsequence (LCS)

## 12.1 INTRODUCTION

In this chapter, we discuss an important algorithm design technique, called dynamic programming (or DP for short). The technique is among the powerful designing algorithms for optimization problems. This is true for two reasons. Dynamic programming solutions are based on a few common elements. Dynamic programming problems are typically optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide and conquer, in the sense that it breaks problem down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide and conquer solutions are not usually efficient.

The basic elements that characterize a dynamic programming algorithm are:

**Substructure:** Decompose your problem into smaller (and hopefully simple) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.

**Table-structure:** Store the answers to the subproblem in a table. This is done because subproblem solutions are reused many times.

**Bottom-up computation:** Combine solutions on smaller subproblems to solve larger subproblems.

The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called **formulation** of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to be applicable.

**Optimal substructure (Sometimes called “Principle of optimality”):** It states that for the global problem to be solved optimally, each subproblem should be solved optimally (Not

all optimization problems satisfy this. Sometimes it is better to lose a little on one subproblem in order to make a big gain on another.)

**Polynomially many subproblems:** An important aspect to the efficiency of DP is that the total number of subproblems to be solved should be at most a polynomial number.

## 12.2 LONGEST COMMON SUBSEQUENCE (LCS)

Let us think of character strings as sequence of characters. Given two sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Z = \langle z_1, z_2, \dots, z_m \rangle$ , we say that  $Z$  is a subsequence of  $X$  if there is strictly increasing sequence of  $k$  indicates,  $\langle i_1, i_2, \dots, i_k \rangle$  ( $1 \leq i_1 < i_2 < \dots < i_k \leq n$ ) such that  $Z = \langle x_{i1}, x_{i2}, \dots, x_{im} \rangle$ . For example, let  $X = \langle \text{ABRACADABRA} \rangle$  and  $Z = \langle \text{AADAA} \rangle$ , then  $Z$  is a subsequence of  $X$ .

Given two strings  $X$  and  $Y$  the longest common subsequence of  $X$  and  $Y$  is longest sequence  $Z$  that is a subsequence of both  $X$  and  $Y$ .

**Example 12.1** Let  $X = \langle \text{ABRACADABRA} \rangle$  and let  $Y = \langle \text{YABBADABBADOO} \rangle$

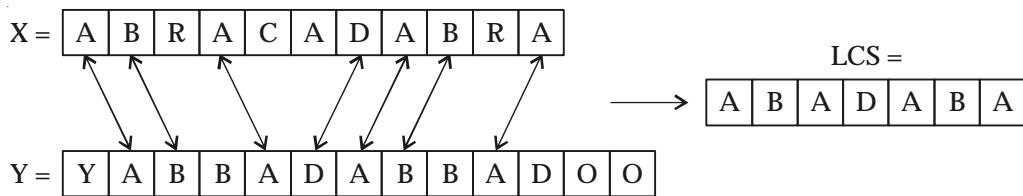


Fig. 12.1. A example of the LCS of two strings X and Y.

**Note:** LCS of two strings is not always unique. For example, the LCS of  $\langle \text{ABC} \rangle$  and  $\langle \text{BAC} \rangle$  is either  $\langle \text{AC} \rangle$  or  $\langle \text{BC} \rangle$ .

### 12.2.1 DP Formulation of LCS

The simple brute-force solution to the problem would be to try all possible subsequences from one string and search for matches in the other string, but this is hopelessly inefficient, since there are an exponential number of possible subsequences.

Instead, we will derive a dynamic programming solution. In typical DP fashion, we need to break the problem into smaller pieces. There are many ways to do this for strings, but it turns out for this problem that considering all pairs of prefixes will suffice for us. A prefix of a sequence is just an initial string of values,  $X_i = \langle x_1, x_2, \dots, x_i \rangle$ .  $X_0$  is the empty sequence.

The idea will be to compute the longest common subsequence for every possible pairs of prefixes. Let  $c[i, j]$  denote the length of the longest common subsequence of  $X_i$  and  $Y_j$ . For example, in the above case we have  $X_5 = \langle \text{ABRAC} \rangle$  AND  $Y_6 = \langle \text{YABBAD} \rangle$ . Their longest common subsequence is  $\langle \text{ABA} \rangle$ . Thus  $C[5, 6] = 3$ .

Here question is which of the  $c[i, j]$  values do we compute? Since, we don't know which will lead to the final optimum, we compute all of them. Eventually we are instead in  $c[m, n]$  since this will be the LCS of the two entire strings. The idea is to compute  $c[i, j]$  assuming that we already known the value of  $c[i', j']$ . For  $i' \leq i$  and  $j' \leq j$  (but not both equal). Here are the possible cases.

**Case 1: Basis:**  $c[i, 0] = c[j, 0] = 0$ . If either sequence is empty, then the longest common subsequence is empty.

### Case 2: Last characters match

Suppose  $x_i = y_j$ . For example: Let  $X_i = \langle ABCA \rangle$  and let  $Y_j = \langle DACA \rangle$ . Since both end in A, we claim that the LCS must also end in A. Since the A is part of the LCS we may find the overall LCS by removing A from both sequences and taking the LCS of  $X_{i-1} = \langle ABC \rangle$  and  $Y_{j-1} = \langle DAC \rangle$  which is  $\langle AC \rangle$  and then adding A to the end, giving  $\langle ACA \rangle$  as the answer. (At first you might object: But how did you know that these two A's matched with each other. The answer is that we don't but it will not make the LCS any smaller if we do.) This is illustrated in Fig. 12.2.

if  $x_i = y_j$  then  $c[i, j] = c[i - 1, j - 1] + 1$

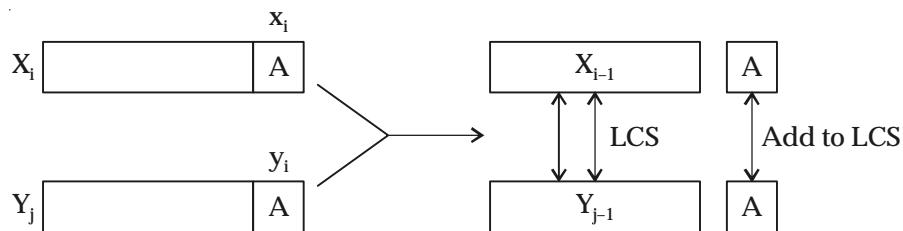


Fig. 12.2. LCS finding if last characters match.

### Case 3: Last characters don't match

Suppose that  $x_i \neq y_j$ . In this case  $x_i$  and  $y_j$  cannot both be in the LCS (Since they would have to be last character of the LCS). Thus either  $x_i$  is not part of LCS, or  $y_j$  is not part of the LCS (and possibly both are not part of LCS).

At this point it may be tempting to try to make a = “smart” choice. By analyzing the last few characters of  $x_i$  and  $y_j$ , perhaps we can figure out which character is best to discard. However, this approach is doomed to failure. Instead, our approach is to take advantage of the fact that we have already precomputed smaller subproblems, and use these results to guide us.

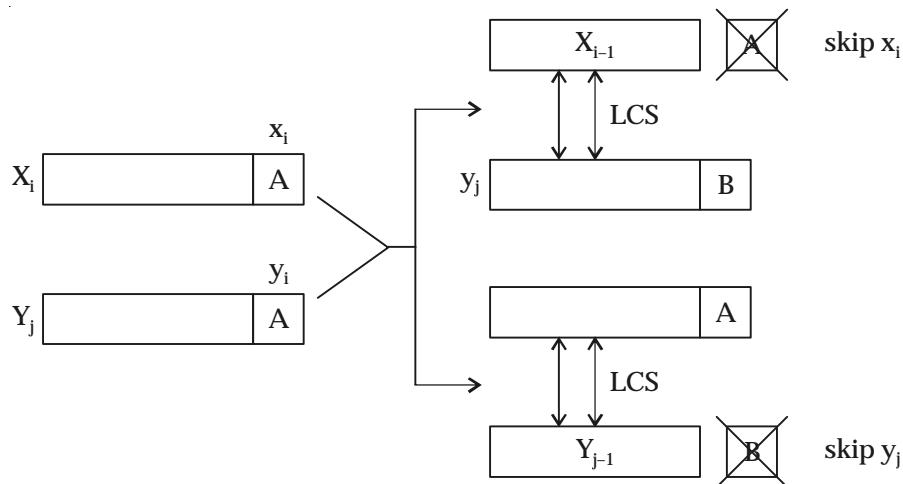


Fig. 12.3. LCS finding if last characters match.

In the first case ( $x_i$  is not in the LCS) the LCS of  $x_i$  and  $y_j$  is the LCS of  $x_{i-1}$  and  $y_j$  which is  $c[i-1, j]$ . In the second case ( $y_j$  is not in the LCS) the LCS is the LCS of  $x_i$  and  $y_{j-1}$  which is  $c[i, j-1]$ . We do not know which is the case, so we try both and take the one that gives us the longer LCS. This is illustrated in Fig. 12.3.

If  $x_i \neq y_j$  then  $c[i, j] = \max(c[i-1, j], c[i, j-1])$ . combining these observations we have the following formulation.

$$c[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

### 12.2.2 Implementing the Formulation

The task is now to implement this formulation. We concentrate only on computing the maximum length of the LCS. Later we will see how to extract the actual sequence. We will store some helpful pointers in the array  $b[0, \dots, m, 0, \dots, n]$ . Let us see at the algorithm steps.

#### Algorithm for Building LCS Table and Finding LCS Length

```
LCS-Length(X[1 . . . m], Y[1 . . . n]) // compute LCS table
1. int c[0 . . . m, 0 . . . n]
2. for i = 0 to m // init column 0
3.   c[i, 0] = 0; b[i, 0] = skip X
4. for j = 0 to n // init row 0
5.   c[0, j] = 0; b[0, j] = skip Y
6. for i = 1 to m // fill rest of the table
7. for j = 1 to n
8. if (X[i] = Y[j]) // take X[i] (Y[j]) for LCS
9.   c[i, j] = c[i - 1, j - 1] + 1, b[i, j] = add XY
10. else if (c[i - 1, j] >= c[i, j - 1]) // X[i] not in LCS
11.   c[i, j] = c[i - 1, j], b[i, j] = skip X
12. else // Y[j] not in LCS
13.   c[i, j] = c[i, j - 1], b[i, j] = skip Y
14. return c[m, n] // return length of LCS
```

**Running Time:** The running time of the algorithm is clearly  $O(mn)$  since there are two nested for loops with  $m$  and  $n$  iterations, respectively. The algorithm uses  $O(mn)$  space.

### 12.2.3 Extracting the Actual Sequence

Extracting the actual sequence of LCS is done by using the back pointers stored in  $b[0, \dots, m, 0, \dots, n]$ . Intuitively  $b[i, j] = \text{add XY}$  means that  $X[i]$  and  $Y[j]$  together form the last character of the LCS.

So we take this common character and continue with entry  $b[i - 1, j - 1]$  to the northwest (↖). If  $b[i, j] = \text{skip X}$ , then we know that  $X[i]$  is not in the LCS, and so we skip it and go to  $b[i - 1, j]$  above us (↑). Similarly, if  $b[i, j] = \text{skip Y}$ , then we know that  $Y[j]$  is not in the LCS, and so we skip it and go to  $b[i, j - 1]$  to the left (←). Following these back pointers and outputting a character with each diagonal move gives the final subsequence.

#### Algorithm for Extracting LCS

```

Extract LCS (b, X, i, j)
1. get LCS (X[1 .... m], Y[1 .... n], b[0 .... m, 0 .... n]
2. LCS string = empty string
3. i = m; j = n                                // start at lower right
4. while (i != 0 & j != 0)                      // go until upper left
5. switch b[i, j]
6. case add XY:                               // add X[i] (= Y[j])
7.   add X[i] to front of LCS string
8.   i--; j--; break
9. case skip X : i --; break                  // skip X[i]
10. case skip Y : j --; break                 // skip Y[j]
11. return LCS string.

```

**Example 12.2** Determine an LCS of  $\langle A, B, C, B, D, A, B \rangle$  and  $\langle B, D, C, A, B, A \rangle$  by using dynamic programming. Find its running time.

**Solution:** Let  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$

Then  $X[1] = A, X[2] = B, X[3] = C, X[4] = B, X[5] = D, X[6] = A$  and  $X[7] = B$   
 $Y[1] = B, Y[2] = D, Y[3] = C, Y[4] = A, Y[5] = B$  and  $Y[6] = A$

		Y → 0	1	2	3	4	5	6 n = 7	
		X ↓	y <sub>i</sub>	B	D	C	A	B	A
		x <sub>i</sub>	0	0	0	0	0	0	0
1	A	0	0	↑ 0	↑ 0	↑ 0	↖ 1	↖ 1	↖ 1
		0	0	↖ 1	↖ 1	↖ 1	↑ 1	↖ 2	↖ 2
2	B	0	0	↖ 1	↑ 1	↑ 1	↖ 2	↑ 2	↑ 2
		0	0	↖ 1	↖ 1	↖ 2	↖ 2	↑ 2	↑ 2
3	C	0	0	↑ 1	↑ 1	↖ 2	↖ 2	↑ 2	↑ 2
		0	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	↖ 3
4	B	0	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	↖ 3
		0	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
5	D	0	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
		0	0	↑ 1	↑ 2	↖ 3	↑ 3	↖ 4	↑ 4
6	A	0	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
		0	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4
7	B	0	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

m = 8      ↑ Start here

(LCS length table with back pointers included)

So,  $X = \langle A, (\textcircled{B}), (\textcircled{C}), (\textcircled{B}), (\textcircled{D}), (\textcircled{A}), (\textcircled{B}) \rangle$

$Y = \langle (\textcircled{B}), D, (\textcircled{C}), (\textcircled{A}), (\textcircled{B}), A, \rangle$

$\text{LCS}(X, Y) = \langle B, C, A, B \rangle$

Running time =  $T(n) = O(mn)$  (where  $m \rightarrow$  number of columns and  $n \rightarrow$  number of rows)

**Note:** (How to fill the table)

- First fill 1st row and 1st column with zeros.
- If corresponding Y's and X's row and column respectively matching then corner (left side box of diagonal element) + 1.
- If not matching then find the maximum element between top and left side value.

**Example 12.3** Determine an LCS of  $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$  and  $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$ .

**Solution:** Let

$$X = \langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle \quad \text{and} \quad Y = \langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle.$$

Then

$$X[1] = 1, X[2] = 0, X[3] = 0, X[4] = 1, X[5] = 0, X[6] = 1, X[7] = 0 \text{ and } X[8] = 1.$$

$Y[1] = 0, Y[2] = 1, Y[3] = 0, Y[4] = 1, Y[5] = 1,$   
 $Y[6] = 0, Y[7] = 1, Y[8] = 1 \quad \text{and} \quad Y[9] = 0.$

		Y → 0	1	2	3	4	5	6	7	8	9	n = 10
		X ↓	$y_i$	0	1	0	1	1	0	1	1	0
X ↓	x_i	0	0	0	0	0	0	0	0	0	0	
1	1	0	0	↑ 1	↖ 1	← 1	↖ 1	↖ 1	↖ 1	↖ 1	↖ 1	↖ 1
2	0	0	↖ 1	↑ 1	↖ 2	↖ 2	↖ 2	↖ 2	↖ 2	↖ 2	↖ 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↖ 2	↖ 2	↖ 3	↖ 3	↖ 3	↖ 3	↖ 3
4	1	0	↑ 1	↖ 2	↖ 2	↖ 3	↖ 3	↖ 3	↖ 4	↖ 4	↖ 4	↖ 4
5	0	0	↖ 1	↑ 2	↖ 3	↖ 3	↖ 3	↖ 4	↖ 4	↖ 4	↖ 4	↖ 5
6	1	0	↑ 1	↖ 2	↖ 3	↖ 4	↖ 4	↖ 4	↖ 5	↖ 5	↖ 5	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↖ 4	↖ 4	↖ 5	↖ 5	↖ 5	↖ 5	↖ 6
8	1	0	↑ 1	↖ 2	↖ 3	↖ 4	↖ 5	↖ 5	↖ 6	↖ 6	↖ 6	↑ 6

$m = 9$       ↑  
Start here

$$X = < (1, 0), 0, (1, 0), (1, 0), 0, (1) >$$

$$Y = < 0, (1, 0), 1, (1, 0), (1, 0), (1, 0) >$$

$$\text{LCS}(X, Y) = < 1, 0, 1, 0, 1, 1 >$$

### 12.3 CHAIN MATRIX MULTIPLICATION

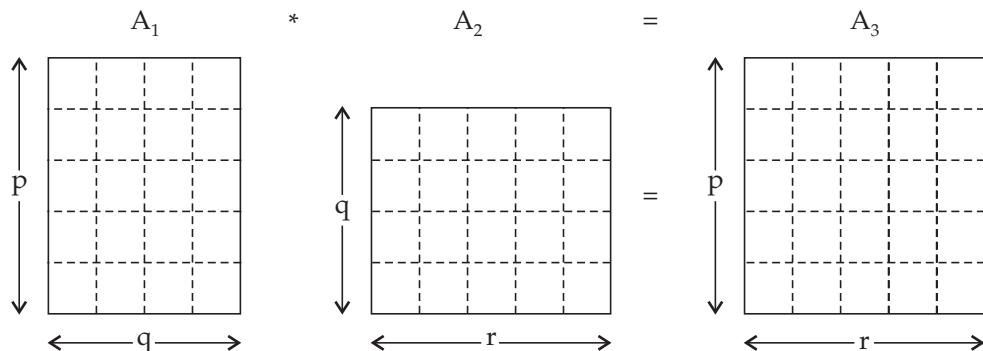
This problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in compiler design for code optimization and in database for query optimization. We will study the problem in a very restricted instance, where the dynamic programming issues are easiest to see.

Suppose that we wish to multiply a series of matrices  $A_1 A_2 \dots A_n$ . Matrix multiplication is an associative but not commutative operation. This means that we are free to parenthesize the above multiplication however we like, but we are not free to arrange the order of the

matrices. Also recall that when two (non-square) matrices are being multiplied, there are restrictions on the dimensions. A  $p \times q$  matrix has  $p$  rows and  $q$  columns. You can multiply a  $p \times q$  matrix  $A_1$  and a  $q \times r$  matrix  $A_2$  and the result will be a  $p \times r$  matrix  $A_3$ . (The number of columns of  $A_1$  must equal the number of rows of  $A_2$ ). In particular for

$$1 \leq i \leq P \text{ and } 1 \leq j \leq r, A_3[i, j] = \sum_{k=1}^q A_1[i, k] A_2[k, j].$$

This corresponds to the (hopefully familiar) rule that the  $[i, j]$  entry if  $A_3$  is the dot product of the  $i^{\text{th}}$  (horizontal) row of  $A_1$  and the  $j^{\text{th}}$  (vertical) column of  $A_2$ . Observe that there are  $pr$  total entries in  $A_3$  and each takes  $O(q)$  time to compute, thus the total time to multiply these two matrices is proportional to the product of the dimensions  $pqr$ .



**Fig. 12.4.** Matrix multiplication.

Note that although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices:  $A_1$  be  $5 \times 4$ ,  $A_2$  be  $4 \times 6$  and  $A_3$  be  $6 \times 2$ .

$$\text{Mult Cost } [(A_1, A_2) A_3] = (5 \times 4 \times 6) + (5 \times 6 \times 2) = 180$$

$$\text{Mult Cost } [(A_1 (A_2, A_3))] = (4 \times 6 \times 2) + (5 \times 4 \times 2) = 88$$

Even for the small example, considerable saving can be achieved by reordering the evaluation sequence.

### 12.3.1 Chain Matrix Multiplication Problem

Given a sequence of matrices  $A_1, A_2, \dots, A_n$  and dimensions  $p_0, p_1, \dots, p_n$  where  $A_i$  is of dimension  $p_{i-1} \times p_i$  determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.

**Note:** Chain matrix multiplication algorithm doesn't perform the multiplications, it just determines the best order in which to perform the multiplications.

### 12.3.2 Native Algorithm

We could write a procedure which tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If you have  $n$  items, then  $n - 1$  places where you could break the list with the outermost pair of parenthesis, namely just after the 1st item, just after the 2nd item, etc. and just after the  $(n - 1)$ st item. When we split just after the  $k$ th item, we create two sublists to be parenthesized, one with  $k$  items and other with  $(n - k)$  items. Then, we could consider all the ways of parenthesizing these. Since these are independent choices, if there are  $L$  ways to parenthesize the left sublist and  $R$  way to parenthesize the right sublist, then the total  $L \cdot R$ . This suggest the following recurrence for  $P(n)$ , the number of different ways of parenthesizing  $n$  items.

$$P(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

This is related to a famous function in combinatorics called the Catalan numbers (which in turn is related to the number of different binary trees on  $n$  nodes). In particular  $P(n) = C(n - 1)$ , where  $C(n)$  is the  $n$ th Catalan number which can be written as

$$C(n) = \frac{1}{n+1} \binom{2n}{n}$$

Applying Stirling's formula, we find that  $C(n) \approx \frac{1}{4\pi n} \left(\frac{2n}{e}\right)^{2n}$ . Since  $4^n$  is exponential and  $n^{3/2}$  is just polynomial, the exponential will dominate, implying that functions grows very fast. Thus, this will not be practical except for very small  $n$ . Therefore we conclude that, brute force is not an option.

### 12.3.3 Dynamic Programming Approach

This problem, like other dynamic programming problems involves determining a structure. We want to break the problem into subproblems, whose solutions can be combined to solve the global problem. As is common to any DP solution, we need to find some way to break the problem into smaller subproblems, and we need to determine a recursive formulation, which represents the optimum solution to each problem in terms of solutions to the subproblems. Let us think how we can do this.

Since matrices can not be reordered, it make sense to think about sequences of matrices. Let  $A_{i \dots j}$  denote the result of multiplying matrices  $i$  through  $j$ . It is easy to see that  $A_{i \dots j}$  is  $p_{i-1} \times p_j$  matrix. (Think about this for a second to be sure you see why). Now, in order to determine how to perform this multiplication optimally, we need to make many decisions. What we want to do is to break the problem into problems of a similar structure. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is, for any  $k$ ,  $1 \leq k \leq n - 1$ .

$$A_{1 \dots n} = A_{1 \dots k} \cdot A_{k+1 \dots n}$$

Thus the problem of determining the optimal sequence of multiplications is broken up into two questions: how do we decide where to split the chain (what is  $k$ ) and how do we parenthesize the subchains  $A_1 \dots k$  and  $A_{k+1} \dots n$ ? The subchain problems can be solved recursively, by applying the same scheme.

So, let us think about the problem of determining the best value of  $k$ . At this point, you may be tempted to consider some clever ideas. For example, since we want matrices with small dimensions pick the value of  $k$  that minimizes  $p_k$ . Although this is not a bad idea, in principle. (After all it might work. It just turns out that it doesn't in this case. This takes a bit of thinking, which you should try). Instead, as it is true in almost all dynamic programming solutions, we will do the dumbest things of simply considering all possible choices of  $k$  and taking the best of them. Usually trying all possible choices is bad, since it quickly leads to an exponential number of total possibilities. What saves us here is that there are only  $O(n^2)$  different

sequences of matrices. (There are  $\binom{n}{2} = \frac{n(n-1)}{2}$  ways of choosing  $i$  and  $j$  to form  $A_i \dots j$  to be precise). Thus, we don't encounter the exponential growth.

Notice that our chain matrix multiplication problem satisfies the principle of optimality, because once we decide to break the sequence into the product  $A_1 \dots k \cdot A_{k+1} \dots n$ , we would compute each subsequence optimally. This is for the global problem to be solved optimally, the subsequence must be solved optimally as well.

#### 12.3.4 Dynamic Programming Formulation

We will store the solutions to the subproblem in a table, and build the table in a bottom up manner. For  $1 \leq i \leq j \leq n$ , let  $m_{ij}$  denote the minimum number of multiplications needed to compute  $A_i \dots j$ . The optimum cost can be described by the following recursive formula.

- (i) if  $i = j$ : then the sequence contains only one matrix and so the cost is 0. (There is nothing to multiply). Thus  $m_{ij} = 0$ .
- (ii) if  $i < j$ : then we are asking about the product  $A_i \dots j$ . This can be split by considering each  $k$ ,  $i \leq k < j$ , as  $A_i \dots k$  time  $A_{k+1} \dots j$ .

The optimum times to compute  $A_i \dots k$  and  $A_{k+1} \dots j$  are by definition  $m_{ik}$  and  $m_{(k+1)j}$  respectively. We may assume that these values have been computed previously and are already stored in our array. Since  $A_i \dots k$  is a  $p_{i-1} \times p_k$  matrix, and  $A_{k+1} \dots j$  is a  $p_k \times p_j$  matrix, the time to multiply them is  $p_{i-1} \dots 1' p_k p_j$ . This suggests the following recursive rule for computing  $m_{ij}$ .

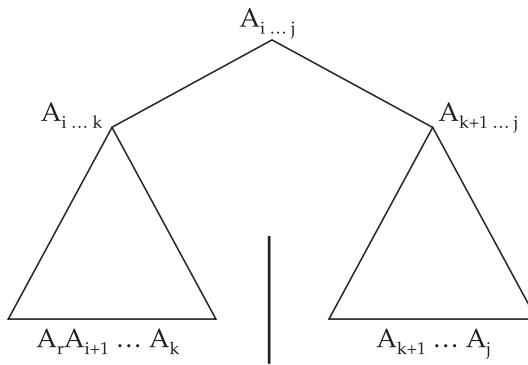


Fig. 12.5. Dynamic programming formulation.

$$m_{ij} = 0$$

$$m_{ij} = \min_{i \leq k < j} (m_{ik} + m_{(k+1)j} + p[i-1] p[k] p[j]) \text{ for } i < j$$

It is not hard to convert this rule into a procedure, which is given below. The only tricky part is arranging the order in which to compute the values. In the process of computing  $m_{ij}$  we need to access values  $m_{ik}$  and  $m_{(k+1)j}$  for  $k$  lying between  $i$  and  $j$ . This suggests that we should organize our computation according to the number of matrices in the subsequence. Let  $L = j - i + 1$  denote the length of the subchain being multiplied. The subchains of length 1  $m_{ij}$  are trivial to compute. Then we build up by computing the subchains of lengths 2, 3, ...,  $n$ . The final answer is  $m_{1n}$ . We need to be a little careful in setting up the loops. If a subchain of length  $L$  starts at position  $i$ , then  $j = i + L - 1$ . Since, we want  $j \leq n$ , this means that  $i + L - 1 \leq n$ ; or in other words,  $i \leq n - L + 1$ . So our loop for  $i$  runs from 1 to  $n - L + 1$ . The algorithm is presented below.

The running time of the procedure is  $\Theta(n^3)$ . We'll leave this as an exercise in solving sums, but the key is that there are three nested loops, and each can iterate at most  $n$  times.

### 12.3.5 Extracting the Final Sequence

Extracting the actual multiplication sequence is a fairly easy extension. The basic idea is to level a split marker indicating what the best split is, that is the value of  $k$  that leads to the minimum value of  $m_{ij}$ . We can maintain a parallel array  $s_{ij}$  in which we will store the value of  $k$  providing the optimal split. For example, suppose that  $s_{ij} = k$ . This tells us that the best way to multiply the subchain  $A_{i \dots j}$  is to first multiply the subchain  $A_{i \dots k}$  and then multiply the subchain  $A_{k+1 \dots j}$  and finally multiply these together.

#### Algorithm for Chain-Matrix Multiplication

```

CHAIN-MATRIX-ORDER (array p[1 . . . n])
1. array s[1 . . . n - 1, 2 . . . n]
2. for i = 1 to n do mij = 0 // initialization
3. for L = 2 to n do // L = length of subchain
4. for i = 1 to n - L + 1 do
5.   j = i + L - 1
6.   mij = ∞
7.   for k = i to j - 1 do // check all splits
8.     q = mik + m(k+1)j + p[i - 1] * p[k] * p[j]
9.     if (q < mij)
10.    mij = q
11.    sij = k
12. return m1n (final cost) and s(splitting markers)

```

**Running Time:** The running time of CHAIN-MATRIX-ORDER algorithm is  $O(n^3)$ .

The actual multiplication algorithm uses the  $s_{ij}$  value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices  $A[1 \dots n]$  and that  $s_{ij}$  is global to this recursive procedure. The recursive procedure EXTRACT-MULT does this computation.

### Algorithm for Extracting Optimum Sequence

```

EXTRACT-MULT (i, j)
1. if (i = j)                                // section 12.3.4, part (i)
2. return Ai
3. else k = sij
4. X = mult (i, k)                          // X = Ai . . . Ak
5. Y = mult (k + 1, j)                      // Y = Ak+1 . . . Aj
6. return X * Y                            // multiply matrices X and Y

```

**Example 12.4** Find an optimal parenthesization of a matrix - chain product whose sequence of dimensions is  $\langle 10, 20, 50, 1, 100 \rangle$ .

**Solution:**

	1	2	3	4
1	$m_{11}$	$m_{12}$	$m_{13}$	$m_{14}$
2		$m_{22}$	$m_{23}$	$m_{24}$
3			$m_{33}$	$m_{34}$
				$m_{44}$

**Given dimensions:**  $p[0] = 10, p[1] = 20, p[2] = 50, p[3] = 1, p[4] = 100$

As  $i = j$  in case of  $m_{11}, m_{22}, m_{33}$  and  $m_{44}$  so they are all equal to zero.

$$\begin{aligned}
m_{12} &= \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\
&= \min_{i \leq k < 1} [m_{11} + m_{22} + p[0] \cdot p[1] \cdot p[2]] \\
&= 0 + 0 + (10 \times 20 \times 50) = 10,000
\end{aligned}$$

$$\begin{aligned}
m_{23} &= \min_{2 \leq k \leq 2} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\
&= \min_{2 \leq k \leq 2} [m_{22} + m_{33} + p[1] \cdot p[2] \cdot p[3]] \\
&= 0 + 0 + (20 \times 50 \times 1) = 1000
\end{aligned}$$

$$m_{34} = \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i] \cdot p[k] \cdot p[j]]$$

$$\begin{aligned}
&= \min_{3 \leq k \leq 3} [m_{33} + m_{44} + p[2] \cdot p[3] \cdot p[4]] \\
&= 0 + 0 (50 \times 1 \times 100) = 5000 \\
m_{13} &= \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\
&= \min_{1 \leq k \leq 2} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]
\end{aligned}$$

When  $k = 1, m_{13} = m_{11} + m_{23} + p[0] \cdot p[1] \cdot [3]$   
 $= 0 + 1000 + 10 \cdot 20 \cdot 1 = 1000 + 200 = 1200$

When  $k = 2, m_{13} = m_{12} + m_{33} + p[0] \cdot p[2] \cdot p[3]$   
 $= 10000 + 0 + (10 \times 50 \times 1)$   
 $= 10500$

For  $k = 1, m_{13}$  is minimum i.e., 1200.

$$\begin{aligned}
m_{24} &= \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\
&= \min_{2 \leq k \leq 3} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]
\end{aligned}$$

When  $k = 2, m_{24} = m_{22} + m_{34} + p[1] \cdot p[2] \cdot p[4]$   
 $= 0 + 5000 + (20 \times 50 \times 100)$   
 $= 5000 + 100000 = 105000$

When  $k = 3, m_{24} = m_{23} + m_{44} + p[1] \cdot p[3] \cdot p[4]$   
 $= 1000 + 0 + (20 \times 1 \times 100)$   
 $= 1000 + 2000 = 3000$

For  $k = 3, m_{24}$  is minimum i.e., 3000.

$$\begin{aligned}
m_{14} &= \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\
&= \min_{i \leq k \leq j-3} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]
\end{aligned}$$

When  $k = 1, m_{14} = m_{11} + m_{24} + p[0] \cdot p[1] \cdot p[4]$   
 $= 0 + 3000 + (10 \times 20 \times 100)$   
 $= 23000$

When  $k = 2, m_{14} = m_{12} + m_{34} + p[0] \cdot p[2] \cdot p[4]$   
 $= 10,000 + 5000 + (10 \times 50 \times 100)$   
 $= 65000$

When  $k = 3, m_{14} = m_{13} + m_{44} + p[0] \cdot p[3] \cdot p[4]$   
 $= 1200 + 0 + (10 \times 1 \times 100)$   
 $= 2200$

For  $k = 3$ ,  $m_{14}$  is minimum i.e., 2200.

	1	2	3	4
1	0	10000	1200	2200
Optimal parenthesization :	2	0	1000	3000
3		0	5000	
4			0	

**Example 12.5** Find the optimal parenthesization of a matrix-chain product whose sequence of dimensions is  $\langle 5, 10, 3, 12, 5, 50 \rangle$

**Solution:** P :  $\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 50 \\ \hline 5 & 10 & 3 & 12 & 5 & 50 \end{array}$

$$m_1 = p[0] \times p[1] = 5 \times 10$$

$$m_2 = p[1] \times p[2] = 10 \times 3$$

$$m_3 = p[2] \times p[3] = 3 \times 12$$

$$m_4 = p[3] \times p[4] = 12 \times 5$$

$$m_5 = p[4] \times p[5] = 5 \times 50$$

	1	2	3	4	5
1	$m_{11}$	$m_{12}$	$m_{13}$	$m_{14}$	$m_{15}$
2		$m_{22}$	$m_{23}$	$m_{24}$	$m_{25}$
3			$m_{33}$	$m_{34}$	$m_{35}$
4				$m_{44}$	$m_{45}$
5					$m_{55}$

$$m_{12} = \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

Here  $i = 1, j = 2, k = 1$

$$m_{12} = m_{11} + m_{22} + p[0] \cdot p[1] \cdot p[2]$$

$$m_{12} = 0 + 0 + (5 \times 10 \times 3) = 150$$

$$m_{23} = \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

Here  $i = 2, k = 2, j = 3$

$$m_{23} = m_{22} + m_{33} + p[1] \cdot p[2] \cdot p[3]$$

$$= 0 + 0 + (10 \times 3 \times 12) = 360$$

$$m_{34} = \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

Here  $i = 3, k = 3, j = 4$

$$m_{34} = m_{33} + m_{44} + p[2] \cdot p[3] \cdot p[4]$$

Here  $i = 4, k = 4, j = 5$

$$m_{45} = m_{44} + m_{55} + p[3] \cdot p[4] \cdot p[5]$$

$$= 0 + 0 + (12 \times 5 \times 50) = 3000$$

$$\begin{aligned}m_{13} &= \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\&= \min_{i \leq k \leq 2} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]\end{aligned}$$

When  $k = 1$ ,  $m_{13} = m_{13} + m_{11} + m_{23} + p[0] \cdot p[1] \cdot p[3]$   
 $= 0 + 360 + (5 \times 10 \times 12) = 960$

When  $k = 2$ ,  $m_{13} = m_{12} + m_{33} + p[0] \cdot p[2] \cdot p[3]$   
 $= 150 + 0 + (5 \times 3 \times 12) = 330$

For  $k = 2$ ,  $m_{13}$  is minimum i.e., 330.

$$m_{24} = \min_{2 \leq k \leq 3} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

When  $k = 2$ ,  $m_{24} = m_{22} + m_{34} + p[1] \cdot p[2] \cdot p[4]$   
 $= 0 + 180 + (10 \times 3 \times 5) = 330$

When  $k = 3$ ,  $m_{24} = m_{23} + m_{44} + p[1] \cdot p[3] \cdot p[4]$   
 $= 360 + 0 + (10 \times 12 \times 5) = 960$

For  $k = 2$ ,  $m_{24}$  is minimum i.e., 330.

$$m_{35} = \min_{3 \leq k \leq 4} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

When  $k = 3$ ,  $m_{35} = m_{33} + m_{45} + p[2] \cdot p[3] \cdot p[5]$   
 $= 0 + 3000 + (3 \times 12 \times 50) = 4800$

When  $k = 4$ ,  $m_{35} = m_{34} + m_{55} + p[2] \cdot p[4] \cdot p[5]$   
 $= 180 + 0 + (3 \times 5 \times 50) = 930$

For  $k = 4$ ,  $m_{35}$  is minimum i.e., 930.

$$m_{14} = \min_{1 \leq k \leq 3} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

When  $k = 1$ ,  $m_{14} = m_{11} + m_{24} + p[0] \cdot p[1] \cdot p[4]$   
 $= 0 + 330 + (5 \times 10 \times 5) = 330 + 250 = 580$

When  $k = 2$ ,  $m_{14} = m_{12} + m_{34} + p[0] \cdot p[2] \cdot p[4]$   
 $= 150 + 180 + (5 \times 3 \times 5)$   
 $= 330 + 75 = 405$

When  $k = 3$ ,  $m_{14} = m_{13} + m_{44} + p[0] \cdot p[3] \cdot p[4]$   
 $= 330 + 0 + (5 \times 12 \times 5) = 330 + 300 = 630$

For  $k = 2$ ,  $m_{14}$  is minimum i.e., 405.

$$m_{25} = \min_{2 \leq k \leq 4} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

When  $k = 2$ ,  $m_{25} = m_{22} + m_{35} + p[1] \cdot p[2] \cdot p[5]$   
 $= 0 + 930 + (10 \times 3 \times 50) = 2430$

When  $k = 3$ ,

$$\begin{aligned} m_{25} &= m_{23} + m_{45} + p[1] \cdot p[3] \cdot p[5] \\ &= 360 + 3000 + (10 \times 12 \times 50) \\ &= 3360 + 6000 = 9360 \end{aligned}$$

When  $k = 4$ ,

$$\begin{aligned} m_{25} &= m_{24} + m_{55} + p[1] \cdot p[4] \cdot p[5] \\ &= 330 + (10 \times 5 \times 50) = 330 + 2500 = 2830 \end{aligned}$$

For  $k = 2$ ,  $m_{25}$  is minimum i.e., 2430.

$$m_{15} = \min_{1 \leq k \leq 4} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

When  $k = 1$ ,

$$\begin{aligned} m_{15} &= m_{11} + m_{25} + p[0] \cdot p[1] \cdot p[5] \\ &= 0 + 2430 + (5 \times 10 \times 50) = 4930 \end{aligned}$$

When  $k = 2$ ,

$$\begin{aligned} m_{15} &= m_{12} + m_{35} + p[0] \cdot p[2] \cdot p[5] \\ &= 150 + 930 + (5 \times 30 \times 50) \\ &= 1080 + 750 = 1830 \end{aligned}$$

When  $k = 3$ ,

$$\begin{aligned} m_{15} &= m_{13} + m_{45} + p[0] \cdot p[3] \cdot p[5] \\ &= 330 + 3000 + (5 \times 12 \times 50) = 6330 \end{aligned}$$

When  $k = 4$ ,

$$\begin{aligned} m_{15} &= m_{14} + m_{55} + p[0] \cdot p[4] \cdot p[5] \\ &= 405 + 0 + (5 \times 5 \times 50) = 1655 \end{aligned}$$

So, for  $k = 4$ ,  $m_{15}$  is minimum i.e., 1655.

Now the optimum parenthesization becomes

	1	2	3	4	5
1	$m_{11}$ 0	$m_{12}$ 150	$m_{13}$ 330	$m_{14}$ 405	$m_{15}$ 1655
2		$m_{22}$ 0	$m_{23}$ 360	$m_{24}$ 330	$m_{25}$ 2430
3			$m_{33}$ 0	$m_{34}$ 180	$m_{35}$ 930
4				$m_{44}$ 0	$m_{45}$ 3000
5					$m_{55}$ 0

**Example 12.6** Let  $R(i, j)$  be the number of times that table entry  $m[i, j]$  is referenced while computing other table entries in a call of CHAIN-MATRIX-ORDER. Show that the total number of references for

the entire is  $\sum_{i=1}^n \sum_{j=i}^n R(i, j) = \frac{n^3 - n}{3}$ .

**Solution:** We can get a series of references like

$$\begin{aligned}
 & n \times (n-1) + (n-1) \times (n-2) + (n-2) \times (n-3) + \dots + 3 \times 2 + 2 \times 1 \\
 &= \sum_{i=0}^{n-2} (n-i)(n-i-1) \\
 \text{So, } &= \sum_{i=0}^{n-2} (n^2 - 2ni + i^2 - n + i) \\
 &= \sum_{i=0}^{n-2} n^2 - \sum_{i=0}^{n-2} 2ni + \sum_{i=0}^{n-2} i^2 - \sum_{i=0}^{n-2} n + \sum_{i=0}^{n-2} i \quad \dots(i)
 \end{aligned}$$

From equation (i)

$$\sum_{i=0}^{n-2} n^2 = n^2 \sum_{i=0}^{n-2} 1 = n^2 (n-1) \quad \dots(ii)$$

$$\begin{aligned}
 \sum_{i=0}^{n-2} 2ni &= 2n \sum_{i=0}^{n-2} i = 2n \frac{(n-2)(n-2+1)}{2} \left( \text{As } \sum_{i=0}^n i = \frac{n(n+1)}{2} \right) \\
 &= n(n-2)(n-1) \quad \dots(iii)
 \end{aligned}$$

$$\begin{aligned}
 \sum_{i=0}^{n-2} i^2 &= \frac{(n-2)(n-2+1)(2(n-2)+1)}{6} \\
 &= \frac{(n-2)(n-1)(2n-3)}{6} \quad \dots(iv)
 \end{aligned}$$

$$\sum_{i=0}^{n-2} n = n \sum_{i=0}^{n-2} 1 = n(n-1) \quad \dots(v)$$

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

Now substituting the values from equation (ii), (iii), (iv), (v) and (vi) in equation (i) we have

$$\begin{aligned}
 & \sum_{i=0}^{n-2} (n-i)(n-i+1) \\
 &= (n^3 - n^2 - n)(n^2 - 3n - 2) + \frac{(n^2 - 3n + 2)(2n + 3)}{6} - n(n-1) + \frac{n^2 - 3n + 2}{2} \\
 &= \frac{-12n + 2n^3 + 6n + 4n}{6} = \frac{2n^3 - 2n}{6} = \frac{n^3 - n}{6}
 \end{aligned}$$

**Example 12.7** Give a  $O(n)$  - time dynamic programming algorithm to compute the  $n$ th Fibonacci number.

**Solution:** Before writing the steps of algorithm, we need to represent the Fibonacci series in an array.

$$0 + 1 = 1$$

$$1 + 1 = 2$$

$$1 + 2 = 3$$

$$2 + 3 = 5$$

$3 + 5 = 8$  and so on.

0	1	1	2	3	5	8	13	...
---	---	---	---	---	---	---	----	-----

Here  $T(n) = O(n)$ . So it is a linear time Fibonacci series.

#### Algorithm for linear time Fibonacci series

```

1. int F(int n) {
2.     int X = 0, Y = 1, Z = i = 2
3.     if (n ≤ 1)
4.         return n
5.     else {
6.         while (i ≤ n) {
7.             Z = X + Y
8.             X = Y
9.             Y = Z
10.        }
11.    return Z
12. }
```

#### CHAPTER NOTES

---

- Dynamic programming is typically applicable to optimization problem in which we make a set of choices in order to arrive at an optimal solution.
- The principle of optimality states that in an optimal sequence of decisions or choices, each sub-sequences must be optimal.
- Basic elements of dynamic programming are: substructure, table-structure and bottom up computation.
- Dynamic programming algorithm for LCS computation takes  $O(mn)$  running time and  $O(mn)$  space complexity.
- Dynamic programming algorithm for chain-matrix-order takes  $O(n^3)$  running time.

## EXERCISES

---

1. Explain how dynamic programming is used to solve matrix-chain multiplication problem.
2. Explain how dynamic programming is used to solve LCS problem.
3. Describe a dynamic programming algorithm to find the maximum length common subsequence X of A and B (clearly define the notation). Analyze the running time and space requirements of your algorithm (in terms of big-O notation).
4. Determine as LCS of X = {babbbab} and Y = {bbbabbaab} by using dynamic programming and find its running time.
5. Find the optimal parenthesization of multiplying given matrix from  $m[A_1, A_4]$ .

$$A_0 = 30 \times 25$$

$$A_1 = 25 \times 15$$

$$A_2 = 15 \times 20$$

$$A_3 = 20 \times 12$$

$$A_4 = 12 \times 50$$

$$A_5 = 50 \times 05$$

# **GREEDY ALGORITHMS**

**13**  
CHAPTER

## **OBJECTIVES OF LEARNING**

**After going through this chapter, the reader would be able to understand :**

- Greedy algorithms vs Dynamic programming
- Greedy algorithms for Activity scheduling
- Elements of Greedy strategy
- Greedy-choice property
- Optimal substructure
- 0–1 Knapsack problem and fractional knapsack problem
- Correctness of fractional knapsack problem
- Prefix codes
- Huffman's algorithm

# Chapter 13 GREEDY ALGORITHMS

## INSIDE THIS CHAPTER

- |                                  |   |
|----------------------------------|---|
| 13.1 Introduction                | 13.2 Activity Selection Problem/Activity scheduling |
| 13.3 Elements of Greedy Strategy | 13.4 Knapsack Problem (Rucksack Problem)            |
| 13.5 Huffman Coding              |   |

## 13.1 INTRODUCTION

In many optimization algorithms a series of selections need to be made. In dynamic programming we saw one way to make these selections. Namely, the optimal solution is described in a recursive manner, and then is computed “bottom-up”. Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times. Today we will consider an alternative design technique, called **greedy algorithms**. This method typically leads to simple and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming. We will give some examples of problems that can be solved by greedy algorithms. Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (non-optimal solution strategies) are often used in finding good approximations.

### Greedy Algorithm Versus Dynamic Programming

<b>Greedy Algorithms</b>	<b>Dynamic Programming</b>
(i) A problem in which set of feasible solution are there. We have to choose the optimal solution from that feasible solution.	(i) A problem is divided into number of subproblems and each subproblem depends on next subproblem in a hierarchical way.
(ii) It is used for optimization problem.	(ii) It is used for optimization problem. Optimal solution is described in a recursive manner.
(iii) Solution is computed in a top-down approach.	(iii) Solution is computed in a bottom-up manner.
(iv) In greedy method only one decision sequence is generated.	(iv) In dynamic programming many decision sequences may be generated. However, sequence containing suboptimal sequences can not be optimal.

## 13.2 ACTIVITY SELECTION PROBLEM/ACTIVITY SCHEDULING

Activity scheduling is a very simple scheduling problem. We are given a set  $S = \{1, 2, \dots, n\}$  of  $n$  activities that are to be scheduled to use some resource, where each activity must be started at a given start time  $s_i$  and ends at a given finish time  $f_i$ . For example, there might be lectures that are to be given in a lecture hall, where the lecture times have been set up in advance or requests for boats to use a repair facility while they are in port.

Because there is only one resource, and some start and finish times may overlap (and two lectures cannot be given in the same room at the same time), not all the requests can be honored. We say that two activities  $i$  and  $j$  are non-interfacing if their start-finish intervals do not overlap, more normally,  $[s_p, f_p] \cap [s_j, f_j] = \emptyset$ . (Note that, by making the intervals half open, two consecutive activities are not considered to interface).

The activities scheduling problem is to select a maximum size set of mutually noninterfacing activities for use of the resources. (Notice that here goal is maximum number of activities, not maximum utilization. Of course different criteria could be considered, but the greedy approach may not be optimal in general).

How do we schedule the largest number of activities on the resource ? Intuitively, we do not like long activities, because they occupy the resource and keep us from honoring other requests. This suggests the following greedy strategy : repeatedly select the activity with the smallest duration ( $f_i - s_i$ ) and schedule it, provided that it does not interface with any previously scheduled activities. Although this seems like a reasonable strategy, this turns out to be non-optimal. Sometimes the design of a correct greedy algorithm requires trying a different strategies, until hitting on one that works.

Here is a greedy strategy that does work. The intuition is the same. Since we do not like activities that take a long time, let us select the activity that finishes first and schedule it. Then, we skip all activities that interface with this one, and schedule the next one that has the earliest finish time and so on. To make the selection process faster, we assume that the activities have sorted by their finish times, that is  $f_1 \leq f_2 \leq f_3 \leq \dots \leq f_n$ .

Assuming this sorting, the pseudocode for the algorithm is given below.

### 13.2.1 Iterative Greedy Algorithm for Activity Scheduling

**Step 1:** First sort the input activities by the sorted order like  $f_1 \leq f_2 \leq f_3 \leq \dots \leq f_n$ .

**Step 2:** Then call GREEDY – ACTIVITY – SELECTOR ( $s, f$ )

```

1. n = length [s] // length [s] is the length of given activities
2. A = {A1}
3. j = 1
4. for k = 2 to n
5. do if sk ≥ fj
6. then A = A ∪ {Ak}
7. j = k
8. return A

```

**Running Time:** It is clear that the algorithm is quite simple and efficient. The most costly activity is that of sorting the activities by finishing time, which takes  $\Theta(n \log n)$  time. The GREEDY – ACTIVITY – SELECTOR procedure takes  $\Theta(n)$  running time.

### 13.2.2 Recursive Greedy Algorithm for Activity Scheduling

**Step 1:** First sort the input activities by the sorted order like  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**Step 2:** The call RECURSIVE – ACTIVITY – SELECTOR ( $s, f, k, n$ )

1.  $m = k + 1$
2. while  $m \leq n$  and  $s[m] < f[k]$  // find the first activity in  $s_k$  to finish.
3.  $m = m + 1$
4. if  $m \leq n$
5. return  $\{a_m\} \cup$  RECURSIVE – ACTIVITY – SELECTOR ( $s, f, m, n$ )
6. else return  $\emptyset$ .

**Running Time:** The step-1 of the algorithm takes  $\Theta(n \log n)$  for sorting, step-2 of the algorithm takes  $\Theta(n)$  running time.

**Example 13.1** Given 10 activities along with their start ( $s_i$ ) and finish ( $f_i$ ) time as follows

$s$	$\langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10} \rangle$
$s_i$	$\langle 1, 2, 3, 4, 7, 8, 9, 9, 11, 12 \rangle$
$f_i$	$\langle 3, 5, 4, 7, 10, 9, 11, 13, 12, 14 \rangle$

Compute a schedule where the largest number of activities takes place.

**Solution:** The 10 activities are:

$$\begin{aligned} A_1 &= (1, 3), A_2 = (2, 5), A_3 = (3, 4), A_4 = (4, 7), A_5 = (7, 10) \\ A_6 &= (8, 9), A_7 = (9, 11), A_8 = (9, 13), A_9 = (11, 12), A_{10} = (12, 14). \end{aligned}$$

Arranging the 10 activities according to the increasing order of finishing time stamp we get:

$$\begin{array}{cccccccccc} (1, 3), & (3, 4), & (2, 5), & (4, 7), & (8, 9), & (7, 10), & (9, 11), & (11, 12), & (9, 13), & (12, 14) \\ \downarrow & \downarrow \\ A_1 & A_3 & A_2 & A_4 & A_6 & A_5 & A_7 & A_9 & A_8 & A_{10} \end{array}$$

Applying GREEDY – ACTIVITY – SELECTION algorithm (iterative version)

$$A = \{A_1\} \text{ (Initialisation of line 2)}$$

$$A = \{A_1, A_3\} \text{ (According to line 6, 1st iteration of for loop)}$$

$$A = \{A_1, A_3, A_4\} \text{ (According to line 6, 2nd iteration of for loop)}$$

$$A = \{A_1, A_3, A_4, A_5\} \text{ (According to line 6, 3rd iteration of for loop)}$$

$$A = \{A_1, A_3, A_4, A_5, A_9\} \text{ (According to line 6, 4th iteration of for loop)}$$

$$A = \{A_1, A_3, A_4, A_5, A_9, A_{10}\} \text{ (According to line 6, 5th iteration of for loop)}$$

$$\text{So, } \{A_1, A_3, A_4, A_5, A_9, A_{10}\} = \{(1, 3), (3, 4), (4, 7), (7, 10), (11, 12), (12, 14)\}$$

**Example 13.2** There are 11 activities given i.e.,  $S = \{p, q, r, s, t, u, v, w, x, y, z\}$  and their start and finish times are given by  $(1, 4), (3, 5), (0, 6), (5, 7), (3, 8), (5, 9), (6, 10), (8, 11), (8, 12), (2, 13)$  and  $(12, 14)$  respectively. Compare a schedule where largest number of activities take place.

**Solution:** Given:  $S = \{p, q, r, s, t, u, v, w, x, y, z\}$

$$\begin{aligned} p &= (1, 4), q = (3, 5), r = (0, 6), s = (5, 7), t = (3, 8), u = (5, 9), v = (6, 10) \\ w &= (8, 11), x = (8, 12), y = (2, 13), z = (12, 14). \end{aligned}$$

Arrange the 11 activities according to the increasing order of finishing time stamp.

$$\begin{array}{cccccccccccc} (1, 4), & (3, 5), & (0, 6), & (5, 7), & (3, 8), & (5, 9), & (6, 10), & (8, 11), & (8, 12), & (2, 13), & (12, 14) \\ \downarrow & \downarrow \\ p & q & r & s & t & u & v & w & x & y & z \end{array}$$

Applying greedy algorithm for selection problem

$$\begin{aligned} A &= \{p\} && \text{(Initialisation at line 2)} \\ A &= \{p, s\} && \text{(According to line 6, 1st iteration of for loop)} \\ A &= \{p, s, w\} && \text{(According to line 6, 2nd iteration of for loop)} \\ A &= \{p, s, w, z\} && \text{(According to line 6, 3rd iteration of for loop)} \end{aligned}$$

$$\{p, s, w, z\} = \{(1, 4), (5, 7), (8, 11), (12, 14)\}$$

### 13.3 ELEMENTS OF GREEDY STRATEGY

---

**(i) How to develop a greedy algorithm?**

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Show that if we make the greedy choice, then only one subproblem remains.
- Prove that it is always safe to make greedy choice.
- Develop a recursive algorithm that implements greedy strategy and then convert it to iterative one.

**(ii) Greedy - choice property**

- In greedy algorithm when we are considering which choice to make, we make the choice that looks best in the current problem, without considering the results from subproblems.
- Greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each given problem to a smaller one.

**(iii) Optimal Substructure**

- A problem exhibits optimal substructure if an optimal solution to the problem contains within its optimal solutions to subproblems. This property is the key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

## 13.4 KNAPSACK PROBLEM (RUCKSACK PROBLEM)

---

Knapsack problem is the problem where we want to find an optimal object from a finite set of objects. Here, we have added the concept of “thief robbing”. The thief with a knapsack (bag) with some capacity and the knapsack can not withstand the weight more than the capacity. So it is the choice of the thief to select which item he/she will put in knapsack to fit the capacity. So various situation arises selecting the item: thief may end with the knapsack filled with costly item or chip item or the knapsack may have some space vacant. We have to solve this problem greedily. Mainly these problems are of two types:

- (i) 0 – 1 knapsack problem
- (ii) Fractional knapsack problem

### 13.4.1 0 – 1 Knapsack Problem

The classical 0 – 1 knapsack problem is a famous optimization problem. A thief is robbing a store, and finds  $n$  items which can be taken. The  $i$ th item is worth  $v_i$  dollars and weights  $w_i$  pounds, where  $v_i$  and  $w_i$  are integers. He wants to take as valuable a load as possible, but has a knapsack that can only carry  $W$  total pounds. Which items should he take? (The reason that this is called 0 – 1 knapsack is that each item must be left (0) or taken entirely (1). It is not possible to take a fraction of an item or multiple copies of an item). This optimization problem arises in industrial packing applications. For example, 0 – 1 knapsack problem is applicable if you want to ship some subset of items on a truck of limited capacity.

**Note:** The 0 – 1 knapsack problem is hard to solve, and in fact it is an NP – COMPLETE problem (meaning that there probably doesn't exist an efficient solution). However there is a simple and efficient greedy algorithm for the fractional knapsack problem.

### 13.4.2 Fractional Knapsack Problem

In contrast, the fractional knapsack problem the setup is exactly the same as that of 0 – 1 knapsack problem, but the thief is allowed to take any fraction of an item for a fraction of the weight and a fraction of the value. So, you might think of each object as being a sack of gold, which you can partially empty out before taking.

As in the case of other greedy algorithms we have seen, the idea is to find the right order in which process items. Intuitively, it is good to have high value and bad to have high weight. This suggests that we first sort the items according to some function that decreases with value and increases with weight. There are a few choices that you might try here, but only one works. Let  $\rho_i = v_i/w_i$  denote value-per-pound ratio. We sort the items in decreasing order of  $\rho_i$ , and add them in this order. If the item fits, we take it all. At some points there is an item that does not fit in the remaining space. We take as much of this item as possible, thus filling the knapsack entirely. This is illustrated through an example shown in Fig. 13.1.

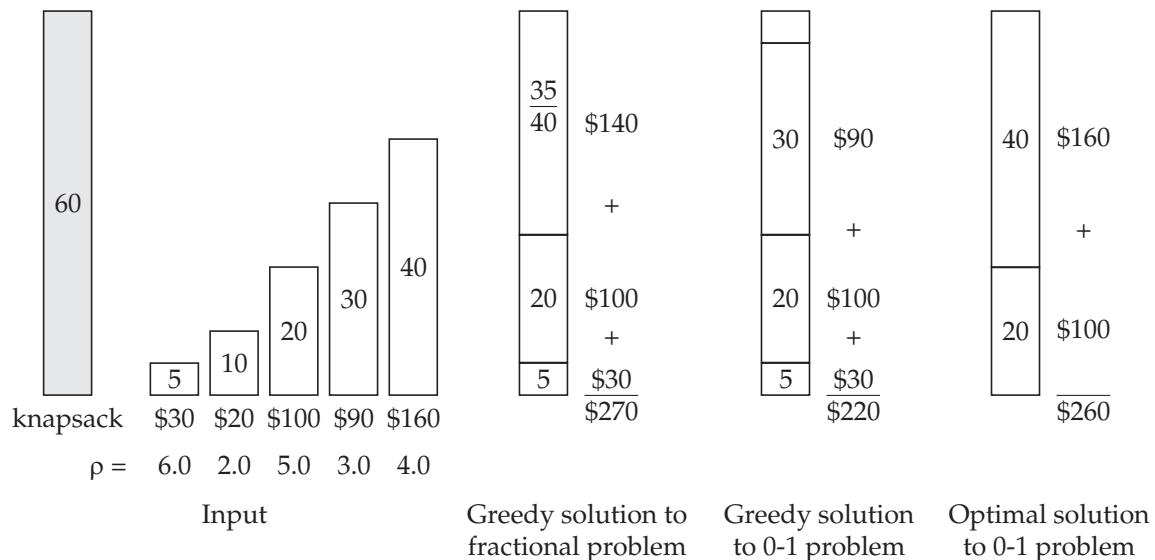


Fig. 13.1. Example of fractional knapsack problem.

**Correctness of Fractional Knapsack Problem:** It is intuitively easy to see that greedy algorithm is optimal for the fractional problem. Given a room with sacks of gold, silver, and bronze, you would obviously take as much gold as possible, then take as much silver as possible, and then as much bronze as possible. But it would never benefit you to take a little less gold so that you could replace it with an equal volume of bronze.

More formally, suppose to the contrary that the greedy algorithm is not optimal. This would mean that there is an alternative selection that is optimal. Sort the items of the alternate selection in decreasing order by  $\rho$  values. Consider the first item  $i$  on which the two selections differ. By definition, greedy takes a greater amount of item  $i$  than the alternate. Let us say that greedy takes  $x$  more units of object  $i$  than the alternate does. All the subsequent element of the alternate selection are of lesser value than  $v_i$ . By replacing  $x$  units of any such items with  $x$  units of item  $i$ , we could increase the overall value of the alternate selection. However this implies that the alternate selection is not optimal, a contradiction.

**Non-Optimality for 0 – 1 Knapsack:** Next we show that the greedy algorithm is not generally optimal in the 0–1 knapsack problem. Consider the example shown in Fig. 13.1. If you were to sort the items by  $\rho_p$ , then you would first take the items of weight 5, then 20, and then (since the item of weight 40 doesn't fit) you would settle for the item of weight 30, for a total value of  $\$30 + \$100 + \$90 = \$220$ . On the other hand if you had been less greedy, and ignored the item of weight 5, then you could take the items of weights 20 and 40 for a total value of  $\$100 + \$160 = \$260$ . This feature of “delaying gratification” in order to come up with a better overall solution which indicates that the greedy solution is not optimal.

### 13.5 HUFFMAN CODING

---

Huffman codes provide a method of encoding data efficiently. Normally when characters are encoded using standard codes like ASCII, each character is represented by a fixed-length codeword of bits (e.g., 8 bits per character). Fixed length codes are popular, because it is very easy to break a string up into its individual characters, and to access individual characters and substrings by direct indexing. However, fixed length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

**Example 13.3** Suppose that we want to encode strings over the (rather limited) 4-character alphabet  $C = \{a, b, c, d\}$ . We could use the following fixed-length code:

Character	a	b	c	d
Fixed-length code word	00	01	10	11

A string such as “abacdaacac” would be encoded by replacing each of its characters by the corresponding binary codeword.

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	11

The final 20 - character binary string would be “00010010110000100010.” Now, suppose that you know the relative probabilities of characters in advance. (This might happen by analyzing many strings over a long period of time. In applications like data compression, where you want to encode one file, you can just scan the file and determine the exact frequencies of all characters). You can use this knowledge to encode strings differently. Frequently occurring characters are encoded using fewer bits and less frequent characters are encoded using more bits. Suppose that characters are expected to occur with the following probabilities. We could design a variable-length code which would do a better job.

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-length codeword	0	110	10	110

Notice that there is no requirement that the alphabetical order of character correspond to any sort ordering applied to the codewords. Now, the same string would be encoded as follows.

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

Thus the resulting 17 characters string would be “01100101110010010”. Thus, we have achieved a saving of 3 characters, by using this alternative code. More generally, what would be the expected saving for a string of length  $n$ ? For the 2nd fixed-length code, the length of the encoded string is just  $2n$  bits, for the variable-length code, the expected length of a single encoded character is equal to the sum of code lengths times the respective probabilities of their occurrences. The expected encoded string length is just  $n$  times the expected encoded character length.

$$n((0.60 \times 1) + (0.05 \times 3) + (0.30 \times 2) + (0.05 \times 3)) = n(0.60 + 0.15 + 0.60 + 0.15) = 1.5n$$

Thus, this would represent a 25% savings in expected encoding length. The question that we will consider now is how to form the best code, assuming that the probabilities of character occurrences are known.

### 13.5.1 Prefix Codes

One issue that we didn't consider in the above example is whether we will be able to decode the string, once encoded. In fact, this code was chosen quite carefully. Suppose that instead of coding the character 'a' as 0, we had encoded it as 1. Now the encoding string "111" is ambiguous. It might be "d" and it might be "aaa". How can we avoid this sort of ambiguity? You might suggest that we add separation markers between the encoded characters, but this will tend to lengthen the encoding, which is undesirable. Instead, we would like the code to have the property that it can be uniquely decoded.

Note that in both the variable length codes given in the example above no codeword is a prefix of another. This turns out to be the key property. Observe that if two codewords did share a common prefix, e.g.,  $a \rightarrow 011$  and  $b \rightarrow 00101$ , then when we see 00101 ... how do we know whether the first character of the encoded message is  $a$  or  $b$ . Conversely, if no codeword is a prefix of any other, then as soon as we see a codeword appearing as a prefix in the encoded text, then we know that we may decode this without fear of it matching some longer codeword. Thus we have the following definition for prefix code.

**Definition 13.1 (Prefix code):** Prefix code is an assignment of codewords to characters so that no codeword is a prefix of other.

Observe that any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means "0" and a right branch means "1". The length of the codeword is just its depth in the tree. The code given earlier is a prefix code, and its corresponding tree is shown in Fig. 13.2.

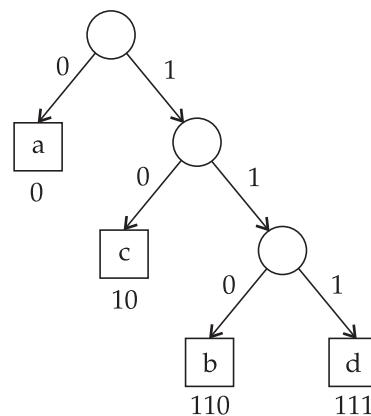


Fig. 13.2. Prefix codes.

Decoding a prefix code is simple. We just traverse a tree from root to leaf, lettering the input character tell us which branch to take. On reaching a leaf, we output the corresponding character, and return to the root to continue the process.

### 13.5.2 Expected Encoding Length

Once we know the probabilities of the various characters, we can determine the total length of the encoded text. Let  $p(x)$  denote the probability of setting character  $x$  and let  $d_T(x)$  denote the length of the codeword (depth in the tree) relative to some prefix tree  $T$ . The expected number of bits needed to encode a text with  $n$  characters is given in the following formula:

$$B(T) = n \sum_{x \in C} p(x) d_T(x)$$

This suggests the following problem:

**Optimal Code Generation:** Given an alphabet  $C$  and the probabilities  $p(x)$  of occurrence for each character  $x \in C$ , compute a prefix code  $T$  that minimizes the expected length of the encoded bit-string,  $B(T)$ .

### 13.5.3 Huffman's Algorithm

Recall that we are given the occurrence probabilities for the characters. We are going to build, the tree up from the leaf level. We will take two characters  $x$  and  $y$ , and “merge” them into single super-character called  $Z$ , which then replaces  $x$  and  $y$  in the alphabet. The character  $Z$  will have a probability equal to the sum of  $x$  and  $y$ 's probabilities. Then we continue recursively building the code on the new alphabet, which has one fewer character. When the process is completed, we know the code for  $Z$ , say 010. Then we append a 0 and 1 to this codeword, given 0100 for  $x$  and 0101 for  $y$ .

#### Algorithm Steps

```
HUFFMAN (int n, character [1 ..... n])
1. Q = C // C denote the set of characters [1 ..... n]
2. for i = 1 to n - 1
3. Z = new internal tree node
4. Z - left = x = Q. extract Min ( )
           // extract smallest probabilities
5. Z - right = y = Q. extract Min ( )
6. Z - prob = x. prob + y. prob // Z's probability is their sum
7. Q - insert (Z)           // insert Z into queue
8. return the last element left in Q as the root.
```

**Running Time:** In the Huffman's pseudocode algorithm each character  $x \in C$  is associated with an occurrence probability  $x. prob$ . Initially, the characters are all stored in a priority queue  $Q$ . Recall that this data structure can be built initially in  $O(n)$  time, and we can

extract the element with the smallest key in  $O(n \log n)$  time and insert a new element in  $O(n \log n)$  time. After  $n - 1$  iterations, there is exactly one element left in queue, and this is the root of final prefix code tree.

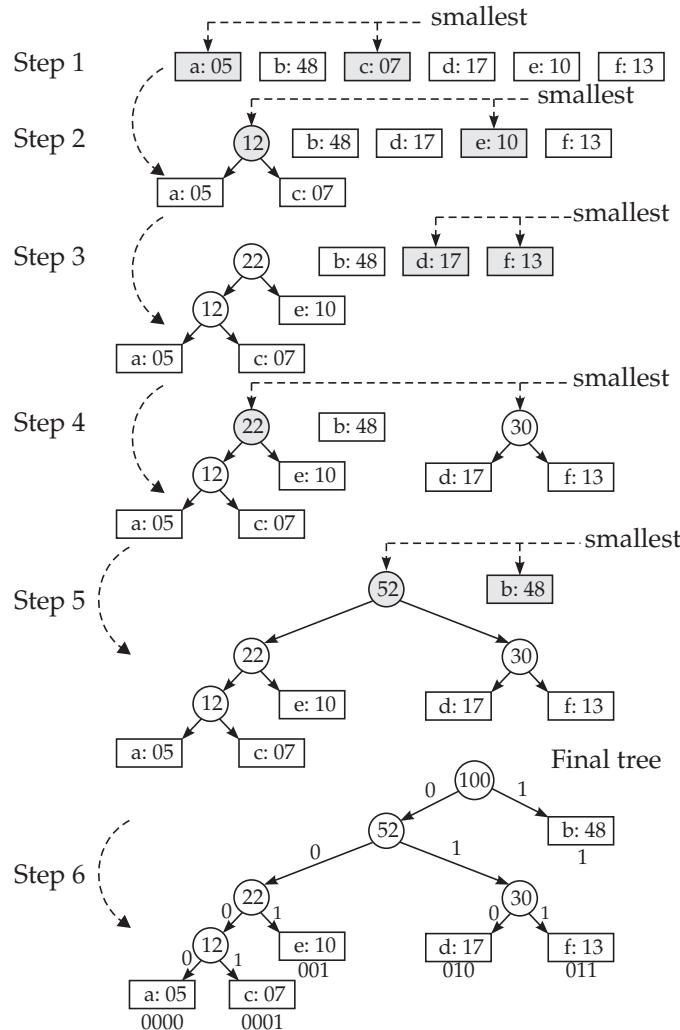
Running time of Huffman algorithm is  $O(n \log n)$ .

#### 13.5.4 Disadvantages of Huffman Coding

- If the frequencies and probabilities changes then the optimal coding changes.
- It does not consider ‘blocks of symbols’ like ‘strings-of-ch’ the next nine symbols are predictable ‘aracters’, but bits are used without conveying any new information.

**Example 13.4** Apply Huffman algorithm to  $a : 05$ ,  $b : 48$ ,  $c : 07$ ,  $d : 17$ ,  $e : 10$ ,  $f : 13$ . Find its time complexity.

**Solution:**



Now the codes for the character are shown below in the table.

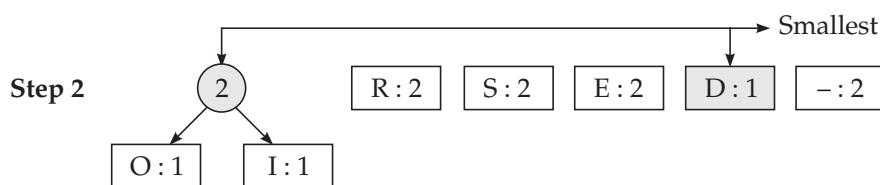
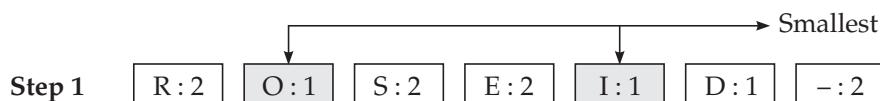
Character	Code
a	0000
b	1
c	0001
d	010
e	001
f	011

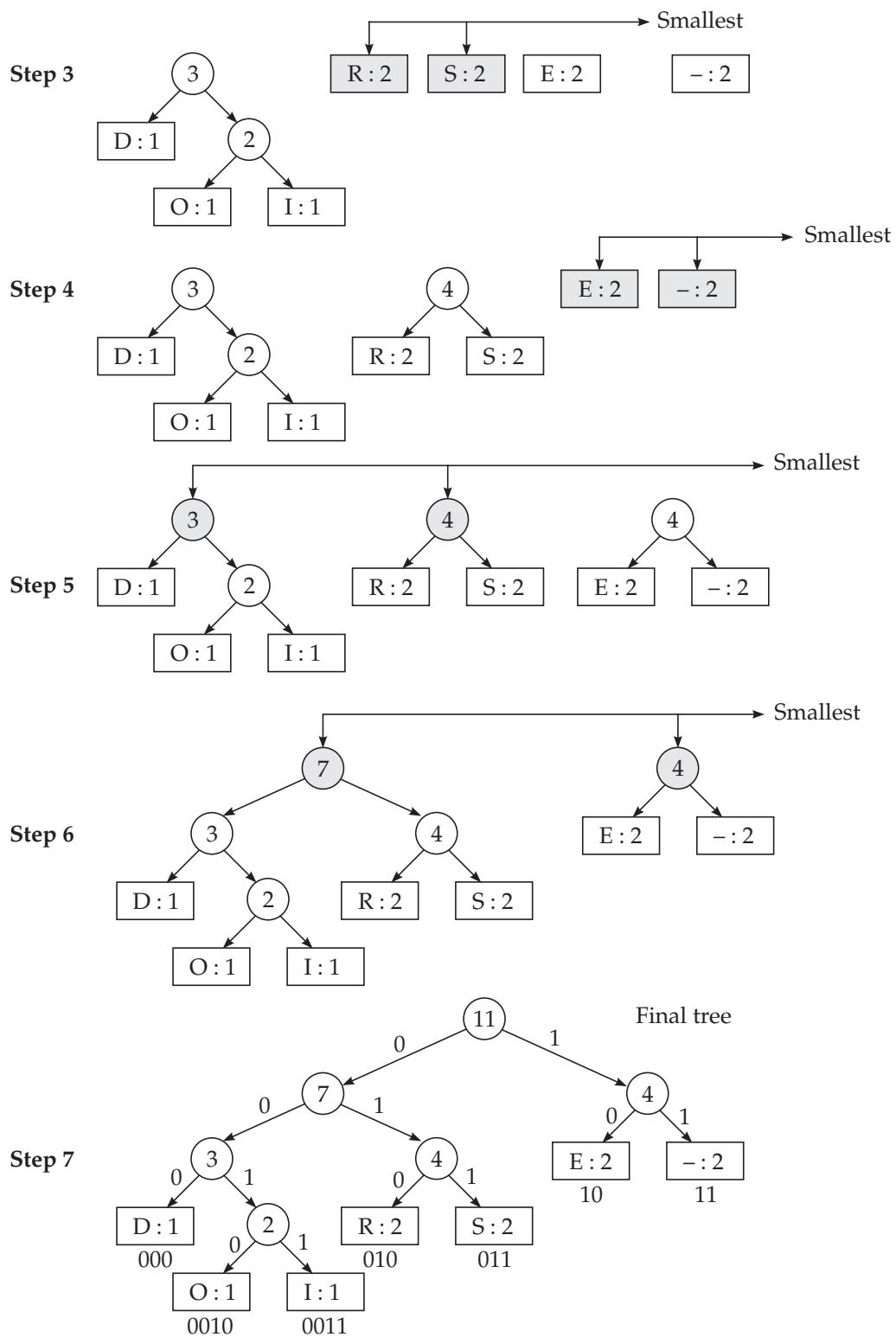
Time complexity  $T(n) = O(n \log n)$

**Example 13.5** Huffman code of the input string “ROSE IS RED” considering space as character. Find the generated string.

**Solution:** The given string is “ROSE IS RED”. Here space is a character.

Character	Frequency
R	2
O	1
S	2
E	2
I	1
D	1
-	2
(space)	





Codes for the characters are:

Character	Code
R	010
O	0010
S	011
E	10
I	0011
D	000
-	11
(space)	

**Example 13.6** Find the Huffman code for the given characters (in ASCII) and their frequencies:

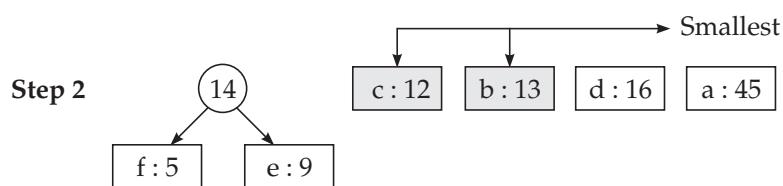
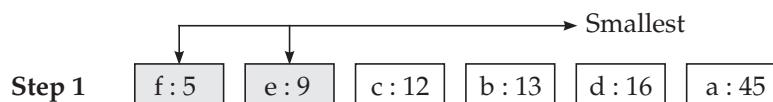
$f : 5, e : 9, c : 12, b : 13, d : 16, a : 45$ . After finding the Huffman code; calculate the percentage of bit reduced.

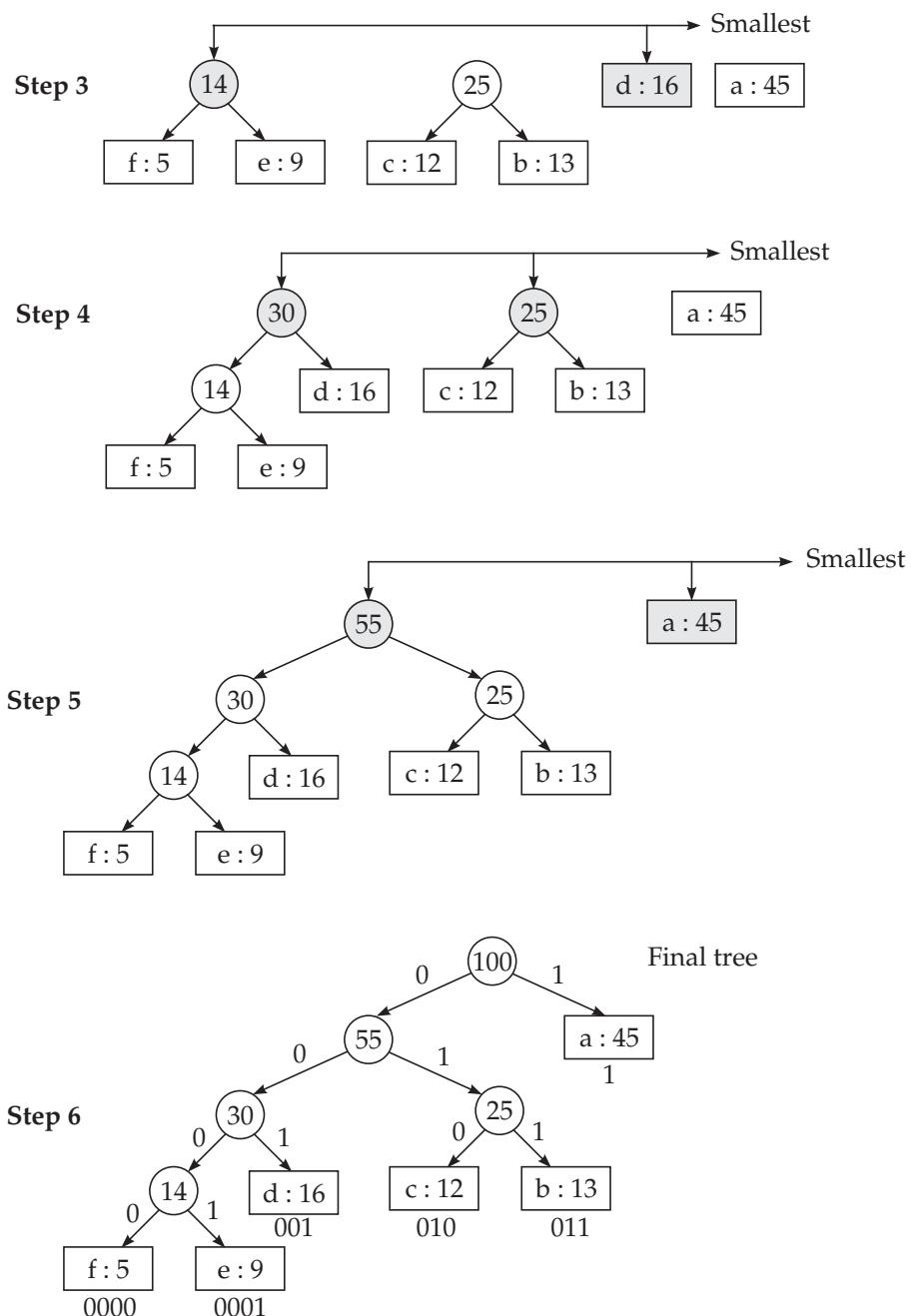
**Note:** Each ASCII character is 8 bits long.

**Solution:**

ASCII Character	Frequency	No. of Bits
f	5	$5 \times 8 = 40$
e	9	$9 \times 8 = 72$
c	12	$12 \times 8 = 96$
b	13	$13 \times 8 = 104$
d	16	$16 \times 8 = 128$
a	45	$45 \times 8 = 360$
		Total = 800

Now performing the Huffman coding for the given characters:





Code for each character and the length of each code is:

<b>Character</b>	<b>Code</b>	<b>Length of code</b>
f	0000	4
e	0001	4
c	010	3
b	011	3
d	001	3
a	1	1

After getting the Huffman code for each character on the above table now calculate the number of bits required (= length of the character  $\times$  frequency) to represent it.

<b>Charater (In Huffman Code)</b>	<b>No. of Bits</b>
f	$4 \times 5 = 20$
e	$4 \times 9 = 36$
c	$3 \times 12 = 36$
b	$3 \times 13 = 39$
d	$3 \times 16 = 48$
a	$1 \times 45 = 45$
	Total = 224

Now the percentage of bit reduced

$$\begin{aligned}
 &= \frac{\text{Total number of bits in ASCII} - \text{Total number of bits in Huffman code}}{\text{Total number of bits in ASCII}} \times 100 \\
 &= \frac{800 - 224}{800} \times 100 = 72\% \approx 75\%
 \end{aligned}$$

So near about 75% of bits we are reducing in Huffman code.

## CHAPTER NOTES

---

- Greedy strategy: the choice that seems best at the moment is the one we go with. When there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it is always safe to make this choice.
- Dynamic programming can be over kill; greedy algorithms tend to be easier to code.

- Activity selection problem:

Input: set S of  $n$  activities,  $a_1, a_2, \dots, a_n$

$s_i$  = start time of activity  $i$ .

$f_i$  = finish time of activity  $i$ .

Output: Subset A of maximum number of compatible activities. Two activities are compatible, if their intervals don't overlap.

- More formally in the 0 – 1 knapsack problem:

- Thief cannot take a fraction of any item.
- It cannot be solved by using greedy algorithm.
- In the Fractional knapsack problem.
- Thief can take fractions of items.
- It can be solved by greedy algorithm.

- The Huffman coding algorithm produces optimal symbol codes.

## EXERCISES

---

1. Write a recursive greedy algorithm for activity scheduling problem. Find its running time.
2. Explain the elements of greedy strategy.
3. What is Greedy algorithm? What are its properties? Explain with an example the difference between 0/1 knapsack problem and Fractional knapsack problem.
4. What are the steps to design greedy algorithm? Discuss briefly.
5. How does the dynamic programming differs from the greedy approach?
6. Construct the Huffman code for the following data:

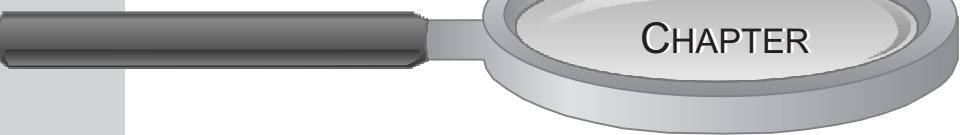
Character	A	B	C	D	-
Probability	0.4	0.1	0.2	0.15	0.15

Encode the text ABACABAD using the Huffman code obtained. Decode the text, whose encoding is 100010111001010 using the above coding scheme.

7. Find the Huffman code of input string “A ROSE FOR MY ROSE”. Consider space as a character. Find the generated string and also compute the time complexity of entire process.
8. Construct the Huffman tree corresponding to the following set of data:

	a	b	c	d	e	f
Frequency (in thousands)	48	12	10	15	8	4
Fixed length code word:	000	001	010	011	100	010
Variable length coding:	0	101	100	111	1101	1100

# ELEMENTARY GRAPH ALGORITHMS



14

CHAPTER

## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- Concepts of graphs and diagraphs, its types and its ways of representation
- Paths and cycles
- Breadth-first-search and Depth-first search
- Topological sort
- Strongly connected components
- Spanning tree
- Minimum spanning tree algorithms
- (Single source) shortest paths
- Relaxation procedure
- Dijkstra's algorithm and Bellman-Ford algorithm
- Floyd-Warshall algorithm

# 14 ELEMENTARY GRAPH ALGORITHMS

## INSIDE THIS CHAPTER

- |                                 |  |
|---------------------------------|--|
| 14.1 Introduction               | 14.2 Representation of Graphs and Digraphs |
| 14.3 Graph Traversal Techniques | 14.4 Breadth-first Search (BFS)            |
| 14.5 Depth-first Search (DFS)   | 14.6 Application of DFS                    |
| 14.7 Minimum Spanning Tree      | 14.8 Kruskal's Algorithm                   |
| 14.9 Prim's Algorithm           | 14.10 Single Source Shortest Paths         |
| 14.11 Dijkstra's Algorithm      | 14.12 The Bellman-Ford Algorithm           |
| 14.13 All-Pairs Shortest Paths  | 14.14 Floyd-Warshall Algorithm             |

## 14.1 INTRODUCTION

We are now beginning a major new section of this book. We will be discussing algorithms for both directed and undirected graphs intuitively a graph is a collection of vertices or nodes, connected by a collection of edges. Graphs are extremely important because they are a very flexible mathematical model for many application problems. Basically any time you have a set of objects, and there is some “connection” or “relationship” or “interaction” between pairs of objects, a graph is a good way to model this. Example of graphs in application includes communication and transportation networks, VLSI and other sorts of logic circuits, surface meshes used for shape description in communication - aided design and geographic information systems, precedence constraints in scheduling system. The list of application is too long to even consider enumerating it.

Most of the problems in computational graph theory that we will consider arise because they are of importance to one or more of these application areas. Furthermore many of these problems form the basic building blocks from which more complex algorithms are then built.

### 14.1.1 Graphs and Diagraphs

Most of you have encountered the notices of directed and undirected graphs in other courses, so we will give a quick overview here.

**Definition 14.1 (Digraph):** A directed graph (or diagraph)  $G = (V, E)$  consists of a finite set  $V$  called the vertices or nodes, and  $E$ , a set of ordered pairs, called the edges of  $G$ .

Observe that self-loops are allowed by this definition. Some definitions of graph disallow this. Multiple edges are not permitted (although the edges  $(v, w)$  and  $(w, v)$  are distinct).

**Example 14.1**

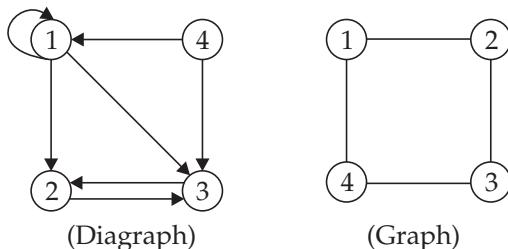


Fig. 14.1. Diagraph and graph.

**Definition 14.2 (Undirected graph):** An undirected graph (or graph)  $G = (V, E)$ , consists of finite set  $V$  of vertices and a set  $E$  of unordered pairs of distinct vertices, called the edges. Here self loops are not allowed.

**Note:** Directed and undirected graphs are different but similar objects mathematically. Certain notions (such as path) are defined for both but other notions (such as connectivity) may only be defined for one or may be defined alternatively.

We may say that vertex  $v$  is adjacent to vertex  $u$  if there is an edge  $(u, v)$ . In a directed graph, given the edge  $e = (u, v)$  we say that  $u$  is the origin of  $e$  and  $v$  is the destination of  $e$ . In undirected graph  $u$  and  $v$  are the endpoints of the edge. The edge  $e$  is incident (meaning that it touches) both  $u$  and  $v$ .

In a digraph, the number of edges coming out of a vertex is called the **out-degree** of that vertex, and the number of edges coming in is called **in-degree**. In an undirected graph we just talk about the degree of a vertex as the number of incident edges. By the degree of a graph, we usually mean the maximum degree of its vertices.

When discussing the size of a graph, we typically consider both the number of vertices and the number of edges. The number of vertices is typically written as  $n$  or  $V$ , and the number of edges is written as  $m$  or  $E$  or  $e$ . Here are some combinatorial facts about graphs and diagraphs. Given a graph with  $V$  vertices and  $E$  edges then:

**In a graph:**

$$\text{Number of edges: } 0 \leq E \leq (2^n) = \frac{n(n-1)}{2} \in O(n^2)$$

$$\text{Sum of degrees: } \sum_{v \in V} \deg(v) = 2E$$

**In a digraph:**

**Number of edges :**  $0 \leq E \leq n^2$

**Sum of degrees :**  $\sum_{v \in V} \text{in-deg}(v) = \sum_{v \in V} \text{out-deg}(v) = E$ .

Notice that generally the number of edges in graph may be as large as quadratic in the number of vertices. However, the large graphs that arise in practice typically have much fewer edges. A graph is said to be **Sparse** if  $E \in O(V)$ , and **dense**, otherwise.

#### 14.1.2 Paths and Cycles

A **path** in a graph or digraph is a sequence of vertices  $\langle v_0, v_1, \dots, v_k \rangle$  such that  $(v_{i-1}, v_i)$  is an edge for  $i = 1, 2, \dots, k$ . The length of the path is the number of edges,  $k$ . A path is simple if all vertices and all the edges are distinct.

A **cycle** is a path containing at least one edge for which  $v_0 = v_k$ . A cycle is simple if its vertices (except  $v_0$  and  $v_k$ ) are distinct, and all the edges are distinct.

A **graph** or **digraph** is said to be acyclic if it contains no simple cycles. An acyclic connected graph is called a **free tree** or simply **tree** for short. An acyclic digraph is called a **directed acyclic graph** or **DAG** for short.

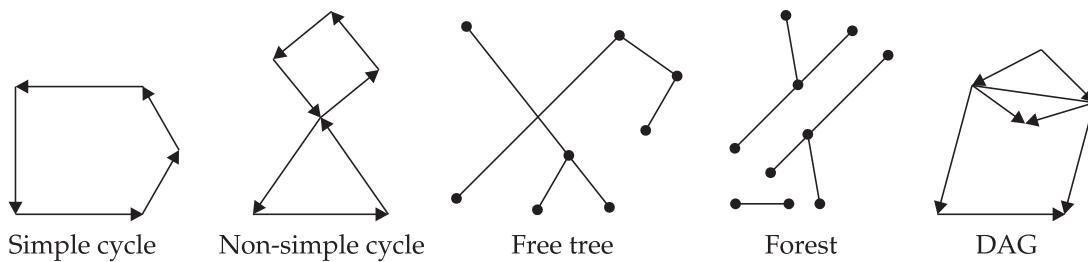


Fig. 14.2. Simple cycle, non-simple cycle, free tree, forest, DAG.

We say that  $w$  is reachable from  $u$  if there is a path from  $u$  to  $w$ .

**Note:** Every vertex is reachable from itself by a trivial path that uses zero edges.

An undirected graph is **connected** if every vertex can reach every other vertex. The subsets of mutually reachable vertices partition the vertices of the graph into disjoint subsets, called the **connected components** of graph.

## 14.2 REPRESENTATION OF GRAPHS AND DIGRAPHS

---

There are two common ways of representing graphs and digraphs. First we show how to represent digraphs. Let  $G = (V, E)$  be a digraph with  $n = |V|$  and let  $|E| = |E|$ . We will assume that the vertices of  $G$  are indexed  $\{1, 2, \dots, n\}$ .

### (i) Adjacency Matrix Representation

An  $n \times n$  matrix defined for  $1 \leq v, w \leq n$ .

$$A[v, w] = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases}$$

If the digraph has weight we can store the weights in the matrix. For example, if  $(v, w) \in E$  then  $A[v, w] = W(v, w)$  the weight on edge  $(v, w)$ . If  $(v, w) \notin E$  then generally  $W(v, w)$  need not be defined, but often we set it to some “special” value, e.g.,  $A(v, w) = -1$ , or  $\infty$ . By  $\infty$  we mean some number which is larger than any allowable weight. In practice, this might be some machine dependent constant like MAXINT.

#### Example 14.2

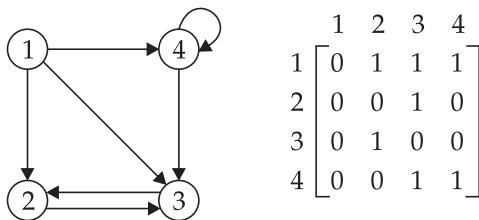


Fig. 14.3. Adjacency matrix for digraph.

#### Example 14.3

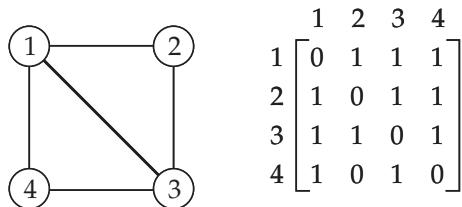


Fig. 14.4. Adjacency matrix for undirected graph.

### (ii) Adjacency List Representation (Linked-list Representation)

An array  $Adj[1 \dots n]$  of pointers where for  $1 \leq v \leq n$ ,  $Adj[v]$  points to a linked list containing the vertices which are adjacent to  $v$  (i.e., the vertices that can be reached from  $v$  by a single edge). If the edges have weights then these weight may also be stored in the linked list elements.

#### Example 14.4

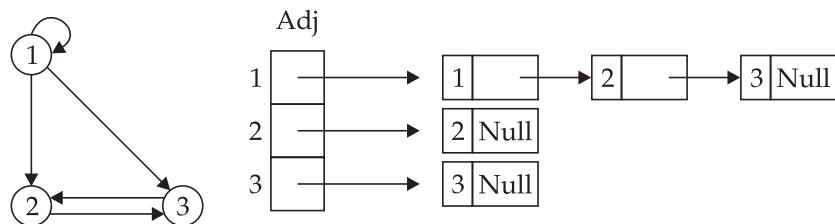


Fig. 14.5. Adjacency list for digraph.

We can represent undirected graphs using exactly same representation, but we will store each edge twice. In particular, we represent the undirected edge  $\{v, w\}$  by the two oppositely directed edges.  $(v, w)$  and  $(w, v)$ .

#### Example 14.5

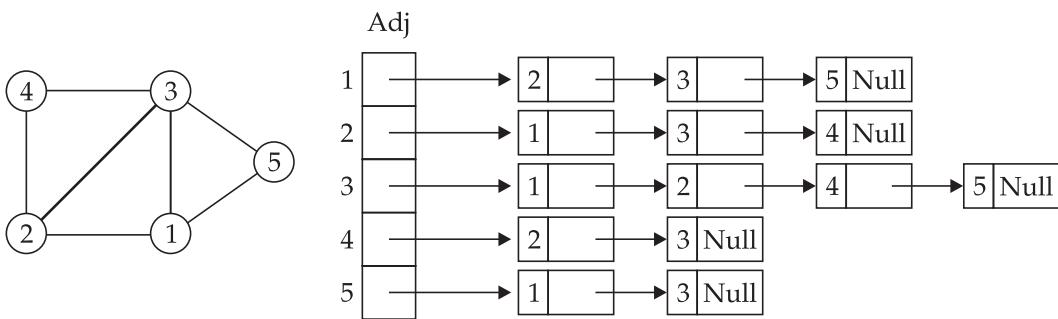


Fig. 14.6. Adjacency list for undirected graph.

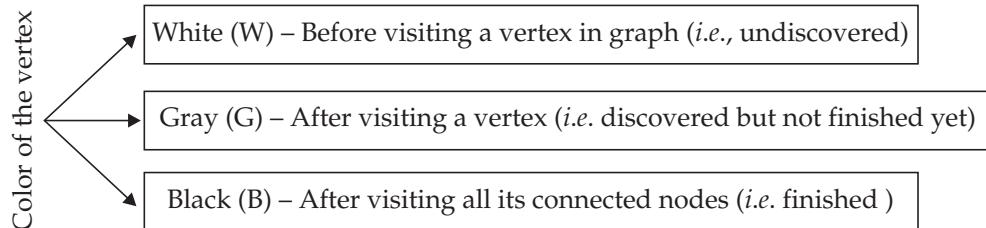
**Note:** An adjacency matrix requires  $\theta(v^2)$  storage and the adjacency list require  $\theta(V + E)$  storage. The V arises because there is one entry for each vertex in Adj. Since each list has out-deg ( $v$ ) entries, when this summed over all vertices, the total number of adjacency list records is  $\theta(E)$ .

### 14.3 GRAPH TRAVERSAL TECHNIQUES

Graph traversal is an important and major part of this chapter. Unlike *tree traversal*, *graph traversal* is a technique through which we visit all vertices in a graph in a particular order, updating and checking their values along the way. The hardest part about traversing a graph is making sure that we don't process some vertices twice. But sometimes we are essentially in need to some nodes to be visited more than once. This is called *redundancy*. Thus, it is usually necessary to remember which nodes have already been explored by the algorithm, so that nodes are revisited as infrequently as possible. For that we impose certain *parameters* while traversing a graph. These parameters are:

- Color of the vertex
- Distance from source vertex
- Predecessor vertex (or parent vertex)

Again the parameter color of the vertex takes three cases into consideration. They are shown below:



We are now going to study two common graph traversal algorithms in the next two sections. They are:

- Breadth-first search (in short called BFS)
- Depth-first search (in short called DFS)

#### 14.4 BREADTH-FIRST SEARCH (BFS)

Given a graph  $G = (V, E)$ , breadth-first search starts at some *source vertex*  $S$  and “*discovers*” which vertices are reachable from  $S$ . Define the *distance* between a vertex  $u$  and  $S$  to be the minimum number of edges on a path from  $S$  to  $u$ . Breadth-first search discovers vertices in *increasing order of distance*, and hence can be used as an algorithm for computing shortest paths. At any given times there is a “frontier” of vertices that have been discovered, but not yet processed. Breadth-first search is named because it visits vertices across the entire “breadth” of this frontier as shown in Fig. 14.7.

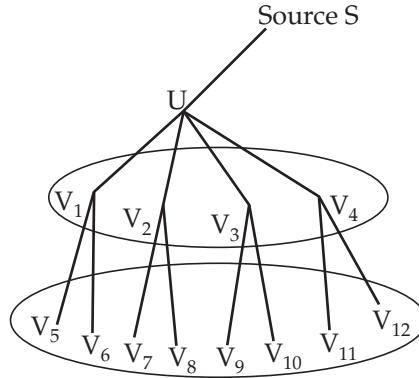


Fig. 14.7. Traversal in BFS.

Initially all vertices (except the source  $S$ ) are colored white, meaning that they are *undiscovered*. When a vertex has first been *discovered*, it is coloured gray (and is part of the frontier). When a gray vertex is *processed*, then it becomes black.

The search makes use of a *queue*, a first-in first-out (FIFO) manner, where elements are removed in the same order they are inserted. The first item in the queue the next to be removed. We will also maintain arrays  $\text{color}[u]$  which holds the color of vertex  $u$  (white, gray or black),  $\pi[u]$  which points to the predecessor of  $u$  (*i.e.*, the vertex who first discovers  $u$ ) and  $d[u]$  represents the distance from  $s$  to  $u$ . Only the color is really needed for the search (in fact it is only necessary to know whether a node is nonwhite). Including all the above informations we are now going to write the algorithm for BFS traversal.

##### Algorithm For BFS ( $G, S$ )

```
BFS ( $G, S$ ) //  $G \rightarrow$  graphs,  $S \rightarrow$  Source vertex
1. For all vertex  $u \in V [G]$ 
2. do colour  $[u] = \text{White}$ ,  $d[u] = \infty$ ,  $\pi[u] = \text{Null}$  // Initialization
3. colour  $[S] = \text{Gray}$ ,  $d[S] = 0$  // Initialize source  $S$ 
4. ENQUEUE ( $Q, S$ ) // ENQUEUE is a function to insert elements into an empty queue ( $Q$ ).
5. While ( $Q$  is not empty)
6. do  $u = \text{DEQUEUE} (Q)$  // DEQUEUE is a function to delete elements from queue.
7. For all  $V \in \text{adjacent} (u)$ 
8. do if colour  $[V] = \text{White}$  // if neighbour  $V$  undiscovered
9. then colour  $[V]$  Gray,  $d[V] = d [u] + 1$ ,  $\pi[V] = u$ 
```

10. ENQUEUE (Q, V)
11. colour [V] = Black.

Observe that the predecessor pointers of the BFS search define an *inverted tree* (an acyclic directed graph in which the source is the root, and every other node has a unique path to the root). If we reverse these edges we get a rooted unordered tree called a *BFS tree* for G. (Note that there are many potential BFS trees for a given graph, depending on where the search starts, and in what order vertices are placed on the queue.) These edges of G are called *tree edges* and the remaining edges of G are called *cross edges*.

### Running Time of BFS Algorithm

- Algorithm steps 1, 2, 3, runs for all the vertex present in the graph i.e., total no of vertex present in the given graph =  $O(|V|)$
- Step 4 run =  $O(1)$  for each case and in total  $V$  times.
- So, total =  $O(V)$ . Similarly for step 6. So ENQUEUE and DEQUEUE function runs  $O(V)$  times.
- Step 7 runs max of  $(V - 1)$  times.

So step 7, 8, 9, 10 and 11 runs in total  $O(V + E)$  times.

Hence total running time of the algorithm is

$$\begin{aligned}
 &= O(\max(f_1(n), f_2(n), \dots, f_n(n))) \\
 &= O(\max(O(|V|), O(V), O(V + E))) \\
 &= O(V + E)
 \end{aligned}$$

**Example 14.6** Perform BFS traversal on the graph G given in Fig. 14.8 taking 'a' as the starting node and find BFS tree and BFS branches.

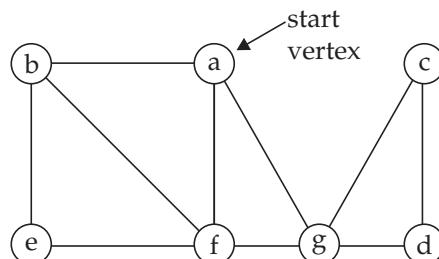


Fig. 14.8. Graph G of Example 14.6.

**Solution:** Adjacency list of the given graph G

- Adjacent of  $a - b, f, g$
- Adjacent of  $b - a, e, f$
- Adjacent of  $c - d, g$
- Adjacent of  $d - c, g$
- Adjacent of  $e - b, f$

Adjacent of  $f$  –  $a, e, g, b$

Adjacent of  $g$  –  $a, c, d, f$

Now performing the operation BFS (G, S):

**Step 1.** Make colour of all the vertex in the graph G as white (W), distance ( $d$ ) of the vertex set to infinity ( $\infty$ ) and predecessor of all the vertex = NULL.

So colour  $[u] = \text{white}$ ,  $d[u] = \infty$  and  $\pi[u] = \text{NULL}$ .

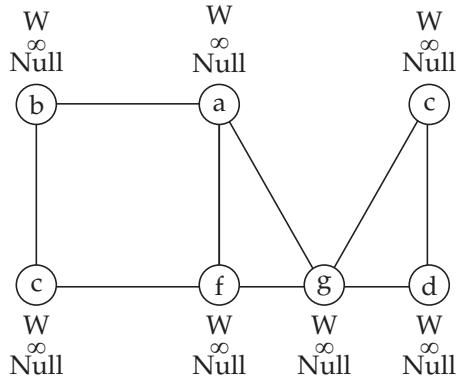


Fig.14.9. Result of step-1.

**Step 2.** Now make colour of the starting vertex as Gray. Enter it in the queue. Change distance of  $a$  from  $\infty$  to 0. So colour  $[a] = \text{Gray}$ ,  $d[a] = 0$ .

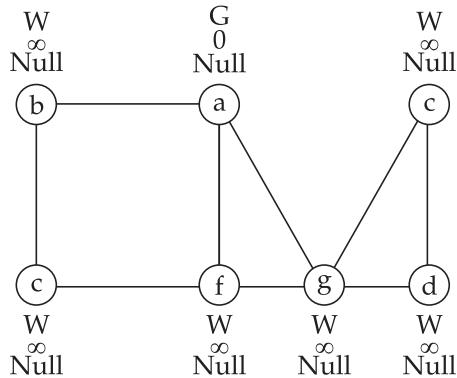


Fig. 14.10. Result of step-2.

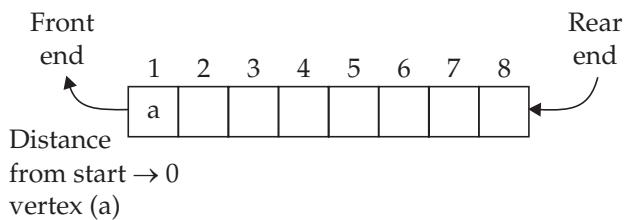


Fig. 14.11. Queue for step-2.

**Step 3.** Delete a from queue. Visit the adjacent vertex of a i.e., b, f, g and enter them into the queue. Make b, f, g from white to gray. Make a as the predecessor of b, f, g. Now a is a black.

1	2	3	4	5	6	7	8
	b	f	g				

Distance  
from start →  
vertex (a)

Fig. 14.12. Queue for step-3.

So, colour [b] = Gray

colour [f] = Gray

colour [g] = Gray

$d[b] = 1, d[f] = 1, d[g] = 1$

$\rho[b] = a, \pi[f] = a, [g] = a$

colour [a] = black.

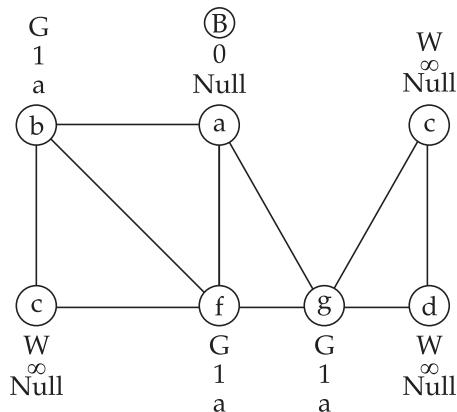


Fig. 14.13. Result of step-3.

**Step 4.** Now delete b from the queue. Enter its adjacent which is white in colour i.e., e. Change the colour of e to Gray. Predecessor of e is b. Make b black.

1	2	3	4	5	6	7	8
		f	g	e			

Distance  
from start →  
vertex (a)

Fig. 14.14 Queue for step-4

So, colour  $[e] = \text{Gray}$ ,  $d[e] = 2$ ,  $\pi[e] = b$ , colour  $[b] = \text{black}$

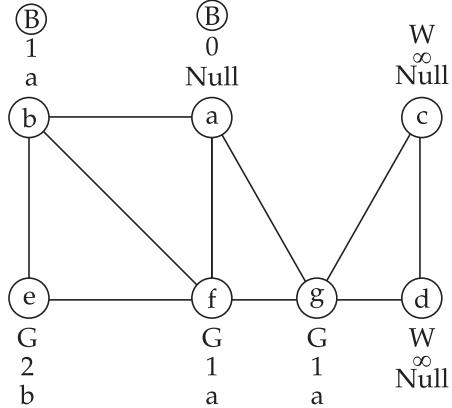


Fig. 14.15. Result of step-4.

**Step 5.** Now delete F from queue. But there is no white node present which are adjacent to f. So simply make f as black.

	1	2	3	4	5	6	7	8
Distance from start → vertex (a)				g	e			

1      2      3      4      5      6      7      8

Fig. 14.16. Queue for step-5.

So, colour  $[f] = \text{black}$ .

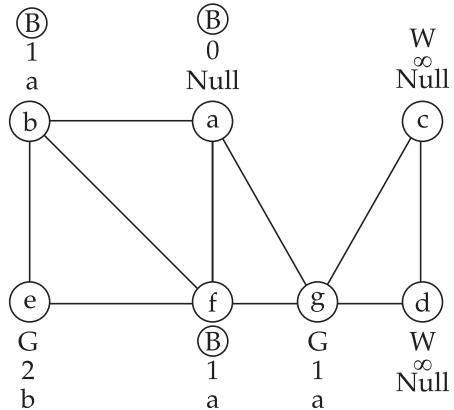


Fig. 14.17. Result of step-5.

**Step 6.** Now delete g from queue. Visit all its adjacent nodes which are white in color i.e., c and d. Make c, d as gray. Make g as black.

	1	2	3	4	5	6	7	8
Distance from start → vertex (a)					e	c	d	

1      2      3      4      5      6      7      8

Fig. 14.18. Queue for step-6.

So, colour  $[c] = \text{Gray}$      $d[c] = 2$      $\pi[c] = g$

colour  $[d] = \text{Gray}$      $d[d] = 2\pi[d] = g$

colour  $[g] = \text{black}$

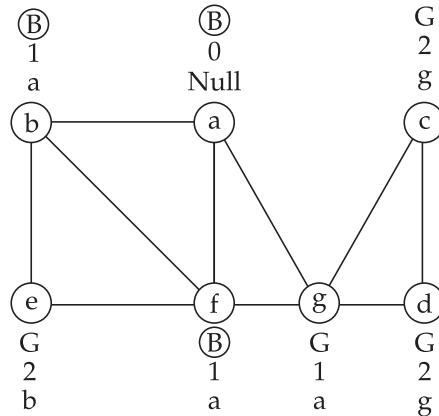


Fig. 14.19. Result for step-6.

**Step 7.** Delete  $e$  from queue. No white node found which is adjacent to  $e$  and white in colour. So make  $e$  as black.

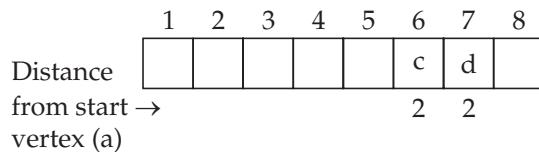


Fig. 14.20. Queue for step-7.

Color  $[e] = \text{black}$

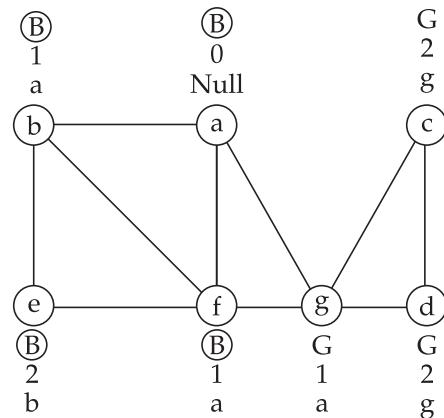


Fig. 14.21. Result of step-7.

**Step 8.** Delete  $c$  from queue. No white node found which is adjacent to  $c$ . Make  $c$  black.

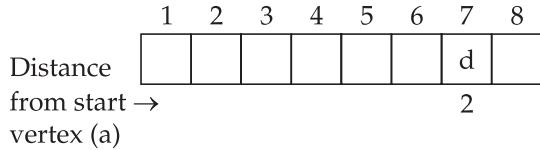


Fig. 14.22. Queue for step-8.

Color  $[c] = \text{black}$

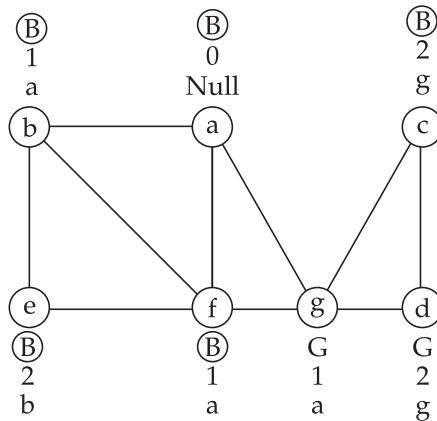


Fig. 14.23. Result of step-8.

**Step 9.** Delete  $d$  from queue. No white node is present which is adjacent to  $d$ . Make  $d$  black.

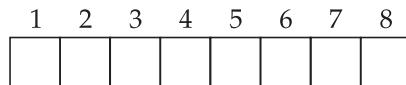


Fig. 14.24. Empty queue.

Color  $[d] = \text{black}$

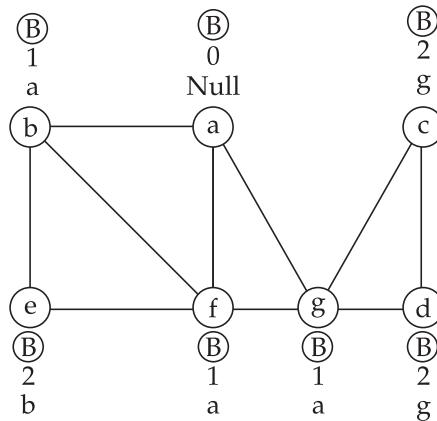


Fig. 14.25. Result of step-9.

Now queue is empty.

**BFS Tree:** Now the BFS tree for the graph G given in Fig. 14.8 is shown in Fig. 14.26.

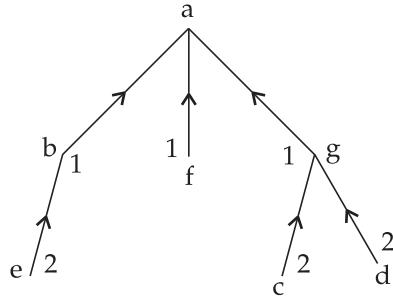


Fig. 14.26. BFS trees for Example 14.26.

**BFS branches :**  $a - b$ ,  $a - f$ ,  $a - g$ ,  $b - e$ ,  $g - c$ ,  $g - d$ .

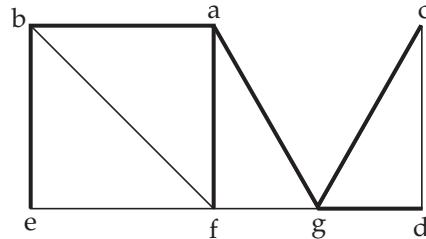


Fig. 14.27. BFS branches.

**Example 14. 7** Do the BFS traversal on the given graph given in Fig. 14.8, taking 'e' as the starting node and find the BFS tree and BFS branches of the given graph.

**Solution.** Task for the reader to work out the entire BFS traversal by your own for practice. The resultant BFS tree will be like as shown in Fig. 14. 28.

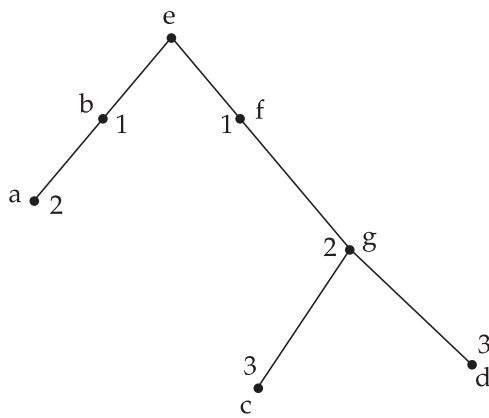


Fig. 14.28. BFS tree.

**BFS Branches :**  $e - b, e - f, b - a, f - g, g - c, g - d.$

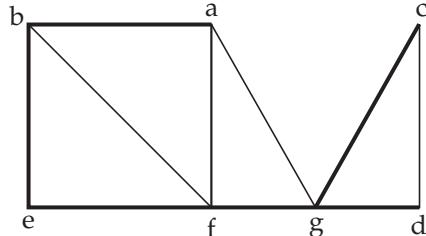


Fig. 14.29. BFS branches.

**Example 14.8** Perform the BFS traversal on the graph given in Fig. 14.30 taking 3 as the starting node. Also find the BFS tree and BFS branches.

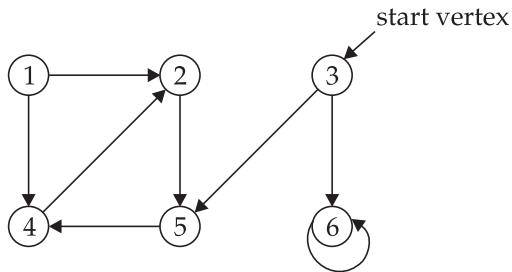


Fig. 14.30. Graph G of Example 14.30.

#### Solution. Adjacency list of the given graph G

Adjacent of 1 – 2, 4

Adjacent of 2 – 5

Adjacent of 3 – 5, 6

Adjacent of 4 – 2

Adjacent of 5 – 4

Adjacent of 6 – 6

Now calling the procedure BFS (G, S) :

**Step 1.** Make colour of all the vertex in graph G as white ( $w$ ), distance ( $d$ ) of the vertex set to infinity ( $\infty$ ) and predecessor of all the vertex = NULL (as shown in Fig .14.31).

So, colour  $[u] = \text{white}$ ,

$$d[u] = \infty \text{ and}$$

$$\pi[u] = \text{NULL}.$$

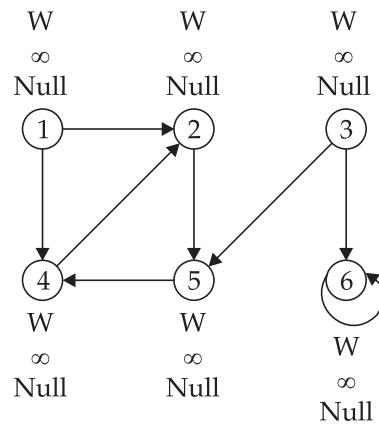


Fig. 14.31. Result of step-1.

**Step 2.** Now make colour of the starting vertex as Gray. Enter it in the queue. Change distance of 3 from  $\infty$  to 0. So, colour [3] = Gray,  $d[3] = 0$ .

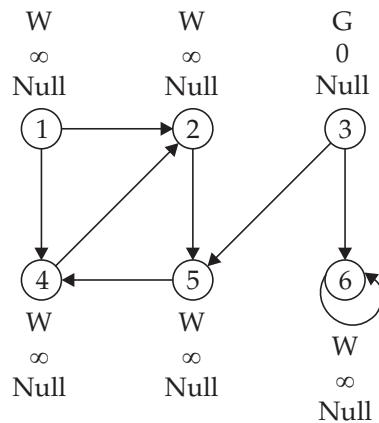


Fig. 14.32. Queue of step-2.

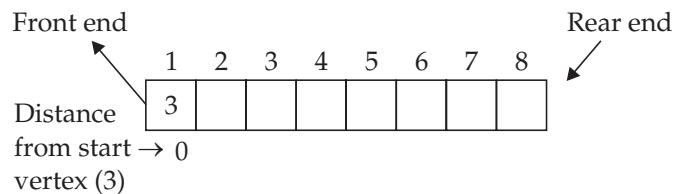


Fig. 14.33. Queue of step-2.

**Step 3.** Delete 3 from queue. Enter the adjacency nodes of 3 which are white in colour into the queue, that is nothing but the 5 and 6. Make 5 and 6 as gray. Make 3 as black.

	1	2	3	4	5	6	7	8
Distance		5	6					
from start → vertex (3)		1	1					

Fig. 14.34. Queue for step-3.

$$\begin{array}{lll}
 \text{So, colour [5] = Gray} & d[5] = 1 & \pi[5] = 3 \\
 \text{colour [6] = Gray} & d[6] = 1 & \pi[6] = 3 \\
 \text{colour [3] = Black} & &
 \end{array}$$

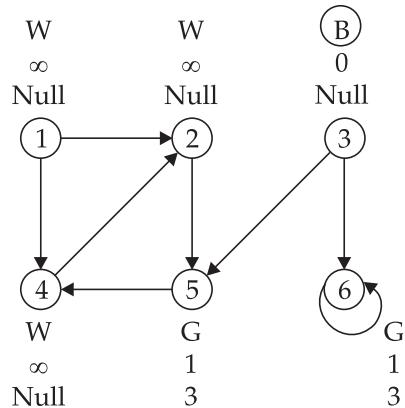


Fig. 14.35. Result of step-3.

**Step 4.** Now delete 5 from queue. Enter the adjacent nodes of 5 into the queue which are white in colour i.e., 4. Make colour of 4 as gray. Make 5 as black.

	1	2	3	4	5	6	7	8
Distance			6	4				
from start → vertex (3)			1	2				

Fig. 14.36. Queue for step-4.

So, color [4] = Gray,  $d[4] = 2$ ,  $\pi[4] = 5$

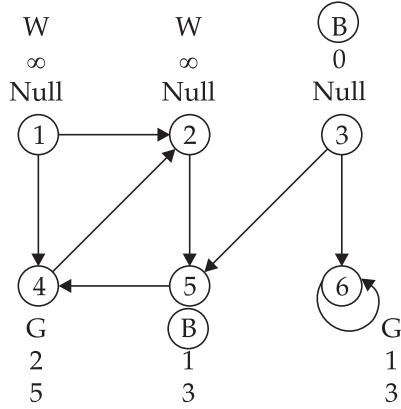


Fig. 14. 37. Result of step-4.

**Step 5.** Delete 6 from queue. As 6 points to itself and there is no adjacent node exist which is white in colour. Hence make 6 black.

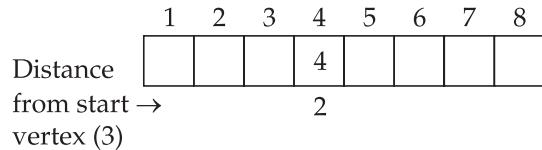


Fig. 14. 38. Queue for step-5.

So, colour [6] = Black

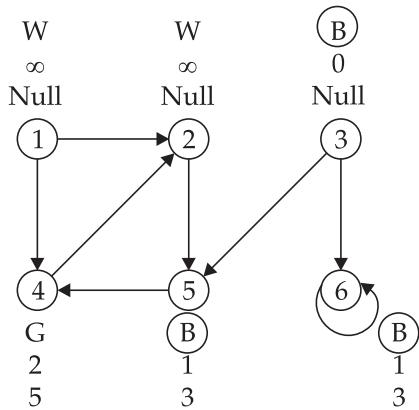


Fig. 14. 39. Result of step-5.

**Step 6.** Now delete 4 from queue. Enter its adjacent node, i.e., 2 into the queue as it is white in colour. Make 2 gray and 4 as black.

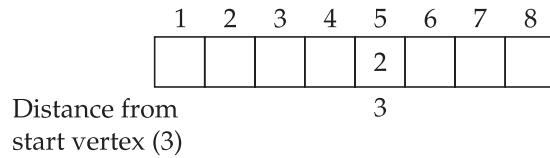


Fig. 14.40. Queue for step-6.

So, colour [2] = Gray,  $d[2] = 3$ ,  $\pi[2] = 4$   
 colour [4] = Black.

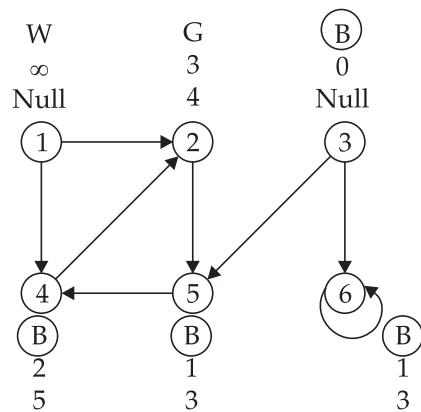


Fig. 14.41. Result of step-6.

**Step 7.** Now delete 2 from queue. But there is no adjacent node of 2 present which is white in colour. So make 2 black.

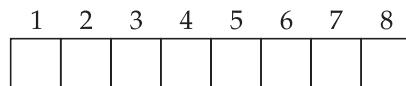


Fig. 14.42. Empty queue.

So, colour [2] = Black.

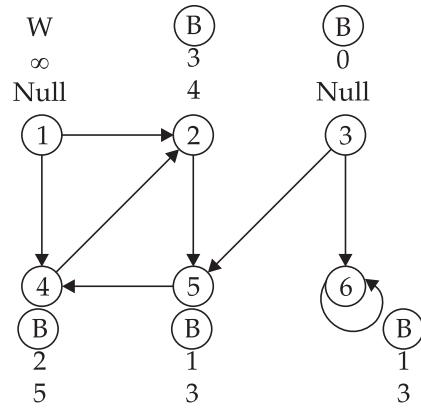


Fig. 14.43. Empty queue.

Now queue is completely empty.

**BFS Tree :** The BFS tree for the graph G is shown in Fig. 14.44.

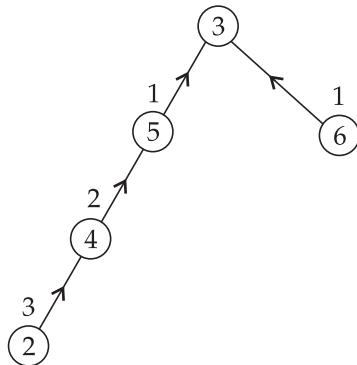


Fig. 14. 44. BFS tree for Example 14.8.

**BFS branches :**  $4 \rightarrow 2$ ,  $5 \rightarrow 4$ ,  $3 \rightarrow 5$  and  $3 \rightarrow 6$ .

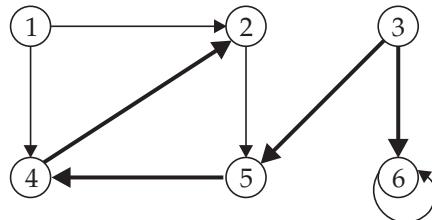


Fig. 14. 45. BFS branches.

## 14.5 DEPTH-FIRST SEARCH (DFS)

The next traversal algorithm that we will study is called *depth-first* search, and it has the nice property that non-tree edges have a good deal of mathematical structure. We assume we are given a graph  $G = (V, E)$ . The graph can be directed or undirected one. As BFS, here we also maintain a colour for each vertex: white means *undiscovered*, gray means *discovered* but not finished processing, and black means *finished*. We also store predecessor pointers, pointing back to the vertex that discovered a given vertex. We will also associate two numbers with each vertex. These are *time stamps*. When we first discover a vertex  $u$  store a counter in  $d[u]$  and when we are finished processing a vertex we store a counter in  $f[u]$ . The purpose of the time stamps will be explained later.

The algorithm is shown below. DFS induces a tree structure while will be discussed further below.

### Algorithm For DFS

DFS ( $G$ ) // main program

1. For all  $u \in V[G]$
2. do Colour [ $u$ ] = white,  $\pi [u] = \text{Null}$  // initialization

```

3. Time = 0
4. For all  $u \in V[G]$ 
5. if Colour [ $u$ ] = white
6. then DFSVISIT ( $u$ )
    DFSVISIT ( $u$ ) // start a search at  $u$ 
1. Colour[ $u$ ] = Gray
2. Time = Time + 1
3.  $d[u] = \text{Time}$  //  $d[u]$  refers to starting time stamp
4. For all  $V \in \text{Adjacent } [u]$ 
5. do if Colour [ $V$ ] = White
6. then  $\pi[V] = \text{White}$ 
7. DFSVISIT ( $V$ )
8. Colour [ $u$ ] = Black
9. Time = Time + 1
10.  $f(u) = \text{Time}$  //  $f(u)$  refers to finishing time stamp.

```

### Running Time of DFS Algorithm

DFS algorithm takes time  $\theta(v)$  to call DFSVISIT. Again DFSVISIT is called exactly once for each vertex  $v \in V$ . Since DFSVISIT is invoked only on white colour vertices paints it gray so that execution of DFSVISIT within the loop is executed  $| \text{Adj}(V) |$  times.

Hence, the total running time =  $O(V + E)$  or  $O(|V| + |E|)$

### Different Types of Edges in DFS Tree:

DFS naturally imposes a tree structure (actually a collection of trees, or a forest) on the structure of the graph. This is just the recursion tree, where the edge  $(u, v)$  arises when processing vertex  $u$  we call DFSVISIT ( $u$ ) for some neighbor  $V$ . For directed graphs the other edges of the graph can be classified as follows:

- Back edges:  $(u, v)$  where  $v$  is a (not necessarily proper) ancestor of  $u$  in the tree. Thus a self loop is considered to be a back edge.
- Forward edges:  $(u, v)$  where  $v$  is the proper descendent of  $u$  in the tree.
- Cross edges:  $(u, v)$  where  $u$  and  $v$  are not ancestors or descendants of one another (in fact, the edge may go between different trees of the forest).

**Example 14.9** Do DFS traversal on the graph given in Fig. 14. 46. Also find its DFS tree.

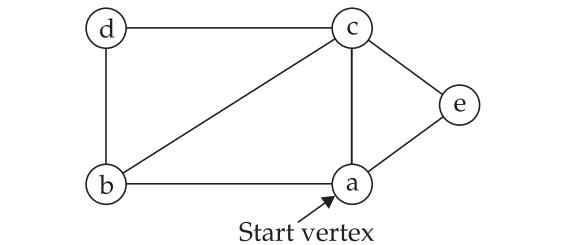


Fig. 14. 46. Graph G for example 14.46 for graph G.

**Solution:** Adjacency list

$a : b, c, e$

$b : a, c, d$

$c : a, b, d, e$

$d : b, c$

$e : a, c$

**Step 1.** Initially make all vertex to be white (w) in colour, starting time stamp is  $\infty$  and predecessor fileds are Null.

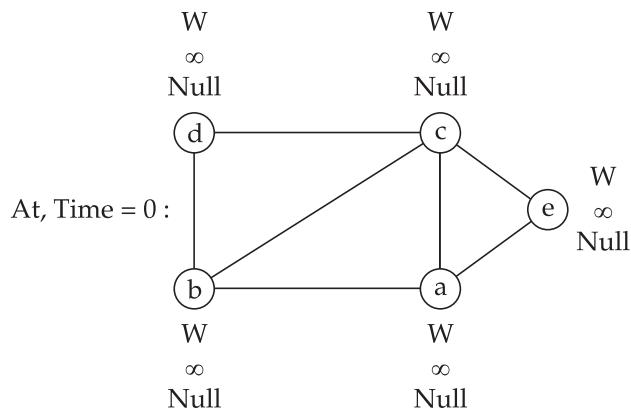


Fig. 14.47. Result of step-1.

**Step 2. DFSVISIT (a)**

Colour [a] = Gray, Time =  $0 + 1 = 1$ ,  $d[a]$  = starting time stamp = 1

Adjacent nodes of a which are white in colour =  $b, c, e$ .

Take  $b$  first.  $\pi[b] = \text{predecessor of } b = a$ .

Do DFSVISIT ( $b$ ).

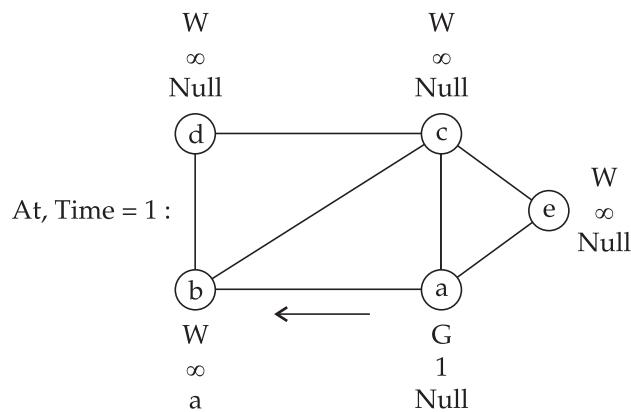


Fig. 14. 48. Result of step-2.

**Step 3. DFSVISIT (b)**

Colour [b] = Gray, Time =  $1 + 1 = 2$ ,  $d[b] = \text{starting time stamp for } b = 2$

Adjacency nodes of  $b$  which are white in color =  $c, d$ .

Take  $c$  first.  $\pi[c] = b$ .

Do DFSVISIT (c)

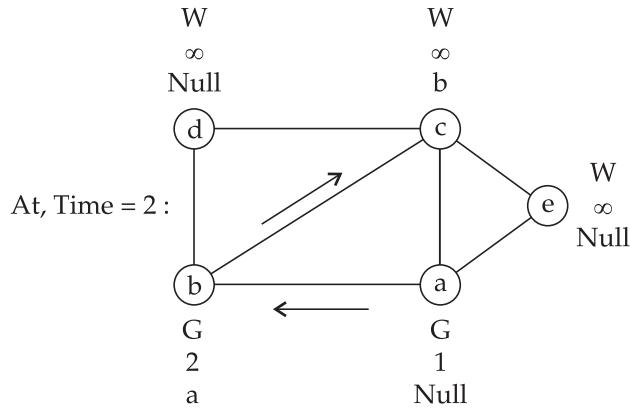


Fig. 14.49. Result of step-3.

**Step 4. DFSVISIT (c)**

Colour [c] = Gray, Time =  $2 + 1 = 3$ .  $d[c] = \text{starting time stamp of } c = 3$ .

Adjacency nodes of  $c$  which are white in colour =  $d, e$ .

Take  $d$  first.  $\pi[d] = c$ .

Do DFSVISIT (c)

At time = 3

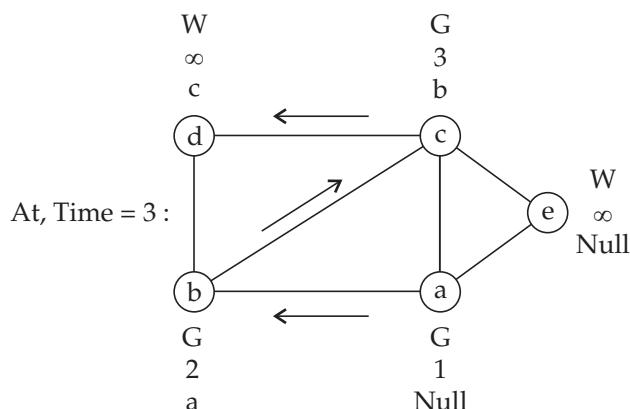


Fig. 14.50. Result of step-4.

**Step 5. DFSVISIT ( $d$ )**

Colour  $[d] = \text{Gray}$ , Time =  $3 + 1 = 4$ ,  $d[d] = \text{starting time stamp} = 4$

There is no white node exists, which is adjacent to  $d$ .

So make  $d$  black.

Time = Time + 1 =  $4 + 1 = 5$

So  $f(d) = \text{finishing time stamp of } d = 5$ .

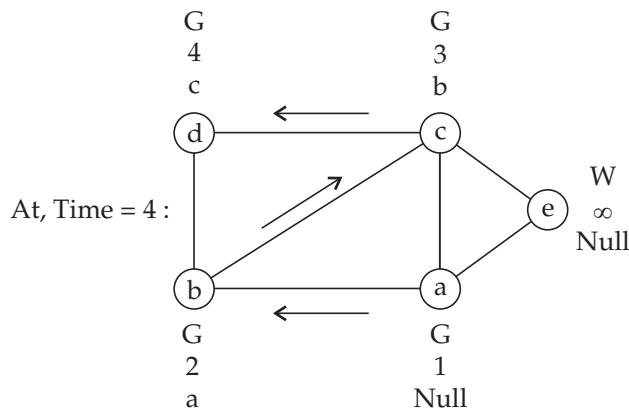


Fig. 14.51. (a) Result of step-5.

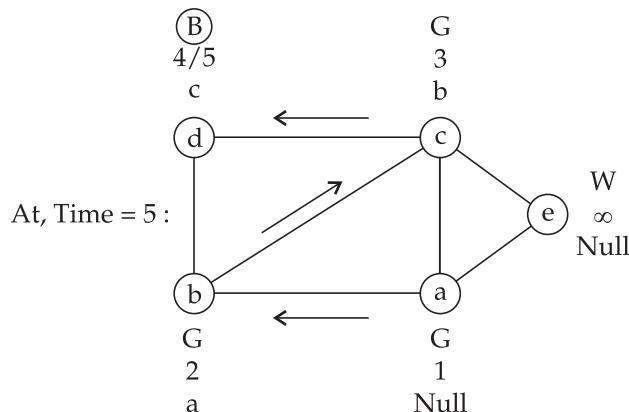


Fig. 14.51. (b) Result of step-5.

Now, we have to move from  $d$  to  $c$  and  $c$  to  $e$  to reach at  $e$ .

**Step 6. DFSVISIT ( $e$ )**

Colour  $[e] = \text{Gray}$ , Time =  $5 + 1 = 6$ ,  $d[e] = \text{starting time stamp of } e = c$ .

From  $e$  we cannot visit any node which is adjacent to  $e$  and white in colour. So make  $e$  black.

$$\text{Now Time} = \text{Time} + 1 = 6 + 1 = 7$$

$$\text{So, } f(e) = \text{finishing time stamp of } e = 7.$$

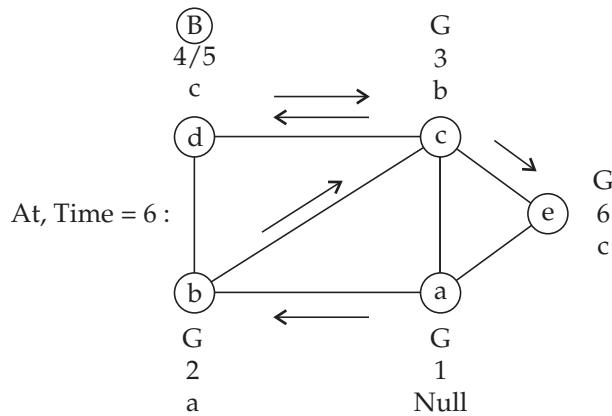


Fig. 14.52. (a) Result of step-6.

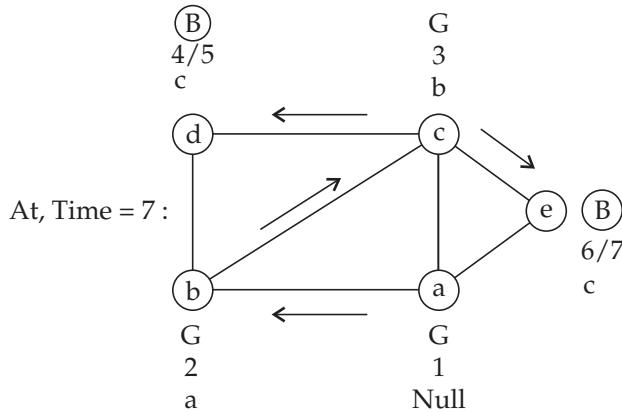


Fig. 14.52. (b) Result of step-6.

From  $e$  we cannot move to any other node. So by recursion move from  $e$  to  $c$  then  $c$  to  $b$  and at last  $b$  to  $a$ .

#### Step 7. Move from $e$ to $c$

Colour [c] = Black, Time =  $7 + 1 = 8$ ,  $f(c) = \text{finishing time stamp} = 8$ .

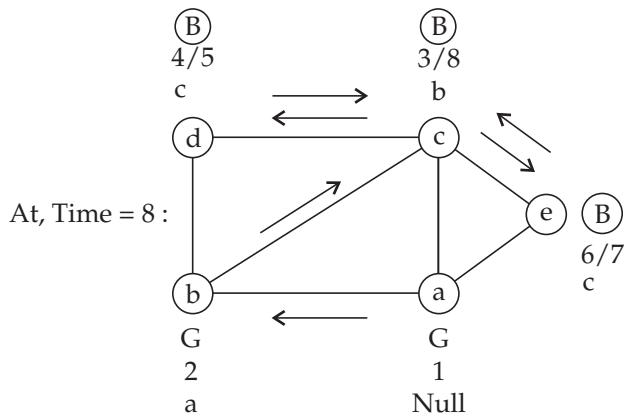


Fig. 14.53. Result of step-7.

#### Step 8. Move from c to b

Color [b] = Black, time =  $8 + 1 = 9$ ,  $f(b)$  = finishing time stamp = 9

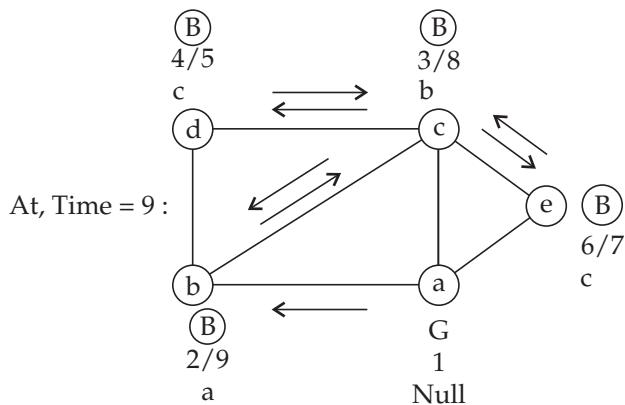


Fig. 14.54. Result of step-8.

#### Step 9. Move from b to a

Colour [a] = Black, Time =  $9 + 1 = 10$ ,  $f(a)$  = 10

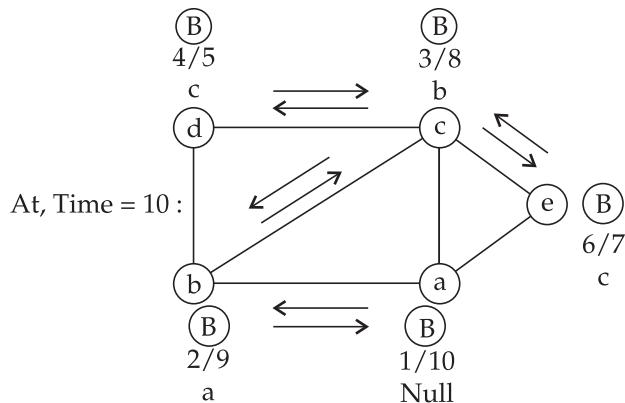
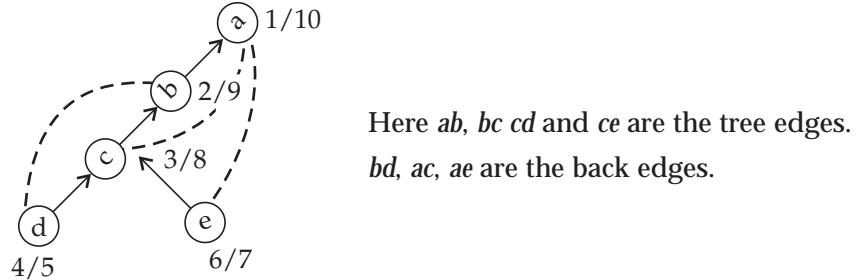


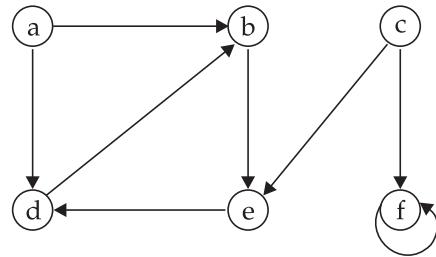
Fig. 14.55. Result of step-9.

So, from the above DFS traversal it is clearly visible that edge  $ac$ ,  $ae$  and  $bd$  are never used during the entire traversal process. These are the back edges. So the DFS tree of the given graph becomes:



**Fig. 14.56.** DFS tree for example 14.9.

**Example 14.10** Do DFS traversal on the graph  $G$ . Find its DFS tree.



**Fig. 14.57.** Graph  $G$  for (example 14.10) for graph  $G$ .

**Solution :** Adjacency list

- $a : b, d$
- $b : e$
- $c : e, f$
- $d : b$
- $e : d$
- $f : f$

**Step 1.** Initially make all vertex to be white (W) in colour, starting time stamps is  $\infty$  and predecessor fields are NULL.

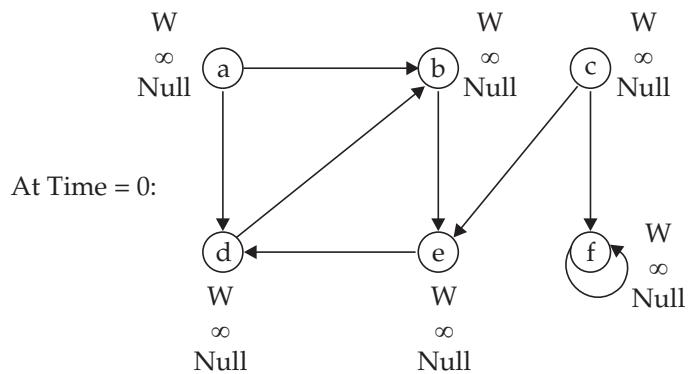


Fig. 14.58. Result of step-1.

**Step 2. DFSVISIT (a)**

Colour [a] = Gray

Time =  $0 + 1 = 1$  $d[a]$  = starting time stamp = 1

Adjacency nodes of a which are white in colour i.e., b, d.

Take b first.  $\pi[b]$  = predecessor of b = a

Do DFSVISIT (b)

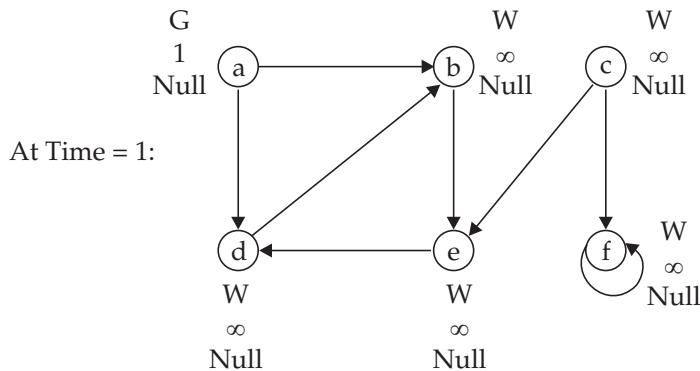


Fig. 14.59. Result of step-2.

**Step 3. DFSVISIT (b)**

Colour [b] = Gray

Time =  $1 + 1 = 2$  $d[b]$  = starting time stamp = 2Adjacency nodes of b which are white in colour i.e., e. Make predecessor of e is b i.e.,  $\pi[e] = b$ . Do DFSVISIT (e).

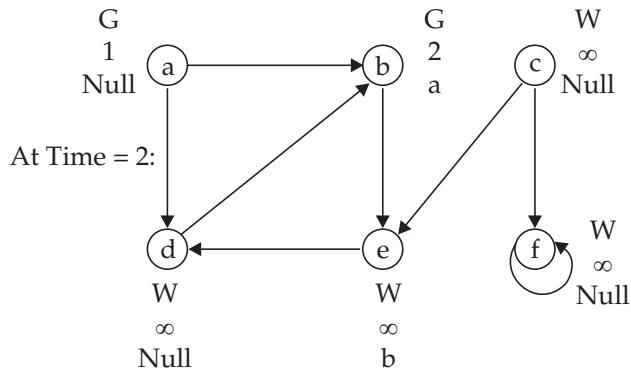


Fig. 14.60. Result of step-3.

**Step 4. DFSVISIT (c)**

Colour [c] = Gray

Time =  $2 + 1 = 3$

d [c] = starting time stamp = 3

d is the only one node which is adjacent to e and white in colour.

Take d. Make predecessor of d is e. So  $\pi[d] = e$ 

Do DFSVISIT (d)

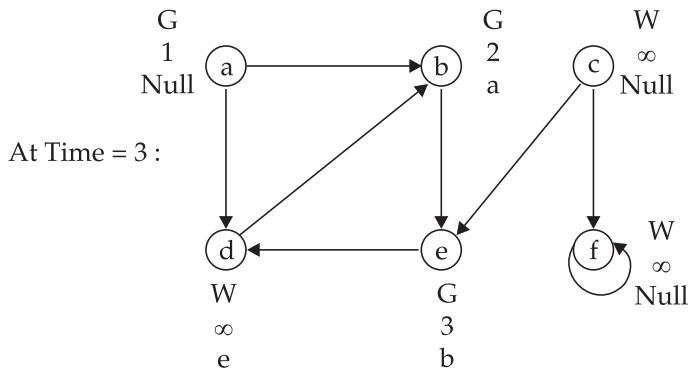


Fig. 14.61. Result of step-4.

**Step 5. DFSVISIT (d)**

Colour [d] = Gray,

Time =  $3 + 1 = 4$

d[d] = Starting time stamp = 4

There doesn't exist any node adjacent to d and white in colour. So make d black.

Now, Time = Time + 1 =  $4 + 1 = 5$

So,  $f(d)$  = finishing time stamp of d = 5

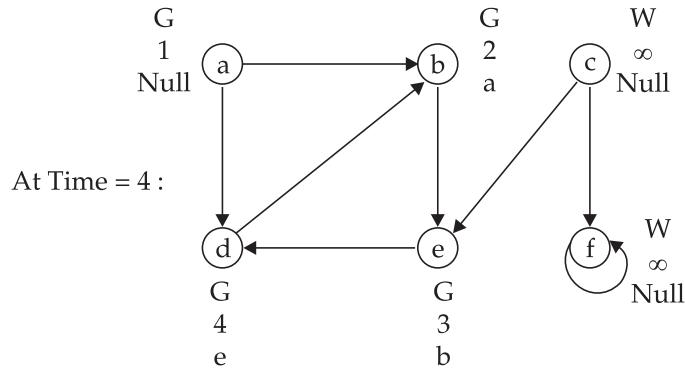


Fig. 14.62. (a) Result of step-5.

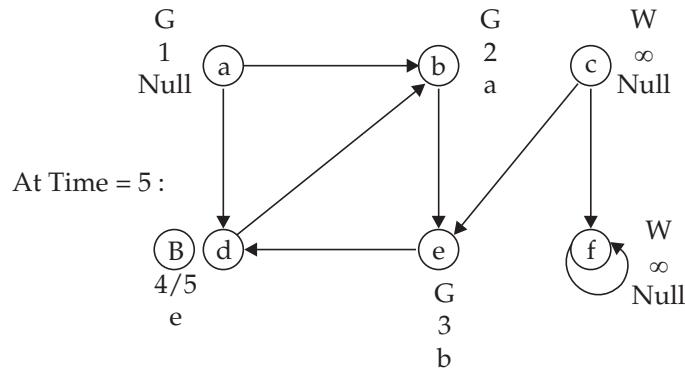


Fig. 14.62. (b) Result of step-5.

Now from  $d$  we cannot go to any other node. By recursion move from  $d$  to  $e$ ,  $e$  to  $b$  then at last  $b$  to  $a$ .

**Step 6.** Make  $e$  black. So, colour  $[e] = \text{Black}$ .

Time =  $5 + 1 = 6$

$f(e)$  = finishing time stamp = 6.

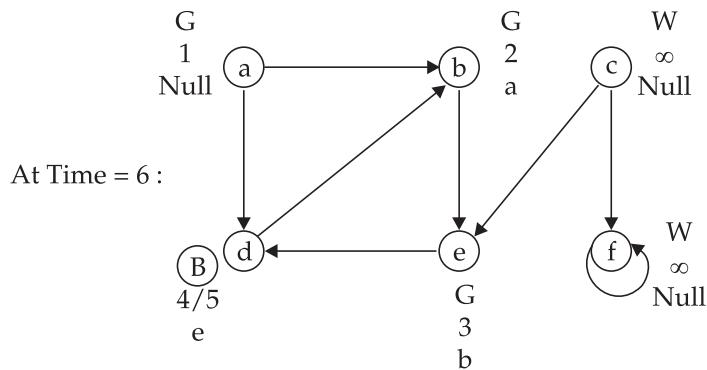


Fig. 14.63. Result of step-6.

**Step 7.** Make  $b$  black. colour  $[b] = \text{Black}$

$$\text{Time} = 6 + 1 = 7$$

$$f(b) = \text{finishing time stamp} = 7$$

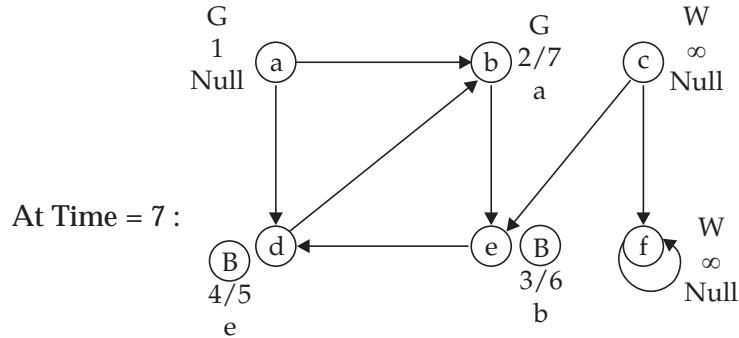


Fig. 14.64. Result of step-7.

**Step 8**

Make  $a$  black. Colour  $[a] = \text{Black}$ .

$$\text{Time} = 7 + 1 = 8. \text{ So } f(a) = \text{finishing time stamp} = 8.$$

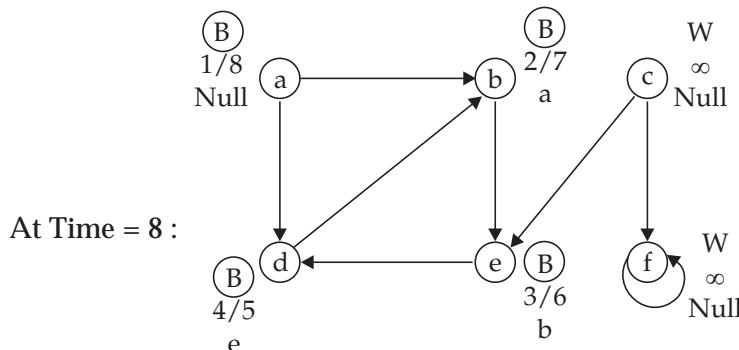


Fig. 14.65. Result of step-8.

After reaching  $a$  we cannot move to any white node. So Do DFSVISIT( $c$ ).

**Step 9 DFSVISIT ( $c$ )**

Colour  $[c] = \text{Gray}$

$$\text{Time} = 8 + 1 = 9$$

$$d[c] = \text{starting time stamp} = 9$$

Adjacency nodes of  $c$  that are white in colour i.e.,  $f$ .

So predecessor of  $f$  is  $e$ .  $\pi[f] = c$ .

Do DFSVISIT (*f*)

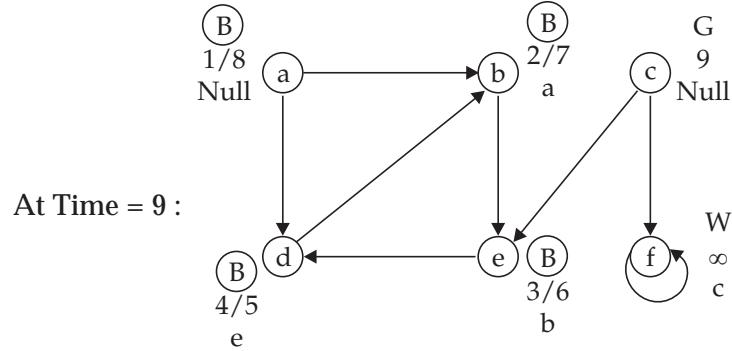


Fig. 14.66. Result of step-9.

#### Step 10 DFSVISIT (*f*)

Colour [*f*] = Gray

Time = 9 + 1 = 10

*d*[*f*] = starting time stamp = 10

There does not exist any node which is adjacent to *f* and white in colour.

So make *f* black. colour [*f*] = Black.

Now time = Time + 1 = 10 + 1 = 11

*f*[*f*] = Finishing time stamp = 11.

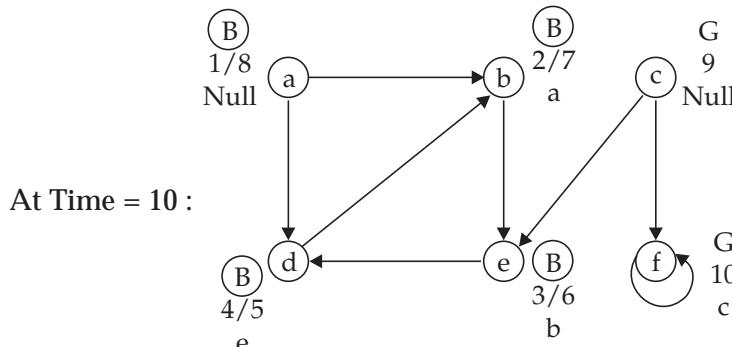


Fig. 14.67. (a) Result of step-10.

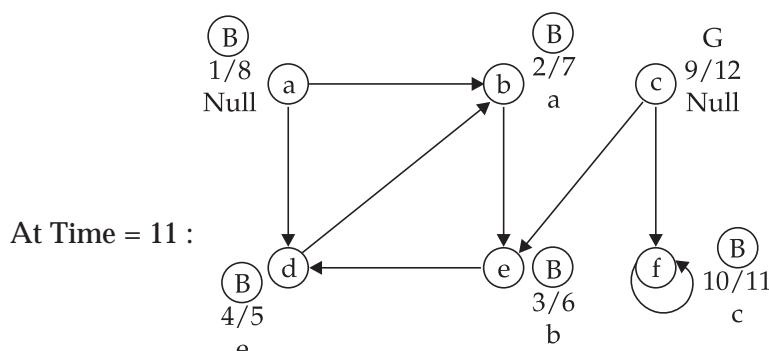


Fig. 14.67. (b) Result of step 10.

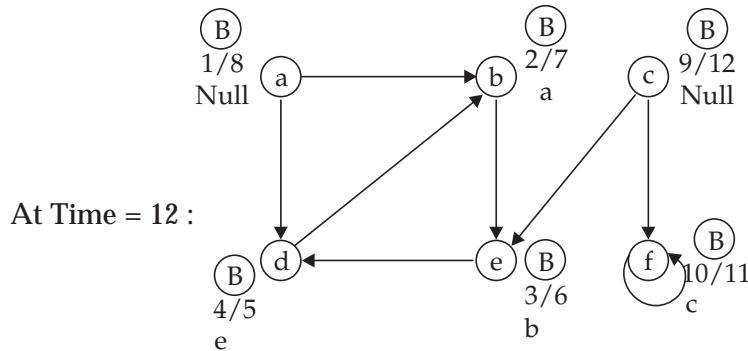
From  $f$  we can not move to any other node. So move to  $c$ .

### Step - 11 Move from $f$ to $c$

Make  $c$  black. So colour  $[c] = \text{Black}$

$$\text{Time} = 11 + 1 = 12$$

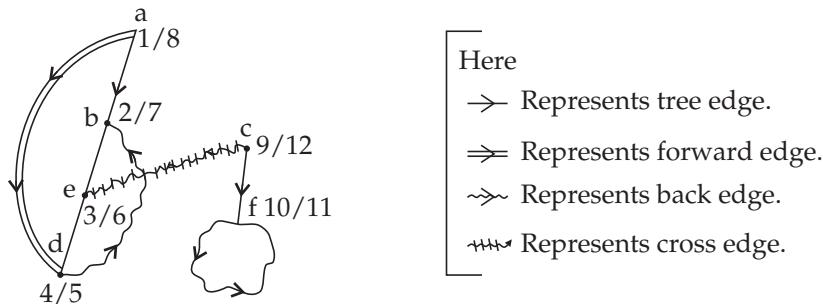
$$f[c] = \text{Finishing time stamp} = 12$$



**Fig. 14.68.** Result of step-11.

### DFS Forest

So the DFS forest of the given graph becomes



**Fig. 14.69.** DFS forest for Example 14.9.

In the above DFS forest

Tree edges are -  $ab$ ,  $be$ ,  $ed$  and  $cf$

Forward edge is :  $ad$

Back edge is :  $db$

Cross edge is :  $ce$

## 14.6 APPLICATION OF DFS

---

We are now going to study two different applications of DFS. They are : Topological sort and Strongly connected components.

**(i) Topological Sort:** One of the applications of topological DFS is sort which is only applicable on “directed acyclic graph” or “DAG”. A topological sort of a dag  $G = (V, E)$  is a linear ordering of all its vertices such that if  $G$  contains on edge  $(u, v)$  then  $u$  appears before  $v$  in the ordering.

In topological sort we arrange the elements along a horizontal line so that all directed edges go from left to right, i.e., we arrange in descending order of finishing time stamp. The partial implementation of topological sort in day to day of life is we wear our shirts, paints, socks in an order.

**Note:** If the graph contains a cycle then no linear ordering is possible.

### Algorithm for Topological-sort

TOPOLOGICAL-SORT (G)

1. Call DFS (G) to determine the finishing time stamp  $f(v)$  for each vertex  $v$ .
2. As each vertex is finished, insert it onto the front of a linked list.
3. Return the linked list of vertices.

### Running Time of TOPOLOGICAL - SORT (G)

In the above algorithm the call procedure for DFS (G) takes  $\theta(V + E)$  and it takes  $O(1)$  time to insert each of  $|V|$  nos of vertices (i.e., total  $O(V)$ ) onto the front of linked list.

**(ii) Strongly Connected Components:** Strongly connected components is an another application of DFS. In this technique we decompose a directed graph into its strongly connected components. After decomposing the graph into strongly connected connections among components, such algorithms run separately on each one and then combine the solutions according to the structure of the connections among components.

A strongly connected component of a directed graph  $G = (V, E)$  is a maximal set of vertices  $C \subseteq V$  such that for every pair of vertices  $u$  and  $v$  in  $C$ , we have  $u \rightsquigarrow v$  and  $v \rightsquigarrow u$ ; i.e., vertices  $u$  and  $v$  are reachable from each other.

### Algorithm for Strongly - connected - components

STRONGLY - CONNECTED - COMPONENTS (G)

1. Call DFS (G) to compute finishing times u.f. for each vertex  $u$ .
2. Compute transpose of  $G$  (i.e.,  $G^T$ )
3. Call DFS ( $G^T$ ), but in the main loop of DFS, consider the vertices in order of decreasing u.f. (as computed in line 1)
4. Output the vertices of each tree in the depth- first formed in line 3 as a separate strongly connected component.

### Running time of STRONGLY- CONNECTED-COMPONENTS(G)

The above linear-time algorithm computer the strongly connected components of a directed graph  $G = (V, E)$  in  $\theta(V + E)$  times (where  $V \rightarrow$  set of vertices and  $E \rightarrow$  set of edges in graph  $G$ .)

**Example 14.11** Find the ordering of vertices produced by TOPOLOGICAL-SORT when it run on the given DAG “G” in Fig. 14.70.

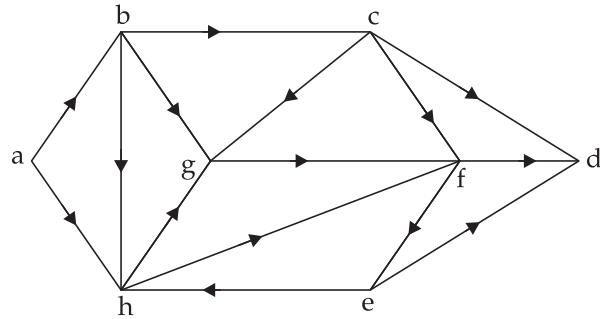


Fig. 14.70. DAG for Example 14.11.

### Solution : Adjacency list for DAG G

$a : b, h$

$b : c, g, h$

$c : d, f, g$

$d : Null$

$e : d, h$

$f : e$

$g : f$

$h : f, g$

Applying DFS traversal on graph G, taking vertex a as the start node.

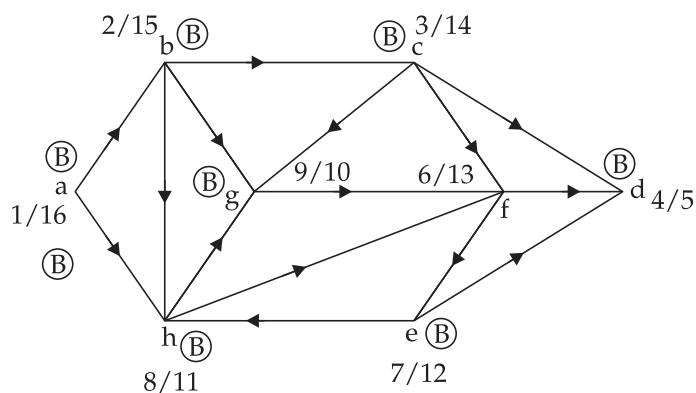


Fig. 14.71. DFS traversal on DAG.

**Note :** To see details about DFS traversal refers section 14.3.2.

### DFS Forest

Now the DFS forest of the above graph becomes

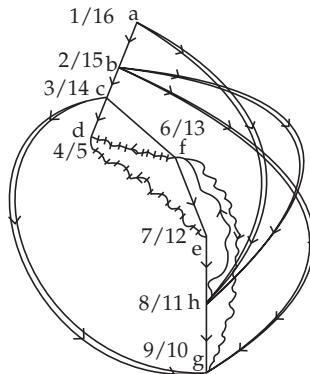


Fig. 14.72. DFS forest for Example 14.11.

In the above DFS forest

Tree edges are –  $ab, bc, cd, ef, fe, eh$  and  $hg$  represented as

Forward edges are –  $ah, bh, bg$  and  $cg$  represented as

Back edges are –  $gf$  and  $hf$  represented as

Cross edges are –  $fd$  and  $cd$  represented as

### Topological order

Name of the vertices	a	b	c	f	e	h	g	d
Finishing time stamp	16	15	14	13	12	11	10	5

Here all directed edges run from left to right.

**Example 14.12** Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the DAG of given in Fig. 14. 73.

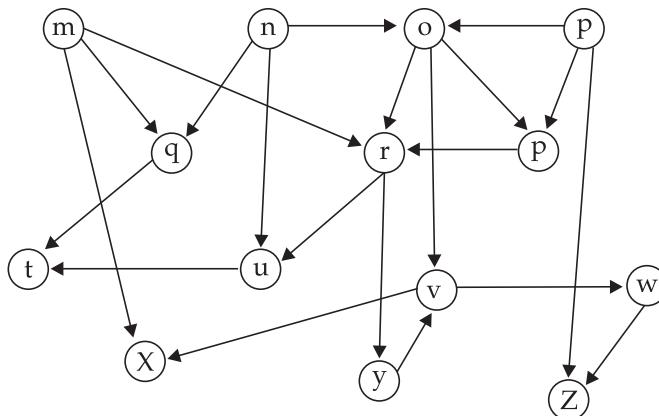


Fig. 14.73. (a) DAG for Example 14.12.

**Solution : Adjacency list for DAG G**

$m : q, r, x$	$t : \text{Null}$
$n : o, q, u$	$u : t$
$o : r, s, v$	$v : w, x$
$p : o, s, z$	$w : z$
$q : t$	$x : \text{Null}$
$r : u, y$	$y : v$
$s : r$	$z : \text{Null}$

Applying DFS traversal on the above graph, taking  $m$  as the start vertex :

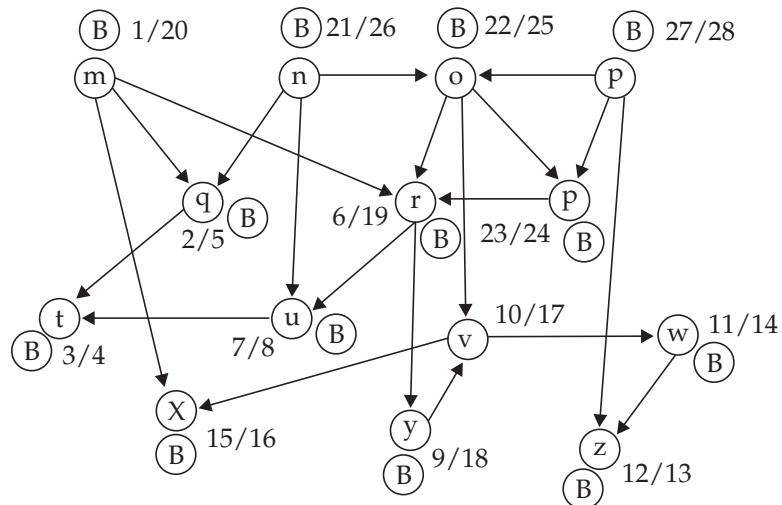


Fig. 14.73. (b) DFS traversal on DAG.

**DFS Forest**

Now the DFS forest of the above graph becomes

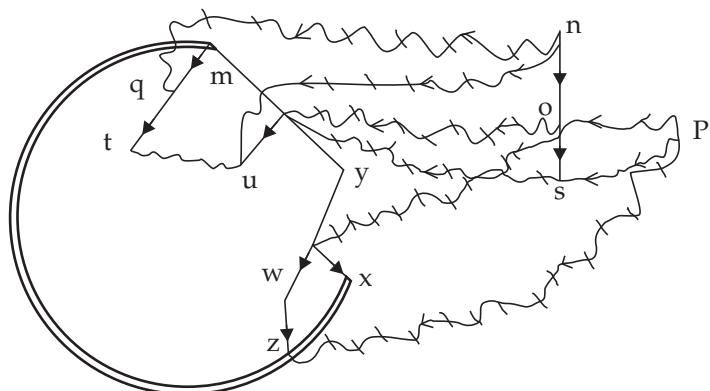


Fig. 14.74. DFS forest of Example 14.72.

In the above DFS forest

Tree edges are :  $mq$ ,  $qt$ ,  $mr$ ,  $ru$ ,  $yu$ ,  $vw$ ,  $wz$ , no and os represented as  $\rightarrow$

Forward edges are :  $mx$ ,  $ry$  and  $vx$  represented as  $\overrightarrow{\overrightarrow{}}$

Back edges are :  $ut$  represented as  $\overleftarrow{\overrightarrow{\overleftarrow{}}}$

Cross edges are :  $nm$ ,  $nu$ ,  $or$ ,  $ov$ ,  $sr$ ,  $po$ ,  $ps$ ,  $pz$  and represented as  $\overleftarrow{\overrightarrow{\overleftarrow{\overrightarrow{\overleftarrow{}}}}}$

Now the topological order becomes

Name of the vertices	$p$	$n$	$o$	$s$	$m$	$r$	$y$	$v$	$x$	$w$	$z$	$u$	$q$	$t$
Finishing time stamp	28	26	25	24	20	19	18	17	16	14	13	8	5	4

**Example 14.13** Find the strongly connected components of the directed graph give in Fig. 14.75.

Take 'a' as the start vertex.

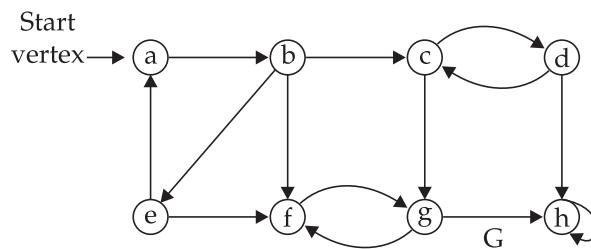


Fig. 14.75. Directed graph for Example 14.13.

**Solution :** For the given  $G$  the adjacency list is

$a : b$

$e : a, f$

$b : c, e, f$

$f : g$

$c : d, g$

$g : f, h$

$d : c, h$

$h : h$  (as there is a self-loop on  $h$ ).

**Step 1:** Applying DFS traversal on the given  $G$  (Here detail steps of DFS traversal has been omitted. Readers are advised to do this by your own for practice).

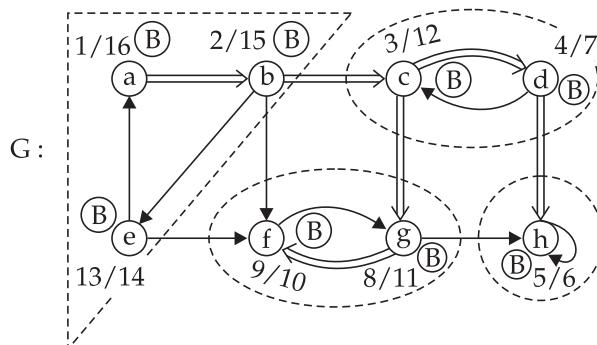


Fig. 14.76. DFS traversal on graph  $G$ .

**Step 2.** Finding the transpose of  $G$  i.e.,  $G^T$  and thereafter applying DFS traversal on  $G^T$  (according to the decreasing order of finishing time stamp.)

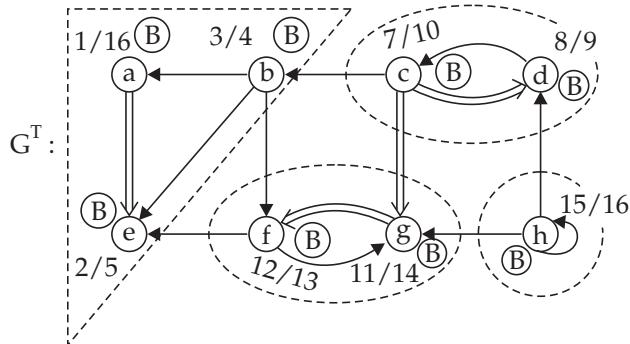


Fig. 14.77. Transpose of graph  $G$ .

**Adjacency list of  $G$ :**

$a : e$	$e : b$
$b : a$	$f : b, e, g$
$c : b, d$	$g : c, f$
$d : c$	$h : d, g, h$

**Step 3.** So finally we get four strongly connected components. They are represented as strongly connected graph.

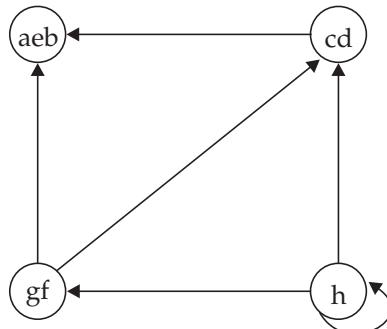


Fig. 14.78. Strongly connected components of graph  $G$ .

Each one i.e.,  $aeb$ ,  $cd$ ,  $gf$  and  $h$  standing alone makes a tree.

## 14.7 MINIMUM SPANNING TREE

Before moving to minimum spanning tree (in short-MST) first we see what is spanning tree.

### Spanning Tree

The graph  $G = (V, E)$  is traversed in such a way that if forms an acyclic graph which is a tree consisting of all the vertices of graph  $G$  but not edges, such a tree is called spanning tree. Therefore, the graph  $G$  has many spanning trees.

If there are  $n$  number of vertices in a weighted graph, then there must be  $(n - 1)$  spanning tree edges present. Lets take an example to clear our ideas about spanning tree. Consider the weighted graph given in Fig. 17.79.

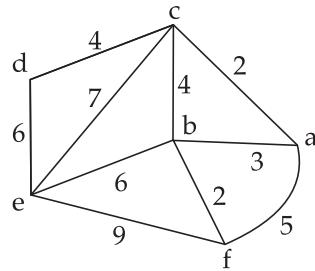


Fig. 14.79. Graph G.

Different scenario of spanning tree formation of graph G are given:

**(i) First spanning tree:** Here the spanning tree edges  $cd$ ,  $bc$ ,  $ab$ ,  $af$  and  $fe$  are represented as  $\sim\sim\sim\sim$  line. So cost of the spanning tree =  $4 + 4 + 3 + 5 + 9 = 25$

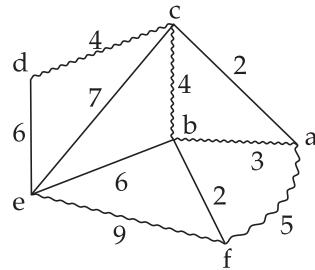


Fig. 14.80. 1<sup>st</sup> attempt for spanning tree formation.

**(ii) Second spanning tree:** The spanning tree edges are  $cd$ ,  $ac$ ,  $be$ ,  $bf$  and  $af$ .

Cost of the spanning tree =  $4 + 2 + 6 + 2 + 5 = 19$

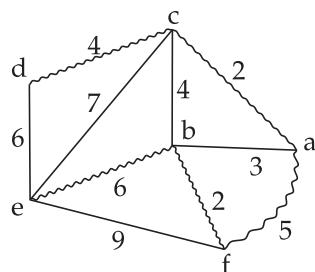
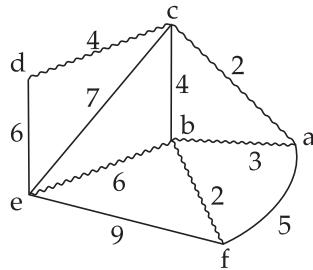


Fig. 14.81. 2<sup>nd</sup> attempt for spanning tree formation.

**(iii) Third spanning tree:** In the 3<sup>rd</sup> case, the spanning tree edges are  $cd$ ,  $ac$ ,  $be$ ,  $bf$  and  $ab$ .

$$\text{Cost of the spanning tree} = 4 + 2 + 6 + 2 + 3 = 17$$



**Fig. 14. 82.** 3<sup>rd</sup> attempt for spanning tree formation.

Likewise we may get many number of spanning trees having different costs (costs may be different or same). Finding the spanning tree having minimum cost out of so much choices is the idea behind the minimum spanning tree. Let's look at its formal definition.

### Minimum Spanning Tree (MST)

While considering the minimum spanning tree, we have to first think for an edge-weighted graph. In an edge-weighted graph  $G = (V, E)$  we associate weights or costs through weight function  $W : E \Rightarrow R^+$  with each edge. A minimum spanning tree (MST) of an edge weighted graph is a spanning tree whose weight (the sum of the weight of its edges) is no larger than the weight of any other spanning tree.

If there are  $n$  vertices in the MST, then there is  $(n - 1)$  no of minimum spanning tree edge present.

### Minimum Spanning Tree Algorithms

There are basically two methods using which we can find the MST for a given weighted undirected graph. These are:

\* Kruskal's algorithm      }  
 \* Prim's algorithm      } (Based on Greedy approach)

Both these algorithms differ in their methodology, but both eventually ends with the MST. Kruskal's algorithm uses edges and Prim's algorithm uses vertex connections in determining the MST. Both algorithms are greedy algorithms that run in polynomial time. At each step of an algorithm, one of the several possible choices must be made. Let's now study in detail about these algorithms.

## 14.8 KRUSKAL'S ALGORITHM

Kruskal's algorithm is a graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of edges that forms a tree that includes every vertex, where the total weights of all edges in the tree is minimized. If the graph is not connected it finds a minimum spanning tree forest. Kruskal's algorithm is a suitable example of Greedy-algorithm.

### MST - Kruskal Algorithm

**KRUSKAL (G, W)**

1.  $T = \emptyset$  //  $T$  is the set of edges for the given graph  $G$ .
2. Sort all edges at  $G$  in non-decreasing order i.e.,  
 $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$
3. For all  $u \in V(G)$  to **MAKESET** ( $u$ )
4. For  $i = 1$  to  $m$  //  $m$  is the total number of edges.
5. do let  $e_i = (u, v)$
6. if **FINDSET** ( $u$ )  $\neq$  **FINDSET** ( $v$ ) // to check whether  $u$  and  $v$  in the same set or not.
7. then  $T = T \cup \{(u, v)\}$
8. **Union** (**FINDSET** ( $u$ ), **FINDSET** ( $v$ ))
9. return  $T$

### Finding Running Time of Kruskal's Algorithm

- (i) Line 1 of the algorithm takes  $O(1)$  time as well see we will see the given graph and write down the edges which are present.
- (ii) Line 2 takes  $O(E \log E)$  to sort all the weighted edges in non-decreasing order.
- (iii) Line 3 takes  $O(V)$  times as for each vertex in the graph we have to make a set.
- (iv) Line 4 takes  $O(E)$  times, because the for loop runs 1 to no of edges times.
- (v) Line 5 runs no of edges times i.e.  $O(E)$  times.
- (vi) Line 6 run  $O(E)$  times.
- (vii) Line 7 runs  $O(V)$  times.
- (viii) Line 8 runs  $O(E)$  times.

So the total running time of the Kruskal's algorithm is  $O(m \log m)$ , where  $m$  is the total no of edges in a given graph.

We also know that  $|E| < |V|^2$

Then  $\log |E| = O(\log V)$

The running time of Kruskal's algorithm is  $O(E \log V)$

**Example 14.14** Find the MINIMUM SPANNING TREE of the graph  $G$  given in Fig. 14.83 by using Kruskal's algorithm.

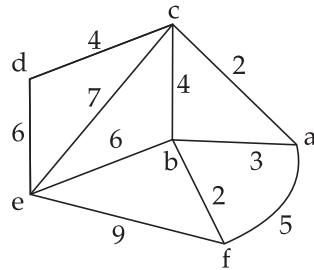


Fig. 14.83. Graph  $G$  for Example 14.14.

**Solution:** Our primary step is to arrange the edges in non-decreasing order.

$$\text{So, } e_1 = (a, c) = 2$$

$$e_2 = (b, f) = 2$$

$$e_3 = (a, b) = 3$$

$$e_4 = (c, d) = 4$$

$$e_5 = (c, b) = 4$$

$$e_6 = (a, f) = 5$$

$$e_7 = (d, e) = 6$$

$$e_8 = (b, e) = 6$$

$$e_9 = (c, e) = 6$$

$$e_{10} = (e, f) = 9$$

Now MAKESET ( $u$ ) becomes  $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$

(1) Take  $e_1$  i.e.,  $(a, c)$ . FINDSET ( $a$ )  $\neq$  FINDSET ( $c$ )

Then MAKESET becomes  $\{a, c\}, \{b\}, \{d\}, \{e\}, \{f\}$ .

(2) Take  $e_2$  i.e.,  $(b, f)$ . FINDSET ( $b$ )  $\neq$  FINDSET ( $f$ ). Then MAKESET becomes

$\{a, c\}, \{b, f\}, \{d\}, \{e\}$ .

(3) Take  $e_3$  i.e.,  $(a, b)$ . FINDSET ( $a$ )  $\neq$  FINDSET ( $b$ ). Then MAKESET becomes

$\{a, c, b, f\}, \{d\}, \{e\}$ .

(4) Take  $e_4$  i.e.,  $(c, d)$ . FINDSET ( $c$ )  $\neq$  FINDSET ( $d$ ). Then MAKESET becomes

$\{a, b, c, d, f\}, \{e\}$ .

(5) Take  $e_5$  i.e.,  $(c, b)$ . FINDSET ( $c$ ) = FINDSET ( $b$ ). Nothing to do as  $c$  and  $b$  are in the same set.

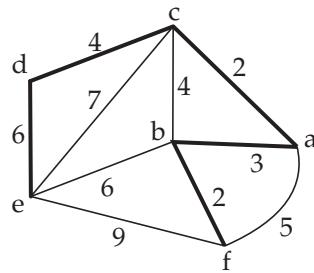
(6) Take  $e_6$  i.e.,  $(a, f)$ . FINDSET ( $a$ ) = FINDSET ( $f$ ). Then nothing to do as  $a$  and  $f$  are in the same set.

(7) Take  $e_7$  i.e.,  $(d, e)$ . FINDSET ( $d$ )  $\neq$  FINDSET ( $e$ ). Then MAKESET is  $\{a, b, c, d, e, f\}$ .

- (8) Take  $e_8$  i.e.,  $(b, e)$ , FINDSET ( $b$ ) = FINDSET ( $e$ ). Nothing to do as  $b$  and  $e$  are in the set.  
 (9) Take  $e_9$  i.e.,  $(c, e)$ , FINDSET ( $c$ ) = FINDSET ( $e$ ). Nothing to do as  $c$  and  $e$  are in the same set.  
 (10) Take  $e_{10}$  i.e.,  $(e, f)$ . FINDSET ( $e$ ) = FINDSET ( $f$ ). So nothing to do as  $e$  and  $f$  are in the same set.

From the above operation spanning tree edges are :

$$\begin{array}{ll} e_1 = (a, c) = 2 & e_2 = (b, f) = 2 \\ e_3 = (a, b) = 3 & e_4 = (c, d) = 4 \\ \text{and} & e_7 = (d, e) = 6 \end{array}$$



**Fig. 14. 84** Representing spanning tree edges.

**Note :** (Bold shaded edges represent spanning tree edges.)

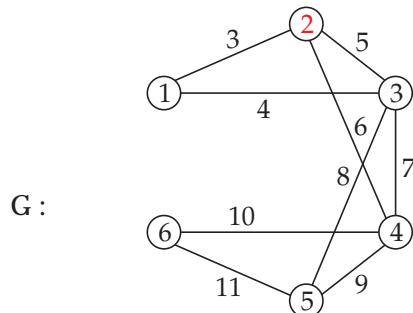
Cost of the MST =  $2 + 2 + 3 + 4 + 6 = 17$

**Example 14.15** Find the MST of graph G with n vertex when  $n \leq 6$ . There is an edge from  $v_i$  to  $v_j$  if  $|i - j| \leq 2$  and  $w(v_i, v_j) = i + j$ .

**Solution :** Given conditions

- $n$  must be  $\leq 6$
- There is an edge from  $v_i$  to  $v_j$  exist provided  $|i - j| \leq 2$
- Weight of an edge from  $i$  to  $j$  is  $w(v_i, v_j) = i + j$

If  $n = 6$  the graph G will be like



**Fig. 14. 85.** Graph G.

Arranging edges in the non-decreasing order

$$e_1 = \{1, 2\} = 3, \quad e_2 = \{1, 3\} = 4, \quad e_3 = \{2, 3\} = 5, \quad e_4 = \{2, 4\} = 6$$

$$e_5 = \{3, 4\} = 7, \quad e_6 = \{3, 5\} = 8, \quad e_7 = \{4, 5\} = 9, \quad e_8 = \{4, 6\} = 10, \quad e_9 = \{5, 6\} = 11$$

Applying the Kruskal's algorithm for finding the MST, the minimum spanning tree edges are =  $e_1, e_2, e_4, e_6, e_8$ .

MST becomes :

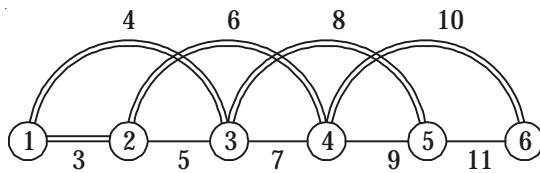


Fig. 14.86. MST edges of graph G.

Here double line (=) represent MST edges. So the total cost of the MST =  $3 + 4 + 6 + 8 + 10 = 31$

Applying the same procedure for  $n$  vertices we get the MST like below.

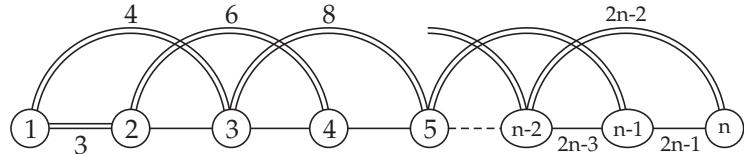


Fig. 14.87. MST edges for  $n$  vertex graph.

Total cost of the MST for  $n$  vertex becomes

$$= 3 + 4 + 6 + 8 + \dots + (2n - 2) = 3 + 2[2 + 3 + 4 + \dots + (n - 1)]$$

$$= 3 + 2 \left[ \frac{n(n-1)}{2} - 1 \right] \left[ \begin{array}{l} \text{As we know } 1 + 2 + 3 + \dots + n(n-1) = \frac{n(n-1)}{2} \\ \Rightarrow 2 + 3 + \dots + (n-1) = \frac{n(n-1)}{2} - 1 \end{array} \right]$$

$$= 3 + 2 \left[ \frac{n(n-1)-2}{2} \right]$$

$$= 3 + (n^2 - n - 2) = n^2 - n + 1.$$

## 14.9 PRIM'S ALGORITHM

Prim's algorithm is a Greedy algorithm which finds the minimum spanning tree of a connected weighted undirected graph. In other words, it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

In this section, we describe two different ways of solving Prim's algorithm to find minimum spanning tree of a given graph. One is based on adjacency matrix and the other is based on priority queue.

### Prim's Algorithm using Adjacency Matrix

Below we write the steps of the algorithm. These steps are very much simple and easy. The algorithm takes  $O(V^2)$  time to run ; where V is the number of vertices of graph G.

1. Create a set MST-SET that keeps track of vertices already included in the MST.
2. Assign a key value to all vertices in the input graph. Assign the key value of the start vertex to be 0 and rest other to be assigned as infinity.
3. While MST-SET doesn't include all vertices
  - (i) Pick a vertex  $u$  which is not there in MST-SET and has minimum key value.
  - (ii) Include  $u$  to MST-SET.
  - (iii) Update a key value of all adjacent vertices of  $u$ . To update the key value, iterate through all adjacent vertices. For every adjacent vertex  $v$ , if weight of edge  $u-v$  is less than the previous key value of  $v$ , update the weight of  $u-v$ .

**Note:** Key value denotes the distance.

**Example 14.16** Run the Prim's algorithm using adjacency matrix on graph G to find the minimum spanning tree. Take b as the start vertex.

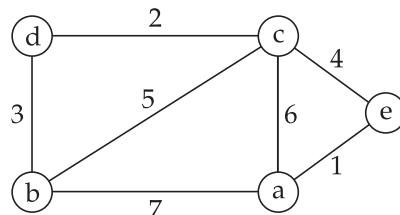
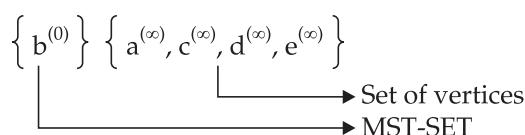


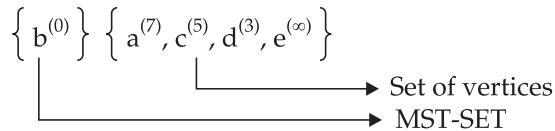
Fig. 14.88. Graph G for Example 14.16.

**Solution :**

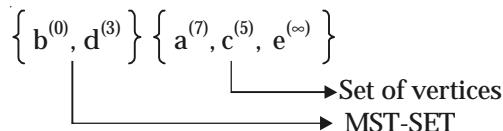
**Step 1.** Key value of vertex  $b$  is 0. The key values of other vertices are infinity. So we get the following two sets.



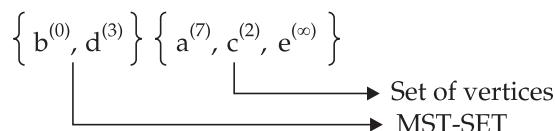
**Step 2.** The distance from  $b$  to  $a$  is 7,  $b$  to  $c$  is 5, and  $b$  to  $d$  is 3. As we cannot reach from  $b$  to  $e$  directly  $b$  to  $e$  distance is infinity. Now we get the following two sets



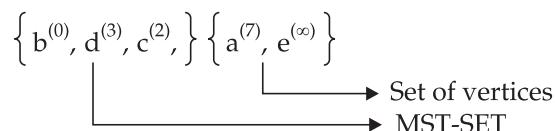
**Step 3.** We pick  $d^{(3)}$  from the set of vertices (as it has the minimum key value) and put it in the MST-SET. Now the two sets become



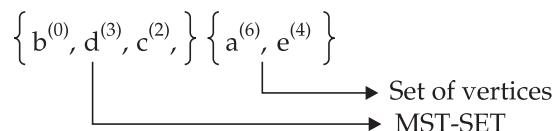
**Step 4.** We cannot move from  $d^{(3)}$  to  $a^{(7)}$  or  $e^{(\infty)}$  directly. But we can move from  $d^{(3)}$  to  $c^{(5)}$  at cost 2. So change the key value of  $c$  as 2 i.e.,  $c^{(2)}$ . Now the two sets become



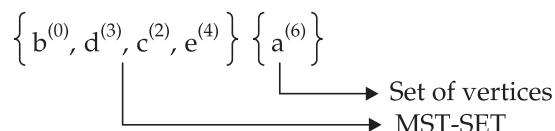
**Step 5.** Now we pick  $c^{(2)}$  from the set of vertices and put it in MST-SET. The corresponding changes are reflected as follows.



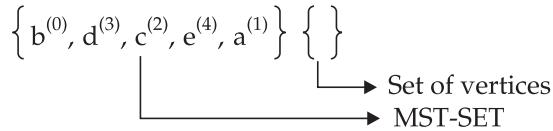
**Step 6.** We can move from  $c^{(2)}$  to  $a^{(7)}$  and  $e^{(\infty)}$  at cost 6 and 4 respectively. These are minimum from their previous key values . Now the two sets become



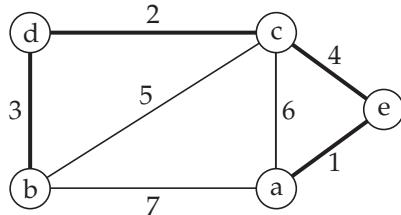
**Step 7.** As  $e^{(4)}$  holds the minimum key value in the set of vertices add it to MST-SET. The corresponding changes are reflected as shown



**Step 8.** We can move from  $e^{(4)}$  to  $a^{(6)}$  at cost 1 . So change the key value of  $a$  to 1. Move  $a^{(1)}$  to MST-SET. Now MST-SET contains all the vertices of graph G.



Therefore, the MST edges of the graph G are  $bd$ ,  $dc$ ,  $ce$  and  $ea$  and its cost =  $3 + 2 + 4 + 1 = 10$ .



**Fig. 14.89.** Showing MST edges of graph G

### Prim's Algorithm using Priority Queue

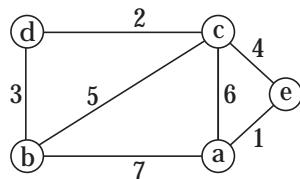
**PRIMS (G, w, S)** //w refers to the weight function  $w : E \rightarrow R^+$  S is the source vertex of graph  $G = (V, E)$

1. For all  $v \in V[G] - S$
2. do  $v.\text{key} = \infty$ ,  $v.\pi = \text{NIL}$  // key is the distance from it predecessor  $\pi$  is the predecessor.
3.  $S.\text{key} = 0$ ,  $S.\pi = \text{NIL}$
4. For all  $v \in V[G]$  do ENQUEUE (PQ, v) // entering elements into the priority queue
5. While PQ is not empty
6. do  $u = \text{DELETE}$  PQ // deleting elements from the priority queue
7. For all  $v \in \text{Adjacent}[u]$
8. do if  $v$  is in PQ and  $v.\text{key} > w(u, v)$
9. then  $v.\text{key} = w(u, v)$
10. Make  $v.\pi = u$  // Make  $u$  as the predecessor of  $v$ .

### Running Time

As each vertex deletion occurs exactly  $\log V$  times the total running time of Prim's algorithm using binary heap is  $O(E \log V)$  which is asymptotically same as for our implementation of Kruskal's algorithm. Instead of using binary heap if we use fibonacci heap then the running becomes  $O(E + V \log V)$ .

**Example 14.17** Find the MST of the graph G given in Fig. 14.89 through Prim's algorithm. Take b as the start vertex.



**Fig. 14. 90.** Graph G for Example 14.18.

**Solution : PRIMS (G, w, b)**

**Step – 1**

Priority queue (PQ) →

Vertex	a	b	c	d	e
Key	$\infty$	0	$\infty$	$\infty$	$\infty$
$\pi$	NIL	NIL	NIL	NIL	NIL

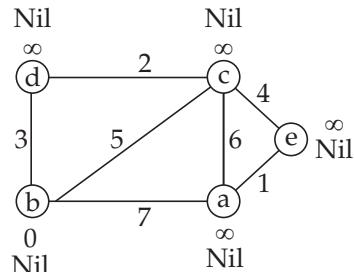


Fig. 14.91. Result of step 1.

**Step – 2**

As  $b$  carries the smallest key value delete  $b$  from PQ. Adjacent of  $b$  are  $a, d, c$

Make            a. key = 7

and a.  $\pi = b$

c. key = 5

and c.  $\pi = b$

d. key = 3

and d.  $\pi = b$

Now priority queue becomes

Vertex	a	b	c	d	e
Key	7	$\infty$	5	3	$\infty$
$\pi$	b	NIL	b	b	NIL

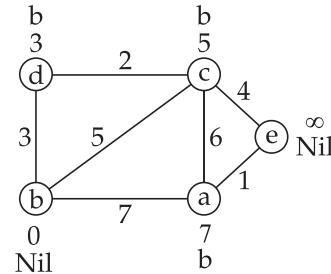


Fig. 14.92. Result of step 2.

**Step – 3**

As a carries the smallest key value, so delete  $d$  from PQ. Adjacent of  $d$  are  $b$  and  $c$ . But  $b$  is already deleted.

Make            c. key = 2

and c.  $\pi = d$

Now PQ becomes

Vertex	a	b	c	d	e
Key	7	$\infty$	2	$\infty$	$\infty$
$\pi$	b	NIL	d	b	NIL

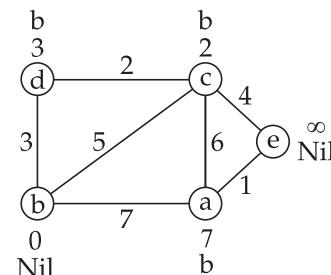


Fig. 14.93. Result of step 3.

**Step – 4**

As  $c$  carries the smallest key value, so delete  $c$  from PQ. Adjacent of  $c$  are  $a$  and  $e$  (as  $b$  and  $d$  are already deleted.)

Make

a. key = 6,

e. key = 4,

a.  $\pi = c$ e.  $\pi = c$ 

So, PQ becomes

Vertex	a	b	c	d	e
Key	1	0	2	3	4
$\pi$	e	NIL	d	b	c

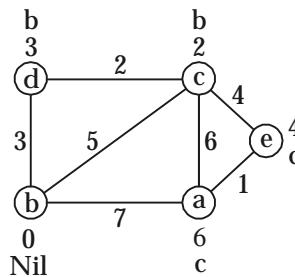


Fig. 14.94. Result of step 4.

**Step – 5**

As  $e$  carries the smallest key value, so delete  $e$  from PQ. Adjacent of  $e$  is  $a$ . Make  $a$ . key = 1,  $a \pi = e$

So PQ becomes

Vertex	a	b	c	d	e
Key	1	0	2	3	4
$\pi$	e	NIL	d	b	c

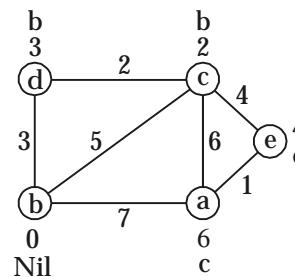


Fig. 14.95. Result of step 5.

**Step – 6**Now delete  $a$  from PQ.

Finally PQ becomes empty.

Vertex	a	b	c	d	e
Key	1	0	2	3	4
$\pi$	e	NIL	d	b	c

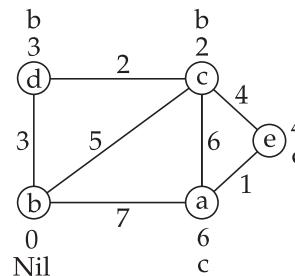


Fig. 14.96. Result of step 6.

So MST edges are :  $bd$ ,  $dc$ ,  $ce$  and  $ea$ .

Total cost of the MST is  $3 + 2 + 4 + 1 = 10$ .

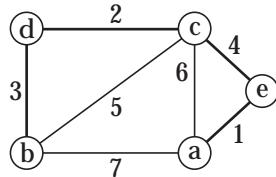


Fig. 14.97. Showing MST edges of graph G.

**Example 14.18** Run the Prim's algorithm on graph G given in Fig. 14.98 to construct a minimum spanning tree. Take  $d$  as the start vertex.

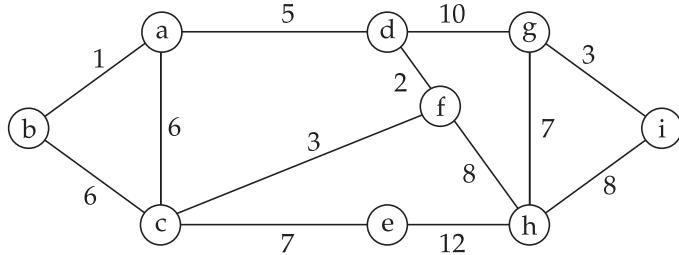


Fig. 14.98. Graph G for Example 14.17.

**Solution:** Call the procedure PRIMS ( $G, w, d$ )

**Step 1.** We first arrange the vertices in priority queue as shown :

Vertex	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$
Key	$\infty$	$\infty$	$\infty$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
$\pi$	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

**Step 2.** As  $d$  carries the smallest key value it from priority queue. Adjacent vertices of  $d$  are  $a$ ,  $f$ , and  $g$ . Make

$$\text{a. key} = 5 \quad \text{a. } \pi = d$$

$$\text{f. key} = 2 \quad \text{f. } \pi = d$$

$$\text{g. key} = 10 \quad \text{g. } \pi = d$$

Vertex	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$
Key	5	$\infty$	$\infty$	0	$\infty$	2	10	$\infty$	$\infty$
$\pi$	$d$	NIL	NIL	<del>Nil</del>	NIL	$d$	$d$	NIL	NIL

**Step 3.** As  $f$  carries the smallest key value, delete it from the priority queue. Adjacent vertices of  $f$  are  $c$ ,  $d$  and  $h$ . Mark that the vertex  $d$  is previously deleted. So we change the things and reflect those in the priority queue.

$$c. \text{key} = 3$$

$$h. \text{key} = 8$$

$$c. \pi = f$$

$$h. \pi = f$$

Vertex	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$
Key	5	$\infty$	3	0	$\infty$	2	10	8	$\infty$
$\pi$	$d$	NIL	$f$	NIL	NIL	$d$	$d$	$f$	NIL

**Step 4.** Now delete vertex  $c$  from the priority queue since it consists minimum key value. Vertices  $a$ ,  $b$ ,  $e$  and  $f$  are the adjacent vertices of vertex  $c$ . Notice that the vertex  $f$  is previously deleted.

We make the following changes in the priority queue.

$$b. \text{key} = 6$$

$$e. \text{key} = 7$$

$$b. \pi = c$$

$$e. \pi = c$$

Vertex	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$
Key	5	6	3	0	7	2	10	8	$\infty$
$\pi$	$d$	$c$	$f$	NIL	$c$	$d$	$d$	$f$	NIL

**Step 5.** Delete vertex  $a$  from the priority queue. Adjacent vertices to  $a$  are  $b$ ,  $c$  and  $d$ . Vertices  $c$  and  $d$  are previously deleted. We make the following changes in the priority queue.

$$b. \text{key} = 1$$

$$b. \pi = a$$

Vertex	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$
Key	5	1	3	0	7	2	10	8	$\infty$
$\pi$	$d$	$a$	$f$	NIL	$c$	$d$	$d$	$f$	NIL

**Step 6.** Delete vertex  $b$  from the priority queue since it carries the smallest key value among the key value of non-deleted vertex. No adjacent vertex of  $b$  remain to be accomplished. The priority queue becomes

Vertex	$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$	$i$
Key	5	1	3	0	7	2	10	8	$\infty$
$\pi$	$d$	$a$	$f$	NIL	$c$	$d$	$d$	$f$	NIL

**Step 7.** As vertex  $e$  hold the minimum key value delete it from the priority queue. Deleting vertex  $e$  we get the following priority queue.

Vertex	a	b	c	d	e	f	g	h	i
Key	5	1	3	0	7	2	10	8	$\infty$
$\pi$	d	a	f	NIL	c	d	d	f	NIL

**Step 8.** Delete  $h$  from the priority queue as it holds the smallest key value. Adjacent vertices of  $h$  are  $e, f, g$  and  $i$ . Vertices  $e$  and  $f$  are already deleted. We update the following key value and predecessor in priority queue.

$g. \text{key} = 7$	$g. \pi = h$
Vertex	
Key	
π	

Vertex	a	b	c	d	e	f	g	h	i
Key	5	1	3	0	7	2	7	8	$\infty$
$\pi$	d	a	f	NIL	c	d	h	f	NIL

**Step 9.** Delete 9 from priority queue. Make  $i. \text{key} = 3$  and  $i. \pi = g$ .

The priority queue becomes

Vertex	a	b	c	d	e	f	g	h	i
Key	5	1	3	0	7	2	7	8	3
$\pi$	d	a	f	NIL	c	d	h	f	9

**Step 10.** At last we remain with vertex  $i$ . Deleting this the priority queue became empty. So the minimum spanning tree edges are :  $df, fa, da, ab, ce, fh, gh$ , and  $gi$ . These edges are shown in Fig. 14.99.

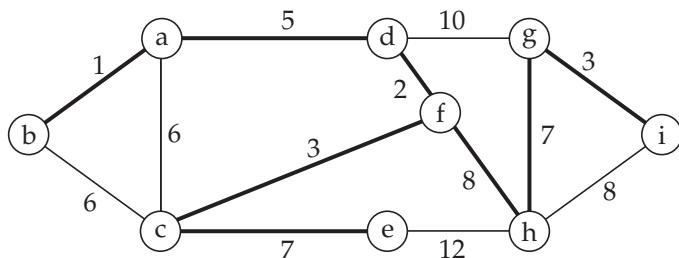


Fig. 14.99. Showing the MST edges of graph G.

## 14.10 SINGLE SOURCE SHORTEST PATHS

In this section, we consider the single-source shortest path problem : Given an edge-weighted graph  $G = (V, E)$  and a vertex  $V_0 \in V$ , find the shortest weighted path from  $V_0$  to every other vertex in  $V$ .

Why do we find the shortest path to every other vertex if we are interested only in the shortest path from say  $V_0$  to  $V_d$ ? It turns out that in order to find out the shortest path from  $V_0$  to  $V_d$  it is necessary to find the shortest path from  $V_0$  to every other vertex in  $G$  !

Clearly when we search for shortest path, we must consider all the vertices in  $V$ . If a vertex is ignored, say  $V_p$  then we will not consider any of the paths from  $V_0$  to  $V_d$  that pass through  $V_p$ . But if we fail to consider all the paths from  $V_0$  to  $V_d$  we can not be assured of finding the shortest one.

Furthermore, suppose the shortest path from  $V_0$  to  $V_d$  passes through some intermediate node  $V_i$  i.e., the shortest path is of the form of  $P = \{V_0, \dots, V_i, \dots, V_d\}$ . It must be the case that the portion of  $P$  between  $V_0$  to  $V_i$  is also shortest path from  $V_0$  to  $V_i$  because we could obtain a shorter one by replacing the portion of  $P$  between  $V_0$  and  $V_i$  by the shortest path.

Here, we will study two techniques for solving single source shortest path problems. They are :

- (i) Dijkstra algorithm
- (ii) Bellman-Ford algorithm

Dijkstra's algorithm operates on positive weight edge whereas Bellman-Ford operates on negative weight edge. Both these algorithms use relaxation procedure which is described below.

### 14.10.1 Relaxation Procedure

It is a technique of determining the shortest path of any vertex  $v$  from some vertex  $u$  inspite of the presence of many paths between  $u$  and  $v$ .

**RELAX ( $u, v, w$ ) //  $w(u, v)$**

1. if  $d[v] > d[u] + w(u, v)$
2. then  $d[v] = d[u] + w(u, v)$
3.  $\pi[v] = u$

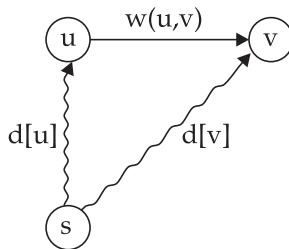


Fig. 14.100. Relaxation procedure.

Here  $d[u]$  is the shortest path so far calculated from  $s$  to  $u$ .  $d[v]$  is the shortest path so far calculated from  $s$  to  $v$ . If  $d[v] > d[u] + w(u, v)$  satisfies then we move to  $d[v] = d[u] + w(u, v)$  and then making predecessor of  $v$  is  $u$ .

#### 14.10.2 Principles of Shortest Path and Relaxation

- (i) **Triangle inequality** : For any edge  $(u, v) \in E$ , then  $\delta(s, v) \leq \delta(s, u) + w(u, v)$ .
- (ii) **Upper bound property** : We have  $d[v] \geq \delta(s, v)$  for all vertices  $v \in V$  and once  $d[v]$  achieves the value  $\delta(s, v)$  it never changes.
- (iii) **No-path property** : If there is no path from  $s$  to  $v$ , then we always have  $d[v] = \delta(s, v) = \infty$ .
- (iv) **Convergence property** : If  $s \rightsquigarrow u \rightarrow v$  is a shortest path in  $G$  for some  $u, v \in V$  and if  $d[u] = \delta(s, u)$  at any time prior to relaxing edge  $(u, v) \in E$ , then  $d[v] = \delta(s, v)$  at all times afterward.
- (v) **Path relaxation property** : If  $p = (v_0, v_1, \dots, v_k)$  is a shortest path from  $s = v_0$  to  $v_k$  and we relax the edge of  $p$  in the order  $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ , then  $d[v_k] = \delta(s, v_k)$ .
- (vi) **Predecessor-subgraph property** : Once  $d[v] = \delta(s, v)$  for all  $v \in V$  the predecessor sub-graph is a shortest paths tree rooted at  $s$ .

### 14.11 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is a Greedy algorithm, which is used to solve the single-source shortest-paths problem on a non-negative weighted directed graph  $G = (V, E)$ . Algorithm starts at the source vertex  $S$ ; it grows a tree that ultimately spans all vertices that are reachable from  $S$ . Vertices are added to the tree in terms of distance i.e., first  $S$ , then the vertex close to  $S$ , then next closest and so on.

So overall it is just like Prim's algorithm and we have to follow same procedure to solve this problem.

#### 14.11.1 Dijkstra's Algorithm Steps

```

DIJKSTRA (G, w, S) // w : E → R+, S is the source vertex
1. For all vertex v ∈ V[G] – S
2. do d[v] = ∞, π[v] = NIL // d is the distance from source and π is the predecessor.
3. d[S] = 0, π[S] = NIL
4. For all v ∈ V[G] do ENQUEUE (PQ, V) // ENQUEUE is the process through which we enter the elements
   into the priority queue (PQ)
5. While (PQ ≠ ∅)
6. do u = DEQUEUE (PQ) // DEQUEUE is the operation through which we delete elements from priority
   queue. Highest priority element (element having smallest d value) is deleted first.
7. For all v ∈ Adjacent [u]
8. do RELAX (u, v, x) // Call RELAX procedure given in section 14.10.1.

```

### Running Time

The running time of Dijkstra's algorithm is  $O(E \log V)$  where  $E$  denotes the number of edges in the given graph  $G$  and  $V$  denotes the number of vertices.

**Example 14.19** Find the shortest path from the source vertex 'a' to all the remaining vertices in the weighted graph given in Fig. 14.101 and find its running time?

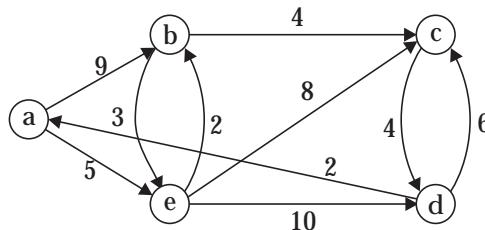


Fig. 14.101. Weighted graph  $G$  for example 14.19.

**Solution:** Dijkstra ( $G, w$  and  $S$ )

**Step – 1**

Priority queue (PQ) →					
Vertex	a	b	c	d	e
d	Nil	∞	∞	∞	∞
π	NIL	NIL	NIL	NIL	NIL

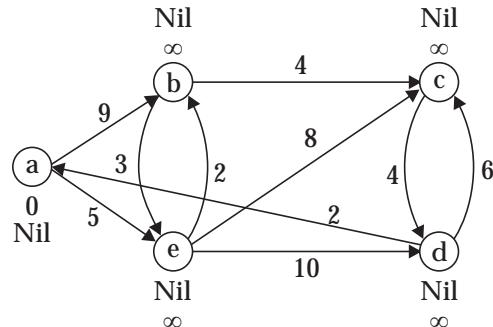


Fig. 14.102. Result of step 1.

**Step – 2**

As  $a$  carries the smallest  $d$  value delete  $a$  from PQ. Adjacent of  $a$  are  $b, e$ .

Make  $d[b] = 9$  and  $\pi[b] = a$

$d[e] = 5$  and  $\pi[e] = a$

Now the priority queue becomes

Vertex	a	b	c	d	e
d	Nil	9	∞	∞	5
π	NIL	a	NIL	NIL	a

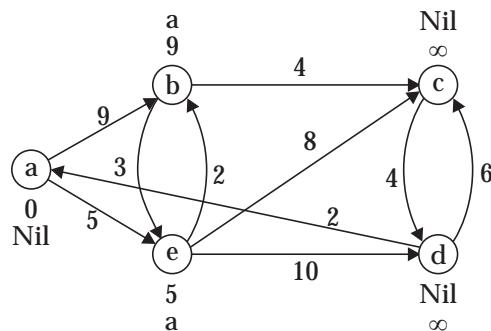


Fig. 14.103. Result of step 2.

**Step - 3**

As  $e$  carries the smallest  $d$  value delete  $e$  from PQ. Adjacent of  $e$  are  $b$ ,  $c$  and  $d$ .

Make  $d[b] = 7$  and  $\pi[b] = e$  (As  $2 < 9$ )

$d[c] = 13$  and  $\pi[c] = e$

$d[d] = 15$  and  $\pi[d] = e$

Priority queue becomes

Vertex	a	b	c	d	e
d	0	7	13	15	5
$\pi$	NIL	e	e	e	a

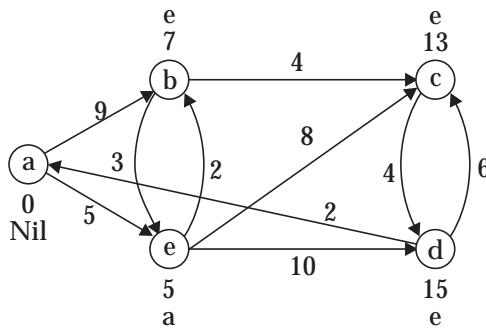


Fig. 14.104. Result of step 3.

**Step - 4**

From the above priority queue table, we can clearly see vertex  $b$  carries the smallest  $d$  value. So delete it, from priority queue. Adjacent of  $b$  are  $c$  and  $e$ . But  $e$  is already deleted.

Make  $d[e] = 11$  and  $\pi[c] = b$

Priority queue becomes

Vertex	a	b	c	d	e
d	0	7	11	15	5
$\pi$	NIL	e	b	e	a

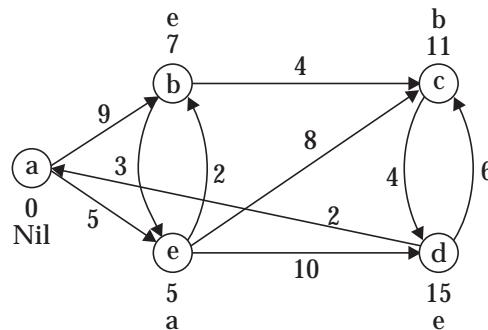


Fig. 14.105. Result of step 4.

**Step - 5**

As  $c$  carries the smallest  $d$  value delete  $c$  from PQ. Adjacent of  $c$  is  $d$ . As distance of  $d$  from start vertex  $a$  through  $e$  = distance of  $d$  through  $c$  = 15. So no change takes place in predecessor field or  $d$  according to Relaxation procedure.

Priority queue becomes →

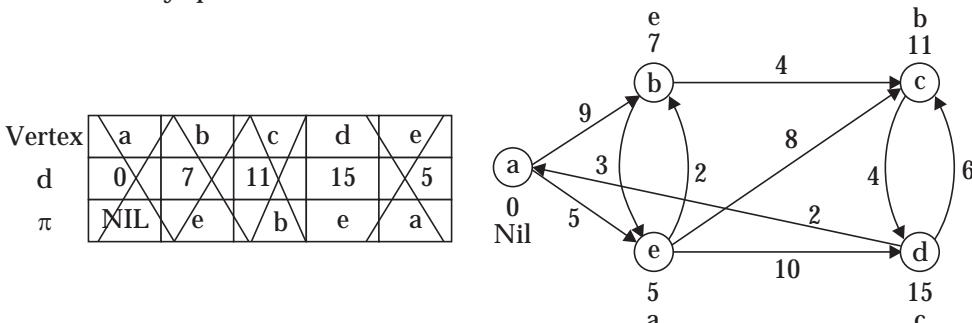


Fig. 14.106. Result of step 5.

#### Step – 6

In priority queue only one vertex remain i.e., d. So delete vertex d. Finally priority queue becomes empty.

Vertex	a	b	c	d	e
d	0	7	11	15	5
π	NIL	e	b	e	a

Shortest paths from source vertex 'a' to all remaining vertices are

$$a - b = 7, a - c = 11, a - d = 15, a - e = 5$$

Running time =  $O(E \log V)$  (where  $E \rightarrow$  no edges in graph G and  $V \rightarrow$  no of vertices in graph G)

## 14.12 THE BELLMAN-FORD ALGORITHM

Let we are given a graph  $G = (V, E)$  with numeric edge weights,  $w(u, v)$ . Like Dijkstra's algorithm, the Bellman-Ford algorithm is based on performing repeated relaxations. (Recall that relaxation updates shortest path information along a single edge.) Dijkstra's algorithm was based on the idea of organizing the relaxations in best possible manner, namely in increasing order of distance. Once relaxation is applied to an edge, it need never to be relaxed again. This trick doesn't seem to work when dealing with graphs will negative edge weights. To solve this problem, we shall present Bellman-Ford algorithm. It simply applies a relaxation to every edge in the graph, and repeat this  $V - 1$  times.

**Note :** Bellman-Ford algorithm does not applicable to the graphs which contains negative cycles.

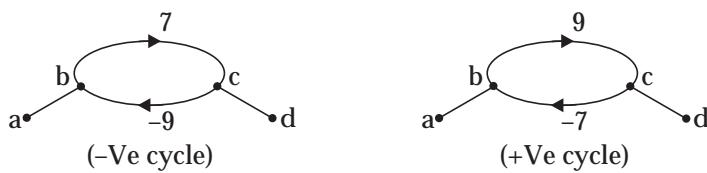


Fig. 14.107. Negative and positive cycles.

#### 14.12.1 Bellman-Ford Algorithm Steps

BELLMAN-FORD ( $G, w, S // w : E \rightarrow R$  and  $S$  is the source vertex).

1. For all vertex  $v \in V[G] - S$
2. do  $d[v] = \infty, \pi[v] = \text{NIL} // d$  is the distance from the source and  $\pi$  is the predecessor.
3.  $d[S] = 0, \pi[S] = \text{NIL}$
4. for  $i = 1$  to  $|G.V| - 1$
5. For each edge  $(u, v) \in G.E$
6. RELAX  $(u, v, w)$
7. For each edge  $(u, v) \in G.E$
8. if  $d[v] > d[u] + w(u, v)$
9. return false
10. else return true.

#### Running Time

Bellman-Ford algorithm is slower than Dijkstra's algorithm, running in  $O(VE)$ , since initialization in lines 1 to 3 takes  $O(V)$  time, each of the  $|V| - 1$  passes over the edges in lines 4 to 6 takes  $O(E)$  time and the for loop of lines 7 – 9 takes  $O(E)$  time.

**Example 14.20** Find the shortest path from the source vertex 'a' to vertex 'd' in the given weighted graph  $G$  and find the complexity of this operation to perform ?

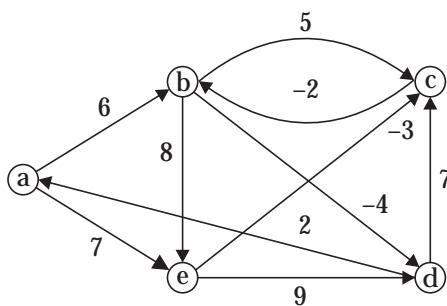


Fig. 14.108. Graph  $G$  for Example 14.20

#### Solution:

##### Step – 1

Initialize all nodes of graph  $G$  except the start node 'a' has distance  $(d) = \infty$  and set predecessor  $(\pi) = \text{NIL}$ . Make  $d[a] = 0$  and  $\pi[a] = \text{NIL}$

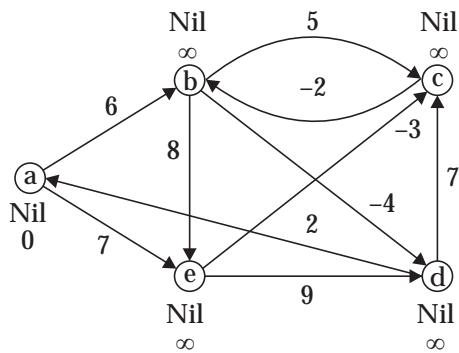


Fig. 14.109. Result of step 1.

**Step – 2**

Apply RELAX ( $a, b, 6$ ) and RELAX ( $a, e, 7$ )

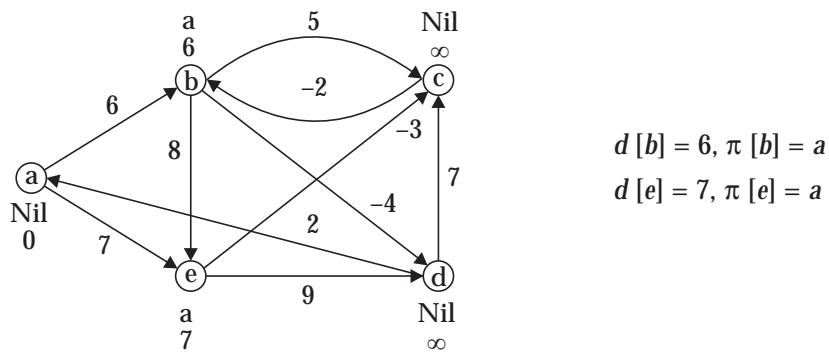


Fig. 14.110. Result of step 2.

**Step – 3**

RELAX ( $b, c, 5$ )

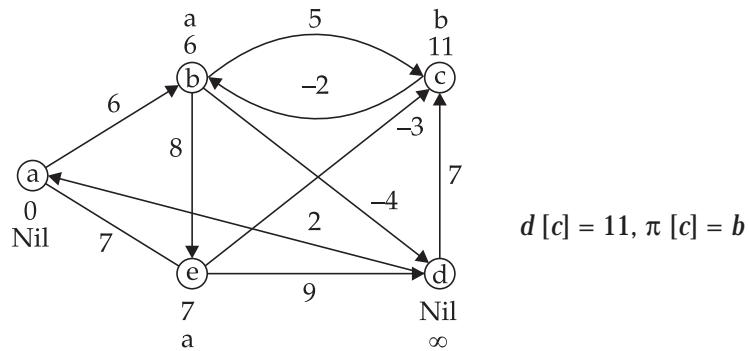


Fig. 14.111. Result of step 3.

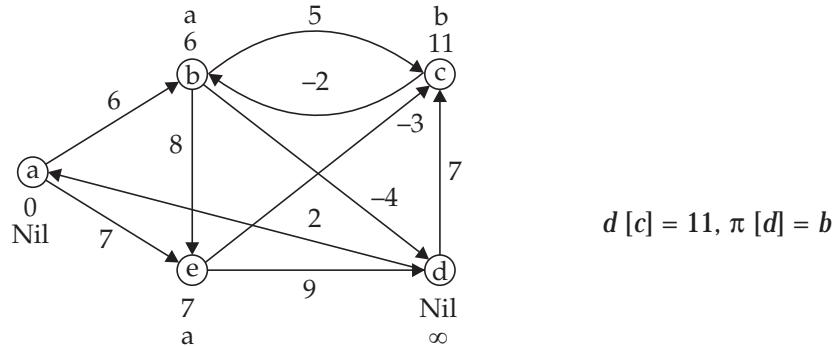
**Step – 4**RELAX ( $b, d, -4$ )

Fig. 14.112. Result of step 4.

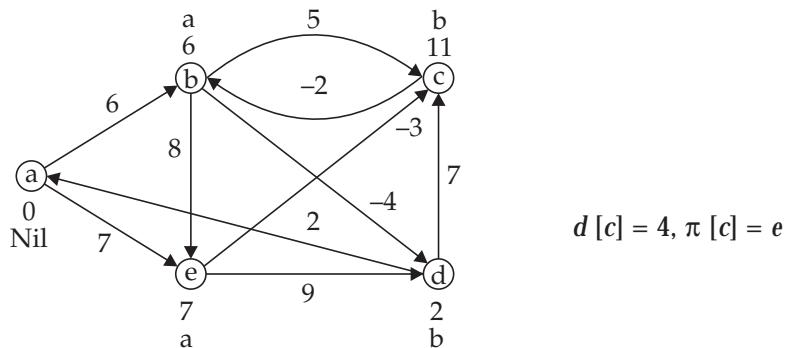
**Step – 5**RELAX ( $e, c, -3$ )

Fig. 14.113. Result of step 6.

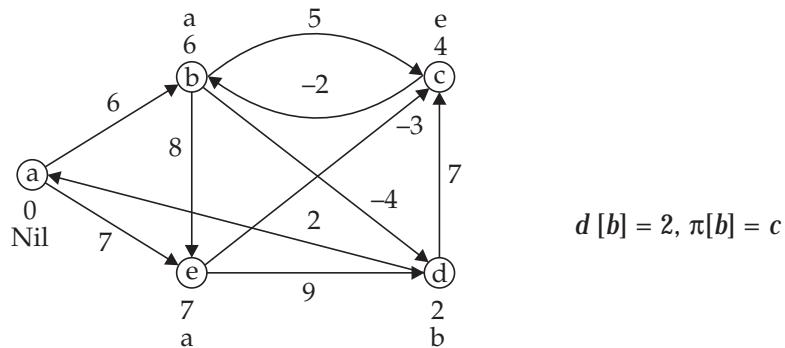
**Step – 6**RELAX ( $c, b, -2$ )

Fig. 14.114. Result of step 7.

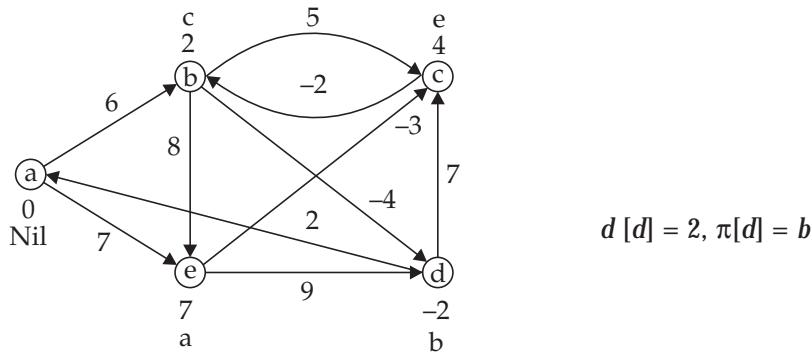
**Step - 7**RELAX ( $b, d, -4$ )

Fig. 14.115. Result of step 8.

Shortest path from  $a$  to  $d$  is  $7 + (-3) + (-2) + (-4) = -2$ .

Time complexity of the entire process =  $O(VE)$ , where  $V$  is the number of vertices and  $E$  is the number of edges.

### 14.13 ALL-PAIRS SHORTEST PATHS

We consider the generalization of the shortest path problem, to computer the shortest paths between all pairs of vertices. Let  $G = (V, E)$  be a directed graph with edge weights. If  $(u, v)$ , is an edge of  $G$ , then the weight of this edge is denoted  $w(u, v)$ . Recall that the cost of a path is the sum of edge weights along the path. The distance between two vertices  $\delta(u, v)$  is the cost of minimum cost path between them. We will allow  $G$  to have negative cost edges, but we will not allow  $G$  to have any negative cost cycles.

We consider the problem of determining the cost of the shortest path between all pairs of vertices in a weighted directed graph. We will present a  $\Theta(n^3)$  algorithm, called the Floyed-Warshall algorithm. This algorithm is based on dynamic programming.

For this algorithm, we will assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. Although adjacency lists are generally more efficient for sparse graphs, storing all the inter-vertex distances will require  $\Omega(n^2)$  storages, so the savings is not justified here.

**Input Format**

The input is an  $n \times n$  matrix  $w$  of edge weights, which are based on the edge weights in the digraph. We let  $w_{ij}$  denote the entry in row  $i$  and column  $j$  of  $w$ .

$$w_{ij} = \begin{cases} 0 & \text{if } i=j \\ w(i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \\ +\infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

Setting  $w_{ij} = \infty$  if there is no edge, intuitively means that there is no direct link between these two nodes, and hence the direct cost is infinite. The reason for setting  $w_{ij} = 0$  is that there is always a trivial path of length 0 (using no edges) from any vertex to itself.

**Note:** In digraph, it is possible to have self loop edges, and so  $w(i, j)$  may generally be non-zero. It cannot be negative, since we assume that there are no negative cost cycles, and if it is positive, there is no point in using it as part of any shortest path.

### Output Format

The output will be an  $n \times n$  distance matrix  $D = d_{ij}$  where  $d_{ij} = \delta(i, j)$ , the shortest path cost from vertex  $i$  to  $j$ . Recovering the shortest paths will also be an issue. To help us do this, we will also computer an auxiliary matrix  $\text{mid}[i, j]$ . The value of  $\text{mid}[i, j]$  will be a vertex that is somewhere along the shortest path from  $i$  to  $j$ . If the shortest path travels directly from  $i$  to  $j$  without passing through any other vertices, then  $\text{mid}[i, j]$  will be set to null. These intermediate values behave somewhat like the predecessor pointers in Dijkstra's algorithm, in order to reconstruct the final shortest path in  $\Theta(n)$  time.

---

## 14.14 FLOYD-WARSHALL ALGORITHM

---

The Floyd-Warshall algorithm dates back to the early 60's. Warshall was interested in the weaker question of reachability : determine for each pair of vertices  $u$  and  $v$ , whether  $u$  can reach  $v$ . Floyd realized that the same technique could be used to computer shortest paths will only minor variations.

As with any DP algorithm, the key is reducing a large problem to smaller problems. A natural way of doing this is by limiting the number of edges of the path but it turns out that this does not lead to faster algorithm (but is an approach worthy of consideration). The main feature of Floyd-Warshall algorithm is to find a best formulation for shortest path subproblem. Rather than limiting the number of edges on the path, they instead limit the set of vertices through which the path is allowed to pass. In particular, for a path  $p = \langle v_1, v_2, \dots, v_l \rangle$  we say that the vertices  $v_2, v_3, \dots, v_{l-1}$  are the intermediate vertices of this path. Note that, a path consisting of a single edge has no intermediate vertices.

**Formulation** – Define  $d_{ij}^{(k)}$  to be the shortest path from  $i$  to  $j$  such that any intermediate vertices on the path are chosen from set  $\{1, 2, \dots, k\}$ .

In other words, we consider a path from  $i$  to  $j$  which either consists of the single edge  $(i, j)$ , or it visits some intermediate vertices along the way, but these intermediate can only be chosen from among  $\{1, 2, \dots, k\}$ . The path is free to visit any subset of these vertices, and to do so in any order. For example, the digraph shown in Fig. 14.116 (a) notice how the value of  $d_{5,6}^{(k)}$  changes as  $k$  varies.

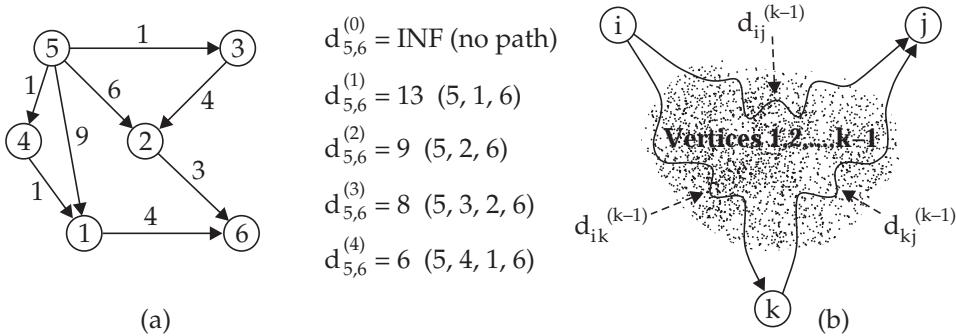


Fig. 14.116. Limiting intermediate vertices.

Here  $d_{5,6}^{(3)}$  can go through any combination of the intermediate vertices {1, 2, 3} of which (5, 3, 2, 6) has the lowest cost of 8.

#### 14.14.1 Floyd-Warshall Update Rule

How do we compute  $d_{ij}^{(k)}$  assuming that we have already computed the previous matrix  $d(k-1)$ ? There are two basic cases, depending on the ways that we might get from vertex  $i$  to  $j$ , assuming that intermediate vertices are chosen from {1, 2, ...,  $k$ }:

**Don't go through  $k$  at all:** Then the shortest path from  $i$  to  $j$  uses only intermediate vertices {1, ...,  $k-1$ } and hence the length of the shortest path is  $d_{ij}^{(k-1)}$ .

**Do go through  $k$ :** First observe that a shortest path does not pass through the same vertex twice, so we can assume that we pass through  $k$  exactly once. (The assumption that there are no negative cost cycles is being used here.) That is, we go from  $i$  to  $k$ , and then from  $k$  to  $j$ . In order for overall path to be as short as possible we should take the shortest path from  $i$  to  $k$ , and the shortest path from  $k$  to  $j$ . Since of these paths uses intermediate vertices only in {1, 2, ...,  $k-1$ }, the length of the path is  $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ .

This suggest the following recursive rule (the DP formulation) for computing  $d^{(k)}$ , which is illustrated in Fig 14.116 (b)

$$\begin{aligned} d_{ij}^{(0)} &= w_{ij} \\ d_{ij}^{(k)} &= \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \text{ for } k \geq 1. \end{aligned}$$

The final answer is  $d_{ij}^{(n)}$  because this allows all possible vertices as intermediate vertices.

We could write a recursive algorithm to compute  $d_{ij}^{(k)}$ , but this will be prohibitively slow, because the same value may be reevaluated many times. Instead we computer it by storing the values in a table, and looking the values up as we need them. Here is the complete algorithm.

#### 14.14.2 Floyd Warshall Algorithm Steps

```

FLOYD – WARSHALL (int n, int w[1, .....n, 1 .....n]) // n is the number of vertices in given graph
1. array d[1 .....n, 1 .....n]
2. for i = 1 to n do      // initialize
3. for j = 1 to n
4.     d[i, j] = w[i,j]
5.     mid[i, j] = null
6. for k = 1 to n          // use intermediates {1, .....k}
7.     for i = 1 to n do    // ... form i
8.         for j = 1 to n do // ... to j
9.             if ((d[i, k] + d[k, j]) < d[i, j])
10.                d[i, j] = d[i, k] + d[k, j] // new shorter path length
11.                mid [i, j] = k // new path is through k
12. return d(n)           // matrix of distances

```

#### Running Time

Clearly the algorithm's running time is  $O(n^3)$ . The space used by the algorithm is  $O(n^2)$ .

#### 14.14.3 Finding the Predecessor Matrix

We can compute the predecessor matrix  $\pi$  while the algorithm computes the matrices  $d^{(n)}$ . So we compute a sequence of matrices  $\pi^{(0)}, \pi^{(1)}, \dots, \pi^{(n)}$ , where  $\pi = \pi^{(n)}$  and we define  $\pi_{ij}^{(k)}$  as the predecessor of vertex  $j$  on a shortest path from vertex with all intermediate vertices in the set  $\{1, 2, 3, \dots, k\}$ .

For  $k \geq 1$ , there is a path exist between  $i$  and  $j$  through an mediator  $k$  i.e.,  $i \rightsquigarrow k \rightsquigarrow j$  where  $k \neq j$ , then the predecessor of  $j$  we choose is the same as the predecessor of  $j$  we choose on a shortest path from  $k$  with all intermediate vertices in the set  $\{1, 2, \dots, k-1\}$ . Alternatively, we choose the same predecessor of  $j$  that we chose on a shortest path from  $i$  with all intermediate vertices in the set  $\{1, 2, \dots, k-1\}$ .

$$\text{Hence } \pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

**Note:** We compute  $\pi_{ij}^{(k)}$  only when  $k \geq 1$  (not for  $k = 0$ ). When  $k = 0$ , we consider there is no intermediate between the two vertices  $i$  and  $j$ .

**Example 14.21** Run the Floyd-Warshall algorithm on the given weighted, directed graph to find all pairs shortest paths and find the predecessor matrix at each stage.

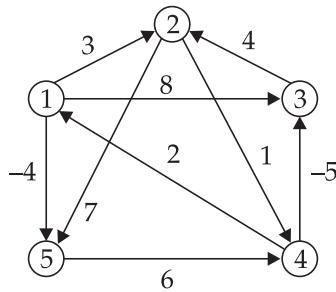


Fig. 14.117. Graph for Example 14.20.

**Solution:**

**Step – 1**

$$d^{(0)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & \infty & -4 \\ 2 & \infty & 0 & \infty & 1 & 7 \\ 3 & \infty & 4 & 0 & \infty & \infty \\ 4 & 2 & \infty & -5 & 0 & \infty \\ 5 & \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$\pi^{(0)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ 3 & \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ 5 & \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

**Step – 2**

For calculating  $d^{(1)}$ , trick is fix 1st row and 1st column of  $d^{(0)}$  matrix and then start calculation. Corresponding changes should be reflected in  $d^{(1)}$  and  $\pi^{(1)}$  matrices.

$$d^{(1)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & \infty & -4 \\ 2 & \infty & 0 & \infty & 1 & 7 \\ 3 & \infty & 4 & 0 & \infty & \infty \\ 4 & 2 & 5 & -5 & 0 & -2 \\ 5 & \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$\pi^{(1)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ 3 & \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 4 & 1 & 4 & \text{NIL} & 1 \\ 5 & \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

**Step - 3**

For calculating  $d^{(2)}$ , fix 2nd row and 2nd column of  $d^{(1)}$  matrix and start calculation. Corresponding changes should be done in  $d^{(2)}$  and  $\pi^{(2)}$  matrices.

$$d^{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & 4 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & \infty & 0 & \infty & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & \infty & 4 & 0 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 5 & -5 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & \infty & \infty & \infty & 6 \end{bmatrix}$$

$$\pi^{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & \text{NIL} & 1 & 1 & 2 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 2 \\ 3 & \text{NIL} & 3 & \text{NIL} & 2 \\ 4 & 4 & 1 & 4 & \text{NIL} \\ 5 & \text{NIL} & \text{NIL} & \text{NIL} & 5 \end{bmatrix}$$

**Step - 4**

For calculating  $d^{(3)}$ , fix 3rd row and 3rd column of  $d^{(2)}$  matrix and then start calculation. Corresponding changes should be made in  $d^{(3)}$  and  $\pi^{(3)}$  matrices.

$$d^{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & 4 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & \infty & 0 & \infty & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & \infty & 4 & 0 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & -1 & -5 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & \infty & \infty & \infty & 6 \end{bmatrix}$$

$$\pi^{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & \text{NIL} & 1 & 1 & 2 \\ 2 & \text{NIL} & \text{NIL} & \text{NIL} & 2 \\ 3 & \text{NIL} & 3 & \text{NIL} & 2 \\ 4 & 4 & 3 & 4 & \text{NIL} \\ 5 & \text{NIL} & \text{NIL} & \text{NIL} & 5 \end{bmatrix}$$

**Step - 5**

For calculating  $d^{(4)}$  fix the 4th row and 4th column of  $d^{(3)}$  matrix and then start calculation. Corresponding changes should be made in  $d^{(4)}$  and  $\pi^{(4)}$  matrices.

$$d^{(4)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & -1 & 4 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 0 & -4 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 7 & 4 & 0 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & -1 & -5 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 8 & 5 & 1 & 6 \end{bmatrix}$$

$$\pi^{(4)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & \text{NIL} & 1 & 4 & 2 \\ 2 & \text{NIL} & 1 & 4 & 2 \\ 3 & 4 & 3 & \text{NIL} & 2 \\ 4 & 4 & 3 & 4 & \text{NIL} \\ 5 & 4 & 4 & 4 & 5 \end{bmatrix}$$

**Step-6**

For calculating  $d^{(5)}$  fix the 5th row and 5th column of  $d^{(4)}$  matrix and then start calculation. Corresponding changes should be marked in  $d^{(5)}$  and  $\pi^{(5)}$ .

$$d^{(5)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & -3 & 2 & -4 \\ 2 & 3 & 0 & -4 & 1 & -1 \\ 3 & 7 & 4 & 0 & 5 & 3 \\ 4 & 2 & -1 & -5 & 0 & -2 \\ 5 & 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$\pi^{(5)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & \text{NIL} & 5 & 5 & 5 & 1 \\ 2 & 4 & \text{NIL} & 4 & 2 & 4 \\ 3 & 4 & 3 & \text{NIL} & 2 & 4 \\ 4 & 4 & 3 & 4 & \text{NIL} & 1 \\ 5 & 4 & 4 & 4 & 5 & \text{NIL} \end{bmatrix}$$

**Example 14.22** Run the Floyd-Warshall algorithm on the weighted, directed graph given in Fig. 14.118 to find all pairs shortest paths and the predecessor matrix at each stage.

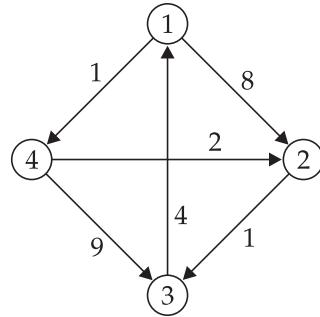


Fig. 14.118. Graph for Example 14.21.

**Solution :**

**Step-1**

$$d^{(0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & \infty & 0 & \infty \\ 4 & \infty & 2 & 9 & 0 \end{bmatrix}$$

$$\pi^{(0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & \text{NIL} & 1 & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ 3 & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & 4 & \text{NIL} \end{bmatrix}$$

**Step-2**

Fix the 1st row 1st column of  $d^{(0)}$  matrix and start calculation. Corresponding changes should be reflected in  $d^{(1)}$  and  $\pi^{(1)}$  matrices.

$$d^{(1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 8 & 2 & 9 & 0 \end{bmatrix}$$

$$\pi^{(1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & \text{NIL} & 1 & \text{NIL} & 1 \\ 2 & \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ 3 & 3 & 1 & \text{NIL} & 1 \\ 4 & \text{NIL} & 4 & 4 & \text{NIL} \end{bmatrix}$$

**Step-3**

Fix the 2nd row and 2nd column of  $d^{(1)}$  matrix and start calculation. Corresponding changes should be reflected in  $d^{(2)}$  and  $\pi^{(2)}$  matrices.

$$d^{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{bmatrix}$$

$$\pi^{(2)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & \text{NIL} & 1 & 2 & 1 \\ 2 & \text{NIL} & \text{NIL} & 2 & \text{NIL} \\ 3 & 3 & 1 & \text{NIL} & 1 \\ 4 & \text{NIL} & 4 & 2 & \text{NIL} \end{bmatrix}$$

**Step-4**

Fix the 3rd row and 3rd column of  $d^{(2)}$  matrix and start calculation. Corresponding changes should be reflected in  $d^{(3)}$  and  $\pi^{(3)}$  matrices.

$$d^{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{bmatrix}$$

$$\pi^{(3)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & \text{NIL} & 1 & 2 & 1 \\ 2 & 3 & \text{NIL} & 2 & \text{NIL} \\ 3 & 3 & 1 & \text{NIL} & 1 \\ 4 & 3 & 4 & 2 & \text{NIL} \end{bmatrix}$$

**Step -5**

Fix the 4th row and 4th column of  $d^{(3)}$  matrix and start calculation. Corresponding changes should be reflected in  $d^{(4)}$  and  $\pi^{(4)}$  matrices.

$$d^{(4)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{bmatrix}$$

$$\pi^{(4)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & \text{NIL} & 4 & 4 & 1 \\ 2 & 3 & \text{NIL} & 2 & \text{NIL} \\ 3 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 2 & \text{NIL} \end{bmatrix}$$

## CHAPTER NOTES

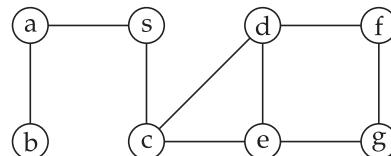
---

- A graph is said to be complete if every node  $u$  in  $G$  adjacent to every other node  $V$  in  $G$ . Clearly such a graph is connected. A complete graph with  $n$  nodes will have  $n(n - 1)/2$  edges.
- A connected graph without any cycle is called tree graph or free graph.
- A graph is said to be labeled if its edges are assigned data. In particular,  $G$  is said to be weighted if each edge  $e$  in  $G$  is assigned with a non-negative numerical value  $w(e)$  called weight or length of  $e$ .
- We can represent a graph in two ways: (i) Linked list representation and (ii) Adjacency matrix representation.
- We have seen searching a graph can be done by either using breadth first search or depth-first search. There are two applications of depth-first search: topologically sorting a directed graph into its strongly connected components.
- Minimum weight spanning tree of a graph is the least weight way of connecting all of the vertices together when each edge has an associated weight. The algorithm for computing minimum spanning tree serve as good example of greedy algorithms.
- Single source shortest path problem determiners how to find the shortest path from a given source vertex to all other vertices. There are two algorithms to handle it. One is Dijkstra's algorithm (meant for non-negative weighted directed graph) and another one is the Bellman-Ford algorithm (designed for negative weights but not for negative cycle.)
- One of the major application of Dijkstra's algorithm in the field of networking is that it is used in routing protocols like OSPF (Open Source Shortest Path First) to find the shortest root for data transmission.

## EXERCISES

---

1. A graph can be represented using adjacency matrix and adjacency list in memory. Discuss about the time complexity and space complexity of BFS algorithm with reference to the two said data structure.
2. Write the algorithm to perform Breadth First Search (BFS) and find its complexity. Apply BFS on the undirected graph given in Fig. 14.119 taking 's' as the start vertex.



**Fig. 14.119.** Undirected graph for 14.2.

3. Explain the properties of strongly connected components.
4. Write a Greedy algorithm to generate shortest path.
5. Prove that Kruskal's algorithm is correct.
6. Explain single source shortest path problem using Bellman-Ford algorithm and find its complexity.
7. Apply Dijkstra's algorithm to the graph given in Fig. 14.120. Find its time complexity.

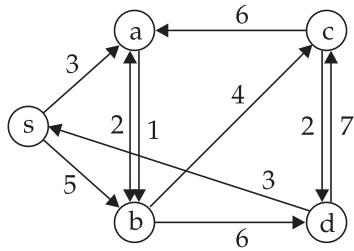


Fig. 14.120. Graph for 14.7.

8. Write the Kruskal's algorithm and find the MST for given in Fig.14.121 graph.

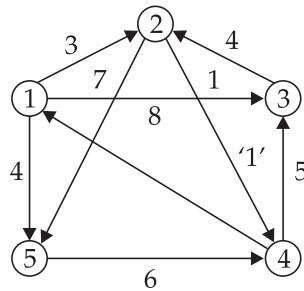


Fig.14.121. Graph for 14.8.

9. Run the Floyd - Warshall algorithm on the weighted, directed graph shown in Fig. 14.122 to find the all pairs shortest paths.

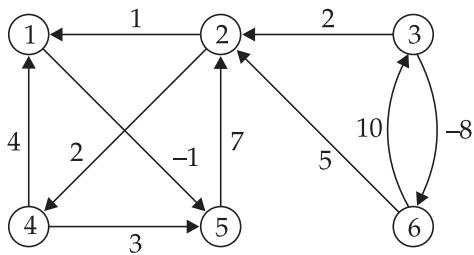
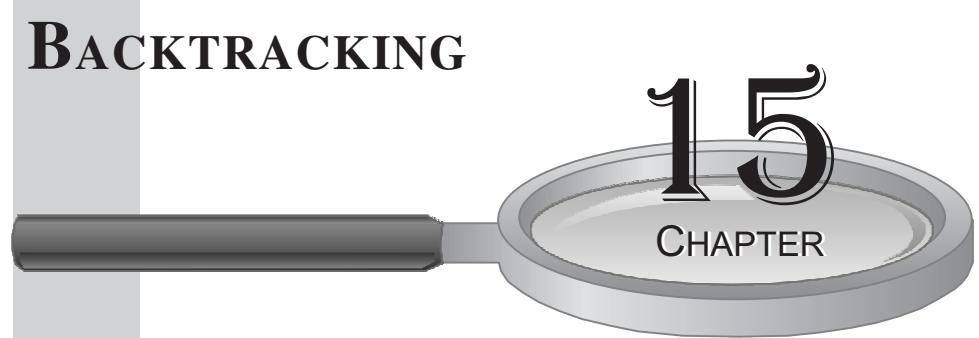


Fig. 14.122. Graph for 14.9.

# BACKTRACKING



15

CHAPTER

## OBJECTIVES OF LEARNING

After going through this chapter, the reader would be able to understand:

- Recursive algorithm strategy
- Solving 4 queens problem
- Recursion tree / State-space tree
- Subset sum problem
- Graph coloring problem

# Chapter 15 BACKTRACKING

## INSIDE THIS CHAPTER

- |                         |                             |
|-------------------------|-----------------------------|
| 15.1 Introduction       | 15.2 N-queens Problem       |
| 15.3 Subset Sum Problem | 15.4 Graph Coloring Problem |

## 15.1 INTRODUCTION

In this chapter, we discuss a recursive algorithm strategy called **backtracking**. It is an important tool for solving queen problem, puzzle problem, Sudoku etc. Moreover, backtracking is the most convenient technique for parsing, knapsack problem and other combinatorial optimization problems.

The idea about backtracking technique is very much simple which can be illustrated as follows. We have a recursive algorithm that tries to build a solution part by part, and when it gets into a dead end, then it has either built a solution or it needs to go back (backtrack) and try picking different values for some of the parts. We check whether the solution we have built is a valid solution only at the deepest level of recursion – when we have all parts picked out.

## 15.2 N-QUEENS PROBLEM

The typical **N-queens problem** was first proposed by German chess enthusiast Max Bezzel in the year of 1848 for the standard  $8 \times 8$  board. This problem is also known as eight queens puzzle. In other words, it is the problem of placing eight chess queens on an  $8 \times 8$  chess board so that no two queens attack each other. **The eight queens puzzle** is an example of the more general N queens problem of placing  $N$  queens on an  $N \times N$  chessboard, so that no two queens attack each other by being on the same row or same column or on same diagonal.

If there exists any solution to N queens problem, then there exists one queen in each row. Therefore, we will represent our possible solution using an array  $Q[1.....n]$ , where  $Q[i]$  indicates which square in row  $i$  contains a queen, or zero if no queen has been placed in row  $i$ . To get a solution, we need to put queens on the board row by row, starting from the top.

We can give a partial solution for the array  $Q[1.....n]$  whose first  $r - 1$  entries are positive and whose last  $n - r + 1$  entries are all zeros, for some integer  $r$ .

To get the solution of the N-queens problem we use the following recursive algorithm which recursively enumerates all complete  $n$ -queens solution that are consistent with a given partial solution.

```
RECURSIVE N-QUEENS (Q[1.....n], r)
1. If r = n + 1 // r is the first empty row
2. print Q
3. else
4. For j = 1 to n
5. legal = TRUE
6. For i = 1 to r - 1
7. If (Q[i] = j) or (Q[i] = j + r - i) or (Q[i] = j - r + i)
8. legal = FALSE
9. if legal
10. Q[r] = j
11. RECURSIVE N QUEENS (Q[1.....n], r + 1)
```

There are various unique solution present for  $8 \times 8$  chessboard. One unique solution can be given as [3, 6, 8, 1, 4, 7, 5, 2]. This is represented in Fig. 15.1

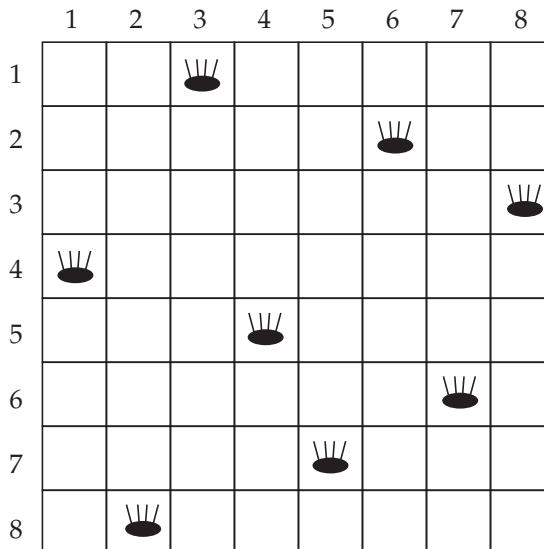


Fig. 15.1. A solution to 8 queens problem.

### 15.2.1 Representation Through Recursion Tree

Recursion tree<sup>[\*]</sup> is the best possible way of representing backtracking algorithms. The root of the recursion tree corresponds to the originity of the algorithm ; whereas edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the N-queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that can not be extended, because there exists a queen on every row, or because every position in the next empty row is in the same row, column or diagonal as an existing queen.

**Example 15.1** Solve the 4-queens problem using backtracking.

**Solution:** We solve the 4-queens problem by using recursion tree as shown in Fig. 15.2.

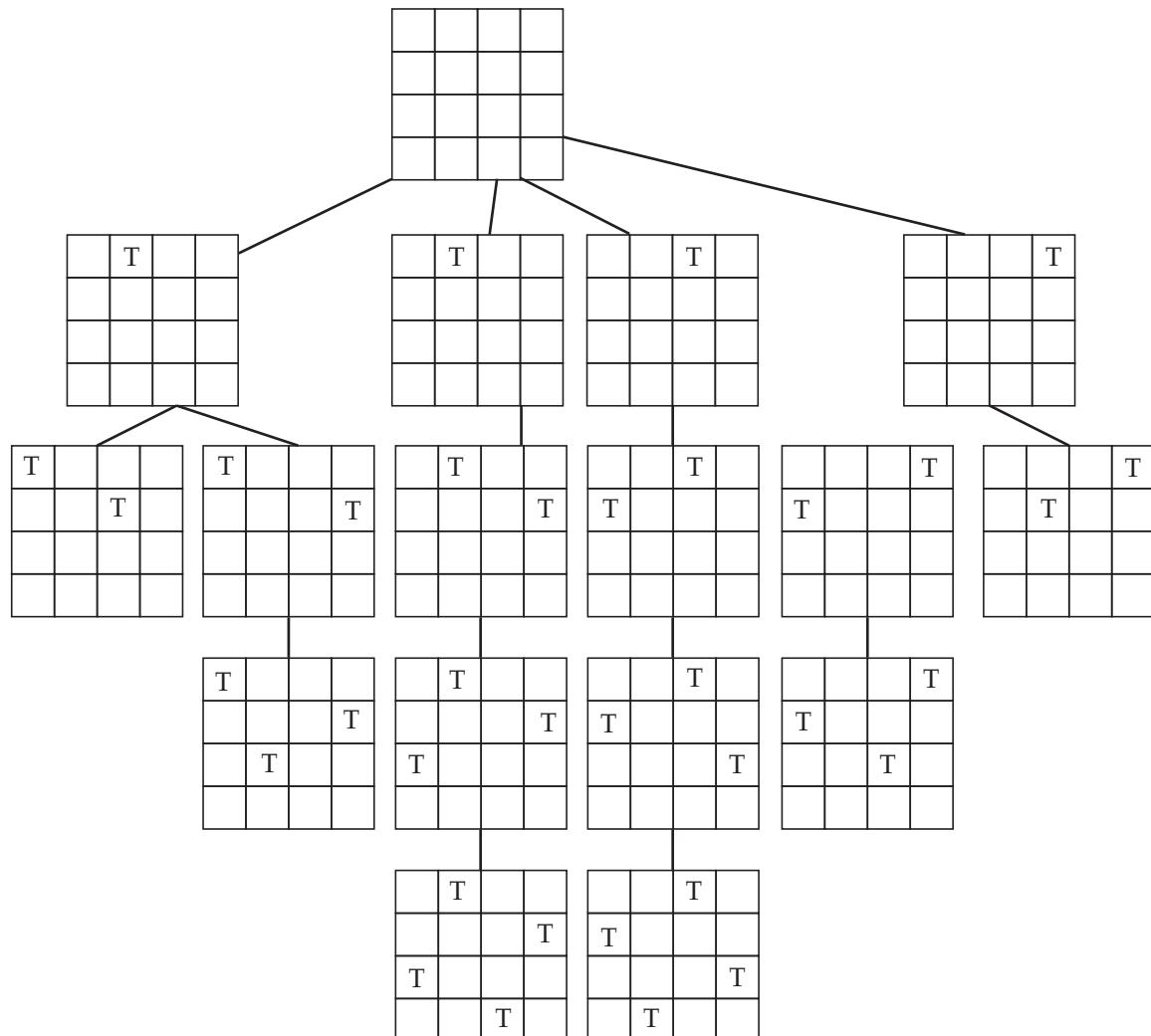


Fig. 15.2. Recursion tree for 4 queens problem.

[\*] This is also called state-space tree.

We found when none of the children nodes of a placed queen leads to a promising solution, the algorithm backtracks and removes the last queen placed. So the two unique solutions that we get are given as [2, 4, 1, 3] and [3, 1, 4, 2].

### 15.3 SUBSET SUM PROBLEM

---

The problem subset sum belongs to NP-complete class (Refer chapter 17 to know about NP-complete problems). In this section, we discuss a recursive algorithm for the subset sum problem. The problem can be stated as follows:

Given a set  $S$  of positive integer and target integer  $T$ , is there a subset of elements in  $S$  that add upto  $T$  ?

For example, if  $S = \{1, 2, 3, 4, 6, 7, 8\}$  and  $T = 10$ , then the subset sum includes the subsets  $\{1, 2, 3, 4\}$  or  $\{1, 3, 6\}$  or  $\{2, 8\}$  or  $\{3, 7\}$  or  $\{4, 6\}$  for which the answer is TRUE. Whereas for the subset  $\{1, 2, 3, 6\}$  the answer is FALSE.

There are two special cases.

**Case I :** If the target value  $T$  is zero and simultaneously the set  $S$  is empty, then we can immediately return TRUE. This is so, because empty set is a subset of every set  $S$ , and the elements of the empty set add upto zero.

**Case II :** If the target value  $T$  is less than zero ( $T < 0$ ) or  $T$  is not equal to zero ( $T \neq 0$ ) but the set  $S$  is empty, then we can immediately return FALSE.

Let us take an arbitrary element  $x \in S$ . There is a subset of  $S$  that sums to  $T$  if and only if the following statements hold good :

1. There is a subset of  $S$  that excludes  $x$  and whose sum is  $T$ . This shows that there must be a subset of  $S \setminus \{x\}$  that sums to  $T-x$ .
2. There is a subset of  $S$  that excludes  $x$  and whose sum is  $T$ . This implies that there must be a subset of  $S \setminus \{x\}$  that sums to  $T$ .

Therefore, we can solve the subset sum problem by reducing it to two simpler instances: SUBSETSUM ( $S \setminus \{x\}$ ,  $T-x$ ) and SUBSETSUM ( $S \setminus \{x\}$ ,  $T$ ). The following resulting Recursive algorithm gives the result of subset sum problem.

```

SUBSETSUM (S[1.....n] , T)
1. if (T = 0)
2.   return TRUE
3. else if (T < 0 or n = 0)
4.   return FALSE
5. else
6.   return SUBSETSUM (S[2.....n] , T) or SUBSETSUM S[2.....n] ,
T-S[1] .

```

## 15.4 GRAPH COLORING PROBLEM

---

Traditionally graph coloring problem is categorised into two types—Vertex coloring and edge coloring. These two coloring problems are moreover similar and can be defined as given. Given as undirected graph and a number  $m$ , determine if the graph can be colored with at most  $m$  colors such that no two adjacent vertices (or edges) of the graph are colored with same color.

The convention of using colors originates from coloring the countries of a map, where each face is literally colored. This was generalized to coloring the faces of a graph embedded in the plane. Moreover graph coloring enjoys many practical as well as theoretical challenges and it is still a very active field of research.

Let us take an example to color the vertices of the following graph  $G$  with 3 colors i.e., red (R), blue (B) and Green (G).

The idea behind the backtracking algorithm for graph coloring problem can be outlined as follows.

We start from initial vertex. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we don't find a color due to clashes then we backtrack and return false. Based on these idea we design the following RECURSIVE  $m$ \_COLORING algorithm. The graph is represented by its adjacency matrix  $G[1....n, 1.....n]$ . All assignments of 1, 2, .....  $m$  colors to the vertices of the graph such that adjacent vertices are assigned distinct color (integer) are printed. We take  $k$  as the index of the next vertex to color.

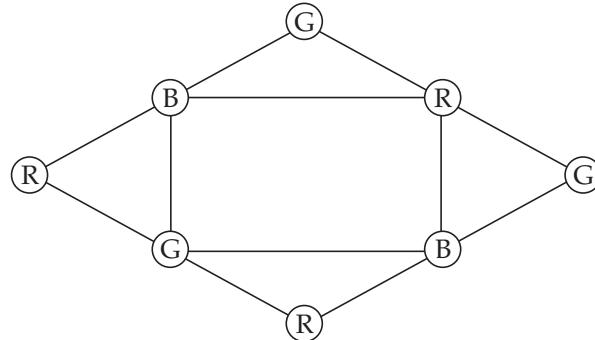


Fig. 15.3. Coloring graph  $G$ .

```

RECURSIVE m_COLORING (k)
1. {repeat
2. {generate all legal assignment for x[k]
3. next value (k) // assign to x[k] a legal value
4. If (n[k] = 0) then return // no new color possible
5. If (k = n) then // at most m color have been used to color n
   vertices
6. Write (x[1.....n])
7. Else m_COLORING (k + 1)
8. } until (false)
9. }

```

### Running Time

The running time of above recursive algorithm is  $O(nm^n)$ , where  $n$  is the number of vertices of graph  $G$  and  $m$  is the number of colors to be used.

### CHAPTER NOTES

---

- Backtracking is one of the strategies to reduce the complexity of a problem. It is mainly useful when there is no solution by going forward in that direction so we require backtracking from it to reduce the complexity and save the time. It has the ability to give same result in far fewer attempts than the exhaustive or brute force method.
- Backtracking approach is based on recursion.
- In typical  $N$ -queens problem we are given with  $n$  numbers of queens and an  $N \times N$  chess board. We need to figure out a way to place all  $N$  queens on the board so that no queens are attacking another queen.
- We can use recursion tree (or state space tree) to solve backtracking algorithm effectively and efficiently.
- In subset sum problem we are given with a set  $S$  of positive integers and target integer  $T$ , is there a subset of elements in  $S$  that add upto  $T$ ?
- In graph coloring problem we are given with an undirected graph and a number  $m$  and we need to determine if the graph can be colored with at most  $m$  colors such that no two adjacent vertices (or edges) of the graph are colored with same color.

### EXERCISES

---

1. Describe the 4-queens problem using backtracking.
2. Let  $S = \{15, 7, 20, 5, 18, 10, 12\}$  and  $T = 35$ . Find all possible subsets of  $S$  that sum to  $T$ . Draw the portion of the state space tree (recursion tree) that is generated.
3. Compare and contrast brute force approach Vs backtracking.
4. Suggest a solution for 8-queens problem.
5. What is graph coloring ? Present an algorithm which finds  $m$ -coloring of a graph.
6. Write an algorithm to generate next vertex color in  $m$ -coloring problem.
7. Write a recursive backtracking algorithm for subset sum problem.
8. Write an algorithm for  $N$ -queens problem. Analyze its time complexity.

# STRING MATCHING

# 16

CHAPTER

## OBJECTIVES OF LEARNING

After going through this chapter, the reader would be able to understand:

- Objectives of string matching
- Valid shift and Invalid shift
- Brute-Force string matching algorithm
- Efficient string matching algorithm: Rabin-Karp algorithm
- Spurious hit and valid match

# Chapter 16 STRING MATCHING

## INSIDE THIS CHAPTER

16.1 Introduction

16.2 Brute-Force String Matching Algorithm

16.3 Rabin-Karp String Matching Algorithm

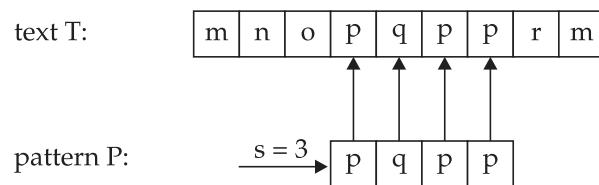
## 16.1 INTRODUCTION

In this chapter, we are going to study the problem of finding all occurrences of a given pattern string in a given text string, a problem that arises frequently in text editing programs internet searches and DNA pattern matching. This kind of problem is called “String Matching”. So our objective is to find the location of a specific text pattern within a large body of text (e.g., a sentence, a paragraph, a book, etc.)

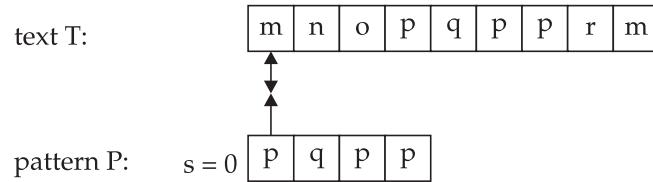
We formally define the string matching problem as follows. Let us assume that, the text in the array  $T[1 \dots n]$  of length  $n$  and the pattern is an array  $P[1 \dots m]$  of length  $m \leq n$ ; where character arrays  $T$  and  $P$  called strings of character. We have to also assume that the elements of  $T$  and  $P$  are the characters drawn from a finite alphabet denoted as  $\Sigma$ . For example,  $\Sigma = \{0, 1\}$  or  $\Sigma = \{a, b, c, \dots, z\}$  or  $\Sigma = \{A, B, \dots, Z\}$ . We say that pattern  $P$  occurs with shift  $s$  in text  $T$  if  $0 \leq s \leq n - m$  and  $T[s + 1 \dots s + m] = P[1 \dots m]$ .

**Valid Shift** – If  $P$  occurs with shift  $s$  in  $T$ , then  $s$  is a valid shift; otherwise,  $s$  is an **invalid shift**.

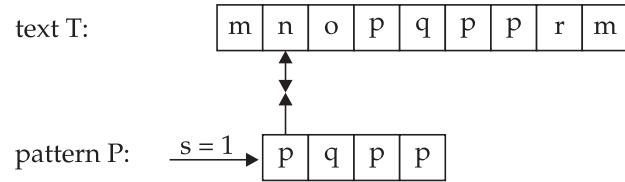
### Example 16.1



In this example, we want to find all the occurrences of given pattern  $P = "pqpp"$  in the given text = "mnopqpprm". At first case i.e., at no shift ( $s = 0$ ) there is a miss match occurs. So, it is invalid shift.

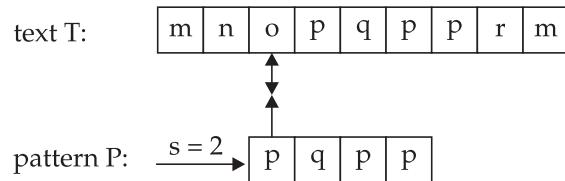


**When  $s = 1$ :**



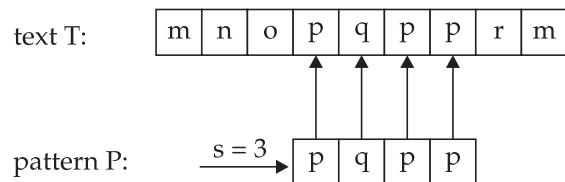
There is also a miss match occurs. So it is also an invalid shift.

**When  $s = 2$ :**



As  $p$  doesn't matches to  $o$ ,  $s = 2$  is also an invalid shift.

**When  $s = 3$ :**



So now by doing shift ( $s = 3$ ), we found the pattern occurs in the text T. So  $s = 3$  is a valid shift.

### String Matching Algorithms

There are number of string matching algorithms in existence today. They are:

- Brute-Force (or Navie String Matching)
- Robin-Karp

- Knuth-Morris-Pratt
- Boyer-Moore

In this chapter, we will only study the first two algorithms in the detail.

## 16.2 BRUTE-FORCE STRING MATCHING ALGORITHM

---

The brute-force string matching is the fundamental and most general string matching algorithm. The basic mechanism behind this technique can be underlined as follows. Let the text is an array  $T[1 \dots n]$  of length  $n$  and the pattern is an array  $P[1 \dots m]$  of length  $m \leq n$ . We further assume that the elements of  $P$  and  $T$  are characters drawn from a finite alphabet  $\Sigma$ . For example, we may have  $\Sigma = \{0, 1, \dots, 9\}$  or  $\Sigma = \{a, b, \dots, z\}$ . The character arrays  $P$  and  $T$  are often called strings of characters.

We say that pattern  $P$  occurs with shift  $s$  in text  $T$  if  $0 \leq s \leq n - m$  and  $T[s + 1 \dots s + m] = P[1 \dots m]$ . If  $P$  occurs with shift  $s$  in  $T$ , then we call  $s$  a “valid shift”; otherwise we call  $s$  an invalid shift. The string matching problem tells us to find all valid shifts with which a given pattern  $P$  occurs in a given text  $T$ .

The brute-force algorithm finds all valid shifts using a loop that checks the condition  $P[1 \dots m] = T[s + 1 \dots s + m]$  for each of the  $n - m + 1$  possible values of  $s$ . The pseudocode for brute-force string matching is given as follows.

### 16.2.1 Algorithm Steps

```

BRUTE-FORCE-STRING-MATCHER (T, P)
1.    $n \leftarrow \text{length}[T]$            //  $n$  is the length of text  $T$ .
2.    $m \leftarrow \text{length}[P]$            //  $m$  is the length of pattern  $P$ .
3.   for  $s \leftarrow 0$  to  $n - m$       //  $s$  refers to shift.
4.     do if  $P[1 \dots m] = T[s + 1 \dots s + m]$ .
5.     then print "Pattern occurs with shift"  $s$ .

```

### Running Time

The for loop beginning on line 3 consider each possible shift explicitly whereas the test on line 4 determines the current shift is valid or not. However this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 outputs each valid shift  $s$ .

The procedure BRUTE-FORCE-STRING-MATCHER takes  $O((n - m + 1) m)$  time in worst case, which is  $O(n^2)$  if  $m = \lfloor n/2 \rfloor$ . When  $m = n$ , then the running time becomes  $O(m)$ , which is best case.

**Example 16.2** Show the comparisons the Brute-Force string matcher makes for the pattern  $P = aab$  in the text  $acaabc$ . Find its time complexity.

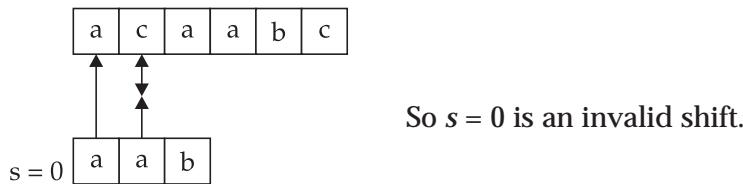
**Solution :** Given text  $T = acaabc$  and the given pattern  $P = aab$

$$n = \text{length of text } T = 6$$

$$m = \text{length of pattern } P = 3$$

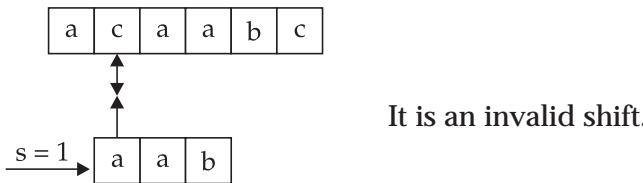
So we take  $s$  from 0 to 3.

**When  $s = 0$**



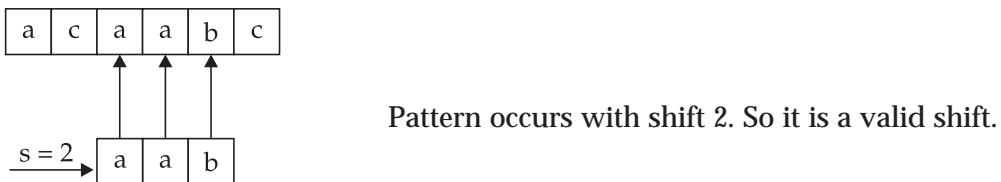
So  $s = 0$  is an invalid shift.

**When  $s = 1$**



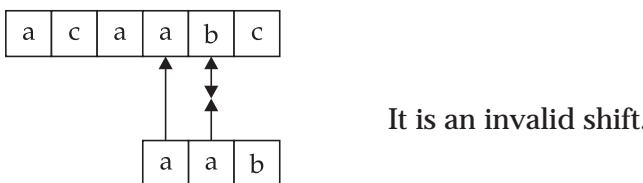
It is an invalid shift.

**When  $s = 2$**



Pattern occurs with shift 2. So it is a valid shift.

**When  $s = 3$**



It is an invalid shift.

Time complexity of the operation =  $O(m(n - m + 1))$ .

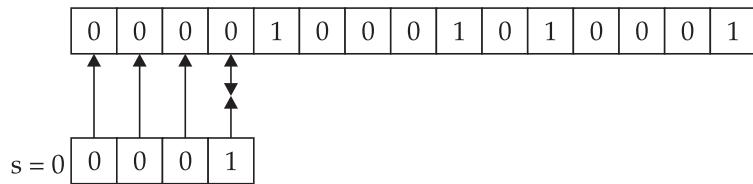
**Example 16.3** Show the comparisons the Brute-Force string matcher makes for the pattern  $P = 0001$  in the text  $T = 000010001010001$ .

**Solution :** Given text  $T = 000010001010001$  and the given pattern  $P = 0001$

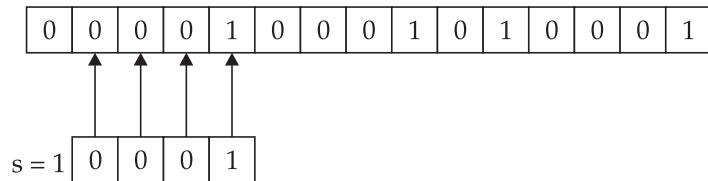
$$n = \text{length of text } T = 15$$

$$m = \text{length of pattern } P = 4$$

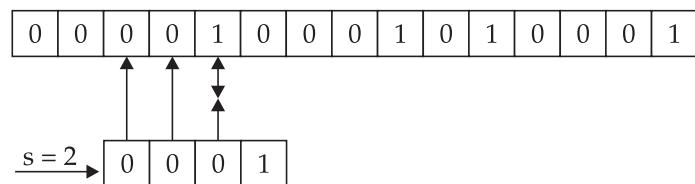
So we take  $s$  from 0 to 11.

**When  $s = 0$** 

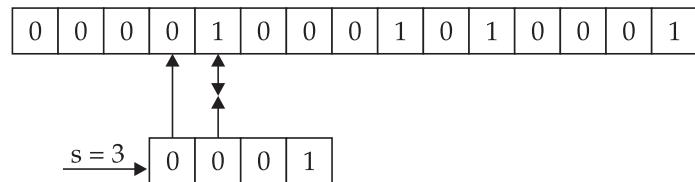
It is an invalid shift.

**When  $s = 1$** 

$s = 1$  is a valid shift.

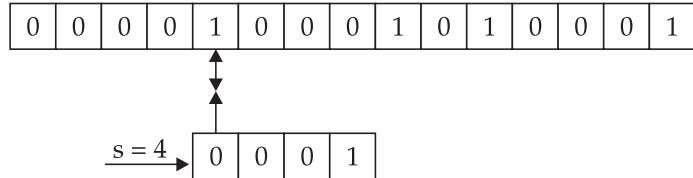
**When  $s = 2$** 

$s = 2$  is an invalid shift.

**When  $s = 3$** 

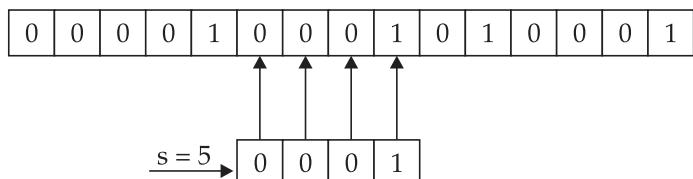
$s = 3$  is an invalid shift.

**When  $s = 4$**



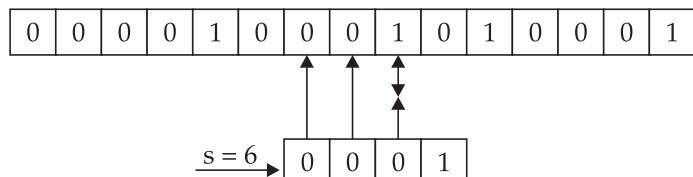
$s = 4$  is an invalid shift.

**When  $s = 5$**



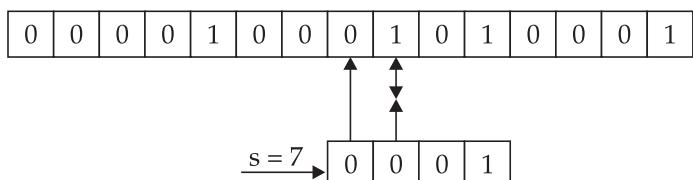
$s = 5$  is a valid shift.

**When  $s = 6$**



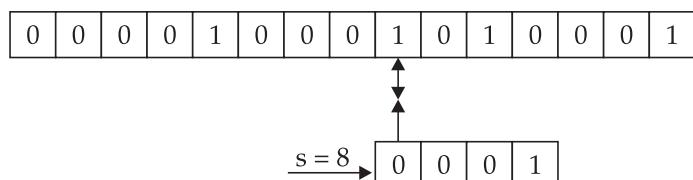
$s = 6$  is an invalid shift.

**When  $s = 7$**



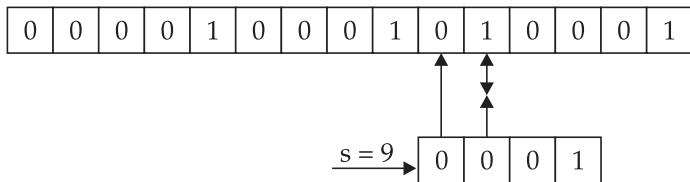
$s = 7$  is an invalid shift.

**When  $s = 8$**



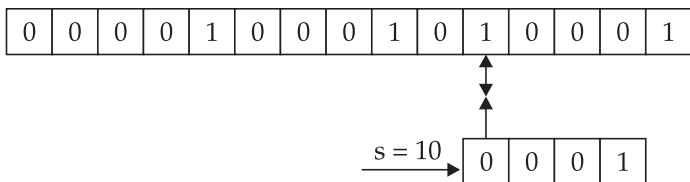
This is an invalid shift.

**When  $s = 9$**



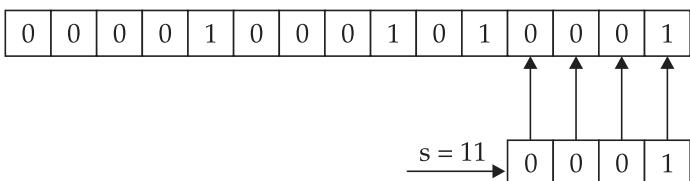
So  $s = 9$  is an invalid shift.

**When  $s = 10$**



So  $s = 10$  is an invalid shift.

**When  $s = 11$**



As the pattern P found in text T,  $s = 11$  is a valid shift.

### 16.3 RABIN-KARP STRING MATCHING ALGORITHM

---

Rabin-Karp string matching algorithm was proposed by Michael O. Rabin and Richard M. Karp in 1987 which is practically used for detecting **plagiarism**. This string matching algorithm is based on the idea of comparing string's hash values, rather than the strings themselves. Thus, it would seem all we have to do is to compute the hash value of the pattern we are searching for, and then looking for a substring having the same hash value. If the hash value matches and the substring doesn't then it is a **spurious hit**. If both match then it is a **valid match**.

**Note:** Hash value is a fixed-length value (we may treat it as a decimal number) which acts as shortened reference to the original string of characters.

The Rabin-Karp algorithm uses number-theoretic notices such as equivalence of two number modulo a third number. Let the pattern P [1 ..... m] and text T [1 ..... n] be digits in radix- $\Sigma$  notation; where  $\Sigma = (0, 1, \dots, 9)$ . Let  $p$  be the corresponding decimal value for the pattern P [1..... m] which can be computed in  $\Theta(m)$  using Horner's rule in the following way.

$$p = P[m] + 10 (P[m-1] + 10(P[m-2] + \dots + 10 (P[2] + 10P[1]) \dots))$$

In similar manner , for the given text T[1..... n] we can calculate its corresponding decimal value  $t_s$  of length- $m$  substring T[s + 1..... s + m], for  $s = 0, 1, \dots, n - m$ . If  $T[s+1\dots s+m] = P[1\dots m]$  then  $t_s = p$ ; which tell us that  $s$  is a valid shift. The value  $t_0$  can be computed similarly from the text T[1.....m] in time  $\Theta(m)$ . The remaining values  $t_1, t_2, \dots, t_{n-m}$  can be computed in time  $\Theta(n - m)$ . It result that  $t_{s+1}$  can be computed from  $t_s$  in constant time, since

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

We remove the high-order digit  $t_s$  by subtracting  $10^{m-1}T[s+1]$  from  $t_s$  and by multiplying the result by 10 we shift the number left one position. By adding  $T[s+m+1]$  to the result, it brings appropriate lower order digit.

The entire process seems to be inconvenit if  $p$  and  $t_s$  are too large. If this is so, we compute  $p$  and  $t_s$ 's modulo a suitable modulus  $q$ . That is

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

where  $h = d^{m-1} \bmod q$  is the value of the digit '1' in the high-order position of an  $m$ -digit text window. The entire procedure is given in the following algorithm.

### 16.3.1 Algorithm Steps

RABIN-KARP-MATCHER (T, P, d, q)

1.  $n \leftarrow \text{length}[T]$  //  $n$  is the length of given text T.
2.  $m \leftarrow \text{length}[P]$  //  $m$  is the length of given pattern P.
3.  $h \leftarrow d^{m-1} \bmod q$  //  $d$  is the radix (which is  $|\Sigma|$ ) and  $q$  is the prime number.
4.  $p \leftarrow 0$
5.  $t_0 \leftarrow 0$
6. for  $i \leftarrow 1$  to  $m$  // Preprocessing
7. do  $p \leftarrow (dp + P[i]) \bmod q$
8.  $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9. for  $s \leftarrow 0$  to  $n - m$  // Matching
10. do if  $p = t_s$
11. then if  $P[1\dots m] = T[s+1\dots s+m]$
12. Output "pattern occurs with shift"  $s$
13. if  $s < n - m$
14. then  $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

### Running Time

In the above algorithm all characters are interpreted as radix- $d$  digits. Line 3 initializes  $h$  to the value of the high-order digit position of an  $m$ -digit window. Lines 4 - 8 compute  $p$  as the value of  $P[1.....m] \bmod q$  and  $t_0$  as the value of  $T[1.....m] \bmod q$ . Lines 9-14 maintains a for loop that iterates through all possible shifts  $s$ . If  $p = t_s$  in line 10, it indicates a hit and then we check to see if  $P[1.....m] = T[s + 1.....m]$  is true in line 11 which result no spurious hit is possible. Line 12 prints all possible valid shifts. If  $s < n - m$  then the for loop is executed at least one more time and computes the value of  $t_{s+1} \bmod q$  from the value of  $t_s \bmod q$ .

The RABIN-KARP-MATCHER takes  $\theta(m)$  processing time and its worst case running time is  $\theta((n - m + 1)m)$ .

**Example 16.4** Working modulo  $q = 11$ , how many spurious hits does the Rabin-Karp matcher encounter in the text  $T = 3141592653589793$  when looking for the pattern  $P = 26$ ?

**Solution :** Given text  $T = 3141592653589793$  and pattern  $P = 26$ .

Working modulo  $q = 11$

Therefore,  $P \bmod q = 26 \bmod 11 = 4$

Rabin-Karp matcher performs the following steps to suitably calculate the number of spurious hits.

Step 1	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$$31 \bmod 11 = 9 \text{ not equal to } 4$$

Step 2	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$$14 \bmod 11 = 3 \text{ not equal to } 4$$

Step 3	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$$41 \bmod 11 = 8 \text{ not equal to } 4$$

Step 4	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$$15 \bmod 11 = 4 \text{ equal to } 4 \rightarrow \text{spurious hit}$$

Step 5	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$$59 \bmod 11 = 4 \text{ equal to } 4 \rightarrow \text{spurious hit}$$

Step 6	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$92 \bmod 11 = 4$  equal to 4  $\rightarrow$  spurious hit

Step 7	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$26 \bmod 11 = 4$  equal to 4  $\rightarrow$  valid (or exact) match

Step 8	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$65 \bmod 11 = 10$  not equal to 4

Step 9	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$53 \bmod 11 = 9$  not equal to 4

Step 10	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$35 \bmod 11 = 2$  not equal to 4

Step 11	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$58 \bmod 11 = 3$  not equal to 4

Step 12	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$89 \bmod 11 = 1$  not equal to 4

Step 13	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$97 \bmod 11 = 9$  not equal to 4

Step 14	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>1</td><td>5</td><td>9</td><td>2</td><td>6</td><td>5</td><td>3</td><td>5</td><td>8</td><td>9</td><td>7</td><td>9</td><td>3</td></tr></table>	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3		

$79 \bmod 11 = 2$  not equal to 4

Step 15 

3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$93 \bmod 11 = 5 \text{ not equal to } 4$$

Therefore the total number of spurious hits are 3.

**Example 16.5** Working modulo  $q = 5$ , how many spurious hits does the Rabin-Karp matcher encounter in the text  $T = 289402340370694045$  when looking for the pattern  $P = 234$ .

**Solution:** Given text  $T = 289402340370694045$  and pattern  $P = 234$

$$\text{Working modulo } q = 5$$

$$\text{Therefore, } P \bmod q = 234 \bmod 5 = 4$$

Rabin-Karp matcher performs the following steps to suitably calculate the number of spurious hits.

Step 1 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$289 \bmod 5 = 4 \text{ equal to } 4 \rightarrow \text{spurious hit}$$

Step 2 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$894 \bmod 5 = 4 \text{ equal to } 4 \rightarrow \text{spurious hit}$$

Step 3 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$940 \bmod 5 = 0 \text{ not equal to } 4$$

Step 4 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$402 \bmod 5 = 2 \text{ not equal to } 4$$

Step 5 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$023 \bmod 5 = 3 \text{ not equal to } 4$$

Step 6 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$234 \bmod 5 = 4 \text{ equal to } 4 \rightarrow \text{valid match}$$

Step 7 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$340 \bmod 5 = 0 \text{ not equal to } 4$$

Step 8 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$403 \bmod 5 = 3 \text{ not equal to } 4$$

Step 9 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$037 \bmod 5 = 2 \text{ not equal to } 4$$

Step 10 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$370 \bmod 5 = 0 \text{ not equal to } 4$$

Step 11 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$706 \bmod 5 = 1 \text{ not equal to } 4$$

Step 12 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$069 \bmod 5 = 4 \text{ equal to } 4 \longrightarrow \text{spurious hit}$$

Step 13 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$694 \bmod 5 = 4 \text{ equal to } 4 \longrightarrow \text{spurious hit}$$

Step 14 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$940 \bmod 5 = 0 \text{ not equal to } 4$$

Step 15 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$404 \bmod 5 = 4 \text{ equal to } 4 \longrightarrow \text{spurious hit}$$

Step 16 

2	8	9	4	0	2	3	4	0	3	7	0	6	9	4	0	4	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$045 \bmod 5 = 0 \text{ not equal to } 4$$

Hence the total number of spurious hits are 5.

## CHAPTER NOTES

---

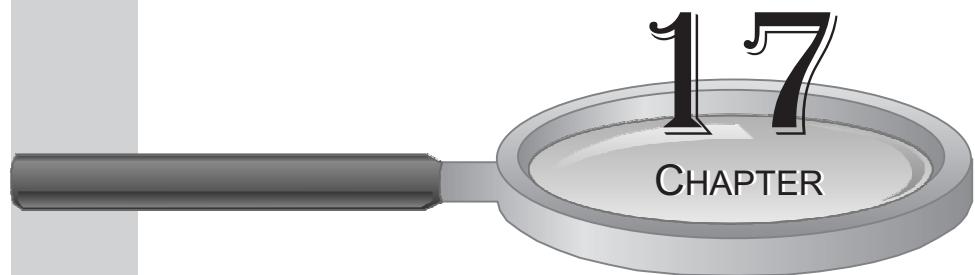
- The idea behind the string matching problem is that we want to find all occurrences of the pattern P in the given text T.
- Different string matching algorithms are : brute-force (naive) string matching, Rabin-Karp string matching, Knuth-Morris-Pratt string matching, Boyer-Moore string matching etc.
- The brute-force string matching utilities iteration over given text T. At each iteration, we compare the sequence against P until all letters match till the end of the alphabet is reached.
- Rabin-Karp string matching algorithm works on comparing strings' hash values rather than strings themselves. In this case we compute the hash value of the pattern we are searching for, and then looking for the substring in the given text having the same hash value. If the hash value of both the strings match but the strings are different then it is a spurious hit. If both matches then it is a valid match.
- Both the brute-force and Rabin-Karp string matching algorithms have the worst case running time of  $\theta((n - m + 1)m)$  where  $n$  is the length of text T and  $m$  is the length of pattern P.
- Different applications of Rabin-Karp string matching includes detecting plagiarism, DNA pattern matching, internet searches etc.

## EXERCISES

---

1. Describe the naive string matching algorithm. Show the comparisons the algorithm makes for the pattern  $P = aabab$  in the text  
 $T = aaababaabaababaab.$
2. What is the basic objective of Rabin-Karp string matching algorithm? Explain the role of hash function? What is its worst case running time of this algorithm?  
**(Hints**-Hash function generator hash value for the given characters of strings).
3. Working modulo  $q = 11$ , how many spurious hits does Rabin-Karp matcher encounter in the given text  $T = 31415926$  when looking for the pattern  $P = 26$ ?
4. Working modulo  $q = 13$ , how many spurious hits does Rabin-Karp matcher encounter in the given text  $T = 2359023141526739921$  when looking for the pattern  $P = 34145$ .
5. What are the basic objectives of string matching ? Illustrate through an example.

# NP-COMPLETENESS AND APPROXIMATION ALGORITHMS



## OBJECTIVES OF LEARNING

**After going through this chapter, the reader would be able to understand:**

- Polynomial time
- Reduction technique
- Decision problems, Search problems and Optimization problems
- The classes : P, NP, NP-hard and NP-complete
- Satisfiability
- Cook's theorem
- Approximation algorithm (TSP, Vertex cover problem, 0-1 Knapsack problem)
- Approximation schemes

# NP-COMPLETENESS AND APPROXIMATION ALGORITHMS

## INSIDE THIS CHAPTER

17.1 Introduction	17.2 Matching Problem
17.3 Reductions and its Applications	17.4 The Classes : P and NP
17.5 NP-Hard and Np-Complete	17.6 Satisfiability (SAT)
17.7 Cook's Theorem	17.8 Common NP-complete Problems with Proofs
17.9 Other Useful Np-Complete Problems	17.10 Other Important Complexity Classes
17.11 Approximation Algorithms	17.12 Different Appoximation Schemes

## 17.1 INTRODUCTION

Suppose you are working in a multinational company as an “Algorithm Designer”. Your job is to design **efficient algorithms** which will take less time complexity and will produce optimum solution. One day your boss call you and ask you to **schedule** some set of jobs with the following sizes by the help of two processors  $P_1$  and  $P_2$  ; where these two processors are identical in nature to automate the process.

The set of jobs are  $\langle J_1, J_2, \dots, J_n \rangle$  and their sizes are given by  $\langle S_1, S_2, \dots, S_n \rangle$  respectively. Task is to schedule the jobs in such a way so that last job finishes fastest. Then after getting this task from your boss how you will approach ! There are various ways to solve this problem which might be murmuring at this time.

### Algorithm 17.1

The general idea behind this algorithm is **least lightly loaded** to the processors where tie can be arbitrary break up.

$$\begin{array}{ll}
 \frac{P_1}{S_1} & \frac{P_2}{S_2} \\
 & S_3 \text{ (if } S_1 > S_2) \\
 S_4 \text{ (if } S_2 + S_3 > S_1)
 \end{array}$$

But this doesn't work always for time T optimum schedule which is less than T. Take an example there are five set of jobs  $J_1, J_2, J_3, J_4$  and  $J_5$ . Their sizes are given as :  $S_1 = 2, S_2 = 3, S_3 = 2, S_4 = 2$  and  $S_5 = 3$ . The below schedule produces an optimum result where time taken in each case is 6 units.

$P_1$	$P_2$
$J_1(2)$	$J_2(3)$
$J_3(2)$	$J_5(3)$
$J_4(2)$	
	6
	6

On the other hand the following schedule shows the algorithm fails to work.

$P_1$	$P_2$
$J_1(2)$	$J_2(3)$
$J_3(2)$	$J_4(2)$
$J_5(3)$	
	5
	7

After this failure you start thinking for an another approach and develop the following idea.

### Algorithm 17.2

The idea is to sort the jobs according to the size and implement **greedy strategy** to lightly loaded processor. For the same set of jobs the first schedule takes place in the following way.

$P_1$	$P_2$
$J_1(2)$	$J_2(3)$
$J_3(2)$	$J_5(3)$
$J_4(2)$	
	6
	6

Again on the other hand the following schedule shows the algorithm fails to work.

$P_1$	$P_2$
$J_1(2)$	$J_3(2)$
$J_4(2)$	$J_2(3)$
$J_5(3)$	
	5
	7

Since failure is the stepping stone for success, you again start to think to develop an algorithm based on brute force approach in the following way.

### Algorithm 17.3

**Brute force algorithm** is a technique for solving the problem systematically enumerating all possible candidates for the solution and checking each candidate satisfies the problem statement or not. This algorithm is very simple but it requires huge number of steps to complete. So it is least desirable but there is no hope remain in you because of last two failures.

From  $S_1, S_2, \dots, S_n$  take the subsets of  $S$ . Then compute the size of  $S$  and the size of  $\bar{S}$ . Choose the maximum among  $S$  and  $\bar{S}$  and then find for all possible subset and choose the minimum.

This technique works correctly for small value of  $n$ . But when the size of  $n$  is 1000 ! For  $n = 1000$  we have  $2^{1000}$  numbers of subsets possible. Then how to calculate them with the help of a computer.

A faster computer can execute  $10^{20}$  instruction/second. Here one instruction means one calculation of  $S$  and  $\bar{S}$ . Currently in the world there are near about  $10^{20}$  numbers of computer present. If all of them are used to solve the above problem then

$$\text{Number of instruction per year} = 10^{40} \times \frac{60}{10^2} \times \frac{60}{10^2} \times \frac{24}{10^2} \times \frac{365}{10^3} \approx 10^{50} \text{ instruction/year.}$$

Number of year it requires =  $\frac{2^{1000}}{10^{50}} = \frac{2^{1000}}{2^{200}} = 2^{800}$  years. So practically how many generations after you it required to compute !

If your boss is little intelligent he can figure out and fire. Even if not figure out then also fire because your process has not yet stopped after a week or more than that. This is a motivation for studying NP-complete. So this problem is among the hard problems in this world. This shows that so far no body else has able to find the solution to this problem. If your boss doesn't agree to this statement then you may convince him by saying if somebody manages to solve this problem, then a lot of other problems which have not solved by other people can be solved. This kind of statement you have to make to save your job.

## 17.2 MATCHING PROBLEM

---

Matching (denoted as  $M$ ) in a graph  $G = (V, E)$  is a subset  $M$  of  $E$  such that no two edges in  $M$  have the same end points. So every vertex should appear only once. For the graph  $G$  given in Fig. 17.1  $\{(a, b), (e, d)\}$  and  $\{(a, d), (b, e)\}$  vertex sets are matching whereas the vertex sets  $\{(a, b),$

$\{a, d\}$  and  $\{(a, b), (b, e)\}$  are not matching. Again matching problem is of two types. They are **perfect matching** and **maximum matching**.

**Perfect matching** is a matching  $M$  such that all vertices are end points of exactly one edge in  $M$ . Every perfect matching is a maximum matching. For the graph  $G$  given in Fig. 17.1 we have the following sets of perfect match :  $\{(a, b), (e, d), (c, f)\}$  and  $\{(a, d), (b, e), (c, f)\}$ .

**Maximum matching** is a matching that contains the largest possible number of edges. For the graph  $G$  the maximum matching sets are  $\{(a, b), (e, d), (c, f)\}$ , and  $\{(a, d), (b, e), (c, f)\}$ .

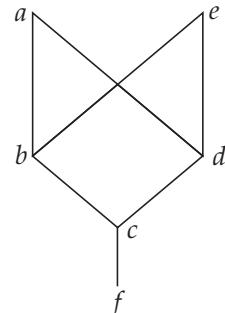


Fig. 17.1. Graph  $G$ .

### 17.3 REDUCTIONS AND ITS APPLICATIONS

Reduction is the transformation process through which we design an efficient algorithm (decision/search type) for problem  $\pi_2$  from a given efficient algorithm (decision/search type) for problem  $\pi_1$ .

At this point one might ask the following questions.

- What is an efficient algorithm?
- What is a decision problem?
- What is a search problem?

We now discuss the answers to these questions. A minimum requirement for an algorithm to be considered as efficient algorithm is that its running time is polynomial i.e.,  $O(n^c)$  for some constant  $c$  and  $n$  is the size of the input. A problem is said to be solvable in polynomial time if there is a polynomial time algorithm that solves it.

A problem is called decision problem which outputs either yes or no. An algorithm for a decision problem is termed as decision algorithm.

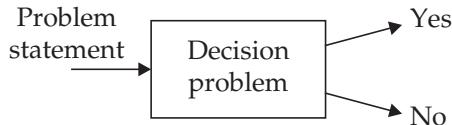


Fig. 17.2. Decision problem.

For example, the decision type of perfect matching and maximum matching look like as follows.

Perfect matching : Decision type

Input : Graph  $G$

Output : Does  $G$  have a perfect match?

Maximum matching : Decision type



**Input :** Graph G  
**Output :** Does G forms a maximum no of  $k$  matching?  
In search problem we always try to figure out the possible reason(s) behind the yes or no output. Therefore, the decision version of a problem is easier to handle rather than search type.

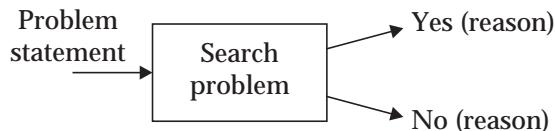
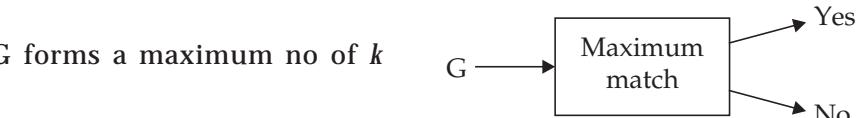


Fig.17.3. Search problem.

For example, the search version of perfect matching and maximum matching look like as follows.

Perfect matching : Search type

Input : Graph G

Output : If G has a perfect match then output the set of edges that forms a matching.

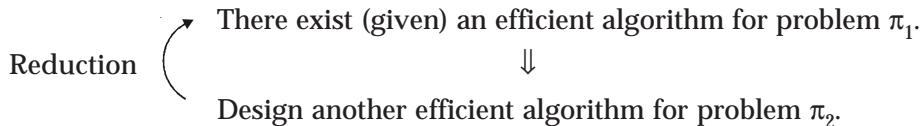
Maximum matching : Search type

Input : Graph G

Output : Set of edges in matching of maximum match.

Apart from decision and search versions of problems there is also another special kind of problem called **optimization problem** which involves the identification of an optimal (either minimum or maximum) value of a given cost function. In both search and optimization problems we always try to figure out associated value and wish to find a **feasible solution** with best value.

Hopefully we finish answering all questions that arises in our mind ! Before moving towards different applications of reduction let us see how the process of reduction takes place. As we have previously told the technique of reduction happens in the following way.



**Note :** The problems  $\pi_1$  and  $\pi_2$  may be search type or decision type.

In this situation, we are reducing the problem  $\pi_2$  to problem  $\pi_1$  which can be represented as  $\pi_2 \leq_R \pi_1$  and  $\pi_2 \alpha \pi_1$ . As this is a **polynomial reduction** it is represented as  $\pi_2 \leq_{poly} \pi_1$ . Throughout this chapter our focus will be on polynomial reduction.

**Note :** The relation polynomial time reduction ( $\leq_{poly}$ ) is transitive i.e., if  $\pi_2 \leq_{poly} \pi_1$  and  $\pi_1 \leq_{poly} \pi_3$  then  $\pi_2 \leq_{poly} \pi_3$ .

Apart from this there are many other kinds of reduction techniques are present; like many-one reduction (Karp reduction), logspace reduction, etc. Here we sketch an intuition behind these reduction techniques.

In the context of decision problems, a problem  $\pi_2$  is **many-one reducible** to  $\pi_1$  (represented as  $\pi_2 \leq_m \pi_1$ ) if there is a many-to-one function  $f()$  that maps an instance  $I_2 \in \pi_2$  to instance  $I_1 \in \pi_1$  such that the output to  $I_1$  is yes iff the output to  $I_2$  is yes.

Note that the mapping need not be one-one and therefore the process of reduction isn't a symmetric relation. If the mapping function  $f()$  can be computed in polynomial time then it is a polynomial reduction.

Another class of reduction is known as **logspace reduction**; which is denoted as  $\pi_2 \leq_{\log} \pi_1$ . If  $\pi_2 \leq_{\log} \pi_1$  then it is obvious that  $\pi_2 \leq_{\text{poly}} \pi_1$ .

**Example 17.1** *Polynomially reduce maximum matching problem to perfect matching problem.*

**Solution :** The reduction that we are going to perform looks like as follows. We assume there exists a polynomial time algorithm for perfect matching from this we design another polynomial time algorithm for maximum matching. For this let us take a graph G. Test whether graph G has a perfect match. If yes then the size of maximum matching is  $|V|/2$ ; where  $|V|$  denotes the number of vertices in graph G. If the answer is no then take a new vertex  $u_1$  and connect edge from  $u$  to all the vertex in G. Name the new graph as  $G_1$ .

Again feed  $G_1$  to the problem of perfect match. If this says yes then  $G_1$  has a maximum matching of size  $(|V| - 1)/2$ . If the output to the problem is no then add a new vertex  $u_2$  and connect edges from  $u_2$  to all the vertex in  $G_1$ . Name the new graph as  $G_2$  and repeat the above process like  $G_1$ . After continuing this process if we get the perfect match at  $k$ th instance i.e.,  $G_k$  has a perfect match then the size of maximum match for graph  $G_k$  will be  $(|V| - k)/2$ . Therefore, we successfully reduced maximum matching problem to perfect matching problem.

### 17.3.1 Hamiltonian Path and Hamiltonian Cycle

**Definition 17.1.** (Hamiltonian Path). A *Hamiltonian path* (in short H.P) in a graph G is a path of length  $n - 1$  where  $n$  is the number of vertices of graph G.

In other words, Hamiltonian path contains all vertices of graph G. Moreover, it is an open path. For the graph G given in Fig. 17.4 the vertex set  $a-b-c-d-e$  forms a Hamiltonian path. Now, we define Hamiltonian cycle in the following way.

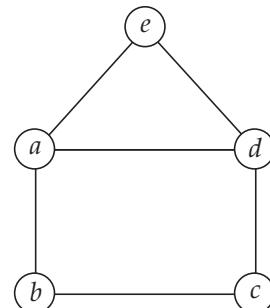


Fig. 17.4. Graph G.

**Definition 17.2 (Hamiltonian Cycle).** A *Hamiltonian cycle* (in short H.C.) in a graph G is a cycle which spans all vertices.

For the graph G given in Fig. 17.4 the vertex set  $a-b-c-d-e-a$  forms a Hamiltonian cycle. It is a longest close path.

**Example 17.2.** Polynomially reduce H.P to H.C (i.e.,  $H.P \leq_{poly} H.C$ ).

**Solution :** The reduction that we are going to perform can be described as : given an efficient algorithm for H.C, construct another efficient algorithm for H.P. The process of reduction is shown in Fig. 17.5.

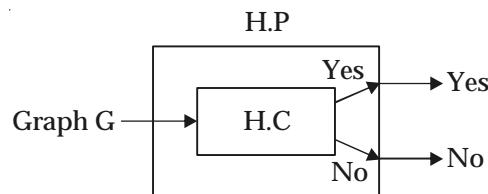


Fig. 17.5.  $H.P \leq_{poly} H.C$ .

Suppose take a graph G. Feed this to the problem of H.C. If the output is yes then the corresponding output for H.P is yes. This is because a graph which contains H.C also contains H.P. If the output to H.C is no then we cannot guarantee that the graph G has a H.P. To overcome this problem we need to perform the below transformation process.

In this process, we add a new vertex  $u_1$ . Then connect every vertex of G to  $u_1$ . Let the new graph formed is  $G_1$ . Now feed  $G_1$  to H.C. If H.C outputs yes then  $G_1$  has H.P. On the other hand if H.C outputs no then we need to adopt the transformation process again. Add a new vertex  $u_2$  and connect every vertex of  $G_1$  to  $u_2$ . Rename the new graph as  $G_2$ . Continue this process for  $k$  number of times (polynomial time) to successfully reduce H.P to H.C.

## 17.4 THE CLASSES : P AND NP

In the previous section, we have discussed few efficient algorithms (known as fast algorithms) which run in polynomial time. Based on these concepts we are now going to discuss two important classes **P** and **NP**.

**Definition 17.3 (Class P).** **P** is the class of decision problems that can be solved in polynomial time. In other words, given an instance of a decision problem in this class, there is a polynomial time algorithm for deciding if the answer is yes or no.

Let us take an example of a problem which is in class **P**. Suppose, we are given with a weighted graph G and a positive integer  $k$ . Our task is to figure out whether there is a shortest path from vertex  $u$  to  $v$  of length  $k$ ? The output to this problem is simply yes or no, since we can solve this problem by running any shortest path finding algorithm (let Dijkstra's algorithm that runs in  $O(E + V \log V)$ ) in polynomial time.

**Definition 17.4 (Class NP).** A decision problem is said to be in *class NP* (nondeterministic polynomial) if for all yes input there is a proof (in the form of advice/string) using which it can be verified in polynomial time that the input is indeed yes input.

To show a problem is in class **NP** we adopt a very tricky method which says : there are two players named Prover and Verifier. The job of these two players is to combinely prove a problem belongs to class **NP**. The following steps describes how these two carry out their jobs.

Prover	Verifier
<p>Prover is a powerful and intelligent person.</p> <ul style="list-style-type: none"> <li>(i) Prover takes the decision version of problem and outputs yes.</li> <li>(iii) Prover supplies a piece of evidence called certificate to Verifier.</li> </ul>	<p>Verifier has limited resources ; no too much time to verify any assigned task.</p> <ul style="list-style-type: none"> <li>(ii) Verifier asks Prover why this is so.</li> <li>(iv) Verifier verifies the evidence provided by Prover in polynomial time and announces success.</li> </ul>

Now we come to the biggest problem of 21st century :

**Does  $P = NP$ ?** In other words, it asks whether the classes **P** and **NP** are identical. This is an open unsolved problem in computer science. Informally it asks whether every problem whose solution be quickly verified (by a computer) can also be solved (by a computer). This problem is presented by Stephen Cook during 1971 in his research paper. Whoever will solve this problem correctly with valid justification will get US \$ 1000000 prize.

Hence, it is one of the millennium prize problems out of six open problems announced by Clay Mathematics Institute, USA. However many researchers proposed many solution and found  $P \neq NP$ .

**Example 17.3** Show that Hamiltonian cycle (H.C) decision problem belongs to class **NP**.

**Solution :** To proof Hamiltonian cycle is in **NP**-class we need to formulate its decision type as follows:

H.C (Decision type) :

Input : Graph G

Output : Does G has a H.C?

Prover and Verifier solves the problem in the following way.

Prover	Verifier
<p>(i) Prover checks graph G and announces that G has a H.C.</p> <p>(iii) Prover provides a set of edges <math>\langle u_1, u_2, \dots, u_n \rangle</math> which forms a H.C to Verifier for verification.</p>	<p>(ii) Verifier asks Prover why the graph G has a H.C ? Justify your answer.</p> <p>(iv) Verifier has to solve the problem by running a polynomial time algorithm for verification. The steps are given below.</p> <ul style="list-style-type: none"> <li>• Edges should come in sequence.</li> <li>• Check every vertex to be unique except the first vertex in given sequence.</li> <li>• Total number of vertex should be <math>n + 1</math>.</li> </ul>

Therefore, this shows the problem H.C is in NP class.

**Example 17.4** *Prove that Hamiltonian path (H.P) decision problem belongs to class NP.*

**Solution :** To show Hamiltonian path problem is in NP class we formulate its decision version as described below.

H.P (Decision type) :

Input : Graph G

Output : Does G has a H.P?

Prover and Verifier solves the problem in the following way.

Prover	Verifier
<p>(i) Prover is given the graph G. After checking Prover announces that G has a H.P?</p> <p>(iii) Prover supplies a set of edges <math>\langle u_1, u_2, \dots, u_n \rangle</math> to Verifier which forms a H.P in graph G.</p>	<p>(ii) Verifier asks Prover why the graph G has a H.P? Justify your answer.</p> <p>(iv) Verifier does the verification process in the following way.</p> <ul style="list-style-type: none"> <li>• Edges should come in sequence.</li> <li>• No repetition of vertex is allowed.</li> <li>• Total number of vertex should be <math>n</math>.</li> </ul>

As the verification process takes polynomial time the problem Hamiltonian path is in class NP.

**Example 17.5** *Prove that perfect match problem belongs to class NP.*

**Solution :** Recall that perfect match problem is defined as the matching such that all vertices end point of exactly one edge in matching. In order to show this problem belongs to class NP formulate its decision version which is given as follows.

**Perfect match (Decision type) :**

Input : Graph G

Output : Does G have a perfect match ?

Prover and Verifier solves this problem in the following way.

Prover	Verifier
<ul style="list-style-type: none"> <li>(i) Prover checks graph G and announces that graph G has a perfect match.</li> <li>(ii) Prover provides Verifier <math>k</math> number of set of edges which forms the perfect match.</li> </ul>	<ul style="list-style-type: none"> <li>(ii) Verifier asks Prover why this is so?</li> <li>(iv) Verifier does the verification process in the following way in polynomial time.           <ul style="list-style-type: none"> <li>• First checks whether <math>k</math> number of set of edges which indeed form a perfect match.</li> </ul> <math display="block">k = \frac{\text{total number of vertices in } G}{2}</math> <ul style="list-style-type: none"> <li>• Then Verifier takes an edge from perfect match set. Matches it with the set of all vertex in graph G. If match then put a cross mark (<math>\times</math>) on this vertex in graph G. Continue this process for <math>k</math> set of edges.</li> <li>• During the checking process if any vertex crossed once and found again to be crossed; this helps the Verifier to conclude that graph G doesn't have a perfect match.</li> </ul> </li> </ul>

As the verification process is performed in polynomial time; this concludes that the perfect match problem is in class **NP**.

## 17.5 NP-HARD AND NP-COMPLETE

---

In this section, we discuss two important classes of problems of complexity theory ; one is **NP-hard** and the other is **NP-complete**.

We formally define the class **NP-hard** in the following way.

**Definition 17.5 (Class NP-hard).** A problem  $\pi$  is in *class NP-hard* if and only if satisfiability (SAT) problem reduces to  $\pi$  (written as  $SAT \leq_{poly} \pi$ ). In otherwords if there is present an efficient polynomial time algorithm for  $\pi$  then there is one for every problem in NP.

**Note :** The problem SAT has discussed in section 17.6.

Recall that the shortest path problem belongs to class **P**. But the corresponding longest path problem is in **NP-hard**.

The problem says given a non-negatively weighted graph  $G$  and two vertices  $u$  and  $v$  then what is the longest path from  $u$  to  $v$  in the graph? Few other example of NP-hard problem are SAT, clique, independent set, vertex cover etc.

**Definition 17.6 (Class NP-complete).** A problem  $\pi$  is said to be in *class NP-complete* if and only if  $\pi$  is NP-hard and  $\pi$  is NP.

Example of NP-complete problem are : SAT (according to Cook's theorem), clique, independent set, vertex cover, graph coloring etc.

## 17.6 SATISFIABILITY (SAT)

---

Let get introduce to few basic terms before knowing SAT.

- **Boolean Variables :** There are two types of **boolean variables** i.e., True (or Yes) represented as T and the other is False (or No) represented as F. If  $x_i$  denote the boolean variable then its complement is denoted as  $\bar{x}_i$ .
- **Literal :** A variable ( $x_i$ ) or its complement ( $\bar{x}_i$ ) is called **literal**.
- **Propositional Calculus :** A formula in **propositional calculus** is an expression that is constructed by connecting literals using the operations logical AND ( $\wedge$ ) and logical OR ( $\vee$ ). Examples of propositional calculus are  $(x_1 \wedge \bar{x}_2) \wedge (x_2 \wedge x_3)$ ,  $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$  etc.
- **Clause :** A **clause** is a disjunction (OR) of literals.

$$c = x_1 \vee \bar{x}_2 \vee \dots \vee x_m$$

- **Boolean Formula :** A boolean formula is either a **conjunction** or **disjunction** of clauses. It is of two types.

(i) **Conjunctive normal form (CNF):** A boolean formula is in CNF iff it is represented as  $\wedge_{i=1}^m c_i$ ; where  $c_i$  are called clauses. So CNF in expanded form is shown as follows.

$$c_1 \wedge c_2 \vee c_3 \wedge \dots \wedge c_m$$

(ii) **Disjunctive normal form (DNF):** A boolean formula is in DNF iff it is represented as  $\vee_{i=1}^m c_i$ ; where  $c_i$  are called clauses. DNF in expanded form can be shown as given.

$$c_1 \vee c_2 \vee c_3 \vee \dots \vee c_m$$

- **Satisfiability Problem (SAT) :** Given  $m$  boolean variables  $x_1, x_2, \dots, x_m$  (each can be True or False), is there a way to assign values to variables so that a given boolean formula evaluates to True. The boolean formula could take any one of the two forms : CNF or DNF. The problem satisfiability is of two types.
  - If a boolean formula is of the form  $(x_1 \vee x_2 \vee \dots) \wedge (x_i \vee x_j \vee \dots) \wedge \dots \wedge (x_m \vee \dots \vee x_m)$  then the problem is known as **CNF-Satisfiability**. It is also represented as  $c_1 \wedge c_2 \wedge \dots \wedge c_m$ ; where each  $c_i$  is a clause.

→ If a boolean formula looks like  $(x_1 \wedge x_2 \wedge \dots) \vee (x_i \wedge x_j \wedge \dots) \vee \dots \vee (x_m \wedge \dots \wedge x_n)$  then the problem is known as **DNF-Satisfiability**. It is also represented as  $c_1 \vee c_2 \vee \dots \vee c_m$ ; where each  $c_i$  is a clause.

## 17.7 COOK'S THEOREM

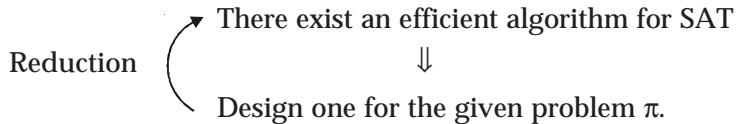
---

Cook's theorem (also known as **Cook-Levin Theorem**) was proposed by Stephen Arthur Cook during 1971. Cook is also known as the father of complexity theory. The theorem is given as follows.

**Theorem 17.1 (Cook's Theorem).** If there exist a polynomial time algorithm for SAT, there is one for all problems in NP.

**Proof :** At this stage as we may realize that there are potentially infinite number of problem in the class NP so it is not possible to design a reduction function for each problem. A detailed proof of Cook's theorem requires a computational model very precisely. This is probably beyond our scope. Therefore, we sketch an intuition behind the proof.

Given an arbitrary problem  $\pi \in \text{NP}$ , we want to show  $\pi \leq_{\text{poly}} \text{SAT}$ . So the reduction process looks like as follows.



In other words, given any instance (value) of  $\pi$ , let say  $I_\pi$ , we would like to define a boolean formula  $F(I_\pi)$  which has satisfiable assignment if  $I_\pi$  is a yes instance. Moreover, the length of  $F(I_\pi)$  should be polynomial time constructable.

A computing machine (such as Turing machine) is a machine where we have

- (a) A **start configuration**
- (b) A **final configuration** that indicates the output is accept or reject.
- (c) A sequence of **intermediate configuration**  $S_i$  where  $S_{i+1}$  follows from  $S_i$  using a valid transition.

In a non-deterministic system there can be more than one possible transition from a configuration. A nondeterministic machine accepts a given input iff there is some valid sequence of configurations that verifies the input is a yes instance.

All the above properties can be expressed in propositional logic, i.e., by an unquantified boolean formula in CNF. Using the fact that the number of transitions is polynomial, we can bound the size of this formula by a polynomial. The details can be quite messy and the interested reader can consult a formal proof in the context of Turing machine model.

Consider a situation where we want to write a propositional formula to assert that a machine is in exactly one of the  $k$  states at any given time  $1 \leq i \leq T$ . Let us use the boolean variables  $x_{1,i}, x_{2,i}, \dots, x_{m,i}$  where  $x_{j,i} = 1$  iff the machine is in state  $j$  at time  $i$ . Now we write a formula which will be a conjunction of two conditions.

(a) At least one variable is true at any time  $i$ .  $(x_{1,i} \vee x_{2,i} \vee \dots \vee x_{m,i})$

(b) At most one variable is true.

$$(x_{1,i} \Rightarrow \bar{x}_{2,i} \wedge \bar{x}_{3,i} \wedge \dots \wedge \bar{x}_{m,i}) \wedge (x_{2,i} \Rightarrow \bar{x}_{1,i} \bar{x}_{3,i} \wedge \dots \wedge \bar{x}_{m,i}) \dots \wedge (x_{m,i} \Rightarrow \bar{x}_{1,i} \wedge \bar{x}_{2,i} \wedge \dots \wedge \bar{x}_{m-1,i})$$

Here the implication  $a \Rightarrow b$  is equivalent to  $\bar{a} \vee b$ .

A conjunction of the above formula overall  $1 \leq i \leq T$  has a satisfiable assignment of  $x_{j,i}$  iff the machine is in exactly one state (not necessarily in the same state) at each of the time instances. The other condition should capture which states can succeed a given state.

We have argued that SAT is NP-hard. Since we can guess an assignment and verify the truth value in boolean formula in linear time, we can claim that SAT is in NP. Hence SAT is NP-complete.

## 17.8 COMMON NP-COMPLETE PROBLEMS WITH PROOFS

---

To proof a given problem  $\pi$  is NP-complete the standard procedure is to show two things.

- (i)  $\pi$  is NP. This seems to be easy in the light of Prover and Verifier.
- (ii) Again  $\pi$  is NP-hard. For that we need to perform a correct reduction from any known NP-hard problem to  $\pi$ .

In complexity theory there are thousands of NP-complete problems. The following are some commonly known problems of this class.

- Clique
- Independent set
- Vertex cover
- $\leq 3$  Vertex cover
- Exact cover

We perform the following reductions to show the above problems are NP-complete ; which will be discussed briefly later.

- SAT to Clique
- Clique to Independent set
- Independent set to Vertex cover
- Vertex cover to  $\leq 3$  Vertex cover
- $\leq 3$  Vertex cover to Exact cover

### 17.8.1 Clique Problem

The problem clique refers to any of the problems related to finding **complete subgraphs** in a graph. Let us define it formally.

**Definition 17.7 (Clique).** A *clique* in a graph  $G = (V, E)$  is a subset  $U$  of  $V$  such that  $\forall u_1, u_2 \in U$  and there exists an edge between  $u_1$  and  $u_2$ .

For the graph  $G$  given in Fig. 17.6 the vertex set  $\{a, b, f\}$ ,  $\{a, b, e\}$ ,  $\{a, b, d\}$  etc. forms a clique as there is an edge exists in between each pair of vertex in each set. Simply the vertex set which don't form clique are  $\{a, b, c, e, f\}$ ,  $\{a, b, c\}$ ,  $\{b, c, e, f\}$  etc.

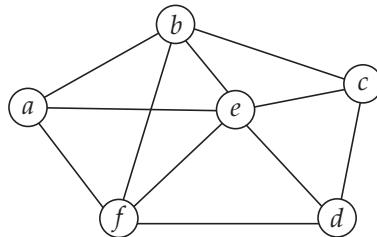


Fig. 17.6. Graph G.

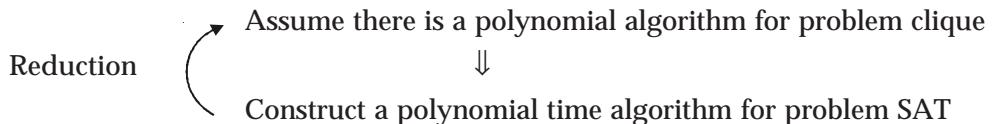
**Example 17.6** Prove that clique problem is NP-complete.

**Solution :** The problem clique is NP-complete provided it is NP and NP-hard. First we show clique is NP. To show clique is NP we adopt Prover-Verifier method in the following way.

Prover	Verifier
(i) Prover is given graph $G$ . After checking Prover announces that graph $G$ has a clique of size $k$ .	(ii) Verifier asks Prover why this is so ?
(iii) Prover supplies $k$ set of vertices of graph $G$ which forms a clique i.e., $\langle u_1, u_2, \dots, u_k \rangle$ to Verifier as replay.	(iv) Verifier job is to check whether there is an edge exist between each pair of vertices. Verifier checks in $k(k-1)/2$ time which is polynomial in nature.

As the verification process takes polynomial time therefore clique problem is NP.

Now we move to the second part of proof. To show clique is NP-hard we need to use the process of reduction; which takes place from SAT to clique. The process of reduction is show below.

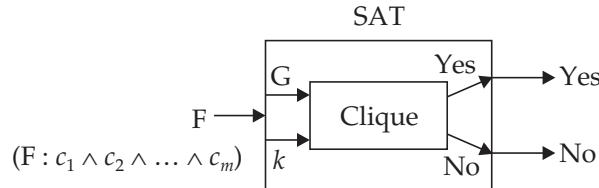


Cook's theorem tells if there is polynomial time algorithm for SAT then there is one for everything in NP. Based on these implications we proceed for reduction with the decision version of clique problem given as follows.

**Input :** Graph  $G$  and positive integer  $k$ .

**Output :** Does  $G$  have a clique of size  $k$ ?

The reduction from SAT to clique is given in Fig. 17.7. The input to SAT is a boolean formula which consists of clauses such as  $c_1, c_2, \dots, c_m$ . From these clauses we need to construct a graph  $G$  and a clique of size  $k$ . The transformation from clauses to a graph takes place in the following manner. For each clause  $c_i$  we will have a number of vertices. For example, the clauses  $c_1$  consists of vertices given as  $\langle c_1, x_1 \rangle, \langle c_1, \bar{x}_2 \rangle, \dots, \langle c_1, x_k \rangle$ .

Fig. 17.7.  $SAT \leq_{poly} \text{Clique}$ .

Here each vertex set consists of **clause-literal pair**. Based on these concepts we build a clique of size  $m$  as given in Fig. 17.8.

$$\begin{array}{ccccccc}
 F: & c_1 & \wedge & c_2 & \wedge & c_3 & \dots \dots \dots \wedge c_m \\
 & \langle c_1, x_1 \rangle & & \langle c_2, \bar{x}_1 \rangle & & \langle c_3, x_1 \rangle & \dots \dots \dots \langle c_m, - \rangle \\
 & \langle c_1, \bar{x}_2 \rangle & & \langle c_2, \bar{x}_2 \rangle & & \langle c_3, x_2 \rangle & \dots \dots \dots \langle c_m, - \rangle \\
 & \langle c_1, x_3 \rangle & & \langle c_2, x_3 \rangle & & \langle c_3, \bar{x}_3 \rangle & \dots \dots \dots \langle c_m, - \rangle \\
 & \vdots & & \vdots & & \vdots & \vdots \\
 & \langle c_1, x_{37} \rangle & & \langle c_2, x_{37} \rangle & & \langle c_3, \bar{x}_{37} \rangle & \dots \dots \dots \langle c_m, - \rangle \\
 & \vdots & & \vdots & & \vdots & \vdots \\
 & \langle c_1, x_k \rangle & & \langle c_2, - \rangle & & \langle c_3, - \rangle & \dots \dots \dots \langle c_m, - \rangle
 \end{array}$$

Fig. 17.8. Clique of size  $m$ .

Boolean formula  $F$  is true provided each individual clauses are true i.e.  $c_1, c_2, c_3, \dots, c_m$  all have to be true. Again  $c_1$  is true if any one of literals  $x_1, \bar{x}_2, x_3, \dots, x_{37}, \dots, x_k$  is true. Clause  $c_2$  is true if any one of  $\bar{x}_1, \bar{x}_2, x_3, \dots, x_{37}, \dots$  is true. Similar case for  $c_3, c_4, \dots, c_m$ . For every two vertex set  $\langle c_i, x \rangle$  and  $\langle c_j, y \rangle$  there exists an edge iff  $i \neq j$  and  $x \neq \bar{y}$ . Mark that  $i \neq j$ , this is because as we put an edge between vertices of different clauses. Finally  $k$  is equal with  $m$ . Thus a SAT problem has a solution if the corresponding vertices form a clique. We have successfully reduced SAT problem to clique problem. This shows the clique problem is NP-hard. We have previously shown it is in NP. Therefore, clique problem is NP-complete.

### 17.8.2 Independent Set Problem

We define the problem independent set as follows.

**Definition 17.8 (Independent Set).** Given a graph  $G = (V, E)$  a subset  $U \subseteq V$  is called an **independent set** if  $\forall u_1, u_2 \in U$ , and there shouldn't be an edge between  $u_1$  and  $u_2$ .

For the graph  $G$  given in Fig. 17.9 the independent sets are  $\{f, d, b\}, \{f, d\}, \{a, e, c\}, \{b, e\}$  etc. Clearly no two vertices in each set contains an edge. However the set  $\{a, b, c\}, \{a, g, c\}$ , and  $\{a, f\}$  don't form an independent set.

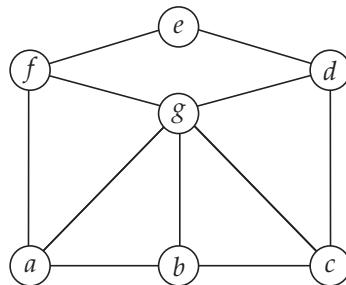


Fig. 17.9. Graph G.

**Example 17.7** Show that the problem independent set in NP-complete.

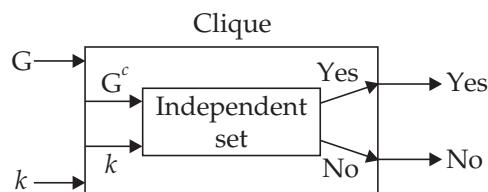
**Solution :** The problem independent set in NP-complete if it is NP as well as NP-hard. To show this problem is NP we adopt the traditional method of conversation between Prover and Verifier in the following way.

Prover	Verifier
(i) Prover is given a graph G. After checking he announces that G has an independent set of size k, where k is any positive integer.	(ii) Verifier asks Prover to justify the reason.
(iii) Prover supplies Verifier k set of vertices $\langle u_1, u_2, \dots, u_k \rangle$ of graph G which forms an independent set. This is a certificate to Verifier.	(iv) Verifier verifies the k set of vertices in $k(k-1)/2$ steps to figure out whether there exists an edge between each pair of vertices.

As the verification process gets performed in polynomial time the problem independent set belongs to NP.

Now to prove independent set problem belongs to NP-hard we reduce a clique problem which is already in NP-hard. It is clear that if there is a polynomial time algorithm for clique then there is one for everything in NP.

Suppose we have a polynomial time algorithm for independent set, then how do we find a clique of size k in graph G. The transformation looks like as follows. Take graph G and find its complement  $G^c$  (whenever there is an edge exist between two vertices we remove it and if there is no edge we add an edge). Then we feed  $G^c$  into the subroutine for independent set as shown in Fig. 17.10. If this says yes then the corresponding output is yes otherwise no.

Fig. 17.10. Clique  $\leq_p$  independent set.

This shows if  $G$  has clique of size  $k$  then  $G^c$  will have an independent set of size  $k$  otherwise  $G^c$  doesn't have an independent set of size  $k$ . So we successfully reduce the clique problem to independent set. Therefore the problem independent set is NP-hard. It is also in NP. Hence independent set problem is NP-complete.

### 17.8.3 Vertex Cover Problem

The formal definition of vertex cover problem is given as follow.

**Definition 17.9 (Vertex Cover)** . A *vertex cover* in a graph  $G$  is a set  $U$  of vertices such that every edge has at most one end point in  $U$ .

Consider the graph  $G$  given in Fig. 17.11. The graph has a vertex cover set  $\{a, b, f, d\}$ . This shows vertex  $a$  covers the edges  $ab$  and  $ag$ , vertex  $b$  covers the edges  $bc$ ,  $bf$ , and  $bg$ , vertex  $f$  cover  $fc$ ,  $fd$ ,  $fe$  and  $fg$  and finally vertex  $d$  cover  $dc$  and  $de$ . So the given set cover all the 11 edges of graph  $G$ .

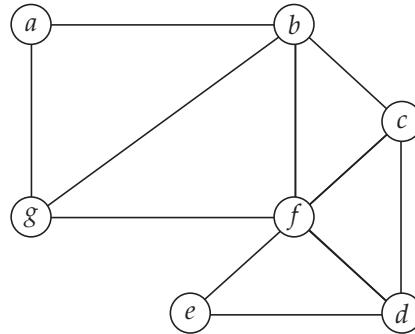


Fig. 17.11. Graph  $G$ .

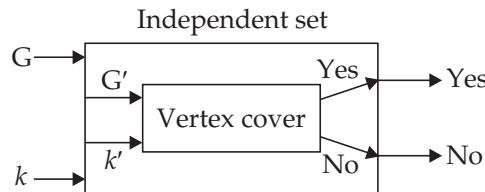
**Example 17.8** Prove that vertex cover problem is NP-complete.

**Solution :** To proof vertex cover problem is NP-complete we need to show it is NP as well as NP-hard. To show vertex cover is NP we take the help of Prover and Verifier in the following way.

Prover	Verifier
(i) Prover is given a graph $G$ . After Checking Prover announces that $G$ has a vertex cover of size $k$ .  (iii) Prover supplies Verifier $k$ set of vertices $\langle u_1, u_2, \dots, u_k \rangle$ of graph $G$ which forms a vertex cover. This is a certificate to Verifier.	(ii) Verifier asks Prover why the graph $G$ has a vertex cover of size $k$ ? Justify your answer.  (iv) Verifier simultaneously looks at the subset $\langle u_1, u_2, \dots, u_k \rangle$ and at every edge of graph $G$ . The job of Verifier is to check that for every edge at least one of the two end points belongs to the subset. If there are $n$ number of edges of graph $G$ then Verifier verifies in $O(nk)$ time.

As the verification process takes polynomial time ; therefore the problem vertex cover is **NP**.

Now we are keen to show vertex cover is **NP-hard**. For that we reduce a known problem in **NP-hard**, let say independent set to vertex cover. Take a graph  $G$  and a positive integer  $k$ . We want to figure out if  $G$  has an independent set of size  $k$ . For that we build a graph  $G'$  (a copy of graph  $G$ ) and take  $k' = n - k$  where  $n$  is the number of vertices of graph  $G$ . The vertex set of  $G'$  is the same as the vertex set of  $G$  (i.e.,  $V(G') = V(G)$ ) and edge set of  $G'$  is also same as the edge set of  $G$  (i.e.,  $E(G') = E(G)$ ). Now we feed the graph  $G'$  and  $k'$  to the subroutine of vertex cover as shown in Fig. 17.12. If this says yes then corresponding output for independent set is yes, otherwise no.



**Fig. 17.12.** Independent set  $\leq_p$  vertex cover.

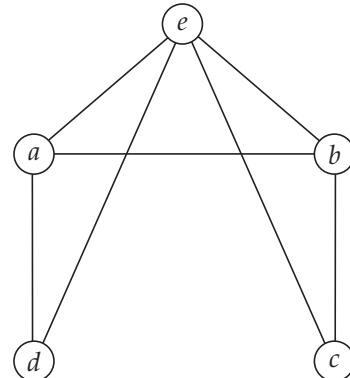
Now we can argue if  $G$  has a vertex cover of size  $n-k$  if and only if  $G$  has an independent set of size  $k$ , on the other hand if  $G$  doesn't have a vertex cover of size  $n-k$  then it doesn't have an independent set of size  $k$ . Therefore, we comprehensibly reduced the independent set problem to vertex cover problem. Hence vertex cover problem is **NP-hard**. It is also **NP**. So finally we get to the conclusion that it is **NP-complete**.

#### 17.8.4 $\leq 3$ Vertex Cover Problem

$\leq 3$  vertex cover (in short  $\leq 3$  VC) is the most general restricted form of vertex cover problem. The formal definition is given as follows.

**Definition 17.10 ( $\leq 3$  VC).** A  $\leq 3$  VC in a graph  $G$  is the set  $U$  of vertices such that maximum degree of any vertex belong to this set is  $\leq 3$ .

Consider the graph  $G$  given in Fig. 17.13. An example of  $\leq 3$  VC set for the graph  $G$  is  $\{a, b, c, d\}$ . The maximum degree of each vertex in this set is less than or equal to 3. However vertex  $e$  cannot be considered as  $\leq 3$  VC set since its degree is 4.



**Fig. 17.13.** Graph  $G$ .

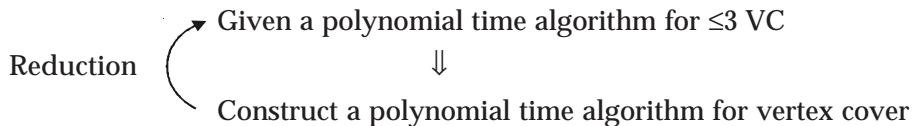
**Example 17.9** Show that  $\leq 3$  vertex cover is **NP-complete**.

**Solution:**  $\leq 3$  VC is **NP-complete** provided it is **NP** as well as **NP-hard**. To show it is **NP** we take the help of Prover and Verifier in the following manner.

Prover	Verifier
(i) Prover is given graph $G$ and a positive integer $k$ . After checking Prover announces that $G$ has $\leq 3$ VC of size $k$ .	(ii) Verifier asks Prover why this is so.
(iii) Now Prover gives Verifier $k$ set of vertices i.e., $\langle u_1, u_2, \dots, u_k \rangle$ for verification.	(iv) Verifier randomly takes any edge (let say $u_1, u_2$ ) from the given graph $G$ and checks one of these two end points present in the $k$ set of vertices. If there are total $m$ number of edges in graph $G$ then total number of attempts taken by Verifier for checking is $O(mk)$ .

As the verification process takes polynomial time the problem  $\leq 3$  VC is NP.

To show  $\leq 3$  VC is NP-hard we perform reduction technique in the following way.



While performing the reduction if all vertex degrees are less than or equal to three then we have no problem. We just feed this to the algorithm for  $\leq 3$  VC and we are done. However our focus is for those vertices whose degree is greater than 3. To handle this situation we use a trick **degree reduction** while controlling the size of the vertex. During the process of degree

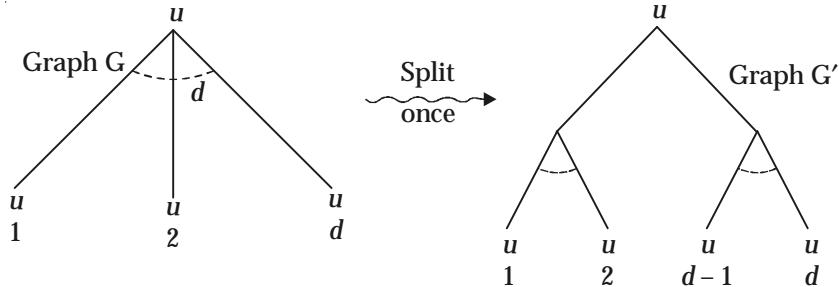


Fig. 17.14. Splitting graph  $G$ .

reduction we split each and every vertex so that its degree is  $\leq 3$ . Each time we are splitting means we are adding two new vertices to the graph. Then how many times can we split a vertex  $u$  having degree  $d > 3$ ? So the maximum number of time splitting occurs is  $d/2$ . If  $s$  is the number of splitting step then  $G_s$  is our final splitted graph where each vertex has degree less than or equals to 3.

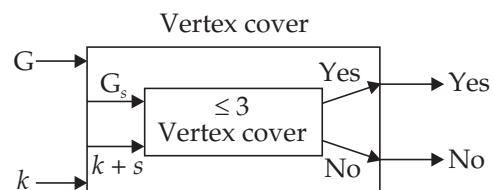


Fig. 17.15. Vertex cover  $\leq_p \leq 3$  VC

So  $G$  has a vertex cover of size  $k$  provided  $G_s$  has a vertex cover of size  $k+s$ . It is important that number of splitting steps to get  $G_s$  shouldn't be too large but polynomially bounded. This shows that we successfully reduced vertex cover problem to  $\leq 3$  VC. Therefore  $\leq 3$  VC is NP-hard. It is also NP. So we finally get to the conclusion that  $\leq 3$  VC is NP-complete.

#### 17.8.5 Exact Cover Problem

Exact cover problem is related to mathematics and the name suggests that it acts like a cover. It is a type of constraint satisfaction problem (CSP) which is represented by a set of objects whose state must satisfy a number of constraints or limitations. So the exact cover problem can be generalized slightly to involve exactly one problem at a time.

**Definition 17.11 (Exact Cover).** Given a collection  $S$  of subsets of a set  $A$ , an *exact cover* of  $A$  is a subcollection  $S^*$  of  $S$  that satisfies two conditions.

- (i) The *union* of the subsets in  $S^*$  in  $A$ , i.e., the subsets in  $S^*$  cover  $A$ .
- (ii) The *intersection* of any two distinct subsets in  $S^*$  is empty.

**Note :** The word exact signifies each element in  $A$  is contained in exactly one subset in  $S^*$ .

Let  $S = \{M, N, O, P\}$  be a collection of subsets of a set  $A = \{1, 2, 3, 4\}$  such that the sets  $M, N, O, P$  are given as follows.

$$\begin{aligned} M &= \{\} \\ N &= \{1, 3\} \\ O &= \{2, 4\} \text{ and} \\ P &= \{2, 3\}. \end{aligned}$$

Clearly the subcollection  $\{N, O\}$  is an exact cover of  $A$ , since the subsets  $N = \{1, 3\}$  and  $O = \{2, 4\}$  are disjoint and their union is  $A = \{1, 2, 3, 4\}$ . Again the subcollection  $\{M, N, O\}$  is also an exact cover of  $A$ . But the subcollection  $\{M, N, O\}$  is also an exact cover of  $A$ . But the subcollection  $\{N, P\}$  is not an exact cover of  $A$ . The intersection of the subsets  $N$  and  $P$  is  $\{3\}$ . So the subsets  $N$  and  $P$  are not disjoint. Moreover the union of the subsets  $N$  and  $P$  is  $\{1, 2, 3\}$  which is not equal to set  $A$ . Similarly the subcollection  $\{M, N, P\}, \{O, P\}$  and  $\{M, O, P\}$  are not exact cover of set  $A$ .

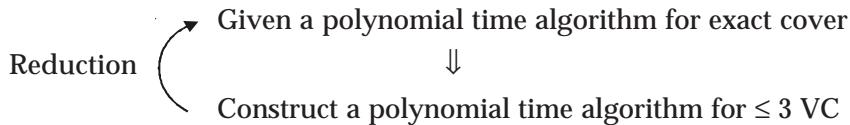
**Example 17.10** Prove that exact cover problem is NP-complete.

**Solution :** To show exact cover problem is NP-complete we need to prove it is NP as well as NP-hard. We adopt the typical Prover and Verifier method to show exact cover problem is NP in the following way.

Prover	Verifier
<p>(i) Prover is given set S which is a collection of subsets of a set A = {A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>}. Prover announces that yes there exists a subcollection such that the union is S and the sets in the subcollection are disjoint.</p> <p>(iii) Prover provides Verifier with all sets in this subcollection (let say S*).</p>	<p>(ii) Verifier asks Prover to provide proper evidence for verification.</p>
	<p>(iv) Verifier verifies that the intersection of any two subsets are disjoint and the union is S in polynomial time.</p>

Since the verification process gets performed in polynomial time the problem exact cover is **NP**.

To show exact cover is **NP-hard** we perform a reduction from problem  $\leq 3$  VC in the following way.



Recall that the inputs to  $\leq 3$  VC problem are graph G and positive integer k with the constraint that the maximum degree of each vertex is less than or equal to 3. In this case we are interested to figure out whether the graph has vertex cover of size k. To solve this problem we use exact cover as subroutine. In case of  $\leq 3$  VC we require vertices to cover all edges whereas in case of exact cover problem we require subsets to cover all elements of a set. Then how can we relate these two problems ?

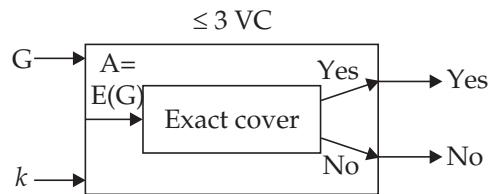


Fig. 17.16.  $\leq 3$  VC  $\leq_{poly}$  Exact cover.

Consider the transformation process given in Fig. 17.16. Our input is graph G and k to the problem  $\leq 3$  VC. Then, we create edge set of graph G (written as E (G)) as set A. The subsets of set A must correspond to vertices of graph G. For example, take a set A<sub>v</sub> (for vertex v). The elements of set A<sub>v</sub> are the edges incident on vertex v. So picking a subset from set A<sub>v</sub> corresponds to pick a vertex in graph G. But make sure that only k number of vertices can pick. For this we take k numbers of extra elements. Now set A becomes

$$A = E(G) \cup \{x_1, x_2, \dots, x_k\}$$

The collection of set A becomes

$$\begin{array}{lll}
 A_1 \cup \{x_1\} & A_2 \cup \{x_1\} & \dots \dots \dots A_n \cup \{x_1\} \\
 A_2 \cup \{x_2\} & A_2 \cup \{x_2\} & \dots \dots \dots A_n \cup \{x_2\} \\
 \vdots & \vdots & \vdots \\
 \vdots & \vdots & \vdots \\
 A_2 \cup \{x_k\} & A_2 \cup \{x_k\} & \dots \dots \dots A_n \cup \{x_k\}
 \end{array}$$

Note that  $n$  is the number of vertices of graph G. The number of collection of set A has grown by a factor of  $k$ . Initially we have  $n$  sets and now we have total  $nk$  sets. To pick an exact cover from the above collection we perform the following steps :

**Step 1** : Pick only one set across each column.

**Step 2** : Pick only one set across each row. But note that the two picks of step 1 and step 2 are different.

**Step 3** : Finally we need to pick a set which contains  $x_1$ , another set which contains  $x_2$ , another set contains  $x_3$  and so on. So there are  $k$  numbers of set formation are possible.

Well, is there any other problem related to our transformation procedure ! Yes there is. What happened if there is an edge exists between any two vertices in vertex cover ? Whenever we pick the corresponding sets for the vertices then this get covered twice. To overcome this problem consider all the possible subsets of edges incident on each vertex  $u$ . If vertex  $u$  is of degree  $d$  then the total number of subsets corresponding to this will be  $2^d - 1$ . Hence we successfully reduced  $\leq 3$  VC to exact cover problem. So it is NP-hard. Previously we have proved it is NP. Therefore, exact cover problem is NP.

## 17.9 OTHER USEFUL NP-COMPLETE PROBLEMS

---

Apart from Richard Karp's 21 NP-complete problems there are literally thousands of different problems have been proved to be NP-complete. We can find many such problems in the book titled by "Computers and Intractability : A Guide to the Theory of NP-completeness" which was written by Mike Garey and David Johnson in 1977. In this section, we list an intuition behind few NP-complete problems.

- **Subset sum** : Given a set S of  $n$  numbers of elements having sizes  $s_1, s_2, \dots, s_n$  and we are also given a positive integer  $k$ . We need to figure out is there a subset T of S whose size is  $k$ ? The size of the subset T is defined as

$$\text{size}(T) = \sum_{e \in T} \text{size}(e)$$

In the subset sum problem we would like to pick up the subset of S whose size is exactly  $k$ .

- **Partition** : Given a set  $S$  of  $n$  integers, are there subsets  $A$  and  $B$  such that

$$A \cup B = S, A \cap B = \emptyset \text{ and } \sum_{a \in A} a = \sum_{b \in B} b?$$

This is the partition problem which is **NP**-complete.

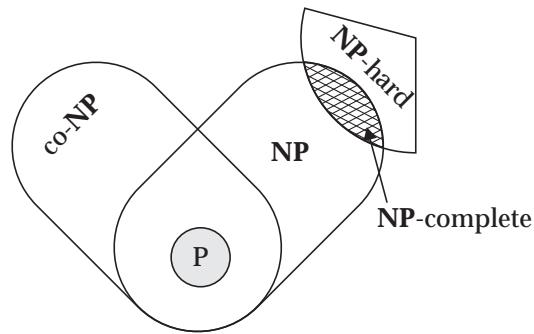
- **3 SAT** : It is a special case of SAT. In SAT we use CNF to show it is **NP**-complete whereas in case of 3SAT we use 3CNF. A 3CNF is a CNF formula with exactly three literals per clause. So 3SAT is just SAT restricted to 3CNF formulas which asks : given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to true?
- **Set cover** : Given a collection of set  $S = \{S_1, S_2, \dots, S_m\}$ , find the smallest sub-collection of  $S_i$ 's that contains all the elements of  $\bigcup S_i$ .
- **Hitting set** : Given a collection of set  $S = \{S_1, S_2, \dots, S_m\}$ , find the minimum number of element of  $\bigcup S_i$  that hit every set in  $S$ .
- **Longest path** : Given a non-negatively weighted graph  $G$  and two vertices  $u$  and  $v$ , what is the longest simple path from  $u$  to  $v$  in the graph ? A path is simple if it visits each vertex at most once. This is the generalization of Hamiltonian path problem. However the corresponding shortest path problem is in class **P**.

## 17.10 OTHER IMPORTANT COMPLEXITY CLASSES

While the classes, **P**, **NP**, **NP-hard** and **NP**-complete hogs the maximum limelight in complexity theory, there are many other classes in their own right. Some of them are described as follows.

- **co-NP** : A problem whose complement is in **NP** belongs to this class. In other words, if the answer to a problem in co-NP is no, then there is a proof of this fact that can be verified in polynomial time.

Even if we can verify every yes answer quickly, there is no reason to think that we can also verify no answers quickly. For instance how would you verify that a boolean formula is unsatisfiable (all assignments make it false) ? It is generally believed that **NP** = co-NP, but nobody knows how to prove it. It is also an open question is whether **NP** and co-NP are different.



**Fig. 17.17.** Relationship between five complexity classes.

- **PSPACE** : This is a special class of complexity theory which contains problem that run in polynomial space but not necessarily polynomial time. The satisfiability of **quantified boolean formula** (QBF) problem belongs to this class.
- **Randomized classes** : Depending on the type of **randomized algorithms** we have the following important classes.

**RP** : RP stands for **randomized polynomial** class of problems which are characterised by randomized algorithms.

**BPP** : The class BPP stands for **bounded error probabilistic polynomial** time. It consists of all languages L which can be recognized by a randomized polynomial time algorithm, with at most an exponentially small error probability on every input.

**ZPP** : The class ZPP stands for **zero error probabilistic polynomial** time. These don't have any errors in the output but the running time is expected polynomial time.

## 17.11 APPROXIMATION ALGORITHMS

---

In real life situation we come across many NP-complete problems and solving those problems is a greatest challenge. Usually NP-completeness arise when we talk about optimization problems. So if finding an optimal solution is NP-complete, then does it possible to get nearly optimal solution in polynomial time ? Of course. This approach seems to be a promising one and it is also helpful to find fast **approximation algorithms**. We are much more interested in finding approximately optimal solution and therefore we will be devising algorithms which are called approximation algorithms.

Let P denotes the optimization problem which asks **minimize** the **objective functions** subject to following constraints where

$A(i)$  : cost of solution found by an algorithm A on instance  $i$ .

$OPT(i)$  : cost of an optimal solution on instance  $i$  where we assume  $OPT(i) > 0$ .

The **approximation ratio** is given as  $\rho_i = \frac{A(i)}{OPT(i)} \geq 1$ .

So **approximation factor** of A is given as  $\rho(n)$  and given as

$$\rho(n) = \min \{\rho_i \mid i \text{ is the instance of size } n\}$$

Our objective is to design algorithm such that  $\rho(n)$  is small for large  $n$ . The resultant algorithm A is an **approximation algorithm**.

Sometimes we want to **maximize** the **objective function**. At this situation  $\rho_i = OPT(i)/A(i) \geq 1$  and we design an approximation algorithm such that  $\rho(n)$  is maximum for large  $n$ . Now lets discuss different applications of approximation algorithm to solve various problems.

### 17.11.1 Travelling Salesman Problem

In this section, we device an **approximation algorithm** for travelling salesman problem (in short TSP) which tells a traveling salesman has to travel through a bunch of cities in such a way such that the expenses on traveling is minimized. In other words the connected cities forms a complete undirected graph  $G = (V, E)$  that has non-negatively integer cost associated with each edge  $E$ . From this we need to find hamiltonian cycle having minimum cost. Therefore, some authors name this problem as **metric TSP problem** or **minimum Hamiltonian cycle**.

The input and output to the TSP are defined as follows.

**Input :** A graph  $G$  specified as an matrix  $D$  in which  $d[i, j]$  denotes the distance between vertex  $i$  and vertex  $j$  such that  $D$  forms a metric i.e.,

- (1)  $\forall i : d[i, i] = 0$  (distance from a node to itself)
- (2)  $\forall i, j : d[i, j] = d[j, i]$  (distances have to be symmetric)
- (3)  $\forall i, j, k : d[i, j] \leq d[i, k] + d[k, j]$  (triangle inequality)

**Output :** A cycle in the graph passing through all vertices exactly once such that the sum of the distances associated with the edges in the cycle is as small as possible.

Now we solve TSP using two approaches : brute-force search method and approximation algorithm in the following manner and then compare these two techniques.

**TSP using Brute-Force Search :** Suppose there are five cities named as  $a, b, c, d$  and  $e$ . All cities are interconnected with each other as shown in Fig. 17.18 as given in graph  $G$ . A salesman given the job of selling a product by visiting all five cities with an optimized cost.

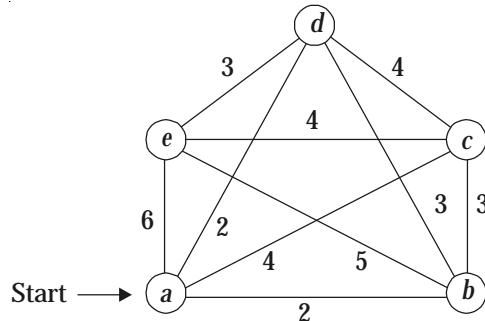


Fig. 17.18. Graph  $G$  for TSP problem.

The salesmen starts his journey from city  $a$ . From city  $a$  he has four choices to visit i.e.,  $b, c, d$  and  $e$ . Among these four he has to visit one city. Suppose he visits city  $b$ . Then from  $b$  he has three choices to visit i.e., city  $c, d$  and  $e$  (since he has already visited city  $a$  and  $b$ ) and so on. Likewise we represent the choices and order of visit of travelling salesman from city  $a$  as the tree shown in Fig. 17.19. This is moreover a brute-force type of search.

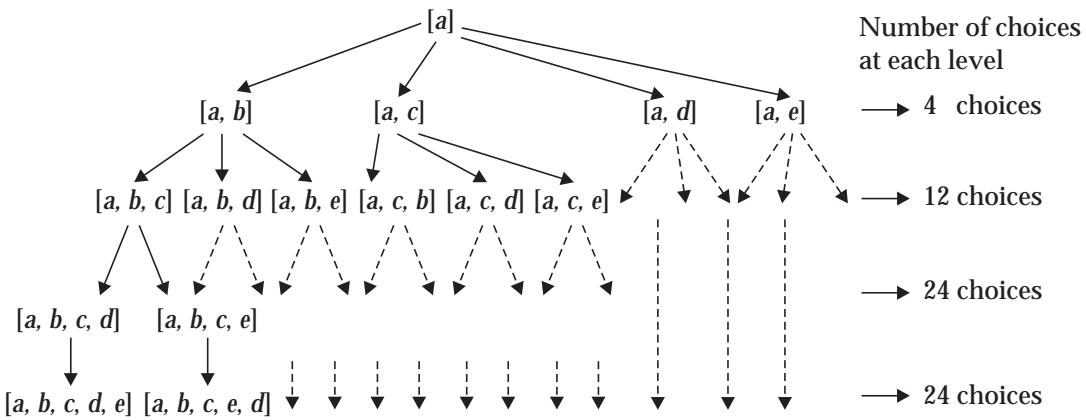


Fig. 17.19. Traversal order from city a and number of choices at each level for graph G.

The salesman can travel 24 different ways (such as  $[a, b, c, d, e]$ ,  $[a, b, c, e, d]$ ,  $[a, b, d, c, e]$ ,  $[a, b, d, e, c]$ , ....) to cover all five cities. Note that at this case the journey of salesman started from city a. He can start his journey from any one of the cities like b, c, d, and e. Therefore the total number of choices  $= 24 \times 5 = 120 = 5!$ . So we get to the conclusion that the number of choices is equal to the factorial of the number of cities.

But at this context if anyone interested to find out the optimum one (least cost) among 120 choices then it may take one or two days to manually calculate and get the result. The time taken for calculation is purely exponential in nature for such a small value of  $n$  (i.e.,  $n=5$ ). For larger value of  $n$ , the task of finding the optimum one will be a tedious job. Therefore, we need approximation algorithm which may not always produce the exact solution but it can provide solution which is very close to the exact solution. By the virtue of this we reduce searching time so that the task can be completed in time. The approximation algorithm steps for TSP is given as follows.

#### Approximation Algorithm for TSP :

- Step 1:** Arbitrarily choose as one of the vertices as start vertex for the given graph G.
- Step 2:** Find the MST (minimum spanning tree) of graph G by using Prim's algorithm taking the start vertex as source vertex.
- Step 3:** Find the sequence of vertices to be visited from source vertex by using DFS traversal.
- Step 4:** This sequence of vertices is the order to be visited by the salesman and make it cycle (known as **minimum Hamiltonian cycle**).

Then how much time does the algorithm take to execute. Clearly step 2 takes  $O(E + V \log V)$  which is equal to the running time of Prim's algorithm for MST. Step 3 takes  $O(V + E)$  time to execute. So the total running time is dominated by the time for finding minimum spanning tree ; therefore, it is  $O(|E| + |V| \log |V|)$ . It is a polynomial time.

**Example 17.11** Solve the TSP for graph G given in Fig. 17.18. Take vertex c as the start vertex. (Using approximation algorithm)

**Solution :**

**Step 1 :** Start vertex is c.

**Step 2 :** The MST for graph G using Prim's algorithm is given in Fig. 17.20.

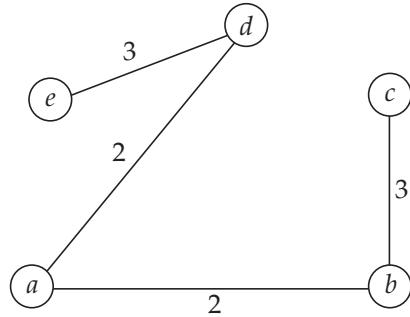


Fig. 17.20. MST for graph G.

**Step 3 :** We need to perform DFS traversal on MST. For that we need to find adjacency list which is given in Table 17.1 and the DFS traversal is given in Fig. 17.21. The sequence of edges are given by c-b-a-d-e.

Table 17.1. Adjacency List

Vertex	Adjacent vertex
a	b, d
b	a, c
c	b
d	a, e
e	d

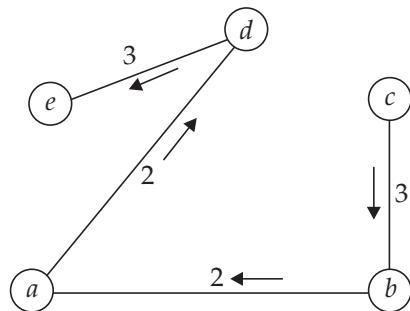


Fig. 17.21. DFS traversal.

**Step 4 :** So the minimum Hamiltonian cycle is given as  $c - b - a - d - e - c$ . This is the sequence in which the travelling salesman has to travel to optimise the cost of travel. Therefore total cost is 14 which is approximately minimum.

### 17.11.2 Vertex Cover Problem

Recall that in section 17.8.3, we have shown the problem vertex cover is NP-complete. In this section, we develop an approximation algorithm for vertex cover problem to find vertex cover of minimum size for a given graph  $G$ . In fact we will use **greedy strategy** to develop **greedy approximation** for vertex cover problem. The steps of greedy approximation algorithm are given as follows.

#### Greedy Approximation for Vertex Cover

- Step 1 :** Let  $C$  be an empty set (called cover). Select the vertex with maximum degree from given graph. Put this vertex in set  $C$ .
- Step 2 :** Delete all the edges that are incident to this vertex since they have been covered.
- Step 3 :** Repeat this operation on the remaining graph, until no more edges remain.

As we are concentrating only on maximum degree vertex our algorithm will always take polynomial time to get the result. At worst cast it takes  $O(\log V)$ , where  $V$  is the number of vertices of given graph  $G$ .

**Example 17.12** Find the vertex cover of minimum size of graph  $G$  given in Fig. 17.22.

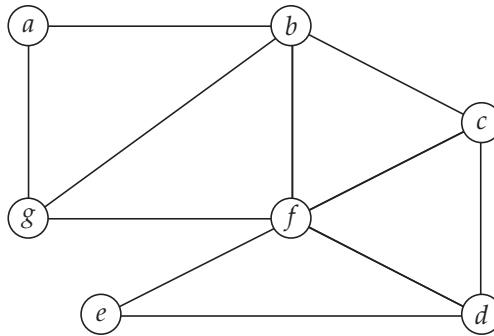
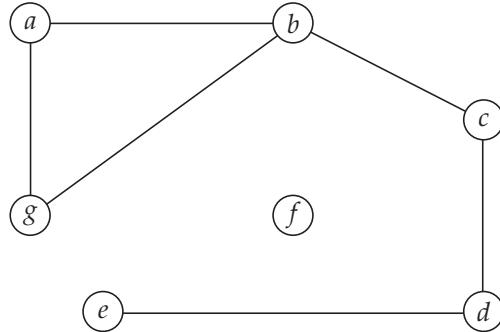


Fig. 17.22. Graph  $G$  for Example 17.12.

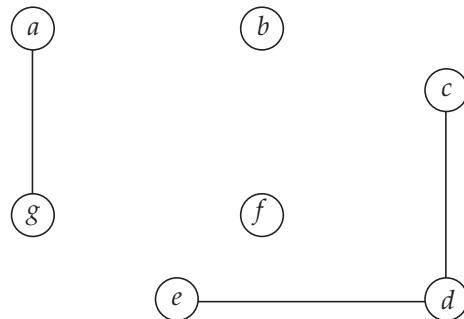
**Solution :**

- Step 1 :** Let  $C$  be the set denote the cover for graph  $G$ . Initially the set  $C$  is empty. The maximum degree vertex of graph  $G$  is  $f$ . So  $C = \{f\}$ .
- Step 2 :** Delete all the edges incident of  $f$  as shown in Fig. 17.22.

**Fig. 17.23.** Deleting the edges incident on vertex f.

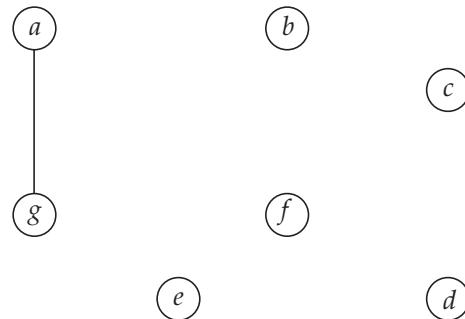
Now the vertex of maximum degree is  $b$ . So cover set  $C = \{f, b\}$

**Step 3 :** Delete all the edges incident on  $b$  as shown in Fig. 17.24.

**Fig. 17.24.** Deleting the edges incident on vertex b.

At this stage the vertex having maximum degree is  $d$ . Cover set  $C = \{f, b, d\}$ .

**Step 4 :** Delete all the edges incident on vertex  $d$ . The resultant figure is shown in Fig. 17.25.

**Fig. 17.25.** Deleting the edges incident on vertex d.

Now both vertex  $a$  and  $g$  are having maximum degree. We take vertex  $a$  and add this to cover set  $C = \{a, b, d, f\}$ .

**Step 5 :** Delete the edge incident on vertex  $a$ . Therefore, we remain with no edge. Hence vertex cover of minimum size set  $C = \{a, b, d, f\}$ .

### 17.11.3. 0 –1 Knapsack Problem

Recall that in section 13.4, we have discussed about knapsack problems. We can solve fractional knapsack problem by using greedy approach but for 0-1 knapsack problem we don't have. Therefore 0-1knapsack problem is NP-complete. In this section, we outline an approximation algorithm based on dynamic programming approach.

Imagine that a thief breaks into a jewellery shop and find  $n$  items. Let  $v_i$  denote the value of the  $i$ -th item and let  $w_i$  denote the weight of the  $i$ -th item. The thief carries a knapsack capable of holding total weight  $W$ . The thief wishes to carry away the most valuable subset items subject to the weight constraint. For example, the thief would rather steal diamond before gold because the value per pound is better. But he would rather steal gold before lead for the same reason. We assume that the thief cannot take a fraction of an object, so he must make a decision to take the object entirely or leave it behind.

In other words, given  $(v_1, v_2, \dots, v_n)$  and  $(w_1, w_2, \dots, w_n)$  and  $W > 0$ , we wish to determine the subset  $T \subseteq \{1, 2, \dots, n\}$  that maximizes  $\sum_{i \in T} v_i$  subject to

$$\sum_{i \in T} w_i \leq W$$

Here  $v_i$ 's,  $w_i$ 's and  $W$  all are positive integers. It turns out that this problem is NP-complete. But however, we have an reasonable approximation algorithm to solve this problem for smaller value of  $w_i$ 's and  $W$ . This algorithm uses dynamic programming technique as described follows.

We construct an array  $V[0 \dots n, 0 \dots W]$ . For  $1 \leq i \leq n$  and  $0 \leq j \leq W$ , the entry  $V[i, j]$  will store the maximum value of any subset of objects  $\{1, 2, \dots, i\}$  which can fit into a knapsack of weight  $j$ . If we can compute all the entries of this array, then the array entry  $V[n, W]$  will contain the maximum value of all  $n$  objects that fit into the entire knapsack of weight  $W$ .

To compute the entires of the array  $V$  we will imply an inductive approach. If we have no item then  $V[0, j] = 0$  for  $0 \leq j \leq W$ . Therefore the following two cases are obvious.

**Don't take object  $i$  :** If we choose not to take object  $i$ , then we get the optimal value by considering how to fill a knapsack of size  $j$  with remaining objects  $\{1, 2, \dots, i-1\}$ . This is just  $V[i-1, j]$ .

**Take object  $i$  :** If we take  $i$ , then we gain a value of  $v_i$  but have used up  $w_i$  of our capacity. With the remaining  $j-w_i$  capacity in the knapsack, we can fill it in the best possible way with objects  $\{1, 2, \dots, i-1\}$ . This implies  $v_i + V[i-1, j-w_i]$ . This is only possible if  $w_i \leq j$ .

From the above two cases we can formulate the following rules for constructing the array V. The ranges on  $i$  and  $j$  are given as  $i \in [0 \dots n]$  and  $j \in [0 \dots W]$ .

$$V[0, j] = 0$$

$$V[i, j] = \begin{cases} V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], v_i + V[i-1, j-w_i]) & \text{if } w_i \leq j \end{cases}$$

Now it is very easy to take these rules to construct an algorithm that computes the maximum value for the knapsack in time proportional to the size of the array  $O((n+1)(W+1)) = O(nW)$ .

The step of approximation algorithm are given follows.

#### Dynamic Programming Approximation Algorithm for 0-1 Knapsack Problem

```

Knapsack (v [1..... n], w [1.....n], n, W)
1. allocate V [0.....n, 0..... W]
2. for j = 0 to W do
3.   do V [0, j] = 0 // initialization
4. for i = 1 to n do
5.   don't_take_val = V[i-1, j] // total value without taking i
6.   if (j ≥ w[i]) // enough capacity to take i
7.     take_val = v[i] + V[i - 1, j-w[i]] // total value if we take i
8.   else take_val = NULL // not possible to take i
9.   V[i, j] = maximum (don't_take_val, take_val) // final value
10. return V[n, W]

```

The above algorithm takes  $O(nW)$  time, where  $n$  is the number of items and  $W$  is the maximum weight of items that the thief can put in knapsack.

**Example 17.13** Solve the following 0-1 knapsack problem using dynamic programming.

Value of the objects = (10, 40, 30, 50)

Weights of the objects = (5, 4, 6, 3)

Capacity of the knapsack = 10

Number of objects = 4

**Solution :** Given values of objects =  $(v_1, v_2, v_3, v_4) = (10, 40, 30, 50)$

weights of objects =  $(w_1, w_2, w_3, w_4) = (5, 4, 6, 3)$

capacity of knapsack =  $W = 10$

number of objects =  $n = 4$

We construct the Table 17.2 to solve the knapsack problem by using dynamic programming.

**Table 17.2** Dynamic Programming for 0-1 Knapsack

Capacity →			0	1	2	3	4	5	6	7	8	9	10
Item	Value	Weight	0	1	2	3	4	5	6	7	8	9	10
1	10	5	0	0	0	0	0	10	10	10	10	10	10
2	40	4	0	0	0	0	40	40	40	40	40	50	50
3	30	6	0	0	0	0	40	40	40	40	40	50	70
4	50	3	0	0	0	50	50	50	50	90	90	90	90

The final result is  $V[n, W] = V[4, 10] = 90$ . This show the selection of items 2 and 4 of values 40 and 50 respectively and the total weight the knapsack can hold is 7.

## 17.12 DIFFERENT APPROXIMATION SCHEMES

**Definition 17.12** (Approximation Scheme). The *approximation scheme* of an algorithm is the tradeoff between running time and the approximation factor.

Mainly approximation scheme is of two types. They are given as follows.

- Polynomial time approximation scheme (PTAS) and
- Fully polynomial time approximation scheme (FPTAS)

We formally define these two as follows.

**Definition 17.13** (PTAS). An algorithm A for a problem P is said to be *polynomial time approximation* scheme (PTAS) if

- (i) A takes two arguments : The problem instance and a number  $\varepsilon > 0$ .
- (ii) A turns a solution with approximation ratio  $1 + \varepsilon$ .
- (iii) The time taken by A is polynomial in the length of the instance for any fixed  $\varepsilon$ .

**Definition 17.14** (FPTAS). An algorithm A for the problem P is said to be a fully polynomial time approximation scheme (FPTAS) if the time taken by A is polynomial in  $1/\varepsilon$ .

So these are the two schemes which include a family of algorithms for different values of  $\varepsilon$ . For example, FPTAS for the knapsack problem runs in  $O(n^3/\varepsilon)$ ; where  $n$  denotes the size of input instance. This running time is polynomial in nature. Moreover knapsack problem is NP- complete. But the proof is not required at this point.

### 17.13 HISTORY OF NP-COMPLETENESS

---

We are at the concluding stage of the chapter. Before ending up here is a quick recap of few dazzling moments in the history of NP-completeness as given in Table 17.3.

**Table 17.3** NP-completeness Theory History

Year	Event
1971	Stephen Cook published his paper “The complexity of theorem proving procedures”, which provided the first published NP-complete results.
1972	Richard Karp published his paper “Reducibility among combinatorial problems”. This paper was first to demonstrate the concept of NP-completeness.
1979	Michael Garey and David Johnson published the famous book on NP-completeness titled “Computer and Intractability : A guide to the theory of NP-completeness”.
1990	Godel's lost letter found and popular blog on his letter $P = NP$ was created. Weblink : <a href="http://rilipton.wordpress.com">http://rilipton.wordpress.com</a>
1992	Annual prize for outstanding journal papers in theoretical computer science was named after Godel prize.
1997	First book published on “Approximation algorithm for NP-hard problems” written by Dorit Hochbaum.

### CHAPTER NOTES

---

- An algorithm is said to be efficient if its running time is polynomial in nature.
- Reduction technique is basically used to proof NP-complete results. Different reduction techniques are polynomial reduction, many-one reduction, log-space reduction, etc.
- In decision version of problem we always find the answer in the form of yes or no; whereas in case of search version we always try to find the answer in the form of reason. In case of optimization problem our focus will be on the optimal value of given cost function.
- A problem is in class **P** if it is solvable in polynomial time.
- A problem is in class **NP** if it is verified in polynomial time.
- To show a problem  $\pi$  is NP-hard, reduce a known NP-hard problem to  $\pi$ .
- A problem  $\pi$  is NP-complete if it is NP as well as NP-hard.
- According to Cook's theorem SAT is NP-complete.
- Examples of different NP-complete problem are clique, independent set, vertex cover,  $\leq 3$  vertex cover and exact cover.

- An approximation algorithm doesn't necessarily return an optimal solution, but settles for some feasible solution which one hopes will be near optimal. Approximation algorithm exists for travelling salesmen problem (TSP), vertex cover problem and 0-1 knapsack problem.

## EXERCISES

---

1. Show that  $P \subseteq NP$ .
2. What is reduction ? Discuss different types of reduction techniques.
3. Polynomially reduce a perfect matching decision problem to a maximum matching decision problem.
4. Prove that the problem Hamiltonian cycle is **NP**-complete. (Hint : Use reduction from vertex cover).
5. Prove that the problem Hamiltonian path is **NP**-complete.
6. What are class **P**, class **NP**, class **NP-hard** and **NP-complete** set of problems?
7. Show that clique problem is **NP**-complete.
8. Show that independent set problem is **NP**-complete.
9. Show that the following problem is **NP**-complete.  
**Input :** Graph  $G$  and a positive integer  $k$ .  
**Output :** Does  $G$  has a vertex cover of size  $k$ ?
10. What is approximation algorithm ? Give the approximation algorithm for travelling salesman problem.
11. What is vertex cover for an undirected graph? Write the approximation algorithm for vertex cover problem. What is its running time?
12. Write the approximation algorithm for 0-1 knapsack problem. What is the running time?
13. Solve the following 0-1 knapsack problem using dynamic programming :  
Value of items = (30, 120, 100, 50, 110)  
Weights of items = (5, 3, 4, 2, 6)  
Capacity of knapsack = 12  
Number of items = 5
14. What is approximation scheme? Discuss different types of it.

# MATHEMATICAL BACKGROUND

## Algorithm Analysis

Now we will review some of the basic elements of algorithm analysis, which were covered in chapters 1, 2 and 3. These include asymptotics, summations and recurrences.

### Asymptotics

Asymptotics involves O-notation (“big-Oh”) and its many relatives  $\Omega$ ,  $\theta$ ,  $o$  (“little-oh”),  $\omega$ . Asymptotic notation provides us with a way to simplify the function that arise in analyzing algorithm running times by ignoring constant factors and concentrating on the trends for large values of  $n$ . For example, it allows us to reason that for algorithms with the respective running times.

$$n^3 \log n + 4n^2 + 52n \log n \in \Theta(n^3 \log n)$$

$$15n^2 + 7n \log^3 n \in \Theta(n^2)$$

$$3n + 4 \log_5 n + 19n^2 \in \Theta(n^2)$$

Thus the first algorithm is significantly slower for large  $n$ , while the other two are comparable, up to a constant factor.

Since asymptotics were covered in earlier chapters, we will assume that this is familiar to you. Nonetheless, here are a few facts to remember about asymptotic notation.

#### Ignore constant factors

Multiplicative constant factors are ignored. For example,  $476n$  is  $\Theta(n)$ . Constant factors appearing exponents can not be ignored. For example,  $2^{3n}$  is not  $O(2^n)$ .

#### Focus on large $n$

Asymptotic analysis means that we consider trends for large  $n$  values of  $n$ . Thus, the fastest growing function of  $n$  is the only one that needs to be considered. For example,  $3n^2 \log n + 25n \log n + (\log n)^7$  is  $\Theta(n^2 \log n)$ .

## Polylog, Polynomial and Exponential

These are the most common functions that arise in analyzing algorithms.

**Polylogarithmic:** Powers of  $\log n$ , such as  $(\log n)^7$ . We will usually write this as  $\log^7 n$ .

**Polynomial:** Powers of  $n$ , such as  $n^4$  and  $\sqrt{n} = n^{\frac{1}{2}}$ .

**Exponential:** A constant (not 1) raised to the power  $n$ , such as  $3^n$ .

An important point is that polylogarithmic functions are strictly asymptotically smaller than polynomial function, which are strictly asymptotically smaller than exponential functions (assuming the base of the exponent is bigger than 1). For example, if we let  $<$  mean “asymptotically smaller” than

$\log^a n < b^b < c^n$  for any  $a, b, c$  provided  $b > 0, c > 1$ .

## Logarithm Simplification

It is a good idea to first simplify terms involving logarithms. For example, the following formulas are useful. Here  $a, b, c$  are constants.

$$\log_b n = \frac{\log_b n}{\log_a b} = \theta(\log_b n)$$

$$\log_a(n^c) = c \log_a n = \theta(\log_a n)$$

$$b^{\log_a n} = n^{\log_a b}$$

Avoid using  $\log n$  in exponents. The last rule above can be used to achieve this. For example, rather than saying  $3^{\log_2 n}$ , express this as  $n^{\log_2 3} \approx n^{1.585}$ .

Following the conventional sloppiness, we will often say  $O(n^2)$ , when in fact the stronger statement  $\theta(n^2)$  holds. (This is just because it is easier to say “oh” than “theta”)

## Summations

Summations naturally arise in the analysis of iterative algorithms. Also, more complex forms of analysis, such as recurrences, are often solved by reducing them to summations. Solving a summation means reducing it to a closed form formula, that is, one having no summations, recurrences, integrals, or other complex operators. In algorithm design it is often not necessary to solve a summation exactly, since an asymptotic approximation or closer upper bound is usually good enough. Here are some common summations and some tips to use in solving summations.

## Constant Series

For integers  $a$  and  $b$ ,

$$\sum_{i=a}^b 1 = \max(b - a + 1, 0)$$

Notice that, when  $b = a - 1$ , there are no terms in the summation (since the index is assumed to count upwards only), and the result is 0. Be careful to check that  $b \geq a - 1$  before applying formula blindly.

### Arithmetic Series

For  $n \geq 0$

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

This is  $\Theta(n^2)$ . (This starting bound could have just as easily been set to 1 as 0)

### Geometric Series

Let  $x \neq 1$  be any constant (independent of  $n$ ), then  $n \geq 0$ .

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

If  $0 < x < 1$  then this is  $\Theta(1)$ . If  $x > 1$ , then this is  $\Theta(x^n)$ , that is, the entire sum is proportional to the last element of the series.

### Quadratic Series

For  $n \geq 0$ ,

$$\sum_{i=0}^n i^2 = 1^2 + 2^2 + \dots + n^2 = \frac{2n^3 + 3n^2 + n}{6}$$

### Linear Geometric Series

This arise in some algorithms based on trees and recursion. Let  $x \neq 1$  be any constant, then for  $n \geq 0$ ,

$$\sum_{i=0}^{n-1} ix^i = x + 2x^2 + 3x^3 + \dots + nx^n = \frac{(n-1)x^{(n+1)} - nx^n + x}{(x-1)^2}$$

As  $n$  becomes large, this is asymptotically dominated by the term  $(n-1)x^{(n+1)}/(x-1)^2$ . The multiplicative term  $n-1$  is very nearly equal to  $n$  for large  $n$ , and since  $x$  is a constant, we may multiply this times the constant  $(x-1)^2/x$  without changing the asymptotics. What remains is  $\Theta(nx^n)$ .

### Harmonic Series

This arises often in probabilistic analysis of algorithms. It does not have an exact closed form solution, but it can be closely approximated. For  $n \geq 0$ .

$$H_n = \sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = (\ln n) + O(1)$$

There are also few tips to learn about solving summations.

### Summations with General Bounds

When a summation does not start at the 1 or 0, as most of the above formulas assume, you can just split it up into the difference of two summations. For example, for  $1 \leq a \leq b$

$$\sum_{i=a}^b f(i) = \sum_{i=0}^b f(i) - \sum_{i=0}^{a-1} f(i)$$

### Linearity of Summation

Constant factors and added terms can be split out to make summations simpler.

$$\sum (4 + 3i(i-2)) = \sum 4 + 3\sum i^2 - 6\sum i$$

Now the formulas can be to each summation individually.

### Approximate Using Integrals

Integration and summation are closely related (Integration is in some sense a continuous form of summation). Here is a handy formula, let  $f(x)$  be any monotonically increasing function (the function increases as  $x$  increases).

$$\int_0^n f(x) dx \leq \sum_{i=1}^n f(i) \leq \int_1^{n+1} f(x) dx$$

### Probability Distributions

The probability is a measure over a set of events that satisfies three axioms.

**Axiom – 1:** The measurement of each event is between 0 and 1. We write this as  $0 \leq P(E = e_i) \leq 1$ , where  $E$  is a random variable representing an event and  $e_i$  are the possible values of  $E$ . In general, random variables are denoted by upper case letters and their values by lower case letters.

**Axiom – 2:** The measure of the whole set is 1, that is  $\sum_{i=1}^n P(E = e_i) = 1$ .

**Axiom – 3:** The probability of a union of disjoint events is the sum of the probabilities of the individual events; that is,  $P(E = e_1 \cup E = e_2) = P(E = e_1) + P(E = e_2)$ , where  $e_1$  and  $e_2$  are disjoint.

### Probability Density Function

The probability density function denoted as  $P(X = c)$  is defined as the ratio of the probability that  $x$  falls into an interval around  $c$ , divided by the width of the interval, as the interval width goes to zero.

$$P(X = c) = \lim_{dx \rightarrow 0} P(c \leq x \leq c + dx)/dx$$

The density function must be non-negative for all  $x$  and must have  $\int_{-\infty}^{\infty} P(X) dx = 1$ .

### Cumulative Probability Density Function

We can also define cumulative probability density function  $F(x)$ , which is the probability of a random variable being less than  $x$ .

$$f(x) = \int_{-\infty}^x P(Z) dz$$

### Gaussian Distribution

One of the most important probability distributions is the Gaussian distribution, also known as the normal distribution. A Gaussian distribution with mean  $\mu$  and standard deviation  $\sigma$  (and therefore variance  $\sigma^2$ ) is defined as

$$P(X) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)/(2\sigma^2)}$$

where  $x$  is a continuous random variable ranging from  $-\infty$  to  $+\infty$ .

### Standard Normal Distribution

With mean  $\mu = 0$  and variance  $\sigma^2 = 1$ , we get the special case of the standard normal distribution.

### Multivariate Gaussian Distribution

For a distribution over a vector  $x$  in  $d$  dimensions, there is the multivariate Gaussian distribution.

$$P(X) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} e^{\left(\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)}$$

where  $\mu$  is the mean vector and  $\Sigma$  is the covariance matrix of the distribution.

### Cumulative Distribution

In one dimension, we can also define the cumulative distribution function  $F(x)$  as the probability that a random variable will be less than  $x$ . For the standard normal distribution, this is given by

$$F(x) = \int_{-\infty}^x P(x) dx = \frac{1}{2} \left( 1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \right)$$

where  $\operatorname{erf}(x)$  is the so called error function, which has no closed form representation.

### Central Limit Theorem

The central limit theorem states that the mean of  $n$  random variables tends to a normal distribution as  $n$  tends to infinity. This holds for almost any collection of random variables, unless the variance of any finite subset of variables dominates the others.



## TABLE FOR TIME COMPLEXITY OF ALGORITHMS

The run time of algorithms could be measured using various notations like theta, big- $\Omega$ , big O, little- $\omega$ , and little- $o$ . Therefore, the run times below are applicable for all the 5 notations, Consider binary logarithm i.e.,  $\log n = \log_2 n$ .

Serial Number	Algorithm	Running Time		
		Worst-Case	Average-Case	Best-Case
1	Insertion Sort	$n^2$	$n^2$	$n$
2	Bubble Sort	$n^2$	$n^2$	$n$
3	Merge Sort	$n \log n$	$n \log n$	$n \log n$
4	Quick Sort	$n^2$	$n \log n$	$n \log n$
5	Selection Sort	$n^2$	$n^2$	$n^2$
6	Heap Sort	$n \log n$	$n \log n$	$n \log n$
7	Counting Sort	$n + m$	$n + m$	$n$
8	Radix Sort	$d(n + k)$	$d(n + k)$	$(n + k)$
9	Bucket Sort	$nd$	$nd$	$nd$
10	Square-Matrix Multiply	$n^3$	$n^3$	$n^3$
11	Strassen's algorithm for matrix multiplication	—	$n^{\log 7}$	—
12	Binary Search	$\log n$	$\log n$	1
13	Finding Max or Min element in the array	$n$	$n$	$n$
14	Finding $i^{\text{th}}$ smallest element (Random Select)	$n^2$	$n$	$n$

Serial Number	Algorithm	Running Time		
		Worst-Case	Average-Case	Best-Case
15	Max-Heapify Procedure	$\log n$	$\log n$	$\log n$
16	Build Max-Heap Procedure	$n$	$n$	$n$
17	Heap-Extract-Max (for max-priority queue)	$\log n$	—	—
18	Heap-Maximum (for max-priority queue)	1	1	1
19	Heap-Increase Key (for max-priority queue)	$\log n$	$\log n$	$\log n$
20	Max-Heap-Insert (for max-priority queue)	$\log n$	$\log n$	$\log n$
21	Dynamic programming algorithm for LCS	$mn$	$mn$	$mn$
22	Dynamic programming algorithm for Chain Matrix Multiplication	$n^3$	$n^3$	$n^3$
23	Dynamic programming for 0-1 Knapsack problem	$nW$	$nW$	$nW$
24	Greedy algorithm for Activity Scheduling	$n$	$n$	$n$
25	Greedy algorithm for Fractional Knapsack problem	$n \log n$	$n \log n$	—
26	Huffman Algorithm	$n \log n$	$n \log n$	$n \log \log n$
27	Breadth-First Search (BFS)	E	V + E	V + E
28	Depth-First Search (DFS)	E	V + E	V + E
29	Topological-sort	V + E	V + E	V + E
30	Strongly-connected components	V + E	V + E	V + E
31	Kruskal's algorithm for finding minimum spanning tree	$E \log V$	$E \log V$	$E \log V$
32	Prim's algorithm for finding minimum spanning tree	$V^2$ Using Adjacency matrix	$E \log V$ Using Binary heap	$E + V \log V$ Using Fibonacci heap

<b>Serial Number</b>	<b>Algorithm</b>	<b>Running Time</b>		
		<b>Worst-Case</b>	<b>Average-Case</b>	<b>Best-Case</b>
33	Dijkstra's algorithm for finding shortest paths	$V^2$ Using Adjacency matrix	$E \log V$ Using Binary heap	$E + V \log V$ Using Fibonacci heap
34	Bellman-Ford algorithm for finding shortest paths	VE	VE	VE
35	Floyd-Warshall algorithm for finding all pairs shortest paths	$n^3$	$n^3$	$n^3$
36	Brute-Force string matching algorithm	$(n - m + 1) m$	$n^2$	$m$
37	Rabin-Karp string matching algorithm	$(n - m + 1) m$	$n + m$	$n + m$
38	Approximation algorithm for TSP problem	$E + V \log V$	$E + V \log V$	—
39	Approximation algorithm for Vertex Cover problem	$\log V$	$\log V$	—



# *SOLVED UNIVERSITY QUESTION PAPERS*

**Fourth Semester Examination–2013**  
**DESIGN AND ANALYSIS OF ALGORITHM**  
**BRANCH : CSE/IT**  
**QUESTION CODE : A352**  
**Full Marks–70**  
**Time : 3 Hours**

*Answer Question No. 1 which is compulsory and any five from the rest. The figures in the right-hand margin indicate marks.*

1. Answer the following questions: (2 × 10)
  - (a) What is master method?
  - (b) What do you mean by priority queue? Write the uses of priority queue.
  - (c) Sort the following expressions in ascending order of their value:  
 $n, n^2, \log_2 n, 2n, n \log_2 n, \sqrt{n}, n^3, \log_2 \log_2 n$
  - (d) What is the time complexity of Quick sort?
  - (e) What is the difference between Dynamic Programming and Divide and Conquer mechanism?
  - (f) What is the minimum and maximum number of elements in a heap of height  $h$ ?
  - (g) What is Huffman code? Find a set of Huffman codes for the message  
( $m_1, m_2, m_3, m_4, m_5, m_6, m_7$ ) with relative frequencies ( $q_1, q_2, q_3, q_4, q_5, q_6, q_7$ )  
= (4, 5, 7, 8, 10, 12, 20).
  - (h) What do you mean by approximation algorithms ? Give two examples.
  - (i) Compare Backtracking with Branch and bound algorithm with example.
  - (j) What do you mean by lower bounds for sorting?
2. (a) What do you mean by asymptotic notations? Describe, in brief, about the standard asymptotic notations. (5)

(b) Solve the recurrence by repeated substitution method. (5)

$$\begin{aligned} T(n) &= 1, \quad n <= 2 \\ &= 2T(n/4) + n^2, \quad n > 2 \end{aligned}$$

3. (a) Use a recursion tree and give an asymptotically tight solution to the recurrence.

$$T(n) = T(n/3) + T(2n/3) + cn, \quad \text{where } c \text{ is a constant.} \quad (5)$$

(b) Differentiate between best case, average case and worst case time complexity of an algorithm. Discuss with example the worst case analysis of merge sort. (5)

4. (a) What is matrix chain multiplication problem? Write the algorithm for matrix chain multiplication. Find the  $m$  and  $s$  table computed by the algorithm for the following matrix dimensions: (5)

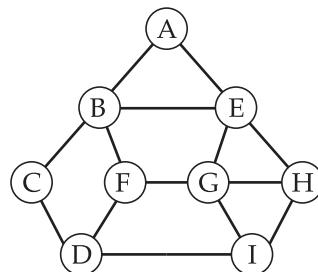
Matrix	Dimension
A1	25*35
A2	35*15
A3	15*5
A4	5*40

(b) Define disjoint sets. Discuss the linked list representation of disjoint sets. (5)

5. (a) What is Activity Selection Problem? Find the solution of Activity Selection Problem for the following set of activities. What is the time complexity of it? (5)

i	1	2	3	4	5	6	7	8	9	10	11
Si	0	5	12	1	5	2	3	3	8	5	6
Fi	6	6	14	4	9	13	8	5	12	7	10

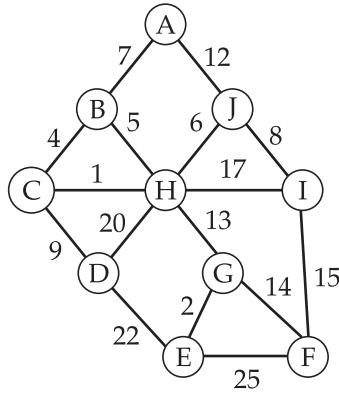
(b) Write the algorithm of Depth First Search of a graph. Find the DFS of the following graph. Take "A" as the start node. (5)



6. (a) What is back tracking? Write the algorithm for N Queen's problem using back tracking. (5)

(b) What is Floyd's algorithm? Explain with a suitable example. What is the time complexity of this algorithm? (5)

7. (a) Find the minimum cost spanning tree using Prim's algorithm for the following graph. (5)



(b) Write the Rabin Karp's String matching algorithm. Find the time complexity of it. (5)

8. Explain the following terms in brief: 4 × 2.5
- (a) NP Completeness
  - (b) Dijkstra's algorithm
  - (c) Assembly Line Scheduling
  - (d) Longest Common Sub Sequence.

### **SOLUTION TO FOURTH SEMESTER EXAMINATION-2013**

---

1. (a) Master method is defined by the three cases which are used to find the time complexity of the recurrence of the form of  $T(n) = aT(n/b) + f(n)$ , where  $a \geq 1$  and  $b > 1$ , as follows:

**Case-I:** If  $f(n) = O(n^{\log_b a - \varepsilon})$  for some constant  $\varepsilon > 0$  then  $T(n) = \Theta(n^{\log_b a})$ .

**Case-II:** If  $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$  for some constant  $k > 0$  then  $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+\frac{1}{n}} n)$ .

**Case-III:** If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$  then  $T(n) = \Theta(f(n))$ .

Moreover, in each of the above three cases we compare the function  $f(n)$  with the function  $n^{\log_b a}$ . The larger of the two functions gives the solution to the recurrence.

- (b) Priority queue is an abstract data type which consists of collection of items that get processed according to the priority.

Depending on the situation the priority can be maximum or minimum.

The different uses of priority queue involves:

- Holding the packets of data waiting to be transmitted over a network.
- Holding the jobs (i.e. programs) that are waiting for execution in an operating system.

(c) The ascending order of the expressions are given as:  $\log_2 \log_2 n, \log_2 n, n \log_2 n, \sqrt{n}, n, 2n, n^2, n^3$ .

(d) The time complexity of quick sort is given by  $\theta(n \log n)$ .

(e)

Divide and conquer mechanism	Dynamic programming
<ul style="list-style-type: none"> <li>(i) In divide and conquer approach we break the problem into several sub-problems that are similar to the original problem and solve them recursively.</li> <li>(ii) It follows top-down approach.</li> <li>(iii) Merge sort uses divide and conquer mechanism to sort.</li> </ul>	<ul style="list-style-type: none"> <li>(i) In dynamic programming we take the smallest sub-problem, memorize it (store somewhere) which is used to calculate the bigger problem iteratively.</li> <li>(ii) It follows bottom-up approach.</li> <li>(iii) Chain matrix multiplication uses this technique.</li> </ul>

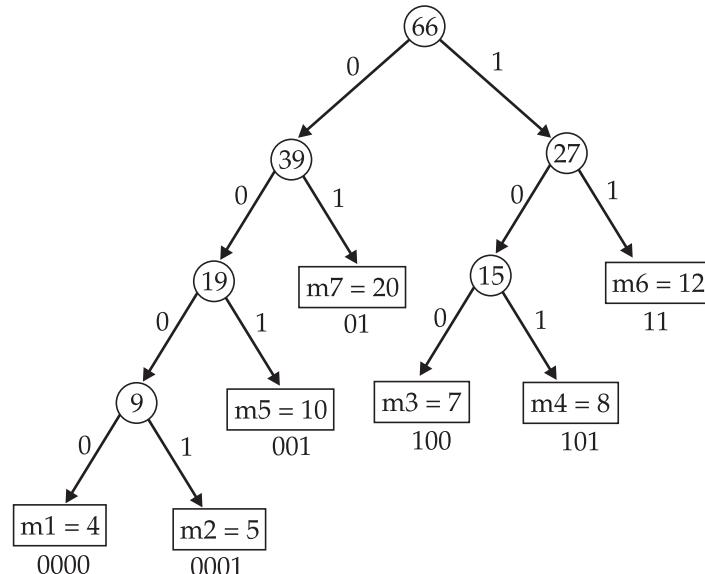
(f) The minimum number of elements in a heap of height  $h$  is  $2^h$ . The maximum number of elements in a heap of height  $h$  is  $2^{h+1} - 1$ .

(g) Huffman code is an efficient data encoding technique which drastically reduces the expected encoding length.

The given set of messages =  $(m_1, m_2, m_3, m_4, m_5, m_6, m_7)$

and their relative frequencies =  $(q_1, q_2, q_3, q_4, q_5, q_6, q_7) = (4, 5, 7, 8, 10, 12, 20)$

The final tree for Huffman code becomes



So the Huffman codes for  $m_1, m_2, m_3, m_4, m_5, m_6, m_7$  are 0000, 0001, 100, 101, 001, 11, 01 respectively.

(h) Approximation algorithm is an algorithm that runs in polynomial time and produces a solution that is within a guaranteed factor of the optimum solution. Examples are– travelling salesman problem and greedy vertex cover.

(i) Backtracking is a recursive algorithm strategy that tries to build a solution to a computational problem incrementally. In other words, it is a methodical way of trying out various sequences of decisions until we find one that works.

In backtracking, we use depth-first search with pruning to traverse the state space. However, we achieve better performance for many problems using a breadth-first search with pruning. This approach is called branch and bound which is nothing but a general search method. In this method we use both lower-bounding and upper-bounding procedures to the root problem. If the bounds match, then an optimal solution has been found and the procedure terminates.

(j) Lower bounds for sorting is a property of sorting problem called comparison based sorting algorithm that at least take  $T(n)$  time in the worst case; where  $T(n)$  is the minimum over all possible algorithms of the maximum complexity.

2. (a) Asymptotic running time of an algorithm is described by asymptotic notation. Asymptotic notations are the functions whose domains are the set of natural numbers i.e.,  $N = \{0, 1, 2, \dots\}$ .

The different standard asymptotic notations are:

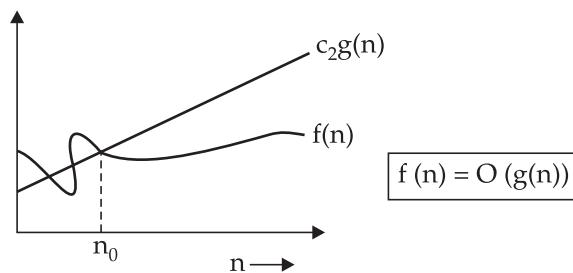
- (i) Big-oh notation ( $O$ -notation)
- (ii) Big-omega notation ( $\Omega$ -notation)
- (iii) Theta notation ( $\Theta$ -notation)
- (iv) Little-oh notation ( $o$ -notation)
- (v) Little-omega notation ( $\omega$ -notation)

These notations can be described as follows:

- (i)  $O$ -notation. It gives an asymptotic upper bound for a function to within a constant factor.

$O(g(n)) = \{f(n) \mid \text{there exist positive constants } C_2 \text{ and } n_0 \text{ such that}$

$$0 \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

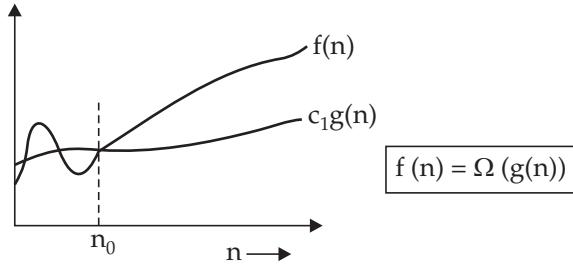


(ii)  $\Omega$ -notation.

It provides us asymptotic lower bound for a function within a constant factor.

$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1 \text{ and } n_0 \text{ such that}$

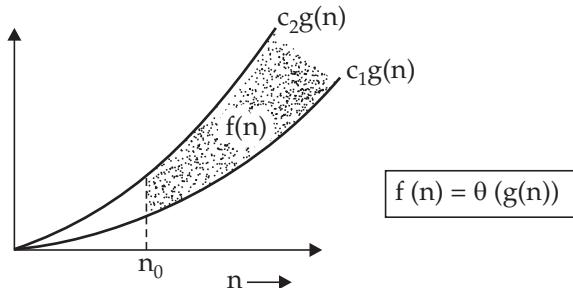
$$0 \leq c_1 g(n) \leq f(n), \forall n \geq n_0\}$$

(iii)  $\Theta$ -notation

$\Theta$ -notation bounds a function to within constant factors. It can be defined as

$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that}$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

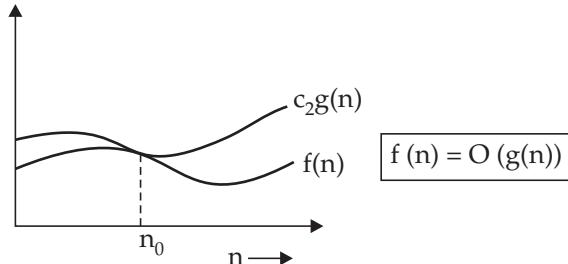


So here  $c_2 g(n)$  and  $c_1 g(n)$  are asymptotic upper and asymptotic lower bound for  $f(n)$  respectively.

(iv)  $o$ -notation

Little-oh notation is used to denote an upper bound that is not asymptotically tight. So it is defined as

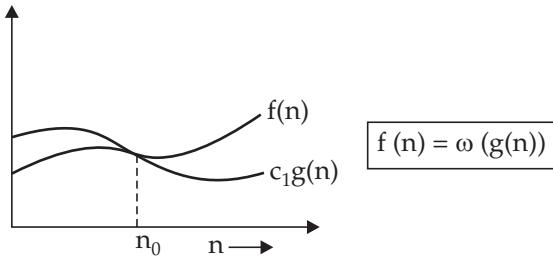
$o(g(n)) = \{f(n) \mid \text{for any positive constant } c_2 > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < c_2 g(n), \forall n \geq n_0\}$



(v)  $\omega$ -notation

Little- $\omega$  notation is used to denote a lower bound that is not asymptotically tight. So we define  $\omega(g(n))$  as follows

$\omega(g(n)) = \{f(n) \mid \text{for any positive constant } c_1 > 0, \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq c_1 g(n) < f(n), \forall n \geq n_0\}$



$$(b) \text{ Given } T(n) = \begin{cases} 1, & n \leq 2 \\ 2T(n/4) + n^2, & n > 2 \end{cases}$$

Let the solution be  $O(n^2)$ . Remember that this is only a guess. It may work or may not.

$$T(n) \leq cn^2, \forall n \geq n_0$$

$$T(k) \leq ck^2, \forall k < n$$

For

$$k = n/4$$

$$T(n/4) \leq c(n^2/16)$$

$$T(n) = 2T(n/4) + n^2$$

$$= 2c\left(\frac{n^2}{16}\right) + n^2$$

$$= n^2 \left(\frac{2c}{16} + 1\right) = O(n^2),$$

Since the term  $\left(\frac{2c}{16} + 1\right)$  is a constant. Therefore our guess is correct.

3. (a) The given recurrence equation is

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \quad \dots(i)$$

$$\text{When } n = \frac{n}{3}, \text{ then } T\left(\frac{n}{3}\right) = T\left(\frac{n}{9}\right) + T\left(\frac{2n}{9}\right) + cn \quad \dots(ii)$$

When  $n = \frac{2n}{3}$ , then equation (i) becomes

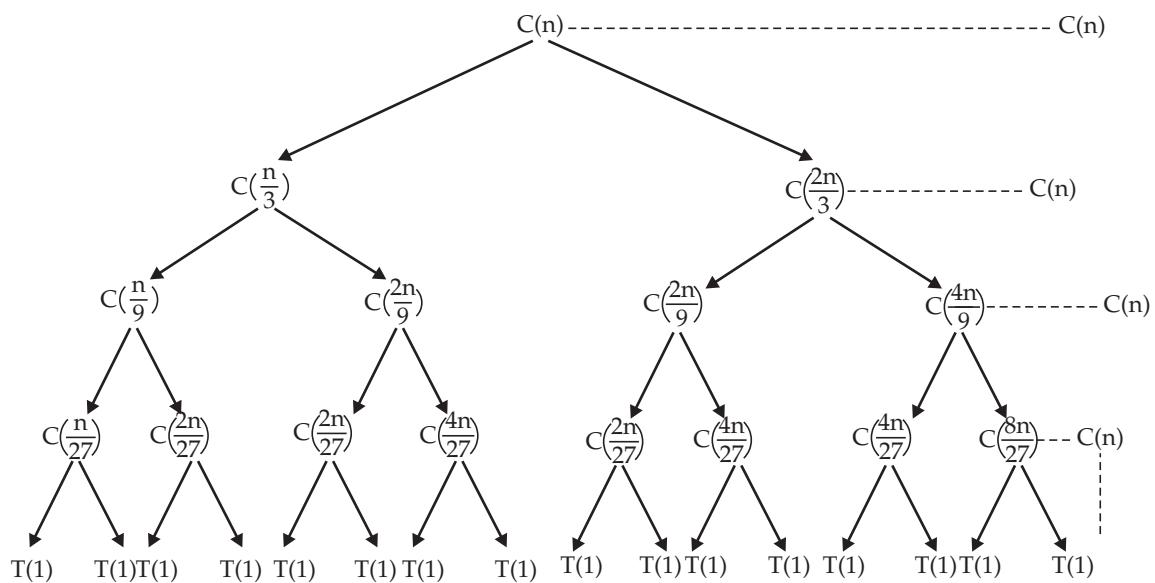
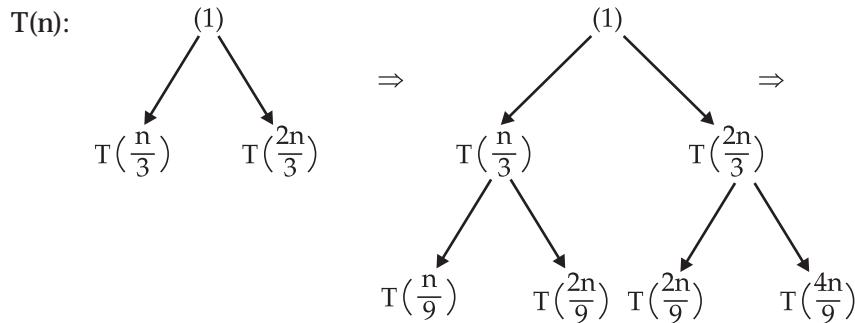
$$T\left(\frac{2n}{3}\right) = T\left(\frac{2n}{9}\right) + T\left(\frac{4n}{9}\right) + cn \quad \dots(iii)$$

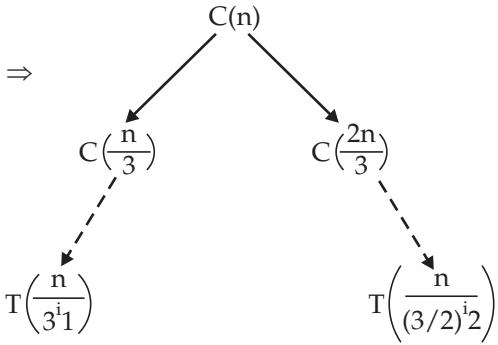
When  $n = \frac{n}{9}$  in equation (i) then

$$T\left(\frac{n}{9}\right) = T\left(\frac{n}{27}\right) + T\left(\frac{2n}{27}\right) + cn \quad \dots(iv)$$

and so on.

Now lets draw the recursion tree





Here

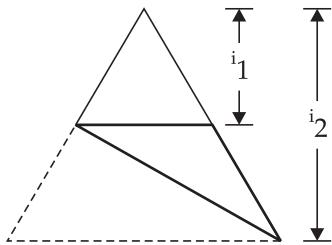
$$\frac{n}{3^{i_1}} = 1 \Rightarrow n = 3^{i_1} \Rightarrow i_1 = \log_3 n$$

Again

$$\frac{n}{(3/2)^{i_2}} = 1 \Rightarrow n = \left(\frac{3}{2}\right)^{i_2} \Rightarrow i_2 = \log_{3/2} n$$

It is clear that  $\log_{3/2} n > \log_3 n$ . Therefore  $i_2 > i_1$ .

Now the tree looks like as follows:



We have

$$T(n) \leq (i_2 + 1)cn \quad \dots(v)$$

Again

$$(1 + i_1) cn \leq T(n) \quad \dots(vi)$$

Now

$$(1 + i_1) cn \leq T(n) \leq (1 + i_2) cn$$

(combining equation (v) and (vi))

$$\Rightarrow cn + i_1 cn \leq T(n) \leq cn + i_2 cn$$

$$\Rightarrow i_1 cn \leq T(n) \leq i_2 cn$$

Substituting the values of  $i_1$  and  $i_2$  we get

$$c(\log_3 n)n \leq T(n) \leq c(\log_{3/2} n)n$$

$$\Rightarrow c \frac{\log n}{\log 3} n \leq T(n) \leq c \frac{\log n}{\log \frac{3}{2}} n$$

$$\Rightarrow c_1 n \log n \leq T(n) \leq c_2 n \log n \quad \left( \text{where } c_1 = \frac{c}{\log 3} \text{ and } c_2 = \frac{c}{\log \frac{3}{2}} \right)$$

Hence,  $T(n) = \Theta(n \log n)$ .

(b) Best Case Time Complexity: The best case time complexity of an algorithm is its efficiency for the best case input of size  $n$  for which the algorithm runs the fastest among all possible inputs of that size. For example, if you want to sort a list of numbers in ascending order when the numbers are given in ascending order. In this running time will be the smallest.

Worst Case Time Complexity: The worst case time complexity of an algorithm is its efficiency for the worst case input of size  $n$ , which is an input of size  $n$  for which the algorithm runs the longest among all possible inputs of that size. For example, if you want to sort a list of numbers in ascending order when the numbers are given in descending order. In this running time will be the longest.

Average Case Time Complexity: The average case time complexity of an algorithm is its efficiency for the input of size  $n$ , for which the algorithm runs between the best case and the worst case among all possible inputs of that size.

The worst case analysis of merge sort can be given as follows:

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

We can rewrite the above recurrence as follows:

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

where  $c$  is the constant. Applying Master theorem to solve the recurrence  $T(n) = 2T(n/2) + cn$  which is of the form of  $T(n) = 2T(n/b) + f(n)$ , where  $a = 2$ ,  $b = 2$ ,

$f(n) = cn$  and  $n^{\log_b a} = n^{\log_2 2} = n$ .

By applying case-II of Master theorem we get

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

Hence the worst case running time of merge-sort is  $\Theta(n \log n)$ .

4. (a) Given a sequence of matrices  $A_1, A_2, \dots, A_n$  and the dimensions  $p_0, p_1, \dots, p_n$  where  $A_i$  is of dimensions  $p_{i-1} \times p_i$  determines the order of multiplication that minimizes the number of operations. So this algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

Algorithm for Chain-Matrix Multiplication:

CHAIN-MATRIX ORDER (array  $p[1 \dots n]$ )

1. array  $s[1 \dots n-1, 2 \dots n]$
2. for  $i = 1$  to  $n$  do  $m_{ij} = 0$  // initialization
3. for  $L = 2$  to  $n$  do //  $L$  = length of sub-chain
4. for  $i = 1$  to  $n - i + 1$  do

5.  $j = i + L - 1$
6.  $m_{ij} = \infty$  //  $m$  denotes final cost
7. for  $k = i$  to  $j - 1$  do // check all splits
8.  $q = m_{ik} + m_{(k+1)j} + p[i-1] * p[k] * p[j]$
9. if ( $q < m_{ij}$ )
10.  $m_{ij} = q$
11.  $s[i+j] = k$  // splitting markers
12. return  $m_{1n}$ , and  $s$ .

Given dimensions  $p[0] = 25, p[1] = 35, p[2] = 15, p[3] = 5, p[4] = 40$

The values of  $m_{11}, m_{22}, m_{33}$  and  $m_{44}$  are all equal to zero since  $i = j$ .

	1	2	3	4
1	$m_{11}$	$m_{12}$	$m_{13}$	$m_{14}$
2		$m_{22}$	$m_{23}$	$m_{24}$
3			$m_{33}$	$m_{34}$
4				$m_{44}$

$$\begin{aligned}m_{12} &= \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\&= \min_{1 \leq k \leq 1} [m_{11} + m_{22} + p[0] \cdot p[1] \cdot p[2]] \\&= 0 + 0 + (25 \times 35 \times 15) = 13125\end{aligned}$$

$$\begin{aligned}m_{23} &= \min_{2 \leq k \leq 2} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\&= \min_{2 \leq k \leq 2} [m_{22} + m_{33} + p[1] \cdot p[2] \cdot p[3]] \\&= 0 + 0 + (35 \times 15 \times 5) = 2625\end{aligned}$$

$$\begin{aligned}m_{34} &= m_{33} + m_{44} + p[2] \cdot p[3] \cdot p[4] \\&= 0 + 0 + (15 \times 5 \times 40) = 3000\end{aligned}$$

$$m_{13} = \min_{1 \leq k \leq 2} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

$$\begin{aligned}\text{when } k = 1, \quad m_{13} &= m_{11} + m_{23} + p[0] \cdot p[1] \cdot p[3] \\&= 0 + 2625 + (25 \times 35 \times 5) = 7000\end{aligned}$$

$$\begin{aligned}\text{When } k = 2, \quad m_{13} &= m_{12} + m_{33} + p[0] \cdot p[2] \cdot p[3] \\&= 13125 + 0 + (25 \times 15 \times 5) = 15000\end{aligned}$$

For  $k = 1, m_{13}$  is minimum i.e., 7000.

$$m_{24} = \min_{2 \leq k \leq 3} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

$$\begin{aligned}\text{When } k = 2, \quad m_{24} &= m_{22} + m_{34} + p[1] \cdot p[2] \cdot p[4] \\&= 0 + 3000 + (35 \times 15 \times 40) = 24000\end{aligned}$$

When  $k = 3$ ,  $m_{24} = m_{23} + m_{44} + p[1] \cdot p[3] \cdot p[4]$   
 $= 2625 + 0 + (35 \times 5 \times 40) = 9625$

For  $k = 3$ ,  $m_{24}$  is minimum i.e., 9625.

$$m_{14} = \min_{1 \leq k \leq 3} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

When  $k = 1$ ,  $m_{14} = m_{11} + m_{24} + p[0] \cdot p[1] \cdot p[4]$   
 $= 0 + 9625 + (25 \times 35 \times 40) = 44625$

When  $k = 2$ ,  $m_{14} = m_{12} + m_{34} + p[0] \cdot p[2] \cdot p[4]$   
 $= 13125 + 3000 + (25 \times 15 \times 40) = 31125$

When  $k = 3$ ,  $m_{14} = m_{13} + m_{44} + p[0] \cdot p[3] \cdot p[4]$   
 $= 7000 + 0 + (25 \times 5 \times 40) = 12000$

For  $k = 3$ ,  $m_{14}$  is minimum i.e., 12000.

The final table becomes

	1	2	3	4
1	0	13125	7000	12000
2		0	2625	9625
3			0	3000
4				0

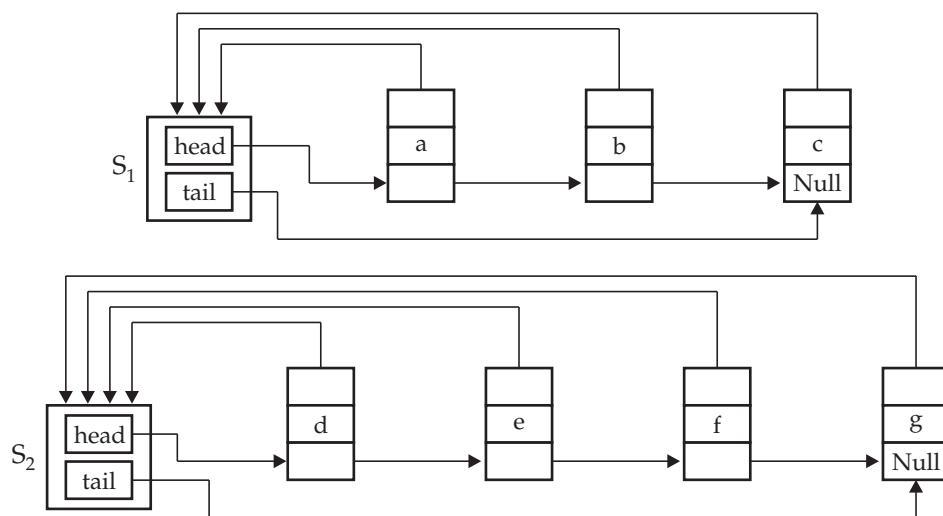
(b) A disjoint set is a data structure that maintains a collection of disjoint dynamic sets.

Let  $S_1$  and  $S_2$  be two disjoint sets given as

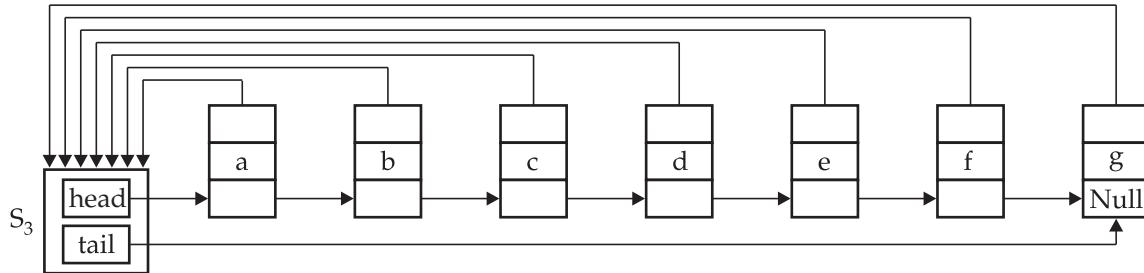
$$S_1 = \{a, b, c\}$$

$$S_2 = \{d, e, f, g\}$$

The linked-list representation of the two sets are given as follows:



The attributes head and tail point to the first and last object in each set. Each object in the list contains a set member, a pointer to the next object in the list and a pointer back to the set object. We can perform a simple union operation on two lists by appending one list onto the end of other list. The union of two lists  $S_1$  and  $S_2$  can be represented as follows:



5. (a) Activity selection problem is a very simple scheduling problem. In this problem we are given a set  $S = \{1, 2, \dots, n\}$  of  $n$  activities that are to be scheduled to use some resource, where each activity must be started at a given start time  $s_i$  and ends at a given finish time  $f_i$ . Two activities  $i$  and  $j$  are non-interfacing if their start-finish intervals don't overlap, i.e.,  $[s_i, f_i] \cap [s_j, f_j] = \emptyset$ . The basic objective of activity scheduling problem is to select a maximum size set of mutually non-interfacing activities for the use of resource.

The given 11 activities are

$$\begin{aligned} A_1 &= (0, 6), A_2 = (5, 6), A_3 = (12, 14), A_4 = (1, 4), A_5 = (5, 9), \\ A_6 &= (2, 13), A_7 = (3, 8), A_8 = (3, 5), A_9 = (8, 12), A_{10} = (5, 7), A_{11} = (6, 10). \end{aligned}$$

Arranging the 11 activities according to the increasing order of finishing time stamp we get

$$\begin{array}{cccccccccccc} (1, 4), & (3, 5), & (0, 6), & (5, 6), & (5, 7), & (3, 8), & (5, 9), & (6, 10), & (8, 12), & (2, 13) & (12, 14) \\ \downarrow & \downarrow \\ A_4 & A_8 & A_1 & A_2 & A_{10} & A_7 & A_5 & A_{11} & A_9 & A_6 & A_3 \end{array}$$

Applying (Iterative) Greedy–Activity–Selector algorithm

$$A = \{A_4\} \quad (\text{Initialisation})$$

$$A = \{A_4, A_2, A_{11}, A_3\} = \{(1, 4), (5, 6), (6, 10), (12, 14)\}$$

The time complexity of sorting the activities by finishing time which takes  $\theta(n \log n)$  time and the Greedy–Activity–Selector takes  $\theta(n)$  running time. Therefore, the total time =  $\theta(n \log n) + \theta(n)$ .

(b) Algorithm for DFS traversal:

DFS (G)

1. For all  $u \in V[G]$
2. do Colour  $[u] = \text{white}$ ,  $\pi[u] = \text{Null}$  // initialization

```

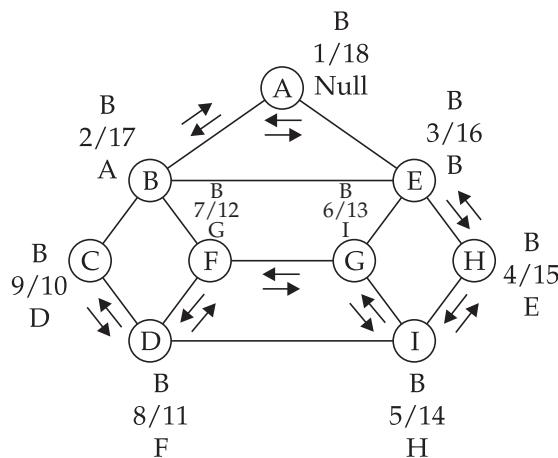
3. Time = 0
4. For all  $u \in V[G]$ 
5. if Colour [u] = white
6. then DFSVISIT (u)
DFSVISIT (u) // start a search at u
1. Colour [u] = Gray
2. Time = Time + 1
3.  $d[u] = \text{Time}$  //  $d[u]$  refers to starting time stamp
4. For all  $V \in \text{Adjacent}[u]$ 
5. do if Colour [V] = white
6. then  $\pi[V] = u$ 
7. DFSVISIT (V)
8. Colour [u] = Black
9. Time = Time + 1
10.  $f(u) = \text{Time}$  //  $f(u)$  refers to finishing time stamp

```

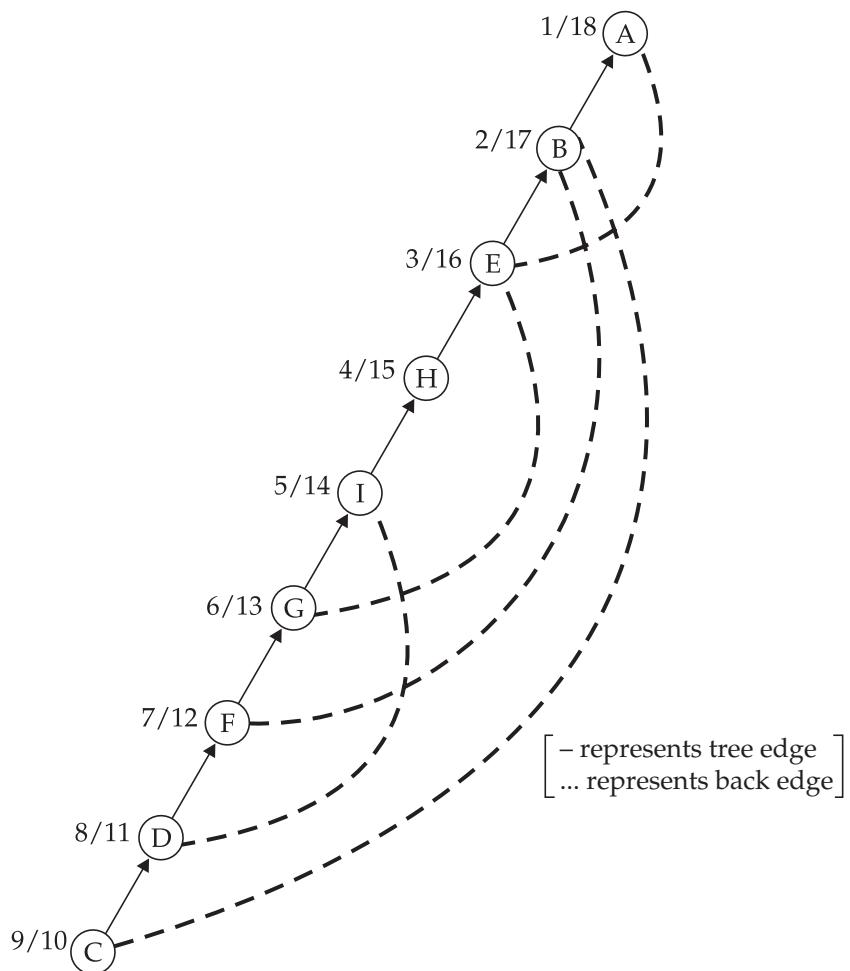
For the given graph the list of adjacent vertices are

A : B, E	E : A, B, G, H
B : A, E, F, C	F : B, G, D,
	G : E, F, H,
C : B, D	H : E, G, I
D : C, F, I	I : G, H, D

First initialize all vertex to be white in colour, starting time stemp is  $\infty$  and predecessor fields are Null. Start DFSVISIT from vertex A, then vertex B, vertex E and so on. Finally we get the following.



The above DFS traversal shows that the edges AB, BE, EH, HI, IG, GF, FD and DC are the tree edges and the edges AE, EG, ID, BF and BC are the back edges. So the DFS tree becomes



6. (a) Backtracking is a recursive algorithm strategy. The principal idea behind this strategy is to construct a solution part by part and when it gets into a dead end, then it has either built a solution or it needs to go back and try picking different values for some of the parts. We always check whether the solution we have built is a valid solution only at the deepest level of recursion.

Backtracking is used to solve queen problem, puzzle problem, sudoku etc.

Algorithm for N Queens problem:

Recursive N Queens (Q [1 ..... n], r)

1. if  $r = n + 1$  //  $r$  is the first empty row
2. print Q
3. else
4. for  $j = 1$  to  $n$
5. legal = True
6. for  $i = 1$  to  $r - 1$

```

7. if (Q [i] = j) or (Q [i] = j + r - i) or (Q [i] = j - r + i)
8. legal = False
9. if legal
10. Q [r] = j
11. Recursive N Queens (Q [1 ..... n], r + 1)

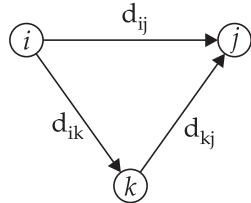
```

(b) Floyd's algorithm is also known as Floyd-Warshall algorithm. This technique is used to find the shortest path between two vertices in a given graph. The problem can be stated in the following way.

Let  $d_{ij}^{(k)}$  be the shortest path from vertex  $i$  to  $j$  which consists of the single edge  $(i, j)$  or it visits some intermediate vertices along the way. Note that these intermediate vertices can only be chosen from the set  $\{1, 2, \dots, k\}$ . This arises two cases.

(i) Without passing through the intermediate vertex  $k$ .

(ii) Passing through the intermediate vertex  $k$ .

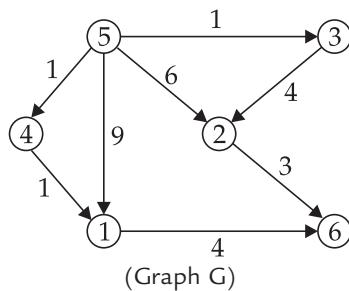


So the dynamic programming formulation for Floyd's algorithm can be given as

$$\begin{cases} d_{ij}^{(0)} = w_{ij}, & \text{when } k = 0 \\ d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & \text{when } k \geq 1 \end{cases}$$

For example, in the given graph G

$d_{5,6}^{(0)} = \infty$	(since no path exist)
$d_{5,6}^{(1)} = 13$	(path is 5-1-6)
$d_{5,6}^{(2)} = 9$	(path is 5-2-6)
$d_{5,6}^{(3)} = 8$	(path is 5-3-2-6)
$d_{5,6}^{(4)} = 6$	(path is 5-4-1-6)



Our basic objective is to limit the intermediate vertices.  $d_{5,6}^{(2)}$  can go through any combination of the intermediate vertices {1, 3, 4} for which 5–2–6 has the lowest cost 9.

Floyd-Warshall Algorithm:

Floyd-Warshall (int n, int w[1 ..... n, 1 ..... n]) // n is the number of vertices in given graph

1. array d[1 ..... n, 1 ..... n]
2. For i = 1 to n do // initialization
3. For j = 1 to n
4. d[i, j] = w[i, j]
5. mid [i, j] = null
6. for k = 1 to n // use intermediates {1, ..... , k}
7. for i = 1 to n do // form i
8. for j = 1 to n do // to j
9. if ((d[i, k] + d[k, j]) < d[i, j])
10. d[i, j] = d[i, k] + d[k, j] // new shortest path length
11. mid [i, j] = k // new path is through k
12. return d<sup>(n)</sup> // matrix of distances

Running time of the above algorithm is  $\theta(n^3)$ , where  $n$  is the total number of vertices in given graph.

7. (a) To find MST of the given graph we take A as the start vertex.

Step 1:

vertex	A	B	C	D	E	F	G	H	I	J
key	0	$\infty$								
$\pi$	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Step 2:

vertex	A	B	C	D	E	F	G	H	I	J
key	0	7	$\infty$	12						
$\pi$	NIL	A	NIL	A						

Step 3:

vertex	A	B	C	D	E	F	G	H	I	J
key	0	7	4	$\infty$	$\infty$	$\infty$	$\infty$	5	$\infty$	12
$\pi$	NIL	A	B	NIL	NIL	NIL	NIL	B	NIL	A

Step 4:

vertex	A	B	C	D	E	F	G	H	I	J
key	0	7	4	9	$\infty$	$\infty$	$\infty$	X	$\infty$	12
$\pi$	NIL	A	B	C	NIL	NIL	NIL	C	NIL	A

Step 5:

vertex	A	B	C	D	E	F	G	H	I	J
key	0	7	4	9	$\infty$	$\infty$	13	X	17	12
$\pi$	NIL	A	B	C	NIL	NIL	H	C	H	A

Step 6:

vertex	A	B	C	D	E	F	G	H	I	J
key	0	7	4	9	22	$\infty$	13	X	17	12
$\pi$	NIL	A	B	C	D	NIL	H	C	H	A

Step 7:

A	B	C	D	E	F	G	H	I	J
0	7	4	9	22	$\infty$	13	X	8	12
NIL	A	B	C	D	NIL	H	C	J	A

Step 8:

A	B	C	D	E	F	G	H	I	J
0	7	4	9	22	15	13	X	8	12
NIL	A	B	C	D	I	H	C	J	A

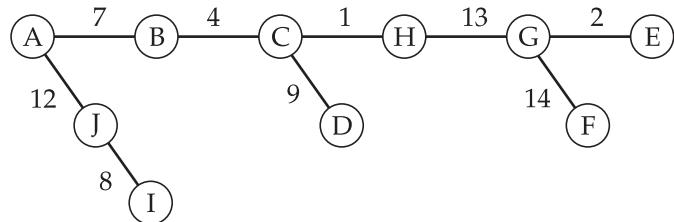
Step 9:

A	B	C	D	E	F	G	H	I	J
0	7	4	9	2	14	13	X	8	12
NIL	A	B	C	G	G	H	C	J	A

Step 10:

A	B	C	D	E	F	G	H	I	J
0	7	4	9	2	14	13	1	8	12
NIL	A	B	C	G	G	H	C	J	A

The MST edges are AB, BC, CH, HG, GE, AJ, JI, CD and GF. So the minimum cost spanning tree looks like as follows.



(MST for the given graph)

Its cost is  $= 7 + 4 + 1 + 13 + 2 + 12 + 8 + 9 + 14 = 70$ .

(b) Rabin Karp's string matching algorithm:

RABIN-KARP MATCHER ( $T, p, d, q$ )

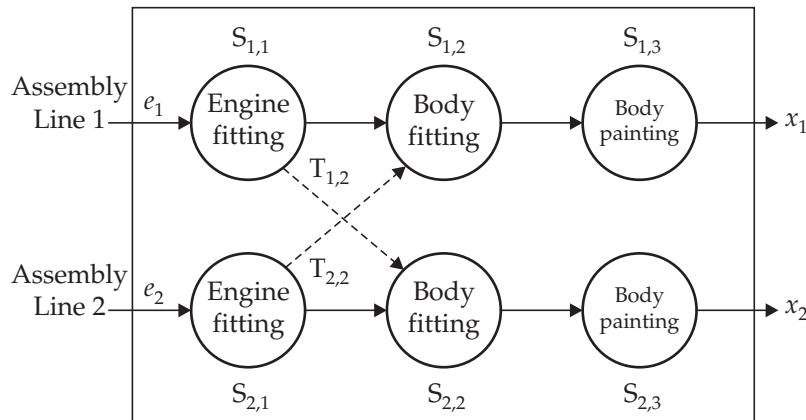
1.  $n = \text{length}[T]$  //  $n$  is the length of the give text  $T$
2.  $m = \text{length}[P]$  //  $m$  is the length of the given pattern  $P$
3.  $h = d^{m-1} \bmod q$  //  $d$  is the radix (which is  $|\Sigma|$ ) and  $q$  is the prime number
4.  $p = 0$
5.  $t_0 = 0$
6. for  $i = 1$  to  $m$
7.  $p = (dp + P[i]) \bmod q$
8.  $t_0 = (dt_0 + T[i]) \bmod q$
9. for  $s = 0$  to  $n - m$
10. if  $P == t_s$
11. if  $P[1.....m] = T[s+1 ..... s+m]$
12. Output is "pattern occurs with shift"  $s$
13. if  $s < n - m$
14. then  $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

Time complexity for matching =  $\Theta(m(n - m + 1))$ .

8. (a) NP Completeness: The concept of NP-Completeness was first introduced by Stephen Cook in 1971. A decision problem  $\pi$  is said to be NP-Complete if
  - (i)  $\pi$  is in NP and
  - (ii)  $\pi$  is in NP Hard (i.e., every problem in NP is reducible to it in polynomial time)

Moreover a decision problem is said to be in NP by demonstrating a certificate which can be verified in polynomial time. Common NP-Complete problems are clique, independent set, vertex cover etc.

- (b) Dijkstra's algorithm: Dijkstra's algorithm is one of those techniques of solving the single source shortest-paths problem on a non-negative weighted directed graph  $G = (V, E)$ . It is based on Greedy approach. Algorithm starts at the source vertex  $S$ ; it grows a tree that ultimately spans all vertices that are reachable from  $S$ . Vertices are added to the tree in terms of distance i.e., first  $S$ , then the vertex close to  $S$ , then next closest and so on. The running time of Dijkstra's algorithm is  $O(E \log V)$  where  $E$  denotes the number of edges in the given graph  $G$  and  $V$  denotes the number of vertices.
- (c) Assembly line Scheduling: This is a type of job scheduling problem. The problem can be described as follows. A car factory has two assembly lines, each with  $n$  stations. A station is denoted by  $S_{i,j}$  where  $i$  is either 1 or 2 and indicates the assembly line the station is on, and  $j$  indicates the number of stations as shown in the following figure.



The time taken per station is denoted by  $a_{i,j}$ . Each station is dedicated to some sort of work like engine fitting, body parts fitting, painting and so on. So a car chassis must pass through each of the  $n$  stations in order before exiting from factory. The parallel stations of the two assembly lines perform the same task. After it passes through the station  $S_{i,j}$ , it will continue to the station  $S_{i,j+1}$  unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line  $i$  at station  $j-1$  to station  $j$  on the other line takes time  $t_{i,j}$ . Each assembly line takes an entry time  $e_i$  and exit time  $x_i$  which may differ for two lines. Problem is how the scheduling process will take minimum time to build a car chassis. This can be solved by applying dynamic programming.

- (d) Longest common subsequence: LCS is one of the applications of dynamic programming approach. Given two sequences of characters  $X = \langle x_1, x_2, \dots, x_n \rangle$  and  $Y = \langle y_1, y_2, \dots, y_m \rangle$ , we say the longest common sub-sequence on  $X$  and  $Y$  is the longest sequence  $Z$  that is a sub-sequence of both  $X$  and  $Y$ . For example, if  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$  then the LCS is  $Z = \langle B, C, A, B \rangle$ .

**Fourth Semester Examination–2012**  
**DESIGN AND ANALYSIS OF ALGORITHM**  
**BRANCH : CSE/IT**  
**QUESTION CODE : 4204**  
**Full Marks–70**  
**Time : 3 Hours**

*Answer Question No. 1 which is compulsory and any five from the rest. The figures in the right-hand margin indicate marks.*

1. Answer the following questions: (2 × 10)
  - (a) Compare the two functions  $n^2$  and  $2^n/4$  for various values of  $n$ . Determine when the second becomes larger than the first.
  - (b) Prove that  $o(g(n)) \cap w(g(n))$  is the empty set.
  - (c) Give asymptotic upper and lower bounds for  $T(n)$  for  $n \leq 2$  where  $T(n) = 2T(n/4) + \sqrt{n}$ .
  - (d) What are the minimum and maximum numbers of elements in a heap of height  $h$ ?
  - (e) What is the running time of QUICKSORT when all elements of array A contains distinct elements and the elements are stored in decreasing order?
  - (f) Find an optimal parenthesization of matrix-chain product whose sequence of dimension is  $< 5, 10, 3, 12, 5, 50, 6 >$ .
  - (g) Prove that the fractional knapsack problem has the greedy-choice property.
  - (h) How can the output of the Floyd-Warshall algorithm be used detect the presence of a negative-weight cycle.
  - (i) Working modulo = 11, how many spurious hits does the Rabin-Karp matcher encounter in the text  $T = 3141592653589793$  when looking for the pattern  $p = 26$ ?
  - (j) Show that Hamiltonian-path problem is NP-complete.
2. (a) Use a recursion tree to give an asymptotically tight solution to the recurrence  $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$ , where  $\alpha$  is a constant in the range  $0 < \alpha < 1$  and  $c > 0$  is also constant. (5)
  - (b) Show that the worst-case time of procedure Mergesort is  $O(n \log n)$ . What is its best-case time? Can it be  $\Theta(n \log n)$ ? (5)
3. (a) Determine an LCS of  $< 1, 0, 0, 1, 0, 1, 0, 1 >$  and  $< 0, 1, 0, 1, 1, 0, 1, 1, 0 >$ . (5)
  - (b) Obtain a set of optimal Huffman codes for the messages  $(M_1, \dots, M_y)$  with relative frequencies  $(q_1, q_2, \dots, q_y) = (4, 5, 7, 8, 10, 12, 20)$ . Draw the decode tree for this set of codes. (5)

4. (a) Write the pseudo code for MAKE-SET, FIND-SET and UNION using the linked-list representation and the weighted-union heuristic. (5)
- (b) Suppose that the graph  $G = (V, E)$  is represented as an adjacency matrix, give a simple implementation of Prim's algorithm for this case that runs in  $O(V^2)$  time. (5)
5. (a) Describe the generalization of the FFT procedure to the case in which 'n' is a power of 3. Give a recurrence for the running time, and solve the recurrence. (5)
- (b) Suppose that all characters in the pattern  $p$  are different. Show how to accelerate NAÏVE-STRING-MATCHER to run in time  $O(n)$  on an  $n$ -character text  $T$ . (5)
6. (a) Write a backtracking algorithm for the sum of subsets problem using the state space tree corresponding to the variable tuple size formulation. (5)
- (b) Present an algorithm for a FIFO branch-and-bound search for a least-cost answer node. (5)
7. (a) Is the following problem in P ? If yes, give a polynomial time algorithm, if not, show it is NP-complete.  
Input are an undirected graph  $G = (V, E)$  of degree 1000 and an integer  $k(\leq |V|)$ . Decide whether  $G$  has a clique of size  $k$ . (5)
- (b) Write the approximation algorithm for solving the TSP problem. (5)
8. Write short notes on any two: (5  $\times$  2)
- (a) Elements of Greedy strategy
  - (b) Hamiltonian cycle
  - (c) Lower bounds on sorting
  - (d) NP-hard problems.

### SOLUTION TO FOURTH SEMESTER EXAMINATION–2012

---

1. (a) The given two functions  $f_1(n) = n^2$  and  $f_2(n) = 2^n/4 = 2^{n-2}$

$n$	$f_1(n) = n^2$	$f_2(n) = 2^{n-2}$
$n = 0$	0	1/4
$n = 1$	1	1/2
$n = 2$	4	1
$n = 3$	9	2
$n = 4$	16	4
$n = 5$	25	8
$n = 6$	36	16
$n = 7$	49	32
$n = 8$	64	64
$n = 9$	81	128

For  $n = 9$  the function  $f_2(n) = 2^{n-2}$  becomes larger than  $f_1(n) = n^2$ .

(b) Suppose  $o(g(n)) \cap w(g(n))$  is not an empty set. Then rightly  $f(n) \in o(g(n)) \cap w(g(n))$ .

Now  $f(n) = w(g(n))$ , if and only if  $g(n) = o(f(n))$  and  $f(n) = o(g(n))$  by assumption.

By transitive property  $f(n) = o(f(n))$  i.e., for all constants  $c > 0$ ,  $f(n) < cf(n)$ . For  $c < 1$ , we have the desired contradiction from the asymptotic non-negativity of  $f(n)$ .

$$(c) T(n) = 2T(n/4) + \sqrt{n}$$

Here

$$a = 2, b = 4, f(n) = \sqrt{n}.$$

Therefore,

$$n^{\log_b a} = n^{\log 4^2} = n^{0.5} = \sqrt{n}$$

So,

$$f(n) = n^{\log_b a} = \sqrt{n}$$

Applying case-II of Master theorem we have

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a} \log^{k+1} n) \\ &= \Theta(n^{0.5} \log^{1.5} n) = \Theta(\sqrt{n}(\log n + \sqrt{\log n})) \end{aligned}$$

Upper bound is  $O(n^{0.5} \log^{1.5} n)$  and lower bound is  $\Omega(n^{0.5} \log^{1.5} n)$ .

- (d) The minimum and maximum number of elements in a heap of height  $h$  are  $2^h$  and  $(2^{h+1} - 1)$  respectively.
- (e) If an array A contains  $n$  numbers of distinct elements and the elements are stored in decreasing order, then the running time of quicksort is  $\Theta(n^2)$ .
- (f) The optimal parenthesization of matrix-chain product of the given sequences of dimension  $< 5, 10, 3, 12, 5, 50, 6 >$  becomes

	1	2	3	4	5	6
1	$m_{11}$ 0	$m_{12}$ 150	$m_{13}$ 330	$m_{14}$ 405	$m_{15}$ 1655	$m_{16}$ 2010
2		$m_0$ 0	$m_{23}$ 360	$m_{24}$ 330	$m_{25}$ 2430	$m_{26}$ 1950
3			$m_{33}$ 0	$m_{34}$ 180	$m_{35}$ 930	$m_{36}$ 1770
4				$m_{44}$ 0	$m_{45}$ 3000	$m_{46}$ 1860
5					$m_{55}$ 0	$m_{56}$ 1500
6						$m_{66}$ 0

- (g) To prove the fractional knapsack has the greedy-choice property we need to show that as much as possible of the highest value/pound item must be included in the optimal solution.

Let item  $h$  having weigh  $w_h$  and value  $V_h$  and it has got the highest value/pound ratio ( $V_h/w_h$ ). Again  $L(i)$  be the weight of item  $i$  contained in the thief's loot  $L$ . Then

total value  $V = \sum_{i=1}^n L(i) V_i / w_i$ . If some of item  $h$  is left and  $L(j) \neq 0$  for some  $j \neq h$  then replacing  $j$  with  $h$  will yield a higher value. This means  $L(j) (V_j/w_j) \leq L(h) (V_h/w_h)$ . This implies  $(V_j/w_j) \leq (V_h/w_h)$ .

- (h) The presence of a negative weight cycle can be determined by looking at the diagonal of the predecessor matrix computed by an all-pairs shortest path algorithm. If the diagonal contains any negative number, there must be a negative weight cycle.

(i) The number of spurious hits = 3.

- (ii) To show Hamiltonian path is NP-complete we take the help of Prover and Verifier. Through their conversation process we will reach to any final verdict.

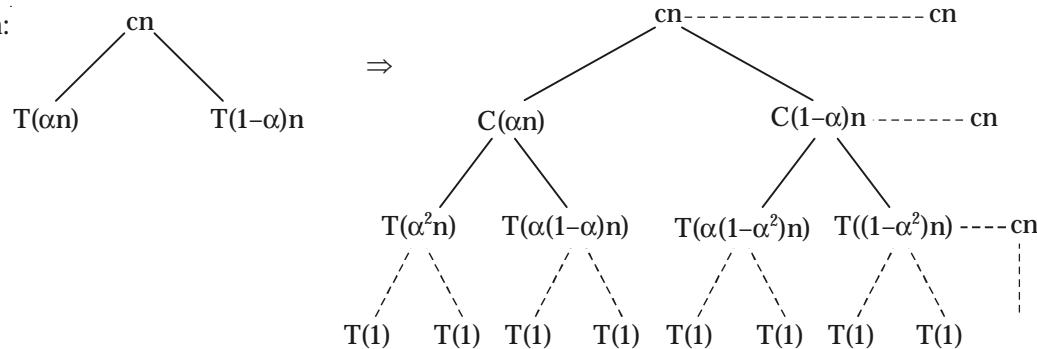
Prover	Verifier
<p>(1) Prover is given the input graph <math>G</math>. After checking he announces <math>G</math> has a Hamiltonian path.</p> <p>(3) Prover supplies a set of edges <math>\langle u_1, u_2, \dots, u_n \rangle</math> to Verifier which forms a Hamiltonian path in the graph <math>G</math>.</p>	<p>(2) Verifier asks Prover: why the graph <math>G</math> has a Hamiltonian path? Justify your answer.</p> <p>(4) Verifier performs the verification process that takes polynomial time in the following way.</p> <ul style="list-style-type: none"> <li>• Edges should come in sequence.</li> <li>• No repetition of vertex is allowed.</li> <li>• Total number of vertex to be <math>n</math>.</li> </ul>

As the verification process takes place in polynomial time the Hamiltonian path problem is in class NP.

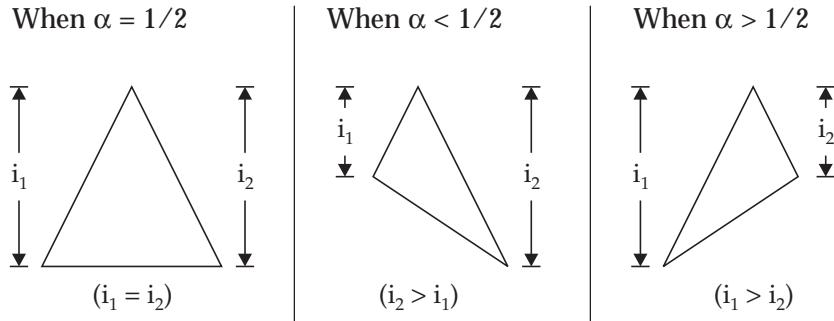
2. (a) The given recursion equation  $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$

The recurrence tree becomes

Tn:



To solve the recurrence we have now three situation i.e.,  $\alpha = 1/2$ ,  $\alpha < 1/2$  and  $\alpha > 1/2$ .



Therefore,

$$i_1 = \log_{1/\alpha} n = \frac{\log n}{\log 1/\alpha} \quad (\text{Applying } \log_a b = \frac{\log_c b}{\log_c a})$$

and

$$i_2 = \log \frac{1}{1-\alpha} n$$

Now the recurrence relation becomes

$$\begin{aligned} (1 + i_1)cn &\leq T(n) \leq (1 + i_2)cn \\ \Rightarrow c_1 n \log n &\leq T(n) \leq c_2 n \log n \\ \Rightarrow T(n) &= \Theta(n \log n) \end{aligned}$$

- (b) The best case time complexity of Mergesort is  $\Theta(n)$ , where  $n$  is the given number of elements.

The recurrence for the Mergesort is given by

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

It is of the form

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

Here,

$$a = 2, b = 2, f(n) = cn.$$

Therefore,

$$n \log_b^a = n \log_2^2 = n.$$

By applying case-II of Master theorem we get

$$T(n) = \Theta(n \log_b^a \log n) = \Theta(n \log n).$$

This shows the worst case running time of Mergesort is  $\Theta(n \log n)$ .

As the growth rate of function  $f(n \log n)$  is faster than  $f(n)$  it is obvious that  $\Theta(n)$  is the best case time complexity and  $\Theta(n \log n)$  is the worst case running time.

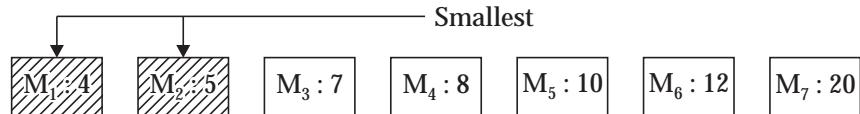
3. (a) Refer example 12.3 for solution.

- (b) For the messages  $(M_1, M_2, \dots, M_y)$  the characters and their relative frequencies are given as follows.

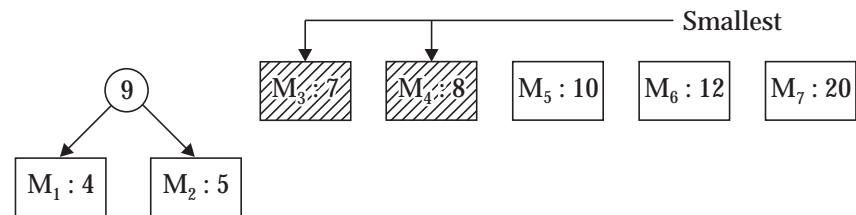
$$M_1 : 4, M_2 : 5, M_3 : 7, M_4 : 8, M_5 : 10, M_6 : 12, M_7 : 20.$$

Now performing the Huffman coding for the given characters:

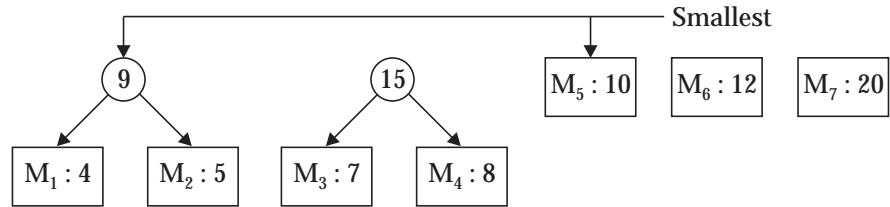
**Step 1 :**



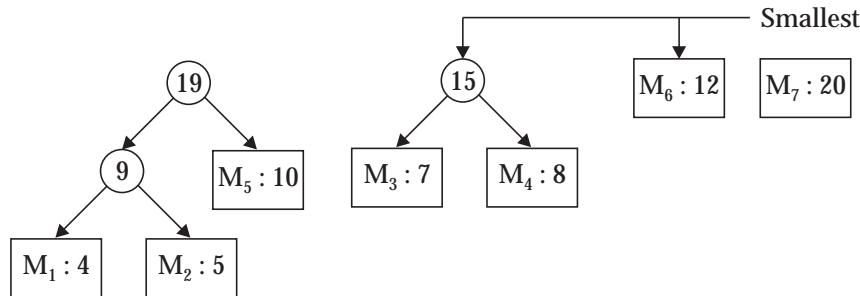
**Step 2 :**



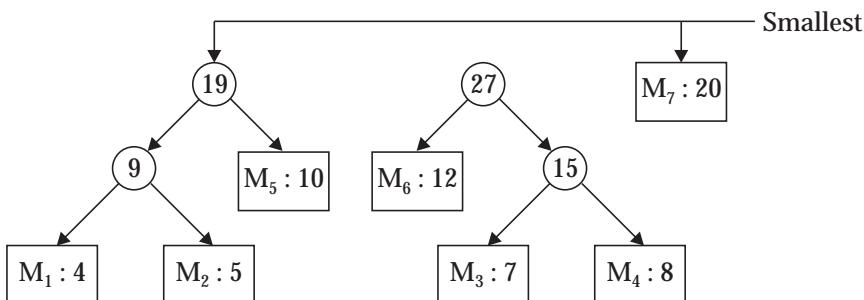
**Step 3 :**



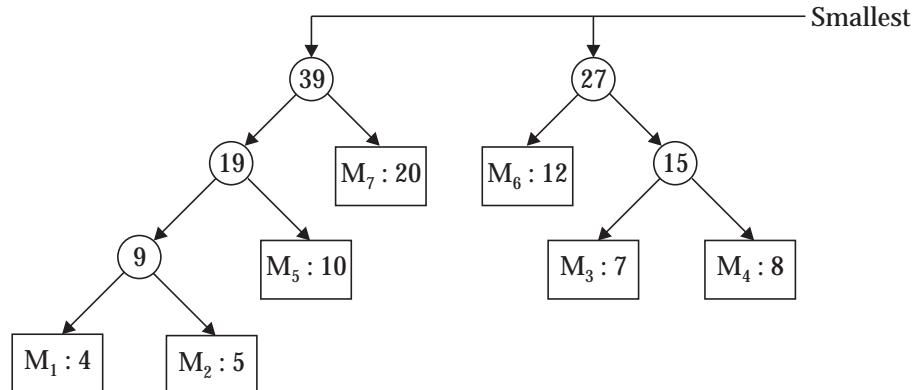
**Step 4 :**



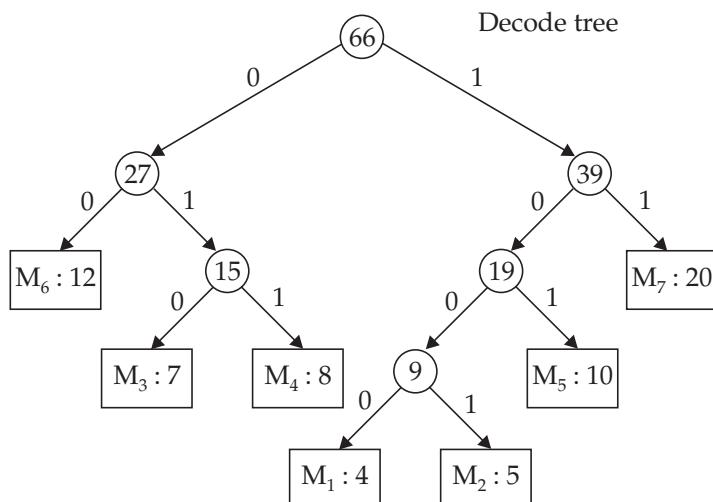
**Step 5 :**



## Step 6:



### **Step 7:**



Codes for the each characters are  $M_1 : 1000$ ,  $M_2 : 1001$ ,  $M_3 : 010$ ,  $M_4 : 011$ ,  $M_5 : 101$ ,  $M_6 : 00$ ,  $M_7 : 11$ .

#### 4. Prim's algorithm using adjacency matrix:

1. Create a set MST-SET that keeps track of vertices already included in the MST.
  2. Assign a key value to all vertices in the input graph. Assign the key value of the start (first) vertex to be 0 and rest others to be assigned as Infinite.
  3. While MST-SET doesn't include all vertices:
    - (i) Pick a vertex  $u$  which is not there in MST-SET and has minimum key value.
    - (ii) Include  $u$  to MST-SET.
    - (iii) Update a key value of all adjacent vertices of  $u$ . To update the key values, iterate through all adjacent vertices. For every adjacent vertex  $v$ , if weight of edge  $u-v$  is less than the previous key value of  $v$ , update the weight of  $u-v$ .

**5. Naive string matching algorithm:**

NAIVE-STRING-MATCHER (T, P)

1.  $n = \text{length}[T]$  //  $n$  is the length of the given text T
2.  $m = \text{length}[P]$  //  $m$  is the length of the given pattern P
3. For  $s = 0$  to  $n - m$  //  $s$  refers to the shift
4. if  $P[1, \dots, m] = T[s + 1, \dots, s + m]$  // P and T are strings of characters.
5. then print "patterns occur with shift"  $s$ .

When  $m = n$  i.e., the length of the given pattern is equal to the length of the given text then the running time of NAIVE-STRING-MATCHER takes  $O(n)$  time-complexity to run.

**6. (a) The backtracking algorithm for subset problem can be given as follows:**

SUBSETSUM (S[1, ..., n], T) // where T is the target value

1. if ( $T = 0$ )
2. return TRUE
3. else if ( $T < 0$  or  $T \neq 0$ )
4. return FALSE
5. else
6. return SUBSETSUM (S[2, ..., n], T) or SUBSETSUM (S[2, ..., n], T - S[1])

For example, if  $S = \{1, 2, 3, 4, 6, 7, 8\}$  and  $T = 10$ , then the subset sum includes the subsets  $\{1, 2, 3, 4\}$  or  $\{1, 3, 6\}$  or  $\{2, 8\}$  or  $\{3, 7\}$  or  $\{4, 6\}$  for which the answer is TRUE. For the subset  $\{1, 2, 3, 6\}$  the answer is false.

**(b) Algorithm for a FIFO branch-and-bound search for a least cost answer node:**

LC-SEARCH // search for answer node

1. if ( $t$  be the answer node)
2. then return  $t$
3. Initialize the list of live nodes to be empty i.e.,  $E = t$
4. while (each child  $x$  of  $E$ )
5. if ( $x$  is the answer node)
6. then return the path from  $x$  to  $t$
7. add  $x$  to the list of live nodes
8. parent of  $x$  becomes  $E$
9. else if (there is no more live nodes present)
10. then no answer node is present
11. Find the live node with least estimated cost and delete this node from the list of live nodes.

7. (a) The clique problem is NP-complete. It has no polynomial time algorithm exists.

To show clique problem is NP-complete and to decide whether graph G has a clique of size  $k$  refer example 17.13.

(b) **Approximation algorithm for TSP problem:**

1. Arbitrary choose as one of the vertices as start vertex for the given graph G.
2. Find the MST (minimum spanning tree) by using Prim's algorithm taking the start vertex as source vertex.
3. Find the sequence of vertices to be visited by DFS traversal. Do this operation from the source vertex of MST of step 2.
4. This sequence of vertex is the order to be visited by the salesman and make it cycle (called minimum Hamiltonian cycle).

8. (a) **Elements of Greedy Strategy:** The different elements of Greedy strategy includes:

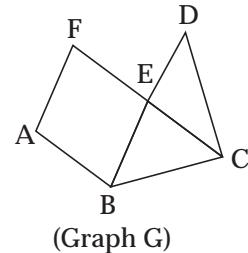
(i) **Greedy-choice property:** In Greedy algorithm when we are considering which choice to make, we make the choice that looks best in the current problem, without considering the results from subproblem. Greedy strategy usually progresses in top-down fashion, making one Greedy choice after another, reducing each given problem to a smaller one.

(ii) **Optimal substructure:** A problem exhibits optimal substructure if an optimal solution to the problem contains within its optimal solutions to the subproblems.

(b) **Hamiltonian cycle:** A Hamiltonian cycle in a graph G is a cycle which spans all vertices exactly except the start vertex. Moreover it is considered as a close path. For example, for the given graph G A-B-C-D-E-F-A forms a Hamiltonian cycle.

(c) **Lower bounds on sorting:** The lower bounds on sorting helps us to understand how close we are to the best possible solution to some problem, i.e., if we have an algorithm that runs in  $O(n \log^2 n)$  time complexity and a lower bound of  $\Omega(n \log n)$  then we have a  $\log(n)$  gap which denotes the maximum possible saving we could hope to achieve by improving our algorithm. In lower bound if we say the algorithm takes  $\Omega(g(n))$  time then we are much more concern about  $g(n)$  as large as possible. For example, mergesort is optimal as it requires  $\Omega(n \log n)$  comparisons are necessary for sorting elements.

(d) **NP-hard problems:** A problem P is NP-hard if and only if satisfiability (SAT) reduces to P. In other words if there is an efficient polynomial time algorithm for P then there is one for every problem in NP. Different examples of problems that are in NP-hard include: longest path problem, clique problem, vertex cover, independent set, subset sum, etc.



**Fourth Semester Examination–2011**  
**DESIGN AND ANALYSIS OF ALGORITHM**  
**BRANCH : CSE/IT**  
**QUESTION CODE : 4204**  
**Full Marks–70**  
**Time : 3 Hours**

*Answer Question No. 1 which is compulsory and any five from the rest. The figures in the right-hand margin indicate marks.*

1. Answer the following questions: (2 × 10)
  - (a) Differentiate between priori analysis and posteriori analysis.
  - (b) Which of the following is not  $O(n^2)$ ? Justify your answer.  
 $n + 10000 n, n^{1.9999}, 10^5 n + 2^6 n, n^3 / \sqrt{n}$
  - (c) Working modulo  $q = 11$ , how many spurious hits does the Rabin-Karp matcher encounter in the text  $T = 31415926$  when looking for the pattern  $P = 26$ .
  - (d) Write the formula used in Euclid's algorithm for finding the greatest common divisor of two numbers ?
  - (e) Give the recurrence equation for the worst case behavior of merge sort.
  - (f) Name any two methods for pattern matching ?
  - (g) When do you say a tree as minimum spanning tree ?
  - (h) How will you construct an optimal binary search tree ?
  - (i) Define backtracking.
  - (j) What is Hamiltonian cycle in an undirected graph ?
2. (a) Write and explain the control abstraction for Divide and conquer. (5)
  - (b) Consider a set  $S$  of  $n \geq 2$  distinct numbers given in unsorted order and,  $x$  and  $y$  are two distinct numbers in the set  $S$ . Write an  $O(n)$  time algorithm to determine,  $x, y \in S$  such that  $|x - y| \geq |w - z|$  for all  $w, z \in S$
3. (a) Write Greedy algorithm to generated shortest path. (5)
  - (b) Find the minimum number of operations required for the following chain matrix multiplication using dynamic programming. (5)

$$A(30, 40) * B(40, 5) * C(5, 15) * D(15, 6).$$
4. (a) Show that  $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$  where  $f_1(n) = O(g_1(n))$  and  $f_2(n) = O(g_2(n))$ . (5)
  - (b) Solve the following 0/1 Knapsack problem using dynamic programming : (5)

$$P = (11, 21, 31, 33), W = (2, 11, 22, 15), C = 40, n = 4.$$

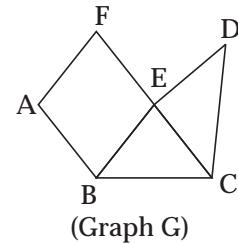
5. (a) Explain the properties of strongly connected components. (5)  
 (b) Write a non-recursive algorithm of In-order traversal of a tree and also analyze its time complexity. (5)
6. (a) Sort the given set of surnames using Heap sort Algorithm. Find its worst case time complexity  
 MISHRA, ACHRAYA, MOHANTY, RATHA, PATTNAYAK, BRAHMA, DASH, MISTRY, MAJHI, BEHERA  
 (b) Solve the 4-queens problem using backtracking. (5)
7. (a) Describe problem state, solution state and answer state with an example. (5)  
 (b) Explain the general method of Branch and Bound. (5)
8. (a) Explain the classes of P and NP. (5)  
 (b) Explain the method of reduction to solve TSP problem using Branch and Bound. (5)

### **SOLUTION TO FOURTH SEMESTER EXAMINATION-2011**

---

1. (a) Priori analysis is based on determining the order of the magnitude of the statement. This method is independent of machine, programming language and operating system.  
 Posteriori analysis refers to the technique of coding a given solution and then measuring its efficiency which provides the actual time taken by the program.
- (b) We can say  $n^3/\sqrt{n} \neq O(n^2)$  because  $n^3/\sqrt{n} = n^{2.5} > n^2$ . So  $O(n^2)$  cannot be treated as the upper bound of  $n^3/\sqrt{n}$ .
- (c) Number of spurious hits is 3.
- (d) Euclid's algorithm :  
 Euclid ( $m, n$ ) // Assuming  $m$  and  $n$  are two integers and  $n > m$   
 {  
   while  $m$  doesn't divide  $n$   
   {  
      $r = n \bmod m$   
      $n = m$   
      $m = r$   
   }  
   return  $m$   
 }
- (e) The recurrence for the worst case behaviour of merge sort is given as  $T(n) = 2T(n/2) + cn$ .

- (f) Two methods for pattern matching are : Rabin- Karp pattern matching algorithm and Knuth-Morris-Pratt pattern matching algorithm.
  - (g) A tree is called minimum spanning tree if the sum of the weights of its edges is no larger than the weight of any other spanning tree.
  - (h) We will construct an optimal binary search tree by using dynamic programming or by using a recursive solution.
  - (i) Backtracking is a recursive algorithm strategy that tries to build a solution to a computational problem incrementally. In other words, it is a methodical way of trying out various sequences of decisions until we find one that works.
  - (j) A Hamiltonian cycle in a graph G is a cycle that spans all vertices exactly once except the start vertex. It is a closed path. For the given graph G A-B-C-D-E-F-A forms a Hamiltonian cycle.
2. (a) A control abstraction is a procedure that reflects the way an actual program based on divide and conquer will look like. The control abstraction shows clearly the flow of control but the primary operations are specified by other procedures. The control abstraction can be written either iteratively or recursively.



If we are given a problem with  $n$  inputs and if it is possible for splitting the  $n$  inputs into  $k$  subsets where each subset represents a sub problem similar to the main problem then it can be achieved by using divide and conquer strategy.

A general divide and conquer design strategy (control abstraction) is illustrated as follows:

```

Algorithm DC (P)
{
  If small (P) then return P
  else Divide P into smaller instances P1, P2, P3, ......., Pk where 1 ≤ k ≤ n.
  Apply DC to each of these subproblems.
  Return combine (DC(P1), DC(P2), .....DC(Pk))
}
  
```

The above blocks of codes represent control abstraction for divide and conquer strategy. Small (P) is a boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting. If P is small then we terminate the condition. Otherwise the problem P is divided into sub problems. These sub problems are solved by recursive application of divide- and-conquer. Finally the solution from  $k$  sub problems is combined to obtain the solution to the given problem.

**(b) Algorithm steps :**

1. Let S be the set of  $n \geq 2$  distinct numbers.

2. Let  $x, y, w, z$  be the four elements of  $S$ .
  3. If mod of  $(x - y) \geq$  mod of  $(w - z)$
  4. then return TRUE
  5. else return FALSE
3. (a) Dijkstra's algorithm is a Greedy algorithm which is used to generate shortest path. The algorithm can be described by the following steps:

DIJKSTRA ( $G, w, S$ ) //  $w : E \rightarrow R^+$ ,  $S$  is the source vertex

1. For all  $v \in V[G] - S$
2. do  $d[v] = \infty, \pi[v] = NIL$  //  $d$  is the distance from source and  $\pi$  is the predecessor
3.  $d[S] = 0, \pi[S] = NIL$
4. For all  $v \in V[G]$  do ENQUEUE ( $PQ, V$ ) // ENQUEUE is the process through which we enter elements into priority queue.
5. While ( $PQ \neq \emptyset$ )
6. do  $u = DEQUEUE(PQ)$  // DEQUEUE is the operation through which we delete elements from priority queue according to the priority.
7. For all  $v \in \text{Adjacent}[u]$
8. do RELAX ( $u, v, x$ )

(b) Given dimensions for matrix-chain-multiplication

$$p[0] = 30, p[1] = 40, p[2] = 5, p[3] = 15, p[4] = 6$$

The values of  $m_{11}, m_{22}, m_{33}$  and  $m_{44}$  are all equal to zero since  $i = j$ .

	1	2	3	4
1	$m_{11}$	$m_{12}$	$m_{13}$	$m_{14}$
2		$m_{22}$	$m_{23}$	$m_{24}$
3			$m_{33}$	$m_{34}$
4				$m_{44}$

$$\begin{aligned} m_{12} &= \min_{i \leq k \leq j-1} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]] \\ &= \min_{1 \leq k \leq 1} [m_{11} + m_{22} + p[0] \cdot p[1] \cdot p[2]] = 0 + 0 + (30 \times 40 \times 5) = 6000 \end{aligned}$$

$$m_{23} = m_{22} + m_{33} + p[1] \cdot p[2] \cdot p[3] = 0 + 0 + (40 \times 5 \times 15) = 3000$$

$$m_{34} = m_{33} + m_{44} + p[2] \cdot p[3] \cdot p[4] = 0 + 0 + (5 \times 15 \times 6) = 450$$

$$m_{13} = \min_{1 \leq k \leq 2} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

when  $k = 1$ ,  $m_{13} = m_{11} + m_{23} + p[0] \cdot p[1] \cdot p[3] = 0 + 3000 + (30 \times 40 \times 15) = 21000$

when  $k = 2$ ,  $m_{13} = m_{12} + m_{33} + p[0] \cdot p[2] \cdot p[3] = 6000 + (30 \times 5 \times 15) = 8250$

For  $k = 2$ ,  $m_{13}$  is minimum i.e., 8250.

$$m_{24} = \min_{2 \leq k \leq 3} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

when  $k = 2$ ,  $m_{24} = m_{22} + m_{34} + p[1] \cdot p[2] \cdot p[4] = 450 + (40 \times 5 \times 6) = 1650$

when  $k = 3$ ,  $m_{24} = m_{23} + m_{44} + p[1] \cdot p[3] \cdot p[4] = 3000 + (40 \times 15 \times 6) = 6600$

For  $k = 2$ ,  $m_{24}$  is minimum i.e., 1650.

$$m_{14} = \min_{1 \leq k \leq 4} [m_{ik} + m_{(k+1)j} + p[i-1] \cdot p[k] \cdot p[j]]$$

when  $k = 1$ ,  $m_{14} = m_{11} + m_{24} + p[0] \cdot p[1] \cdot p[4] = 1650 + (30 \times 40 \times 6) = 8850$

when  $k = 2$ ,  $m_{14} = m_{12} + m_{34} + p[0] \cdot p[2] \cdot p[4] = 6000 + 450 + (30 \times 5 \times 6) = 7350$

when  $k = 3$ ,  $m_{14} = m_{13} + m_{44} + p[0] \cdot p[3] \cdot p[4] = 8250 + (30 \times 15 \times 6) = 10950$

For  $k = 2$ ,  $m_{14}$  is minimum i.e., 7350.

So the final table becomes

	1	2	3	4
1	0	6000	8250	7350
2		0	3000	1650
3			0	450
4				0

4. (a) According to the basic definition of big-O,

$$f_1(n) = f_2(n) \leq c (\max(f_1(n), f_2(n))) \text{ where } c \text{ is a constant.}$$

$$\text{Again, } f_1(n) + f_2(n) = \max(f_1(n), f_2(n)) + \min(f_1(n), f_2(n))$$

$$\text{and } \max(f_1(n), f_2(n)) \geq \min(f_1(n), f_2(n))$$

Therefore we can write,

$$f_1(n) + f_2(n) \leq 2 \max(f_1(n), f_2(n))$$

so  $c = 2$ . We have given  $f_1(n) = O(g_1(n))$ , and  $f_2(n) = O(g_2(n))$

This implies,  $f_1(n) + f_2(n) \leq 2 \max(O(g_1(n)), O(g_2(n)))$

Hence,  $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$ .

(b) Given specifications of 0/1 Knapsack problem

Profit (or value) =  $P = (11, 21, 31, 33)$

Weight =  $W = (2, 11, 22, 15)$

Capacity of Knapsack =  $C = 40$

Number of items =  $n = 4$

Applying dynamic programming to solve this problem, we construct the following table.

Capacity →			0	2	11	13	15	17	22	24	26	28	33	35	37	39
Item	Profit	Weight	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	11	2	0	11	11	11	11	11	11	11	11	11	11	11	11	11
2	21	11	0	11	21	32	32	32	32	32	32	32	32	32	32	32
3	31	22	0	11	21	32	32	32	32	44	44	44	52	63	63	63
4	33	15	0	11	21	32	33	44	44	44	54	65	65	65	65	75

$$P[4, 40] = 75 \text{ (for taking items 1, 3 and 4)}$$

5. (a) Strongly connected component is an application of DFS. A strongly connected component of a directed graph  $G = (V, E)$  is a maximal set of vertices  $C \subseteq V$  such that for every pair of vertices  $u$  and  $v$  in  $C$ , we have both  $u \rightsquigarrow v$  and  $v \rightsquigarrow u$ ; that is, vertices  $u$  and  $v$  are reachable from each other.

While finding the strongly connected components of a given graph  $G$  we first traverse the graph in DFS and compute the finishing times for each vertex. Then we compute the transpose of graph  $G$  i.e.,  $G^T$ . Then we again apply DFS on  $G^T$  and arrange the vertices in order of decreasing finishing time stamp. Then we output the vertices of each tree in the DFS formed above and separate strongly connected components. This component graph is a DAG (directed acyclic graph).

Another key property is :

- Let  $C_1$  and  $C_2$  be distinct strongly connected components in a directed graph  $G = (V, E)$ . Suppose that there is an edge  $(u, v) \in E$ , where  $u \in C_1$  and  $v \in C_2$ . The finishing time stamps of vertex in  $C_1$  are greater than finishing time stamp of vertex in  $C_2$  i.e.,  $f(C_1) > f(C_2)$ .

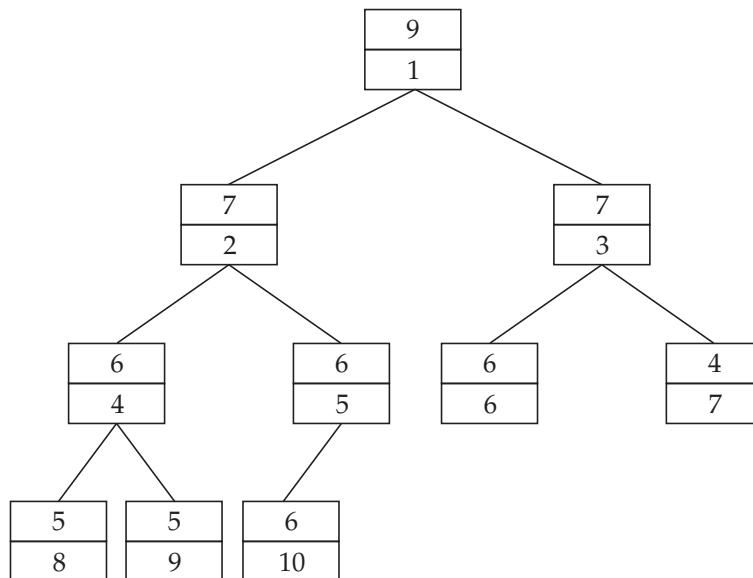
- (b) Using stack (which is a LIFO data structure) we can traverse a tree without recursion. Below is an algorithm for traversing binary tree using stack.

#### ALGORITHM IN-ORDER TRAVERSAL

- Create an empty stack  $S$
- Initialize current node as root
- Push the current node to  $S$  and set  $current = current \rightarrow left$  until  $current$  is NULL

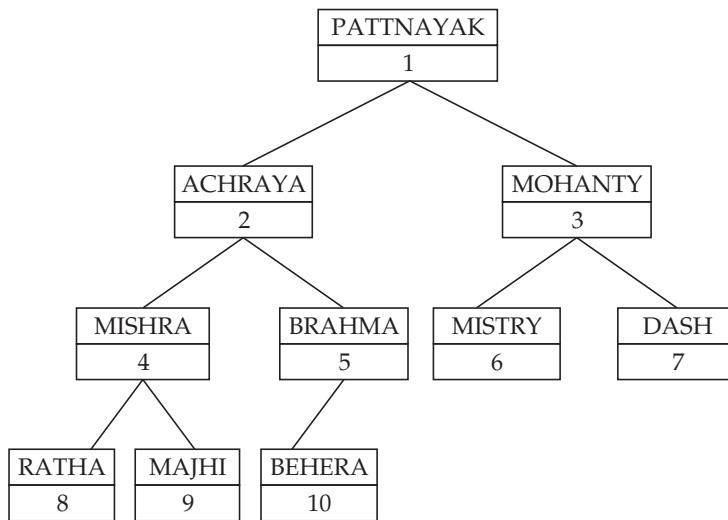
4. If current is NULL and stack is not empty then
  - (i) Pop the top item from stack
  - (ii) Print the current item, set current = current → right
  - (iii) Go to step 3
5. If current is NULL and stack is empty then we are done. Running time of the above algorithm is  $O(n)$ ; where  $n$  is the number of nodes in the binary tree.
6. (a) We sort the given surnames according to their length. First find the length of each surname and then apply Heap sort algorithm. MISHRA-6, ACHRAYA-7, MOHANTY-7, RATHA-5, PATTNAYAK-9, BRAHMA-6, DASH-4, MISTRY-6, MAJHI-5, BEHERA-6.

We build max-heap from the array  $<6, 7, 7, 5, 9, 6, 4, 6, 5, 6>$  where number of elements  $n = 10$ . Applying MAX-HEAPIFY for  $i = \lfloor 10/2 \rfloor$  to 1 i.e., 5 to 1 we get the following MAX-HEAP. Here  $i$  denotes index value.



**(Note.** Readers are advised to perform the entire MAX-HEAPIFY procedure here to get this final result).

Now putting the surnames in their corresponding length, the MAX-HEAP will look like as follows:



The worst case time complexity of heapsort is  $O(n \log n)$ ; where  $n$  is the number of elements.

(b) Refer example 15.1.

7. (a) Problem state is each node in the depth-first search tree. Solution state is the problem state  $s$  for which the path from the root no to  $s$  defines a tuple in the solution space. Answer state is the solution state  $s$  for which the path from root node to  $s$  defines a tuple that is a member of this set. Example- refer the example 15.1 for 4-queens problem.

(b) Branch and bound is a general algorithm for finding optimal solutions of various optimization problems. In this method, we assume that the goal is to find the minimum value of a function  $f(x)$ , where ranges over some set of admissible or candidate solution (the search space or feasible region).

The branch-and-bound procedure requires two tools. The first one is a splitting procedure that given a set  $S$  of candidates, returns two or more smaller sets  $S_1, S_2, \dots$  whose union covers  $S$ . This step is called branching, since its recursive application defines a tree structure (called recursion tree) whose nodes are the subsets of  $S$ .

The second tool is a general procedure that computes upper and lower bounds for the minimum value of  $f(x)$  within a given subset of  $S$ . This process is called bounding.

8. Class P : P is the class of decision problems that can be solved in polynomial time. In other words, given an instance of a decision problem in this class, there is a polynomial time algorithm for deciding if the answer is either YES or NO. For example, let us consider the shortest path problem. Given a graph  $G = (V, E)$  and a positive integer  $k$ , does there a shortest path from vertex  $u$  to  $v$  of length  $k$ ? We can certainly solve this problem by running Dijkstra's algorithm in polynomial time. Therefore, the shortest path problem is in class P.

Class NP : NP stands for nondeterministic polynomial and defined as it is class of decision problems that have polynomial time verifiers. Examples of problems that are in NP class: Clique, Hamiltonian Cycle, Hamiltonian Path etc.

**CHAPTER – 1 and 2****Overview of Design and Analysis of Algorithms and Framework of Algorithm Analysis****1. What do you mean by “Algorithm” ?**

**Ans.** Algorithm is an abstract computational procedure which takes some value or values as input and produces a value or values as output.

**2. What is a “sorting problem” ?**

**Ans.** We define sorting problem as below:

Input: A sequence of  $n$  numbers,  $\langle x_1, x_2, x_3, \dots, x_n \rangle$

Output: A permutation (reordering)  $\langle x'_1, x'_2, x'_3, \dots, x'_n \rangle$  of the input sequence such that  $x'_1 \leq x'_2 \leq x'_3, \dots, \leq x'_n$ .

**Example:** For the given input sequence  $\langle 4, 12, 15, 3, 8, 13 \rangle$  a sorting algorithm returns  $\langle 3, 4, 8, 12, 13, 15 \rangle$ .

**3. When the newly made algorithm is said to be correct?**

**Ans.** A newly made algorithm is said to be correct if for every input instance, it halts with the correct output.

**4. Give some practical day to day life examples where we use algorithms?**

**Ans.** We use algorithms in hardware design, GUI (Graphical User Interface) based operating systems, Routers in networks (Dijkstra algorithms, Minimum Spanning Tree), Compiler, interpreter or assembler.

**5. What is the need of loop invariants in algorithm? What are the basic steps of loop invariants?**

**Ans.** Loop invariants are used to find out why an algorithm is correct.

Basic steps are (i) Initialization (ii) Maintenance (iii) Termination.

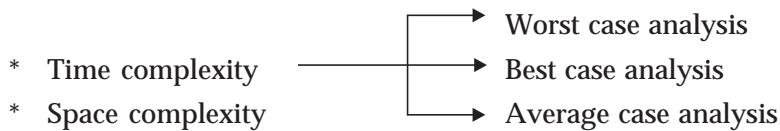
**Initialization:** It is true prior to the first iteration of the loop.

**Maintenance:** It is true before an iteration of the loop, it remains true before the next iteration.

**Termination:** When the loop terminates, the invariants gives us a useful property that helps us to know how that the algorithm is correct.

#### 6. How can you analyze the algorithm?

**Ans.** We can analyze the algorithm through



#### 7. What do you mean by "time complexity"?

**Ans.** Time complexity is the amount of time required by the algorithm to successfully complete its execution.

#### 8. What is worst case analysis?

**Ans.** It is the maximum time required by the algorithm to successfully complete execution while performing time complexity analysis.

#### 9. What is best case analysis?

**Ans.** It is the minimum time required by the algorithm to successfully complete execution while performing time complexity analysis.

**Or**

It is the minimum amount of memory space required to successfully execute execution while performing space complexity analysis.

#### 10. What is average case analysis?

**Ans.** It is the average time taken on an average input size while performing time complexity.

#### 11. How testing of algorithm takes place?

**Ans.** Testing of algorithm is accompanied by 2 phases

(i) Debugging (it is the process of finding an error and fixing it)

(ii) Performance measurement (better performance algorithm is accepted)

#### 12. What is the complexity of an algorithm? Explain with example.

**Ans.** Complexity of an algorithm is of two types

(i) Time complexity - is the amount of time needed for the algorithm to complete its execution.

(ii) Space complexity - is the amount of memory space needed to successfully complete its execution.

### **Example**

- (i) Time complexity of heapsort algorithm is  $O(n \log n)$  whereas auxiliary space required for the algorithm is  $O(1)$ .
- (ii) Time complexity of quicksort is  $O(n \log n)$  whereas auxiliary space used in the average case for implementing recursive function calls is  $O(\log n)$ .

### **13. Briefly discuss the Algorithm Analysis Framework?**

**Ans.** The framework that we design could be used not only for comparing the execution time of algorithms which is what we primarily mean algorithm analysis but it could be also be used for comparing other resources that an algorithm might use.

For example, an algorithm might use varying amounts of memory. So we could use this framework to formally compare the memory requirements of different programs or different algorithms.

### **14. Define Worst case, Average case and Best case efficiencies.**

**Ans.** **Worst case:** The worst case efficiencies of an algorithm is the function defined by the maximum number of steps taken on any instance of size  $n$ . It gives us an upper bound on the running time.

**Average case:** The average case efficiencies of an algorithm is the function defined by an average number of steps taken on any instance of size  $n$ .

**Best case:** The best case efficiencies of an algorithm is the function defined by the minimum number of steps taken on any instance of size  $n$ . It is the lower bound on the running time.

### **15. What is the difference between performance analysis and performance measurement?**

**Ans.** **Performance Analysis:** To analyze performance of an algorithm is or determining the amount of resource (time and space) needed to execute it efficiently is stated as a reaction relating the input length to the number of steps (time complexity) or storage location (space complexity).

**Performance Measurement:** It describes the correct program execution for all possible data sets and it takes time and space to compute results.

### **16. Differentiate between “Priori Analysis” and “Posteriori Analysis”.**

**Ans.** **Priori Analysis:** The bounds of algorithms computing time are obtained by formulating a function. It is on theoretical basis that, it is independent of programming languages and machine structures. The stress is laid on the frequency of execution of statements.

**Posteriori Analysis:** The actual amount of space and time taken by the algorithms are recorded during execution. It is dependent on programming languages used and machine structures.

### **17. What is the need of randomized algorithm?**

**Ans.** Randomized algorithm makes random choices which allow a probabilistic analysis and yields an expected (average) running time.

**18. What do you mean by “analyzing algorithms”?**

**Ans.** Analyzing an algorithms means predicting the resources that the algorithm requires. Here resources such as memory, communication bandwidth or computer hardware are of primary concern, but most often it is the computational time that we want to measure.

**19. What are the different techniques used for designing algorithm?**

**Ans.** The different techniques used for designing algorithms are: divide and conquer approach, dynamic programming, greedy strategy etc.

**20. Write the formulae used in Euclid’s algorithm for finding the greatest common divisor of two numbers?**

**Ans.** **Input:** Two integers  $m$  and  $n$ .

**Ouput:** Largest integer that divides both.

Euclid’s formulae is

If  $m$  divides  $n$  then  $\text{GCD}(m, n) = m$

If not, then  $\text{GCD}(m, n) = \text{GCD}(n \bmod m, m)$

**21. How can you modify any algorithm to have a good best case running time?**

**Ans.** To improve the best case, all we have to do it to be able to solve one instance of each size efficiently. We could modify our algorithm to first test whether the input is the special instance we know how to solve, and then output the pre-computed answer.

**Example:** For a searching algorithm, the best case arises when the element we are searching is present at the first position. The best case running time is  $\theta(1)$ .

Similarly for a sorting algorithm, check whether the list is already sorted before trying to sort it, then the best case will be  $\theta(n)$ .

---

## CHAPTER –3

---

### Asymptotic Notation

**1. Define asymptotic notation.**

**Ans.** The asymptotic notations are used to describe the asymptotic running time of an algorithm and defined by the functions whose domains are the set of natural numbers i.e.,  $N = \{0, 1, 2, \dots\}$ .

**2. What are the different types of asymptotic notations are used?**

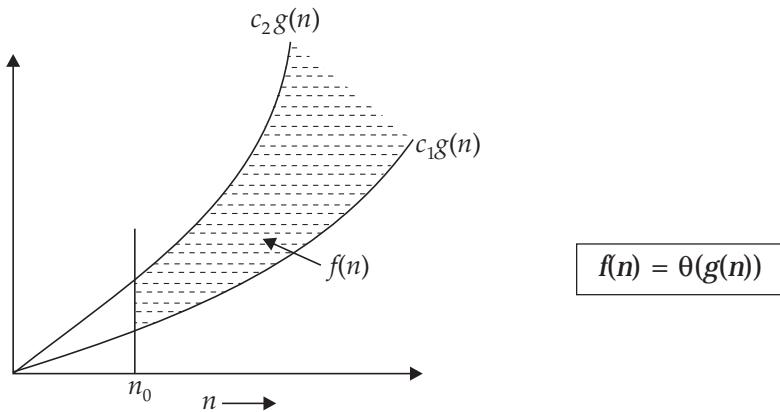
**Ans.** The different types of asymptotic notations are:

(i) Big-oh notation ( $O$ -notation), (ii) Big-Omega notation ( $\Omega$ -notation), (iii) Theta notation ( $\Theta$ -notation), (iv) Little-oh notation ( $o$ -notation) and (v) Little-omega notation ( $\omega$ -notation).

**3. Define  $\Theta$  – notation.**

**Ans.**  $\Theta$  notation bounds a function to within constant factors.

$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0\}$



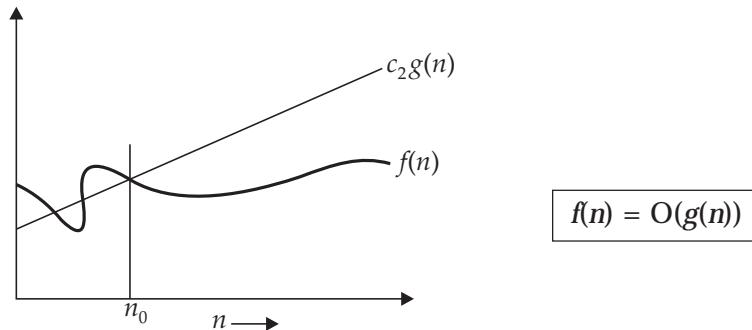
Here,  $c_2g(n)$  and  $c_1g(n)$  are asymptotic upper bound and asymptotic lower bound for  $f(n)$  respectively.

#### 4. Define O-notation.

Or

**Define  $O(g(n))$ .**

- Ans.** O-notation gives an asymptotic upper bound for a function to within a constant factor.  
 $O(g(n)) = \{f(n) \mid \text{there exist positive constant } c_2 \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$

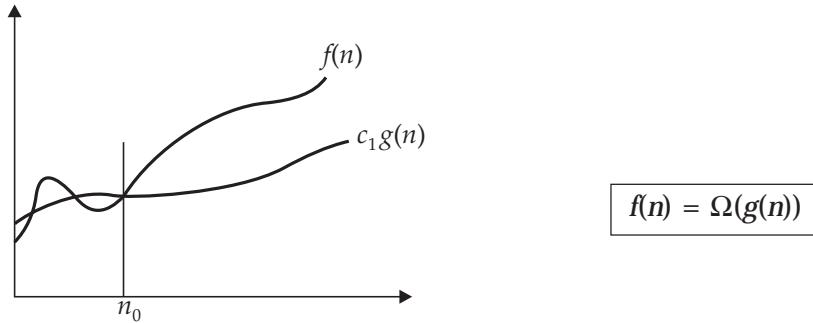


#### 5. Define $\Omega$ -notation.

Or

**Define  $\Omega(g(n))$**

- Ans.**  $\Omega$ -notation provides us asymptotic lower bound for a function within a constant factor.  
 $\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n), \forall n \geq n_0\}$



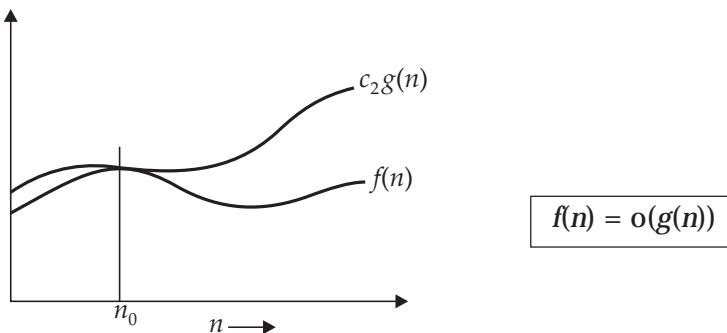
**6. Define little-oh notation (O-notation).**

Or

Define  $o(g(n))$ .

**Ans.** Little-oh notation is used to denote an upper bound that is not asymptotically tight. So it is defined as

$o(g(n)) = \{f(n) \mid \text{for any positive constant } c_2 > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < c_2 g(n), \forall n \geq n_0\}$



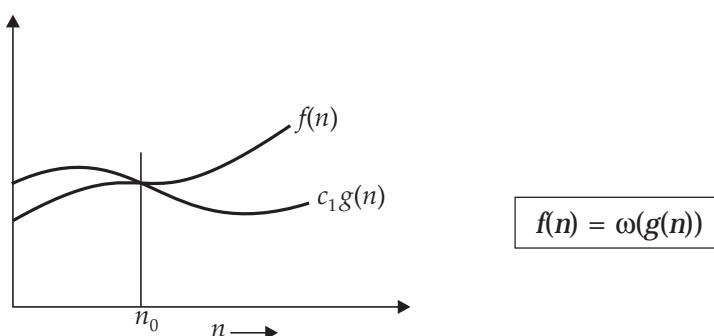
**7. Define little-omega notation ( $\omega$ -notation).**

Or

Define  $\omega(g(n))$ .

**Ans.** Little-omega ( $\omega$ -notation) notation is used to denote a lower bound that is not asymptotically tight.

$\omega(g(n)) = \{f(n) \mid \text{for any positive constant } c_1 > 0, \text{ there exist a constant } n_0 > 0 \text{ such that } 0 \leq c_1 g(n) < f(n), \forall n \geq n_0\}$



**8. Represent Little-oh notation in terms of limit.**

**Ans.** If  $f(n) = o(g(n))$

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

**9. Represent  $\omega$ -notation in terms of limit.**

**Ans.** If  $f(n) = \omega(g(n))$

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

**10. Can you compare the two functions  $n$  and  $n^{1 + \sin n}$  using asymptotic notation? Justify your answer.**

**Ans.** We can not compare the two functions  $n$  and  $n^{1 + \sin n}$  using asymptotic notation, as the value of the exponent  $n^{1 + \sin n}$  varies between 0 and 2.

**11. What is the relationship between  $\Theta$ ,  $O$  and  $\Omega$  notation?**

**Ans.**  $\Theta$  is the intersection of  $O$  and  $\Omega$ .

$$\text{So, } O(g(n)) \cap \Omega(g(n)) = \Theta(g(n)).$$

**12. Give an example of single non-negative function  $f(n)$  such that for all functions  $g(n)$ ,  $f(n)$  is neither  $O(g(n))$  nor  $\Omega(g(n))$ .**

**Ans.**  $f(n) = (1 + \sin n) \cdot 2^{2n+2}$

**13. Give an example of single non-negative function  $f(n)$  such that for all functions  $g(n)$ ,  $f(n) = O(g(n))$ .**

**Ans.**  $f(n) = 1/n$ .

**14. Give an example of single non-negative function  $f(n)$  such that for all functions  $g(n)$ ,  $f(n) = \omega(g(n))$ .**

**Ans.**  $f(n) = 2^{2n}$

**15. Find the growth of order in increasing sequence of  $n^2 \log n$ ,  $\log n$ ,  $n!$ ,  $2^n$ .**

**Ans.** The growth of order in increasing sequence is  $\log n$ ,  $n^2 \log n$ ,  $2^n$ ,  $n!$ .

**16. Which function grows at faster rate  $e^n$  or  $2^n$ ? Justify your answer.**

**Ans.** The function  $e^n$  grow faster than  $2^n$  as exponential function grows faster than power.

## CHAPTER – 4 to 9

**Insertion-sort, Divide and conquer approach, Recurrence, Binary search, Quicksort and Order-statistics**

**1. What is comparison based sorting? Explain with example.**

**Ans.** A sorting algorithm is comparison based if the only operation that we can perform on keys is to compare two keys. The comparsion sorts (such as insertion sort, merge sort, heapsort, quicksort) determine the sorted order of an input array by comparing elements.

**2. What is non-comparison based sorting? Give two examples.**

**Ans.** The non-comparison based sorting algorithm sort numbers by means other than comparing two elements.

**Examples:** Counting sort, Radix sort.

**3. What are the worst case, best case and average case running times of Insertion sort ?**

**Ans.** Insertion sort

Worst case	$\Theta(n^2)$
Best case	$\Theta(n)$
Avg. case	$\Theta(n^2)$

**4. What is the basic objective of the divide and conquer approach?**

**Ans.** In divide and conquer approach we break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively (by using any one of the three methods i.e., substitution method, recursion tree method and Master theorem) and then combine these solutions to create a solution to the original subproblem.

**5. Define “Recurrences”. Name and explain its three methods.**

**Ans.** A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Its three methods are

(i) **Substitution method:** In this method we guess a bound and then use mathematical induction to prove our guess is correct.

(ii) **Recursion tree method:** This converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. Then sum up the costs with each level to get level cost and at last sum up all the level costs to get total cost.

(iii) **Master theorem:** Master theorem provide bounds for recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n). \text{ Where } a \geq 1, b > 1 \text{ and } f(n) \text{ is the given function.}$$

**6. Mergesort is a divide-and-conquer algorithm. Justify.**

**Ans.** Divide step: divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  element each.

Conquer step: sort the two subsequences recursively using merge sort.

Combine step: merge the two sorted subsequences to produce the sorted answer.

**7. Provide an algorithm for MERGE-SORT (A, p, r).**

**Ans.** MERGE-SORT (A, p, r)

1. if  $p < r$
2.  $q = \lfloor (p + r)/2 \rfloor$
3. MERGE-SORT (A, p, q)
4. MERGE-SORT (A, q + 1, r)
5. MERGE (A, p, q, r)

**8. Give the recurrence equation for the worst case behaviour of merge sort.**

**Ans.** The recurrence equation for the worst case behaviour of merge sort is

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

In worst case  $T(n) = \theta(n \log n)$ .

**9. Provide an algorithm for multiplication of  $n \times n$  matrix.**

Or

**Provide an algorithm for SQUARE-MATRIX-MULTIPLICATION.**

**Ans.** SQUARE-MATRIX-MULTIPLY (A, B)

1.  $n = \text{rows}$
2. Let C be a new  $n \times n$  matrix
3. for  $i = 1$  to  $n$
4. for  $j = 1$  to  $n$
5.  $C_{ij} = 0$
6. for  $k = 1$  to  $n$
7.  $C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$
8. return C

**10. Give the recurrence relation for SQUARE-MATRIX-MULTIPLICATION algorithm.**

$$\text{Ans. } T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 8T(n/2) + \theta(n^2) & \text{if } n>1 \end{cases}$$

**11. Give the recurrence equation for Strassen's matrix multiplication algorithm.**

$$\text{Ans. } T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 7T(n/2) + \theta(n^2) & \text{if } n>1 \end{cases}$$

**12. Justify that Strassen's matrix multiplication algorithm is asymptotically better than SQUARE-MATRIX-MULTIPLICATION algorithm.**

**Ans.** Strassen's algorithm runs in  $\theta(n^{\log 7}) = \theta(n^{2.81})$  time whereas SQUARE-MATRIX-MULTIPLICATION algorithm takes  $\theta(n^3)$ . As  $\theta(n^{2.81}) < \theta(n^3)$  so Strassen's algorithm is asymptotically better than SQUARE-MATRIX-MULTIPLICATION algorithm.

**13. Quicksort is a simple divide and conquer algorithm. Justify.**

**Ans.** Divide step: pick the pivot element and PARTITION (A, p, r).



Conquer step: recursively sort two subarrays.

Combine step: do nothing, quicksort is an in place algorithm.

**14. What are the advantages of quicksort over mergesort ?**

- Ans.** (i) Quicksort is likely to run a bit faster than mergesort (perhaps 1.2 to 1.4 times as fast).
- (ii) Although quicksort performs 39% more comparisons than mergesort but it requires much less movement (copying) of array elements.
- (iii) Quicksort requires less memory (space) than mergesort.
- (iv) A good implementation of quicksort is probably easier to code than a good implementation of mergesort.

**15. Explain why we expect the average case for mergesort to be almost the same as the worst case?**

- Ans.** Merge sort is based on divide and conquer algorithm. Each divide step yields two subsequences of size  $(n/2)$ . For division it will take  $O(n)$  time that equals to  $\theta(1)$ , for combination of  $n$  elements it will take  $\theta(n)$  time, so that the recurrence is  $T(n) = 2T(n/2) + \theta(n)$ .

By applying Master theorem,  $a = 2$ ,  $b = 2$ ,  $f(n) = \theta(n)$ ,  $n^{\log_b a} = n^{\log_2 2} = n$

Applying Case – II,  $T(n) = \theta(n \log n)$ .

So in both cases i.e, average case and worst case, the set is divided into  $n/2$  subsets. Hence the time complexity of average and worst case is almost same.

**16. Which algorithm takes more space? (a) Mergesort (b) Quicksort. Justify your answer.**

- Ans.** Merge sort takes more space than quicksort; as in case of merge sort an auxiliary array is used to store the result.

**17. Give the recurrence relation for the worst case complexity of quicksort. Determine its time complexity.**

- Ans.** The recurrence relation for the worst case complexity of quicksort is

$$T(n) = T(n - 1) + T(0) + \theta(n) = T(n - 1) + \theta(n)$$

By applying substitution method, we can get the worst case time complexity as  $T(n) = \theta(n^2)$ .

**18. Give the recurrence relation for the best case complexity of quicksort. Determine its time complexity.**

- Ans.** The recurrence relation for the best case complexity of quicksort is :

$$T(n) = 2T(n/2) + \theta(n)$$

By applying Master theorem case – II, we can get the best case time complexity as  $T(n) = \theta(n \log n)$ .

**19. What is the running time of Quick sort when all elements of array A have the same value?**

- Ans.** Since all elements of array A have the same value, subroutine PARTITION will always split into a subproblem of size  $n - 1$  and a subproblem of size 0. This is the worst case. So we expect that the running time is  $\theta(n^2)$ .

**20. What is the running time of Quicksort when the array A contains distinct elements and is sorted in decreasing order? Justify why ?**

**Ans.** The running time of quick sort is  $\theta(n^2)$  when the array A contains distinct elements and is sorted in decreasing order.

Since array A is in decreasing order and all elements are distinct, the pivot will be the smallest element in the subroutine PARTITION. It will always split into a subproblem of size  $n - 1$  and a subproblem of size 0. This is the worst case. So expected running time is  $\theta(n^2)$ .

**21. How to modify quicksort to get good average case behaviour on all inputs ?**

**Ans.** By using “Randomization” we can get good average case behaviour of quicksort on all inputs.

⇒ Randomly permute input.

⇒ Choose partitioning element  $e$  randomly at each iteration.

**22. Write the running time of Binary search and Quick sort.**

**Ans.** Running time of Binary search =  $\theta(\log n)$  and Running time of Quick sort =  $\theta(n \log n)$ . (where,  $n$  is the number of elements in the given array).

**23. Define order statistic.**

**Ans.** The  $i$ th order statistics of a set of  $n$  - element is the  $i$ th smallest element.

**24. What are the input and output of the selection problem?**

**Ans.** **Input:** A set of  $n$  distinct numbers and an integer  $i$ , with  $1 \leq i \leq n$ .

**Output:** The element  $x \in A$  that is larger than exactly  $i - 1$  other element of A.

**25. Write an algorithm for finding minimum of a set of n elements.**

**Ans.** MINIMUM (A)

1.  $\min = A[1]$
2. for  $j = 2$  to  $A$ .
3. if  $\min > A[i]$
4.  $\min = A[i]$
5. return  $\min$ .

**26. Define Master theorem.**

**Ans.** If the recurrence is of the form  $T(n) = aT(n/b) + f(n)$ ; Where  $a \geq 1$  and  $b > 1$ ,

We can apply Master theorem to find the running time. It has three cases.

**Case - I :** If  $f(n) = O(n^{\log_b a - \varepsilon})$  where  $\varepsilon > 0$

Then  $T(n) = \Theta(n^{\log_b a})$

**Case – II :** If  $f(n) = \theta(n^{\log_b a} \cdot \log^k n)$  where  $k \geq 0$

Then  $T(n) = \theta(n^{\log_b a} \cdot \log^{k+1} n)$

**Case – III :** If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$

Then  $T(n) = \Theta(f(n))$

### 27. State the conditions when Master theorem is not satisfied?

**Ans.** There are three conditions when Master theorem can not be satisfied. These are also called as limitations of Master theorem. They are

**(i) Gap between Case – I and Case-II :**  $f(n)$  is smaller than  $n^{\log_b a}$  but not polynomially smaller.

**(ii) Gap between Case – II and Case-III :**  $f(n)$  is larger than  $n^{\log_b a}$  but not polynomially larger.

**(iii) Regularity fails in Case – III :** if the regularity fails to hold i.e.,  $af(n/b) < f(n)$  for some  $c < 1$ .

## CHAPTER – 10 and 11

### Heapsort and Priority Queue

#### 1. How many types of Heaps are there? What are they?

**Ans.** There are two types of Heaps present. They are: (i) Max-heap and (ii) Min-heap.

#### 2. Write the heap-property.

**Ans.** For max-heap, we have max-heap property which says that the priority of every node is greater than or equal to the priority of any child nodes of that node.

For min-heap, we have min-heap property, which says that the priority of every node is less than or equal to the priority of any child nodes of that node.

#### 3. What are the minimum and maximum number of elements in a heap of height h? Justify your answers.

**Ans.** A heap of height  $h$  has the minimum number of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary tree of height  $h - 1$  and  $2^h - 1$  nodes. Hence the minimum number of nodes possible in a heap of height  $h = 2^h - 1 + 1 = 2^h$ .

Similrly, a heap of height  $h$  has the maximum number of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height  $h$  and hence it has  $2^{h+1} - 1$  maximum numbers of nodes possible.

**4. Give a recurrence relation for MAX-HEAPIFY procedure. What is the running time?**

**Ans.** The recurrence relation for MAX-HEAPIFY procedure is  $T(n) \leq T(2n/3) + \theta(1)$

The children's subtrees each have size at most  $(2n/3)$

Running time of MAX-HEAPIFY procedure =  $O(\log n) = O(h)$

$(h \rightarrow \text{a node of height } h \text{ in the heap}).$

**5. Write the algorithm for BUILD-MAX-HEAP.**

**Ans.** BUILD-MAX-HEAP(A)

1. A. heapsize = A. length
2. for  $i = \lfloor A. \text{length}/2 \rfloor$  downto 1
3. MAX-HEAPIFY (A, i)

**6. What is the height of n-element heap? What is the maximum number of nodes present in this heap of height  $h$ ?**

**Ans.**  $n$ -element heap has height  $\lfloor \log n \rfloor$ .

There are at most  $\left\lfloor \frac{n}{2^{h+1}} \right\rfloor$  number of nodes present in  $n$ -element heap of height  $h$ .

**7. Find the running time of BUILD-MAX-HEAP procedure.**

$$\text{Ans. } \sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h\right)$$

$$\text{Again } \sum_{h=0}^{\infty} h/2^h \leq 2$$

Thus we can find the running time of BUILD-MAX-HEAP as

$$O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} h/2^h\right) = O\left(n \sum_{h=0}^{\infty} h/2^h\right) = O(n)$$

**8. Write down the HEAPSORT algorithm.**

**Ans.** HEAPSORT (A)

1. BUILD-MAX-HEAP (A)
2.  $n = \text{Heap-size (A)}$
3. For  $i = n$  down to 2
4. do interchange ( $A[i], A[1]$ )
5.  $\text{Heap-size (A)} = \text{Heap-size (A)} - 1$
6. Max-Heapify (A, 1)

9. How many comparisons does heapsort do in the worst case on an array with four keys?

**Ans.** Number of comparisons in the worst case = 6.

10. Define Priority Queue. What are the different operations it supports?

**Ans.** Priority queue is a data structure for maintaining a set S of elements, each with an associated key. It supports following operations.

insert (S, x): inserts element x into S.

max (S): returns element of S with largest key.

extract-max (S): removes and returns element of S with largest key.

## CHAPTER – 12

### Dynamic Programming Algorithms

1. How can you develop a dynamic programming algorithm?

**Ans.** We can develop a dynamic programming algorithm by applying below listed four steps.

(i) Characterize the structure of an optimal solution.

(ii) Recursively define the value of optimal solution.

(iii) Compute the value of optimal solution in bottom-up fashion.

(iv) Construct an optimal solution from computed information.

2. Define the basic working principle of dynamic programming algorithm? Where dynamic programming is used?

**Ans.** In dynamic programming we take the smallest subproblem (smallest among all subproblems), memorize it (store it in a table) which is used to calculate the bigger problem iteratively.

Dynamic programming is basically used to solve optimization problems; which has many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

3. How dynamic programming algorithm differs from divide and conquer approach?

**Ans.**

Dynamic Programming	Divide-and-conquer
<ul style="list-style-type: none"> <li>(i) Here we take the smallest among the sub-problems into consideration.</li> <li>(ii) It is an iterative process.</li> <li>(iii) It follows bottom-up approach</li> <li>(iv) It is more efficient than divide and conquer.</li> </ul>	<ul style="list-style-type: none"> <li>(i) To get the result we take all the sub-problems into account.</li> <li>(ii) It is a recursive process.</li> <li>(iii) It follows top-down approach.</li> <li>(iv) It is less efficient than dynamic programming.</li> </ul>

**4. What are the basic elements of dynamic programming algorithm ?**

**Ans.** The basic elements of dynamic programming are:

**(i) Optimal Substructure:** Decompose the problem into smaller (and hopefully simpler) sub-problems.

**(ii) Table-Structure:** Store the computed answers in a table.

**(iii) Bottom-up Computation:** Combine solutions on smaller subproblems to solve large sub-problems.

**5. State the “principle of optimality”.**

**Ans.** The “principle of optimality” states that for the global problem to be solved optimally, each subproblem should be solved optimally.

**6. Define “subsequence” in the context of string.**

**Ans.** Given two sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Z = \langle z_1, z_2, \dots, z_k \rangle$ , we say that  $Z$  is a subsequence of  $X$  if there is a strictly increasing sequence  $\langle i_1, i_2, \dots, i_k \rangle$  of indices of  $X$  such that for all  $i = 1, 2, \dots, k$  we have  $x_{i_j} = z_j$ .

**Example :** Let  $X = \langle A B B C D A A \rangle$  and

$$Z = \langle A B C D \rangle$$

So  $Z$  is a sub-sequence of  $X$ .

**7. What is a common sub-sequence?**

**Ans.** Given two sequences  $X$  and  $Y$ , we say that a sequence  $Z$  is a common subsequence of  $X$  and  $Y$  if  $Z$  is a subsequence of both  $X$  and  $Y$ .

**Example :** If  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$ . The sequence  $\langle B, C, A, \rangle$  is a common sub-sequence of both  $X$  and  $Y$ .

**8. What is longest common sub-sequence?**

**Ans.** Given two sequences  $X$  and  $Y$ , we say the longest common sub-sequence of  $X$  and  $Y$  is longest sequence  $Z$  that is a sub-sequences of both  $X$  and  $Y$ .

**Example :** If  $X = \langle A, B, C, B, D, A, B \rangle$  and  $Y = \langle B, D, C, A, B, A \rangle$  Then the longest common sub-sequence is  $Z = \langle B, C, A, B \rangle$ .

**9. Write the optimal sub-structure of the LCS problem.**

Or

**Give a recursive solution to the LCS problem.**

**Ans.** Let  $c[i, j]$  be the length of an LCS of the sequence  $X_i$  and  $Y_j$ . Then we have the following optimal substructure formulation of the LCS problem.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ i.e., either sequence is empty.} \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \text{ i.e., last character match.} \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \text{ i.e., last character don't match.} \end{cases}$$

**10. What is the basic objective of chain matrix multiplication problem?**

**Ans.** Given a sequence of matrices  $A_1, A_2, \dots, A_n$  and the dimensions  $p_0, p_1, \dots, p_n$  where  $A_i$  is of dimension  $p_{i-1} \times p_i$ , determine the order of multiplication that minimizes the number of operations. So this algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

**11. Explain the need of parent-hesitation in chain matrix multiplication.**

**Ans.** If we have just one or two matrices, then there is only one way to parenthesize. If we have  $n$  items, then there are  $n - 1$  places where we could break the list with the outermost pair of parentheses, namely just after the 1st item, just after the 2nd item, etc. and just after the  $(n - 1)$ st item. When we split after the  $k$ th item, we create two sub-lists to be parenthesized, one with  $k$  items, and the other with  $n - k$  items. By doing this we evaluate the order in which matrix to multiply.

**12. Give a recursive solution for chain matrix multiplication.**

**Ans.** For  $1 \leq i \leq j \leq n$ , let  $m[i, j]$  denote the minimum number of multiplication needed to compute  $A_{i, \dots, j}$ . The optimum cost can be described by the following recursive formula:

$$m[i, j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1}p_kp_j) & \text{if } i < j \end{cases}$$

## CHAPTER – 13

### Greedy Algorithms

#### 1. How the Greedy paradigm of algorithm differs from Dynamic programming?

**Ans.**

Greedy Algorithm	Dynamic Programming
<ul style="list-style-type: none"> <li>(i) Greedy algorithm is applicable to a problem in which set of feasible solutions are there. We have to choose the optimal solution from that feasible solutions.</li> <li>(ii) Greedy algorithm is not as powerful or as widely applicable as dynamic programming but it typically generates simple and faster algorithm.</li> <li>(iii) Solution is computed in “top down” manner.</li> <li>(iv) In greedy method only one decision sequence is generated.</li> </ul>	<ul style="list-style-type: none"> <li>(i) A problem is divided into the number of sub-problems and each sub-problem is depend on next sub-problem in a hierarchical way.</li> <li>(ii) Dynamic programming is a powerful technique, but it often leads to algorithm with higher than desired running times.</li> <li>(iii) Optimal solution is described in a recursive, manner, and then computed in “bottom up” fashion.</li> <li>(iv) In dynamic programming many decision sequence generate. However sequence containing sub-optimal sequences can not be optimal.</li> </ul>

**2. What is the feasible solution in activity selection problem/activity scheduling?**

- Ans.** The feasible solution in activity selection problem is that we need to check whether the solution satisfies all possible problem constraints or not. It means a set of activities  $i$  and  $j$  are feasible if the half open intervals  $[s_i, f_i)$  and  $[s_j, f_j)$  do not overlap, if  $s_i \geq f_j$  and  $s_j \geq f_i$ .

**3. What is the basic objective of activity selection problem?**

- Ans.** An activity selection is the problem of scheduling a resource among several competing activities. Here a set of  $n$  activities has been given and the name of the set is  $S = \{1, 2, \dots, n\}$ .

Each activity has a start time  $s_i$  and ends at a given finish time  $f_i$ . We have to find the maximum size of mutually compatible activities.

**4. State the greedy choice property.**

- Ans.** According to greedy choice property we can assemble a globally optimal solution by making locally optimal choices or we may say that when we are considering which choice to make, we make the choice that looks best in the current problem, without considering the results from sub-problem.

**5. Define optimal sub-structure.**

- Ans.** A problem exhibits optimal sub-structure if an optimal solution to the problem contains within it optimal solutions to the sub-problems. This property is a key ingredient of dynamic programming as well as greedy algorithms.

**6. Define Fractional Knapsack Problem. What is the time complexity of the algorithm to solve Fractional Knapsack Problem using greedy algorithm.**

- Ans.** The classical Fractional Knapsack Problem is an optimization problem. Here the thief is allowed to take any fraction of an item for a fraction of the weight and a fraction of the value. The running time of Fractional Knapsack Problem using greedy algorithm is  $O(n \log n)$ ,

where  $n \rightarrow$  number of items.

$\log n \rightarrow$  we use a heap based priority queue to store the items.

**7. Define 0 – 1 Knapsack problem. What is the time complexity of the algorithm to solve 0 – 1 Knapsack problem by dynamic programming solution?**

- Ans.** The classical 0 – 1 Knapsack problem is a famous optimization problem. A thief is robbing a store, and finds  $n$  items which can be taken. The  $i$ th item is worth  $v_i$  dollars and weights  $w_i$  pounds, where  $v_i$  and  $w_i$  are integers. He wants to take as valuable a load as possible, but has a knapsack that can only carry  $W$  total pounds. So he is not allowed to take a fraction of an item or multiple copies of an item.

The time complexity of the algorithm for 0 - 1 knapsack problem by dynamic programming solution is  $O(n W)$  where  $n$  is the number of items and  $W$  is the maximum weight of items that the thief can put in his knapsack.

**8. How Huffman code differs from ASCII code?**

**Ans.**

Huffman Code	ASCII Code
<ul style="list-style-type: none"> <li>(i) It is an efficient method of encoding data and widely used.</li> <li>(ii) It is represented by variable length code word.</li> </ul>	<ul style="list-style-type: none"> <li>(i) It is an inefficient standard for encoding data and not widely used.</li> <li>(ii) It is represented by fixed length codeword which is inefficient from the perspective of minimizing the total quantity of data.</li> </ul>

**9. How variable length code is better than fixed length code?**

**Ans.** In fixed length code each character is represented by a fixed length codeword of bits (like 8 bits per character in case of ASCII). Fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

In case of variable length code frequently occurring characters are encoded using fewer bits and less frequent characters are encoded using more bits, if we have the knowledge of relative probabilities of characters in advance. It can save 25% in expected coding length.

**10. What is prefix code?**

**Ans.** Prefix code is an assignment of codewords to characters so that no codeword is a prefix of any other.

**11. What is expected encoding length ? How can we calculate it?**

**Ans.** After knowing the probabilities of the various characters, we can determine the total length of the encoded text.

The expected number of bits needed to encode a text with  $n$  characters is given by

$$B(T) = n \sum_{x \in C} p(x) d_T(x)$$

Where  $p(x) \rightarrow$  probability of seeing character  $x$ .

$d_T(x) \rightarrow$  length of the codeword relative to some prefix tree  $T$ .

**12. What is Optimal Code Generation? Which algorithm is used for finding this code?**

**Ans.** **Optimal Code Generation:** Given an alphabet  $C$  and the probabilities  $p(x)$  of occurrence for each character  $x \in C$ , compute a prefix code  $T$  that minimizes the expected length of the encoded bit-string,  $B(T)$ .

For finding this code we use Huffman's algorithm.

**13. What is Huffman code? What is the running time of Huffman's algorithm?**

**Ans.** **Huffman code:** Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code. The running time of Huffman's algorithm is  $O(n \log n)$  where  $n$  is the number of characters.

## CHAPTER - 14

### Elementary Graph Algorithms

**1. What are the different applications of graph?**

**Ans.** Different applications of graph includes communication and transportation networks, VLSI and other sorts of logic circuits, CAD and GIS systems, scheduling systems.

**2. Differentiate between directed graph and undirected graph.**

**Ans.** **Directed graph :** A directed graph  $G = (V, E)$  consists of a finite set  $V$ , called vertices of nodes and  $E$ , a set of ordered pairs, called the edges of  $G$ .

**Undirected graph :** An undirected graph  $G = (V, E)$  consists of a finite set  $V$  of vertices, and set  $E$  of unordered pairs of distinct vertices, called the edges.

**3. Differentiate between path and cycle?**

**Ans.** **Path :** A path in a graph or digraph is a sequence of vertices  $(v_0, v_1, v_2, \dots, v_k)$  such that  $(v_{i-1}, v_i)$  is an edge for  $i = 1, 2, \dots, k$ . The length of the path is the number of edges.

**Cycle :** A cycle is a path containing at least one edge and for which  $v_0 = v_k$

**4. Define sparse graph and dense graph?**

**Ans.** **Sparse graph :** A graph is said to be sparse graph if  $|E| \ll |V^2|$

**Dense graph :** A graph is said to be dense if  $|E| \approx |V^2|$ .

**5. Define common ways for representing graphs.**

**Ans.** There are two common ways for representing graph

**(i) Adjacency matrix :** An  $n \times n$  matrix defined for  $1 \leq v, w \leq n$ .

$$A[v, w] = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases}$$

**(ii) Adjacency list :** An array  $\text{Adj}[1, \dots, n]$  of pointers where for  $1 \leq v \leq n$ ,  $\text{Adj}[v]$  points to a linked list containing the vertices which are adjacent to  $v$ . If the edges have weights then these weights may also be stored in the linked list elements.

**6. What are the storage spaces required in adjacency matrix and adjacency list representations?**

**Ans.** An adjacency matrix requires  $\Theta(v^2)$  storage whereas adjacency list require  $\Theta(V + E)$  storage.

**7. What are the different parameters used during BFS traversal of a graph?**

**Ans.** The different parameters that we are using during the BFS traversal of a graph given as follows:

- (i) Colour of the vertex
  - White (Undiscovered)
  - Gray (Discovered)
  - Black (Discovered completely with all its adjacent nodes)

(ii) Distance “d”

(iii) Predecessor  $\pi$ (parent vertex)

**8. Name two popular techniques that are used during the traversal of a graph?**

**Ans.** Two popular techniques are: (i) Breadth-first-search (BFS) and (ii) Depth-first-search (DFS).

**9. What is the basic principle of BFS?**

**Ans.** The name breadth-first-search reflects the fact that BFS attempts to spread out from the source vertex as much as possible. After the source vertex is visited, all its neighbours (adjacent nodes) are visited one by one.

**10. What are the different edges that we found during BFS traversal?**

**Ans.** We mainly get two types of edges. They are: (i) Tree edge (ii) Cross edge.

**11. What is the running time of BFS procedure?**

**Ans.**  $\theta(V + E)$ , Where V → number of vertices. E → number of edges.

**12. Define shortest-path-distance.**

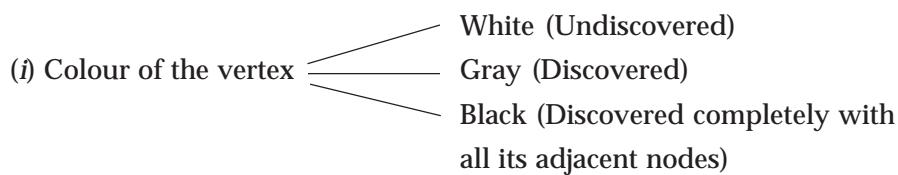
**Ans.** The shortest-path-distance from vertex  $u$  to  $v$  is  $\delta(u, v)$ ; defined as the minimum number of edges in any path from vertex  $u$  to vertex  $v$ ; if there is no path from  $u$  to  $v$ , then  $\delta(u, v) = \infty$ .

**13. What is the basic principle of DFS?**

**Ans.** The name depth-first-search reflects the fact that it is used to search deeper in the graph whenever possible. The edges are explored out of most recently discovered vertex  $V$  that still has unexplored edges leaving it. When all the  $V$ 's edges have been explored, the search backtracks to explore edge leaving the vertex from which  $V$  was discovered. This process continues till we reach the source.

**14. What are the different parameters used in case of DFS traversal?**

**Ans.** Different parameters that are used



(ii) Starting time stamp (represented as  $d$ )

(iii) Finishing time stamp (represented as  $f$ )

(iv) Predecessor (parent vertex  $\pi$ )

**15. What are the different edges that we found in DFS forest?**

**Ans.** We mainly get four types of edges in DFS forest. They are

(i) Tree edges (ii) Back edges (iii) Forward edges (iv) Cross edges.

**16. Name two applications of DFS.**

**Ans.** Applications of DFS are: (i) Topological sort and (ii) Strongly connected components.

**17. Define topological sort.**

**Ans.** One of the applications of DFS is topological sort. It is only applicable on “directed acyclic graph” or “DAG”. A topological sort of a dag  $G = (V, E)$  is a linear ordering of all its vertices such that if  $G$  contains an edge  $(u, v)$  then  $u$  appears before  $v$  in the ordering. In topological sort we arrange the elements along a horizontal line so that all directed edges go from left to right, i.e., we arrange in descending order of finishing time stamp. The practical implementation of topological sort in day to day of life is we put our shirts, paints, socks before i.e., everything put in an order.

**18. Write the algorithm for TOPOLOGICAL-SORT( $G$ ). What is its running time?**

**Ans.** TOPOLOGICAL-SORT( $G$ )

1. Call DFS ( $G$ ) to compute finishing times  $v.f$  for each vertex  $v$ .
2. as each vertex is finished, insert it onto the front of a linked list.
3. return the linked list vertices.

Running time =  $\theta(V + E)$

**19. Define the strongly connected component.**

**Ans.** Strongly connected components is another application of DFS. In this technique we decompose a directed graph into its strongly connected components. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

A strongly connected component of a directed graph  $G = (V, E)$  is a maximal set of vertices  $C \subseteq V$  such that for every pair of vertices  $u$  and  $v$  in  $C$ , we have both  $u \leadsto v$  and  $v \leadsto u$ ; i.e., vertices  $u$  and  $v$  are reachable from each other.

**20. Write the algorithm for finding strongly connected component graph  $G$  and state running time.**

**Ans.** STRONGLY-CONNECTED-COMPONENT( $G$ )

1. Call DFS ( $G$ ) to compute finishing times  $u.f$  for each vertex  $u$
2. Calculate  $G^T$
3. Call DFS ( $G^T$ ), but in the main loop of DFS, consider the vertices in order of decreasing  $u.f$
4. Output the vertices of each tree in the depth-first-forest formed in line 3 as a separate strongly connected component.

Running time =  $\theta(V + E)$

- 21. What is a spanning tree? What is the number of spanning tree possible for a given graph G?**

**Ans.** **Spanning tree:** The graph  $G = (V, E)$  is traversed in such a way that it forms an acyclic graph which is a tree consisting of all the vertices of graph G but not edges; such a tree is called a spanning tree. For a given graph G there are many number of spanning trees are possible.

- 22. Define minimum spanning tree (MST). For a given graph G how many MST you can generate?**

**Ans.** **Minimum spanning tree:** A MST of an edge weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree. For a given graph G. We can generate only one MST.

- 23. Name two methods for finding MST of a given graph G.**

**Ans.** Two methods are (i) Kruskal's algorithm and (ii) Prim's algorithm.

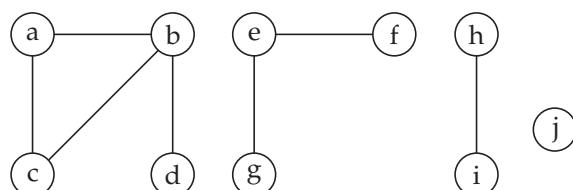
- 24. How Kruskal's algorithm works for finding MST? What is its running time?**

**Ans.** Kruskal's algorithm finds a MST for a connected weighted graph. This means that it finds a subset of edges that forms a tree that includes every vertex, where the total weight of all the edges in a tree is minimized. If the graph is not connected it finds a MST forest. Kruskal's algorithm is a suitable example of Greedy algorithm. Running time of the Kruskal's algorithm is  $O(E \log V)$ ; where  $E \rightarrow$  number of edges and  $V \rightarrow$  number of vertices.

- 25. In Kruskal's method for finding the minimum spanning tree, how does the algorithm know, when the addition of an edge will generate a cycle?**

**Ans.** In Kruskal's algorithm if the vertices of the edge are belong to some set then that edge is not safe, that's why we can not include that edge in the spanning tree. If  $(u, v) \in E$  and if  $\text{FINDSET}(u) = \text{FINDSET}(v)$  then that edge can not be included.

- 26. Determine the collection of disjoint sets for the following graph using the functions MAKESET, UNION, FINDSET**



**Ans.**

Edge processed	Collection of disjoint sets			
initial sets	{a} {b} {c} {d}	{e} {f} {g}	{h} {i}	{j}
(a, b)	{a, b} {c} {d}	{e} {f} {g}	{h} {i}	{j}
(a, c)	{a, b, c} {d}	{e} {f} {g}	{h} {i}	{j}
(b, d)	{a, b, c, d}	{e} {f} {g}	{h} {i}	{j}
(b, c)	{a, b, c, d}	{e} {f} {g}	{h} {i}	{j}
(e, f)	{a, b, c, d}	{e, f} {g}	{h} {i}	{j}
(e, g)	{a, b, c, d}	{e, f, g}	{h} {i}	{j}
(h, i)	{a, b, c, d}	{e, f, g}	{h, i}	{j}

So, the sets {a, b, c, d}, {e, f, g}, {h, i}, {j} are the collection of disjoint sets of the given graphs.

**27. What is the basic objective of Prim's algorithm? What is its time complexity?**

**Ans.** Prim's algorithm is a Greedy algorithm which finds the minimum spanning tree of a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized . The time complexity of the Prim's algorithm is same as that of Kruskal's algorithm i.e., O(E log V)

where E → the number of edges and V → the number of vertices

**28. What are the disjoint set data structures used in Kruskal's algorithm?**

- Ans.** (i) MAKESET (u) : It creates a new set whose only member is pointed to u.  
(ii) FINDSET (u) : It returns a pointer to the set containing u.  
(iii) UNION (u, v) : It unites the sets that contain u and v.

**29. What is the basic difference between Kruskal's and Prime's algorithm?**

**Ans.** Kruskal's algorithm works on edges and Prim's algorithm works on vertices in determining the minimum sanning tree.

**30. What is a single sources shortest path problem? Name two methods for solving shortest path problems.**

**Ans.** In a single source shortest path problem, we are given a graph  $G = (V, E)$ . We want to find a shortest path from a given source vertex  $\in V$  to each vertex  $u \in V$  . For solving this kind of problem we use either Dijksta's algorithm or Bellmanford algorithm.

**31. What is the need of “0-weight cycle” in shortest path problem?**

**Ans.** If there is a shortest path from a source vertex  $s$  to a destination vertex  $v$  that contains a 0-weight cycle, then there is another shortest path from  $s$  to  $v$  without this cycle. As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free. So the shortest paths having no cycles are simple paths.

**32. Can a shortest path contain a cycle? Justify your answer.**

**Ans.** A shortest path cannot contain a negative-weight cycle nor can it contain a positive weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.

**33. Define shortest path weight.**

**Ans.** The shortest path weight from vertex  $u$  to  $v$  is given as  $\delta(u, v)$  and defined as

$$\delta(u, v) = \begin{cases} \min \{w(p) : u \xrightarrow{p} v\} & \text{if there is a path from } u \text{ to } v \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

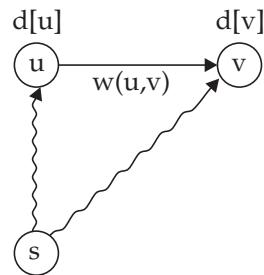
**34. What do you mean by relaxation, while designing an algorithm for shortest path? Write its procedure and time complexity.**

**Ans.** Relaxation is a technique of determining the shortest path of any vertex  $v$  from some vertex  $u$  inspite of the presence of many paths between  $u$  and  $v$ .

**RELAXATION PROCEDURE ( $u, v, w$ )**

1. if  $d[v] > d[u] + w(u, v)$
2. then  $d[v] = d[u] + w(u, v)$
3.  $\pi[v] = u$

Time complexity =  $O(1)$ .



**35. State any four properties of shortest paths and relaxation.**

**Ans.** The properties are (i) Triangle inequality, (ii) Upper-bound property (iii) Convergence property (iv) No path property (v) Path relaxation property (vi) Predecessor subgraph property.

**36. What is the basic objective of Dijkstra's algorithm in shortest path problem?**

**Ans.** Dijkstra's algorithm is a greedy algorithm which solves the single-source shortest path problem when all edges have non-negative weights. Algorithm starts at the source vertex and grows a tree that ultimately spans all vertices reachable from source. Vertices are added to the tree in order of distance; first start vertex, then the vertex close to it, then the next closest and so on.

**37. What is the basic working principle of Bellman-Ford algorithm?**

**Ans.** Bellman-Ford algorithm solves the single source shortest path problem in which edges of a given digraph can have negative weights, but no negative cost cycles. This algorithm is based on performing repeated relaxations. So Bellman-Ford algorithm simply applies a relaxation to every edge in the graph, and repeats this  $V - 1$  times.

**38. What are the running times of Dijksta's algorithm and Bellman-Ford algorithm?**

**Ans.** Dijksta's algorithm –  $O(E \log V)$ , Bellman-Ford algorithm –  $O(VE)$

- 39. What is the basic objective of all-pairs shortest path problem? How to solve this kind of problem?**

**Ans.** The basic objective of all-pairs shortest path problem is to find a shortest path from  $u$  to  $v$  for every pair of vertices  $u$  and  $v$ . We can solve this problem by using Floyd-Warshall algorithm.

- 40. Floyed-Warshall algorithm is based on \_\_\_\_ approach. What is its running time? How much space the algorithm use?**

**Ans.** Floyd-Warshall algorithm is based on dynamic programming approach. Time complexity of the algorithm is  $\Theta(n^3)$ . The space used by the algorithm is  $\Theta(n^2)$ .

- 41. How dynamic programming formulation used in Floyd-Warshall algorithm?**

**Ans.**  $d_{ij}^k$  = The shortest path from  $i$  to  $j$  with  $k$  number of intermediate vertex.

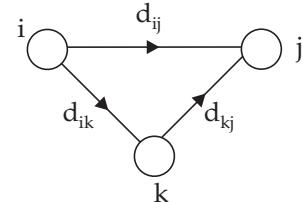
$k$  can be 0, 1, 2, ....,  $n$ .

$$d_{ij}^{(0)} = W_{ij}$$

$$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \text{ (for } k \geq 1\text{)}$$

- 42. What is a shortest path?**

**Ans.** When every edge of a directed path  $G$  has a non-negative weight attached, then a path from one vertex  $v$  to another  $w$  such that sum of weights on the path is as small as possible; such a path is called shortest path.



## CHAPTER – 15

### Backtracking

- 1. Define backtracking. Give two examples about its application.**

**Ans.** Backtracking is a recursive algorithm strategy for finding all solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate (*i.e.*, backtracks) as soon as it determines that it cannot possibly be completed to a valid solution. This technique is used to solve puzzle problem, queen problem, graph coloring problem etc.

- 2. Explain N Queens problem.**

**Ans.** The typical N Queens problem is the problem of placing N queens on an  $N \times N$  chessboard, so that no two queens attack each other by being on the same row or same column or on same diagonal.

- 3. What is state-space tree?**

**Ans.** State space tree (or recursion tree) is the best possible way to implement backtracking by constructing a tree of choices being made. The root represents an initial state before the search for the solution begins; whereas edges in the tree corresponds to recursive calls.

**4. Explain subset-sum problem.**

**Ans.** The subset-sum problem can be explained as follows: Given a set S of positive integers  $S = \{S_1, S_2, \dots, S_n\}$  and target integer T, is there a subset of elements in S that add upto T?

**5. Explain graph coloring problem?**

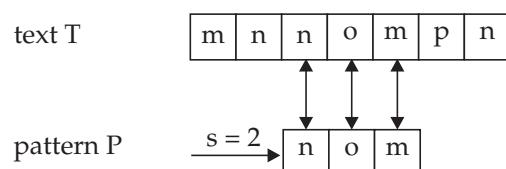
**Ans.** The graph coloring problem asks us to assign the smallest number of colors to vertices of a graph so that no two adjacent vertices are the same color.

## CHAPTER – 16

### String Matching

**1. What is the basic objective of string matching problem?**

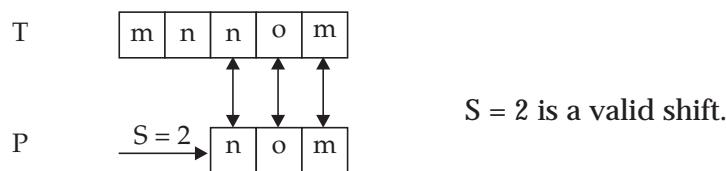
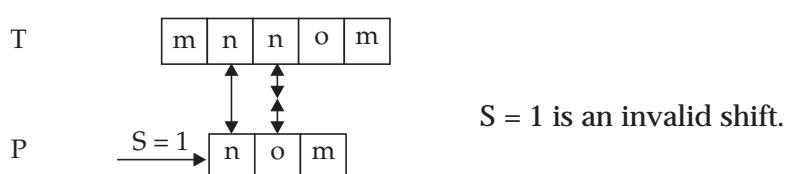
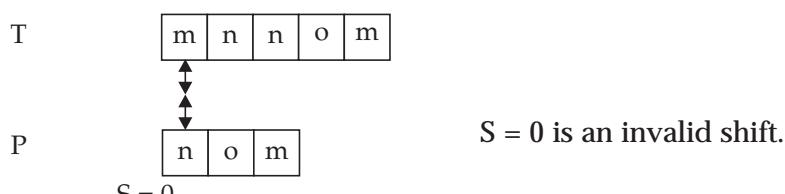
**Ans.** The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.



**2. Define “valid shift” and “invalid shift” in the context of string matching with an example.**

**Ans.** In the string matching problem if the pattern P occurs with shift S in text T, then S is the valid shift; otherwise S is an invalid shift.

**Example**



3. Name any two efficient string matching algorithms. Also write its corresponding matching time and processing time.

Ans.	Efficient string-matching Algorithm	Matching time	Processing time
	(i) Rabin-Karp	$O((n - m + 1)m)$	$\theta(m)$
	(ii) Knuth-Morris-Pratt	$\theta(n)$	$\theta(m)$

where  $n$  is the length of given text T,  $m$  is the length of given pattern P.

4. What is the basic objective of Brute-Force String matching algorithm (or Navie string-matching algorithm)? What is the running time of this algorithm?

Ans. The basic objective of Brute-Force string matching algorithm is to compare the pattern to the text, one character at a time, until unmatching characters are found.

The running time (worst case) of the algorithm is  $O((n - m + 1)m)$ , where  $n$  is the length of given text T,  $m$  is length of given pattern P.

5. What is spurious hit.

Ans. If the hash values for the given pattern P and the substring in the given text match but the characters differ then it is a spurious hit.

6. What is the basic objective of Robin-Karp string matching algorithm?

Ans. The Robin-Karp algorithm is a string searching algorithm calculates a hash value for the pattern, and for each  $m$ -character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next  $m$ -character sequence. If the hash values are equal, the algorithm will do a brute-force comparison between the pattern and the  $m$ -character sequence. So there is only one comparison per text subsequence and brute force is needed when hash values match.

## CHAPTER – 17

### NP-Completeness

1. Define and differentiate between NP and NP-Complete problems with examples?

Ans. The class NP: NP stands for non-deterministic polynomial and defined as it is the class of problems that have polynomial time verifies.

To prove a problem is in class NP or not we need to take the help of “PROVER” (all powerful) and “VERIFIER” (Limited resources, no too much time for verification).

In general a decision problem is said to be in class NP if for all YES input there is a proof (advice/string) using which it can verified (quickly) that the input is indeed a YES input.

**Example :** CLIQUE is NP, Hamiltonian cycle (HC) is NP, etc.

#### The class NP-Complete

A problem  $\pi$  is NP-complete if it satisfies two conditions i.e.,  $\pi$  is NP and it is NP-HARD as well.

**Example :** CLIQUE is NP-Complete, SAT is NP-Complete, Hamiltonian Path (HP or HAMPATH) is NP-Complete, etc.

**2. State three properties of NP-Complete problems.**

**Ans.** The three properties of NP-complete problems are

- (i) If problem B is NP-complete and  $B \in P$ , then  $P = NP$
- (ii) If any NP-complete problem is solvable in polynomial time, then every NP-complete problem (and in fact every problem in NP) is also solvable in polynomial time.
- (iii) Cook's theorem tells SAT is NP-complete.

**3. How NP problems differ from NP-Hard problems?**

**Ans.** **NP problem :** NP is the class of languages that have polynomial time verifiers. A decision problem is said to be in NP class if for all YES input there is a proof (advice/string) using which it can be verified (quickly) that the input is indeed a YES input.

To check whether the given problem is in NP class or not we need to take the help of "PROVER" (who is very intelligent) and "VERIFIER" (who has limited resource and his job is to verify quickly in polynomial time).

**NP-Hard Problem :** A problem  $\pi$  is in NP-Hard if it satisfies the below characteristics.

→ If there exists a polynomial time algorithm for  $\pi$  then there is one for all problem in NP.

→ So, SAT is NP-Hard.

We use "Reduction" technique for proving the given problem is in NP-Hard or not.

**4. Why do we need approximation algorithm?**

**Ans.** We need approximation algorithm to find the near-optimal (approximate) solution for NP-Hard problems (like travelling-salesman problem, vertex cover problem etc). Depending on the problem, we may define an optimal solution as one with maximum possible cost or one with minimum possible cost; that is, the problem may be either a maximization or minimization problem.

**5. What is an absolute approximation algorithm?**

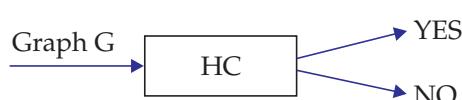
**Ans.** One way to compare the output of an approximation algorithm to the optimal solution is to take the absolute difference between them. If we can bound this difference to a constant for any input, then we have an absolute approximation algorithm.

**6. Differentiate between a decision problem, a search problem and an optimization problem with examples?**

**Ans. Decision problem**

In decision problem the output is only one bit i.e., "YES" or "No" (or more formally "1" or "0").

**Example**



Does G have a HC?

### Search Problem

In search problem the output consists of many bits. Here we search for a solution to output the solution.



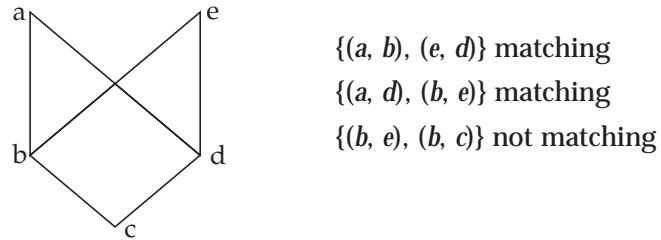
**Optimization problem :** In optimization problem each feasible solution has an associated value and we wish to find a feasible solution with the best value.

**Example :** In Shortest-path problem we are given an undirected graph G and vertices  $u$  and  $v$ , and we wish to find a path from  $u$  to  $v$  that uses a fewest edges.

### 7. Define “Matching” in a graph G.

**Ans.** Matching (M) in a graph  $G = (V, E)$  is a subset M of E such that no two edges in M have the same end points.

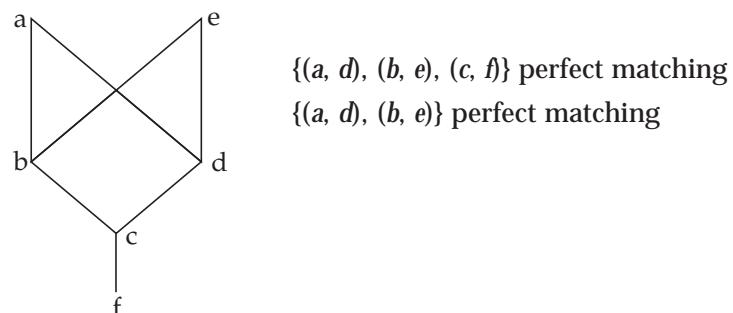
**Example :**



### 8. Define “Perfect Matching” in a graph G.

**Ans.** It is the matching such that all vertices are end points of exactly one edge in M.

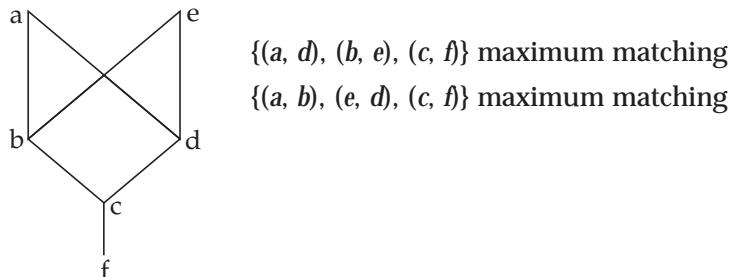
**Example :**



### 9. Define “Maximum Matching” in graph G.

**Ans.** If G has a perfect matching then the size of a maximum matching is  $\frac{|V|}{2}$  where maximum number of matching takes place.

**Example :**



**10. Define the Reduction technique with an example.**

**Ans.** **Reduction :** There exists an efficient algorithm for problem  $\pi_1$  (search/decision) using this we construct (design) an efficient algorithm for problem  $\pi_2$ .

**Example :**

$\pi_1$  : A perfect match for a graph G.



$\pi_2$  : A size of the maximum matching

Reduce the problem  $\pi_1$  into  $\pi_2$ , if  $\pi_1$  is solved then  $\pi_2$  is solved.

Reduction technique is mainly used for proving NP-Hard problems.

**11. Define complexity class with an example.**

**Ans.** A complexity class as a set of languages, membership in which is determined by a complexity measure, such as running time, of an algorithm that determines whether a given string  $x$  belongs to language L.

**Example :** The complexity class P:

$P = \{L \subseteq \{0, 1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}$

**12. Define “verification algorithm” with an example.**

**Ans.** We define a verification algorithm as being a two-argument algorithm A, where one argument is an ordinary input string  $x$  and the other is a binary string  $y$  called a certificate. A two-argument algorithm A verifies an input string  $x$  if there exists a certificate such that  $A(x, y) = 1$ .

The language verified by a verification algorithm A is  $L = \{x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}$

**13. Define the class P. Give an example of a problem in P.**

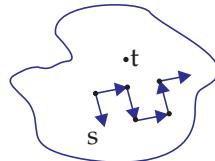
**Ans.** P is the class of languages that are decidable in polynomial time.

In other words,  $P = \sum_k \text{Time}(n^k)$

**Example :** The PATH problem : is there a path from  $s$  to  $t$  ?

$\text{PATH} = \{(G, s, t) \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

Graph  $G$  :



**14. Differentiate P class and NP class with examples.**

**Ans.** P is the class of languages that are **decidable** in polynomial time whereas NP is the class of languages that have polynomial time **verifiers**.

**Example of P class**

Relative prime finding is in P class.

$\text{RELPRIIME} = \{(x, y) \mid x \text{ and } y \text{ are relative prime}\}$

So,  $\text{RELPRIIME} \in P$

**Example of NP class**

CLIQUE is in NP.

**15. State the Cook-Levin Theorem (or Cook's Theorem)**

**Ans.** Cook-Levin Theorem states that “If there exist a polynomial time algorithm for SAT (satisfy-Ability or Satisfiability), there is one for all problem in NP”.

This implies  $\text{SAT} \in P$  iff  $P = NP$ .

**16. What is “Approximation Algorithm” ? Name its two applications.**

**Ans.** Approximation algorithm is an algorithm that runs in polynomial time and produces a solution that is within a guaranteed factor of the optimum solution. Its two applications are travelling salesman problem and greedy vertex cover.

**17. Name any two problems which are in NP-Hard but not in NP. Define those problems.**

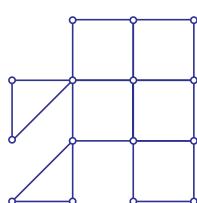
**Ans.** Two problems which are in NP-Hard but not in NP :

(i) QBF (Quantified Boolean Formulas)

In this problem, we are given a boolean formula with quantifiers ( $\exists$  and  $\forall$ ) and we want to know whether the formula is true or false.

(ii) No. Hamiltonian cycle

In this problem, we are given with a graph, we need to show the given graph doesn't consist any hamiltonian cycle.



(No-Hamiltonian cycle)

- 18. Is there any problem which is in NP-class but not in NP-Hard? If yes, then give an example and define that problem.**

**Ans.** Yes. One of the example of a problem which is in NP-class but not in NP-Hard is “Graph isomorphism”. Graph isomorphism problem asks whether two graphs are identical up to a renaming of their vertices.

- 19. What is “Certificate” in NP problems? Explain with an example.**

**Ans.** Certificate is a piece of evidence that allows us to verify in polynomial time that a string is in a given language.

**Example :** Suppose that a language is the set of Hamiltonian graphs. To convince someone that a graph is in this language, we could supply a certificate consisting of a sequence of vertices along the cycle. We could access the adjacency matrix to determine that this is a legitimate cycle in  $G$ . So  $HC \in NP$ .

- 20. For problem P, if we are given an input I and a possible answer A, and we find a way to verify whether or not A really is a valid answer to P given I, then what kind of problem is P?**

**Ans.** We know that, NP is the class of languages that have polynomial time verifiers. So the verification process takes place in NP problem. This implies the problem P is a NP problem.

- 21. What is Hamiltonian path in an undirected graph?**

**Ans.** A Hamiltonian Path (H.P) in a graph  $G$  is a path of length  $n - 1$  where  $n$  is the number of vertices. In other words, it is a path that contains all vertices.

- 22. What is Hamiltonian cycle in an undirected graph ?**

**Ans.** A Hamiltonian cycle (H.C.) in a graph  $G$  is a cycle which spans all vertices.

- 23. What is Clique?**

**Ans.** A Clique in a graph  $G = (V, E)$  is a subset  $U$  of  $V$  such that for all  $u_1, u_2 \in U$ ,  $\{u_1, u_2\} \in E(G)$ .

- 24. What is Independent set in a graph G?**

**Ans.** Given a graph  $G = (V, E)$ , a subset  $U \subseteq V$  is called an independent set if for all  $u_1, u_2 \in U$  and  $\{(u_1, u_2)\} \notin E$

- 25. What is Vertex Cover problem?**

**Ans.** A Vertex cover in a graph  $G$  is a set of vertices such that every edge has at least one end points in  $U$ .

- 26. Why we reduce a problem to another? What is its significance?**

**Ans.** A reduction let us compare the time complexities and underlying structure of two problems. They are useful in providing algorithms for new problems, for giving evidence that there no fast algorithms for certain problems and for classifying problems according to their difficulty.



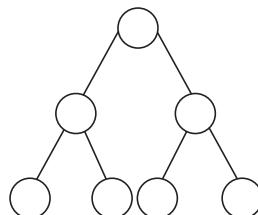
# BRAIN TEASERS FROM GATE (SOLVED)

**CS-GATE-2010**

1. In a binary tree with  $n$  nodes, every node has an odd number of descendants. Every node is considered to be its own descendant. What is the number of nodes in the tree have exactly one child?
 

(a) 0	(b) 1
(c) $(n - 1)/2$	(d) $n - 1$

**Ans.** Given that a node has odd number of descendants possible only when it is fully binary tree like.



→ So in fully binary tree, ever node has two child or zero child (zero child when there is one node exactly).

→ So the number of nodes in the tree that type having exactly one child is zero.

So the answer is option (a).

2. Consider a complete undirected graph with vertex set  $\{0, 1, 2, 3, 4\}$ . Entry  $W_{ij}$  in the matrix  $W$  below is the weight of the edge  $\{ij\}$ .

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

**What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T?**

- |       |        |
|-------|--------|
| (a) 7 | (b) 8  |
| (c) 9 | (d) 10 |

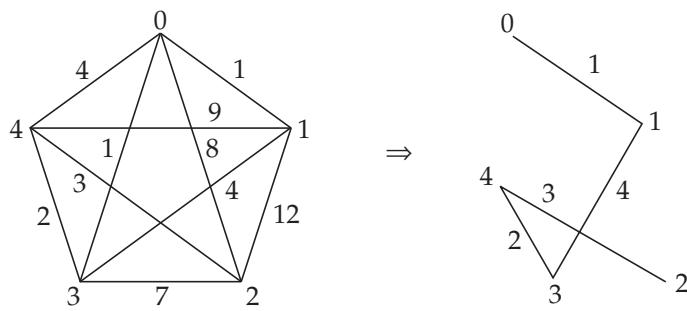
**Ans.**  $V_0 \ V_1 \ V_2 \ V_3 \ V_4$

For the minimum cost we goes to

$$V_0 \xrightarrow{1} V_1 \xrightarrow{4} V_4 \xrightarrow{2} V_3 \xrightarrow{3} V_2$$

Hence the total cost = 10

**Or**



$$\text{Total cost} = 1 + 2 + 3 + 4 = 10$$

So the answer is option (d).

**What is the minimum possible weight of a path P from vertex 1 to 2 in this graph such that P contains at most 3 edges?**

- |       |        |
|-------|--------|
| (a) 7 | (b) 8  |
| (c) 9 | (d) 10 |

**Ans.** From the above figure, move from vertex 1 to vertex 2 can be done in

$$V_1 \xrightarrow{1} V_0 \xrightarrow{4} V_4 \xrightarrow{3} V_2$$

So the total weight = 1 + 4 + 3 = 8 and it contains 3 edges.

Here weight is given in matrix. We use directly such that,  $w_{ij} = V_i \xrightarrow{w} V_j$ .

$$V_1 \ V_0 \rightarrow V_0 \ V_4 \rightarrow V_4 \ V_2$$

So the answer is option (b).

## CS-GATE - 2011

1. An algorithm to find the length of the longest monotonically increasing sequence of numbers in an array  $A[0 : n - 1]$  is given below.

Let  $L_i$  denote the length of the longest monotonically increasing sequence starting at index  $i$  in the array.

**Initialize**  $L_{n-1} = 1$

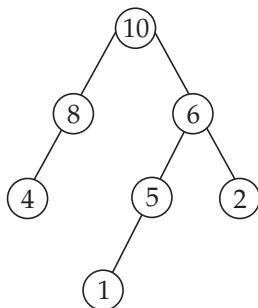
**For all  $i$  such that  $0 \leq i \leq n - 2$**

$$L_i = \begin{cases} 1 + L_{i+1} & \text{if } A[i] < A[i+1] \\ 1 & \text{otherwise} \end{cases}$$

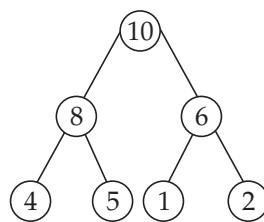
Finally the length of the longest monotonically increasing sequence is  $\text{Max}(L_0, L_1, \dots, L_{n-1})$ . Which of the following statement is TRUE?

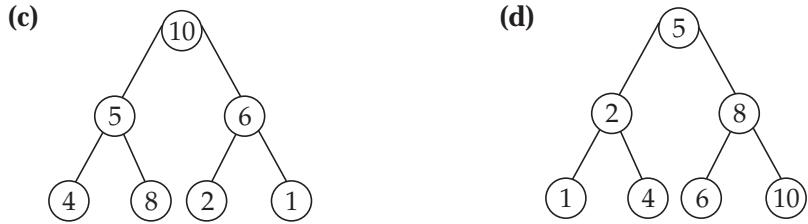
- (a) The algorithm uses dynamic programming paradigm.
  - (b) The algorithm has a linear complexity and uses branch and bound paradigm.
  - (c) The algorithm has a non-linear polynomial complexity and uses branch and bound paradigm.
  - (d) The algorithm uses divide and conquer paradigm.
- Ans.** In longest increasing sub-sequence problem we need to find a sub-sequence of a given sequence in which the subsequence elements are in sorted order, lowest to highest and in which the subsequence is as long as possible. This sub-sequence is not necessarily to be unique. The longest increasing sub-sequence problem is closely related to longest common sub-sequence problem, which has a quadratic time dynamic program solution. So clearly the given algorithm uses dynamic programming paradigm. So answer is option (a).
2. A max-heap is heap where the value of each parent is greater than or equal to the value of its children. Which of the following is a max-heap?

(a)



(b)





- Ans.** Here option (c) and option (d) does not satisfy the max-heap property. Again we also know that heap is a full binary tree; which is violated in option (a).

So answer is option (b).

3. Four matrices  $M_1$ ,  $M_2$ ,  $M_3$  and  $M_4$  of dimensions  $p \times q$ ,  $q \times r$ ,  $r \times s$  and  $s \times t$  respectively can be multiplied in several ways with different number of total scalar multiplications. For example when multiplied as  $((M_1 \times M_2) \times (M_3 \times M_4))$  the total number of scalar multiplications is  $pqr + rst + prt$ . When multiplied as  $((M_1 \times M_2) \times M_3) \times M_4$ ) the total number of scalar multiplications is  $pqr + prs + pst$ .

If  $p = 10$ ,  $q = 100$ ,  $r = 20$ ,  $s = 5$  and  $t = 80$ , then the minimum number of scalar multiplications needed is



- Ans.** Multiply as  $(M_1 \times (M_2 \times M_3)) \times M_4$

Total number of scalar multiplications required =  $qrs + pqs + pst = 10000 + 5000 + 4000 = 19000$

So answer is option (c)

4. Which of the given options provides the increasing order of asymptotic complexity of functions  $f_1$ ,  $f_2$ ,  $f_3$  and  $f_4$ ?

$$f_1(n) = 2^n \qquad \qquad f_2(n) = n^{3/2}$$

$$f_3(n) = n \log_2 n \quad f_4(n) = n^{\log_2 n}$$

- (a)  $f_3, f_2, f_4, f_1$       (b)  $f_3, f_2, f_1, f_4$   
 (c)  $f_2, f_3, f_1, f_4$       (d)  $f_2, f_3, f_4, f_1$

- Ans.**  $n = 1024$

$$f_1(n) = 2^{1024}$$

$$f_0(n) = 2^{15}$$

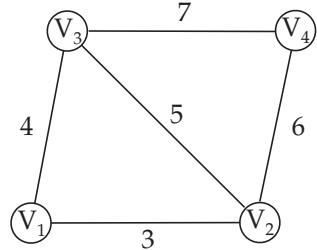
$$f_2(n) = 10 \times 2^{10}$$

$$f_4(n) = 1024^{10} = 2^{100}$$

So,  $f_2, f_2, f_4, f_1$  is required increasing order.

Answer is option (a).

5. An undirected graph  $G(V, E)$  contains  $n$  ( $n > 2$ ) nodes named  $v_1, v_2, \dots, v_n$ . Two nodes  $v_i, v_j$  are connected if and only if  $0 < |i - j| \leq 2$ . Each edge  $(v_i, v_j)$  is assigned with a weight  $i + j$ . A sample graph  $n = 4$  is shown below.



What will be the cost of the Minimum Spanning Tree (MST) of such a graph with  $n$  nodes ?

- (a)  $1/12 (11n^2 - 5n)$       (b)  $n^2 - n + 1$   
 (c)  $6n - 11$       (d)  $2n + 1$

**Ans.** Answer is option (b).

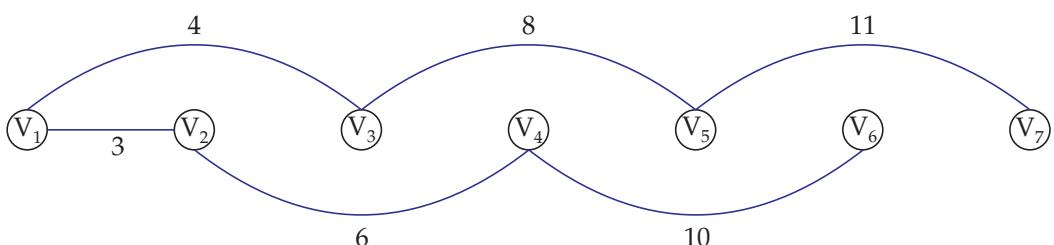
Refer Chapter – 14 solved examples – 14.10.

The length of the path from  $v_5$  to  $v_6$  in the MST of the previous question with  $n = 10$  is

- (a) 11      (b) 25  
 (c) 31      (d) 41

**Ans.**  $V_5V_3 + V_3V_1 + V_1V_2 + V_2V_4 + V_4V_6$   
 $= 8 + 4 + 3 + 6 + 10 = 31$

So answer is option (c).



## CS-GATE – 2012

1. Let  $W(n)$  and  $A(n)$  denote respectively, the worst case and average case running time of an algorithm executed on an input of size  $n$ . Which of the following is ALWAYS TRUE?

- (a)  $A(n) = \Omega(W(n))$       (b)  $A_b(n) = \Theta(W(n))$   
 (c)  $A(n) = O(W(n))$       (d)  $A(n) = o(W(n))$

**Ans.** If for an algorithm  $S$ ;  $X(n)$ ,  $A(n)$  and  $W(n)$  denote best case, average case and worst case running times respectively then we may say  $X(n) = O(A(n))$  and  $A(n) = O(W(n))$   
 So, answer is option (c).

2. The recurrence relation capturing the optimal execution time of the “Towers of Hanoi” problem with  $n$  discs is

- (a)  $T(n) = 2T(n - 2) + 2$       (b)  $T(n) = 2T(n - 1) + n$   
 (c)  $T(n) = 2T(n/2) + 1$       (d)  $T(n) = 2T(n - 1) + 1$

**Ans.** Answer is option (d). Refer example 6.16.

3. The worst case running time to search for an element in a balanced binary search tree with  $n^{2^n}$  elements is

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

**Ans.** The worst case running time to search for an element in a balanced BST =  $\log n$  = Height of the tree and  $n$  denotes the number of elements.

Here  $n$  is  $n \cdot 2^n$  (given)

So the running time will be

$$\begin{aligned} &= \log(n \cdot 2^n) = \log n + \log 2^n \\ &= \log n + n \log 2 = \Theta(n) \end{aligned}$$

Answer is option (c).

4. Assuming  $P \neq NP$ , which of the following is TRUE ?

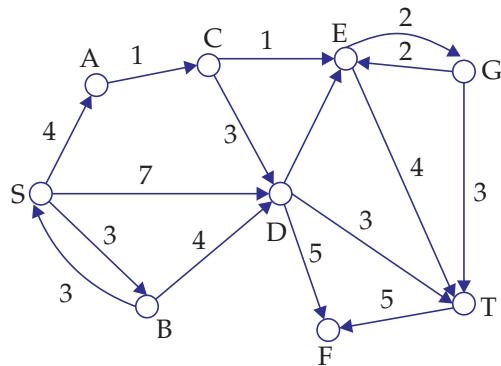
- (a)  $NP\text{-complete} = NP$       (b)  $NP\text{-complete} \cap P = \emptyset$   
 (c)  $NP\text{-hard} = NP$       (d)  $P = NP\text{-complete}$

**Ans.** If  $P \neq NP$

- $\Rightarrow$  No NP-complete problem can be solved in polynomial time.  
 $\Rightarrow$   $P \cap NP\text{-complete} = \emptyset$  (these sets are disjoint).

So answer is option (b).

5. Consider the directed graph shown in figure below. There are multiple shortest paths between vertices  $S$  and  $T$ . Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to vertex  $v$  is updated only when a strictly shorter path to  $v$  is discovered?



(a) SDT

(c) SACDT

(b) SBDT

(d) SACET

**Ans. Step-1:**

Vertex (v)	S	A	B	C	D	E	F	G	T
Distance (d)	0								
Predecessor ( $\pi$ )	NIL								

**Step-2:**

v	S	A	B	C	D	E	F	G	T
d	0	4	3	$\infty$	7	$\infty$	$\infty$	$\infty$	$\infty$
$\pi$	NIL	S	S	NIL	S	NIL	NIL	NIL	NIL

**Step-3:**

v	S	A	B	C	D	E	F	G	T
d	0	4	3	$\infty$	7	$\infty$	$\infty$	$\infty$	$\infty$
$\pi$	NIL	S	S	S	NIL	S	NIL	NIL	NIL

**Step-4:**

v	S	A	B	C	D	E	F	G	T
d	0	4	3	5	7	$\infty$	$\infty$	$\infty$	$\infty$
$\pi$	NIL	S	S	A	S	NIL	NIL	NIL	NIL

**Step-5:**

	1	3	2	4	5	F	G	T
v	S	A	B	C	D	E		
d	0	4	3	5	7	6	$\infty$	$\infty$
$\pi$	NIL	S	S	A	S	C	NIL	NIL

**Step-6:**

	1	3	2	4	5	F	G	T
v	S	A	B	C	D	E		
d	0	4	3	5	7	6	$\infty$	10
$\pi$	NIL	S	S	A	S	C	NIL	E

After Step – 6 we can observe that,

$$\pi[T] = E, \pi[E] = C, \pi[C] = A, \pi[A] = S$$

So the shortest path from S to T is SACET.

Answer is option (d).

6. A list of  $n$  strings, each of length  $n$ , is sorted into lexicographic order using the merge-sort algorithm. The worst case running time of this computation is
- $O(n \log n)$
  - $O(n^2 \log n)$
  - $O(n^2 + \log n)$
  - $O(n^2)$

**Ans.** The height of the recursion tree using merge sort is  $\log n$  and at most  $n^2$  comparisons are done at each level, where at most  $n$  pairs of strings are compared at each level and  $n$  comparisons are required to compare any two strings. So the worst case running time =  $O(n^2 \log n)$ . Answer is option (b).

7. Let  $G$  be a weighted graph with edge weights greater than one and  $G'$  be the graph constructed by squaring the weights of edges in  $G$ . Let  $T$  and  $T'$  be the minimum spanning trees of  $G$  and  $G'$  respectively, with total weights  $t$  and  $t'$ . Which of the following statements is TRUE?
- $T' = T$  with total weight  $t' = t^2$
  - $T' = T$  with total weight  $t' < t^2$
  - $T' \neq T$  but total weight  $t' = t^2$
  - None of the above

**Ans.** Let  $G$  be the graph given as  $A \xrightarrow{2} B$ .

$G'$  be the graph represented as  $A' \xrightarrow{2} B' \xrightarrow{3} C'$ .

Graph  $G$  is the counter example for options (b) and (c).

Graph  $G'$  is the counter example for option (a).

So answer is option (d).

CS-GATE-2013

1. What is the time complexity of Bellman–Ford single source shortest path algorithm on a complete graph of  $n$  vertices?

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

**Ans.** The time complexity of Bellman-Ford algorithm is  $\theta(|V| \times |E|)$ . For a complete graph  $|E| = n(n - 1)/2$  and  $|V| = n$ . Therefore, the running time of Bellman-Ford algorithm becomes

$$\theta\left(n \times \frac{n(n-1)}{2}\right) = \theta(n^3).$$

Answer is option (c).

2. Which of the following statements are TRUE?

  - (A) The problem of determining whether there exists a cycle in an undirected graph is in P.
  - (B) The problem of determining whether there exists a cycle in an undirected graph is in NP.
  - (C) If a problem A is in NP-Complete, there exists a non-deterministic polynomial time algorithm to solve A.

(a) A, B and C	(b) A and B
(c) B and C only	(d) A and C only

**Ans.** The following statements are true.

- (A) This statement is true since cycle detection can be possible by DFS in  $O(V + E) = O(V^2)$  time which is a polynomial.
  - (B) The statement is true since every P problem is in NP i.e.,  $P \subset NP$ .
  - (C) This statement is also true because NP-Complete  $\in NP$ . Therefore the problem A can be solved in non-deterministic polynomial time. So answer is option (a).

3. Which one of the following is the tightest upper bound that represents the time complexity of inserting an object into a binary search tree of  $n$  nodes?

  - (a) O(1)
  - (b) O(log n)
  - (c) O(n)
  - (d) O( $n \log n$ )

**Ans.** For a binary search tree on  $n$  nodes, the tightest upper bound to insert a node is  $O(n)$ . The answer is option (c).

4. Which one of the following is the tightest upper bound that represents the number of swaps required to sort n numbers using selection sort?

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

**Ans.** The maximum number of swaps that takes place in selection sort on numbers is  $n$ .  
Answer is option (b)

5. Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.

Multi Dequeue (Q) {

    m = k

    while (Q is not empty) and m > 0 {

        Dequeue (Q)

        m = m - 1

    }

}

What is the worst case time complexity of a sequence of n queue operations on an initially empty queue?

- |                  |                     |
|------------------|---------------------|
| (a) $\Theta(n)$  | (b) $\Theta(n + k)$ |
| (c) $\Theta(nk)$ | (d) $\Theta(n^2)$   |

Ans. Answer is option (c).

6. Consider the following function. What is its running time?

int unknown (int n) {

    int i, j, k = 0;

    for (i = n/2; i <= n; i++)

        for (j = 2; j <= n; j = j \* 2)

            k = k + n/2

    return (k);

}

- |                   |                          |
|-------------------|--------------------------|
| (a) $\Theta(n^2)$ | (b) $\Theta(n^2 \log n)$ |
|-------------------|--------------------------|

- |                   |                          |
|-------------------|--------------------------|
| (c) $\Theta(n^3)$ | (d) $\Theta(n^2 \log n)$ |
|-------------------|--------------------------|

Ans.  $i = \left( \frac{n}{2}, \frac{n}{2} + 1, \frac{n}{2} + 2, \dots, n \right)$  repeats  $\frac{n}{2}$  to  $n = \left( \frac{n}{2} + 1 \right)$  times

$$\left. \begin{array}{l} j = 2, 2^2, 2^3, 2^4, \dots, n \\ k = k + n/2 \end{array} \right\} \quad k = \Theta(n \log n)$$

$$k = \frac{n}{2} + \frac{n}{2} + \dots \log n \text{ times} = \frac{n}{2} \log n$$

$$\text{So, } i = \frac{n}{2} \log n + \frac{n}{2} \log n + \dots + \left( \frac{n}{2} + 1 \right) \text{ times}$$

$$= \left( \frac{n}{2} + 1 \right) \frac{n}{2} \log n = \Theta(n^2 \log n)$$

Answer is option (b).

7. The number of elements that can be sorted in  $\Theta(\log n)$  time using heap sort is

(a)  $\Theta(1)$

(b)  $\Theta(\sqrt{\log n})$

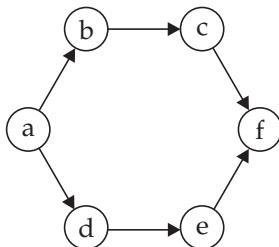
(c)  $\Theta\left(\frac{\log n}{\log \log n}\right)$

(d)  $\Theta(\log n)$

**Ans.** After constructing a max-heap in the heap-sort, the time to extract maximum element and then heapifying the heap takes  $\Theta(\log n)$  time by which we could say that  $\Theta(\log n)$  time is required to correctly place an element in sorted array. If  $\Theta(\log n)$  time is taken to sort using heap sort, then the number of elements that can be sorted is constant i.e.,  $\Theta(1)$ . Answer is option (a).

## CS-GATE-2016

1. Consider the following directed graph:



The number of different topological orderings of the vertices of the graph is .....

Ans. 6.

- a b c d e f
- a d e b c f
- a b d c e f
- a d b c e f
- a b d e c f
- a d b e c f

2. The worst case running times of Insertion sort, Merge sort and Quick sort, respectively, are:

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

Ans. (d)

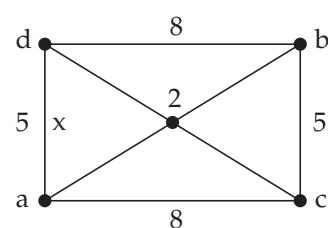
3. Consider the weighted undirected graph with 4 vertices, where the weight of edge  $\{i, j\}$  is given by the entry  $W_{ij}$  in the matrix  $W$ .

$$W = \begin{bmatrix} 0 & 2 & 8 & 5 \\ 2 & 0 & 5 & 8 \\ 8 & 5 & 0 & x \\ 5 & 8 & x & 0 \end{bmatrix}$$

The largest possible integer value of  $x$ , for which at least one shortest path between some pair of vertices will contain the edge with weight  $x$  is .....

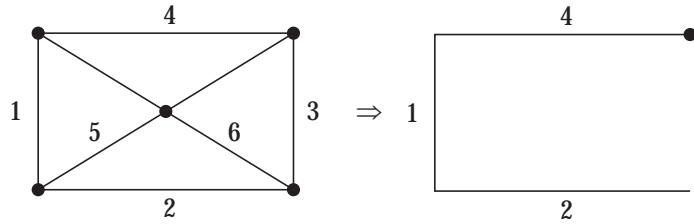
Ans. 12

If  $x = 12$  then the shortest path between  $d$  and  $c$  will contain edge with label 'x'.



4. Let  $G$  be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5, and 6. The maximum possible weight that a minimum weight spanning tree of  $G$  can have is ..... .

Ans. 7



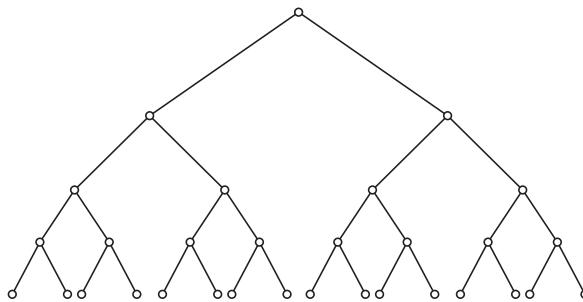
5.  $G = (V, E)$  is an undirected simple graph in which each edge has a distinct weight, and  $e$  is a particular edge of  $G$ . Which of the following statements about the minimum spanning trees (MSTs) of  $G$  is/are TRUE?

- I. If  $e$  is the lightest edge of some cycle in  $G$ , then every MST of  $G$  includes  $e$
- II. If  $e$  is the heaviest edge of some cycle in  $G$ , then every MST of  $G$  excludes  $e$
- (a) I only
- (b) II only
- (c) both I and II
- (d) neither I nor II

Ans. (b)

6. Breadth First Search (BFS) is started on a binary tree beginning from the root vertex. There is a vertex  $t$  at a distance four from the root. If  $t$  is the  $n$ -th vertex in this BFS traversal, then the maximum possible value of  $n$  is ..... .

Ans. 31.



Required vertex is 31st vertex.

7. Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are TRUE?

- I. Quick sort runs in  $\Theta(n^2)$  time
- II. Bubble sort runs in  $\Theta(n^2)$  time
- III. Merge sort runs in  $\Theta(n)$  time
- IV. Insertion sort runs in  $\Theta(n)$  time

- (a) I and II only
- (b) I and III only
- (c) II and IV only
- (d) I and IV only

**Ans.** (d)

8. The Floyd-Warshall algorithm for all-pair shortest paths computation is based on
- (a) Greedy paradigm
  - (b) Divide-and-Conquer paradigm
  - (c) Dynamic Programming paradigm
  - (d) Neither Greedy nor Divide-and-Conquer nor Dynamic Programming paradigm.

**Ans.** (c)

## CS-GATE-2017

1. Consider the following table:

Algorithms	Design Paradigms
P. Kruskal	i. Divide and Conquer
Q. Quicksort	ii. Greedy
R. Floyd-Warshall	iii. Dynamic Programming

Match the algorithms to the design paradigms they are based on.

- |                            |                            |
|----------------------------|----------------------------|
| (a) P-(ii), Q-(iii), R-(i) | (b) P-(iii), Q-(i), R-(ii) |
| (c) P-(ii), Q-(i), R-(iii) | (d) P-(i), Q-(ii), R-(iii) |

Ans. (c)

2. Let  $G = (V, E)$  be any connected undirected edge-weighted graph. The weights of the edges in  $E$  are positive and distinct. Consider the following statements:

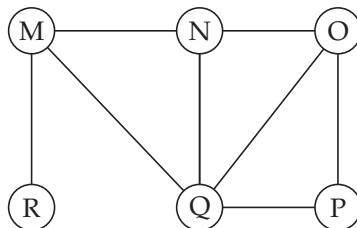
- (I) Minimum spanning tree of  $G$  is always unique.  
 (II) Shortest path between any two vertices of  $G$  is always unique.

Which of the above statements is/are necessarily true?

- |                       |                          |
|-----------------------|--------------------------|
| (a) (I) only          | (b) (II) only            |
| (c) Both (I) and (II) | (d) Neither (I) nor (II) |

Ans. (a)

3. The Breadth First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below?



- |            |            |
|------------|------------|
| (a) MNOPQR | (b) NQMPOR |
| (c) QMNROP | (d) POQNMR |

Ans. (d)

## 4. Match the following algorithms with their time complexities.

Algorithm	Time complexity
P. Tower of Hanoi with $n$ disks	i. $\Theta(n^2)$
Q. Binary search given $n$ sorted numbers	ii. $\Theta(n \log n)$
R. Heap sort given $n$ numbers at the worst case	iii. $\Theta(2^n)$
S. Addition of two $n \times n$ (matrices)	iv. $\Theta(\log n)$

(a) P-(iii), Q-(iv), R-(i), S-(ii)      (b) P-(iv), Q-(iii), R-(i), S-(ii)  
 (c) P-(iii), Q-(iv), R-(ii), S-(i)      (d) P-(iv), Q-(iii), R-(ii), S-(i)

Ans. (c)

## 5. Consider the recurrence function

$$T(n) = \begin{cases} 2T(\sqrt{n}) + 1, & n > 2 \\ 2, & 0 < n \leq 2 \end{cases}$$

Then  $T(n)$  in terms of  $\Theta$  notation is

- |                           |                      |
|---------------------------|----------------------|
| (a) $\Theta(\log \log n)$ | (b) $\Theta(\log n)$ |
| (c) $\Theta(\sqrt{n})$    | (d) $\Theta(n)$      |

Ans. (b)

Appendix

# VI MODEL QUESTION PAPERS FOR PRACTICE

## MODEL QUESTION PAPERS FOR PRACTICE SET-1

Time : 3 Hours

Full marks-70

Time 3 hours

Answer Question No. 1 which is compulsory and any five from the rest. The figure in the right-hand margin indicates marks.

1. Answer the following questions: (2 × 10)
  - (a) Explain time complexity of an algorithm.
  - (b) Compare the two functions  $n^2$  and  $2n/4$  for various values of  $n$ . Determine when the second becomes larger than the first function.
  - (c) Arrange the following functions by order of growth:  
 $e^n, n!, \sqrt{\log n}, n \log n, 2^n, n^3$
  - (d) What is backtracking? Write two applications of it.
  - (e) Why the recurrence  $T(n) = 2T(n/2) + n \log n$  is not solvable by Master method?
  - (f) What are the minimum and maximum number of elements in a heap of height  $h$ ?
  - (g) How dynamic programming differs from divide and conquer approach?
  - (h) What is Huffman code? What is the running time of Huffman's algorithm?
  - (i) How Prim's algorithm differs from Kruskal's algorithm?
  - (j) What do you mean by NP-COMPLETENESS?
2. (a) Show that  $f(n) = O(n^2)$  where  $f(n) = 3n^2 - n + 4$  and  $g(n) = n \log n + 5$ . (5)  
(b) Let  $f(n)$  and  $g(n)$  be asymptotically non-negative functions. Prove that  $\max(f(n), g(n)) = \Theta(f(n) + g(n))$ . (5)
3. (a) Determine an LCS of  $< 1, 0, 0, 1, 0, 1, 0, 1>$  and  $<0, 1, 0, 1, 1, 0, 1, 0>$ . (5)  
(b) Find the optimal parenthesization of a matrix-chain product of whose sequence dimensions are  $<5, 10, 3, 12, 5, 50, 6>$ . (5)
4. (a) Explain the properties of strongly connected components. (5)  
(b) Write a non-recursive algorithm of in-order traversal of a tree and also analyze its time complexity. (5)

5. (a) Solve the following recurrence relation (5)

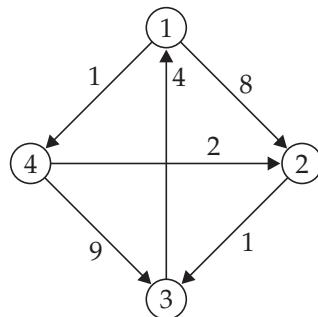
$$T(n) = \begin{cases} 1 & \text{where } n \leq 4 \\ T(\sqrt{n}) + c & \text{where } n > 4 \end{cases}$$

(b) Explain in detail how the technique of backtracking can be applied to solve 4-queens problem. Write an algorithm for this. (5)

6. (a) The edge length of a directed graph are given by the below matrix. Using the traveling salesman algorithm calculate the optimal tour. (5)

$$\begin{bmatrix} 0 & 20 & 30 & 10 & 11 \\ 15 & 0 & 16 & 4 & 2 \\ 3 & 5 & 0 & 2 & 4 \\ 19 & 6 & 18 & 0 & 3 \\ 16 & 4 & 7 & 16 & 0 \end{bmatrix}$$

(b) Run the Floyd-Warshall algorithm on the weighted, directed graph on the following graph to find all pairs shortest paths. (5)



7. (a) What is graph traversal? Write the algorithm for breadth-first-search (BFS)? What is its time complexity. (5)

(b) Prove that Kruskal's algorithm generates a minimum-cost spanning tree for every connected undirected graph G. (5)

8. (a) What are classes P, NP, NP-Hard and NP-Complete. Give examples of problems of each class.

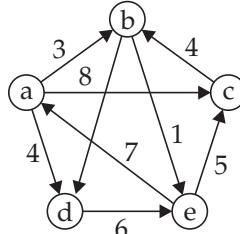
(b) Solve the Towers of Hanoi problem using recurrences. (5)

**MODEL QUESTION PAPERS FOR PRACTICE SET-2****Time : 3 Hours***Full marks-70**Time 3 hours*

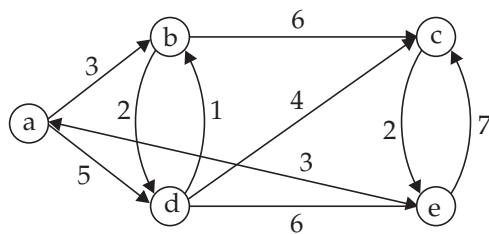
*Answer Question No. 1 which is compulsory and any five from the rest. The figures in the right-hand margin indicate marks.*

1. Answer the following questions: *(2 × 10)*
  - (a) What do you mean by performance analysis of an algorithm?
  - (b) Write the non-recursive algorithm to find the Fibonacci sequence and what is its running time?
  - (c) What are the worst case and best case time complexity of merge sort?
  - (d) What is comparison based sorting? Explain with example.
  - (e) What do you mean by lower bound for sorting?
  - (f) Does the Huffman coding help us to reduce the number of bits of expected encoding length ? Explain how?
  - (g) Give an instance when the quicksort algorithm has worst case time complexity.
  - (h) State the conditions when Master theorem is not satisfied.
  - (i) What is topological sort?
  - (j) State Cook's theorem.
2. (a) What is a RAM model? How will you analyze any algorithm through it? *(5)*
  - (b) What are asymptotic notation and asymptotic analysis? Discuss any three asymptotic notations. *(5)*
3. (a) Analyze the running time of quicksort using divide and conquer approach. *(5)*
  - (b) Does the Strassen's procedure for matrix multiplication is efficient than simple matrix multiplication? Explain how? Multiply the two given matrices using Strassen's method  $\begin{bmatrix} 1 & 3 \\ 7 & 5 \end{bmatrix} \begin{bmatrix} 5 & 4 \\ 2 & 1 \end{bmatrix}$ . *(5)*
4. (a) Use the recursion tree method to find the asymptotic solution to the recurrence  $T(n) = T(n/2) + T(n/4) + T(n/8) + n$ . *(5)*
  - (b) Show that the second smallest of  $n$  elements can be found with  $n + \lceil \log n \rceil - 2$  comparisons in the worst case. *(5)*
5. (a) What is activity scheduling? Give the greedy algorithm for activity scheduling? *(5)*
  - (b) Find the Huffman code of input string "ROSE IS RED". Consider space as character. Find the generated string. *(5)*

6. (a) Write the Prim's algorithm to find MST for the below graph. (5)

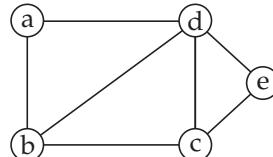


- (b) Write the Dijkstra's algorithm and find the shortest path from the source vertex a to all the remaining vertex.

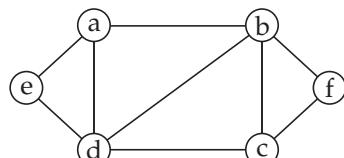


7. (a) Write the Robin-Karp pattern matching algorithm. Calculate the number of spurious hits for working modulo  $q = 11$  and text  $T = 3141592653589793$  when looking for the pattern  $p = 26$ . (5)

- (b) Perform the DFS traversal on the following graph taking a as start vertex. (5)



8. (a) Write the Greedy approximation for Vertex Cover problem. Find the Vertex Cover of minimum size for the following graph. (5)



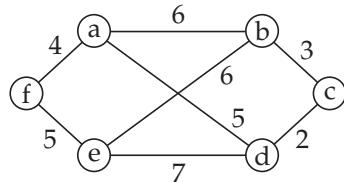
- (b) Show that Hamiltonian cycle is in NP.

**MODEL QUESTION PAPERS FOR PRACTICE SET-3****Time : 3 Hours***Full marks-70**Time 3 hours*

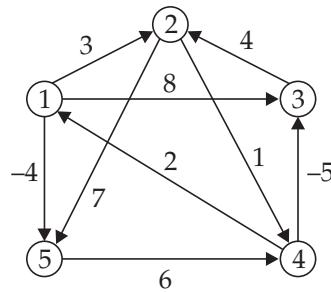
*Answer Question No. 1 which is compulsory and any five from the rest. The figures in the right-hand margin indicate marks.*

1. Answer the following questions: (2 × 10)
  - (a) Differentiate between comparison sorting and non-comparison sorting. Give examples of each.
  - (b) Which of the following functions is not  $\Omega(n^2)$ . Justify your answer.  $n^2 + 1000n$ ,  $n^{2.00001}$ ,  $n^2 + n$ ,  $n^{1.99999}$ .
  - (c) Write the Euclid's algorithm for finding the greatest common divisor of two numbers.
  - (d) What is a priority queue? Name any two applications of it.
  - (e) What is branch and bound?
  - (f) Explain how Greedy paradigm differs from dynamic programming?
  - (g) What is a spanning tree? What is minimum spanning tree?
  - (h) How Dijkstra's algorithm for shortest path problem differs from Bellman-Ford algorithm?
  - (i) Name any two efficient techniques for pattern matching?
  - (j) How does the polynomial time verification takes place?
2. Solve the following recurrences (2.5 × 4)
  - (i)  $T(n) = 2T(n/2) + n \log n$
  - (ii)  $T(n) = 3T(n/2) + n \log n$
  - (iii)  $T(n) = T(n - 1) + n$
  - (iv)  $T(n) = 2T(n/4) + n$
3. (a) Show how quicksort sorts the following sequences of keys in ascending order. (5)  
22, 55, 33, 11, 99, 77, 55, 66, 54, 21, 32  
(b) Explain the 0-1 Knapsack problem with Greedy concept. (5)
4. (a) What is disjoint set data structures? Explain its operations and different ways of representations. (5)  
(b) Write the dynamic programming algorithm for matrix-chain multiplication. (5)
5. (a) Find the LCS of the following two strings and what is its time complexity. (5)  
 $X = <\text{ABCBDAB}>$ ,  $Y = <\text{BDCABA}>$   
(b) Given 10 activities along with their start ( $S_i$ ) and finish ( $f_i$ ) time as. (5)  
 $S_i <1\ 2\ 3\ 4\ 7\ 8\ 9\ 9\ 11\ 12>$   
 $f_i <3\ 5\ 4\ 7\ 10\ 9\ 11\ 13\ 12\ 14>$   
Compute a schedule where largest number of activities take place.

6. (a) Write the Kruskal's algorithm for finding MST. Find the MST of the following graph. (5)



- (b) Write the Floyd-Warshall algorithm. Find the all pairs shortest paths of the following graph. (5)



7. (a) Working modulo  $q = 13$ , how many spurious hits does the Rabin-Karp matcher encounter in the text  $T = 2359023141526739921$  when looking for the pattern  $P = 34145$ . (5)

- (b) Write a dynamic programming algorithm for 0-1 Knapsack problem that runs in  $O(nW)$  time, where  $n$  is the number of items and  $W$  is the maximum weight. (5)

8. (a) Write short notes on any two: (2.5 × 2)

- (i) Assembly-line scheduling
- (ii) Elements of Greedy Strategy
- (iii) Strongly connected components

- (b) Prove that Clique problem is NP-COMPLETE. (5)

## FURTHER READINGS

1. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley Longmans, 1998.
2. M. De Berg, O. Cheong, M. Van Kreveld and M. Overmars, *Computational Geometry-Algorithms and Applications*, Springer, 3<sup>rd</sup> Edition, 2008.
3. M.T. Goodrich and R.Tamassaia, *Algorithm Design: Foundations, Analysis and Internet Examples*, John Willey and Sons, 2002.
4. T.H. Cormen, C.E.Leiserson, R.L.Rivest and C. Stein, *Introduction to Algorithms*, 3<sup>rd</sup> Edition, PHI, 2010.
5. M.R.Garey and D.S. Jhonson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H.Freeman, 1979.
6. Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, Second Edition, 1998.
7. Anany Levitin, *Introduction to the Design and Analysis of Algorithms*, 3<sup>rd</sup> Edition, Pearson, 2012.
8. Kurt Mehlhorn, *Data Structure and Algorithms I, Sorting and Searching*, Springer, 1984.
9. Kurt Mehlhorn, *Data Structure and Algorithms II, Graph Algorithms and NP-Completeness*, Springer, 1984.
10. Clifford A. Shaffer, *Data Structures and Algorithm Analysis*, <https://people.cs.vt.edu/shaffer/Book/JAVA3elatest.pdf>, 2013.
11. Jeffrey H. Kingston, *Algorithms and Data Structures: Design, Correctness and Analysis*, Addison-Wesley, 1997.
12. Sara Baase and Allen Van Gelder, *Computer Algorithms: Introduction to Design and Analysis*, Pearson Education, 2008.
13. R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, 2<sup>nd</sup> Edition, Addison-Wesley.
14. Mark A. Weiss, *Data Structures and Algorithm Analysis in C*, 2<sup>nd</sup> Edition, Pearson Education.
15. E. Horowitz, S. Sahni and S. Rajasekaran, *Fundamentals of Computer Algorithms*, 2<sup>nd</sup> Edition, Universities Press, 2008.
16. Hari Mohan Pandey, *Design and Analysis of Algorithms*, 1<sup>st</sup> Edition, University Science Press, Laxmi Publications, 2008.
17. S. R. Jena and S.K. Swain, *Theory of Computation and Application*, 1<sup>st</sup> Edition, University Science Press, Laxmi Publications, 2017.

# INDEX

$\leq 3$  Vertex cover problem, 4, 317  
0-1 Knapsack problem, 4, 195, 329  
3 SAT, 322

## A

---

Activity selection problem, 4, 192  
Adjacency list representation, 211  
Adjacency matrix representation, 211  
Algorithm, 2, 15  
Analysis of algorithm, 16  
Approximation algorithms, 4, 323  
Approximation scheme, 331  
Arithmetic series, 336  
Asymptotic analysis, 23  
Asymptotic notations, 4, 23  
Asymptotics, 334  
Average case analysis, 17

## B

---

Backtracking, 4, 279  
Bellman-Ford algorithm, 264  
Best case analysis, 17  
Big-O notation, 24  
Big- $\Omega$  notation, 25  
Binary search, 4, 109  
Biological computing, 20  
Boolean formula, 310  
Boolean variables, 310  
Bottom-up computation, 171  
BPP, 323  
Branch and bound approach, 16

Breadth-first search, 213  
Brute force approach, 15  
Bubble sort, 4, 49  
Bucket sort, 4, 56

## C

---

Ceil, 27  
Central limit theorem, 338  
Chain matrix multiplication, 4, 177  
Class NP, 307  
Class NP-complete, 310  
Class NP-hard, 309  
Class P, 306  
Clause, 310  
Clique problem, 4, 312  
CNF-satisfiability, 310  
Conjunctive normal form, 310  
co-NP, 322  
Constant series, 335  
Cook's theorem, 311  
Counting sort, 4, 52  
Cumulative distribution, 338  
Cumulative probability density function, 338  
Cycle, 210

## D

---

Debugging, 17  
Decision problem, 303  
Depth-first search, 213  
Digraph, 209, 210  
Dijkstra's algorithm, 261

Disjunctive normal form, 310  
 Divide and conquer approach, 4, 15, 60  
 DNF-satisfiability, 310  
 Dynamic programming, 4, 15, 171, 191

**E**


---

Efficient algorithm, 303  
 EM model, 19  
 Euclid's algorithm, 9  
 Exact cover problem, 4, 319  
 Exponential functions, 28  
 Exponential, 335  
 External memory model, 19

**F**


---

Floor, 27  
 Floyd-Warshall algorithm, 268, 269  
 FPTAS, 331  
 Fractional Knapsack problem, 4, 195

**G**


---

Gaussian distribution, 338  
 GCD, 8, 9  
 Geometric series, 336  
 Graph coloring problem, 4, 283  
 Graph traversal, 212  
 Greatest common divisor, 8  
 Greedy algorithms, 4, 191  
 Greedy approach, 16  
 Greedy choice property, 194

**H**


---

Hamiltonian cycle, 306  
 Hamiltonian path, 305  
 Harmonic series, 336  
 Heap property, 137  
 Heaps, 137  
 Heapsort, 4, 136, 149

Hitting set, 322  
 Huffman coding, 197  
 Huffman's algorithm, 199

**I**


---

Independent set problem, 4, 314  
 Input instance, 14  
 Insertion sort, 4, 42  
 Invalid shift, 286

**K**


---

Knapsack problem, 4, 195  
 Kruskal's algorithm, 247, 248

**L**


---

LCS, 4  
 Linear geometric series, 336  
 Literal, 310  
 Little-o notation, 25  
 Little- $\omega$  notation, 25  
 Logarithmic functions, 27  
 Logspace reduction, 305  
 Longest common subsequence, 172  
 Longest path, 322

**M**


---

Macro analysis, 16  
 Many-one reducible, 305  
 Master theorem, 4, 73, 100  
 Matching problem, 302  
 Max-heapify, 139, 140  
 Maximum heap, 139  
 Maximum priority queue, 160  
 Maximum matching, 303  
 Merge sort, 4, 60  
 Micro analysis, 16  
 Min-heapify, 156  
 Minimum hamiltonian cycle, 324

Minimum heap, 139  
 Minimum priority queue, 161  
 Minimum spanning tree, 245  
 Modular arithmetic, 27  
 Monotonicity, 26  
 Multivariate gaussian distribution, 338

**N**

Navie string matching, 287  
 NP, 4  
 NP-Complete, 4  
 NP-hard, 4  
 N-queens problem, 4, 279

**O**

Optimal substructure, 171  
 Optimal substructure, 194  
 Optimization problem, 304  
 Order statistics, 127

**P**

P, 4  
 Parallel random access machine, 20  
 Partition, 322  
 Paths, 210  
 Perfect matching, 303  
 Performance measurement, 17  
 Polylogarithmic, 335  
 Polynomial reduction, 304  
 Polynomial, 27, 335  
 Posteriori analysis, 16  
 PRAM model, 20  
 Prim's algorithm, 247, 252  
 Principle of optimality, 171  
 Priori analysis, 16  
 Priority queue, 160  
 Probability density function, 337  
 Probability distributions, 337

Problem, 14  
 Program, 15  
 Propositional calculus, 310  
 Prover, 307  
 PSPACE, 323  
 PTAS, 331

**Q**

Quadratic series, 336  
 Quantum model, 20  
 Quicksort, 4, 115

**R**

Radix sort, 4, 54  
 RAM model, 8, 17  
 Random access machine, 8, 17  
 Random quicksort, 120  
 Randomized classes, 323  
 Recurrences, 4, 73  
 Recursion tree, 281  
 Recursion-tree method, 4, 73, 83  
 Recursive N-queens, 280  
 Reduction, 303  
 Relaxation procedure, 260  
 Robin-Karp string matching, 292  
 RP, 323

**S**

Satisfiability problem, 310  
 Search problem, 304  
 Selection problem, 128  
 Selection sort, 4, 51  
 Set cover, 322  
 Size of an instance, 14  
 Space complexity, 16  
 Spanning tree, 245  
 Spurious hit, 292  
 Standard normal distribution, 338

Strassen's matrix multiplication, 67  
String matching, 286  
Strongly connected components, 240  
Subset sum problem, 4, 282, 321  
Substitution method, 4, 73  
Substructure, 171  
Summations, 335

**T**

---

Table structure, 171  
Testing of algorithm, 17  
Theta notation, 24  
Time complexity, 16  
Topological sort, 240  
Travelling salesman problem, 4, 324

**U**

---

Undirected graph, 209

**V**

---

Valid shift, 286  
Validation of algorithm, 16  
Verifier, 307  
Vertex cover problem, 4, 316, 327

**W**

---

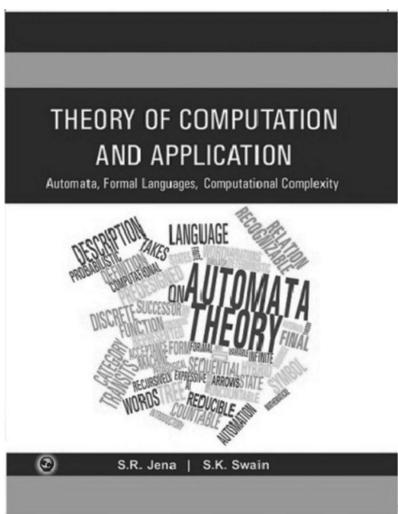
Worst case analysis, 17

**Z**

---

ZPP, 323

## YOU MAY BE ALSO INTERESTED IN



**Title:** Theory of Computation and Application (Automata, Formal Languages, Computational Complexity)

**Authors:** S.R. Jena, S.K. Swain

### About The Book

This book is intended for the students who are pursuing courses in B.Tech/B.E (CSE/IT), M.Tech/M.E (CSE/IT), MCA, M.Sc (CS/IT). The book covers different crucial theoretical aspects such as of Automata Theory, Formal Language Theory, Computability Theory and Computational Complexity Theory and their applications. This book can be used as a text or reference book for a one-semester course in theory of computation or automata theory.

It includes the detailed coverage of:

- Introduction to Theory of Computation
- Essential Mathematical Concepts
- Finite State Automata
- Formal Language & Formal Grammar
- Regular Expressions & Regular Languages
- Context-Free Grammar
- Pushdown Automata
- Turing Machines
- Recursively Enumerable & Recursive Languages
- Complexity Theory

### Key Features

- ◆ Presentation of concepts in clear, compact and comprehensible manner
- ◆ Chapterwise supplement of theorems and formal proofs
- ◆ Display of chapterwise appendices with case studies, applications and some prerequisites
- ◆ Pictorial two-minute drill to summarize the whole concept
- ◆ Inclusion of more than 400 solved with additional problems
- ◆ Questions of GATE with their keys for the aspirants to have the thoroughness, practice and multiplicity
- ◆ Key terms, Review questions and Problems at chapterwise termination