SAGAR INSTITUTE OF RESEARCH AND TECHNOLOGY-EXCELLENCE, BHOPAL

Department of CSE

Subject Name /Code:	ADA/CS-402
Semester:	IV CSE
Faculty Name:	Prof. Ankur Pathrey.

S.No	Date	VN	MN	PEO & Pos	COs	MAP	AC	Scheme of Exam	CS & Gate Syllabus	TT	LP	AS	MID SEM I & II	University papers	Lect Notes	CBS	Remarks
1	03/5/23	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Pl. Add ~~Experiential~~ Learning. / Experiential. 08/5/23 (5 Hours)

Pl. update experential learning as discussed (Upto 15/05/2023).

Vlogey 06/05/2023

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

Course File

Subject Name	:	**Analysis Design of Algorithm**
Subject Code	:	**CS-402**
Academic Year	:	**Jan-June 2023**
Semester	:	**IV th**
Designation	:	**Assistant Professor**

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

LIST OF CONTENT

Vision and Mission of the Institute

VISION

To motivate and mold students into world class professionals who will excel in their fields and effectively meet challenges of the dynamic global scenario.

MISSION

Working towards being the best by incorporating the principles of total quality management (TQM) and excellence. Adopting IT based knowledge management to meet global challenges.

Mission and Vision of the Department

DEPARMENT OF COMPUTER SCIENCE & ENGINEERING

Vision

To produce talented engineers to meet the challenges of the modern world and train the students to understand the human values.

Mission

To impart, maintain and improve technical excellence, ethical values and overall development of the students for meeting global standards with ever enhancing competence in teaching and research.

PEOs, POs AND PSOs

PEOs AND POs

Programme Educational Outcomes (PEOs)

PEO 1: Graduates will have able to find employment in research sectors, software & IT industries and related sectors.

PEO 2: Graduates will have prepared for admission in the institute of repute for postgraduate program.

PEO 3: Graduates will have able to demonstrate skills in presentation, leadership, teamwork, social responsibility and entrepreneurship.

PEO 4: Graduates will have able to work for interdisciplinary research by using modern computer techniques.

PEO 5: Graduates will have able to Learn to apply modern skills, techniques, and **engineering** tools to create computational systems.

Programme Outcomes (POs)

PO 1: Graduates will demonstrate the ability to use basic knowledge in mathematics, science and engineering and apply them to solve problems specific to Computer engineering.

PO 2: Graduates will demonstrate the ability to design and conduct experiments, interpret and analyze data, and report results.

PO 3: Graduates will demonstrate the ability to design any Computer Software that meets desired specifications and requirements.

PO 4: Graduates will demonstrate the ability to identify, formulate and solve Computer engineering problems of a complex kind.

PO 5: Graduates will be familiar with applying software methods and modern computer tools to analyze Computer engineering problems.

PO 6: Graduates will develop an open mind and have an understanding of the impact of engineering on society and demonstrate awareness of contemporary issues.

PO 7: Graduates will able to design a system, component, or process to meet desired needs within realistic constraints such as economic, environmental, social, political, ethical, health care and safety, manufacturability, and sustainability.

PO 8: Graduates will demonstrate an understanding of their professional and ethical responsibilities, and use technology for the benefit of mankind.

PO 9: Graduates will demonstrate the ability to function as a coherent unit in multidisciplinary design teams and deliver results through collaborative research.

PO 10: Graduates will be able to communicate effectively in both verbal and written forms.

PO 11: Graduates will have the confidence to apply engineering solutions in global contexts

PO 12: Graduates should be capable of self-education and clearly understand the value of lifelong learning.

Program Specific Outcomes (PSOs)

PSO 1: Ability to implement the learned principles of computer Engineering to analyze, evaluate and create more advanced application.

PSO 2: Ability to apply the acquired computer Engineering knowledge for the advancement of society and self.

Course objectives
&
Course outcomes (COs)

Course Objectives:

To introduce the fundamental concept of data structures and to emphasize the importance of data structures in developing and implementing efficient algorithms.

COURSE OUTCOME

Program Name : **B. Tech.**
Branch : **CSE**
Semester : **IV**
Course Name : **Analysis and Design of Algorithm**
Course Code : **CS-402**
Session : **JAN-JUNE 2023**

The student will be able to

S. No.	Description	Bloom's Taxonomy Level
CO1.	**Understand** and **Explain** the concept of various techniques of analaysis and design of alogirthm	L1/L2
CO2.	Ability to **Apply** the concept of design algorithms using divide and conquer, greedy, dynamic programming, backtracking, branch & bound and graph-tree	L3
CO3.	Ability to **Analyze** the asymptotic performance of algorithms.	L4
CO4.	Ability to **Evaluate** efficient algorithms in common engineering design situations.	L5
CO5.	Ability to **conduct experiment** to implement various algorithm and submit file individualy.	L3

L1:Remember
L2: Understand
L3: Apply
L4 : Analyze
L5 : Evaluate
L6 : Create

Mapping of COs with POs

Mapping with COs and POs

	CO1	CO2	CO3	CO4	CO5
PO1					
PO2	✓		✓		
PO3					
PO4		✓		✓	
PO5					
PO6		✓	✓		
PO7					
PO8					
PO9	✓			✓	
PO10					
PO11			✓		
PO12	✓	✓			

Academic calendar

Sagar I[redacted]of Research & Technology - Excellence, Bhopal

ACADEMIC CALENDAR - 2023 (EVEN SEMESTER)

For 4th Semester

Date: 16.03.2023

S.No.	Particular	Dates
1	Duration of semester	Jan- June 2023
2	Commencement of Classess	20th Feb 2023
3	First faculty feedback	30 March 23
4	First Mid Sem Exam (2 units)	11-13 April '23
5	Display of First Mid Sem Marks	18-Apr-23
6	Second Mid Sem Exam (3 units)	24th May - 26th May '23
7	Display of Second Mid Sem Marks	30th May '23
8	Last date of teaching	27th May 23
9	Submission of Internal Marks to University	29 May- 26th June 23
10	Submission of Exam Forms through - without late fee	10th June - 20th June 23
12	Submission of Exam Forms through - with late fee	21nd June - 25th June 23
13	End Semester Examination Theory	27th June - 11th July 23
14	End Semester Examination Pratical Examination	12th July - 17th July 23
15	End Semester Break/Interaship	29th May - 26th June 23
16	Declaration of result	End of August 23

Dr. Vikas S. Pagey
Director, SIRTE

SCHEME OF EXAMINATION

Rajiv Gandhi Proudhyogiki Vishwavidyalaya, Bhopal

Scheme of Examination as per AICTE Flexible Curricula

IV Semester **Bachelor of Technology (B.Tech.) |Computer Science and Engineering/ Computer Engineering/Computer Science & Technology|**

For batches admitted in July, 2020 (w.e.f. Jan, 2022)

S. No.	Subject Code	Category	Subject Name	Maximum Marks Allotted					Total Marks	Contact Hours per week			Total Credits
				Theory			Practical						
				End Sem.	Mid Sem. Exam.	Quiz/ Assignment	End Sem.	Term work / Lab Work & Sessional		L	T	P	
1.	BT401	BSC	Mathematics- III	70	20	10	-	-	100	3	1	-	4
2.	CS402	DC	Analysis Design of Algorithm	70	20	10	30	20	150	2	1	2	4
3.	CS403	DC	Software Engineering	70	20	10	30	20	150	3	1	2	5
4.	CS404	DC	Computer Org. & Architecture	70	20	10	30	20	150	3	1	2	5
5.	CS405	DC	Operating Systems	70	20	10	30	20	150	3	0	2	4
6.	CS406	DLC*	Programming Practices	-	-	-	30	20	50	-	-	4	2
7.	BT407	DLC	90 hrs Internship based on using various software's –Internship -II	To be completed anytime during fourth semester. Its evaluation/credit to be added in fifth semester.									3
			Total	350	100	50	150	100	750	14	4	12	24
8.	BT408	MC	Cyber Security	Non-credit course									
9.	BT409I	MC	Indian Knowledge System	Non-credit course									
	NC001		NSS/NCC										

*A minimum of 2hours per week should be allotted for the Virtual Lab along with the slot fixed for the conventional lab classes.

MST: Minimum of two mid semester tests to be conducted.

*Students can earn additional credits from the University recognized MOOC courses.

1 Hr Lecture	1 Hr Tutorial	2 Hr Practical
1 Credit	1 Credit	1 Credit

Course Syllabus

&

Gate Syllabus

RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Computer Science and Engineering, IV-Semester

CS402 Analysis Design of Algorithm

Algorithms, Designing algorithms, analyzing algorithms, asymptotic notations, heap and heap sort. Introduction to divide and conquer technique, analysis, design and comparison of various algorithms based on this technique, example binary search, merge sort, quick sort, strassen's matrix multiplication.

Study of Greedy strategy, examples of greedy method like optimal merge patterns, Huffman coding, minimum spanning trees, knapsack problem, job sequencing with deadlines, single source shortest path algorithm

Concept of dynamic programming, problems based on this approach such as 0/1 knapsack, multistage graph, reliability design, Floyd-Warshall algorithm

Backtracking concept and its examples like 8 queen's problem, Hamiltonian cycle, Graph coloring problem etc. Introduction to branch & bound method, examples of branch and bound method like traveling salesman problem etc. Meaning of lower bound theory and its use in solving algebraic problem, introduction to parallel algorithms.

Binary search trees, height balanced trees, 2-3 trees, B-trees, basic search and traversal techniques for trees and graphs (In order, preorder, postorder, DFS, BFS), NP-completeness.

References:

1. Coremen Thomas, Leiserson CE, Rivest RL; Introduction to Algorithms; PHI.
2. Horowitz & Sahani; Analysis & Design of Algorithm
3. Dasgupta; algorithms; TMH
4. Ullmann; Analysis & Design of Algorithm;
5. Michael T Goodrich, Robarto Tamassia, Algorithm Design, Wiely India
6. Rajesh K Shukla: Analysis and Design of Algorithms: A Beginner's Approach; Wiley

List of Experiments(expandable):

1. Write a program for Iterative and Recursive Binary Search.
2. Write a program for Merge Sort.
3. Write a program for Quick Sort.
4. Write a program for Strassen's Matrix Multiplication.
5. Write a program for optimal merge patterns.
6. Write a program for Huffman coding.
7. Write a program for minimum spanning trees using Kruskal's algorithm.
8. Write a program for minimum spanning trees using Prim's algorithm.
9. Write a program for single sources shortest path algorithm.
10. Write a program for Floye-Warshal algorithm.
11. Write a program for traveling salesman problem.
12. Write a program for Hamiltonian cycle problem.

CS: Computer Science and Information Technology

Section 1: Engineering Mathematics

Discrete Mathematics: Propositional and first order logic. Sets, relations, functions, partial orders and lattices. Groups. Graphs: connectivity, matching, coloring. Combinatorics: counting, recurrence relations, generating functions.

Linear Algebra: Matrices, determinants, system of linear equations, eigenvalues and eigenvectors, LU decomposition.

Calculus: Limits, continuity and differentiability. Maxima and minima. Mean value theorem. Integration.

Probability: Random variables. Uniform, normal, exponential, poisson and binomial distributions. Mean, median, mode and standard deviation. Conditional probability and Bayes theorem.

Section 2 – 10: Computer Science and Information Technology

Section 2: Digital Logic

Boolean algebra. Combinational and sequential circuits. Minimization. Number representations and computer arithmetic (fixed and floating point).

Section 3: Computer Organization and Architecture

Machine instructions and addressing modes. ALU, data-path and control unit. Instruction pipelining. Memory hierarchy: cache, main memory and secondary storage; I/O interface (interrupt and DMA mode).

Section 4: Programming and Data Structures

Programming in C. Recursion. Arrays, stacks, queues, linked lists, trees, binary search trees, binary heaps, graphs.

Section 5: Algorithms

Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques: greedy, dynamic programming and divide-and-conquer. Graph search, minimum spanning trees, shortest paths.

Section 6: Theory of Computation

Regular expressions and finite automata. Context-free grammars and push-down automata. Regular and contex-free languages, pumping lemma. Turing machines and undecidability.

Section 7: Compiler Design

Lexical analysis, parsing, syntax-directed translation. Runtime environments. Intermediate code generation.

Section 8: Operating System

Processes, threads, inter-process communication, concurrency and synchronization. Deadlock. CPU scheduling. Memory management and virtual memory. File systems.

Section 9: Databases

ER-model. Relational model: relational algebra, tuple calculus, SQL. Integrity constraints, normal forms. File organization, indexing (e.g., B and B+ trees). Transactions and concurrency control.

Section 10: Computer Networks

Concept of layering. LAN technologies (Ethernet). Flow and error control techniques, switching. IPv4/IPv6, routers and routing algorithms (distance vector, link state). TCP/UDP and sockets, congestion control. Application layer protocols (DNS, SMTP, POP, FTP, HTTP). Basics of Wi-Fi. Network security: authentication, basics of public key and private key cryptography, digital signatures and certificates, firewalls.

Sagar Institute o

Depar

Time Table (A

ssion: (Jan to June 2023)

w.e.f. 14-03-2023

SEMESTER		AI & DS IV (211)	CS-A IV SEM (205)	CS-B IV	CS-A+B VIII SEM (207)
Time	Period	Subject	Subject	Subject	Subject
8:30-9:20	I	AD-405 OS (IS)	CS-405 OS (SG)	CS-403 SE (PT)	CS 802 CC (GS)
09:20-10:10	II	BT-401 M-III (USS)	CS-403 SE (PT)	CS-405 OS (SG)	CS 803 IPCV (IS)
10:10-11:00	III	AD-403 SE with Agile (GS)	SE LAB (PT) A1 L5/ JAVA LAB (VL) A2 L10	COA LAB (HN) A1/ LAB (SG) A2 L3	MAJOR PROJECT LAB (IS
11:00-11:50	IV	AD-404 DS (AT)			
11:50-12:30					
12:30-1:20	V	AD-402 DBMS (AKS)	CS-404 COA (HN)	BT-401 M-III (USS)	CS 801 IOT (VL)
1:20-2:10	VI	DBMS LAB(AKS) A1 L3/OS LAB (IS) A2 L5	BT-401 M-III (USS)	CS-404 COA (HN)	IOT Lab (VL)
2:10-3:00	VII		CS-402 ADA (AP)	CS-402 ADA (AT)	
8:30-9:20	I	AD-403 SE with Agile (GS)	SE LAB (PT) A2 L5/OS LAB (SG) A1 L3	CS-404 COA (HN)	CS 803 IPCV (IS)
09:20-10:10	II	AD-402 DBMS (AKS)		BT-401 M-III (USS)	CS 801 IOT (VL)
10:10-11:00	III	AD-404 DS (AT)	T&P (GG)	CS-403 SE (PT)	CS 802 CC LAB (GS)
11:00-11:50	IV	BT-401 M-III (USS)	LIBRARY (PT)	T&P (GG)	
11:50-12:30					
12:30-1:20	V	T&P (GG)	BT-401 M-III (USS)	CS-405 OS (SG)	CS 802 CC (GS)
1:20-2:10	VI	SE LAB (GS) A2 L5/OS LAB (IS) A1 L3	CS-405 OS (SG)	ADA LAB (AT) B1 L2/JAVA LAB (VL) B2 L9	MAJOR PROJECT LAB (VL)
2:10-3:00	VII		CS-402 ADA (AP)		
8:30-9:20	I	BT-401 M-III (USS)	CS-402 ADA (AP)	SE LAB (PT) B2 L5/OS LAB (SG) B1 L3	CS 802 CC (GS)
09:20-10:10	II	AD-402 DBMS (AKS)	CS-404 COA (HN)		CS 801 IOT (VL)
10:10-11:00	III	DS LAB (AT) A2 L10/ R LAB (AKS) A1 L9	CS-403 SE (PT)	BT-401 M-III (USS)	MAJOR PROJECT LAB (IS)
11:00-11:50	IV		BT-401 M-III (USS)	CS-405 OS (SG)	
11:50-12:30					
12:30-1:20	V	AD-403 SE with Agile (GS)	T&P (DA)	CS-404 COA (HN)	CS 803 IPCV (IS)
1:20-2:10	VI	AD-405 OS (IS)	COA LAB (HN) A1/ADA LAB (AP) A2 L2	T&P (DA)	IOT Lab (VL)
2:10-3:00	VII	T&P (DA)		CS-402 ADA (AT)	
8:30-9:20	I	BT-401 M-III (USS)	CS-403 SE (PT)	CS-404 COA (HN)	CS 803 IPCV (IS)
09:20-10:10	II	AD-405 OS (IS)	CS-404 COA (HN)	CS-405 OS (SG)	CS 802 CC (GS)
10:10-11:00	III	DS LAB (AT) A1 L10/ DBMS LAB (AKS) A2 L3	ADA LAB (AP) A1 L2/OS LAB (SG) A2 L3	CS-403 SE (PT)	CC Lab (GS)
11:00-11:50	IV			BT-401 M-III (USS)	
11:50-12:30					
12:30-1:20	V	AD-402 DBMS (AKS)	CS-405 OS (SG)	CS-402 ADA (AT)	CS 801 IOT (VL)
1:20-2:10	VI	AD-404 DS (AT)	CS-402 ADA (AP)	JAVA LAB (VL) B1 L10/COA LAB (HN) B2 L3	MAJOR PROJECT LAB (IS/VL)
2:10-3:00	VII	LIBRARY(HP/AT)	BT-401 M-III (USS)		
8:30-9:20	I	AD-402 DBMS (AKS)	CS-403 SE (PT)	BT-401 M-III (USS)	CS 803 IPCV (IS)
09:20-10:10	II	AD-405 OS (IS)	CS-402 ADA (AP)	CS-403 SE (PT)	CS 802 CC (GS)
10:10-11:00	III	SE LAB (GS) A1 L5/R LAB (AKS) A2 L9	BT-401 M-III (USS)	SE LAB (PT) B1 L5/ADA LAB (AT) B2 L3	MAJOR PROJECT LAB (IS)
11:00-11:50	IV		CS-404 COA (HN)		
11:50-12:30					
12:30-1:20	V	AD-404 DS (AT)	CS-405 OS (SG)	CS-404 COA (HN)	CS 801 IOT (VL)
1:20-2:10	VI	BT-401 M-III (USS)	COA LAB (HN) A2/JAVA LAB (VL) A1 L10	CS-402 ADA (AT)	CS 803 IPCV (IS)
2:10-3:00	VII	AD-403 SE with Agile (PS)		LIBRARY(AP/SG)	LIBRARY(LS/NK)

Dr. KALPANA RAI - KR	PROF. VINAY LOWANSHI - VL
Dr. SNEHA SONI - SS	PROF. ATUL KU. SHRIVASTAVA - AS
PROF. NITYA KHARE - NK	PROF. SHUMAILI GUPTA - SG
PROF. ANKITA TIWARI - AT	PROF. HIMANSHU HAUTIYAL - HN
PROF. ANKUR PATHEY - AP	PROF. INDU SHRIVASTAVA - IS
PROF. POOJA TIWARI - PT	PROF. GOLDY SAINI - GS
PROF. KAMINI PACHLASIYA -KP	

aculties -

PROF. MAHENDRA JOSHI- MJ	
PROF. ROLI SHRIVASTAVA-RS	
PROF. GAURAV GAUR- GG	
PROF. DIGANT ARORA - DA	

TT I/C Name & Signature

HOD CSE

Dy. Director/Director

LECTURE PLAN

SAGAR INSTITUTE OF RESEARC[redacted]NOLOGY - EXCELLENCE, BHOPAL

[redacted]PLAN

Institute Name	SIRT[redacted]
Program Name	BTECH
Course Code	CS-402
Course Name	Analysis and Design Of Algorithm
L-T-P	3-1-2
Course Coordinator Name	Prof Ankur Patney

Subject: Analysis and Design Of Algorithm **Semester: IV** **Branch: CSE** **Session: JAN –JUNE 2023**

Topics	Lecture Number	Date on which the Lecture was taken
Unit 1		
Introduction to Algorithms, Designing algorithms	L1	
Analyzing algorithms, Step Count and Complexity	L2	
Recurrence Equation	L3	
Recurrence Relation	L4	
Heap and Heap Sort ,operations	L5	
Tutorial 1	L6	
Introduction to divide and conquer technique	L7	
Binary search	L8	
Quick sort	L9	
Merge sort,	L10	
strassen's matrix multiplication	L11	
Tutorial 2	L12	
Unit Test 1	L13	
Unit 2		
Introduction to Greedy strategy	L1	
Knapsack Problem	L2	
Job Sequencing With Deadline	L3	
Minimum Cost Spanning Tree- Prim's Algorithm	L4	
Minimum Cost Spanning Tree- Prim's Algorithm	L5	
Tutorial 3	L6	
Kruskal's Algorithm	L7	
Optimal Merge Patterns	L8	
Optimal Merge Patterns	L9	
Single Source Shortest path Algorithm (Dijkstra's)	L10	
single source shortest path algorithm,	L11	
Tutorial 4	L12	
Unit Test 2	L13	
Unit3		
Concept of dynamic programming	L1	
problems based on this approach such as 0/1 knapsack	L2	

problems based on this approach such as 0/1 knapsack	L3	
multistage graph	L4	
multistage graph	L5	
Tutorial 5	L6	
reliability design	L7	
Floyd-Warshall algorithm	L8	
Floyd-Warshall algorithm numerical	L9	
Floyd-Warshall algorithm numerical	L10	
Tutorial 6	L11	
Unit Test 3	L12	
Unit-4		
Backtracking concept and its examples like 8 queen's problem	L1	
Hamiltonian cycle	L2	
Graph coloring problem	L3	
examples of branch and bound method like traveling salesman problem	L4	
examples of branch and bound method	L5	
Tutorial 7	L6	
Meaning of lower bound theory and its use in solving algebraic problem	L7	
introduction to parallel algorithms	L8	
parallel algorithms	L9	
Tutorial 8	L10	
Unit Test 4	L11	
Unit 5		
Binary search trees	L1	
Binary search trees operation	L2	
height balanced trees	L3	
height balanced trees operation	L4	
height balanced trees operation	L5	
Tutorial 9	L6	
B-trees, 2-3 trees	L7	
2-3 trees	L8	
basic search and traversal techniques for trees and graphs	L9	
Tutorial 10	L10	
Unit test 5	L11	

Assignment

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

Assignment Sheet

SUBJECT NAME & CODE: ADA(CS-402)

SESSION: Jan –June 2023 *UNIT NO.:1*

ASSIGNMENT SHEET NO.:1 *DATE OF SUBMISSION* _____

Q. No.	COs (Course Outcome)	Level (Bloom's Taxonomy)	Questions Descriptions	Marks
1.	1	2(R)	**Explain** asymptotic notation with suitable example.	10
2.	1	4(A)	**Apply** the divide and conquer strategy for solving the binary search problem and **analyze** its complexity	10
3.	1	5(E)	**Prove** that Strassen's matrix multiplications advantages over ordinary matrix multiplication.	10
4.	1	4(A)	**Examine** the procedure of merge sort and sort the given array of elements using merge sort 35,10,7,22,5,13,16,3	10
5.	1	1(R)	**What** are the factors which affect the running time of an algorithm?	10
6.	1	3(A)	**Apply** the heap sort and sort the following array elements. 66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65.	10

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

Assignment Sheet

SUBJECT NAME & CODE: ADA(CS-402)

SESSION: Jan-June2023 *UNIT NO.:2*

ASSIGNMENT SHEET NO.:2 *DATE OF SUBMISSION* _____

Q. No.	Cos (Course Outcome)	Level (Bloom's Taxonomy)	Questions Descriptions	Marks
1.	1	2(R)	**Apply** the divide and conquer strategy for solving the binary search problem and **analyze** its complexity	10
2.	2	3(A)	**What** is Knapsack problem? How can we solve using greedy approach?	10
3.	2	5(E)	**Which** algorithm is good Prim's or Kruskal's? **Justify** the answer.	10
4.	2	4(A)	**Distinguish** Greedy Method and Dynamic programming.	10
5.	2	3(A)	Find an optimal code for the following set of frequencies A:50,B25,C:15,D:40,E:75 by **applying** Huffman code.	10
6.	2	3(A)	A Knapsack carry weights not exceeding 100.The weight and values of five objects are as follows: Wi:10 20 30 40 50 Pi:23 34 66 20 10 **Solve** the knapsack problem and find the maximum profit that can be obtained.	10

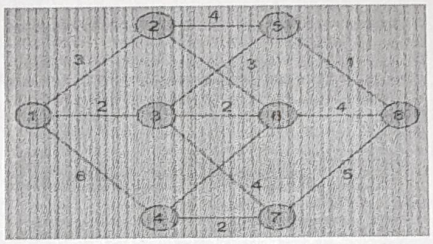
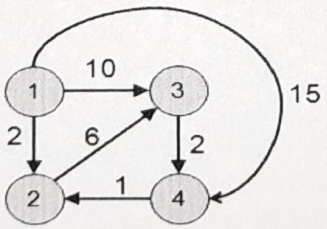
SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

Assignment Sheet

SUBJECT NAME & CODE: ADA(CS-402)

SESSION: Jan-June2023 *UNIT NO.:3*

ASSIGNMENT SHEET NO.:3 *DATE OF SUBMISSION* _____

Q. No.	Cos (Course Outcome)	Level (Bloom's Taxonomy)	Questions Descriptions	Marks
1.	CO4	L1	Define dynamic programming and there components	10
2.	CO4	L2	Differentiate dynamic programming with Greedy approach and divide-and-conquer technique?	10
3.	CO5	L3	Solve the following multistage problem. 	10
4.	CO5	L3	Use the Floyd-Warshall algorithm and find all pairs shortest paths for the following adjacency weighted matrix 	10
5.	CO4	L1	Define how knapsack problem is solved by using dynamic programming approach? Consider. N=3,(w1,w2,w3)=(2,3,3),(p1,p2,p3)=(1,2,4) and m = 6. Find optimal solution for the given data.	10
6.	CO4	L2	Explain Reliability Design.	

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

Assignment Sheet

SUBJECT NAME & CODE: ADA(CS-402)

SESSION: Jan-June2023 *UNIT NO.:4*

ASSIGNMENT SHEET NO.:4 *DATE OF SUBMISSION* _____

Q. No	Question Description	Level (Bloom's Taxonomy	COs (Course Outcomes)	Marks
1	What is backtracking? Explain recursive and non recursive backtracking.	L1	CO4	10
2	Explain 4-Queens problems with example.	L2	CO4	10
3	What is Hamiltonian cycle? Write an algorithm to find all Hamiltonian cycles in graph.	L1	CO4	10
4	Define graph Colouring problem? give an algorithm to solve this problem.	L1	CO4	10
5	Explain branch and bound method with example.	L2	CO4	10
6	Define parallel algorithm.	L1	CO4	10

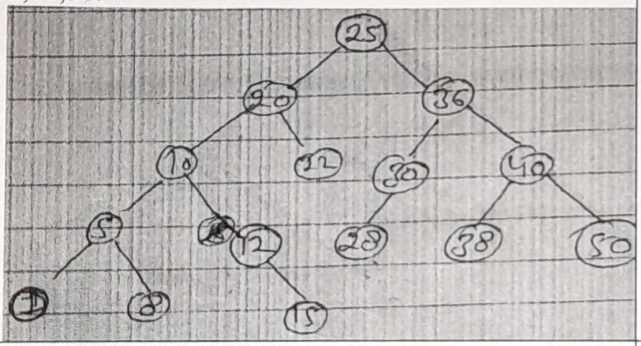
SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

Assignment Sheet

SUBJECT NAME & CODE: ADA(CS-402)

SESSION: Jan-June2023 *UNIT NO.:5*

ASSIGNMENT SHEET NO.:5 *DATE OF SUBMISSION* _____

Q. No	Question Description	Level (Bloom's Taxonomy	COs (Course Outcomes)	Marks
1	Delete the following elements from binary search tree 1,10,36. 	L3	CO5	10
2	Explain all balancing method of Height balance tree with example.	L2	CO5	10
3	Define all traversal techniques for trees with example	L2	CO5	10
4	Define insertion and deletion operation in 2-3 tree with example.	L2	CO5	10
5	Define insertion and deletion operation in B- tree with example.	L2	CO5	10
6	Explain NP-completeness with example.	L2	CO5	10

UNIT TEST

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

UNIT TEST 1

Branch: CSE
Semester: IV

Session: Jan-June2023
Date: _____

SUBJECT :Analysis and Design Of Algorithm
Max. Marks:20

Sub. Code :CS-402
Allotted Time: 50 mins.

Q. No	Question Description	Level (Bloom's Taxonomy	COs (Course Outcomes)	Marks

1	What does f(250,2) return?
	`f(m,n)`
	`{`
	`ans = 1;`
	`count = 0;`
	`while (ans <= m)`
	`{`
	`count = count + 1;`
	`ans = ans * n;`
	`}`
	`return(count)`
	`}`
		L1	CO1	10

2	You are executing an algorithm with worst-case time complexity O(n4) on a CPU that can perform 108 operations per second. if the input size is 750 Examine the most accurate time required to solve a worst case?	L4	CO1	10
3	Analyze its time complexity Binary search algorithm	L4	CO1	10

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

UNIT TEST 2

Branch: CSE **Session: Jan-June2023**

Semester: IV **Date: _____**

SUBJECT :Analysis and Design Of Algorithm **Sub. Code :CS-402**

Max. Marks:20 **Allotted Time: 50 mins.**

<table>
<tr><th>Q. No</th><th>Question Description</th><th>Level (Bloom's Taxonomy</th><th>COs (Course Outcomes)</th><th>Marks</th></tr>
<tr><td>Q.1</td><td>Define the general characteristics of Greedy algorithm.</td><td>L1</td><td>CO2</td><td>10</td></tr>
<tr><td>Q.2</td><td>Distinguish Prim's and Kruskal's algorithms</td><td>L4</td><td>CO2</td><td>10</td></tr>
<tr><td>Q.3</td><td>Apply the greedy approach to a machine-scheduling problem where dealings are involved. the details of the jobs are as follows:
<table>
<tr><th>Job</th><th>Deadline</th><th>Profit</th></tr>
<tr><td>1</td><td>1</td><td>10</td></tr>
<tr><td>2</td><td>3</td><td>75</td></tr>
<tr><td>3</td><td>2</td><td>15</td></tr>
<tr><td>4</td><td>4</td><td>40</td></tr>
</table>
Find the optimal scheduling order of the jobs considering deadline constrains.</td><td>L3</td><td>CO2</td><td>10</td></tr>
<tr><td colspan="5">OR</td></tr>
<tr><td>Q.4</td><td>A knapsack capacity is 100.the weights and values of five objects are as follows:
Weight: W=10, 20, 30, 40, 50
Value: P=20, 30, 66, 20, 60
Solve the knapsack problem and find the maximum profit that can be obtained.</td><td>L3</td><td>CO2</td><td>10</td></tr>
<tr><td>Q.5</td><td>Consider five characters (A,B,C,D,*) following probability:
<table>
<tr><td>Characters</td><td>A</td><td>B</td><td>C</td><td>D</td><td>*</td></tr>
<tr><td>Probability</td><td>0.35</td><td>0.1</td><td>0.2</td><td>0.2</td><td>0.15</td></tr>
</table>
Solve the construction of a Huffman tree and also show the efficiency.</td><td>L2</td><td>CO2</td><td>10</td></tr>
</table>

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

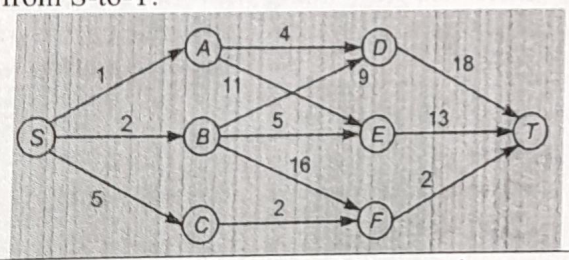
UNIT TEST 3

Branch: CSE
Semester: IV

Session: Jan-June2023
Date: _____

SUBJECT :Analysis and Design Of Algorithm
Max. Marks:20

Sub. Code :CS-402
Allotted Time: 50 mins.

Q. No	Question Description	Level (Bloom's Taxonomy	COs (Course Outcomes)	Marks
Q.1	Write the Difference between Divide-and-Conquer & Dynamic Programming	L1	CO3	10
Q.2	Solve the knapsack problem using Dynamic Programmin.Where the weight and profits are as follows (w1,w2,w3)=(2,3,3); (p1,p2,p3)=(1,2,4) capacity of knapsack is M = 6.	L3	CO3	10
Q.3	Explain Floyd-Warshall algorithm. Write its pseudo code.	L2	CO3	10
	OR			
Q.4	Explain multistage graphs and find the shortest path from S-to-T. 	L2	CO3	10
Q.5	Explain reliability design with example.	L2	CO3	10

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

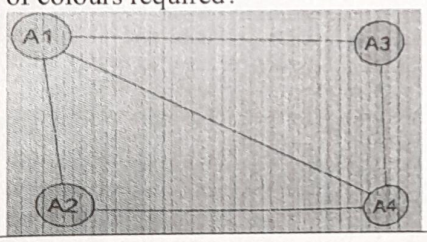
UNIT TEST 4

Branch: CSE **Session: Jan-June2023**

Semester: IV **Date: _____**

SUBJECT :Analysis and Design Of Algorithm **Sub. Code :CS-402**

Max. Marks:20 **Allotted Time: 50 mins.**

Q. No	Question Description	Level (Bloom's Taxonomy	COs (Course Outcomes)	Marks
Q.1	Explain 4-queen problem and how can we solve it using backtracking?	L2	CO4	10
	OR			
Q.2	Explain how branch and bound techniques can be used to solve travelling sales person problem.	L2	CO4	10
Q.3	What is Hamiltonian cycle? Write an algorithm to find all Hamiltonian cycle in graph?	L1	CO4	10
Q.4	Solve the sum of subsets for the following set of integers [5 10 25 50 100] for W=75	L3	CO4	10
Q.5	Colour the following graph using the vertex colouring algorithm. What is the minimum numbers of colours required? 	L3	CO4	10

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL

UNIT TEST 5

Branch: CSE **Session: Jan-June2023**

Semester: IV **Date: _____**

SUBJECT :Analysis and Design Of Algorithm **Sub. Code :CS-402**

Max. Marks:20 **Allotted Time: 50 mins.**

Q. No	Question Description	Level (Bloom's Taxonomy	COs (Course Outcomes)	Marks
Q.1.	Define B-Tree with example.	L1	CO5	10
Q.2.	Classify and explain NP complete problems with example.	L4	CO6	10
Q.3.	Distinguish the properties of BFS and DFS.	L2	CO5	10
	OR			
Q.4.	Explain 2-3 tree with the help of suitable example.	L2	CO5	10
Q.5.	Define height balance tree. Explain all the rotations perform to balance the tree with example.	L2	CO5	10

Elements a[1] and a[2] are merged to yield.
(285, 310)[179]652, 351] 423, 861, 254, 450, 520)

Then a[3] is merged with a[1:2] and
(179, 285, 310[652, 351] 423, 861, 254, 450, 520)

Next, elements a[4] & a[5] are merged.
(179, 285, 310, 351, 652, 423, 861, 254, 450, 520)

And then a[1:3] & a[4:5]
(179, 285, 310, 351, 652] 423, 861, 254, 450, 520)

Repeated recursive calls are invoked producing the following sub arrays.
(179, 285, 310, 351, 652, 423] 861] 254, 450, 520)

Elements a[6] &a[7] are merged.
Then a[8] is merged with a[6:7]
(179, 285, 310, 351, 652] 254,423, 861] 450, 520)

Next a[9] &a[10] are merged, and then a[6:8] & a[9:10]
(179, 285, 310, 351, 652, 254, 423, 450, 520, 861)

At this point there are 2 sorted sub arrays & the final merge produces the fully sorted result.
(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

If the time for the merging operations is proportional to 'n', then the computing time for merge sort is described by the recurrence relation.

$T(n) = \begin{cases} a & n=1, \text{'a' a constant} \\ 2T(n/2)+cn & n>1, \text{'c' a constant} \end{cases}$

- When 'n' is a power of 2, n= 2^k, we can solve this equation by successive substitution.

$T(n) = 2(2T(n/4) + cn/2) + cn$
$= 4T(n/4)+2cn$
$= 4(2T(n/8)+cn/4)+2cn$
$\vdots$
$= 2^k T(1)+kCn.$
$= an + cn \log n.$

- It is easy to see that if $2^k \le n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore,

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

Recurrence Relation

Change of variables:

- It is sometimes possible to solve more complicated recurrences by making a change of variable.
- In the following example, we write $T(n)$ for the term of a general recurrences, and t_i for the term of a new recurrence obtained from the first by a change of variable.

Example: (1) ----------

Consider the recurrence,

$$T(n) = \begin{cases} 1 & \text{, if } n=1 \\ 3T(n/2) + n & \text{, if 'n' is a power of 2, } n>1 \end{cases}$$

→ Reconsider the recurrence we solved by intelligent guesswork in the previous section, but only for the case when 'n' is a power of 2

$$T(n) = \begin{cases} 1 \\ 3T(n/2) + n \end{cases}$$

- We replace 'n' by 2^i.
- This is achieved by introducing new recurrence t_i, define by $t_i = T(2^i)$
- This transformation is useful because $n/2$ becomes $(2^i)/2 = 2^{i-1}$
- In other words, our original recurrence in which $T(n)$ is defined as a function of $T(n/2)$ given way to one in which t_i is defined as a function of t_{i-1}, precisely the type of recurrence we have learned to solve.

$t_i = T(2^i) = 3T(2^{i-1}) + 2^i$

$t_i = 3t_{i-1} + 2^i$

$t_i - 3t_{i-1} = 2^i$ →(A)

In this case,

$b = 2$, $p(n) = 1$, degree = 0

So, the characteristic equation,

$(x - 3)(x - 2) = 0$

The roots are, $r1 = 3$, $r2 = 2$.

The general equation,

$t_n = C_1 r_1^i + C_2 r_2^i$

sub. r_1 & r_2: $t_n = 3^n C_1 + C_2 2^n$

$t_i = C_1 3^i + C_2 2^i$

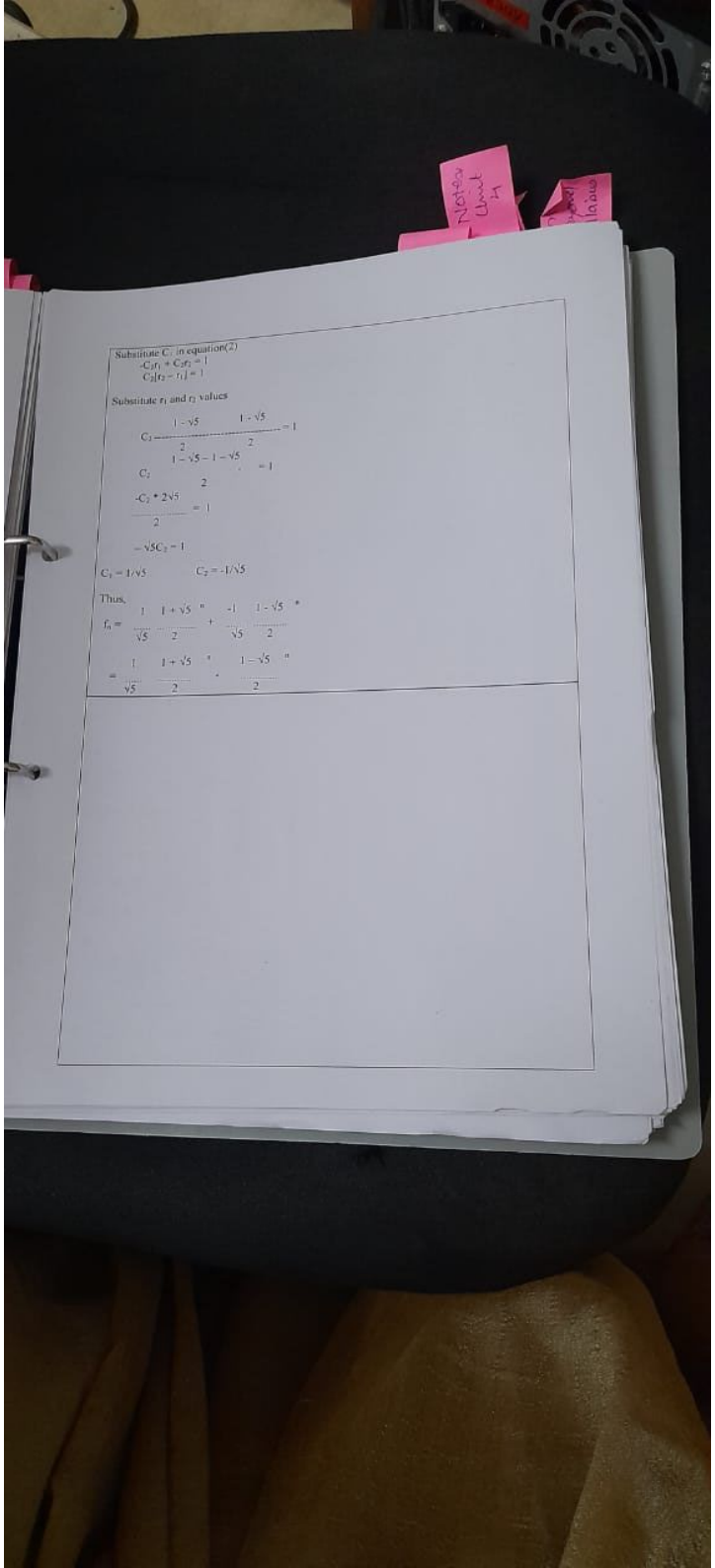
We use the fact that, $T(2^i) = t_i$ & thus $T(n) = t\log n$ when $n = 2^i$ to obtain,

$T(n) = C_1 \cdot 3^{\log_2 n} + C_2 \cdot 2^{\log_2 n}$

$T(n) = C_1 \cdot n^{\log_2 3} + C_2 n$ [$i = \log n$]

When 'n' is a power of 2, which is sufficient to conclude that,

$T(n) = O(n^{\log_2 3})$ 'n' is a power of 2


Substitute C_1 in equation(2)
$-C_1r_1 + C_2r_2 = 1$
$C_2[r_2 - r_1] = 1$
Substitute r_1 and r_2 values
$C_2 \frac{1-\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2} = 1$
$C_2 \frac{1-\sqrt{5}-1+\sqrt{5}}{2} = 1$
$\frac{-C_2 * 2\sqrt{5}}{2} = 1$
$-\sqrt{5}C_2 = 1$
$C_1 = 1/\sqrt{5}$ $C_2 = -1/\sqrt{5}$
Thus,
$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{-1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$
$= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n$

Strassen's matrix multiplication

STRASSON'S MATRIX MULTIPLICAION

- Let A and B be the 2 n*n Matrix. The product matrix C=AB is calculated by using the formula.
  C (i, j)= A(i,k) B(k,j) for all 'i' and and j between 1 and n.
- The time complexity for the matrix Multiplication is $O(n^3)$.
- Divide and conquer method suggest another way to compute the product of n*n matrix.
- We assume that N is a power of 2. In the case N is not a power of 2 ,then enough rows and columns of zero can be added to both A and B ,SO that the resulting dimension are the powers of two.
- If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of A & B.
- If n>2,Then the elements are partitioned into sub matrix n/2*n/2..since 'n' is a power of 2 these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until 'N" become suitable small(n=2) so that the product is computed directly .
- The formula are

$$\begin{matrix} A11 & A12 \\ A21 & A21 \end{matrix} * \begin{matrix} B11 & B12 \\ B21 & B22 \end{matrix} = \begin{matrix} C11 & C12 \\ C21 & C22 \end{matrix}$$

C11 = A11 B11 + A12 B21
C12 = A11 B12 + A12 B22
C21 = A21 B11 + A22 B21
C22 = A21 B12 + A22 B22

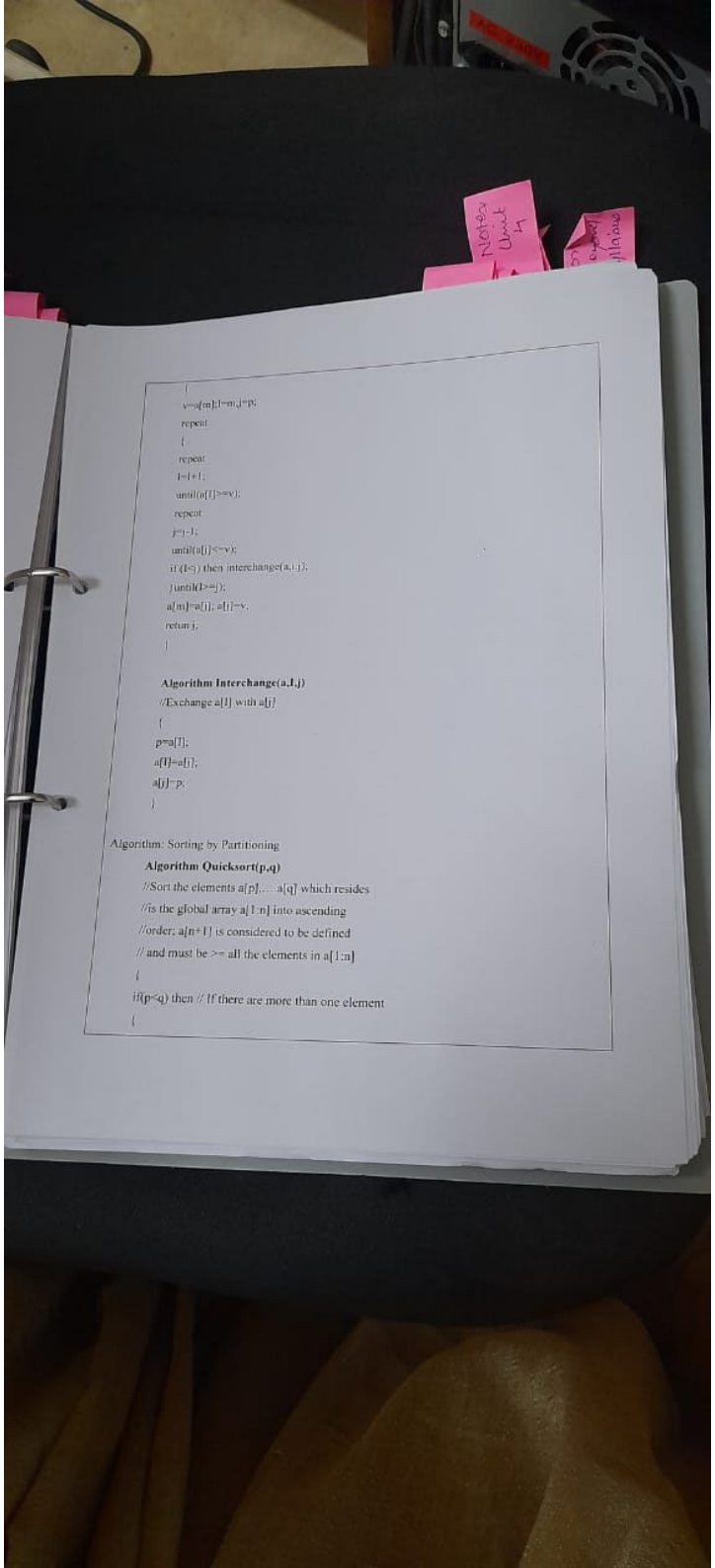
For EX:

4 * 4 = $\begin{pmatrix} 2&2&2&2 \\ 2&2&2&2 \\ 2&2&2&2 \\ 2&2&2&2 \end{pmatrix} * \begin{pmatrix} 1&1&1&1 \\ 1&1&1&1 \\ 1&1&1&1 \\ 1&1&1&1 \end{pmatrix}$

The Divide and conquer method

$$\left(\begin{array}{cc|cc} 2&2&2&2 \\ 2&2&2&2 \\ \hline 2&2&2&2 \\ 2&2&2&2 \end{array}\right) * \left(\begin{array}{cc|cc} 1&1&1&1 \\ 1&1&1&1 \\ \hline 1&1&1&1 \\ 1&1&1&1 \end{array}\right) = \left(\begin{array}{cc|cc} 4&4&4&4 \\ 4&4&4&4 \\ \hline 4&4&4&4 \\ 4&4&4&4 \end{array}\right)$$

- To compute AB using the equation we need to perform 8 multiplication of n/2*n/2 matrix and from 4 addition of n/2*n/2 matrix.
- C_{ij} are computed using the formula in equation →4




```
    {
    v=a[m];i=m;j=p;
    repeat
    {
    repeat
    i=i+1;
    until(a[i]>=v);
    repeat
    j=j-1;
    until(a[j]<=v);
    if (i<j) then interchange(a,i,j);
    }until(i>=j);
    a[m]=a[j]; a[j]=v;
    return j;
    }

    Algorithm Interchange(a,I,j)
    //Exchange a[I] with a[j]
    {
    p=a[I];
    a[I]=a[j];
    a[j]=p;
    }
```

Algorithm: Sorting by Partitioning

```
    Algorithm Quicksort(p,q)
    //Sort the elements a[p],....a[q] which resides
    //is the global array a[1:n] into ascending
    //order; a[n+1] is considered to be defined
    // and must be >= all the elements in a[1:n]
    {
    if(p<q) then // If there are more than one element
    {
```

```
// divide p into 2 subproblems
j=partition(a,p,q+1);
//'j' is the position of the partitioning element.
//solve the subproblems.
quicksort(p,j-1);
quicksort(j+1,q);
//There is no need for combining solution.
}
}
```

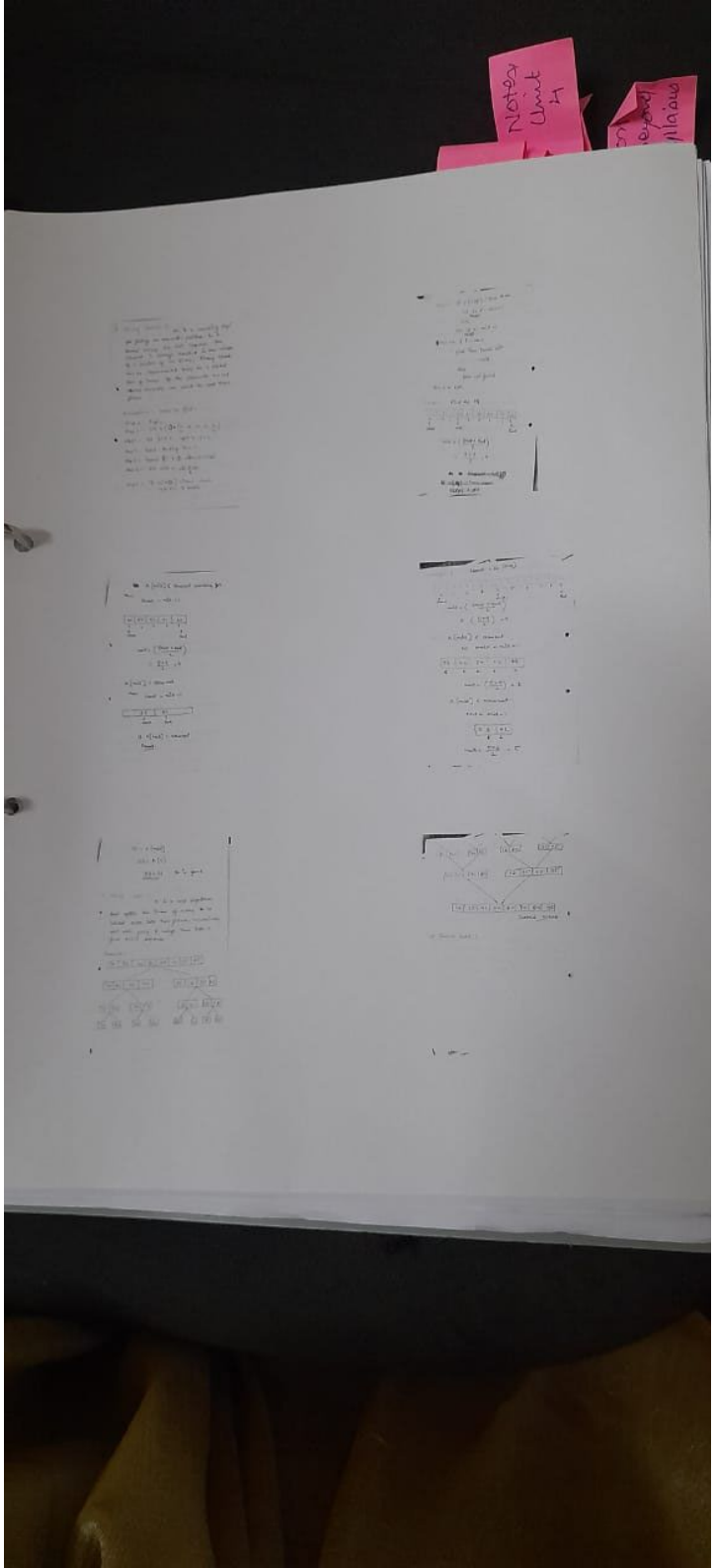
Complexity: O(n log n)- Best and Average case

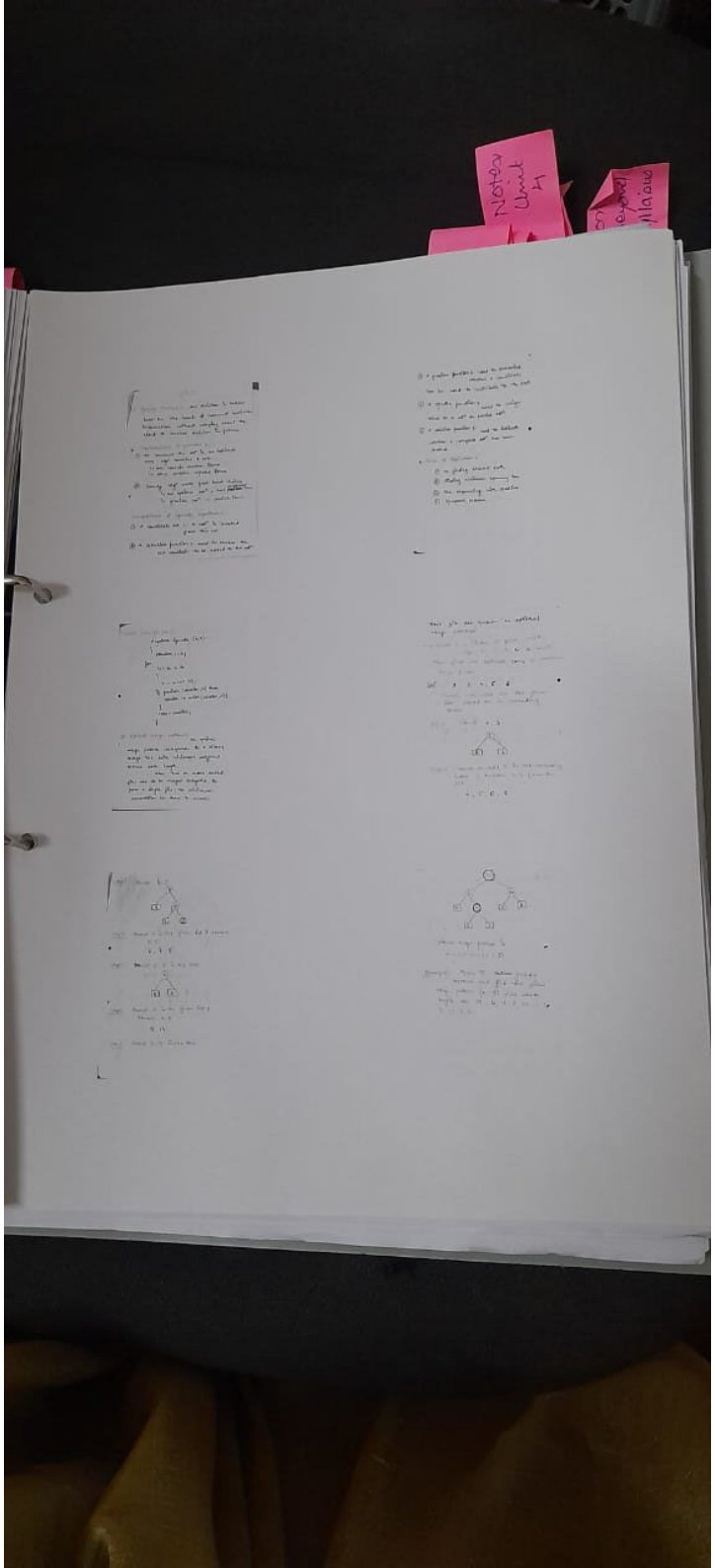
O(n^2) in worst case

Example:

As an example of how PARTITION works consider the following array of 9 elements. The procedure is initially invoked as call PARTITION (1, 10). The vertical bars connected by a horizontal line indicate those elements which were interchanged to produce the next row. A(1) = 65 is the partitioning element and it is eventually (in the sixth row) determined to be the 5th smallest element of the set. Notice that the remaining elements are unsorted but they are partitioned about A(5) = 65.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	+ ∞	2	9
65	45	75	80	85	60	55	50	70	+ ∞	3	8
65	45	50	80	85	60	55	75	70	+ ∞	4	7
65	45	50	55	85	60	80	75	70	+ ∞	5	6
65	45	50	55	60	85	80	75	70	+ ∞	6	5
60	45	50	55	65	85	80	75	70	+ ∞		


Notes
Unit
4



Knapsack problem

Problem:

- input:
  - ✓ n objects.
  - ✓ each object i has a weight w_i and a profit p_i
  - ✓ Knapsack : M
- output:
  - ✓ Fill up the Knapsack s.t. the total profit is maximized.
  - ✓ Feasible solution: $(x_1,\ldots\ldots,x_n)$.

Formally,

✓ Let x_i be the fraction of object i placed in the Knapsack, $0 \le x_i \le 1$. For $1 \le I \le n$.

✓ Then :

$$P = \sum_{1\le i\le n} p_i x_i$$

$$\text{And} \quad \sum_{1\le i\le n} w_i x_i \le M$$

Assumptions:

- $\sum_{i=1}^{n} w_i > M$; not all $x_i = 1$.
- $\sum_{1\le i\le n} w_i x_i = M$

1 an unsuccessful search.

- Table. Shows the traces of Bin search on these 3 steps.

X=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			Found

x=-14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	Not found

x=9	low	high	mid
	1	14	7
	1	6	3
	4	6	5
			Found

Theorem: Algorithm Binsearch(a,n,x) works correctly.

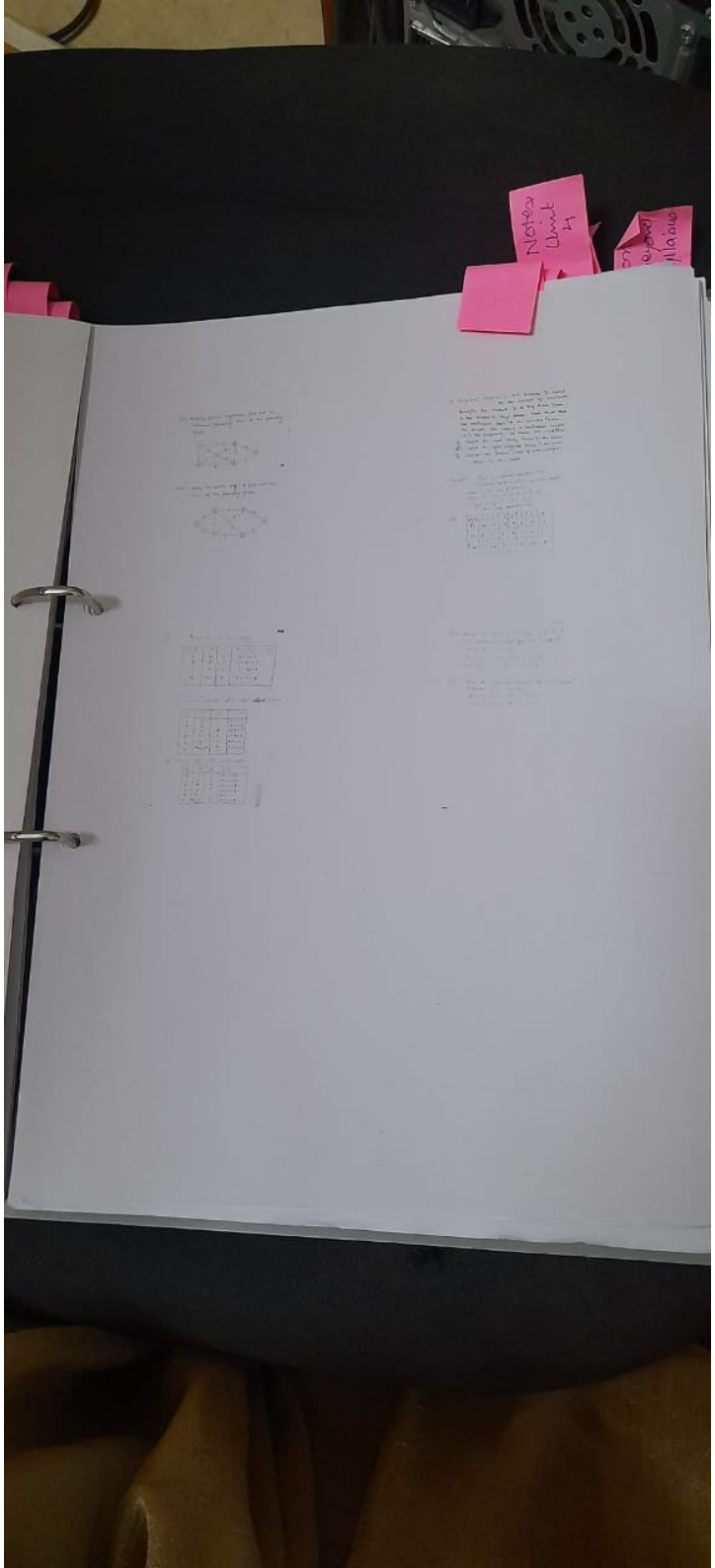
Proof:
We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

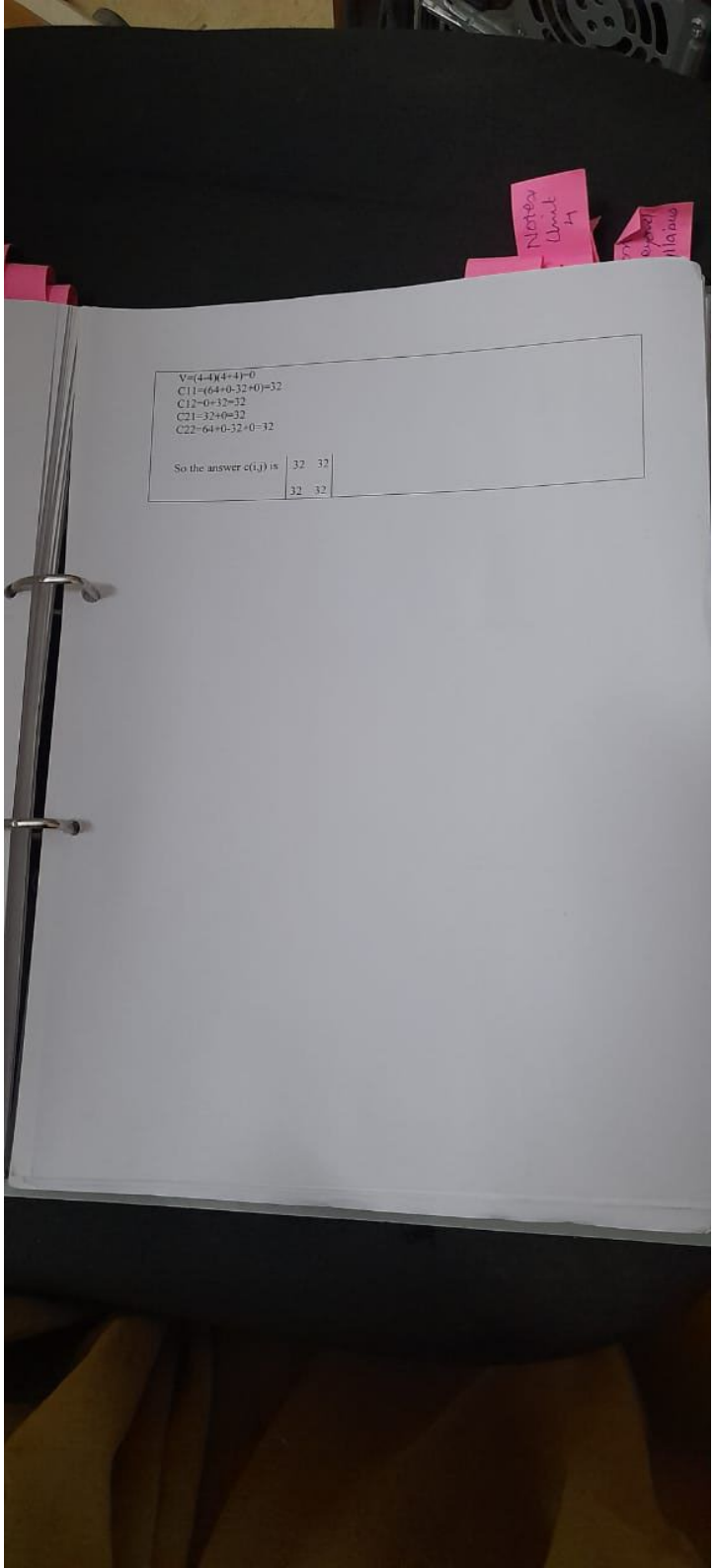
- Initially low =1, high= n,n>=0, and a[1]<=a[2]<= ..... <=a[n]
- If n=0, the while loop is not entered and is returned.
- Otherwise we observe that each time thro' the loop the possible elements to be checked of or equality with x and a[low], a[low+1], ... a[mid], ...a[high].
- If x=a[mid], then the algorithm terminates successfully.
- Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
- Clearly, this narrowing of the range does not affect the outcome of the search.
- If low becomes > than high, then 'x' is not present & hence the loop is exited.

Complexity: O(log n)

UNIT-2

Notes
Unit
2


Notes
Unit
4


Notes Unit 4
V=(4-4)(4+4)=0
C11=(64+0-32+0)=32
C12=0+32=32
C21=32+0=32
C22=64+0-32+0=32
So the answer c(i,j) is
32 32
32 32

Merge sort

MERGE SORT

- As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n)
- This algorithm is called merge sort
- We assume throughout that the elements are to be sorted in non-decreasing order.
- Given a sequence of 'n' elements a[1],....a[n] the general idea is to imagine then split into 2 sets a[1], ...a[n/2] and a[n/2]+1], ...a[n]
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of 'n' elements.
- Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Algorithm For Merge Sort:

```
Algorithm MergeSort(low,high)
//a[low high] is a global array to be sorted
//Small(P) is true if there is only one element
//to sort. In this case the list is already sorted.
{
if (low<high) then //if there are more than one element
{
//Divide P into subproblems
//find where to split the set
mid = [(low+high)/2];
//solve the subproblems.
mergesort (low,mid);
mergesort(mid+1,high);
//combine the solutions .
merge(low,mid,high);
}
}
```

- Consider the array of 10 elements a[1:10] =(310, 285. 179, 652, 351, 423, 861, 254, 450, 520)

- Algorithm Mergesort begins by splitting a[] into 2 sub arrays each of size five (a[1:5] and a[6:10]).
- The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3]) and 2(a[4:5])
- Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])
- The 2 values in a[1:2] are split to find time into one-element sub arrays, and now the merging begins.

  (310|285| 179| 652, 351| 423, 861, 254, 450, 520)

  Where vertical bars indicate the boundaries of sub arrays.

- **Analysis:**

T= O (n-1) * max (O (Least), O (Insert)).

- Case 1 L is not sorted.
O (Least)= O (n).
O (Insert)= O (1).

$\Rightarrow$ T= O (n_2).

- Case 2 L is sorted.

Case 2.1

O (Least)= O (1)
O (Insert)= O (n)

$\Rightarrow$ T= O (n_2)

Case 2.2

L is represented as a min-heap. Value in the root is $\leq$ the values of its children.

O (Least)= O (1)
O (Insert)= O (log n)

$\Rightarrow$ T= O (n log n).

Greedy Method

Objective:

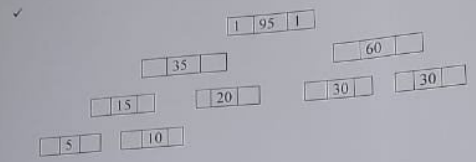
General approach:

- Given a set of n inputs.
- Find a subset, called feasible solution, of the n inputs subject to some constraints, and satisfying a given objective function.
- If the objective function is maximized or minimized, the feasible solution is optimal.

- It is a locally optimal method.

Algorithm:

- Step 1: Choose an input from the input set, based on some criterion. If no more input exit.
- Step 2: Check whether the chosen input yields to a feasible solution. If no, discard the input and goto step 1.
- Step 3: Include the input into the solution vector and update the objective function. Goto step 1.



- **Algorithm:**

  - Least (L): find a tree in L whose root has the smallest weight.
  - Function : Tree (L,n).

```
Integer i;
Begin
For i=1 to n -1 do
    Get node (T)        /* create a node pointed by T */
    Left child (T)= Least (L)      /* first smallest */
    Right child (T)= Least (L)     /* second smallest */
    Weight (T) = weight (left child (T))
                + weight (right child (T))
    Insert (L,T);       /* insert new tree with root T in L */
End for
Return (Least (L))  /* tree left in L */
End.
```

Kruskal's algorithm

Problem:

- T form a forest.
- The edge e added to T is always least-weight edge in the graph that connects two distinct trees of T.
- At the end of the algorithm T becomes a single tree.

Example:

Notes
Unit
4

Largest profit-weight ratio strategy:

- ✓ Order profit-weight ratios of all objects.
- ✓ $P_i/w_i \geq (p_i+1)/(w_i+1)$ for $1 \leq i \leq n-1$
- ✓ Pick the object with the largest p/w
- ✓ If the weight of the object exceeds the remaining knapsack capacity, take a fraction of the object.

Example:

$P_1/w_1=25/18=1.389$
$P_2/w_2=24/15=1.6$
$P_3/w_3=15/10=1.5$

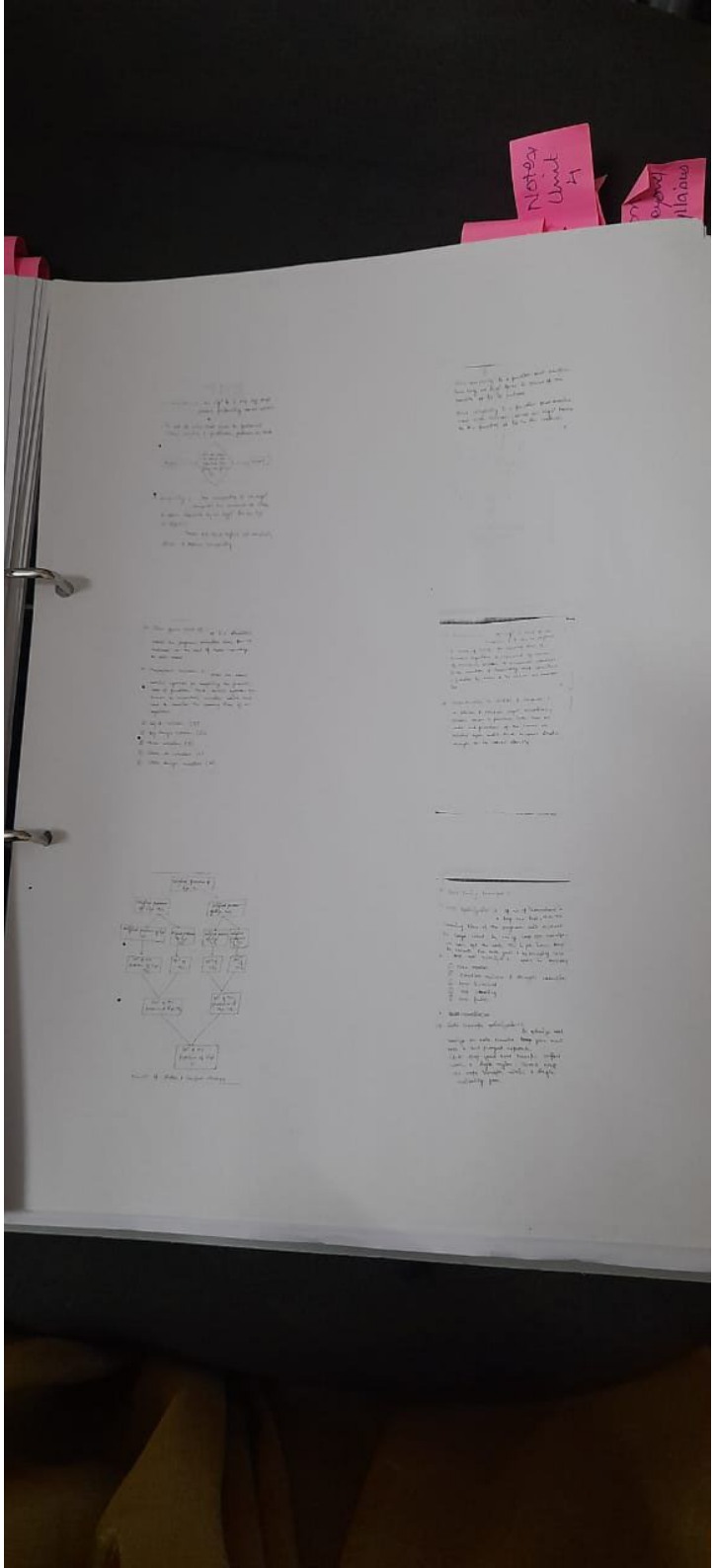
→ $P_2/w_2>=P_2/w_2>= P_3/w_3$

Cu=20; p=0

- ✓ Pick object 2
  Since cu ≥ w_2 then $x_2=1$
  cu=20-15=5 and p=24

- ✓ Pick object 3
  Since cu<w_3 then $x_3=cu/w_3=5/10=1/2$
  cu= 0 and P= 24+1/2.15=24+7.5=31.5

- ✓ Feasible solution (0,1,1/2) p=31.5


Notes
Unit
4

Example:

- Let's start form v= 1

Merge the first two files:

(5, 10, 20, 30, 30) ➔ (15, 20, 30, 30)

Merge the next two files:

(15, 20, 30, 30) ➔ (30, 30, 35)

Merge the next two files:

(30, 30, 35) ➔ (35, 60)

Merge the last two files:

(35,60) ➔ (95)

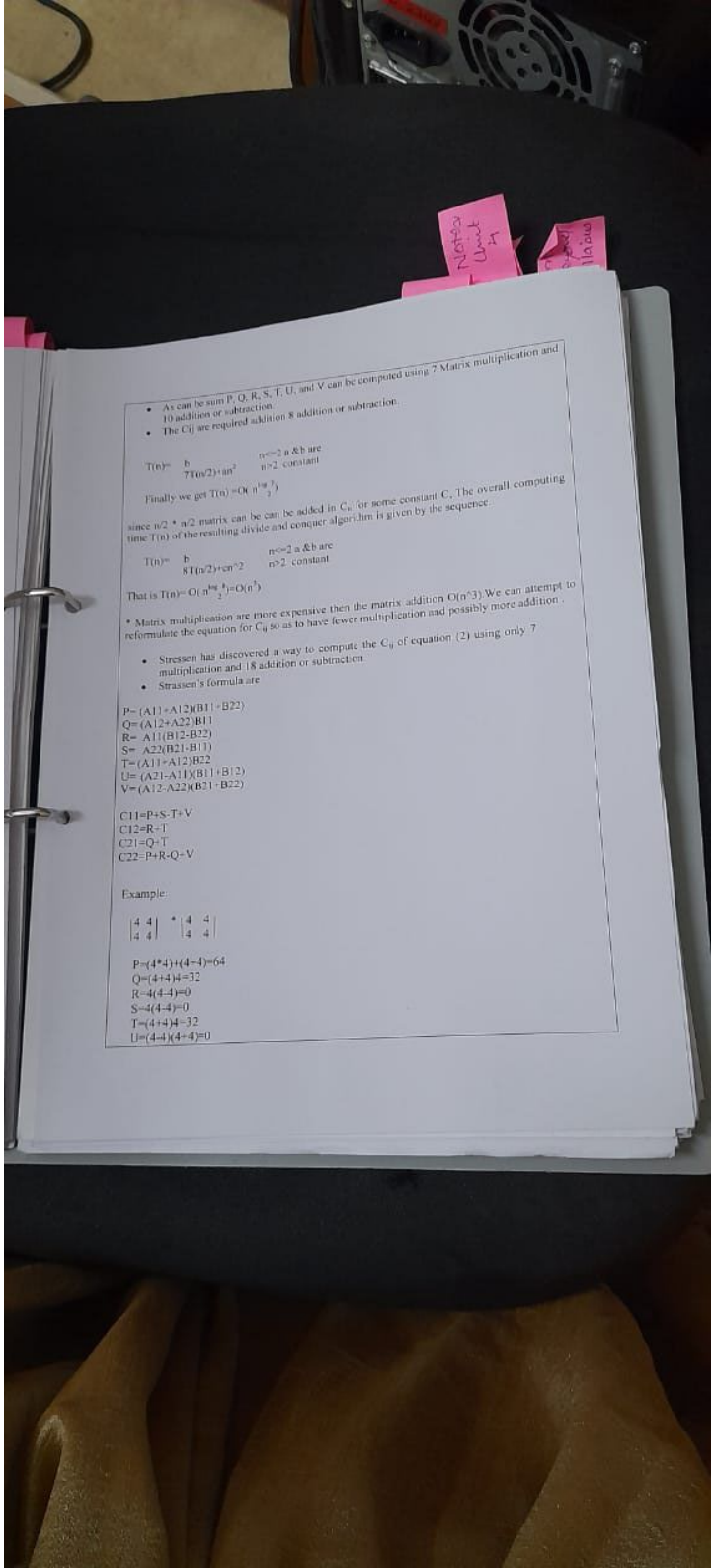
Total time: 15 + 35 + 60 + 95 = 205

⇒ This is called a 2-way merge pattern.

- **Problem:**
  - ✓ Given n sorted files
  - ✓ Merge n files in a minimum amount of time.

- **Algorithm**:
  - ✓ We associate with each file a node

Left	Weight	Right



- As can be sum P, Q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.
- The Cij are required addition 8 addition or subtraction.

$$T(n)=\begin{cases} b & n<=2 \text{ a \&b are} \\ 7T(n/2)+an^2 & n>2 \text{ constant} \end{cases}$$

Finally we get $T(n)=O(n^{\log_2 7})$

since n/2 * n/2 matrix can be can be added in C_n for some constant C, The overall computing time T(n) of the resulting divide and conquer algorithm is given by the sequence.

$$T(n)=\begin{cases} b & n<=2 \text{ a \&b are} \\ 8T(n/2)+cn^2 & n>2 \text{ constant} \end{cases}$$

That is $T(n)=O(n^{\log_2 8})=O(n^3)$

• Matrix multiplication are more expensive then the matrix addition $O(n^3)$.We can attempt to reformulate the equation for C_{ij} so as to have fewer multiplication and possibly more addition .

- Stressen has discovered a way to compute the C_{ij} of equation (2) using only 7 multiplication and 18 addition or subtraction.
- Strassen's formula are

P= (A11+A12)(B11+B22)
Q= (A12+A22)B11
R= A11(B12-B22)
S= A22(B21-B11)
T= (A11+A12)B22
U= (A21-A11)(B11+B12)
V= (A12-A22)(B21+B22)

C11=P+S-T+V
C12=R+T
C21=Q+T
C22=P+R-Q+V

Example:

$$\begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} * \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix}$$

P=(4*4)+(4+4)=64
Q=(4+4)4=32
R=4(4-4)=0
S=4(4-4)=0
T=(4+4)4=32
U=(4-4)(4+4)=0

Example:

- ✓ 3 objects (n=3).
- ✓ $(w_1,w_2,w_3)=(18,15,10)$
- ✓ $(p_1,p_2,p_3)=(25,24,15)$
- ✓ M=20

Largest-profit strategy: (Greedy method)

- ✓ Pick always the object with largest profit.
- ✓ If the weight of the object exceeds the remaining Knapsack capacity, take a fraction of the object to fill up the Knapsack.

Example:

- ✓ P=0 , C=M=20 /* remaining capacity */
- ✓ Put object 1 in the Knapsack.

  P=25 Since $w_1 < M$ then $x_1=1$
  C=M-18=20-18=2
- ✓ Pick object 2

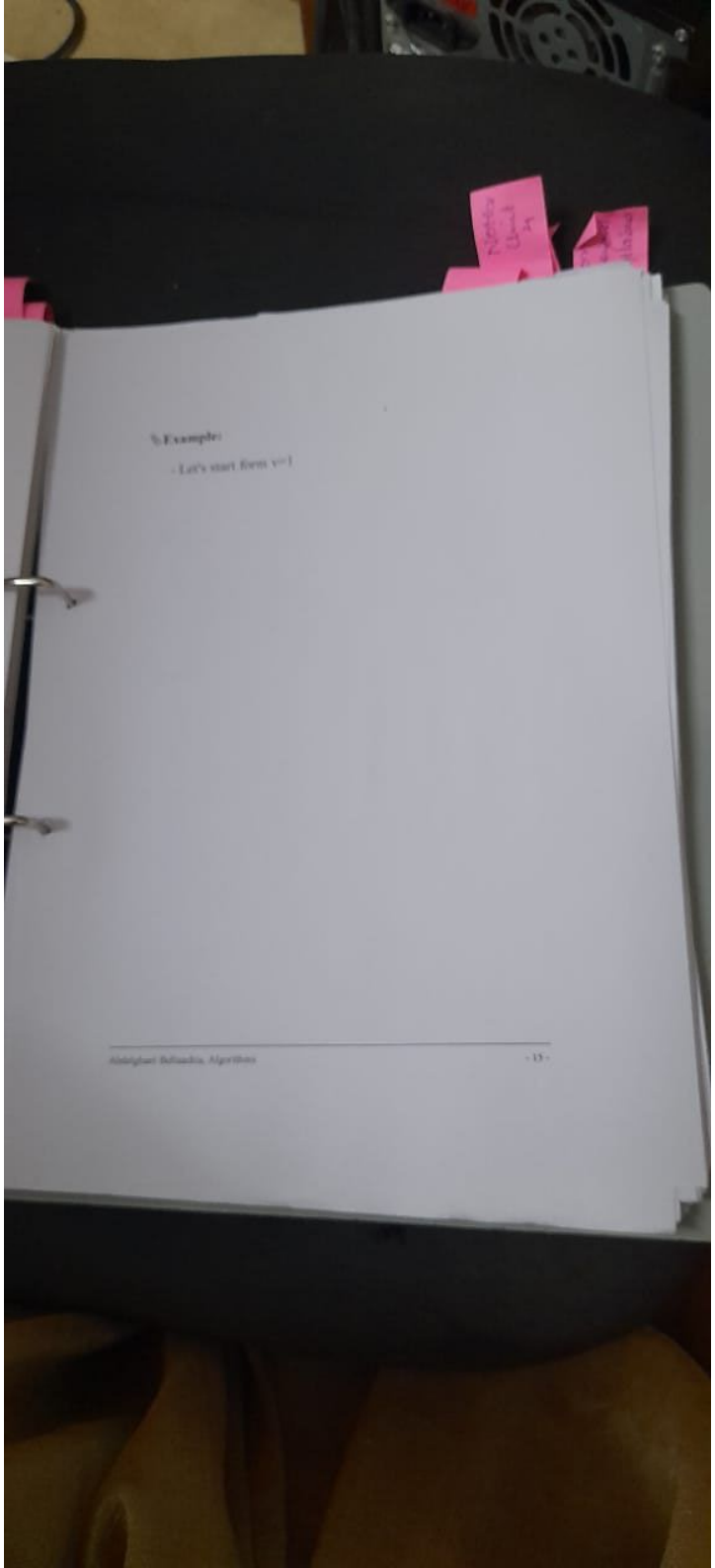
  Since $C< w_2$ then $x_2= C/w_2=2/15$.
  P=25+2/15*24 =25+3.2=28.2
- ✓ Since the Knapsack is full then $x_3=0$.
- ✓ The feasible solution is (1, 2/15,0).

Smallest-weight strategy:

- ✓ be greedy in capacity: do not want to fill the knapsack quickly.
- ✓ Pick the object with the smallest weight.
- ✓ If the weight of the object exceeds the remaining knapsack capacity, take a fraction of the object.

Example:

- ✓ cu=M=20
- ✓ Pick object 3
  Since $w_3 < cu$ then $x_3=1$
  P= 15 cu =20-10 = 10 , x_3 =1
- ✓ Pick object 2
  Since $w_2 > cu$ then $x_2 = 10/15 = 2/3$
  P = 15+ 2/3.24
  = 15+ 16 = 31 cu= 0.
- ✓ Since cu=0 then $x_1=0$
- ✓ Feasible solution : (0,2/3,1) p=31.

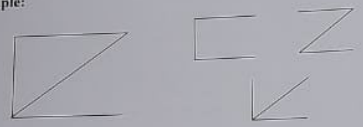

Example:
- Let's start form v=1

Minimum Spanning Tree.

Definition:

Let G=(V,E) be an undirected connected graph.
T=(V,E') is a spanning tree iff T is a tree.

Example:



Definition:

- If each edge of E has a weight, G is called a weighted graph.

Problem:

- Given an undirected, connected, weighted graph G=(V,E).
- We wish to find an acyclic subset $T \subseteq E$ that connects all the vertices and whose total weight:

$$w(T)= \sum_{(u,v)\in T} w(u,v) \text{ is minimized.}$$

Where w(u,v) is the weight of edge (u,v).

- T is called a minimum spanning tree of G.

Optimal merge patterns

- Introduction:

- Merge two files each has n & m elements, respectively:
  ⇒ takes O (n+m).

- Given n files
  What's the minimum time needed to merge all n files?

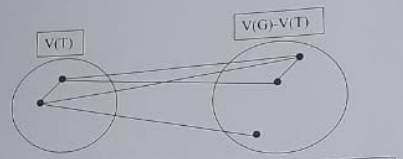
- Example:
  $(F_1, F_2, F_3, F_4, F_5) = (20, 30, 10, 5, 30)$.

$M_1 = F_1 \& F_2 \quad \Rightarrow 20+30 = 50$
$M_2 = M_1 \& F_3 \quad \Rightarrow 50+10 = 60$
$M_3 = M_2 \& F_4 \quad \Rightarrow 60+5 = 65$
$M_4 = M_3 \& F_5 \quad \Rightarrow 65+30 = 95$
270

- **Optimal merge pattern**: Greedy method.

  Sort the list of files:

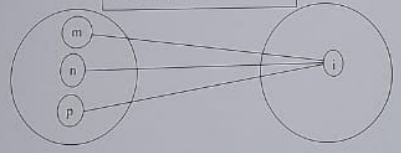
  $(5,10, 20, 30, 30) = (F_4, F_3, F_1, F_2, F_5)$

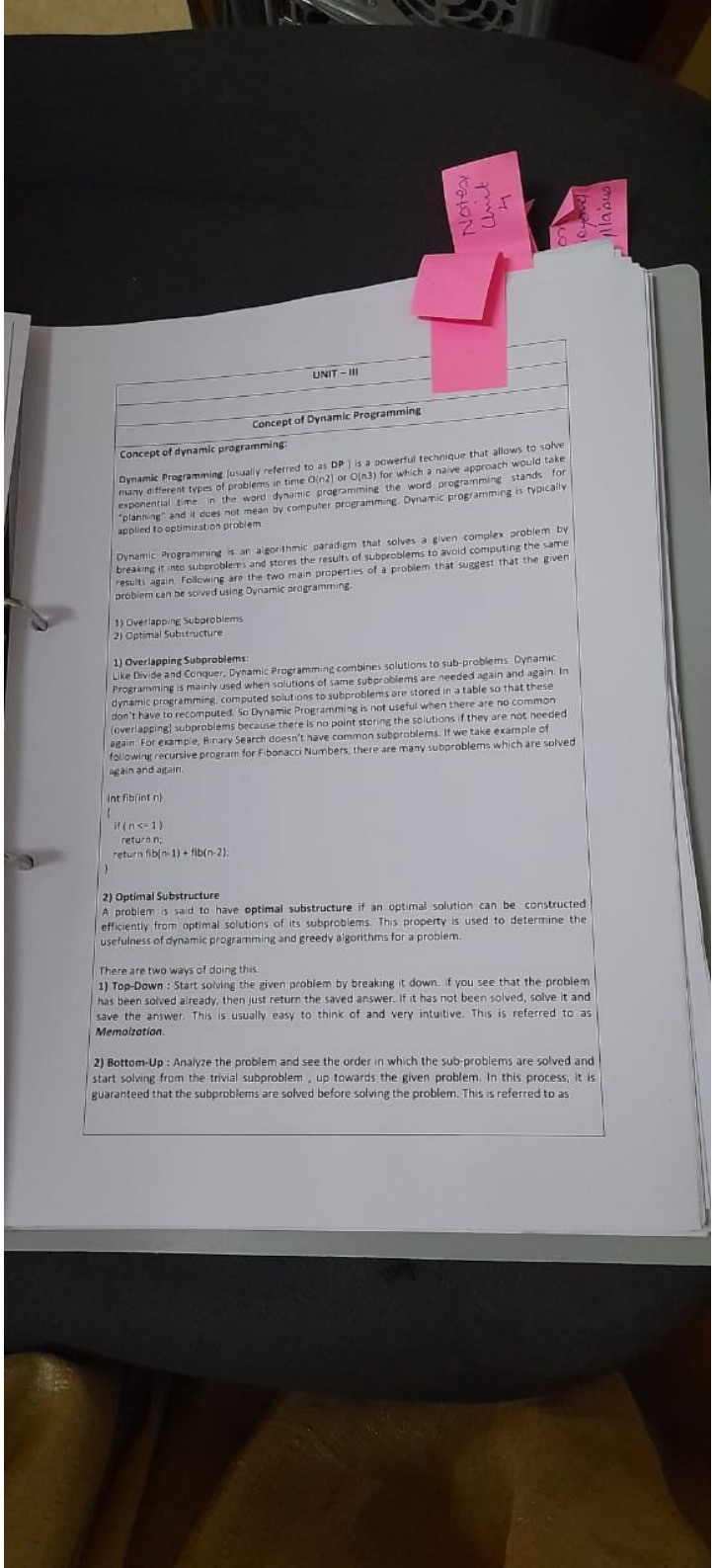


NEAR(I)=0
i∈V(T)

NEAR(I)=v s.t. i∈V(T), v∈V(T) and cost(i,v) is min among all choices for NEAR(i).

NEAR(I) =p s.t. cost(I,p) is min cost(I,w) where p<=w<=m





UNIT – III

Concept of Dynamic Programming

Concept of dynamic programming:

Dynamic Programming (usually referred to as **DP**) is a powerful technique that allows to solve many different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time. In the word dynamic programming the word programming stands for "planning" and it does not mean by computer programming. Dynamic programming is typically applied to optimization problem.

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Subproblems
2) Optimal Substructure

1) Overlapping Subproblems:
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
int fib(int n)
{
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```

2) Optimal Substructure
A problem is said to have **optimal substructure** if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.

There are two ways of doing this.
1) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***.

2) Bottom-Up : Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem , up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as

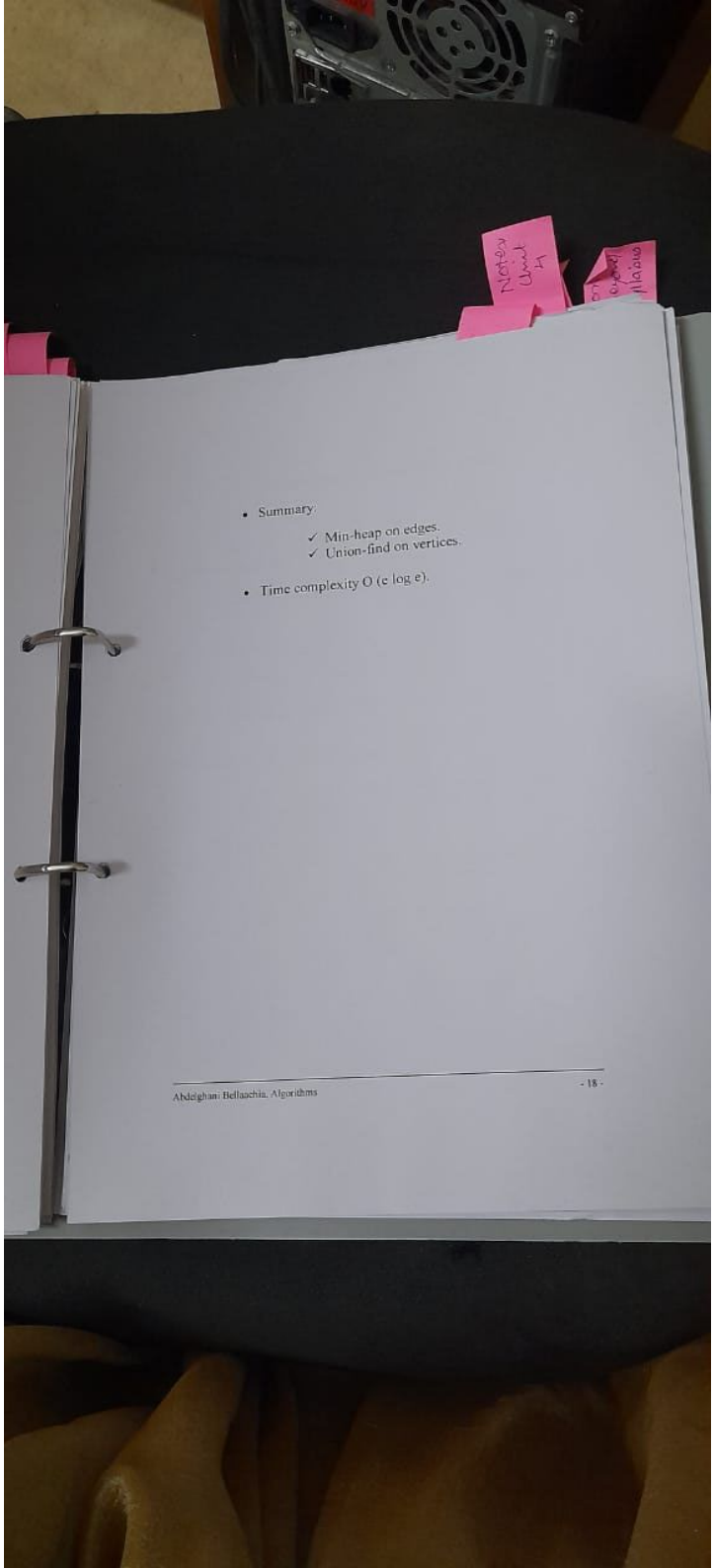
Single Source Shortest Paths.

Requirements:

- Given a weighted digraph G= (V,E) where the weights are >0.
- A source vertex, $v_o \in V$.
- Find the shortest path from v_o to all other nodes in G.
- Shortest paths are generated in increasing order: 1,2,3,.....

Algorithm Description: Dijkstra

- S: Set of vertices (including v_o) whose final shortest paths from the source v_o have already been determined.
- For each node w∈ V-S,
  Dist (w): the length of the shortest path starting from v_o going through only vertices which are in S and ending at w.
- The next path is generated as follows:
  - It's the path of a vertex u which has Dist (u) minimum among all vertices in V-S
  - Put u in S.
- Dist (w) for w in V-S may be decreased going though u.



- Summary:
  - ✓ Min-heap on edges.
  - ✓ Union-find on vertices.
- Time complexity O (e log e).

```
U=n;
  for i=1 to n do
  {
  if (w[i]>u)then break;
  x[i]=1.0;U=U-w[i]
  }
if(i<=n)then x[i]=U/w[i];
}
```

Example:

Capacity=20
N=3 ,M=20
Wi=18,15,10
Pi=25,24,15

Pi/Wi=25/18=1.36, 24/15=1.6, 15/10=1.5

Descending Order → Pi/Wi→ 1.6 1.5 1.36
Pi = 24 15 25
Wi = 15 10 18
Xi = 1 5/10 0

Pi*Xi= 1*24+0.5*15→31.5

The optimal solution is →31.5

X1	*X2*	*X3*	*WiXi*	*PiXi*
½	1/3	¼	16.6	24.25
1	2/5	0	20	18.2
0	2/3	1	20	31
0	1	½	20	31.5

Of these feasible solutions Solution No 4 yields the Max profit. And this solution is optimal for the given problem instance

Tree Vertex Splitting

Tree vertex splitting

- Directed and weighted binary tree
- Consider a network of power line transmission
- The transmission of power from one node to the other results in some loss, such as drop in voltage
- Each edge is labeled with the loss that occurs (edge weight)
- Network may not be able to tolerate losses beyond a certain level
- You can place boosters in the nodes to account for the losses

Definition 1 Given a network and a loss tolerance level, the tree vertex splitting problem is to determine the optimal placement of boosters. We can place boosters only in the vertices and now here else.

Let T = (V; E; w) be a weighted directed tree

- V is the set of vertices
- E is the set of edges
- w is the weight function for the edges
- wij is the weight of the edge hi; ji 2 E
- We say that $w_i = \infty$ if $\langle i,j \rangle \notin E$
- A vertex with in-degree zero is called a source vertex
- A vertex with out-degree zero is called a sink vertex
- For any path P 2 T, its delay d(P) is defined to be the sum of the weights (wij) of that path, or

$$d(P) = \sum_{(i,j) \in P} w_{ij}$$

- Delay of the tree T, d(T) is the maximum of all path delays
- Splitting vertices to create forest
- Let T/X be the forest that results when each vertex u ∈ X is split into two nodes u^i and u^o such that all the edges $\langle u, j \rangle \in E$ [$\langle j,u \rangle \in E$] are replaced by edges of the form $\langle u^o, j \rangle \in E$ [$\langle j,u^i \rangle \in E$]
  - Outbound edges from u now leave from u^o
  - Inbound edges to u now enter at u^i
- Split node is the booster station
- Tree vertex splitting problem is to identify a set $X \subseteq V$ of minimum cardinality (minimum number of booster stations) for which $d(T/X) \leq \delta$ for some specifiedtolerance limit δ.
  - TVSP has a solution only if the maximum edge weight is ≤δ
- Given a weighted tree T = (V,E,w) and a tolerance limit δ, any $X \subseteq V$ is a feasible solution if $d(T/X) \leq \delta$
  - Given an X, we can compute d(T/X) in O(|V|) time

Solution:

- Using greedy method.
- Two algorithms:
  - ✓ Prim's algorithm.
  - ✓ Kruskal's algorithm.

Approach:

- The tree is built edge by edge.
- Let T be the set of edges selected so far.
- Each time a decision is made:
  - Include an edge e to T s.t. :
    Cost (T)+w (e) is minimized, and
    T∪{e} does not create a cycle.

Prim's algorithm:

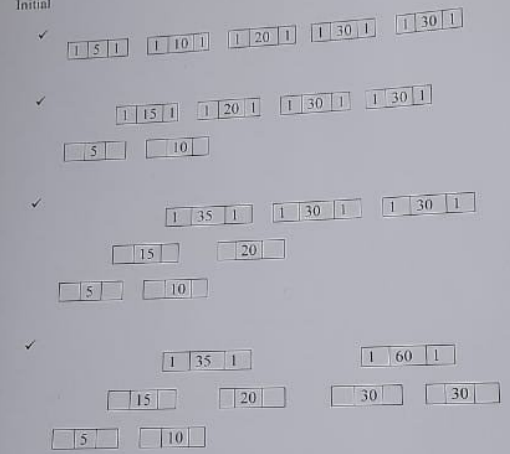
- T forms a single tree.
- The edge e added to T is always least-weight edge connecting the tree, T, to a vertex not in the tree

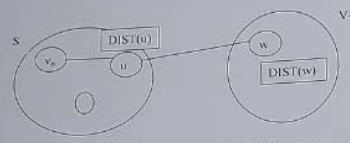
Implementation:

- To choose the next edge to be included in T, NEAR (i:n) array is used.

Example:

Initial



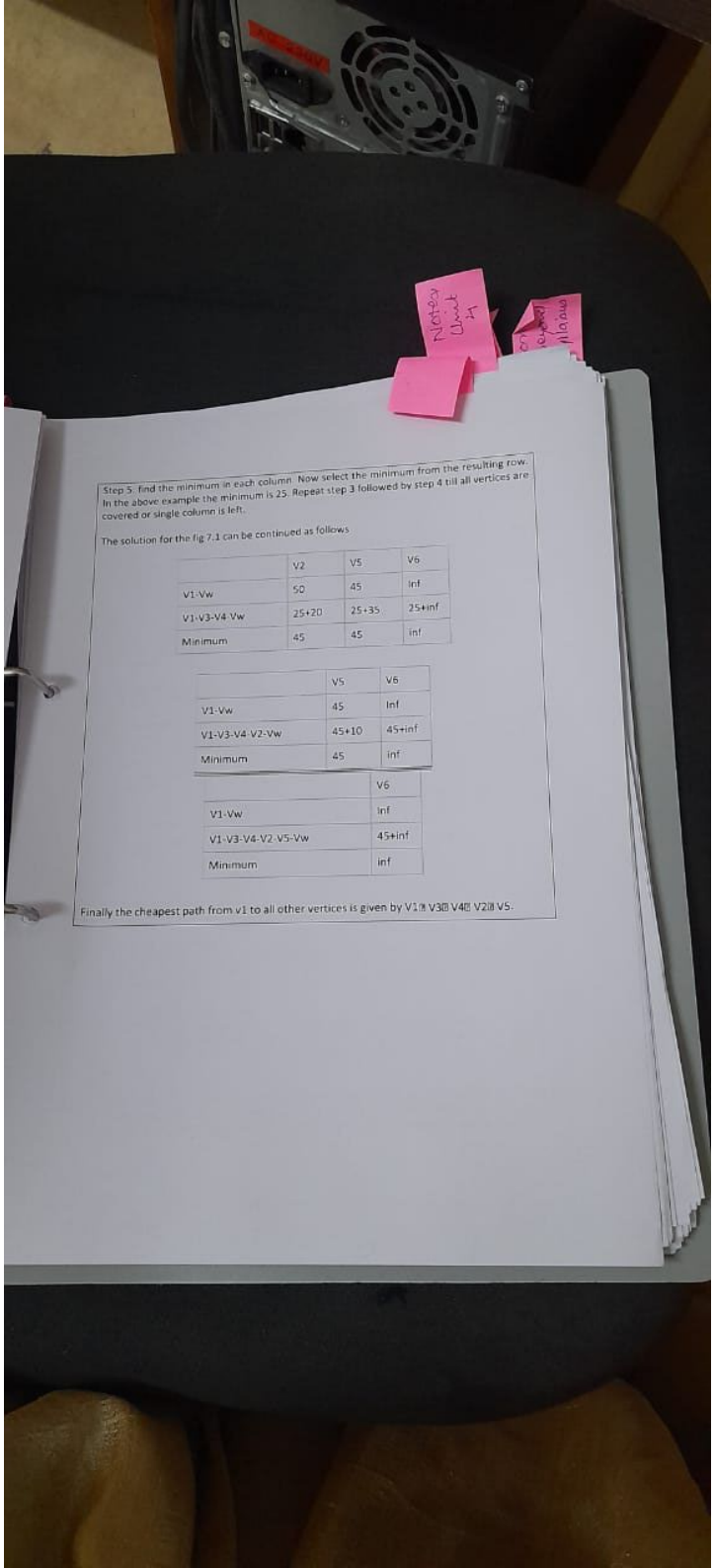


Compare Dist (u)+ cost (u,w) with Dist (w).

Algorithm:

```
Procedure SSSP (v_o, cost, n)
   Array  S (1:n);
  Begin
     /* initialization*/
     For i=1 to n do
          S(i)=o, Dist (i)= cost (v_o,i)
     End for.
     S(v_o)=1, Dist (v_o)=o;
     For i=1 to n-1 do.
                Choose u  s.t.  Dist (u)= min {Dist (w) }
                                          S(w)=0
                S(u)=1;
                For all w with S(w)=o do.
                     Dist (w)= min (Dist (w), Dist (u) +
                                              Cost (u,w) )
                End for.
     end for.
end.
```

✓ Time complexity: $O(n^2)$.



Step 5: find the minimum in each column. Now select the minimum from the resulting row. In the above example the minimum is 25. Repeat step 3 followed by step 4 till all vertices are covered or single column is left.

The solution for the fig 7.1 can be continued as follows

	V2	V5	V6
V1-Vw	50	45	inf
V1-V3-V4-Vw	25+20	25+35	25+inf
Minimum	45	45	inf

	V5	V6
V1-Vw	45	Inf
V1-V3-V4-V2-Vw	45+10	45+inf
Minimum	45	inf

	V6
V1-Vw	Inf
V1-V3-V4-V2-V5-Vw	45+inf
Minimum	inf

Finally the cheapest path from v1 to all other vertices is given by V1 V3 V4 V2 V5.

Knapsack Problem

- We are given n objects and knapsack or bag with capacity M. Object 'i' has a weight Wi and profit Pi where i varies from 1 to N.
- The problem is we have to fill the bag with the help of N objects and the resulting profit has to be maximum.
- Formally the problem can be stated as

  Maximize XiPi subject to XiWi<=M

  Where Xi is the fraction of object and it lies between 0 to 1.
- There are so many ways to solve this problem, which will give many feasible solutions for which we have to find the optimal solution.
- But in this algorithm, it will generate only one solution which is going to be feasible as well as optimal.
- First, we find the profit & weight rates of each and every object and sort it according to the descending order of the ratios.
- Select an object with highest p/w ratio and check whether its height is lesser than the capacity of the bag.
- If so place 1 unit of the first object and decrement the capacity of the bag by the weight of the object you have placed.
- Repeat the above steps until the capacity of the bag becomes less than the weight of the object you have selected in this case place a fraction of the object and come out of the loop.
- Whenever you selected.

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le M \tag{4.2}$$

$$\text{and} \quad 0 \le x_i \le 1, \quad 1 \le i \le n \tag{4.3}$$

The Profits and Weights are positive.

ALGORITHM:

```
Algorityhm Greedy knapsack (m,n)
//P[1:n] and the w[1:n]contain the profit
// & weight res' of the n object ordered.
//such that p[i]/w[i] >=p[i+1]/W[i+1]
//n is the Knapsack size and x[1:n] is the solution vertex.
{
  for i=1 to n do a[i]=0.0;
```

```
Procedure kruskal (G, cost).
Begin
    T: forest
    T= ∅
    while |T| ≤ n-1 & E≠∅ do
        choose an edge (v,w) ∈E of least weight
        delete (v,w) form E
        If (v,w) does not create a cycle in T
        then
            add (v,w) ∈o  T
        else
            discard (v,w);
        endif
    end while.
```

Implementation:

- Choose the edge with the smallest weight:

  ✓ Use min-heap:
  - Get the min & read just the heap takes O (log e).
  - Construct the heap takes O (e).

- Be sure that the chosen edge does not create a cycle in the so far built forest, T:

  ✓ Use union-find:
  Once (u,v) is selected.
  Check if Find (u) ≠ Find (v).

- solving algebraic problem,
- introduction to parallel algorithms
- Meaning of lower bound theory and its use in
- solving algebraic problem,
- introduction to parallel algorithms

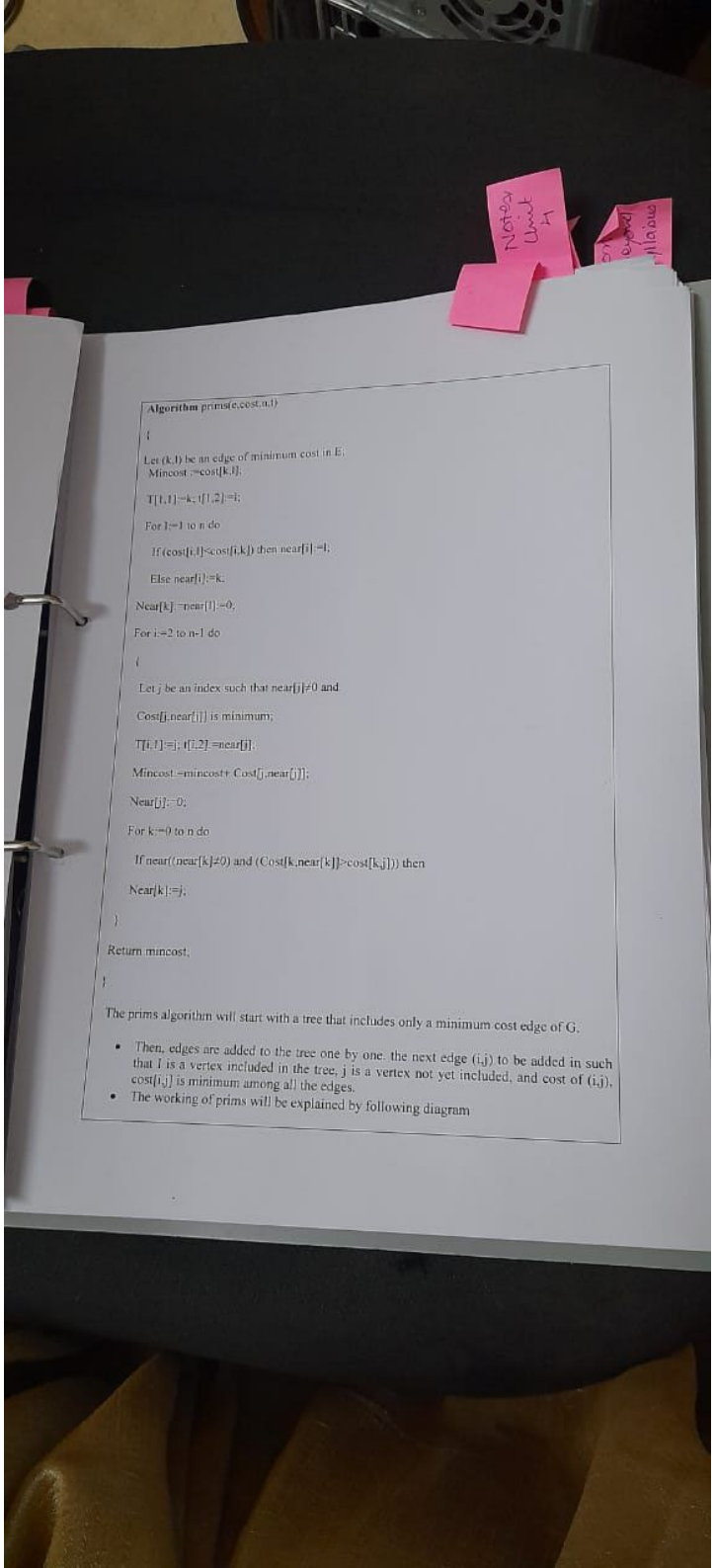
Concept of Back Tracking

- A trivial way of solving TVSP is to compute d(T/X) for every $X \subseteq V$, leading to a possible $2^{|V|}$ computations.

Solving the above tree with $\delta = 5$

- Greedy solution for TVSP
- We want to minimize the number of booster stations (X)
- For each node $u \in V$, compute the maximum delay d(u) from u to any other node in its subtree.
- If u has a parent v such that $d(u) + w(v,u) > \delta$, split u and set d(u) to zero
- Computation proceeds from leaves to root
- Delay for each leaf node is zero
- The delay for each node v is computed from the delay for the set of its children C(v)
- If $d(v) > \delta$, split v

```
Algorithm TVS(T,l)
//Determine and output the nodes to be split.
//w() is the weighting function for the edges.
{
 if(T!=0) then
 {
  d[T]:=0;
  for each child v of T do
   {
   TVS(v,l);
   d[T]:=max(d[T],d[v] + w(T,v));
   }
 if((T is not the root) and (d[T] + w(parent(T),T)>l)) then
  {
  write(T); d[T]:=0;
  }
 }
}
```



Algorithm prims(e,cost,n,t)

{

Let (k,l) be an edge of minimum cost in E;

Mincost :=cost[k,l];

T[1,1]=k; t[1,2]:=l;

For i:=1 to n do

If (cost[i,l]<cost[i,k]) then near[i]:=l;

Else near[i]:=k;

Near[k]:=near[l]:=0;

For i:=2 to n-1 do

{

Let j be an index such that near[j]≠0 and

Cost[j,near[j]] is minimum;

T[i,1]=j; t[i,2]:=near[j];

Mincost:=mincost+ Cost[j,near[j]];

Near[j]:=0;

For k:=0 to n do

If near((near[k]≠0) and (Cost[k,near[k]]>cost[k,j])) then

Near[k]:=j;

}

Return mincost;

}

The prims algorithm will start with a tree that includes only a minimum cost edge of G.

- Then, edges are added to the tree one by one. the next edge (i,j) to be added in such that I is a vertex included in the tree, j is a vertex not yet included, and cost of (i,j), cost[i,j] is minimum among all the edges.
- The working of prims will be explained by following diagram

Minimum Cost Spanning Tree- Prim's Algorithm

- Let G(V,E) be an undirected connected graph with vertices 'v' and edge 'E'.
- A sub-graph t=(V,E') of the G is a Spanning tree of G iff 't' is a tree.3
- The problem is to generate a graph G'= (V,E) where 'E' is the subset of E,G' is a Minimum spanning tree.
- Each and every edge will contain the given non-negative length .connect all the nodes with edge present in set E' and weight has to be minimum.

NOTE:

- We have to visit all the nodes.
- The subset tree (i.e) any connected graph with 'N' vertices must have at least N-1 edges and also it does not form a cycle.

Definition:

- A spanning tree of a graph is an undirected tree consisting of only those edge that are necessary to connect all the vertices in the original graph.
- A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Application of the spanning tree:

1. Analysis of electrical circuit.
2. Shortest route problems.

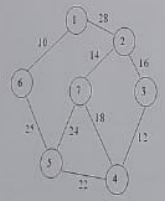
Minimum cost spanning tree:

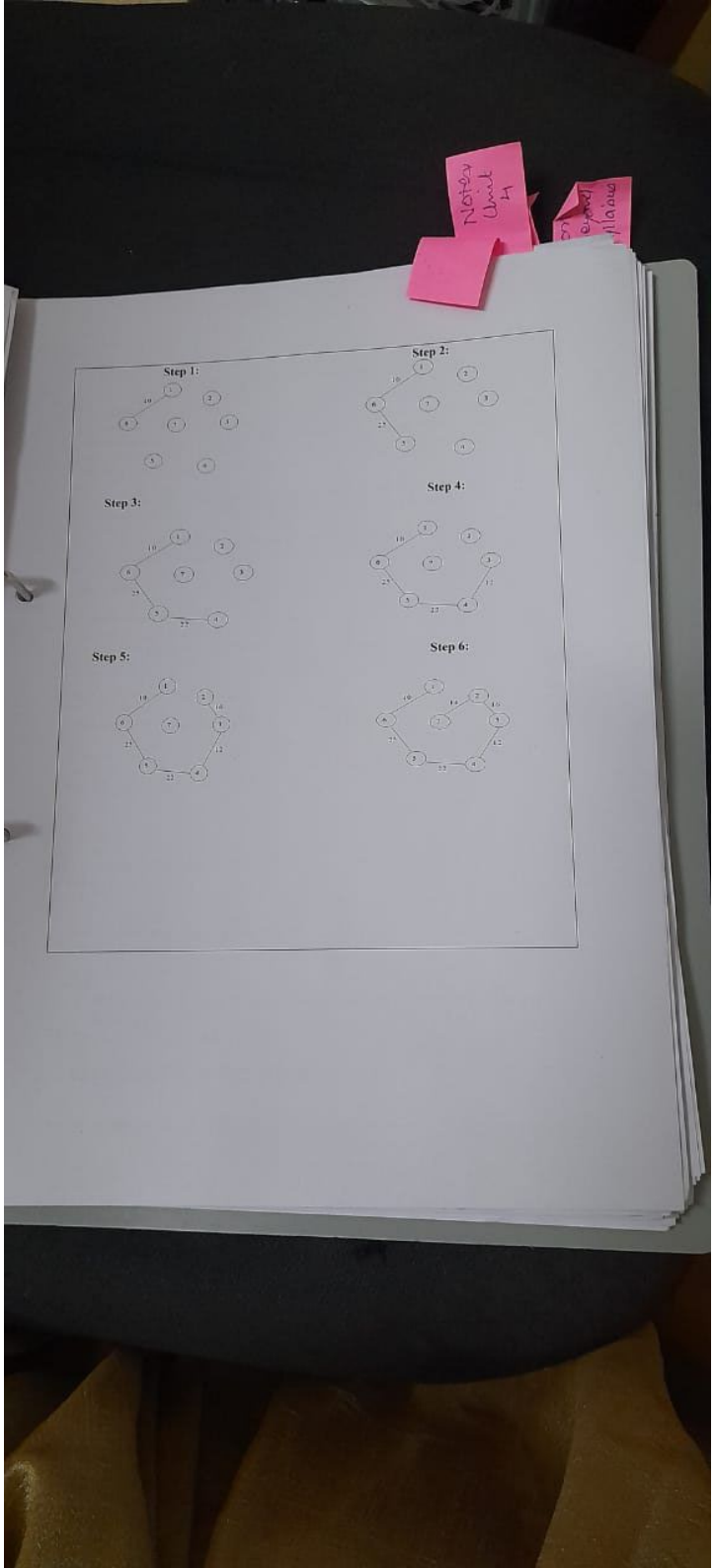
- The cost of a spanning tree is the sum of cost of the edges in that trees.
- There are 2 method to determine a minimum cost spanning tree are

1. Kruskal's Algorithm
2. Prom's Algorithm.

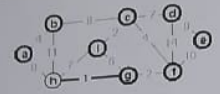
PRIM'S ALGORITHM

Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have been reached.

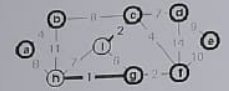



Notes
Unit
H
Step 1:
Step 2:
Step 3:
Step 4:
Step 5:
Step 6:

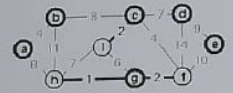
representative. Lets choose vertex g arbitrarily.



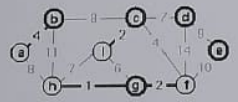
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



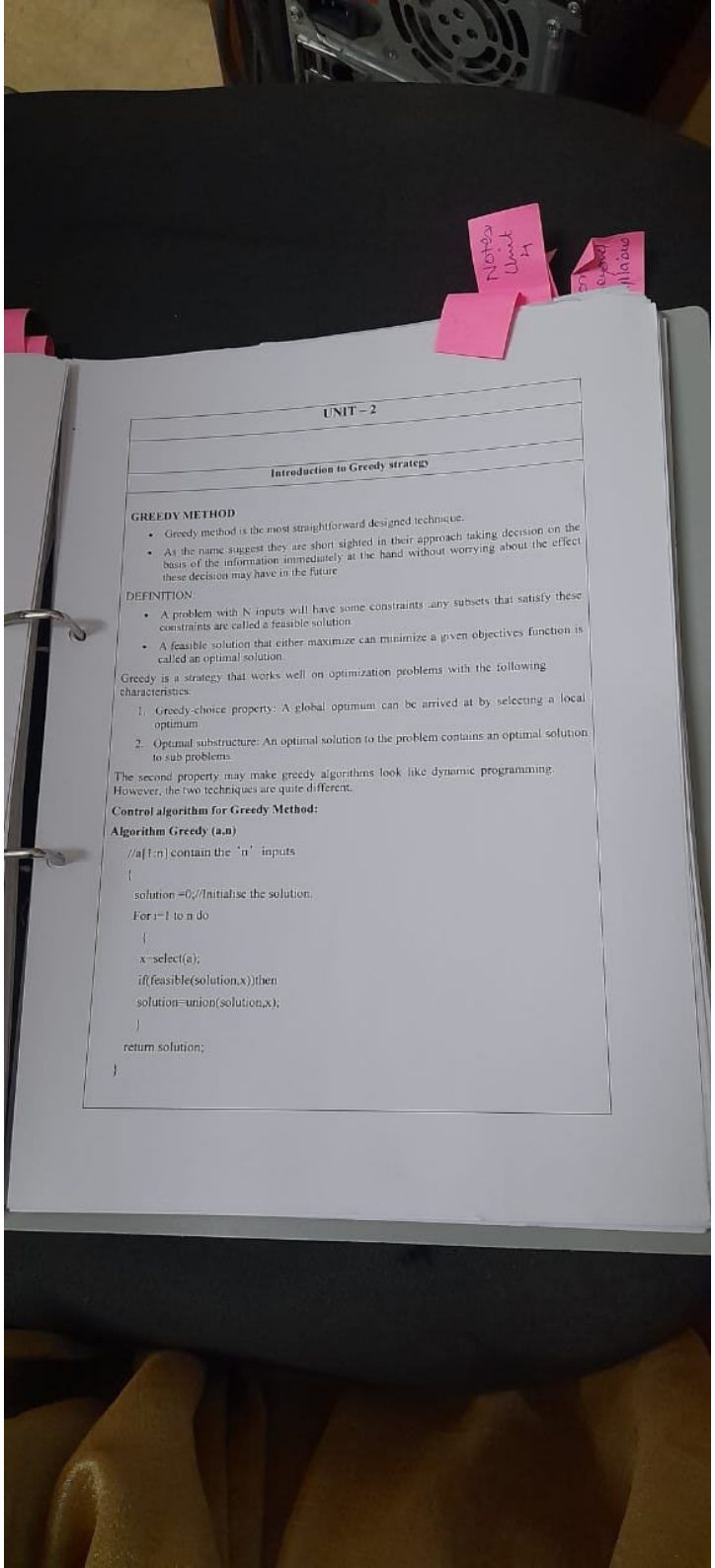
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



Step 4. Edge (a, b) creates a third tree.



Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



UNIT – 2

Introduction to Greedy strategy

GREEDY METHOD

- Greedy method is the most straightforward designed technique.
- As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future

DEFINITION:

- A problem with N inputs will have some constraints .any subsets that satisfy these constraints are called a feasible solution.
- A feasible solution that either maximize can minimize a given objectives function is called an optimal solution.

Greedy is a strategy that works well on optimization problems with the following characteristics:

1. Greedy-choice property: A global optimum can be arrived at by selecting a local optimum
2. Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems

The second property may make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Control algorithm for Greedy Method:

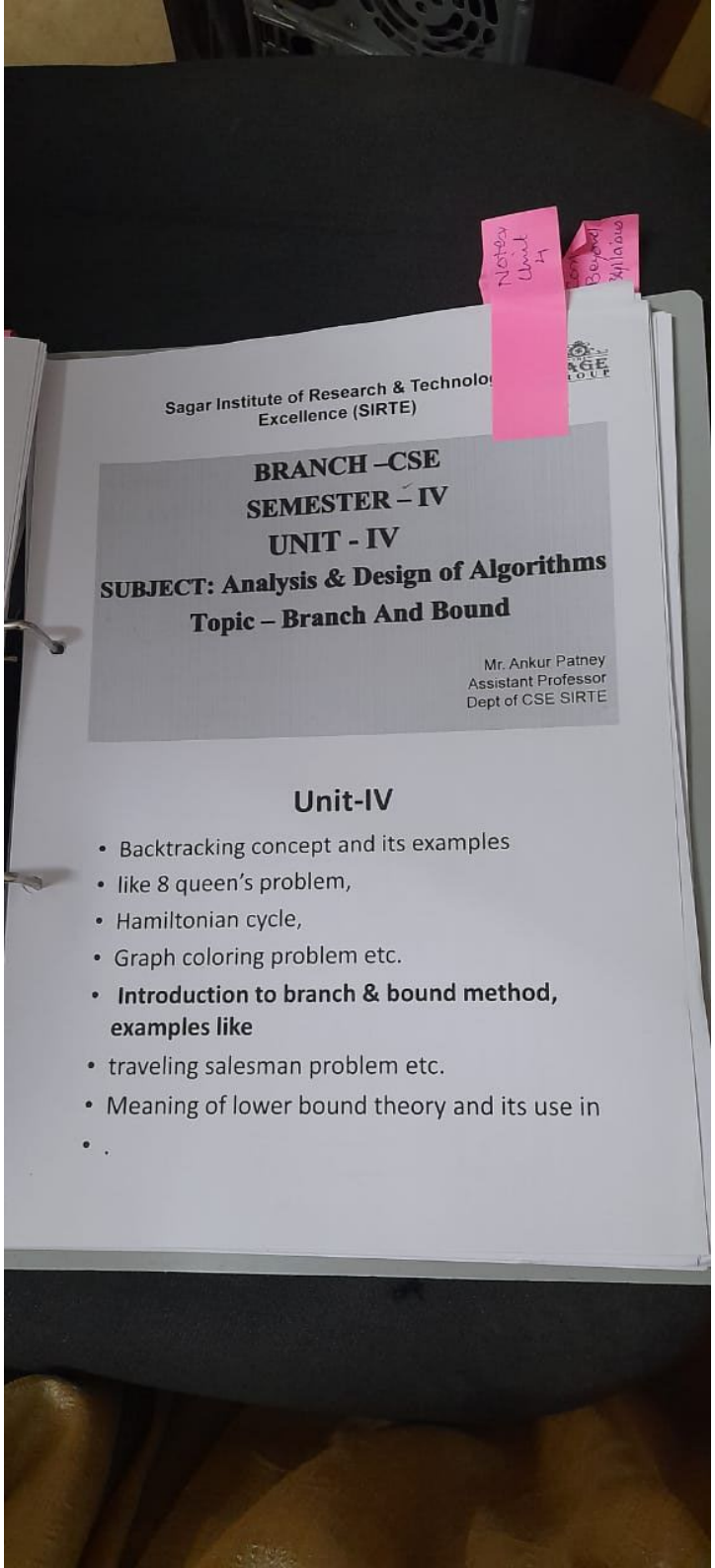
Algorithm Greedy (a,n)

```
//a[1:n] contain the 'n' inputs
{
solution =0;//Initialise the solution.
For i=1 to n do
  {
  x=select(a);
  if(feasible(solution,x))then
  solution=union(solution,x);
  }
return solution;
}
```

```
Procedure PRIM (G, Cost, mincost)
  /* Let n be # of vertices */
  Integer NEAR (1:n);
  Integer u,w,p,I;
1. Begin
2.    Choose an arbitrary vertex v_o.
3.    mincost=o; NEAR (v_o)=o
4.    For each vertex w ≠ v_o do
5.       NEAR (w)=v_o;
6.    End for

7.    For I=1 to n-1 do  /* fin n-1 edges of T */
8.       Choose a vertex w s.t.
9.          cost(w,NEAR(w) )= min (cost (u, NEAR(u)) )
10.         where NEAR (u) ≠ o
11.      mincost = mincost+ cost (w, NEAR(w));
12.      NEAR (w)=o
13.      For each vertex p do
14.         if NEAR(p) ≠ o & cost (p, NEAR(p) ) > cost (p,w)
15.         then NEAR (p)= w;
16.         endif
17.      end for
18.   End for
19.   End.
```

- Analysis:

  - ✓ The for loop between 4 and 6 takes O(n).
  - ✓ Lines between 8 and 10 take O(n)
  - ✓ The For loop between 13 and 17 takes O(n)
  - ✓ Finally, the main For loop that starts at line 7 takes O(n)
  - ✓ the overall algorithm takes $O(n^2)$.


Sagar Institute of Research & Technolo
Excellence (SIRTE)
BRANCH –CSE
SEMESTER – IV
UNIT - IV
SUBJECT: Analysis & Design of Algorithms
Topic – Branch And Bound
Mr. Ankur Patney
Assistant Professor
Dept of CSE SIRTE
Unit-IV
• Backtracking concept and its examples
• like 8 queen's problem,
• Hamiltonian cycle,
• Graph coloring problem etc.
• Introduction to branch & bound method, examples like
• traveling salesman problem etc.
• Meaning of lower bound theory and its use in
• .

lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.
A much simpler method would be to solve it using matrix representation. The steps that should be followed is as follows.

Step 1: find the adjacency matrix for the given graph. The adjacency matrix for above graph is given below

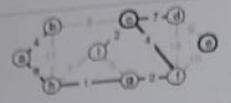
	V1	V2	V3	V4	V5	V6
V1	-	50	10	Inf	45	Inf
V2	Inf	-	15	Inf	10	Inf
V3	20	Inf	-	15	Inf	Inf
V4	Inf	20	Inf	-	35	Inf
V5	Inf	Inf	Inf	30	-	Inf
V6	Inf	Inf	Inf	3	Inf	-

Step 2: consider v1 to be the source and choose the minimum entry in the row v1. In the above table the minimum in row v1 is 10.

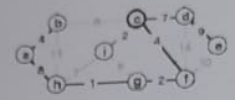
Step 3: find out the column in which the minimum is present, for the above example it is column v3. Hence, this is the node that has to be next visited.

Step 4: compute a matrix by eliminating v1 and v3 columns. Initially retain only row v1. The second row is computed by adding 10 to all values of row v3.
The resulting matrix is

	V2	V4	V5	V6
V1-Vw	50	Inf	45	Inf
V1-V3-Vw	10+inf	10+15	10+inf	10+inf
Minimum	50	25	45	inf



Step 10. Again, if we add edge (h, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



0/1 Knapsack using Branch & Bound Method

Now we have to arrange the items according to descending order of value/weight

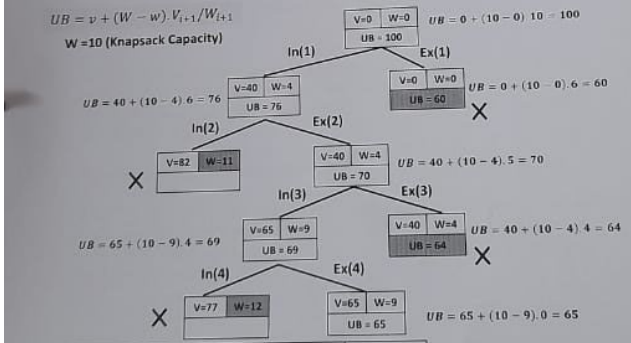
Items	Weight	Value	Value/Weight
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

Now we have to calculate the upper bound for each node by using formula:

$$UB = v + (W - w).V_{i+1}/W_{i+1}$$

$UB = v + (W - w).V_{i+1}/W_{i+1}$

W =10 (Knapsack Capacity)



Items	1	2	3	4
0/1	1	0	1	0
Weight	4	-	5	-
Profit	40	-	25	-

Ans: Thus the solution is to pick up item 1, 3 and Max profit = 65

0/1 knapsack Problem

0/1 knapsack: [RGPV June-2014(3)]

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

Problem Description
If we are given n objects and a knapsack or a bag in which the object i that has weight w_i is to be placed. The knapsack has a capacity W. Then the profit that can be earned is $p_i x_i$. The objective is to obtain filling of knapsack with maximum profit earned.Maximized $p_i x_i$ subject to constraint $w_i x_i <= W$ Where $1<=i<=n$ and n is total no. of objects and $x_i=0$ or 1.

Steps and Notations

Step-1:
Let $f_i(y_j)$ be the value of optimal solution. Then s^i is pair (p,w) where $p=f_i(y_j)$ and $w=y_j$. Initially
$s^0 = \{(0,0)\}$
We can compute s^{i+1} from s^i . These computations of s^i are basically the sequence of decisions made for obtaining the optimal solutions.

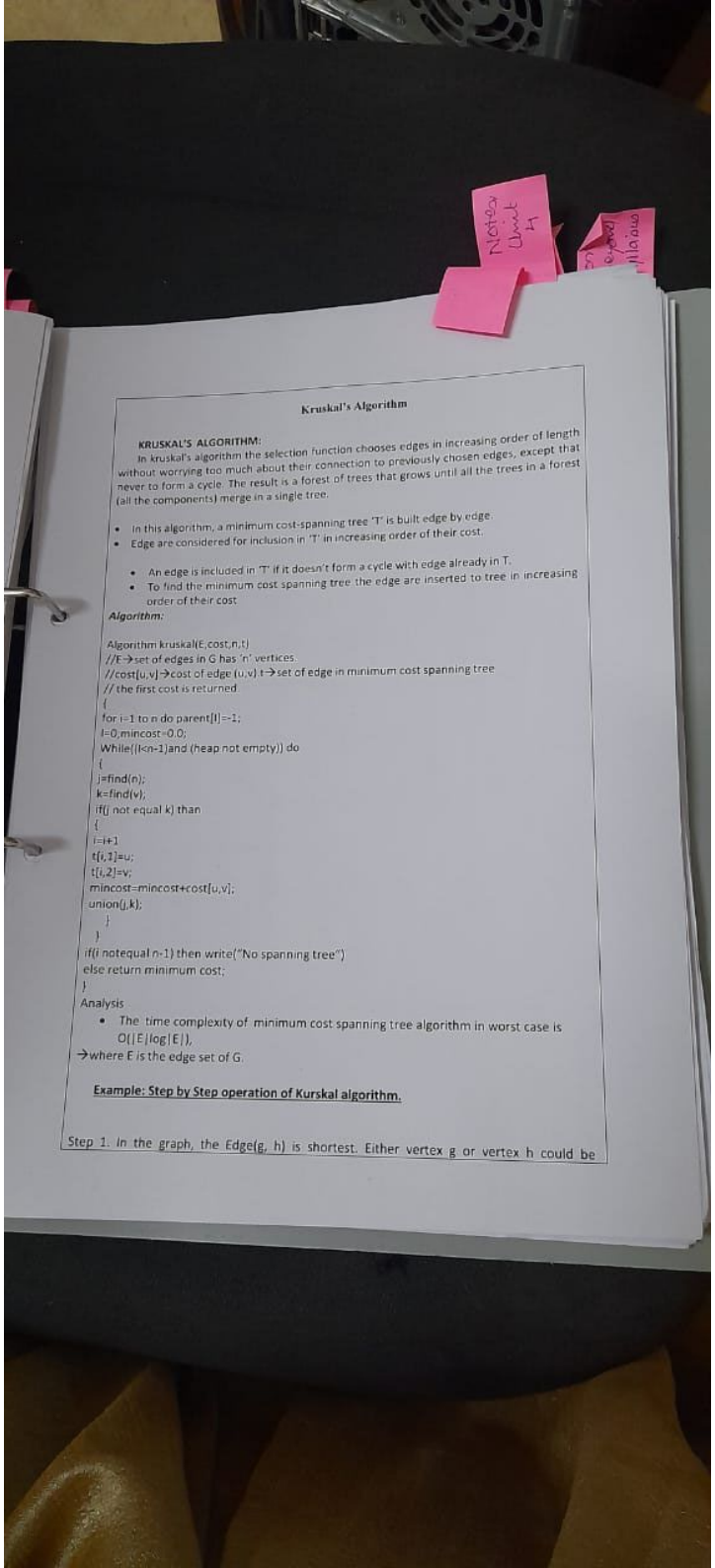
Step-2:
We can generate the sequence of decisions in order to obtain the optimum selection for solving the knapsack problem.

Let x_n be the optimum sequence. Then there are two instances $\{x_n\}$ and $\{x_{n-1},x_{n-2},\ldots,x_1\}$.
So from $\{x_n,x_{n-1},x_{n-2},\ldots,x_1\}$ we will choose the optimum sequence with respect to x_n.
The selection of sequence from remaining set should be such that we should be able to fulfill the condition of filling knapsack capacity W with maximum profit.
Otherwise $\{x_{n-1},x_{n-2},\ldots,x_1\}$ is not optimum.
This proves that 0/1 knapsack problem is solved using principle of optimality.

Step-3:
Let $f_i(y_j)$ be the value of optimal solution. Then $f_i(y) = \max\{f_{i-1}(y), f_{i-1}(y-w_i)+p_i\}$
Initially compute
$s^0=\{(0,0)\}$
$s^1_i = \{(P,W)|(P-p_1, W-w_1) \text{ belongs to } s^1\}$

S^{i+1} can be computed by merging s^i and s^1_i

Purging rule
If s^{i+1} contains (P_j, W_j) and (P_k, W_k) these two pairs such that $P_j<=P_k$ and $W_j>=W_k$, then (P_j,W_j) can be eliminated . This purging rule is also called as dominance rule. In purging rule basically the dominated tuples gets purged. In short remove the pair with less profit and more weight.



Kruskal's Algorithm

KRUSKAL'S ALGORITHM:

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

- In this algorithm, a minimum cost-spanning tree 'T' is built edge by edge.
- Edge are considered for inclusion in 'T' in increasing order of their cost.
  - An edge is included in 'T' if it doesn't form a cycle with edge already in T.
  - To find the minimum cost spanning tree the edge are inserted to tree in increasing order of their cost.

Algorithm:

```
Algorithm kruskal(E,cost,n,t)
//E→set of edges in G has 'n' vertices
//cost[u,v]→cost of edge (u,v) t→set of edge in minimum cost spanning tree
// the first cost is returned.
{
for i=1 to n do parent[I]=-1;
I=0,mincost=0.0;
While((I<n-1)and (heap not empty)) do
{
j=find(n);
k=find(v);
if(j not equal k) than
{
i=i+1
t[i,1]=u;
t[i,2]=v;
mincost=mincost+cost[u,v];
union(j,k);
   }
  }
if(i notequal n-1) then write("No spanning tree")
else return minimum cost;
}
```

Analysis

- The time complexity of minimum cost spanning tree algorithm in worst case is $O(|E|\log|E|)$,

→where E is the edge set of G.

Example: Step by Step operation of Kurskal algorithm.

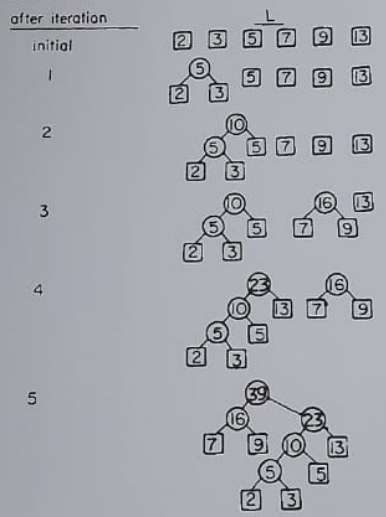
Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be

of records) of the file represented by that node.

The external node F4 is at a distance of 3 from the root node Z4 (a node at level i is at a distance of i - 1 from the root). Hence, the records of file F4 will be moved three times, once to get Z1, once again to get Z2 and finally one more time to get Z 4 . If d_i is the distance from the root to the external node for file F; and q_i the length of F; then the total number of record moves for this binary merge tree is

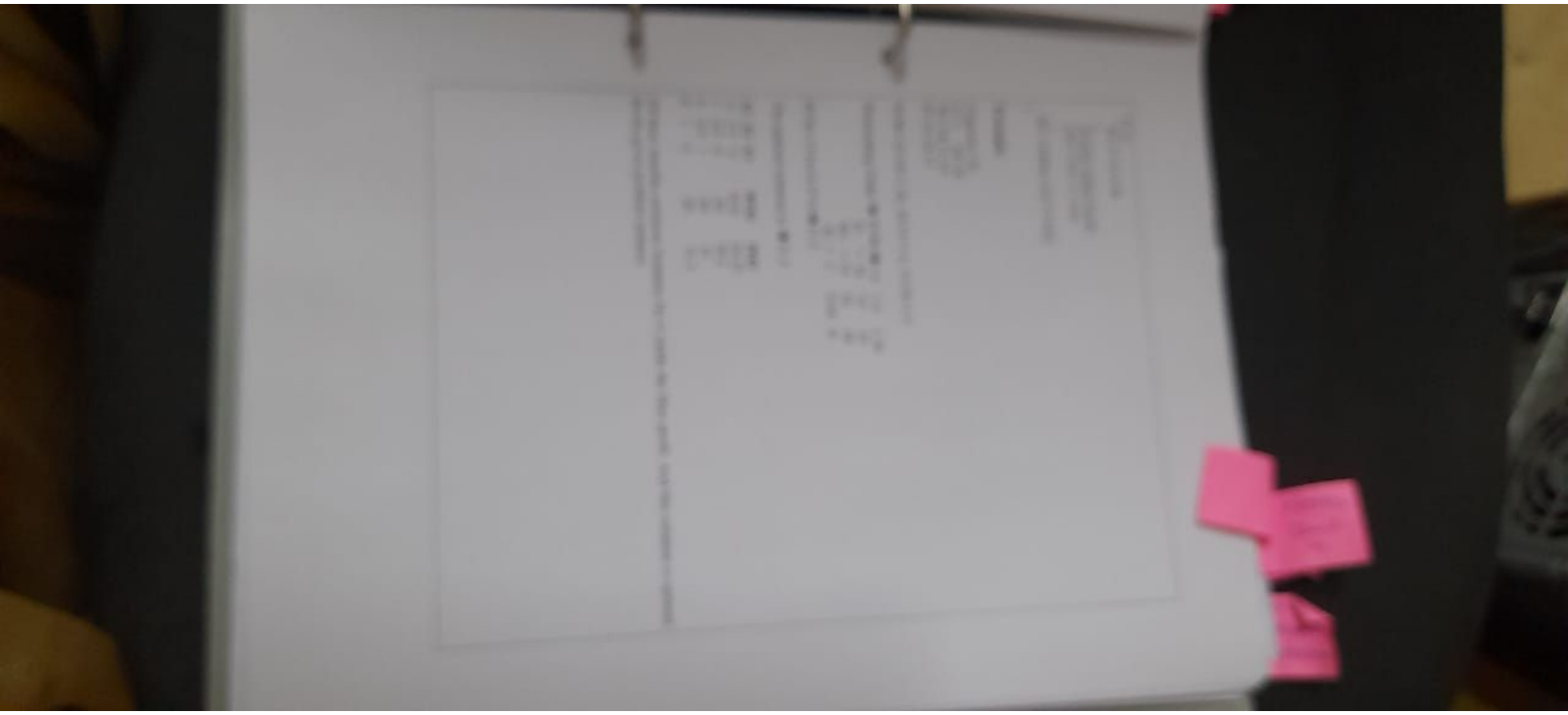
$$\sum_{i=1}^{n} d_i q_i .$$

This sum is called the weighted external path length of the tree



line procedure TREE(L, n)

//L is a list of n single node binary trees as described above//

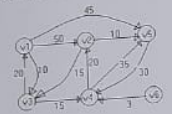


Single Source Shortest path Algorithm (Dijkstra's)

Single-source shortest path:

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:

1. Is there a path from A to B?
2. If there is more than one path from A to B? Which is the shortest path?



The problems defined by these questions are special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination. The graphs are digraphs representing streets. Consider a digraph G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:

Consider the digraph of above figure. Let the numbers on the edges be the costs of travelling along that route. If a person is interested travel from v1 to v2, then he encounters many paths. Some of them are

v1- v2 = 50 units
v1- v3- v4- v2 = 10+15+20=45 units
v1- v5- v4- v2 = 45+30+20= 95 units
v1- v3- v4- v5- v4- v2 = 10+15+35+30+20=110 units

The cheapest path among these is the path along v1-v3-v4-v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1-v2 that costs 50 units. One can also notice that, it is not possible to travel to v6 from any other node.

To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the

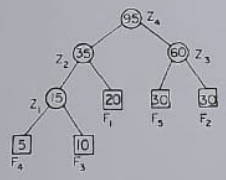
Optimal Merge Patterns

Optimal Merge Patterns

In merging two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time O(n + m). When more than two sorted files are to be merged together the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if files X1, X2, X3 and X4 are to be merged we could first merge X1 and X2 to get a file Y1. Then we could merge Y1 and X3 to get Y2. Finally, Y2 and X4 could be merged to obtain the desired sorted file. Alternatively, we could first merge X1 and X2 getting Y1, then merge X3 and X 4 getting Y2 and finally Y1 and Y2 getting the desired sorted file. Given n sorted files there are many ways in which to pair wise merge them into a single sorted file. Different pairings require differing amounts of computing time. Here determining an optimal (i.e. one requiring the fewest comparisons) way to pair wise merge n sorted files together.

Example: X1, X2 and X3 are three sorted files of length 30, 20 and 10 records each. Merging X1 and X2 requires 50 record moves. Merging the result with X3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If instead, we first merge X2 and X3 (taking 30 moves) and then X1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first.

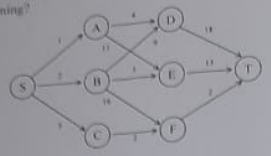
A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n record file and an m record file requires possibly n + m records moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files (F1, ..., F 5) with sizes (20, 30, 10, 5, 30) our greedy rule would generate the following merge pattern: merge F4 and F3 to get Z1 (|Z1| = 15); merge Z1 and F1 to get Z2 (|Z2| = 35); merge F2 and F5 to get Z3 (|Z3| = 60); merge Z2 and Z3 to get the answer Z4. The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.



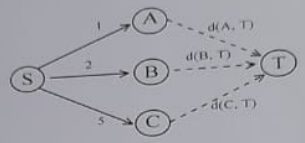
The merge pattern such as the one just described will be referred to as a 2-way merge pattern (each merge step involves the merging of two files). 2-way merge patterns may be represented by binary merge trees. Figure shows a binary merge tree representing the optimal merge pattern obtained for the above five files. The leaf nodes are drawn as squares and represent 170 The Greedy Method the given five files. These nodes will be called external nodes. The remaining nodes are drawn circular and are called internal nodes. Each internal node has exactly two children and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number

Problems based on Multistage graph

Q.1 Find a minimum cost path from 'S' to 't' in multistage graph using dynamic programming?

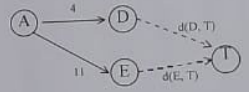


Solution:

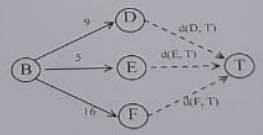


d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}

d(A, T) = min{4+d(D,T), 11+d(E,T)}
= min{4+18, 11+13} = 22.



d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
= min{9+18, 5+13, 16+2} = 18.
d(C, T) = min{ 2+d(F, T) } = 2+2 = 4
d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
= min{1+22, 2+18, 5+4} = 9.



Q.2 Find a minimum cost path from 'S' to 't' in multistage graph using dynamic programming? [RGPV JUNE-2014]

Branch and Bound Method

- After computation of bounding value at each node, we compare it with best solution and the expanding the node further can lead to a final solution.
- The branching procedure replaces an original problem by a new set of new problem that are:
  - Partially solved version of the original problem.
  - Smaller problem than original problem.
- There are various reason for terminating search path in state space tree of branch and bound:
  - The value of the node's bound is better than the best solution
  - There is no feasible solution.
  - A new better solution is obtained rather then continuing further.

0/1 Knapsack using Branch & Bound Method

Question: Solve the following instance of the knapsack problem by the branch and bound algorithm.

Items	Weight	Value
1	4	40
2	7	42
3	5	25
4	3	12

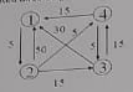
W = 10

Solution: First we compute value/weight for each item.

Items	Weight	Value	Value/Weight
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

algorithm is $O(n^3)$ because of the triple-nested loops; the space complexity is $O(n^2)$ because only one matrix is used.

Example: We demonstrate Floyd's algorithm for computing $D_k[i,j]$ for $k = 0$ through $k = 4$, for the following weighted directed graph:



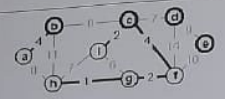
Solution:

$$D_0 = W = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

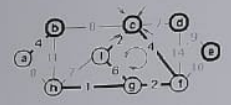
$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & (35) & 0 & 15 \\ 15 & (20) & 5 & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 5 & (20) & (10) \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

reduced from ∞ because the path (3,1,2) going thru node 1 is possible in D_1

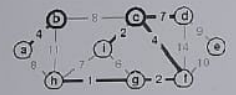
$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ (45) & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_4 = \begin{pmatrix} 0 & 5 & (15) & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 25 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$



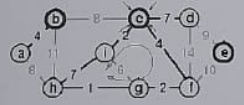
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



Step 7. Instead, add edge (c, d).



Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).

Formal definition of NP-completeness

A decision problem c is NP-complete if:

1. c is in NP, and
2. Every problem in NP is reducible to c in polynomial time.

c can be shown to be in NP by demonstrating that a candidate solution to c can be verified in polynomial time.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition

A consequence of this definition is that if we had a polynomial time algorithm (on a UTM, or any other Turing-equivalent abstract machine) for$c$, we could solve all problems in NP in polynomial time.

NP-complete problems

An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices. Consider these two problems:

- Graph Isomorphism: Is graph G1 isomorphic to graph G2?
- Sub graph Isomorphism: Is graph G1 isomorphic to a sub graph of graph G2?

The Sub graph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be hard, but is not thought to be NP-complete.

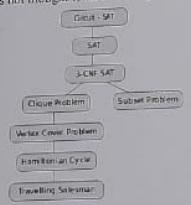


Figure: Some NP complete problem, indicating the reductions typically used to prove their NP completeness

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems.

The list below contains some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (Sat.)
- Knapsack problem

Concept of Back Tracking

- The term Backtrack was given by American Mathematician D.H. Lehmer in the 1950.
- In this technique we search for a set of solutions or optimal solution which satisfies some constraints.
- It incrementally builds candidates to the solutions, and abandons each partial candidate(Backtrack) as soon as it determines that the candidate can not possibly be completed to a valid solution.
- Backtracking can be applied only for those 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.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e. set of all feasible solutions) for given problem.
- Backtracking is a important tool for solving constraints satisfaction problems such as crosswords, verbal arithmetic's, Sudoku and many other puzzles.
- In this method all possible solutions are tried. If the solution are not satisfying the constraint then backtrack and then backtrack and try another solution.

Applications of Back Tracking

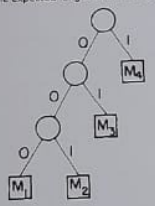
Standard problems that can be solved using Backtracking algorithm.

1. **N Queens problem (4 Queens and 8 Queens Problem)**
2. **Sum of Subset Problem**
3. **Finding Hamilton Cycle**
4. **Graph Coloring Problem**

Huffman Tree

Another application of binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages $M_1, \ldots, M_{n+1}$. Each code is a binary string which will be used for transmission of the corresponding message. At the receiving end the code will be decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages. The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure corresponds to codes 000, 001, 01, and 1 for messages M_1, M_2, M_3 and M_4 respectively. These codes are called Huffman codes.

The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\Sigma_{1 \leq i \leq n+1} q_i d_i$, where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that $\Sigma_{1 \leq i \leq n+1} q_i d_i$ is also the expected length of a transmitted message. Hence the code which minimizes expected decode time also minimizes the expected length of a message.



The remainder of this paper describes a portion of Grumman Data Systems' Adaptable Distributed Ada Programming Testbed (ADAPT) project. The overall goal of this project is to develop a testbed environment that will be used for research on distributed systems construction and software fault tolerance technologies. The current project focus is a distributed implementation of Ada, supported by a tool set that facilitates the construction and understanding of distributed Ada programs. ADAPT is a long term project, providing a phased implementation of distributed Ada and supporting tools. A subset of the proposed testbed was completed in 1987, providing a partial implementation of distributed Ada (see Section 4). This paper describes the full tool set and the expected benefit to be gained through its use. Detailed information on the distributed Ada implementation may be found in [3].

2. ADAPT OVERVIEW

This section provides an overview of ADAPT facilities, the basic approach for constructing distributed programs, base technologies, problems addressed, and a description of distributed program execution. Additionally, we survey related work and compare that with ADAPT. The following section describes ADAPT's tools.

ADAPT provides the following four facilities. First, the concurrency constructs defined within the Ada programming language describe the concurrency of the program. Second, the complete concurrent program is analyzed for potential distribution. (Complete program analysis is required to support the strong typing of the Ada language.) Potential distribution is determined from the program's use of Ada's tasking constructs, an evaluation of the resulting control flow, and an evaluation of data access patterns. Third, distributed interfaces are generated automatically and safely, but only where possible and desired. (This removes a burden from the programmer and assures that the distribution does not add unnecessary overhead.) Fourth, the distributed mapping and its effects are reported to the distributed system builders. These builders are allowed to modify that mapping to adjust performance as required.

With these four facilities, ADAPT permits concurrent programs to be developed independent of the details of distribution. Distribution is implemented with no syntactic or semantic changes to the Ada code nor any requirement for explicit operating system calls to invoke remote interfaces. (See [4] for a relevant distributed application example.) ADAPT implements concurrency at the task level to meet the particular needs of distributed processing with large-grained decomposition and loose coordination. To minimize the difficulties of hidden couplings in large systems, ADAPT provides analysis of subprogram interaction within the entire program, adding distributed interfaces where needed. ADAPT's tools analyze the program, inform the user of the potential effect of all distributed interfaces, and allow the user to modify the distribution of a program (see Figure 1).

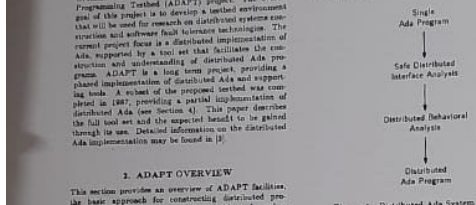


Figure 1. Distributed Ada System Overview

ADAPT is built upon a commercially available Ada compiler. Our host environment is a network of loosely connected computing nodes, each running a Unix based operating system including facilities for transaction management, process imaging and a distributed file system. All tools, the compiler, and the runtime environment are written in Ada.

2.1 DISTRIBUTION ISSUES

The design of ADAPT has been motivated by three distributed issues: distributed systems design, distributed Ada design, and distributed Ada implementation. Distributed Ada implementation issues have been widely discussed in the literature [1, 2, 3, 6, 7, 8, 9]. Details of our distributed Ada implementation may be found in [3]. ADAPT is concerned with distributed Ada design and distributed systems design issues.

First consider a distributed systems design issue: the task of partitioning a software design into several concurrent modules, and the allocation of these onto a distributed target. One must consider the allocation of modules to machines given specific hardware or peripheral constraints and intermachine communication overhead as it affects, for example, efficiency, latency, and dynamic load balancing. For large systems, software changes during normal maintenance activities can have an important impact on these partitioning and allocation decisions. ADAPT supports the human evaluation of these decisions by providing evaluation and automation for distributed Ada design and distributed Ada implementation.

Now consider a distributed Ada design issue: the decision to allocate a task to a specific processor. Because Ada makes the implicit assumption of a static, universal

Problems based on Reliability design:

Q.1 Design a three stage system with device types D1,D2,D3. The costs are Rs. 30, Rs. 15 and Rs. 20 respectively. The cost of the system is to be no more than Rs. 105. The reliability of each device type is 0.9,0.8 and 0.5 respectively.

Solution:

We will first compute u_1, u_2, u_3 using following formula.

$u_i = (C + C_i - \text{sigma } C_j) / C_i$

For computing u_1

$u_1 = 2$(approx value)

For computing u_2

$u_2 = 3$(approx value)

For computing u_3

$u_3 = 3$

Hence (u_1, u_2, u_3)

Computing subsequences-

$S^0 = (1,0)$

Let Si consist of tuples of the form $(f, x) = (r, c)$

$S^0 = \{(1,0)\}$

For device D_1 for 1 D_1

$r_1=0.9, c_1=30$

$S^1_1 = \{(0.9, 30)\}$

For device D_1 for 2 D_1

$m_1=2$(2 D_1 device in parallel)

Reliability of stage $1 = 1-(1-r_1)^2$

Reliability of stage $1 = 1-(1-0.9)^2 = 0.99$

Cost $=30*2 = 60$

$S^1_2 = \{(0.99, 60)\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

$S^1 = \{(0.9, 30), (0.99, 60)\}$

For one Device D2:-

$S^2_1 = \{(0.72, 45), (0.792, 75)\}$

For two Device D2:-

$S^2_2 = \{(0.864, 60), (0.9504, 90)\}$

For three Device D2:-

$S^2_3 = \{(0.8928, 75), (0.98208, 105)\}$

$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.9504, 90), (0.8928, 75), (0.98208, 105)\}$

(0.792, 75), (0.9504, 90) is eleminated due to purging or dominance rule and (0.98208, 105) is eleminated due to access cost 105.

After this we got

$S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

For one Device D3:-

$S^3_1 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

For Two Device D3:-

$S^3_2 = \{(0.54, 85), (0.648, 100)\}$

For Three Device D3:-

Problem-1
Solve knapsack instance M=8, and n=4. let p_i and w_i are as shown below.

i	p_i	w_i
1	1	2
2	2	3
3	5	4
4	6	5

Solution-

s^0={(0,0)} initially
s^0_1={(1,2)}
That means while building s^0_1 we select the next i^{th} pair.
For s^0_1we have selected first (P,W) pair which (1,2).

S^1 ={merge s^0 and s^0_1}
={(0,0),(1,2)}
s^1_1={select next (P,W) pair and add it with s^1}
={(2,3),(2+0,3+0), (2+1,3+2)}
s^1_1 ={(2,3),(3,5)} // repetition of (2,3) avoided.

S^2 ={merge candidates from s^1 and s^1_1}
={ (0,0), (1,2), (2,3), (3,5)}
S^2_1 ={select next (P,W) pair and add it with s^2}
={(5,4),(6,6),(7,7),(8,9)}
S^1 ={merge candidates from s^2 and s^2_1}
={ (0,0), (1,2), (2,3),(5,4),(6,6),(7,7),(8,9)}

Note that the pair (3, 5) is purged form s^3.
Because let us assume (P_i, W_i) = (3, 5) and (P_k, W_k) = (5, 4).
Here P_i<=P_k and W_i>W_k is true hence we will eliminate pair (P_i, W_i) = (3, 5) from s^3.

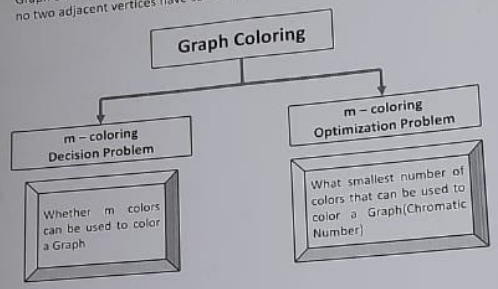
S^3_1 ={select next (P,W) pair and add it with s^1}
S^3. ={(6,5),(7,7),(8,8),(11,9),(12,11) ,(13,12) ,(14,14)}

S^4={(0,0),(1,2),(2,3) ,(5,4) ,(6,6) ,(7,7),(8,9) ,(6,5) ,(8,8),(11,9),(12,11) ,(13,12) ,(14,14)}
Now we are interested in M=8. We get pair (8, 8) in s^4.
Hence we will set x_4=1.

Now we select next object (P-p_4) and (W-w_4).
i.e (8-6) and (8-5).
i.e (2,3)
Pair (2,3) belongs to s^2,hence set x_2=1.
So we get the final solution as {0, 1, 0, 1}.

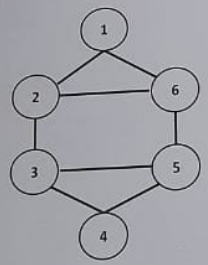
Graph Coloring Problem

Graph coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have same color.

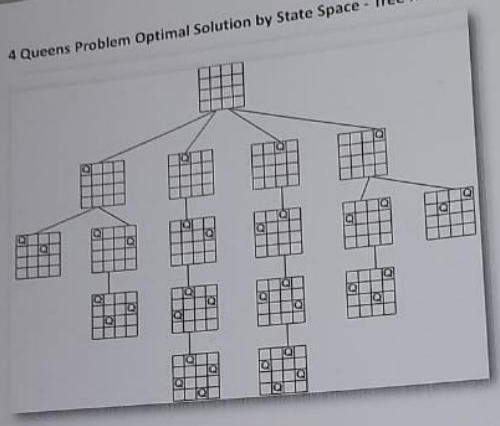


Graph Coloring Problem

Example:



4 Queens Problem Optimal Solution by State Space - Tree Method



8 Queens Problem Optimal Solution by State Space - Tree Method

Trick - 3 6 2 7 1 4 8 5

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

NP-Hard and NP- Complete Problems

Introduction: NP-hard (Non-deterministic Polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". More precisely, a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H. As a consequence, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP, which is unlikely as many of them are considered hard.

A common mistake is to think that the NP in NP-hard stands for non-polynomial. Although it is widely suspected that there are no polynomial-time algorithms for NP-hard problems, this has never been proven. Moreover, the class NP also contains all problems which can be solved in polynomial time.

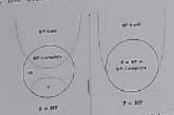
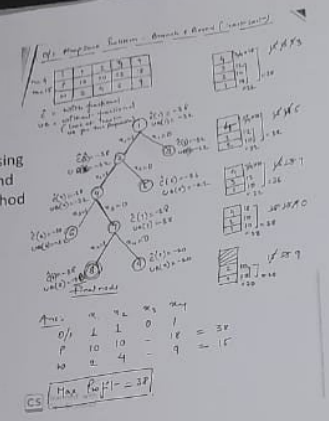


Figure: Euler diagram for P, NP, NP complete, NP hard set of problem.

Definitions :
A decision problem H is NP-hard when for any problem L in NP, there is a polynomial-time reduction from L to H. An equivalent definition is to require that any problem L in NP can be solved in polynomial time by an oracle machine with an oracle for H. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving H, and solves L in polynomial time, if the subroutine call takes only one step to compute.

Another definition is to require that there is a polynomial-time reduction from an NP-complete problem G to H.As any problem L in NP reduces in polynomial time to G, L reduces in turn to H in polynomial time so this new definition implies the previous one. It does not restrict the class NP-hard to decision problems, for instance it also includes search problems, or optimization problems.

0/1 Knapsack using Branch & Bound (Least Cost)Method



Travelling Salesman Problem (TSP)

- **Problem Statement for TSP:** If there are n cities and cost of travelling from any city to any other city is given. Then we have to obtain the cheapest round – trip such that each city is visited exactly once and then returning to starting city, complete the tour.
- Typically travelling salesman problem is represented by weighted graph.
- **Tour(x)** is the path that begins at root and reaching to node x in a state space tree and returns to root. In branch and bound strategy cost of each node x is computed. The travelling salesman problem is solved by choosing the node with optimum cost.

Floyd-Warshall algorithm

Floyd-Warshall algorithm is a procedure, which is used to find the shortest (longest) paths among all pairs of nodes in a graph, which does not contain any cycles of negative lenght. The main advantage of Floyd-Warshall algorithm is its simplicity.

Description

Floyd-Warshall algorithm uses a matrix of lengths D_0 as its input. If there is an edge between nodes i and j, than the matrix D_0 contains its length at the corresponding coordinates. The diagonal of the matrix contains only zeros. If there is no edge between edges i and j, than the position (i, j) contains positive infinity.

In other words, the matrix represents lengths of all paths between nodes that does not contain any intermediate node.

In each iteration of Floyd-Warshall algorithm is this matrix recalculated, so it contains lengths of paths among all pairs of nodes using gradually enlarging set of intermediate nodes. The matrix D_1, which is created by the first iteration of the procedure, contains paths among all nodes using exactly one (predefined) intermediate node. D_2 contains lengths using two predefined intermediate nodes. Finally the matrix D_n uses n intermediate nodes. This transformation can be described using the following recurrent formula:

Floyd's Algorithm:

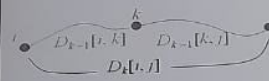
Define the notation $D_k[i, j]$, $1 \le i, j \le n$, and $0 \le k \le n$, that stands for the shortest distance (via a shortest path) from node i to node j, passing through nodes whose number (label) is $\le k$. Thus, when $k = 0$, we have

$D_0[i, j] = W[i][j]$ = the edge weight from node i to node j

This is because no nodes are numbered < 0 (the nodes are numbered 1 through n). In general, when $k \ge 1$,

$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

The reason for this recurrence is that when computing $D_k[i, j]$, this shortest path either doesn't go through node k, or it passes through node k exactly once. The former case yields the value $D_{k-1}[i, j]$; the latter case can be illustrated as follows:



Implementation of Floyd's Algorithm:

Input: The weight matrix W[1..n][1..n] for a weighted directed graph, nodes are labeled 1 through n.

Output: The shortest distances between all pairs of the nodes, expressed in an n × n matrix.

Algorithm:

```
Create a matrix D and initialize it to W.
for k = 1 to n do
   for i = 1 to n do
      for j = 1 to n do
         D[i][j] = min(D[i][j], D[i][k] + D[k][j])
```

Note that one single matrix D is used to store D_{k-1} and D_k, i.e., updating from D_{k-1} to D_k is done immediately. This causes no problems because in the k^{th} iteration, the value of $D_k[i, k]$ should be the same as it was in $D_{k-1}[i, k]$; similarly for the value of $D_k[k, j]$. The time complexity of the above

maximize $\prod_{1<=i<=n} \Phi_i(m_i)$

subject to $\sum_{1<=i<=n} c_i m_i <= c$

$m_i >= 1$ and integer i, $1<=i<=n$

A dynamic programming solution may be obtained in a manner similar to that used for the knapsack problem. Since, we may assume each $c_i > 0$, each m_i must be in the range $1<=m_i<=u_i$ where

$$u_i = \lfloor (c + c_i - \sum_{i=1..n} c_j) / c_i \rfloor$$

The upper bound u_i follows from the observation that $m_j >= 1$. An optimal solution $m_1, m_2, \ldots, m_n$ is the result of a sequence of decisions, one decision for each m_i

Let $f_i(x)$ represent the maximum value of $\prod(m_j)$, $1<=j<=i$ subject to the constraints $\sum_{1<=j<=i} c_j m_j <= x$ and $1<=m_j<=u_j$, $1<=j<=i$. Then, the value of an optimal solution is $f_n(c)$. The last decision made requires one to choose m_n from one of $\{1, 2, 3, \ldots, u_n\}$. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $c - c_n m_n$ in an optimal way. The principal of optimality holds and

$$f_n(c) = \max_{1<=m_n<=u_n} \{ \Phi_n(m_n) f_{n-1}(c - c_n m_n) \}$$

For any $f(x)$, $i>=1$ this equation generalizes to

$$f_i(x) = \max_{1<=m_i<=u_i} \{ \Phi_i(m_i) f_{i-1}(c - c_i m_i) \}$$

Reliability Design:

Reliability means the ability of an apparatus, machine, or system to consistently perform its intended or required function or mission, on demand and without degradation or failure.

Reliability design using dynamic programming is used to solve a problem with a multiplicative optimization function. The problem is to design a system which is composed of several devices connected in series (below Fig-3.3(b)). Let r_i be the reliability of device D_i (i.e. r_i is the probability that device i will function properly). Then, the reliability of the entire system is πr_i. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good.

Fig-3.3(a) Devices connected in series

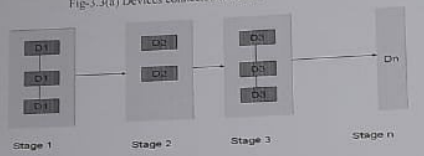


Fig-3.3(b) Multiple Devices Connected in Parallel in each stage

Multiple copies of the same device type are connected in parallel (Fig-3.3(b)) through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

If stage i contains m_i copies of device D_i then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$. Thus, if $r_i = 0.99$ and $m_i = 2$ the stage reliability becomes 0.9999. In any practical situation, the stage reliability will be a little less than $1 - (1 - r_i)^{m_i}$ because the switching circuits themselves are not fully reliable. Also, failures of copies of the same device may not be fully independent (e.g. if failure is due to design defect). Let us assume that the reliability of stage i is actually given by a function $\phi_i(m_i)$, $1<=i<=n$. (It is quite conceivable that $\phi_i(m_i)$ may decrease after a certain value of m_i;). The reliability of the system of stages is $\prod_{1<=i<=n}\phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint.

Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed.
We wish to solve the following maximization problem:

Binary search Tree

Introduction

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a class of data structures used to implement lookup tables and dynamic sets. They store data items, known as keys, and allow fast insertion and deletion of such keys, as well as checking whether a key is present in a tree.

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right sub trees. On average, this means that each comparison allows the operations to skip over half of the tree, so that each lookup/insertion/deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an unsorted array, but slower than the corresponding operations on hash tables.

Definition

A binary search tree is a node-based binary tree data structure where each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub-tree and smaller than the keys in all nodes in that node's right sub-tree. Each non-leaf node has exactly two child nodes. Each child must either be a leaf node or the root of another binary search tree. BSTs are also dynamic data structures, and the size of a BST is only limited by the amount of free memory in the operating system. The main advantage of binary search trees is that it remains ordered, which provides quicker search times than many other data structures. The common properties of binary search trees are as follows:

- One node is designated the root of the tree.
- Each internal node contains a key and has two sub trees.
- The leaves (final nodes) of the tree contain no key. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.
- Each subtree is itself a binary search tree.
- The left sub tree of a node contains only nodes with keys strictly less than the node's key.
- The right sub tree of a node contains only nodes with keys strictly greater than the node's key.

Operations

Searching

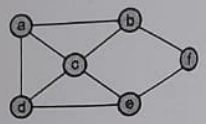
Searching a binary search tree for a specific key can be a recursive or an iterative process. We begin by examining the root node. If the tree is null, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is null. If the searched key is not found before a null subtree is reached, then the item must not be present in the tree. This is easily expressed as a recursive algorithm:

```
function Find-recursive(key, node):  // call initially with node = root
  if node = Null or node.key = key then
    return node
  else if key < node.key then
    return Find-recursive(key, node.left)
  else
    return Find-recursive(key, node.right)
```

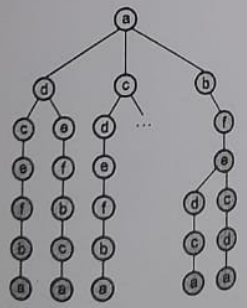
Hamiltonian Cycle

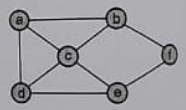
Hamiltonian Cycle: Given an undirected connected graph and two nodes x and y then find a path from x and y visiting each node in the graph exactly once.

Applying Backtracking Algorithm: The state space tree is generated in order to find Hamiltonian Cycles. Using Backtracking Algorithm the Hamiltonian Cycle can be obtained.



Hamiltonian Cycle



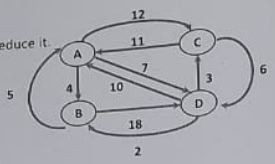


Thus Hamiltonian Circuits are:

1. a-d-c-e-f-b-a
2. a-d-e-f-b-c-a
3. a-c-d-e-f-b-a
4. a-b-f-e-c-d-a
5. a-b-f-e-d-c-a

Travelling Salesman Problem (TSP)

Solution:
Step 1: Write the initial cost matrix and reduce it.



	A	B	C	D
A	∞	4	12	7
B	5	∞	∞	18
C	11	∞	∞	6
D	10	2	3	∞

Travelling Salesman Problem (TSP)

We have

Note:
If 0 present in row and col then no need to do reduction.

	A	B	C	D	Reduce By
A	∞	4	12	7	4
B	5	∞	∞	18	5
C	11	∞	∞	6	6
D	10	2	3	∞	2

Initial Cost Matrix

After Row Reduction

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞
Reduce By	-	-	1	-

After Col Reduction

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

So, The Cost of Node A is –

$$Cost(A) = 4 + 5 + 6 + 2 + 1 = 18$$

The part that is rebuilt uses O(log n) space in the average case and O(n) in the worst case (see big-O notation).

In either version, this operation requires time proportional to the height of the tree in the worst case, which is O(log n) time in the average case over all trees, but O(n) time in the worst case.
Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with a leaf node, and then it is added as this node's right or left child, depending on its key.

There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

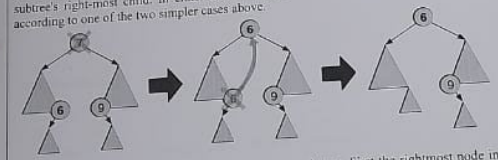
Deletion

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted N. Do not delete N.

Instead, choose either its in-order successor node or its in-order predecessor node, R. Copy the value of R to N, then recursively call delete on R until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of first 2 cases.

Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



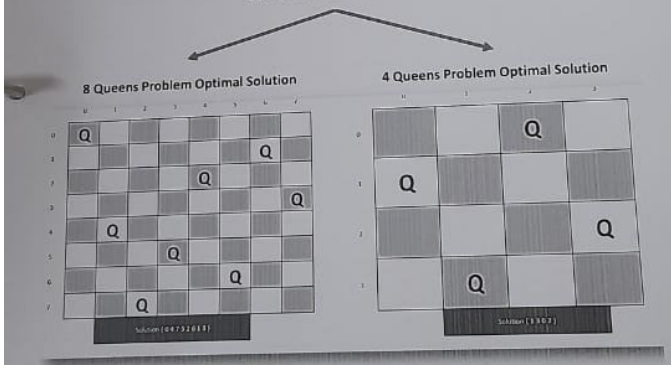
Deleting a node with two children from a binary search tree. First the rightmost node in the left sub tree, the inorder predecessor 6, is identified. Its value is copied into the node being deleted. The inorder predecessor can then be easily deleted because it has at most one child.
The same method works symmetrically using the inorder successor labelled 9.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so some implementations select one or the other at different times.

N Queens Problem

- This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board.
- The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way.
- A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

N Queens Problem

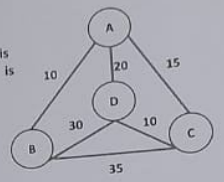


Travelling Salesman Problem (TSP)

Problem: Given a set of cities and distance between every pair of vertices, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting city.

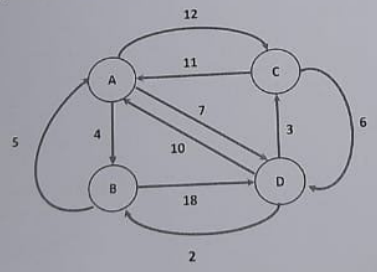
Example:

Consider the graph. A TSP tour in the graph is A-B-D-C-A. The cost of the tour is 10+25+30+15 i.e. 80



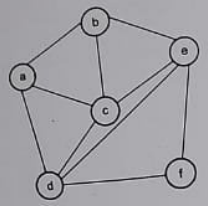
Travelling Salesman Problem (TSP)

Example: Solve the Travelling salesman problem for the following graph by using the Branch and Bound Method.

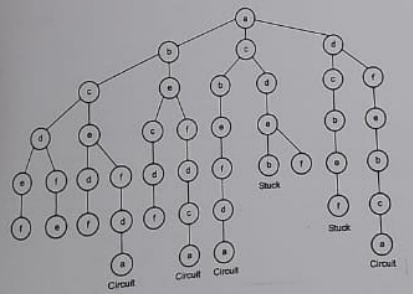


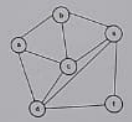
Hamiltonian Cycle

Question: Find the Hamiltonian Circuit using Backtracking Method.



Hamiltonian Cycle





Thus Hamiltonian Circuits are:

1. a-b-c-e-f-d-a
2. a-b-e-f-d-c-a
3. a-c-b-e-f-d-a
4. a-d-f-e-b-c-a

AVL Tree

Introduction

The AVL tree is named after its two Soviet inventors, Georgy Adelson-Velsky and E. M. Landis, who published it in their 1962 paper "An algorithm for the organization of information".

In computer science, an AVL tree (Georgy Adelson-Velsky and Landis' tree, named after the inventors) is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child sub trees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

AVL trees are often compared with red-black trees because both support the same set of operations and take O(log n) time for the basic operations. For lookup-intensive applications, AVL trees are faster than red-black trees because they are more rigidly balanced.[3] Similar to red-black trees, AVL trees are height-balanced. Both are in general not weight-balanced nor μ-balanced for any ; that is, sibling nodes can have hugely differing numbers of descendants.

Searching

Searching for a specific key in an AVL Tree can be done the same way as that of a normal unbalanced Binary Search Tree.

Traversal

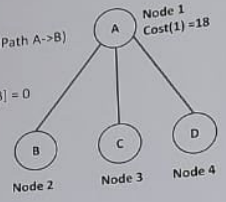
Once a node has been found in a balanced tree, the next or previous nodes can be explored in amortized constant time. Some instances of exploring these "nearby" nodes require traversing up to log (n) links (particularly when moving from the rightmost leaf of the root's left sub tree to the root or from the root to the leftmost leaf of the root's right sub tree; in the example AVL tree, moving from node 14 to the next but one node 19 takes 4 steps). However, exploring all n nodes of the tree in this manner would use each link exactly twice: one traversal to enter the sub tree rooted at that node, another to leave that node's sub tree after having explored it. And since there are n-1 links in any tree, the amortized cost is found to be 2× (n-1)/n, or approximately 2.

Insertion

Pictorial description of how rotations rebalance an AVL tree. The numbered circles represent the nodes being rebalanced. The lettered triangles represent sub trees which are themselves balanced AVL trees. A blue number next to a node denotes possible balance factors (those in parentheses occurring only in case of deletion).

Travelling Salesman Problem (TSP)

Step 2: Choosing to go to vertex B: Node – B (Path A->B)



- From the reduced matrix of step 1, M[A,B] = 0
- Set Row A and Col B to ∞
- Set M[B,A] = ∞
- Now resulting cost matrix is

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	5	∞	∞	0
D	8	∞	0	∞

- Now we reduce this matrix and find the cost of Node - A

Travelling Salesman Problem (TSP)

We have

	A	B	C	D	Reduce By
A	∞	∞	∞	∞	-
B	∞	∞	∞	13	13
C	5	∞	∞	0	-
D	8	∞	0	∞	-

Initial Cost Matrix

After Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	5	∞	∞	0
D	8	∞	0	∞
Reduce By	5	-	-	-

After Col Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	0	∞	∞	0
D	3	∞	0	∞

So, Thus the Cost of Node B is –

$$Cost(B) = Cost(A) + Reduction + M[A,B]$$
$$= 18 + 18 + 0 = 36$$
$$Cost\ (B) = 36$$

Sum of Subset Problem

The Subset-Sum Problem is to find a subset s' of the given set $S = (S_1, S_2, S_3 \ldots S_n)$ where the elements of the set S are n positive integers in such a manner that $s' \subseteq S$ and sum of the elements of subset's' is equal to some positive integer 'X.'

The Subset-Sum Problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in increasing order:

$$S_1 \leq S_2 \leq S_3 \ldots \leq S_n$$

The left child of the root node indicated that we have to include 'S_1' from the set 'S' and the right child of the root indicates that we have to exclude 'S_1'. Each node stores the total of the partial solution elements. If at any stage the sum equals to 'X' then the search is successful and terminates.
The dead end in the tree appears only when either of the two inequalities exists:

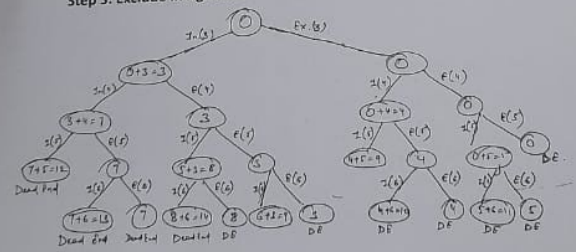
- The sum of s' is too large
- The sum of s' is too small

Sum of Subset Problem

Example: Given a set S = {3, 4, 5, 6} and X =9. Obtain the subset sum using Backtracking approach.

Solution:

Step 1: Start with Zero (as root node)
Step 2: Include in left side
Step 3: Exclude in right side



Ans: Subset 1 = {3, 6} Subset 2 = {4, 5}

the balance factors of all nodes, first observe that all nodes requiring correction lie along the path used during the initial insertion. If the above procedure is applied to nodes along this path, starting from the bottom (i.e. the inserted node), then every node in the tree will again have a balance factor of -1, 0, or 1.
The time required is O (log n) for lookup, plus a maximum of O (log n) retracing levels on the way back to the root, so the operation can be completed in O (log n) time.

Deletion

Let node X be the node with the value we need to delete, and let node Y be a node in the tree we need to find to take node X's place, and let node Z be the actual node we take out of the tree.

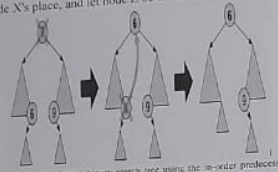


Figure: Deleting a node with two children from a binary search tree using the in-order predecessor (rightmost node in the left sub tree, labelled 6).

Steps to consider when deleting a node in an AVL tree are the following:

1. If node X is a leaf or has only one child, skip to step 5 with Z: =X.
2. Otherwise, determine node Y by finding the largest node in node X's left sub tree (the in-order predecessor of X - it does not have a right child) or the smallest in its right sub tree (the in-order successor of X - it does not have a left child).
3. Exchange all the child and parent links of node X with those of node Y. In this step, the in-order sequence between nodes X and Y is temporarily disturbed, but the tree structure doesn't change.
4. Choose node Z to be all the child and parent links of old node Y = those of new node X.
5. If node Z has a sub tree (which then is a leaf) attach it to Z's parent.
6. If node Z was the root (its parent is null), update root.
7. Delete node Z.
8. Retrace the path back up the tree (starting with node Z's parent) to the root, adjusting the balance factors as needed.

Since with a single deletion the height of an AVL sub tree cannot decrease by more than one, the temporary balance factor of a node will be in the range from -2 to +2.
If the balance factor becomes ±2 then the sub tree is unbalanced and needs to be rotated. The various cases of rotations are depicted in section "Insertion".

UNIT – V

Computationability

Computational complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used.

A computational problem can be viewed as an infinite collection of instances together with a solution for every instance. The input string for a computational problem is referred to as a problem instance, and should not be confused with the problem itself. In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem.

When considering computational problems, a problem instance is a string over an alphabet. Usually, the alphabet is taken to be the binary alphabet (i.e., the set {0,1}), and thus the strings are bitstrings. As in a real-world computer, mathematical objects other than bitstrings must be suitably encoded. For example, integers can be represented in binary notation, and graphs can be encoded directly via their adjacency matrices, or by encoding their adjacency lists in binary.

Decision problems are one of the central objects of study in computational complexity theory. A decision problem is a special type of computational problem whose answer is either yes or no, or alternately either 1 or 0. A decision problem can be viewed as a formal language, where the members of the language are instances whose output is yes, and the non-members are those instances whose output is no. The objective is to decide, with the aid of an algorithm, whether a given input string is a member of the formal language under consideration. If the algorithm deciding this problem returns the answer yes, the algorithm is said to accept the input string, otherwise it is said to reject the input.

To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem. However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve. Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as a function of the size of the instance. This is usually taken to be the size of the input in bits. Complexity theory is interested in how algorithms scale with an increase in the input size.

Beyond Syllabus

Operations

Basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are followed by zero or more operations called tree rotations, which help to restore the height balance of the sub trees.

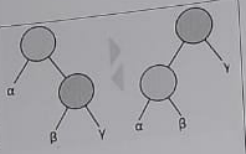
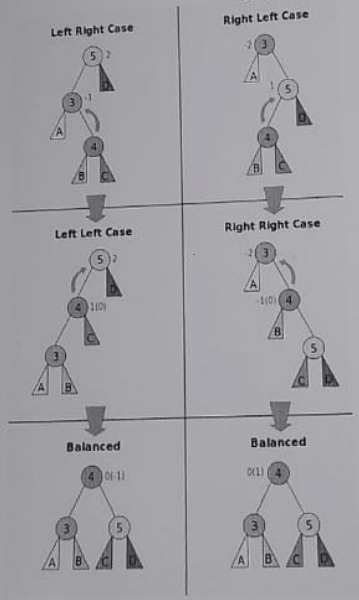


Figure: Tree rotations.



Lower Bound Theory

- Lower Bound theory concept is based upon the calculation of minimum time that is required to execute an algorithm.
- Lower Bound Theory is used to calculate a minimum number of comparisons required to execute an algorithm
- **Lower Bound Theory**: According to lower bound theory, for a lower bound $L(n)$ of an algorithm, it is not possible to have any algorithm (for a common problem) whose time complexity is less than $L(n)$ for random inputs
- Once the lower bound is calculated, then we can compare it with actual complexity of the algorithm & if their order are same then we can declare our algorithm as optimal.

Introduction to Parallel Algorithm

- Consider that, we have to fill up 5 water tanks with water. One tank can be filled up in 10 minutes the 5 such tanks will take in all 50 minutes to get filled up using single tap. But if there are 5 such taps then the task of filling 5 tanks will be done in 10 minutes only. That means if certain task is executed simultaneously then total execution time will be less.

Parallel Computing is a technique in which given problem is subdivided into many sub problems so that each processor can work on a sub problem; and solution from all processors are collected together to form a final solution.

- The algorithm required by a single processor to solve the entire problem is called sequential algorithm
- The algorithm required by multiple processors to solve the sub problem simultaneously is called parallel algorithm.

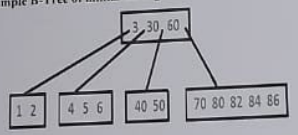
- keeps the index balanced with an elegant recursive algorithm

In addition, a B-tree minimizes waste by making sure the interior nodes are at least half full. A B-tree can handle an arbitrary number of insertions and deletions.

Properties of B-Tree

1) All leaves are at same level.
2) A B-Tree is defined by the term minimum degree'**t**'. The value of t depends upon disk block size.
3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
4) All nodes (including root) may contain at most 2t – 1 key.
5) Number of children of a node is equal to the number of keys in it plus 1.
6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in range from k1 and k2.
7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O (Logn).

Following is **an example B-Tree of minimum degree 3**.



Note that in practical B-Trees, the value of minimum degree is much more than 3.

Search

Search is similar to search in Binary Search Tree. Let the key to be searched be k. We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

address space, it is common to find many subtle couplings between modules that affect the utility of any particular assignment of task to processor. Allocation of a particular task to a processor may restrict our ability to allocate other program elements freely. Therefore, a program's design has hidden effects on its distribution potential. ADAPT's tools ease this problem by providing visibility and modifiability of the distributed nature of the program.

3.3 DISTRIBUTED ADA PROGRAMS

To motivate the following section on the ADAPT tools, we now describe the mask-up and execution profile of a distributed Ada program in ADAPT. Our model places restrictions upon the distributable features of Ada to limit translation complexity and execution overhead. Our contention is that well structured Ada programs will distribute easily even in light of these restrictions. Our basic approach requires that the Ada tasking facilities be used to describe a distributed program. We allow task allocation, activation, rendezvous, and termination of a single Ada program to occur in multiple processors across our distributed hardware base. However, we do not allow an uncontrolled distributed execution, but instead modify a program's distribution to ensure semantic correctness. A performance factor in ensuring this is providing correct access to data objects. In particular, static data items (i.e. declared in package) have a single value across the entire program, and we maintain these in only a single target process. We have chosen to *move the strand of execution* to the location of static data, and to do so on subprogram boundaries. Thus, tasks and subprograms which access static data are suitable for execution only in the single target process where that data exists, and calls to these will result in distributed entry or subprogram calls. Additionally, access of upscope and heap data places further restrictions on the locations of calling and called units to ensure that these forms of data access are satisfied locally.

The model we have described is in direct contrast with distributed systems which provide access to global data on a per access basis. We feel justified in our approach for two related reasons. We have stressed that *well-structured* Ada programs are suitable for distribution by ADAPT. It is commonly agreed that static or global data is to be hidden or encapsulated, with one or more subprograms providing access. In our model these subprograms would be restricted to execute at a single location, but no such restriction is placed upon the callers of these modules. Additionally, we expect that multiple static data accesses will be found in a single subprogram, and thus feel that distributing on subprogram boundaries will provide a good compromise between complete generality and execution efficiency.

Thus, our restrictions define a pragmatic approach to distributed Ada. We ensure a safe and efficient distribution of well-structured programs. Only through actual future experience with ADAPT will we be able to determine the validity of our assumptions and the usefulness of the distributed programming model that ADAPT defines.

3.4 RELATED WORK

Before describing ADAPT tools in detail, we survey and compare related work. Many approaches to create distributed Ada programs have been suggested. All have stressed the importance of tools to eliminate programmer burden and error. One difference among these approaches is the time to specify the mapping of Ada software to distributed hardware.

Tedd [10] has defined the virtual node approach, where the program is to be designed and coded such that tasks in one virtual node communicate only via the rendezvous with tasks in other virtual nodes. A checker tool is provided to ensure no other forms of communication exist between the virtual nodes. Volz [11] has developed an Ada pre-processor which translates a single Ada program into multiple communicating programs. A special pragma, SITE, is used to specify in which target compilable portions of the Ada software are to be assigned. The pre-processor translates accesses and calls to data, tasks, and subprograms assigned to a target distinct from the accessing or calling software, into calls to a local agent. The agents are responsible for interprocessor communication and are transparent to the programmer. The resulting programs are suitable for compilation by a standard Ada compiler. Both of these approaches require that the allocation of software to distributed target be specified at compile time. The advantage is an up-front understanding of the distributed communication and associated overhead that will be borne by the program. The disadvantage of these approaches is the difficulty in modifying the distribution of the program. The virtual node approach may require redesign of the software to incorporate a change in the program's distribution, while the pre-processor approach may only require a re-compilation after adjustment of the pragmas controlling the distribution.

Honeywell's Distributed Ada Project [1,9] has defined a partitioning language that maps an Ada program to a distributed target. CDC, under the ALS/N project, is developing the ability to map large Ada programs onto small memory machines. This is effectively the same as mapping Ada programs to a distributed target, and CDC intends to extend their work to include distributed programming. These efforts allow the allocation of software to distributed hardware to be described independently of the software itself. Advantages are claimed to include separation of target architecture concerns from the design and coding phase, and the ability to easily modify a distributed mapping. ADAPT is designed to support these same ideas.

A Tool Set for Distributed Ada Programming

Gregory McFarland, Peter Brennan, John D. Lipke, Michael S. Restivo

Grumman Data Systems
1000 Woodbury Road, D13
Woodbury N.Y. 11797

ABSTRACT

This paper describes a tool set for a distributed Ada programming testbed. The testbed includes an Ada compiler to produce distributed programs and a unique tool set to permit safe and rapid program partitioning. The testbed, and the tool set in particular, are designed to support rapid prototyping of distributed systems and to support research on software fault tolerant and distributed program construction techniques.

1. INTRODUCTION

The construction of large, complex computing applications is made difficult by many factors, including concurrency and a requirement for fault tolerance. Distributed computing architectures provide suitable support for attacking these problems. The potential advantages of a distributed computing architecture include high throughput, fault tolerance, extensibility, and heterogeneity. Unfortunately, most distributed implementations are application specific. Each solves the distributed communication and control problems specific to its needs. This results in an inefficient development process and a system that is difficult to modify and maintain. We are interested in providing a general purpose implementation vehicle for a wide range of applications so that we may share the development and maintenance costs across these systems.

In a general purpose approach, one should implement distributed systems using a language that supports communication and synchronization of parallel activities. The Ada programming language includes such features and could be used to construct distributed applications. The advantages of writing distributed applications as single programs have been discussed in [1,2]. These include the ability to design the application as one program with a single consistent view, to have distributed interfaces defined and implemented by the language and translator. The result of this unified view is expected improved maintainability and modifiability of the distributed application.

To build a distributed system one must analyze, partition, allocate, refine, and implement (normally more than once). Commonly known methods and tools for constructing systems generally focus on the requirements analysis and detailed design phases, and are especially lacking for the other phases. Of these, the partition and allocate phases are known to be difficult for distributed systems.

Allocation takes on a new meaning when constructing distributed systems implemented as a single program. Though a point of disagreement among current research groups, we consider it preferable to delay allocation until after the software has been written and undergone initial debugging. This simplifies the software implementation effort by allowing designers and programmers to maintain the concept of an abstract target machine, and eliminates the tendency to design for a specific hardware configuration. It also provides the opportunity to modify the allocation without redesigning the software.

By postponing the allocation process, additional complexities are introduced into that process. The allocation phase must not simply assign units of software to units of hardware, but must ensure the semantic integrity of the software for any particular assignment. Given a distributed hardware base and a programmed solution with concurrent activities, there is a potentially large number of ways to allocate software to hardware. This allocation will be constrained by requirements, but rarely to such a degree that only a single allocation is viable. Different assignments may require different software configurations or translations, and tool support is needed to automate the construction of these different allocations and to help system builders understand the effects of a particular allocation. The allocation of software to hardware will also have direct impact upon system throughput. Automated modification of the allocation will simplify the effort required to extend the hardware base and/or the software functionality. Finally, the ability to control and modify the placement of software, both statically and dynamically, provides good support for software fault tolerance.

CH2475-3/88/0000/0071$01.00 © 1988 IEEE

BFS

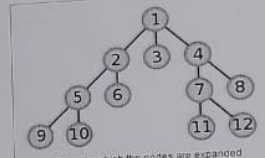
Introduction: Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbour nodes first, before moving to the next level neighbours. Compare BFS with the equivalent, but more memory-efficient Iterative deepening depth-first search and contrast with depth-first search.

BFS was invented in the late 1950s by E. F. Moore, who used it to find the shortest path out of a maze, and discovered independently by C. Y. Lee as a wire routing algorithm (published 1961).

This can be thought of as being like Dijkstra's algorithm for shortest paths, but with every edge having the same length. However it is a lot simpler and doesn't need any data structures. We just keep a tree (the breadth first search tree), a list of nodes to be added to the tree, and markings (Boolean variables) on the vertices to tell whether they are in the tree or list.

breadth first search:
unmark all vertices
choose some starting vertex x
mark x
list L = x
tree T = x
while L nonempty
choose some vertex v from front of list
visit v
for each unmarked neighbor w
mark w
add it to end of list
add edge vw to T

Breadth-first search



Order in which the nodes are expanded

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|E\|) = O(b^d)$
Worst case space complexity	$O(\|V\|) = O(b^d)$

It's very important that you remove vertices from the other end of the list than the one you add them to, so that the list acts as a queue (FIFO storage) rather than a stack (LIFO). The "visit v" step would be filled out later depending on what you are using BFS for, just like the tree traversals usually involve doing something at each vertex that is not specified as part of the basic algorithm. If a vertex has several unmarked neighbours, it would be equally correct to visit them in any order. Probably the easiest method to implement would be simply to visit them in the order the adjacency list for v is stored in.

Let's prove some basic facts about this algorithm. First, each vertex is clearly marked at most once, added to the list at most once (since that happens only when it's marked), and therefore removed from the list at most once. Since the time to process a vertex is proportional to the length of its adjacency list, the total time for the whole algorithm is O(m).

NP-Hard and NP- Complete Problems

P: Problems in class P can be solved with algorithms that run in polynomial time.

Let you have an algorithm that finds the smallest integer in an array. One way to do this is by iterating over all the integers of the array and keeping track of the smallest number you've seen up to that point. Every time you look at an element, you compare it to the current minimum, and if it's smaller, you update the minimum.

How long does this take? Let's say there are n elements in the array. For every element the algorithm has to perform a constant number of operations. Therefore we can say that the algorithm runs in O (n) time, or that the runtime is a linear function of how many elements are in the array. So this algorithm runs in linear time.

You can also have algorithms that run in quadratic time (O (n^2)), exponential time (O (2^n)), or even logarithmic time (O(log n)). Binary search (on a balanced tree) runs in logarithmic time because the height of the binary search tree is a logarithmic function of the number of elements in the tree.

If the running time is some polynomial function of the size of the input**, for instance if the algorithm runs in linear time or quadratic time or cubic time, then we say the algorithm runs in polynomial time and the problem it solves is in class P.

NP: Now there are a lot of programs that don't (necessarily) run in polynomial time on a regular computer, but do run in polynomial time on a nondeterministic Turing machine. These programs solve problems in NP, which stands for nondeterministic polynomial time. A nondeterministic Turing machine can do everything a regular computer can and more.*** This means all problems in P are also in NP.

An equivalent way to define NP is by pointing to the problems that can be verified in polynomial time. This means there is not necessarily a polynomial-time way to find a solution, but once you have a solution it only takes polynomial time to verify that it is correct.

Say P = NP, which means any problem that can be verified in polynomial time can also be solved in polynomial time and vice versa.

NP-hard: What does NP-hard mean? A lot of times you can solve a problem by reducing it to a different problem. I can reduce Problem B to Problem A if, given a solution to Problem A, I can easily construct a solution to Problem B. (In this case, "easily" means "in polynomial time.")

If a problem is NP-hard, this means I can reduce any problem in NP to that problem. This means if I can solve that problem, I can easily solve any problem in NP. If we could solve an NP-hard problem in polynomial time, this would prove P = NP.

NP-complete: A problem is NP-complete if the problem is both

NP-hard, and in NP.

* A technical point: O(n) actually means the algorithm runs in asymptotically linear time, which means the time complexity approaches a line as n gets very large. Also, O(n) is technically an upper bound, so if the algorithm ran in sublinear time you could still say it's O(n), even if that's not the best description of it.

** Note that if the input has many different parameters, like n and k, it might be polynomial in n and exponential in k

3. ADAPT TOOLS

The ADAPT design contains a compiler and runtime environment supporting distributed execution of Ada programs, a distributed program analysis tool, and a distributed program allocation tool. ADAPT generates distributed programs by sequential application of these tools (see Figure 2.). Debugging and program monitoring facilities are also to be included. This section describes the ADAPT compiler, analysis tool, and allocation tool.

3.1 COMPILER

The base compiler is a commercially available, retargetable Ada compiler. The front-end and runtime environment was developed by Telesoft (San Diego, CA). Code generator development and runtime environment modifications necessary for the non-distributed version of our hardware base were completed by Grumman in the summer of 1987. Since one project goal was to produce Ada programs that could be executed or debugged as either single or distributed processes, there were no significant changes to the modified compiler, other than to increase the information collecting capability of the code generator. This additional information provides the necessary data access and control flow information to allow analysis of Ada programs for distribution.

An advantage of this approach is that different distributed control and communication paradigms may be added to the testbed without any change to the compiler. The distributed interfaces are implemented by post-compilation tools and supported by the runtime environment. A further advantage is that a distributed implementation of Ada can be developed relatively quickly and inexpensively in comparison with the effort required to develop a distributed Ada compiler from scratch. A disadvantage of the approach is that the analyses that can be performed is limited by the interface, or information flow, between the compiler's front-end and code generator. This interface provides a very low level program representation, in order to simplify the code generation task. The result is that many source level constructs are transparent to the code generation process. Therefore, program distribution, which requires knowledge of source level constructs (such as types) based solely on code generator analysis is quite complex for some language features. In the remainder of this section we describe a few of the requirements and associated difficulties placed upon the code generator as a collector of program distribution information. Our discussion must remain somewhat general so as not to disclose proprietary front-end compiler technology.

Section 2, *ADAPT Overview*, described the basic make-up and execution profile of a distributed Ada program. From this it should be clear that for each subprogram and task body, the code generator must collect data

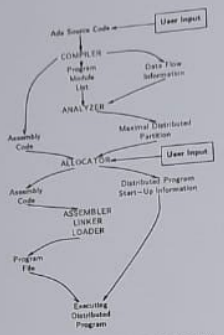


Figure 3. Construction of a Distributed Program

access and call pattern information. This information determines what subprograms and tasks are free to execute in any address space, which are restricted to a single address space, which are restricted to execute in the address space of a lexically superior unit, and which are restricted to execute in a callers address space. These restrictions result from the presence of static data references, upscope references, and some forms of actual parameters, respectively. Collection of this information is not difficult, and its interpretation is discussed in section 3.2.

A more difficult problem for the ADAPT code generator is identification, classification of subcomponent types, and determination of size for composite objects. A distributed implementation requires this information if composite objects are to be passed as parameters. A non-distributed implementation will, for efficiency purposes, pass composites by reference. There are no more than addresses and present no difficulty for the normal code generator. ADAPT has chosen to pass composite objects by copy-restore linkage [12] across any distributed interface. This is justified by our contention that

Depth-first search (DFS)

Introduction : Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.
A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux as a strategy for solving mazes.
Depth first search is another way of traversing graphs, which is closely related to preorder traversal of a tree. Recall that preorder traversal simply visits each node before its children. It is most easy to program as a recursive routine:

```
Preorder (node v)
{
visit(v);
for each child w of v
   preorder(w);
}
```

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|E\|)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor b searched to depth d
Worst case space complexity	$O(\|V\|)$ if entire graph is traversed without repetition, O(longest path length searched) for implicit graphs without elimination of duplicate nodes

To turn this into a graph traversal algorithm, we basically replace "child" by "neighbor". But to prevent infinite loops, we only want to visit each vertex once. Just like in BFS we can use marks to keep track of the vertices that have already been visited, and not visit them again. Also, just like in BFS, we can use this search to build a spanning tree with certain useful properties.

```
dfs(vertex v)
{
visit(v);
for each neighbor w of v
   if w is unvisited
   {
   dfs(w);
   add edge vw to tree T
   }
}
```

The overall depth first search algorithm then simply initializes a set of markers so we can tell which vertices are visited, chooses a starting vertex x, initializes tree T to x, and calls dfs(x). Just like in breadth first search, if a vertex has several neighbours it would be equally correct to go through them in any order. I didn't simply say "for each unvisited neighbour of v" because it is very important to delay the test for whether a vertex is visited until the recursive calls for previous neighbours are finished.

The proof that this produces a spanning tree (the depth first search tree) is essentially the same as that for BFS, so I won't repeat it. However while the BFS tree is typically "short and bushy", the DFS tree is typically "long and stringy".

SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BH

Content Beyond Syllabus

programs are well structured. Passing composite objects in this manner presents a number of difficulties for any implementation. These problems are related to Ada's discriminant record and unconstrained array types. These require runtime size calculation and storage allocation for efficiency purposes. Furthermore, the expected technique is to pass composites by reference.

There are three basic problems presented by composite objects in a distributed execution. First, composite objects are not always easily identifiable. For instance, composites whose size is calculated and storage allocated at runtime are referenced by an address. These addresses are difficult to distinguish from scalar objects. Second, composite objects have arbitrarily typed components. Of these, task, access, and composite object components need to be distinguished. Task components limit the location where the composite can be allocated to where the task may execute. Access and task components restrict a composite object from being a suitable parameter in a remote interface. Finally, composite object components require a similar recursive analysis. Third, to implement pass by copy-restore linkage, the size of the composite object must be known at runtime. For composites whose size is calculated and storage allocated at runtime, special handling is required. In particular, additional code is needed to maintain the dynamic sizes of these objects. The intermediate program representations used to calculate composite object size, allocate composite object storage, and initialize composite object fields (which is necessary to determine the composite object's make-up) is highly varied. This complicates the analysis.

The ADAPT code generator is also responsible for identifying tasks. Tasks may be declared statically in packages, locally in subprograms, task bodies, and blocks, or dynamically as the result of an Ada *new* operation. Tasks may be named or components of composite objects. We require identification of tasks so that post-compilation tools can restrict distribution for tasks accessing static data, and where appropriate, allow the user to specify in what process particular tasks are to be activated. A particularly interesting case is an array of tasks. ADAPT does not require these tasks to be activated in a single process. Instead, post-compilation tools allow the user to selectively assign individual tasks to individual processes if desired.

3.2 ANALYZER

The Analyzer performs a complete program analysis to determine the maximum potential distribution of an Ada program. Recall that ADAPT's realization of a distributed execution is to provide local access to all data items referenced. Thus, static, upscope, and heap data referenced by any subprogram or task must be valid in the execution context of that subprogram or task. The Analyzer is responsible for determining placement of remote subprogram and task entry calls to ensure the required locality of all data references. The result of this analysis will be the *Maximal Distributed Partition* for the given Ada program. By this we mean the maximum number of distinct address spaces (i.e. target processes), each with assigned static data, subprograms, and tasks, such that the distribution rules imposed by ADAPT are satisfied. We refer to these address spaces as *Distribution Units*.

Three forms of analysis are performed by the ADAPT Analyzer tool, each relating to the data access and calling pattern of tasks and subprograms. The combination of the three is used to determine if subprograms and tasks may execute in any Distribution Unit, or are to be restricted to execute in a particular Distribution Unit. Restriction to a particular Distribution Unit results in calls to these tasks and subprograms being remote calls. For purposes of this discussion, a task body can be considered a subprogram with multiple interfaces. In the following discussion we use the term *Callable Unit* to mean procedure, function, or task.

Distribution analysis is dependent solely upon the information collected by the ADAPT Code Generator. This information, referred to generally as *Distribution Information*, includes lexical placement, formal parameter characteristics, call pattern, actual parameter characteristics, and data access pattern for each Callable Unit. Lexical trees and call graphs are built from this information prior to the analysis phase. The Analyzer need only interface with the Distribution Information, lexical trees, and call graphs, and does not need to know intermediate program representations or object code formats.

The first form of analysis defines program distribution based upon static data access. The Analyzer begins by assigning the static data of each compilation in a given program to a unique Distribution Unit. Each Callable Unit is then analyzed for access to static data, which when identified results in that Callable Unit being assigned to the particular Distribution Unit maintaining the referenced static data. This assignment indicates that the Callable Unit may only execute in this particular Distribution Unit. Thus any calls to the Callable Unit from execution activity in other Distribution Units will be remote. Callable Units referencing static data from multiple compilations cause a merge of Distribution Units maintaining the static data of those compilations. Thus, the overall distribution of a program is decreased by a single function, procedure, or task referencing the static data of multiple compilations. We do not expect this form of behavior to be common in a well structured program, although, if present, the Analyzer will restrict distribution to ensure a correct execution.

Notice that ADAPT does not track a Callable Unit's access of individual static data items, but instead tracks access of a compilation's static data. This effectively limits distribution to the number of compilation unit's.

The third Allocator activity is modification of the generated assembly code to achieve the specified distributed execution. Distributed execution of Ada's tasking constructs is implemented purely by the runtime environment. Thus, for tasking, the Allocator needs only to modify program start-up code that will inform the runtime environment of task type location, number of distributed processors, and assignment of processors to target machines. Additionally, the Allocator will add calls to runtime environment modules that implement distributed interfaces to appropriate subprograms. Note that if the program references static data in a restricted fashion, and the user does not assign subprograms to Distribution Units during the allocation process, no distributed subprogram interfaces need be added. The effect is that only small amounts of object code modification are expected.

4. IMPLEMENTATION STATUS

As mentioned in section 1, the ADAPT project represents a long term research project in distributed systems construction and software fault tolerance. The near-term goal of this project is to provide a tasked programming environment to support this research. The approach taken to achieve this goal was to re-target an Ada compiler to a distributed hardware base. This Ada compiler, generating non-distributed programs, was completed in the summer of 1987. The compiler was internally certified to version 1.8 of the ACVC validation suite. The distributed program compiler for a subset of Ada and a prototype tool set was completed in the fourth quarter of 1987. This current version of ADAPT allows named tasks declared in packages of a single Ada program to be distributed among multiple processors. Rendezvous between these tasks is provided by the runtime environment, but currently only scalar parameters may be passed. Timed and conditional entry calls, selective accepts, and exceptions propagated from rendezvous are supported. ADAPT implements the distribution automatically, requiring no user input except to select a process site for each named task. A more complete distributed implementation and tool set, handling compatible parameters, locally declared tasks, remote subprogram invocation, and a safe distributed execution are expected by the end of this year. Advanced distributed and fault tolerant behavior, such as replicated calls, at least once semantics, etc., are to be implemented thereafter.

5. SUMMARY

The unique property of ADAPT is the delineation of distributed program structure to the user. From our experience subtle coupling between distributed elements, as implemented, is common even if the original design is sound. Such coupling is either introduced by incorrect implementation or by the inclusion of existing library routines, and in either case requires automated aids to detect and correct the error. Furthermore, automated analysis permits adjustment of program allocations for performance or behavioral reasons without the risk of introducing subtle module coupling errors. ADAPT addresses these issues by automating the distribution of Ada programs and providing the ability to visualize and modify the distribution.

6. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the helpful comments of Derek Morris and the conference technical committee.

7. REFERENCES

1 Cornhill, Dennis, "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," *Proc. Conf. on Ada Applications and Environments*, Oct 1984, pp 153-162.

2 Volz, Richard A., Trevor N. Mudge, Gregory D. Buzzard and Padmanabhan Krishnan, "Translation and Execution of Distributed Ada Programs: Is It Still Ada?" *IEEE Transactions on Software*, Spring 1987.

3 Brennan, Peter, John D. Lilke, Gregory McFarland, and Michael S. Restivo, "An Adaptable Distributed Ada Programming Testbed" *Grumman Data Systems, RR #3-008*, August 1987.

4 Morris, D. S. and T. Wheeler, "Distributed Program Design in Ada: An Example," *The Second International IEEE Conference on Ada Applications and Environments*, April 6-10, 1986, Miami, Fl.

5 Jha, Rakesh and Dennis Kafura, "Implementation of Ada Synchronization in Embedded, Distributed Systems," *Virginia Polytechnic Institute Tech Rept. TR-85-22*.

6 Kafura, Dennis, "Termination of Ada Tasks in a Distributed Environment," *Virginia Polytechnic Institute Tech Rept. TR-85-16*.

7 Weatherly, Richard M., "A message-based Kernel to Support Ada Tasking," *Proc. Conf. on Ada Applications and Environments*, Oct 1984, pp 136-144.

8 Knight, John C. and John I.A. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems," *Univ. of Virginia Tech Report TR-83-06*, June 1983.

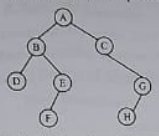
9 Cornhill, Dennis, "Partitioning Ada Programs for Execution on Distributed Systems," *Proc. Intl. Conf. on Data Engr.*, 1984, pp 364-370.

10 Tedd, Mike, Stefano Crespi-Reghizzi and Antonio Natali, Ada for multi-microprocessors, Cambridge University Press, Cambridge, 1984.

11 Volz, R.A., P. Krishnan and R. Theriault, "An Approach To Distributed Execution of Ada Programs," JPL Workshop on Ada Programs, Pasadena, CA, Jan. 20-22, 1987.

12 Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, Compilers, Principles, Techniques, and Tools, Addison-Wesley, Reading, Mass., 1986.

5. a) Show preorder, inorder and postorder for the following tree. 7
निम्न tree से preorder, inorder और postorder दिखाइए।



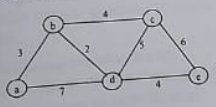
b) Discuss in detail NP complete problems with example. 7
उदाहरण के साथ NP complete problem को विस्तार से समझाइए।

6. a) Construct B tree of order 5 for the list of elements. 7
दी गयी list से order 5 का B tree बनाइए।
2, 8, 5, 6, 13, 9, 14, 12, 19, 24, 18, 15, 5, 16, 20, 21

b) Compare Bfs and Dfs. 7
Bfs और Dfs में तुलना करिये।

7. a) Explain Prim's algorithm with example 7
उदाहरण के साथ Prim's algorithm समझाइए।

b) Solve the following instances of the single source shortest path problem with vertex 'a' as the source. 7
vertex 'a' को source ले के निम्न instances को single source shortest path problem से हल करिए।



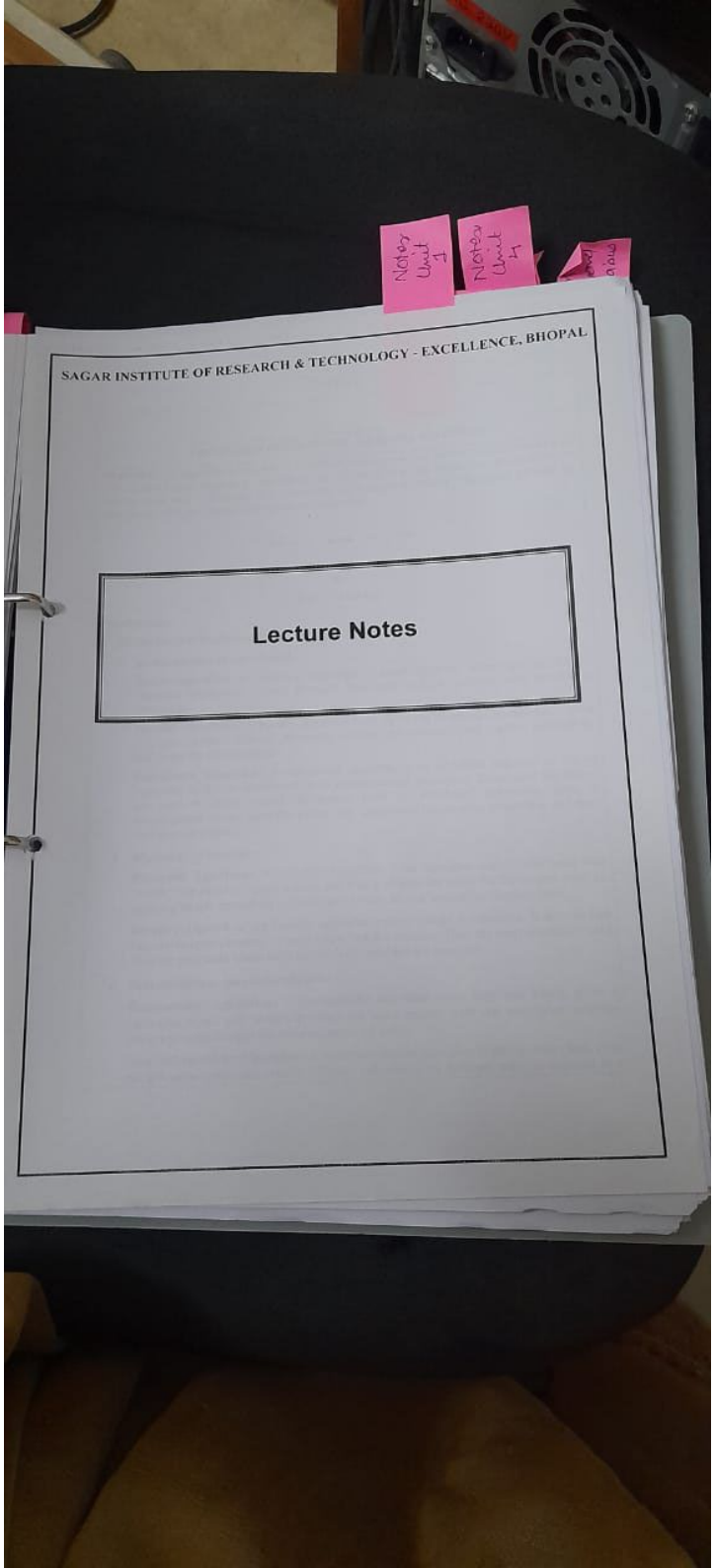
PTO

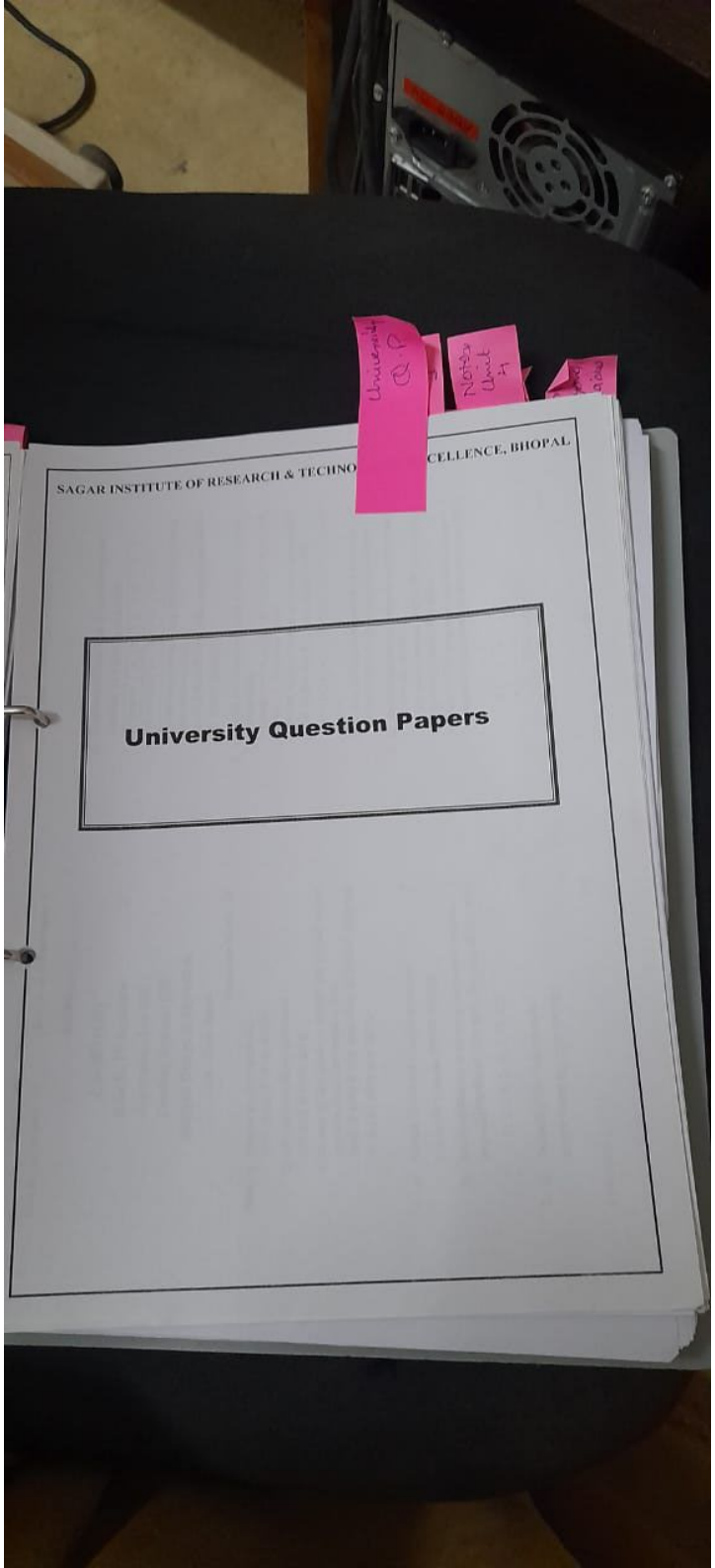
8. Write short notes. 14
संक्षिप्त नोट लिखें।
i) Parallel Algorithm
ii) Graph Coloring
iii) Quick Sort

******

Notes Unit 1

Notes Unit 2


Notes Unit 1
Notes Unit 1
SAGAR INSTITUTE OF RESEARCH & TECHNOLOGY - EXCELLENCE, BHOPAL
Lecture Notes


SAGAR INSTITUTE OF RESEARCH & TECHNO
CELLENCE, BHOPAL
University Question Papers

Recurrence Relation

SOLVING RECURRENCES :- (Happen again (or) repeatedly)

- The indispensable last step when analyzing an algorithm is often to solve a recurrence equation.
- With a little experience and intention, most recurrence can be solved by intelligent guesswork.
- However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically.
- This is a main topic of this section the technique of the characteristic equation.

1. Intelligent guess work:

This approach generally proceeds in 4 stages.

1. Calculate the first few values of the recurrence
2. Look for regularity
3. Guess a suitable general form.
4. And finally prove by mathematical induction (perhaps constructive induction).

Then this form is correct.

Consider the following recurrence.

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ 3T(n-2)+n & \text{otherwise} \end{cases}$$

- First step is to replace $n \div 2$ by $n/2$.
- It is tempting to restrict 'n' to being ever since in that case $n-2 = n/2$, but recursively dividing an even no. by 2, may produce an odd no. larger than 1.
- Therefore, it is a better idea to restrict 'n' to being an exact power of 2.
- First, we tabulate the value of the recurrence on the first few powers of 2.

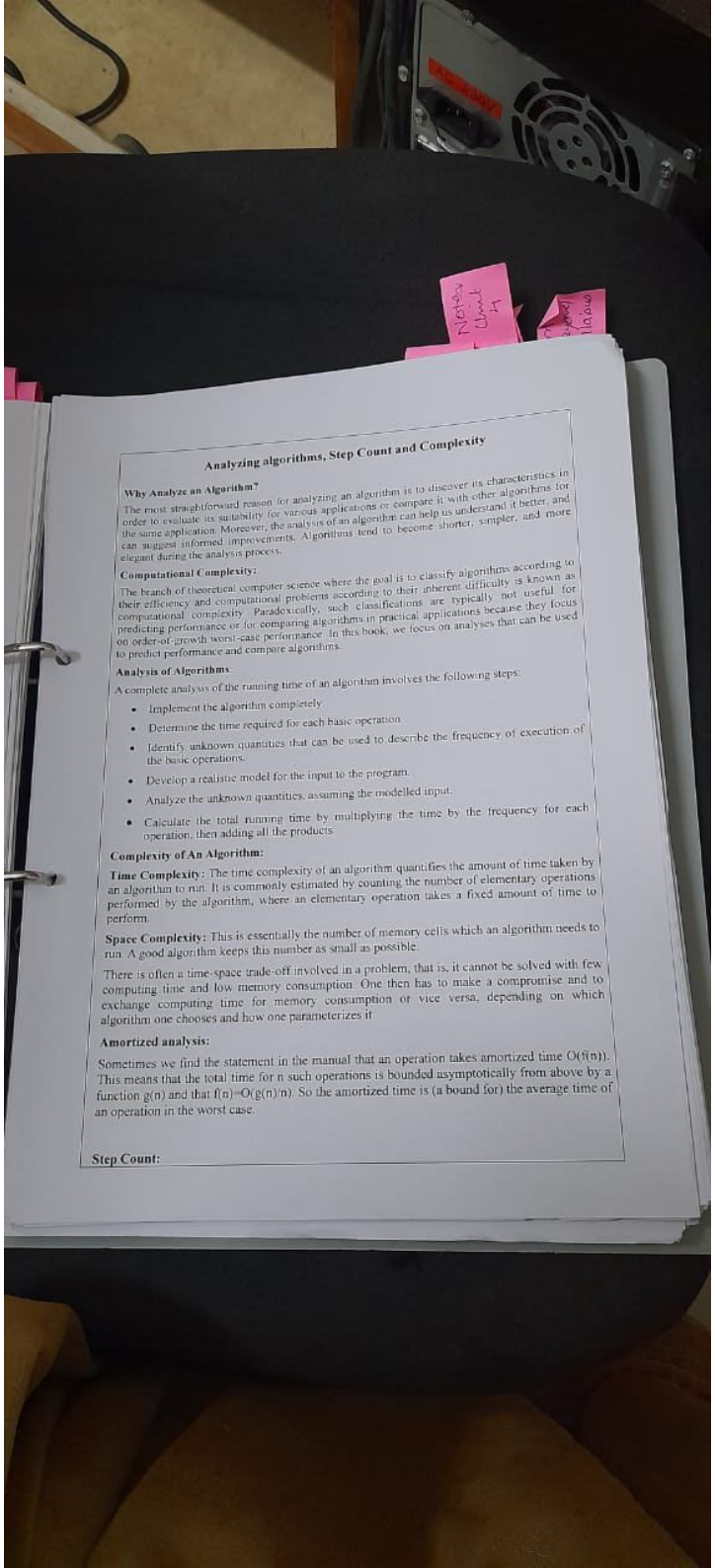
n	1	2	4	8	16	32
T(n)	1	5	19	65	211	665

- For instance, $T(16) = 3 * T(8) + 16$
  $= 3 * 65 + 16$
  $= 211.$
- Instead of writing $T(2) = 5$, it is more useful to write $T(2) = 3 * 1 + 2$.

Then,

$T(A) = 3 * T(2) + 4$
$= 3 * (3 * 1 + 2) + 4$
$= (3^2 * 1) + (3 * 2) + 4$

- We continue in this way, writing 'n' as an explicit power of 2.



Analyzing algorithms, Step Count and Complexity

Why Analyze an Algorithm?

The most straightforward reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. Moreover, the analysis of an algorithm can help us understand it better, and can suggest informed improvements. Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

Computational Complexity:

The branch of theoretical computer science where the goal is to classify algorithms according to their efficiency and computational problems according to their inherent difficulty is known as computational complexity. Paradoxically, such classifications are typically not useful for predicting performance or for comparing algorithms in practical applications because they focus on order-of-growth worst-case performance. In this book, we focus on analyses that can be used to predict performance and compare algorithms.

Analysis of Algorithms

A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modelled input.
- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

Complexity of An Algorithm:

Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run. It is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform.

Space Complexity: This is essentially the number of memory cells which an algorithm needs to run. A good algorithm keeps this number as small as possible.

There is often a time-space trade-off involved in a problem, that is, it cannot be solved with few computing time and low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

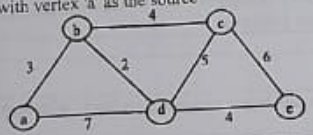
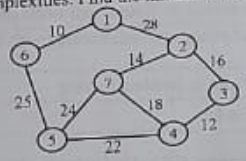
Amortized analysis:

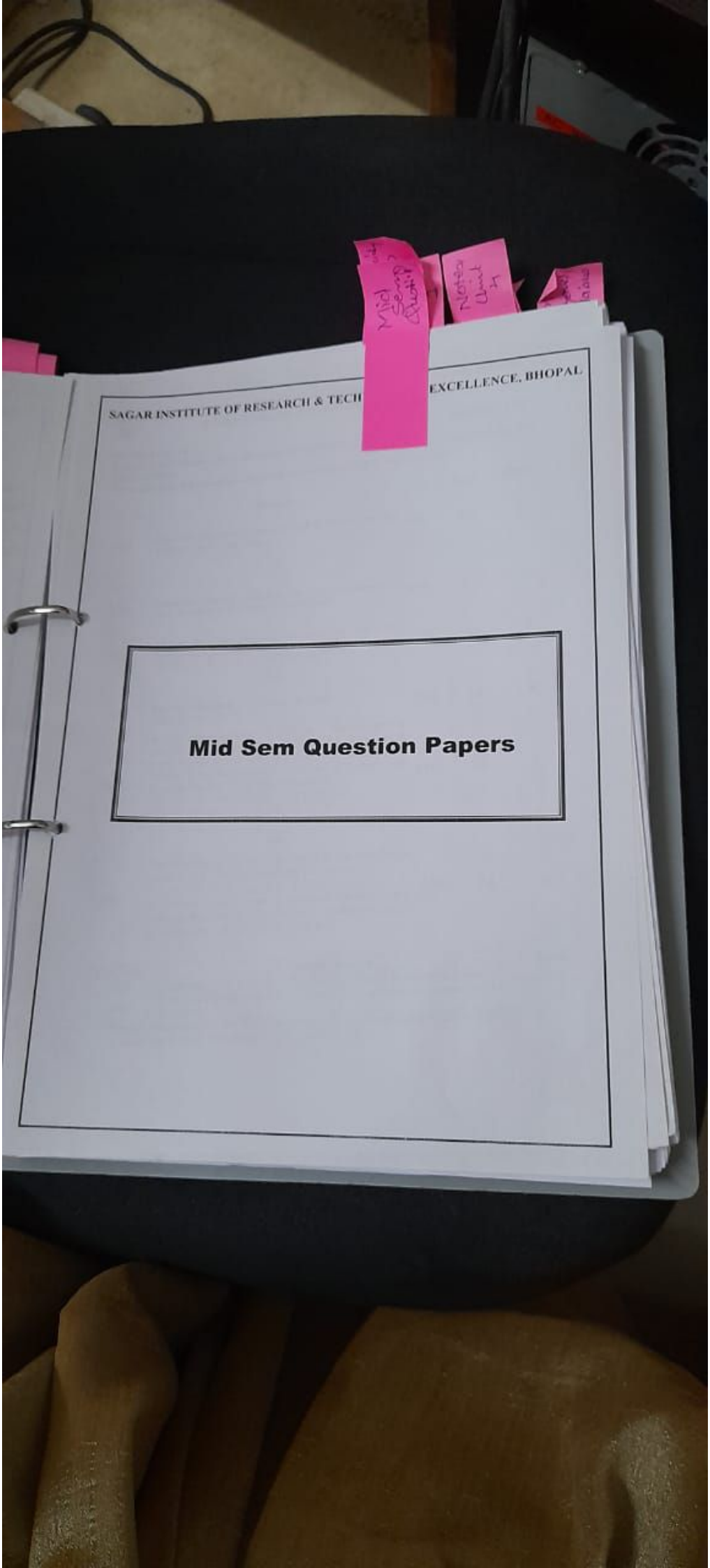
Sometimes we find the statement in the manual that an operation takes amortized time $O(f(n))$. This means that the total time for n such operations is bounded asymptotically from above by a function $g(n)$ and that $f(n)=O(g(n)/n)$. So the amortized time is (a bound for) the average time of an operation in the worst case.

Step Count:

Sagar Institute of Research & Technology-Excellence, Bhopal
II Mid Semester Examination, June - 2023

SEMESTER: IV **BRANCH: CSE**
Subject Code/ Name: CS-402/ ANALYSIS DESIGN OF ALGORITHM
Time: 3:00 Hrs. **Max. Marks: 70**
NOTE: Attempt any five questions. All questions carry equal marks.

Q. No.		Question	COs	Level	Marks
Q.1	(A)	What is the need of obtaining the time and space complexity measures of an algorithm? Justify your answer by some example.	CO1	L2	14
	(B)	Sort the following array using Heap sort. 66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65	CO1	L3	
Q.2	(A)	Show the various steps involved in the quick sorting of (23, 67, 12, 78, 33, 28, 97, 10, 6, 87, 39)	CO1	L3	14
	(B)	Discuss Strassen's matrix multiplication and derive its time complexity.	CO1	L4	
Q.3	(A)	Consider the Knapsack instance n= 4, (w1, w2, w3, w4) = (2, 3, 4, 5) and (p1, p2, p3 ,p4) = (3, 2, 5, 7) and m = 10. Find the optimal solution using greedy method.	CO2	L3	14
	(B)	Obtain a set of optimal Huffman codes for the seven messages (M1,.... , M7) with relative frequencies (q1,.... q7) = (4, 5, 7, 8, 10, 22, 15). Draw the decode tree for this set of codes.	CO2	L3	
Q. 4	(A)	Solve the following instances of the single source shortest path problem with vertex 'a' as the source 	CO2	L3	14
	(B)	Apply Kruskal's and Prim's algorithm for the following graph? Write their time complexities. Find the minimum cost in each case. 	CO2	L3	
Q.5	(A)	Find shortest Travelling salesman path for following graph using branch and bound method. $\begin{array}{c\|ccccc} & 0 & 1 & 2 & 3 & 4 \\ 0 & \infty & 10 & 20 & 0 & 1 \\ 1 & 13 & \infty & 14 & 2 & 0 \\ 2 & 3 & 5 & \infty & 2 & 4 \\ 3 & 19 & 6 & 18 & \infty & 3 \\ 4 & 16 & 4 & 7 & 16 & \infty \end{array}$	CO3	L3	14
	(B)	Explain Hamiltonian cycle and write an algorithm to find all Hamiltonian cycle in graph.	CO4	L3	


SAGAR INSTITUTE OF RESEARCH & TECH
EXCELLENCE, BHOPAL
Mid Sem Question Papers

We set $t_n = C f_n + d g_n$ for arbitrary constants C & d, then t_n is also a solution to equation.

* This is true because.

$a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k}$
$= a_0(C f_n + d g_n) + a_1(C f_{n-1} + d g_{n-1}) + \ldots + a_k(C f_{n-k} + d g_{n-k})$
$= C(a_0 f_n + a_1 f_{n-1} + \ldots + a_k f_{n-k}) +$
$d(a_0 g_n + a_1 g_{n-1} + \ldots + a_k g_{n-k})$
$= C * 0 + d * 0$
$= 0.$

1) (Fibonacci) Consider the recurrence.

$$f_n = \begin{cases} n & \text{if } n=0 \text{ or } n=1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

We rewrite the recurrence as.

$f_n - f_{n-1} - f_{n-2} = 0.$

The characteristic polynomial is.

$x^2 - x - 1 = 0.$

The roots are.

$$x = \frac{-(-1) \pm \sqrt{((-1)2 + 4)}}{2}$$

$$= \frac{1 \pm \sqrt{(1 + 4)}}{2}$$

$$= \frac{1 \pm \sqrt{5}}{2}$$

$$r_1 = \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

The general solution is.

$f_n = C_1 r_1^n + C_2 r_2^n$

when n=0, $f_0 = C_1 + C_2 = 0$
when n=1, $f_1 = C_1 r_1 + C_2 r_2 = 1$

$C_1 + C_2 = 0 \quad \rightarrow (1)$
$C_1 r_1 + C_2 r_2 = 1 \quad \rightarrow (2)$

From equation (1)

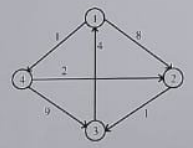
$C_1 = -C_2$

b) What is multistage graph? Write down its properties?
मल्टीस्टेज ग्राफ क्या है? इसके गुणों को लिखिए।

5. Write short notes (any two)
संक्षिप्त नोट लिखे (कोई दो)
a) Subset sort problem
b) Big 'oh' Notation

6. a) In what way is an AVL tree is better than a binary tree insert these keys in to an AVL tree.
किस प्रकार से AVL tree, Binary tree से बेहतर है, इन keys को AVL tree में insert कराइए।
342, 206, 444, 523, 607, 301, 142, 183, 102, 157, 149

b) Use the Floyd-Warshall algorithm and find shortest path between all pairs of vertices for the following graph.
Floyd-Warshall algorithm का प्रयोग करके shortest path निकालना है, हर pair of vertices के बीच, नीचे दिये गये ग्राफ से।



Sagar Institute of Research & Technology-Excellence, Bhopal
I Mid Semester Examination, April -2023

BRANCH: CSE

SEMESTER: IV
Subject Code/ Name: Analysis Design of Algorithm (CS-402) **Max. Marks: 20**
Time: 2:00Hrs
Note: Attempt All Questions. Internal choice is provided in each question.

S. No	Question	CO	Level	Marks
1(a)	Describe Asymptotic Notations? Add different asymptotic notations used in algorithms.	CO1	L2	4
	OR			
1(b)	Articulate Strassen's algorithm for matrix multiplication with an example. Derive its time complexity.	CO1	L2	4
2(a)	Analyse the time complexity of quick sort. Show the various steps involved in the quick sorting of (23, 67, 12, 78, 33, 28, 97, 10, 6, 87, 39)	CO2	L4	8
	OR			
2(b)	Sort the following array using heap sort 5,8,3,9,2,10,1,45,32	CO2	L4	8
3(a)	A thief enters a house for robbing it. He can carry a maximal weight of 60 kg into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him? (w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25) (v1, v2, v3, v4, v5) = (30, 40, 45, 77, 90)	CO2	L4	8
	OR			
3(b)	The characters a to h have the set of frequencies based on the first 8 Fibonacci numbers as follows: a : 1, b : 1, c : 2, d : 3, e : 5, f : 8, g : 13, h : 21 A Huffman code is used to represent the characters. Evaluate the sequence of characters corresponding to the following code : 110111100111010	CO2	L4	8

CO_NO	Course outcome	Bloom
CO1	To apply knowledge of computing and mathematics to algorithm design.	L3
CO2	To analyze a problem and identify the computing requirements appropriate for its solution.	L4

Bloom's Taxonomy Levels:-R: Remembering (L1), U: Understanding (L2), A: Applying (L3), A: Analyzing (L4), E: Evaluating (L5), C: Creating (L6)

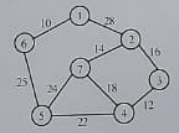
2. a) Sort the following array using Heap sort.
Heap sort तकनीक का प्रयोग करके निम्न array सार्ट करें।
66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65

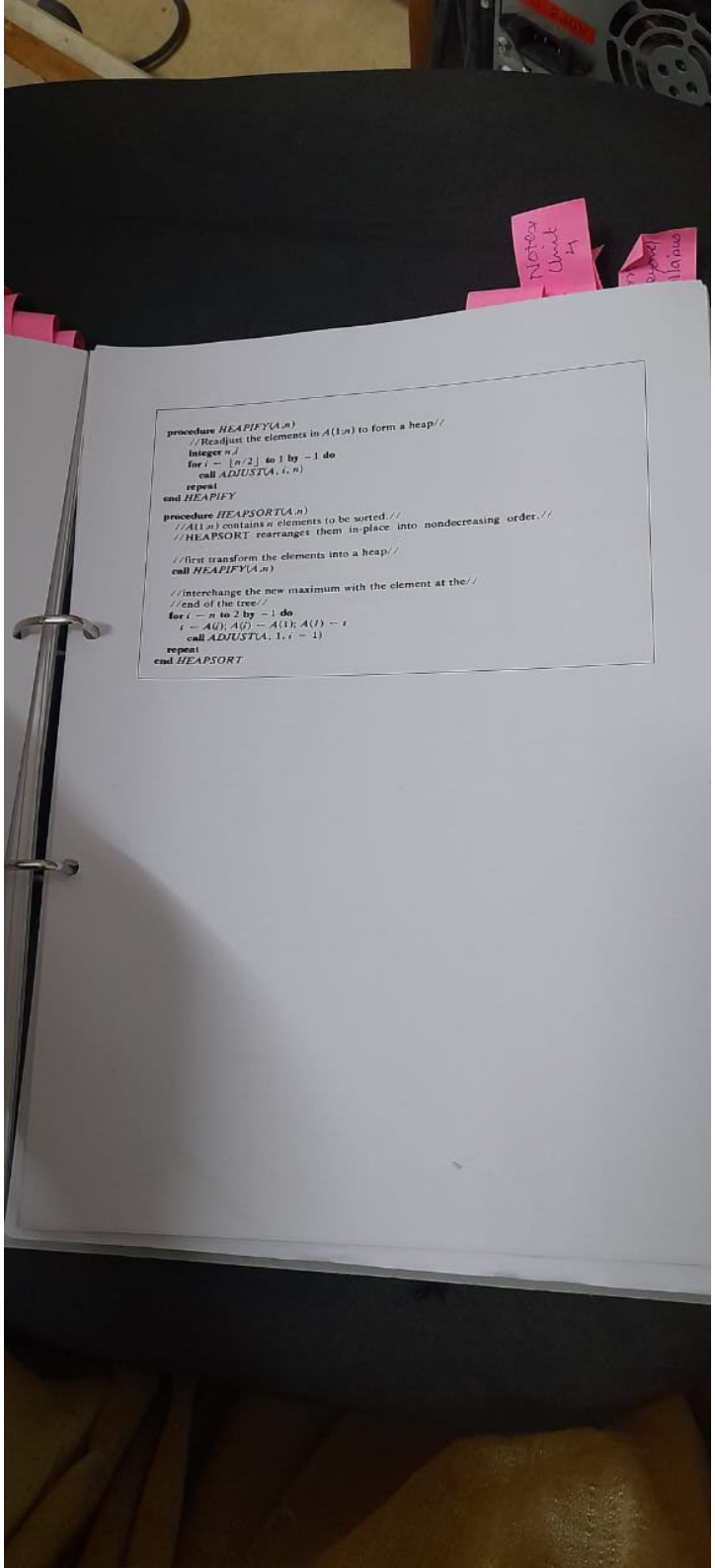
b) Tabulate the differences between Kruskal's and Prim's algorithm.
Kruskal's and Prim's algorithm के बीच अंतर सारणीबद्ध करें।

3. a) What is Knapsack problem? How can we solve using Greedy approach?
Knapsack की समस्या क्या है? हम Greedy approach का उपयोग करके कैसे हल कर सकते है?

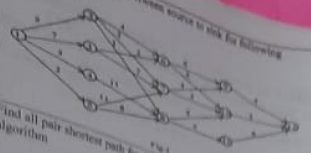
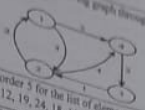
b) Write algorithm for single source shortest path and find its complexity?
Single source shortest path के लिए एल्गोरिदम लिखें और इसकी जटिलता का पता लगाइये।

4. a) Apply Kruskal's and Prim's algorithm for the following graph? Write their time complexities. Find the minimum cost in each case.
निम्नलिखित ग्राफ के लिए Kruskal और Prim's का एल्गोरिदम लागू करें उनके time complexities लिखिए। प्रत्येक मामले में minimum cost ज्ञात कीजिये।



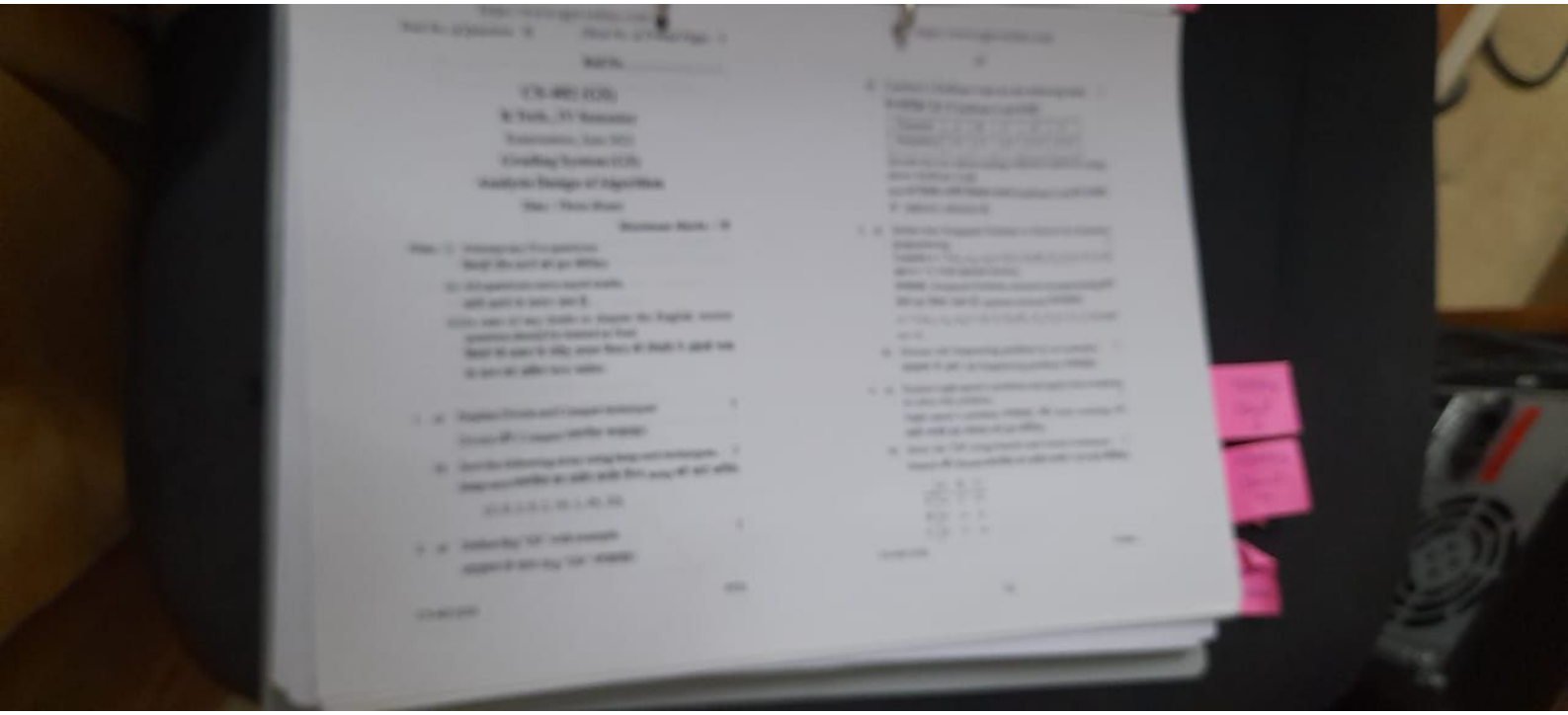



```
procedure HEAPIFY(A,n)
    //Readjust the elements in A(1:n) to form a heap//
    integer n,i
    for i ← ⌊n/2⌋ to 1 by −1 do
      call ADJUST(A, i, n)
    repeat
end HEAPIFY

procedure HEAPSORT(A,n)
  //A(1:n) contains n elements to be sorted.//
  //HEAPSORT rearranges them in-place into nondecreasing order.//

  //first transform the elements into a heap//
  call HEAPIFY(A,n)

  //interchange the new maximum with the element at the//
  //end of the tree//
  for i ← n to 2 by −1 do
    t ← A(i); A(i) ← A(1); A(1) ← t
      call ADJUST(A, 1, i − 1)
  repeat
end HEAPSORT
```

Q.6	(A)	Evaluvate optimum path between source to sink for following multistage graph	CO3	L4	14
					
	(B)	Find all pair shortest path for following graph through Floyd warshal algorithm	CO3	L3	
					
Q.7	(A)	Construct B tree of order 5 for the list of elements 2, 8, 5, 6, 13, 9, 14, 12, 19, 24, 18, 15, 5, 16, 20, 21	CO5	L4	14
	(B)	Consider the following inorder and preorder, reconstruct tree and calculate postorder In order : C,A,D,I,F,B,H,J,G Preorder : D,A,C,B,F,I,G,H,J	CO5	L4	
Q. 8	(A)	Write short notes on (any three) a) Parallel Algorithm b) NP completeness c) Reliability Design d) N queen problem	CO4 CO5 CO3 CO4	L2	14

COs	Course Outcomes	Bloom's Taxonomy Level
CO1	Analyze and compare various divide and conquer algorithm with respect to time and space complexity.	L4
CO2	Analyze various algorithm based on greedy strategies and device mechanism to ensure its correctness.	L4
CO3	Apply concept of Dynamic Programming on various real world problem	L3
CO4	Examine the concept of branch and bound and backtracking with its examples.	L3
CO5	Apply the concepts of trees and graphs in problem solving	L3
CO6	Implement the various algorithms in lab	L3

Bloom's Taxonomy Levels:- R: Remembering (L1), U: Understanding (L2), A: Applying (L3), A: Analyzing (L4), E: Evaluating (L5), C: Creating (L6).



n	T(n)
1	1
2^1	$3^1 * 1 + 2$
2^2	$3^2 * 1 + 3^1 * 2 + 2^2$
2^3	$3^3 * 1 + 3^2 * 2 + 3^1 * 2^2 + 2^3$
2^4	$3^4 * 1 + 3^3 * 2 + 3^2 * 2^2 + 3^1 * 2^3 + 2^4$
2^5	$3^5 * 1 + 3^4 * 2 + 3^3 * 2^2 + 3^2 * 2^3 + 3^1 * 2^4 + 2^5$

- The pattern is now obvious.

$T(2^k) = 3^k 2^0 + 3^{k-1} 2^1 + 3^{k-2} 2^2 + \ldots + 3^1 2^{k-1} + 3^0 2^k$.

$= \sum_{i=0}^{k} 3^{k-i} 2^i$

$= 3^k \sum_{i=0}^{k} (2/3)^i$

$= 3^k * [(1 - (2/3)^{k+1}) / (1 - (2/3))]$

$= 3^{k+1} - 2^{k+1}$

Proposition: (Geometric Series)

Let S_n be the sum of the first n terms of the geometric series a, ar, ar^2.... Then $S_n = a(1-r^n) / (1-r)$, except in the special case when $r = 1$; when $S_n = a_n$

$= 3^k * [(1 - (2/3)^{k+1}) / (1 - (2/3))]$

$= 3^k * [((3^{k+1} - 2^{k+1}) / 3^{k+1}) / ((3 - 2) / 3)]$

$= 3^k * \left[\frac{3^{k+1} - 2^{k+1}}{3^{k+1}} * \frac{3}{1}\right]$

$= 3^k * \frac{3^{k+1} - 2^{k+1}}{3^{k+1}}$

$= 3^{k+1} - 2^{k+1}$

* It is easy to check this formula against our earlier tabulation.

Eg 2

$$t_n = \begin{cases} 0 & n=0 \\ 5 & n=1 \\ 3t_{n-1} - 4t_{n-2}, \text{ otherwise} \end{cases}$$

$t_n = 3t_{n-1} - 4t_{n-2} = 0$ → General function

Characteristics Polynomial, $x^2 - 3x - 4 = 0$

$(x - 4)(x + 1) = 0$

Roots $r_1 = 4$, $r_2 = -1$

General Solution, $f_n = C_1 r_1^n + C_2 r_2^n$ → (A)

$n=0$ → $C_1 + C_2 = 0$ → (1)

UNIT – I

Introduction to Algorithms, Designing algorithms

Algorithm: an algorithm is any set of detailed instructions which results in a predictable end-state from a known beginning. Algorithms are only as good as the instructions given, however, and the result will be incorrect if the algorithm is not properly defined. Algorithms are used for calculation, data processing, and automated reasoning.

Fig 1.1.1: Algorithm

Classification:

- **On the Basis of Implementation**
  - **Serial, parallel or distributed:**

    Serial Algorithm: A sequential algorithm or serial algorithm is an algorithm that is executed sequentially – once through, from start to finish, without other processing executing.

    Parallel Algorithm: A parallel algorithm is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result.

    Distributed Algorithm: A distributed algorithm is an algorithm designed to run on computer hardware constructed from interconnected processors. Distributed algorithms are used in many varied application areas of distributed computing, such as telecommunications, scientific computing, distributed information processing, and real-time process control.

  - **Recursion or iteration:**

    Recursive Algorithm: A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.

    Iterative Algorithm: An iterative algorithm executes steps in iterations. It aims to find successive approximation in sequence to reach a solution. They are most commonly used in linear programs where large numbers of variables are involved.

  - **Deterministic or non-deterministic:**

    Deterministic Algorithm: A deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states.

    Non-Deterministic Algorithm: A nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviours on different runs, as opposed to a

Total No. of Questions : 8] [Total No. of Printed Pages : 4

Roll No .....

CS-402 (GS)

B.Tech., IV Semester

Examination, June 2022

Grading System (GS)

Analysis Design of Algorithm

Time : Three Hours *Maximum Marks : 70*

Note: i) Attempt any five questions.
किन्हीं पाँच प्रश्नों को हल कीजिए।

ii) All questions carry equal marks.
सभी प्रश्नों के समान अंक हैं।

iii) In case of any doubt or dispute the English version question should be treated as final.
किसी भी प्रकार के संदेह अथवा विवाद की स्थिति में अंग्रेजी भाषा के प्रश्न को अंतिम माना जायेगा।

1. a) Explain Divide and Conquer techniques 7
Divide और Conquer तकनीक समझाइए।

b) Sort the following array using heap-sort techniques. 7
heap-sort तकनीक का प्रयोग करके निम्न array को शार्ट करिये।

(5, 8, 3, 9, 2, 10, 1, 45, 32)

2. a) Define Big "Oh" with example. 7
उदाहरण के साथ Big "Oh" समझाइए।

PTO

b) Construct a Huffman Code for the following data: 7
निम्नलिखित डेटा से Huffman Code बनाइए

Character	A	B	C	D	E
Probability	0.4	0.1	0.2	0.15	0.15

Decode the text whose ending 1000101111001010 using above Huffman Code.
text को डिकोड करिये जिसका समाप्त Huffman Code के उपयोग से 1000101111001010 है।

3. a) Define how Knapsack Problem is Solved by dynamic programming. 7
Consider $n = 3$ $(w_1, w_2, w_3) = (2, 3, 3)$, $(P_1, P_2, P_3) = (1, 2, 4)$ and $m = 6$. Find optimal solution.
समझाइए, Knapsack Problem, dynamic programming द्वारा कैसे हल किया जाता है। optimal solution निकालिये।
$n = 3$ $(w_1, w_2, w_3) = (2, 3, 3)$, $(P_1, P_2, P_3) = (1, 2, 4)$ and $m = 6$

b) Discuss Job Sequencing problem by an example. 7
उदाहरण के द्वारा Job Sequencing problem समझाइए।

4. a) Explain eight queen's problem and apply back tracking to solve this problem. 7
Eight queen's problem समझाइए और back tracking का प्रयोग करते इस समस्या को हल कीजिए।

b) Solve the TSP using branch and bound technique. 7
Branch और Bound तकनीक का प्रयोग करते TSP हल कीजिए।

	A	B	C
A	∞	3	4
B	6	∞	4
C	3	5	∞

Contd...

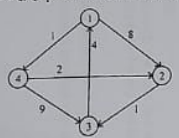
Note Unit 1

Note Unit 4

5. Write short notes (any two) 14
संक्षिप्त नोट लिखे (कोई दो)
   a) B Tree
   b) Subset sort problem
   c) Big 'oh' Notation

6. a) In what way is an AVL tree is better than a binary tree insert these keys in to an AVL tree. 7
किस प्रकार से AVL tree, Binary tree से बेहतर है, इन keys को AVL tree में insert कराइए।
342, 206, 444, 523, 607, 301, 142, 183, 102, 157, 149

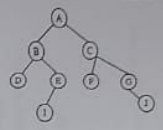
   b) Use the Floyd-Warshall algorithm and find shortest path between all pairs of vertices for the following graph. 7
Floyd-Warshall algorithm का प्रयोग करके shortest path निकालना है, हर pair of vertices के बीच, नीचे दिये गये ग्राफ से।



7. a) Solve the TSP using Branch and Bound Techniques. 7
Branch और Bound तकनीक का प्रयोग करके TSP हल कीजिए।

	A	B	C
A	∞	3	4
B	6	∞	4
C	3	5	∞

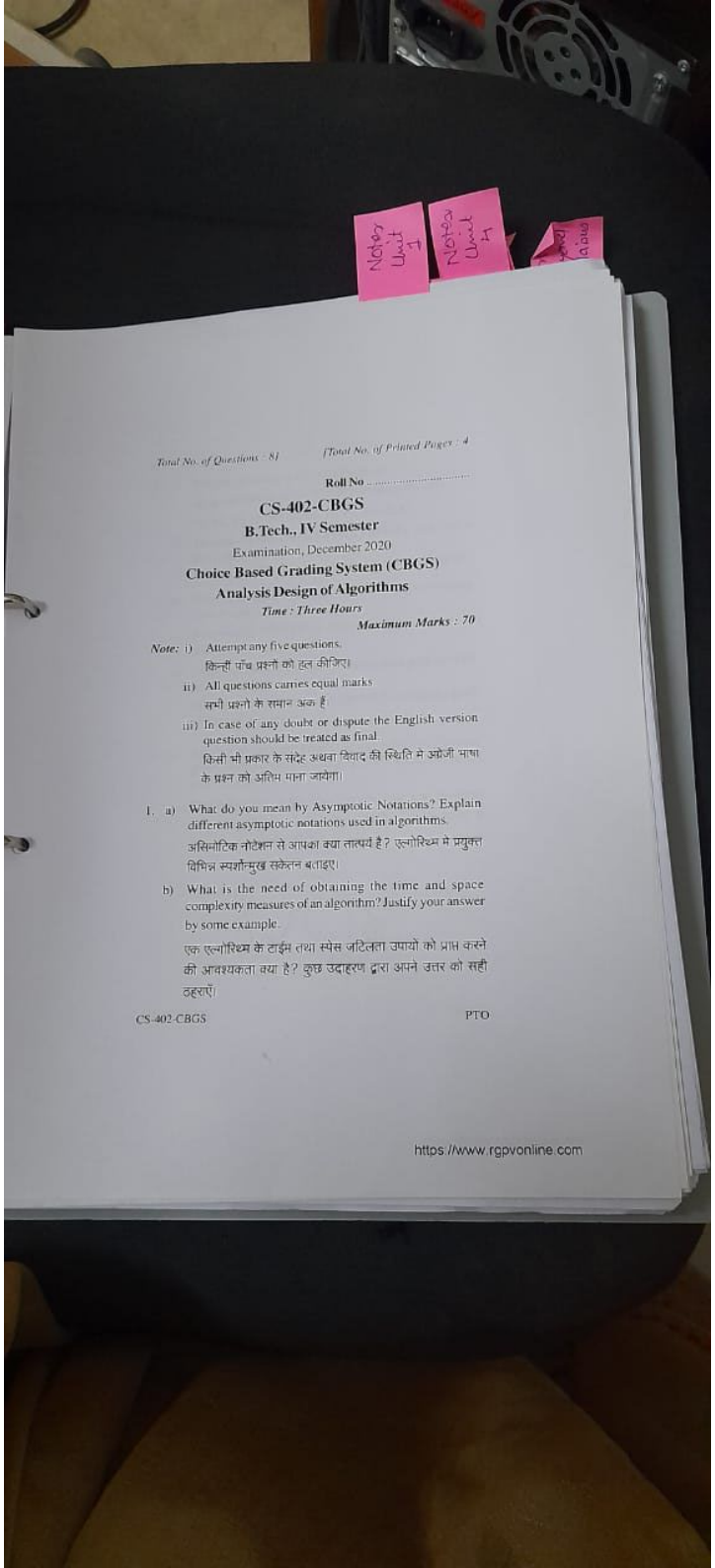
   b) Show Preorder, Inorder and Postorder for the following tree. 7
निम्न tree से Preorder, Inorder, और Postorder दिखाइए।



8. Write short notes. 14
संक्षिप्त नोट लिखें।
   i) Multistage graphs
   ii) Parallel algorithm
   iii) NP complete problem

******

Notes Unit 1
Notes Unit 2



Total No. of Questions : 8] [Total No. of Printed Pages : 4

Roll No .....

CS-402-CBGS

B.Tech., IV Semester

Examination, December 2020

Choice Based Grading System (CBGS)

Analysis Design of Algorithms

Time : Three Hours *Maximum Marks : 70*

Note: i) Attempt any five questions.
किन्हीं पाँच प्रश्नों को हल कीजिए।

ii) All questions carries equal marks
सभी प्रश्नो के समान अंक हैं

iii) In case of any doubt or dispute the English version question should be treated as final.
किसी भी प्रकार के संदेह अथवा विवाद की स्थिति मे अंग्रेजी भाषा के प्रश्न को अंतिम माना जायेगा।

1. a) What do you mean by Asymptotic Notations? Explain different asymptotic notations used in algorithms.
असिम्पोटिक नोटेशन से आपका क्या तात्पर्य है? एल्गोरिथ्म मे प्रयुक्त विभिन्न स्पर्शोन्मुख संकेतन बताइए।

b) What is the need of obtaining the time and space complexity measures of an algorithm? Justify your answer by some example.
एक एल्गोरिथ्म के टाईम तथा स्पेस जटिलता उपायों को प्राप्त करने की आवश्यकता क्या है? कुछ उदाहरण द्वारा अपने उत्तर को सही ठहराएँ।

Total No. of Questions : 8] [Total No. of Printed Pages : 4

Roll No.....

CS-402 (CBGS)

B.Tech., IV Semester

Examination, November 2019

Choice Based Grading System (CBGS)

Analysis Design of Algorithm

Time : Three Hours *Maximum Marks : 70*

Note: i) Attempt any five questions.
किन्हीं पाँच प्रश्नों को हल कीजिए।

ii) All questions carry equal marks.
सभी प्रश्नों के समान अंक हैं।

iii) In case of any doubt or dispute the English version question should be treated as final.
किसी भी प्रकार के संदेह अथवा विवाद की स्थिति में अंग्रेजी भाषा के प्रश्न को अंतिम माना जायेगा।

1. a) What is the running time of Quick Sort Algorithm when all elements of array a have the same value? 7
जब array A के सभी तत्वों (elements) का समान Value होता है तो Quick Sort Algorithm का running time क्या होगा।

b) Define algorithm. Discuss how to analyse Algorithms. 7
Algorithms समझाइए। Algorithms कैसे analyse होता है समझाइए।

2. a) Sort the following array using Heap sort. 7
Heap sort तकनीक का प्रयोग करके निम्न array शार्ट करे।
66, 33, 40, 20, 50, 88, 60, 11, 77, 30, 45, 65

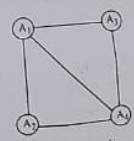
b) Tabulate the differences between Kruskal's and Prim's algorithm. 7
Kruskal's and Prim's algorithm के बीच अंतर सारणीबद्ध करे।

3. a) Discuss job sequencing problem by an example. 7
Job sequencing problem को उदाहरण के साथ समझाइए।

b) Find an optimal solution to the Knapsack instance 7
$n = 3, m = 20, (P_1, P_2, P_3) = (25, 24, 15)$ and
$(w_1, w_2, w_3) = (18, 15, 10)$
Knapsack Instance का Opimal solution निकालिए।
$n = 3, m = 20, (P_1, P_2, P_3) = (25, 24, 15)$ and
$(w_1, w_2, w_3) = (18, 15, 10)$

4. a) A text is made up of the character a, b, c, d, e each occuring with the probability 0.11, 0.40, 0.16, 0.09 and 0.24 respectively the optimal Huffman coding technique will have the average length of. 7
एक text अक्षर a, b, c, d, e से बना है, जो क्रमशः 0.11, 0.40, 0.16, 0.09 और 0.24 probability के साथ होता है, तो Optimal Huffman Coding तकनीक की औसत लंबाई क्या होगी?

b) Colour the following graph using a vertex colouring algorithm. What is the minimum number of colour required? 7
निम्न ग्राफ को कलर करिए, vertex colouring algorithm का प्रयोग करके। रंग की न्यूनतम संख्या क्या है?



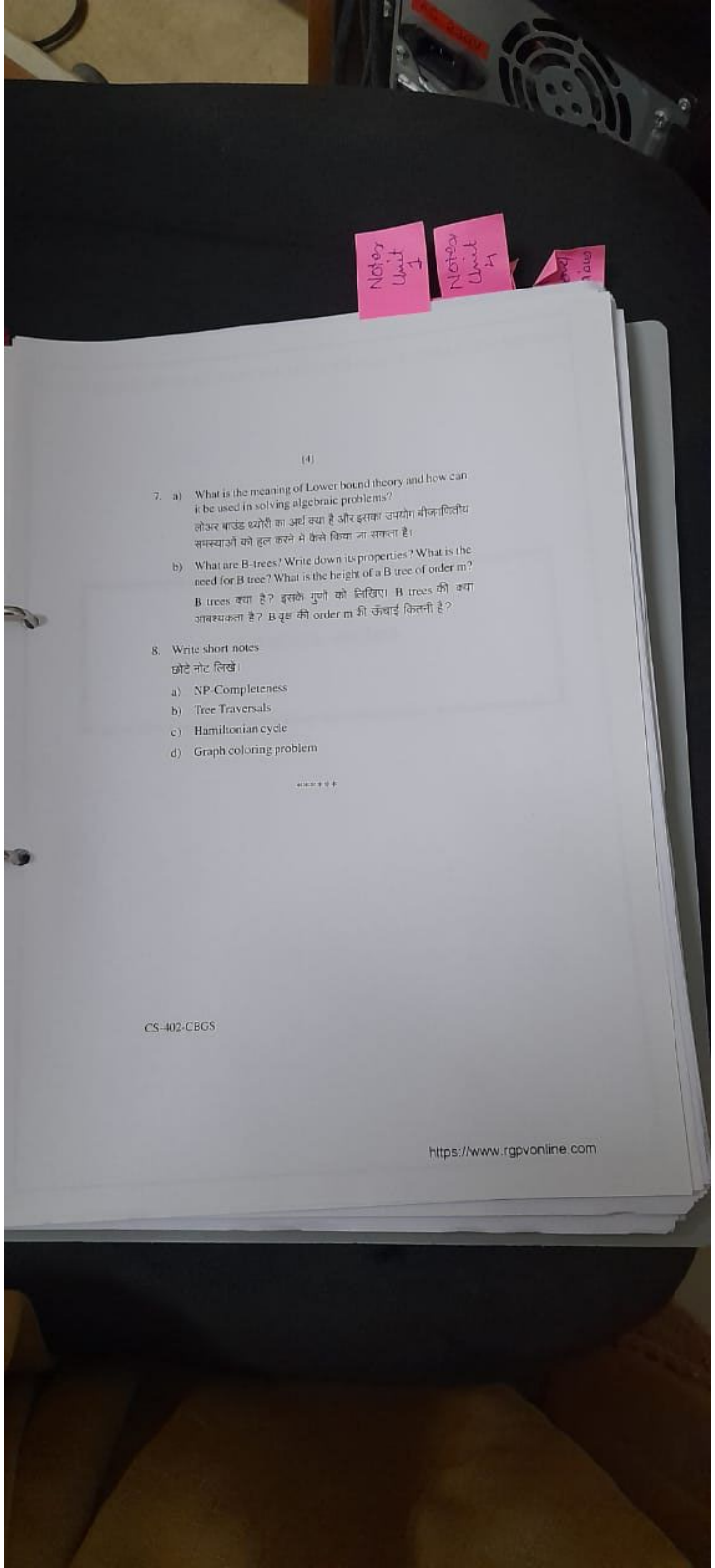
Notes Unit 1

Notes Unit 4

deterministic algorithm. Exact or approximate

➢ **On the Basis of Design**

- **Divide and conquer:** A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily.
- **Dynamic programming:** It is a method for efficiently solving a broad range of search and optimization problems which exhibit the characteristics of overlapping sub problems and optimal substructure.
- **Greedy Algorithms:** A greedy algorithm is a mathematical process that looks for simple, easy-to-implement solutions to complex, multi-step problems by deciding which next step will provide the most obvious benefit. Such algorithms are called greedy because while the optimal solution to each smaller instance will provide an immediate output, the algorithm doesn't consider the larger problem as a whole. Once a decision has been made, it is never reconsidered.
- **Backtracking Algorithm:** Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.
- **Branch & Bound:** Branch and bound algorithms are a variety of adaptive partition strategies have been proposed to solve global optimization models. These are based upon partition, sampling, and subsequent lower and upper bounding procedures



7. a) What is the meaning of Lower bound theory and how can it be used in solving algebraic problems?
लोअर बाउंड थ्योरी का अर्थ क्या है और इसका उपयोग बीजगणितीय समस्याओं को हल करने में कैसे किया जा सकता है।

b) What are B-trees? Write down its properties? What is the need for B tree? What is the height of a B tree of order m?
B trees क्या है? इसके गुणों को लिखिए। B trees की क्या आवश्यकता है? B वृक्ष की order m की ऊँचाई कितनी है?

8. Write short notes
छोटे नोट लिखें।
a) NP-Completeness
b) Tree Traversals
c) Hamiltonian cycle
d) Graph coloring problem

******

- s/e is the number of steps per execution of the statement.
- Frequency is how often each statement is executed.
- The time complexity is estimated as Total steps.

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2. {	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

Table 1.2.1: Step Count of Sequential Search

Worst-case complexity: The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. It represents the curve passing through the highest point of each column.

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. It represents the curve passing through the lowest point of each column.

Average-case complexity: The average-case complexity of the algorithm is the function defined by the average number of steps taken on any instance of size n.

Asymptotic Notations:

The goal of computational complexity is to classify algorithms according to their performances.

Definition of "big Oh"

For any monotonic functions f(n) and g(n) from the positive integers to the positive integers, we say that f(n) = O(g(n)) when there exist constants c > 0 and n0 > 0 such that

f(n) ≤ c * g(n), for all n ≥ n0

Intuitively, this means that function f(n) does not grow faster than g(n), or that function g(n) is an upper bound for f(n), for all sufficiently large n→∞.

Here is a graphic representation of f(n) = O(g(n)) relation:

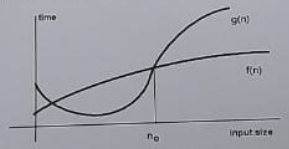
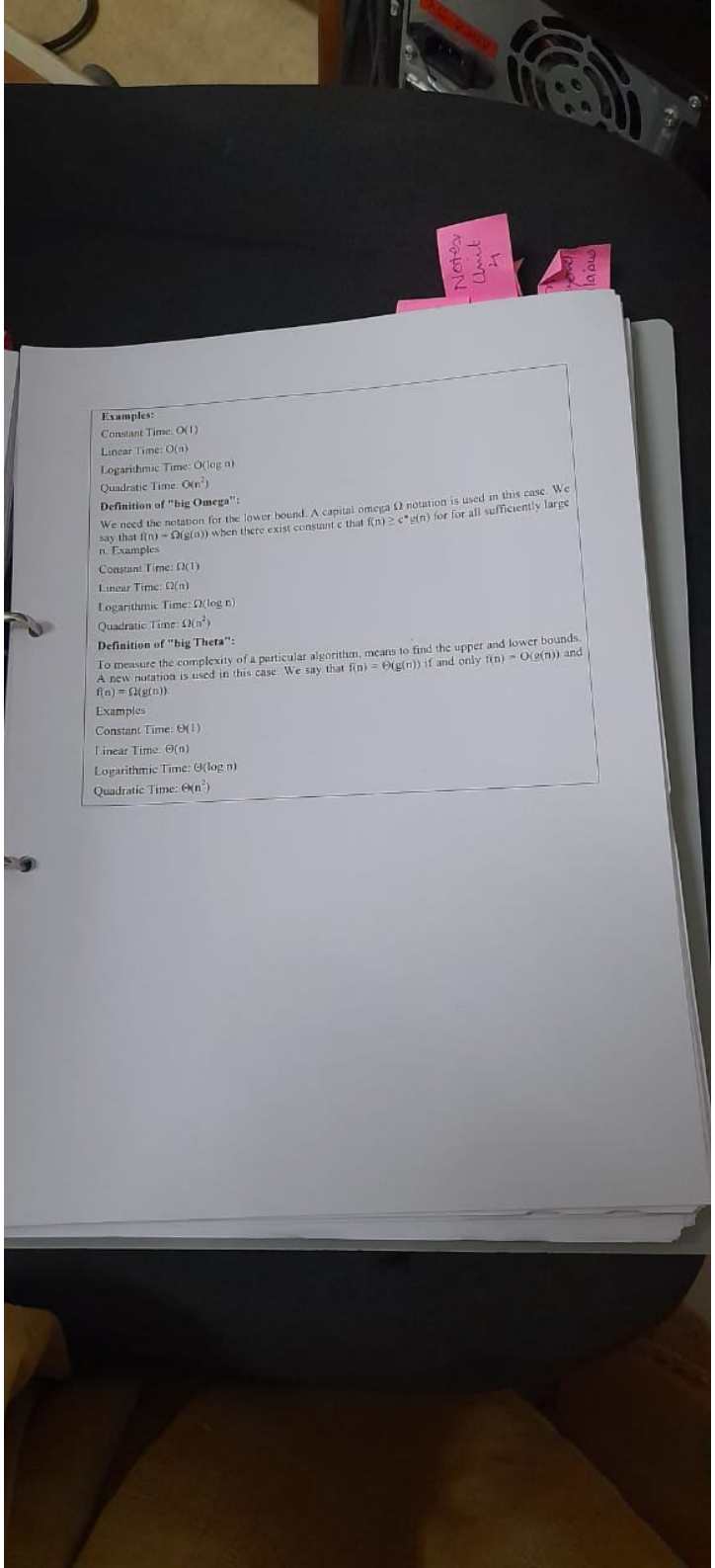


Fig 1.2.1: Graph of Big Oh Notation



Examples:

Constant Time: O(1)

Linear Time: O(n)

Logarithmic Time: O(log n)

Quadratic Time: $O(n^2)$

Definition of "big Omega":

We need the notation for the lower bound. A capital omega Ω notation is used in this case. We say that f(n) = Ω(g(n)) when there exist constant c that f(n) ≥ c*g(n) for for all sufficiently large n. Examples

Constant Time: Ω(1)

Linear Time: Ω(n)

Logarithmic Time: Ω(log n)

Quadratic Time: $\Omega(n^2)$

Definition of "big Theta":

To measure the complexity of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case. We say that f(n) = Θ(g(n)) if and only f(n) = O(g(n)) and f(n) = Ω(g(n)).

Examples

Constant Time: Θ(1)

Linear Time: Θ(n)

Logarithmic Time: Θ(log n)

Quadratic Time: $\Theta(n^2)$

Heap and Heap Sort

Heap definition: A heap is a complete binary tree with the property that the value at each node is at least as large as the values at its children.

This definition implies that a largest element is at the root of the heap. If the elements are distinct, then the root contains the largest item. The relation greater than or equal to may be reversed so that the parent node contains a value as small as or smaller than its children. In this case the root contains the smallest element.

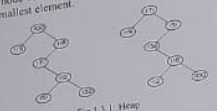


Fig 1.3.1: Heap

Heap Insertion:

procedure *INSERT*(A, n)
//inserts the value in $A(n)$ into the heap which is stored//
//at $A(1)$ to $A(n-1)$//
  integer i, j, n;
  $j \leftarrow n$; $i \leftarrow \lfloor n/2 \rfloor$; $item \leftarrow A(n)$
  while $i > 0$ **and** $A(i) < item$ **do**
    $A(j) \leftarrow A(i)$ //move the parent down//
    $j \leftarrow i$; $i \leftarrow \lfloor i/2 \rfloor$ //the parent of $A(i)$ is at $A(\lfloor i/2 \rfloor)$//
  repeat
  $A(j) \leftarrow item$ //a place for $A(n)$ is found//
end *INSERT*

Fig 1.3.2: Heap Insertion Algorithm

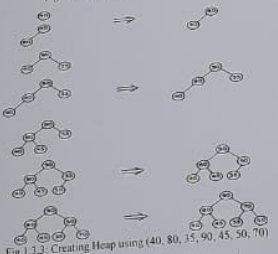
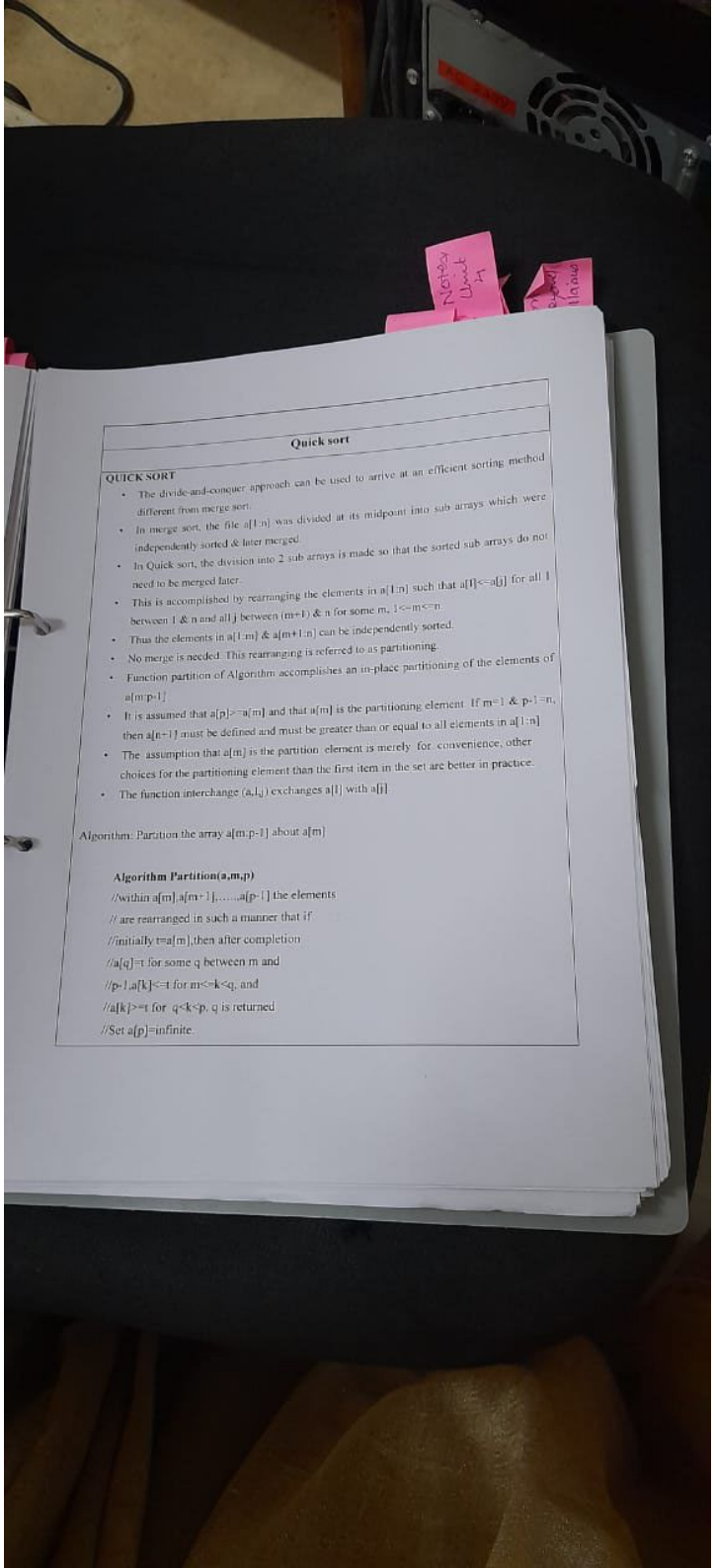


Fig 1.3.3: Creating Heap using (40, 80, 35, 90, 45, 50, 70)



Quick sort

QUICK SORT

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.
- In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.
- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.
- This is accomplished by rearranging the elements in a[1:n] such that a[i]<=a[j] for all i between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.
- Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.
- No merge is needed. This rearranging is referred to as partitioning.
- Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1].
- It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to all elements in a[1:n].
- The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in the set are better in practice.
- The function interchange (a,i,j) exchanges a[i] with a[j].

Algorithm: Partition the array a[m:p-1] about a[m]

```
Algorithm Partition(a,m,p)
//within a[m],a[m+1],.....,a[p-1] the elements
// are rearranged in such a manner that if
//initially t=a[m],then after completion
//a[q]=t for some q between m and
//p-1,a[k]<=t for m<=k<q, and
//a[k]>=t for  q<k<p. q is returned
//Set a[p]=infinite
```

$n=1 \rightarrow C_1 r_1 + C_2 r_2 = 5 \rightarrow (2)$

Eqn 1 $\rightarrow C_1 = -C_2$

sub C_1 value in Eqn (2)

$-C_2 r_1 + C_2 r_2 = 5$

$C_2(r_2 - r_1) = 5$

$$C_2 = \frac{5}{r_2 - r_1}$$

$$= \frac{5}{-1+4}$$

$= 5 / (-5) = -1$

$C_2 = -1$, $C_1 = 1$

Sub C_1, C_2, r_1 & r_2 value in equation $\rightarrow$ (A)

$f_n = 1.4^n + (-1).(-1)^n$

$\mathbf{f_n = 4^n + 1^n}$

2. Homogenous Recurrences :

* We begin our study of the technique of the characteristic equation with the resolution of homogenous linear recurrences with constant co-efficient, i.e the recurrences of the form.

$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$

where the t_i are the values we are looking for.

* The values of t_i on 'K' values of i (Usually $0 \le i \le k-1$ (or) $0 \le i \le k$) are needed to determine the sequence.
* The initial condition will be considered later.
* The equation typically has infinitely many solutions.
* The recurrence is,
  - → linear because it does not contain terms of the form t_{n-k}, t_{n-1}, t^2_{n-1} and soon.
  - → Homogeneous because the linear combination of the t_{n-i} is equal to zero.
  - → With constant co-efficient because the a_i are constants
* Consider for instance our non familiar recurrence for the Fibonacci sequence,

$f_n = f_{n-1} + f_{n-2}$

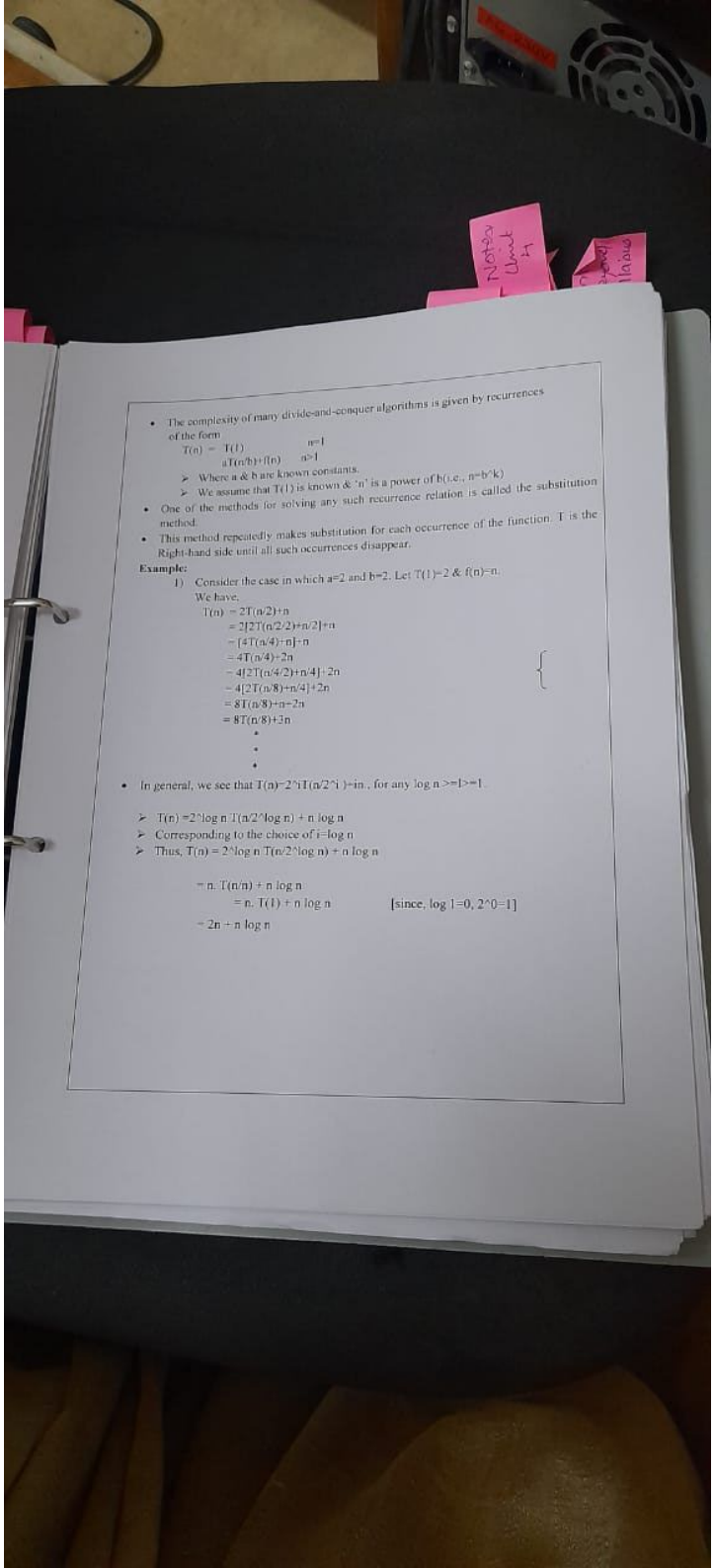
* This recurrence easily fits the mould of equation after obvious rewriting.

$f_n - f_{n-1} - f_{n-2} = 0$

* Therefore, the fibonacci sequence corresponds to a homogenous linear recurrence with constant co-efficient with $k=2, a_0=1 \& a_1=a_2=-1$.
* In other words, if f_n & g_n satisfy equation.

$$\text{So } \sum_{i=0}^{k} a_i f_{n-i} = 0 \text{ \& similarly for } g_n \text{ \& } f_n$$

Notes
Unit
4



- The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n=1 \\ aT(n/b)+f(n) & n>1 \end{cases}$$

  - Where a & b are known constants.
  - We assume that T(1) is known & 'n' is a power of b(i.e., $n=b^k$)
- One of the methods for solving any such recurrence relation is called the substitution method.
- This method repeatedly makes substitution for each occurrence of the function T is the Right-hand side until all such occurrences disappear.

Example:

1) Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n.

We have,

$T(n) = 2T(n/2)+n$
$= 2[2T(n/2/2)+n/2]+n$
$= [4T(n/4)+n]+n$
$= 4T(n/4)+2n$
$= 4[2T(n/4/2)+n/4]+2n$
$= 4[2T(n/8)+n/4]+2n$
$= 8T(n/8)+n+2n$
$= 8T(n/8)+3n$

.
.
.

- In general, we see that $T(n)=2^iT(n/2^i)+in$, for any $\log n >=i>=1$.
  - $T(n) = 2^{\log n} T(n/2^{\log n}) + n \log n$
  - Corresponding to the choice of $i=\log n$
  - Thus, $T(n) = 2^{\log n} T(n/2^{\log n}) + n \log n$

$= n. T(n/n) + n \log n$
$= n. T(1) + n \log n$ [since, $\log 1=0$, $2^0=1$]
$= 2n + n \log n$

Introduction to divide and conquer technique

DIVIDE AND CONQUER:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.
- For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.
- D And C(Algorithm) is initially invoked as D and C(P), where 'p' is the problem to be solved.
- Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting.
- If this so, the function 'S' is invoked.
- Otherwise, the problem P is divided into smaller sub problems.
- These sub problems P1, P2 ... Pk are solved by recursive application of D And C.
- Combine is a function that determines the solution to p using the solutions to the 'k' sub problems.
- If the size of 'p' is n and the sizes of the 'k' sub problems are n1, n2 ....nk, respectively, then the computing time of D And C is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n1)+T(n2)+\ldots\ldots\ldots\ldots+T(nk)+f(n); & \text{otherwise.} \end{cases}$$

Where T(n) → is the time for D And C on any I/p of size 'n'.
g(n) → is the time of compute the answer directly for small I/ps.
f(n) → is the time for dividing P & combining the solution to sub problems.

Algorithm D And C(P)

```
{
if small(P) then return S(P);
else
{
   divide P into smaller instances
                P1, P2... Pk, k>=1;
   Apply D And C to each of these sub problems;
   return combine (D And C(P1), D And C(P2),.....,D And C(Pk));
}
}
```