



QUANTUM Series

Semester - 5

CS & IT

Design and Analysis of Algorithms



- Topic-wise coverage of entire syllabus in Question-Answer form.
- Short Questions (2 Marks)

Includes solution of following AKTU Question Papers

2015-16 • 2016-17 • 2017-18 • 2018-19 • 2019-20

QUANTUM SERIES

For

B.Tech Students of Third Year
of All Engineering Colleges Affiliated to

**Dr. A.P.J. Abdul Kalam Technical University,
Uttar Pradesh, Lucknow**

(Formerly Uttar Pradesh Technical University)

Design and Analysis of Algorithm

By

Prashant Agrawal



QUANTUM PAGE PVT. LTD.
Ghaziabad ■ New Delhi

PUBLISHED BY : **Apram Singh**
Quantum Publications®
(A Unit of Quantum Page Pvt. Ltd.)
Plot No. 59/2/7, Site - 4, Industrial Area,
Sahibabad, Ghaziabad-201 010

Phone : 0120-4160479

Email : pagequantum@gmail.com **Website:** www.quantumpage.co.in

Delhi Office : 1/6590, East Rohtas Nagar, Shahdara, Delhi-110032

© ALL RIGHTS RESERVED

*No part of this publication may be reproduced or transmitted,
in any form or by any means, without permission.*

Information contained in this work is derived from sources believed to be reliable. Every effort has been made to ensure accuracy, however neither the publisher nor the authors guarantee the accuracy or completeness of any information published herein, and neither the publisher nor the authors shall be responsible for any errors, omissions, or damages arising out of use of this information.

Design and Analysis of Algorithm (CS/IT : Sem-5)

1st Edition : 2010-11

2nd Edition : 2011-12

3rd Edition : 2012-13

4th Edition : 2013-14

5th Edition : 2014-15

6th Edition : 2015-16

7th Edition : 2016-17

8th Edition : 2017-18

9th Edition : 2018-19

10th Edition : 2019-20

11th Edition : 2020-21 (*Thoroughly Revised Edition*)

Price: Rs. 90/- only

Printed Version : e-Books.

CONTENTS

KCS-503 : DESIGN AND ANALYSIS OF ALGORITHM

UNIT-1 : INTRODUCTION

(1-1 B to 1-36 B)

Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements, Sorting and Order Statistics - Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time.

UNIT-2 : ADVANCED DATA STRUCTURE

(2-1 B to 2-52 B)

Red-Black Trees, B – Trees, Binomial Heaps, Fibonacci Heaps, Tries, Skip List.

UNIT-3 : GRAPH ALGORITHMS

(3-1 B to 3-42 B)

Divide and Conquer with Examples Such as Sorting, Matrix Multiplication, Convex Hull and Searching.

Greedy Methods with Examples Such as Optimal Reliability Allocation, Knapsack, Minimum Spanning Trees – Prim's and Kruskal's Algorithms, Single Source Shortest Paths - Dijkstra's and Bellman Ford Algorithms.

UNIT-4 : DYNAMIC PROGRAMMING

(4-1 B to 4-34 B)

Dynamic Programming with Examples Such as Knapsack. All Pair Shortest Paths – Warshal's and Floyd's Algorithms, Resource Allocation Problem.

Backtracking, Branch and Bound with Examples Such as Travelling Salesman Problem, Graph Coloring, n-Queen Problem, Hamiltonian Cycles and Sum of Subsets.

UNIT-5 : SELECTED TOPICS

(5-1 B to 5-33 B)

Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NP-Completeness, Approximation Algorithms and Randomized Algorithms.

SHORT QUESTIONS

(SQ-1 B to SQ-23 B)

SOLVED PAPERS (2015-16 TO 2019-20)

(SP-1 B to SP-17 B)



QUANTUM Series

Related titles in Quantum Series

For Semester - 5

(Computer Science & Engineering / Information Technology)

- Database Management System
- Design and Analysis of Algorithm
- Compiler Design
- Web Technology

Departmental Elective-I

- Data Analytics
- Computer Graphics
- Object Oriented System Design

Departmental Elective-II

- Machine Learning Techniques
- Application of Soft Computing
- Human Computer Interface

Common Non Credit Course (NC)

- Constitution of India, Law & Engineering
- Indian Tradition, Culture & Society

- Topic-wise coverage in Question-Answer form.
- Clears course fundamentals.
- Includes solved University Questions.

A comprehensive book to get the big picture without spending hours over lengthy text books.

Quantum Series is the complete one-stop solution for engineering student looking for a simple yet effective guidance system for core engineering subject. Based on the needs of students and catering to the requirements of the syllabi, this series uniquely addresses the way in which concepts are tested through university examinations. The easy to comprehend question answer form adhered to by the books in this series is suitable and recommended for student. The students are able to effortlessly grasp the concepts and ideas discussed in their course books with the help of this series. The solved question papers of previous years act as a additional advantage for students to comprehend the paper pattern, and thus anticipate and prepare for examinations accordingly.

The coherent manner in which the books in this series present new ideas and concepts to students makes this series play an essential role in the preparation for university examinations. The detailed and comprehensive discussions, easy to understand examples, objective questions and ample exercises, all aid the students to understand everything in an all-inclusive manner.

- The perfect assistance for scoring good marks.
- Good for brush up before exams.
- Ideal for self-study.



Quantum Publications®

(A Unit of Quantum Page Pvt. Ltd.)

Plot No. 59/2/7, Site-4, Industrial Area, Sahibabad,
Ghaziabad, 201010, (U.P.) Phone: 0120-4160479

E-mail: pagequantum@gmail.com Web: www.quantumpage.co.in



Find us on: facebook.com/quantumseriesofficial

Design and Analysis of Algorithm (KCS503)		
Course Outcome (CO)		Bloom's Knowledge Level (KL)
At the end of course , the student will be able to:		
CO 1	Design new algorithms, prove them correct, and analyze their asymptotic and absolute runtime and memory demands.	K ₄ , K ₆
CO 2	Find an algorithm to solve the problem (create) and prove that the algorithm solves the problem correctly (validate).	K ₅ , K ₆
CO 3	Understand the mathematical criterion for deciding whether an algorithm is efficient, and know many practically important problems that do not admit any efficient algorithms.	K ₂ , K ₅
CO 4	Apply classical sorting, searching, optimization and graph algorithms.	K ₂ , K ₄
CO 5	Understand basic techniques for designing algorithms, including the techniques of recursion, divide-and-conquer, and greedy.	K ₂ , K ₃
DETAILED SYLLABUS		3-1-0
Unit	Topic	Proposed Lecture
I	Introduction: Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements, Sorting and Order Statistics - Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time.	08
II	Advanced Data Structures: Red-Black Trees, B – Trees, Binomial Heaps, Fibonacci Heaps, Tries, Skip List	08
III	Divide and Conquer with Examples Such as Sorting, Matrix Multiplication, Convex Hull and Searching. Greedy Methods with Examples Such as Optimal Reliability Allocation, Knapsack, Minimum Spanning Trees – Prim's and Kruskal's Algorithms, Single Source Shortest Paths - Dijkstra's and Bellman Ford Algorithms.	08
IV	Dynamic Programming with Examples Such as Knapsack. All Pair Shortest Paths – Warshall's and Floyd's Algorithms, Resource Allocation Problem. Backtracking, Branch and Bound with Examples Such as Travelling Salesman Problem, Graph Coloring, n-Queen Problem, Hamiltonian Cycles and Sum of Subsets.	08
V	Selected Topics: Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NP-Completeness, Approximation Algorithms and Randomized Algorithms	08
Text books:		
1. Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, "Introduction to Algorithms", Prentice Hall of India. 2. E. Horowitz & S Sahni, "Fundamentals of Computer Algorithms", 3. Aho, Hopcraft, Ullman, "The Design and Analysis of Computer Algorithms" Pearson Education, 2008. 4. LEE "Design & Analysis of Algorithms (POD)", McGraw Hill 5. Richard E. Neapolitan "Foundations of Algorithms" Jones & Bartlett Learning 6. Jon Kleinberg and Éva Tardos, Algorithm Design, Pearson, 2005. 7. Michael T Goodrich and Roberto Tamassia, Algorithm Design: Foundations, Analysis, and Internet Examples, Second Edition, Wiley, 2006. 8. Harry R. Lewis and Larry Denenberg, Data Structures and Their Algorithms, Harper Collins, 1997 9. Robert Sedgewick and Kevin Wayne, Algorithms, fourth edition, Addison Wesley, 2011. 10. Harsh Bhasin, "Algorithm Design and Analysis", First Edition, Oxford University Press. 11. Gilles Brassard and Paul Bratley, Algorithmics: Theory and Practice, Prentice Hall, 1995.		

1**UNIT**

Introduction

CONTENTS

- | | |
|---|-----------------------|
| Part-1 : Algorithms, Analyzing | 1-2B to 1-3B |
| Algorithms, Complexity
of Algorithms | |
| Part-2 : Growth of Functions, | 1-3B to 1-13B |
| Performance
Measurements | |
| Part-3 : Sorting and Order Statistic : | 1-14B to 1-20B |
| Shell Sort, Quick Sort | |
| Part-4 : Merge Sort | 1-20B to 1-23B |
| Part-5 : Heap Sort | 1-23B to 1-30B |
| Part-6 : Comparison of Sorting | 1-30B to 1-35B |
| Algorithms, Sorting in
Linear Time | |

PART- 1

Introduction : Algorithms, Analyzing Algorithms, Complexity of Algorithms.

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 1.1. What do you mean by algorithm ? Write the characteristics of algorithm.

Answer

1. An algorithm is a set of rules for carrying out calculation either by hand or on machine.
2. It is a finite step-by-step procedure to achieve a required result.
3. It is a sequence of computational steps that transform the input into the output.
4. An algorithm is a sequence of operations performed on data that have to be organized in data structures.

Characteristics of algorithm are :

1. **Input and output :** The algorithm must accept zero or more inputs and must produce at least one output.
2. **Definiteness :** Each step of algorithm must be clear and unambiguous.
3. **Effectiveness :** Every step must be basic and essential.
4. **Finiteness :** Total number of steps used in algorithm should be finite.

Que 1.2. What do you mean by analysis or complexity of an algorithm ? Give its types and cases.

Answer

Analysis/complexity of an algorithm :

The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n .

Types of complexity :

1. **Space complexity :** The space complexity of an algorithm is the amount of memory it needs to run to completion.
2. **Time complexity :** The time complexity of an algorithm is the amount of time it needs to run to completion.

Cases of complexity :

1. **Worst case complexity :** The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.

2. **Average case complexity :** The running time for any given size input will be the average number of operations over all problem instances for a given size.
3. **Best case complexity :** The best case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n .

PART-2

Growth of Functions, Performance Measurements.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.3. What do you understand by asymptotic notations ?

Describe important types of asymptotic notations.

OR

Discuss asymptotic notations in brief.

Answer

1. Asymptotic notation is a shorthand way to represent the fastest possible and slowest possible running times for an algorithm.
2. It is a line that stays within bounds.
3. These are also referred to as 'best case' and 'worst case' scenarios and are used to find complexities of functions.

Notations used for analyzing complexity are :

1. **Θ -Notation (Same order) :**

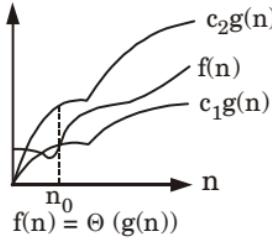


Fig. 1.3.1.

- a. This notation bounds a function within constant factors.
 - b. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0 , c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.
2. **O-Notation (Upper bound) :**
 - a. Big-oh is formal method of expressing the upper bound of an algorithm's running time.

- b. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.
- c. More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$,

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

- d. Then, $f(n)$ is big-oh of $g(n)$. This is denoted as :

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

i.e., the set of functions which, as n gets large, grow faster than a constant time $f(n)$.

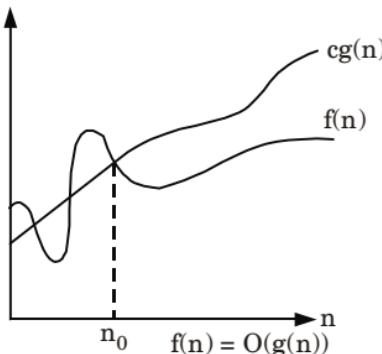


Fig. 1.3.2.

3. **Ω -Notation (Lower bound) :**

- a. This notation gives a lower bound for a function within a constant factor.
- b. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 , the value of $f(n)$ always lies on or above $cg(n)$.

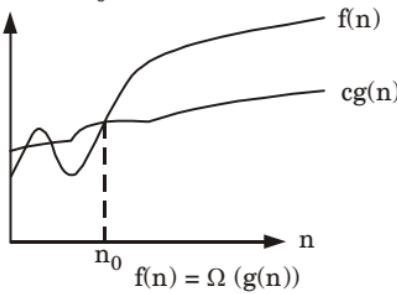


Fig. 1.3.3.

4. **Little-oh notation (o) :** It is used to denote an upper bound that is asymptotically tight because upper bound provided by O-notation is not tight.

$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \ \forall n \geq n_0\}$

5. **Little omega notation (ω) :** It is used to denote lower bound that is asymptotically tight.

$\omega(g(n)) = \{f(n) : \text{For any positive constant } c > 0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \ \forall n \geq n_0\}$

Que 1.4. If $f(n) = 100 * 2^n + n^5 + n$, then show that $f(n) = O(2^n)$.

Answer

$$\text{If } f(n) = 100 * 2^n + n^5 + n$$

$$\text{For } n^5 \geq n$$

$$\begin{aligned} 100 * 2^n + n^5 + n &\leq 100 * 2^n + n^5 + n^5 \\ &\leq 100 * 2^n + 2n^5 \end{aligned}$$

$$\text{For } 2^n \geq n^5$$

$$\begin{aligned} 100 * 2^n + n^5 + n &\leq 100 * 2^n + 2.2^n \\ &\leq 102 * 2^n \end{aligned}$$

$$\text{Thus, } f(n) = O(2^n)$$

$$[\because n \leq 1, n_0 = 23]$$

Que 1.5. Write Master's theorem and explain with suitable examples.

Answer

Master's theorem :

Let $T(n)$ be defined on the non-negative integers by the recurrence.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b > 1 \text{ are constants}$$

a = Number of sub-problems in the recursion

$1/b$ = Portion of the original problem represented by each sub-problem

$f(n)$ = Cost of dividing the problem and the cost of merging the solution

Then $T(n)$ can be bounded asymptotically as follows :

Case 1 :

If it is true that : $f(n) = O(n^{\log_b a - E})$ for $E > 0$

It follows that : $T(n) = \Theta(n^{\log_b a})$

$$\text{Example : } T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

In the given formula, the variables get the following values :

$$\begin{aligned} a &= 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3 \\ n^{\log_b a} &= n^{\log_2 8} = n^3 \\ f(n) &= O(n^{\log_b a - E}) = O(n^3 - E) \end{aligned}$$

For $E = 1$, we get

$$f(n) = O(n^{3-1}) = O(n^2)$$

Since this equation holds, the first case of the Master's theorem applies to the given recurrence relation, thus resulting solution is

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

Case 2 :

If it is true that : $f(n) = \Theta(n^{\log_b a})$

It follows that : $T(n) = \Theta(n^{\log_b a} \log(n))$

Example :

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

In the given formula, the variables get the following values :

$$\begin{aligned} a &= 2, b = 2, f(n) = n, \log_b a = \log_2 2 = 1 \\ n^{\log_b a} &= n^{\log_2 2} = n \\ f(n) &= \Theta(n^{\log_b a}) = \Theta(n) \end{aligned}$$

Since this equation holds, the second case of the Master's theorem applies to the given recurrence relation, thus resulting solution is :

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

Case 3 :

If it is true that : $f(n) = \Omega(n^{\log_b a} + E)$ for $E > 0$

and if it is also true that :

if $af\left(\frac{n}{b}\right) \leq cf(n)$ for $a, c < 1$ and all sufficiently large n

It follows that : $T(n) = \Theta(f(n))$

$$\text{Example : } T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

In the given formula, the variables get the following values :

$$\begin{aligned} a &= 2, b = 2, f(n) = n^2, \log_b a = \log_2 2 = 1 \\ n^{\log_b a} &= n^{\log_2 2} = n \\ f(n) &= \Omega(n^{\log_b a} + E) \end{aligned}$$

For $E = 1$ we get

$$f(n) = \Omega(n^{1+1}) = \Omega(n^2)$$

Since the equation holds, third case of Master's theorem is applied.

Now, we have to check for the second condition of third case, if it is true that :

$$af\left(\frac{n}{b}\right) \leq c f(n)$$

If we insert once more the values, we get :

$$2\left(\frac{n}{2}\right)^2 \leq cn^2 \Rightarrow \frac{1}{2}n^2 \leq cn^2$$

If we choose $c = \frac{1}{2}$, it is true that :

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 \quad \forall n \geq 1$$

So, it follows that : $T(n) = \Theta(f(n))$

If we insert once more the necessary values, we get :

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

Thus, the given recurrence relation $T(n)$ was in $\Theta(n^2)$.

Que 1.6. The recurrence $T(n) = 7T(n/2) + n^2$ describe the running time of an algorithm A. A competing algorithm A has a running

time $T'(n) = aT' (n/4) + n^2$. What is the largest integer value for a 'is asymptotically faster than A ?

AKTU 2017-18, Marks 10

Answer

Given that :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \quad \dots(1.6.1)$$

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2 \quad \dots(1.6.2)$$

Here, eq. (1.6.1) defines the running time for algorithm A and eq. (1.6.2) defines the running time for algorithm A' . Then for finding value of a for which A' is asymptotically faster than A we find asymptotic notation for the recurrence by using Master's method.

Now, compare eq. (1.6.1) by $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we get,

$$a = 7$$

$$b = 2$$

$$f(n) = n^2$$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

Now, apply cases of Master's theorem as :

Case 1 :	$f(n) = O(n^{\log_2 7 - E})$
\Rightarrow	$f(n) = O(n^{2.81 - E})$
\Rightarrow	$f(n) = O(n^{2.81 - 0.81})$
\Rightarrow	$f(n) = O(n^2)$

Hence, case 1 of Master's theorem is satisfied.

Thus,

$T(n) = \Theta(n^{\log_b a})$
$\Rightarrow T(n) = \Theta(n^{2.81})$

Since recurrence given by eq. (1.6.1) is asymptotically bounded by Θ -notation by which is used to show optimum time we have to show that recurrence given by eq. (1.6.2) is bounded by Ω -notation which shows minimum time (best case).

For the use satisfy the case 3 of Master theorem, let $a = 16$

$$T'(n) = 16T'\left(\frac{n}{4}\right) + n^2$$

\Rightarrow

$a = 16$
$b = 4$
$f(n) = n^2$

$$\Omega(n^{\log_b a + E}) = \Omega(n^{2+E})$$

Hence, case 3 of Master's theorem is satisfied.

$\Rightarrow T(n) = \Theta(f(n))$
$\Rightarrow T(n) = \Theta(n^2)$

Therefore, this shows that A' is asymptotically faster than A when $a = 16$.

Que 1.7. The recurrence $T(n) = 7T(n/3) + n^2$ describes the running time of an algorithm A. Another competing algorithm B has a running time of $S(n) = aS(n/9) + n^2$. What is the smallest value of a such that B is asymptotically faster than A?

AKTU 2018-19, Marks 10

Answer

Given that :

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2 \quad \dots(1.7.1)$$

$$S'(n) = aS'\left(\frac{n}{9}\right) + n^2 \quad \dots(1.7.2)$$

Here, eq. (1.7.1) defines the running time for algorithm A and eq. (1.7.2) defines the running time for algorithm B. Then for finding value of a for which B is asymptotically faster than A we find asymptotic notation for the recurrence by using Master's method.

Now, compare eq. (1.7.1) with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we get, $a = 7, b = 3$

$$f(n) = n^2$$

$$n^{\log_b a} = n^{\log_3 7} = n^{2.81}$$

Now, apply cases of Master's theorem as :

Case 3 : $f(n) = O(n^{\log_3 7 + \epsilon})$

$\Rightarrow f(n) = O(n^{1.77 + \epsilon})$

$\Rightarrow f(n) = O(n^{1.77 + 0.23})$

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

Hence, case 3 of Master's theorem is satisfied.

Thus, $T(n) = \theta(f(n))$

$\Rightarrow T(n) = \theta(n^2)$

Since recurrence (1) is asymptotically bounded by θ -notation which is used to show optimum time we have to show that recurrence given by eq. (1.7.2) is bounded by Ω -notation which shows minimum time (best case).

For the use satisfy the case 2 of Master theorem, Guess $a = 81$

$$S'(n) = f(n) = 81 S'\left(\frac{n}{9}\right) + n^2$$

$\Rightarrow a = 81, b = 9$

$$f(n) = n^{\log_9 81}$$

$$f(n) = \Omega(n^{\log_b a}) = \Omega(n^2)$$

Hence, case 2 of Master's theorem is satisfied.

$\Rightarrow T(n) = \theta(n^{\log_9 81} \log n)$

$$\Rightarrow T(n) = \theta(n^2 \log n)$$

Therefore, this shows that B is asymptotically faster than A when $a = 81$.

Que 1.8. Solve the following recurrences :

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

Answer

$$T(n) = T(\sqrt{n}) + O(\log n) \quad \dots(1.8.1)$$

$$\begin{aligned} \text{Let } & m = \log n \\ & n = 2^m \\ & n^{1/2} = 2^{m/2} \end{aligned} \quad \dots(1.8.2)$$

Put value of \sqrt{n} in eq. (1.8.1) we get

$$T(2^m) = T(2^{m/2}) + O(\log 2^m) \quad \dots(1.8.3)$$

$$x(m) = T(2^m) \quad \dots(1.8.4)$$

Putting the value of $x(m)$ in eq. (1.8.3)

$$x(m) = x\left(\frac{m}{2}\right) + O(m) \quad \dots(1.8.5)$$

Solution of eq. (1.8.5) is given as

$$\begin{aligned} a &= 1, \quad b = 2, \quad f(n) = O(m) \\ m^{\log_b a} &= m^{\log_2 1 + E} \quad \text{where } E = 1 \\ x(m) &= \theta(\log m) \\ T(n) &= \theta(\log \log n) \end{aligned}$$

Que 1.9.

- Solve the recurrence $T(n) = 2T(n/2) + n^2 + 2n + 1$
- Prove that worst case running time of any comparison sort is $\Omega(n \log n)$.

AKTU 2019-20, Marks 07

Answer

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

$$\text{Compare it with } T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$\text{we have, } a = 2, b = 2, f(n) = n^2$$

Now, we apply cases for Master's theorem.

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

This satisfies case 3 of Master's theorem.

$$\begin{aligned} \Rightarrow f(n) &= \Omega(n^{\log_b a + E}) = \Omega(n^{1+E}) \\ &= \Omega(n^{1+1}) \\ &= \Omega(n^2) \end{aligned}$$

where $E = 1$

Again $2f\left(\frac{n^2}{2}\right) \leq c f(n^2)$... (1.9.1)

eq. (1.9.1) is true for $c = 2$

$$\Rightarrow T(n) = \theta(f(n))$$

$$\Rightarrow T(n) = \theta(f(n^2))$$

- ii. Let $T(n)$ be the time taken by merge sort to sort any array of n elements.

Therefore, $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \left(\left\lceil \frac{n}{2} \right\rceil\right) + g(n)$

where $g(n) \in \Theta(n)$

This recurrence, which becomes :

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

when n is even is a special case of our general analysis for divide-and conquer algorithms.

Compare the above given recurrence with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we get

$$a = 2$$

$$b = 2$$

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

Now we find, $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

$$\Rightarrow f(n) = \Theta(n)$$

i.e., case 2 of Master's theorem applied then

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

$$\Rightarrow T(n) = \Omega(n \log n)$$

Hence, the worst case running time of merge sort is $\Omega(n \log n)$.

Que 1.10. What do you mean by recursion ? Explain your answer with an example.

Answer

1. Recursion is a process of expressing a function that calls itself to perform specific operation.
2. Indirect recursion occurs when one function calls another function that then calls the first function.
3. Suppose P is a procedure containing either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure P . Then P is called recursive procedure.
4. A recursive procedure must have the following two properties :
 - a. There must be certain criteria, called base criteria, for which the procedure does not call itself.

- b. Each time the procedure does call itself, it must be closer to the criteria.
- 5. A recursive procedure with these two properties is said to be well-defined.

For example :

The factorial function may also be defined as follows :

- a. If $n = 0$, then $n! = 1$.

Here, the value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value).

- b. If $n > 0$, then $n! = n \cdot (n - 1)!$

Here, the value of $n!$ for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0.

Observe that this definition of $n!$ is recursive, since it refers to itself when it uses $(n - 1)!$

Que 1.11. What is recursion tree ? Describe.

Answer

1. Recursion tree is a pictorial representation of an iteration method, which is in the form of a tree, where at each level nodes are expanded.
2. In a recursion tree, each node represents the cost of a single subproblem.
3. Recursion trees are particularly useful when the recurrence describes the running time of a divide and conquer algorithm.
4. A recursion tree is best used to generate a good guess, which is then verified by the substitution method.
5. It is a method to analyze the complexity of an algorithm by diagramming the recursive function calls in the form of tree.

Que 1.12. Solve the recurrence :

$$T(n) = T(n - 1) + T(n - 2) + 1, \text{ when } T(0) = 0 \text{ and } T(1) = 1.$$

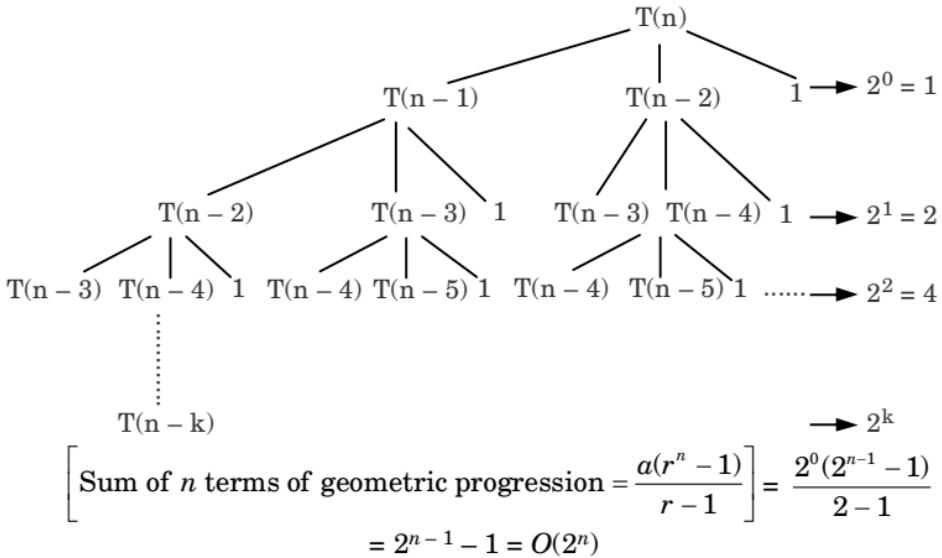
Answer

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

At k^{th} level, $T(1)$ will be equal to 1

when, $n - k = 1$

$$\begin{aligned} k &= n - 1 \\ &= 2^0 + 2^1 + 2^2 + \dots + 2^k \\ &= 2^0 + 2^1 + 2^2 + \dots + 2^{n-1} \end{aligned}$$



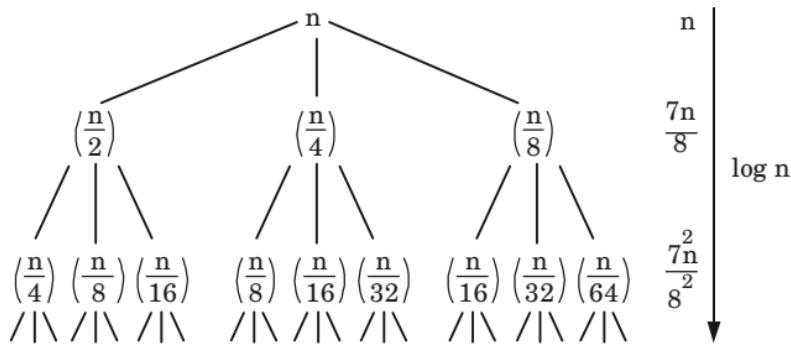
Que 1.13. Solve the following recurrences :

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

AKTU 2019-20, Marks 07

Answer

$$\begin{aligned} T(n) &= n + \frac{7n}{8} + \frac{7^2 n}{8^2} + \dots + \log n \text{ times} \\ &= \Omega(n \log n) \end{aligned}$$

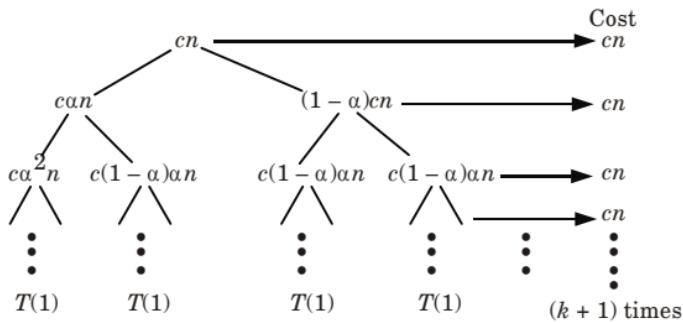


Que 1.14. Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

AKTU 2018-19, Marks 07

Answer

$$T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$$

Recursion tree :Assuming $T(1) = 1$ So $c(1 - \alpha)k n = 1$

$$cn = \frac{1}{(1 - \alpha)^k} = \left(\frac{1}{1 - \alpha}\right)^k$$

$$\log(n) = k \log\left(\frac{1}{1 - \alpha}\right)$$

$$k = \frac{\log cn}{\log\left(\frac{1}{1 - \alpha}\right)} = \log_{\frac{1}{1 - \alpha}}(cn)$$

So, Total cost = $cn + cn + \dots (k + 1) \text{ times} = cn(k + 1)$
 $= cn \times \log_{\frac{1}{1 - \alpha}}(cn)$

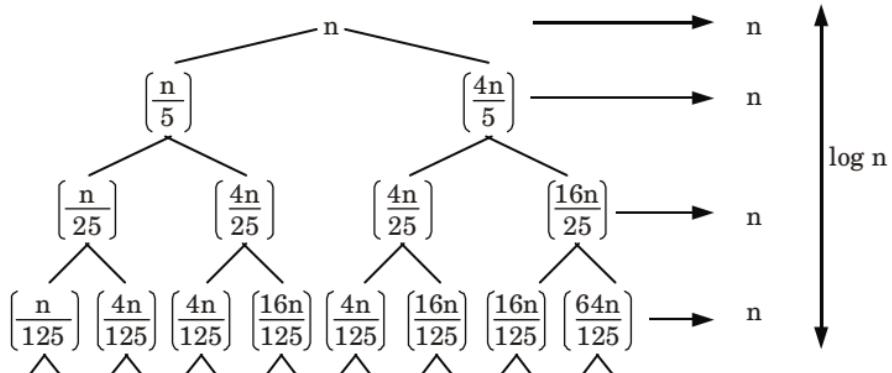
$$\text{Time complexity} = O\left(n \log_{\frac{1}{1 - \alpha}}(cn)\right) = O\left(n \log_{\frac{1}{1 - \alpha}} n\right)$$

Que 1.15. Solve the following by recursion tree method

$$T(n) = n + T(n/5) + T(4n/5)$$

AKTU 2017-18, Marks 10**Answer**

$$T(n) = n + n + n + \dots + \log n \text{ times} = \Omega(n \log n)$$



PART-3

Sorting and Order Statistic : Shell Sort, Quick Sort.

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 1.16. Explain shell sort with example.

Answer

1. Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm and we can code it easily.
2. It roughly sorts the data first, moving large elements towards one end and small elements towards the other.
3. In shell sort several passes over the data is performed.
4. After the final pass, the data is fully sorted.
5. The shell sort does not sort the data itself; it increases the efficiency of other sorting algorithms.

Algorithm :

Input : An array a of length n with array elements numbered 0 to $n - 1$.

1. $\text{inc} \leftarrow \text{round}(n/2)$
2. while $\text{inc} > 0$
3. for $i = \text{inc}$ to $n - 1$
 - $\text{temp} \leftarrow a[i]$
 - $j \leftarrow i$
 - while $j \geq \text{inc}$ and $a[j - \text{inc}] > \text{temp}$
 - $a[j] \leftarrow a[j - \text{inc}]$
 - $j \leftarrow j - \text{inc}$
 - $a[j] \leftarrow \text{temp}$
4. $\text{inc} \leftarrow \text{round}(\text{inc}/2.2)$

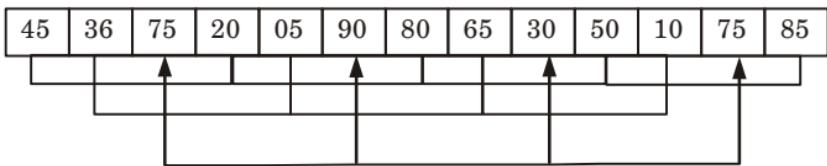
For example :

45	36	75	20	05	90	80	65	30	50	10	75	85
----	----	----	----	----	----	----	----	----	----	----	----	----

The distance between the elements to be compared is 3. The subfiles generated with the distance of 3 are as follows :

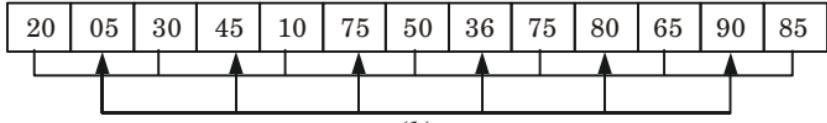
Subfile 1	$a[0]$	$a[3]$	$a[6]$	$a[9]$	$a[12]$
Subfile 2	$a[1]$	$a[4]$	$a[7]$	$a[10]$	
Subfile 3	$a[2]$	$a[5]$	$a[8]$	$a[11]$	

Input to pass 1 with distance = 3



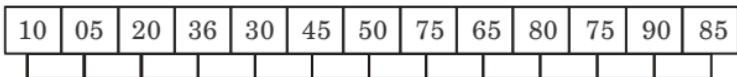
(a)

Output of pass 1 is input to pass 2 and distance = 2



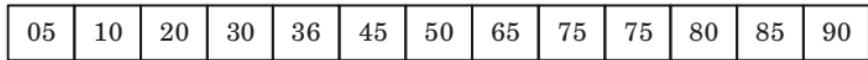
(b)

Output of pass 2 is input to pass 3 and distance = 1



(c)

Output of pass 3



(d)

Fig. 1.16.1.

Que 1.17. Describe any one of the following sorting techniques :

- Selection sort
- Insertion sort

Answer

- Selection sort (A) :
 - $n \leftarrow \text{length}[A]$
 - for $j \leftarrow 1$ to $n-1$
 - $\text{smallest} \leftarrow j$
 - for $i \leftarrow j+1$ to n
 - if $A[i] < A[\text{smallest}]$
 - then $\text{smallest} \leftarrow i$
 - exchange ($A[j], A[\text{smallest}]$)

- ii. Insertion_Sort(A) :
 - for $j \leftarrow 2$ to $\text{length}[A]$
 - do $\text{key} \leftarrow A[j]$
 - Insert $A[j]$ into the sorted sequence $A[1 \dots j-1]$
 - $i \leftarrow j-1$
 - while $i > 0$ and $A[i] > \text{key}$
 - do $A[i+1] \leftarrow A[i]$
 - $i \leftarrow i-1$
 - $A[i+1] \leftarrow \text{key}$

Que 1.18. Write non-deterministic algorithm for sorting.

AKTU 2016-17, Marks 10

Answer

Non-deterministic algorithms are algorithm that, even for the same input, can exhibit different behaviours on different runs, iterations and executions.

N SORT(A, B) :

1. for $i = 1$ to n do
2. $j = \text{choice}(1 \dots n)$
3. if $B[j] \neq 0$ then failure
4. $B[j] = A[i]$
5. endfor
6. for $i = 1$ to $n - 1$ do
7. if $B[i] < B[i + 1]$ then failure
8. endfor
9. print(B)
10. success

Que 1.19. Explain the concepts of quick sort method and analyze

its complexity with suitable example.

AKTU 2016-17, Marks 10

Answer

Quick sort :

Quick sort works by partitioning a given array $A[p \dots r]$ into two non-empty subarray $A[p \dots q - 1]$ and $A[q + 1 \dots r]$ such that every key in $A[p \dots q - 1]$ is less than or equal to every key in $A[q + 1 \dots r]$. Then the two subarrays are sorted by recursive calls to quick sort.

Quick_Sort (A, p, r)

1. If $p < r$ then
2. $q \leftarrow \text{Partition} (A, p, r)$
3. Recursive call to Quick_Sort ($A, p, q - 1$)
4. Recursive call to Quick_Sort ($A, q + 1, r$)

As a first step, Quick sort chooses as pivot one of the items in the array to be sorted. Then array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partition (A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. for $j \leftarrow p$ to $r - 1$
4. do if $A[j] \leq x$
5. then $i \leftarrow i + 1$
6. then exchange $A[i] \leftrightarrow A[j]$

7. exchange $A[i + 1] \leftrightarrow A[r]$

8. return $i + 1$

Example : Given array to be sorted

3	1	4	1	5	9	2	6	5	3	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---

Sort the array A using quick sort algorithm.

Step 1 : The array is Pivoted about its first element i.e., Pivot (P) = 3

3		1	4	1	5	9	2	6	5	3	5	8	9
	P												

Step 2 : Find first element larger than pivot (make underline) and find element not larger than pivot from end make overline.

3		1	<u>4</u>	1	5	9	2	6	5	<u>3</u>	5	8	9
	P		Underline							Overline			

Step 3 : Swap these elements and scan again.

3		1	3	1	5	9	2	6	5	4	5	8	9
	P									↓			

Array after swapping

3		1	3	1	<u>5</u>	9	<u>2</u>	6	5	4	5	8	9
	P		Underline		Overline								

Apply swapping,

3		1	3	1	2	9	5	6	5	4	5	8	9
	P												

Again apply scanning,

3		1	3	1	<u>2</u>	<u>9</u>	5	6	5	4	5	8	9
	P		Overline	Underline									

The pointers have crossed

i.e., overline on left of underlined

Then, in this situation swap pivot with overline.

2	1	3	1		3		9	5	6	5	4	5	8	9
				P										

Now, pivoting process is complete.

Step 4 : Recursively sort subarrays on each side of pivot.

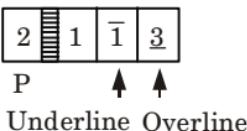
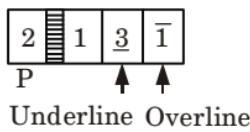
Subarray 1 :

2	1	3	1
---	---	---	---

Subarray 2 :

9	5	6	5	1	5	8	9
---	---	---	---	---	---	---	---

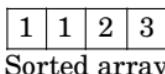
First apply Quick sort for subarray 1.



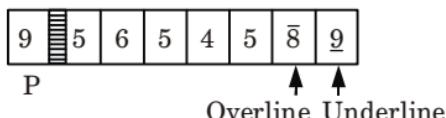
The pointers have crossed.

i.e., overline on left of underlined.

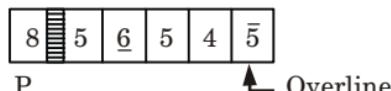
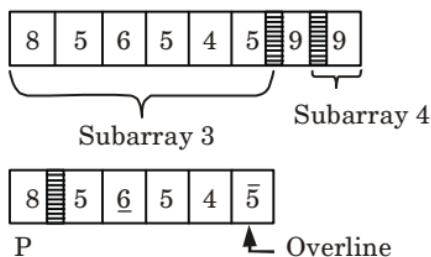
Swap pivot with overline



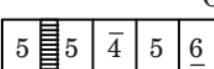
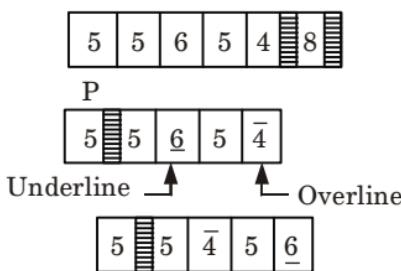
Now, for subarray 2 we apply Quick sort procedure.



The pointer has crossed. Then swap pivot with overline.



Swap overline with pivot.

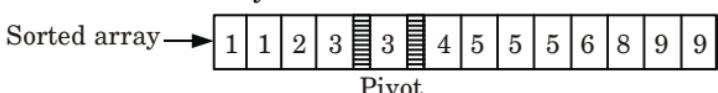


Overline on left of underlined.

Swap pivot with overline.



Now combine all the subarrays



Analysis of complexity :**i. Worst case :**

- Let $T(n)$ be the worst case time for quick sort on input size n . We have a recurrence

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n) \quad \dots(1.19.1)$$

where q ranges from 0 to $n - 1$, since the partition produces two regions, each having size $n - 1$.

- Now we assume that $T(n) \leq cn^2$ for some constant c .

Substituting our assumption in eq. (1.19.1) we get

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n) \end{aligned}$$

- Since the second derivative of expression $q^2 + (n - q - 1)^2$ with respect to q is positive. Therefore, expression achieves a maximum over the range $0 \leq q \leq n - 1$ at one of the endpoints.
- This gives the bound
 $\max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) \leq (n - 1)^2 = n^2 - 2n + 1$
- Continuing with the bounding of $T(n)$ we get
 $T(n) \leq cn^2 - c(2n - 1) + \Theta(n) \leq cn^2$
- Since we can pick the constant c large enough so that the $c(2n - 1)$ term dominates the $\Theta(n)$ term. We have
 $T(n) = O(n^2)$
- Thus, the worst case running time of quick sort is $\Theta(n^2)$.

ii. Average case :

- If the split induced of RANDOMIZED_PARTITION puts constant fraction of elements on one side of the partition, then the recurrence tree has depth $\Theta(\log n)$ and $\Theta(n)$ work is performed at each level.
- This is an intuitive argument why the average case running time of RANDOMIZED_QUICKSORT is $\Theta(n \log n)$.
- Let $T(n)$ denotes the average time required to sort an array of n elements. A call to RANDOMIZED_QUICKSORT with a 1 element array takes a constant time, so we have $T(1) = \Theta(1)$.
- After the split RANDOMIZED_QUICKSORT calls itself to sort two subarrays.
- The average time to sort an array $A[1 .. q]$ is $T[q]$ and the average time to sort an array $A[q+1 .. n]$ is $T[n - q]$. We have

$$T(n) = 1/n (T(1) + T(n - 1) + \sum_{q=1}^{n-1} S_{q-1} T(q) T(n - q)) + \Theta(n) \quad \dots(1.19.1)$$

We know from worst-case analysis

$$T(1) = \Theta(1) \text{ and } T(n - 1) = O(n^2)$$

$$T(n) = 1/n (\Theta(1) + O(n^2)) + 1/n \sum_{q=1}^{n-1} \Theta_q(r(q) + T(n - q)) + Q(n) \quad \dots(1.19.2)$$

$$= 1/n \sum_{q=1}^{n-1} (T(q) + T(n - q)) + Q(n) \quad \dots(1.19.2)$$

$$= 1/n [2 \sum_{k=1}^{n-1} T(k)] + \Theta(n)$$

$$= 2/n \sum_{k=1}^{n-1} T(k) + \Theta(n) \quad \dots(1.19.3)$$

6. Solve the above recurrence using substitution method. Assume that $T(n) \leq an n \log n + b$ for some constants $a > 0$ and $b > 0$.

If we can pick 'a' and 'b' large enough so that $n \log n + b > T(1)$. Then for $n > 1$, we have

$$\begin{aligned} T(n) &\geq \sum_{k=1}^{n-1} 2/n (ak \log k + b) + \Theta(n) \\ &= 2a/n \sum_{k=1}^{n-1} k \log k - 1/8(n^2) + 2b/n \\ &\quad (n-1) + \Theta(n) \end{aligned} \quad \dots(1.19.4)$$

At this point we are claiming that

$$\sum_{k=1}^{n-1} k \log k \leq 1/2 n^2 \log n - 1/8(n^2)$$

Substituting this claim in the eq. (1.19.4), we get

$$\begin{aligned} T(n) &\leq 2a/n [1/2n^2 \log n - 1/8(n^2)] + 2/n b(n-1) + \Theta(n) \\ &\leq an \log n - an/4 + 2b + \Theta(n) \end{aligned} \quad \dots(1.19.5)$$

In the eq. (1.19.5), $\Theta(n) + b$ and $an/4$ are polynomials and we can choose 'a' large enough so that $an/4$ dominates $\Theta(n) + b$.

We conclude that QUICKSORT's average running time is $\Theta(n \log n)$.

Que 1.20. Discuss the best case and worst case complexities of quick sort algorithm in detail.

Answer

Best case :

1. The best thing that could happen in quick sort would be that each partitioning stage divides the array exactly in half.
2. In other words, the best to be a median of the keys in $A[p .. r]$ every time procedure 'Partition' is called.
3. The procedure 'Partition' always split the array to be sorted into two equal sized arrays.
4. If the procedure 'Partition' produces two regions of size $n/2$, then the recurrence relation is :

$$T(n) \leq T(n/2) + T(n/2) + \Theta(n) \leq 2T(n/2) + \Theta(n)$$

And from case (2) of master theorem

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

Worst case : Refer Q. 1.19, Page 1-16B, Unit-1.

PART-4

Merge Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.21. Explain the concept of merge sort with example.

Answer

1. Merge sort is a sorting algorithm that uses the idea of divide and conquer.
2. This algorithm divides the array into two halves, sorts them separately and then merges them.
3. This procedure is recursive, with the base criteria that the number of elements in the array is not more than 1.

Algorithm :**MERGE_SORT (a, p, r)**

1. if $p < r$
2. then $q \leftarrow \lfloor (p + r)/2 \rfloor$
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. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Create arrays $L [1 \dots n_1 + 1]$ and $R [1 \dots n_2 + 1]$
4. for $i = 1$ to n_1
do
 $L[i] = A [p + i - 1]$
endfor
5. for $j = 1$ to n_2
do
 $R[j] = A[q + j]$
endfor
6. $L[n_1 + 1] = \infty, R[n_2 + 1] = \infty$
7. $i = 1, j = 1$
8. for $k = p$ to r
do
if $L[i] \leq R[j]$
then $A[k] \leftarrow L[i]$
 $i = i + 1$
else $A[k] = R[j]$
 $j = j + 1$
endif
endfor

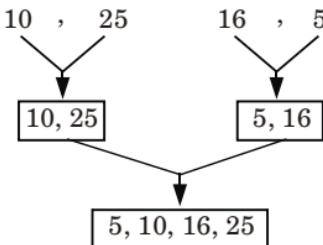
9. exit

Example :

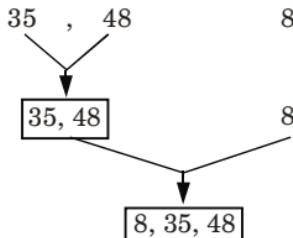
10, 25, 16, 5, 35, 48, 8

1. **Divide into two halves :**

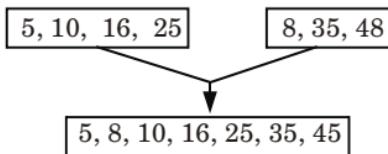
10, 25, 16, 5	35, 48, 8
---------------	-----------
2. **Consider the first part :** 10, 25, 16, 5 again divide into two sub-arrays



3. Consider the second half: 35, 48, 5 again divide into two sub-arrays



4. Merge these two sorted sub-arrays,



This is the sorted array.

Que 1.22. | Determine the best case time complexity of merge sort algorithm.

Answer

1. The best case of merge sort occurs when the largest element of one array is smaller than any element in the other array.
2. For this case only $n/2$ comparisons of array elements are made.
3. Merge sort comparisons are obtained by the recurrence equation of the recursive calls used in merge sort.
4. As it divides the array into half so the recurrence function is defined as :

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = 2T\left(\frac{n}{2}\right) + n \quad \dots(1.22.1)$$

5. By using variable k to indicate depth of the recursion, we get

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn \quad \dots(1.22.2)$$

6. For the best case there are only $n/2$ comparisons hence equation (1.22.2) can be written as

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

7. At the last level of recursion tree

$$2^k = n$$

$$k = \log_2 n$$

8. So the recurrence function is defined as :

$$T(n) = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + \frac{n}{2} \log_2 n$$

$$= nT(1) + \frac{n}{2} \log_2 n = \frac{n}{2} \log_2 n + n$$

$$T(n) = O(n \log_2 n)$$

Hence, the best case complexity of merge sort is $O(n \log_2 n)$.

PART-5

Heap Sort.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.23. Explain heap sort algorithm with its complexity.

OR

Discuss Max-Heapify and Build-Max-Heap procedures.

Answer

1. Heap sort is a comparison based sorting technique based on binary heap data structure.
2. Heap sort finds the largest element and puts it at the end of array, then the second largest item is found and this process is repeated for all other elements.
3. The general approach of heap sort is as follows :
 - a. From the given array, build the initial max heap.
 - b. Interchange the root (maximum) element with the last element.
 - c. Use repetitive downward operation from root node to rebuild the heap of size one less than the starting.
 - d. Repeat step a and b until there are no more elements.

Analysis of heap sort :

Complexity of heap sort for all cases is $O(n \log_2 n)$.

MAX-HEAPIFY (A, i) :

1. $i \leftarrow \text{left}[i]$
2. $r \leftarrow \text{right}[i]$
3. if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$
4. then $\text{largest} \leftarrow l$
5. else $\text{largest} \leftarrow i$
6. if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$
7. then $\text{largest} \leftarrow r$

8. if largest $\neq i$
9. then exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY [A, largest]

HEAP-SORT(A) :

1. BUILD-MAX-HEAP (A)
2. for $i \leftarrow \text{length}[A]$ down to 2
3. do exchange $A[1] \leftrightarrow A[i]$
4. heap-size [A] $\leftarrow \text{heap-size}[A] - 1$
5. MAX-HEAPIFY (A, 1)

BUILD-MAX-HEAP (A)

1. heap-size (A) $\leftarrow \text{length}[A]$
2. for $i \leftarrow (\text{length}[A]/2)$ down to 1 do
3. MAX-HEAPIFY (A, i)

We can build a heap from an unordered array in linear time.

Average case and worst case complexity :

1. We have seen that the running time of BUILD-HEAP is $O(n)$.
2. The heap sort algorithm makes a call to BUILD-HEAP for creating a (max) heap, which will take $O(n)$ time and each of the $(n - 1)$ calls to MAX-HEAPIFY to fix up the new heap (which is created after exchanging the root and by decreasing the heap size).
3. We know 'MAX-HEAPIFY' takes time $O(\log n)$.
4. Thus the total running time for the heap sort is $O(n \log n)$.

Que 1.24. How will you sort following array A of element using heap sort : $A = (23, 9, 18, 45, 5, 9, 1, 17, 6)$.

AKTU 2018-19, Marks 10

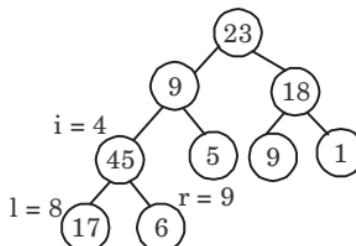
Answer

Given array :

23	9	18	45	5	9	1	17	6
----	---	----	----	---	---	---	----	---

First we call Build-Max heap

heap size [A] = 9



so $i = 4$ to 1 call MAX HEAPIFY (A, i)

i.e., first we call MAX HEAPIFY (A, 4)

$$A[l] = 7, A[i] = A[4] = 45, A[r] = 6$$

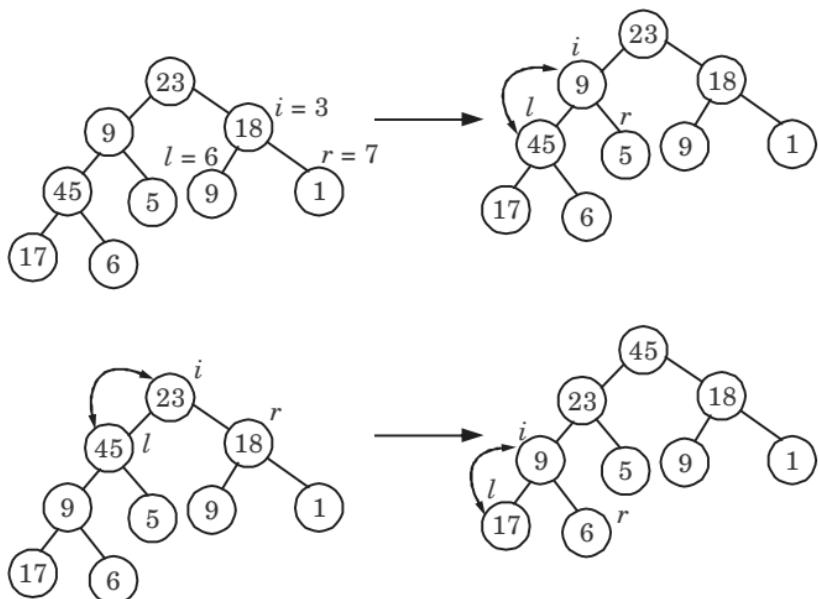
$$l \leftarrow \text{left}[4] = 2 \times 4 = 8$$

$$r \leftarrow \text{right}[4] = 2 \times 4 + 1 = 9$$

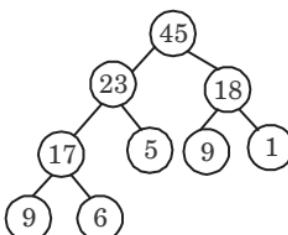
$$8 \leq 9 \text{ and } A[8] = 17 < 45 \text{ (False)}$$

Then, largest $\leftarrow 4$.

Similarly for $i = 3, 2, 1$ we get the following heap tree :

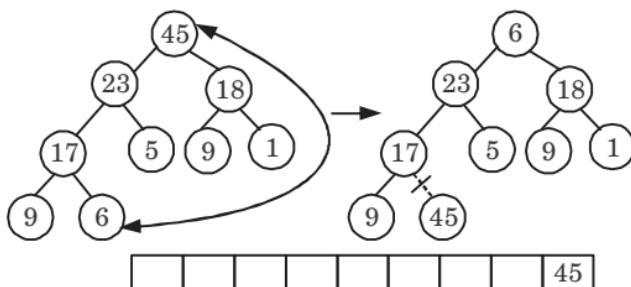


So, final tree after Build-Max heap is

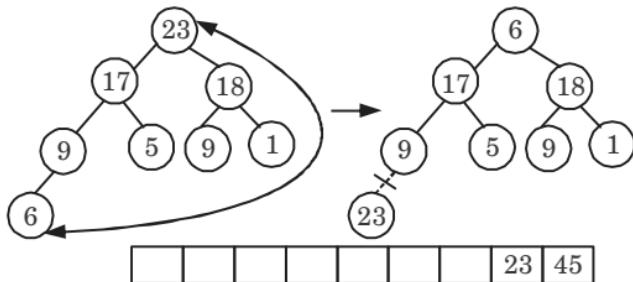


Now $i = 9$ down to 2 and size $= 10 - 1 = 9$ and call MAX HEAPIFY ($A, 1$) each time

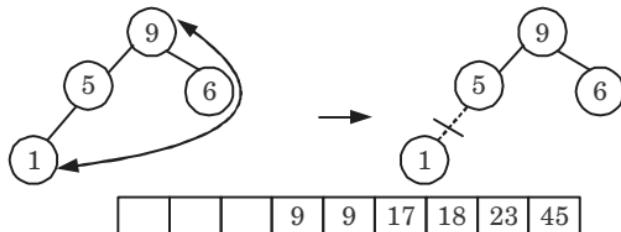
Exchanging $A[1] \leftrightarrow A[9]$



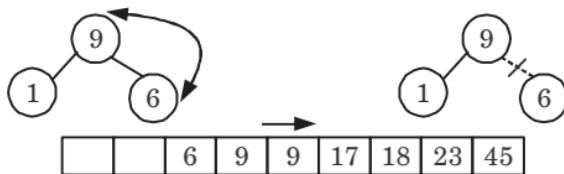
Now call MAX HEAPIFY ($A, 1$) and
Exchange $A[1]$ and $A[8]$, size $= 9 - 1 = 8$



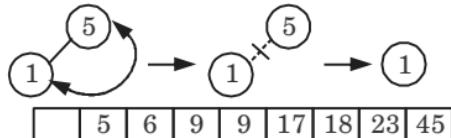
Now, call MAX HEAPIFY ($A, 1$), exchange $A[1]$ and $A[4]$ and size = $5 - 1 = 4$



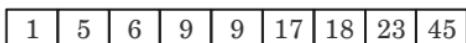
Now, call MAX HEAPIFY ($A, 1$), exchange $A[1]$ and $A[3]$ and size = $4 - 1 = 3$



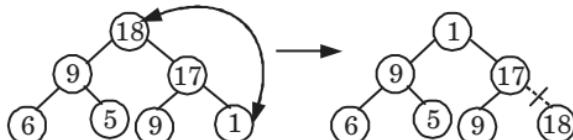
Exchange $A[1]$ and $A[2]$ size $3 - 1 = 2$



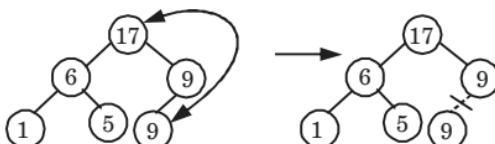
The sorted array



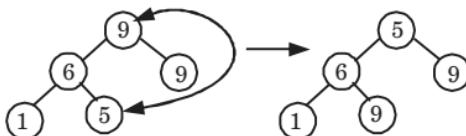
Now call MAX HEAPIFY ($A, 1$) and exchange $A[1]$ and $A[7]$



Now call MAX HEAPIFY ($A, 1$) and size = $7 - 1 = 6$ exchange $A[1]$ and $A[6]$



Exchange $A[1]$ and $A[5]$ and size = $6 - 1 = 5$



Que 1.25. What is heap sort ? Apply heap sort algorithm for sorting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Also deduce time complexity of heap sort.

AKTU 2015-16, Marks 10

Answer

Heap sort and its time complexity : Refer Q. 1.23, Page 1-23B, Unit-1.

Numerical : Since the given problem is already in sorted form. So, there is no need to apply any procedure on given problem.

Que 1.26. Explain HEAP SORT on the array. Illustrate the operation HEAP SORT on the array $A = \{6, 14, 3, 25, 2, 10, 20, 7, 6\}$

AKTU 2017-18, Marks 10

Answer

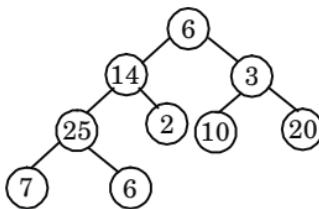
Heap sort : Refer Q. 1.23, Page 1-23B, Unit-1.

Numerical :

Originally the given array is : [6, 14, 3, 25, 2, 10, 20, 7, 6]

First we call BUILD-MAX-HEAP

heap size $[A] = 9$



so, $i = 4$ to 1, call MAX-HEAPIFY (A, i)

i.e., first we call MAX-HEAPIFY ($A, 4$)

$$Az [l] = 7, A [i] = A [4] = 25, A [r] = 6$$

$$l \leftarrow \text{left}[4] = 8$$

$$r \leftarrow \text{right}[4] = 9$$

$8 \leq 9$ and $7 > 25$ (False)

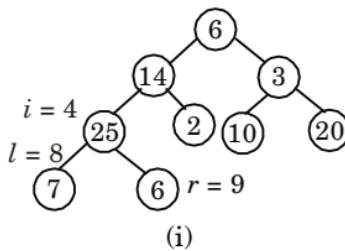
Then, largest $\leftarrow 4$

$9 \leq 9$ and $6 > 25$ (False)

Then, largest = 4

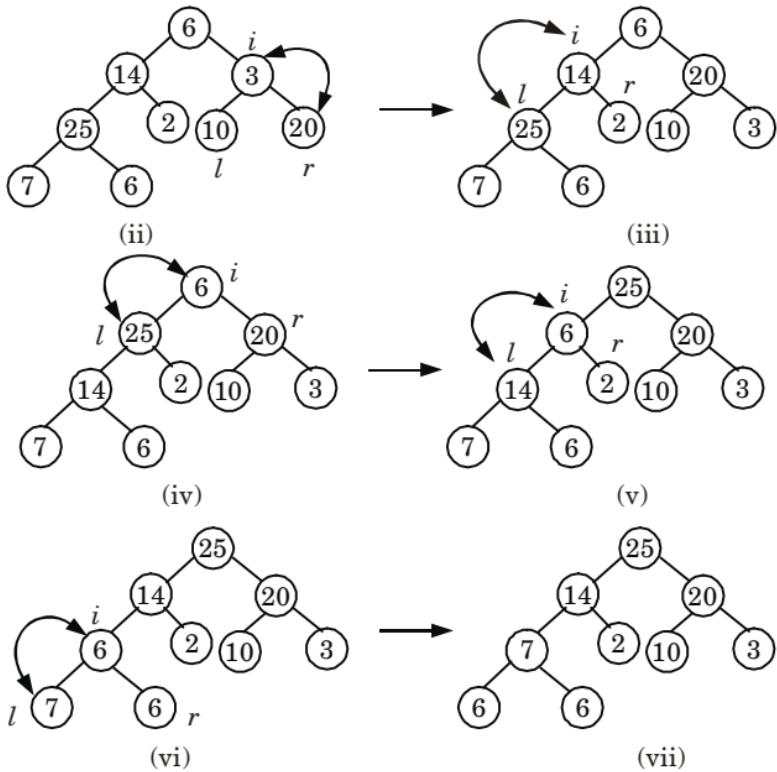
$A[i] \leftrightarrow A[4]$

Now call MAX-HEAPIFY ($A, 2$)

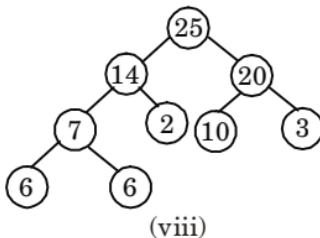


(i)

Similarly for $i = 3, 2, 1$ we get the following heap tree.

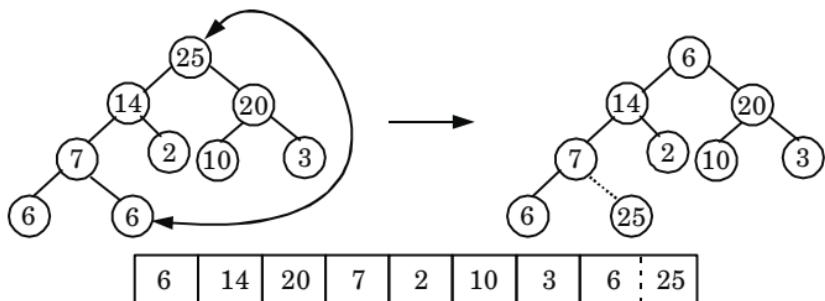


So final tree after BUILD-MAX-HEAP is



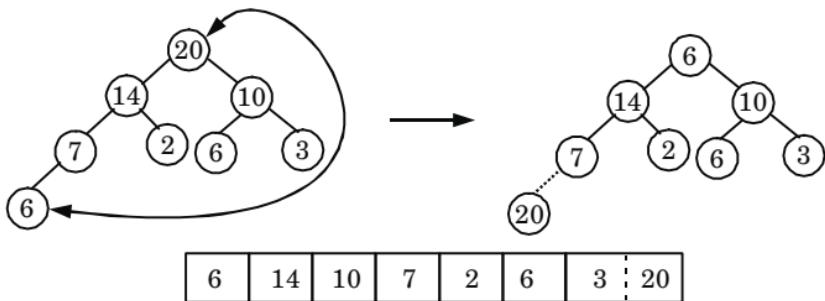
Now $i = 9$ down to 2, and size = size - 1 and call MAX-HEAPIFY($A, 1$) each time.

exchanging $A[1] \leftrightarrow A[9]$



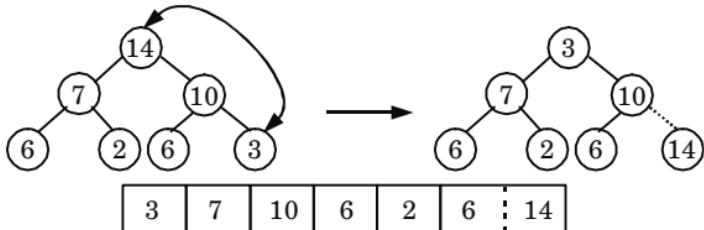
Now call MAX-HEAPIFY ($A, 1$) we get

Now exchange $A[1]$ and $A[8]$ and size = $8 - 1 = 7$



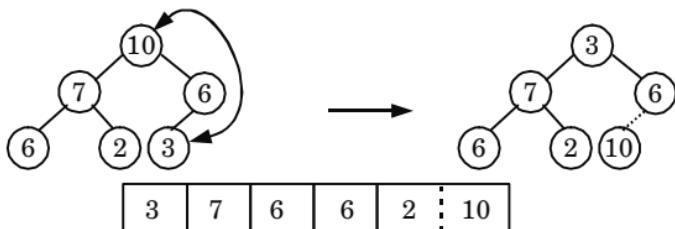
Again call MAX-HEAPIFY ($A, 1$), we get

exchange $A[1]$ and $A[7]$ and size = $7 - 1 = 6$



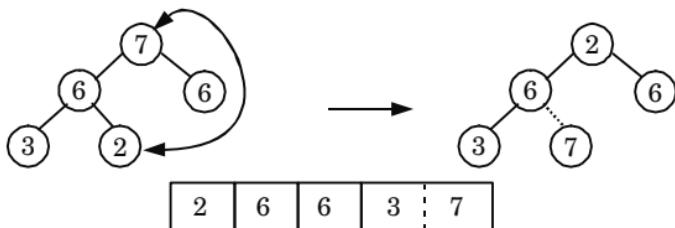
Again call MAX-HEAPIFY ($A, 1$), we get

exchange $A[1]$ and $A[6]$ and now size = $6 - 1 = 5$

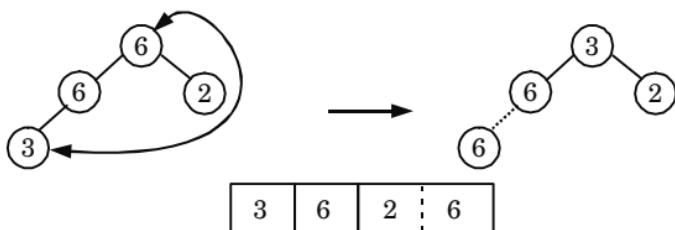


Again call MAX-HEAPIFY ($A, 1$)

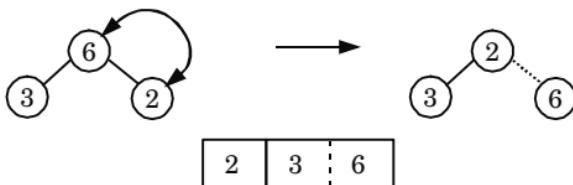
exchange $A[1]$ and $A[5]$ and now size = $5 - 1 = 4$



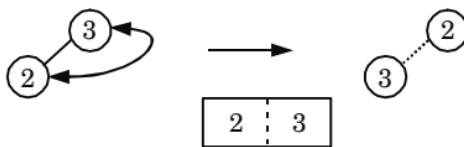
Again, call MAX-HEAPIFY ($A, 1$)
 exchange $A[1]$ and $A[4]$ and size = $4 - 1 = 3$



call MAX-HEAPIFY ($A, 1$)
 exchange $A[1]$ and $A[3]$, size = $3 - 1 = 2$



call MAX-HEAPIFY ($A, 1$)
 exchange $A[1]$ and $A[2]$ and size = $2 - 1 = 1$



Thus, sorted array :

2	3	6	6	7	10	14	20	25
---	---	---	---	---	----	----	----	----

PART-6

Comparison of Sorting Algorithms, Sorting in Linear Time.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 1.27. How will you compare various sorting algorithms ?

Answer

Name	Average case	Worst case	Stable	Method	Other notes
Selection sort	$O(n^2)$	$O(n^2)$	No	Selection	Can be implemented as a stable sort
Insertion sort	$O(n^2)$	$O(n^2)$	Yes	Insertion	Average case is also $O(n + d)$, where d is the number of inversion
Shell sort	-	$O(n \log^2 n)$	No	Insertion	No extra memory required
Merge sort	$O(n \log n)$	$O(n \log n)$	Yes	Merging	Recursive, extra memory required
Heap sort	$O(n \log n)$	$O(n \log n)$	No	Selection	Recursive, extra memory required
Quick sort	$O(n \log n)$	$O(n^2)$	No	Partitioning	Recursive, based on divide conquer technique

Que 1.28. Explain the counting sort algorithm.**Answer**

Counting sort is a linear time sorting algorithm used to sort items when they belong to a fixed and finite set.

Algorithm :

Counting_Sort(A, B, k)

1. let $C[0..k]$ be a new array
2. for $i \leftarrow 0$ to k
3. do $C[i] \leftarrow 0$
4. for $j \leftarrow 1$ to $\text{length}[A]$
5. do $C[A[j]] \leftarrow C[A[j]] + 1$
 // $C[i]$ now contains the number of elements equal to i .
6. for $i \leftarrow 1$ to k
7. do $C[i] \leftarrow C[i] + C[i - 1]$
 // $C[i]$ now contains the number of elements less than or equal to i .
8. for $j \leftarrow \text{length}[A]$ down to 1
9. do $B[C[A[j]]] \leftarrow A[j]$
10. $C[A[j]] \leftarrow C[A[j]] - 1$

Que 1.29. What is the time complexity of counting sort ? Illustrate the operation of counting sort on array $A = \{1, 6, 3, 3, 4, 5, 6, 3, 4, 5\}$.

Answer

Given array : Time complexity of counting sort is $O(n)$.

1 2 3 4 5 6 7 8 9 10

A	1	6	3	3	4	5	6	3	4	5
---	---	---	---	---	---	---	---	---	---	---

Step 1 : $i = 0$ to 6 $k = 6$ (largest element in array A)

$C[i] \leftarrow 0$

0 1 2 3 4 5 6

C	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Step 2 : $j = 1$ to 10 (\because length $[A] = 10$)

$C[A[j]] \leftarrow C[A[j]] + 1$

For $j = 1$

0 1 2 3 4 5 6

$C[A[1]] \leftarrow C[1] + 1 = 0 + 1 = 1$ C 0 1 0 0 0 0 0

$C[1] \leftarrow 1$

For $j = 2$

0 1 2 3 4 5 6

$C[A[2]] \leftarrow C[6] + 1 = 0 + 1 = 1$ C 0 1 0 0 0 0 1

$C[6] \leftarrow 1$

0 1 2 3 4 5 6

Similarly for $j = 3, 4, 5, 6, 7, 8, 9, 10$ C 0 1 0 3 2 2 2

Step 3 :

For $i = 1$ to 6

$C[i] \leftarrow C[i] + C[i - 1]$

For $i = 1$

0 1 2 3 4 5 6

$C[1] \leftarrow C[1] + C[0]$ C 0 1 0 3 2 2 2

$C[1] \leftarrow 1 + 0 = 1$

For $i = 2$

0 1 2 3 4 5 6

$C[2] \leftarrow C[2] + C[1]$ C 0 1 1 3 2 2 2

$C[1] \leftarrow 1 + 0 = 1$

0 1 2 3 4 5 6

Similarly for $i = 3, 4, 5, 6$ C 0 1 1 4 6 8 10

Step 4 :

For $j = 10$ to 1

$B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j] - 1]$

<i>j</i>	<i>A[j]</i>	<i>C[A[j]]</i>	<i>B[C[A[j]]] ← A[j]</i>	<i>C[A[j]] ← C[A[j]] - 1</i>
10	5	8	<i>B</i> [8] ← 5	<i>C</i> [5] ← 7
9	4	6	<i>B</i> [6] ← 4	<i>C</i> [4] ← 5
8	3	4	<i>B</i> [4] ← 3	<i>C</i> [3] ← 3
7	6	10	<i>B</i> [10] ← 6	<i>C</i> [6] ← 9
6	5	7	<i>B</i> [7] ← 5	<i>C</i> [5] ← 6
5	4	5	<i>B</i> [5] ← 4	<i>C</i> [4] ← 4
4	3	3	<i>B</i> [3] ← 3	<i>C</i> [3] ← 2
3	3	2	<i>B</i> [2] ← 3	<i>C</i> [3] ← 1
2	6	9	<i>B</i> [9] ← 6	<i>C</i> [6] ← 8
1	1	1	<i>B</i> [1] ← 1	<i>C</i> [1] ← 0

1 2 3 4 5 6 7 8 9 10
B

1	3	3	3	4	4	4	5	5	6	6
---	---	---	---	---	---	---	---	---	---	---

Que 1.30. Write the bucket sort algorithm.

Answer

1. The bucket sort is used to divide the interval $[0, 1]$ into n equal-sized sub-intervals, or bucket, and then distribute the n -input numbers into the bucket.
2. Since the inputs are uniformly distributed over $[0, 1]$, we do not expect many numbers to fall into each bucket.
3. To produce the output, simply sort the numbers in each bucket and then go through the bucket in order, listing the elements in each.
4. The code assumes that input is in n -element array A and each element in A satisfies $0 \leq A[i] \leq 1$. We also need an auxiliary array $B[0 \dots n - 1]$ for linked-list (buckets).

BUCKET_SORT (A)

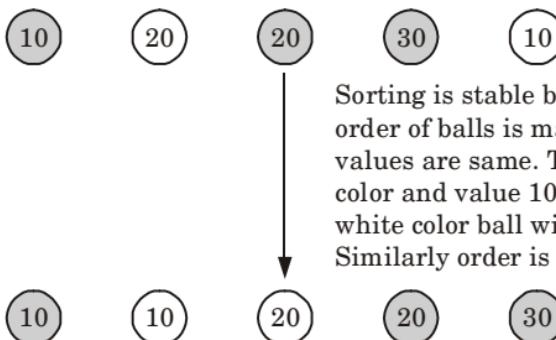
1. $n \leftarrow \text{length } [A]$
2. for $i \leftarrow 1$ to n
3. do Insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. for $i \leftarrow 0$ to $n - 1$
5. do Sort list $B[i]$ with insertion sort
6. Concatenate the lists $B[0], B[1], \dots, B[n - 1]$ together in order.

Que 1.31. What do you mean by stable sort algorithms ? Explain it with suitable example.

Answer

1. A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input sorted array.
2. A stable sort is one where the initial order of equal items is preserved.
3. Some sorting algorithms are stable by nature, such as bubble sort, insertion sort, merge sort, counting sort etc.
4. Let A be an array, and let $<$ be a strict weak ordering on the elements of A . Sorting algorithm is stable if :
 $i < j$ and $A[i] \equiv A[j]$ i.e., $A[i]$ comes before $A[j]$.
5. Stability means that equivalent elements retain their relative positions, after sorting.

For example :



Que 1.32. Write a short note on radix sort.

Answer

1. Radix sort is a sorting algorithm which consists of list of integers or words and each has d -digit.
2. We can start sorting on the least significant digit or on the most significant digit.
3. On the first pass entire numbers sort on the least significant digit (or most significant digit) and combine in a array.
4. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combine in an array and so on.

RADIX_SORT (A, d)

1. for $i \leftarrow 1$ to d do
2. use a stable sort to sort array A on digit i
// counting sort will do the job

The code for radix sort assumes that each element in the n -element array A has d -digits, where digit 1 is the lowest-order digit and d is the highest-order digit.

Analysis :

1. The running time depends on the table used as an intermediate sorting algorithm.

2. When each digit is in the range 1 to k , and k is not too large, COUNTING_SORT is the obvious choice.
3. In case of counting sort, each pass over n d -digit numbers takes $\Theta(n + k)$ time.
4. There are d passes, so the total time for radix sort is $\Theta(n + k)$ time. There are d passes, so the total time for radix sort is $\Theta(dn + kd)$. When d is constant and $k = \Theta(n)$, the radix sort runs in linear time.

For example : This example shows how radix sort operates on seven 3-digit number.

Table 1.32.1.

Input	1 st pass	2 nd pass	3 rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the table 1.32.1, the first column is the input and the remaining shows the list after successive sorts on increasingly significant digits position.

Que 1.33. Among Merge sort, Insertion sort and quick sort which sorting technique is the best in worst case. Apply the best one among these algorithms to sort the list E, X, A, M, P, L, E in alphabetic order.

AKTU 2019-20, Marks 07

Answer

Merge sort technique is best in worst case because of its time complexity $O(n \log n)$.

Numerical :

Given : E, X, A, M, P, L, E

Pass 1 : Merge each pair of element to obtain sorted list :

E	X	A	M
---	---	---	---

P	L	E
---	---	---

After sorting each pair, we get

E	X	A	M	L	P	E
---	---	---	---	---	---	---

Pass 2 : Merge each pair to obtain the list :

A	E	M	X	E	L	P
---	---	---	---	---	---	---

Pass 3 : Again merge the two sub arrays to obtain the list :

A	E	E	L	M	P	X
---	---	---	---	---	---	---

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What do you mean by algorithm ? Write its characteristics.

Ans. Refer Q. 1.1.

Q. 2. Write short note on asymptotic notations.

Ans. Refer Q. 1.3.

Q. 3. i. Solve the recurrence $T(n) = 2T(n/2) + n^2 + 2n + 1$

ii. Prove that worst case running time of any comparison sort is $\Omega(n \log n)$.

Ans. Refer Q. 1.9.

Q. 4. Explain shell sort with example.

Ans. Refer Q. 1.16.

Q. 5. Discuss quick sort method and analyze its complexity.

Ans. Refer Q. 1.19.

Q. 6. Explain the concept of merge sort with example.

Ans. Refer Q. 1.21.

Q. 7. Write short note on heap sort algorithm with its analysis.

Ans. Refer Q. 1.23.

Q. 8. Explain HEAP SORT on the array. Illustrate the operation HEAP SORT on the array $A = \{6, 14, 3, 25, 2, 10, 20, 7, 6\}$

Ans. Refer Q. 1.26.



2**UNIT**

Advanced Data Structure

CONTENTS

Part-1 : Red-Black Trees	2-2B to 2-19B
Part-2 : B-Trees	2-19B to 2-33B
Part-3 : Binomial Heaps	2-33B to 2-44B
Part-4 : Fibonacci Heaps	2-44B to 2-48B
Part-5 : Tries, Skip List	2-48B to 2-51B

PART - 1*Red-Black Trees.***Questions-Answers****Long Answer Type and Medium Answer Type Questions**

Que 2.1. Define a red-black tree with its properties. Explain the insertion operation in a red-black tree.

Answer**Red-black tree :**

A red-black tree is a binary tree where each node has colour as an extra attribute, either red or black. It is a self-balancing Binary Search Tree (BST) where every node follows following properties :

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendent leave contain the same number of black nodes.

Insertion :

- i. We begin by adding the node as we do in a simple binary search tree and colouring it red.

RB-INSERT(T, z)

1. $y \leftarrow \text{nil}[T]$
2. $x \leftarrow \text{root}[T]$
3. while $x \neq \text{nil}[T]$
 4. do $y \leftarrow x$
 5. if $\text{key}[z] < \text{key}[x]$
 6. then $x \leftarrow \text{left}[x]$
 7. else $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. if $y = \text{nil}[T]$
10. then $\text{root}[T] \leftarrow z$
11. else if $\text{key}[z] < \text{key}[y]$
12. then $\text{left}[y] \leftarrow z$
13. else $\text{right}[y] \leftarrow z$
14. $\text{left}[z] \leftarrow \text{nil}[T]$
15. $\text{right}[z] \leftarrow \text{nil}[T]$
16. $\text{colour}[z] \leftarrow \text{RED}$
17. RB-INSERT-FIXUP(T, z)

ii. Now, for any colour violation, RB-INSERT-FIXUP procedure is used.

RB-INSERT-FIXUP(T, z)

1. while colour [$p[z]$] = RED
2. do if $p[z] = \text{left}[p[p[z]]]$
3. then $y \leftarrow \text{right}[p[p[z]]]$
4. if colour[y] = RED
5. then colour[p[z]] \leftarrow BLACK ⇒ case 1
6. colour[y] \leftarrow BLACK ⇒ case 1
7. colour[p[p[z]]] \leftarrow RED ⇒ case 1
8. $z \leftarrow p[p[z]]$ ⇒ case 1
9. else if $z = \text{right}[p[z]]$
10. then $z \leftarrow p[z]$ ⇒ case 2
11. LEFT-ROTATE(T, z) ⇒ case 2
12. colour[p[z]] \leftarrow BLACK ⇒ case 3
13. colour[p[p[z]]] \leftarrow RED ⇒ case 3
14. RIGHT-ROTATE($T, p[p[z]]$) ⇒ case 3
15. else (same as then clause with “right” and “left” exchanged)
16. colour[root[T]] \leftarrow BLACK

Cases of RB-tree for insertion :

Case 1 : z 's uncle is red :

$$P[z] = \text{left}[p[p[z]]]$$

then uncle $\leftarrow \text{right}[p[p[z]]]$

- change z 's grandparent to red.
- change z 's uncle and parent to black.
- change z to z 's grandparent.

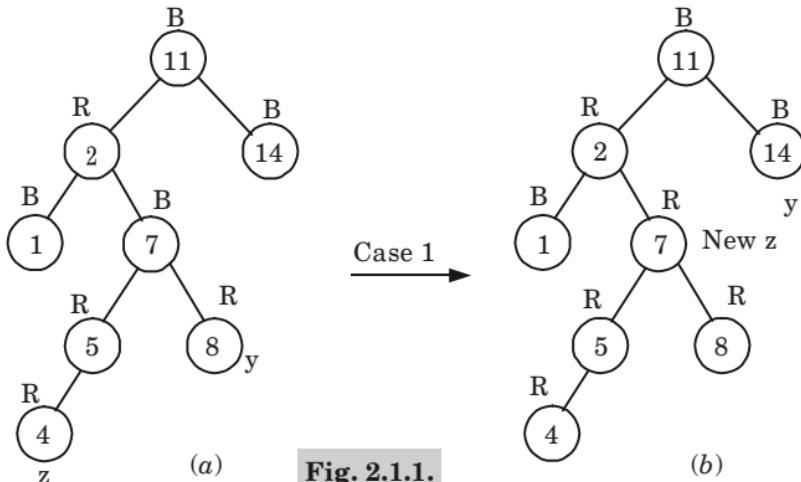
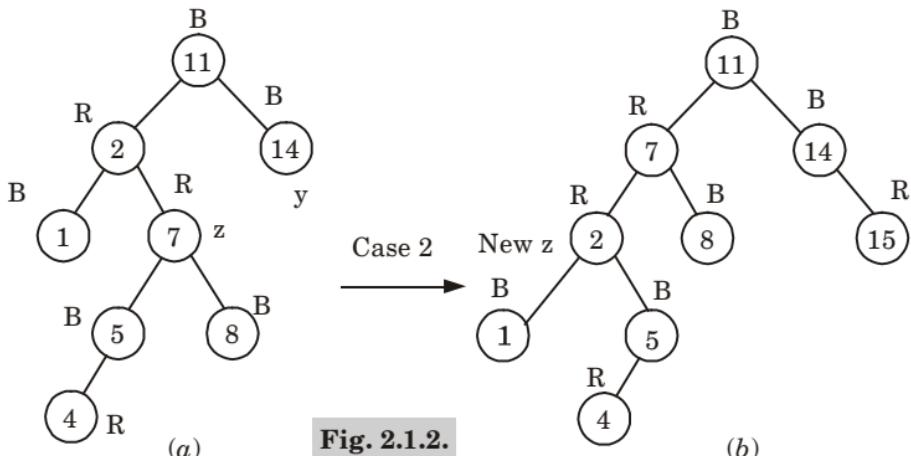


Fig. 2.1.1.

Now, in this case violation of property 4 occurs, because z 's uncle y is red, then case 1 is applied.

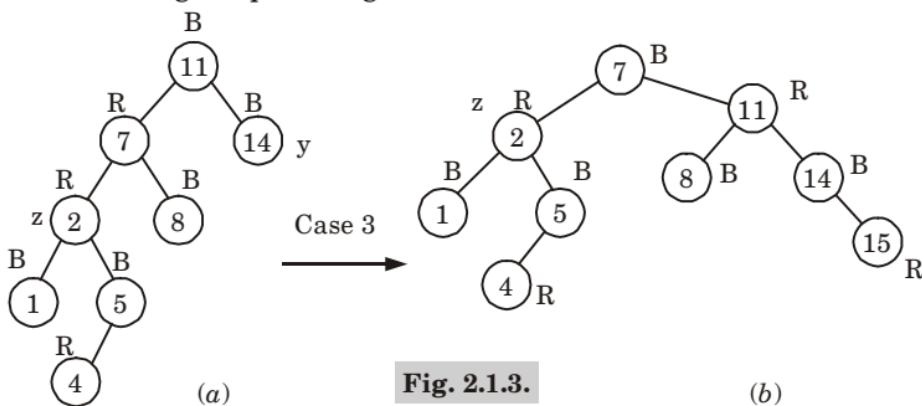
Case 2 : z 's uncle is black, z is the right of its parent :

- Change z to z 's parent.
- Rotate z 's parent left to make case 3.



Case 3 : z's uncle is black, z is the left child of its parent :

- Set z's parent black.
- Set z's grandparent to red.
- Rotate z's grandparent right.



Que 2.2. What are the advantages of red-black tree over binary search tree ? Write algorithms to insert a key in a red-black tree insert the following sequence of information in an empty red-black tree 1, 2, 3, 4, 5, 5.

Answer

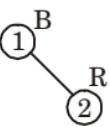
Advantages of RB-tree over binary search tree :

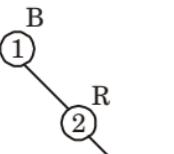
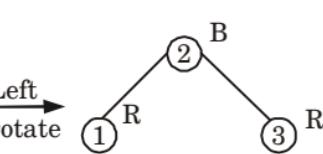
- The main advantage of red-black trees over AVL trees is that a single top-down pass may be used in both insertion and deletion operations.
- Red-black trees are self-balancing while on the other hand, simple binary search trees are unbalanced.
- It is particularly useful when inserts and/or deletes are relatively frequent.
- Time complexity of red-black tree is $O(\log n)$ while on the other hand, a simple BST has time complexity of $O(n)$.

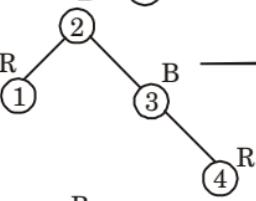
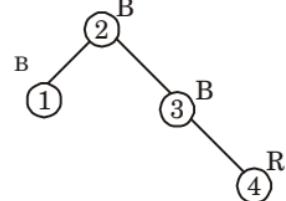
Algorithm to insert a key in a red-black tree : Refer Q. 2.1, Page 2-2B, Unit-2.

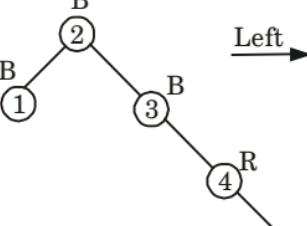
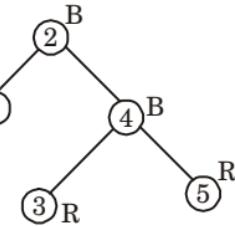
Numerical :

Insert 1 : 

Insert 2 : 

Insert 3 :  Left rotate 

Insert 4 :  → 

Insert 5 :  Left → 

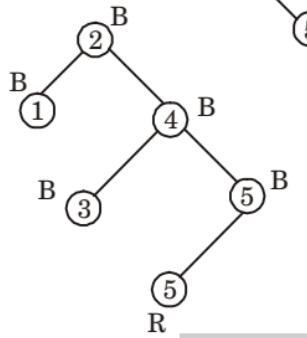
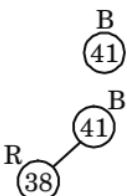
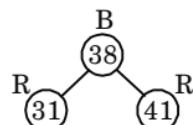
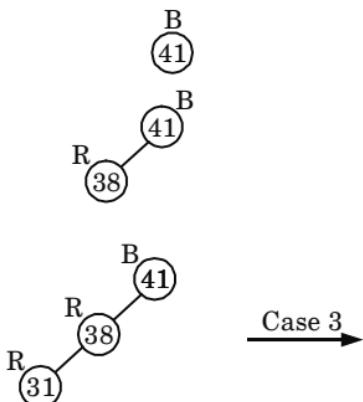
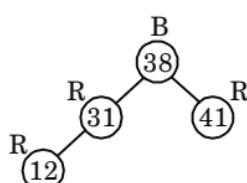
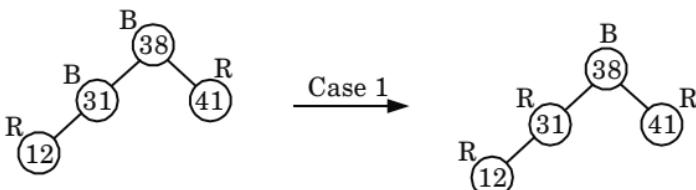
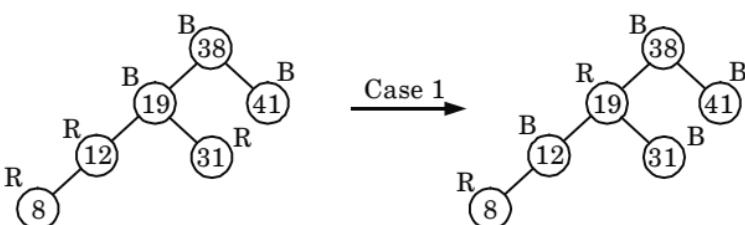
Insert 5 : 

Fig. 2.2.1.

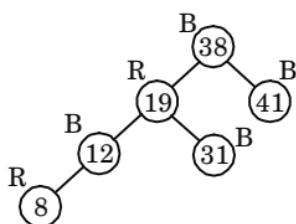
Que 2.3. Explain red-black tree. Show steps of inserting the keys 41, 38, 31, 12, 19, 8 into initially empty red-black tree.

OR

What is red-black tree ? Write an algorithm to insert a node in an empty red-black tree explain with suitable example.

Answer**Red-black tree and insertion algorithm :** Refer Q. 2.1, Page 2-2B, Unit-2.**Numerical :****Insert 41 :****Insert 38 :****Insert 31 :****Insert 12 :****Insert 19 :****Insert 8 :**

Thus final tree is



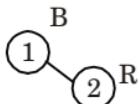
Que 2.4. Explain insertion in red-black tree. Show steps for inserting 1, 2, 3, 4, 5, 6, 7, 8 and 9 into empty RB-tree.

AKTU 2015-16, Marks 10

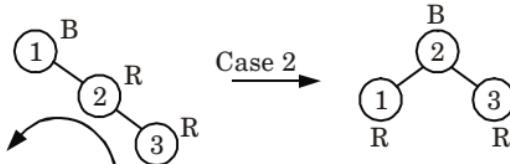
Answer

Insertion in red-black tree : Refer Q. 2.1, Page 2-2B, Unit-2.

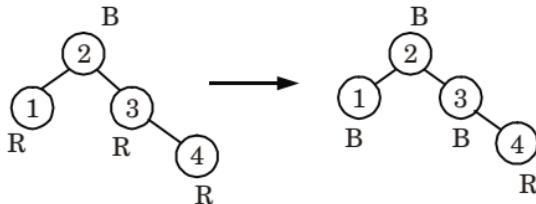
Insert 1 : 

Insert 2 : 

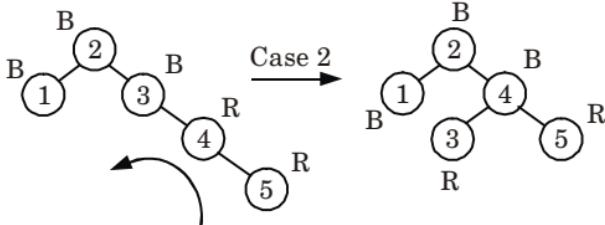
Insert 3 :



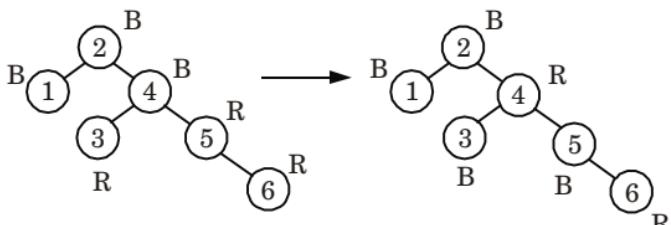
Insert 4 :

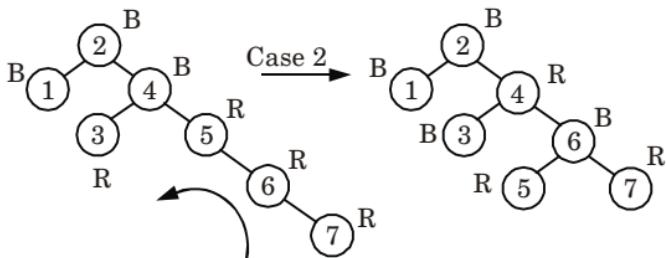
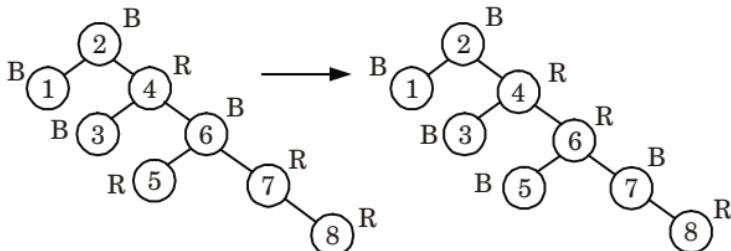
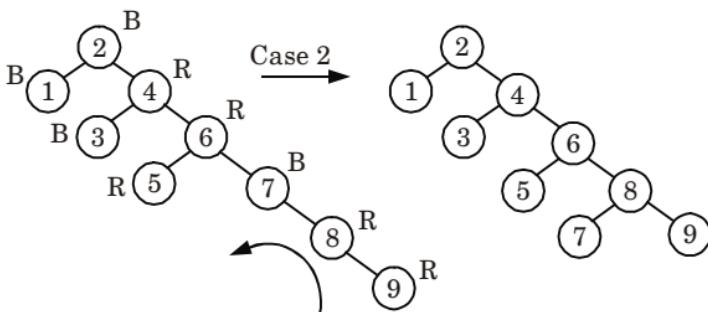


Insert 5 :



Insert 6 :



Insert 7 :**Insert 8 :****Insert 9 :**

Que 2.5. How to remove a node from RB-tree ? Discuss all cases and write down the algorithm.

Answer

To remove a node from RB-tree RB-DELETE procedure is used. In RB-DELETE procedure, after splitting out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colours and performs rotations to restore the red-black properties.

RB-DELETE(T, z)

1. if $\text{left}[z] = \text{nil}[T]$ or $\text{right}[z] = \text{nil}[T]$
2. then $y \leftarrow z$
3. else $y \leftarrow \text{TREE-SUCCESSOR}(z)$
4. if $\text{left}[y] \neq \text{nil}[T]$
5. then $x \leftarrow \text{left}[y]$
6. else $x \leftarrow \text{right}[y]$
7. $p[x] \leftarrow p[y]$
8. if $p[y] = \text{nil}[T]$

```

9.    then root[T] ← x
10.   else if  $y = \text{left}[p[y]]$ 
11.     then left[p[y]] ← x
12.     else right[p[y]] ← x
13.   if  $y \neq z$ 
14.     then key[z] ← key[y]
15.     copy  $y$ 's sibling data into  $z$ 
16.   if colour[y] = BLACK
17.     then RB-DELETE-FIXUP( $T, x$ )
18.   return  $y$ 

```

RB-DELETE-FIXUP(T, x)

```

1.   while  $x \neq \text{root}[T]$  and colour[x] = BLACK
2.     do if  $x = \text{left}[p[x]]$ 
3.       then  $w \leftarrow \text{right}[p[x]]$ 
4.       if colour[w] = RED
5.         then colour[w] ← BLACK          ⇒ case 1
6.         colour[p[x]] ← RED            ⇒ case 1
7.         LEFT-ROTATE( $T, p[x]$ )        ⇒ case 1
8.          $w \leftarrow \text{right}[p[x]]$       ⇒ case 1
9.     if colour[left[w]] = BLACK and colour[right[w]] = BLACK
10.    then colour[w] ← RED           ⇒ case 2
11.     $x \leftarrow p[x]$               ⇒ case 2
12.    else if colour[right[w]] = BLACK
13.      then colour[left[w]] ← BLACK  ⇒ case 3
14.      colour[w] ← RED            ⇒ case 3
15.      RIGHT-ROTATE( $T, w$ )        ⇒ case 3
16.       $w \leftarrow \text{right}[p[x]]$   ⇒ case 3
17.      colour[w] ← colour[p[x]]  ⇒ case 4
18.      colour[p[x]] ← BLACK      ⇒ case 4
19.      colour[right[w]] ← BLACK  ⇒ case 4
20.      LEFT-ROTATE( $T, p[x]$ )    ⇒ case 4
21.       $x \leftarrow \text{root}[T]$       ⇒ case 4
22.    else (same as then clause with "right" and "left" exchanged).
23.  colour[x] ← BLACK

```

Cases of RB-tree for deletion :**Case 1 : x 's sibling w is red :**

1. It occurs when node w the sibling of node x , is red.
2. Since w must have black children, we can switch the colours of w and $p[x]$ and then perform a left-rotation on $p[x]$ without violating any of the red-black properties.
3. The new sibling of x , which is one of w 's children prior to the rotation, is now black, thus we have converted case 1 into case 2, 3 or 4.
4. Case 2, 3 and 4 occur when node w is black. They are distinguished by colours of w 's children.

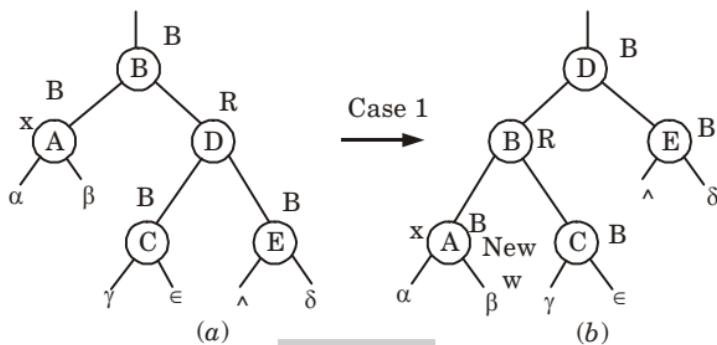


Fig. 2.5.1.

Case 2 : x 's sibling w is black, and both of w 's children are black :

- Both of w 's children are black. Since w is also black, we take one black of both x and w , leaving x with only one black and leaving w red.
- For removing one black from x and w , we add an extra black to $p[x]$, which was originally either red or black.
- We do so by repeating the while loop with $p[x]$ as the new node x .
- If we enter in case 2 through case 1, the new node x is red and black, the original $p[x]$ was red.
- The value c of the colour attribute of the new node x is red, and the loop terminates when it tests the loop condition. The new node x is then coloured black.

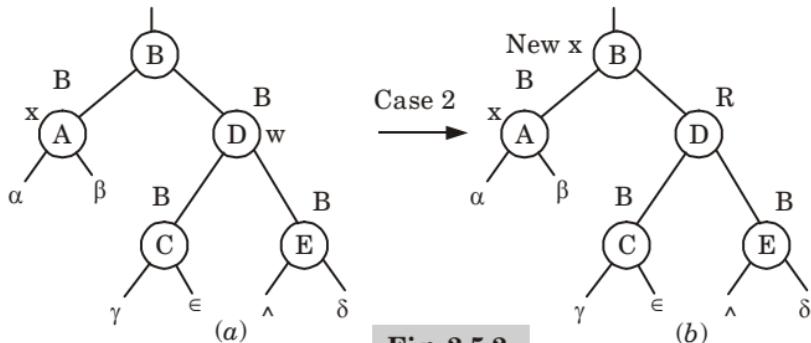
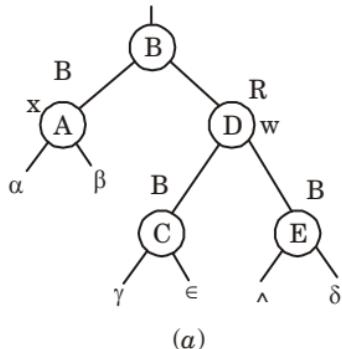


Fig. 2.5.2.

Case 3 : x 's sibling w is black, w 's left child is red, and w 's right child is black :

- Case 3 occurs when w is black, its left child is red and its right child is black.
- We can switch the colours of w and its left child $left[w]$ and then perform a right rotation on w without violating any of the red-black properties, the new sibling w of x is a black node with a red right child and thus we have transformed case 3 into case 4.



Case 3

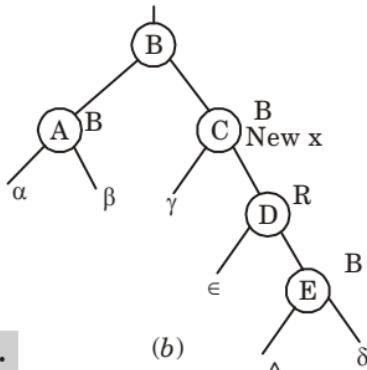
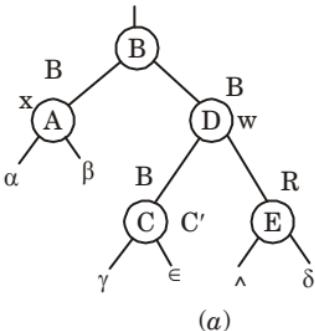


Fig. 2.5.3.

Case 4 : x's sibling w is black, and w's right child is red :

1. When node x 's sibling w is black and w 's right child is red.
2. By making some colour changes and performing a left rotation on $p[x]$, we can remove the extra black on x , making it singly black, without violating any of the red-black properties.



Case 4

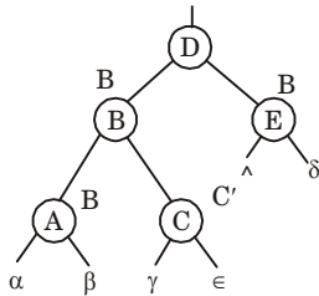


Fig. 2.5.4.

Que 2.6. Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion.

AKTU 2016-17, Marks 10

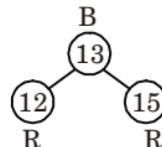
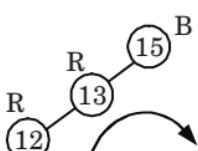
Answer**Insertion :**

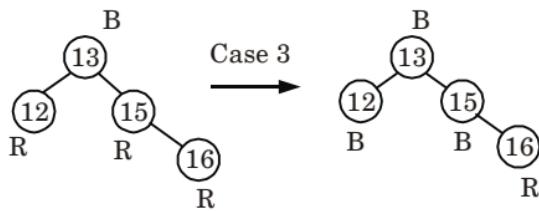
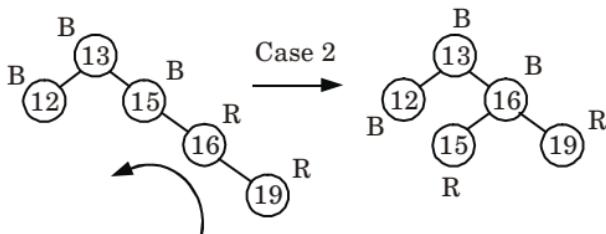
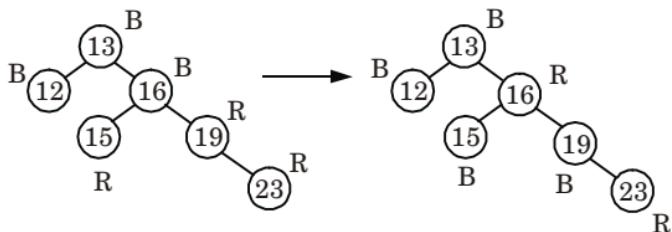
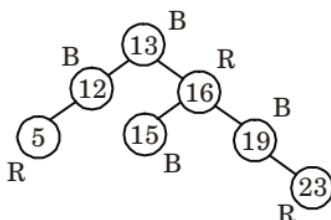
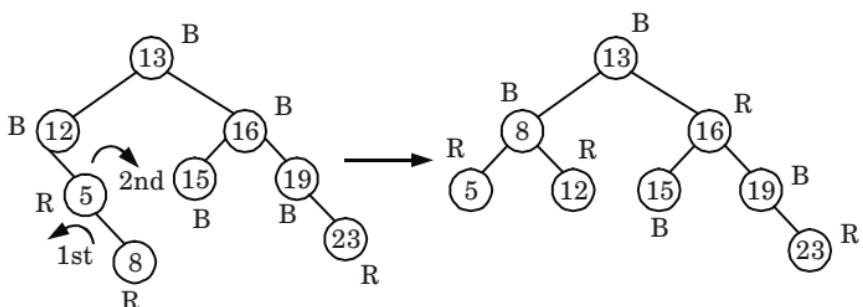
Insert 15 : (15) ^B

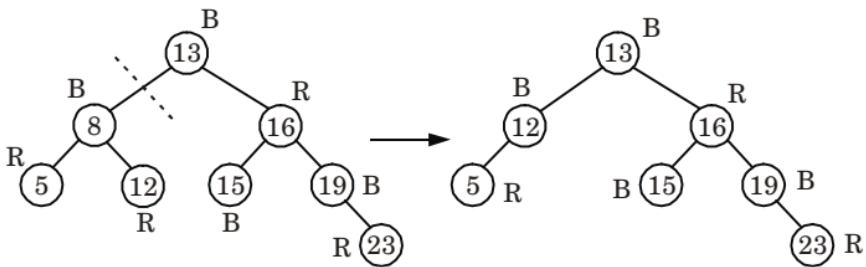
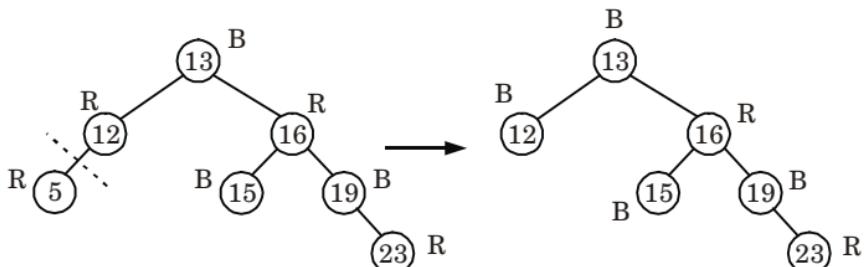
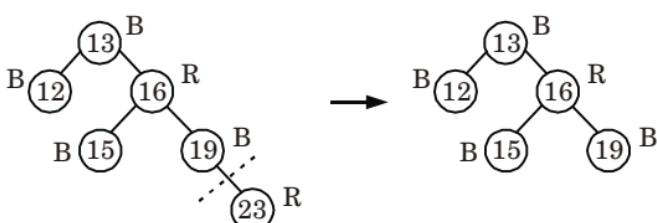
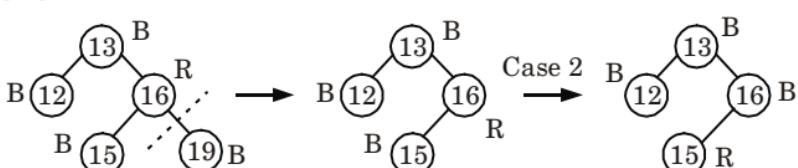
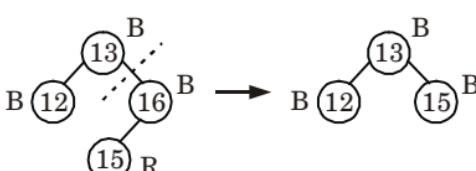
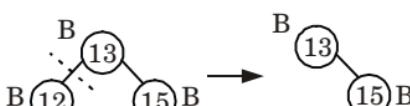
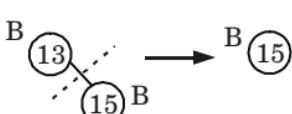


Insert 13 : (13) ^R

Insert 12 :



Insert 16 :**Insert 19 :****Insert 23 :****Insert 5 :****Insert 8 :**

Deletions :**Delete 8 :****Delete 5 :****Delete 23 :****Delete 19 :****Delete 16 :****Delete 12 :****Delete 13 :**

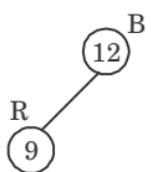
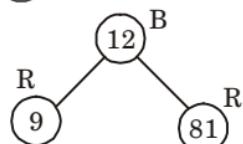
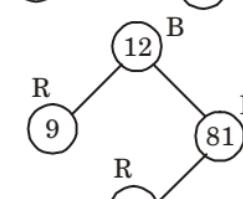
Delete 15 :

No tree

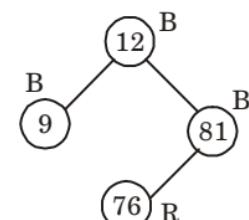
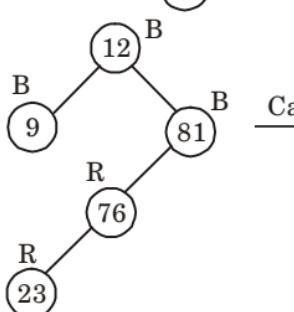
Que 2.7. Insert the following element in an initially empty RB-Tree.

12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now delete 23 and 81.

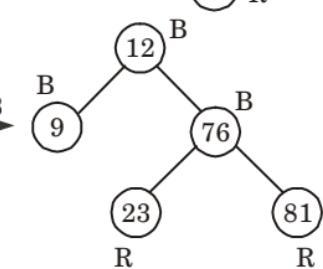
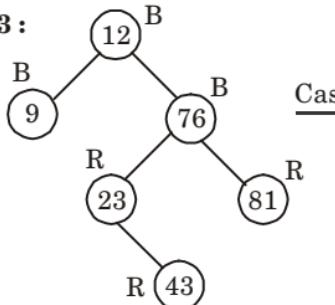
AKTU 2019-20, Marks 07

Answer**Insert 12 :****Insert 9 :****Insert 81 :****Insert 76 :**

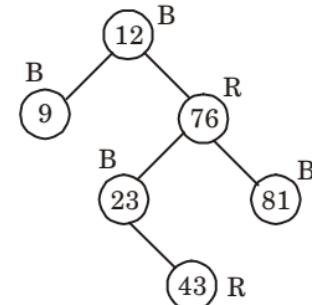
Case 1 →

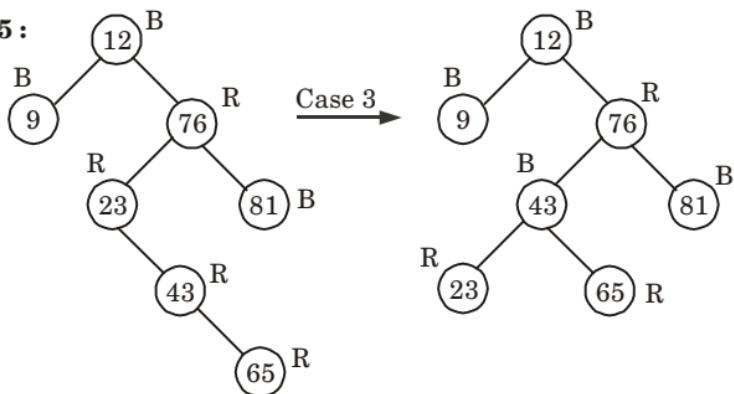
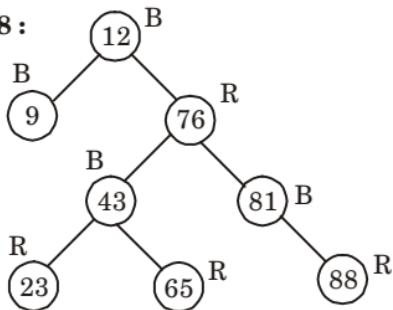
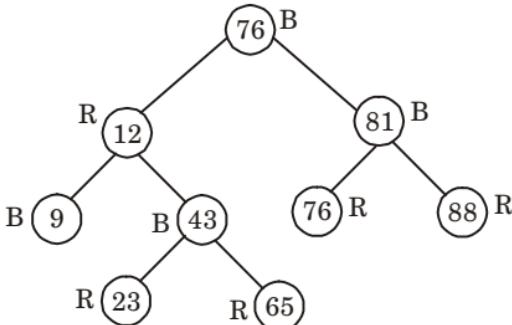
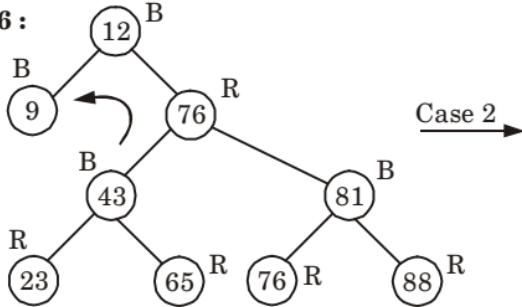
**Insert 23 :**

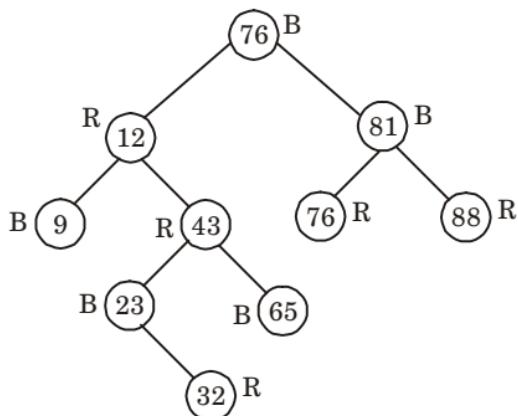
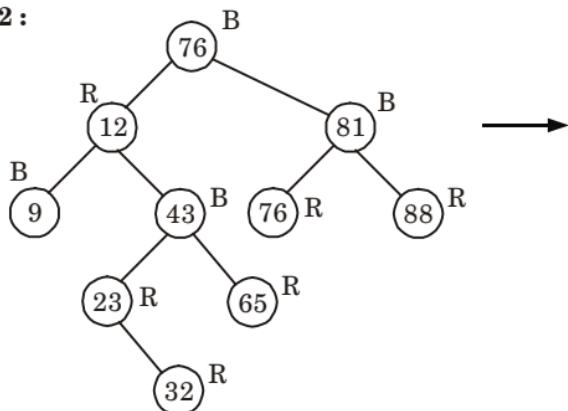
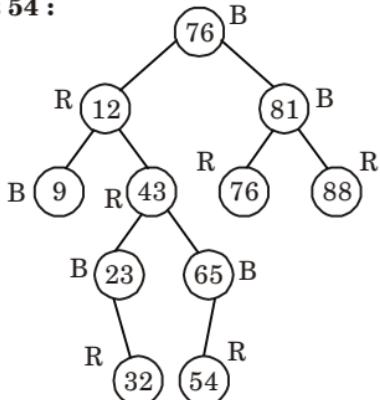
Case 3 →

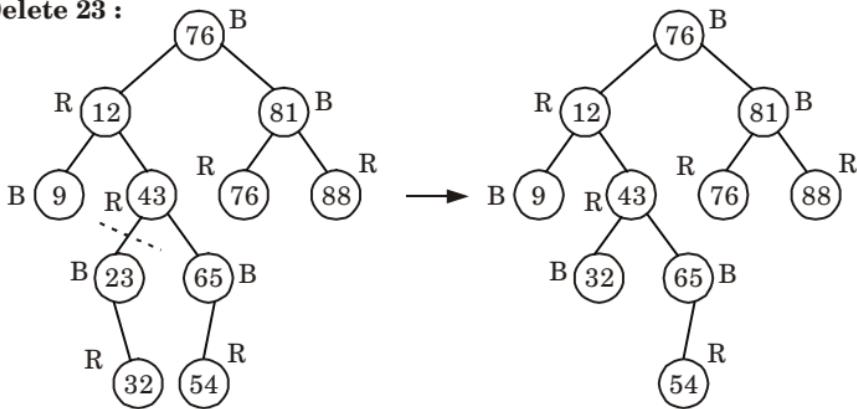
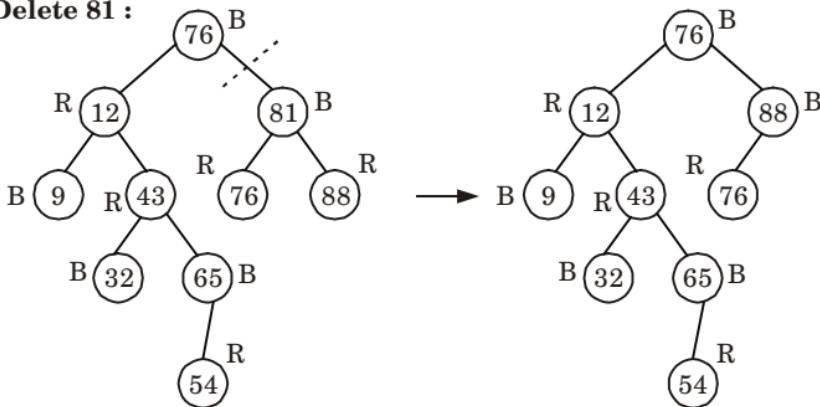
**Insert 43 :**

Case 1 →



Insert 65 :**Insert 88 :****Insert 76 :**

Insert 32 :**Insert 54 :**

Delete 23 :**Delete 81 :**

Que 2.8. Describe the properties of red-black tree. Show the red-black tree with n internal nodes has height at most $2 \log(n + 1)$.

OR

Prove the height h of a red-black tree with n internal nodes is not greater than $2 \log(n + 1)$.

Answer

Properties of red-black tree : Refer Q. 2.1, Page 2-2B, Unit-2.

1. By property 5 of RB-tree, every root-to-leaf path in the tree has the same number of black nodes, let this number be B .
2. So there are no leaves in this tree at depth less than B , which means the tree has at least as many internal nodes as a complete binary tree of height B .
3. Therefore, $n \leq 2^B - 1$. This implies $B \leq \log(n + 1)$.
4. By property 4 of RB-tree, at most every other node on a root-to-leaf path is red, therefore, $h \leq 2B$.

Putting these together, we have

$$h \leq 2 \log(n + 1).$$

Que 2.9. Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black tree and delete 12, 4, 9, 14 respectively.

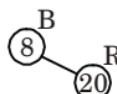
AKTU 2018-19, Marks 10

Answer

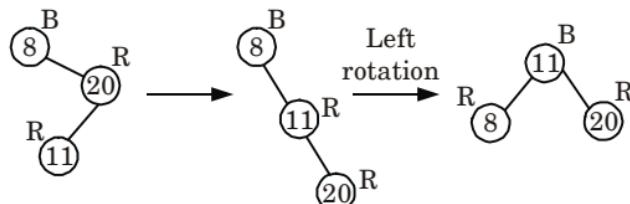
Insert 8 :



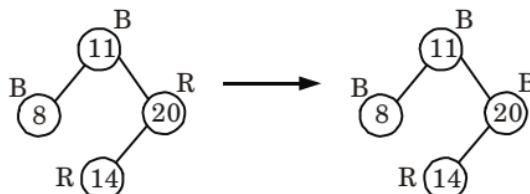
Insert 20 :



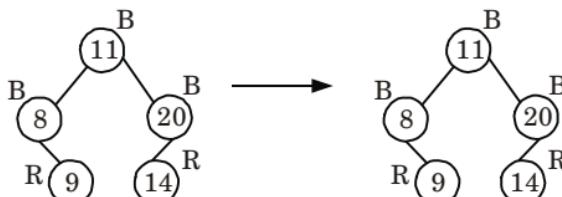
Insert 11 : Since, parent of node 11 is red. Check the colour of uncle of node 11. Since uncle of node 11 is nil than do rotation and recolouring.



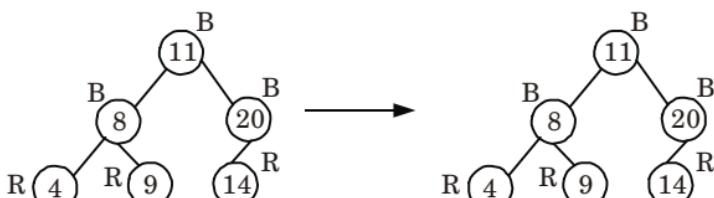
Insert 14 : Uncle of node 14 is red. Recolour the parent of node 14 i.e., 20 and uncle of node 14 i.e., 8. No rotation is required.



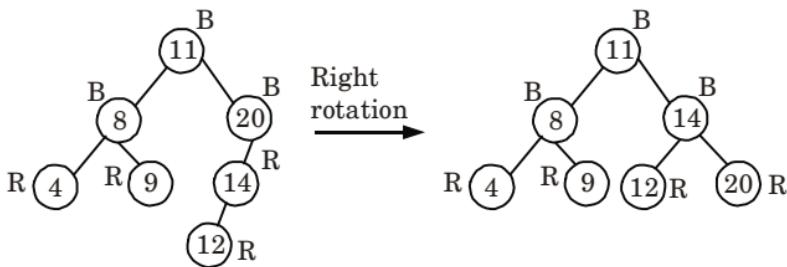
Insert 9 : Parent of node 9 is black. So no rotation and no recolouring.



Insert 4 : Parent of node 4 is black. So no rotation and no recolouring.



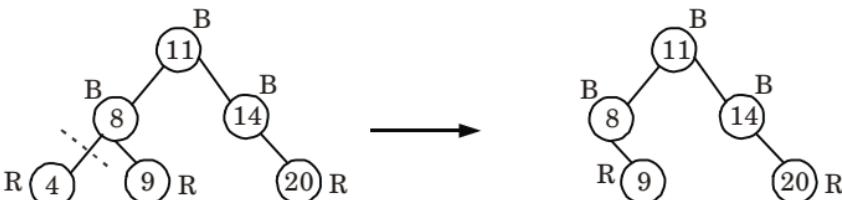
Insert 12 : Parent of node 12 is red. Check the colour of uncle of node 12, which is nil. So do rotation and recolouring.



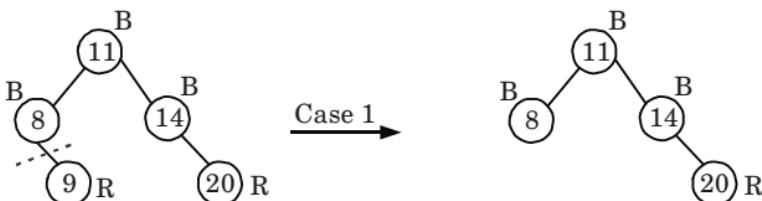
Delete 12 : Node 12 is red and leaf node. So simply delete node 12.



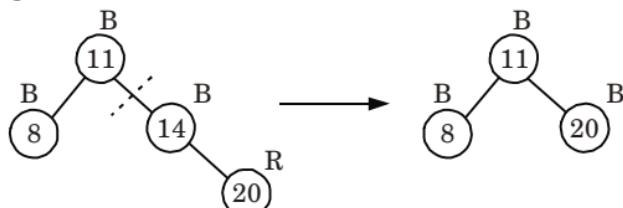
Delete 4 : Node 4 is red and leaf node. So simply delete node 4.



Delete 9 : Node 9 is red and leaf node. So simply delete node 9.



Delete 14 : Node 14 is internal node replace node 14 with node 20 and do not change the colour.



Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 2.10. Define a B-tree of order m . Explain the searching and insertion algorithm in a B-tree.

Answer

A B-tree of order m is an m -ary search tree with the following properties :

1. The root is either leaf or has atleast two children.
2. Each node, except for the root and the leaves, has between $m/2$ and m children.
3. Each path from the root to a leaf has the same length.
4. The root, each internal node and each leaf is typically a disk block.
5. Each internal node has upto $(m - 1)$ key values and upto m children.

SEARCH(x, k)

1. $i \leftarrow 1$
2. while $i \leq n[x]$ and $k > \text{key}_i[x]$
3. do $i \leftarrow i + 1$
4. if $i \leq n[x]$ and $k = \text{key}_i[x]$
5. then return(x, i)
6. if leaf[x]
7. then return NIL
8. else DISK-READ($c_i[x]$)
9. return B-TREE-SEARCH ($c_i[x], k$)

B-TREE-INSERT(T, k)

1. $r \leftarrow \text{root}[T]$
2. if $n[r] = 2t - 1$
3. then $s \leftarrow \text{ALLOCATE-NODE}()$
4. $\text{root}[T] \leftarrow S$
5. leaf[s] \leftarrow FALSE
6. $n[s] \leftarrow 0$
7. $c_1[s] \leftarrow r$
8. B-TREE SPLIT CHILD(S, l, r)
9. B-TREE-INSERT-NONFULL(s, k)
10. else B-TREE-INSERT-NONFULL(r, k)

B-TREE SPLIT CHILD(x, i, y)

1. $z \leftarrow \text{ALLOCATE-NODE}()$
2. leaf[z] \leftarrow leaf[y]
3. $n[z] \leftarrow t - 1$

4. for $j \leftarrow 1$ to $t - 1$
5. do $\text{key}_j[z] \leftarrow \text{key}_{j+1}[y]$
6. if not leaf[y]
7. then for $j \leftarrow 1$ to t
8. do $c_j[z] \leftarrow c_{j+1}[y]$
9. $n[y] \leftarrow t - 1$
10. for $j \leftarrow n[x] + 1$ down to $i + 1$
11. do $c_{j+1}[x] \leftarrow c_j[x]$
12. $c_{i+1}[x] \leftarrow z$
13. for $j \leftarrow n[x]$ down to i
14. do $\text{key}_{j+1}[x] \leftarrow \text{key}_j[x]$
15. $\text{key}_i[x] \leftarrow \text{key}_t[y]$
16. $n[x] \leftarrow n[x] + 1$
17. DISK-WRITE[y]
18. DISK-WRITE[z]
19. DISK-WRITE[x]

The CPU time used by B-TREE SPLIT CHILD is $\theta(t)$. The procedure performs $\theta(1)$ disk operations.

B-TREE-INSERT-NONFULL(x, k)

1. $i \leftarrow n[x]$
2. if leaf[x]
3. then while $i \geq 1$ and $k < \text{key}_i[x]$
4. do $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$
5. $i \leftarrow i - 1$
6. $\text{key}_{i+1}[x] \leftarrow k$
7. $n[x] \leftarrow n[x] + 1$
8. DISK-WRITE(x)
9. else while $i \geq 1$ and $k < \text{key}_i[x]$
10. do $i \leftarrow i - 1$
11. $i \leftarrow i + 1$
12. DISK-READ($c_i[x]$)
13. if $n[c_i[x]] = 2t - 1$
14. then B-TREE-SPLIT-CHILD($x, i, c_i[x]$)
15. if $k > \text{key}_i[x]$
16. then $i \leftarrow i + 1$
17. B-TREE INSERT NONFULL($c_i[x], k$)

The total CPU time use is $O(th) = O(t \log_t n)$

Que 2.11. What are the characteristics of B-tree ? Write down the steps for insertion operation in B-tree.

Answer**Characteristic of B-tree :**

1. Each node of the tree, except the root node and leaves has at least $m/2$ subtrees and no more than m subtrees.
2. Root of tree has at least two subtree unless it is a leaf node.
3. All leaves of the tree are at same level.

Insertion operation in B-tree :

In a B-tree, the new element must be added only at leaf node. The insertion operation is performed as follows :

Step 1 : Check whether tree is empty.

Step 2 : If tree is empty, then create a new node with new key value and insert into the tree as a root node.

Step 3 : If tree is not empty, then find a leaf node to which the new key value can be added using binary search tree logic.

Step 4 : If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.

Step 5 : If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.

Step 6 : If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Que 2.12. | Describe a method to delete an item from B-tree.**Answer**

There are three possible cases for deletion in B-tree as follows :

Let k be the key to be deleted, x be the node containing the key.

Case 1 : If the key is already in a leaf node, and removing it does not cause that leaf node to have too few keys, then simply remove the key to be deleted. Key k is in node x and x is a leaf, simply delete k from x .

Case 2 : If key k is in node x and x is an internal node, there are three cases to consider :

- a. If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k' in the subtree rooted at y . Recursively delete k' and replace k with k' in x .
- b. Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k' and delete and replace as before.
- c. Otherwise, if both y and z have only $t - 1$ (minimum number) keys, merge k and all of z into y , so that both k and the pointer to z are removed from x , y now contains $2t - 1$ keys, and subsequently k is deleted.

Case 3 : If key k is not present in an internal node x , determine the root of the appropriate subtree that must contain k . If the root has only $t - 1$ keys,

execute either of the following two cases to ensure that we descend to a node containing at least t keys. Finally, recurse to the appropriate child of x .

- If the root has only $t - 1$ keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x , and moving the appropriate child from the sibling to x .
- If the root and all of its siblings have $t - 1$ keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

Que 2.13. How B-tree differs with other tree structures ?

Answer

- In B-tree, the maximum number of child nodes a non-terminal node can have is m where m is the order of the B-tree. On the other hand, other tree can have at most two subtrees or child nodes.
- B-tree is used when data is stored in disk whereas other tree is used when data is stored in fast memory like RAM.
- B-tree is employed in code indexing data structure in DBMS, while, other tree is employed in code optimization, Huffman coding, etc.
- The maximum height of a B-tree is $\log mn$ (m is the order of tree and n is the number of nodes) and maximum height of other tree is $\log_2 n$ (base is 2 because it is for binary).
- A binary tree is allowed to have zero nodes whereas any other tree must have atleast one node. Thus binary tree is really a different kind of object than any other tree.

Que 2.14. Insert the following key in a 2-3-4 B-tree :

40, 35, 22, 90, 12, 45, 58, 78, 67, 60 and then delete key 35 and 22 one after other.

AKTU 2018-19, Marks 07

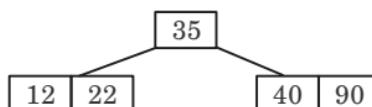
Answer

In 2-3-4 B-trees, non-leaf node can have minimum 2 keys and maximum 4 keys so the order of tree is 5.

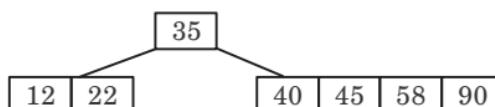
Insert 40, 35, 22, 90 :

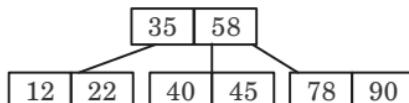
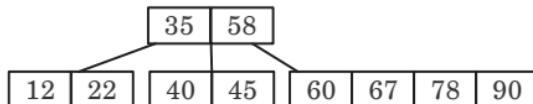
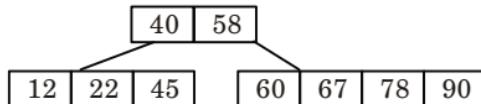
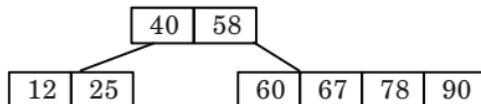
22	35	40	90
----	----	----	----

Insert 12 :



Insert 45, 58 :



Insert 78 :**Insert 67, 60 :****Delete 35 :****Delete 22 :**

Que 2.15. Explain B-tree and insert elements **B, Q, L, F** into B-tree Fig. 2.15.1 then apply deletion of elements **F, M, G, D, B** on resulting B-tree.

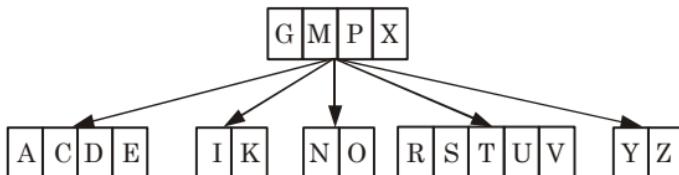
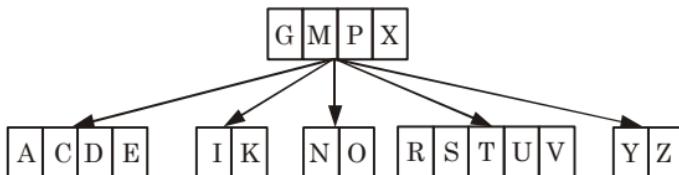
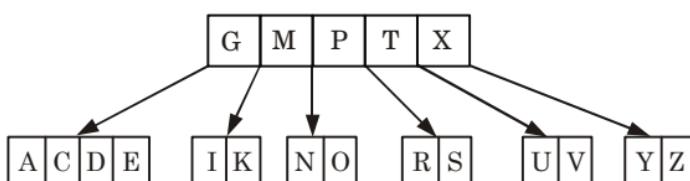


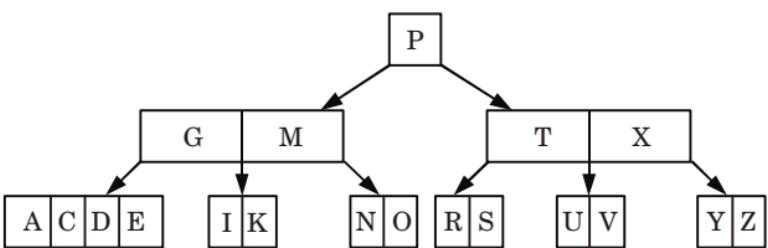
Fig. 2.15.1.

AKTU 2015-16, Marks 10

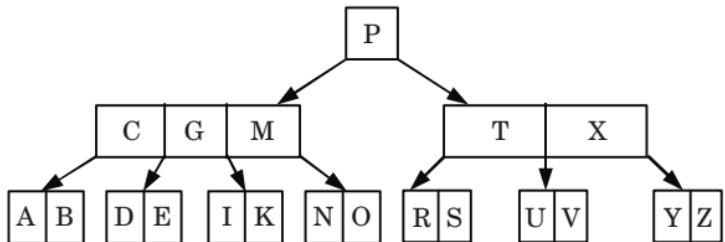
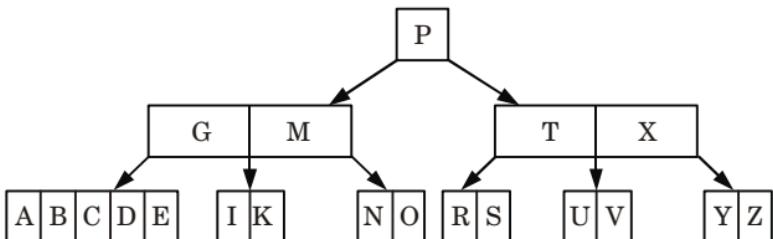
Answer**B-tree :** Refer Q. 2.10, Page 2-20B, Unit-2.**Numerical :****Insertion :**

Assuming, order of B-tree = 5

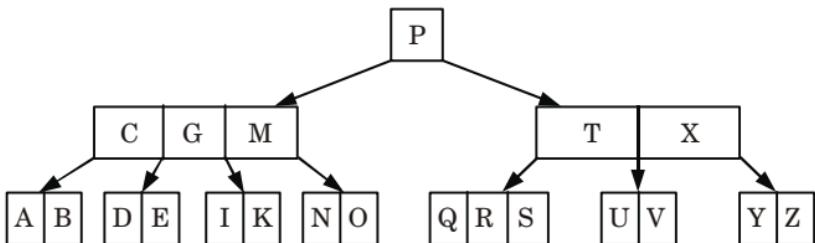




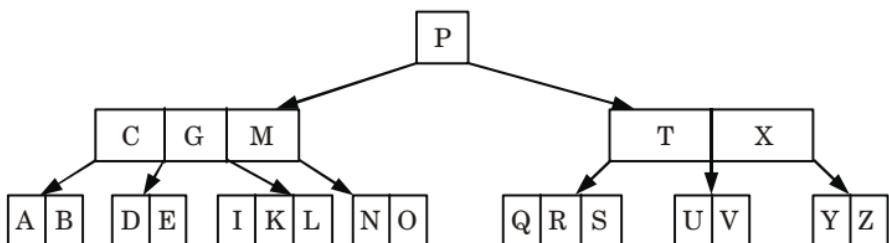
Insert B :

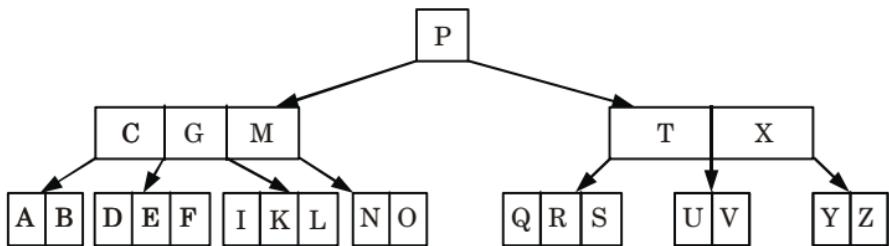
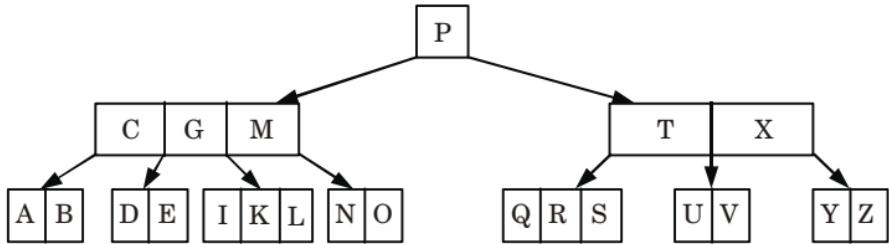
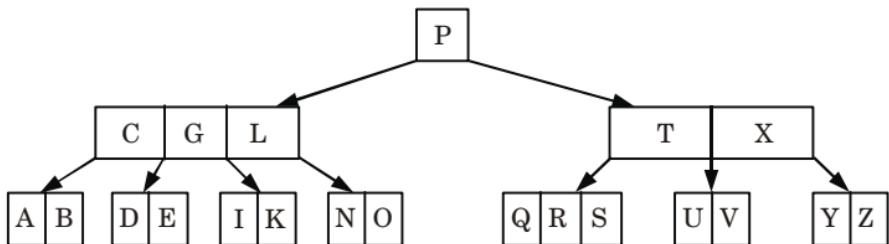
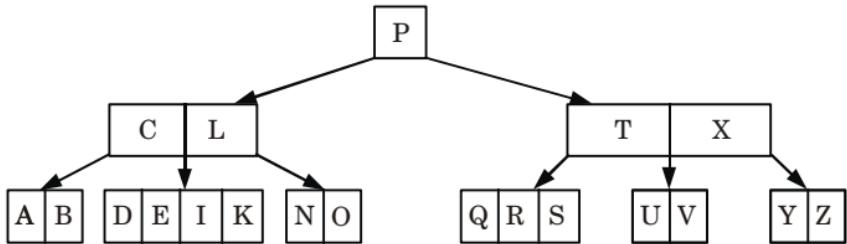
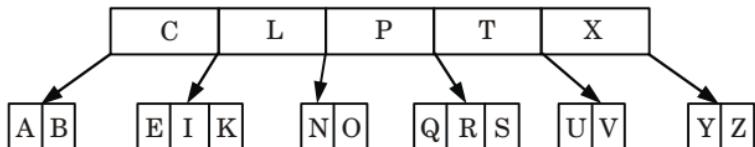
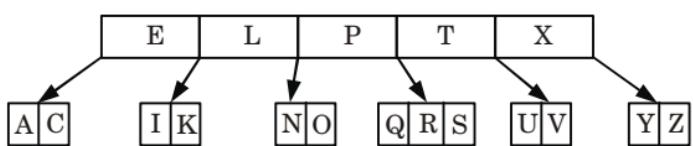


Insert Q :



Insert L :



Insert F :**Deletion :****Delete F :****Delete M :****Delete G :****Delete D :****Delete B :**

Que 2.16. Insert the following information, $F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I$ into an empty B-tree with degree $t = 3$.

AKTU 2017-18, Marks 10

Answer

Assume that

$$t = 3$$

$$2t - 1 = 2 \times 3 - 1 = 6 - 1 = 5$$

and

$$t - 1 = 3 - 1 = 2$$

So, maximum of 5 keys and minimum of 2 keys can be inserted in a node.

Now, apply insertion process as :

Insert F :



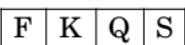
Insert S :



Insert Q :



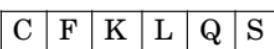
Insert K :



Insert C :



Insert L :



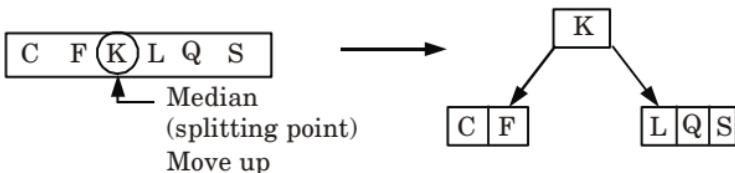
As, there are more than 5 keys in this node.

∴ Find median, $n[x] = 6$ (even)

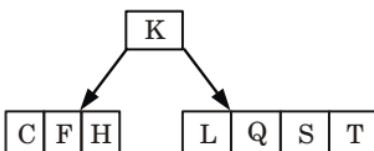
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$

Now, median = 3,

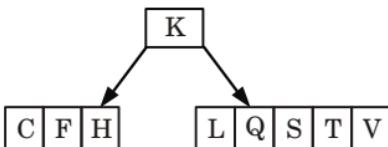
So, we split the node by 3rd key.

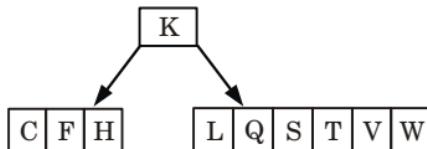


Insert H, T :



Insert V :

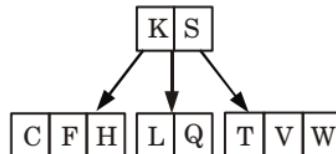
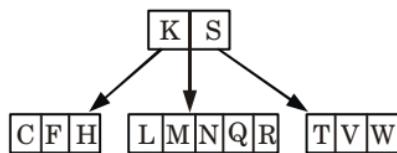
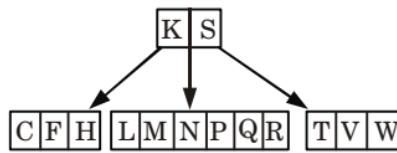


Insert W :

More than 5 keys split node from Median.

$$n[x] = 6 \text{ [even]}$$

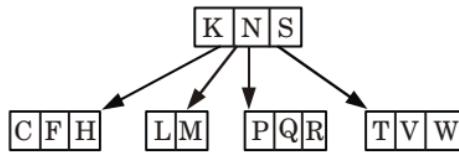
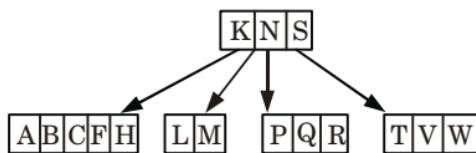
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$

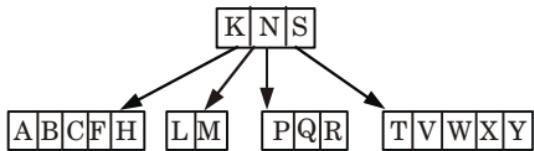
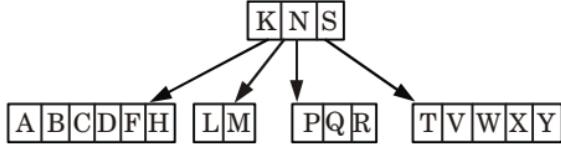
**Insert M, R, N :****Insert P :**

More than 5 key
split the node

$$n[x] = 6$$

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$

**Insert A, B :**

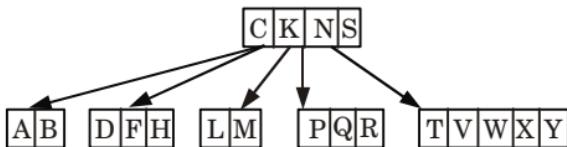
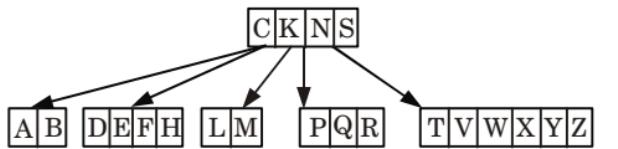
Insert X, Y :**Insert D :**

More than 5 key

split the node

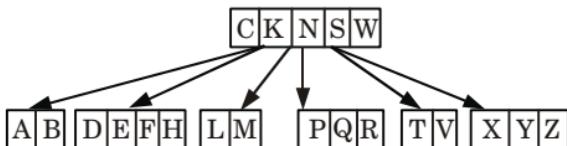
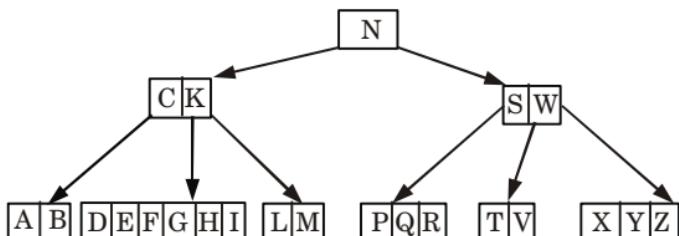
$$n[x] = 6 \text{ (even)}$$

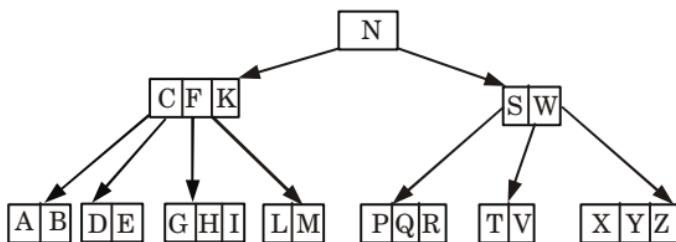
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$

**Insert Z, E :**More than 5 key
split the node

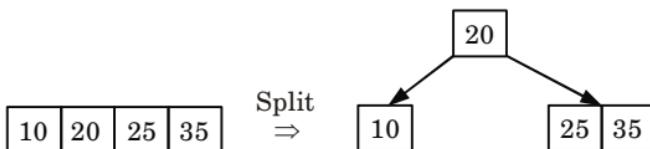
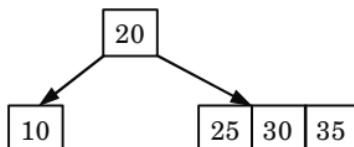
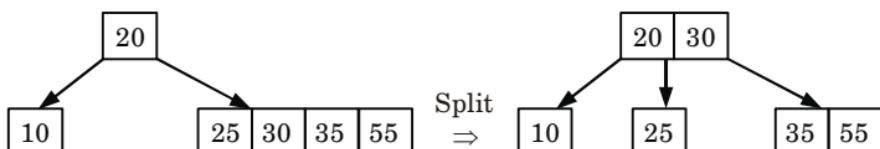
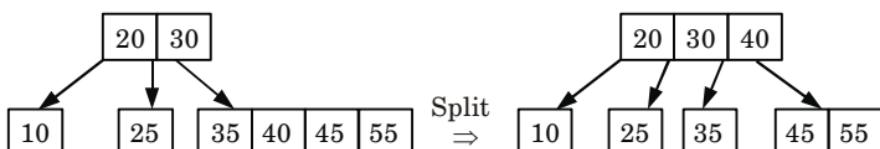
$$n[x] = 6$$

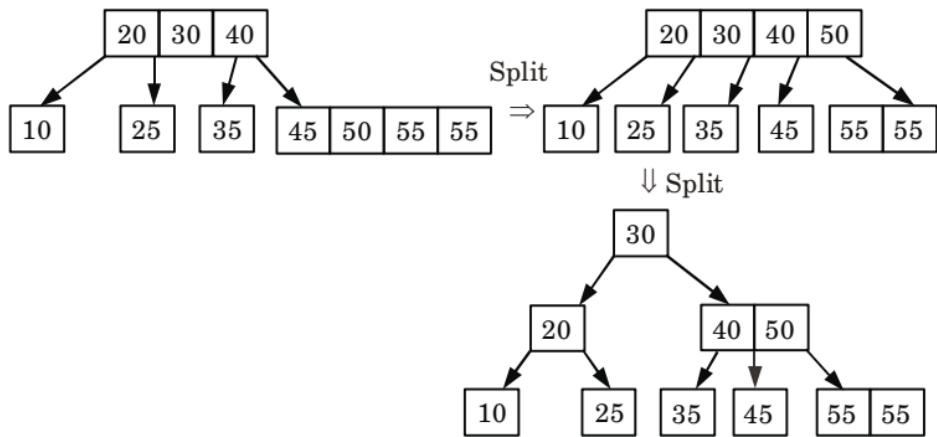
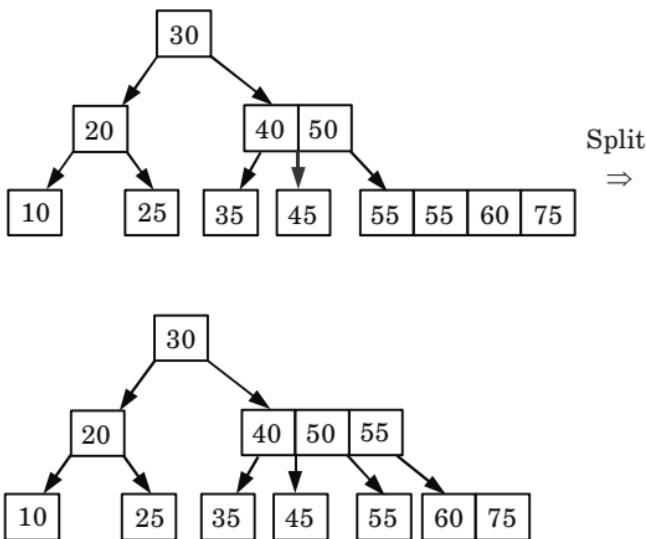
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$

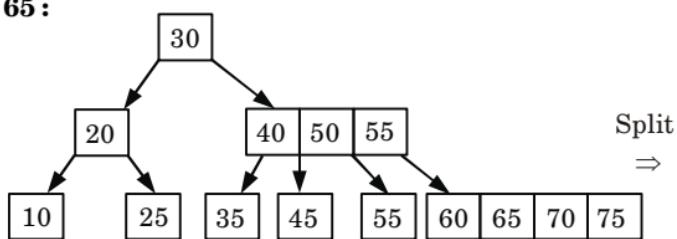
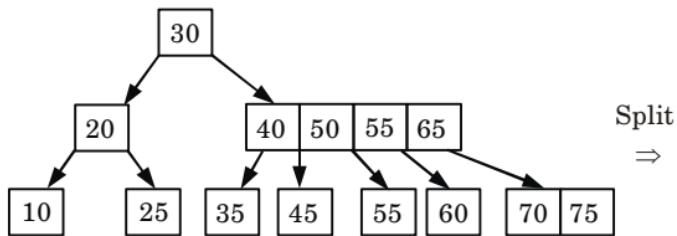
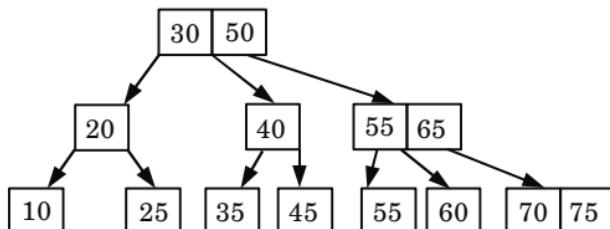
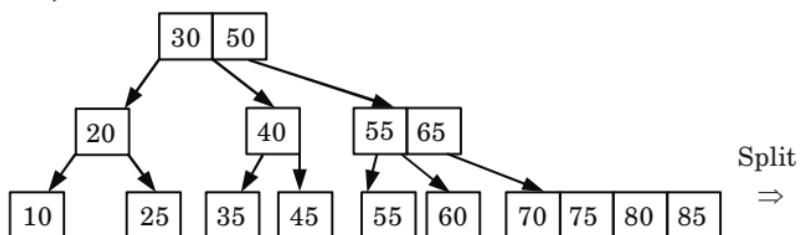
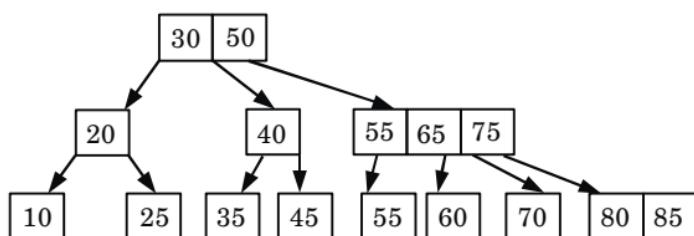
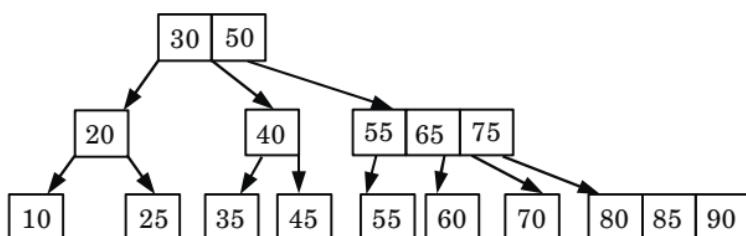
**Insert G, I :**

**Fig. 2.16.1.** Inserted all given information with degree $t = 3$.

Que 2.17. Using minimum degree ' t ' as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.

AKTU 2019-20, Marks 07**Answer****Insert 10, 25, 20, 35 :****Insert 30 :****Insert 55 :****Insert 40, 45 :**

Insert 50, 55 :**Insert 60, 75 :**

Insert 70, 65 :Split
⇒Split
⇒**Insert 80, 85 :**Split
⇒**Insert 90 :**

Number of nodes splitting operations = 9.

PART-3

Binomial Heaps.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.18. Explain binomial heap and properties of binomial tree.

Answer

Binomial heap :

1. Binomial heap is a type of data structure which keeps data sorted and allows insertion and deletion in amortized time.
2. A binomial heap is implemented as a collection of binomial tree.

Properties of binomial tree :

1. The total number of nodes at order k are 2^k .
2. The height of the tree is k .
3. There are exactly $\binom{k}{i}$ i.e., ${}^k C_i$ nodes at depth i for $i = 0, 1, \dots, k$ (this is why the tree is called a “binomial” tree).
4. Root has degree k (children) and its children are $B_{k-1}, B_{k-2}, \dots, B_0$ from left to right.

Que 2.19. What is a binomial heap ? Describe the union of binomial heap.

OR

Explain the different conditions of getting union of two existing binomial heaps. Also write algorithm for union of two binomial

heaps. What is its complexity ?

AKTU 2018-19, Marks 10

Answer

Binomial heap : Refer Q. 2.18, Page 2-33B, Unit-2.

Union of binomial heap :

1. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees where roots have the same degree.
2. The following procedure links the B_{k-1} tree rooted at node x to the B_{k-1} tree rooted at node y , that is, it makes y the parent of x . Node y thus becomes the root of a B_k tree.

BINOMIAL-LINK (y, z)

- i. $p[y] \leftarrow z$
- ii. $sibling[y] \leftarrow child[z]$
- iii. $child[z] \leftarrow y$
- iv. $degree[z] \leftarrow degree[z] + 1$

3. The BINOMIAL-HEAP-UNION procedure has two phases :
 - a. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order.
 - b. The second phase links root of equal degree until at most one root remains of each degree. Because the linked list H is sorted by degree, we can perform all the like operations quickly.

BINOMIAL-HEAP-UNION(H_1, H_2)

1. $H \leftarrow \text{MAKE-BINOMIAL-HEAP}()$
2. $\text{head}[H] \leftarrow \text{BINOMIAL-HEAP-MERGE}(H_1, H_2)$
3. Free the objects H_1 and H_2 but not the lists they point to
4. if $\text{head}[H] = \text{NIL}$
5. then return H
6. $\text{prev-}x \leftarrow \text{NIL}$
7. $x \leftarrow \text{head}[H]$
8. $\text{next-}x \leftarrow \text{sibling}[x]$
9. while $\text{next-}x \neq \text{NIL}$
10. do if ($\text{degree}[x] \neq \text{degree}[\text{next-}x]$) or
 ($\text{sibling}[\text{next-}x] \neq \text{NIL}$ and $\text{degree}[\text{sibling}[\text{next-}x]] = \text{degree}[x]$)
 11. then $\text{prev-}x \leftarrow x$ ⇒ case 1 and 2
 12. $x \leftarrow \text{next-}x$ ⇒ case 1 and 2
 13. else if $\text{key}[x] \leq \text{key}[\text{next-}x]$
 14. then $\text{sibling}[x] \leftarrow \text{sibling}[\text{next-}x]$ ⇒ case 3
 15. $\text{BINOMIAL-LINK}(\text{next-}x, x)$ ⇒ case 3
 16. else if $\text{prev-}x = \text{NIL}$ ⇒ case 4
 17. then $\text{head}[H] \leftarrow \text{next-}x$ ⇒ case 4
 18. else $\text{sibling}[\text{prev-}x] \leftarrow \text{next-}x$ ⇒ case 4
 19. $\text{BINOMIAL-LINK}(x, \text{next-}x)$ ⇒ case 4
 20. $x \leftarrow \text{next-}x$ ⇒ case 4
 21. $\text{next-}x \leftarrow \text{sibling}[x]$
 22. return H

BINOMIAL-HEAP-MERGE(H_1, H_2)

1. $a \leftarrow \text{head}[H_1]$
2. $b \leftarrow \text{head}[H_2]$
3. $\text{head}[H_1] \leftarrow \text{min-degree}(a, b)$
4. if $\text{head}[H_1] = \text{NIL}$
5. return
6. if $\text{head}[H_1] = b$
7. then $b \leftarrow a$
8. $a \leftarrow \text{head}[H_1]$
9. while $b \neq \text{NIL}$

10. do if $\text{ sibling}[a] = \text{NIL}$
11. then $\text{ sibling}[a] \leftarrow b$
12. return
13. else if $\text{ degree } [\text{ sibling}[a]] < \text{ degree } [b]$
14. then $a \leftarrow \text{ sibling}[a]$
15. else $c \leftarrow \text{ sibling}[b]$
16. $\text{ sibling}[b] \leftarrow \text{ sibling}[a]$
17. $\text{ sibling}[a] \leftarrow b$
18. $a \leftarrow \text{ sibling}[a]$
19. $b \leftarrow c$

There are four cases/conditions that occur while performing union on binomial heaps.

Case 1 : When $\text{ degree } [x] \neq \text{ degree } [\text{ next-}x] = \text{ degree } [\text{ sibling } [\text{ next-}x]]$, then pointers moves one position further down the root list.

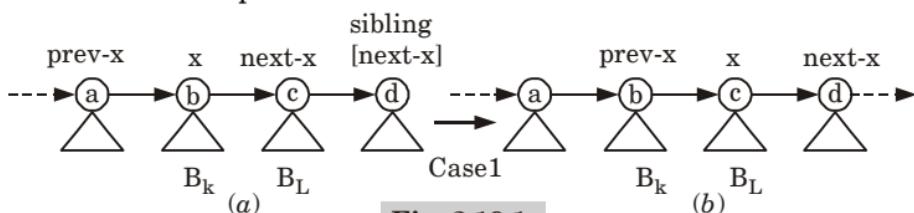


Fig. 2.19.1.

Case 2 : It occurs when x is the first of three roots of equal degree, that is, $\text{ degree } [x] = \text{ degree } [\text{ next-}x] = \text{ degree } [\text{ sibling } [\text{ next-}x]]$, then again pointer move one position further down the list, and next iteration executes either case 3 or case 4.

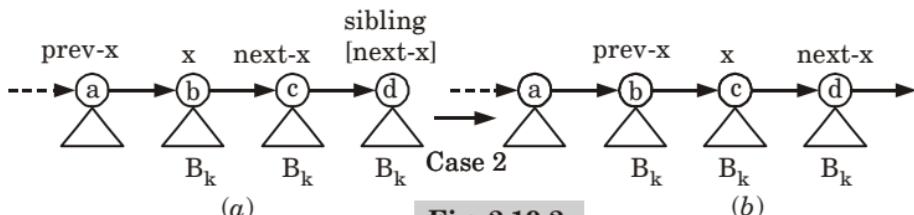


Fig. 2.19.2.

Case 3 : If $\text{ degree } [x] = \text{ degree } [\text{ next-}x] \neq \text{ degree } [\text{ sibling } [\text{ next-}x]]$ and $\text{ key } [x] \leq \text{ key } [\text{ next-}x]$, we remove $\text{ next-}x$ from the root list and link it to x , creating B_{k+1} tree.

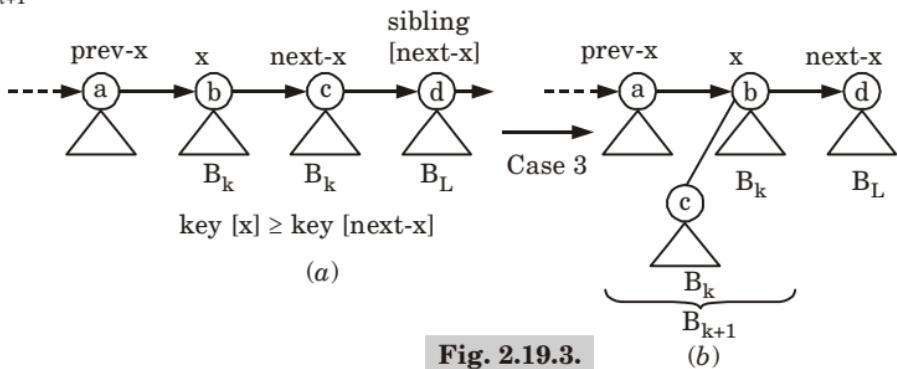


Fig. 2.19.3.

Case 4 : $\text{degree}[x] = \text{degree}[\text{next-}x] \neq \text{degree}[\text{sibling}[\text{next-}x]]$ and $\text{key}[\text{next-}x] \leq \text{key } x$, we remove x from the root list and link it to $\text{next-}x$, again creating a B_{k+1} tree.

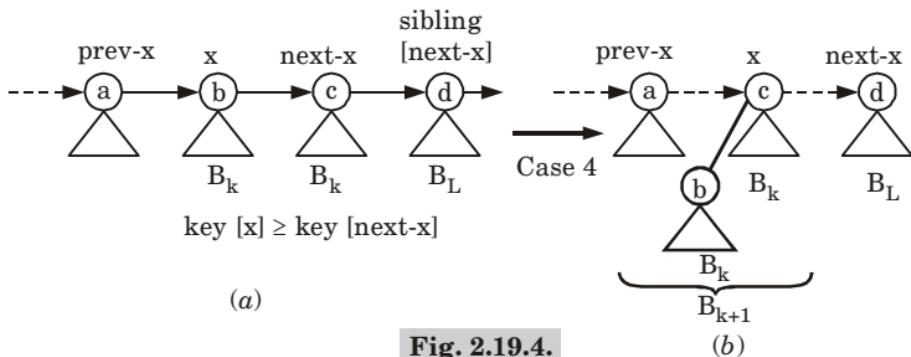


Fig. 2.19.4.

Time complexity of union of two binomial heap is $O(\log n)$.

Que 2.20. Explain properties of binomial heap. Write an algorithm to perform uniting two binomial heaps. And also to find Minimum key.

AKTU 2017-18, Marks 10

Answer

Properties of binomial heap : Refer Q. 2.18, Page 2-33B, Unit-2.

Algorithm for union of binomial heap : Refer Q. 2.19, Page 2-33B, Unit-2.

Minimum key :

BINOMIAL-HEAP-EXTRACT-MIN (H) :

- Find the root x with the minimum key in the root list of H , and remove x from the root list of H .
- $H' \leftarrow \text{MAKE-BINOMIAL-HEAP}()$.
- Reverse the order of the linked list of x 's children, and set $\text{head}[H']$ to point to the head of the resulting list.
- $H \leftarrow \text{BINOMIAL-HEAP-UNION}(H, H')$.
- Return x

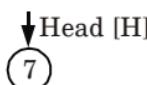
Since each of lines 1-4 takes $O(\log n)$ time of H has n nodes, BINOMIAL-HEAP-EXTRACT-MIN runs in $O(\log n)$ time.

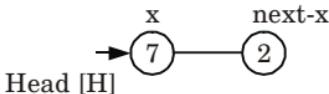
Que 2.21. Construct the binomial heap for the following sequence of number 7, 2, 4, 17, 1, 11, 6, 8, 15.

Answer

Numerical :

Insert 7 :



Insert 2 :

$\text{prev-}x = \text{NIL}$

$\text{degree } [x] = 0$. So, $\text{degree } [x] \neq \text{degree } [\text{next-}x]$ is false.

$\text{degree } [\text{next-}x] = 0$ and $\text{Sibling } [\text{next-}x] = \text{NIL}$

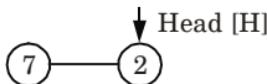
So, case 1 and 2 are false here.

Now key $[x] = 7$ and key $[\text{next-}x] = 2$

Now $\text{prev-}x = \text{NIL}$

then $\text{Head } [H] \leftarrow \text{next-}x$ and

i.e.,

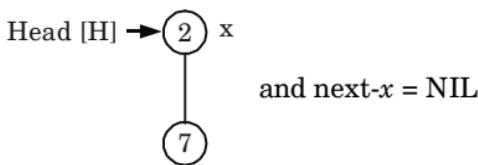


and $\text{BINOMIAL-LINK}(x, \text{next-}x)$

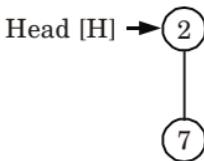
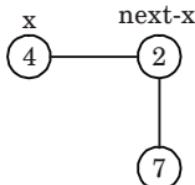
i.e.,



Now



So, after inserting 2, binomial heap is

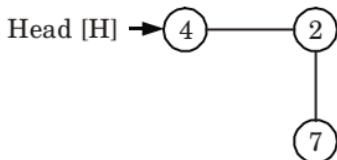
**Insert 4 :**

$\text{degree } [x] \neq \text{degree } [\text{next-}x]$

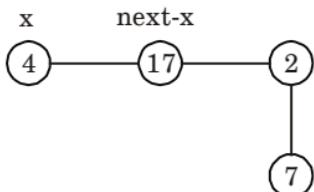
So, Now $\text{next-}x$ makes x and x makes $\text{prev-}x$.

Now $\text{next-}x = \text{NIL}$

So, after inserting 4, final binomial heap is :

**Insert 17 :**

After Binomial-Heap-Merge, we get



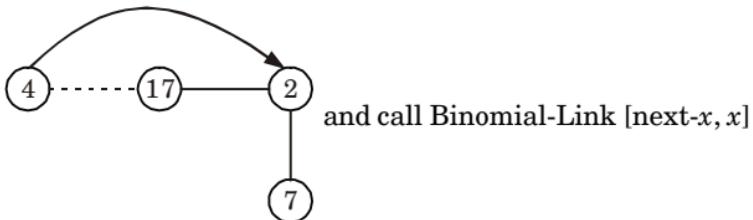
$\text{degree}[x] = \text{degree}[\text{next-}x]$

$\text{degree}[\text{Sibling-}[\text{next-}x]] \neq \text{degree}[x]$

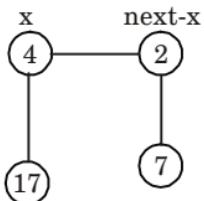
$\text{key}[x] \leq \text{key}[\text{next-}x]$

$4 \leq 17$ [True]

So,



We get



$\text{degree}[x] = \text{degree}[\text{next-}x]$

$\text{Sibling}[\text{next-}x] = \text{NIL}$

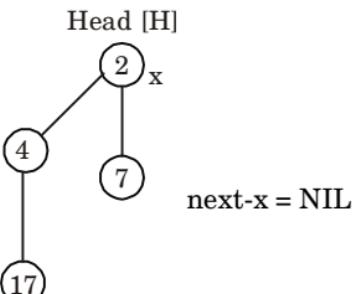
$\text{Key}[x] \leq \text{key}[\text{next-}x]$ [False]

$\text{prev-}x = \text{NIL}$ then

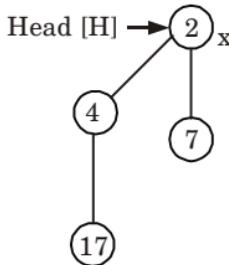
$\text{Head}[\text{H}] \leftarrow [\text{next-}x]$

Binomial-Link [x , $\text{next-}x$]

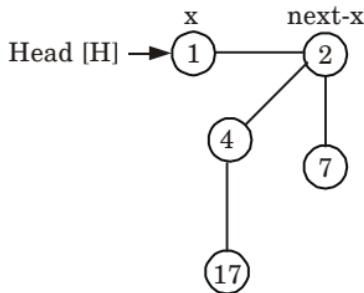
$x \leftarrow \text{next-}x$



So, after inserting 17, final binomial heap is :

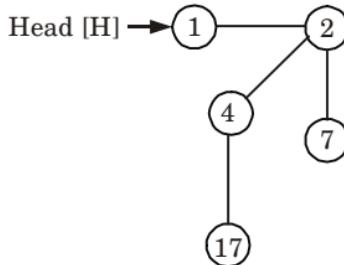


Insert 1 :



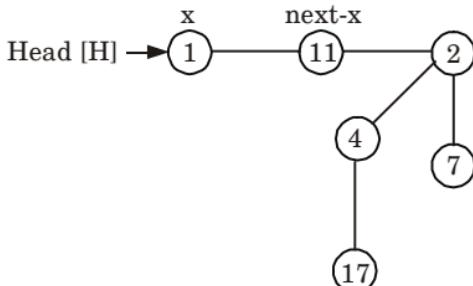
degree [x] ≠ degree [next-x]

So, next-x makes x and next-x = NIL
and after inserting 1, binomial heap is :



Insert 11 :

After Binomial-Heap-Merge, we get

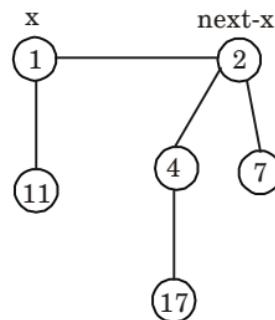


degree [x] = degree [next-x]

degree [Sibling [next-x]] ≠ degree [x]

key [x] ≤ key [next-x] [True]

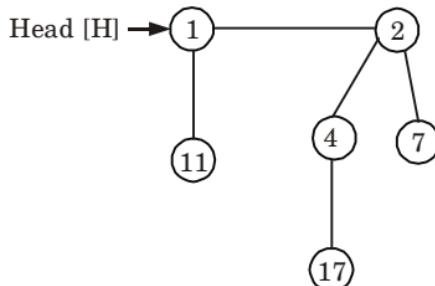
So,



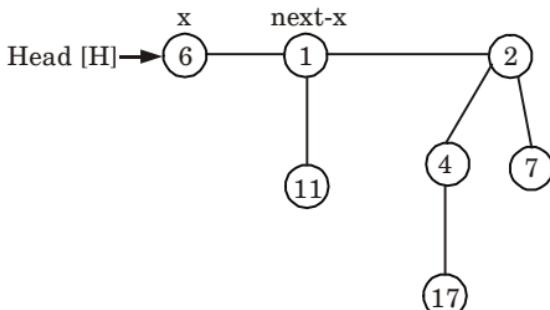
$\text{degree}[x] \neq \text{degree}[\text{next-}x]$

So, $\text{next-}x$ makes x and $\text{next-}x = \text{NIL}$

and final binomial heap after inserting 11 is



Insert 6 :

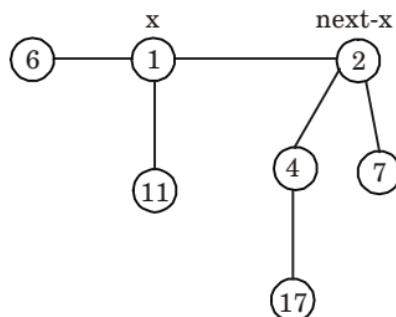


$\text{degree}[x] \neq \text{degree}[\text{next-}x]$

So, $\text{next-}x$ becomes x

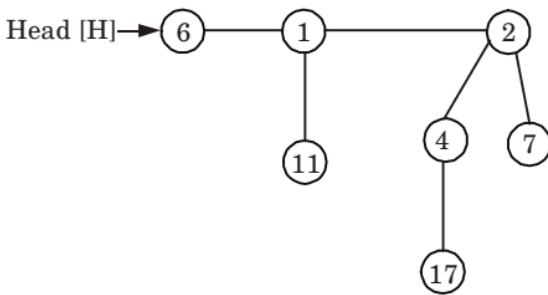
Sibling $[\text{next-}x]$ becomes $\text{next-}x$.

i.e.,

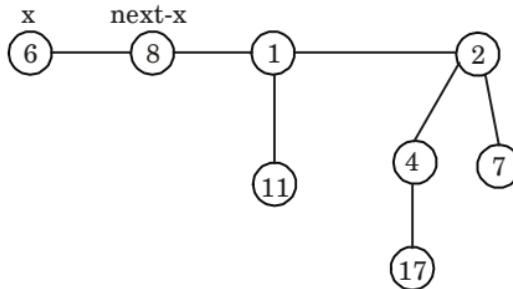


degree [x] ≠ degree [next-x]

So, no change and final heap is :



Insert 8 :

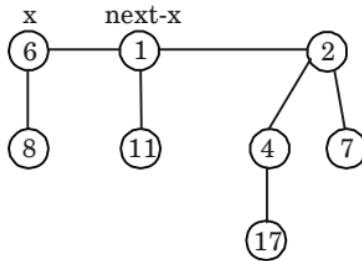


degree [x] = degree [next-x]

degree [Sibling [next-x]] ≠ degree [x]

key [x] ≤ key [next-x] [True]

So,



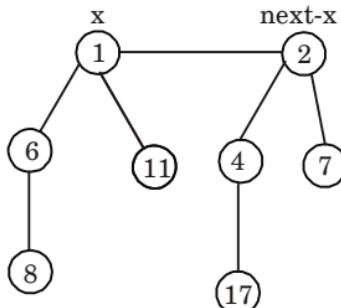
degree [x] = degree [next-x]

degree [Sibling p[next-x]] ≤ degree [x]

key [x] ≤ key [next-x] [False]

prev-x = NIL

So,



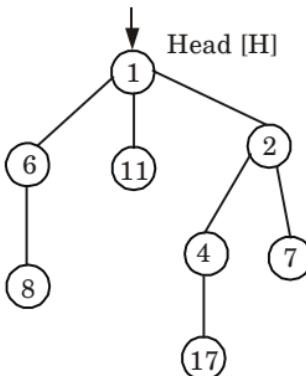
degree [x] = degree [next-x]

Sibling [next-x] = NIL

key [x] ≤ key [next-x] [True]

So, Sibling [x] = NIL.

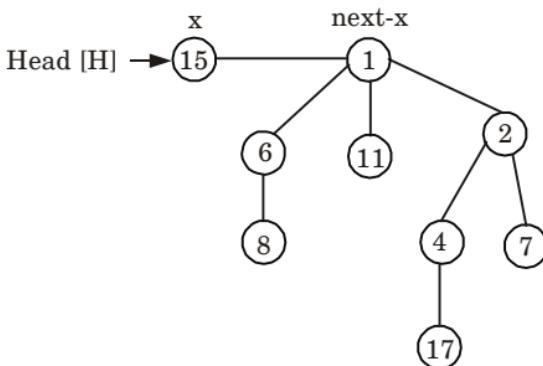
and



next [x] = NIL

So, this is the final binomial heap after inserting 8.

Insert 15 :



degree [x] ≠ degree [next-x]

So, no change and this is the final binomial heap after inserting 15.

Que 2.22. Explain the algorithm to delete a given element in a binomial heap. Give an example for the same.

AKTU 2019-20, Marks 07

Answer

Deletion of key from binomial heap :

The operation BINOMIAL-HEAP-DECREASE (H, x, k) assigns a new key ' k ' to a node 'x' in a binomial heap H .

BINOMIAL-HEAP-DECREASE-KEY (H, x, k)

1. if $k > \text{key}[x]$ then
2. Message "error new key is greater than current key"
3. $\text{key}[x] \leftarrow k$

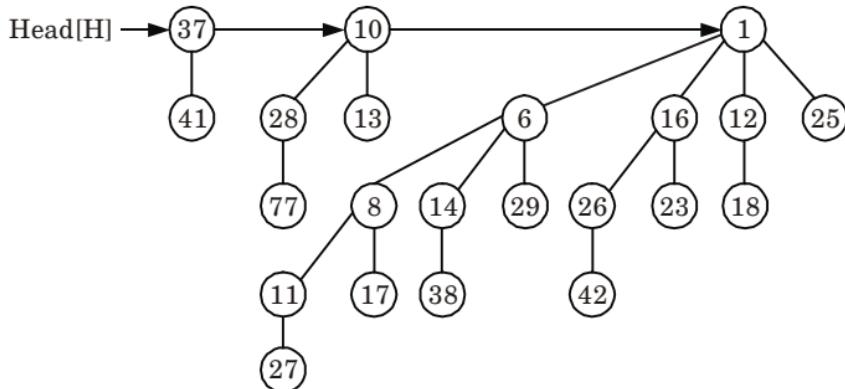
4. $y \leftarrow x$
5. $z \leftarrow P[y]$
6. While ($z \neq \text{NIL}$) and key [y] < key [z]
7. do exchange key [y] \leftrightarrow key [z]
9. $y \leftarrow z$
10. $z \leftarrow P[y]$

Deleting a key : The operation BINOMIAL-HEAP-DELETE (H, x) is used to delete a node x 's key from the given binomial heap H . The following implementation assumes that no node currently in the binomial heap has a key of $-\infty$.

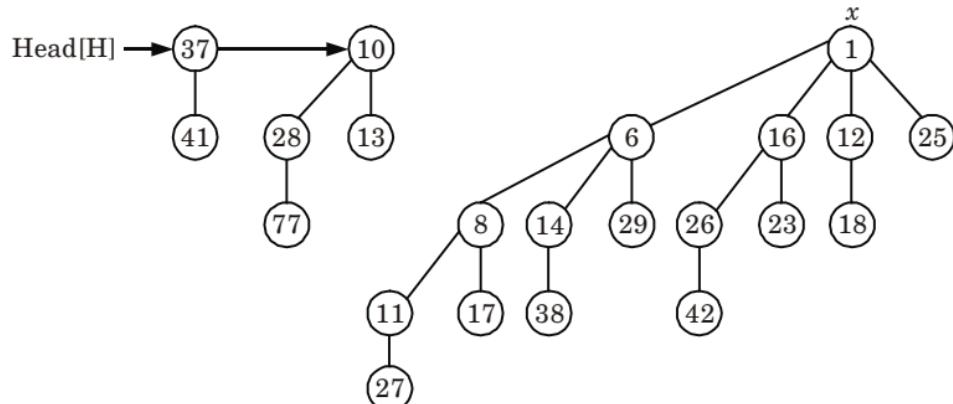
- BINOMIAL-HEAP-DELETE (H, x)
1. BINOMIAL-HEAP-DECREASE-KEY ($H, x, -\infty$)
 2. BINOMIAL-HEAP-EXTRACT-MIN(H)

For example : Operation of Binomial-Heap-Decrease (H, x, k) on the following given binomial heap :

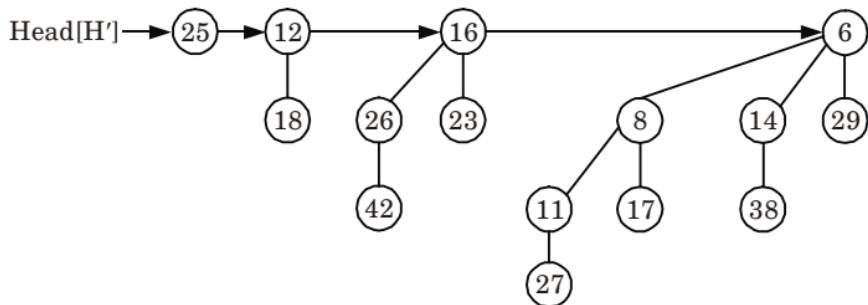
Suppose a binomial heap H is as follows :



The root x with minimum key is 1. x is removed from the root list of H . i.e.,

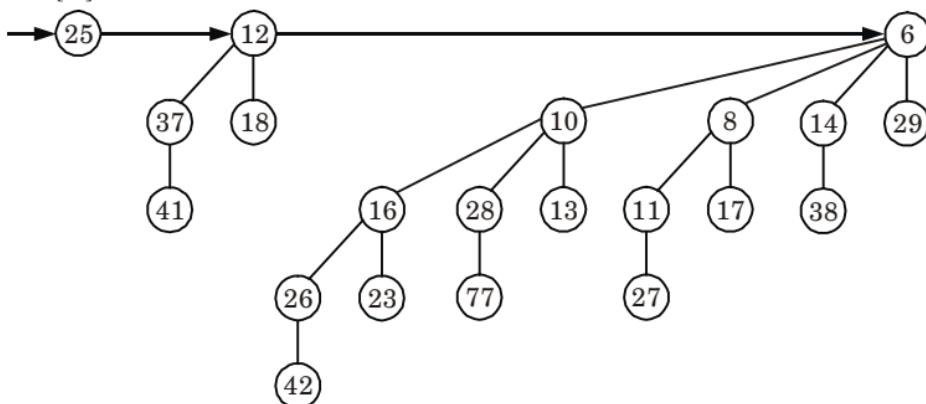


Now, the linked list of x 's children is reversed and set head[H'] to point to the head of the resulting list, i.e., another binomial heap H' .



Now, call BINOMIAL-HEAP-UNION (H, H') to uniting the two binomial heaps H and H' . The resulting binomial heap is

Head[H]

**PART-4***Fibonacci Heaps.***Questions-Answers****Long Answer Type and Medium Answer Type Questions**

Que 2.23. What is a Fibonacci heap ? Discuss the applications of Fibonacci heaps.

Answer

1. A Fibonacci heap is a set of min-heap-ordered trees.
2. Trees are not ordered binomial trees, because

- Children of a node are unordered.
- Deleting nodes may destroy binomial construction.

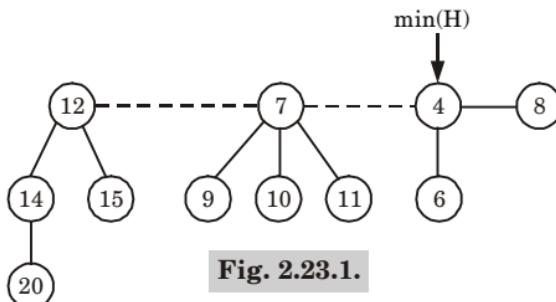


Fig. 2.23.1.

- Fibonacci heap H is accessed by a pointer $\text{min}[H]$ to the root of a tree containing a minimum key. This node is called the minimum node.
- If Fibonacci heap H is empty, then $\text{min}[H] = \text{NIL}$.

Applications of Fibonacci heap :

- Fibonacci heap is used for Dijkstra's algorithm because it improves the asymptotic running time of this algorithm.
- It is used in finding the shortest path. These algorithms run in $O(n^2)$ time if the storage for nodes is maintained as a linear array.

Que 2.24. What is Fibonacci heap ? Explain CONSOLIDATE operation with suitable example for Fibonacci heap.

AKTU 2015-16, Marks 15

Answer

Fibonacci heap : Refer Q. 2.23, Page 2-44B, Unit-2.

CONSOLIDATE operation :

CONSOLIDATE(H)

- for $i \leftarrow 0$ to $D(n[H])$
- do $A[i] \leftarrow \text{NIL}$
- for each node w in the root list of H
- do $x \leftarrow w$
- $d \leftarrow \text{degree}[x]$
- while $A[d] \neq \text{NIL}$
- do $y \leftarrow A[d]$ \triangleright Another node with the same degree as x .
- if $\text{key}[x] > \text{key}[y]$
- then exchange $x \leftrightarrow y$
- FIB-HEAP-LINK(H, y, x)
- $A[d] \leftarrow \text{NIL}$
- $d \leftarrow d + 1$
- $A[d] \leftarrow x$
- $\text{min}[H] \leftarrow \text{NIL}$
- for $i \leftarrow 0$ to $D(n[H])$
- do if $A[i] \neq \text{NIL}$
- then add $A[i]$ to the root list of H

18. if $\min[H] = \text{NIL}$ or $\text{key}[A[i]] < \text{key}[\min[H]]$
19. then $\min[H] \leftarrow A[i]$

FIB-HEAP-LINK(H, y, x)

1. remove y from the root list of H
2. make y a child of x , incrementing $\text{degree}[x]$
3. $\text{mark}[y] \leftarrow \text{FALSE}$

Que 2.25. Define Fibonacci heap. Discuss the structure of a Fibonacci heap with the help of a diagram. Write a function for uniting two Fibonacci heaps.

Answer

Fibonacci heap : Refer Q. 2.23, Page 2-44B, Unit-2.

Structure of Fibonacci heap :**1. Node structure :**

- a. The field “mark” is True if the node has lost a child since the node became a child of another node.
- b. The field “degree” contains the number of children of this node. The structure contains a doubly-linked list of sibling nodes.

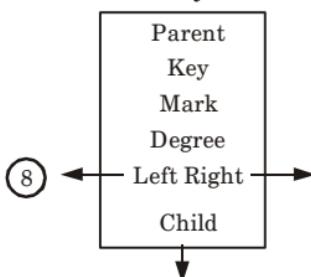


Fig. 2.25.1. Node structure.

2. Heap structure :

min(H) : Fibonacci heap H is accessed by a pointer $\min[H]$ to the root of a tree containing a minimum key; this node is called the minimum node.

If Fibonacci heap H is empty, then $\min[H] = \text{NIL}$.

n(H) : Number of nodes in heap H

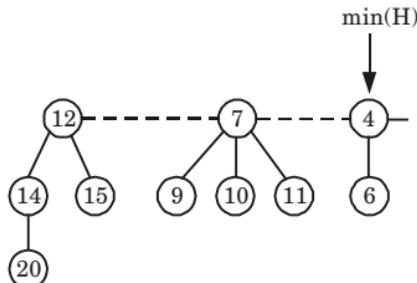


Fig. 2.25.2. Heap structure.

Function for uniting two Fibonacci heap :**Make-Heap :**

MAKE-FIB-HEAP()

$\text{allocate}(H)$
 $\text{min}(H) = \text{NIL}$
 $n(H) = 0$

FIB-HEAP-UNION(H_1, H_2)

1. $H \leftarrow \text{MAKE-FIB-HEAP}()$
2. $\text{min}[H] \leftarrow \text{min}[H_1]$
3. Concatenate the root list of H_2 with the root list of H
4. if ($\text{min}[H_1] = \text{NIL}$) or ($\text{min}[H_2] \neq \text{NIL}$ and $\text{min}[H_2] < \text{min}[H_1]$)
5. then $\text{min}[H] \leftarrow \text{min}[H_2]$
6. $n[H] \leftarrow n[H_1] + n[H_2]$
7. Free the objects H_1 and H_2
8. return H

Que 2.26. Discuss following operations of Fibonacci heap :

- i. **Make-Heap**
- ii. **Insert**
- iii. **Minimum**
- iv. **Extract-Min**

Answer

- i. **Make-Heap** : Refer Q. 2.25, Page 2-46B, Unit-2.
- ii. **Insert** : (H, x)
 1. $\text{degree}[x] \leftarrow 0$
 2. $p[x] \leftarrow \text{NIL}$
 3. $\text{child}[x] \leftarrow \text{NIL}$
 4. $\text{left}[x] \leftarrow x$
 5. $\text{right}[x] \leftarrow x$
 6. $\text{mark}[x] \leftarrow \text{FALSE}$
 7. concatenate the root list containing x with root list H
 8. if $\text{min}[H] = \text{NIL}$ or $\text{key}[x] < \text{key}[\text{min}[H]]$
 9. then $\text{min}[H] \leftarrow x$
 10. $n[H] \leftarrow n[H] + 1$

To determine the amortized cost of FIB-HEAP-INSERT, Let H be the input Fibonacci heap and H' be the resulting Fibonacci heap, then $t(H) = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is, $(t(H) + 1) + 2m(H) - (t(H) + 2m(H)) = 1$

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$

- iii. **Minimum** :

The minimum node of a Fibonacci heap H is always the root node given by the pointer $\text{min}[H]$, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

FIB-HEAP-EXTRACT-MIN(H)

1. $z \leftarrow \text{min}[H]$
2. if $z \neq \text{NIL}$
3. then for each child x of z

4. do add x to the root list of H
5. $p[x] \leftarrow \text{NIL}$
6. remove z from the root list of H
7. if $z = \text{right}[z]$
8. then $\text{min}[H] \leftarrow \text{NIL}$
9. else $\text{min}[H] \leftarrow \text{right}[z]$
10. CONSOLIDATE (H)
11. $n[H] \leftarrow n[H] - 1$
12. return z

PART-5

Tries, Skip List.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 2.27. What is trie ? What are the properties of trie ?

Answer

1. A trie (digital tree / radix tree / prefix free) is a kind of search tree i.e., an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.
2. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.
3. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.
4. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest.

Properties of a trie :

1. Trie is a multi-way tree.
2. Each node has from 1 to d children.
3. Each edge of the tree is labeled with a character.
4. Each leaf node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

Que 2.28. Write an algorithm to search and insert a key in trie data structure.

Answer**Search a key in trie :**

Trie-Search($t, P[k..m]$) // inserts string P into t

1. if t is leaf then return true
2. else if $t.\text{child}(P[k]) = \text{nil}$ then return false
3. else return Trie-Search($t.\text{child}(P[k]), P[k + 1..m]$)

Insert a key in trie :

Trie-Insert($t, P[k..m]$)

1. if t is not leaf then //otherwise P is already present
2. if $t.\text{child}(P[k]) = \text{nil}$ then
//Create a new child of t and a “branch” starting with that child and storing $P[k..m]$
3. else Trie-Insert($t.\text{child}(P[k]), P[k + 1..m]$)

Que 2.29. What is skip list ? What are its properties ?**Answer**

1. A skip list is built in layers.
2. The bottom layer is an ordinary ordered linked list.
3. Each higher layer acts as an “express lane”, where an element in layer i appears in layer $(i + 1)$ with some fixed probability p (two commonly used values for p are $\frac{1}{2}$ and $\frac{1}{4}$).
4. On average, each element appears in $1/(1-p)$ lists, and the tallest element (usually a special head element at the front of the skip list) in all the lists.
5. The skip list contains $\log_{1/p} n$ (i.e., logarithm base $1/p$ of n).

Properties of skip list :

1. Some elements, in addition to pointing to the next element, also point to elements even further down the list.
2. A level k element is a list element that has k forward pointers.
3. The first pointer points to the next element in the list, the second pointer points to the next level 2 element, and in general, the i^{th} pointer points to the next level i element.

Que 2.30. Explain insertion, searching and deletion operation in skip list.**Answer****Insertion in skip list :**

1. We will start from highest level in the list and compare key of next node of the current node with the key to be inserted.

2. If key of next node is less than key to be inserted then we keep on moving forward on the same level.
3. If key of next node is greater than the key to be inserted then we store the pointer to current node i at $\text{update}[i]$ and move one level down and continue our search.

At the level 0, we will definitely find a position to insert given key.

Insert(list, searchKey)

1. local $\text{update}[0..\text{MaxLevel}+1]$
2. $x := \text{list} \rightarrow \text{header}$
3. for $i := \text{list} \rightarrow \text{level down to } 0$ do
4. while $x \rightarrow \text{forward}[i] \rightarrow \text{key} \neq \text{forward}[i]$
5. $\text{update}[i] := x$
6. $x := x \rightarrow \text{forward}[0]$
7. $\text{lvl} := \text{randomLevel}()$
8. if $\text{lvl} > \text{list} \rightarrow \text{level}$ then
9. for $i := \text{list} \rightarrow \text{level} + 1$ to lvl do
10. $\text{update}[i] := \text{list} \rightarrow \text{header}$
11. $\text{list} \rightarrow \text{level} := \text{lvl}$
12. $x := \text{makeNode}(\text{lvl}, \text{searchKey}, \text{value})$
13. for $i := 0$ to level do
14. $x \rightarrow \text{forward}[i] := \text{update}[i] \rightarrow \text{forward}[i]$
15. $\text{update}[i] \rightarrow \text{forward}[i] := x$

Searching in skip list :

Search(list, searchKey)

1. $x := \text{list} \rightarrow \text{header}$
2. loop invariant : $x \rightarrow \text{key} \leq \text{level down to } 0$ do
3. while $x \rightarrow \text{forward}[i] \rightarrow \text{key} < \text{forward}[i]$
4. $x := x \rightarrow \text{forward}[0]$
5. if $x \rightarrow \text{key} = \text{searchKey}$ then return $x \rightarrow \text{value}$
6. else return failure

Deletion in skip list :

Delete(list, searchKey)

1. local $\text{update}[0..\text{MaxLevel}+1]$
2. $x := \text{list} \rightarrow \text{header}$
3. for $i := \text{list} \rightarrow \text{level down to } 0$ do
4. while $x \rightarrow \text{forward}[i] \rightarrow \text{key} \neq \text{forward}[i]$
5. $\text{update}[i] := x$

6. $x := x \rightarrow \text{forward}[0]$
7. if $x \rightarrow \text{key} = \text{searchKey}$ then
8. for $i := 0$ to $\text{list} \rightarrow \text{level}$ do
9. if $\text{update}[i] \rightarrow \text{forward}[i] \neq x$ then break
10. $\text{update}[i] \rightarrow \text{forward}[i] := x \rightarrow \text{forward}[i]$
11. $\text{free}(x)$
12. while $\text{list} \rightarrow \text{level} > 0$ and $\text{list} \rightarrow \text{header} \rightarrow \text{forward}[\text{list} \rightarrow \text{level}] = \text{NIL}$ do
13. $\text{list} \rightarrow \text{level} := \text{list} \rightarrow \text{level} - 1$

Que 2.31. Given an integer x and a positive number n , use divide and conquer approach to write a function that computes x^n with time complexity $O(\log n)$.

AKTU 2018-19, Marks 10

Answer

Function to calculate x^n with time complexity $O(\log n)$:

```
int power(int x, unsigned int y)
{
    int temp;
    if(y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp * temp;
    else
        return x * temp * temp;
}
```

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Define red-black tree and give its properties.

Ans. Refer Q. 2.1.

Q. 2. Insert the following element in an initially empty RB-Tree. 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now delete 23 and 81.

Ans. Refer Q. 2.7.

Q. 3. Define a B-tree of order m . Explain the searching and insertion algorithm in a B-tree.

Ans. Refer Q. 2.10.

Q. 4 Explain the insertion and deletion algorithm in a red-black tree.

Ans. Insertion algorithm : Refer Q. 2.1.

Deletion algorithm : Refer Q. 2.5.

Q. 5. Using minimum degree ' t ' as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.

Ans. Refer Q. 2.17.

Q. 6. What is binomial heap ? Describe the union of binomial heap.

Ans. Refer Q. 2.19.

Q. 7. What is a Fibonacci heap ? Discuss the applications of Fibonacci heaps.

Ans. Refer Q. 2.23.

Q. 8. What is trie ? Give the properties of trie.

Ans. Refer Q. 2.27.

Q. 9. Explain the algorithm to delete a given element in a binomial heap. Give an example for the same.

Ans. Q. 2.22.

Q. 10. Explain skip list. Explain its operations.

Ans. Skip list : Refer Q. 2.29.

Operations : Refer Q. 2.30.



3

UNIT

Graph Algorithms

CONTENTS

Part-1 :	Divide and Conquer with	3-2B to 3-10B
	Examples such as Sorting, Matrix Multiplication, Convex Hull, Searching	
Part-2 :	Greedy Methods with Examples	3-11B to 3-15B
	such as Optimal Reliability Allocation	
Part-3 :	Knapsack	3-16B to 3-23B
Part-4 :	Minimum Spanning Trees-Prim's	3-23B to 3-33B
	and Kruskal's Algorithm Single	
Part-5 :	Source Shortest	3-33B to 3-41B
	Paths-Dijkstra's and Bellman-Ford Algorithm	

PART- 1

*Divide and Conquer with Examples such as
Sorting, Matrix Multiplication, Convex Hull, Searching.*

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 3.1. Write short note on divide and conquer. Write algorithm for merge sort and quick sort.

Answer**Divide and conquer :**

1. Divide and conquer is an algorithm design paradigm based on multi-branched recursion.
2. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
3. The solutions to the sub-problems are then combined to give a solution to the original problem.
4. Divide and conquer technique can be divided into the following three parts :
 - a. **Divide** : This involves dividing the problem into some sub-problem.
 - b. **Conquer** : Recursively calling sub-problem until it is solved.
 - c. **Combine** : This involves combination of solved sub-problem so that we will get the solution of problem.

Merge Sort : Refer Q. 1.21, Page 1–20B, Unit-1.

Quick Sort : Refer Q. 1.19, Page 1–16B, Unit-1.

Que 3.2. What is matrix chain multiplication problem ? Describe a solution for matrix chain multiplication problem.

Answer

1. Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that can be solved using dynamic programming.
2. MCOP helps to find the most efficient way to multiply given matrices.

3. Solution for matrix chain multiplication problem is Strassen's matrix multiplication.

Strassen's matrix multiplication :

1. It is an application of divide and conquer technique.
2. Suppose we wish to compute the product $C = AB$ where each A, B and C are $n \times n$ matrices.
3. Assuming that n is an exact power of 2. We divide each of A, B and C into four $n/2 \times n/2$ matrices.

Rewriting the equation $C = AB$ as

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \quad \dots(3.2.1)$$

4. For convenience, the sub-matrices of A are labelled alphabetical from left to right, whereas those of B are labelled from top to bottom. So that matrix multiplication is performed.

Equation (3.2.1) corresponds to the four equations :

$$r = ae + bf \quad \dots(3.2.2)$$

$$s = ag + bh \quad \dots(3.2.3)$$

$$t = ce + df \quad \dots(3.2.4)$$

$$u = cg + dh \quad \dots(3.2.5)$$

5. Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products.
6. Using these equations to define a straight-forward divide and conquer strategy. We derive the following recurrence for the time $T(n)$ to multiply two $n \times n$ matrices :

$$T(n) = 8T(n/2) + \theta(n^2)$$

7. Unfortunately, this recurrence has the solution $T(n) = \Theta(n^3)$ and thus, this method is no faster than the ordinary one.

Que 3.3. Describe in detail Strassen's matrix multiplication algorithms based on divide and conquer strategies with suitable example.

Answer

Strassen's matrix multiplication algorithm has four steps :

1. Divide the input matrices A and B into $n/2 \times n/2$ sub-matrices.
2. Using $\Theta(n^2)$ scalar additions and subtraction compute 14 $n/2 \times n/2$ matrices $A_1, B_1, A_2, B_2, \dots, A_7, B_7$.
3. Recursively compute the seven matrix products.

$$P_i = A_i B_i \text{ for } i = 1, 2, 3, \dots, 7$$

4. Compute the desired sub-matrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the P_i matrices using only $\Theta(n^2)$ scalar additions and subtractions.

Suppose, $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ and $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

In Strassen method first compute the $7 n/2 \times n/2$ matrices.

P, Q, R, S, T, U, V as follows :

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

Then the c_{ij} 's are computed using the formulas

$$c_{11} = P + S - T + V$$

$$c_{12} = R + T$$

$$c_{21} = Q + S$$

$$c_{22} = P + R - Q + U$$

As can be seen P, Q, R, S, T, U and V can be computed using 7 matrix multiplication and 10 matrix addition or subtraction and c_{ij} 's require an additional 8 addition or subtraction.

For example :

Suppose,

$$A = \begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 11 \\ 8 & 7 \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$AB = C$$

$$\begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 4 & 11 \\ 8 & 7 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

So,

$$A_{11} = 2, B_{11} = 4$$

$$A_{12} = 9, B_{12} = 11$$

$$A_{21} = 5, B_{21} = 8$$

$$A_{22} = 6, B_{22} = 7$$

Now calculate,

$$S_1 = B_{12} - B_{22} = 5, \quad S_2 = A_{11} + A_{12} = 11,$$

$$S_6 = B_{11} + B_{12} = 15, \quad S_7 = A_{12} - A_{22} = 3$$

$$\begin{array}{ll}
 S_3 = A_{21} + A_{22} = 11, & S_8 = B_{21} + B_{22} = 15 \\
 S_4 = B_{21} - B_{11} = 4, & S_9 = A_{12} - A_{22} = 3 \\
 S_5 = A_{11} + A_{22} = 8, & S_{10} = B_{11} + B_{12} = 15 \\
 R = A_{11} \times S_1 = 10, & P = S_5 \times S_6 = 120 \\
 T = S_2 \times B_{22} = 77, & U = S_7 \times S_8 = 45 \\
 Q = S_3 \times B_{11} = 44, & V = S_9 \times S_{10} = 45 \\
 S = A_{22} \times S_4 = 24 &
 \end{array}$$

Now,

$$\begin{aligned}
 C_{11} &= P + S - T + V = 88 + 24 - 77 + 45 = 80 \\
 C_{12} &= R + T = 10 + 77 = 87 \\
 C_{21} &= Q + S = 44 + 24 = 68 \\
 C_{22} &= P + R - Q + U = 88 + 10 - 44 + 45 = 99
 \end{aligned}$$

Now matrix = $\begin{bmatrix} 80 & 71 \\ 68 & 99 \end{bmatrix}$

Que 3.4. What do you mean by graphs ? Discuss various representations of graphs.

Answer

1. A graph G consists of a set of vertices V and a set of edges E .
2. Graphs are important because any binary relation is a graph, so graphs can be used to represent essentially any relationship.

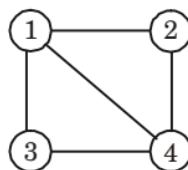


Fig. 3.4.1.

Various representations of graphs :

1. **Matrix representation :** Matrices are commonly used to represent graphs for computer processing. Advantage of representing the graph in matrix lies in the fact that many results of matrix algebra can be readily applied to study the structural properties of graph from an algebraic point of view.

a. Adjacency matrix :

i. Representation of undirected graph :

The adjacency matrix of a graph G with n vertices and no parallel edges is a $n \times n$ matrix $A = [a_{ij}]$ whose elements are given by

$a_{ij} = 1$, if there is an edge between i^{th} and j^{th} vertices
 $= 0$, if there is no edge between them

ii. Representation of directed graph :

The adjacency matrix of a digraph D , with n vertices is the matrix

$A = [a_{ij}]_{n \times n}$ in which
 $a_{ij} = 1$ if arc (v_i, v_j) is in D
 $= 0$ otherwise

b. Incidence matrix :

i. Representation of undirected graph :

Consider an undirected graph $G = (V, E)$ which has n vertices and m edges all labelled. The incidence matrix $I(G) = [b_{ij}]$, is then $n \times m$ matrix, where

$b_{ij} = 1$ when edge e_j is incident with v_i
 $= 0$ otherwise

ii. Representation of directed graph :

The incidence matrix $I(D) = [b_{ij}]$ of digraph D with n vertices and m edges is the $n \times m$ matrix in which.

$b_{ij} = 1$ if arc j is directed away from vertex v_i
 $= -1$ if arc j is directed towards vertex v_i
 $= 0$ otherwise.

2. Linked representation :

- a. In linked representation, the two nodes structures are used :

i. For non-weighted graph,

INFO	Adj-list
------	----------

ii. For weighted graph,

Weight	INFO	Adj-list
--------	------	----------

Where Adj-list is the adjacency list i.e., the list of vertices which are adjacent for the corresponding node.

- b. The header nodes in each list maintain a list of all adjacent vertices of that node for which the header node is meant.

Que 3.5. Explain DFS. Also give DFS algorithm.

Answer

Depth First Search Algorithm :

- Algorithm starts at a specific vertex S in G , which becomes current vertex.
- Then algorithm traverse graph by any edge (u, v) incident to the current vertex u .
- If the edge (u, v) leads to an already visited vertex v , then we backtrack to current vertex u .
- If, on other hand, edge (u, v) leads to an unvisited vertex v , then we go to v and v becomes our current vertex.
- We proceed in this manner until we reach to “dead end”. At this point we start backtracking.

6. The process terminates when backtracking leads back to the start vertex.
7. Edges leading to new vertex are called discovery or tree edges and edges lead to already visited vertex are called back edges.

Algorithm :

In DFS, each vertex v has two timestamps : the first timestamp $d[v]$ i.e., discovery time records when v is first discovered i.e., grayed, and the second timestamp $f[v]$ i.e. finishing time records when the search finishes examining v 's adjacency list i.e., blacked. For every vertex $d[u] < f[u]$.

DFS(G) :

1. for each vertex $u \in V[G]$
2. do $\text{color}[u] \leftarrow \text{WHITE}$
3. $\pi[u] \leftarrow \text{NIL}$
4. time $\leftarrow 0$
5. for each vertex $u \in V[G]$
6. do if $\text{color}[u] = \text{WHITE}$
7. then DFS-VISIT(u)

DFS-VISIT(u) :

1. $\text{color}[u] \leftarrow \text{GRAY}$ // White vertex u has just been discovered.
2. time $\leftarrow \text{time} + 1$
3. $d[u] \leftarrow \text{time}$
4. for each $v \in \text{Adj}[u]$ // Explore edge (u, v)
5. do if $\text{color}[v] = \text{WHITE}$
6. then $\pi[v] \leftarrow u$
7. DFS-VISIT(v)
8. $\text{color}[u] \leftarrow \text{BLACK}$ // Blacken u , it is finished.
9. $f[u] \leftarrow \text{time} \leftarrow \text{time} + 1$

Que 3.6. Explain Breadth First Search (BFS). Give its algorithm.

Answer**Breadth first search :**

1. The general idea behind a breadth first search is as follows :
 - a. First we examine the starting node A .
 - b. Then, we examine all the neighbours of A , and so on.
2. Naturally, we need to keep track of the neighbours of a node, and we need to guarantee that no node is processed more than once.
3. This is accomplished by using a queue to hold nodes that are waiting to be processed.

Algorithm :**BFS (G, s) :**

1. for each vertex $u \in V[G] - \{s\}$
2. do $\text{color}[u] \leftarrow \text{WHITE}$

```
3.            $d[u] \leftarrow \infty$ 
4.            $\pi[u] \leftarrow \text{NIL}$ 
5.            $\text{color}[s] \leftarrow \text{GRAY}$ 
6.            $d[s] \leftarrow 0$ 
7.            $\pi[s] \leftarrow \text{NIL}$ 
8.            $Q \leftarrow \emptyset$ 
9.           ENQUEUE( $Q, s$ )
10.          while  $Q \neq \emptyset$ 
11.            do c
12.              do  $u \leftarrow \text{DEQUEUE}(Q)$ 
13.                for each  $v \in \text{Adj}[u]$ 
14.                  do if color [ $v$ ] = WHITE
15.                      then color [ $v$ ]  $\leftarrow \text{GRAY}$ 
16.                       $d[v] \leftarrow d[u] + 1$ 
17.                       $\pi[v] \leftarrow u$ 
18.                      ENQUEUE( $Q, v$ )
19.            color [ $u$ ]  $\leftarrow \text{BLACK}$ 
```

Que 3.7. Write an algorithm to test whether a given graph is connected or not.

Answer

Test-connected (G) :

1. Choose a vertex x
2. Make a list L of vertices reachable from x ,
and another list K of vertices to be explored.
3. Initially, $L = K = x$.
4. while K is non-empty
5. Find and remove some vertex y in K
6. for each edge (y, z)
7. if (z is not in L)
8. Add z to both L and K
9. if L has fewer than n items
10. return disconnected
11. else return connected.

Que 3.8. Discuss strongly connected components with its algorithm.

Answer

1. The Strongly Connected Components (SCC) of a directed graph G are its maximal strongly connected subgraphs.
2. If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph called as condensation of G .
3. Kosaraju's algorithm is an algorithm to find the strongly connected components of a directed graph.

Kosaraju's algorithm :

1. Let G be a directed graph and S be an empty stack.
2. While S does not contain all vertices :
 - i. Choose an arbitrary vertex v not in S . Perform a depth first search starting at v .
 - ii. Each time that depth first search finishes expanding a vertex u , push u onto S .
3. Reverse the direction of all arcs to obtain the transpose graph.
4. While S is non-empty :
 - i. Pop the top vertex v from S . Perform a depth first search starting at v .
 - ii. The set of visited vertices will give the strongly connected component containing v ; record this and remove all these vertices from the graph G and the stack S .

Que 3.9. Explain convex hull problem.**AKTU 2017-18, Marks 10****OR**

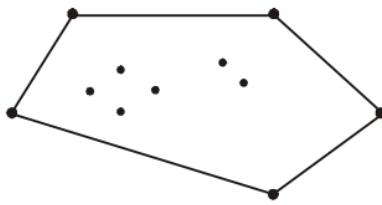
Discuss convex hull. Give Graham-Scan algorithm to compute convex hull.

OR

What do you mean by convex hull ? Describe an algorithm that solves the convex hull problem. Find the time complexity of the algorithm.

AKTU 2019-20, Marks 07**Answer**

1. The convex hull of a set S of points in the plane is defined as the smallest convex polygon containing all the points of S .
2. The vertices of the convex hull of a set S of points form a (not necessarily proper) subset of S .
3. To check whether a particular point $p \in S$ is extreme, see each possible triplet of points and check whether p lies in the triangle formed by these three points.

**Fig. 3.9.1.**

4. If p lies in any triangle then it is not extreme, otherwise it is.
5. We denote the convex hull of S by $\text{CH}(S)$. Convex hull is a convex set because the intersection of convex sets is convex and convex hull is also a convex closure.

Graham-Scan algorithm :

The procedure GRAHAM-SCAN takes as input a set Q of points, where $|Q| \geq 3$. It calls the functions $\text{Top}(S)$, which return the point on top of stack S without changing S , and to $\text{NEXT_TO_TOP}(S)$, which returns the point one entry below the top of stack S without changing S .

GRAHAM-SCAN(Q)

1. Let p_0 be the point in Q with the minimum y -coordinate, or the leftmost such point in case of a tie.
2. Let $\langle p_1, p_2, \dots, p_m \rangle$ be the remaining points in Q , sorted by polar angle in counter clockwise order around p_0 (if more than one point has the same angle remove all but the one that is farthest from p_0).
3. $\text{Top}[S] \leftarrow 0$
4. $\text{PUSH}(p_0, S)$
5. $\text{PUSH}(p_1, S)$
6. $\text{PUSH}(p_2, S)$
7. for $i \leftarrow 3$ to m
8. do while the angle formed by points $\text{NEXT_TO_TOP}(S)$, $\text{Top}(S)$, and p_i makes a non left turn.
9. do $\text{POP}(S)$
10. $\text{PUSH}(p_i, S)$
11. return S

Time complexity :

The worst case running time of GRAHAM-SCAN is

$$T(n) = O(n) + O(n \log n) + O(1) + O(n) = O(n \log n)$$

where

$$n = |Q|$$

Graham's scan running time depends only on the size of the input it is independent of the size of output.

PART-2

Greedy Methods with Examples such as Optimal Reliability Allocation.

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 3.10. Write note on the greedy algorithm.

Answer

1. Greedy algorithms are simple and straight forward.
2. Greedy algorithms are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.
3. Greedy algorithms are easy to invent, easy to implement and most of the time quite efficient.
4. Many problems cannot be solved correctly by greedy approach.
5. Greedy algorithms are used to solve optimization problems.

Que 3.11. What are the four functions included in greedy algorithm ? Write structure of greedy algorithm.

Answer

The greedy algorithm consists of four functions :

- i. A function that checks whether chosen set of items provide a solution.
- ii. A function that checks the feasibility of a set.
- iii. The selection function tells which of the candidates are most promising.
- iv. An objective function, which does not appear explicitly, gives the value of a solution.

Structure of greedy algorithm :

- i. Initially the set of chosen items is empty i.e., solution set.
- ii. At each step
 - a. Item will be added in a solution set by using selection function.
 - b. If the set would no longer be feasible
Reject items under consideration (and is never consider again).
 - c. Else if set is still feasible
add the current item.

Que 3.12. Define activity selection problem and give its solution by using greedy approach with its correctness.

Answer

1. An activity selection is the problem of scheduling a resource among several competing activity. Given a set $S = \{1, 2, \dots, n\}$ of n activities.
2. Each activity has s_i a start time, and f_i a finish time.
3. If activity i is selected, the resource is occupied in the intervals (s_i, f_i) . We say i and j are compatible activities if their start and finish time does not overlap i.e., i and j compatible if $s_i \geq f_j$ and $s_j \geq f_i$
4. The activity selection problem is, to select a maximal sized subset of mutually compatible activities.

Here we maximize the number of activities selected, but if the profit were proportional to $s_i - f_i$, this will not maximize the profit.

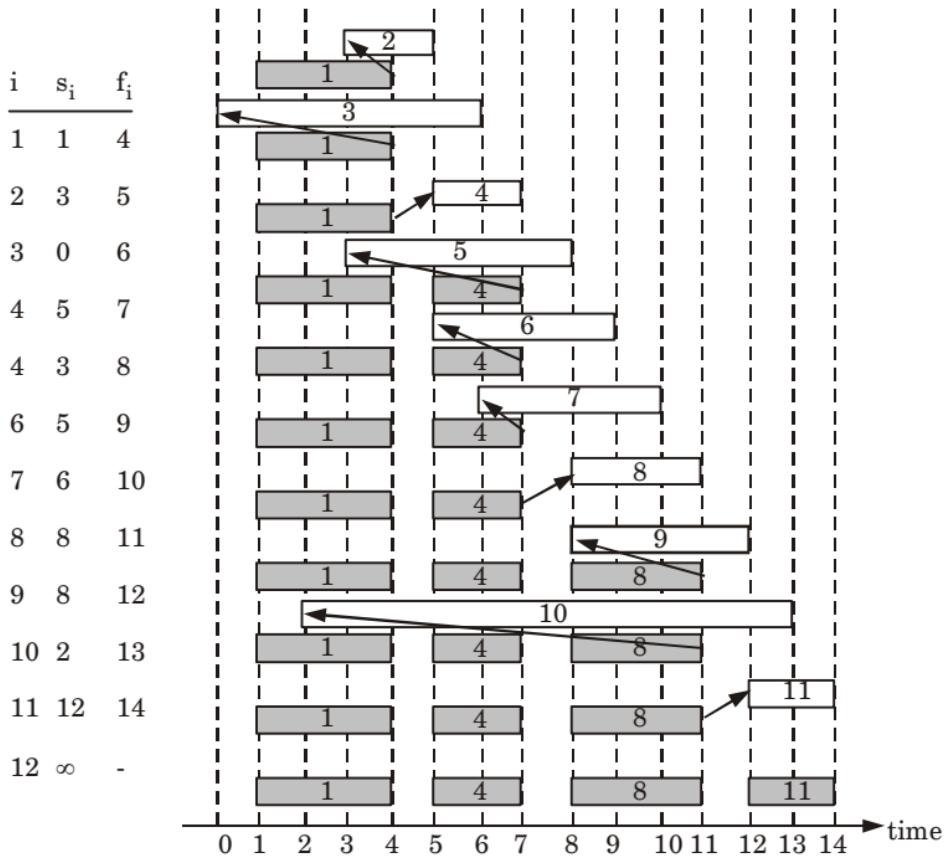


Fig. 3.12.1.

Greedy algorithm :

Assume that $f_1 \leq f_2 \leq \dots \leq f_n$

Greedy-Activity-Selector (s, f)

1. $n \leftarrow \text{length } [s]$
2. $A \leftarrow \{a_1\}$
3. $i \leftarrow 1$
4. for $m \leftarrow 2$ to n
 5. do if $s_m \geq f_i$
 6. then $A \leftarrow A \cup \{(m)\}$
 7. $i \leftarrow m$
8. return A

The algorithm starts with {1} and checks to see which can be added after 1, updating the global “finishing time” and comparing with each start time. The activity picked is always the first that is compatible. Greedy algorithms do not always produce optimal solutions.

Correctness : Greedy algorithm does not always produce optimal solutions but GREEDY-ACTIVITY-SELECTOR does.

Que 3.13. What are greedy algorithms ? Find a solution to the following activity selection problem using greedy technique. The starting and finishing times of 11 activities are given as follows : (2, 3) (8, 12) (12, 14) (3, 5) (0, 6) (1, 4) (6, 10) (5, 7) (3, 8) (5, 9) (8, 11)

OR

What is greedy approach ? Write an algorithm which uses this approach.

Answer

Greedy algorithm : Refer Q. 3.10, Page 3-11B, Unit-3.

Greedy approach : Greedy approach works by making the decision that seems most promising at any moment it never reconsiders this decision, whatever situation may arise later. Activity selection problem uses greedy approach.

Algorithm for greedy activity selection : Refer Q. 3.12, Page 3-12B, Unit-3.

Numerical :

Sorted activities	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
Starting time	2	1	3	0	5	3	5	6	8	8	12
Finish time	3	4	5	6	7	8	9	10	11	12	14

We select first activity a_1
(2, 3)

Check for activity a_2

Starting time of $a_2 \not\geq$ time of a_1

$\therefore a_2$ is not selected

Check for activity a_3

Starting time of $a_3 \geq$ finish time of a_1

$\therefore a_3$ is selected

Check for activity a_4

Starting time of $a_4 \geq$ finish time of a_3

$\therefore a_4$ is not selected

Check for activity a_5

Starting time of $a_5 \geq$ finish time of a_3

$\therefore a_5$ is selected

Check for activity a_6

Starting time of $a_6 \geq$ finish time of a_5

$\therefore a_6$ is not selected

Check for activity a_7

Starting time of $a_7 \geq$ finish time of a_5

$\therefore a_7$ is not selected

Check for activity a_8

Starting time of $a_8 \geq$ finish time of a_5

$\therefore a_8$ is not selected

Check for activity a_9

Starting time of $a_9 \geq$ finish time of a_5

$\therefore a_9$ is selected

Check for activity a_{10}

Starting time of $a_{10} \geq$ finish time of a_9

$\therefore a_{10}$ is not selected

Check for activity a_{11}

Starting time of $a_{11} \geq$ finish time of a_9

$\therefore a_{11}$ is selected.

\therefore Therefore selected activities are :

$a_1 : (2, 3)$

$a_3 : (3, 5)$

$a_5 : (5, 7)$

$a_9 : (8, 11)$

$a_{11} : (12, 14)$

Que 3.14. What is “Greedy Algorithm” ? Write its pseudocode for recursive and iteration process.

Answer

Greedy algorithm : Refer Q. 3.10, Page 3-11B, Unit-3.

Greedy algorithm defined in two different forms :

i. Pseudo code for recursive greedy algorithm : **$R_A_S(s, f, i, j)$**

1. $m \leftarrow i + 1$
2. while $m < j$ and $s_m < f_i$
3. do $m \leftarrow m + 1$
4. if $m < j$
5. then return $\{a_m\} \cup R_A_S(s, f, m, j)$
6. else return \emptyset

ii. Pseudo code for iterative greedy algorithm : **$G_A_S(s, f)$**

1. $n \leftarrow \text{length}[s]$
2. $A \leftarrow [a_1]$
3. $i \leftarrow 1$
4. $m \leftarrow 2$ to n
5. do if $s_m \geq f_i$
6. then $A \leftarrow A \cup \{a_m\}$
7. $i \leftarrow m$
8. return A

Que 3.15. What is an optimization problem ? How greedy method can be used to solve the optimization problem ?

Answer

1. An optimization problem is the problem of finding the best solution from all feasible solutions.
2. Optimization problems can be divided into two categories depending on whether the variables are continuous or discrete.
3. There is no way in general that one can specify if a greedy algorithm will solve a particular optimization problem.
4. However if the following properties can be demonstrated, then it is probable to use greedy algorithm :
 - a. **Greedy choice property :** A globally optimal solution can be arrived at by making a locally optimal greedy choice. That is, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from sub-problems. .
 - b. **Optimal substructure :** A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems.

PART-3*Knapsack.***Questions-Answers****Long Answer Type and Medium Answer Type Questions**

Que 3.16. What is knapsack problem ? Describe an approach used to solve the problem.

Answer

1. The knapsack problem is a problem in combinatorial optimization.
2. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Approach use to solve the problem :

1. In knapsack problem, we have to fill the knapsack of capacity W , with a given set of items $I_1, I_2 \dots I_n$ having weight $w_1, w_2 \dots w_n$ in such a manner that the total weight of items cannot exceed the capacity of knapsack and maximum possible value (v) can be obtained.
2. Using branch and bound approach, we have a bound that none of the items can have total sum more than the capacity of knapsack and must give maximum possible value.
3. The implicit tree for this problem is a binary tree in which left branch implies inclusion and right implies exclusion.
4. Upper bound of node can be calculated as :
$$ub = v + (W - w_{i+1}) (v_{i+1} / w_{i+1})$$

Que 3.17. Write greedy algorithm for discrete knapsack problem with example.

Answer**Greedy algorithm for the discrete knapsack problem :**

1. Compute value/weight ratio v_i/w_i for all items.
2. Sort the items in non-increasing order of the ratios v_i/w_i .
3. Repeat until no item is left in sorted list using following steps :
 - a. If current item fits, use it.
 - b. Otherwise skip this item, and proceed to next item.

For example : Knapsack problem for the following instance using greedy approach. The item can be selected or skipped completely.

Item	Weight	Value
1	7	₹49
2	3	₹12
3	4	₹42
4	5	₹30

Consider $W = 10$.

Solution : This is also called 0/1 knapsack. Either we can completely select an item or skip it. First of all we will compute value-to-weight ratio and arrange them in non-increasing order of the ratio.

Item	Weight	Value	Value/Weight
3	4	₹42	10.5
1	7	₹49	7
4	5	₹30	6
2	3	₹12	4

To fulfill the capacity $W = 10$, we will have

1. Add item of weight 4, $W = 10 - 4 = 6$
2. Skip item of weight 7
3. Add item of weight 5, $W = 6 - 5 = 1$
4. Skip item of weight 3

Maximum value = $10.5 + 6 = 16.5$

But the greedy algorithm does not give optimal solution always rather there is no upper bound on the accuracy of approximate solution.

Que 3.18. What is 0/1-knapsack problem ? Does greedy method effective to solve the 0/1-knapsack problem ?

Answer

The 0/1-knapsack problem is defined as follows :

1. Given, a knapsack of capacity c and n items of weights $\{w_1, w_2, \dots, w_n\}$ and profits $\{p_1, p_2, \dots, p_n\}$, the objective is to choose a subset of n objects that fits into the knapsack and that maximizes the total profit.
2. Consider a knapsack (bag) with a capacity of c .
3. We select items from a list of n items.

4. Each item has both a weight of w_i and profit of p_i .
5. In a feasible solution, the sum of the weights must not exceed the knapsack capacity (c) and an optimal solution is both feasible and reaches the maximum profit.
6. An optimal packing is a feasible solution one with a maximum profit :

$$p_1x_1 + p_2x_2 + p_3x_3 + \dots + p_nx_n = \sum_{i=1}^n p_i x_i$$

which is subjected to constraints :

$$w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_nx_n = \sum_{i=1}^n w_i x_i \leq c$$

and

$$x_i = 1 \text{ or } 0, \quad 1 \leq i \leq n$$

7. We have to find the values of x_i where $x_i = 1$ if i^{th} item is packed into the knapsack and $x_i = 0$ if i^{th} item is not packed.

Greedy strategies for the knapsack problem are :

- i. From the remaining items, select the item with maximum profit that fits into the knapsack.
- ii. From the remaining items, select the item that has minimum weight and also fits into the knapsack.
- iii. From the remaining items, select the one with maximum p_i/w_i that fits into the knapsack.

Greedy method is not effective to solve the 0/1-knapsack problem. By using greedy method we do not get optimal solution.

Que 3.19. Given the six items in the table below and a knapsack with weight 100, what is the solution to the knapsack problem in all concepts. i.e., explain greedy all approaches and find the optimal solution.

Item ID	Weight	Value	Value/Weight
A	100	40	0.4
B	50	35	0.7
C	40	20	0.5
D	20	4	0.2
E	10	10	1
F	10	6	0.6

Answer

We can use 0/1-knapsack problem when the items cannot be divided into parts and fractional knapsack problem when the items can be divided into fractions.

First arrange in non-increasing order of value/weight :

Item ID	Weight	Value	Value/Weight
E	10	10	1
B	50	35	0.7
F	10	6	0.6
C	40	20	0.5
A	100	40	0.4
D	20	4	0.2

According to 0/1-knapsack problem, either we select an item or reject. So the item will be selected according to value per weight.

$$E \text{ is selected} \quad W = 10 < 100$$

$$B \text{ is selected} \quad W = 10 + 50 \\ = 60 < 100$$

$$F \text{ is selected} \quad W = 60 + 10 \\ = 70 < 100$$

C cannot be selected because

$$W = 70 + 40 = 110 > 100$$

Hence we select D

$$W = 70 + 20 = 90 < 100$$

$$\text{Total value} = 10 + 35 + 6 + 4 = 55$$

According to fractional knapsack problem, we can select fraction of any item.

$$E \text{ is selected} \quad W = 10 < 100$$

$$B \text{ is selected} \quad W = 10 + 50 \\ = 60 < 100$$

$$F \text{ is selected} \quad W = 60 + 10 \\ = 70 < 100$$

$$\text{If we select } C \quad W = 70 + 40 \\ = 110 > 100$$

Hence we select the fraction of item C as

$$\frac{100 - W}{\text{Weight of } C} = \frac{100 - 70}{40}$$

$$\text{Weight of } C = 30/40 = 0.75$$

$$\text{So,} \quad W = 0.75 \times 40 = 30 \\ W = 70 + 30 = 100$$

$$\text{Total value} = 10 + 35 + 6 + 0.75(20) \\ = 10 + 35 + 6 + 15 = 66$$

Que 3.20. Consider the weight and values of item listed below.

Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt} . A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy} . Find the value of $V_{\text{opt}} - V_{\text{greedy}}$.

Item	I_1	I_2	I_3	I_4
W	10	7	4	2
V	60	28	20	24

AKTU 2018-19, Marks 07

Answer

For V_{greedy} :

Item	W	V
I_1	10	60
I_2	7	28
I_3	4	20
I_4	2	24

Arrange the items by V/W ratio in descending order :

Item	W	V	V/W
I_4	2	24	12
I_1	10	60	6
I_3	4	20	5
I_2	7	28	4

Total weight $W = 11 \text{ kg}$

I_4 is picked so $W = 11 - 2 = 9 \text{ kg}$

I_1 cannot be picked $10 > 9$

I_3 is picked, $W = 9 - 4 = 5 \text{ kg}$

I_2 cannot be picked $7 > 5$

I_4 and I_3 are picked so

$$V_{\text{greedy}} = V(I_4) + V(I_3) = 24 + 20 = 44$$

For V_{opt} : For calculating V_{opt} we use 0/1 knapsack problem, so only item 1 is picked. Hence, $V_{\text{opt}} = 60$

$$\text{So, } V_{\text{opt}} - V_{\text{greedy}} = 60 - 44 = 16$$

Que 3.21. Consider following instance for simple knapsack problem. Find the solution using greedy method.

$$N = 8$$

$$P = \{11, 21, 31, 33, 43, 53, 55, 65\}$$

$$W = \{1, 11, 21, 23, 33, 43, 45, 55\}$$

$$M = 110$$

AKTU 2016-17, Marks 7.5

Answer

$$N = 8$$

$$W = \{1, 11, 21, 23, 33, 43, 45, 55\}$$

$$P = \{11, 21, 31, 33, 43, 53, 55, 65\}$$

$$M = 110$$

Now, arrange the value of P_i in decreasing order

N	W_i	P_i	$V_i = W_i \times P_i$
1	1	11	11
2	11	21	231
3	21	31	651
4	23	33	759
5	33	43	1419
6	43	53	2279
7	45	55	2475
8	55	65	3575

Now, fill the knapsack according to decreasing value of P_i . First we choose item $N = 1$ whose weight is 1.

Then choose item $N = 2$ whose weight is 11.

Then choose item $N = 3$ whose weight is 21.

Now, choose item $N = 4$ whose weight is 23.

Then choose item $N = 5$ whose weight is 33.

Total weight in knapsack is $= 1 + 11 + 21 + 23 + 33 = 89$

Now, the next item is $N = 6$ and its weight is 43, but we want only 21 because $M = 110$.

So, we choose fractional part of it, i.e.,

The value of fractional part of $N = 6$ is,

$$\frac{2279}{43} \times 21 = 1113$$

21
33
23
21
11
1

} $\Rightarrow 110$

Thus, the maximum value is,

$$\begin{aligned}
 &= 11 + 231 + 651 + 759 + 1419 + 1113 \\
 &= 4184
 \end{aligned}$$

Que 3.22. Solve the following 0/1-knapsack problem using dynamic programming $P = \{11, 21, 31, 33\}$ $w = \{2, 11, 22, 15\}$ $c = 40, n = 4$.

AKTU 2019-20, Marks 07

Answer

Numerical :

$$w = \{2, 11, 22, 15\}$$

$$c = 40$$

$$p = \{11, 21, 31, 33\}$$

Initially,

Item	w_i	p_i
I_1	2	11
I_2	11	21
I_3	22	31
I_4	15	33

Taking value per weight ratio, i.e., p_i/w_i

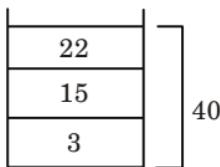
Item	w_i	v_i/w_i	p_i
I_1	2	11	22
I_2	11	21	232
I_3	22	31	682
I_4	15	33	495

Now, arrange the value of p_i in decreasing order.

Item	w_i	p_i	p_i
I_3	22	31	682
I_4	15	33	495
I_2	11	21	232
I_1	2	11	22

Now, fill the knapsack according to decreasing value of p_i .

First we choose item I_3 whose weight is 22, then choose item I_4 whose weight is 15. Now the total weight in knapsack is $22 + 15 = 37$. Now, next item is I_2 and its weight is 11 and then again I_1 . So, we choose fractional part of it, i.e.,



The value of fractional part of I_1 is,

$$= \frac{232}{11} \times 3 = 63$$

Thus, the maximum value is,

$$= 682 + 495 + 63 = 1190$$

PART-4

Minimum Spanning Trees-Prim's and Kruskal's Algorithm, Single.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.23. What do you mean by spanning tree and minimum spanning tree ?

Answer

Spanning tree :

1. A spanning tree of a graph is a subgraph that contains all the vertices and is a tree.
2. A spanning tree of a connected graph G contains all the vertices and has the edges which connect all the vertices. So, the number of edges will be 1 less than the number of nodes.

3. If graph is not connected, i.e., a graph with n vertices has edges less than $n - 1$ then no spanning tree is possible.
4. A graph may have many spanning trees.

Minimum spanning tree :

1. Given a connected weighted graph G , it is often desired to create a spanning tree T for G such that the sum of the weights of the tree edges in T is as small as possible.
2. Such a tree is called a minimum spanning tree and represents the “cheapest” way of connecting all the nodes in G .
3. There are number of techniques for creating a minimum spanning tree for a weighted graph but the most famous methods are Prim’s and Kruskal’s algorithm.

Que 3.24. Write Kruskal’s algorithm to find minimum spanning tree.

Answer

- i. In this algorithm, we choose an edge of G which has smallest weight among the edges of G which are not loops.
- ii. This algorithm gives an acyclic subgraph T of G and the theorem given below proves that T is minimal spanning tree of G . Following steps are required :

Step 1: Choose e_1 , an edge of G , such that weight of e_1 , $w(e_1)$ is as small as possible and e_1 is not a loop.

Step 2: If edges e_1, e_2, \dots, e_i have been selected then choose an edge e_{i+1} not already chosen such that

- i. the induced subgraph

$G[\{e_1, \dots, e_{i+1}\}]$ is acyclic and

- ii. $w(e_{i+1})$ is as small as possible

Step 3: If G has n vertices, stop after $n - 1$ edges have been chosen. Otherwise repeat step 2.

If G be a weighted connected graph in which the weight of the edges are all non-negative numbers, let T be a subgraph of G obtained by Kruskal’s algorithm then, T is minimal spanning tree.

Que 3.25. Describe and compare following algorithms to determine the minimum cost spanning tree :

- i. Kruskal’s algorithm
- ii. Prim’s algorithm

OR

Define spanning tree. Write Kruskal’s algorithm or finding minimum cost spanning tree. Describe how Kruskal’s algorithm is different from Prim’s algorithm for finding minimum cost spanning tree.

Answer

Spanning tree : Refer Q. 3.23, Page 3-23B, Unit-3.

- Kruskal's algorithm :** Refer Q. 3.24, Page 3-24B, Unit-3.
- Prim's algorithm :**

First it chooses a vertex and then chooses an edge with smallest weight incident on that vertex. The algorithm involves following steps :

Step 1 : Choose any vertex V_1 of G .

Step 2 : Choose an edge $e_1 = V_1V_2$ of G such that $V_2 \neq V_1$ and e_1 has smallest weight among the edge e of G incident with V_1 .

Step 3 : If edges e_1, e_2, \dots, e_i have been chosen involving end points V_1, V_2, \dots, V_{i+1} , choose an edge $e_{i+1} = V_jV_k$ with $V_j = \{V_1, \dots, V_{i+1}\}$ and $V_k \notin \{V_1, \dots, V_{i+1}\}$ such that e_{i+1} has smallest weight among the edges of G with precisely one end in $\{V_1, \dots, V_{i+1}\}$.

Step 4 : Stop after $n - 1$ edges have been chosen. Otherwise goto step 3.

Comparison :

S.No.	Kruskal's algorithm	Prim's algorithm
1.	Kruskal's algorithm initiates with an edge.	Prim's algorithm initializes with a node.
2.	Kruskal's algorithm selects the edges in a way that the position of the edge is not based on the last step.	Prim's algorithms span from one node to another.
3.	Kruskal's can be used on disconnected graphs.	In Prim's algorithm, graph must be a connected graph.
4.	Kruskal's time complexity in worst case is $O(E \log E)$.	Prim's algorithm has a time complexity in worst case of $O(E \log V)$.

Que 3.26. What do you mean by minimum spanning tree ? Write an algorithm for minimum spanning tree that may generate multiple forest trees and also explain with suitable example.

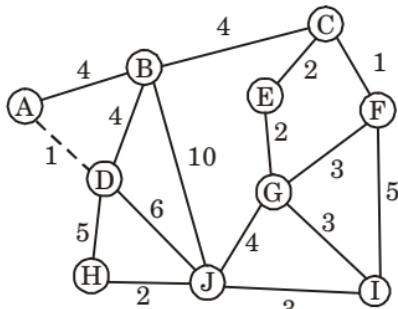
Answer

Minimum spanning tree : Refer Q. 3.23, Page 3-23B, Unit-3.

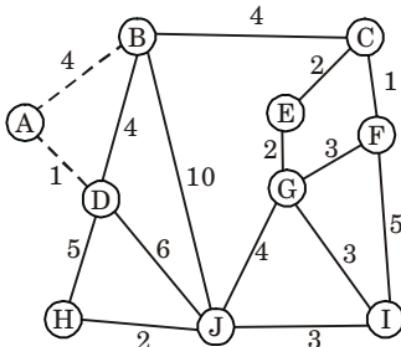
Prim's algorithm : Refer Q. 3.25, Page 3-24B, Unit-3.

For example :

According to algorithm we choose vertex A from the set {A, B, C, D, E, F, G, H, I, J}.



Now edge with smallest weight incident on A is $e = AD$



Now we look on weight

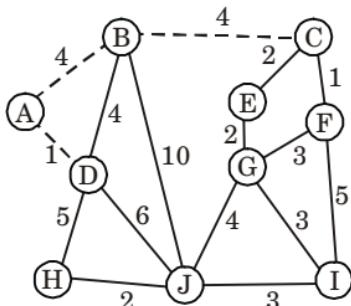
$$W(A, B) = 4$$

$$W(D, B) = 4 \quad W(D, H) = 5$$

$$W(D, J) = 6$$

We choose $e = AB$, since it is minimum.

$W(D, B)$ can also be chosen because it has same value.



Again,

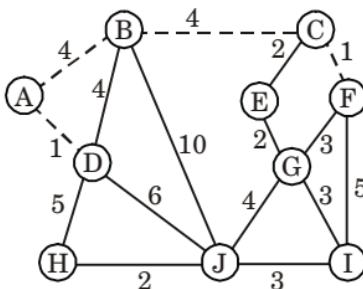
$$W(B, C) = 4$$

$$W(B, J) = 10$$

$$W(D, H) = 5$$

$$W(D, J) = 6$$

We choose $e = BC$, since it has minimum value.

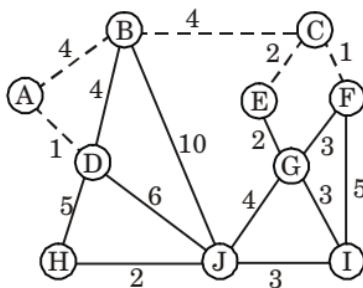


Now, $W(B, J) = 10$

$W(C, E) = 2$

$W(C, F) = 1$

We choose $e = CF$, since it has minimum value.

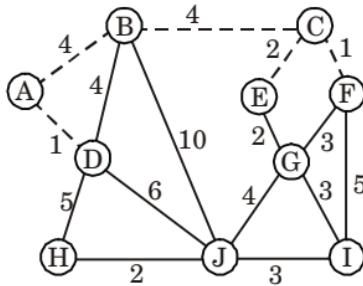


Now, $W(C, E) = 2$

$W(F, G) = 3$

$W(F, I) = 5$

We choose $e = CE$, since it has minimum value.

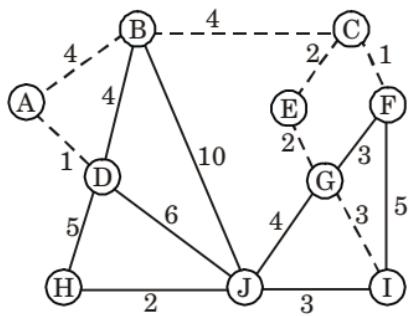


$W(E, G) = 2$

$W(F, G) = 3$

$W(F, I) = 5$

We choose $e = EG$, since it has minimum value.

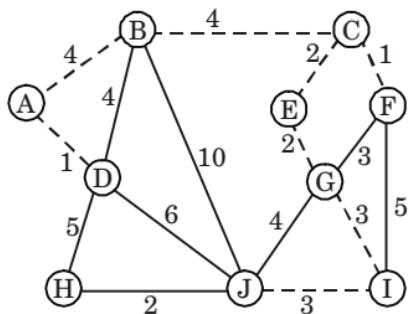


$$W(G, J) = 4$$

$$W(G, I) = 3$$

$$W(F, I) = 5$$

We choose $e = GI$, since it has minimum value.



$$W(I, J) = 3$$

$$W(G, J) = 4$$

We choose $e = IJ$, since it has minimum value.

$$W(J, H) = 2$$

Hence, $e = JH$ will be chosen.

The final minimal spanning tree is given as :

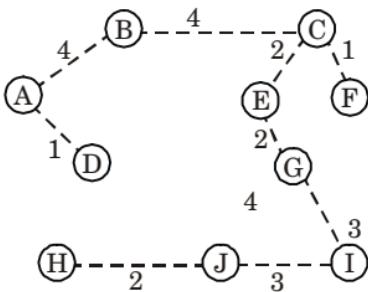


Fig. 3.26.1.

Que 3.27. What is minimum cost spanning tree ? Explain Kruskal's algorithm and find MST of the graph. Also write its time complexity.

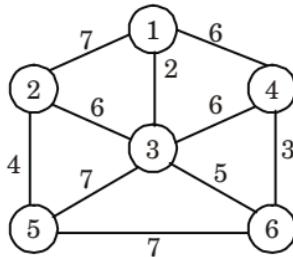


Fig. 3.27.1.

AKTU 2017-18, Marks 10

Answer

Minimum spanning tree : Refer Q. 3.23, Page 3-23B, Unit-3.

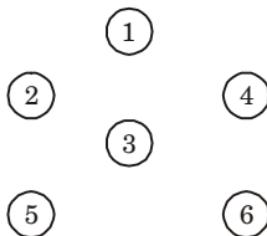
Kruskal's algorithm : Refer Q. 3.24, Page 3-24B, Unit-3.

Numerical :

Step 1 : Arrange the edge of graph according to weight in ascending order.

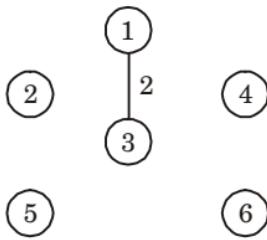
Edges	Weight	Edge	Weight
13	2	32	6
46	3	17	7
25	4	35	7
36	5	56	7
34	6		
41	6		

Step 2 : Now draw the vertices as given in graph,

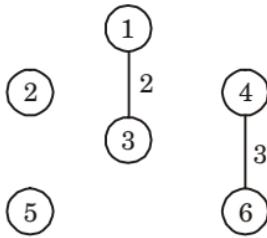


Now draw the edge according to the ascending order of weight. If any edge forms cycle, leave that edge.

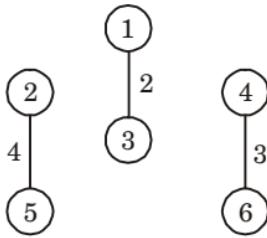
Step 3 : Select edge 13



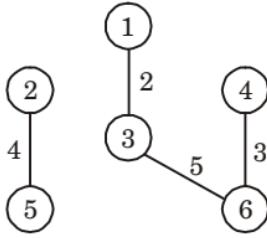
Step 4 : Select edge 46



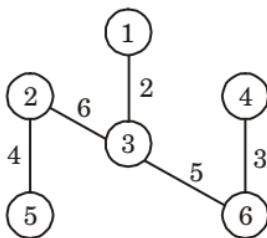
Step 5 : Select edge 25



Step 6 : Select edge 36

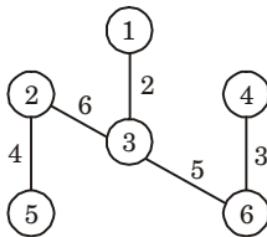


Step 7 : Select edge 23



All the remaining edges, such as 34, 41, 12, 35, 56 are rejected because they form cycle.

All the vertices are covered in this tree. So, the final tree with minimum cost of given graph is



Minimum cost = $2 + 3 + 4 + 5 + 6 = 20$

Time complexity : Time complexity is $O(|E| \log |E|)$.

Que 3.28. What is minimum spanning tree ? Explain Prim's algorithm and find MST of graph Fig. 3.28.1.

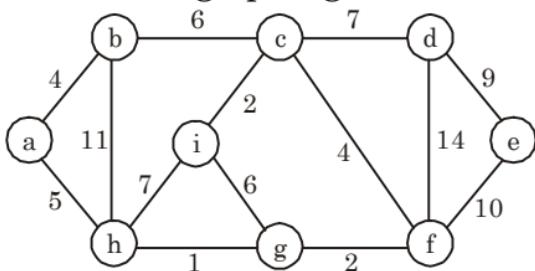


Fig. 3.28.1.

AKTU 2015-16, Marks 05

Answer

Minimum spanning tree : Refer Q. 3.23, Page 3-23B, Unit-3.

Prim's algorithm : Refer Q. 3.25, Page 3-24B, Unit-3.

Numerical :

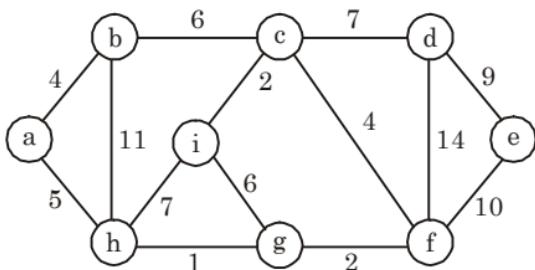
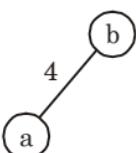
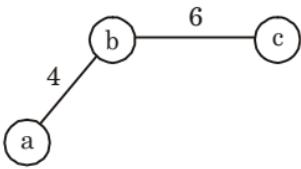


Fig. 3.28.2.

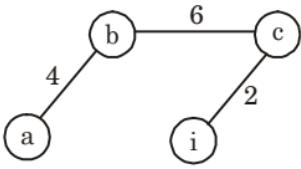
Let a be the source node. Select edge (a, b) as distance between edge (a, b) is minimum.



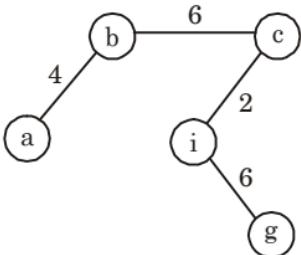
Now, select edge (b, c)



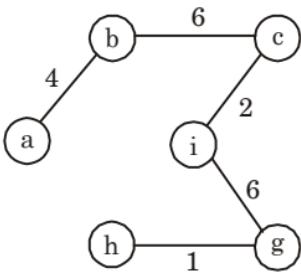
Now, select edge (c, i)



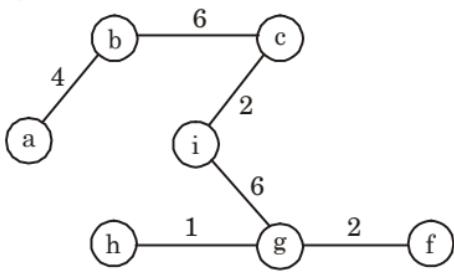
Now, select edge (i, g)



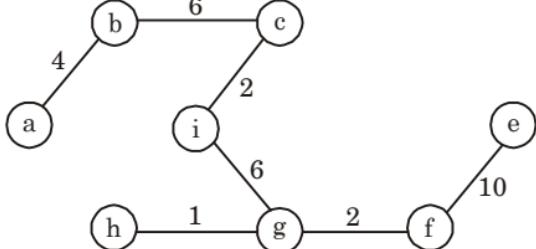
Now, select edge (g, h)



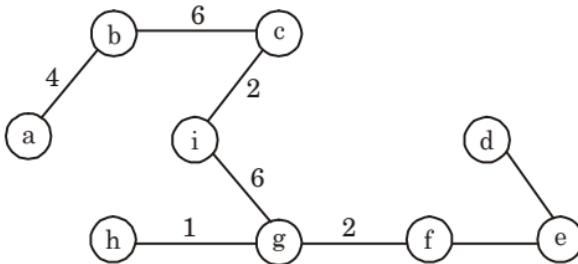
Now, select edge (g, f)



Now, select edge (f, e)



Now, select edge (e, d)



Thus, we obtained MST for Fig. 3.28.1.

PART-5

Source Shortest Paths-Dijkstra's and Bellman-Ford Algorithm.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

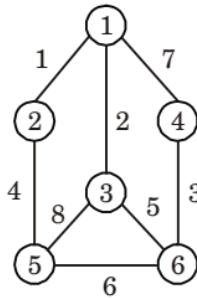
Que 3.29. Prove that the weights on the edge of the connected undirected graph are distinct then there is a unique minimum spanning tree. Give an example in this regard. Also discuss prim's minimum spanning tree algorithm in detail.

AKTU 2018-19, Marks 07

Answer

Proof :

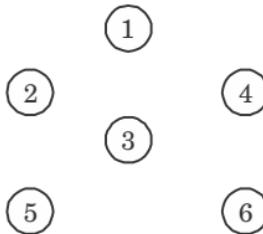
1. Let we have an algorithm that finds an MST (which we will call A) based on the structure of the graph and the order of the edges when ordered by weight.
 2. Assume MST A is not unique.
 3. There is another spanning tree with equal weight, say MST B.
 4. Let e_1 be an edge that is in A but not in B.
 5. Then, B should include at least one edge e_2 that is not in A.
 6. Assume the weight of e_1 is less than that of e_2 .
 7. As B is a MST, $\{e_1\} \cup B$ must contain a cycle.
 8. Replace e_2 with e_1 in B yields the spanning tree $\{e_1\} \cup B - \{e_2\}$ which has a smaller weight compared to B.
 9. This contradicts that B is not a MST.
- So, MST of undirected graph with distinct edge is unique.

Example :**Fig. 3.29.1.**

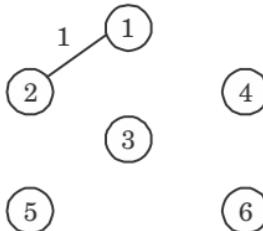
Step 1 : Arrange the edge of graph according to weight in ascending order.

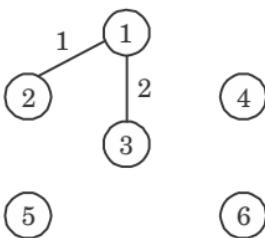
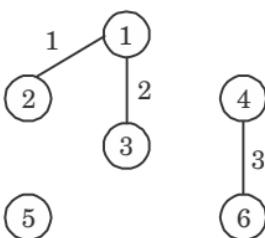
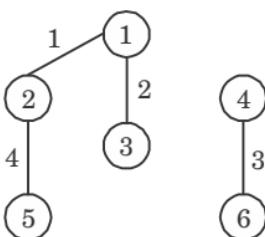
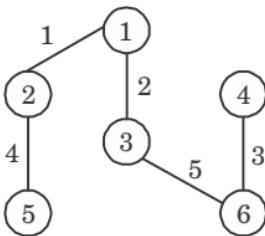
Edges	Weight	Edge	Weight
12	1	14	7
13	2	35	8
46	3		
25	4		
36	5		
56	6		

Step 2 : Now draw the vertices as given in graph,



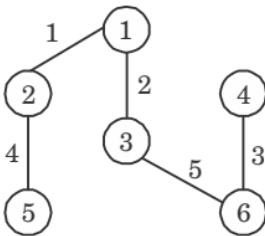
Now draw the edge according to the ascending order of weight. If any edge forms cycle, leave that edge.

Step 3 :

Step 4 :**Step 5 :****Step 6 :****Step 7 :**

All the remaining edges, such as : 14, 35, 56 are rejected because they form cycle.

All the vertices are covered in this tree. So, the final tree with minimum cost of given graph is



Prim's algorithm : Refer Q. 3.25, Page 3-24B, Unit-3.

Que 3.30. Write an algorithm to find shortest path between all pairs of nodes in a given graph.

OR

Explain greedy single source shortest path algorithm with example.

AKTU 2015-16, Marks 10

OR

Write short note on Dijkstra's algorithm shortest paths problems.

AKTU 2016-17, Marks 10

Answer

1. Dijkstra's algorithm, is a greedy algorithm that solves the single source shortest path problem for a directed graph $G = (V, E)$ with non-negative edge weights, i.e., we assume that $w(u, v) \geq 0$ each edge $(u, v) \in E$.
2. Dijkstra's algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined.
3. That is, for all vertices $v \in S$, we have $d[v] = \delta(s, v)$.
4. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest path estimate, inserts u into S , and relaxes all edges leaving u .
5. We maintain a priority queue Q that contains all the vertices in $V - s$, keyed by their d values.
6. Graph G is represented by adjacency list.
7. Dijkstra's always chooses the "lightest or "closest" vertex in $V - S$ to insert into set S that it uses as a greedy strategy.

Dijkstra's algorithm :

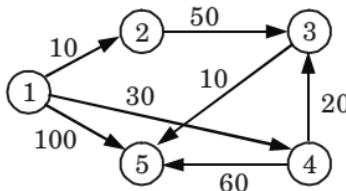
DIJKSTRA(G, w, s)

1. INITIALIZE-SINGLE-SOURCE(G, s)
2. $s \leftarrow \emptyset$
3. $Q \leftarrow V[G]$
4. while $Q \neq \emptyset$
 5. do $u \leftarrow \text{EXTRACT-MIN}(Q)$
 6. $S \leftarrow S \cup \{u\}$
 7. for each vertex $v \in \text{Adj}[u]$
 8. do RELAX(u, v, w)

RELAX(u, v, w) :

1. If $d[u] + w(u, v) < d[v]$
2. then $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

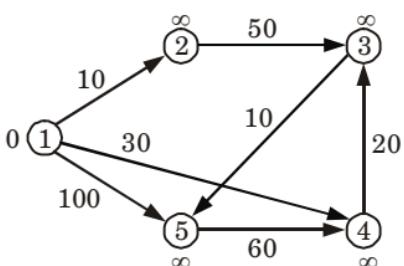
Que 3.31. Find the shortest path in the below graph from the source vertex 1 to all other vertices by using Dijkstra's algorithm.



AKTU 2017-18, Marks 10

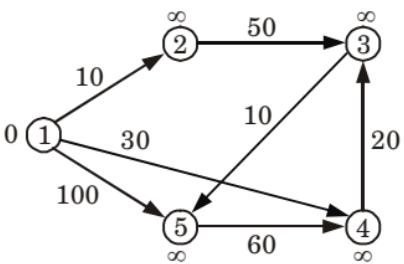
Answer

Initialize :



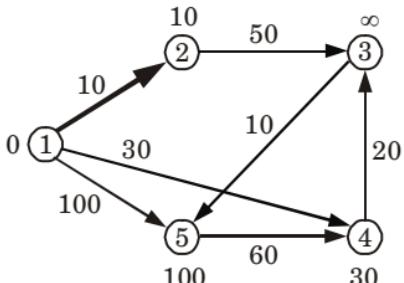
S : {}	Q : 1	2	3	4	5
	0	∞	∞	∞	∞

Extract min (1) :

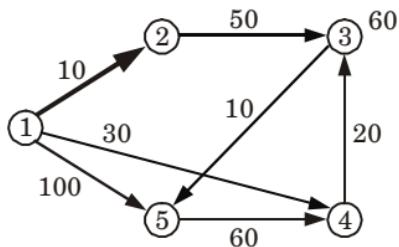


S : {1}	Q : 1	2	3	4	5
	0	∞	∞	∞	∞

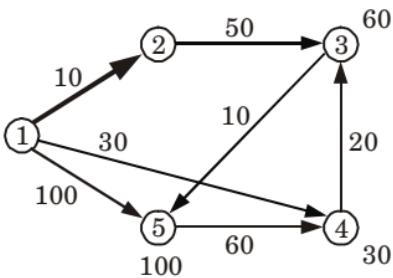
All edges leaving (1) :



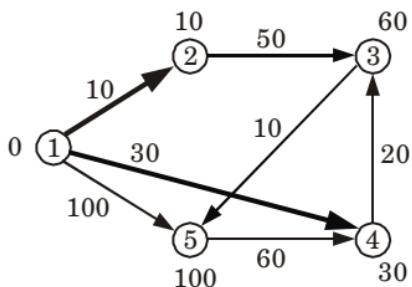
S : {1}	Q : 1	2	3	4	5
	0	∞	∞	∞	∞
	10	∞	30	100	

Extract min(2) :

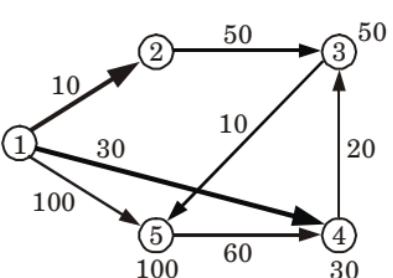
S : {1, 2}				
Q : 1 2 3 4 5				
0	∞	∞	∞	∞
10	60	30	100	

All edges leaving (2) :

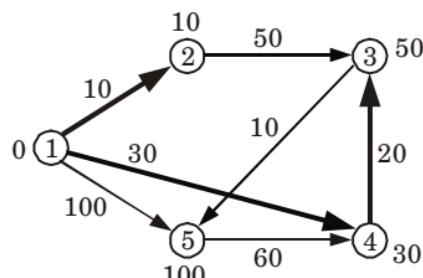
S : {1, 2}				
Q : 1 2 3 4 5				
0	∞	∞	∞	∞
10	60	30	100	
	60	30	100	

Extract min(4) :

S : {1, 2, 4}				
Q : 1 2 3 4 5				
0	∞	∞	∞	∞
10	60	30	100	
	60	30	100	
		30	100	

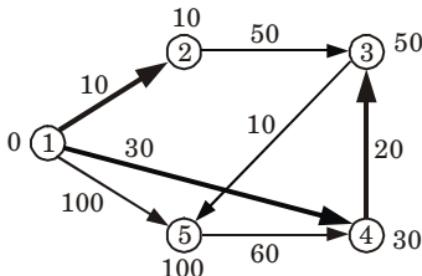
All edges leaving (4) :

S : {1, 2, 4}				
Q : 1 2 3 4 5				
0	∞	∞	∞	∞
10	60	30	100	
	60	30	100	
		30	100	
			50	

Extract min(3) :

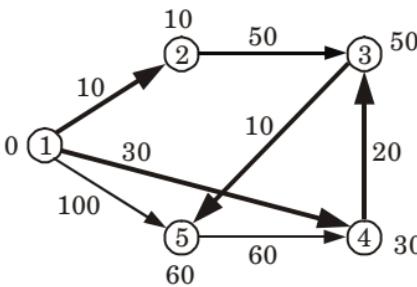
S : {1, 2, 4, 3}				
Q : 1 2 3 4 5				
0	∞	∞	∞	∞
10	∞	30	100	
	60	30	100	
		30	100	
			50	

All edges leaving (3) :



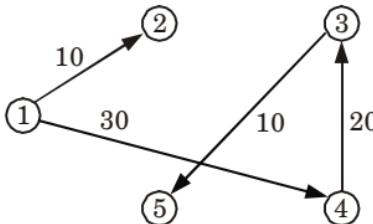
		S : {1, 2, 4, 3, 5}				
		Q : 1	2	3	4	5
	0		∞	∞	∞	∞
	10		∞	30	100	
	60		30		100	
	50				60	

Extract min(5) :



		S : {1, 2, 4, 3, 5}				
		Q : 1	2	3	4	5
	0		∞	∞	∞	∞
	10		∞	30	100	
	60		30		100	
	50				60	
						60

Shortest path



Que 3.32. State Bellman-Ford algorithm.

AKTU 2016-17, Marks 7.5

Answer

1. Bellman-Ford algorithm finds all shortest path length from a source $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.
2. Bellman-Ford algorithm solves the single source shortest path problem in the general case in which edges of a given digraph G can have negative weight as long as G contains no negative cycles.
3. This algorithm, uses the notation of edge relaxation but does not use with greedy method.
4. The algorithm returns boolean TRUE if the given digraph contains no negative cycles that are reachable from source vertex otherwise it returns boolean FALSE.

Bellman-Ford (G, w, s) :

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. for each vertex $i \leftarrow 1$ to $V[G] - 1$
 - 3. do for each edge (u, v) in $E[G]$
 - 4. do RELAX (u, v, w)
5. for each edge (u, v) in $E[G]$ do
6. do if $d[u] + w(u, v) < d[v]$ then
7. then return FALSE
8. return TRUE

RELAX (u, v, w) :

1. If $d[u] + w(u, v) < d[v]$
2. then $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$

If Bellman-Ford returns true, then G forms a shortest path tree, else there exists a negative weight cycle.

Que 3.33. When do Dijkstra and the Bellman Ford algorithm both fail to find a shortest path ? Can Bellman Ford detect all negative weight cycles in a graph ? Apply Bellman Ford algorithm on the following graph :

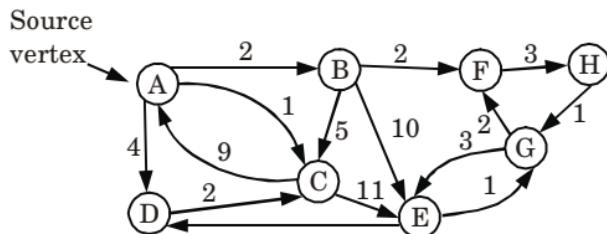


Fig. 3.33.1.

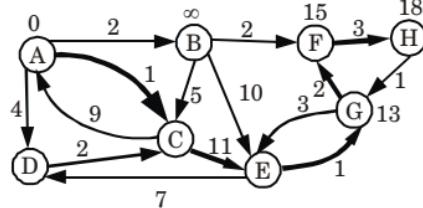
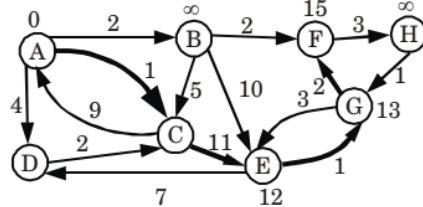
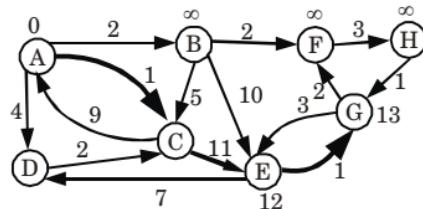
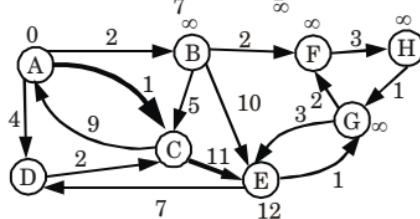
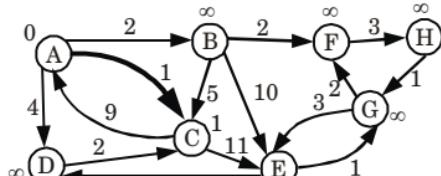
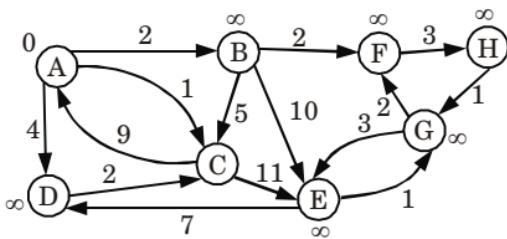
AKTU 2018-19, Marks 07

Answer

Dijkstra algorithm fails to find a shortest path when the graph contains negative edges.

Bellman Ford algorithm fails to find a shortest path when the graph contain negative weight cycle.

No, Bellman Ford cannot detect all negative weight cycle in a graph.

Numerical :

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Discuss matrix chain multiplication problem and its solution.

Ans. Refer Q. 3.2.

Q. 2. Explain graphs with its representations.

Ans. Refer Q. 3.4.

Q. 3. Write short note on convex null problem.

Ans. Refer Q. 3.9.

Q. 4. What is greedy algorithm ? Write its pseudocode for recursive and iterative process.

Ans. Refer Q. 3.14.

Q. 5. Discuss 0/1-knapsack problem.

Ans. Refer Q. 3.18.

Q. 6. Write short note on the following :

- a. Minimum spanning tree
- b. Kruskal's algorithm
- c. Prim's algorithm

Ans.

- a. Refer Q. 3.23.
- b. Refer Q. 3.24.
- c. Refer Q. 3.25.

Q. 7. Write an algorithm to find shortest path between all pairs of nodes in a given graph.

Ans. Refer Q. 3.30.

Q. 8. State Bellman Ford algorithm.

Ans. Refer Q. 3.32.

Q. 9. Solve the following 0/1-knapsack problem using dynamic programming $P = \{11, 21, 31, 33\}$ $w = \{2, 11, 22, 15\}$ $c = 40$, $n = 4$.

Ans. Refer Q. 3.22.



4**UNIT**

Dynamic Programming and Backtracking

CONTENTS

- Part-1 :** Dynamic Programming with **4-2B to 4-8B**
Examples such as Knapsack
- Part-2 :** All Pair Shortest Paths : **4-8B to 4-12B**
Warshall's and Floyd's Algorithm,
Resource Allocation Problem
- Part-3 :** Backtracking, Branch and **4-12B to 4-21B**
Bound with Examples such as
Travelling Salesman Problem
- Part-4 :** Graph Colouring, **4-21B to 4-26B**
N-Queen Problem
- Part-5 :** Hamiltonian Cycles **4-26B to 4-33B**
and Sum of Subsets

PART- 1

Dynamic Programming with Examples such as Knapsack.

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 4.1. What do you mean by dynamic programming ?

OR

What is dynamic programming ? How is this approach different from recursion? Explain with example.

AKTU 2019-20, Marks 07

Answer

1. Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions.
2. It is used when the sub-problems are not independent.
3. Dynamic programming takes advantage of the duplication and arranges to solve each sub-problem only once, saving the solution (in table or something) for later use.
4. Dynamic programming can be thought of as being the reverse of recursion. Recursion is a top-down mechanism i.e., we take a problem, split it up, and solve the smaller problems that are created. Dynamic programming is a bottom-up mechanism i.e., we solve all possible small problems and then combine them to obtain solutions for bigger problems.

Difference :

1. In recursion, sub-problems are solved multiple times but in dynamic programming sub-problems are solved only one time.
2. Recursion is slower than dynamic programming.

For example :

Consider the example of calculating n^{th} Fibonacci number.

$$\text{fibo}(n) = \text{fibo}(n - 1) + \text{fibo}(n - 2)$$

$$\text{fibo}(n - 1) = \text{fibo}(n - 2) + \text{fibo}(n - 3)$$

$$\text{fibo}(n - 2) = \text{fibo}(n - 3) + \text{fibo}(n - 4)$$

.....

.....

.....

$$\text{fibo}(2) = \text{fibo}(1) + \text{fibo}(0)$$

In the first three steps, it can be clearly seen that $\text{fibo}(n - 3)$ is calculated twice. If we use recursion, we calculate the same sub-problems again and again but with dynamic programming we calculate the sub-problems only once.

Que 4.2. What is the principle of optimality? Also give approaches in dynamic programming.

Answer

Principle of optimality : Principle of optimality states that whatever the initial state is, remaining decisions must be optimal with regard to the state following from the first decision.

Approaches in dynamic programming :

There are two approaches of solving dynamic programming problems :

1. **Bottom-up approach :** Bottom-up approach simply means storing the results of certain calculations, which are then re-used later because the same calculation is a sub-problem in a larger calculation.
2. **Top-down approach :** Top-down approach involves formulating a complex calculation as a recursive series of simpler calculations.

Que 4.3. Discuss the elements of dynamic programming.

Answer

Following are the elements of dynamic programming :

1. **Optimal sub-structure :** Optimal sub-structure holds if optimal solution contains optimal solutions to sub-problems. It is often easy to show the optimal sub-problem property as follows :
 - i. Split problem into sub-problems.
 - ii. Sub-problems must be optimal; otherwise the optimal splitting would not have been optimal.

There is usually a suitable “space” of sub-problems. Some spaces are more “natural” than others. For matrix chain multiply we choose sub-problems as sub-chains.

2. Overlapping sub-problem :

- i. Overlapping sub-problem is found in those problems where bigger problems share the same smaller problems. This means, while solving larger problems through their sub-problems we find the same sub-problems more than once. In these cases a sub-problem is usually found to be solved previously.
- ii. Overlapping sub-problems can be found in Matrix Chain Multiplication (MCM) problem.

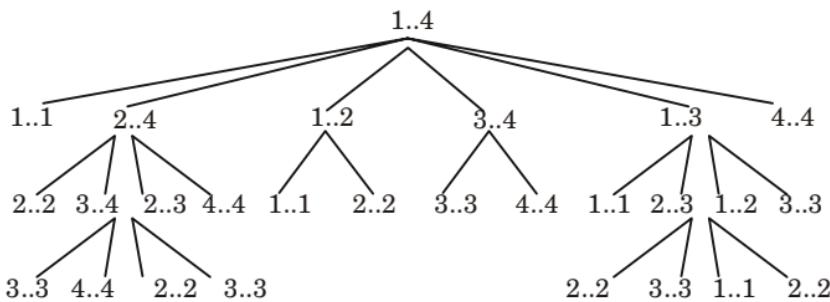


Fig. 4.3.1. The recursion tree for the computation of Recursive-Matrix-Chain (p, 1, 4).

3. Memoization :

- The memoization technique is the method of storing values of solutions to previously solved problems.
- This generally means storing the values in a data structure that helps us reach them efficiently when the same problems occur during the program's execution.
- The data structure can be anything that helps us do that but generally a table is used.

Que 4.4. Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyze its time complexity.

AKTU 2017-18, Marks 10

Answer

LCS-Length (X, Y) :

- $m \leftarrow \text{length}[X]$
- $n \leftarrow \text{length}[Y]$
- for $i \leftarrow 1$ to m
- do $c[i, 0] \leftarrow 0$
- for $j \leftarrow 0$ to n
- do $c[0, j] \leftarrow 0$
- for $i \leftarrow 1$ to m
- do for $j \leftarrow 1$ to n
- do if $x_i = y_j$
- then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
- $b[i, j] \leftarrow "\nwarrow"$
- else if $c[i - 1, j] \geq c[i, j - 1]$
- then $c[i, j] \leftarrow c[i - 1, j]$
- $b[i, j] \leftarrow "\uparrow"$
- else $c[i, j] \leftarrow c[i, j - 1]$
- $b[i, j] \leftarrow "\leftarrow"$

17. return c and b

Note :

1. “↖” means both the same.
2. “↑” means $c[i - 1, j] \geq c[i, j - 1]$.
3. “←” means $c[i - 1, j] < c[i, j - 1]$.
4. The “↖” diagonal arrows lengthen the LCS.

Since, two for loops are present in LCS algorithm first for loop runs upto m times and second for loop runs upto n times. So, time complexity of LCS is $O(mn)$.

Que 4.5. Give the algorithm of dynamic 0/1-knapsack problem.

Answer

Dynamic 0/1-knapsack(v, w, n, W) :

1. for ($w = 0$ to W) $V[0, w] = 0$
2. for ($i = 1$ to n)
3. for ($w = 0$ to W)
4. if ($w[i] \leq w$) then
5. $V[i, w] = \max\{V[i - 1, w], v[i] + V[i - 1, w - w[i]]\};$
6. else $V[i, w] = V[i - 1, w];$
7. return $V[n, W];$

Now, as we know that $V[n, W]$ is the total value of selected items, the can be placed in the knapsack. Following steps are used repeatedly to select actual knapsack item.

Let, $i = n$ and $k = W$ then

while ($1 > 0$ and $k > 0$)

```
{
    if ( $V[i, k] \neq V[i - 1, k]$ ) then
        mark  $i^{\text{th}}$  item as in knapsack
         $i = i - 1$  and  $k = k - w_i$  // selection of  $i^{\text{th}}$  item
    else
         $i = i - 1$  //do not select  $i^{\text{th}}$  item
}
```

Que 4.6. Differentiate between dynamic programming and greedy approach. What is 0/1 knapsack problem ? Solve the following instance using dynamic programming. Write the algorithm also. Knapsack Capacity = 10, $P = < 1, 6, 18, 22, 28 >$ and $w = < 1, 2, 5, 6, 7 >$.

Answer

S. No.	Dynamic programming	Greedy approach
1.	Solves every optimal sub-problem.	Do not solve every optimal problem.
2.	We make a choice at each step, but the choice may depend on the solutions to sub-problem.	We make whatever choice seems best at the moment and then solve the problem.
3.	It is bottom-up approach.	It is top-down approach.
4.	Dynamic programming works when a problem has following properties : a. Optimal sub-structure b. Overlapping sub-problems	Greedy algorithm works when a problem exhibits the following properties : a. Greedy choice property b. Optimal sub-structure

0/1-knapsack problem : Refer Q. 3.18, Page 3–17B, Unit-3.

0/1-knapsack algorithm : Refer Q. 4.5, Page 4–5B, Unit-4.

Numerical :

Item	w_i	$p_i = v_i/w_i$ (Given)	v_i
I_1	1	1	1
I_2	2	6	12
I_3	5	18	90
I_4	6	22	132
I_5	7	28	196

Now, fill the knapsack according to given value of p_i .

First we choose item I_1 whose weight is 1, then choose item I_2 whose weight is 2 and item I_3 whose weight is 5.

∴ Total weight in knapsack : $1 + 2 + 5 = 8$

Now, the next item is I_4 and its weight is 6, but we want only 2 $\{ \because W = 10 \}$. So we choose fractional part of it i.e.,

The value of fractional part of I_4 is $= 132/6 \times 2 = 44$

Thus the maximum value is $= 1 + 12 + 90 + 44 = 147$

$$\left[\begin{array}{c} 2 \\ 5 \\ 2 \\ 1 \end{array} \right] = 10$$

Que 4.7. Discuss knapsack problem with respect to dynamic programming approach. Find the optimal solution for given problem, w (weight set) = {5, 10, 15, 20} and W (Knapsack size) = 25 and v = {50, 60, 120, 100}.

Answer

Knapsack problem with respect to dynamic programming approach : Refer Q. 4.5, Page 4–5B, Unit-4.

Numerical :

$$w = \{5, 10, 15, 20\}$$

$$W = 25$$

$$v = \{50, 60, 120, 100\}$$

Initially,

Item	w_i	v_i
I_1	5	50
I_2	10	60
I_3	15	120
I_4	20	100

Taking value per weight ratio, i.e., $p_i = v_i/w_i$

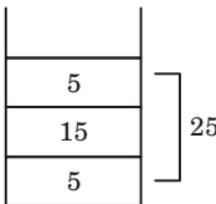
Item	w_i	v_i	$p_i = v_i/w_i$
I_1	5	50	10
I_2	10	60	6
I_3	15	120	8
I_4	20	100	5

Now, arrange the value of p_i in decreasing order.

Item	w_i	v_i	$p_i = v_i/w_i$
I_1	5	50	10
I_3	15	120	8
I_2	10	60	6
I_4	20	100	5

Now, fill the knapsack according to decreasing value of p_i .

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 15. Now the total weight in knapsack is $5 + 15 = 20$. Now, next item is I_2 and its weight is 10, but we want only 5. So, we choose fractional part of it, i.e.,



The value of fractional part of I_2 is,

$$= \frac{60}{10} \times 5 = 30$$

Thus, the maximum value is,

$$= 50 + 120 + 3 = 200$$

Que 4.8. Compare the various programming paradigms such as divide-and-conquer, dynamic programming and greedy approach.

AKTU 2019-20, Marks 07

Answer

S. No.	Divide and conquer approach	Dynamic programming approach	Greedy approach
1.	Optimizes by breaking down a subproblem into simpler versions of itself and using multi-threading and recursion to solve.	Same as Divide and Conquer, but optimizes by caching the answers to each subproblem as not to repeat the calculation twice.	Optimizes by making the best choice at the moment.
2.	Always finds the optimal solution, but is slower than Greedy.	Always finds the optimal solution, but cannot work on small datasets.	Does not always find the optimal solution, but is very fast.
3.	Requires some memory to remember recursive calls.	Requires a lot of memory for tabulation.	Requires almost no memory.

PART-2

All Pair Shortest Paths : Warshall's and Floyd's Algorithm,
Resource Allocation Problem.

Questions-Answers**Long Answer Type and Medium Answer Type Questions**

Que 4.9. **Describe the Warshall's and Floyd's algorithm for finding all pairs shortest paths.**

Answer

1. Floyd-Warshall algorithm is a graph analysis algorithm for finding shortest paths in a weighted, directed graph.
2. A single execution of the algorithm will find the shortest path between all pairs of vertices.
3. It does so in $\Theta(V^3)$ time, where V is the number of vertices in the graph.
4. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.
5. The algorithm considers the “intermediate” vertices of a shortest path, where an intermediate vertex of a simple path $p = (v_1, v_2, \dots, v_m)$ is any vertex of p other than v_1 or v_m , that is, any vertex in the set $\{v_2, v_3, \dots, v_{m-1}\}$.
6. Let the vertices of G be $V = \{1, 2, \dots, n\}$, and consider a subset $\{1, 2, \dots, k\}$ of vertices for some k .
7. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \dots, k\}$, and let p be a minimum-weight path from among them.
8. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \dots, k\}$.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall (W) :

1. $n \leftarrow \text{rows } [W]$
2. $D^{(0)} \leftarrow W$
3. **for** $k \leftarrow 1$ to n
4. **do for** $i \leftarrow 1$ to n
5. **do for** $j \leftarrow 1$ to n
6. **do** $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
7. **return** $D^{(n)}$

Que 4.10. Define Floyd Warshall algorithm for all pair shortest path and apply the same on following graph :

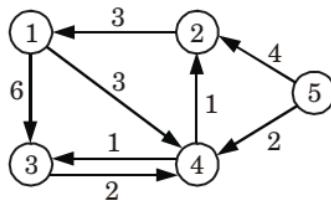


Fig. 4.10.1.

AKTU 2019-20, Marks 07

Answer

Floyd Warshall algorithm : Refer Q. 4.9, Page 4-9B, Unit-4.

Numerical :

$$d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & \infty & 6 & 3 & \infty \\ 2 & 3 & 0 & \infty & \infty & \infty \\ 3 & \infty & \infty & 0 & 2 & \infty \\ 4 & \infty & 1 & 1 & 0 & \infty \\ 5 & \infty & 4 & \infty & 2 & 0 \end{array}$$

$$D^{(1)} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 4 & 6 & 3 & \infty \\ 2 & 3 & 0 & 9 & 6 & \infty \\ 3 & 6 & 4 & 0 & 2 & \infty \\ 4 & 4 & 1 & 1 & 0 & \infty \\ 5 & 7 & 4 & 3 & 2 & 0 \end{array}$$

$$D^{(2)} = \begin{bmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & \begin{bmatrix} 0 & 4 & 6 & 3 & \infty \\ 3 & 0 & 7 & 6 & \infty \\ 6 & 3 & 0 & 2 & \infty \\ 4 & 1 & 1 & 0 & \infty \\ 6 & 3 & 3 & 2 & 0 \end{bmatrix} \\ 2 & \\ 3 & \\ 4 & \\ 5 & \end{bmatrix}$$

Now, if we find $D^{(3)}$, $D^{(4)}$ and $D^{(5)}$ there will be no change in the entries.

Que 4.11. Give Floyd-Warshall algorithm to find the shortest path for all pairs of vertices in a graph. Give the complexity of the algorithm. Explain with example. AKTU 2018-19, Marks 07

Answer

Floyd-Warshall algorithm : Refer Q. 4.9, Page 4-9B, Unit-4.

Time complexity of Floyd-Warshall algorithm is $O(n^3)$.

Example :

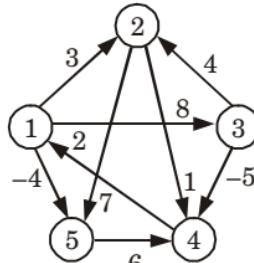


Fig. 4.11.1.

$$d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}; \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}; \pi^{(5)} = \begin{bmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

PART-3

Backtracking, Branch and Bound with Examples such as Travelling Salesman Problem.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.12. What is backtracking? Write general iterative algorithm for backtracking.

AKTU 2016-17, Marks 10

Answer

1. Backtracking is a general algorithm for finding all solutions to some computational problems.
2. Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, and many other puzzles.
3. It is often the most convenient (if not the most efficient) technique for parsing, for the knapsack problem and other combinational optimization problem.
4. It can be applied only for problems which admit the concept of a “partial candidate solution” and a relatively quick test of whether it can possibly be completed to a valid solution.

Iterative backtracking algorithm :

algorithm ibacktrack (n)

// Iterative backtracking process

// All solutions are generated in $x[1 : n]$ and printed as soon as they are found

{

$k = 1;$

while ($k \neq 0$)

{

if (there remains an untried $x[k]$ in $T(x[1], x[2], \dots, x[k-1])$ and $B_k(x[1], \dots, x[k])$ is true)

{

if ($x[1], \dots, x[k]$ is a path to an answer node)

write ($x[1 : k]$);

$k = k + 1;$ // Consider the next set

}

else

$k = k - 1;$ // Backtrack to the previous set

}

}

Que 4.13. Describe backtracking algorithm for Travelling Salesman Problem (TSP). Show that a TSP can be solved using backtracking method in the exponential time.

OR

Explain TSP (Travelling Salesman) problem with example. Write an approach to solve TSP problem.

AKTU 2015-16, Marks 10

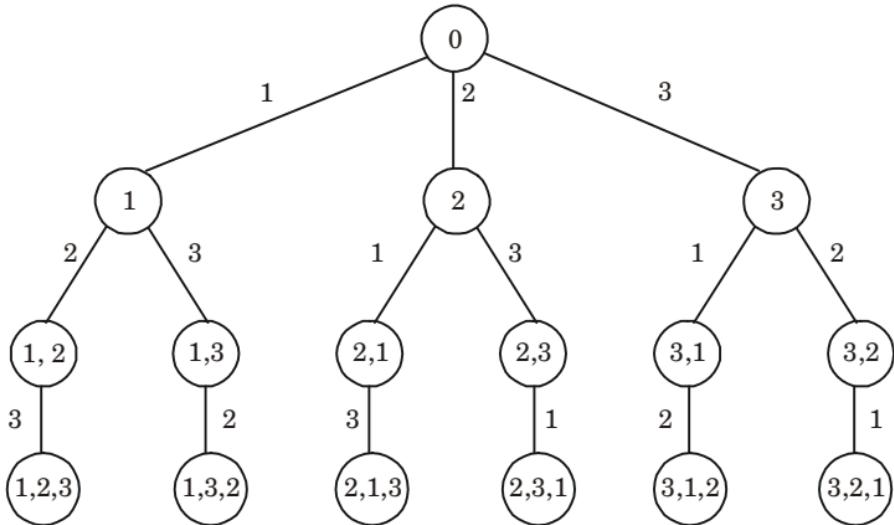
Answer**Travelling Salesman Problem (TSP) :**

Travelling salesman problem is the problem to find the shortest possible route for a given set of cities and distance between the pair of cities that visits every city exactly once and returns to the starting point.

Backtracking approach is used to solve TSP problem.

Backtracking algorithm for the TSP :

1. Let G be the given complete graph with positive weights on its edges.
2. Use a search tree that generates all permutations of $V = \{1 \dots n\}$, specifically the one illustrated in Fig. 4.13.1 for the case $n = 3$.

**Fig. 4.13.1.**

3. A node at depth i of this tree (the root is at depth 0) stores an i -permutation of $\{1, \dots, n\}$. A leaf stores a permutation of $\{1, \dots, n\}$, which is equivalent to saying that it stores a particular Hamiltonian cycle (tour) of G .
4. For the travelling salesman problem, we will not do any static pruning on this tree, we will do dynamic pruning, during the search.

Proof :

1. At some point during the search, let v be a non-leaf node of this tree that is just being visited, and let w be the weight of the shortest tour found to this point.
2. Let $(\pi_1 \pi \dots \pi_k)$ be the k -permutation of $\{1 \dots n\}$ stored at v .

Let, $w_v = \sum_{i=1 \dots k=1} w_{\pi_i \pi_{i+1}}$ denote the sum of the weights on edges whose endpoints are adjacent vertices in this k -permutation.

3. Then, if $w_v \geq w$, the entire subtree of the search tree rooted at v can be pruned, i.e., not searched at all, since every leaf of this subtree represents a tour whose weight must be greater than w_v .
4. This follows from the fact that all edges in the graph have positive weights.
5. There are atmost $O(n^*2^n)$ subproblem, and each one takes linear time to solve.
6. The total running time is therefore $O(n^2*2^n)$.
7. The time complexity is much less than $O(n!)$ but still exponential.
Hence proved.

Que 4.14. Explain the working rule for Travelling Salesman Problem using branch and bound technique.

Answer

For solving travelling salesman problem we represent the solution space by a state space tree. We define three cost functions C , l and u where $l_i \leq C_i \leq u_i$ for all the nodes i of the state space tree.

Step 1 : First find the cost matrix as given cost on the edges of the graph.

Step 2 : Now, find reduced matrix by subtracting the smallest element from row i (column j) introduce a zero in to row i (column j). Repeating this process as often as needed, the cost matrix can be reduced.

Add the total amount subtracted from the column and rows and make this as the root of the state space tree.

Step 3 : Let M be the reduced cost matrix for node A and let B be a child of A such that the tree edge (A, B) corresponds to inclusion of edge (i, j) in the tour. If B is not a leaf, then the reduced cost matrix for B may be obtained by apply following given steps :

- a. Change all the entries in row i , column j of M to ∞ . This includes use of any more edges leaving vertex i or entering vertex j .
- b. Set $M(j, 1) = \infty$. This excludes use of the edge $(j, 1)$.
- c. Reduce all the rows and columns in the resulting matrix except for rows and columns containing only ∞ .

Suppose T is the total amount subtracted in step (c), then

$$I(B) = l(A) + M(i, j) + T$$

For leaf nodes $l = c$ is easily computed as each leaf defines a unique tour. For the upper bound u , we assume $u_i = \infty$ for all nodes i .

Step 4 : Find the root of the node as, combine the total amount subtracted from cost matrix to find the reduced cost matrix M .

Que 4.15. What is travelling salesman problem ? Find the solution of following travelling salesman problem using branch and bound method.

$$\text{Cost matrix} = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 6 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

AKTU 2016-17, Marks 10

Answer

Travelling salesman problem : Refer Q. 4.13, Page 4-13B, Unit-4.
Numerical :

$$\text{Cost matrix} = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 6 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

1. Reduce each column and row by reducing the minimum value from each element in row and column.

Row

Column

$$\begin{array}{l}
 \begin{array}{c}
 \begin{array}{c}
 10 \rightarrow \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \end{bmatrix} \\
 2 \rightarrow \begin{bmatrix} 13 & \infty & 14 & 2 & 0 \end{bmatrix} \\
 2 \rightarrow \begin{bmatrix} 1 & 3 & \infty & 0 & 2 \end{bmatrix} \\
 3 \rightarrow \begin{bmatrix} 16 & 3 & 3 & \infty & 0 \end{bmatrix} \\
 4 \rightarrow \begin{bmatrix} 12 & 0 & 3 & 12 & \infty \end{bmatrix}
 \end{array} \\
 \begin{array}{c}
 \downarrow \quad \quad \quad \downarrow \\
 \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \end{bmatrix} \\
 \begin{bmatrix} 12 & \infty & 11 & 2 & 0 \end{bmatrix} \\
 \begin{bmatrix} 0 & 3 & \infty & 0 & 2 \end{bmatrix} \\
 \begin{bmatrix} 15 & 3 & 0 & \infty & 0 \end{bmatrix} \\
 \begin{bmatrix} 11 & 0 & 0 & 12 & \infty \end{bmatrix}
 \end{array}
 \end{array} \\
 = M_1
 \end{array}$$

2. So, total expected cost is : $10 + 2 + 2 + 3 + 4 + 1 + 3 = 25$.
3. We have discovered the root node V_1 so the next node to be expanded will be V_2, V_3, V_4, V_5 . Obtain cost of expanding using cost matrix for node 2.
4. Change all the elements in 1st row and 2nd column.

$$M_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 0 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

5. Now, reducing M_2 in rows and columns, we get :

$$M_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 0 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

$$\therefore \text{Total cost for } M_2 = 25 + 10 + 0 = 35$$

6. Similarly, for node 3, we have :

$$M_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

7. Now, reducing M_3 , we get :

$$M_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

$$\therefore \text{Total cost for } M_3 = 25 + 17 + 0 = 42$$

8. Similarly, for node 4, we have :

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

9. Now, reducing M_4 , we get :

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 0 + 0 = 25$$

10. Similarly, for node 5, we have :

$$M_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 0 & \infty & \infty \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

11. Now, reducing M_5 , we get :

$$M_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 0 & \infty & \infty \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 1 + 2 = 28$$

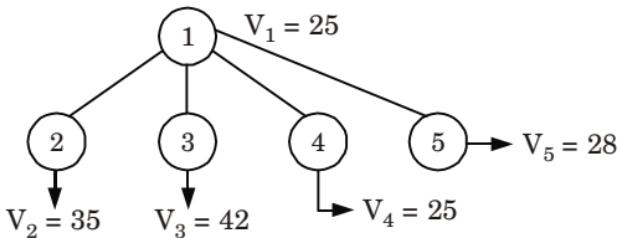


Fig. 4.15.1.

12. Now, the promising node is $V_4 = 25$. Now, we can expand V_2 , V_3 and V_5 . Now, the input matrix will be M_4 .

13. Change all the elements in 4th row and 2nd column.

$$M_6 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

14. On reducing M_6 , we get :

$$M_6 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 3 + 0 = 28$$

15. Similarly, for node 3, we have :

$$M_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

16. On reducing M_7 , we get :

$$M_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 0 + 0 = 25$$

17. Similarly, for node 5, we have :

$$M_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

18. On reducing M_8 , we get :

$$M_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 0 + 11 = 36$$

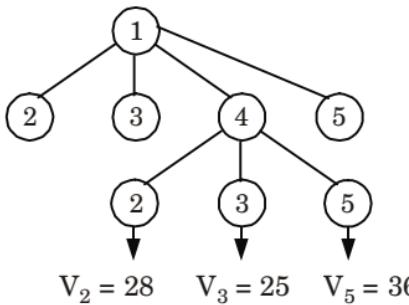


Fig. 4.15.2.

19. Now, promising node is $V_3 = 25$. Now, we can expand V_2 and V_5 . Now, the input matrix will be M_7 .

20. Change all the elements in 3rd row and 2nd column.

$$M_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

21. On reducing M_9 , we get :

$$M_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 3 + 0 = 28$$

22. Similarly, for node 5, we have :

$$M_{10} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

23. On reducing M_{10} , we get :

$$M_{10} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 2 + 12 + 3 = 42.$$

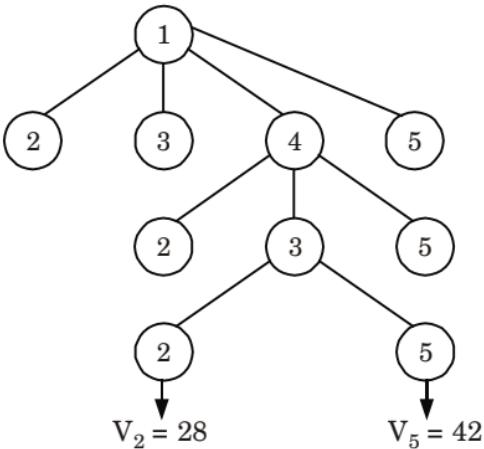


Fig. 4.15.3.

24. Here V_2 is the most promising node so next we are going to expand this node further. Now, we are left with only one node not yet traversed which is V_5 .

$$V_1 \xrightarrow{10} V_4 \xrightarrow{6} V_3 \xrightarrow{5} V_2 \xrightarrow{2} V_5 \xrightarrow{16} V_1$$

So, total cost of traversing the graph is :

$$10 + 6 + 5 + 2 + 16 = 39$$

PART-4

Graph Colouring, N-Queen Problem.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.16. Write short notes on graph colouring.

Answer

1. Graph colouring is a simple way of labelling graph components such as vertices, edges, and regions under some constraints.
2. In a graph, no two adjacent vertices, adjacent edges, or adjacent regions are coloured with minimum number of colours. This number is called the chromatic number and the graph is called a properly coloured graph.
3. While graph colouring, the constraints that are set on the graph are colours, order of colouring, the way of assigning colour, etc.
4. A colouring is given to a vertex or a particular region. Thus, the vertices or regions having same colours form independent sets.

For example :

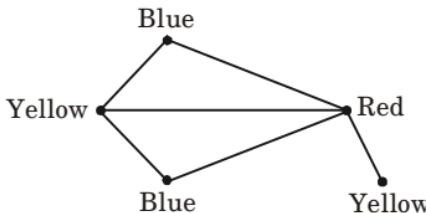


Fig. 4.16.1. Properly coloured graph.

Que 4.17. Write short notes on N-Queens problem.

OR**Write pseudocode for 8-Queens problem.****AKTU 2016-17, Marks 10****Answer**

1. In N -Queens problem, the idea is to place queens one by one in different columns, starting from the leftmost column.
2. When we place a queen in a column, we check for clashes with already placed queens.
3. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.
4. If we do not find such a row due to clashes then we backtrack and return false.

Procedure for solving N -Queens problem :

1. Start from the leftmost column.
2. If all queens are placed return true.
3. Try all rows in the current column. Do following for every tried row :
 - a. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
 - b. If placing queen in [row, column] leads to a solution then return true.
 - c. If placing queen does not lead to a solution then unmark this [row, column] (backtrack) and go to step (a) to try other rows.
4. If all rows have been tried and nothing worked, return false to trigger backtracking.

Algorithm/pseudocode for N -Queens problem :

N -Queens are to be placed on an $n \times n$ chessboard so that no two attack i.e., no two Queens are on the same row, column or diagonal.

PLACE (k, i)

1. for $j \leftarrow 1$ to $k - 1$
2. do if $(x(j) = i)$ or $\text{Abs}(x[j] - i) = (\text{Abs}(j - k))$
3. then return false
4. return true

Place (k, i) returns true if a queen can be placed in the k^{th} row and i^{th} column otherwise return false.

$x[]$ is a global array whose first $k - 1$ values have been set. $\text{Abs}(r)$ returns the absolute value of r .

 N -Queens (k, n)

1. for $i \leftarrow 1$ to n
2. do if PLACE (k, i)
3. then $x[k] \leftarrow i$
4. if $k = n$, then print $x[1 \dots N]$
5. else N -Queens ($k + 1, n$)

[Note : For 8-Queen problem put $n = 8$ in the algorithm.]

Que 4.18. Write an algorithm for solving N -Queens problem.

Show the solution of 4-Queens problem using backtracking approach.

AKTU 2015-16, Marks 10

Answer

Algorithm for N -Queens problem : Refer Q. 4.17, Page 4-21B, Unit-4.

4-Queens problem :

1. Suppose we have 4×4 chessboard with 4-queens each to be placed in non-attacking position.

	1	2	3	4
1				
2				
3				
4				

Fig. 4.18.1.

2. Now, we will place each queen on a different row such that no two queens attack each other.
3. We place the queen q_1 in the very first accept position (1, 1).
4. Now if we place queen q_2 in column 1 and 2 then the dead end is encountered.
5. Thus, the first acceptable position for queen q_2 is column 3 i.e., (2, 3) but then no position is left for placing queen q_3 safely. So, we backtrack one step and place the queen q_2 in (2, 4).
6. Now, we obtain the position for placing queen q_3 which is (3, 2). But later this position lead to dead end and no place is found where queen q_2 can be placed safely.

	1	2	3	4
1	q ₁			
2				q ₂
3		q ₃		
4				

Fig. 4.18.2.

7. Then we have to backtrack till queen q_1 and place it to (1, 2) and then all the other queens are placed safely by moving queen q_2 to (2, 4), queen q_3 to (3, 1) and queen q_4 to (4, 3) i.e., we get the solution $<2, 4, 1, 3>$. This is one possible solution for 4-queens problem.

	1	2	3	4
1		q ₁		
2				q ₂
3	q ₃			
4			q ₄	

Fig. 4.18.3.

8. For other possible solution the whole method is repeated for all partial solutions. The other solution for 4-queens problem is $<3, 1, 4, 2>$ i.e.,

	1	2	3	4
1		q ₁		
2	q ₂			
3				q ₃
4		q ₄		

Fig. 4.18.4.

9. Now, the implicit tree for 4-queen for solution $<2, 4, 1, 3>$ is as follows :
 10. Fig. 4.18.5 shows the complete state space for 4-queens problem. But we can use backtracking method to generate the necessary node and stop if next node violates the rule i.e., if two queens are attacking.

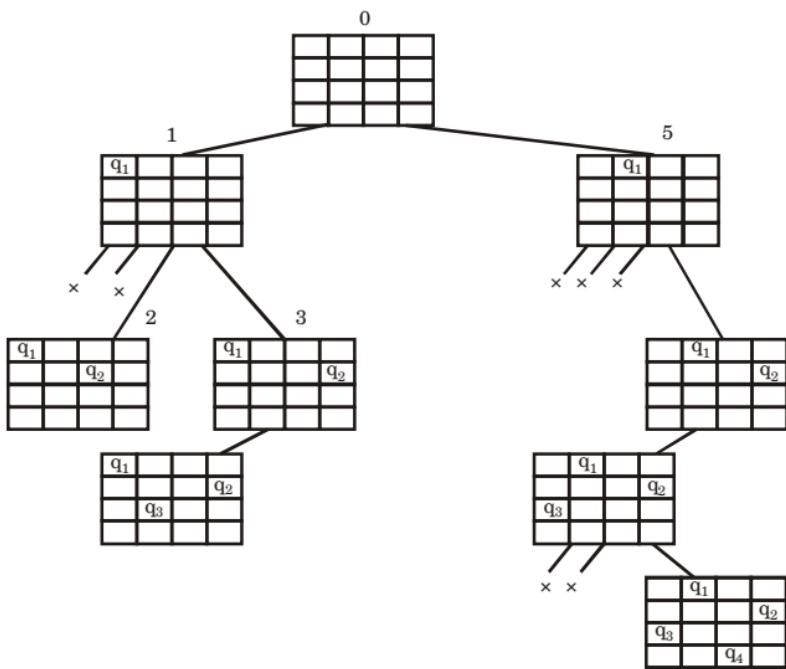


Fig. 4.18.5.

Que 4.19. What is branch and bound technique ? Find a solution to the 4-Queens problem using branch and bound strategy. Draw the solution space using necessary bounding function.

Answer

Branch and bound technique :

1. It is a systematic method for solving optimization problems.
2. It is used where backtracking and greedy method fails.
3. It is sometimes also known as best first search.
4. In this approach we calculate bound at each stage and check whether it is able to give answer or not.
5. Branch and bound procedure requires two tools :
 - a. The first one is a way of covering the feasible region by several smaller feasible sub-regions. This is known as branching.
 - b. Another tool is bounding, which is a fast way of finding upper and lower bounds for the optimal solution within a feasible sub-region.

Solution to 4-Queens problem :

Basically, we have to ensure 4 things :

1. No two queens share a column.

2. No two queens share a row.
3. No two queens share a top-right to left-bottom diagonal.
4. No two queens share a top-left to bottom-right diagonal.

Number 1 is automatic because of the way we store the solution. For number 2, 3 and 4, we can perform updates in O(1) time and the whole updatation process is shown in Fig. 4.19.1.

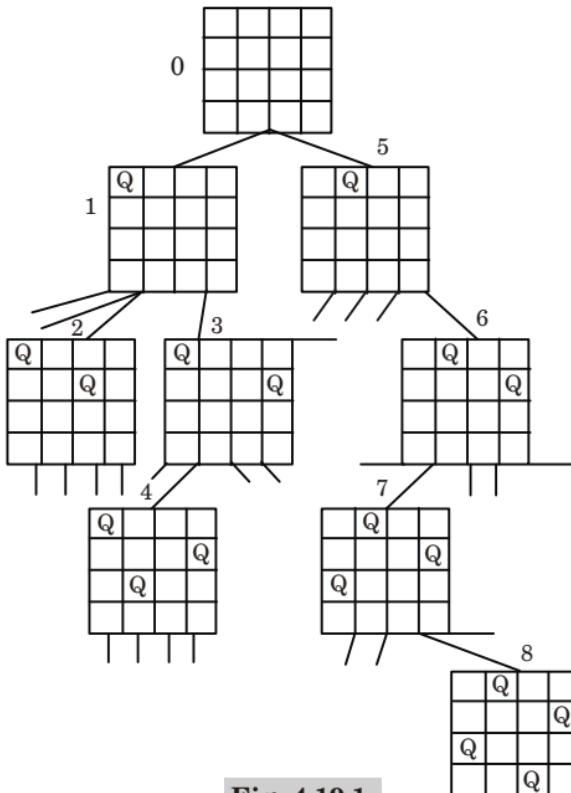


Fig. 4.19.1.

PART-5

Hamiltonian Cycles and Sum of Subsets.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.20. What is backtracking? Discuss sum of subset problem with the help of an example.

AKTU 2017-18, Marks 10

Answer

Backtracking : Refer Q. 4.12, Page 4–13B, Unit-4.

Sum of subset problem with example :

In the subset-sum problem we have to find a subset s' of the given set $S = (S_1, S_2, S_3, \dots, S_n)$ where the elements of the set S are n positive integers in such a manner that $s' \in S$ and sum of the elements of subset ' s' ' is equal to some positive integer 'X'.

Algorithm for sum-subset problem :

Subset-Sum (S, t)

This procedure selects those elements of S whose sum is equal to t . Every time maximum element is found from S , if it is less than t then this element is removed from Z and also it is subtracted from t .

For example :

Given $S = \langle 1, 2, 5, 7, 8, 10, 15, 20, 25 \rangle$ & $m = 35$

$Z \leftarrow S, m = 35$

$$k \leftarrow \max[Z] = 25$$

$$K < m$$

$$Z = Z - K$$

i.e., $Z = \langle 1, 2, 5, 7, 8, 10, 15, 20 \rangle$ & $m_1 \leftarrow m$

Subtracting K from m_1 , we get

$$\text{New } m_1 = m_1(\text{old}) - K = 35 - 25 = 10$$

In new step,

$$K \leftarrow \max[Z] = 20$$

$$K > m_1$$

i.e., $Z = \langle 1, 2, 5, 7, 8, 10, 15 \rangle$

In new step,

$$K \leftarrow \max[Z] = 15$$

$$K > m_1$$

i.e.,

$$Z = \langle 1, 2, 5, 7, 8, 10 \rangle$$

In new step,

$$K \leftarrow \max[Z] = 10$$

$$K > m_1$$

i.e.,

$$Z = \langle 1, 2, 5, 7, 8 \rangle$$

In new step,

$$K \leftarrow \max[Z] = 8$$

$$K > m_1$$

i.e.,

$$Z = \langle 1, 2, 5, 7 \rangle \text{ & } m_2 \leftarrow m_1$$

$$\text{New } m_2 = m_2(\text{old}) - K = 10 - 8 = 2$$

In new step

$$K \leftarrow \max[Z] = 7$$

$$K > m_2$$

i.e.,

$$Z = \langle 1, 2, 5 \rangle$$

In new step,

$$K \leftarrow \max[Z] = 5$$

$$K > m_2$$

i.e.,

$$Z = \langle 1, 2 \rangle$$

In new step,

$$K \leftarrow \max[Z] = 2$$

$$K > m_2$$

i.e.,

$$Z = \langle 1 \rangle$$

In new step,

$$K = 1$$

$$K < m_2$$

∴

$$m_3 = 01$$

Now only those numbers are needed to be selected whose sum is 01, therefore only 1 is selected from Z and rest other number found as $\max[Z]$ are subtracted from Z one by one till Z become \emptyset .

Que 4.21. Solve the subset sum problem using backtracking, where

$$n = 4, m = 18, w[4] = \{5, 10, 8, 13\}.$$

AKTU 2018-19, Marks 07

Answer

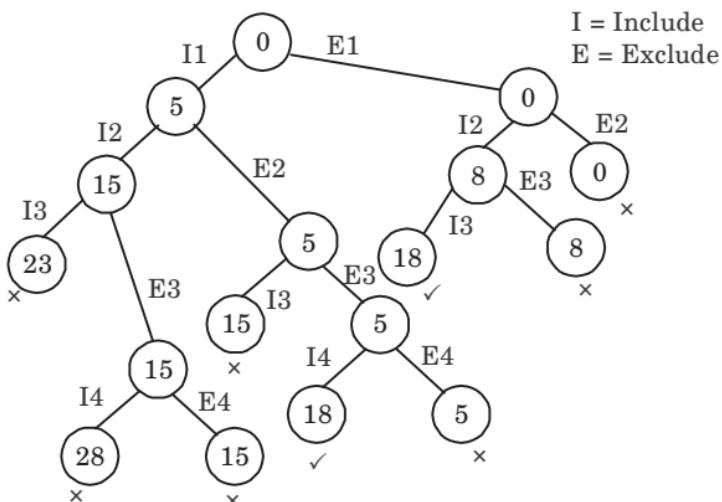
$$n = 4$$

$$m = 18$$

$$w\{4\} = \{5, 10, 8, 13\}$$

Sorted order : $w\{4\} = \{5, 8, 10, 13\}$

Now we construct state-space tree.



First Subset

$$S_1 = \{5, 13\}$$

Similarly,

$$S_2 = \{8, 10\}$$

Que 4.22. Differentiate between backtracking and branch and bound approach. Write an algorithm for sum-subset problem using backtracking approach.

Answer

Difference between backtracking and branch and bound technique :

S.No.	Backtracking	Branch and bound
1.	It is a methodological way of trying out various sequences of decisions.	It is a systematic method for solving optimization problems.
2.	It is applied in dynamic programming technique.	It is applied where greedy and dynamic programming technique fails.
3.	It is sometimes called depth first search.	It is also known as best first search.
4.	This approach is effective for decision problem.	This approach is effective for optimization problems.
5.	This algorithm is simple and easy to understand.	This algorithm is difficult to understand.

Algorithm for sum-subset problem : Refer Q. 4.20, Page 4-26B, Unit-4.

Que 4.23. Explain Hamiltonian circuit problem. Consider a graph $G = (V, E)$ shown in Fig. 4.23.1 and find a Hamiltonian circuit using backtracking method.

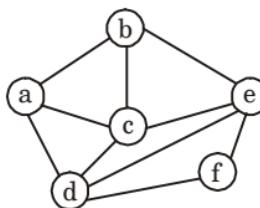


Fig. 4.23.1.

Answer

Hamiltonian circuit problem :

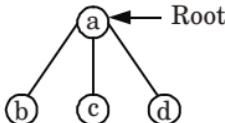
- Given a graph $G = (V, E)$, we have to find the Hamiltonian circuit using backtracking approach.
- We start our search from any arbitrary vertex, say 'a'. This vertex 'a' becomes the root of our implicit tree.
- The first element of our partial solution is the first intermediate vertex of the Hamiltonian cycle that is to be constructed.
- The next adjacent vertex is selected on the basis of alphabetical (or numerical) order.
- If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that dead end is reached.
- In this case we backtrack one step, and again search begins by selecting another vertex and backtrack the element from the partial solution must be removed.
- The search using backtracking is successful if a Hamiltonian cycle is obtained.

Numerical :

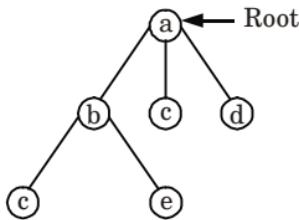
- Firstly, we start our search with vertex 'a', this vertex 'a' becomes the root of our implicit tree.

(a) ← Root

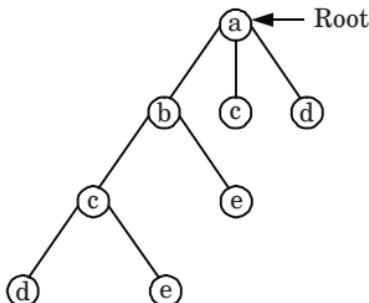
- Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



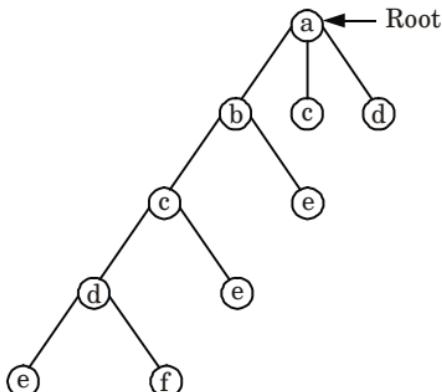
- Next, we select 'c' adjacent to 'b'



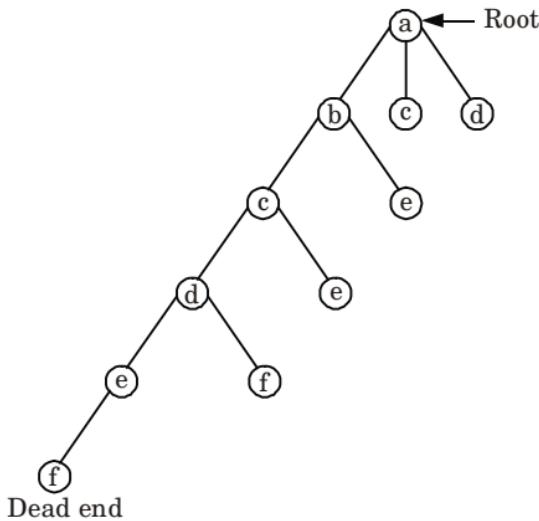
4. Next, we select 'd' adjacent to 'c'



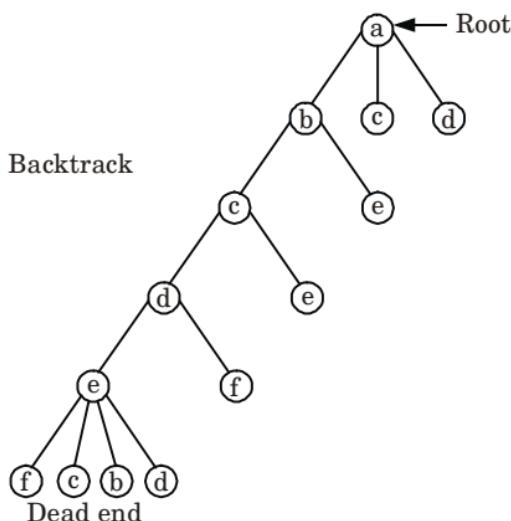
5. Next, we select 'e' adjacent to 'd'

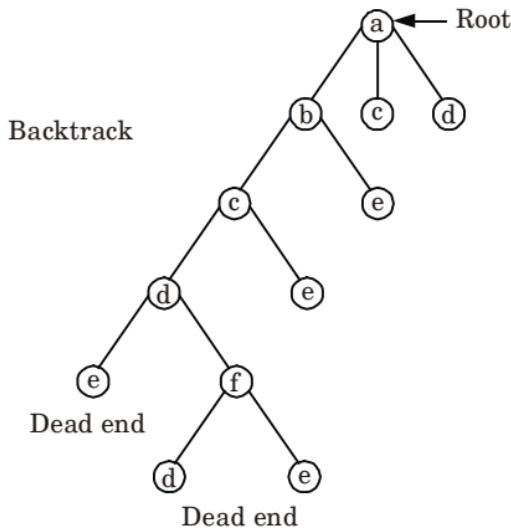


6. Next, we select vertex 'f' adjacent to 'e'. The vertex adjacent to 'f' are 'd' and 'e' but they have already visited. Thus, we get the dead end and we backtrack one step and remove the vertex 'f' from partial solution.

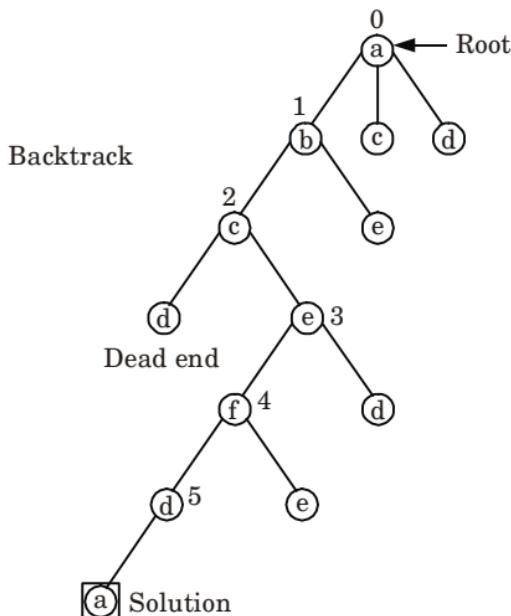


7. From backtracking, the vertex adjacent to 'e' are 'b', 'c', 'd', 'f' from which vertex 'f' has already been checked and 'b', 'c', 'd' have already visited. So, again we backtrack one step. Now, the vertex adjacent to 'd' are 'e', 'f' from which 'e' has already been checked and adjacent of 'f' are 'd' and 'e'. If 'e' vertex visited then again we get dead state. So again we backtrack one step.
8. Now, adjacent to 'c' is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a'. Here, we get the Hamiltonian cycle as all the vertex other than the start vertex 'a' is visited only once ($a - b - c - e - f - d - a$).





Again backtrack



9. Here we have generated one Hamiltonian circuit but other Hamiltonian circuit can also be obtained by considering other vertex.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What do you mean by dynamic programming ?

Ans. Refer Q. 4.1.

Q. 2. Write down an algorithm to compute longest common subsequence (LCS) of two given strings.

Ans. Refer Q. 4.4.

Q. 3. Give the algorithm of dynamic 0/1-knapsack problem.

Ans. Refer Q. 4.5.

Q. 4. Describe the Warshall's and Floyd's algorithm for finding all pair shortest paths.

Ans. Refer Q. 4.9.

Q. 5. What is backtracking ? Write general iterative algorithm for backtracking.

Ans. Refer Q. 4.12.

Q. 6. Write short notes on N-Queens problem.

Ans. Refer Q. 4.17.

Q. 7. Explain travelling salesman problem using backtracking.

Ans. Refer Q. 4.13.

Q. 8. Explain TSP using branch and bound technique.

Ans. Refer Q. 4.14.

Q. 9. Write an algorithm for sum of subset problem.

Ans. Refer Q. 4.20.

Q. 10. Compare the various programming paradigms such as divide-and-conquer, dynamic programming and greedy approach.

Ans. Refer Q. 4.8.



5**UNIT****Selected Topics****CONTENTS**

Part-1	: Algebraic Computation,	5-2B to 5-4B
	Fast Fourier Transform	
Part-2	: String Matching	5-4B to 5-13B
Part-3	: Theory of NP-Completeness	5-13B to 5-24B
Part-4	: Approximation Algorithm	5-24B to 5-30B
Part-5	: Randomized Algorithm	5-30B to 5-32B

PART- 1*Algebraic Computation, Fast Fourier Transform.***Questions-Answers****Long Answer Type and Medium Answer Type Questions**

Que 5.1. What is FFT (Fast Fourier Transformation) ? How the recursive FFT procedure works ? Explain.

Answer

1. The Fast Fourier Transform (FFT) is an algorithm that computes a Discrete Fourier Transform (DFT) of n -length vector in $O(n \log n)$ time.
2. In the FFT algorithm, we apply the divide and conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree $(n - 1)$ polynomial.

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

into two degree $\left(\frac{n}{2} - 1\right)$ polynomials.

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

Where $A^{[0]}$ contains all the even index coefficients of A and $A^{[1]}$ contains all the odd index coefficients and we can combine these two polynomials into A , using the equation,

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \quad \dots(5.1.1)$$

So that the problem of evaluating $A(x)$ at ω_n^k where $k = 0, 1, 2, \dots, n-1$ reduces to,

- Evaluating the degree $\left(\frac{n}{2} - 1\right)$ polynomial $A^{[0]}(x)$ and $A^{[1]}(x)$ at the point $(\omega_n^k)^2$ i.e.,

$$(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$$

because we know that if ω_n^k is a complex n^{th} root of unity then $(\omega_n^k)^2$ is a complex $\frac{n^{th}}{2}$ root of unity. Thus, we can evaluate each $A^{[0]}(x)$ and $A^{[1]}(x)$ at $(\omega_n^k)^2$ values.

- Combining the results according to the equation (5.1.1). This observation is the basis for the following procedure which computes

the DFT of an n -element vector $a = (a_0, a_1, \dots, a_{n-1})$ where for sake of simplicity, we assume that n is a power of 2.

FFT (a, w):

1. $n \leftarrow \text{length } [a]$ n is a power of 2.
2. if $n = 1$
3. then return a
4. $\omega_n \leftarrow e^{2\pi i/n}$
5. $x \leftarrow \omega^0$ x will store powers of ω initially $x = 1$.
6. $a^{[0]} \leftarrow (a_0, a_2, \dots, a_{n-2})$
7. $a^{[1]} \leftarrow (a_1, a_3, \dots, a_{n-1})$
8. $y^{[0]} \leftarrow \text{FFT}(a^{[0]}, \omega^2)$ Recursive calls with ω^2 as $(n/2)^{\text{th}}$ root of unity.
9. $y^{[1]} \leftarrow \text{FFT}(a^{[1]}, \omega^2)$
10. for $k \leftarrow 0$ to $(n/2) - 1$
11. do $y_k \leftarrow y_k^{[0]} + x y_k^{[1]}$
12. $y_{k+(n/2)} \leftarrow y_k^{[0]} - x y_k^{[1]}$
13. $x \leftarrow x \omega_n$
14. return y

Line 2-3 represents the basis of recursion; the DFT of one element is the element itself. Since in this case

$$y_0 = a_0 \omega_1^0 = a_0 1 = a_0$$

Line 6-7 defines the recursive coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. $\omega = \omega_n^k$

Line 8-9 perform the recursive DFT _{$n/2$} computations setting for

$$k = 0, 1, 2, \dots, \frac{n}{2} - 1 \text{ i.e.,}$$

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k}), \quad y_k^{[1]} = A^{[1]}(\omega_n^{2k})$$

Lines 11-12 combine the results of the recursive DFT _{$n/2$} calculations.

For $y_0, y_2, \dots, y_{(n/2)-1}$, line 11 yields.

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) = A(\omega_n^k) \text{ using equation (5.1.1)} \end{aligned}$$

For $y_{n/2}, y_{(n/2)+1}, \dots, y_{n-1}$, line 12 yields.

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} = y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \quad [\because \omega_n^{k+(n/2)} = -\omega_n^k] \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k} \omega_n^n) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k} \omega_n^n) \quad [\because \omega_n^n = 1] \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}) \quad \text{using equation (5.1.1)} \end{aligned}$$

each $k = 0, 1, 2, \dots, (n/2) - 1$.

Thus, the vector y returned by the FFT algorithm will store the values of $A(x)$ at each of the roots of unity.

Que 5.2. What is the application of Fast Fourier Transform (FFT) ? Also write the recursive algorithm for FFT.

AKTU 2018-19, Marks 10

Answer

Application of Fast Fourier Transform :

1. Signal processing.
2. Image processing.
3. Fast multiplication of large integers.
4. Solving Poisson's equation nearly optimally.

Recursive algorithm : Refer Q. 5.1, Page 5-2B, Unit-5.

PART-2

String Matching.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.3. What is string matching ? Discuss string matching problem. Also define string, substring and proper substring.

Answer

String matching is a process of finding one or more occurrences of a pattern in a text.

String matching problem :

Given a text array $T[1 \dots n]$ of n character and a pattern array $P[1 \dots m]$ of m characters.

The problem is to find an integer s , called valid shift where $0 \leq s < n - m$ and $T[s + 1 \dots s + m] = P[1 \dots m]$.

We further assume that the elements of P and T are characters drawn from a finite alphabet Σ such as $\{0, 1\}$ or $\{A, B, \dots, Z, a, b, \dots, z\}$.

String : A string is traditionally a sequence of character, either as a literal constant or as some kind of variable.

Substring : Given a string $T[1 \dots n]$, the substring is defined as $T[i \dots j]$ for some $0 \leq i \leq j \leq n - 1$, that is, the string formed by the characters in T from index j , inclusive. This means that a string is a substring of itself (simply take $i = 0$ and $j = m$).

Proper substring : The proper substring of string $T[1 \dots n]$ is $T[i \dots j]$ for some $0 \leq i \leq j \leq n - 1$, that is, we must have either $i > 0$ or $j < m - 1$.

Using these definition, we can say given any string $T[1 \dots n]$, the substring are

$$T[i \dots j] = T[i] \ T[i+1] \ T[i+2] \dots \ T[j]$$

for some $0 \leq i \leq j \leq n - 1$.

And proper substrings are

$$T[i \dots j] = T[i] \ T[i+1] \ T[i+2] \dots \ T[j]$$

for some $0 \leq i \leq j \leq n - 1$.

Note that if $i > j$, then $T[i \dots j]$ is equal to the empty string or null, which has length zero. Using these notations, we can define of a given string $T[1 \dots n]$ as $T[0 \dots i]$ for some $0 \leq i \leq n - 1$ and suffix of a given string $T[1 \dots n]$ as $T[i \dots n-1]$ for some $0 \leq i \leq n - 1$.

Que 5.4. What are the different types of string matching? Explain one of them.

Answer

Basic types of string matching algorithms are :

String matching algorithm

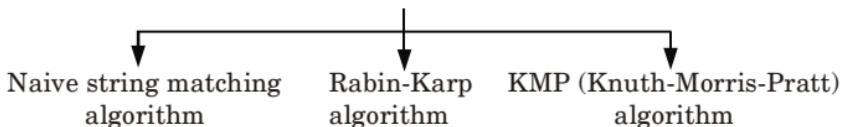


Fig. 5.4.1.

Naive string matching :

The Naive approach simply test all the possible placement of pattern $P[1 \dots m]$ relative to text $T[1 \dots n]$. Specifically, we try shifts $s = [0, 1, \dots, n - m]$, successively and for each shift, s , compare $T[s + 1 \dots s + m]$ to $P[1 \dots m]$.

Naive string matcher (T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. for $s \leftarrow 0$ to $n - m$
4. do if $P[1 \dots m] = T[s + 1 \dots s + m]$
5. then print "pattern occurs with shift" s .

The Naive string matching procedure can be interpreted graphically as a sliding a pattern $P[1 \dots m]$ over the text $T[1 \dots n]$ and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

To analyze the time of Naive matching, the given algorithm is implemented as follows, note that in this implementation, we use notation $P[1 \dots j]$ to denote the substring of P from index i to index j . That is, $P[1 \dots j] = P[i] \ P[i+1] \dots \ P[j]$.

Naive string matcher (T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. for $s \leftarrow 0$ to $n - m$ do
4. $j \leftarrow 1$

5. while $j \leq m$ and $T[s + j] = P[j]$ do
6. $j \leftarrow j + 1$
7. if $j > m$ then
8. return valid shift s
9. return no valid shift exist // i.e., there is no substring of T matching P .

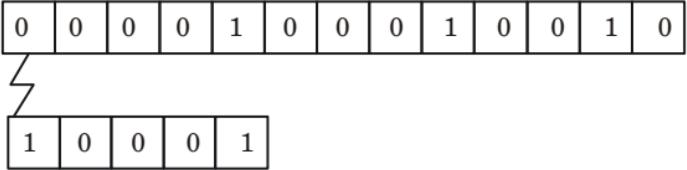
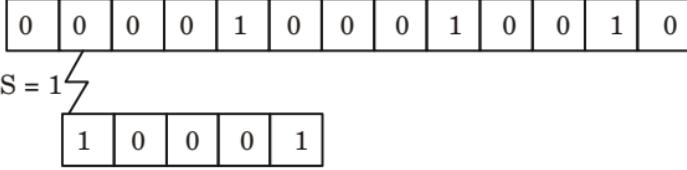
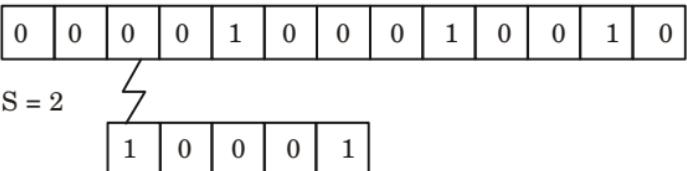
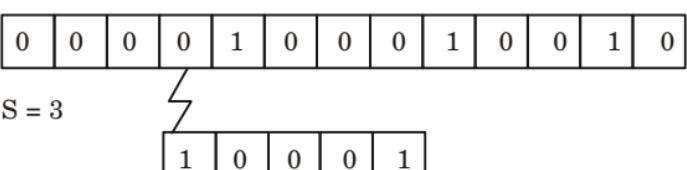
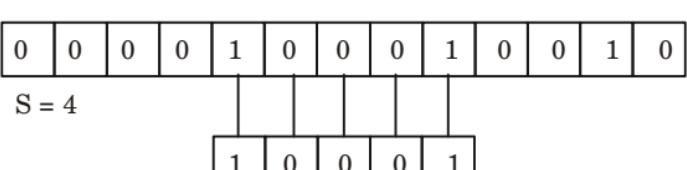
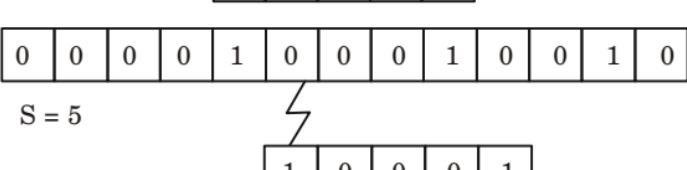
Que 5.5. Show the comparisons that Naive string matcher makes for the pattern $P = \{10001\}$ in the text $T = \{0000100010010\}$

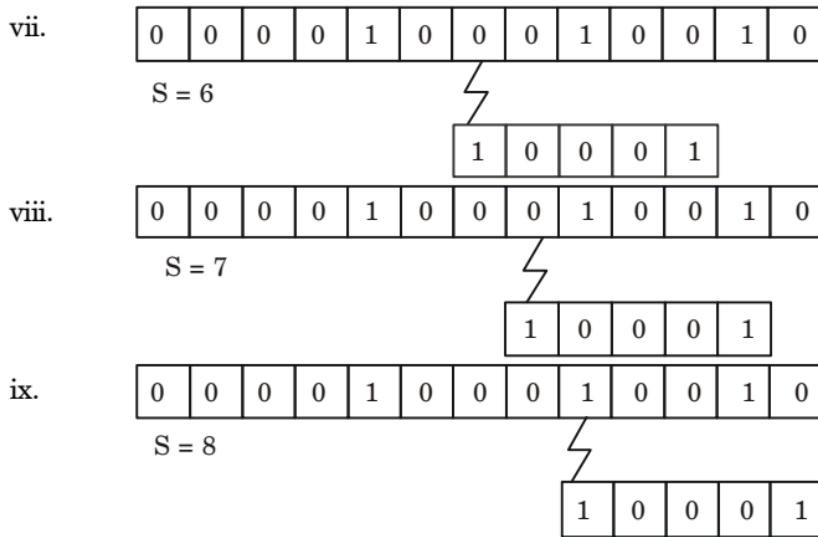
Answer

Given,

$$P = 10001$$

$$T = 0000100010010$$

- i. 
- ii. 
- iii. 
- iv. 
- v. 
- vi. 



Que 5.6. Write down Knuth-Morris-Pratt algorithm for string matching. Find the prefix function of the string *ababababca*.

Answer

Knuth-Morris-Pratt algorithm for string matching :
COMPUTE-PREFIX-FUNCTION (P)

1. $m \leftarrow \text{length } [P]$
2. $\pi [1] \leftarrow 0$
3. $k \leftarrow 0$
4. for $q \leftarrow 2$ to m
 5. do while $k > 0$ and $P [k + 1] \neq P[q]$
 6. do $k \leftarrow \pi [k]$
 7. if $P[k + 1] = P[q]$
 8. then $k \leftarrow k + 1$
 9. $\pi [q] \leftarrow k$
 10. return π

KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π .

KMP-MATCHER (T, p)

1. $n \leftarrow \text{length } [T]$
2. $m \leftarrow \text{length } [P]$
3. $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION } (P)$
4. $q \leftarrow 0$
5. for $i \leftarrow 1$ to n
 6. do while $q > 0$ and $P [q + 1] \neq T[i]$
 7. do $q \leftarrow \pi [q]$
 8. if $P [q + 1] = T[i]$

9. then $q \leftarrow q + 1$
10. if $q = m$
11. then print "pattern occurs with shift" $i - m$
12. $q \leftarrow \pi[q]$

Prefix function of the string *ababababca* :

$$m \leftarrow \text{length}[P]$$

$$\therefore m = 10$$

Initially, $P[1] = 0, k = 0$

for $q \leftarrow 2$ to 10

for $q = 2, k > 0$

$$\& \quad P[0 + 1] = P[2]$$

$$\therefore \pi[2] = 0$$

for $q = 3, k > 0$

$$\& \quad P[0 + 1] = P[3]$$

$$\therefore k \leftarrow k + 1 = 1$$

$$\& \quad \pi[3] \leftarrow 1$$

for $q = 4, k > 0$

$$\& \quad P[1 + 1] = P[4]$$

$$\therefore k \leftarrow 1 + 1 = 2$$

$$\& \quad \pi[4] \leftarrow 2$$

for $q = 5, k > 0$

$$\& \quad P[2 + 1] = P[5]$$

$$\therefore k \leftarrow 2 + 1 = 3$$

$$\& \quad \pi[5] \leftarrow 3$$

for $q = 6, k > 0$

$$\& \quad P[3 + 1] = P[6]$$

$$\therefore k \leftarrow 3 + 1 = 4$$

$$\& \quad \pi[6] \leftarrow 4$$

for $q = 7, k > 0$

$$\& \quad P[4 + 1] = P[7]$$

$$\therefore k \leftarrow 4 + 1 = 5$$

$$\& \quad \pi[7] \leftarrow 5$$

for $q = 8, k > 0$

$$\& \quad P[5 + 1] = P[8]$$

$$\therefore k \leftarrow 5 + 1 = 6$$

$$\& \quad \pi[8] \leftarrow 6$$

for $q = 9, k > 0$

$$\& \quad P[6 + 1] = P[9]$$

$$\therefore k \leftarrow \pi[k] = 6$$

$$\& \quad \pi[9] \leftarrow 6$$

for $q = 10, k > 0$

$$\& \quad P[6 + 1] = P[10]$$

$$\therefore k \leftarrow 6 + 1 = 7$$

$$\& \quad \pi[10] \leftarrow 7$$

String	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>
$P[i]$	1	2	3	4	5	6	7	8	9	10
$\pi[i]$	0	0	1	2	3	4	5	6	6	7

Que 5.7. Compute the prefix function π for the pattern $P = abacab$ using KNUTH-MORRIS-PRATT algorithm. Also explain Naive string matching algorithm.

AKTU 2017-18, Marks 10

Answer

Prefix function of the string *abacab* :

$$m \leftarrow \text{length}[P]$$

$$\therefore m = 6$$

Initially, $\pi[1] = 0, k = 0$

for $q \leftarrow 2$ to 6

for $q = 2, k \neq 0$

& $P[0 + 1] \neq P[2]$

$$\therefore \pi[2] = 0$$

for $q = 3, k \neq 0$

& $P[0 + 1] = P[3]$

$$\therefore k = k + 1 = 1$$

$$\& \pi[3] = 1$$

for $q = 4, k > 0$

& $P[1 + 1] \neq P[4]$

$$\therefore k \leftarrow \pi[1] = 0$$

$$P[1] \neq P[4]$$

$$\& \pi[4] = 0$$

for $q = 5, k > 0$

& $P[0 + 1] = P[5]$

$$\therefore k \leftarrow 0 + 1 = 1$$

$$\& \pi[5] = 1$$

for $q = 6, k > 0$

& $P[1 + 1] = P[6]$

$$\therefore k \leftarrow 1 + 1 = 2$$

$$\& \pi[6] = 2$$

String	<i>a</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
$P[i]$	1	2	3	4	5	6
$\pi[i]$	0	0	1	0	1	2

Naive string matching algorithm : Refer Q. 5.4, Page 5–5B, Unit-5.

Que 5.8. Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function π for the pattern $ababbabbabbababbabb$ when the alphabet is $\Sigma = \{a, b\}$.

AKTU 2019-20, Marks 07

Answer

Knuth-Morris-Pratt string matching algorithm : Refer Q. 5.6, Page 5-7B, Unit-5.

Numerical :

pattern = ababbabbabbababbabb
length 19

Initially, $\pi(1) = 0$ and $k = 0$

For $q \leftarrow 2$ to 9

For $q = 2$ and $k \neq 0$

$P[0 + 1] \neq P[2]$.

$\pi[2] = 0$

For $q = 3$

$P[0 + 1] = P[3]$

$k = k + 1 = 1$

$\pi[3] = 1$

For $q = 4$ $k > 0$

$P[1 + 1] = P[4]$

$P[2] = P[4]$

$k = k + 1 = 2$

$\pi[4] = 2$

For $q = 5$ $k > 0$

$P[2 + 1] \neq P[5]$

$\pi[5] = 0$

For $q = 6$

$P[0 + 1] = P[6]$

$k = k + 1 = 1$

$\pi[6] = 1$

For $q = 7$ $k > 0$

$P[1 + 1] = P[7]$

$P[2] = P[7]$

$k = k + 1 = 2$

$\pi[7] = 0$

For $q = 8$ $k > 0$

$P[2 + 1] = P[8]$

$P[3] \neq P[8]$

$\pi[8] = 0$

For $q = 9$

$P[0 + 1] = P[9]$

$k = k + 1$

```

 $\pi[9] = 2$ 
For  $q = 10 \ k > 0$ 
 $P[2 + 1] \neq P[10]$ 
 $\pi[10] = 0$ 
For  $q = 11 \ k > 0$ 
 $P[0 + 1] \neq P[11]$ 
 $\pi[11] = 0$ 
For  $q = 12 \ k > 0$ 
 $P[0 + 1] \neq P[12]$ 
 $k = k + 1$ 
 $\pi[12] = 2$ 
For  $q = 13 \ k > 0$ 
 $P[2 + 1] \neq P[13]$ 
 $\pi[13] = 0$ 
For  $q = 14$ 
 $P[0 + 1] = P[14]$ 
 $k = k + 1$ 
 $\pi[14] = 2$ 
For  $q = 15 \ k > 0$ 
 $P[2 + 1] \neq P[15]$ 
 $\pi[15] = 0$ 
For  $q = 16 \ k > 0$ 
 $P[0 + 1] \neq P[16]$ 
 $\pi[16] = 0$ 
For  $q = 17$ 
 $P[0 + 1] \neq P[17]$ 
 $k = k + 1$ 
 $\pi[17] = 2$ 
For  $q = 18 \ k > 0$ 
 $P[2 + 1] \neq P[18]$ 
 $\pi[18] = 0$ 
For  $q = 19$ 
 $P[0 + 1] \neq P[19]$ 
 $\pi[19] = 0$ 

```

String	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>b</i>									
$P[i]$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\pi[i]$	0	0	1	2	0	1	2	0	2	0	0	2	0	2	0	0	2	0	0

Que 5.9. Write algorithm for Rabin-Karp method. Give a suitable example to explain it.

OR

What is string matching algorithm ? Explain Rabin-Karp method with examples.

AKTU 2015-16, Marks 10

Answer

String matching algorithm : Refer Q. 5.3, Page 5–4B, Unit-5.

The Rabin-Karp algorithm :

The Rabin-Karp algorithm states that if two strings are equal, their hash values are also equal. This also uses elementary number-theoretic notions such as the equivalence of two numbers modulo a third.

Rabin-Karp-Matcher (T, P, d, q)

1. $n \leftarrow \text{length } [T]$
2. $m \leftarrow \text{length } [P]$
3. $h \leftarrow d^{m-1} \bmod q$
4. $p \leftarrow 0$
5. $t_0 \leftarrow 0$
6. for $i \leftarrow 1$ to m
 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 $n-m$
10. do if $p = t_s$
 11. then if $p[1...m] = T[s+1...s+m]$
 12. then “pattern occurs with shift” s
13. if $s < n-m$
14. then $t_{s+1} \leftarrow (d(t_s - T[s+1] h) + T[s+m+1]) \bmod q$

Example of Rabin-Karp method : Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $p = 26$

Given, $p = 26$ and $q = 11$

Now we divide 26 by 11 i.e.,

Remainder is 4 and $m = 2$.

We know m denotes the length of p .

T	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

Now we divide 31 by 11, and get remainder is 9.

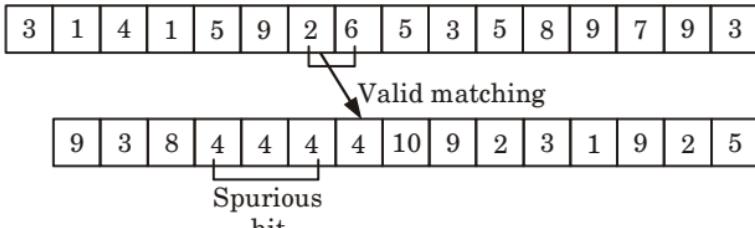
Similarly, 14 by 11 and get remainder is 3.

So, continue this step till last i.e., 93 is divided by 11 and get remainder is 5.

After that we will store all remainder in a table.

9	3	8	4	4	4	4	10	9	2	3	1	9	2	5
---	---	---	---	---	---	---	----	---	---	---	---	---	---	---

Now we find valid matching.



The number of spurious hits is 3.

Que 5.10. Give a linear time algorithm to determine if a text T is a cycle rotation of another string T' . For example : RAJA and JARA are cyclic rotations of each other.

AKTU 2018-19, Marks 10

Answer

Knuth-Morris-Pratt algorithm is used to determine if a text T is a cycle rotation of another string T' .

Knuth-Morris-Pratt algorithm : Refer Q. 5.6, Page 5–7B, Unit-5.

PART-3

Theory of NP – Completeness.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.11. Discuss the problem classes P, NP and NP-complete.

Answer

P : Class P are the problems which can be solved in polynomial time, which take time like $O(n)$, $O(n^2)$, $O(n^3)$.

Example : Finding maximum element in an array or to check whether a string is palindrome or not. So, there are many problems which can be solved in polynomial time.

NP : Class NP are the problems which cannot be solved in polynomial time like TSP (travelling salesman problem).

Example : Subset sum problem is best example of NP in which given a set of numbers, does there exist a subset whose sum is zero, but NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

NP-complete : The group of problems which are both in NP and NP-hard are known as NP-complete problem.

Now suppose we have a NP-complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem, therefore Q will also be at least NP-hard, it may be NP-complete also.

Que 5.12. Discuss the problem classes P, NP and NP-complete with class relationship.

AKTU 2017-18, Marks 10

Answer

1. The notion of NP-hardness plays an important role in the relationship between the complexity classes P and NP .

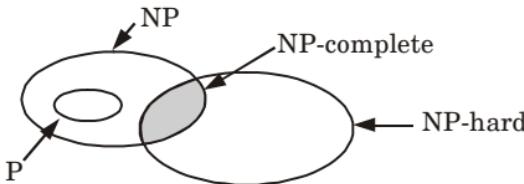


Fig. 5.12.1. Relationship among P , NP , NP -complete and NP -hard problems.

2. It is also often used to define the complexity class NP -complete which is the intersection of NP and NP -hard.
3. Consequently class NP -hard can be understood as the class of problems that are NP -complete or harder.
4. There are no polynomial time algorithms for NP -hard problems.
5. A problem being in NP means that the problem is “easy” (in a certain specific sense), whereas a problem being NP -hard means that the problem is “difficult” (in another specific sense).
6. A problem being in NP and a problem being NP -hard are not mutually exclusive. When a problem is both in NP and NP -hard, we say that the problem is NP -complete.
7. All problems in NP can be solved deterministically in time $O(2n)$.
8. An example of an NP -hard problem is the decision problem subset-sum. Given a set of integers, does any non-empty subset of them add up to zero ? i.e., a yes or no question, and happens to be NP -complete.
9. There are, also decision problems that are NP -hard but not NP -complete.
10. For example, in the halting problem “given a program and its input, will it run forever” i.e., yes or no question, so this is a decision problem. It is case to prove that the halting problem is NP -hard but not NP -complete.

Que 5.13. What is NP-completeness ?**Answer**

A language $L \subseteq \{0,1\}^*$ is NP -complete if it satisfies the following two properties :

- i. $L \in NP$; and
- ii. For every $L' \leq_p L$

NP-hard : If a language L satisfies property (ii), but not necessarily property (i), we say that L is NP -hard.

NP-complete : We use the notation $L \in NPC$ to denote that L is NP -complete.

Theorem : If any NP -complete problem is polynomial time solvable, then $P = NP$. If any problem in NP is not polynomial time solvable, then all NP complete problems are not polynomial time solvable.

Proof : Suppose that $L \in P$ and also that $L \in \text{NPC}$. For any $L' \in \text{NP}$, we have $L' \leq_p L$ by property (ii) of the definition of NP-completeness. We know if $L' \leq_p L$ then $L \in P$ implies $L' \in P$, which proves the first statement.

To prove the second statement, suppose that there exists an $L \in \text{NP}$ such that $L \notin P$. Let $L' \in \text{NPC}$ be any NP-complete language, and for the purpose of contradiction, assume that $L' \in P$. But then we have $L \leq_p L'$ and thus $L \in P$.

Que 5.14. Explain NP-hard and NP-complete problems and also define the polynomial time problems and write a procedure to solve NP-problems.

OR

Write short note on NP-hard and NP-complete problems.

OR

Define NP-hard and NP-complete problems. What are the steps involved in proving a problem NP-complete? Specify the problems already proved to be NP-complete.

AKTU 2019-20, Marks 07

Answer

NP-hard problem :

1. We say that a decision problem P_i is NP-hard if every problem in NP is polynomial time reducible to P_i .
2. In symbols,

$$P_i \text{ is NP-hard if, for every } P_j \in \text{NP}, P_j \xrightarrow{\text{Poly}} P_i.$$
3. This does not require P_i to be in NP.
4. Highly informally, it means that P_i is ‘as hard as’ all the problems in NP.
5. If P_i can be solved in polynomial time, then all problems in NP.
6. Existence of a polynomial time algorithm for an NP-hard problem implies the existence of polynomial solution for every problem in NP.

NP-complete problem :

1. There are many problems for which no polynomial time algorithms are known.
2. Some of these problems are travelling salesman problem, optimal graph colouring, the Knapsack problem, Hamiltonian cycles, integer programming, finding the longest simple path in a graph, and satisfying a Boolean formula.
3. These problems belong to an interesting class of problems called the “NP-complete” problems, whose status is unknown.
4. The NP-complete problems are traceable i.e., require a super polynomial time.

Polynomial time problem :

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k where n is the complexity of input.

Polynomial time verifiable algorithm : A polynomial time algorithm A is said to be polynomial time verifiable if it has following properties :

1. The input to A consists of an instance I of X (X is a decision problem) and a string S such that the length of S is bounded by some polynomial in the size of I .
2. The output of A is either yes or no.
3. If I is a negative instance of X , then the output of A is “no” regardless of the value of S .
4. If I is a positive instance of X , then there is at least one choice of S for which A output “yes”.

Procedure to solve NP-problems :

1. The class NP is the set of all decision problems that have instances that are solvable in polynomial time using a non-deterministic turing machine.
2. In a non-deterministic turing machine, in contrast to a deterministic turing machine, for each state, several rules with different actions can be applied.
3. Non-deterministic turing machine branches into many copies that are represented by a computational tree in which there are different computational paths.
4. The class NP corresponds to a non-deterministic turing machine that guesses the computational path that represents the solution.
5. By doing so, it guesses the instances of the decision problem.
6. In the second step, a deterministic turing machine verifies whether the guessed instance leads to a “yes” answer.
7. It is easy to verify whether a solution is valid or not. This statement does not mean that finding a solution is easy.

Que 5.15. Differentiate NP-complete with NP-hard.

AKTU 2016-17, Marks 10

Answer

S. No.	NP-complete	NP-hard
1.	An NP-complete problems is one to which every other polynomial-time non-deterministic algorithm can be reduced in polynomial time.	NP-hard problems is one to which an NP-complete problem is Turing-reducible.
2.	NP-complete problems do not corresponds to an NP-hard problem.	NP-hard problems correspond to an NP-complete problem.

3.	NP-complete problems are exclusively decision problem.	NP-hard problems need not to be decision problem.
4.	NP-complete problems have to be in NP-hard and also in NP.	NP-hard problems do not have to be in NP.
5.	For example : 3-SAT vertex cover problem is NP-complete.	For example : Halting problem is NP-hard.

Que 5.16. Discuss NP-complete problem and also explain minimum vertex cover problem in context to NP-completeness.

Answer

NP-complete problem : Refer Q. 5.14, Page 5–15B, Unit-5.

Minimum vertex cover problem :

1. A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.
2. The problem of finding a minimum vertex cover is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm.
3. Its decision version, the vertex cover problem, was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in computational complexity theory.
4. Furthermore, the vertex cover problem is fixed-parameter tractable and a central problem in parameterized complexity theory.
5. The minimum vertex cover problem can be formulated as a half-integral linear program whose dual linear program is the maximum matching problem.
6. Formally, a vertex cover V' of undirected graph $G = (V, E)$ is a subset of V such that $uv \in V' \vee v \in V'$ i.e., it is a set of vertices V' where every edge has at least one endpoint in the vertex cover V' . Such a set is said to cover the edges of G .

Example : Fig. 5.16.1 shows examples of vertex covers, with some vertex cover V' marked in dark.

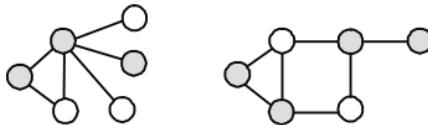


Fig. 5.16.1.

7. A minimum vertex cover is a vertex cover of smallest possible size.
8. The vertex cover number τ is the size of a minimum cover, i.e., $\tau = |V'|$. The Fig. 5.16.2 shows examples of minimum vertex covers in the previous graphs.

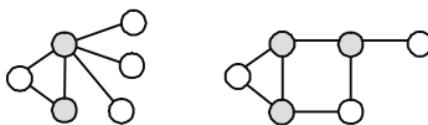


Fig. 5.16.2.

Que 5.17. Discuss different types of NP-complete problem.

Answer

Types of NP-complete problems :

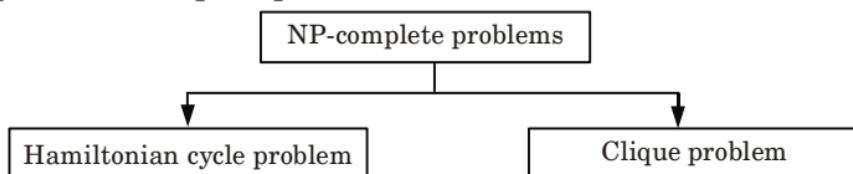


Fig. 5.17.1.

Hamiltonian cycle problem :

1. A Hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V .
2. A graph that contains a Hamiltonian cycle is said to be Hamiltonian, otherwise it is said to be non-Hamiltonian.

The clique problem :

1. A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E .
2. The size of a clique is the number of vertices it contains.
CLIQUE = { $\langle G, K \rangle : G$ is a graph with a clique of size K }
3. The clique problem is the optimization problem of finding a clique of maximum size in a graph.
4. As a decision problem, we ask simply whether a clique of a given size k exists in the graph.

Que 5.18. Show that Hamiltonian circuit is NP-complete.

Answer

Theorem : Hamiltonian circuit (HC) is NP-complete.

Proof :

1. Let us define a non-deterministic algorithm A that takes, as input, a graph G encoded as an adjacency list in binary notation, with the vertices numbered 1 to N .
2. We define A to first iteratively call the choose method to determine a sequence S of $N + 1$ numbers from 1 to N .
3. Then, we have A to check that each number from 1 to N appears exactly once in S (for example, by sorting S), except for the first and last numbers in S , which should be the same.

4. Then, we verify that a sequence S defines a cycle of vertices and edges in G .
5. A binary encoding of the sequence S is clearly of size at most n , where n is the size of the input. Moreover, both of the checks made on the sequence S can be done in polynomial time in n .
6. Observe that if there is a cycle in G that visits each vertex of G exactly once; returning to its starting vertex, then there is a sequence S for which A will output “yes.”
7. Likewise, if A outputs “yes,” then it has found a cycle in G that visits each vertex of G exactly once, returning to its starting point. Hence, Hamiltonian circuit is NP-complete.

Que 5.19. Show that CLIQUE problem is NP-complete.

Answer

Problem : The CLIQUE problem is defined as $\{< G, k > | G \text{ is a graph with a } k\text{-clique}\}$. Show that CLIQUE is NP-complete.

Proof :

1. First, to show that CLIQUE is in NP. Given an instance of $< G, k >$ and a k -clique we can easily verify in $O(n^2)$ time that we do, in fact, have a k -clique.
2. Now, we want to show that 3-SAT is CLIQUE. Let F be a boolean formula in CNF.
3. For each literal in F we will make a vertex in the graph i.e., $(x_1 + \bar{x}_2 + x_3) (\bar{x}_1 + x_2 + \bar{x}_3)$ has 6 vertices.

$(x_1 + \bar{x}_2 + x_3) (\bar{x}_1 + x_2 + \bar{x}_3)$ has 6 vertices.

Let k be the number of clauses in F .

4. We will connect each vertex to all of the other vertices that are logically compatible except for the ones that are in the same clause.
5. Now, if we have a satisfiable assignment we will have a k -clique because the satisfying vertices will all be connected to one another.
6. Thus, we can use CLIQUE to solve 3-SAT so CLIQUE is NP-complete.

Que 5.20. Define different complexity classes in detail with suitable example. Show that TSP problem is NP-complete.

Answer

Different complexity classes :

There are some complexity classes involving randomized algorithms :

1. **Randomized polynomial time (RP) :** The class RP consists of all languages L that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^*

$$\begin{aligned} x \in L \Rightarrow P[A(x) \text{ accepts}] &\geq \frac{1}{2} \\ x \notin L \Rightarrow P[A(x) \text{ accepts}] &= 0 \end{aligned}$$

Independent repetitions of the algorithms can be used to reduce the probability of error to exponentially small.

- 2. Zero-error probabilistic polynomial time (ZPP) :** The class ZPP is the class of languages which have Las Vegas algorithms running in expected polynomial time.

$$\text{ZPP} = \text{RP} \cap \text{co-RP}$$

where a language L is in co- X where X is a complexity class if and only if its complement $\Sigma^* - L$ is in X .

- 3. Probabilistic polynomial time (PP) :** The class PP consists of all languages L that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^* .

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{1}{2}$$

$$x \notin L \Rightarrow P[A(x) \text{ accepts}] < \frac{1}{2}$$

To reduce the error probability, we cannot repeat the algorithm several times on the same input and produce the output which occurs in the majority of those trials.

- 4. Bounded-error probabilistic polynomial time (BPP) :** The class BPP consists of all languages that have a randomized algorithm A running in worst case polynomial time such that for any input x in Σ^* .

$$x \in L \Rightarrow P[A(x) \text{ accepts}] \geq \frac{3}{4}$$

$$x \notin L \Rightarrow P[A(x) \text{ accepts}] \leq \frac{1}{4}$$

For this class of algorithms, the error probability can be reduced to $1/2n$ with only a polynomial number of iterations.

For a given a graph $G = (V, E)$ and a number k , does there exist a tour C on G such that the sum of the edge weights for edges in C is less than or equal to k .

Proof:

Part 1 : TSP is in NP.

Proof:

1. Let a hint S be a sequence of vertices $V = v_1, \dots, v_n$.
2. We then check two things :
 - a. First we check that every edge traversed by adjacent vertices is an edge in G , such that the sum of these edge weights is less than or equal to k .
 - b. Secondly we check that every vertex in G is in V , which assures that every node has been traversed.
3. We accept S if and only if S satisfies these two questions, otherwise reject.
4. Both of these checks are clearly polynomial, thus our algorithm forms a verifier with hint S , and TSP is consequently in NP.

Part 2 : TSP is NP-Hard.**Proof :**

1. To show that TSP is NP-Hard, we must show that every problem y in NP reduces to TSP in polynomial time.
2. To do this, consider the decision version of Hamiltonian Cycle (HC).
3. Take $G = (V, E)$, set all edge weights equal to 1, and let $k = |V| = n$, that is, k equals the number of nodes in G .
4. Any edge not originally in G then receives a weight of 2 (traditionally TSP is on a complete graph, so we need to add in these extra edges).
5. Then pass this modified graph into TSP, asking if there exists a tour on G with cost at most k . If the answer to TSP is YES, then HC is YES. Likewise if TSP is NO, then HC is NO.

First direction : HC has a YES answer \Rightarrow TSP has a YES answer.

Proof :

1. If HC has a YES answer, then there exists a simple cycle C that visits every node exactly once, thus C has n edges.
2. Since every edge has weight 1 in the corresponding TSP instance for the edges that are in the HC graph, there is a Tour of weight n . Since $k = n$, and given that there is a tour of weight n , it follows that TSP has a YES answer.

Second direction : HC has a NO answer \Rightarrow TSP has a NO answer.

Proof :

1. If HC has a NO answer, then there does not exist a simple cycle C in G that visits every vertex exactly once. Now suppose TSP has a YES answer.
2. Then there is a tour that visits every vertex once with weight at most k .
3. Since the tour requires every node be traversed, there are n edges, and since $k = n$, every edge traversed must have weight 1, implying that these edges are in the HC graph. Then take this tour and traverse the same edges in the HC instance. This forms a Hamiltonian Cycle, a contradiction.

This concludes Part 2. Since we have shown that TSP is both in NP and NP-Hard, we have that TSP is NP-Complete.

Que 5.21. Prove that three colouring problem is NP-complete.

AKTU 2016-17, Marks 10

Answer

1. To show the problem is in NP, let us take a graph $G(V, E)$ and a colouring c , and checks in $O(n^2)$ time whether c is a proper colouring by checking if the end points of every edge $e \in E$ have different colours.
2. To show that 3-COLOURING is NP-hard, we give a polytime reduction from 3-SAT to 3-COLOURING.

3. That is, given an instance ϕ of 3-SAT, we will construct an instance of 3-COLOURING (*i.e.*, a graph $G(V, E)$) where G is 3-colourable iff ϕ is satisfiable.
4. Let ϕ be a 3-SAT instance and C_1, C_2, \dots, C_m be the clauses of ϕ defined over the variables $\{x_1, x_2, \dots, x_n\}$.
5. The graph $G(V, E)$ that we will construct needs to capture two things :
 - a. Somehow establish the truth assignment for x_1, x_2, \dots, x_n via the colours of the vertices of G ; and
 - b. Somehow capture the satisfiability of every clause C_i in ϕ .
6. To achieve these two goals, we will first create a triangle in G with three vertices $\{T, F, B\}$ where T stands for True, F for False and B for Base.
7. Consider $\{T, F, B\}$ as the set of colours that we will use to colour (label) the vertices of G .
8. Since this triangle is part of G , we need 3 colours to colour G .
9. Now we add two vertices v_i, \bar{v}_i for every literal x_i and create a triangle B , v_i, \bar{v}_i for every (v_i, \bar{v}_i) pair, as shown in Fig. 5.21.1.

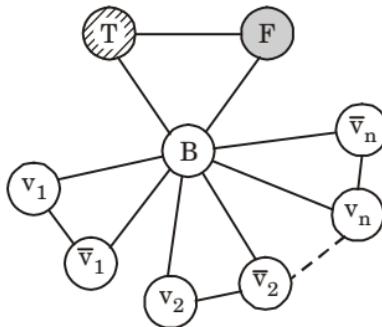


Fig. 5.21.1.

10. This construction captures the truth assignment of the literals.
11. Since if G is 3-colourable, then either v_i or \bar{v}_i gets the colour T , and we interpret this as the truth assignment to v_i .
12. Now we need to add constraints to G to capture the satisfiability of the clauses of ϕ .
13. To do so, we introduce the Clause Satisfiability Gadget, (the OR-gadget). For a clause $C_i = (a \vee b \vee c)$, we need to express the OR of its literals using our colours $\{T, F, B\}$.
14. We achieve this by creating a small gadget graph that we connect to the literals of the clause. The OR-gadget is constructed as follows :

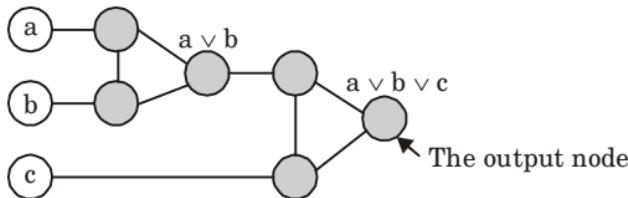


Fig. 5.21.2.

15. Consider this gadget graph as a circuit whose output is the node labeled $a \vee b \vee c$. We basically want this node to be coloured T if C_i is satisfied and F otherwise.
16. This is a two step construction : The node labelled $a \vee b$ captures the output of $(a \vee b)$ and we repeat the same operation for $((a \vee b) \vee c)$. If we play around with some assignments to a, b, c , we will notice that the gadget satisfies the following properties :
- If a, b, c are all coloured F in a 3-colouring, then the output node of the OR-gadget has to be coloured F . Thus capturing the unsatisfiability of the clause $C_i = (a \vee b \vee c)$.
 - If one of a, b, c is coloured T , then there exists a valid 3-colouring of the OR-gadget where the output node is coloured T . Thus again capturing the satisfiability of the clause.

17. Once we add the OR-gadget of every C_i in ϕ , we connect the output node of every gadget to the Base vertex and to the False vertex of the initial triangle, as follows :

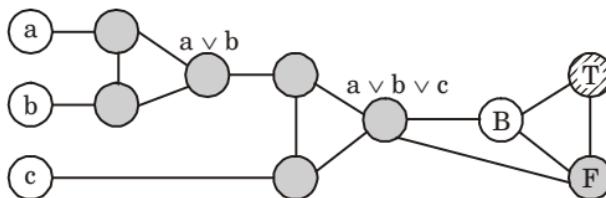


Fig. 5.21.3.

18. Now we prove that our initial 3-SAT instance ϕ is satisfiable if and only the graph G as constructed above is 3-colourable. Suppose ϕ is satisfiable and let $(x_1^*, x_2^*, \dots, x_n^*)$ be the satisfying assignment.
19. If x_i^* is assigned True, we colour v_i with T and \bar{v}_i with F (recall they are connected to the Base vertex, coloured B , so this is a valid colouring).
20. Since ϕ is satisfiable, every clause $C_i = (a \vee b \vee c)$ must be satisfiable, i.e., at least one of a, b, c is set to True. By the property of the OR-gadget, we know that the gadget corresponding to C_i can be 3-coloured so that the output node is coloured T .
21. And because the output node is adjacent to the False and Base vertices of the initial triangle only, this is a proper 3-colouring.
22. Conversely, suppose G is 3-colourable. We construct an assignment of the literals of ϕ by setting x_i to True if v_i is coloured T and vice versa.
23. Now consider this assignment is not a satisfying assignment to ϕ , then this means there exists at least one clause $C_i = (a \vee b \vee c)$ that was not satisfiable.
24. That is, all of a, b, c were set to False. But if this is the case, then the output node of corresponding OR-gadget of C_i must be coloured F .
25. But this output node is adjacent to the False vertex coloured F ; thus contradicting the 3-colourability of G .
26. To conclude, we have shown that 3-COLOURING is in NP and that it is NP-hard by giving a reduction from 3-SAT.

27. Therefore 3-COLOURING is NP-complete.

Que 5.22. Prove that P is the subset of NP.

Answer

To prove : P is the subset of NP.

Proof :

1. If $L \in P$, then $L \in NP$, as there is a polynomial time algorithm to decide L , this algorithm can easily be converted into a row argument verification algorithm that simply ignores any exception and accepts exactly those input strings it determines to be in L .
2. Thus, $P \subseteq NP$.

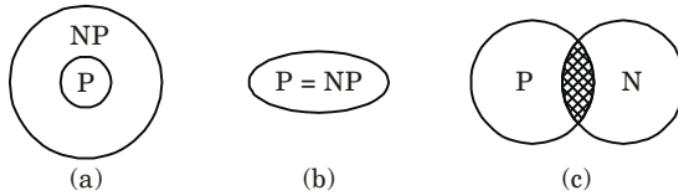


Fig. 5.22.1. P Vs NP .

PART-4

Approximation Algorithm.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.23. Describe approximation algorithm in detail. How it differ with deterministic algorithm. Show that TSP is 2 approximate.

OR

Explain approximation algorithms with suitable examples.

AKTU 2015-16, 2017-18; Marks 10

Answer

Approximation algorithm :

1. An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.
2. The best of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.

3. Let $c(i)$ be the cost of solution produced by approximate algorithm and $c^*(i)$ be the cost of optimal solution for some optimization problem instance i .
4. For minimization and maximization problem, we are interested in finding a solution of a given instance i in the set of feasible solutions, such that $c(i) / c^*(i)$ and $c^*(i) / c(i)$ be as small as possible respectively.
5. We say that an approximation algorithm for the given problem instance i , has a ratio bound of $p(n)$ if for any input of size n , the cost c of the solution produced by the approximation algorithm is within a factor of $p(n)$ of the cost c^* of an optimal solution. That is

$$\max(c(i) / c^*(i), c^*(i) / c(i)) \leq p(n)$$

The definition applies for both minimization and maximization problems.

6. $p(n)$ is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have $p(n) = 1$.
7. For a minimization problem, $0 < c^*(i) < c(i)$, and the ratio $c(i) / c^*(i)$ gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.
8. Similarly, for a maximization problem, $0 < c(i) \leq c^*(i)$, and the ratio $c^*(i) / c(i)$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

Difference between deterministic algorithm and approximation algorithm :

S. No.	Deterministic algorithm	Approximation algorithm
1.	It does not deal with optimization problem.	It deals with optimization problem.
2.	It has initial and final step.	It does not have initial or final state.
3.	It require finite state machine.	It does not require finite state machine.
4.	It fails to deliver a result.	It gives an optimal result.
5.	It does not apply to maximization or minimization problem.	It applies to maximization and minimization problem.

Proof :

TSP is 2-approximate :

Let H^* denote the optimal tour. Observe that a TSP with one edge removed is a spanning tree (not necessarily MST).

It implies that the weight of the MST ' T ' is in lower bound on the cost of an optimal tour.

$$c(T) \leq c(H^*)$$

A "Full" walk, W , traverse every edge of MST, T , exactly twice. That is,

$$c(W) = 2c(T)$$

which means

$$c(W) \leq 2c(H^*)$$

and we have

$$c(W) / c(H^*) \leq p(n) = 2$$

That is, the cost of walk, $c(W)$, of the solution produced by the algorithm is within a factor of $p(n) = 2$ of the cost $c(H^*)$ of an optimal solution.

Que 5.24. Explain vertex cover problem with algorithm and analysis.

Answer

A vertex cover of an undirected graph $G = (V, E)$ is a subset of $V' \subseteq V$ such that if edge $(u, v) \in G$ then $u \in V'$ or $v \in V'$ (or both).

Problem : Find a vertex cover of maximum size in a given undirected graph. This optimal vertex cover is the optimization version of an NP-Complete problem but it is not too hard to find a vertex cover that is near optimal.

Approx-vertex-cover ($G : \text{Graph}$)

1. $c \leftarrow \emptyset$
2. $E' \leftarrow E[G]$
3. while E' is not empty
4. do Let (u, v) be an arbitrary edge of E'
5. $c \leftarrow c \cup \{u, v\}$
6. Remove from E' every edge incident on either u or v
7. return c

Analysis : It is easy to see that the running time of this algorithm is $O(V + E)$, using adjacency list to represent E' .

Que 5.25. Describe approximation algorithm in detail. What is the approximation ratio ? Show that vertex cover problem is 2-approximate.

Answer

Approximation algorithm : Refer Q. 5.23, Page 5-24B, Unit-5.

Proof :

Vertex cover problem is 2-approximate :

Goal : Since this is a minimization problem, we are interested in smallest possible c / c^* . Specifically we want to show $c / c^* = 2 = p(n)$.

In other words, we want to show that Approx-Vertex-Cover algorithm returns a vertex-cover that is almost twice the size of an optimal cover.

Proof : Let the set c and c^* be the sets output by Approx-Vertex-Cover and Optimal-Vertex-Cover respectively. Also, let A be the set of edges.

Because, we have added both vertices, we get $c = 2|A|$ but Optimal-Vertex-Cover would have added one of two.

$$c / c^* \leq p(n) = 2.$$

Formally, since no two edge in A are covered by the same vertex from c^* and the lower bound : $|c^*| \geq A$... (5.25.1)

on the size of an Optimal-Vertex-Cover.

Now, we pick both end points yielding an upper bound on the size of Vertex-Cover :

$$|c| \leq 2|A|$$

Since, upper bound is an exact in this case, we have

$$|c| = 2|A| \quad \dots(5.25.2)$$

Take $|c|/2 = |A|$ and put it in equation (5.25.1)

$$|c^*| \geq |c|/2$$

$$|c^*|/|c| \geq 1/2$$

$$|c^*|/|c| \leq 2 = p(n) \text{ Hence the theorem proved.}$$

Que 5.26. Explain Travelling Salesman Problem (TSP) with the triangle inequality.

Answer

Problem : Given a complete graph with weights on the edges, find a cycle of least total weight that visits each vertex exactly once. When the cost function satisfies the triangle inequality, we can design an approximate algorithm for TSP that returns a tour whose cost is not more than twice the cost of an optimal tour.

APPROX-TSP-TOUR (G, c) :

1. Select a vertex $r \in V[G]$ to be a “root” vertex.
2. Compute a minimum spanning tree T for G from root r using MST-PRIM (G, c, r).
3. Let L be the list of vertices visited in a pre-order tree walk of T .
4. Return the Hamiltonian cycle H that visits the vertices in the order L .

Outline of an approx-TSP tour : First, compute a MST (minimum spanning tree) whose weight is a lower bound on the length of an optimal TSP tour. Then, use MST to build a tour whose cost is no more than twice that of MST's weight as long as the cost function satisfies triangle inequality.

Que 5.27. Write short notes on the following using approximation algorithm with example.

- i. Nearest neighbour
- ii. Multifragment heuristic

Answer

i. Nearest neighbour :

The following well-known greedy algorithm is based on the nearest-neighbour heuristic i.e., always go next to the nearest unvisited city.

Step 1 : Choose an arbitrary city as the start.

Step 2 : Repeat the following operation until all the cities have been visited : go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 : Return to the starting city.

Example :

1. For the instance represented by the graph in Fig. 5.27.1, with a as the starting vertex, the nearest-neighbour algorithm yields the tour (Hamiltonian circuit) $s_a : a - b - c - d - a$ of length 10.

2. The optimal solution, as can be easily checked by exhaustive search, is the tour s^* : $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

i.e., tour s_a is 25 % longer than optimal tour s^* .

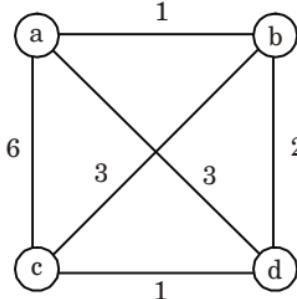


Fig. 5.27.1. Instance of the traveling salesman problem.

3. Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbour algorithm.
 4. In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour.
 5. Indeed, if we change the weight of edge (a, d) from 6 to an arbitrary large number $w \geq 6$ in given example, the algorithm will still yield the tour $a - b - c - d - a$ of length $4 + w$, and the optimal solution will still be $a - b - d - c - a$ of length 8. Hence,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4+w}{8}$$

which can be made as large as we wish by choosing an appropriately large value of w . Hence, $RA = \infty$ for this algorithm.

ii. Multifragment heuristic :

Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have degree 2.

Step 1 : Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

Step 2 : Repeat this step n times, where n is the number of cities in the instance being solved : add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n ; otherwise, skip the edge.

Step 3 : Return the set of tour edges.

Example :

1. Applying the algorithm to the graph in Fig. 5.27.1 yields $\{(a, b), (c, d), (b, c), (a, d)\}$.

2. There is, however, a very important subset of instances, called Euclidean, for which we can make a non-trivial assertion about the accuracy of both the nearest-neighbour and multifragment-heuristic algorithms.
3. These are the instances in which intercity distances satisfy the following natural conditions :
 - a. **Triangle inequality :** $d[i, j] \leq d[i, k] + d[k, j]$ for any triple of cities i, j , and k (the distance between cities i and j cannot exceed the length of a two-leg path from i to some intermediate city k to j).
 - b. **Symmetry :** $d[i, j] = d[j, i]$ for any pair of cities i and j (the distance from i to j is the same as the distance from j to i).
4. A substantial majority of practical applications of the traveling salesman problem are its Euclidean instances.
5. They include, in particular, geometric ones, where cities correspond to points in the plane and distances are computed by the standard Euclidean formula.
6. Although the performance ratios of the nearest-neighbour and multifragment-heuristic algorithms remain unbounded for Euclidean instances, their accuracy ratios satisfy the following inequality for any such instance with $n \geq 2$ cities :

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4+w}{8}$$

where $f(s_a)$ and $f(s^*)$ are the lengths of the heuristic tour and shortest tour.

Que 5.28. What is an approximation algorithm ? What is meant by $p(n)$ approximation algorithms ? Discuss approximation algorithm for Travelling Salesman Problem.

AKTU 2019-20, Marks 07

Answer

Approximation algorithm : Refer Q. 5.23, Page 5-24B, Unit-5.

$p(n)$ approximation algorithm : A is a $p(n)$ approximate algorithm if and only if for every instance of size n , the algorithm achieves an approximation ratio of $p(n)$. It is applied to both maximization ($0 < C(i) \leq C^*(i)$) and minimization ($0 < C^*(i) \leq C(i)$) problem because of the maximization factor and costs are positive. $p(n)$ is always greater than 1.

Approximation algorithm for Travelling Salesman Problem (TSP) :

1. The key to designing approximation algorithm is to obtain a bound on the optimal value (OPT).
2. In the case of TSP, the minimum spanning tree gives a lower bound on OPT.
3. The cost of a minimum spanning tree is not greater than the cost of an optimal tour.

The algorithm is as follows :

1. Find a minimum spanning tree of G .

2. Duplicate each edge in the minimum spanning tree to obtain a Eulerian graph.
3. Find a Eulerian tour (J) of the Eulerian graph.
4. Convert J to a tour T by going through the vertices in the same order of T , skipping vertices that were already visited.

PART-5

Randomized Algorithm.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.29. Write short notes on randomized algorithms.

OR

Explain approximation and randomized algorithms.

AKTU 2017-18, Marks 10

Answer

Approximation algorithm : Refer Q. 5.23, Page 5–24B, Unit-5.

Randomized algorithm :

1. A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits and it is then allowed to use these random bits to influence its computation.
2. An algorithm is randomized if its output is determined by the input as well as the values produced by a random number generator.
3. A randomized algorithm makes use of a randomizer such as a random number generator.
4. The execution time of a randomized algorithm could also vary from run to run for the same input.
5. The algorithm typically uses the random bits as an auxiliary input to guide its behaviour in the hope of achieving good performance in the “average case”.
6. Randomized algorithms are particularly useful when it faces a malicious attacker who deliberately tries to feed a bad input to the algorithm.

Randomized algorithm are categorized into two classes :

- i. **Las Vegas algorithm :** This algorithm always produces the same output for the same input. The execution time of Las Vegas algorithm depends on the output of the randomizer.
- ii. **Monte Carlo algorithm :**
 - a. In this algorithm output might differ from run to run for the same input.

- b. Consider any problem for which there are only two possible answers, say yes and no.
- c. If a Monte Carlo algorithm is used to solve such a problem then the algorithm might give incorrect answers depending on the output of the randomizer.
- d. Then the requirement is that the probability of an incorrect answer from a Monte Carlo algorithm be low.

Que 5.30. Write a short note on randomized algorithm. Write its merits, and applications.

Answer

Randomized algorithm : Refer Q. 5.29, Page 5–30B, Unit-5.

Merits :

1. Simple.
2. High efficiency.
3. Better complexity bounds.
4. Random selection of good and bad choices.
5. Cost efficient.

Applications :

1. Randomized quick sort algorithm
2. Randomized minimum-cut algorithm
3. Randomized algorithm for N-Queens problem
4. Randomized algorithm for majority element

Que 5.31. Write the EUCLID'S GCD algorithm. Compute gcd (99, 78) with EXTENDED-EUCLID.

Answer

Euclid's GCD algorithm :

The inputs a and b are arbitrary non-negative integers.

EUCLID (a, b) {

if ($b == 0$)
then return a ;

else return EUCLID ($b, a \bmod b$); }

EXTEUCLID (a, b) {

//returns a triple (d, x, y) such that $d = \gcd(a, b)$

$//d == (a \times x + b \times y)$

if ($b == 0$) return $(a, 1, 0)$;

$(d_1, x_1, y_1) = \text{EXTEUCLID} (b, a \% b)$;

$d = d_1$;

$x = y_1$;

$y = x_1 - (a \text{ div } b) \times y_1$; //div = integer division

return (d, x, y) ;

}

Numerical :

Let $a = 99$ and $b = 78$

a	b	$[a/b]$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	-	3	1	0

- In the 5th receive ($a = 6, b = 3$), values from the 6th call ($b = 0$) has $d_1 = 3, x_1 = 1$ and $y_1 = 0$. Still within the 5th call we calculate that $d = d_1 = 3, x = y_1 = 0$ and $y = x_1 - (a \text{ div } b) \times y_1 = 1 - 2 \times 0 = 1$.
- In the 4th receive ($a = 15, b = 6$), the values $d_1 = 3, x_1 = 0$ and $y_1 = 1$ from the 5th call, then compute $x = y_1 = 1$ and $y = x_1 - (a \text{ div } b) \times y_1 = 0 - 2 \times 1 = -2$.
- In the 3rd receive ($a = 21, b = 15$), $x = -2$ and $y = x_1 - (a \text{ div } b) \times y_1 = 1 - (1 \times -2) = 3$.
- In the 2nd receive ($a = 78, b = 21$), $x = 3$ and $y = x_1 - (a \text{ div } b) \times y_1 = (-2) - 3 \times 3 = -11$.
- In the 1st receive ($a = 99, b = 78$), $x = -11$ and $y = x_1 - (a \text{ div } b) \times y_1 = 3 - 1 \times (-11) = 14$.
- The call EXTEUCLID (99, 78) return (3, -11, 14), so $\text{gcd}(99, 78) = 3$ and $\text{gcd}(99, 78) = 3 = 99 \times (-11) + 78 \times 14$.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. What is Fast Fourier Transformation and how it works ?

Ans. Refer Q. 5.1.

Q. 2. Explain the following string matching algorithms :

- Naive string matching
- Rabin-Karp algorithm
- Knuth-Morris-Pratt algorithm

Ans.

- Refer Q. 5.4.
- Refer Q. 5.8.
- Refer Q. 5.6.

Q. 3. Discuss the problem classes P, NP and NP-complete.

Ans. Refer Q. 5.11.

Q. 4. Differentiate NP-complete with NP-hard.

Ans. Refer Q. 5.15.

Q. 5. Show that CUQUE NP-complete.

Ans. Refer Q. 5.19.

Q. 6. Explain the following :

- a. Approximation algorithm
- b. Randomized algorithm

Ans.

- a. Refer Q. 5.23.
- b. Refer Q. 5.29.

Q. 7. Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function π for the pattern $ababbabbabbababbabb$ when the alphabet is $\Sigma = \{a, b\}$.

Ans. Refer Q. 5.8.





Introduction (2 Marks Questions)

1.1. Define the term algorithm.

Ans. Algorithm is a set of rules for carrying out calculation either by hand or on a machine. It is a finite step-by-step procedure to achieve a required result.

1.2. What are the steps needed to be followed while designing an algorithm ?

Ans. Steps of designing an algorithm :

1. Understanding the problem.
2. Decision making on capabilities of computational devices.
3. Specification of algorithm.
4. Algorithm verification.
5. Analysis of algorithm.
6. Implementation or coding of algorithm.

1.3. Define the following terms :

- i. Time efficiency
- ii. Space efficiency

Ans.

- i. **Time efficiency** : Time efficiency of an algorithm or a program is the amount of time it needs to run to completion.
- ii. **Space efficiency** : Space efficiency of an algorithm or program is the amount of memory it needs to run to completion.

1.4. Give an example of worst case time complexity.

Ans. While searching a particular element using linear searching method, if desired element is placed at the end of the list then we get worst time complexity.

1.5. Compare time complexity with space complexity.

Ans.

S. No.	Time complexity	Space complexity
1.	Time complexity is the amount of time required for an algorithm to complete its process.	Space complexity is the amount of memory needed for an algorithm to solve the problem.
2.	It is expressed using Big Oh(O), theta (θ) and omega (Ω) notation.	It is expressed only using Big Oh(O) notation.

1.6. What are the characteristics of the algorithm ?**AKTU 2017-18, Marks 02****Ans. Characteristics of algorithm :**

- Input and output :** The algorithm must accept zero or more inputs and must produce at least one output.
- Definiteness :** Each step of algorithm must be clear and unambiguous.
- Effectiveness :** Every step must be basic and essential.
- Finiteness :** Total number of steps used in algorithm should be finite.

1.7. What do you mean by asymptotic notations ?

Ans. Asymptotic notation is a shorthand way to represent the fastest possible and slowest possible running times for an algorithm.

1.8. Differentiate between Big Oh(O) and theta(θ) notations with suitable examples.**Ans.**

S. No.	O-Notation (Upper bound)	θ -Notation (Same order)
1.	Big-oh is formal method of expressing the upper bound of an algorithm's running time.	θ -notation bounds a function to within constant factors.
2.	It is the measure of the longest amount of time it could possibly take for the algorithm to complete.	It is the measure of the average amount of time it could possibly take for the algorithm to complete.
3.	For example : Time complexity of searching an element in array is $O(n)$.	For example : Worst case complexity of insertion sort is $\theta(n^2)$.

1.9. Why are asymptotic notations important ?

Ans. Asymptotic notations are important because :

- They give a simple characterization of algorithm efficiency.
- They allow the comparisons of the performances of various algorithms.

1.10. Solve the given recurrence $T(n) = 4T(n/4) + n$.

AKTU 2015-16, Marks 02

Ans. $T(n) = 4T(n/4) + n$

We will map this equation with

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

$$a = 4 \text{ and } b = 4, f(n) = n$$

$$\text{Now, } n^{\log_b a} = n^{\log_4 4} = n$$

i.e., case 2 of Master's theorem is applied. Thus, resulting solution is

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

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

1.11. What is a heap ? What are the different types of heaps ?

Ans. **Heap:** A heap is a specialized tree-based data structure that satisfies the heap property.

Different types of heap are :

- Min-heap
- Binary heap
- Binomial heap
- Fibonacci heap
- Max-heap
- Radix heap

1.12. Justify why quick sort is better than merge sort ?

AKTU 2015-16, Marks 02

Ans. Theoretically both quick sort and merge sort take $O(n \log n)$ time and hence time taken to sort the elements remains same. However, quick sort is superior to merge sort in terms of space.

Quick sort is in-place sorting algorithm where as merge sort is not in-place. In-place sorting means, it does not use additional storage space to perform sorting. In merge sort, to merge the sorted arrays it requires a temporary array and hence it is not in-place.

1.13. Why counting sort is stable ?

Ans. Counting sort is stable because numbers with the same value appear in the output array in the same order as they do in the input array.

1.14. What are the fundamental steps involved in algorithmic problem solving ?

AKTU 2016-17, Marks 02

Ans. Steps involved in algorithmic problem solving are :

- Characterize the structure of optimal solution.
- Recursively define the value of an optimal solution.
- By using bottom-up technique, compute value of optimal solution.
- Compute an optimal solution from computed information.

1.15. Write recursive function to find n^{th} Fibonacci number.

AKTU 2016-17, Marks 02

Ans. int fibo(int num)

```

{
if (num == 0)
{
return 0;
}
else if (num == 1)
{
return 1;
}
else
{
return(fibo(num - 1) + fibo(num - 2));
}
}
```

1.16. Write the names of various design techniques of algorithm.

AKTU 2016-17, Marks 02

Ans. Various design techniques of algorithm are :

- Divide and conquer
- Greedy approach
- Dynamic programming
- Branch and bound
- Backtracking algorithm

1.17. Solve the following recurrence using master method :

$$T(n) = 4T(n/3) + n^2$$

AKTU 2017-18, Marks 02

Ans. $T(n) = 4T(n/3) + n^2$

$$a = 4, \quad b = 3, \quad f(n) = n^2$$

$$n^{\log_b a} = n^{\log_3 4} = n^{1.261}$$

$$f(n) = \Omega(n^{\log_b a + E})$$

$$\text{Now, } af(n/b) \leq c f(n)$$

$$\frac{4}{3} f(n) \leq c f(n)$$

$$\frac{4}{3} n^2 \leq cn^2$$

$$c = \frac{4}{3}$$

Hence, $T(n) = \Theta(n^2)$

- 1.18. Name the sorting algorithm that is most practically used and also write its time complexity.**

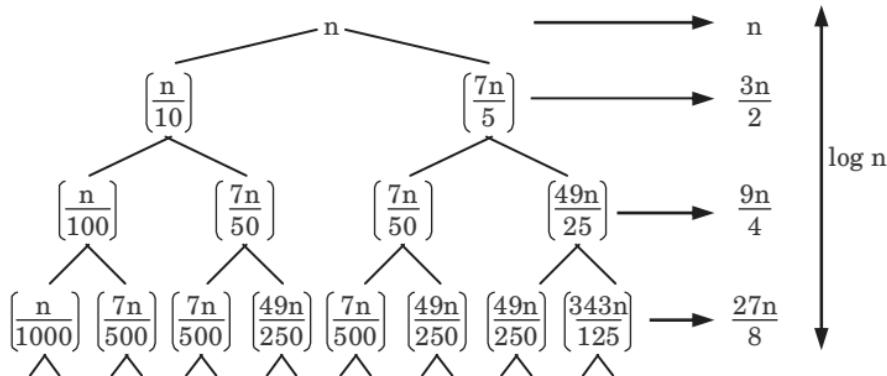
AKTU 2017-18, Marks 02

Ans. Quick sort algorithm is most practically used in sorting.
Time complexity of quick sort is $O(n \log n)$.

- 1.19. Find the time complexity of the recurrence relation
 $T(n) = n + T(n/10) + T(7n/5)$**

AKTU 2017-18, Marks 02

Ans.



$$\begin{aligned} T(n) &= \frac{3^0 n}{2^0} + \frac{3^1 n}{2^1} + \frac{3^2 n}{2^1} + \frac{3^3 n}{2^3} + \dots + \text{log } n \text{ times} \\ &= \Omega(n \log n) \end{aligned}$$

- 1.20. What is priority queue ?**

AKTU 2015-16, Marks 02

Ans. A priority queue is a collection of elements such that each element has been assigned a priority. The order in which the elements are deleted and processed according to the following rules :

- i. An element of higher priority is processed before any element of lower priority.

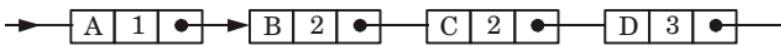


Fig. 1.

- ii. Two elements with the same priority are processed according to the order in which they were added to queue. This can be implemented as linked list, or as 2D array.

1.21. How do you compare the performance of various algorithms ?

AKTU 2019-20, Marks 02

Ans. To compare the performance of various algorithms first we measure its performance which depends on the time taken and the size of the problem. For this we measure the time and space complexity of various algorithms which is divided into different cases such as worst case, average case and best case.

1.22. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ immediately follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$$f_1(n) = n^{2.5}, f_2(n) = \sqrt{2^n}, f_3(n) = n + 10, f_4(n) = 10n, f_5(n) = 100n, \\ \text{and } f_6(n) = n^2 \log n$$

AKTU 2019-20, Marks 02

Ans. $f_3(n) = f_4(n) = f_5(n) < f_2(n) < f_6(n) < f_1(n)$

1.23. Rank the following by growth rate :

$$n, 2 \log \sqrt{n}, \log n, \log(\log n), \log^2 n, (\log n)^{\log n}, 4, (3/2)^n, n!$$

AKTU 2018-19, Marks 02

Ans. Rank in increasing order of growth rate is given as :

$$4, \log n, \log(\log n), \log^2 n, (\log n)\log n, \log n, 2 \log \sqrt{n}, n, n!, \left(\frac{3}{2}\right)^n$$

1.24. What do you mean by stability of a sorting algorithm ?

Explain its application.

AKTU 2018-19, Marks 02

Ans. **Stability of a sorting algorithm :** Let A be an array, and let $<$ be a strict weak ordering on the elements of A .

Sorting algorithm is stable if :

$i < j$ and $A[i] \equiv A[j]$ i.e., $A[i]$ comes before $A[j]$.

Stability means that equivalent elements retain their relative positions, after sorting.

Application : One application for stable sorting algorithms is sorting a list using a primary and secondary key. For example, suppose we wish to sort a hand of cards such that the suits are in the order clubs, diamonds, hearts, spades and within each suit, the cards are sorted by rank. This can be done by first sorting the cards by rank (using any sort), and then doing a stable sort by suit.



2

UNIT

Advanced Data Structures (2 Marks Questions)

- 2.1. What are the advantages of red-black tree over binary search tree ?**

Ans. The red-black tree is a self balanced tree which ensures the worst case time complexity as $O(\log N)$ with N numbers of nodes and worst case time complexity of binary search tree is $O(N)$. Hence, searching of any node in RB-tree becomes efficient than the binary search tree.

- 2.2. What is the largest possible number of internal nodes in a red-black tree with black height k ?**

Ans. Consider a red-black tree with black height k . If every node is black, then total number of internal nodes is $2k - 1$.

- 2.3. Discuss the different kinds of rotations in RB-tree.**

Ans. Rotations in RB-tree :

- Left rotation :** When left rotation on node 'x' is made then its right child becomes its parent.
- Right rotation :** When right rotation on node 'y' is made then its left child becomes its parent.

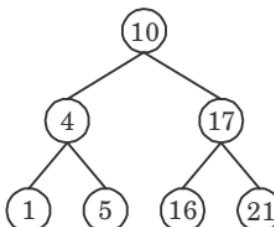
- 2.4. What is the running time of RB-Delete ?**

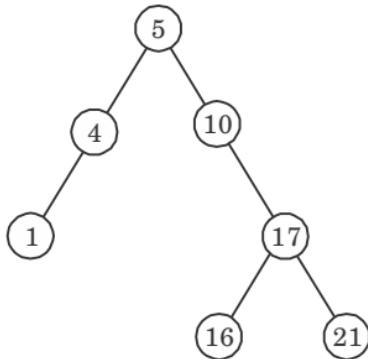
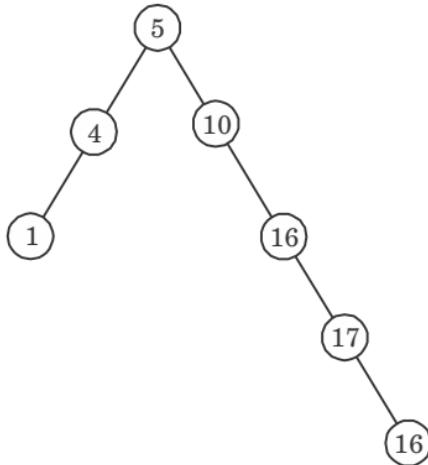
Ans. Running time of RB-delete is $O(\log n)$ time.

- 2.5. Draw BSTs of height 2, 3 and 4 on the set of keys {10, 4, 5, 16, 1, 17, 21}**

Ans. Keys = {10, 4, 5, 16, 1, 17, 21}

BST of height 2 :



BST of height 3 :**BST of height 4 :**

- 2.6. Draw all legal B-trees of minimum degree 2 that represent {10, 12, 13, 14, 15}.**

Ans. Keys : 10, 12, 13, 14, 15
Minimum degree : 2

Insert 10 :

10

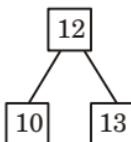
Insert 12 :

10	12
----	----

Insert 13 :

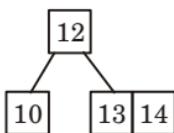
10	12	13
----	----	----

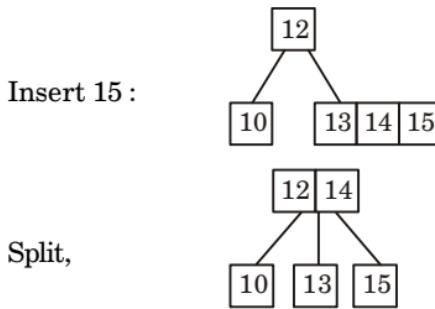
Split,



Insert 14 :

10	13	14
----	----	----





2.7. What are the operations performed for mergeable heaps ?

Ans. Various operations performed for mergeable heaps are :

- Make-heap ()
- Insert (H, x)
- Minimum (H)
- Extract-min (H)
- Union (H_1, H_2)

2.8. Explain binomial heap with properties.

AKTU 2015-16, Marks 02

Ans. Binomial heap is a type of data structure which keeps data sorted and allows insertion and deletion in amortized time.

Properties of binomial heap :

- There are 2^k nodes.
- The height of the tree is k .
- There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \dots, k$ (this is why the tree is called a “binomial” tree).
- Root has degree k (children) and its children are $B_{k-1}, B_{k-2}, \dots, B_0$ from left to right.

2.9. Discuss the application of Fibonacci heaps.

Ans. Application of Fibonacci heaps :

- In Dijkstra's algorithm for computing shortest path.
- In Prim's algorithm for finding minimum spanning tree.

2.10. What is a disjoint set ?

Ans. A disjoint set is a data structure $S = \{S_1, \dots, S_k\}$, or a collection of disjoint dynamic sets. Each set has a representative element, which never changes unless union with another set.

2.11. Define binary heap.

AKTU 2016-17, Marks 02

Ans. The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The tree is completely filled on all levels except possibly lowest.

2.12. Differentiate between complete binary tree and binary tree.

AKTU 2017-18, Marks 02

Ans.

S. No.	Complete binary tree	Binary tree
1.	In a complete binary tree every level, except possibly the last is completely filled, and all nodes in the last level are as far left as possible.	A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.
2	It can have between 1 and 2^{h-1} nodes at the last level h .	It can have between 2^{h+1} and $2^{h+1} - 1$ nodes at the last level h .

**2.13. What is advantage of binary search over linear search?
Also, state limitations of binary search.**

AKTU 2019-20, Marks 02

Ans. Advantages of binary search over linear search :

1. Input data needs to be sorted in binary search but not in linear search.
2. Linear search does the sequential access whereas binary search access data randomly.
3. Time complexity of linear search is $O(n)$ where binary search has time complexity $O(\log n)$.

Limitation of binary search :

1. List must be sorted.
2. It is more complicated to implement and test.

2.14. Prove that if $n >= 1$, then for any n -key B-tree of height h and minimum degree $t >= 2$, $h \leq \log_t ((n + 1)/2)$.

AKTU 2018-19, Marks 02

Ans. Proof :

1. The root contains at least one key.
2. All other nodes contain at least $t - 1$ keys.
3. There are at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^{i-1}$ nodes at depth i and $2t^{h-1}$ nodes at depth h .

$$n \geq 1 + (t - 1) \sum_{i=1}^h 2t^{i-1} = 1 + 2(t - 1) \left(\frac{t^h - 1}{t - 1} \right) = 2t^h - 1$$

4. So $t^h \leq (n + 1)/2$ as required.

Taking log both sides we get,

$$h \leq \log_t (n + 1)/2$$



3

UNIT

Graph Algorithm (2 Marks Questions)

3.1. Describe the general method for divide and conquer.

Ans. In divide and conquer method, a given problem is :

- Divided into smaller sub-problems.
- These sub-problems are solved independently.
- Combining all the solutions of sub-problems into a solution of the large problem.

3.2. Give various applications of divide and conquer.

Ans. Various applications divide and conquer strategy are :

- Binary search
- Merge sort
- Quick sort
- Strassen's matrix multiplication
- Finding maximum and minimum element

3.3. Describe convex hull problem.

Ans. There exists a set of points on a plane which is said to be convex if for any two points A and B in the set, the entire line segment with the end points at A and B belongs to the set.

3.4. Define activity selection problem.

Ans. Activity selection problem is a problem of scheduling a resource among several competing activity.

3.5. List out the disadvantages of divide and conquer algorithm.

AKTU 2016-17, Marks 02

Ans. Disadvantages of divide and conquer algorithm :

- Recursion is slow.
- Algorithm becomes complicated for large value of n .

3.6. When we can say that optimization problem has overlapping sub-problem ?

Ans. When a recursive algorithm revisits the same problem over and over again, we say that optimization problem has overlapping sub-problems.

3.7. Define greedy technique.

- Ans.** Greedy technique solves problem by making the choice that seems best at the particular moment.
 Greedy technique works if a problem exhibit two properties :
 i. Greedy choice property ii. Optimal sub-structure

3.8. Compare between greedy method and divide and conquer algorithm.**Ans.**

S. No.	Greedy algorithm	Divide and Conquer
1.	It is used to obtain optimum solution.	It is used to obtain solution to given problem.
2.	In greedy method, a set of feasible solution is generated and optimum solution is picked up.	In this technique, the problem is divided into smaller sub-problems.

3.9. How greedy method is used to solve the optimization problem ?

- Ans.** In greedy technique, the solution is computed through a sequence of steps. In each step, the partially constructed solution is expanded. This process is repeated until a complete solution to the problem is reached.

3.10. Give comparison between Prim's and Kruskal's algorithm.**Ans.**

S. No.	Prim's algorithm	Kruskal's algorithm
1.	It is used to obtain minimum spanning tree (MST).	It is also used to obtain minimum spanning tree (MST).
2.	It starts to build MST from any of the node.	It starts to build MST from minimum weighted edge in the graph.
3.	Run faster in dense graph.	Run faster in sparse graph.

3.11. Discuss principle of optimality.**AKTU 2016-17, Marks 02**

- Ans.** Principle of optimality states that in an optimal sequence of decisions or choices, each subsequence must also be optimal.

3.12. Define bottleneck spanning tree.

Ans. A bottleneck spanning tree T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G .

3.13. Can the minimum spanning tree change ?

Ans. Yes, the spanning tree will change only in the case when some of the weights of edges are changed.

3.14. Briefly explain the Prim's algorithm.

AKTU 2016-17, Marks 02

Ans. First it chooses a vertex and then chooses an edge with smallest weight incident on that vertex. The algorithm involves following steps :

Step 1 : Choose any vertex V_1 of G .

Step 2 : Choose an edge $e_1 = V_1V_2$ of G such that $V_2 \neq V_1$ and e_1 has smallest weight among the edge e of G incident with V_1 .

Step 3 : If edges e_1, e_2, \dots, e_i have been chosen involving end points V_1, V_2, \dots, V_{i+1} , choose an edge $e_{i+1} = V_jV_k$ with $V_j = \{V_1, \dots, V_{i+1}\}$ and $V_k \notin \{V_1, \dots, V_{i+1}\}$ such that e_{i+1} has smallest weight among the edges of G with precisely one end in $\{V_1, \dots, V_{i+1}\}$.

Step 4 : Stop after $n - 1$ edges have been chosen. Otherwise goto step 3.

3.15. Explain element searching techniques using divide and conquer approach.

AKTU 2015-16, Marks 02

Ans. Binary search is a searching technique which uses divide and conquer.

In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then recursively call the algorithm for left side of middle element, else recursively call the algorithm for right side of middle element.

3.16. What are greedy algorithms ? Explain their characteristics ?

AKTU 2019-20, Marks 02

Ans. **Greedy algorithms :** Greedy algorithms are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.

Characteristics of greedy algorithm :

1. Greedy algorithms are most efficient.
2. For every instance of input greedy algorithms makes a decision and continues to process further set of input.
3. The other input values at the instance of decision are not used in further processing.

3.17. Define feasible and optimal solution.

AKTU 2019-20, Marks 02

Ans. **Feasible solution :** A feasible solution is a set of values for the decision variables that satisfies all of the constraints in an optimization problem. The set of all feasible solutions defines the feasible region of the problem.

Optimal solution : An optimal solution is a feasible solution where the objective function reaches its maximum (or minimum) value.

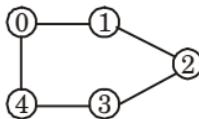
3.18. Compare adjacency matrix and linked adjacency lists representation of graph with suitable example/diagram.

AKTU 2018-19, Marks 02

Ans.

S. No.	Adjacency matrix	Linked adjacency list
1.	An adjacency matrix is a square matrix used to represent a finite graph.	Linked adjacency list is a collection of unordered lists used to represent a finite graph.
2.	The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.	Each list describes the set of adjacent vertices in the graph.
3.	Space complexity in the worst case is $O(V ^2)$.	Space complexity in the worst case is $O(V + E)$.

For example : Consider the graph :



Using adjacency matrix :

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \left[\begin{matrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 1 & 0 \\ 3 & 0 & 0 & 1 & 0 & 1 \\ 4 & 1 & 0 & 0 & 1 & 0 \end{matrix} \right] \end{matrix}$$

Using adjacency list :

0	→ [1] → [4 /]
1	→ [0 /] → [2 /]
2	→ [1 /] → [3 /]
3	→ [4 /] → [2 /]
4	→ [3 /] → [0 /]





4
UNIT

Dynamic Programming, Backtracking and Branch and Bound (2 Marks Questions)

4.1. Describe dynamic programming.

Ans. Dynamic programming is a technique for solving problems with overlapping subproblems.

4.2. Give two differences between dynamic programming and divide and conquer techniques.

Ans.

S. No.	Dynamic programming	Divide and conquer
1.	In dynamic programming, many decision sequences are generated and all the overlapping subinstances are considered.	The problem is divided into small subproblems. These subproblems are solved independently.
2.	In dynamic computing, duplication in solutions is avoided totally.	In this method, duplications in subsolutions are neglected.

4.3. Explain dynamic programming. How it is different from greedy approach ?

AKTU 2015-16, Marks 02

OR

Differentiate between greedy technique and dynamic programming.

AKTU 2017-18, Marks 02

Ans. Dynamic programming is a technique for solving problems with overlapping subproblems.

In dynamic programming, problems are solved by using divide and conquer method but in greedy approach problem are solved by making the choice that seems best at the particular moment.

Difference :

S. No.	Dynamic programming	Greedy method
1.	Dynamic programming can be thought of as ‘smart’ recursion. It often requires one to break down a problem into smaller components that can be cached.	A greedy algorithm is one that at a given point in time makes a local optimization.

2. Dynamic programming would solve all dependent subproblems and then select one that would lead to an optimal solution.	Greedy algorithms have a local choice of the subproblem that will lead to an optimal answer.
--	--

4.4. State the single source shortest path.**AKTU 2017-18, Marks 02**

Ans. Single source shortest path problem states that in a given graph $G = (V, E)$ we can find a shortest path from given source vertex $s \in V$ to every vertex $v \in V$.

4.5. What is the purpose of Floyd-Warshall's algorithm ?

Ans. The purpose of Floyd-Warshall's algorithm is to find the shortest path between all pairs of vertices in a graph. It uses a number of matrices of size $n \times n$ where n is the number of vertices.

4.6. What are the constraints used for solving the problem in backtracking algorithm ?

Ans. There are two types of constraints used to solve the problem in backtracking :

- i. Explicit constraints
- ii. Implicit constraints

4.7. Write the major advantage of backtracking algorithm.

Ans. The major advantage of backtracking algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.

4.8. State the difference between backtracking and branch and bound.**AKTU 2015-16, 2016-17, 2017-18; Marks 02**

Ans.

S. No.	Backtracking	Branch and bound
1.	Solution for backtracking is traced using depth first search.	In this method, it is not necessary to use depth first search for obtaining the solution, even the breadth first search, best first search can be applied.
2.	Typically decision problems can be solved using backtracking.	Typically optimization problems can be solved using branch and bound.

4.9. What is the running time complexity of 8-Queens problem ?**AKTU 2016-17, Marks 02**

Ans. The running time complexity of 8-Queens problem is $O(P(n)n!)$ where $P(n)$ is polynomial in n .

4.10. Define graph colouring.**AKTU 2017-18, Marks 02**

Ans. Graph colouring is a problem of colouring each vertex in graph in such a way that no two adjacent vertices have same colour and m -colours are used.

4.11. What is optimal colouring ?

Ans. If the degree of given graph is d then we colour it with $d + 1$ colours. The least number of colours needed to colour the graph is called its chromatic number. Such type of colouring is called optimal colouring.

4.12. What is Travelling Salesman Problem (TSP) ?

Ans. Travelling salesman problem is a problem in which a salesman visits ' m ' cities, in a way that all cities must be visited at once and in the end, he returns to the city from where he started, with minimum cost.

4.13. Mention the three common techniques used in amortized analysis and write the characteristics of one of its technique.

Ans. Three common techniques used in amortized analysis are :

- Aggregate method
- Accounting method
- Potential method

Characteristics of aggregate method are :

- The amortized cost is $T(n)/n$ per operation.
- It gives the average performance of each operation in the worst case.

4.14. Find out Hamiltonian cycles in complete graph having ' n ' vertices.**AKTU 2015-16, Marks 02**

Ans. The answer depends on how we think that two Hamiltonian cycles are equal. Take K_3K_3 as example. Let's call its vertices $1_1, 2_2$ and 3_3 . Then do we consider $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ same as $2 \rightarrow 3 \rightarrow 1 \rightarrow 2_2 \rightarrow 3 \rightarrow 1 \rightarrow 2$? If yes, $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is the same as $2 \rightarrow 3 \rightarrow 1 \rightarrow 2_2 \rightarrow 3 \rightarrow 1 \rightarrow 2$, which is the same as $3 \rightarrow 1 \rightarrow 2 \rightarrow 3_3 \rightarrow 1 \rightarrow 2 \rightarrow 3$ in K_3K_3 . Then we will have two Hamiltonian cycles $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $1 \rightarrow 3 \rightarrow 2 \rightarrow 1_1 \rightarrow 3 \rightarrow 2 \rightarrow 1$. Moreover, if we consider $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $1 \rightarrow 3 \rightarrow 2 \rightarrow 1_1 \rightarrow 3 \rightarrow 2 \rightarrow 1$ being the same because the second one is

obtained by reversing direction the first one, then we have only one Hamiltonian cycle in $K_3 K_3$.

For general $K_n K_n$, it's the same. $1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1_1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1$ is the same as $2 \rightarrow \dots \rightarrow n \rightarrow 1 \rightarrow 2_2 \rightarrow \dots \rightarrow n \rightarrow 1 \rightarrow 2$ is the same as $\dots n \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow nn \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow n$. And the $1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1_1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1$ and $1 \rightarrow n \rightarrow \dots \rightarrow 2 \rightarrow 1_1 \rightarrow n \rightarrow \dots \rightarrow 2 \rightarrow 1$ being the same because the second one is obtained by reversing direction the first one. So, we have altogether $12(n-1)!$ $12(n-1)!$ Hamiltonian cycles in $K_n K_n$.

- 4.15. Find the subsets of sum of following problem. Given total elements are $(S) = \{4, 2, 7, 6, 8\}$ and maximum SUM is $(X) = 8$.**

AKTU 2015-16, Marks 02

Ans.

Initially subset = {s}	Sum(x) = 8	
4	4	Add next element
4, 2	6 ∵ 6 < 8	Add next element
4, 2, 7	13	Sum exceeds X = 8 Hence backtrack
4, 2, 6	12	Sum exceeds X = 8 ∴ Backtrack
4, 2, 8	14	Sum exceeds X = 8 ∴ Not feasible Hence backtrack
4, 2		Add next element
4, 7	11	Backtrack
2, 7	9	Backtrack
2, 6	8	Solution obtained as sum = 8 = X.
Also		
2		Backtrack
	8	Backtrack and add element
8		Solution obtained as sum = 8 = X

- 4.16. Explain application of graph colouring problem.**

AKTU 2018-19, Marks 02

Ans. Application of graph colouring problem :

- Sudoku :** Sudoku is a variation of graph colouring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.
- Register allocation :** In compiler optimization, register allocation is the process of assigning a large number of target program variables

onto a small number of CPU registers. This problem is also a graph colouring problem.

3. **Bipartite graphs :** We can check if a graph is bipartite or not by colouring the graph using two colours. If a given graph is 2-colourable, then it is bipartite, otherwise not.
4. **Map colouring :** Geographical maps of countries or states where no two adjacent cities cannot be assigned same colour.

4.17. Define principle of optimality. When and how dynamic programming is applicable ? AKTU 2018-19, Marks 02

Ans. **Principle of optimality :** Refer Q. 3.11, Page SQ-12B, Unit-3, Two Marks Questions.

Dynamic programming is mainly applicable where the solution of one sub-problem is needed repeatedly. In this procedure, the solutions of sub-problems are stored in a table, so that there is no need to re-compute the sub-problems and can be directly accessed from the table if required.





Selected Topics (2 Marks Questions)

5.1. Define Fast Fourier Transformation (FFT).

Ans. The Fast Fourier Transform (FFT) is an algorithm that computes a Discrete Fourier Transform (DFT) of n -length vector in $O(n \log n)$ time. In the FFT algorithm, we apply the divide and conquer approach to polynomial evaluation by observing that if n is even, we can divide a degree $(n - 1)$ polynomial.

5.2. Discuss the term string matching.

Ans. String matching algorithms, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

5.3. Write about Knuth-Morris-Pratt algorithm.

Ans. It is the first linear time string matching algorithm. In this, we have to examine all the characters in the text and pattern at least once. The running time is $O(n + m)$.

5.4. Differentiate between decision problem and optimization problem.

Ans.

S. No.	Decision problem	Optimization problem
i.	Any problem for which answer is either yes or no is called decision problem.	Any problem that involves the identification of optimal cost is called optimization problem.
ii.	The algorithm for decision problem is called decision algorithm.	Typically optimization problems can be solved using branch and bound.

5.5. Define P, NP and NP-complete in decision problems.

AKTU 2015-16, 2016-17; Marks 02

Ans. **P-polynomial time :** These problems can be solved in polynomial time, which take time like $O(n)$, $O(n^2)$, $O(n^3)$ like finding maximum

element in an array or to check whether a string is palindrome or not are P problems.

Non deterministic polynomial time : These problem cannot be solved in polynomial time like TSP (Travelling Salesman Problem) or subset sum are NP problem.

But NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

NP-complete : The group of problems which are both in NP and NP-hard are known as NP-complete problem.

5.6. When a language is said to be NP-complete and NP-hard ?

Ans. A language $L \subseteq \{0, 1\}^*$ is NP-complete, if it satisfies the following two properties :

- i. $L \in NP$ and
- ii. For every $L' \in NP$, $L' \leq PL$

If a language L satisfies property (ii), but not necessarily property (i), we say that L is NP-hard.

5.7. Describe satisfiability.

Ans. A given boolean formula is satisfiable if there is a way to assign truth values (0 or 1) to the variable such that the final result is 1.

5.8. Define circuit satisfiability.

Ans. A combinational circuit is called circuit satisfiable if for set of inputs applied, output of this circuit should always be one.

5.9. Name some NP-complete problems.

Ans. **NP-complete problems are :**

- i. The 0/1 knapsack problem
- ii. Hamiltonian cycle
- iii. Travelling salesman problem

5.10. Define vertex cover.

Ans. A vertex cover to an undirected graph $G = (V, E)$ is a subset of $V' \subseteq V$ such that if edge $(u, v) \in G$ then $u \in V'$ or $v \in V'$ (or both). This optimal vertex cover is the optimization version of an NP-complete problem but it is not too hard to find a vertex cover that is near optimal.

5.11. Name any three problems that cannot be solved by polynomial time algorithm.

Ans. **Three problems that cannot be solved by polynomial time algorithm are :**

1. Halting problem
2. Decision problem
3. Optimization problem

5.12. What are polynomial-time solvable and polynomial-time verifiable algorithms ?

Ans. **Polynomial-time solvable algorithm :**

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k where n is the complexity of input.

Polynomial-time verifiable algorithm : A polynomial-time algorithms A is said to be polynomial time verifiable if it has following properties :

1. The input to A consists of an instance I of X (X is a decision problem) and a string s such that the length of s is bounded by some polynomials in the size of I .
2. The output of A is either yes or no.
3. If I is a negative instance of X . Then the output of A is “no” regardless of the value of s .
4. If I is a positive instance of X_1 then there is at least one choice of s for which A output “yes”.

5.13. List three problems that have polynomial time algorithm.

Ans. **Problems that have polynomial time algorithm are :**

- i. Binary search
- ii. Evaluation of polynomial
- iii. Sorting a list

5.14. List any two approximation algorithms.

Ans. **Two approximation algorithms are :**

- i. Nearest neighbour algorithm
- ii. Greedy algorithm

5.15. Mention the types of randomized algorithms.

Ans. **Two types of randomized algorithms are :**

- i. Las Vegas algorithms
- ii. Monte Carlo algorithm

5.16. Describe the advantages of randomized algorithms.

Ans. **Advantages of randomized algorithms are :**

- i. These algorithms are simple to implement.
- ii. These algorithms are many times efficient than traditional algorithms.

5.17. What do you mean by clique ?

Ans. A clique in an undirected graph $G = (V, E)$ is a subset $V' \subset V$ of vertices, each pair of which is connected by an edge in E .
 $CLIQUE = \{(G, R) : G \text{ is a graph with a clique of size } R\}$.

5.18. Explain applications of FFT.

AKTU 2019-20, Marks 02

Ans. Application of Fast Fourier Transform :

1. Signal processing.
2. Image processing.
3. Fast multiplication of large integers.
4. Solving Poisson's equation nearly optimally.

5.19. What do you mean by polynomial time reduction ?

AKTU 2019-20, Marks 02

Ans. A polynomial time reduction is a method for solving one problem using another. For example, if a hypothetical subroutine solving the second problem exists, then the first problem can be solved by transforming or reducing it to inputs for the second problem and calling the subroutine one or more times.

5.20. What are approximation algorithms ? What is meant by $p(n)$ approximation algorithms ?

AKTU 2018-19, Marks 02

Ans. **Approximation algorithm :** An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.

$p(n)$ approximation algorithm : A is a $p(n)$ approximate algorithm if and only if for every instance of size n , the algorithm achieves an approximation ratio of $p(n)$. It is applied to both maximization ($0 < C(i) \leq C^*(i)$) and minimization ($0 < C^*(i) \leq C(i)$) problem because of the maximization factor and costs are positive. $p(n)$ is always greater than 1.



B. Tech.**(SEM. V) ODD SEMESTER THEORY
EXAMINATION, 2015-16****DESIGN AND ANALYSIS OF ALGORITHMS**

Time : 3 Hours**Max. Marks : 100**

SECTION - A

Note : Attempt **all** parts. All parts carry equal marks. Write answer of each part in short. $(2 \times 10 = 20)$

1. a. Justify why quick sort is better than merge sort ?
- b. What is priority queue ?
- c. Find out Hamiltonian cycles in complete graph having ' n ' vertices.
- d. Explain binomial heap with properties.
- e. Explain element searching techniques using divide and conquer approach.
- f. Find the subsets of sum of following problem. Given total elements are (S) = {4, 2, 7, 6, 8} and maximum SUM is (X) = 8.
- g. Explain dynamic programming. How it is different from greedy approach ?
- h. Solve the given recurrence $T(n) = 4T(n/4) + n$.
- i. Differentiate between backtracking and branch and bound programming approach.
- j. Explain the P, NP and NP-complete in decision problems.

SECTION - B

- Note :** Attempt any **five** questions from this section : $(10 \times 5 = 50)$
2. Explain insertion in red-black tree. Show steps for inserting 1, 2, 3, 4, 5, 6, 7, 8 and 9 into empty RB-tree.

3. Discuss knapsack problem with respect to dynamic programming approach. Find the optimal solution for given problem, w (weight set) = {5, 10, 15, 20} and W (Knapsack size) = 25 and v = {50, 60, 120, 100}.
4. What is heap sort ? Apply heap sort algorithm for sorting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Also deduce time complexity of heap sort.
5. Explain B-tree and insert elements B, Q, L, F into B-tree Fig. 1 then apply deletion of elements F, M, G, D, B on resulting B-tree.

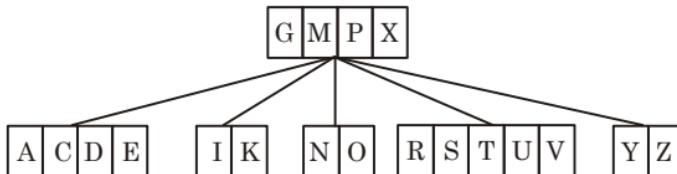


Fig. 1.

6. Write an algorithm for solving N -Queens problem. Show the solution of 4-Queens problem using backtracking approach.
7. Explain greedy single source shortest path algorithm with example.
8. What is string matching algorithm ? Explain Rabin-Karp method with examples.
9. Explain approximation algorithms with suitable examples.

SECTION - C

Note : Attempt any **two** questions from this section : $(15 \times 2 = 30)$

10. What is Fibonacci heap ? Explain CONSOLIDATE operation with suitable example for Fibonacci heap.
11. What is minimum spanning tree ? Explain Prim's algorithm and find MST of graph Fig. 2.

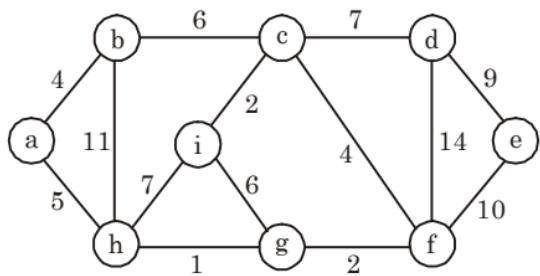


Fig. 2.

12. Explain TSP (Travelling Salesman) problem with example.
Write an approach to solve TSP problem.



SOLUTION OF PAPER (2015-16)

SECTION - A

Note : Attempt **all** parts. **All** parts carry equal marks. Write answer of each part in short. **(2 × 10 = 20)**

1. a. Justify why quick sort is better than merge sort ?

Ans. Theoretically both quick sort and merge sort take $O(n \log n)$ time and hence time taken to sort the elements remains same. However, quick sort is superior to merge sort in terms of space. Quick sort is in-place sorting algorithm where as merge sort is not in-place. In-place sorting means, it does not use additional storage space to perform sorting. In merge sort, to merge the sorted arrays it requires a temporary array and hence it is not in-place.

b. What is priority queue ?

Ans. A priority queue is a collection of elements such that each element has been assigned a priority. The order in which the elements are deleted and processed according to the following rules :

- i. An element of higher priority is processed before any element of lower priority.
- ii. Two elements with the same priority are processed according to the order in which they were added to queue. This can be implemented as linked list, or as 2D array.

c. Find out Hamiltonian cycles in complete graph having ' n ' vertices.

Ans. The answer depends on how we think that two Hamiltonian cycles are equal. Take K_3K_3 as example. Let's call its vertices $1_1, 2_2$ and 3_3 . Then do we consider $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ same as $2 \rightarrow 3 \rightarrow 1 \rightarrow 2_2 \rightarrow 3 \rightarrow 1 \rightarrow 2$? If yes, $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is the same as $2 \rightarrow 3 \rightarrow 1 \rightarrow 2_2 \rightarrow 3 \rightarrow 1 \rightarrow 2$, which is the same as $3 \rightarrow 1 \rightarrow 2 \rightarrow 3_3 \rightarrow 1 \rightarrow 2 \rightarrow 3$ in K_3K_3 . Then we will have two Hamiltonian cycles $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $1 \rightarrow 3 \rightarrow 2 \rightarrow 1_1 \rightarrow 3 \rightarrow 2 \rightarrow 1$. Moreover, if we consider $1 \rightarrow 2 \rightarrow 3 \rightarrow 1_1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ and $1 \rightarrow 3 \rightarrow 2 \rightarrow 1_1 \rightarrow 3 \rightarrow 2 \rightarrow 1$ being the same because the second one is obtained by reversing direction the first one, then we have only one Hamiltonian cycle in K_3K_3 .

For general K_nK_n , it's the same. $1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1_1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1$ is the same as $2 \rightarrow \dots \rightarrow n \rightarrow 1 \rightarrow 2_2 \rightarrow \dots \rightarrow n \rightarrow 1 \rightarrow 2$ is the same as $\dots n \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow nn \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow n$. And the $1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1_1 \rightarrow 2 \rightarrow \dots \rightarrow n \rightarrow 1$ and $1 \rightarrow n \rightarrow \dots \rightarrow 2 \rightarrow 1_1 \rightarrow n \rightarrow \dots \rightarrow 2 \rightarrow 1$ being the same because the second one is obtained by reversing direction the first one. So, we have altogether $12(n-1)! 12(n-1)!$ Hamiltonian cycles in K_nK_n .

d. Explain binomial heap with properties.

Ans. Binomial heap is a type of data structure which keeps data sorted and allows insertion and deletion in amortized time.

Properties of binomial heap :

- i. There are 2^k nodes.
- ii. The height of the tree is k .
- iii. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \dots, k$ (this is why the tree is called a “binomial” tree).

- iv Root has degree k (children) and its children are $B_{k-1}, B_{k-2}, \dots, B_0$ from left to right.

e. Explain element searching techniques using divide and conquer approach.

Ans. Binary search is a searching technique which uses divide and conquer.

In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then recursively call the algorithm for left side of middle element, else recursively call the algorithm for right side of middle element.

f. Find the subsets of sum of following problem. Given total elements are (S) = {4, 2, 7, 6, 8} and maximum SUM is (X) = 8.

Ans.

Initially subset = {s}	Sum(x) = 8	
4	4	Add next element
4, 2	6 ∵ 6 < 8	Add next element
4, 2, 7	13	Sum exceeds X = 8 Hence backtrack
4, 2, 6	12	Sum exceeds X = 8 ∴ Backtrack
4, 2, 8	14	Sum exceeds X = 8 ∴ Not feasible Hence backtrack
4, 2		Add next element
4, 7	11	Backtrack
2, 7	9	Backtrack
2, 6	8	Solution obtained as sum = 8 = X.
Also		
2		Backtrack
8	8	Backtrack and add element Solution obtained as sum = 8 = X

- g. Explain dynamic programming. How it is different from greedy approach ?**

Ans. Dynamic programming is a technique for solving problems with overlapping subproblems.

In dynamic programming, problems are solved by using divide and conquer method but in greedy approach problem are solved by making the choice that seems best at the particular moment.

Difference :

S. No.	Dynamic programming	Greedy method
1.	Dynamic programming can be thought of as 'smart' recursion. It often requires one to break down a problem into smaller components that can be cached.	A greedy algorithm is one that at a given point in time makes a local optimization.
2.	Dynamic programming would solve all dependent subproblems and then select one that would lead to an optimal solution.	Greedy algorithms have a local choice of the subproblem that will lead to an optimal answer.

- h. Solve the given recurrence $T(n) = 4T(n/4) + n$.**

Ans. $T(n) = 4T(n/4) + n$

We will map this equation with

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

$$a = 4 \text{ and } b = 4, f(n) = n$$

Now, $n^{\log_b a} = n^{\log_4 4} = n$

i.e., case 2 of Master's theorem is applied. Thus, resulting solution is

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

$$= \theta(n \log n)$$

- i. Differentiate between backtracking and branch and bound programming approach.**

Ans.

S. No.	Backtracking	Branch and bound
1.	Solution for backtracking is traced using depth first search.	In this method, it is not necessary to use depth first search for obtaining the solution, even the breadth first search, best first search can be applied.
2.	Typically decision problems can be solved using backtracking.	Typically optimization problems can be solved using branch and bound.

j. Explain the P, NP and NP-complete in decision problems.

Ans. **P-polynomial time :** These problems can be solved in polynomial time, which take time like $O(n)$, $O(n^2)$, $O(n^3)$ like finding maximum element in an array or to check whether a string is palindrome or not are *P* problems.

Non deterministic polynomial time : These problem cannot be solved in polynomial time like TSP (Travelling Salesman Problem) or subset sum are NP problem.

But NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

NP-complete : The group of problems which are both in NP and NP-hard are known as NP-complete problem.

SECTION - B

Note : Attempt any **five** questions from this section : **(10 × 5 = 50)**

2. Explain insertion in red-black tree. Show steps for inserting 1, 2, 3, 4, 5, 6, 7, 8 and 9 into empty RB-tree.

Ans. **Insertion :**

- i. We begin by adding the node as we do in a simple binary search tree and colouring it red.

RB-INSERT(T, z)

1. $y \leftarrow \text{nil}$ [T]
2. $x \leftarrow \text{root}$ [T]
3. while $x \neq \text{nil}$ [T]
 4. do $y \leftarrow x$
 5. if $\text{key}[z] < \text{key}[x]$
 6. then $x \leftarrow \text{left}$ [x]
 7. else $x \leftarrow \text{right}$ [x]
8. $p[z] \leftarrow y$
9. if $y = \text{nil}$ [T]
 10. then root [T] $\leftarrow z$
 11. else if $\text{key}[z] < \text{key}[y]$
 12. then left [y] $\leftarrow z$
 13. else right [y] $\leftarrow z$
14. $\text{left}[z] \leftarrow \text{nil}$ [T]
15. $\text{right}[z] \leftarrow \text{nil}$ [T]
16. $\text{colour}[z] \leftarrow \text{RED}$
17. RB-INSERT-FIXUP(T, z)

- ii. Now, for any colour violation, RB-INSERT-FIXUP procedure is used.

RB-INSERT-FIXUP(T, z)

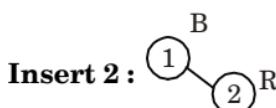
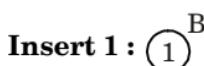
1. while $\text{colour}[p[z]] = \text{RED}$
2. do if $p[z] = \text{left}[p[p[z]]]$
 3. then $y \leftarrow \text{right}[p[p[z]]]$
 4. if $\text{colour}[y] = \text{RED}$

- ```

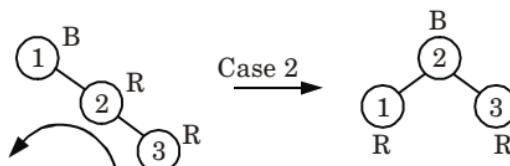
5. then colour[p[z]] ← BLACK ⇒ case 1
6. colour[y] ← BLACK ⇒ case 1
7. colour[p[p[z]]] ← RED ⇒ case 1
8. z ← p[p[z]] ⇒ case 1
9. else if z = right[p[z]]
10. then z ← p[z] ⇒ case 2
11. LEFT-ROTATE(T, z) ⇒ case 2
12. colour[p[z]] ← BLACK ⇒ case 3
13. colour[p[p[z]]] ← RED ⇒ case 3
14. RIGHT-ROTATE(T, p[p[z]]) ⇒ case 3
15. else (same as then clause with "right" and "left" exchanged)
16. colour[root[T]] ← BLACK

```

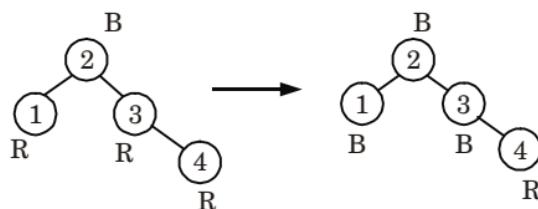
### Numerical :



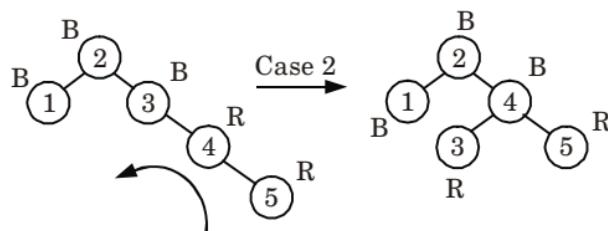
### **Insert 3 :**

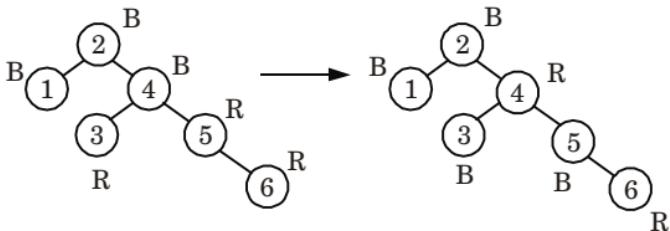
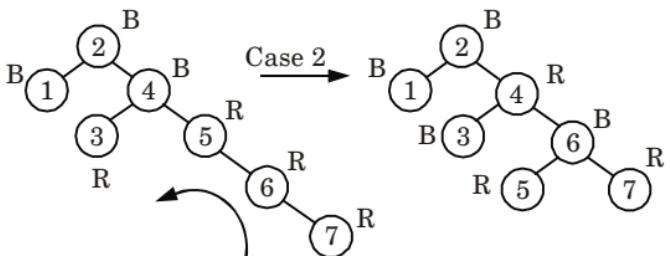
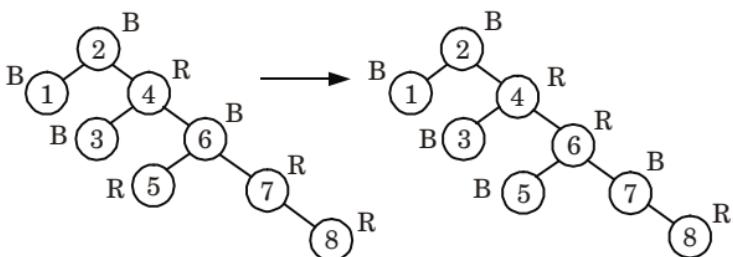
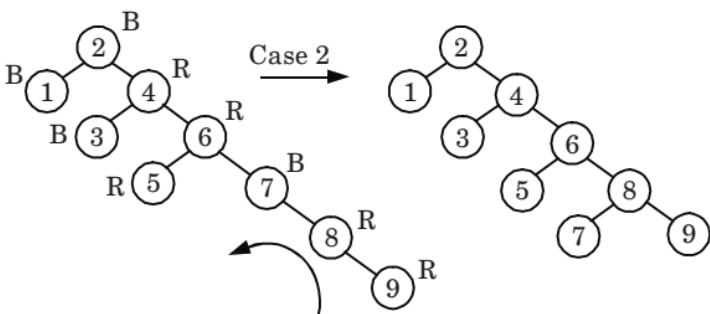


#### **Insert 4 :**



### **Insert 5 :**



**Insert 6 :****Insert 7 :****Insert 8 :****Insert 9 :**

3. Discuss knapsack problem with respect to dynamic programming approach. Find the optimal solution for given problem,  $w$  (weight set) = {5, 10, 15, 20} and  $W$  (Knapsack size) = 25 and  $v$  = {50, 60, 120, 100}.

**Ans.** Knapsack problem with respect to dynamic programming approach :

**Dynamic 0/1-knapsack( $v, w, n, W$ ) :**

1. for ( $w = 0$  to  $W$ )  $V[0, w] = 0$
2. for ( $i = 1$  to  $n$ )

3.     for ( $w = 0$  to  $W$ )
4.        if ( $w[i] \leq w$ ) then
5.             $V[i, w] = \max\{V[i - 1, w], v[i] + V[i - 1, w - w[i]]\};$
6.        else  $V[i, w] = V[i - 1, w];$
7.     return  $V[n, W];$

Now, as we know that  $V[n, W]$  is the total value of selected items, the can be placed in the knapsack. Following steps are used repeatedly to select actual knapsack item.

Let,  $i = n$  and  $k = W$  then

while ( $1 > 0$  and  $k > 0$ )

{

    if ( $V[i, k] \neq V[i - 1, k]$ ) then

        mark  $i^{\text{th}}$  item as in knapsack

$i = i - 1$  and  $k = k - w_i$  // selection of  $i^{\text{th}}$  item

    else

$i = i - 1$  //do not select  $i^{\text{th}}$  item

}

### Numerical :

$$w = \{5, 10, 15, 20\}$$

$$W = 25$$

$$v = \{50, 60, 120, 100\}$$

Initially,

| Item  | $w_i$ | $v_i$ |
|-------|-------|-------|
| $I_1$ | 5     | 50    |
| $I_2$ | 10    | 60    |
| $I_3$ | 15    | 120   |
| $I_4$ | 20    | 100   |

Taking value per weight ratio, i.e.,  $p_i = v_i/w_i$

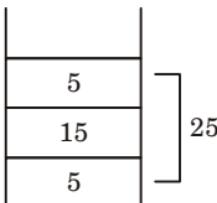
| Item  | $w_i$ | $v_i$ | $p_i = v_i/w_i$ |
|-------|-------|-------|-----------------|
| $I_1$ | 5     | 50    | 10              |
| $I_2$ | 10    | 60    | 6               |
| $I_3$ | 15    | 120   | 8               |
| $I_4$ | 20    | 100   | 5               |

Now, arrange the value of  $p_i$  in decreasing order.

| Item  | $w_i$ | $v_i$ | $p_i = v_i/w_i$ |
|-------|-------|-------|-----------------|
| $I_1$ | 5     | 50    | 10              |
| $I_3$ | 15    | 120   | 8               |
| $I_2$ | 10    | 60    | 6               |
| $I_4$ | 20    | 100   | 5               |

Now, fill the knapsack according to decreasing value of  $p_i$ .

First we choose item  $I_1$  whose weight is 5, then choose item  $I_3$  whose weight is 15. Now the total weight in knapsack is  $5 + 15 = 20$ . Now, next item is  $I_2$  and its weight is 10, but we want only 5. So, we choose fractional part of it, i.e.,



The value of fractional part of  $I_2$  is,

$$= \frac{60}{10} \times 5 = 30$$

Thus, the maximum value is,

$$= 50 + 120 + 3 = 200$$

4. What is heap sort ? Apply heap sort algorithm for sorting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Also deduce time complexity of heap sort.

**Ans. Heap sort :**

1. Heap sort is a comparison based sorting technique based on binary heap data structure.
2. Heap sort finds the largest element and puts it at the end of array, then the second largest item is found and this process is repeated for all other elements.

**MAX-HEAPIFY ( $A, i$ ) :**

1.  $i \leftarrow \text{left}[i]$
2.  $r \leftarrow \text{right}[i]$
3. if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$
4. then  $\text{largest} \leftarrow l$
5. else  $\text{largest} \leftarrow i$
6. if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[\text{largest}]$
7. then  $\text{largest} \leftarrow r$
8. if  $\text{largest} \neq i$
9. then exchange  $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY [ $A, \text{largest}$ ]

**HEAP-SORT(A) :**

1. BUILD-MAX-HEAP (A)
2. for  $i \leftarrow \text{length}[A]$  down to 2
3. do exchange  $A[1] \leftrightarrow A[i]$
4. heap-size [A]  $\leftarrow \text{heap-size}[A] - 1$
5. MAX-HEAPIFY (A, 1)

**BUILD-MAX-HEAP (A)**

1. heap-size (A)  $\leftarrow \text{length}[A]$
2. for  $i \leftarrow (\text{length}[A]/2)$  down to 1 do
3. MAX-HEAPIFY (A, i)

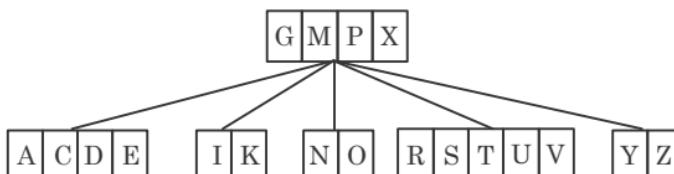
We can build a heap from an unordered array in linear time.

**Average case and worst case complexity :**

1. We have seen that the running time of BUILD-HEAP is  $O(n)$ .
2. The heap sort algorithm makes a call to BUILD-HEAP for creating a (max) heap, which will take  $O(n)$  time and each of the  $(n - 1)$  calls to MAX-HEAPIFY to fix up the new heap (which is created after exchanging the root and by decreasing the heap size).
3. We know 'MAX-HEAPIFY' takes time  $O(\log n)$ .
4. Thus the total running time for the heap sort is  $O(n \log n)$ .

**Numerical :** Since the given problem is already in sorted form. So, there is no need to apply any procedure on given problem.

5. Explain B-tree and insert elements B, Q, L, F into B-tree Fig. 1 then apply deletion of elements F, M, G, D, B on resulting B-tree.



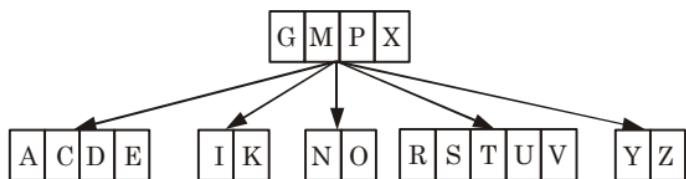
**Fig. 1.**

**Ans. B-tree :**

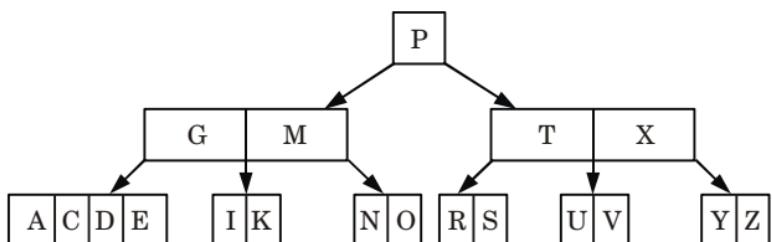
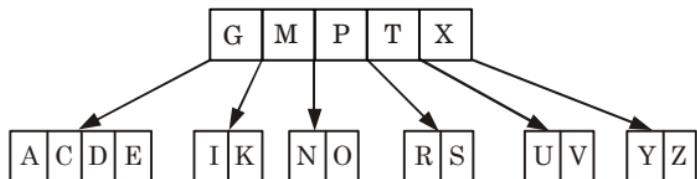
A B-tree of order  $m$  is an  $m$ -ary search tree with the following properties :

1. The root is either leaf or has atleast two children.
2. Each node, except for the root and the leaves, has between  $m/2$  and  $m$  children.
3. Each path from the root to a leaf has the same length.
4. The root, each internal node and each leaf is typically a disk block.
5. Each internal node has upto  $(m - 1)$  key values and upto  $m$  children.

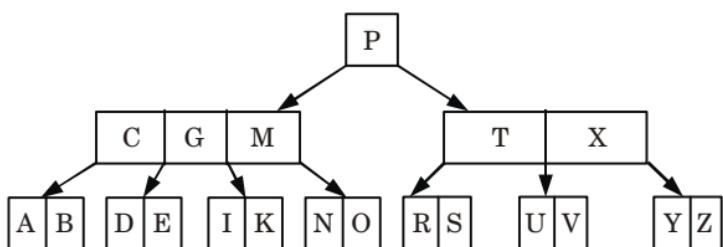
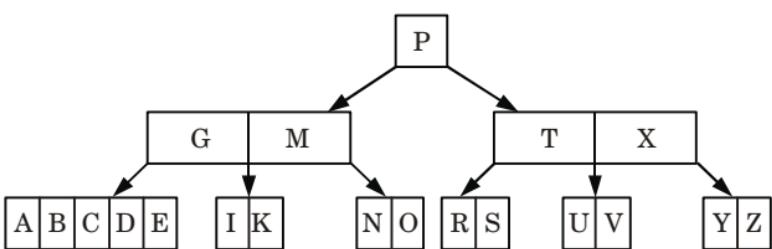
**Numerical :**  
**Insertion :**

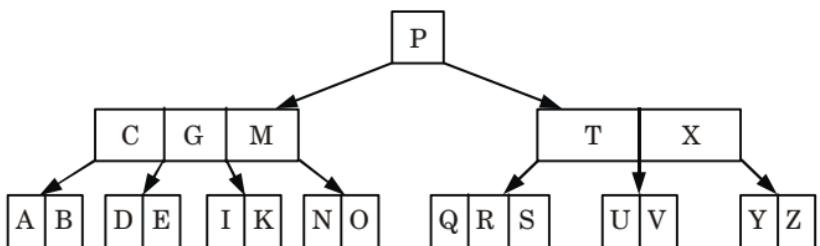
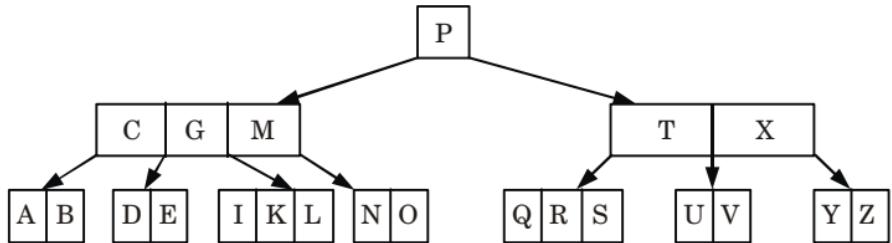
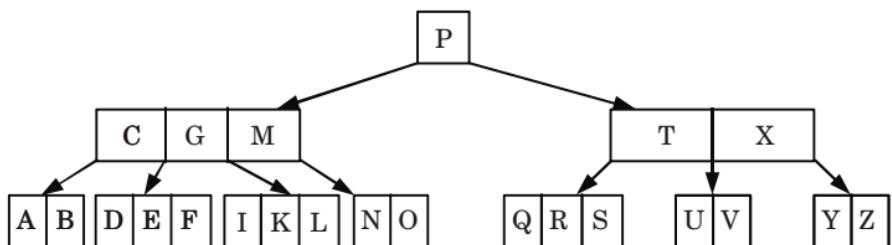
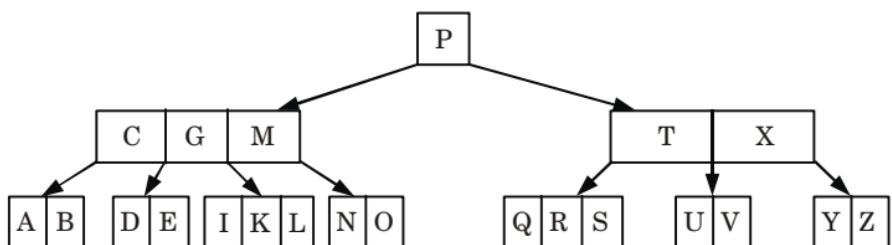
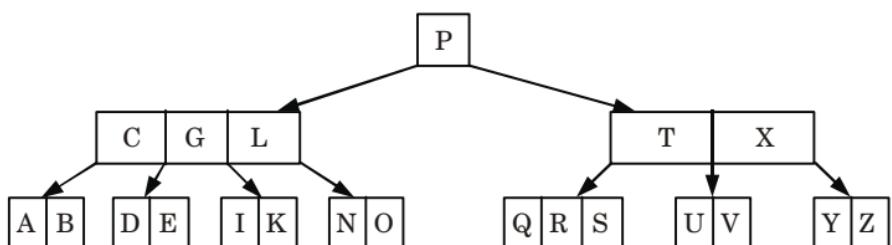


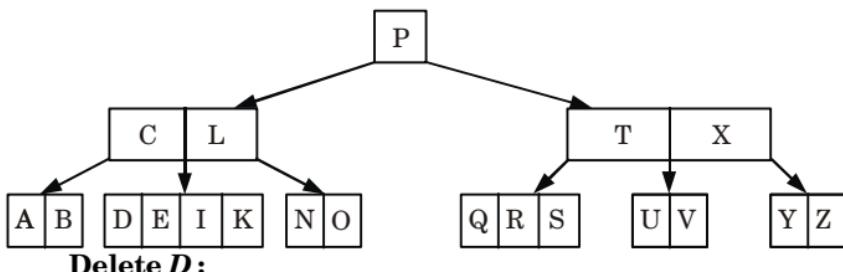
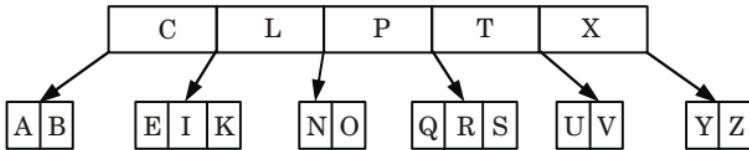
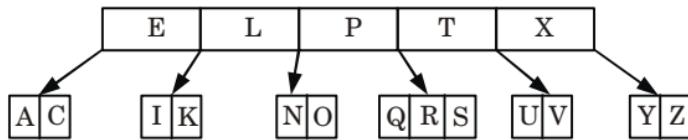
Assuming, order of B-tree = 5



**Insert B :**



**Insert Q :****Insert L :****Insert F :****Deletion :****Delete F :****Delete M :**

**Delete G :****Delete D :****Delete B :**

- 6. Write an algorithm for solving  $N$ -Queens problem. Show the solution of 4-Queens problem using backtracking approach.**

**Ans.**

1. In  $N$ -Queens problem, the idea is to place queens one by one in different columns, starting from the leftmost column.
2. When we place a queen in a column, we check for clashes with already placed queens.
3. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.
4. If we do not find such a row due to clashes then we backtrack and return false.

**Algorithm for  $N$ -Queens problem :**

$N$ -Queens are to be placed on an  $n \times n$  chessboard so that no two attack i.e., no two Queens are on the same row, column or diagonal.

**PLACE ( $k, i$ )**

1. for  $j \leftarrow 1$  to  $k - 1$
2. do if  $(x(j) = i)$  or  $\text{Abs}(x[j] - i) = (\text{Abs}(j - k))$
3. then return false
4. return true

Place  $(k, i)$  returns true if a queen can be placed in the  $k^{\text{th}}$  row and  $i^{\text{th}}$  column otherwise return false.

$x[ ]$  is a global array whose first  $k - 1$  values have been set.  $\text{Abs}(r)$  returns the absolute value of  $r$ .

***N*-Queens ( $k, n$ )**

1. for  $i \leftarrow 1$  to  $n$
2. do if PLACE ( $k, i$ )
3.     then  $x[k] \leftarrow i$
4. if  $k = n$ , then print  $x[1 \dots N]$
5. else  $N$ -Queens ( $k + 1, n$ )

[Note : For 8-Queen problem put  $n = 8$  in the algorithm.]

**4-Queens problem :**

1. Suppose we have  $4 \times 4$  chessboard with 4-queens each to be placed in non-attacking position.

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   |   |   |   |
| 2 |   |   |   |   |
| 3 |   |   |   |   |
| 4 |   |   |   |   |

**Fig. 2.**

2. Now, we will place each queen on a different row such that no two queens attack each other.
3. We place the queen  $q_1$  in the very first accept position (1, 1).
4. Now if we place queen  $q_2$  in column 1 and 2 then the dead end is encountered.
5. Thus, the first acceptable position for queen  $q_2$  is column 3 i.e., (2, 3) but then no position is left for placing queen  $q_3$  safely. So, we backtrack one step and place the queen  $q_2$  in (2, 4).
6. Now, we obtain the position for placing queen  $q_3$  which is (3, 2). But later this position lead to dead end and no place is found where queen  $q_2$  can be placed safely.

|   | 1     | 2     | 3 | 4     |
|---|-------|-------|---|-------|
| 1 | $q_1$ |       |   |       |
| 2 |       |       |   | $q_2$ |
| 3 |       | $q_3$ |   |       |
| 4 |       |       |   |       |

**Fig. 3.**

7. Then we have to backtrack till queen  $q_1$  and place it to (1, 2) and then all the other queens are placed safely by moving queen  $q_2$  to (2, 4), queen  $q_3$  to (3, 1) and queen  $q_4$  to (4, 3) i.e., we get the solution < 2, 4, 1, 3>. This is one possible solution for 4-queens problem.

|   | 1     | 2     | 3     | 4     |
|---|-------|-------|-------|-------|
| 1 |       | $q_1$ |       |       |
| 2 |       |       |       | $q_2$ |
| 3 | $q_3$ |       |       |       |
| 4 |       |       | $q_4$ |       |

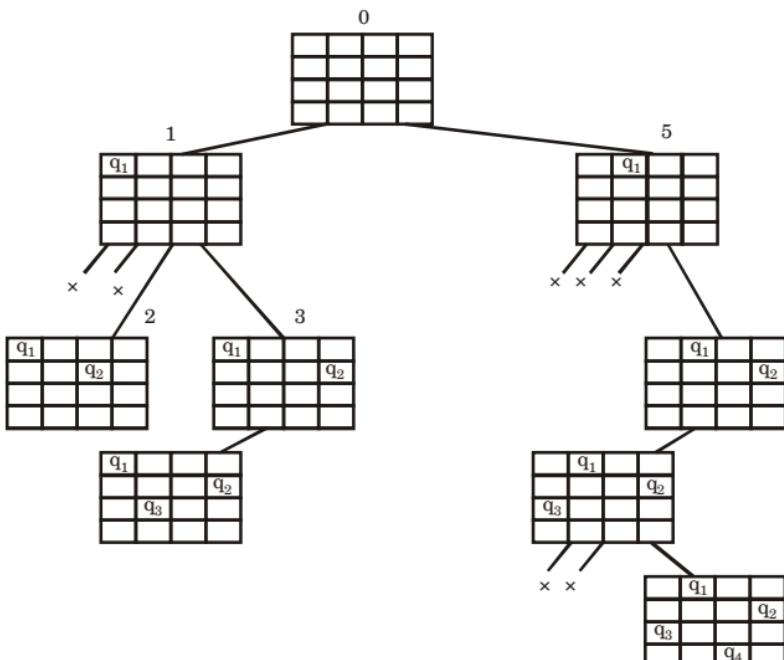
**Fig. 4.**

8. For other possible solution the whole method is repeated for all partial solutions. The other solution for 4-queens problem is  $<3, 1, 4, 2>$  i.e.,

|   | 1     | 2     | 3 | 4     |
|---|-------|-------|---|-------|
| 1 |       | $q_1$ |   |       |
| 2 | $q_2$ |       |   |       |
| 3 |       |       |   | $q_3$ |
| 4 |       | $q_4$ |   |       |

**Fig. 5.**

9. Now, the implicit tree for 4-queen for solution  $<2, 4, 1, 3>$  is as follows :
10. Fig. 6 shows the complete state space for 4-queens problem. But we can use backtracking method to generate the necessary node and stop if next node violates the rule i.e., if two queens are attacking.

**Fig. 6.**

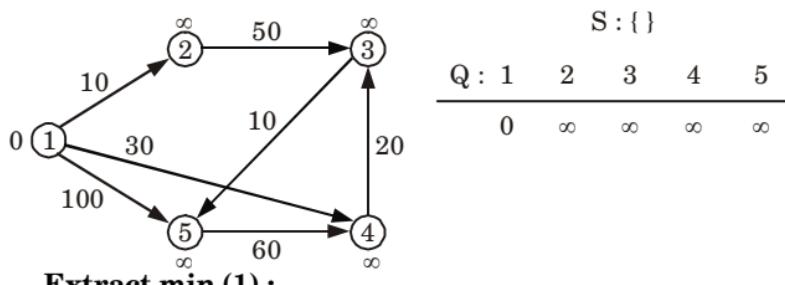
**7. Explain greedy single source shortest path algorithm with example.**

**Ans.**

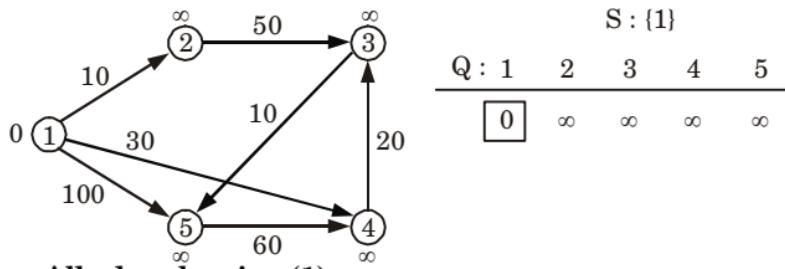
- Dijkstra's algorithm, is a greedy algorithm that solves the single source shortest path problem for a directed graph  $G = (V, E)$  with non-negative edge weights, i.e., we assume that  $w(u, v) \geq 0$  each edge  $(u, v) \in E$ .
- Dijkstra's algorithm maintains a set  $S$  of vertices whose final shortest path weights from the source  $s$  have already been determined.
- That is, for all vertices  $v \in S$ , we have  $d[v] = \delta(s, v)$ .
- The algorithm repeatedly selects the vertex  $u \in V - S$  with the minimum shortest path estimate, inserts  $u$  into  $S$ , and relaxes all edges leaving  $u$ .
- We maintain a priority queue  $Q$  that contains all the vertices in  $V - S$ , keyed by their  $d$  values.
- Graph  $G$  is represented by adjacency list.
- Dijkstra's always chooses the "lightest or "closest" vertex in  $V - S$  to insert into set  $S$  that it uses as a greedy strategy.

**For example :**

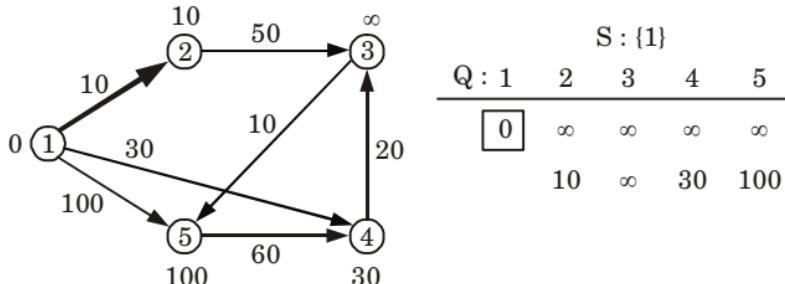
**Initialize :**

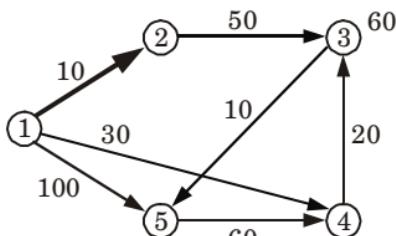


**Extract min (1) :**

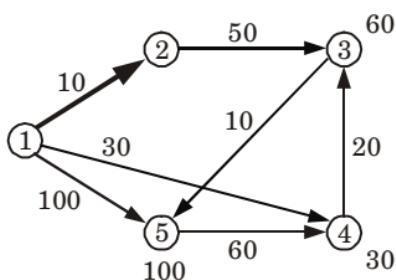


**All edges leaving (1) :**

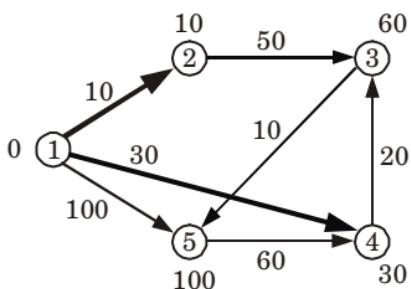


**Extract min(2) :**

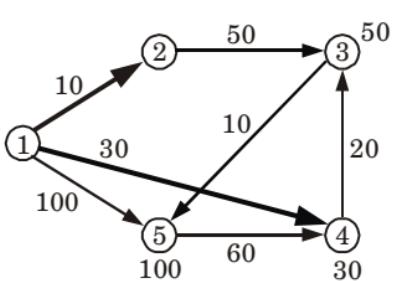
| S : {1, 2}    |          |          |          |          |
|---------------|----------|----------|----------|----------|
| Q : 1 2 3 4 5 |          |          |          |          |
| 0             | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10            | 60       | 30       | 100      |          |

**All edges leaving (2) :**

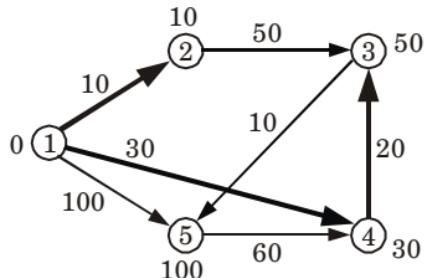
| S : {1, 2}    |          |          |          |          |
|---------------|----------|----------|----------|----------|
| Q : 1 2 3 4 5 |          |          |          |          |
| 0             | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10            | 60       | 30       | 100      |          |

**Extract min(4) :**

| S : {1, 2, 4} |          |          |          |          |
|---------------|----------|----------|----------|----------|
| Q : 1 2 3 4 5 |          |          |          |          |
| 0             | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10            | 60       | 30       | 100      |          |

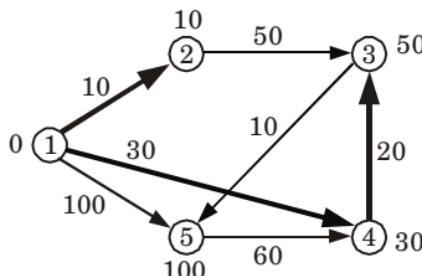
**All edges leaving (4) :**

| S : {1, 2, 4} |          |          |          |          |
|---------------|----------|----------|----------|----------|
| Q : 1 2 3 4 5 |          |          |          |          |
| 0             | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10            | 60       | 30       | 100      |          |

**Extract min(3) :**

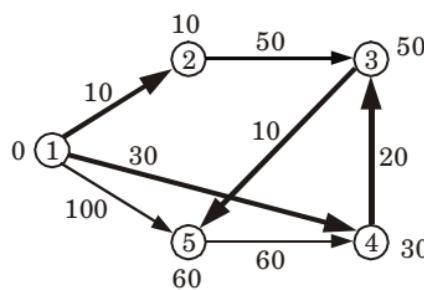
| S : {1, 2, 4, 3} |          |          |          |          |
|------------------|----------|----------|----------|----------|
| Q : 1 2 3 4 5    |          |          |          |          |
| 0                | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10               | $\infty$ | 30       | 100      |          |

**All edges leaving (3) :**



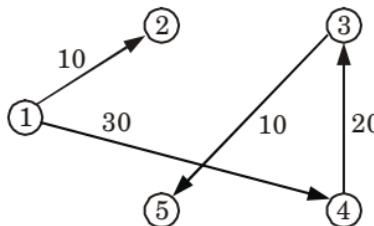
|  |    | S : {1, 2, 4, 3, 5} |    |    |     |   |
|--|----|---------------------|----|----|-----|---|
|  |    | Q : 1               | 2  | 3  | 4   | 5 |
|  | 0  |                     | ∞  | ∞  | ∞   | ∞ |
|  | 10 |                     | ∞  | 30 | 100 |   |
|  | 60 |                     | 30 |    | 100 |   |
|  | 50 |                     |    |    | 60  |   |

**Extract min(5) :**



|  |    | S : {1, 2, 4, 3, 5} |    |    |     |    |
|--|----|---------------------|----|----|-----|----|
|  |    | Q : 1               | 2  | 3  | 4   | 5  |
|  | 0  |                     | ∞  | ∞  | ∞   | ∞  |
|  | 10 |                     | ∞  | 30 | 100 |    |
|  | 60 |                     | 30 |    | 100 |    |
|  | 50 |                     |    |    | 60  |    |
|  |    |                     |    |    |     | 60 |

**Shortest path**



8. What is string matching algorithm ? Explain Rabin-Karp method with examples.

**Ans.** **String matching algorithm :**

String matching is a process of finding one or more occurrences of a pattern in a text.

**String matching problem :**

Given a text array  $T[1 \dots n]$  of  $n$  character and a pattern array  $P[1 \dots m]$  of  $m$  characters.

The problem is to find an integer  $s$ , called valid shift where  $0 \leq s < n - m$  and  $T[s + 1 \dots s + m] = P[1 \dots m]$ .

We further assume that the elements of  $P$  and  $T$  are characters drawn from a finite alphabet  $\Sigma$  such as  $\{0, 1\}$  or  $\{A, B, \dots, Z, a, b, \dots, z\}$ .

**The Rabin-Karp algorithm :**

The Rabin-Karp algorithm states that if two strings are equal, their hash values are also equal. This also uses elementary number-

theoretic notions such as the equivalence of two numbers module a third.

### Rabin-Karp-Matcher ( $T, P, d, q$ )

1.  $n \leftarrow \text{length}[T]$
2.  $m \leftarrow \text{length}[P]$
3.  $h \leftarrow d^{m-1} \bmod q$
4.  $p \leftarrow 0$
5.  $t_0 \leftarrow 0$
6. for  $i \leftarrow 1$  to  $m$
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  $n-m$
10. do if  $p = t_s$
11. then if  $p[1.....m] = T[s+1.....s+m]$
12. then “pattern occurs with shift”  $s$
13. if  $s < n-m$
14. then  $t_{s+1} \leftarrow (d(t_s - T[s+1] h) + T[s+m+1]) \bmod q$

**Example of Rabin-Karp method :** Working modulo  $q = 11$ , how many spurious hits does the Rabin-Karp matcher encounter in the text  $T = 3141592653589793$  when looking for the pattern  $p = 26$

Given,  $p = 26$  and  $q = 11$

Now we divide 26 by 11 i.e.,

Remainder is 4 and  $m = 2$ .

We know  $m$  denotes the length of  $p$ .

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 7 | 9 | 3 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Now we divide 31 by 11, and get remainder is 9.

Similarly, 14 by 11 and get remainder is 3.

So, continue this step till last i.e., 93 is divided by 11 and get remainder is 5. After that we will store all remainder in a table.

|   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|--|
| 9 | 3 | 8 | 4 | 4 | 4 | 4 | 10 | 9 | 2 | 3 | 1 | 9 | 2 | 5 |  |
|   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |  |

Now we find valid matching.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 | 7 | 9 | 3 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Valid matching

|   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|--|
| 9 | 3 | 8 | 4 | 4 | 4 | 4 | 10 | 9 | 2 | 3 | 1 | 9 | 2 | 5 |  |
|   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |  |

Spurious  
hit

The number of spurious hits is 3.

### 9. Explain approximation algorithms with suitable examples.

**Ans.** **Approximation algorithm :**

1. An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.

2. The best of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.
  3. Let  $c(i)$  be the cost of solution produced by approximate algorithm and  $c^*(i)$  be the cost of optimal solution for some optimization problem instance  $i$ .
  4. For minimization and maximization problem, we are interested in finding a solution of a given instance  $i$  in the set of feasible solutions, such that  $c(i)/c^*(i)$  and  $c^*(i)/c(i)$  be as small as possible respectively.
  5. We say that an approximation algorithm for the given problem instance  $i$ , has a ratio bound of  $p(n)$  if for any input of size  $n$ , the cost  $c$  of the solution produced by the approximation algorithm is within a factor of  $p(n)$  of the cost  $c^*$  of an optimal solution. That is  $\max(c(i) / c^*(i), c^*(i) / c(i)) \leq p(n)$   
The definition applies for both minimization and maximization problems.
  6.  $p(n)$  is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have  $p(n) = 1$ .
  7. For a minimization problem,  $0 < c^*(i) < c(i)$ , and the ratio  $c(i) / c^*(i)$  gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.
  8. Similarly, for a maximization problem,  $0 < c(i) \leq c^*(i)$ , and the ratio  $c^*(i) / c(i)$  gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.
- Travelling salesman problem (TSP) is an example of approximation algorithm.

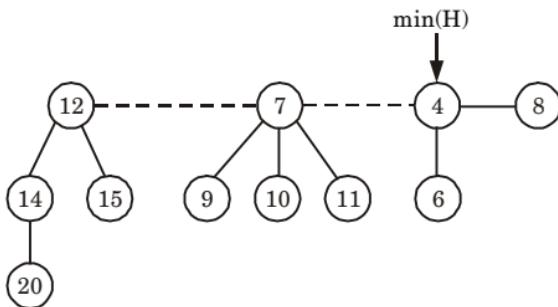
### SECTION - C

**Note :** Attempt any **two** questions from this section : **(15 × 2 = 30)**

10. **What is Fibonacci heap ? Explain CONSOLIDATE operation with suitable example for Fibonacci heap.**

**Ans. Fibonacci heap :**

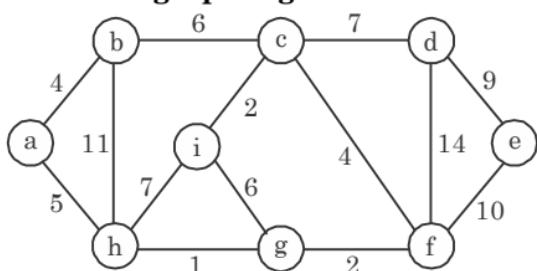
1. A Fibonacci heap is a set of min-heap-ordered trees.
2. Trees are not ordered binomial trees, because
  - a. Children of a node are unordered.
  - b. Deleting nodes may destroy binomial construction.
3. Fibonacci heap  $H$  is accessed by a pointer  $\min[H]$  to the root of a tree containing a minimum key. This node is called the minimum node.
4. If Fibonacci heap  $H$  is empty, then  $\min[H] = \text{NIL}$ .

**Fig. 7.**

**CONSOLIDATE operation :**  
**CONSOLIDATE( $H$ )**

1. for  $i \leftarrow 0$  to  $D(n[H])$
  2. do  $A[i] \leftarrow \text{NIL}$
  3. for each node  $w$  in the root list of  $H$
  4. do  $x \leftarrow w$
  5.  $d \leftarrow \text{degree}[x]$
  6. while  $A[d] \neq \text{NIL}$
  7. do  $y \leftarrow A[d]$   $\triangleright$  Another node with the same degree as  $x$ .
  8. if  $\text{key}[x] > \text{key}[y]$
  9. then exchange  $x \leftrightarrow y$
  10. FIB-HEAP-LINK( $H, y, x$ )
  11.  $A[d] \leftarrow \text{NIL}$
  12.  $d \leftarrow d + 1$
  13.  $A[d] \leftarrow x$
  14.  $\text{min}[H] \leftarrow \text{NIL}$
  15. for  $i \leftarrow 0$  to  $D(n[H])$
  16. do if  $A[i] \neq \text{NIL}$
  17. then add  $A[i]$  to the root list of  $H$
  18. if  $\text{min}[H] = \text{NIL}$  or  $\text{key}[A[i]] < \text{key}[\text{min}[H]]$
  19. then  $\text{min}[H] \leftarrow A[i]$
- FIB-HEAP-LINK( $H, y, x$ )**
1. remove  $y$  from the root list of  $H$
  2. make  $y$  a child of  $x$ , incrementing  $\text{degree}[x]$
  3.  $\text{mark}[y] \leftarrow \text{FALSE}$

- 11. What is minimum spanning tree ? Explain Prim's algorithm and find MST of graph Fig. 8.**

**Fig. 8.**

**Ans. Minimum spanning tree :**

- Given a connected weighted graph  $G$ , it is often desired to create a spanning tree  $T$  for  $G$  such that the sum of the weights of the tree edges in  $T$  is as small as possible.
- Such a tree is called a minimum spanning tree and represents the “cheapest” way of connecting all the nodes in  $G$ .
- There are number of techniques for creating a minimum spanning tree for a weighted graph but the most famous methods are Prim’s and Kruskal’s algorithm.

**Prim's algorithm :**

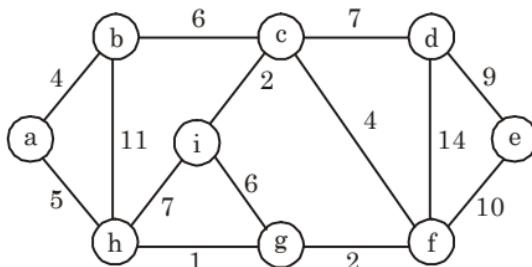
First it chooses a vertex and then chooses an edge with smallest weight incident on that vertex. The algorithm involves following steps :

**Step 1 :** Choose any vertex  $V_1$  of  $G$ .

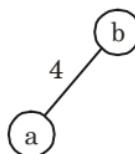
**Step 2 :** Choose an edge  $e_1 = V_1V_2$  of  $G$  such that  $V_2 \neq V_1$  and  $e_1$  has smallest weight among the edge  $e$  of  $G$  incident with  $V_1$ .

**Step 3 :** If edges  $e_1, e_2, \dots, e_i$  have been chosen involving end points  $V_1, V_2, \dots, V_{i+1}$ , choose an edge  $e_{i+1} = V_jV_k$  with  $V_j = \{V_1, \dots, V_{i+1}\}$  and  $V_k \notin \{V_1, \dots, V_{i+1}\}$  such that  $e_{i+1}$  has smallest weight among the edges of  $G$  with precisely one end in  $\{V_1, \dots, V_{i+1}\}$ .

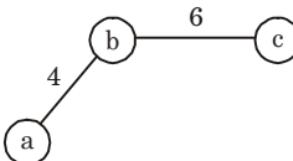
**Step 4 :** Stop after  $n - 1$  edges have been chosen. Otherwise goto step 3.

**Numerical :****Fig. 9.**

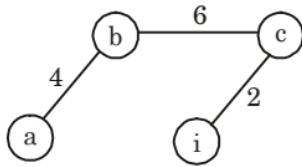
Let  $a$  be the source node. Select edge  $(a, b)$  as distance between edge  $(a, b)$  is minimum.



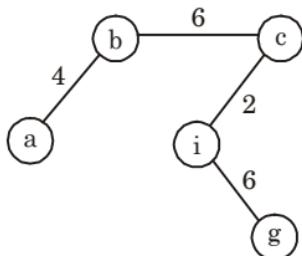
Now, select edge  $(b, c)$



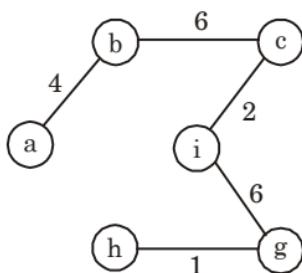
Now, select edge  $(c, i)$



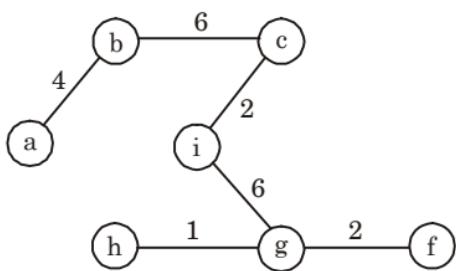
Now, select edge  $(i, g)$



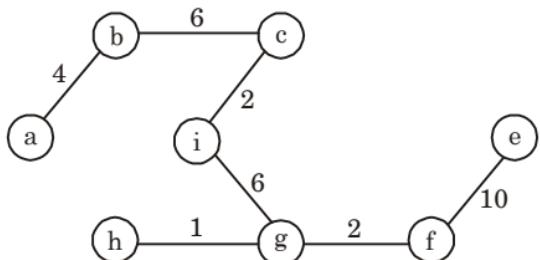
Now, select edge  $(g, h)$



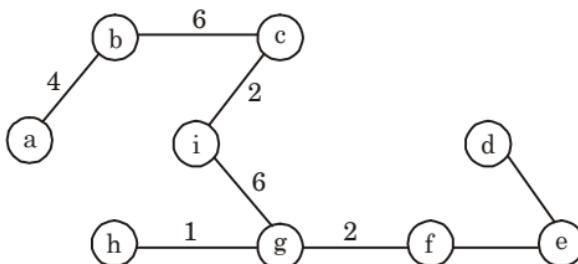
Now, select edge  $(g, f)$



Now, select edge  $(f, e)$



Now, select edge  $(e, d)$



Thus, we obtained MST for Fig. 9.

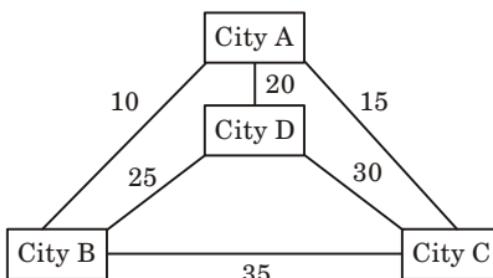
- 12. Explain TSP (Travelling Salesman) problem with example. Write an approach to solve TSP problem.**

**Ans. Travelling Salesman Problem (TSP) :**

Travelling salesman problem is the problem to find the shortest possible route for a given set of cities and distance between the pair of cities that visits every city exactly once and returns to the starting point.

**For example :**

The following graph shows a set of cities and distance between every pair of cities :



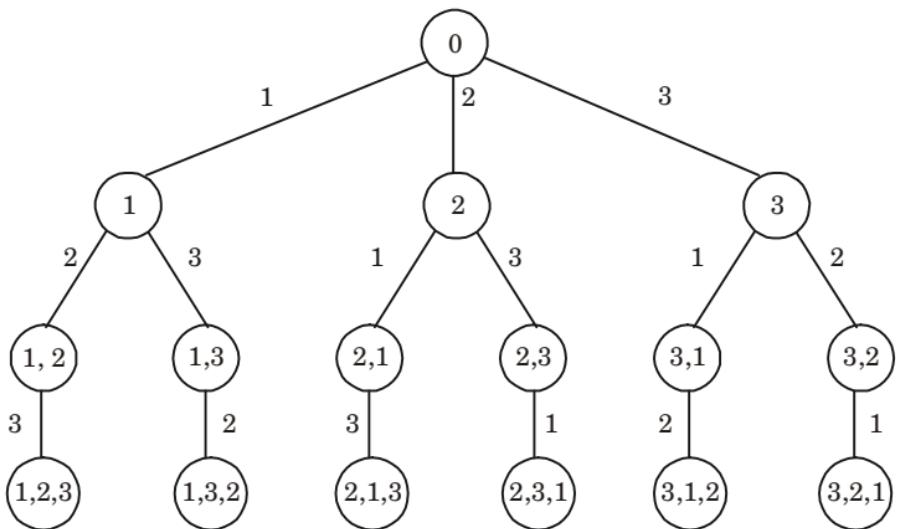
If salesman starting city is A. Then a TSP tour in the graph is  
 $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$

Cost of the tour =  $10 + 25 + 30 + 15 = 80$  Units.

Backtracking approach is used to solve TSP problem.

**Backtracking algorithm for the TSP :**

1. Let  $G$  be the given complete graph with positive weights on its edges.
2. Use a search tree that generates all permutations of  $V = \{1 \dots n\}$ , specifically the one illustrated in Fig. 10 for the case  $n = 3$ .



**Fig. 10.**

3. A node at depth  $i$  of this tree (the root is at depth 0) stores an  $i$ -permutation of  $\{1, \dots, n\}$ . A leaf stores a permutation of  $\{1, \dots, n\}$ , which is equivalent to saying that it stores a particular Hamiltonian cycle (tour) of  $G$ .
4. For the travelling salesman problem, we will not do any static pruning on this tree, we will do dynamic pruning, during the search.



**B. Tech.****(SEM. V) ODD SEMESTER THEORY  
EXAMINATION, 2016-17****DESIGN AND ANALYSIS OF ALGORITHMS**

---

**Time : 3 Hours****Max. Marks : 100**

---

**Section-A**

1. Attempt **all** parts. All parts carry equal marks. Write answer of each part in short. **( $2 \times 10 = 20$ )**
- a. List out the disadvantages of divide and conquer algorithm.
- b. What are the fundamental steps involved in algorithmic problem solving ?
- c. Write recursive function to find  $n^{\text{th}}$  Fibonacci number.
- d. Define binary heap.
- e. Briefly explain the Prim's algorithm.
- f. Define principle of optimality.
- g. Write the names of various design techniques of algorithm.
- h. Differentiate between branch and bound and backtracking technique.
- i. What is the running time complexity of 8-Queens problem ?
- j. Define P, NP and NP-complete in decision problem.

**Section-B****Note :** Attempt any **five** questions from this section.**( $10 \times 5 = 50$ )**

2. Explain the concepts of quick sort method and analyze its complexity with suitable example.
3. Explain the concept of merge sort with example.
4. Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion.

5. Write short note on Dijkstra's algorithm shortest paths problems.
  6. Write pseudocode for 8-Queens problem.
  7. Write non-deterministic algorithm for sorting.
  8. What is backtracking ? Write general iterative algorithm for backtracking.
  9. Differentiate NP-complete with NP-hard.

## **Section-C**

**Note :** Attempt any **two** questions from this section. **(15 × 2 = 30)**

- ### **10. i. State Bellman-Ford algorithm.**

- ii. Consider following instance for simple knapsack problem.  
Find the solution using greedy method.

N = 8

$$P = \{11, 21, 31, 33, 43, 53, 55, 65\}$$

$$W = \{1, 11, 21, 23, 33, 43, 45, 55\}$$

$$M=110$$

11. What is travelling salesman problem ? Find the solution of following travelling salesman problem using branch and bound method.

$$\text{Cost matrix} = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 6 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

- 12.** Prove that three colouring problem is NP-complete.



**SOLUTION OF PAPER (2016-17)****Section-A**

1. Attempt **all** parts. All parts carry equal marks. Write answer of each part in short.  $(2 \times 10 = 20)$

- a. **List out the disadvantages of divide and conquer algorithm.**

**Ans.** **Disadvantages of divide and conquer algorithm :**

- i. Recursion is slow.
- ii. Algorithm becomes complicated for large value of  $n$ .
  
- b. **What are the fundamental steps involved in algorithmic problem solving ?**

**Ans.** **Steps involved in algorithmic problem solving are :**

- i. Characterize the structure of optimal solution.
- ii. Recursively define the value of an optimal solution.
- iii. By using bottom-up technique, compute value of optimal solution.
- iv. Compute an optimal solution from computed information.

- c. **Write recursive function to find  $n^{\text{th}}$  Fibonacci number.**

**Ans.**

```
int fibo(int num)
{
 if (num == 0)
 {
 return 0;
 }
 else if (num == 1)
 {
 return 1;
 }
 else
 {
 return(fibo(num - 1) + fibo(num - 2));
 }
}
```

- d. **Define binary heap.**

**Ans.** The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The tree is completely filled on all levels except possibly lowest.

- e. **Briefly explain the Prim's algorithm.**

**Ans.** First it chooses a vertex and then chooses an edge with smallest weight incident on that vertex. The algorithm involves following steps :

**Step 1 :** Choose any vertex  $V_1$  of  $G$ .

**Step 2 :** Choose an edge  $e_1 = V_1 V_2$  of  $G$  such that  $V_2 \neq V_1$  and  $e_1$  has smallest weight among the edge  $e$  of  $G$  incident with  $V_1$ .

**Step 3 :** If edges  $e_1, e_2, \dots, e_i$  have been chosen involving end points  $V_1, V_2, \dots, V_{i+1}$ , choose an edge  $e_{i+1} = V_j V_k$  with  $V_j = \{V_1, \dots, V_{i+1}\}$  and  $V_k \notin \{V_1, \dots, V_{i+1}\}$  such that  $e_{i+1}$  has smallest weight among the edges of  $G$  with precisely one end in  $\{V_1, \dots, V_{i+1}\}$ .

**Step 4 :** Stop after  $n - 1$  edges have been chosen. Otherwise goto step 3.

**f. Define principle of optimality.**

**Ans.** Principle of optimality states that in an optimal sequence of decisions or choices, each subsequence must also be optimal.

**g. Write the names of various design techniques of algorithm.**

**Ans.** Various design techniques of algorithm are :

1. Divide and conquer
2. Greedy approach
3. Dynamic programming
4. Branch and bound
5. Backtracking algorithm

**h. Differentiate between branch and bound and backtracking technique.**

**Ans.**

| S. No. | Branch and bound                                                                                                                                           | Backtracking                                                  |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------|
| 1.     | In this method, it is not necessary to use depth first search for obtaining the solution, even the breadth first search, best first search can be applied. | Solution for backtracking is traced using depth first search. |
| 2.     | Typically optimization problems can be solved using branch and bound.                                                                                      | Typically decision problems can be solved using backtracking. |

**i. What is the running time complexity of 8-Queens problem ?**

**Ans.** The running time complexity of 8-Queens problem is  $O(P(n)n!)$  where  $P(n)$  is polynomial in  $n$ .

**j. Define P, NP and NP-complete in decision problem.**

**Ans.** **P-polynomial time :** These problems can be solved in polynomial time, which take time like  $O(n)$ ,  $O(n^2)$ ,  $O(n^3)$  like finding maximum element in an array or to check whether a string is palindrome or not are *P* problems.

**Non deterministic polynomial time :** These problem cannot be solved in polynomial time like TSP (Travelling Salesman Problem) or subset sum are NP problem.

But NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

**NP-complete :** The group of problems which are both in NP and NP-hard are known as NP-complete problem.

## Section-B

**Note :** Attempt any five questions from this section. **(10 × 5 = 50)**

**2. Explain the concepts of quick sort method and analyze its complexity with suitable example.**

**Ans.** **Quick sort :**

Quick sort works by partitioning a given array  $A[p \dots r]$  into two non-empty subarray  $A[p \dots q - 1]$  and  $A[q + 1 \dots r]$  such that every key in  $A[p \dots q - 1]$  is less than or equal to every key in  $A[q + 1 \dots r]$ . Then the two subarrays are sorted by recursive calls to quick sort.

**Quick\_Sort ( $A, p, r$ )**

1. If  $p < r$  then
2.  $q \leftarrow \text{Partition} (A, p, r)$
3. Recursive call to Quick\_Sort ( $A, p, q - 1$ )
4. Recursive call to Quick\_Sort ( $A, q + 1, r$ )

As a first step, Quick sort chooses as pivot one of the items in the array to be sorted. Then array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

**Partition ( $A, p, r$ )**

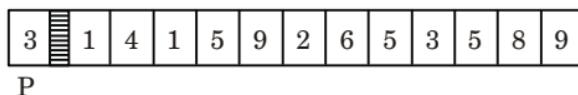
1.  $x \leftarrow A[r]$
2.  $i \leftarrow p - 1$
3. for  $j \leftarrow p$  to  $r - 1$
4. do if  $A[j] \leq x$
5. then  $i \leftarrow i + 1$
6. then exchange  $A[i] \leftrightarrow A[j]$
7. exchange  $A[i + 1] \leftrightarrow A[r]$
8. return  $i + 1$

**Example :** Given array to be sorted

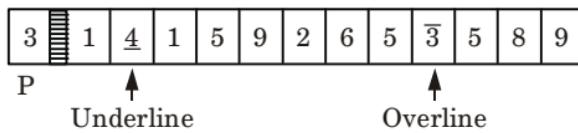
|   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Sort the array  $A$  using quick sort algorithm.

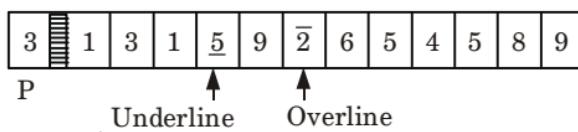
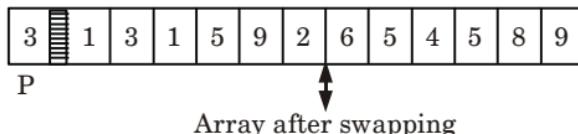
**Step 1 :** The array is Pivoted about its first element i.e., Pivot ( $P$ ) = 3



**Step 2 :** Find first element larger than pivot (make underline) and find element not larger than pivot from end make overline.



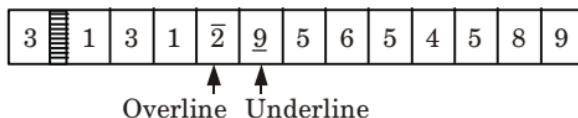
**Step 3 :** Swap these elements and scan again.



Apply swapping,



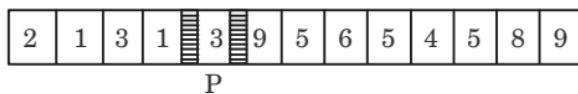
Again apply scanning,



The pointers have crossed

i.e., overline on left of underlined

Then, in this situation swap pivot with overline.



Now, pivoting process is complete.

**Step 4 :** Recursively sort subarrays on each side of pivot.

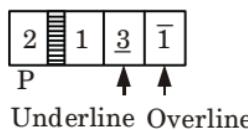
Subarray 1 : 

|   |   |   |   |
|---|---|---|---|
| 2 | 1 | 3 | 1 |
|---|---|---|---|

Subarray 2 : 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 9 | 5 | 6 | 5 | 1 | 5 | 8 | 9 |
|---|---|---|---|---|---|---|---|

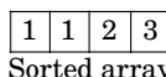
First apply Quick sort for subarray 1.



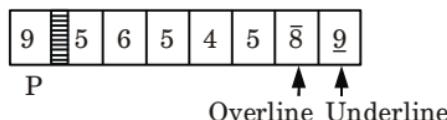
The pointers have crossed.

i.e., overline on left of underlined.

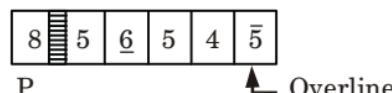
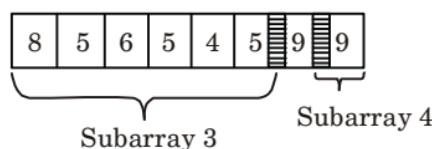
Swap pivot with overline



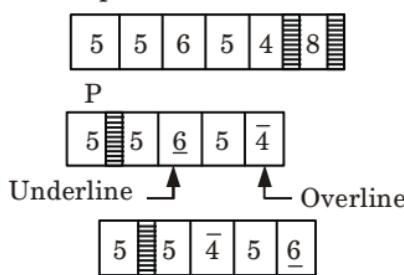
Now, for subarray 2 we apply Quick sort procedure.



The pointer has crossed. Then swap pivot with overline.

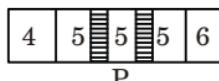


Swap overline with pivot.

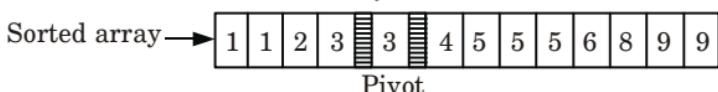


Overline on left of underlined.

Swap pivot with overline.



Now combine all the subarrays



### Analysis of complexity :

#### i. Worst case :

- Let  $T(n)$  be the worst case time for quick sort on input size  $n$ . We have a recurrence

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n) \quad \dots(1)$$

where  $q$  ranges from 0 to  $n - 1$ , since the partition produces two regions, each having size  $n - 1$ .

- Now we assume that  $T(n) \leq cn^2$  for some constant  $c$ .

Substituting our assumption in eq. (1.19.1) we get

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= \max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) + \Theta(n) \end{aligned}$$

- Since the second derivative of expression  $q^2 + (n - q - 1)^2$  with respect to  $q$  is positive. Therefore, expression achieves a maximum over the range  $0 \leq q \leq n - 1$  at one of the endpoints.

- This gives the bound

$$\max_{0 \leq q \leq n-1} (q^2 + (n - q - 1)^2) \leq (n - 1)^2 = n^2 - 2n + 1$$

- Continuing with the bounding of  $T(n)$  we get

$$T(n) \leq cn^2 - c(2n - 1) + \Theta(n) \leq cn^2$$

- Since we can pick the constant  $c$  large enough so that the  $c(2n - 1)$  term dominates the  $\Theta(n)$  term. We have

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

- Thus, the worst case running time of quick sort is  $\Theta(n^2)$ .

#### ii. Average case :

- If the split induced of RANDOMIZED\_PARTITION puts constant fraction of elements on one side of the partition, then the recurrence tree has depth  $\Theta(\log n)$  and  $\Theta(n)$  work is performed at each level.
- This is an intuitive argument why the average case running time of RANDOMIZED\_QUICKSORT is  $\Theta(n \log n)$ .
- Let  $T(n)$  denotes the average time required to sort an array of  $n$  elements. A call to RANDOMIZED\_QUICKSORT with a 1 element array takes a constant time, so we have  $T(1) = \Theta(1)$ .
- After the split RANDOMIZED\_QUICKSORT calls itself to sort two subarrays.
- The average time to sort an array  $A[1 .. q]$  is  $T[q]$  and the average time to sort an array  $A[q+1 .. n]$  is  $T[n - q]$ . We have

$$T(n) = 1/n (T(1) + T(n - 1) + \sum_{q=1}^{n-1} S_{q=1}^{n-1} (T(q) + T(n - q))) + \Theta(n) \quad \dots(1)$$

We know from worst-case analysis

$$T(1) = \Theta(1) \text{ and } T(n - 1) = O(n^2)$$

$$\begin{aligned} T(n) &= 1/n (\Theta(1) + O(n^2)) + 1/n \sum_{q=1}^{n-1} \Theta_q (r(q) + T(n - q)) + Q(n) \end{aligned} \quad \dots(2)$$

$$= 1/n \sum_{q=1}^{n-1} (T(q) + T(n - q)) + Q(n) \quad \dots(2)$$

$$= 1/n [2 \sum_{k=1}^{n-1} (T(k))] + \Theta(n) \quad \dots(2)$$

$$= 2/n \sum_{k=1}^{n-1} (T(k)) + \Theta(n) \quad \dots(3)$$

6. Solve the above recurrence using substitution method. Assume that  $T(n) \leq an n \log n + b$  for some constants  $a > 0$  and  $b > 0$ . If we can pick ‘ $a$ ’ and ‘ $b$ ’ large enough so that  $n \log n + b > T(1)$ . Then for  $n > 1$ , we have

$$\begin{aligned} T(n) &\geq {}^{n-1}\Theta_{k=1} 2/n (ak \log k + b) + \Theta(n) \\ &= 2a/n {}^{n-1}S_{k=1} k \log k - 1/8(n^2) + 2b/n \\ &\quad (n-1) + \Theta(n) \end{aligned} \quad \dots(4)$$

At this point we are claiming that

$${}^{n-1}\Theta_{k=1} k \log k \leq 1/2 n^2 \log n - 1/8(n^2)$$

Substituting this claim in the eq. (4), we get

$$\begin{aligned} T(n) &\leq 2a/n [1/2n^2 \log n - 1/8(n^2)] + 2/n b(n-1) + \Theta(n) \\ &\leq an \log n - an/4 + 2b + \Theta(n) \end{aligned} \quad \dots(5)$$

In the eq. (5),  $\Theta(n) + b$  and  $an/4$  are polynomials and we can choose ‘ $a$ ’ large enough so that  $an/4$  dominates  $\Theta(n) + b$ .

We conclude that QUICKSORT’s average running time is  $\Theta(n \log n)$ .

### 3. Explain the concept of merge sort with example.

**Ans.**

1. Merge sort is a sorting algorithm that uses the idea of divide and conquer.
2. This algorithm divides the array into two halves, sorts them separately and then merges them.
3. This procedure is recursive, with the base criteria that the number of elements in the array is not more than 1.

**Algorithm :**

**MERGE\_SORT ( $a, p, r$ )**

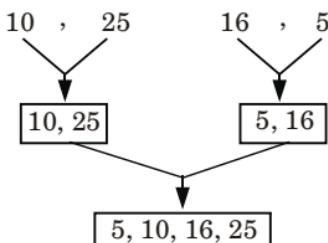
1. if  $p < r$
  2. then  $q \leftarrow \lfloor (p+r)/2 \rfloor$
  3. MERGE-SORT ( $A, p, q$ )
  4. MERGE-SORT ( $A, q+1, r$ )
  5. MERGE ( $A, p, q, r$ )
- MERGE ( $A, p, q, r$ )**
1.  $n_1 = q - p + 1$
  2.  $n_2 = r - q$
  3. Create arrays  $L$  [ $1 \dots n_1 + 1$ ] and  $R$  [ $1 \dots n_2 + 1$ ]
  4. for  $i = 1$  to  $n_1$ 
    - do  $L[i] = A[p+i-1]$
  5. endfor
  5. for  $j = 1$  to  $n_2$ 
    - do  $R[j] = A[q+j]$
  5. endfor
  6.  $L[n_1 + 1] = \infty, R[n_2 + 1] = \infty$

7.  $i = 1, j = 1$   
 8. for  $k = p$  to  $r$   
 do  
 if  $L[i] \leq R[j]$   
 then  $A[k] \leftarrow L[i]$   
 $i = i + 1$   
 else  $A[k] = R[j]$   
 $j = j + 1$   
 endif  
 endfor  
 9. exit

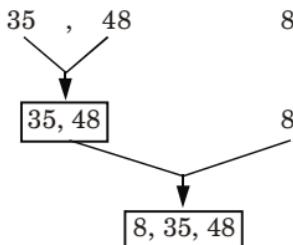
**Example :**

10, 25, 16, 5, 35, 48, 8

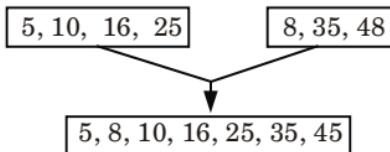
- Divide into two halves : 10, 25, 16, 5   35, 48, 8
- Consider the first part : 10, 25, 16, 5 again divide into two sub-arrays



- Consider the second half : 35, 48, 5 again divide into two sub-arrays



- Merge these two sorted sub-arrays,

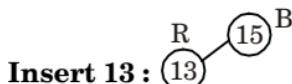


This is the sorted array.

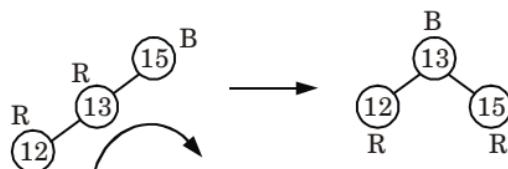
- Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion.

**Ans. Insertion :**

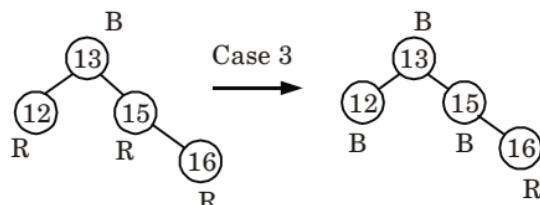
Insert 15 : 15<sup>B</sup>



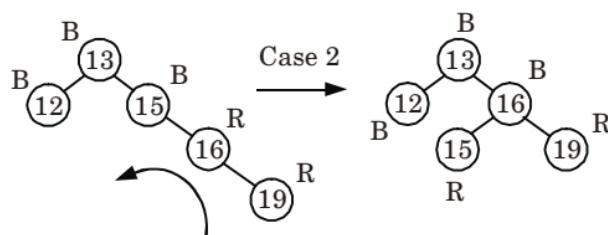
**Insert 12 :**



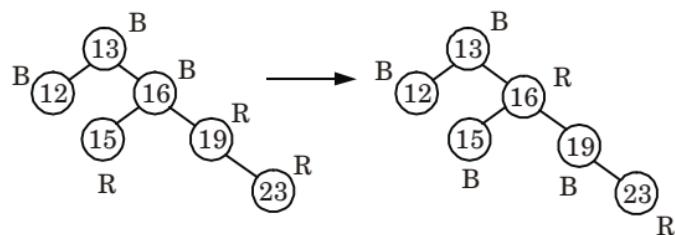
**Insert 16 :**



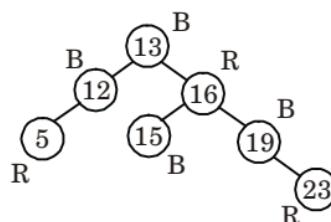
**Insert 19 :**

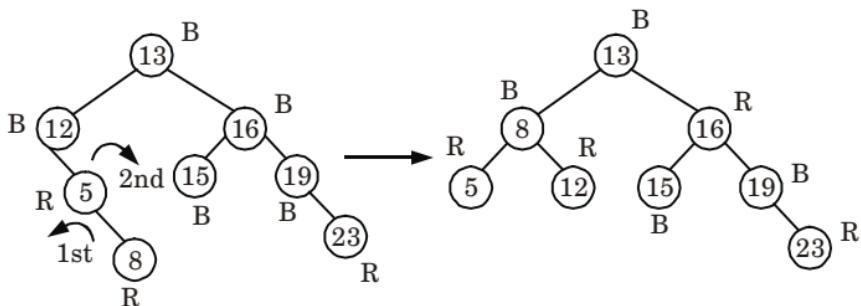
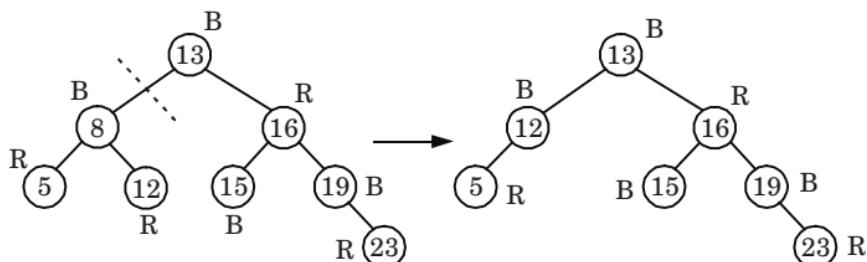
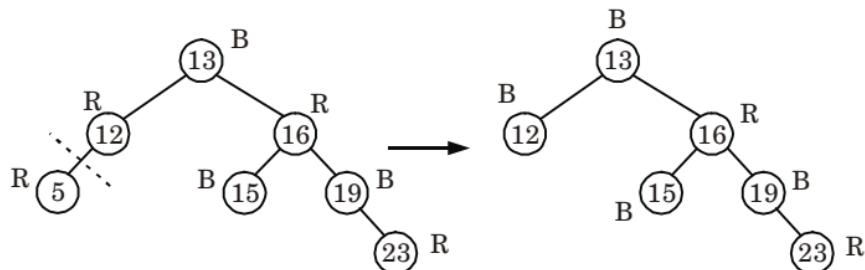
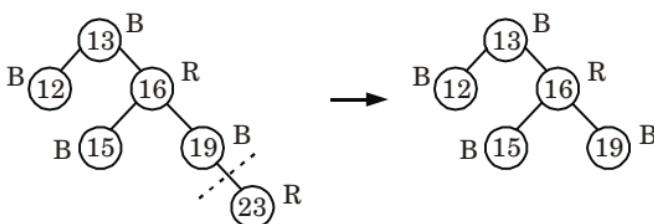
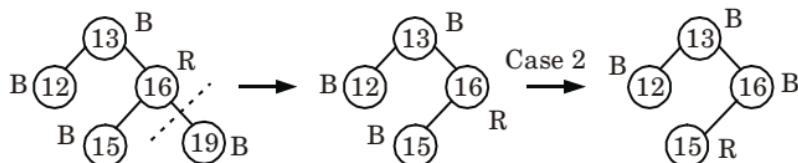


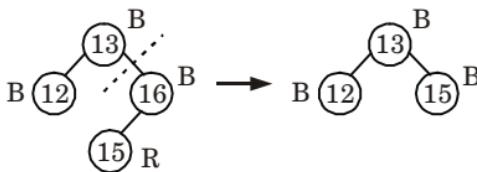
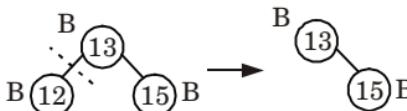
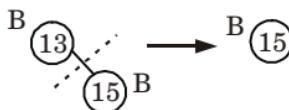
**Insert 23 :**



**Insert 5 :**



**Insert 8 :****Deletions :****Delete 8 :****Delete 5 :****Delete 23 :****Delete 19 :**

**Delete 16 :****Delete 12 :****Delete 13 :****Delete 15 :**

No tree

**5. Write short note on Dijkstra's algorithm shortest paths problems.**

**Ans.**

1. Dijkstra's algorithm, is a greedy algorithm that solves the single source shortest path problem for a directed graph  $G = (V, E)$  with non-negative edge weights, i.e., we assume that  $w(u, v) \geq 0$  each edge  $(u, v) \in E$ .
2. Dijkstra's algorithm maintains a set  $S$  of vertices whose final shortest path weights from the source  $s$  have already been determined.
3. That is, for all vertices  $v \in S$ , we have  $d[v] = \delta(s, v)$ .
4. The algorithm repeatedly selects the vertex  $u \in V - S$  with the minimum shortest path estimate, inserts  $u$  into  $S$ , and relaxes all edges leaving  $u$ .
5. We maintain a priority queue  $Q$  that contains all the vertices in  $V - s$ , keyed by their  $d$  values.
6. Graph  $G$  is represented by adjacency list.
7. Dijkstra's always chooses the "lightest or "closest" vertex in  $V - S$  to insert into set  $S$  that it uses as a greedy strategy.

**Dijkstra's algorithm :**

**DIJKSTRA ( $G, w, s$ )**

1. INITIALIZE-SINGLE-SOURCE ( $G, s$ )
2.  $s \leftarrow \emptyset$
3.  $Q \leftarrow V[G]$

4. while  $Q \neq \emptyset$
5. do  $u \leftarrow \text{EXTRACT-MIN } (Q)$
6.      $S \leftarrow S \cup \{u\}$
7.                 for each vertex  $v \in \text{Adj } [u]$
8.                     do  $\text{RELAX } (u, v, w)$

**RELAX ( $u, v, w$ ) :**

1. If  $d[u] + w(u, v) < d[v]$
2. then  $d[v] \leftarrow d[u] + w(u, v)$
3.              $\pi[v] \leftarrow u$

**6. Write pseudocode for 8-Queens problem.****Ans. Pseudocode for N-Queens problem :**

$N$ -Queens are to be placed on an  $n \times n$  chessboard so that no two attack i.e., no two Queens are on the same row, column or diagonal.

**PLACE ( $k, i$ )**

1. for  $j \leftarrow 1$  to  $k - 1$
2. do if  $(x(j) = i)$  or  $\text{Abs } (x[j] - i) = (\text{Abs } (j - k))$
3. then return false
4. return true

Place  $(k, i)$  returns true if a queen can be placed in the  $k^{\text{th}}$  row and  $i^{\text{th}}$  column otherwise return false.

$x[ ]$  is a global array whose first  $k - 1$  values have been set.  $\text{Abs}(r)$  returns the absolute value of  $r$ .

 **$N$ -Queens ( $k, n$ )**

1. for  $i \leftarrow 1$  to  $n$
2. do if PLACE  $(k, i)$
3.     then  $x[k] \leftarrow i$
4. if  $k = n$ , then print  $x[1 \dots N]$
5. else  $N$ -Queens  $(k + 1, n)$

[Note : For 8-Queen problem put  $n = 8$  in the algorithm.]

**7. Write non-deterministic algorithm for sorting.**

**Ans.** Non-deterministic algorithms are algorithm that, even for the same input, can exhibit different behaviours on different runs, iterations and executions.

 **$N$  SORT( $A, B$ ) :**

1. for  $i = 1$  to  $n$  do
2.  $j = \text{choice}(1 \dots n)$
3. if  $B[j] \neq 0$  then failure
4.  $B[j] = A[i]$
5. endfor

6. for  $i = 1$  to  $n - 1$  do
7. if  $B[i] < B[i + 1]$  then failure
8. endfor
9. print( $B$ )
10. success

**8. What is backtracking ? Write general iterative algorithm for backtracking.****Ans.**

1. Backtracking is a general algorithm for finding all solutions to some computational problems.
2. Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, and many other puzzles.
3. It is often the most convenient (if not the most efficient) technique for parsing, for the knapsack problem and other combinatorial optimization problem.
4. It can be applied only for problems which admit the concept of a “partial candidate solution” and a relatively quick test of whether it can possibly be completed to a valid solution.

**Iterative backtracking algorithm :**algorithm ibacktrack ( $n$ )

```
// Iterative backtracking process
// All solutions are generated in $x[1 : n]$ and
printed as soon as they are found
```

```
{
k = 1;
while (k != 0)
{
if (there remains an untried $x[k]$ in $T(x[1], x[2], \dots, x[k - 1])$
and $B_k(x[1], \dots, x[k])$ is true)
{
if ($x[1], \dots, x[k]$ is a path to an answer node)
write ($x[1 : k]$);
k = k + 1; // Consider the next set
}
else
k = k - 1; // Backtrack to the previous set
}
```

**9. Differentiate NP-complete with NP-hard.**

**Ans.**

| S. No. | <b>NP-complete</b>                                                                                                                 | <b>NP-hard</b>                                                               |
|--------|------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| 1.     | An NP-complete problems is one to which every other polynomial-time non-deterministic algorithm can be reduced in polynomial time. | NP-hard problems is one to which an NP-complete problem is Turing-reducible. |
| 2.     | NP-complete problems do not corresponds to an NP-hard problem.                                                                     | NP-hard problems correspond to an NP-complete problem.                       |
| 3.     | NP-complete problems are exclusively decision problem.                                                                             | NP-hard problems need not to be decision problem.                            |
| 4.     | NP-complete problems have to be in NP-hard and also in NP.                                                                         | NP-hard problems do not have to be in NP.                                    |
| 5.     | <b>For example :</b> 3-SAT vertex cover problem is NP-complete.                                                                    | <b>For example :</b> Halting problem is NP-hard.                             |

### Section-C

**Note :** Attempt any **two** questions from this section. **(15 × 2 = 30)**

**10. i. State Bellman-Ford algorithm.**

**ii. Consider following instance for simple knapsack problem.**

**Find the solution using greedy method.**

$$N = 8$$

$$P = \{11, 21, 31, 33, 43, 53, 55, 65\}$$

$$W = \{1, 11, 21, 23, 33, 43, 45, 55\}$$

$$M = 110$$

**Ans.****i.**

1. Bellman-Ford algorithm finds all shortest path length from a source  $s \in V$  to all  $v \in V$  or determines that a negative-weight cycle exists.
2. Bellman-Ford algorithm solves the single source shortest path problem in the general case in which edges of a given digraph  $G$  can have negative weight as long as  $G$  contains no negative cycles.
3. This algorithm, uses the notation of edge relaxation but does not use with greedy method.
4. The algorithm returns boolean TRUE if the given digraph contains no negative cycles that are reachable from source vertex otherwise it returns boolean FALSE.

**Bellman-Ford ( $G, w, s$ ) :**

1. INITIALIZE-SINGLE-SOURCE ( $G, s$ )
2. for each vertex  $i \leftarrow 1$  to  $V[G] - 1$ 
  3. do for each edge  $(u, v)$  in  $E[G]$
  4. do RELAX ( $u, v, w$ )
5. for each edge  $(u, v)$  in  $E[G]$  do
6. do if  $d[u] + w(u, v) < d[v]$  then
7. then return FALSE

8. return TRUE

**RELAX ( $u, v, w$ ) :**

1. If  $d[u] + w(u, v) < d[v]$
2. then  $d[v] \leftarrow d[u] + w(u, v)$
3.  $\pi[v] \leftarrow u$

If Bellman-Ford returns true, then  $G$  forms a shortest path tree,  
else there exists a negative weight cycle.

**ii.**

$$N = 8$$

$$W = \{1, 11, 21, 23, 33, 43, 45, 55\}$$

$$P = \{11, 21, 31, 33, 43, 53, 55, 65\}$$

$$M = 110$$

Now, arrange the value of  $P_i$  in decreasing order

| <b><math>N</math></b> | <b><math>W_i</math></b> | <b><math>P_i</math></b> | <b><math>V_i = W_i \times P_i</math></b> |
|-----------------------|-------------------------|-------------------------|------------------------------------------|
| 1                     | 1                       | 11                      | 11                                       |
| 2                     | 11                      | 21                      | 231                                      |
| 3                     | 21                      | 31                      | 651                                      |
| 4                     | 23                      | 33                      | 759                                      |
| 5                     | 33                      | 43                      | 1419                                     |
| 6                     | 43                      | 53                      | 2279                                     |
| 7                     | 45                      | 55                      | 2475                                     |
| 8                     | 55                      | 65                      | 3575                                     |

Now, fill the knapsack according to decreasing value of  $P_i$ . First we choose item  $N = 1$  whose weight is 1.

Then choose item  $N = 2$  whose weight is 11.

Then choose item  $N = 3$  whose weight is 21.

Now, choose item  $N = 4$  whose weight is 23.

Then choose item  $N = 5$  whose weight is 33.

Total weight in knapsack is  $= 1 + 11 + 21 + 23 + 33 = 89$

Now, the next item is  $N = 6$  and its weight is 43, but we want only 21 because  $M = 110$ .

So, we choose fractional part of it, i.e.,

The value of fractional part of  $N = 6$  is,

$$\frac{2279}{43} \times 21 = 1113$$

|    |
|----|
| 21 |
| 33 |
| 23 |
| 21 |
| 11 |
| 1  |

}  $\Rightarrow 110$

Thus, the maximum value is,

$$\begin{aligned}
 &= 11 + 231 + 651 + 759 + 1419 + 1113 \\
 &= 4184
 \end{aligned}$$

- 11. What is travelling salesman problem ? Find the solution of following travelling salesman problem using branch and bound method.**

$$\text{Cost matrix} = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 6 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

**Ans. Travelling salesman problem :**

Travelling salesman problem is the problem to find the shortest possible route for a given set of cities and distance between the pair of cities that visits every city exactly once and returns to the starting point.

Branch and bound method and backtracking approach are used to solve TSP problem.

**Numerical :**

$$\text{Cost matrix} = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 6 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

1. Reduce each column and row by reducing the minimum value from each element in row and column.

**Row**

$$\begin{array}{l} 10 \rightarrow \\ 2 \rightarrow \\ 2 \rightarrow \\ 3 \rightarrow \\ 4 \rightarrow \end{array} \left[ \begin{array}{ccccc} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 3 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{array} \right]$$

**Column**

$$\left[ \begin{array}{ccccc} 1 & & 3 & & \\ \downarrow & & \downarrow & & \\ \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{array} \right] = M_1$$

2. So, total expected cost is :  $10 + 2 + 2 + 3 + 4 + 1 + 3 = 25$ .
3. We have discovered the root node  $V_1$  so the next node to be expanded will be  $V_2, V_3, V_4, V_5$ . Obtain cost of expanding using cost matrix for node 2.
4. Change all the elements in 1<sup>st</sup> row and 2<sup>nd</sup> column.

$$M_2 = \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 0 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{array} \right]$$

5. Now, reducing  $M_2$  in rows and columns, we get :

$$M_2 = \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 0 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{array} \right]$$

- $\therefore$  Total cost for  $M_2 = 25 + 10 + 0 = 35$
6. Similarly, for node 3, we have :

$$M_3 = \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{array} \right]$$

7. Now, reducing  $M_3$ , we get :

$$M_3 = \left[ \begin{array}{ccccc} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{array} \right]$$

$\therefore$  Total cost for  $M_3 = 25 + 17 + 0 = 42$

8. Similarly, for node 4, we have :

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

9. Now, reducing  $M_4$ , we get :

$$M_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ 15 & 3 & 0 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 0 + 0 = 25$$

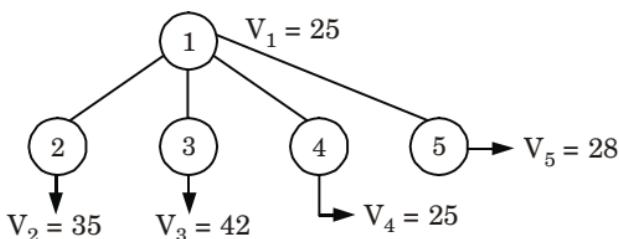
10. Similarly, for node 5, we have :

$$M_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 0 & \infty & \infty \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

11. Now, reducing  $M_5$ , we get :

$$M_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 0 & \infty & \infty \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 1 + 2 = 28$$



**Fig. 1.**

12. Now, the promising node is  $V_4 = 25$ . Now, we can expand  $V_2$ ,  $V_3$  and  $V_5$ . Now, the input matrix will be  $M_4$ .
13. Change all the elements in 4<sup>th</sup> row and 2<sup>nd</sup> column.

$$M_6 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

14. On reducing  $M_6$ , we get :

$$M_6 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 3 + 0 = 28$$

15. Similarly, for node 3, we have :

$$M_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

16. On reducing  $M_7$ , we get :

$$M_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$$\therefore \text{Total cost} = 25 + 0 + 0 = 25$$

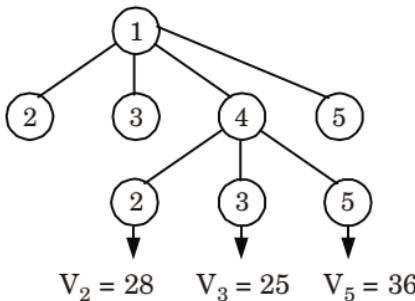
17. Similarly, for node 5, we have :

$$M_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

18. On reducing  $M_8$ , we get :

$$M_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$\therefore$  Total cost =  $25 + 0 + 11 = 36$



**Fig. 2.**

19. Now, promising node is  $V_3 = 25$ . Now, we can expand  $V_2$  and  $V_5$ .  
Now, the input matrix will be  $M_7$ .
20. Change all the elements in 3<sup>rd</sup> row and 2<sup>nd</sup> column.

$$M_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

21. On reducing  $M_9$ , we get :

$$M_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

$\therefore$  Total cost =  $25 + 3 + 0 = 28$

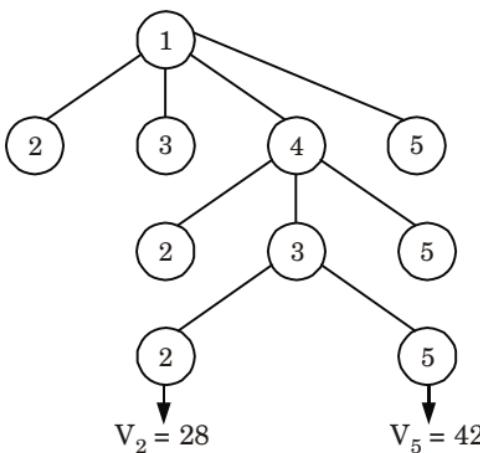
22. Similarly, for node 5, we have :

$$M_{10} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

23. On reducing  $M_{10}$ , we get :

$$M_{10} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$\therefore$  Total cost =  $25 + 2 + 12 + 3 = 42$ .

**Fig. 3.**

24. Here  $V_2$  is the most promising node so next we are going to expand this node further. Now, we are left with only one node not yet traversed which is  $V_5$ .

$$V_1 \xrightarrow{10} V_4 \xrightarrow{6} V_3 \xrightarrow{5} V_2 \xrightarrow{2} V_5 \xrightarrow{16} V_1$$

So, total cost of traversing the graph is :

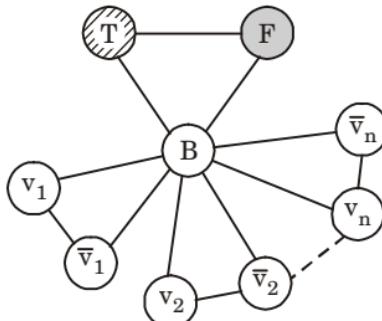
$$10 + 6 + 5 + 2 + 16 = 39$$

## 12. Prove that three colouring problem is NP-complete.

**Ans.**

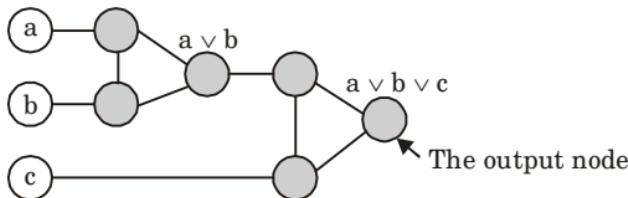
1. To show the problem is in NP, let us take a graph  $G(V, E)$  and a colouring  $c$ , and checks in  $O(n^2)$  time whether  $c$  is a proper colouring by checking if the end points of every edge  $e \in E$  have different colours.
2. To show that 3-COLOURING is NP-hard, we give a polytime reduction from 3-SAT to 3-COLOURING.
3. That is, given an instance  $\phi$  of 3-SAT, we will construct an instance of 3-COLOURING (*i.e.*, a graph  $G(V, E)$ ) where  $G$  is 3-colourable iff  $\phi$  is satisfiable.
4. Let  $\phi$  be a 3-SAT instance and  $C_1, C_2, \dots, C_m$  be the clauses of  $\phi$  defined over the variables  $\{x_1, x_2, \dots, x_n\}$ .
5. The graph  $G(V, E)$  that we will construct needs to capture two things :
  - a. Somehow establish the truth assignment for  $x_1, x_2, \dots, x_n$  via the colours of the vertices of  $G$ ; and
  - b. Somehow capture the satisfiability of every clause  $C_i$  in  $\phi$ .
6. To achieve these two goals, we will first create a triangle in  $G$  with three vertices  $\{T, F, B\}$  where  $T$  stands for True,  $F$  for False and  $B$  for Base.
7. Consider  $\{T, F, B\}$  as the set of colours that we will use to colour (label) the vertices of  $G$ .
8. Since this triangle is part of  $G$ , we need 3 colours to colour  $G$ .

9. Now we add two vertices  $v_i, \bar{v}_i$  for every literal  $x_i$  and create a triangle  $B, v_i, \bar{v}_i$  for every  $(v_i, \bar{v}_i)$  pair, as shown in Fig. 4.



**Fig. 4.**

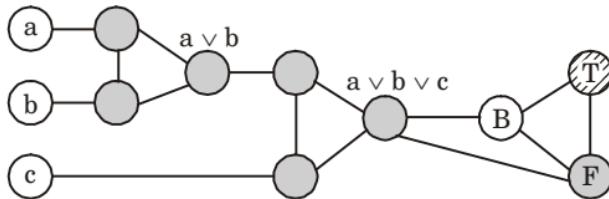
10. This construction captures the truth assignment of the literals.  
 11. Since if  $G$  is 3-colourable, then either  $v_i$  or  $\bar{v}_i$  gets the colour  $T$ , and we interpret this as the truth assignment to  $v_i$ .  
 12. Now we need to add constraints to  $G$  to capture the satisfiability of the clauses of  $\phi$ .



**Fig. 5.**

13. To do so, we introduce the Clause Satisfiability Gadget, (the OR-gadget). For a clause  $C_i = (a \vee b \vee c)$ , we need to express the OR of its literals using our colours  $\{T, F, B\}$ .  
 14. We achieve this by creating a small gadget graph that we connect to the literals of the clause. The OR-gadget is constructed as follows :  
 15. Consider this gadget graph as a circuit whose output is the node labeled  $a \vee b \vee c$ . We basically want this node to be coloured  $T$  if  $C_i$  is satisfied and  $F$  otherwise.  
 16. This is a two step construction : The node labelled  $a \vee b$  captures the output of  $(a \vee b)$  and we repeat the same operation for  $((a \vee b) \vee c)$ . If we play around with some assignments to  $a, b, c$ , we will notice that the gadget satisfies the following properties :  
 a. If  $a, b, c$  are all coloured  $F$  in a 3-colouring, then the output node of the OR-gadget has to be coloured  $F$ . Thus capturing the unsatisfiability of the clause  $C_i = (a \vee b \vee c)$ .  
 b. If one of  $a, b, c$  is coloured  $T$ , then there exists a valid 3-colouring of the OR-gadget where the output node is coloured  $T$ . Thus again capturing the satisfiability of the clause.

17. Once we add the OR-gadget of every  $C_i$  in  $\phi$ , we connect the output node of every gadget to the Base vertex and to the False vertex of the initial triangle, as follows :

**Fig. 6.**

18. Now we prove that our initial 3-SAT instance  $\phi$  is satisfiable if and only the graph  $G$  as constructed above is 3-colourable. Suppose  $\phi$  is satisfiable and let  $(x_1^*, x_2^*, \dots, x_n^*)$  be the satisfying assignment.
19. If  $x_i^*$  is assigned True, we colour  $v_i$  with  $T$  and  $\bar{v}_i$  with  $F$  (recall they are connected to the Base vertex, coloured  $B$ , so this is a valid colouring).
20. Since  $\phi$  is satisfiable, every clause  $C_i = (a \vee b \vee c)$  must be satisfiable, i.e., at least of  $a, b, c$  is set to True. By the property of the OR-gadget, we know that the gadget corresponding to  $C_i$  can be 3-coloured so that the output node is coloured  $T$ .
21. And because the output node is adjacent to the False and Base vertices of the initial triangle only, this is a proper 3-colouring.
22. Conversely, suppose  $G$  is 3-colourable. We construct an assignment of the literals of  $\phi$  by setting  $x_i$  to True if  $v_i$  is coloured  $T$  and vice versa.
23. Now consider this assignment is not a satisfying assignment to  $\phi$ , then this means there exists at least one clause  $C_i = (a \vee b \vee c)$  that was not satisfiable.
24. That is, all of  $a, b, c$  were set to False. But if this is the case, then the output node of corresponding OR-gadget of  $C_i$  must be coloured  $F$ .
25. But this output node is adjacent to the False vertex coloured  $F$ ; thus contradicting the 3-colourability of  $G$ .
26. To conclude, we have shown that 3-COLOURING is in NP and that it is NP-hard by giving a reduction from 3-SAT.
27. Therefore 3-COLOURING is NP-complete.



**B.Tech.****(SEM. V) ODD SEMESTER THEORY  
EXAMINATION, 2017-18****DESIGN AND ANALYSIS OF ALGORITHM****Time : 3 Hours****Max. Marks : 100****Note :** Attempt all sections. Assume any missing data.**Section-A**

- 1. Define/Explain the following : (2 × 10 = 20)**
- a. Differentiate between complete binary tree and binary tree.
- b. Differentiate between greedy technique and dynamic programming.
- c. Solve the following recurrence using master method :  
$$T(n) = 4T(n/3) + n^2$$
- d. Name the sorting algorithm that is most practically used and also write its time complexity.
- e. Find the time complexity of the recurrence relation  
$$T(n) = n + T(n/10) + T(7n/5)$$
- f. Explain single source shortest path.
- g. Define graph colouring.
- h. Compare time complexity with space complexity.
- i. What are the characteristics of the algorithm ?
- j. Differentiate between backtracking and branch and bound techniques.

**SECTION-B**

2. Attempt any three of the following : (10 × 3 = 30)
- a. Solve the following by recursion tree method  
$$T(n) = n + T(n/5) + T(4n/5)$$

- b. Insert the following information,  $F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I$  into an empty B-tree with degree  $t = 3$ .
- c. What is minimum cost spanning tree ? Explain Kruskal's algorithm and Find MST of the graph. Also write it's time complexity.

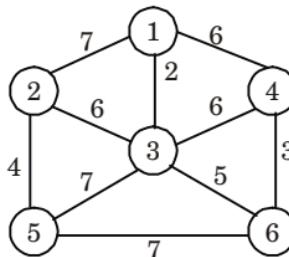


Fig. 2.

- d. What is red-black tree ? Write an algorithm to insert a node in an empty red-black tree explain with suitable example.
- e. Explain HEAP SORT on the array. Illustrate the operation HEAP SORT on the array

$$A = \{6, 14, 3, 25, 2, 10, 20, 7, 6\}$$

### SECTION-C

3. Attempt any one part of the following :  $(10 \times 1 = 10)$
- a. Explain convex-hull problem.
- b. Find the shortest path in the below graph from the source vertex 1 to all other vertices by using Dijkstra's algorithm.

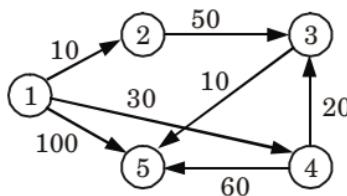


Fig. 2.

4. Attempt any one part of the following :  $(10 \times 1 = 10)$
- a. What is backtracking ? Discuss sum of subset problem with the help of an example.
- b. Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyze its time complexity.

5. Attempt any **one** part of the following :  $(10 \times 1 = 10)$
- The recurrence  $T(n) = 7T(n/2) + n^2$  describe the running time of an algorithm A. A competing algorithm A' has a running time  $T'(n) = aT'(n/4) + n^2$ . What is the largest integer value for a A' is asymptotically faster than A ?**
  - Discuss the problem classes P, NP and NP-complete with class relationship.**
6. Attempt any one part of the following :  $(10 \times 1 = 10)$
- Explain properties of binomial heap. Write an algorithm to perform uniting two binomial heaps. And also to find minimum key.**
  - Given the six items in the table below and a knapsack with weight 100, what is the solution to the knapsack problem in all concepts. i.e., explain greedy all approaches and find the optimal solution.**

| Item ID | Weight | Value | Value/Weight |
|---------|--------|-------|--------------|
| A       | 100    | 40    | .4           |
| B       | 50     | 35    | .7           |
| C       | 40     | 20    | .5           |
| D       | 20     | 4     | .2           |
| E       | 10     | 10    | 1            |
| F       | 10     | 6     | .6           |

7. Attempt any **one** part of the following :  $(10 \times 1 = 10)$
- Compute the prefix function  $\pi$  for the pattern  $P = a b a c a b$  using Knuth-Morris-Pratt algorithm. Also explain Naive string matching algorithm.**
  - Explain approximation and randomized algorithms.**



## SOLUTION OF PAPER (2017-18)

**Note :** Attempt all sections. Assume any missing data.

### Section-A

- 1. Define/Explain the following :  $(2 \times 10 = 20)$**
- a. Differentiate between complete binary tree and binary tree.**

**Ans.**

| S. No. | Complete binary tree                                                                                                                               | Binary tree                                                                                                                                      |
|--------|----------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.     | In a complete binary tree every level, except possibly the last is completely filled, and all nodes in the last level are as far left as possible. | A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. |
| 2      | It can have between 1 and $2^{h-1}$ nodes at the last level $h$ .                                                                                  | It can have between $2^{h+1}$ and $2^{h+1} - 1$ nodes at the last level $h$ .                                                                    |

- b. Differentiate between greedy technique and dynamic programming.**

**Ans.** Difference :

| S. No. | Greedy method                                                                                | Dynamic programming                                                                                                                                   |
|--------|----------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.     | A greedy algorithm is one that at a given point in time makes a local optimization.          | Dynamic programming can be thought of as 'smart' recursion. It often requires one to break down a problem into smaller components that can be cached. |
| 2.     | Greedy algorithms have a local choice of the subproblem that will lead to an optimal answer. | Dynamic programming would solve all dependent subproblems and then select one that would lead to an optimal solution.                                 |

- c. Solve the following recurrence using master method :**

$$T(n) = 4T(n/3) + n^2$$

**Ans.**  $T(n) = 4T(n/3) + n^2$

$$a = 4, \quad b = 3, \quad f(n) = n^2$$

$$n^{\log_b a} = n^{\log_3 4} = n^{1.261}$$

$$f(n) = \Omega(n^{\log_b a + E})$$

Now,  $af(n/b) \leq c f(n)$

$$\frac{4}{3} f(n) \leq c f(n)$$

$$\frac{4}{3} n^2 \leq cn^2$$

$$c = \frac{4}{3}$$

Hence,  $T(n) = \Theta(n^2)$

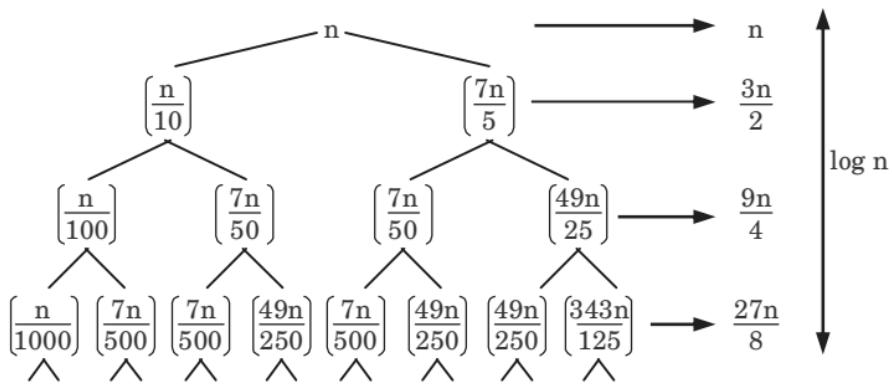
- d. Name the sorting algorithm that is most practically used and also write its time complexity.**

**Ans.** Quick sort algorithm is most practically used in sorting.  
Time complexity of quick sort is  $O(n \log n)$ .

- e. Find the time complexity of the recurrence relation**  

$$T(n) = n + T(n/10) + T(7n/5)$$

**Ans.**



$$\begin{aligned} T(n) &= \frac{3^0 n}{2^0} + \frac{3^1 n}{2^1} + \frac{3^2 n}{2^1} + \frac{3^3 n}{2^3} + \dots + \log n \text{ times} \\ &= \Omega(n \log n) \end{aligned}$$

- f. Explain single source shortest path.**

**Ans.** Single source shortest path problem states that in a given graph  $G = (V, E)$  we can find a shortest path from given source vertex  $s \in V$  to every vertex  $v \in V$ .

- g. Define graph colouring.**

**Ans.** Graph colouring is a problem of colouring each vertex in graph in such a way that no two adjacent vertices have same colour and  $m$ -colours are used.

- h. Compare time complexity with space complexity.**

**Ans.**

| S. No. | Time complexity                                                                          | Space complexity                                                                       |
|--------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| 1.     | Time complexity is the amount of time required for an algorithm to complete its process. | Space complexity is the amount of memory needed for an algorithm to solve the problem. |
| 2.     | It is expressed using Big Oh(O), theta ( $\theta$ ) and omega ( $\Omega$ ) notation.     | It is expressed only using Big Oh(O) notation.                                         |

**i. What are the characteristics of the algorithm ?****Ans. Characteristics of algorithm :**

- Input and output :** The algorithm must accept zero or more inputs and must produce at least one output.
- Definiteness :** Each step of algorithm must be clear and unambiguous.
- Effectiveness :** Every step must be basic and essential.
- Finiteness :** Total number of steps used in algorithm should be finite.

**j. Differentiate between backtracking and branch and bound techniques.****Ans.**

| S. No. | Backtracking                                                  | Branch and bound                                                                                                                                           |
|--------|---------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.     | Solution for backtracking is traced using depth first search. | In this method, it is not necessary to use depth first search for obtaining the solution, even the breadth first search, best first search can be applied. |
| 2.     | Typically decision problems can be solved using backtracking. | Typically optimization problems can be solved using branch and bound.                                                                                      |

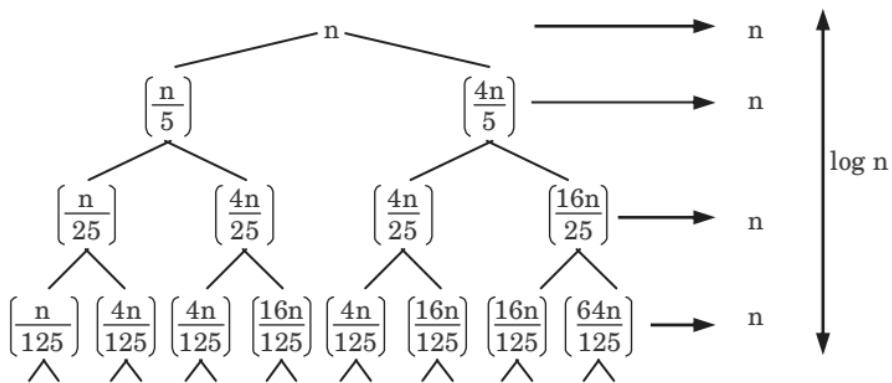
**SECTION-B**

2. Attempt any **three** of the following : **(10 × 3 = 30)**

**a. Solve the following by recursion tree method**

$$T(n) = n + T(n/5) + T(4n/5)$$

**Ans.**  $T(n) = n + n + n + \dots + \log n \text{ times} = \Omega(n \log n)$



- b. Insert the following information,  $F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, G, I$  into an empty B-tree with degree  $t = 3$ .

**Ans.** Assume that  $t = 3$   
 $2t - 1 = 2 \times 3 - 1 = 6 - 1 = 5$   
and  $t - 1 = 3 - 1 = 2$

So, maximum of 5 keys and minimum of 2 keys can be inserted in a node. Now, apply insertion process as :

**Insert F :**

F

**Insert S :**

F | S

**Insert Q :**

F | Q | S

**Insert K :**

F | K | Q | S

**Insert C :**

C | F | K | Q | S

**Insert L :**

C | F | K | L | Q | S

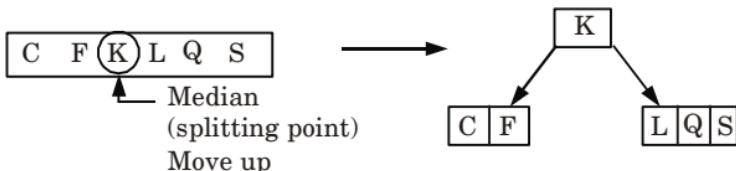
As, there are more than 5 keys in this node.

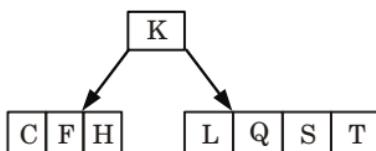
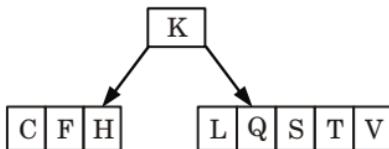
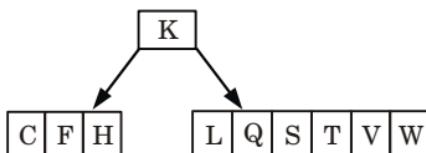
∴ Find median,  $n[x] = 6$  (even)

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3$$

Now, median = 3,

So, we split the node by 3<sup>rd</sup> key.

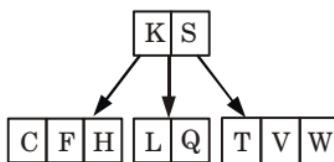
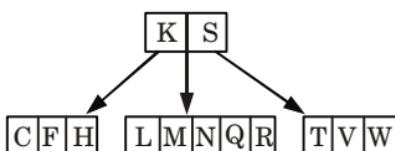
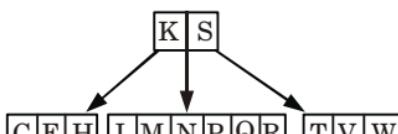


**Insert H, T :****Insert V :****Insert W :**

More than 5 keys split node from Median.

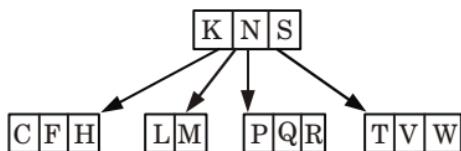
$$n[x] = 6 \text{ [even]}$$

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$

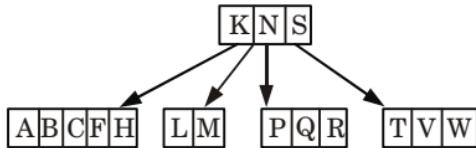
**Insert M, R, N :****Insert P :**More than 5 key  
split the node

$$n[x] = 6$$

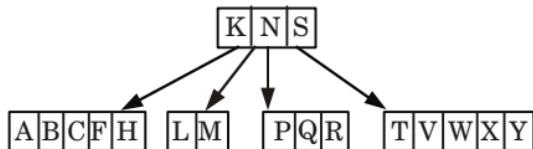
$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$



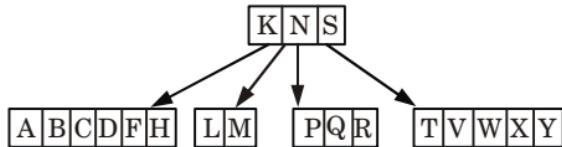
**Insert A, B :**



**Insert X, Y :**



**Insert D :**

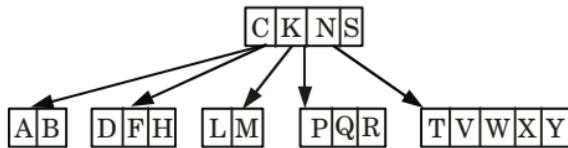


More than 5 key

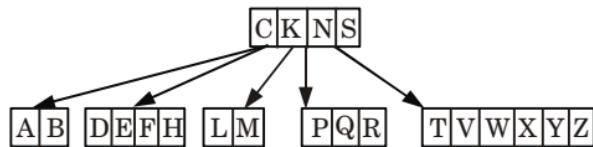
split the node

$$n[x] = 6 \text{ (even)}$$

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$



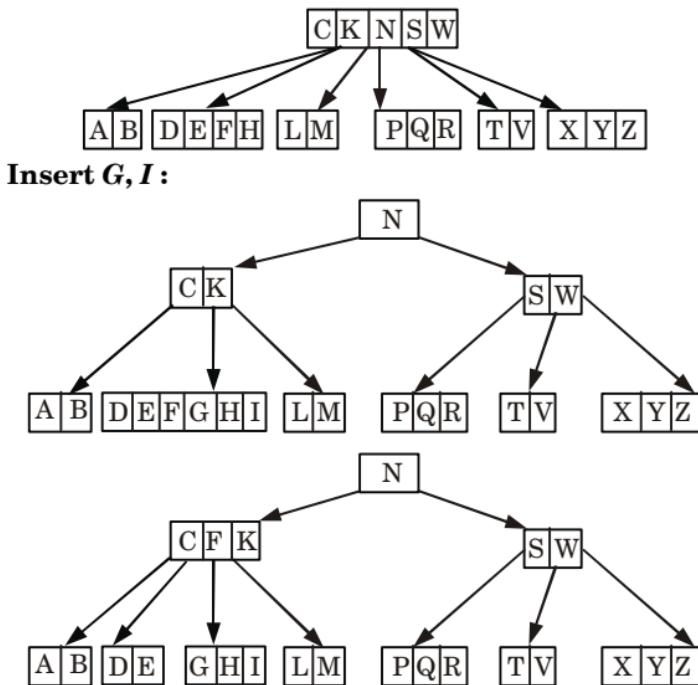
**Insert Z, E :**



More than 5 key  
split the node

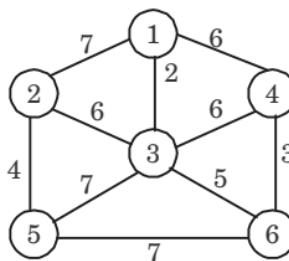
$$n[x] = 6$$

$$\text{Median} = \frac{n[x]}{2} = \frac{6}{2} = 3 \quad (\text{i.e., 3rd key move up})$$



**Fig. 1.** Inserted all given information with degree  $t = 3$ .

- c. What is minimum cost spanning tree ? Explain Kruskal's algorithm and Find MST of the graph. Also write it's time complexity.



**Fig. 2.**

**Ans.** Minimum spanning tree (Minimum cost spanning tree) :

- Given a connected weighted graph  $G$ , it is often desired to create a spanning tree  $T$  for  $G$  such that the sum of the weights of the tree edges in  $T$  is as small as possible.
- Such a tree is called a minimum spanning tree and represents the “cheapest” way of connecting all the nodes in  $G$ .
- There are number of techniques for creating a minimum spanning tree for a weighted graph but the most famous methods are Prim's and Kruskal's algorithm.

**Kruskal's algorithm :**

- In this algorithm, we choose an edge of  $G$  which has smallest weight among the edges of  $G$  which are not loops.

- ii. This algorithm gives an acyclic subgraph  $T$  of  $G$  and the theorem given below proves that  $T$  is minimal spanning tree of  $G$ . Following steps are required :

**Step 1 :** Choose  $e_1$ , an edge of  $G$ , such that weight of  $e_1$ ,  $w(e_1)$  is as small as possible and  $e_1$  is not a loop.

**Step 2 :** If edges  $e_1, e_2, \dots, e_i$  have been selected then choose an edge  $e_{i+1}$  not already chosen such that

- i. the induced subgraph

$G[\{e_1, \dots, e_{i+1}\}]$  is acyclic and

- ii.  $w(e_{i+1})$  is as small as possible

**Step 3 :** If  $G$  has  $n$  vertices, stop after  $n - 1$  edges have been chosen. Otherwise repeat step 2.

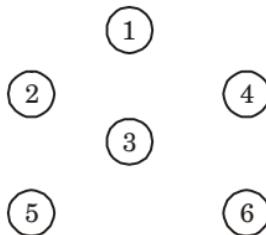
If  $G$  be a weighted connected graph in which the weight of the edges are all non-negative numbers, let  $T$  be a subgraph of  $G$  obtained by Kruskal's algorithm then,  $T$  is minimal spanning tree.

#### Numerical :

**Step 1 :** Arrange the edge of graph according to weight in ascending order.

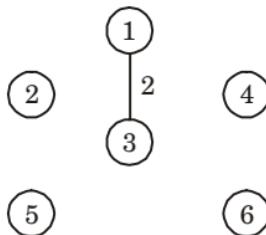
| Edges | Weight | Edge | Weight |
|-------|--------|------|--------|
| 13    | 2      | 32   | 6      |
| 46    | 3      | 17   | 7      |
| 25    | 4      | 35   | 7      |
| 36    | 5      | 56   | 7      |
| 34    | 6      |      |        |
| 41    | 6      |      |        |

**Step 2 :** Now draw the vertices as given in graph,

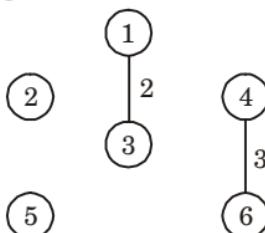


Now draw the edge according to the ascending order of weight. If any edge forms cycle, leave that edge.

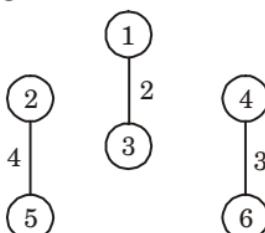
**Step 3 :** Select edge 13



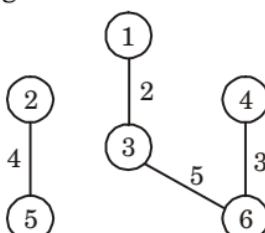
**Step 4 :** Select edge 46



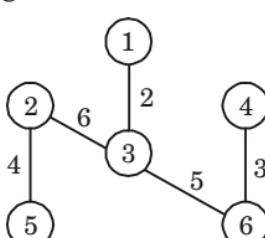
**Step 5 :** Select edge 25



**Step 6 :** Select edge 36

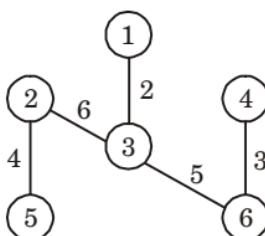


**Step 7 :** Select edge 23



All the remaining edges, such as 34, 41, 12, 35, 56 are rejected because they form cycle.

All the vertices are covered in this tree. So, the final tree with minimum cost of given graph is



$$\text{Minimum cost} = 2 + 3 + 4 + 5 + 6 = 20$$

**Time complexity :** Time complexity is  $O(|E| \log |E|)$ .

- d. What is red-black tree ? Write an algorithm to insert a node in an empty red-black tree explain with suitable example.**

**Ans. Red-black tree :**

- A red-black tree is a binary tree where each node has colour as an extra attribute, either red or black.
- It is a self-balancing Binary Search Tree (BST) where every node follows following properties :
  - Every node is either red or black.
  - The root is black.
  - Every leaf (NIL) is black.
  - If a node is red, then both its children are black.
  - For each node, all paths from the node to descendent leave contain the same number of black nodes.

**Insertion algorithm :**

- We begin by adding the node as we do in a simple binary search tree and colouring it red.

**RB-INSERT( $T, z$ )**

- $y \leftarrow \text{nil } [T]$
- $x \leftarrow \text{root } [T]$
- while  $x \neq \text{nil } [T]$
- do  $y \leftarrow x$
- if  $\text{key}[z] < \text{key}[x]$
- then  $x \leftarrow \text{left } [x]$
- else  $x \leftarrow \text{right } [x]$
- $p[z] \leftarrow y$
- if  $y = \text{nil } [T]$
- then  $\text{root } [T] \leftarrow z$
- else if  $\text{key}[z] < \text{key}[y]$
- then  $\text{left } [y] \leftarrow z$
- else  $\text{right } [y] \leftarrow z$
- $\text{left } [z] \leftarrow \text{nil}[T]$
- $\text{right } [z] \leftarrow \text{nil}[T]$
- $\text{colour } [z] \leftarrow \text{RED}$
- RB-INSERT-FIXUP( $T, z$ )**

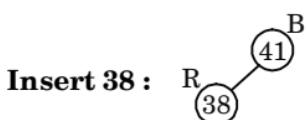
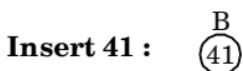
- Now, for any colour violation, RB-INSERT-FIXUP procedure is used.

**RB-INSERT-FIXUP( $T, z$ )**

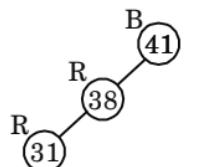
- while  $\text{colour } [p[z]] = \text{RED}$
- do if  $p[z] = \text{left}[p[p[z]]]$
- then  $y \leftarrow \text{right}[p[p[z]]]$
- if  $\text{colour } [y] = \text{RED}$
- then  $\text{colour } [p[z]] \leftarrow \text{BLACK}$   $\Rightarrow \text{case 1}$
- $\text{colour } [y] \leftarrow \text{BLACK}$   $\Rightarrow \text{case 1}$
- $\text{colour } [p[p[z]]] \leftarrow \text{RED}$   $\Rightarrow \text{case 1}$
- $z \leftarrow p[p[z]]$   $\Rightarrow \text{case 1}$
- else if  $z = \text{right}[p[z]]$
- then  $z \leftarrow p[z]$   $\Rightarrow \text{case 2}$

11.  $\text{LEFT-ROTATE}(T, z)$   $\Rightarrow$  case 2  
 12.  $\text{colour}[p[z]] \leftarrow \text{BLACK}$   $\Rightarrow$  case 3  
 13.  $\text{colour}[p[p[z]]] \leftarrow \text{RED}$   $\Rightarrow$  case 3  
 14.  $\text{RIGHT-ROTATE}(T, p[p[z]])$   $\Rightarrow$  case 3  
 15. else (same as then clause with "right" and "left" exchanged)  
 16.  $\text{colour}[\text{root}[T]] \leftarrow \text{BLACK}$

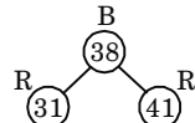
**For example :**



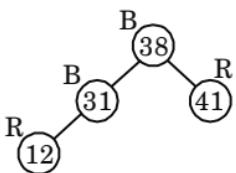
**Insert 31 :**



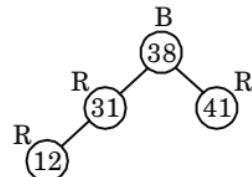
Case 3  $\rightarrow$



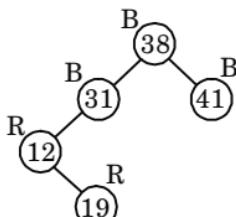
**Insert 12 :**



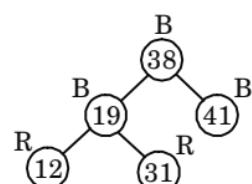
Case 1  $\rightarrow$



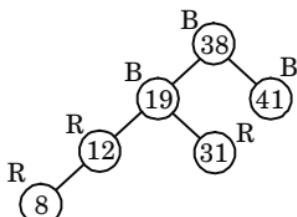
**Insert 19 :**



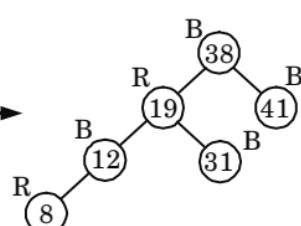
Case 2, 3  $\rightarrow$



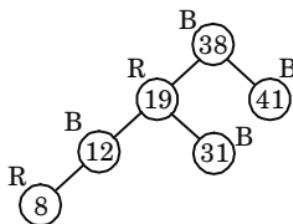
**Insert 8 :**



Case 1  $\rightarrow$



Thus final tree is



- e. Explain HEAP SORT on the array. Illustrate the operation HEAP SORT on the array

$$A = \{6, 14, 3, 25, 2, 10, 20, 7, 6\}$$

**Ans.** **Heap sort :**

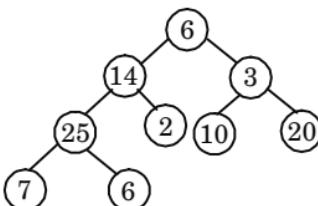
1. Heap sort is a comparison based sorting technique based on binary heap data structure.
2. Heap sort finds the largest element and puts it at the end of array, then the second largest item is found and this process is repeated for all other elements.
3. The general approach of heap sort is as follows :
  - a. From the given array, build the initial max heap.
  - b. Interchange the root (maximum) element with the last element.
  - c. Use repetitive downward operation from root node to rebuild the heap of size one less than the starting.
- d. Repeat step a and b until there are no more elements.

**Numerical :**

Originally the given array is : [6, 14, 3, 25, 2, 10, 20, 7, 6]

First we call BUILD-MAX-HEAP

heap size [A] = 9



so,

$i = 4$  to  $1$ , call MAX-HEAPIFY ( $A, i$ )

i.e., first we call MAX-HEAPIFY ( $A, 4$ )

$Az [l] = 7, A [i] = A [4] = 25, A [r] = 6$

$l \leftarrow \text{left}[4] = 8$

$r \leftarrow \text{right}[4] = 9$

$8 \leq 9$  and  $7 > 25$  (False)

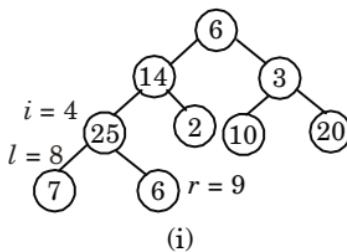
Then, largest  $\leftarrow 4$

$9 \leq 9$  and  $6 > 25$  (False)

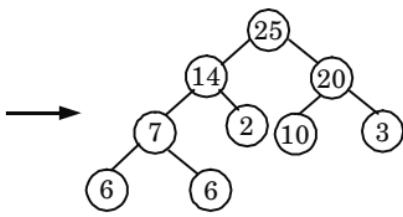
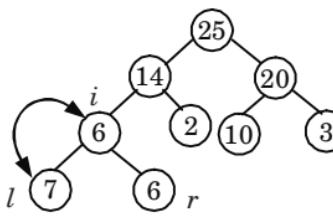
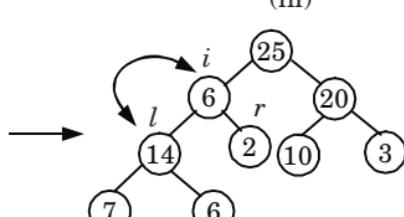
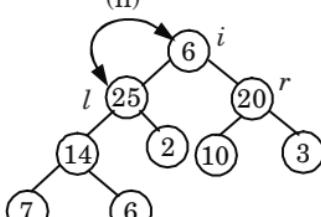
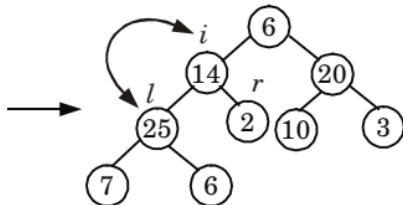
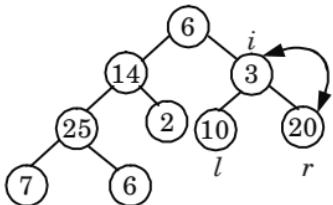
Then, largest = 4

$A [i] \leftrightarrow A [4]$

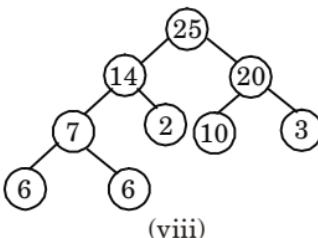
Now call MAX-HEAPIFY ( $A, 2$ )



Similarly for  $i = 3, 2, 1$  we get the following heap tree.

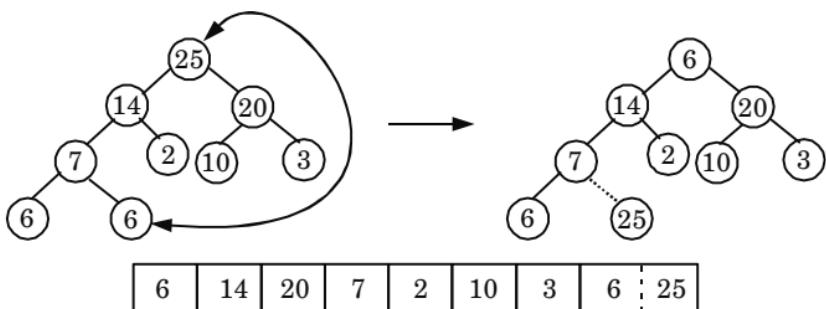


So final tree after BUILD-MAX-HEAP is



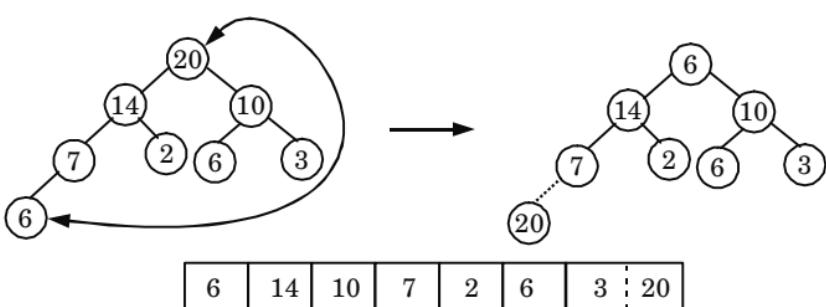
Now  $i = 9$  down to 2, and size = size - 1 and call MAX-HEAPIFY ( $A, 1$ ) each time.

exchanging  $A[1] \leftrightarrow A[9]$



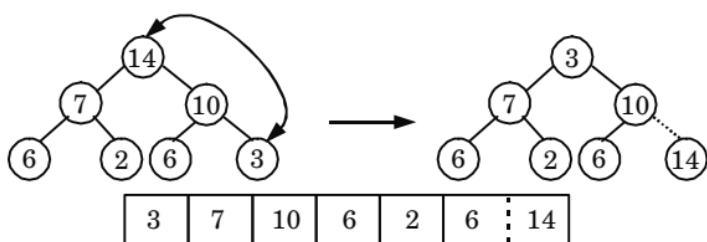
Now call MAX-HEAPIFY ( $A, 1$ ) we get

Now exchange  $A[1]$  and  $A[8]$  and size =  $8 - 1 = 7$



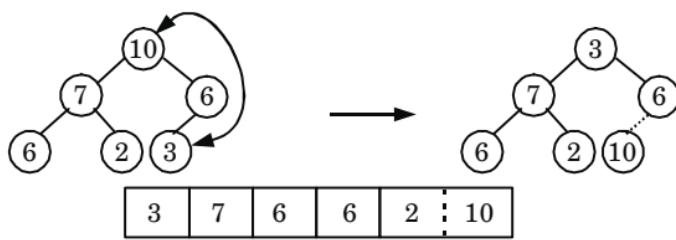
Again call MAX-HEAPIFY ( $A, 1$ ), we get

exchange  $A[1]$  and  $A[7]$  and size =  $7 - 1 = 6$



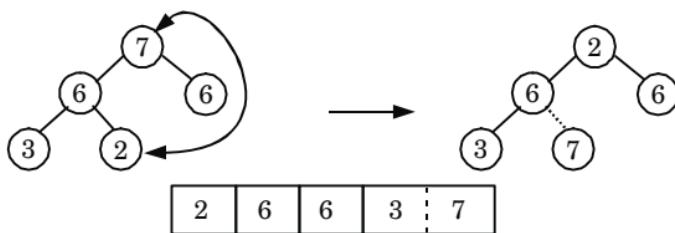
Again call MAX-HEAPIFY ( $A, 1$ ), we get

exchange  $A[1]$  and  $A[6]$  and now size =  $6 - 1 = 5$

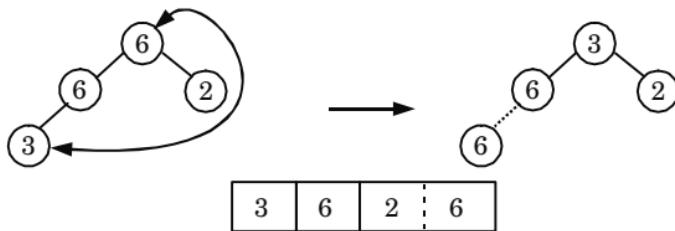


Again call MAX-HEAPIFY ( $A, 1$ )

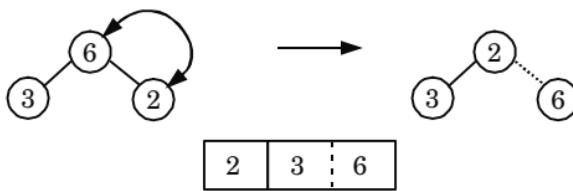
exchange  $A[1]$  and  $A[5]$  and now size =  $5 - 1 = 4$



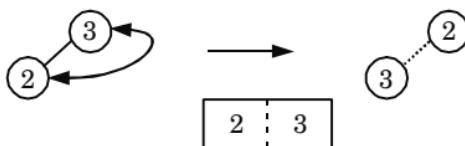
Again, call MAX-HEAPIFY ( $A, 1$ )  
exchange  $A[1]$  and  $A[4]$  and size =  $4 - 1 = 3$



call MAX-HEAPIFY ( $A, 1$ )  
exchange  $A[1]$  and  $A[3]$ , size =  $3 - 1 = 2$



call MAX-HEAPIFY ( $A, 1$ )  
exchange  $A[1]$  and  $A[2]$  and size =  $2 - 1 = 1$



Thus, sorted array :

|   |   |   |   |   |    |    |    |    |
|---|---|---|---|---|----|----|----|----|
| 2 | 3 | 6 | 6 | 7 | 10 | 14 | 20 | 25 |
|---|---|---|---|---|----|----|----|----|

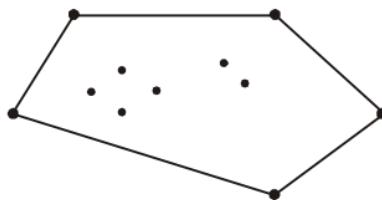
### SECTION-C

3. Attempt any **one** part of the following : **(10 × 1 = 10)**  
**a. Explain convex-hull problem.**

**Ans.**

1. The convex hull of a set  $S$  of points in the plane is defined as the smallest convex polygon containing all the points of  $S$ .
2. The vertices of the convex hull of a set  $S$  of points form a (not necessarily proper) subset of  $S$ .

3. To check whether a particular point  $p \in S$  is extreme, see each possible triplet of points and check whether  $p$  lies in the triangle formed by these three points.

**Fig. 3.**

4. If  $p$  lies in any triangle then it is not extreme, otherwise it is.  
 5. We denote the convex hull of  $S$  by  $\text{CH}(S)$ . Convex hull is a convex set because the intersection of convex sets is convex and convex hull is also a convex closure.

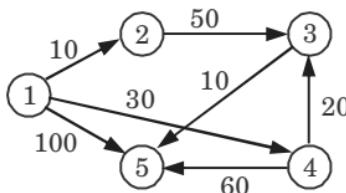
#### **Graham-Scan algorithm :**

The procedure GRAHAM-SCAN takes as input a set  $Q$  of points, where  $|Q| \geq 3$ . It calls the functions  $\text{Top}(S)$ , which return the point on top of stack  $S$  without changing  $S$ , and to  $\text{NEXT-TO-TOP}(S)$ , which returns the point one entry below the top of stack  $S$  without changing  $S$ .

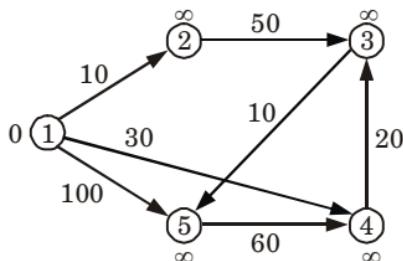
#### **GRAHAM-SCAN( $Q$ )**

1. Let  $p_0$  be the point in  $Q$  with the minimum  $y$ -coordinate, or the leftmost such point in case of a tie.
2. Let  $\langle p_1, p_2, \dots, p_m \rangle$  be the remaining points in  $Q$ , sorted by polar angle in counter clockwise order around  $p_0$  (if more than one point has the same angle remove all but the one that is farthest from  $p_0$ ).
3.  $\text{Top}[S] \leftarrow 0$
4.  $\text{PUSH}(p_0, S)$
5.  $\text{PUSH}(p_1, S)$
6.  $\text{PUSH}(p_2, S)$
7. for  $i \leftarrow 3$  to  $m$
8. do while the angle formed by points  $\text{NEXT-TO-TOP}(S)$ ,  $\text{Top}(S)$ , and  $p_i$  makes a non left turn.
9. do  $\text{POP}(S)$
10.  $\text{PUSH}(p_i, S)$
11. return  $S$

- b. Find the shortest path in the below graph from the source vertex 1 to all other vertices by using Dijkstra's algorithm.

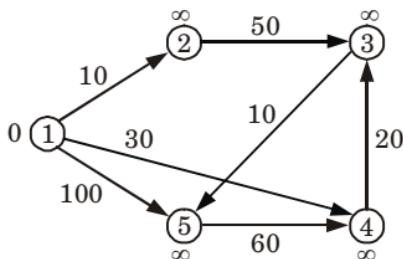
**Fig. 4.**

**Ans. Initialize :**



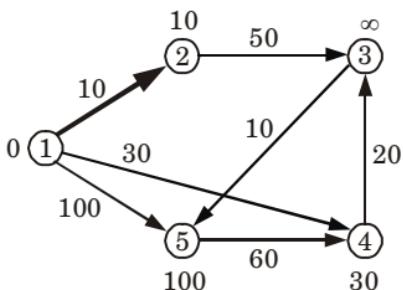
|               |   |          |          |          |
|---------------|---|----------|----------|----------|
| S : { }       |   |          |          |          |
| Q : 1 2 3 4 5 | 0 | $\infty$ | $\infty$ | $\infty$ |

**Extract min (1) :**



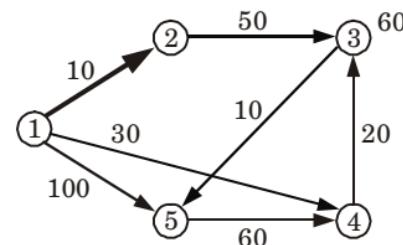
|               |   |          |          |          |
|---------------|---|----------|----------|----------|
| S : {1}       |   |          |          |          |
| Q : 1 2 3 4 5 | 0 | $\infty$ | $\infty$ | $\infty$ |

**All edges leaving (1) :**



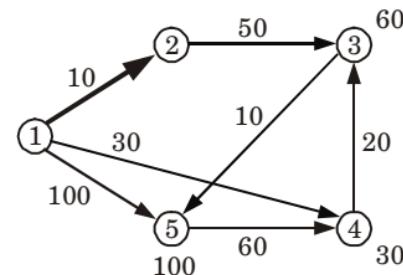
|               |    |          |          |          |
|---------------|----|----------|----------|----------|
| S : {1}       |    |          |          |          |
| Q : 1 2 3 4 5 | 0  | $\infty$ | $\infty$ | $\infty$ |
|               | 10 | $\infty$ | 30       | 100      |

**Extract min(2) :**



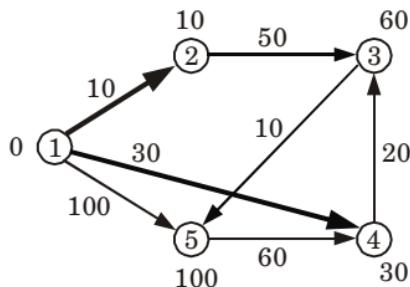
|               |    |          |          |          |
|---------------|----|----------|----------|----------|
| S : {1, 2}    |    |          |          |          |
| Q : 1 2 3 4 5 | 0  | $\infty$ | $\infty$ | $\infty$ |
|               | 10 | 60       | 30       | 100      |

**All edges leaving (2) :**



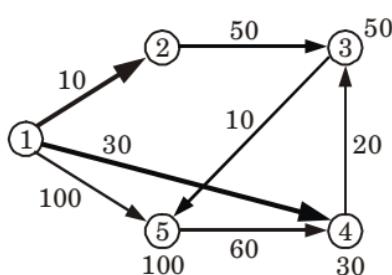
|               |    |          |          |          |
|---------------|----|----------|----------|----------|
| S : {1, 2}    |    |          |          |          |
| Q : 1 2 3 4 5 | 0  | $\infty$ | $\infty$ | $\infty$ |
|               | 10 | 60       | 30       | 100      |
|               | 60 | 30       | 100      |          |

**Extract min(4) :**



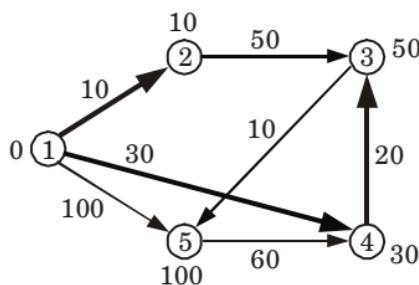
| S : {1, 2, 4} |          |          |          |          |
|---------------|----------|----------|----------|----------|
| Q : 1         | 2        | 3        | 4        | 5        |
| 0             | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10            | 60       | 30       | 100      |          |
| 60            | 30       | 100      |          |          |

**All edges leaving (4) :**



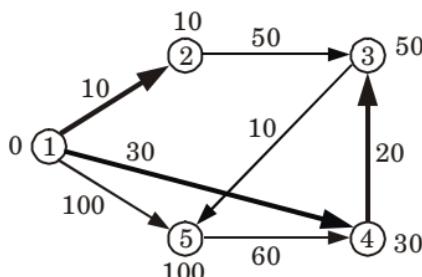
| S : {1, 2, 4} |          |          |          |          |
|---------------|----------|----------|----------|----------|
| Q : 1         | 2        | 3        | 4        | 5        |
| 0             | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10            | 60       | 30       | 100      |          |
| 60            | 30       | 100      |          |          |
|               |          |          | 50       |          |

**Extract min(3) :**

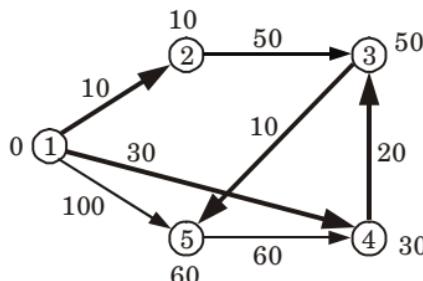


| S : {1, 2, 4, 3} |          |          |          |          |
|------------------|----------|----------|----------|----------|
| Q : 1            | 2        | 3        | 4        | 5        |
| 0                | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10               | $\infty$ | 30       | 100      |          |
| 60               | 30       | 100      |          |          |
| 50               |          |          |          |          |

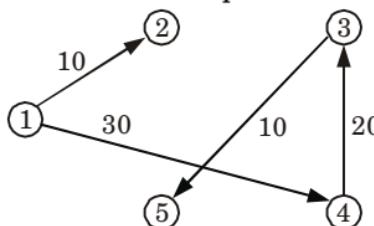
**All edges leaving (3) :**



| S : {1, 2, 4, 3, } |          |          |          |          |
|--------------------|----------|----------|----------|----------|
| Q : 1              | 2        | 3        | 4        | 5        |
| 0                  | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10                 | $\infty$ | 30       | 100      |          |
| 60                 | 30       | 100      |          |          |
| 50                 |          |          |          | 60       |

**Extract min(5) :**

| S : {1, 2, 4, 3, 5}       |          |          |          |          |
|---------------------------|----------|----------|----------|----------|
| Q : 1    2    3    4    5 |          |          |          |          |
| 0                         | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| 10                        | $\infty$ | 30       | 100      |          |
| 60                        | 30       | 100      |          |          |
| 50                        |          |          |          |          |
| 60                        |          |          |          |          |

**Shortest path**

4. Attempt any **one** part of the following : **(10 × 1 = 10)**

- a. **What is backtracking ? Discuss sum of subset problem with the help of an example.**

**Ans.** **Backtracking :**

1. Backtracking is a general algorithm for finding all solutions to some computational problems.
2. Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, and many other puzzles.

**Sum of subset problem with example :**

In the subset-sum problem we have to find a subset  $s'$  of the given set  $S = (S_1, S_2, S_3, \dots, S_n)$  where the elements of the set  $S$  are  $n$  positive integers in such a manner that  $s' \in S$  and sum of the elements of subset ' $s'$ ' is equal to some positive integer ' $X$ '.

**Algorithm for sum-subset problem :**

Subset-Sum ( $S, t$ )

1.  $C \leftarrow \emptyset$
2.  $Z \leftarrow S$
3.  $K \leftarrow \emptyset$
4.  $t_1 \leftarrow t$
5. while ( $Z \neq \emptyset$ ) do
6.      $K \leftarrow \max(Z)$
7.     if ( $K < t$ ) then
8.          $Z \leftarrow Z - K$
9.          $t_1 \leftarrow t_1 - K$
10.          $C \leftarrow C \cup K$
11.     else  $Z \leftarrow Z - K$

12. print  $C$  // Subset Sum elements whose  
// Sum is equal to  $t_1$

This procedure selects those elements of  $S$  whose sum is equal to  $t$ . Every time maximum element is found from  $S$ , if it is less than  $t$  then this element is removed from  $Z$  and also it is subtracted from  $t$ .

**For example :**

Given  $S = \langle 1, 2, 5, 7, 8, 10, 15, 20, 25 \rangle$  &  $m = 35$

$$\begin{aligned} Z &\leftarrow S, m = 35 \\ k &\leftarrow \max[Z] = 25 \\ K &< m \\ Z &= Z - K \end{aligned}$$

$$i.e., \quad Z = \langle 1, 2, 5, 7, 8, 10, 15, 20 \rangle \text{ & } m_1 \leftarrow m$$

Subtracting  $K$  from  $m_1$ , we get

$$\text{New } m_1 = m_1(\text{old}) - K = 35 - 25 = 10$$

In new step,

$$\begin{aligned} K &\leftarrow \max[Z] = 20 \\ K &> m_1 \\ i.e., \quad Z &= \langle 1, 2, 5, 7, 8, 10, 15 \rangle \end{aligned}$$

In new step,

$$\begin{aligned} K &\leftarrow \max[Z] = 15 \\ K &> m_1 \\ i.e., \quad Z &= \langle 1, 2, 5, 7, 8, 10 \rangle \end{aligned}$$

In new step,

$$\begin{aligned} K &\leftarrow \max[Z] = 10 \\ K &> m_1 \\ i.e., \quad Z &= \langle 1, 2, 5, 7, 8 \rangle \end{aligned}$$

In new step,

$$\begin{aligned} K &\leftarrow \max[Z] = 8 \\ K &> m_1 \\ i.e., \quad Z &= \langle 1, 2, 5, 7 \rangle \text{ & } m_2 \leftarrow m_1 \end{aligned}$$

$$\text{New } m_2 = m_2(\text{old}) - K = 10 - 8 = 2$$

In new step

$$\begin{aligned} K &\leftarrow \max[Z] = 7 \\ K &> m_2 \\ i.e., \quad Z &= \langle 1, 2, 5 \rangle \end{aligned}$$

In new step,  $K \leftarrow \max[Z] = 5$

$$\begin{aligned} K &> m_2 \\ i.e., \quad Z &= \langle 1, 2 \rangle \end{aligned}$$

In new step,  $K \leftarrow \max[Z] = 2$

$$\begin{aligned} K &> m_2 \\ i.e., \quad Z &= \langle 1 \rangle \end{aligned}$$

In new step,

$$\begin{aligned} K &= 1 \\ K &< m_2 \\ i.e., \quad m_3 &= 01 \end{aligned}$$

∴

Now only those numbers are needed to be selected whose sum is 01, therefore only 1 is selected from Z and rest other number found as max[Z] are subtracted from Z one by one till Z become  $\phi$ .

- b. Write down an algorithm to compute Longest Common Subsequence (LCS) of two given strings and analyze its time complexity.**

**Ans. LCS-Length (X, Y) :**

1.  $m \leftarrow \text{length}[X]$
2.  $n \leftarrow \text{length}[Y]$
3. for  $i \leftarrow 1$  to  $m$
4. do  $c[i, 0] \leftarrow 0$
5. for  $j \leftarrow 0$  to  $n$
6. do  $c[0, j] \leftarrow 0$
7. for  $i \leftarrow 1$  to  $m$
8. do for  $j \leftarrow 1$  to  $n$
9. do if  $x_i = y_j$
10. then  $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
11.  $b[i, j] \leftarrow \swarrow$
12. else if  $c[i - 1, j] \geq c[i, j - 1]$
13. then  $c[i, j] \leftarrow c[i - 1, j]$
14.  $b[i, j] \leftarrow \uparrow$
15. else  $c[i, j] \leftarrow c[i, j - 1]$
16.  $b[i, j] \leftarrow \leftarrow$
17. return  $c$  and  $b$

**Note :**

1.  $\swarrow$  means both the same.
2.  $\uparrow$  means  $c[i - 1, j] \geq c[i, j - 1]$ .
3.  $\leftarrow$  means  $c[i - 1, j] < c[i, j - 1]$ .
4. The  $\swarrow$  diagonal arrows lengthen the LCS.

**Time complexity :**

Since, two for loops are present in LCS algorithm first for loop runs upto  $m$  times and second for loop runs upto  $n$  times. So, time complexity of LCS is  $O(mn)$ .

- 5. Attempt any one part of the following : (10 × 1 = 10)**
- a. The recurrence  $T(n) = 7T(n/2) + n^2$  describe the running time of an algorithm A. A competing algorithm A' has a running time  $T'(n) = aT'(n/4) + n^2$ . What is the largest integer value for a A' is asymptotically faster than A ?**

**Ans.** Given that :

$$T(n) = 7T\left(\frac{n}{2}\right) + n^2 \quad \dots(1)$$

$$T'(n) = aT'\left(\frac{n}{4}\right) + n^2 \quad \dots(2)$$

Here, eq. (1) defines the running time for algorithm A and eq. (2) defines the running time for algorithm A'. Then for finding value of  $a$  for which A' is asymptotically faster than A we find asymptotic notation for the recurrence by using Master's method.

Now, compare eq. (1) by  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we get,

$$a = 7$$

$$b = 2$$

$$f(n) = n^2$$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

Now, apply cases of Master's theorem as :

$$\text{Case 1 : } f(n) = O(n^{\log_2 7 - E})$$

$$\Rightarrow f(n) = O(n^{2.81 - E})$$

$$\Rightarrow f(n) = O(n^{2.81 - 0.81})$$

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

Hence, case 1 of Master's theorem is satisfied.

$$\text{Thus, } T(n) = \Theta(n^{\log_b a})$$

$$\Rightarrow T(n) = \Theta(n^{2.81})$$

Since recurrence given by eq. (1) is asymptotically bounded by  $\Theta$ -notation by which is used to show optimum time we have to show that recurrence given by eq. (2) is bounded by  $\Omega$ -notation which shows minimum time (best case).

For the use satisfy the case 3 of Master theorem, let  $a = 16$

$$T'(n) = 16T'\left(\frac{n}{4}\right) + n^2$$

$$\Rightarrow a = 16$$

$$b = 4$$

$$f(n) = n^2$$

$$\Omega(n^{\log_b a + E}) = \Omega(n^{2+E})$$

Hence, case 3 of Master's theorem is satisfied.

$$\Rightarrow T(n) = \Theta(f(n))$$

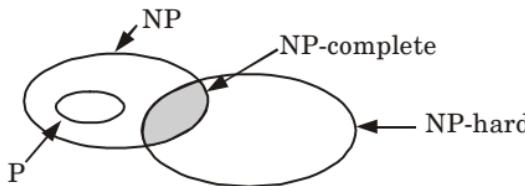
$$\Rightarrow T(n) = \Theta(n^2)$$

Therefore, this shows that A' is asymptotically faster than A when  $a = 16$ .

- b. Discuss the problem classes P, NP and NP-complete with class relationship.**

**Ans.**

- The notion of NP-hardness plays an important role in the relationship between the complexity classes  $P$  and  $NP$ .



**Fig. 5.** Relationship among P, NP, NP-complete and NP-hard problems.

2. It is also often used to define the complexity class NP-complete which is the intersection of NP and NP-hard.
3. Consequently class NP-hard can be understood as the class of problems that are NP-complete or harder.
4. There are no polynomial time algorithms for NP-hard problems.
5. A problem being in NP means that the problem is “easy” (in a certain specific sense), whereas a problem being NP-hard means that the problem is “difficult” (in another specific sense).
6. A problem being in NP and a problem being NP-hard are not mutually exclusive. When a problem is both in NP and NP-hard, we say that the problem is NP-complete.
7. All problems in NP can be solved deterministically in time  $O(2n)$ .
8. An example of an NP-hard problem is the decision problem subset-sum. Given a set of integers, does any non-empty subset of them add up to zero? i.e., a yes or no question, and happens to be NP-complete.
9. There are, also decision problems that are NP-hard but not NP-complete.
10. For example, in the halting problem “given a program and its input, will it run forever” i.e., yes or no question, so this is a decision problem. It is case to prove that the halting problem is NP-hard but not NP-complete.

6. Attempt any one part of the following :  $(10 \times 1 = 10)$

- a. Explain properties of binomial heap. Write an algorithm to perform uniting two binomial heaps. And also to find minimum key.

**Ans.** Properties of binomial heap :

1. The total number of nodes at order  $k$  are  $2^k$ .
2. The height of the tree is  $k$ .
3. There are exactly  $\binom{k}{i}$  i.e.,  ${}^k C_i$  nodes at depth  $i$  for  $i = 0, 1, \dots, k$  (this is why the tree is called a “binomial” tree).
4. Root has degree  $k$  (children) and its children are  $B_{k-1}, B_{k-2}, \dots, B_0$  from left to right.

**Algorithm for union of binomial heap :**

1. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees where roots have the same degree.

2. The following procedure links the  $B_{k-1}$  tree rooted at node to the  $B_{k-1}$  tree rooted at node  $z$ , that is, it makes  $z$  the parent of  $y$ . Node  $z$  thus becomes the root of a  $B_k$  tree.

**BINOMIAL-LINK ( $y, z$ )**

- i  $p[y] \leftarrow z$
- ii.  $sibling[y] \leftarrow child[z]$
- iii.  $child[z] \leftarrow y$
- iv.  $degree[z] \leftarrow degree[z] + 1$

3. The BINOMIAL-HEAP-UNION procedure has two phases :

- a. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps  $H_1$  and  $H_2$  into a single linked list  $H$  that is sorted by degree into monotonically increasing order.
- b. The second phase links root of equal degree until at most one root remains of each degree. Because the linked list  $H$  is sorted by degree, we can perform all the like operations quickly.

**BINOMIAL-HEAP-UNION( $H_1, H_2$ )**

1.  $H \leftarrow \text{MAKE-BINOMIAL-HEAP}()$
2.  $\text{head}[H] \leftarrow \text{BINOMIAL-HEAP-MERGE}(H_1, H_2)$
3. Free the objects  $H_1$  and  $H_2$  but not the lists they point to
4. if  $\text{head}[H] = \text{NIL}$
5. then return  $H$
6.  $\text{prev-}x \leftarrow \text{NIL}$
7.  $x \leftarrow \text{head}[H]$
8.  $\text{next-}x \leftarrow \text{sibling}[x]$
9. while  $\text{next-}x \neq \text{NIL}$
10. do if ( $\text{degree}[x] \neq \text{degree}[\text{next-}x]$ ) or  
 $(\text{sibling}[\text{next-}x] \neq \text{NIL} \text{ and } \text{degree}[\text{sibling}[\text{next-}x]] = \text{degree}[x])$
11. then  $\text{prev-}x \leftarrow x$   $\Rightarrow$  case 1 and 2
12.  $x \leftarrow \text{next-}x$   $\Rightarrow$  case 1 and 2
13. else if  $\text{key}[x] \leq \text{key}[\text{next-}x]$
14. then  $\text{ sibling}[x] \leftarrow \text{ sibling}[\text{next-}x]$   $\Rightarrow$  case 3
15.  $\text{BINOMIAL-LINK}(\text{next-}x, x)$   $\Rightarrow$  case 3
16. else if  $\text{prev-}x = \text{NIL}$   $\Rightarrow$  case 4
17. then  $\text{head}[H] \leftarrow \text{next-}x$   $\Rightarrow$  case 4
18. else  $\text{ sibling}[\text{prev-}x] \leftarrow \text{ next-}x$   $\Rightarrow$  case 4
19.  $\text{BINOMIAL-LINK}(x, \text{next-}x)$   $\Rightarrow$  case 4
20.  $x \leftarrow \text{next-}x$   $\Rightarrow$  case 4
21.  $\text{next-}x \leftarrow \text{ sibling}[x]$   $\Rightarrow$  case 4
22. return  $H$

**BINOMIAL-HEAP-MERGE( $H_1, H_2$ )**

1.  $a \leftarrow \text{head}[H_1]$
2.  $b \leftarrow \text{head}[H_2]$
3.  $\text{head}[H_1] \leftarrow \text{min-degree}(a, b)$
4. if  $\text{head}[H_1] = \text{NIL}$

5. return
6. if  $\text{head}[H_1] = b$
7. then  $b \leftarrow a$
8.  $a \leftarrow \text{head}[H_1]$
9. while  $b \neq \text{NIL}$
10. do if  $\text{sibling}[a] = \text{NIL}$
11. then  $\text{sibling}[a] \leftarrow b$
12. return
13. else if  $\text{degree}[\text{sibling}[a]] < \text{degree}[b]$
14. then  $a \leftarrow \text{sibling}[a]$
15. else  $c \leftarrow \text{sibling}[b]$
16.  $\text{sibling}[b] \leftarrow \text{sibling}[a]$
17.  $\text{sibling}[a] \leftarrow b$
18.  $a \leftarrow \text{sibling}[a]$
19.  $b \leftarrow c$

**Minimum key :****BINOMIAL-HEAP-EXTRACT-MIN ( $H$ ) :**

1. Find the root  $x$  with the minimum key in the root list of  $H$ , and remove  $x$  from the root list of  $H$ .
2.  $H' \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ .
3. Reverse the order of the linked list of  $x$ 's children, and set  $\text{head}[H']$  to point to the head of the resulting list.
4.  $H \leftarrow \text{BINOMIAL-HEAP-UNION}(H, H')$ .
5. Return  $x$

Since each of lines 1-4 takes  $O(\log n)$  time of  $H$  has  $n$  nodes, BINOMIAL- HEAP-EXTRACT-MIN runs in  $O(\log n)$  time.

- b. Given the six items in the table below and a knapsack with weight 100, what is the solution to the knapsack problem in all concepts. i.e., explain greedy all approaches and find the optimal solution.**

| Item ID | Weight | Value | Value/Weight |
|---------|--------|-------|--------------|
| A       | 100    | 40    | .4           |
| B       | 50     | 35    | .7           |
| C       | 40     | 20    | .5           |
| D       | 20     | 4     | .2           |
| E       | 10     | 10    | 1            |
| F       | 10     | 6     | .6           |

**Ans.** We can use 0/1-knapsack problem when the items cannot be divided into parts and fractional knapsack problem when the items can be divided into fractions.

First arrange in non-increasing order of value/weight :

| Item ID | Weight | Value | Value/Weight |
|---------|--------|-------|--------------|
| E       | 10     | 10    | 1            |
| B       | 50     | 35    | 0.7          |
| F       | 10     | 6     | 0.6          |
| C       | 40     | 20    | 0.5          |
| A       | 100    | 40    | 0.4          |
| D       | 20     | 4     | 0.2          |

According to 0/1-knapsack problem, either we select an item or reject. So the item will be selected according to value per weight.

E is selected  $W = 10 < 100$

B is selected  $W = 10 + 50$

$$= 60 < 100$$

F is selected  $W = 60 + 10$

$$= 70 < 100$$

C cannot be selected because

$$W = 70 + 40 = 110 > 100$$

Hence we select D

$$W = 70 + 20 = 90 < 100$$

$$\text{Total value} = 10 + 35 + 6 + 4 = 55$$

According to fractional knapsack problem, we can select fraction of any item.

E is selected  $W = 10 < 100$

B is selected  $W = 10 + 50$

$$= 60 < 100$$

F is selected  $W = 60 + 10$

$$= 70 < 100$$

If we select C  $W = 70 + 40$

$$= 110 > 100$$

Hence we select the fraction of item C as

$$\frac{100 - W}{\text{Weight of } C} = \frac{100 - 70}{40}$$

$$\text{Weight of } C = 30/40 = 0.75$$

So,  $W = 0.75 \times 40 = 30$

$$W = 70 + 30 = 100$$

$$\text{Total value} = 10 + 35 + 6 + 0.75(20)$$

$$= 10 + 35 + 6 + 15 = 66$$

7. Attempt any one part of the following : **(10 × 1 = 10)**
- a. Compute the prefix function  $\pi$  for the pattern  $P = a b a c a b$  using Knuth-Morris-Pratt algorithm. Also explain Naive string matching algorithm.

**Ans. Prefix function of the string *abacab* :**

$m \leftarrow \text{length}[P]$

$\therefore m = 6$

Initially,  $\pi[1] = 0, k = 0$

for  $q \leftarrow 2$  to 6

for  $q = 2, k > 0$

&  $P[0 + 1] \neq P[2]$

$\therefore \pi[2] = 0$

for  $q = 3, k > 0$

&  $P[0 + 1] = P[3]$

$\therefore k = k + 1 = 1$

&  $\pi[3] = 1$

for  $q = 4, k > 0$

&  $P[1 + 1] \neq P[4]$

$\therefore k \leftarrow \pi[1] = 0$

$P[1] \neq P[4]$

&  $\pi[4] = 0$

for  $q = 5, k > 0$

&  $P[0 + 1] = P[5]$

$\therefore k \leftarrow 0 + 1 = 1$

&  $\pi[5] = 1$

for  $q = 6, k > 0$

&  $P[1 + 1] = P[6]$

$\therefore k \leftarrow 1 + 1 = 2$

&  $\pi[6] = 2$

| String   | a | b | a | c | a | b |
|----------|---|---|---|---|---|---|
| $P[i]$   | 1 | 2 | 3 | 4 | 5 | 6 |
| $\pi[i]$ | 0 | 0 | 1 | 0 | 1 | 2 |

**Naive string matching algorithm :**

1. The Naive approach simply test all the possible placement of pattern  $P[1 .. m]$  relative to text  $T[1 .. n]$ . Specifically, we try shifts  $s = [0, 1, \dots, n - m]$ , successively and for each shift,  $s$ , compare  $T[s + 1 .. s + m]$  to  $P[1 .. m]$ .
2. The Naive string matching procedure can be interpreted graphically as a sliding a pattern  $P[1 .. m]$  over the text  $T[1 .. m]$  and noting for which shift all of the characters in the pattern match the corresponding characters in the text.

**Naive string matcher ( $T, P$ )**

1.  $n \leftarrow \text{length}[T]$
2.  $m \leftarrow \text{length}[P]$
3. for  $s \leftarrow 0$  to  $n - m$  do
4.      $j \leftarrow 1$
5.     while  $j \leq m$  and  $T[s + j] = P[j]$  do

6.  $j \leftarrow j + 1$
7. if  $j > m$  then
8. return valid shift  $s$
9. return no valid shift exist // i.e., there is no substring of  $T$  matching  $P$ .

**b. Explain approximation and randomized algorithms.**

**Ans. Approximation algorithm :**

1. An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.
2. The best of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.
3. Let  $c(i)$  be the cost of solution produced by approximate algorithm and  $c^*(i)$  be the cost of optimal solution for some optimization problem instance  $i$ .
4. For minimization and maximization problem, we are interested in finding a solution of a given instance  $i$  in the set of feasible solutions, such that  $c(i)/c^*(i)$  and  $c^*(i)/c(i)$  be as small as possible respectively.
5. We say that an approximation algorithm for the given problem instance  $i$ , has a ratio bound of  $p(n)$  if for any input of size  $n$ , the cost  $c$  of the solution produced by the approximation algorithm is within a factor of  $p(n)$  of the cost  $c^*$  of an optimal solution. That is  $\max(c(i) / c^*(i), c^*(i) / c(i)) \leq p(n)$

The definition applies for both minimization and maximization problems.

6.  $p(n)$  is always greater than or equal to 1. If solution produced by approximation algorithm is true optimal solution then clearly we have  $p(n) = 1$ .

**Randomized algorithm :**

1. A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased bits and it is then allowed to use these random bits to influence its computation.
2. An algorithm is randomized if its output is determined by the input as well as the values produced by a random number generator.
3. A randomized algorithm makes use of a randomizer such as a random number generator.
4. The execution time of a randomized algorithm could also vary from run to run for the same input.
5. The algorithm typically uses the random bits as an auxiliary input to guide its behaviour in the hope of achieving good performance in the “average case”.
6. Randomized algorithms are particularly useful when it faces a malicious attacker who deliberately tries to feed a bad input to the algorithm.



**B.Tech.**  
**(SEM. V) ODD SEMESTER THEORY  
EXAMINATION, 2018-19**

**DESIGN AND ANALYSIS OF ALGORITHM**

---

**Time : 3 Hours**

**Max. Marks : 70**

---

- Note :** 1. Attempt all sections. If require any missing data; then choose suitable.  
2. Any special paper specific instruction.

**Section-A**

1. Attempt all questions in brief : **(2 × 10 = 20)**
- a. Rank the following by growth rate :  
 $n, 2 \log \sqrt{n}, \log n, \log(\log n), \log^2 n, (\log n)^{\log n}, 4, (3/2)^n, n!$
- b. Prove that if  $n >= 1$ , then for any  $n$ -key B-tree of height  $h$  and minimum degree  $t >= 2$ ,  $h <= \log_t ((n + 1)/2)$ .
- c. Define principle of optimality. When and how dynamic programming is applicable ?
- d. Explain application of graph colouring problem.
- e. Compare adjacency matrix and linked adjacency lists representation of graph with suitable example / diagram.
- f. What are approximation algorithms ? What is meant by  $p(n)$  approximation algorithms ?
- g. What do you mean by stability of a sorting algorithm ? Explain its application.

**SECTION-B**

2. Attempt any three of the following : **(7 × 3 = 21)**
- 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 a constant.

- b. Define BNP, NP hard and NP complete problems. Prove that travelling salesman problem is NP complete.
- c. Consider the weight and values of item listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by  $V_{\text{opt}}$ . A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by  $V_{\text{greedy}}$ . Find the value of  $V_{\text{opt}} - V_{\text{greedy}}$ .

| Item | $I_1$ | $I_2$ | $I_3$ | $I_4$ |
|------|-------|-------|-------|-------|
| W    | 10    | 7     | 4     | 2     |
| V    | 60    | 28    | 20    | 24    |

- d. Insert the following key in a 2-3-4 B-tree :  
 40, 35, 22, 90, 12, 45, 58, 78, 67, 60 and then delete key 35 and 22 one after other.
- e. Prove that the weights on the edge of the connected undirected graph are distinct then there is a unique minimum spanning tree. Give an example in this regard. Also discuss prim's minimum spanning tree algorithm in detail.

### SECTION-C

3. Attempt any one part of the following :  $(10 \times 1 = 10)$
- a. The recurrence  $T(n) = 7T(n/3) + n^2$  describes the running time of an algorithm A. Another competing algorithm B has a running time of  $S(n) = a S(n/9) + n^2$ . What is the smallest value of  $a$  such that B is asymptotically faster than A ?
- b. How will you sort following array A of element using heap sort :  $A = (23, 9, 18, 45, 5, 9, 1, 17, 6)$
4. Attempt any one part of the following :  $(10 \times 1 = 10)$
- a. Explain the different conditions of getting union of two existing binomial heaps. Also write algorithm for union of two binomial heaps. What is its complexity ?

- b. Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black tree and delete 12, 4, 9, 14 respectively.
5. Attempt any one part of the following :  $(10 \times 1 = 10)$
- a. When do Dijkstra and the Bellman Ford algorithm both fail to find a shortest path ? Can Bellman Ford detect all negative weight cycles in a graph ? Apply Bellman Ford algorithm on the following graph :

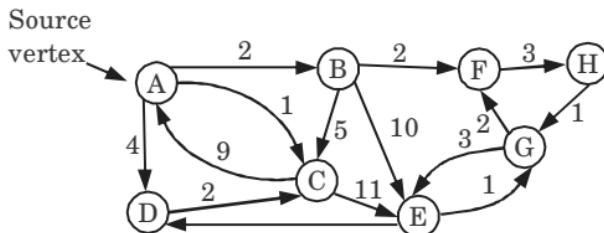


Fig. 1.

- b. Given an integer  $x$  and a positive number  $n$ , use divide and conquer approach to write a function that computes  $x^n$  with time complexity  $O(\log n)$ .
6. Attempt any one part of the following :  $(10 \times 1 = 10)$
- a. Solve the subset sum problem using backtracking, where  $n = 4$ ,  $m = 18$ ,  $w[4] = \{5, 10, 8, 13\}$ .
- b. Give Floyd-Warshall algorithm to find the shortest path for all pairs of vertices in a graph. Give the complexity of the algorithm. Explain with example.
7. Attempt any one part of the following :  $(10 \times 1 = 10)$
- a. What is the application of Fast Fourier Transform (FFT) ? Also write the recursive algorithm for FFT.
- b. Give a linear time algorithm to determine if a text  $T$  is a cycle rotation of another string  $T$ . For example : RAJA and JARA are cyclic rotations of each other.



## SOLUTION OF PAPER (2018-19)

- Note :**
1. Attempt all sections. If require any missing data; then choose suitable.
  2. Any special paper specific instruction.

### Section-A

1. Attempt all questions in brief : **(2 × 10 = 20)**

- a. Rank the following by growth rate :

$$n, 2 \log \sqrt{n}, \log n, \log(\log n), \log^2 n, (\log n)^{\log n}, 4, (3/2)^n, n!$$

**Ans.** Rank in increasing order of growth rate is given as :

$$4, \log n, \log(\log n), \log^2 n, (\log n)\log n, \log n, 2 \log \sqrt{n}, n, n!, \left(\frac{3}{2}\right)^n$$

- b. Prove that if  $n >= 1$ , then for any  $n$ -key B-tree of height  $h$  and minimum degree  $t >= 2$ ,  $h <= \log_t((n+1)/2)$ .

**Ans.** Proof :

1. The root contains at least one key.
2. All other nodes contain at least  $t - 1$  keys.
3. There are at least 2 nodes at depth 1, at least  $2t$  nodes at depth 2, at least  $2t^{i-1}$  nodes at depth  $i$  and  $2t^{h-1}$  nodes at depth  $h$ .

$$n \geq 1 + (t-1) \sum_{i=1}^h 2t^{i-1} = 1 + 2(t-1) \left( \frac{t^h - 1}{t-1} \right) = 2t^h - 1$$

4. So  $t^h \leq (n+1)/2$  as required.

Taking log both sides we get,

$$h \leq \log_t(n+1)/2$$

- c. Define principle of optimality. When and how dynamic programming is applicable ?

**Ans.** **Principle of optimality :** Principle of optimality states that in an optimal sequence of decisions or choices, each subsequence must also be optimal.

Dynamic programming is mainly applicable where the solution of one sub-problem is needed repeatedly. In this procedure, the solutions of sub-problems are stored in a table, so that there is no need to re-compute the sub-problems and can be directly accessed from the table if required.

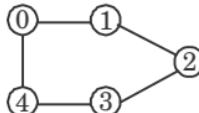
- d. Explain application of graph colouring problem.

**Ans.** Application of graph colouring problem :

- 1. Sudoku :** Sudoku is a variation of graph colouring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.
- 2. Register allocation :** In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph colouring problem.
- 3. Bipartite graphs :** We can check if a graph is bipartite or not by colouring the graph using two colours. If a given graph is 2-colourable, then it is bipartite, otherwise not.
- 4. Map colouring :** Geographical maps of countries or states where no two adjacent cities cannot be assigned same colour.
- e. Compare adjacency matrix and linked adjacency lists representation of graph with suitable example / diagram.**

**Ans.**

| S. No. | Adjacency matrix                                                                                | Linked adjacency list                                                                      |
|--------|-------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| 1.     | An adjacency matrix is a square matrix used to represent a finite graph.                        | Linked adjacency list is a collection of unordered lists used to represent a finite graph. |
| 2.     | The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. | Each list describes the set of adjacent vertices in the graph.                             |
| 3.     | Space complexity in the worst case is $O( V ^2)$ .                                              | Space complexity in the worst case is $O( V  +  E )$ .                                     |

**For example :** Consider the graph :**Using adjacency matrix :**

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 1 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 1 & 0 & 0 & 1 \end{bmatrix}$$

**Using adjacency list :**

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | → | 1 | → | 4 | / |
| 1 | → | 0 | / | 2 | / |
| 2 | → | 1 | → | 3 | / |
| 3 | → | 4 | / | 2 | / |
| 4 | → | 3 | / | 0 | / |

- f. What are approximation algorithms ? What is meant by  $p(n)$  approximation algorithms ?**

**Ans.** **Approximation algorithm :** An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.

**$p(n)$  approximation algorithm :** A is a  $p(n)$  approximate algorithm if and only if for every instance of size  $n$ , the algorithm achieves an approximation ratio of  $p(n)$ . It is applied to both maximization ( $0 < C(i) \leq C^*(i)$ ) and minimization ( $0 < C^*(i) \leq C(i)$ ) problem because of the maximization factor and costs are positive.  $p(n)$  is always greater than 1.

- g. What do you mean by stability of a sorting algorithm ? Explain its application.**

**Ans.** **Stability of a sorting algorithm :** Let  $A$  be an array, and let  $<$  be a strict weak ordering on the elements of  $A$ .

Sorting algorithm is stable if :

$i < j$  and  $A[i] \equiv A[j]$  i.e.,  $A[i]$  comes before  $A[j]$ .

Stability means that equivalent elements retain their relative positions, after sorting.

**Application :** One application for stable sorting algorithms is sorting a list using a primary and secondary key. For example, suppose we wish to sort a hand of cards such that the suits are in the order clubs, diamonds, hearts, spades and within each suit, the cards are sorted by rank. This can be done by first sorting the cards by rank (using any sort), and then doing a stable sort by suit.

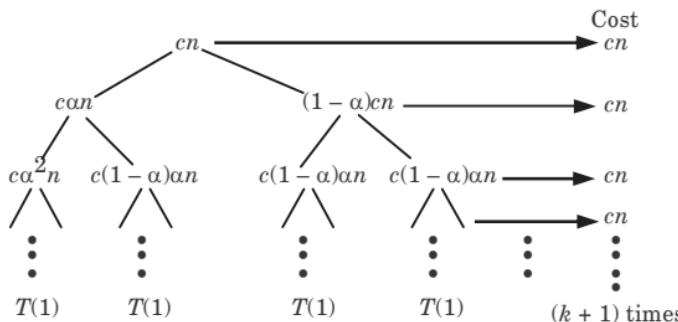
## SECTION-B

- 2. Attempt any three of the following :  $(7 \times 3 = 21)$**

- 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 a constant.**

**Ans.**  $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$

**Recursion tree :**



Assuming  $T(1) = 1$

So  $c(1 - \alpha)k n = 1$

$$cn = \frac{1}{(1 - \alpha)^k} = \left( \frac{1}{1 - \alpha} \right)^k$$

$$\log(n) = k \log\left(\frac{1}{1-\alpha}\right)$$

$$k = \frac{\log cn}{\log\left(\frac{1}{1-\alpha}\right)} = \log_{\frac{1}{1-\alpha}}(cn)$$

So,      Total cost =  $cn + cn + \dots$  ( $k+1$ ) times =  $cn(k+1)$   
 $= cn \times \log_{\frac{1}{1-\alpha}}(cn)$

$$\text{Time complexity} = O\left(n \log_{\frac{1}{1-\alpha}}(cn)\right) = O\left(n \log_{\frac{1}{1-\alpha}} n\right)$$

**b. Define BNP, NP hard and NP complete problems. Prove that travelling salesman problem is NP complete.**

**Ans.** **P (BNP)** : Class P are the problems which can be solved in polynomial time, which take time like  $O(n)$ ,  $O(n^2)$ ,  $O(n^3)$ .

**Example :** Finding maximum element in an array or to check whether a string is palindrome or not. So, there are many problems which can be solved in polynomial time.

**NP** : Class NP are the problems which cannot be solved in polynomial time like TSP (travelling salesman problem).

**Example :** Subset sum problem is best example of NP in which given a set of numbers, does there exist a subset whose sum is zero, but NP problems are checkable in polynomial time means that given a solution of a problem, we can check that whether the solution is correct or not in polynomial time.

**NP-complete** : The group of problems which are both in NP and NP-hard are known as NP-complete problem.

Now suppose we have a NP-complete problem  $R$  and it is reducible to  $Q$  then  $Q$  is at least as hard as  $R$  and since  $R$  is an NP-hard problem, therefore  $Q$  will also be at least NP-hard, it may be NP-complete also.

**NP hard and NP complete :**

1. We say that a decision problem  $P_i$  is NP-hard if every problem in NP is polynomial time reducible to  $P_i$ .
2. In symbols,  
 $P_i$  is NP-hard if, for every  $P_j \in \text{NP}$ ,  $P_j \xrightarrow{\text{Poly}} P_i$ .
3. This does not require  $P_i$  to be in NP.
4. Highly informally, it means that  $P_i$  is 'as hard as' all the problem in NP.
5. If  $P_i$  can be solved in polynomial time, then all problems in NP.
6. Existence of a polynomial time algorithm for an NP-hard problem implies the existence of polynomial solution for every problem in NP.

**NP-complete problem :**

1. There are many problems for which no polynomial time algorithms is known.
2. Some of these problems are travelling salesman problem, optimal graph colouring, the Knapsack problem, Hamiltonian cycles, integer programming, finding the longest simple path in a graph, and satisfying a Boolean formula.
3. These problems belongs to an interesting class of problems called the “NP-complete” problems, whose status is unknown.
4. The NP-complete problems are traceable *i.e.*, require a super polynomial time.

**Proof :****Part 1 : TSP is in NP.****Proof :**

1. Let a hint  $S$  be a sequence of vertices  $V = v_1, \dots, v_n$ .
2. We then check two things :
  - a. First we check that every edge traversed by adjacent vertices is an edge in  $G$ , such that the sum of these edge weights is less than or equal to  $k$ .
  - b. Secondly we check that every vertex in  $G$  is in  $V$ , which assures that every node has been traversed.
3. We accept  $S$  if and only if  $S$  satisfies these two questions, otherwise reject.
4. Both of these checks are clearly polynomial, thus our algorithm forms a verifier with hint  $S$ , and TSP is consequently in NP.

**Part 2 : TSP is NP-Hard.****Proof :**

1. To show that TSP is NP-Hard, we must show that every problem  $y$  in NP reduces to TSP in polynomial time.
2. To do this, consider the decision version of Hamiltonian Cycle (HC).
3. Take  $G = (V, E)$ , set all edge weights equal to 1, and let  $k = |V| = n$ , that is,  $k$  equals the number of nodes in  $G$ .
4. Any edge not originally in  $G$  then receives a weight of 2 (traditionally TSP is on a complete graph, so we need to add in these extra edges).
5. Then pass this modified graph into TSP, asking if there exists a tour on  $G$  with cost at most  $k$ . If the answer to TSP is YES, then HC is YES. Likewise if TSP is NO, then HC is NO.

**First direction :** HC has a YES answer  $\Rightarrow$  TSP has a YES answer.**Proof :**

1. If HC has a YES answer, then there exists a simple cycle  $C$  that visits every node exactly once, thus  $C$  has  $n$  edges.
2. Since every edge has weight 1 in the corresponding TSP instance for the edges that are in the HC graph, there is a Tour of weight  $n$ . Since  $k = n$ , and given that there is a tour of weight  $n$ , it follows that TSP has a YES answer.

**Second direction :** HC has a NO answer  $\Rightarrow$  TSP has a NO answer.

**Proof:**

1. If HC has a NO answer, then there does not exist a simple cycle  $C$  in  $G$  that visits every vertex exactly once. Now suppose TSP has a YES answer.
2. Then there is a tour that visits every vertex once with weight at most  $k$ .
3. Since the tour requires every node be traversed, there are  $n$  edges, and since  $k = n$ , every edge traversed must have weight 1, implying that these edges are in the HC graph. Then take this tour and traverse the same edges in the HC instance. This forms a Hamiltonian Cycle, a contradiction.

This concludes Part 2. Since we have shown that TSP is both in NP and NP-Hard, we have that TSP is NP-Complete.

- c. Consider the weight and values of item listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by  $V_{\text{opt}}$ . A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by  $V_{\text{greedy}}$ . Find the value of  $V_{\text{opt}} - V_{\text{greedy}}$ .

| Item | $I_1$ | $I_2$ | $I_3$ | $I_4$ |
|------|-------|-------|-------|-------|
| $W$  | 10    | 7     | 4     | 2     |
| $V$  | 60    | 28    | 20    | 24    |

**Ans. For  $V_{\text{greedy}}$ :**

| Item  | $W$ | $V$ |
|-------|-----|-----|
| $I_1$ | 10  | 60  |
| $I_2$ | 7   | 28  |
| $I_3$ | 4   | 20  |
| $I_4$ | 2   | 24  |

Arrange the items by  $V/W$  ratio in descending order :

| Item  | $W$ | $V$ | $V/W$ |
|-------|-----|-----|-------|
| $I_4$ | 2   | 24  | 12    |
| $I_1$ | 10  | 60  | 6     |
| $I_3$ | 4   | 20  | 5     |
| $I_2$ | 7   | 28  | 4     |

Total weight

$$W = 11 \text{ kg}$$

$I_4$  is picked so  $W = 11 - 2 = 9 \text{ kg}$

$I_1$  cannot be picked  $10 > 9$

$I_3$  is picked,  $W = 9 - 4 = 5 \text{ kg}$

$I_2$  cannot be picked  $7 > 5$

$I_4$  and  $I_3$  are picked so

$$V_{\text{greedy}} = V(I_4) + V(I_3) = 24 + 20 = 44$$

**For  $V_{\text{opt}}$ :** For calculating  $V_{\text{opt}}$  we use 0/1 knapsack problem, so only item 1 is picked. Hence,

$$V_{\text{opt}} = 60$$

So,

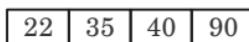
$$V_{\text{opt}} - V_{\text{greedy}} = 60 - 44 = 16$$

**d. Insert the following key in a 2-3-4 B-tree :**

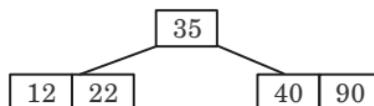
**40, 35, 22, 90, 12, 45, 58, 78, 67, 60 and then delete key 35 and 22 one after other.**

**Ans.** In 2-3-4 B-trees, non-leaf node can have minimum 2 keys and maximum 4 keys so the order of tree is 5.

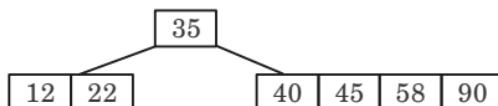
**Insert 40, 35, 22, 90 :**



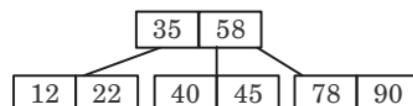
**Insert 12 :**



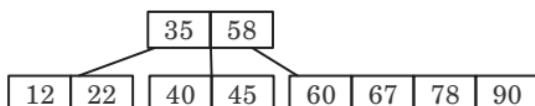
**Insert 45, 58 :**



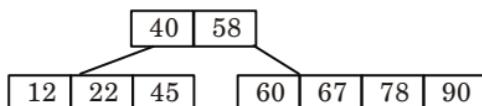
**Insert 78 :**



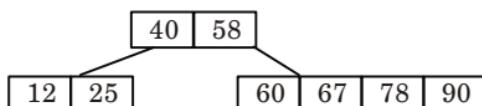
**Insert 67, 60 :**



**Delete 35 :**



**Delete 22 :**



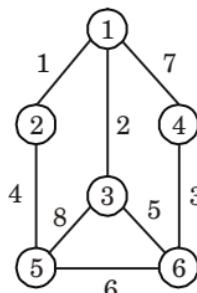
- e. Prove that the weights on the edge of the connected undirected graph are distinct then there is a unique minimum spanning tree. Give an example in this regard. Also discuss prim's minimum spanning tree algorithm in detail.

**Ans. Proof :**

- Let we have an algorithm that finds an MST (which we will call A) based on the structure of the graph and the order of the edges when ordered by weight.
- Assume MST A is not unique.
- There is another spanning tree with equal weight, say MST B.
- Let  $e_1$  be an edge that is in A but not in B.
- Then, B should include at least one edge  $e_2$  that is not in A.
- Assume the weight of  $e_1$  is less than that of  $e_2$ .
- As B is a MST,  $\{e_1\} \cup B$  must contain a cycle.
- Replace  $e_2$  with  $e_1$  in B yields the spanning tree  $\{e_1\} \cup B - \{e_2\}$  which has a smaller weight compared to B.
- This contradicts that B is not a MST.

So, MST of undirected graph with distinct edge is unique.

**Example :**

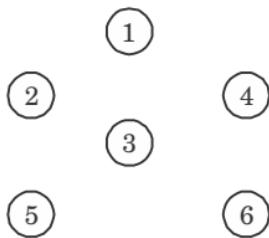


**Fig. 1.**

**Step 1 :** Arrange the edge of graph according to weight in ascending order.

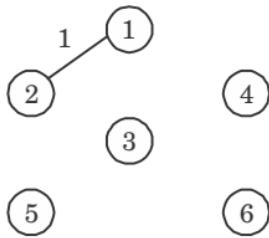
| Edges | Weight | Edge | Weight |
|-------|--------|------|--------|
| 12    | 1      | 14   | 7      |
| 13    | 2      | 35   | 8      |
| 46    | 3      |      |        |
| 25    | 4      |      |        |
| 36    | 5      |      |        |
| 56    | 6      |      |        |

**Step 2 :** Now draw the vertices as given in graph,

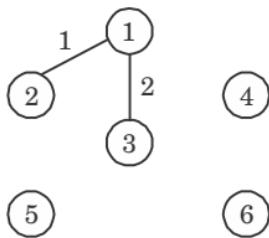


Now draw the edge according to the ascending order of weight. If any edge forms cycle, leave that edge.

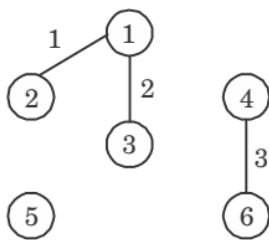
**Step 3 :**



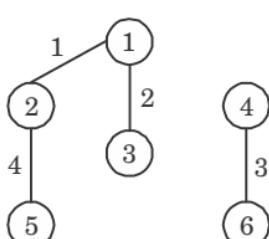
**Step 4 :**

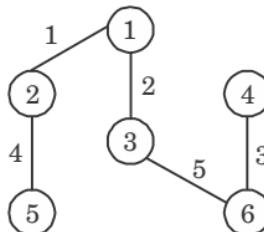


**Step 5 :**



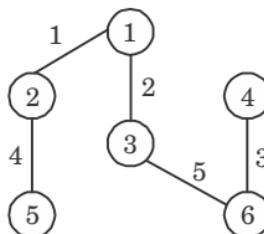
**Step 6 :**



**Step 7 :**

All the remaining edges, such as : 14, 35, 56 are rejected because they form cycle.

All the vertices are covered in this tree. So, the final tree with minimum cost of given graph is

**Prim's algorithm :**

First it chooses a vertex and then chooses an edge with smallest weight incident on that vertex. The algorithm involves following steps :

**Step 1 :** Choose any vertex  $V_1$  of  $G$ .

**Step 2 :** Choose an edge  $e_1 = V_1V_2$  of  $G$  such that  $V_2 \neq V_1$  and  $e_1$  has smallest weight among the edge  $e$  of  $G$  incident with  $V_1$ .

**Step 3 :** If edges  $e_1, e_2, \dots, e_i$  have been chosen involving end points  $V_1, V_2, \dots, V_{i+1}$ , choose an edge  $e_{i+1} = V_jV_k$  with  $V_j \in \{V_1, \dots, V_{i+1}\}$  and  $V_k \notin \{V_1, \dots, V_{i+1}\}$  such that  $e_{i+1}$  has smallest weight among the edges of  $G$  with precisely one end in  $\{V_1, \dots, V_{i+1}\}$ .

**Step 4 :** Stop after  $n - 1$  edges have been chosen. Otherwise goto step 3.

**SECTION-C**

3. Attempt any **one** part of the following :  $(10 \times 1 = 10)$

a. The recurrence  $T(n) = 7T(n/3) + n^2$  describes the running time of an algorithm A. Another competing algorithm B has a running time of  $S(n) = a S(n/9) + n^2$ . What is the smallest value of  $a$  such that B is asymptotically faster than A ?

**Ans.** Given that :

$$T(n) = 7T\left(\frac{n}{3}\right) + n^2 \quad \dots(1)$$

$$S'(n) = aS'\left(\frac{n}{9}\right) + n^2 \quad \dots(2)$$

Here, eq. (1) defines the running time for algorithm A and eq. (2) defines the running time for algorithm B. Then for finding value of  $a$  for which B is asymptotically faster than A we find asymptotic notation for the recurrence by using Master's method.

Now, compare eq. ) with  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we get,  $a = 7, b = 3$   
 $f(n) = n^2$

$$n^{\log_b a} = n^{\log_3 7} = n^{2.81}$$

Now, apply cases of Master's theorem as :

Case 3 :  $f(n) = O(n^{\log_3 7 + \epsilon})$   
 $\Rightarrow f(n) = O(n^{1.77 + \epsilon})$   
 $\Rightarrow f(n) = O(n^{1.77 + 0.23})$   
 $\Rightarrow f(n) = O(n^2)$

Hence, case 3 of Master's theorem is satisfied.

Thus,  $T(n) = \theta(f(n))$   
 $\Rightarrow T(n) = \theta(n^2)$

Since recurrence (1) is asymptotically bounded by  $\theta$ -notation which is used to show optimum time we have to show that recurrence given by eq. (2) is bounded by  $\Omega$ -notation which shows minimum time (best case).

For the use satisfy the case 2 of Master theorem, Guess  $a = 81$

$$S'(n) = f(n) = 81 S'\left(\frac{n}{9}\right) + n^2$$

$\Rightarrow a = 81, b = 9$   
 $f(n) = n^{\log_9 81}$   
 $f(n) = \Omega(n^{\log_b a}) = \Omega(n^2)$

Hence, case 2 of Master's theorem is satisfied.

$\Rightarrow T(n) = \theta(n^{\log_9 81} \log n)$   
 $\Rightarrow T(n) = \theta(n^2 \log n)$

Therefore, this shows that B is asymptotically faster than A when  $a = 81$ .

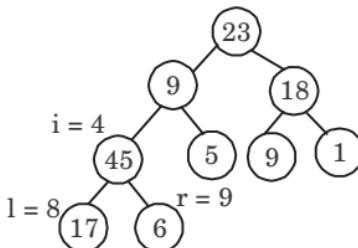
- b. How will you sort following array A of element using heap sort :  $A = (23, 9, 18, 45, 5, 9, 1, 17, 6)$ .

**Ans:** Given array : 

|    |   |    |    |   |   |   |    |   |
|----|---|----|----|---|---|---|----|---|
| 23 | 9 | 18 | 45 | 5 | 9 | 1 | 17 | 6 |
|----|---|----|----|---|---|---|----|---|

First we call Build-Max heap

heap size  $[A] = 9$



so  $i = 4$  to 1 call MAX HEAPIFY ( $A, i$ )

i.e., first we call MAX HEAPIFY ( $A, 4$ )

$$A[l] = 7, A[i] = A[4] = 45, A[r] = 6$$

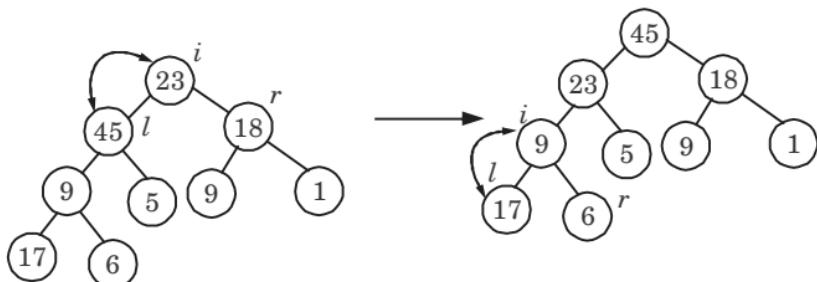
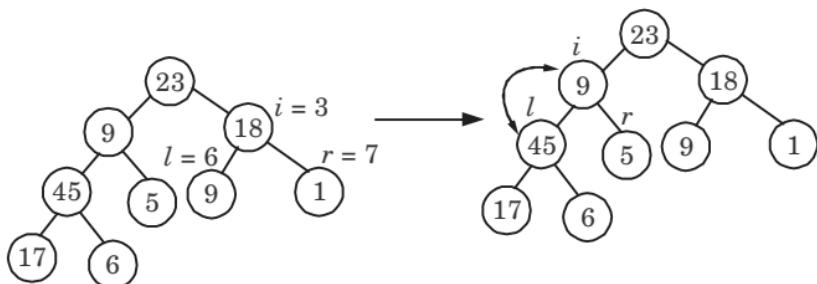
$$l \leftarrow \text{left}[4] = 2 \times 4 = 8$$

$$r \leftarrow \text{right}[4] = 2 \times 4 + 1 = 9$$

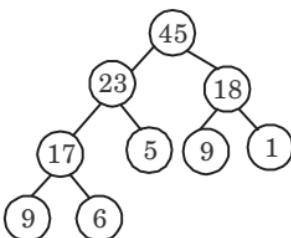
$$8 \leq 9 \text{ and } A[8] = 17 < 45 \text{ (False)}$$

Then, largest  $\leftarrow 4$ .

Similarly for  $i = 3, 2, 1$  we get the following heap tree :

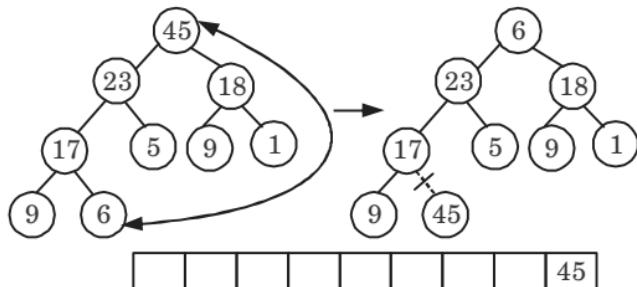


So, final tree after Build-Max heap is

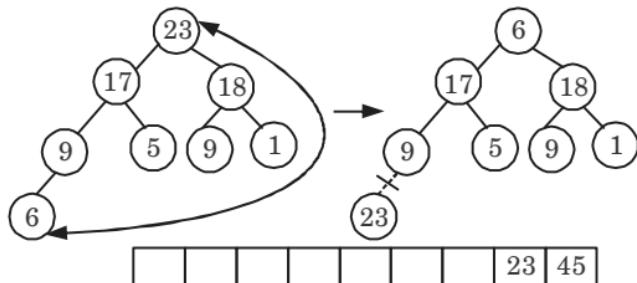


Now  $i = 9$  down to 2 and size  $= 10 - 1 = 9$  and call MAX HEAPIFY ( $A, 1$ ) each time

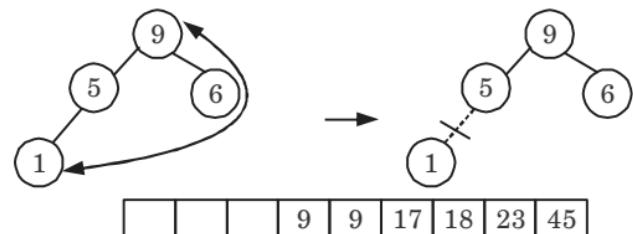
Exchanging  $A[1] \leftrightarrow A[9]$



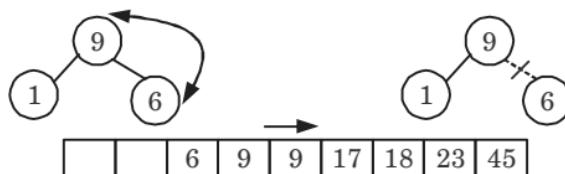
Now call MAX HEAPIFY ( $A, 1$ ) and  
Exchange  $A[1]$  and  $A[8]$ , size =  $9 - 1 = 8$



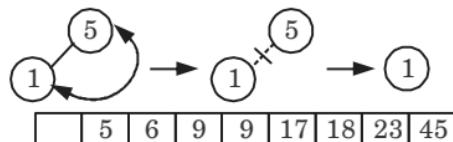
Now, call MAX HEAPIFY ( $A, 1$ ), exchange  $A[1]$  and  $A[4]$  and size =  $5 - 1 = 4$



Now, call MAX HEAPIFY ( $A, 1$ ), exchange  $A[1]$  and  $A[3]$  and size =  $4 - 1 = 3$



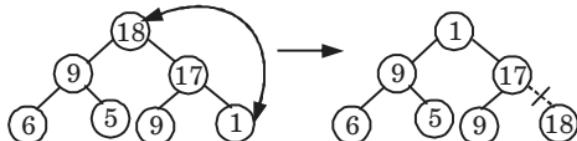
Exchange  $A[1]$  and  $A[2]$  size  $3 - 1 = 2$



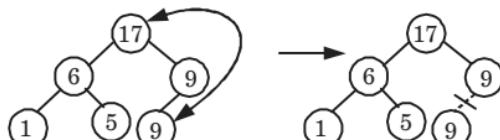
The sorted array

|   |   |   |   |   |    |    |    |    |
|---|---|---|---|---|----|----|----|----|
| 1 | 5 | 6 | 9 | 9 | 17 | 18 | 23 | 45 |
|---|---|---|---|---|----|----|----|----|

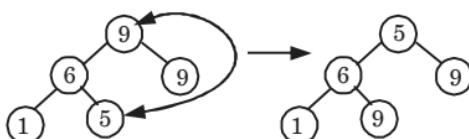
Now call MAX HEAPIFY ( $A$ , 1) and exchange  $A[1]$  and  $A[7]$



Now call MAX HEAPIFY ( $A$ , 1) and size =  $7 - 1 = 6$  exchange  $A[1]$  and  $A[6]$



Exchange  $A[1]$  and  $A[5]$  and size =  $6 - 1 = 5$

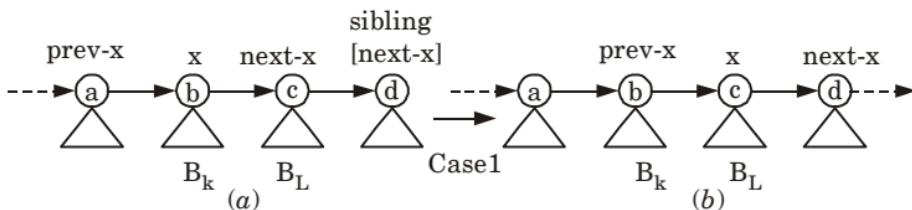


4. Attempt any one part of the following :  $(10 \times 1 = 10)$

- a. Explain the different conditions of getting union of two existing binomial heaps. Also write algorithm for union of two binomial heaps. What is its complexity ?

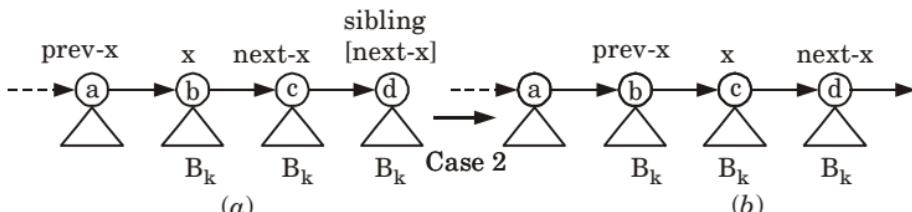
**Ans.** There are four cases/conditions that occur while performing union on binomial heaps.

**Case 1 :** When  $\text{degree}[x] \neq \text{degree}[\text{next}-x] = \text{degree}[\text{sibling}[\text{next}-x]]$ , then pointers moves one position further down the root list.



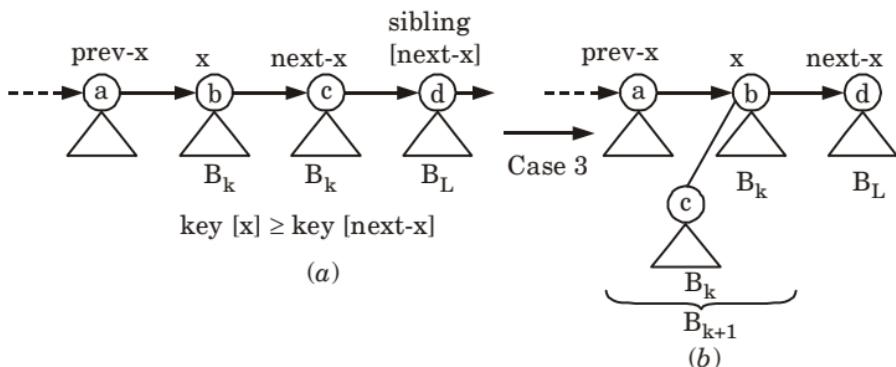
**Fig. 2.**

**Case 2 :** It occurs when  $x$  is the first of three roots of equal degree, that is,  $\text{degree}[x] = \text{degree}[\text{next}-x] = \text{degree}[\text{sibling}[\text{next}-x]]$ , then again pointer move one position further down the list, and next iteration executes either case 3 or case 4.



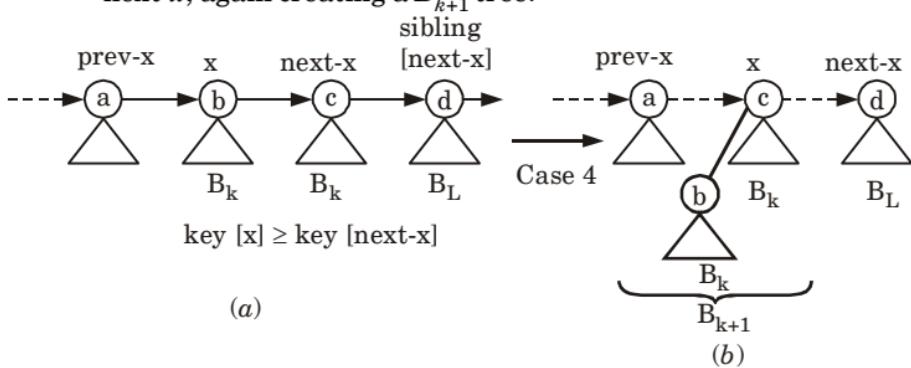
**Fig. 3.**

**Case 3 :** If  $\text{degree}[x] = \text{degree}[\text{next-}x] \neq \text{degree}[\text{sibling}[\text{next-}x]]$  and  $\text{key}[x] \leq \text{key}[\text{next-}x]$ , we remove next- $x$  from the root list and link it to  $x$ , creating  $B_{k+1}$  tree.



**Fig. 4.**

**Case 4 :**  $\text{degree}[x] = \text{degree}[\text{next-}x] \neq \text{degree}[\text{sibling}[\text{next-}x]]$  and  $\text{key}[\text{next-}x] \leq \text{key } x$ , we remove  $x$  from the root list and link it to  $\text{next-}x$ , again creating a  $B_{k+1}$  tree.



**Fig. 5.**

#### Algorithm for union of binomial heap :

1. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees where roots have the same degree.
2. The following procedure links the  $B_{k-1}$  tree rooted at node  $y$  to the  $B_{k-1}$  tree rooted at node  $z$ , that is, it makes  $z$  the parent of  $y$ . Node  $z$  thus becomes the root of a  $B_k$  tree.

#### BINOMIAL-LINK ( $y, z$ )

- i.  $p[y] \leftarrow z$
- ii.  $\text{sibling}[y] \leftarrow \text{child}[z]$
- iii.  $\text{child}[z] \leftarrow y$
- iv.  $\text{degree}[z] \leftarrow \text{degree}[z] + 1$

3. The BINOMIAL-HEAP-UNION procedure has two phases :
  - a. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps  $H_1$  and  $H_2$

- into a single linked list  $H$  that is sorted by degree into monotonically increasing order.
- The second phase links root of equal degree until at most one root remains of each degree. Because the linked list  $H$  is sorted by degree, we can perform all the like operations quickly.

**BINOMIAL-HEAP-UNION( $H_1, H_2$ )**

- $H \leftarrow \text{MAKE-BINOMIAL-HEAP}()$
- $\text{head}[H] \leftarrow \text{BINOMIAL-HEAP-MERGE}(H_1, H_2)$
- Free the objects  $H_1$  and  $H_2$  but not the lists they point to
- if  $\text{head}[H] = \text{NIL}$
- then return  $H$
- $\text{prev-}x \leftarrow \text{NIL}$
- $x \leftarrow \text{head}[H]$
- $\text{next-}x \leftarrow \text{ sibling}[x]$
- while  $\text{next-}x \neq \text{NIL}$
- do if ( $\text{degree}[x] \neq \text{degree}[\text{next-}x]$ ) or  
( $\text{sibling}[\text{next-}x] \neq \text{NIL}$  and  $\text{degree}[\text{sibling}[\text{next-}x]] = \text{degree}[x]$ )
- then  $\text{prev-}x \leftarrow x$  ⇒ case 1 and 2
- $x \leftarrow \text{next-}x$  ⇒ case 1 and 2
- else if  $\text{key}[x] \leq \text{key}[\text{next-}x]$
- then  $\text{ sibling}[x] \leftarrow \text{ sibling}[\text{next-}x]$  ⇒ case 3
- $\text{BINOMIAL-LINK}(\text{next-}x, x)$  ⇒ case 3
- else if  $\text{prev-}x = \text{NIL}$  ⇒ case 4
- then  $\text{head}[H] \leftarrow \text{next-}x$  ⇒ case 4
- else  $\text{ sibling}[\text{prev-}x] \leftarrow \text{next-}x$  ⇒ case 4
- $\text{BINOMIAL-LINK}(x, \text{next-}x)$  ⇒ case 4
- $x \leftarrow \text{next-}x$  ⇒ case 4
- $\text{next-}x \leftarrow \text{ sibling}[x]$  ⇒ case 4
- return  $H$

**BINOMIAL-HEAP-MERGE( $H_1, H_2$ )**

- $a \leftarrow \text{head}[H_1]$
- $b \leftarrow \text{head}[H_2]$
- $\text{head}[H_1] \leftarrow \text{min-degree}(a, b)$
- if  $\text{head}[H_1] = \text{NIL}$
- return
- if  $\text{head}[H_1] = b$
- then  $b \leftarrow a$
- $a \leftarrow \text{head}[H_1]$
- while  $b \neq \text{NIL}$
- do if  $\text{ sibling}[a] = \text{NIL}$
- then  $\text{ sibling}[a] \leftarrow b$
- return
- else if  $\text{degree}[\text{ sibling}[a]] < \text{degree}[b]$
- then  $a \leftarrow \text{ sibling}[a]$
- else  $c \leftarrow \text{ sibling}[b]$

16.  $\text{sibling}[b] \leftarrow \text{sibling}[a]$

17.  $\text{sibling}[a] \leftarrow b$

18.  $a \leftarrow \text{sibling}[a]$

19.  $b \leftarrow c$

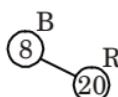
Time complexity of union of two binomial heap is  $O(\log n)$ .

- b. Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black tree and delete 12, 4, 9, 14 respectively.**

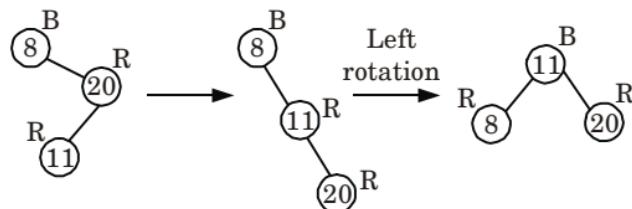
**Ans. Insert 8 :**



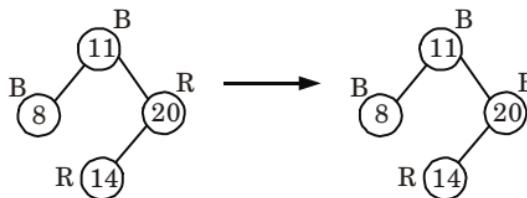
**Insert 20 :**



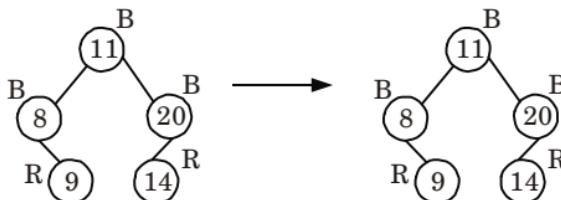
**Insert 11 :** Since, parent of node 11 is red. Check the colour of uncle of node 11. Since uncle of node 11 is nil than do rotation and recolouring.



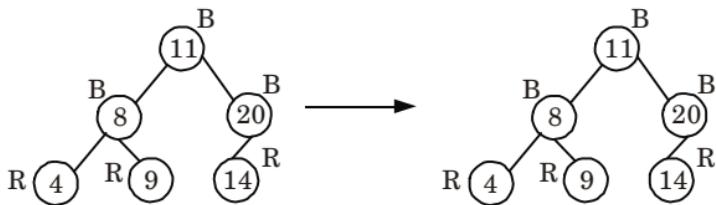
**Insert 14 :** Uncle of node 14 is red. Recolour the parent of node 14 i.e., 20 and uncle of node 14 i.e., 8. No rotation is required.



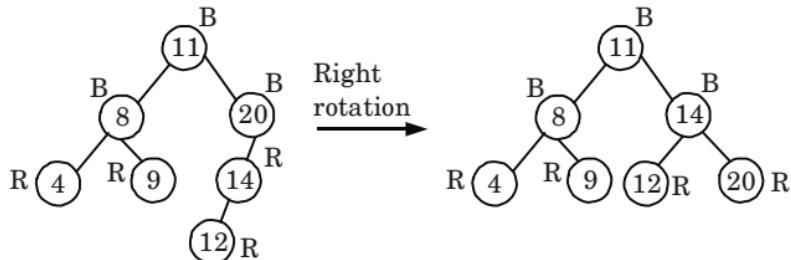
**Insert 9 :** Parent of node 9 is black. So no rotation and no recolouring.



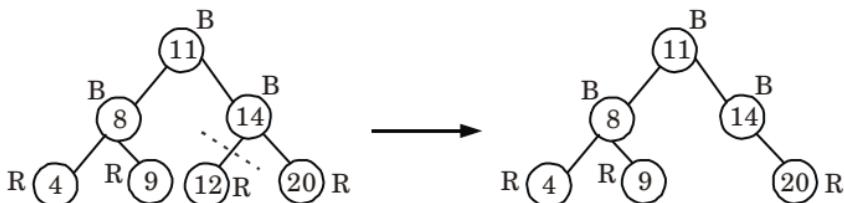
**Insert 4 :** Parent of node 4 is black. So no rotation and no recolouring.



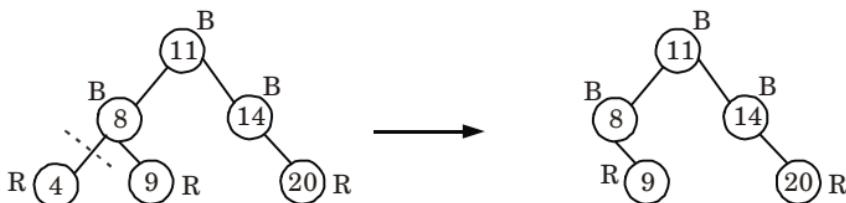
**Insert 12 :** Parent of node 12 is red. Check the colour of uncle of node 12, which is nil. So do rotation and recolouring.



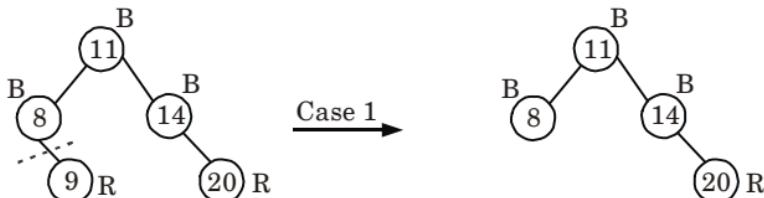
**Delete 12 :** Node 12 is red and leaf node. So simply delete node 12.



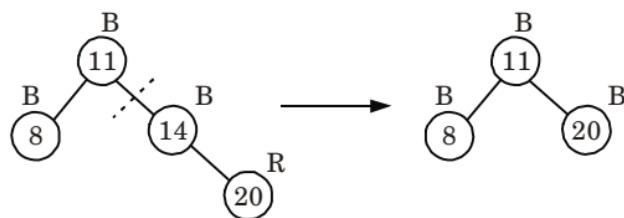
**Delete 4 :** Node 4 is red and leaf node. So simply delete node 4.



**Delete 9 :** Node 9 is red and leaf node. So simply delete node 9.

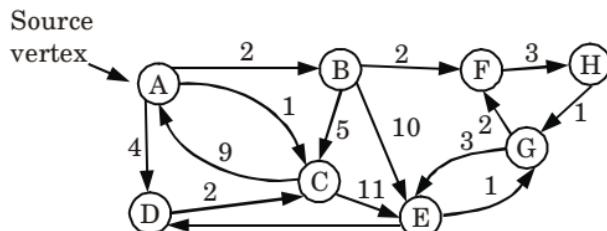


**Delete 14 :** Node 14 is internal node replace node 14 with node 20 and do not change the colour.



5. Attempt any **one** part of the following : **(10 × 1 = 10)**

- a. When do Dijkstra and the Bellman Ford algorithm both fail to find a shortest path ? Can Bellman Ford detect all negative weight cycles in a graph ? Apply Bellman Ford algorithm on the following graph :

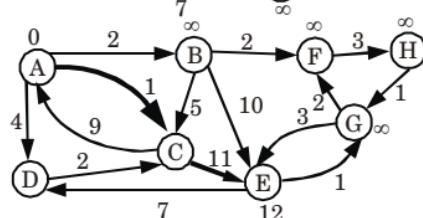
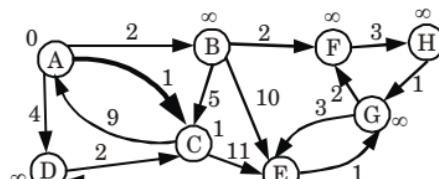
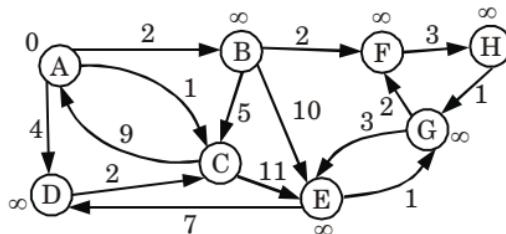


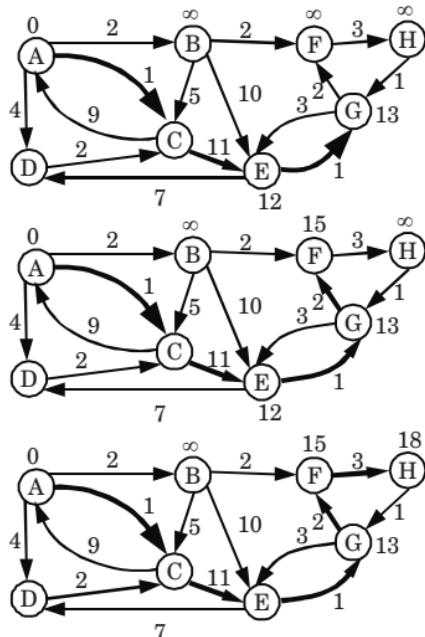
**Ans.** Dijkstra algorithm fails to find a shortest path when the graph contains negative edges.

Bellman Ford algorithm fails to find a shortest path when the graph contain negative weight cycle.

No, Bellman Ford cannot detect all negative weight cycle in a graph.

**Numerical :**





- b. Given an integer  $x$  and a positive number  $n$ , use divide and conquer approach to write a function that computes  $x^n$  with time complexity  $O(\log n)$ .

**Ans.** Function to calculate  $x^n$  with time complexity  $O(\log n)$  :

```
int power(int x, unsigned int y)
{
 int temp;
 if(y == 0)
 return 1;
 temp = power(x, y/2);
 if (y%2 == 0)
 return temp * temp;
 else
 return x * temp * temp;
}
```

6. Attempt any one part of the following :  $(10 \times 1 = 10)$

- a. Solve the subset sum problem using backtracking, where  $n = 4, m = 18, w[4] = \{5, 10, 8, 13\}$ .

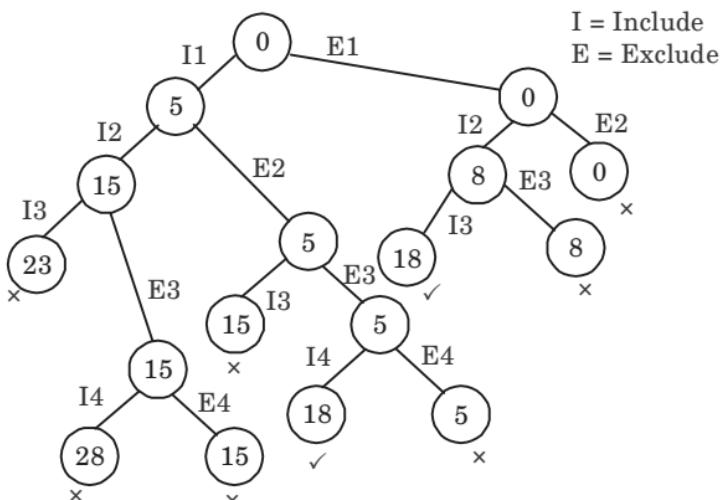
**Ans.**

$$\begin{aligned}n &= 4 \\m &= 18\end{aligned}$$

$$w[4] = \{5, 10, 8, 13\}$$

Sorted order :  $w[4] = \{5, 8, 10, 13\}$

Now we construct state-space tree.



First Subset  $S_1 = \{5, 13\}$

Similarly,  $S_2 = \{8, 10\}$

- b. Give Floyd-Warshall algorithm to find the shortest path for all pairs of vertices in a graph. Give the complexity of the algorithm. Explain with example.

**Ans. Floyd-Warshall algorithm :**

1. Floyd-Warshall algorithm is a graph analysis algorithm for finding shortest paths in a weighted, directed graph.
2. A single execution of the algorithm will find the shortest path between all pairs of vertices.
3. The algorithm considers the “intermediate” vertices of a shortest path, where an intermediate vertex of a simple path  $p = (v_1, v_2, \dots, v_m)$  is any vertex of  $p$  other than  $v_1$  or  $v_m$ , that is, any vertex in the set  $\{v_2, v_3, \dots, v_{m-1}\}$ .
4. Let the vertices of  $G$  be  $V = \{1, 2, \dots, n\}$ , and consider a subset  $\{1, 2, \dots, k\}$  of vertices for some  $k$ .
5. For any pair of vertices  $i, j \in V$ , consider all paths from  $i$  to  $j$  whose intermediate vertices are all drawn from  $\{1, 2, \dots, k\}$ , and let  $p$  be a minimum-weight path from among them.
6. Let  $d_{ij}^{(k)}$  be the weight of a shortest path from vertex  $i$  to vertex  $j$  with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ .

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

**Floyd-Warshall ( $W$ ) :**

1.  $n \leftarrow \text{rows } [W]$
2.  $D^{(0)} \leftarrow W$
3. for  $k \leftarrow 1$  to  $n$
4. do for  $i \leftarrow 1$  to  $n$

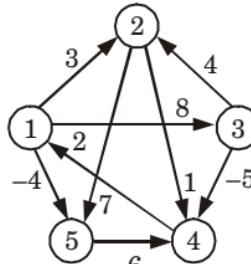
5. do for  $j \leftarrow 1$  to  $n$

6.       $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

7. return  $D^{(n)}$

Time complexity of Floyd-Warshall algorithm is  $O(n^3)$ .

**Example :**



**Fig. 6.**

$$d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(0)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(1)} = \begin{bmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(2)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}; \pi^{(3)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}; \pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}; \pi^{(5)} = \begin{bmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

7. Attempt any **one** part of the following : **(10 × 1 = 10)**

- a. **What is the application of Fast Fourier Transform (FFT) ?  
Also write the recursive algorithm for FFT.**

**Ans.** **Application of Fast Fourier Transform :**

1. Signal processing.
2. Image processing.
3. Fast multiplication of large integers.
4. Solving Poisson's equation nearly optimally.

**Recursive algorithm :**

1. The Fast Fourier Transform (FFT) is a algorithm that computes a Discrete Fourier Transform (DFT) of  $n$ -length vector in  $O(n \log n)$  time.
2. In the FFT algorithm, we apply the divide and conquer approach to polynomial evaluation by observing that if  $n$  is even, we can divide a degree  $(n - 1)$  polynomial.

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

into two degree  $\left(\frac{n}{2} - 1\right)$  polynomials.

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \dots + a_{n-2}x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \dots + a_{n-1}x^{n/2-1}$$

Where  $A^{[0]}$  contains all the even index coefficients of  $A$  and  $A^{[1]}$  contains all the odd index coefficients and we can combine these two polynomials into  $A$ , using the equation,

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \quad \dots(1)$$

**FFT ( $a, w$ ) :**

1.  $n \leftarrow \text{length } [a]$        $n$  is a power of 2.
2. if  $n = 1$
3. then return  $a$
4.  $\omega_n \leftarrow e^{2\pi i/n}$
5.  $x \leftarrow \omega^0$        $x$  will store powers of  $\omega$  initially  $x = 1$ .

6.  $a^{[0]} \leftarrow (a_0, a_2, \dots, a_{n-2})$
7.  $a^{[1]} \leftarrow (a_1, a_3, \dots, a_{\frac{n}{2}-1})$
8.  $y^{[0]} \leftarrow \text{FFT}(a^{[0]}, \omega^2)$  Recursive calls with  $\omega^2$  as  $(n/2)^{\text{th}}$  root of unity.
9.  $y^{[1]} \leftarrow \text{FFT}(a^{[0]}, \omega^2)$
10. for  $k \leftarrow 0$  to  $(n/2) - 1$
11. do  $y_k \leftarrow y_k^{[0]} + x y_k^{[1]}$
12.  $y_{k+(n/2)} \leftarrow y_k^{[0]} - x y_k^{[1]}$
13.  $x \leftarrow x \omega_n$
14. return  $y$

Line 2-3 represents the basis of recursion; the DFT of one element is the element itself. Since in this case

$$y_0 = a_0 \omega_1^0 = a_0 1 = a_0$$

Line 6-7 defines the recursive coefficient vectors for the polynomials  $A^{[0]}$  and  $A^{[1]}$ .  $\omega = \omega_n^k$

Line 8-9 perform the recursive DFT <sub>$n/2$</sub>  computations setting for

$$k = 0, 1, 2, \dots, \frac{n}{2} - 1 \text{ i.e.,}$$

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k}), \quad y_k^{[1]} = A^{[1]}(\omega_n^{2k})$$

Lines 11-12 combine the results of the recursive DFT <sub>$n/2$</sub>  calculations.

For  $y_0, y_2, \dots, y_{(n/2)-1}$ , line 11 yields.

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) = A(\omega_n^k) \end{aligned} \quad \text{using equation (1)}$$

For  $y_{n/2}, y_{(n/2)+1} \dots y_{n-1}$ , line 12 yields.

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} = y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \quad [\because \omega_n^{k+(n/2)} = -\omega_n^k] \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k} \omega_n^n) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k} \omega_n^n) \quad [\because \omega_n^n = 1] \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}) \end{aligned} \quad \text{using equation (1)}$$

$$\text{each } k = 0, 1, 2, \dots, (n/2) - 1.$$

Thus, the vector  $y$  returned by the FFT algorithm will store the values of  $A(x)$  at each of the roots of unity.

- b. Give a linear time algorithm to determine if a text  $T$  is a cycle rotation of another string  $T$ . For example : RAJA and JARA are cyclic rotations of each other.

- Ans.** Knuth-Morris-Pratt algorithm is used to determine if a text  $T$  is a cycle rotation of another string  $T'$ .

**Knuth-Morris-Pratt algorithm for string matching :**

**COMPUTE-PREFIX-FUNCTION ( $P$ )**

1.  $m \leftarrow \text{length } [P]$
2.  $\pi [1] \leftarrow 0$
3.  $k \leftarrow 0$
4. for  $q \leftarrow 2$  to  $m$ 
  - 5. do while  $k > 0$  and  $P [k + 1] \neq P[q]$ 
    - 6. do  $k \leftarrow \pi [k]$
    - 7. if  $P[k + 1] = P[q]$
    - 8. then  $k \leftarrow k + 1$
    - 9.  $\pi [q] \leftarrow k$
  - 10. return  $\pi$

KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute  $\pi$ .

**KMP-MATCHER ( $T, p$ )**

1.  $n \leftarrow \text{length } [T]$
2.  $m \leftarrow \text{length } [P]$
3.  $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION } (P)$
4.  $q \leftarrow 0$
5. for  $i \leftarrow 1$  to  $n$ 
  - 6. do while  $q > 0$  and  $P [q + 1] \neq T[i]$ 
    - 7. do  $q \leftarrow \pi [q]$
    - 8. if  $P [q + 1] = T[i]$
    - 9. then  $q \leftarrow q + 1$
  - 10. if  $q = m$
  - 11. then print "pattern occurs with shift"  $i - m$
  - 12.  $q \leftarrow \pi [q]$



**B. Tech.****(SEM. V) ODD SEMESTER THEORY  
EXAMINATION, 2019-20****DESIGN AND ANALYSIS OF ALGORITHMS**

---

**Time : 3 Hours****Max. Marks : 100**

---

**Note :** Attempt all sections. If require any missing data; then choose suitably.

**Section-A**

1. Attempt all questions in brief. **(2 × 7 = 14)**
- a. How do you compare the performance of various algorithms ?
- b. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function  $g(n)$  immediately follows function  $f(n)$  in your list, then it should be the case that  $f(n)$  is  $O(g(n))$ .  
 $f_1(n) = n^{2.5}$ ,  $f_2(n) = \sqrt{2^n}$ ,  $f_3(n) = n + 10$ ,  $f_4(n) = 10n$ ,  $f_5(n) = 100n$ , and  $f_6(n) = n^2 \log n$
- c. What is advantage of binary search over linear search? Also, state limitations of binary search.
- d. What are greedy algorithms ? Explain their characteristics ?
- e. Explain applications of FFT.
- f. Define feasible and optimal solution.
- g. What do you mean by polynomial time reduction ?

**Section-B**

2. Attempt any three of the following : **(7 × 3 = 21)**
- a. i. Solve the recurrence  $T(n) = 2T(n/2) + n^2 + 2n + 1$
- ii. Prove that worst case running time of any comparison sort is  $\Omega(n \log n)$ .
- b. Insert the following element in an initially empty RB-Tree. 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now delete 23 and 81.

- c. Define spanning tree. Write Kruskal's algorithm or finding minimum cost spanning tree. Describe how Kruskal's algorithm is different from Prim's algorithm for finding minimum cost spanning tree.
- d. What is dynamic programming ? How is this approach different from recursion? Explain with example.
- e. Define NP-hard and NP-complete problems. What are the steps involved in proving a problem NP-complete ? Specify the problems already proved to be NP-complete.

### Section-C

- 3. Attempt any **one** part of the following :  $(7 \times 1 = 7)$ 
  - a. Among Merge sort, Insertion sort and quick sort which sorting technique is the best in worst case. Apply the best one among these algorithms to sort the list  $E, X, A, M, P, L, E$  in alphabetic order.
  - b. Solve the recurrence using recursion tree method :  

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$
- 4. Attempt any **one** part of the following :  $(7 \times 1 = 7)$ 
  - a. Using minimum degree ' $t$ ' as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.
  - b. Explain the algorithm to delete a given element in a binomial heap. Give an example for the same.
- 5. Attempt any **one** part of the following :  $(7 \times 1 = 7)$ 
  - a. Compare the various programming paradigms such as divide-and-conquer, dynamic programming and greedy approach.
  - b. What do you mean by convex hull ? Describe an algorithm that solves the convex hull problem. Find the time complexity of the algorithm.
- 6. Attempt any **one** part of the following :
  - a. Solve the following 0/1 knapsack problem using dynamic programming  $P = \{11, 21, 31, 33\}$   $w = \{2, 11, 22, 15\}$   $c = 40$ ,  $n = 4$ .

- b. Define Floyd Warshall algorithm for all pair shortest path and apply the same on following graph :

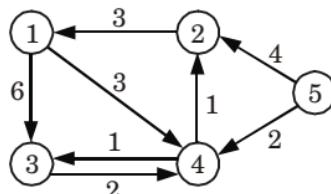


Fig. 1.

7. Attempt any one part of the following :  $(7 \times 1 = 7)$
- a. Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function  $\pi$  for the pattern  $ababbabbabbabbabbabb$  when the alphabet is  $\Sigma = \{a, b\}$ .
- b. What is an approximation algorithm ? What is meant by  $p(n)$  approximation algorithms ? Discuss approximation algorithm for Travelling Salesman Problem.



## SOLUTION OF PAPER (2019-20)

**Note:** Attempt all sections. If require any missing data; then choose suitably.

### Section-A

1. Attempt all questions in brief. **(2 × 7 = 14)**

**a. How do you compare the performance of various algorithms ?**

**Ans.** To compare the performance of various algorithms first we measure its performance which depends on the time taken and the size of the problem. For this we measure the time and space complexity of various algorithms which is divided into different cases such as worst case, average case and best case.

- b. Take the following list of functions and arrange them in ascending order of growth rate. That is, if function  $g(n)$  immediately follows function  $f(n)$  in your list, then it should be the case that  $f(n) = O(g(n))$ .

$$f_1(n) = n^{2.5}, f_2(n) = \sqrt[4]{2^n}, f_3(n) = n + 10, f_4(n) = 10n, f_5(n) = 100n, \text{ and } f_6(n) = n^2 \log n$$

**Ans.**  $f_3(n) = f_4(n) = f_5(n) < f_2(n) < f_6(n) < f_1(n)$

- c. What is advantage of binary search over linear search? Also, state limitations of binary search.

**Ans.** Advantages of binary search over linear search :

1. Input data needs to be sorted in binary search but not in linear search.
2. Linear search does the sequential access whereas binary search access data randomly.
3. Time complexity of linear search is  $O(n)$  where binary search has time complexity  $O(\log n)$ .

**Limitation of binary search :**

1. List must be sorted.
2. It is more complicated to implement and test.

- d. What are greedy algorithms ? Explain their characteristics ?

**Ans.** Greedy algorithms : Greedy algorithms are shortsighted in their approach in the sense that they take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.

**Characteristics of greedy algorithm :**

1. Greedy algorithms are most efficient.
2. For every instance of input greedy algorithms makes a decision and continues to process further set of input.

3. The other input values at the instance of decision are not used in further processing.

**e. Explain applications of FFT.**

**Ans. Application of Fast Fourier Transform :**

1. Signal processing.
2. Image processing.
3. Fast multiplication of large integers.
4. Solving Poisson's equation nearly optimally.

**f. Define feasible and optimal solution.**

**Ans. Feasible solution :** A feasible solution is a set of values for the decision variables that satisfies all of the constraints in an optimization problem. The set of all feasible solutions defines the feasible region of the problem.

**Optimal solution :** An optimal solution is a feasible solution where the objective function reaches its maximum (or minimum) value.

**g. What do you mean by polynomial time reduction ?**

**Ans.** A polynomial time reduction is a method for solving one problem using another. For example, if a hypothetical subroutine solving the second problem exists, then the first problem can be solved by transforming or reducing it to inputs for the second problem and calling the subroutine one or more times.

## Section-B

2. Attempt any three of the following :  $(7 \times 3 = 21)$

a. i. **Solve the recurrence  $T(n) = 2T(n/2) + n^2 + 2n + 1$**

ii. **Prove that worst case running time of any comparison sort is  $\Omega(n \log n)$ .**

**Ans.**

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

Compare it with  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

we have,  $a = 2, b = 2, f(n) = n^2$

Now, we apply cases for Master's theorem.

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

This satisfies case 3 of Master's theorem.

$$\begin{aligned} \Rightarrow f(n) &= \Omega(n^{\log_b a + E}) = \Omega(n^{1+E}) \\ &= \Omega(n^{1+1}) \\ &= \Omega(n^2) \end{aligned}$$

where  $E = 1$

$$\text{Again } 2f\left(\frac{n^2}{2}\right) \leq c f(n^2) \quad \dots(1)$$

eq. (1) is true for  $c = 2$

$$\Rightarrow T(n) = \theta(f(n))$$

$$\Rightarrow T(n) = \theta(f(n^2))$$

- ii. Let  $T(n)$  be the time taken by merge sort to sort any array of  $n$  elements.

$$\text{Therefore, } T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \left(\left\lceil \frac{n}{2} \right\rceil\right) + g(n)$$

$$\text{where } g(n) \in \Theta(n)$$

This recurrence, which becomes :

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

when  $n$  is even is a special case of our general analysis for divide-and conquer algorithms.

Compare the above given recurrence with  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$\text{we get } a = 2$$

$$b = 2$$

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

$$\text{Now we find, } n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

$$\Rightarrow f(n) = \Theta(n)$$

i.e., case 2 of Master's theorem applied then

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

$$\Rightarrow T(n) = \Omega(n \log n)$$

Hence, the worst case running time of merge sort is  $\Omega(n \log n)$ .

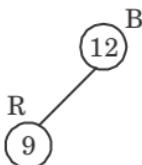
- b. Insert the following element in an initially empty RB-Tree. 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now delete 23 and 81.

**Ans.**

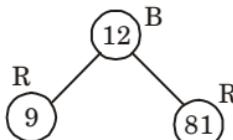
Insert 12 :

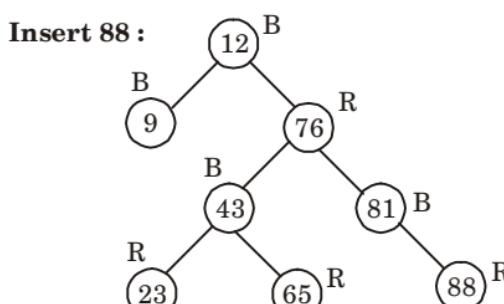
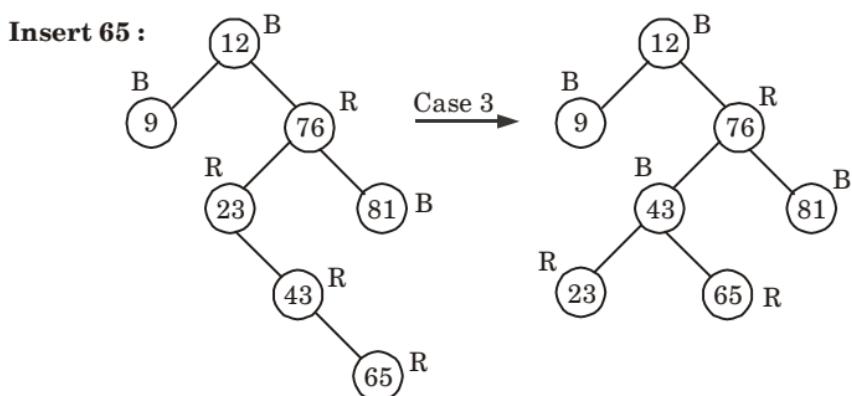
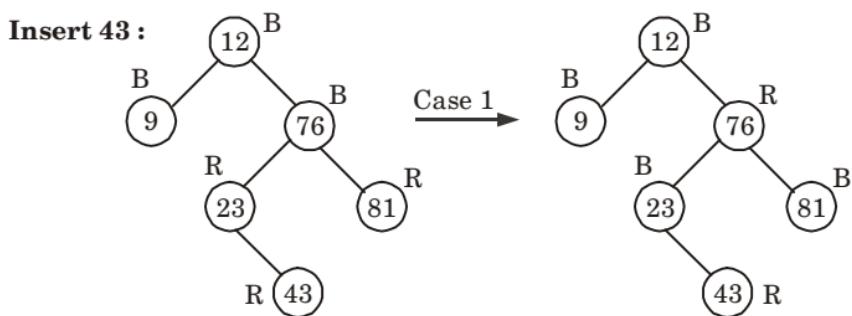
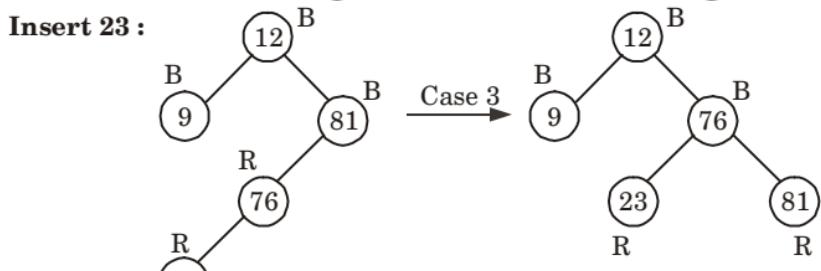
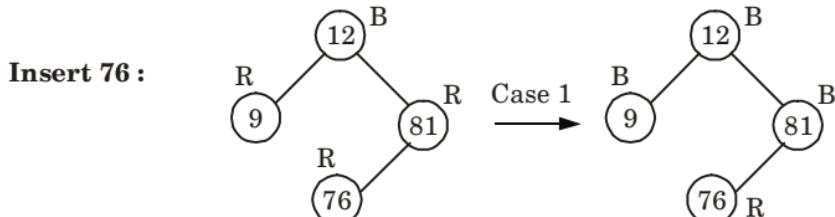


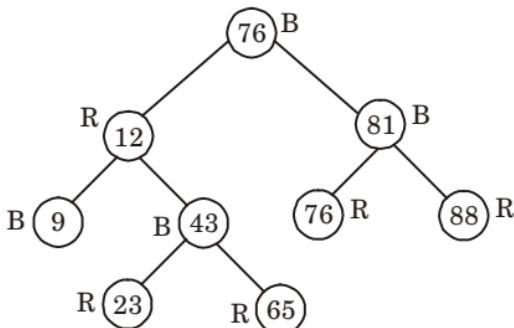
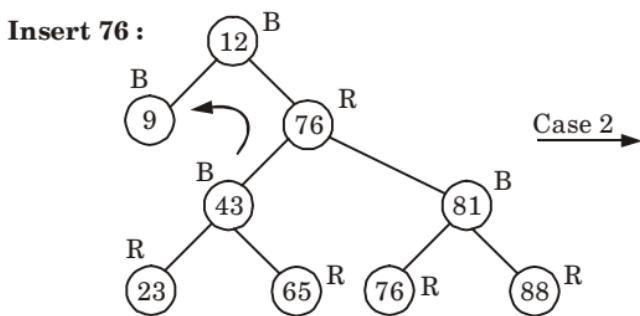
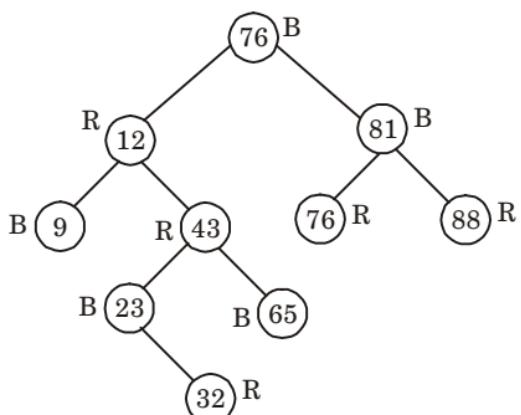
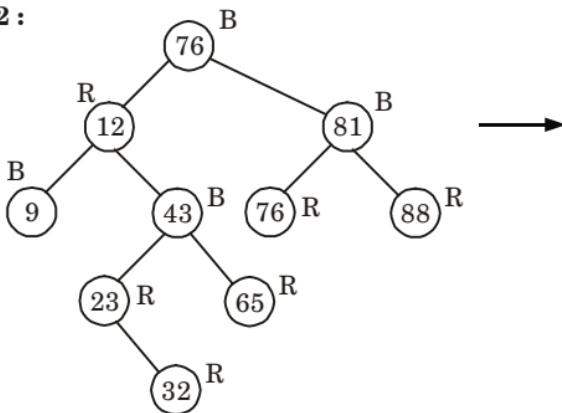
Insert 9 :

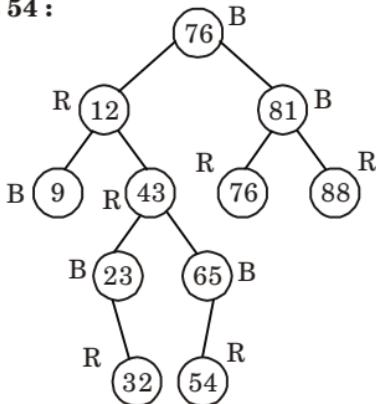
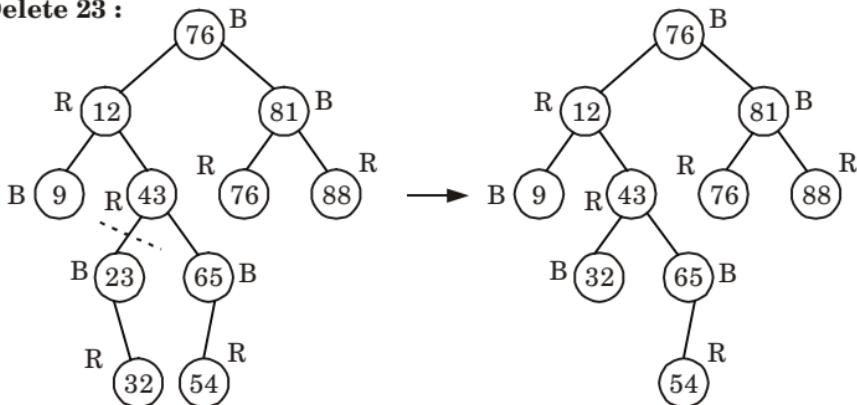
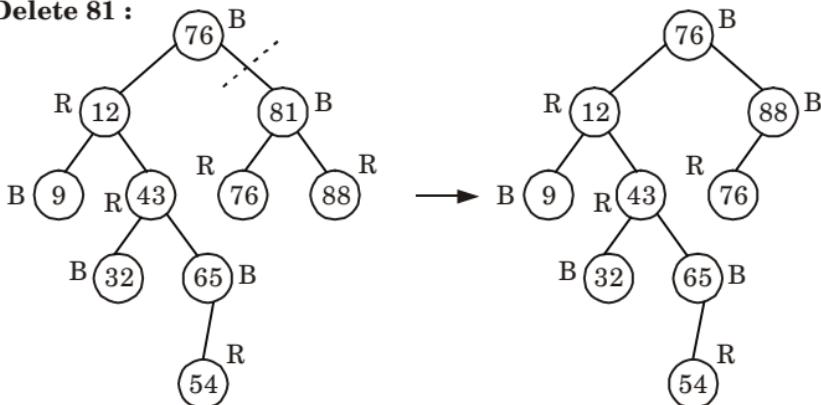


Insert 81 :





**Insert 76 :****Insert 32 :**

**Insert 54 :****Delete 23 :****Delete 81 :**

- c. Define spanning tree. Write Kruskal's algorithm or finding minimum cost spanning tree. Describe how Kruskal's algorithm is different from Prim's algorithm for finding minimum cost spanning tree.

**Ans.** Spanning tree :

1. A spanning tree of a graph is a subgraph that contains all the vertices and is a tree.

2. A spanning tree of a connected graph  $G$  contains all the vertices and has the edges which connect all the vertices. So, the number of edges will be 1 less than the number of nodes.
3. If graph is not connected, i.e., a graph with  $n$  vertices has edges less than  $n - 1$  then no spanning tree is possible.
4. A graph may have many spanning trees.

**Kruskal's algorithm :**

- i. In this algorithm, we choose an edge of  $G$  which has smallest weight among the edges of  $G$  which are not loops.
- ii. This algorithm gives an acyclic subgraph  $T$  of  $G$  and the theorem given below proves that  $T$  is minimal spanning tree of  $G$ . Following steps are required :

**Step 1 :** Choose  $e_1$ , an edge of  $G$ , such that weight of  $e_1$ ,  $w(e_1)$  is as small as possible and  $e_1$  is not a loop.

**Step 2 :** If edges  $e_1, e_2, \dots, e_i$  have been selected then choose an edge  $e_{i+1}$  not already chosen such that

- i. the induced subgraph

$G[\{e_1, \dots, e_{i+1}\}]$  is acyclic and

- ii.  $w(e_{i+1})$  is as small as possible

**Step 3 :** If  $G$  has  $n$  vertices, stop after  $n - 1$  edges have been chosen. Otherwise repeat step 2.

If  $G$  be a weighted connected graph in which the weight of the edges are all non-negative numbers, let  $T$  be a subgraph of  $G$  obtained by Kruskal's algorithm then,  $T$  is minimal spanning tree.

**Difference :**

| S. No. | Kruskal's algorithm                                                                                         | Prim's algorithm                                                        |
|--------|-------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------|
| 1.     | Kruskal's algorithm initiates with an edge.                                                                 | Prim's algorithm initializes with a node.                               |
| 2.     | Kruskal's algorithm selects the edges in a way that the position of the edge is not based on the last step. | Prim's algorithms span from one node to another.                        |
| 3.     | Kruskal's can be used on disconnected graphs.                                                               | In Prim's algorithm, graph must be a connected graph.                   |
| 4.     | Kruskal's time complexity in worst case is $O(E \log E)$ .                                                  | Prim's algorithm has a time complexity in worst case of $O(E \log V)$ . |

- d. What is dynamic programming ? How is this approach different from recursion? Explain with example.

**Ans.**

1. Dynamic programming is a stage-wise search method suitable for optimization problems whose solutions may be viewed as the result of a sequence of decisions.
2. It is used when the sub-problems are not independent.
3. Dynamic programming takes advantage of the duplication and arranges to solve each sub-problem only once, saving the solution (in table or something) for later use.
4. Dynamic programming can be thought of as being the reverse of recursion. Recursion is a top-down mechanism *i.e.*, we take a problem, split it up, and solve the smaller problems that are created. Dynamic programming is a bottom-up mechanism *i.e.*, we solve all possible small problems and then combine them to obtain solutions for bigger problems.

**Difference :**

1. In recursion, sub-problems are solved multiple times but in dynamic programming sub-problems are solved only one time.
2. Recursion is slower than dynamic programming.

**For example :**

Consider the example of calculating  $n^{\text{th}}$  Fibonacci number.

$$\text{fibo}(n) = \text{fibo}(n - 1) + \text{fibo}(n - 2)$$

$$\text{fibo}(n - 1) = \text{fibo}(n - 2) + \text{fibo}(n - 3)$$

$$\text{fibo}(n - 2) = \text{fibo}(n - 3) + \text{fibo}(n - 4)$$

.....  
.....  
.....

$$\text{fibo}(2) = \text{fibo}(1) + \text{fibo}(0)$$

In the first three steps, it can be clearly seen that  $\text{fibo}(n - 3)$  is calculated twice. If we use recursion, we calculate the same sub-problems again and again but with dynamic programming we calculate the sub-problems only once.

- e. Define NP-hard and NP-complete problems. What are the steps involved in proving a problem NP-complete ? Specify the problems already proved to be NP-complete.

**Ans. NP-hard problem :**

1. We say that a decision problem  $P_i$  is NP-hard if every problem in NP is polynomial time reducible to  $P_i$ .
  2. In symbols,
- $P_i$  is NP-hard if, for every  $P_j \in \text{NP}$ ,  $P_j \xrightarrow{\text{Poly}} P_i$ .
3. This does not require  $P_i$  to be in NP.
  4. Highly informally, it means that  $P_i$  is 'as hard as' all the problem in NP.
  5. If  $P_i$  can be solved in polynomial time, then all problems in NP.
  6. Existence of a polynomial time algorithm for an NP-hard problem implies the existence of polynomial solution for every problem in NP.

**NP-complete problem :**

1. There are many problems for which no polynomial time algorithms is known.
2. Some of these problems are travelling salesman problem, optimal graph colouring, the Knapsack problem, Hamiltonian cycles, integer programming, finding the longest simple path in a graph, and satisfying a Boolean formula.
3. These problems belongs to an interesting class of problems called the “NP-complete” problems, whose status is unknown.
4. The NP-complete problems are traceable i.e., require a super polynomial time.

**Steps involved in proving a problem is NP-complete :**

**Step 1 :** Given a problem U, show that U is NP.

**Step 2 :** Select a known NP-complete problem V.

**Step 3 :** Construct a reduction from V to U.

**Step 4 :** Show that the reduction requires polynomial time.

**List of NP-complete problems :**

1. 3SAT
2. Vertex-cover
3. Clique
4. Hamiltonian circuits.

**Section-C**

3. Attempt any **one** part of the following : **(7 × 1 = 7)**
- a. Among Merge sort, Insertion sort and quick sort which sorting technique is the best in worst case. Apply the best one among these algorithms to sort the list E, X, A, M, P, L, E in alphabetic order.**

**Ans.** Merge sort technique is best in worst case because of its time complexity  $O(n \log n)$ .

**Numerical :**

**Given :** E, X, A, M, P, L, E

**Pass 1 :** Merge each pair of element to obtain sorted list :

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| E | X | A | M | P | L | E |
|---|---|---|---|---|---|---|

After sorting each pair, we get

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| E | X | A | M | L | P | E |
|---|---|---|---|---|---|---|

**Pass 2 :** Merge each pair to obtain the list :

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| A | E | M | X | E | L | P |
|---|---|---|---|---|---|---|

**Pass 3 :** Again merge the two sub arrays to obtain the list :

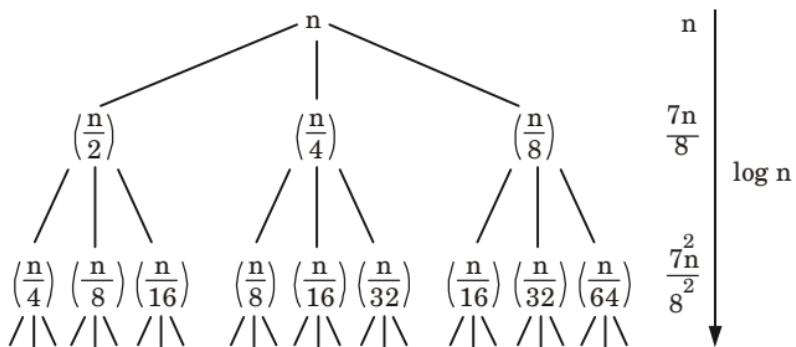
|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| A | E | E | L | M | P | X |
|---|---|---|---|---|---|---|

- b. Solve the recurrence using recursion tree method :**  
 $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

**Ans.**

$$T(n) = n + \frac{7n}{8} + \frac{7^2 n}{8^2} + \dots + \log n \text{ times}$$

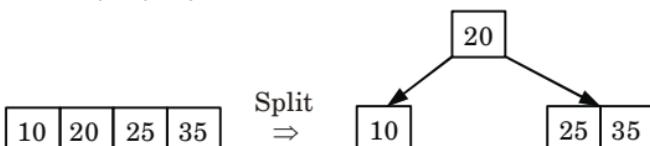
$$= \Omega(n \log n)$$



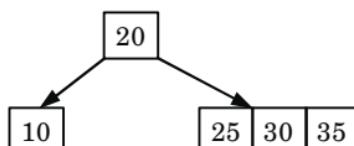
4. Attempt any **one** part of the following :  $(7 \times 1 = 7)$

- a. Using minimum degree 't' as 3, insert following sequence of integers 10, 25, 20, 35, 30, 55, 40, 45, 50, 55, 60, 75, 70, 65, 80, 85 and 90 in an initially empty B-Tree. Give the number of nodes splitting operations that take place.

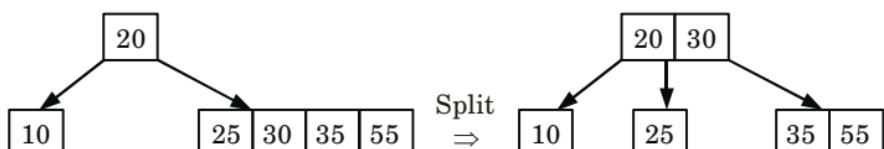
**Ans.** Insert 10, 25, 20, 35 :



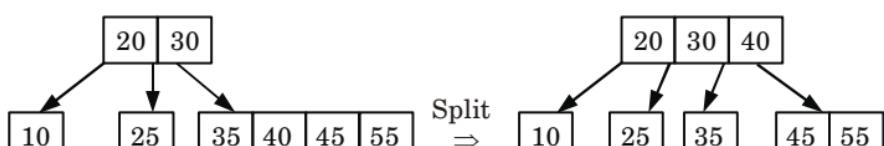
**Insert 30 :**



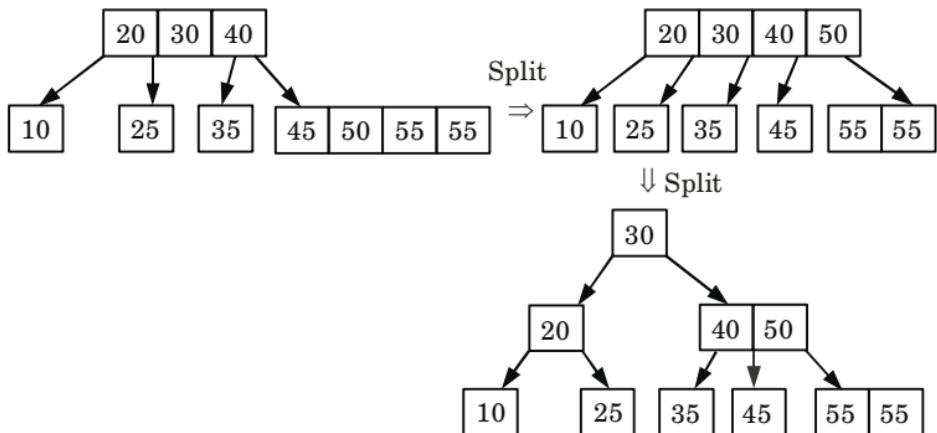
**Insert 55 :**



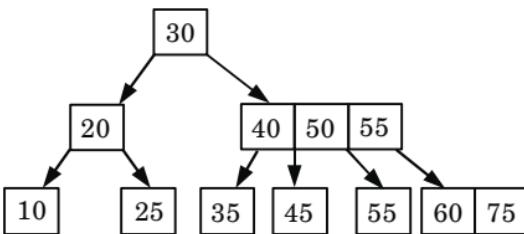
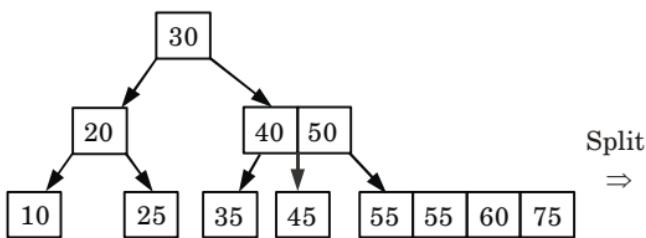
**Insert 40, 45 :**



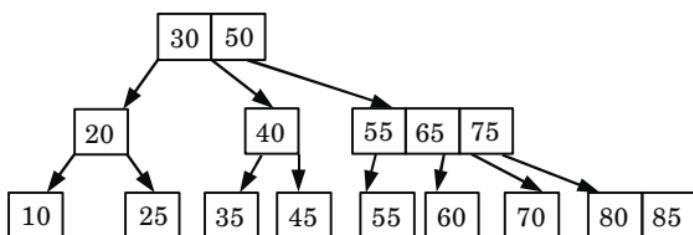
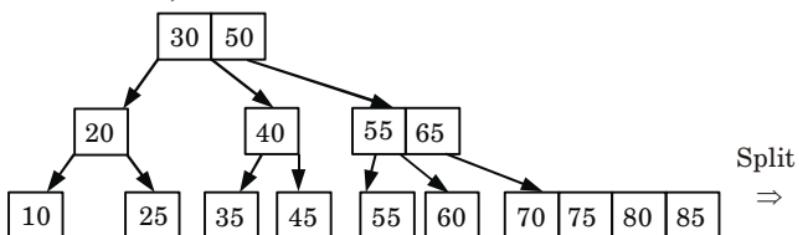
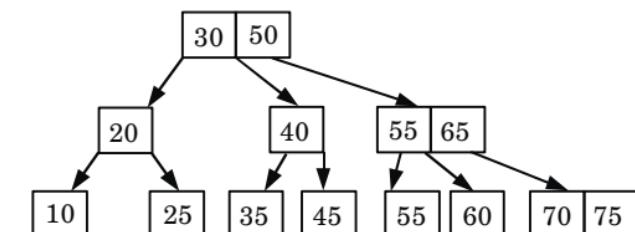
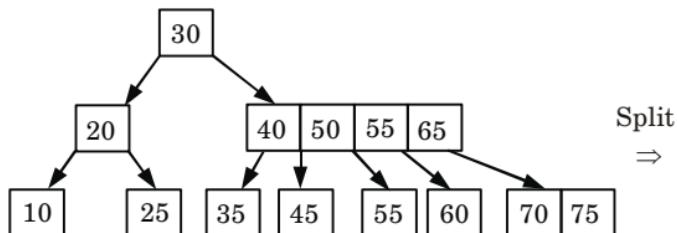
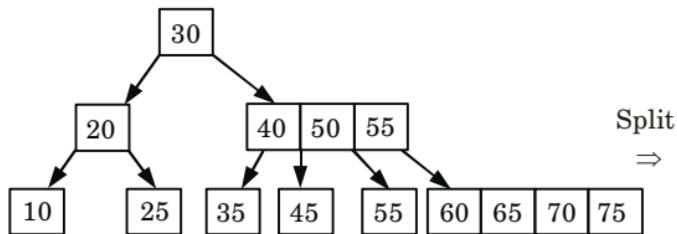
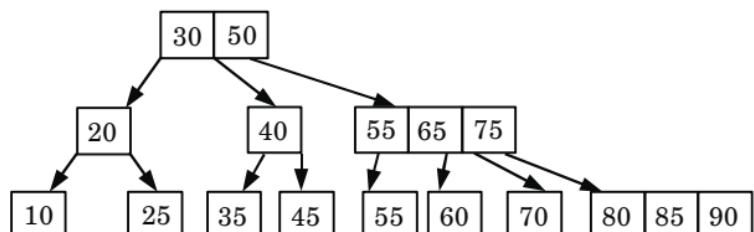
**Insert 50, 55 :**



**Insert 60, 75 :**



**Insert 70, 65 :**

**Insert 90 :**

Number of nodes splitting operations = 9.

- b. Explain the algorithm to delete a given element in a binomial heap. Give an example for the same.

**Ans.** **Deletion of key from binomial heap :**

The operation BINOMIAL-HEAP-DECREASE ( $H, x, k$ ) assigns a new key ' $k$ ' to a node ' $x$ ' in a binomial heap  $H$ .

**BINOMIAL-HEAP-DECREASE-KEY ( $H, x, k$ )**

1. if  $k > \text{key}[x]$  then
2. Message "error new key is greater than current key"
3.  $\text{key}[x] \leftarrow k$
4.  $y \leftarrow x$
5.  $z \leftarrow P[y]$
6. While ( $z \neq \text{NIL}$ ) and  $\text{key}[y] < \text{key}[z]$
7. do exchange  $\text{key}[y] \leftrightarrow \text{key}[z]$
8.  $y \leftarrow z$
9.  $z \leftarrow P[y]$

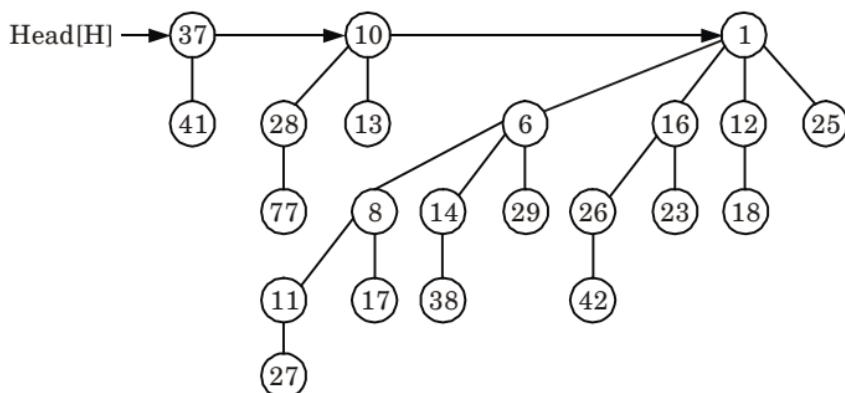
**Deleting a key :** The operation BINOMIAL-HEAP-DELETE ( $H, x$ ) is used to delete a node  $x$ 's key from the given binomial heap  $H$ . The following implementation assumes that no node currently in the binomial heap has a key of  $-\infty$ .

**BINOMIAL-HEAP-DELETE ( $H, x$ )**

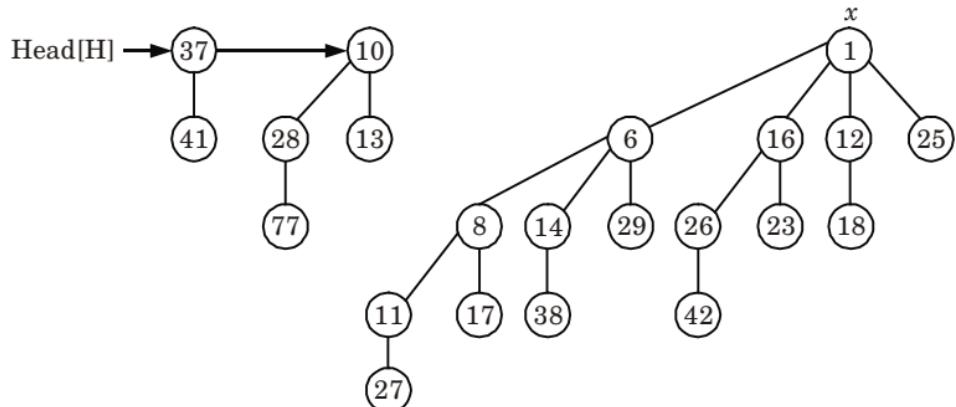
1. BINOMIAL-HEAP-DECREASE-KEY ( $H, x, -\infty$ )
2. BINOMIAL-HEAP-EXTRACT-MIN( $H$ )

**For example :** Operation of Binomial-Heap-Decrease ( $H, x, k$ ) on the following given binomial heap :

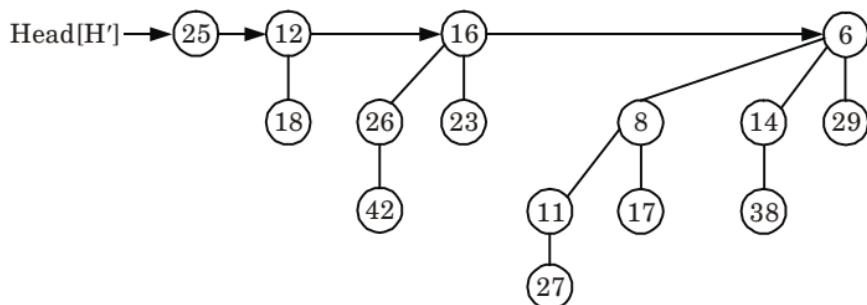
Suppose a binomial heap  $H$  is as follows :



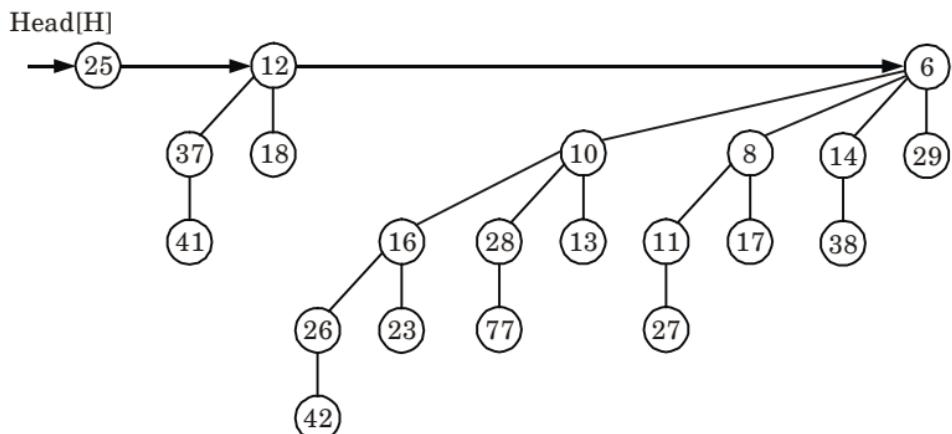
The root  $x$  with minimum key is 1.  $x$  is removed from the root list of  $H$ . i.e.,



Now, the linked list of  $x$ 's children is reversed and set  $\text{head}[H']$  to point to the head of the resulting list, i.e., another binomial heap  $H'$ .



Now, call **BINOMIAL-HEAP-UNION** ( $H, H'$ ) to unite the two binomial heaps  $H$  and  $H'$ . The resulting binomial heap is



5. Attempt any **one** part of the following :  $(7 \times 1 = 7)$
- Compare the various programming paradigms such as divide-and-conquer, dynamic programming and greedy approach.**

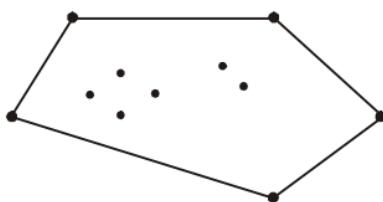
**Ans.**

| S.No. | Divide and conquer approach                                                                                               | Dynamic programming approach                                                                                                | Greedy approach                                              |
|-------|---------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|
| 1.    | Optimizes by breaking down a subproblem into simpler versions of itself and using multi-threading and recursion to solve. | Same as Divide and Conquer, but optimizes by caching the answers to each subproblem as not to repeat the calculation twice. | Optimizes by making the best choice at the moment.           |
| 2.    | Always finds the optimal solution, but is slower than Greedy.                                                             | Always finds the optimal solution, but cannot work on small datasets.                                                       | Does not always find the optimal solution, but is very fast. |
| 3.    | Requires some memory to remember recursive calls.                                                                         | Requires a lot of memory for tabulation.                                                                                    | Requires almost no memory.                                   |

- b. What do you mean by convex hull ? Describe an algorithm that solves the convex hull problem. Find the time complexity of the algorithm.**

**Ans.**

1. The convex hull of a set  $S$  of points in the plane is defined as the smallest convex polygon containing all the points of  $S$ .
2. The vertices of the convex hull of a set  $S$  of points form a (not necessarily proper) subset of  $S$ .
3. To check whether a particular point  $p \in S$  is extreme, see each possible triplet of points and check whether  $p$  lies in the triangle formed by these three points.

**Fig. 1.**

4. If  $p$  lies in any triangle then it is not extreme, otherwise it is.
5. We denote the convex hull of  $S$  by  $\text{CH}(S)$ . Convex hull is a convex set because the intersection of convex sets is convex and convex hull is also a convex closure.

### Graham-Scan algorithm :

The procedure GRAHAM-SCAN takes as input a set  $Q$  of points, where  $|Q| \geq 3$ . It calls the functions  $\text{Top}(S)$ , which return the point on top of stack  $S$  without changing  $S$ , and to  $\text{NEXT-TO-TOP}(S)$ , which returns the point one entry below the top of stack  $S$  without changing  $S$ .

#### **GRAHAM-SCAN( $Q$ )**

1. Let  $p_0$  be the point in  $Q$  with the minimum  $y$ -coordinate, or the leftmost such point in case of a tie.
2. Let  $\langle p_1, p_2, \dots, p_m \rangle$  be the remaining points in  $Q$ , sorted by polar angle in counter clockwise order around  $p_0$  (if more than one point has the same angle remove all but the one that is farthest from  $p_0$ ).
3.  $\text{Top}[S] \leftarrow 0$
4.  $\text{PUSH}(p_0, S)$
5.  $\text{PUSH}(p_1, S)$
6.  $\text{PUSH}(p_2, S)$
7. for  $i \leftarrow 3$  to  $m$
8. do while the angle formed by points  $\text{NEXT-TO-TOP}(S)$ ,  $\text{Top}(S)$ , and  $p_i$  makes a non left turn.
  9.           do  $\text{POP}(S)$
  10.           $\text{PUSH}(p_i, S)$
11. return  $S$

#### **Time complexity :**

The worst case running time of GRAHAM-SCAN is

$$T(n) = O(n) + O(n \log n) + O(1) + O(n) = O(n \log n)$$

where  $n = |Q|$

Graham's scan running time depends only on the size of the input it is independent of the size of output.

6. Attempt any **one** part of the following :

- Solve the following 0/1 knapsack problem using dynamic programming**  $P = \{11, 21, 31, 33\}$   $w = \{2, 11, 22, 15\}$   $c = 40$ ,  $n = 4$ .

#### **Ans. Numerical :**

$$w = \{2, 11, 22, 15\}$$

$$c = 40$$

$$p = \{11, 21, 31, 33\}$$

Initially,

| Item  | $w_i$ | $p_i$ |
|-------|-------|-------|
| $I_1$ | 2     | 11    |
| $I_2$ | 11    | 21    |
| $I_3$ | 22    | 31    |
| $I_4$ | 15    | 33    |

Taking value per weight ratio, i.e.,  $p_i/w_i$

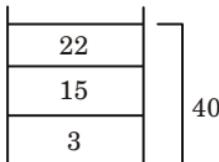
| Item  | $w_i$ | $v_i/w_i$ | $p_i$ |
|-------|-------|-----------|-------|
| $I_1$ | 2     | 11        | 22    |
| $I_2$ | 11    | 21        | 232   |
| $I_3$ | 22    | 31        | 682   |
| $I_4$ | 15    | 33        | 495   |

Now, arrange the value of  $p_i$  in decreasing order.

| Item  | $w_i$ | $p_i$ | $p_i$ |
|-------|-------|-------|-------|
| $I_3$ | 22    | 31    | 682   |
| $I_4$ | 15    | 33    | 495   |
| $I_2$ | 11    | 21    | 232   |
| $I_1$ | 2     | 11    | 22    |

Now, fill the knapsack according to decreasing value of  $p_i$ .

First we choose item  $I_3$  whose weight is 22, then choose item  $I_4$  whose weight is 15. Now the total weight in knapsack is  $22 + 15 = 37$ . Now, next item is  $I_2$  and its weight is 11 and then again  $I_1$ . So, we choose fractional part of it, i.e.,



The value of fractional part of  $I_1$  is,

$$= \frac{232}{11} \times 3 = 63$$

Thus, the maximum value is,

$$= 682 + 495 + 63 = 1190$$

- b. Define Floyd Warshall algorithm for all pair shortest path and apply the same on following graph :**

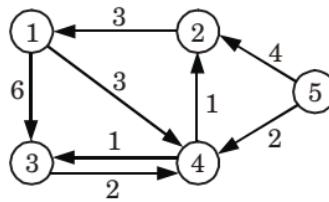


Fig. 2.

**Ans. Floyd Warshall algorithm :**

1. Floyd-Warshall algorithm is a graph analysis algorithm for finding shortest paths in a weighted, directed graph.
2. A single execution of the algorithm will find the shortest path between all pairs of vertices.
3. It does so in  $\Theta(V^3)$  time, where  $V$  is the number of vertices in the graph.
4. Negative-weight edges may be present, but we shall assume that there are no negative-weight cycles.
5. The algorithm considers the “intermediate” vertices of a shortest path, where an intermediate vertex of a simple path  $p = (v_1, v_2, \dots, v_m)$  is any vertex of  $p$  other than  $v_1$  or  $v_m$ , that is, any vertex in the set  $\{v_2, v_3, \dots, v_{m-1}\}$ .
6. Let the vertices of  $G$  be  $V = \{1, 2, \dots, n\}$ , and consider a subset  $\{1, 2, \dots, k\}$  of vertices for some  $k$ .
7. For any pair of vertices  $i, j \in V$ , consider all paths from  $i$  to  $j$  whose intermediate vertices are all drawn from  $\{1, 2, \dots, k\}$ , and let  $p$  be a minimum-weight path from among them.
8. Let  $d_{ij}^{(k)}$  be the weight of a shortest path from vertex  $i$  to vertex  $j$  with all intermediate vertices in the set  $\{1, 2, \dots, k\}$ .

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

**Numerical :**

$$d_{ij}^{(k)} = \min[d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}]$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & \infty & 6 & 3 & \infty \\ 3 & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & 2 & \infty \\ \infty & 1 & 1 & 0 & \infty \\ \infty & 4 & \infty & 2 & 0 \end{matrix} \right] \end{matrix}$$

$$D^{(1)} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & 4 & 6 & 3 & \infty \\ 3 & 0 & 9 & 6 & \infty \\ 6 & 4 & 0 & 2 & \infty \\ 4 & 1 & 1 & 0 & \infty \\ 7 & 4 & 3 & 2 & 0 \end{matrix} \right] \end{matrix}$$

$$D^{(2)} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \left[ \begin{matrix} 0 & 4 & 6 & 3 & \infty \\ 3 & 0 & 7 & 6 & \infty \\ 6 & 3 & 0 & 2 & \infty \\ 4 & 1 & 1 & 0 & \infty \\ 6 & 3 & 3 & 2 & 0 \end{matrix} \right] \end{matrix}$$

Now, if we find  $D^{(3)}$ ,  $D^{(4)}$  and  $D^{(5)}$  there will be no change in the entries.

7. Attempt any **one** part of the following : **(7 × 1 = 7)**
- a. **Describe in detail Knuth-Morris-Pratt string matching algorithm. Compute the prefix function  $\pi$  for the pattern  $ababbabbabbababbabb$  when the alphabet is  $\Sigma = \{a, b\}$ .**

**Ans.** Knuth-Morris-Pratt algorithm for string matching :  
COMPUTE-PREFIX-FUNCTION ( $P$ )

1.  $m \leftarrow \text{length } [P]$
2.  $\pi [1] \leftarrow 0$
3.  $k \leftarrow 0$

4. for  $q \leftarrow 2$  to  $m$
5.     do while  $k > 0$  and  $P[k + 1] \neq P[q]$
6.         do  $k \leftarrow \pi[k]$
7.         if  $P[k + 1] = P[q]$
8.             then  $k \leftarrow k + 1$
9.          $\pi[q] \leftarrow k$
10. return  $\pi$

KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute  $\pi$ .

### **KMP-MATCHER ( $T, P$ )**

1.  $n \leftarrow \text{length}[T]$
2.  $m \leftarrow \text{length}[P]$
3.  $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$
4.  $q \leftarrow 0$
5. for  $i \leftarrow 1$  to  $n$
6.     do while  $q > 0$  and  $P[q + 1] \neq T[i]$
7.         do  $q \leftarrow \pi[q]$
8.         if  $P[q + 1] = T[i]$
9.             then  $q \leftarrow q + 1$
10. if  $q = m$
11. then print "pattern occurs with shift"  $i - m$
12.  $q \leftarrow \pi[q]$

### **Numerical :**

pattern = ababbabbabbababbabb  
 length                          19

Initially,       $\pi(1) = 0$  and  $k = 0$

For                           $q \leftarrow 2$  to 9

For                           $q = 2$  and  $k \neq 0$

$P[0 + 1] \neq P[2]$ .

$\pi[2] = 0$

For                           $q = 3$

$P[0 + 1] = P[3]$

$k = k + 1 = 1$

$\pi[3] = 1$

For                           $q = 4$   $k > 0$

$P[1 + 1] = P[4]$

$P[2] = P[4]$

$k = k + 1 = 2$

$\pi[4] = 2$

For                           $q = 5$   $k > 0$

$P[2 + 1] \neq P[5]$

$\pi[5] = 0$

For                           $q = 6$

$P[0 + 1] = P[6]$

$k = k + 1 = 1$   
 $\pi[6] = 1$

For             $q = 7 \ k > 0$   
 $P[1 + 1] = P[7]$   
 $P[2] = P[7]$   
 $k = k + 1 = 2$   
 $\pi[7] = 0$

For             $q = 8 \ k > 0$   
 $P[2 + 1] = P[8]$   
 $P[3] \neq P[8]$   
 $\pi[8] = 0$

For             $q = 9$   
 $P[0 + 1] = P[9]$   
 $k = k + 1$   
 $\pi[9] = 2$

For             $q = 10 \ k > 0$   
 $P[2 + 1] \neq P[10]$   
 $\pi[10] = 0$

For             $q = 11 \ k > 0$   
 $P[0 + 1] \neq P[11]$   
 $\pi[11] = 0$

For             $q = 12 \ k > 0$   
 $P[0 + 1] \neq P[12]$   
 $k = k + 1$   
 $\pi[12] = 2$

For             $q = 13 \ k > 0$   
 $P[2 + 1] \neq P[13]$   
 $\pi[13] = 0$

For             $q = 14$   
 $P[0 + 1] = P[14]$   
 $k = k + 1$   
 $\pi[14] = 2$

For             $q = 15 \ k > 0$   
 $P[2 + 1] \neq P[15]$   
 $\pi[15] = 0$

For             $q = 16 \ k > 0$   
 $P[0 + 1] \neq P[16]$   
 $\pi[16] = 0$

For             $q = 17$   
 $P[0 + 1] \neq P[17]$   
 $k = k + 1$   
 $\pi[17] = 2$

For             $q = 18 \ k > 0$   
 $P[2 + 1] \neq P[18]$   
 $\pi[18] = 0$

For             $q = 19$   
 $P[0 + 1] \neq P[19]$

$$\pi[19] = 0$$

|          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| String   | <i>a</i> | <i>b</i> | <i>a</i> | <i>b</i> | <i>a</i> | <i>b</i> | <i>b</i> | <i>a</i> | <i>b</i> | <i>b</i> |
| $P[i]$   | 1        | 2        | 3        | 4        | 5        | 6        | 7        | 8        | 9        | 10       | 11       | 12       | 13       | 14       | 15       | 16       | 17       | 18       | 19       |
| $\pi[i]$ | 0        | 0        | 1        | 2        | 0        | 1        | 2        | 0        | 2        | 0        | 0        | 2        | 0        | 2        | 0        | 0        | 2        | 0        | 0        |

- b. What is an approximation algorithm ? What is meant by  $p(n)$  approximation algorithms ? Discuss approximation algorithm for Travelling Salesman Problem.**

**Ans. Approximation algorithm :**

1. An approximation algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution.
2. The best of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.
3. Let  $c(i)$  be the cost of solution produced by approximate algorithm and  $c^*(i)$  be the cost of optimal solution for some optimization problem instance  $i$ .
4. For minimization and maximization problem, we are interested in finding a solution of a given instance  $i$  in the set of feasible solutions, such that  $c(i)/c^*(i)$  and  $c^*(i)/c(i)$  be as small as possible respectively.
5. We say that an approximation algorithm for the given problem instance  $i$ , has a ratio bound of  $p(n)$  if for any input of size  $n$ , the cost  $c$  of the solution produced by the approximation algorithm is within a factor of  $p(n)$  of the cost  $c^*$  of an optimal solution. That is  $\max(c(i) / c^*(i), c^*(i) / c(i)) \leq p(n)$

**$p(n)$  approximation algorithm :** A is a  $p(n)$  approximate algorithm if and only if for every instance of size  $n$ , the algorithm achieves an approximation ratio of  $p(n)$ . It is applied to both maximization ( $0 < C(i) \leq C^*(i)$ ) and minimization ( $0 < C^*(i) \leq C(i)$ ) problem because of the maximization factor and costs are positive.  $p(n)$  is always greater than 1.

**Approximation algorithm for Travelling Salesman Problem (TSP) :**

1. The key to designing approximation algorithm is to obtain a bound on the optimal value (OPT).
2. In the case of TSP, the minimum spanning tree gives a lower bound on OPT.
3. The cost of a minimum spanning tree is not greater than the cost of an optimal tour.

The algorithm is as follows :

1. Find a minimum spanning tree of  $G$ .

2. Duplicate each edge in the minimum spanning tree to obtain a Eulerian graph.
3. Find a Eulerian tour ( $J$ ) of the Eulerian graph.
4. Convert  $J$  to a tour  $T$  by going through the vertices in the same order of  $T$ , skipping vertices that were already visited.

