MCS-507 Design and Analysis of Algorithm

School of Computer Science & IT
Uttarakhand Open University, Haldwani

Table of Contents

BLOCK 1

UNIT 1: Introduction To Algorithms

S.No.	Topic	Page Number
1.1	Learning Objectives	2
1.2	Introduction TO COMPUTING	2
1.3	What is computer Science?	3
1.3.1	Formal and mathematical Properties of algorithms	3
1.3.2	Computer Hardware used by algorithms	3
1.3.3	Linguistic Realizations of algorithms	3
1.3.4	Applications of Algorithms	3
1.4	Important Terminologies	4
1.4.1	Algorithm	4
1.4.2	Algorithmic	4
1.4.3	Input Instance	4
1.4.4	Domain	4
1.4.5	Input Size	4
1.5	Simple Example	4
1.6	Characteristics of an Algorithm	5
1.7	Answer to Check Your Progress	6
2.1	Computational Problem	6
2.1.1	Structuring Problems	7
2.1.2	Search Problems	7
2.1.3	Construction Problems	7
2.1.4	Decision problems	7
2.1.5	Optimization Problems	7
2.2	Fundamental Stages of Problem Solving	10
2.2.1	Problem Understanding	10
2.2.2	Algorithm Planning	11
2.2.3	Design of algorithms	12
2.2.4	Algorithm Verification and Validation	12
2.2.5	Algorithm analysis	13
2.2.6	Algorithm Implementation	14
2.2.7	Perform post-mortem analysis	14
2.3	Summary	14
2.4	Answer to Check Your Progress	15
2.5	References	16

2.6	Model Questions	**16**
UNIT 2: Classification of Algorithm		
1.1	Learning Objectives	**17**
1.2	Classification of Algorithm	**18**
1.2.1	Classification by Implementation	**18**
1.2.2	Classification by Design	**18**
1.2.3	Classification by Problem Types	**19**
1.2.4	Classification by Tractability	**19**
1.3	Basics of Algorithm Writing	**19**
1.3.1	Sequencing	**20**
1.3.2	Decision or Conditional Branching	**20**
1.3.3	Recursion	**21**
1.4	Basics of recursion	**27**
1.4.1	Algorithm Sigma (N)	**28**
1.4.2	Algorithm iterative-sigma(N)	**29**
1.4.3	Algorithm towersofhanoi(A,B,C,n)	**30**
1.5	Answer to Check Your Progress	**31**
UNIT 3: Analysis of Algorithm		
1.1	Learning Objectives	**32**
1.2	Analysis of Algorithm	**32**
1.2.1	Subjective measures	**33**
1.2.2	Objective measures	**33**
1.3	Apriori analysis (or Mathematical analysis)	**33**
1.3.1	Step count	**34**
1.3.2	Operation count	**34**
1.3.3	Second Philosophy: Count of Basic Operations	**36**
1.4	Summary	**38**
1.5	Answer to Check Your Progress	**39**
1.6	References	**39**
1.7	Model Questions	**40**
UNIT 4: Analysis of Recursive Algorithm		
1.1	Learning Objectives	**41**
1.2	What Is Recursive Algorithm	**41**
1.2.1	What is Recurrence Equation?	**41**
1.3	Formulation of Recurrence Equation	**41**
1.4	Solution of Recurrence Equation	**44**
1.4.1	Methods to solve Recurrence Equation	**44**
1.5	Master Theorem	**47**
1.6	Summery	**48**

1.7	Answer to Check Your Progress	**49**
1.8	References	**49**
1.9	Model Questions	**49**
BLOCK 2 **UNIT 5: Brute Force Technique and Unintelligent Search**		
1.1	Learning Objectives	**51**
1.2	Brute Force Technique	**51**
1.2.1	The advantages of the brute force approach	**52**
1.3	Optimization Problem and Exhaustive Searching	**52**
1.4	Solution Space and Unintelligent Search techniques	**53**
1.4.1	DFS Search	**53**
1.4.2	BFS Search	**55**
1.4.3	15-Puzzle Problem	**56**
1.4.4	8-Queen Problem	**58**
1.4.5	Knapsack problem	**61**
1.4.6	Assignment Problem	**64**
1.5	Answer to Check Your Progress	**66**
2.1	Introduction to Divide and Conquer Technique	**66**
2.1.1	Advantages of Divide and Conquer Technique	**67**
2.1.2	Disadvantages of Divide and Conquer Technique	**67**
2.2	Marge Sort	**67**
2.3	Quicksort	**73**
2.4	Partitioning Algorithms	**74**
2.5	Hoare Algorithm	**77**
2.6	Answer to Check Your Progress	**82**
3.1	What is divide and Conquer design paradigm?	**82**
3.2	Multiplication of Long Integers	**83**
3.3	Strassen Matrix multiplication Algorithm	**89**
3.4	Strassen Matrix Multiplication	**91**
3.5	Summary	**98**
3.6	Answer to Check Your Progress	**99**
3.7	References	**99**
3.8	Model Questions	**99**
UNIT 6: Divide and Conquer Technique		
1.1	Learning Objective	**101**
1.2	Closest pair and Convex Hull Problems using Divide and Conquer	**101**
1.2.1	Closest Pair problem	**101**
1.2.2	Convex Hull	**107**
1.2.3	Quick Hull	**108**

1.2.4	Merge Hull	**111**
1.3	Answer to Check Your Progress	**114**
2.1	Applications of Divide and Conquer	**114**
1.2.1	Finding Maximum and Minimum Elements	**114**
2.2	Tiling Problem	**117**
2.2.1	Fourier Transform	**119**
2.2.2	Polynomial Multiplication	**128**
2.3	Answer to Check Your Progress	**130**
3.1	Introduction to Decrease and Conquer Design paradigm	**130**
3.2	Categories of Decrease and Conquer Design paradigm	**131**
3.2.1	Decrease by a constant	**131**
3.2.1.1	Insertion Sort	**132**
3.2.1.2	Topological Sorting	**135**
3.2.1.3	Permutations	**138**
3.2.1.4	Johnston–Trotter Algorithm	**140**
3.3	Summary	**144**
3.4	Answer to Check Your Progress	**144**
3.5	References	**144**
3.6	Model Questions	**145**
UNIT 7: Transform and Conquer Design paradigm		
1.1	Learning Objectives	**146**
1.2	Transform and Conquer Design paradigm	**146**
1.3	Variations of Transform and Conquer	**147**
1.4	Presorting	**148**
1.4.1	Finding unique elements in an array	**148**
1.4.2	Search using Presorting	**149**
1.4.3	Mode	**150**
1.5	Transform and conquer approach	**151**
1.6	Matrix Operations	**152**
1.6.1	Gaussian elimination method	**152**
1.7	Answer to Check Your Progress	**159**
2.1	Transform and Conquer Design paradigm	**160**
2.1.1	Variations of Transform and Conquer	**160**
2.2	Gaussian Elimination method for LU Decomposition	**161**
2.2.1	Recursive procedure	**164**
2.2.2	LUP decomposition	**166**
2.3	Crout’s Method of Decomposition	**166**
2.4	Finding Inverse of a matrix	**170**

2.5	Finding Determinant of a matrix	**173**
2.6	Answer to Check Your Progress	**175**
3.1	More Transform and Conquer Design problems	**175**
3.1.1	Variations of Transform and Conquer	**176**
3.2	Polynomial Evaluation	**176**
3.3	Faster Exponentiation	**180**
3.4	Right – to – Left Computation	**182**
3.5	Summary	**185**
3.6	Answer to Check Your Progress	**186**
3.7	References	**186**
3.8	Model Questions	**186**
UNIT 8: Greedy Algorithm		
1.1	Learning Objectives	**187**
1.2	Greedy Algorithm	**187**
1.3	Applications of Greedy Algorithm	**189**
1.4	Answers to check your progress	**194**
2.1	Coin Change Problem	**194**
2.1.1	Coin Change Problem using Dynamic Programming Approach	**195**
2.1.2	Failure of coin change problem Check Your Progress	**195**
2.2	Answers to check your progress	**197**
2.3	Scheduling problem	**197**
2.3.1	Types of Scheduling problem	**198**
2.3.2	Scheduling problem without deadline	**198**
2.4	Answers to check your progress	**201**
2.5	References	**202**
BLOCK 3 **UNIT 9: Knapsack problem**		
1.1	Learning Objectives	**203**
1.1	Knapsack problem	**203**
1.1.1	0 – 1 Knapsack problem	**203**
1.1.2	Fractional Knapsack Problem	**203**
1.2	Huffman Algorithm	**207**
1.2.1	Answers to check your Progress	**212**

1.3	Minimum Spanning Tree	**215**
1.3.1	Kruskal Algorithm	**217**
1.3.2	Prim's Algorithm	**217**
1.3.3	Answers to check your Progress	**220**
1.4	Optimal merge Short	**220**
1.5	Single-source Shortest Path Problems	**225**
1.5.1	Answers to check your Progress	**228**
1.5.2	Model Questions	**228**
UNIT 10: Dynamic Programming		
1.1	Learning objectives	**230**
1.2	Dynamic Programming	**230**
1.2.1	Answers to check your Progress	**231**
1.3	Binomial Coefficient	**234**
1.3.1	Answers to check your Progress	**237**
1.4	Transitive Closure	**237**
1.4.1	Answers to check your Progress	**241**
1.5	Shortest path algorithm	**241**
1.5.1	Answers to check your Progress	**245**
1.6	The Multistage Graph Problem	**245**
1.7	Traveling Salesman Problem	**249**
1.7.1	Answers to check your Progress	**249**
1.8	Chained matrix multiplication	**252**
1.9	Bellman –ford algorithm	**259**
1.9.1	Answers to check your Progress	**265**
1.9.2	Model Questions	**265**
Unit 11: Longest Common Subsequence		
1.0	Learning objectives	**267**

1.1	Longest Common Subsequence	**267**
1.1.1	Answers to check your Progress	**272**
1.2	String Edit Problem	**272**
1.2.1	Answers to check your Progress	**276**
1.3	Binary Search Tree	**276**
1.4	Optimal Binary Search Tree	**278**
1.5	Knapsack Problem	**288**
1.6	Flow Shop Scheduling	**292**
1.7	Concept of computational complexity	**296**
1.8	Model Questions	**301**
BLOCK 4 **UNIT 12: Solvability of Problems**		
1.0	Learning Objective	**302**
1.1	Solvability of Problems	**302**
1.1.1	Polynomial Time Algorithms	**302**
1.1.2	Unsolvable problems	**303**
1.1.3	Hard Problems	**303**
1.1.4	Turing machine	**304**
1.1.5	NP Problems	**305**
1.1.6	Answers to check your progress	**307**
1.2	Class P problem	**307**
1.2.1	What is NP Class?	**307**
1.2.2	Co – NP Problem	**308**
1.2.3	NP – I Problem	**308**
1.2.4	What is NP Hard Problem?	**308**
1.2.5	NP complete problem?	**309**
1.2.6	Answers to check your progress	**313**
1.3	Class P and NP class Problem	**314**
1.3.1	Reduction	**314**
1.3.2	Turing Reduction	**314**
1.3.3	Karp Reduction	**315**
1.3.4	NP Complete Proof Out- line	**317**
1.4	Answers to check your progress	**322**
1.5	Model Questions	**322**
UNIT 13: NP – Complete Problem		
1.0	Learning Objective	**323**

1.1	NP – Complete Problem	**323**
1.1.1	Reduction	**323**
1.1.2	Travelling Salesmen Problem	**324**
1.1.3	Clique Problem	**324**
1.1.4	Vertex Cover	**326**
1.1.5	Answers to Check your Progress	**329**
1.2	NP – Complete Problem	**329**
1.2.1	Backtracking Technique	**329**
1.2.2	Sum of sub – set	**330**
1.2.3	Branch and bound Technique	**331**
1.2.4	Assignment Problem	**332**
1.2.5	Answers to Check your Progress	**336**
1.3	Randomized Algorithm	**336**
1.3.1	Concept of witness	**338**
1.3.2	Randomized sampling and ordering	**338**
1.3.3	Foiling Adversary	**338**
1.3.4	Types of Randomized Algorithms	**338**
1.3.5	Complexity Class	**339**
1.3.6	Random Number	**339**
1.3.7	Hiring Problem	**340**
1.4	Answers to check your Progress	**342**
1.5	Model Questions	**343**
UNIT 14: Randomized Algorithm		
1.0	Learning Objective	**344**
1.1	Randomized Algorithm	**344**
1.1.1	Randomized Analysis	**345**
1.1.2	Primality Testing	**347**
1.1.3	Randomized large string Comparison	**347**
1.1.4	Randomized quick sort	**348**
1.1.5	Answers to check your progress	**351**
1.2	Approximation algorithm	**352**
1.2.1	Vertex Cover Problem	**352**
1.2.2	Travelling Salesmen Problems	**353**
1.3	Heuristic	**359**
1.3.1	Greedy Approach	**359**
1.3.2	Linear Programming	**360**
1.3.3	Dynamic Programming	**361**

1.4	Answers to check your progress	**364**
1.5	Model Questions	**364**

BLOCK 1
UNIT 1
INTRODUCTION TO ALGORITHMS

1.1 Learning Objectives
1.2 Introduction TO COMPUTING
1.3 What is computer Science?
- 1.3.1 Formal and mathematical Properties of algorithms
- 1.3.2 Computer Hardware used by algorithms
- 1.3.3 Linguistic Realizations of algorithms
- 1.3.4 Applications of Algorithms

1.4 Important Terminologies
- 1.4.1 Algorithm
- 1.4.2 Algorithmic
- 1.4.3 Input Instance
- 1.4.4 Domain
- 1.4.5 Input Size

1.5 Simple Example
1.6 Characteristics of an Algorithm
1.7 Answer to Check Your Progress
2.1 Computational Problem
- 2.1.1 Structuring Problems
- 2.1.2 Search Problems
- 2.1.3 Construction Problems
- 2.1.4 Decision problems
- 2.1.5 Optimization Problems

2.2 Fundamental Stages of Problem Solving
- 2.2.1 Problem Understanding
- 2.2.2 Algorithm Planning
- 2.2.3 Design of algorithms

2.2.4 Algorithm Verification and Validation

2.2.5 Algorithm analysis

2.2.6 Algorithm Implementation

2.2.7 Perform post-mortem analysis

2.3 Summary
2.4 Answer to Check Your Progress
2.5 References
2.6 Model Questions

1.1 Learning objectives

This unit focuses on the basics of the algorithm study. The Learning objectives of this unit are as follows:

- Need, role and characteristics of Algorithms
- To understand stages of Problem solving process
- To understand Classification of Algorithms and recursive algorithms
- To understand Computational Problem and stages of Problem solving process
- Characteristics of an Efficient Algorithm
- To categorize algorithms.

1.2 Introduction to computing

Computers have become integral part of our life. It is difficult to imagine a life without computers. Individuals use computers for variety of reasons such as document preparation, Internet browsing, sending emails, video games, performing numeric calculations, and this list goes on.

Industries and governments, on the other hand, use computers much more productivity such as airline reservation, video surveillance, biometric recognition, e-governance and e-commerce. Industries use computers for process control and quality testing. These applications improve the efficiency and productivity.

The popularity and necessity of computer systems have prompted schools and universities to introduce computer science as an integral part of our modern education.

The usage of computers has brought importance for three types of thinking.

- **Computational Thinking:** Computational thinking is the knowledge of using computers to perform our day to day activities. Computational thinking is the need of the hour and a necessary to survive in this world.
- **Analytical Thinking**: Computer science professionals are expected to do know the usage and also are required to provide computer based solutions for problems by writing computer programs for it. Writing programs requires analytical skills.
- **Algorithmic thinking** is a necessary analytical skill that is required for solving problems and writing effective programs. Algorithmic thinking is generic and is not restricted to computer science domain.

1.3 What is computer Science?

There are many definitions of computer science. An apt definition of computer science is provided by two persons, Norman E Gibbs and Allen B Tucker[1]. It is proposed as part of the ACM curriculum forliberal art. The proposed definition is given as follows:

"Computer science is the study of algorithms or more precisely its formal (or mathematical) properties, hardware and linguistic realizations along with its applications".

So, one can safely conclude that computer science is the study of algorithms and its manifestations. There are four types of manifestations given by their definition. They are listed as below:

1.3.1 Formal and mathematical properties of algorithms
This includes the study of algorithm correctness, algorithm design and algorithm analysis for understanding the behaviour of algorithms.

1.3.2 Computer Hardware used by Algorithms:
These are Hardware realizations that includes the computerhardware which is necessary to run the algorithms in the form of programs.

1.3.3 Linguistic Realizations:
These include the study of programming languages and its design, translatorslike interpreters, compilers, linkers and loaders.

1.3.4 Applications of Algorithms
This includes the study of tools and packages.

1.4 Important Terminologies

Let us introduces some of the important terminologies.

1.4.1 Algorithm: An **algorithm** is a set of unambiguous instructions or procedures for solving a given problem to provide correct and expected outputs for all valid and legal input data. Some of the other word words that are used for algorithms in literature are **recipe, prescription, process or computationalprocedure.**

1.4.2 Algorithmic: The study of designing, implementing, analyzing algorithms is called **Algorithmic**.

1.4.3 Input Instance: A valid input from legal set of input data for the algorithm is called an **instance** of an algorithm. A valid input can be called as an **instance** of a problem. For example, factorial of a negative number is not possible. So, all valid positive integers {0,1,2,…} can serves input and every legal input is called an instance.

1.4.4 Domain: All possible inputs of a problem are often called **domain** of the input data. The input should be encoded in a suitable form so that the computers can process.

1.4.5 Input Size: The number of binary bits used to represent the given input, say N, is called **input size**.

The word Algorithm is derived from the name of a Persian mathematician, Abu Ja'fer Mohammed Ibn Musa al Khowarizmi, who lived sometime around 780 – 850 AD. He wrote a book titled "Kitab al Jabr w'al Muqabala" (or Rules of Restoration and Reduction) where he introduced the old Indian-Arabic number systems to Europe. His name was quoted as Algorismus in Latin books and Algorithm is emerged as its corrupted form.

1.5 Simple Example

To illustrate the process of writing an elementary algorithm, let us try to write a procedure for preparing a tea. The input is tea powder, milk etc. The environment of a typical algorithm involves an agent, input, process and output. Here agent is the performer. **Agent** can be a human or a computer. The agent for preparing the tea is human and output of the procedure is tea. The procedure can be written as a stepby step procedure as follows:

1. Pour tea powder in a cup

2. Boil the water and pour it into the cup and filter it

3. Pour milk

5. Put sugar if necessary

6. Pour it into a cup.

This kind of procedure can be called an **algorithm**.

Likewise, algorithms can be written for all tasks. The algorithms, thus developed, for all the tasks share some commonalities. The commonalities are called characteristics of an algorithm.

1.6 Characteristics Of An Algorithm

The following characteristics of an algorithm are important. They are listed as below:

- **Input:** An algorithm can have zero or more inputs.
- **Output:** An algorithm should produce at least one or more outputs.
- **Definiteness:** By definiteness, it is meant that the instructions should be clear and unambiguouswithout any confusion. For example, division by zero is not a well-defined instruction.
- **Uniqueness:** The order of the instructions of an algorithm should be well defined as a change in theorder of execution leads to wrong result or uncertainty.
- **Correctness:** The algorithm should be correct and yield expected results.
- **Effectiveness:** The algorithm should be traceable.
- **Finiteness:** An algorithm should have finite number of steps and should terminate.

Additionally, the algorithm should be simple (i.e., Ease of implementation, generality (An algorithm should be generic and not specific).

What is a program? A computer program is an algorithm in action. Algorithm thus acts as a blueprints that are used for constructing a house. In short, Program is an expression of that idea in a programming language.

Check Your Progress

Choose the Correct One.

Q.1: Fill the blank with correct word.

"Computer science is the study of ___________or more precisely its formal (or mathematical) properties, hardware and linguistic realizations along with its applications".

A. Algorithm

B. Program

C. Computer

D. Function

Q.2: Fill the blank with correct word.

"All possible inputs of a problem are often called ________of the input data."

A. Domain

B. Program

C. Function

D. Range

Q.3: What are the characteristics of algorithm?

A. Uniqueness

B. Correctness

C. Effectiveness

D. Finiteness

E. All of the above

1.7 Answer to check your progress

Ans to Q.1: A

Ans to Q.2: A

Ans to Q.3: E

2.1 Computational Problem

Algorithms are feasible for computational problems. A **computational problem** is characterized by two factors –

- The formalization of all legal inputs and expected outputs of a given problem
- Thecharacterization of the relationship between problem output and input.

Non-computational problems are problems that cannot be solved by the computer system. For example, emotions and opinions.

For example, it is not feasible to write a program for, say, offering opinion on Indian Literature.

Only computational problems can be solved by computers. One encounters many types of computational problems in the domain of computer science. Some of the problems are listed below:

2.1.1 Structuring Problems: In structuring problems, the input is restructured based on certain conditions or properties. For example, Sorting is an example of a structuring problem.

2.1.2 Search Problems: Search problem involves searching for a target in a list of all possibilities. Puzzles are good examples of search problems where the target or goal is searched in a huge list of feasible solutions.

2.1.3 Construction Problems: These are the problems that involve the construction of solutions based on the constraints that are associated with the given problem.

2.1.4 Decision problems: Decision problems are Yes/No type of problems where the algorithm output is restricted to answering Yes or No.

2.1.5 Optimization Problems: Optimization problems are very important set of problems that are often encountered in computer science domain. The problems like finding shortest path or least cost solutions are optimization problems. These problems have objective function and a set of constraints. The solution is finding a solution that maximizes or minimizes the objective function while satisfying the constraints.

Let us discuss about the ways of writing a simple algorithm. Let us assume that there is a class (say, a tuition centre) where the following set of students is studying. The course marks are given as shown in Table 1. The students are expected to get 50 marks to pass. Given this table, find how many failures are there?

Table 1 Students Course Mark

Register Number	Student Name	Course Marks
1	Raghav	80
2	Preeti	30
3.	John	83

4.	Jones	23
5.	Joseph	90

This can be done manually by checking each mark with 50. If the mark is less that 50, then the fail count can be incremented. Finally, when the list is over, the fail count is printed.

The informal algorithm is given as follows:

1. Let counter = 1, Number of Students = 5

2. Fail count = 0

2. While (counter <= Number of students)

   2.1 Read the student marks of the students

   2.2 Compare the marks of the student with 50

   2.3 If Student marks is less than 50
   Then increment the fail count

   2.4 Increment the counter

3. Print fail count

4. Exit

The above algorithm is a simple algorithm. But in reality, efficient algorithms are preferred. What is an efficient algorithm? An efficient algorithm is one that uses computer resources optimally. Why? Computer resources are limited.

Let us take a simple example of traveling salesperson problem (TSP). Informally, travelling salesperson is one starts in a city visits all other cities only once and returns back to the original city where he had started. This problem can be modeled as a graph.. A graph is a set of nodes and edges that interconnect the nodes. In this case, the cities are the nodes and edges are the path that connects the cities. Edges are undirected in this case. Alternatively, it is possible to move from a particular city to any other city or vice versa.

A brute force technique can be used for solving this problem.

1. TSP problem involving only one city, there is no path!

2. For two cities, there is only one path (A-B).

3. For three cities, A, B, and C, the two paths are A – B – C and A – C – B, assuming A is theorigin from where the traveling salesperson started.
4. For four cities, A,B, C and D, the paths are { A –D- B-C-A, A-D-C-B-A, A-B-C-D-A, A-B-D-C-A, A-C-D-B-A and A-C-D-B-A}.
This is shown in Table 2.

Table 2: Complexity of TSP

Number of Cities	Number of routes
1	0 (As there is no route)
2	1
3	2
4	6

Thus it can be noted that every addition of a city increases the path exponentially. Table 2 showsthe number of possible routes. It can be seen that for N cities, the number of paths are

(N-1)! Thus, for 51 cities, the paths are (51-1)! = 50!

50! is 30414093201713378043612608166064768844377641568960512000000000000.

Thus, one can imagine that for larger values of N (Say all the cities of India or China), the number of paths are very large and even if a computer takes a second for exploring a route, the algorithm would run for many months. This shows that efficiency of algorithms is very important and thus designing such algorithms are very important.

2.2 FUNDAMENTAL STAGES OF PROBLEM SOLVING

Problem solving is both an art and science. As there are no guidelines available to solve the problem, the problem solving is called an art as high creativity is needed for solving problems effectively. Thus by art, we mean one has to be creative, novel and adventurous and by science, we mean the problem solving should be based on sound mathematical and logical guidelines.

The problem solving stages are given as follows:

2.2.1 Problem Understanding
2.2.2 Algorithm Planning
2.2.3 Design of algorithms
2.2.4 Algorithm Verification and Validation
2.2.5 Algorithm analysis
2.2.6 Algorithm Implementation
2.2.7 Perform post-mortem analysis

2.2.1 Problem Understanding

Is it possible to solve the given problem? This falls under the domain called computability theory. Computability theory deals with the solvability of the given problem.

To deal with the solvability, one require analytical thinking. Analytical thinking deals with the ways of solving the given problems. It is not possible to solve the problem if the problem is ill- defined. Often puzzles depict the limitations of computing power. Sometimes, there may be some solution but one doesn"t have the knowledge or means to analyze the algorithms. Thus problems can be as follows:

- **Computationally hard problems:** It is not possible to solve this problem.
- **Analytically hard problems**: These problems, like Traveling salesperson problem, runs effectively for small instances. But for larger problems, it may be computationally feasible. Also for some problems analysis is difficult.

In computability theory, these sort of problems are often considered. The problem solving starts with understanding of the problem and aims at providing a detailed problem statement. There should not be any confusion in the problem statement. As the mistakes in understanding the problem may affect the development of algorithms.

2.2.2 Algorithm Planning

Once problem statement is produced, the planning of algorithm starts. This phase requires selection of computing model and data structures.

Computational model is the selection of a computing device. A computational model is a mathematical model. Why? It is meaningless to talk about fastness of the algorithm with respect to a particular machine or environment. An algorithm may run faster in machine A compared to machine B. Hence, analysis of algorithms based on a particular brand of machines is meaningless. Hence, the algorithm analysis should be should be independent of machines.

Normally, two theoretical machines are used – One is called Random Access Machine (RAM) and another is called Turing machine.

- **Data Structures**

Second major decision in algorithm planning is selection of a data structure. Data structure is a domain that deals with data storage along with its relationships. Data structures can have impact on the efficiency of the algorithms. Therefore, algorithms and data structures together often constitute important aspect of problem solving.

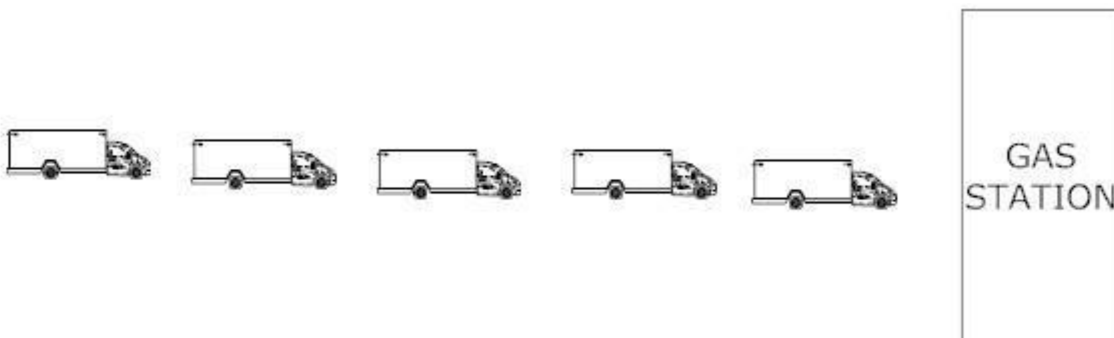


Fig. 1 : Example of a Queue

Data organization is something we are familiar with in our daily life. The Fig 1 shows a data organization called „Queue". A Gas station with one servicing point should have a queue like

above to avoid chaos. A queue (or FCFS – First Come First Serve) is an organization where the processing (filling of gas) is done in one end and addition of a vehicle is done in another end. A popular statement in computer science is “Algorithm + Data structure = Program” as a wrong selection of a data structure often proves fatal in problem solving.

The planning of a data structure can be based on the solution also as some of the problems can be solved exactly. Sometimes, getting the exact solution is impossible for computationally hard problems. Therefore, for such problems approximate solutions are planned.

2.2.3 Design of Algorithms

Algorithm design is the next stage of problem solving. Algorithm design is a way of developing the algorithmic solutions using the appropriate design strategy.

Different types of problem demands different strategies for finding the solution. Just imagine, searching a name in a telephone directory. If one has to search for a name “Qadir”, then the search can be done from starting page to the end page. This is a crude way of solving the problem. Instead, one can prefer to search using indexed entries. It can be done using interpolation search also. Thus design strategy selection plays an important role in solving problems effectively.

After the algorithm is designed, it should be communicated to a programmer so that the algorithms can be coded as a program. This stage is called **algorithm specification.** The choices of communication can be natural language, programming language and pseudocode. Natural language is preferable, but has some disadvantages such as lack of precision and ambiguity. Programming language may create a dependency on that particular language. Hence, often algorithm is written and communicated through pseudocode. Pseudocode is a mix of natural language and mathematics.

2.2.4 Algorithm Verification and validation

Algorithm verification and validation is the process of checking algorithm correctness. An algorithm is expected to give correct output for all valid inputs. This process is called **algorithm validation**. Once validation is over, **program proving** or **program verification** starts.

Verification is done by giving mathematical proofs. Mathematical proofs are rigorous and better than scientific methods. Program correctness itself a major study by itself. Proofs are given as follows:

- **Proof by assertion:** Assertion assert some facts. It can be given throughout the program and assertions expressed are in predicate calculus. If it can be proved that, if for every legal input, ifit leads to a logical output, then it can be said that the given algorithm is correct.

- **Mathematical Induction:** Mathematical induction is a technique can be used to prove recursive algorithms.

2.2.5 Algorithm Analysis

Once the algorithm is proved correct, then the analysis starts. Analysis is important for two reasons. They are as follows:

- **To decide efficiency of algorithms**
- **To compare algorithms for deciding the effective solutions for a given problem.**

Algorithm analysis as a domain is called Algorithmic Complexity theory. What is a complexity? Well, complexity is assumed to be with respect to size of the input. Sorting of an array with 10 numbers is easy: but the same problem with 1 million is difficult. So there is a connection with the complexity and the size of the input.

The complexity of two or more algorithms can be done based on measures that can be categorized into two types.

- **Subjective measures.**
- **Objective measures.**

Subjective measures include factors like ease of implementation or style of the algorithm or understandability of algorithms. But the problem with the subjective measures are that they vary from person to person. Hence Objective measures are preferable.

Objective measures include factors like time complexity, space complexity and measures like disk usage and network usage. By and large, algorithmic analysis is the estimation of computing resources.

Out of all objective measures, two measures are popular. They are time and space.

- **Time complexity:** Time complexity refers to the time taken by the algorithm to run for scaled input. By time, it is often meant run time. Actually, there are two time factors. One is execution time and another is run time. Execution (or compile time) does not depend on problem instances and compilers often ignore instances. Hence, time always refers to run or execution time only.

- **Space Complexity**: Space complexity is meant the memory required to store the program.

2.2.6 Implementation of algorithm as a program and performance analysis

After the algorithms are designed, it is implemented as a program. This stage requires the selection of a programming language. After the program is written, then the program must be tested. Sometimes, the program may not give expected results due to errors. The process of removing the errors is called **debugging**. Once program is available, they can be verified with a bench mark dataset. It is called experimental algorithmics. This is called performance analysis. Thus, P**rofiling is a** process of running a program on a datasets and measuring the time/space requirement of the program.

2.2.7 Perform post-mortem analysis

Problem solving process ends with postmortem analysis. Any analysis should end with the valuable insight. Insights like Is the problem solvable? Are there any limits of this algorithm? and Is the algorithm efficient? are asked and collected.

A theoretical best solution for the given problem is called lower bound of the algorithm. The worst case estimate of behavior of the algorithm is called upper bound. The difference between upper and lower bound is called **algorithmic gap**. Technically, no algorithmic gaps should exist.But practically, there may be vast gap present between two bounds.

Refining the algorithms is to bring these gaps short by reefing planning, design, implementation and analysis. Thus problem solving is not linear but rather this is a cycle.

2.3 Summary

In short, one can conclude as part of this unit 1 that

1- Computer Science's core is algorithm study and Algorithmic thinking is necessary to solve problems.
2- Algorithms are the blueprint of how to solve problems and Efficiency is a necessity for algorithms.
3- Problem solving stages are problem understanding, planning, design, analysis, implementationand post-analysis.
4- Problem understanding and planning is important. Algorithm design and analysis is crucial.

Check Your Progress

Choose the Correct One.

Q.1: There are ___________ types of problems.

A. 2
B. 3
C. 4
D. 5

Q.2: The problems like finding shortest path or least cost solutions are __________ problems.

A. Optimization Problems
B. Search Problems
C. Construction Problems
D. Decision problems

Q.3: Algorithm correctness is checking with_________.

A. Verification
B. Validation
C. Both A & B
D. None of the above

Q.4: First stage of problem solving is:

A. Problem Understanding
B. Algorithm Planning
C. Design of algorithms
D. Algorithm Verification and Validation

2.4 Answer to Check Your Progress

Ans to Q.1: A

Ans to Q.2: A

Ans to Q.3: C

Ans to Q.4: A

2.5 References

1. S.Sridhar – Design and Analysis of Algorithms, Oxford University Press, 2014.
2. Cormen, T.H., C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press,Cambridge, MA 1992.
3. Gibbs, N.E., Tucker, A.B. "A model Curriculum for a Liberal Arts Degree in ComputerScience", Communications of ACM, Vol. 29, No. 3 (1986).
4. *S.Sridhar – Design and Analysis of Algorithms , Oxford University Press, 2014.*
5. Cormen, T.H., C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press,Cambridge, MA 1992.
6. *S.Sridhar – Design and Analysis of Algorithms, Oxford University Press, 2014.*
7. Cormen, T.H., C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge,MA 1992.

2.6 Model Questions

1. What is Computer Science?
2. What is Algorithms?
3. What are the Characteristics of Algorithm?
4. What is computational and non-computational problems?
5. What are the different stages of problem solving?
6. What do you understand by algorithm verification and validation?
7. What do you understand by traveling salesperson problem (TSP)?
8. Explain recursion with example.

BLOCK 1
UNIT 2: Classification of Algorithm

1.1 Learning Objectives
1.2 Classification of Algorithm
    1.2.1 Classification by Implementation
    1.2.2 Classification by Design

1.2.3 Classification by Problem Types

1.2.4 Classification by Tractability

1.3 Basics of Algorithm Writing

1.3.1 Sequencing

1.3.2 Decision or Conditional Branching

1.3.3 Recursion

1.4 Basics of recursion

1.4.1 Algorithm Sigma (N)

1.4.2 Algorithm iterative-sigma(N)

1.4.3 Algorithm towersofhanoi(A,B,C,n)

2.1 Analysis of Algorithm

2.1.1 Subjective measures

2.1.2 Objective measures

2.2 Apriori analysis (or Mathematical analysis)

2.2.1 Step count

2.2.2 Operation count

2.2.3 Second Philosophy: Count of Basic Operations

2.3 Summary

2.4 Answer to Check Your Progress

2.5 References

2.6 Model Questions

1.1 Learning Objectives

- To understand the process of algorithm analysis
- To know the types of analysis
- To understand the concept of step count

1.2 Classification of Algorithms

For effective, often algorithms are categorized. The algorithm categorization can be done basedon various criteria such as

1.2.1 Implementation
1.2.2 Design
1.2.3 Problem Types
1.2.4 Tractability

1.2.1 Classification by Implementation

Based on implementation, one can classify the algorithms as recursive algorithm and iterative algorithms.

Recursive algorithm is an algorithm that uses functions that call itself conditionally. Recursion isa top-down approach of solving the given problem by reducing the problem to another problem with a unit decrease in input instance. The transformed problem is same as the original problem but with a difference that its input is one less than the original problem. This strategy uses the principles of **work postponement** or **delaying the work**. Iterative algorithms on the other hand are deductive.

Based on implementation, the algorithms can be classified as sequential or parallel algorithms. An algorithm that uses only one processor is called sequential algorithm. On the other hand, a parallel algorithm uses a set of processors to find a solution of the problem. Similarly, the algorithm can be classified as exact or approximation based on the solution it provides for the given problem. Again, based on implementation, the algorithms can be categorized as deterministic and randomized. Deterministic algorithms have predictable results for a given input. If this is not possible, then the algorithm is called non-deterministic.

1.2.2 Classification by Design

Based on design technique, the algorithms can be classified. Some of the popular algorithm design categories brute force method, Divide and Conquer, Dynamic programming, greedy approach, and backtracking.

1.2.3 Classification by Problem Types

Based on problem types or domain, the algorithm can be classified as follows:

1. Sorting Algorithms
2. Searching Algorithms
3. String Algorithms
4. Graph Algorithms
5. Combinatorial Algorithms
6. Geometric Algorithms

1.2.4 Classification based on Tractability

Based on tractability, the algorithms can be categorized as polynomial algorithms and Non-deterministic polynomial problems. Easy solvable problems are called polynomial algorithms and the problems for which solution is not possible or difficult to solve given the limited natureof resources are called NP algorithms.

1.3 Basics of Algorithm Writing

Algorithms as discussed earlier are step by step procedure for solving a given problem. As said earlier, algorithms can be written using natural language or pseudocode. There is no standard way of writing algorithms in pseudocode. So there is a need for basic guidelines for writing algorithms. The problem solving starts with stepwise refinement. The idea of stepwise refinement is to take a problem and try to divide it into many subproblems. The subproblems can further be divided more subproblems. The subdivision will be carried out till the problem can’t further be divided. Hence, the idea of stepwise refinement is to evolve structures that can be directly implemented in a programming language.

The kinds of structures thus evolved are sequence, decision and Iteration. These are called

control structures and are described below:

1.3.1 Sequence

Sequence is a structure whereby the computer system executes tasks one by one. This is given as follows:

Ta

sk

P

Ta

sk

Q

Ta

sk

R

Here, the task P is executed first, followed by tasks Q and R.

1.3.2 Decision or Conditional Branching

This is a control structure where the condition is evaluated first and based on its condition the course of action is decided. The control structure for decision is given as follows:

IF (Condition C) Then

Perform Task A

Else

Perform Task B

It can be observed that the condition C is evaluated first. Then based on the results of the

condition, taskP is performed if the condition is true or task Q is performed if the condition is false.

1.3.3 Repetition

Computers are known for their effectiveness in solving repetitive tasks. Repetition control structure is given as follows:

While (condition C) do

R

For example, informally we say often in English language “Perform the task 100 times”. This is a kind of repetition.

There are two types of iteration. Iteration like saying “Perform task A exactly 500 times” is called a bounded iteration. Programming languages do provide a ‘For – Statement” that implements a bounded iteration. On the other hand, statements like performing a task for a specific condition are called unbound iteration. Good examples of unbounded iteration are statements like “While…End while” and “repeat …. until”.

Once the control structures are evolved, then it has to be written as an algorithm. There are only three ways of writing algorithms:

- **Using a natural language like English.**
- **Using a programming languages, say C++ or Java.**

- **Pseudocode: Using a natural language like English.**

English, or any natural language, is obvious choice for writing algorithms. But the problem is most of the natural languages are ambiguous as a wrong interpretation of a word may lead to imprecise implementation. Hence, natural languages are not considered suitable for algorithm writing. Similarly, the usage of a programming language makes algorithm dependent on a particular language. Hence, pseudocode is preferable for writing algorithms. One can recollect from module 1 that pseudocode is a mix of natural language like English and mathematical constructs.

Let us evolve a pseudocode conventions so that the algorithm can be written. The pseudocode conventions of the algorithms are specified below:

- **Assignment Statement**

Assignment statement is a statement for assigning a value or expression to a variable. For example, the following assignment statements are valid.

$x = 20$

z = r + k

- **Input/output Statements**

Input statement is used by the user to give values to the variables of the algorithm. For example, the following statement is right.

Input x, y

Similarly, the print or write statement is used to print the values of the variables. For example, the following statement is used to print the value of the variable x and y.

Print x,y or Write x,y

- **Conditional Statements**

Algorithm can have conditional statement. The syntax of the conditional statement can be shown as:

If (condition)

then

Statement

(s)

End if

Here, the condition (True or false type) is evaluated first. If the condition is true, then statement(s) are executed. “If- Endif” serves as brackets for the conditional statement.

Conditional statement can have else part also as shown below:

If (condition)

then

Statement A

;

else

Statement B

End if

Here, If the condition is true, then statement A (This can be a set of statements also) are executed. Otherwise, if the condition is false, then statement B (This also can be a single or multiple statements) is executed.

- **Repetition Statement**

Algorithms can have repetitive statements. Three repetitive statements are supported by most of theprogramming languages.

Unconditional repetitive statement is 'For' statement. This is an example of bounded iteration. Thesyntax of this statement is given as follows

For variable = value1 to

value2 doStatement(s)

End for

Computer system executes this statement like this: First the variable is to value1. Value1 and value2 can be a value or an expression. Then the statement(s) is executed till the variable values reaches value2.

- **Conditional Loop:**

One useful repetitive statement is “While” statement that provides a conditional loop. The syntax of the statement is given below:

```
While (Condition) do
    begin
    Statement(s);
End while.
```

Repeat…Until also provides repetition control structure. The difference between this statement and While statement is that “repeat – until” statement first executes the task and then checks condition of the statement. The syntax of repeat statement is given below:

```
    repeat
      Statemen
      t(s)
    until (condition)
```

Using these statements, some elementary algorithms can be designed.

let us practice some elementary algorithms and let us consider the problem of converting Fahrenheit to Celsius. The problem can be directly solved using a simple formula. The formula for conversion is given as

$$celsius = \left(\frac{5}{9}\right) * (Farenheit - 32)$$

The algorithm can be written informally as follows:

1. Input Fahrenheit temperature
2. Apply the formula for temperature conversion
3. Display the results.

The algorithm can be given formally as follows:
Algorithm
FtoC(F) Input
: Fahrenheit F
Output:
Celsius Begin

Celsius =
(5/9)*(F-32)
Return Celsius;

End.

Let us practice one more algorithm for finding the count and sum of even/odd numbers of an array. Thealgorithm can be given informally as follows;

1. Initialize oddcount, evencount, oddsum and evensum

2. Read the sumber

3. If it is odd or even, then increment the appropriate counters

4. Display the results

The algorithm is formally given as follows:

Algorithm sumoddeven

(A[1..n])Input: An array

A[1..n]

Output: sum on odd and even number count

oddcount = 0

evencount = 0

oddsum = 0

evensum = 0

for i = 1 to n

```
reminder = A[i]
mod 2if (reminder
= 0) then

        evensum = evensum + A[i]
        evencount = evencount + 1

    else
        oddsum = oddsum + A[i]
        oddcount = oddcount + 1

    Endif
  End for

Return evensum, evencount, oddsum,
oddcountEnd
```

Another popular algorithm is linear search. Here, a target is given and the objective of the linear search is to display whether the target is present in the given array, if so where? And failure message is the target is not present in the array.

The informal algorithm is given as follows:

1. Read the value of the target and array A
2. Index = 1, found = false
3. Repeat until found = true of index >
   n if the value at index = target then
   return the index and set found =
   true else index = index + 1
4. If (not found) then output message that target is not found
5. Exit

It can be seen that initially index and flag found is initialized to 1 and „false" respectively. Everyvalue guided by index pointer is checked and if the target is found, then the flag is set

true and the corresponding index is sent. Otherwise, the failure message to find target is printed.

Formally this can be written as follows:

```
Algorithm Search(list, n , target)
Begin
index
= 1
found = false
repeat until found = true of index
    > nif (list_index = target ) then
        print “target found at”,
        indexfound = true
    else index = index + 1
if (not found) then print „Target is not
found“End
```

Thus one can conclude that after stepwise refinement, the control structures are evolved and can be written suitable as a pseudocode. Now let us discuss about recursive algorithms.

1.4 Basics of Recursion

Recursion is a way of defining an object using the concept of self-reference. The statements like “A length of the linked list of size N is 1 plus the length of the remaining linked list of size N-1” are examples of recursive definition.

Let us start with recursive definitions. The basic components of a recursive algorithm definition are given as follows:

- **Base case**: This is also known as initial values. This is the non-recursive part.

- **Recursive step**: It is the recursive part of the algorithm. Often this is a rule used for calculating a function in terms of the previous values of the function.

- **Step for progress towards base case**: This is a condition that ensures the algorithm eventually converges by bringing the recursive step towards the base case.

Let us discuss some recursive algorithms now. Let us consider the problem of finding summation problem given as $\sum_{i=1}^{n} n$. The recursive definition of summation can be given compactly as follows

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

A simple recursive algorithm for implementing the above recursive formula is given as follows:

1.4.1 Algorithm Sigma (N)

Program: Compute sum recursivelyInput: N

Output: Sum of N numbers

```
Begin

  if (N == 0)

  return

  0;

  else

  return N + Sigma(N

 -1);End if
```

End

1.4.2 Algorithm iterative-sigma(N)

The above algorithm can also be written as an iterative algorithm. The iterative algorithm isgiven as follows:

Program: Compute sum recursivelyInput: N

Output: Sum of N numbers

Begin

sum = 0;

for i = 1 to N do

sum = sum + I

end

for

End

It can be observed that both versions give identical answers. It can be observed that recursive algorithms are compact and it is easier to formulate the algorithms. But , the disadvantage is that recursive algorithms take extra space and require more memory.

Let us discuss about one more recursive algorithm for finding factorial of a number. The recursive function for finding the factorial of a number is given as follows:

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \text{x} \text{factorial}(n-1) & \text{if } n \geq 1 \end{cases}$$

The pseudocode for finding factorial of a number is given as

follows:Algorithm MFactorial(m)

Begin

If m < 0 then

print “factorial for negative numbers is not

possible”End if

If ((m=0) OR (m = 1)

thenReturn 1;

Else

Return m * MFactorial(m-1);

En

d

if

En

d.

Another important problem is towers of Hanoi. There are three pegs – source, intermediate and destination. The objective of this problem is to move the disks from source peg to the destination peg using the intermediate peg with the following rules:

- **At any point of time, only one disk should be moved**
- **A larger disk cannot be placed on the smaller disk at any point of time**

The logic for solving this problem is that, for N disks, N-1 disks are moved to the intermediate tower and then the larger disk is moved to the destination. Then the disks are moved from the intermediate tower to the destination.

The formal algorithm is given as follows:

1.4.3 Algorithm towersofhanoi(A,B,C,n)

Input: Source Peg A, intermediate Peg B and Destination Peg C, n disks

Output: All the disks in the tower C

Begin

if n = 1 the move disk from A

to Clse

towersofhanoi

(A,C,B,n-1);move disk

from A to C

towersofhanoi

(B,A,C,n-1);End if

End.

Check Your Progress
Fill in the Blanks.

Q.1: Recursive algorithm is an algorithm that uses functions that call __________conditionally.

Q.2: ___________is a structure whereby the computer system executes tasks one by one.

Q.3: ___________ can be written using natural language or pseudocode.

Q.4: Objective of the ________is to display whether the target is present in the given array, if so where? And failure message is the target is not present in the array.

1.5 Answer to Check Your Progress

Ans 1. Itself
Ans 2. Sequence
Ans 3. Algorithms
Ans 4. linear search

BLOCK-1
UNIT-3 ANALYSIS OF ALGORITHM

1.1 Learning Objectives

1.2 Analysis of Algorithm

1.2.1 Subjective measures

1.2.2 Objective measures

1.3 Apriori analysis (or Mathematical analysis)

1.3.1 Step count

1.3.2 Operation count

1.3.3 Second Philosophy: Count of Basic Operations

1.4 Summary

1.5 Answer to Check Your Progress

1.6 References

1.7 Model Questions

1.1 Learning Objectives

- To understand the process of algorithm analysis
- To know the types of analysis
- To understand the concept of step count
- To understand the Apriori analysis

1.2 Analysis of Algorithm

Algorithm analysis is necessary to determine the efficiency of an algorithm and to decide the efficiency of the given algorithm. Algorithmic Complexity theory is a field of algorithm that deals with the analysis of algorithmsin terms of computational resources and its optimization. An efficient algorithm is the one that uses lesser computer resources such as time and memory.

Analysis of algorithms is important for two reasons

- To estimate the efficiency of the given algorithm
- To find a framework for comparing the algorithms or solutions for the given problem.

But the analysis of an algorithm is not easy as analysis is dependent on many factors such as speed of machine, programming language, efficiency of language translators, program size, data structures used, algorithm design style, ease of its implementation, understandability and its simplicity.

Hence, analysis focus on two aspects – subjective measures and objective measures.

1.2.1 Subjective measures: Subjective measures are algorithm design style, ease of its implementation, understandability and its simplicity. For example, two algorithms can be compared based on ease of implementation. But, the subjective measures differ from person to person and time to time. Hence, to compare algorithms some objective measures or metrics are required.

1.2.2 Objective measures: Objective measures are measures like running time, memory space, disk usage and network usage.

Out of all objective measures, time complexity and space complexity are important.

Time complexity: Time complexity measures are very popular. Time efficiency or time complexity means how much run time an algorithm takes for execution for the scaled input. It expresses the complexity as some measure of input size and thus predicts the behavior of the given algorithm. By time, the run time is referred as compilation time is not important. Also the unit of running time is in terms of CPU cycle.

Let us recollect some of the important concepts for algorithm analysis.

- **Instance:** All valid inputs are called instances of the given algorithm.
- **Domain:** All the valid inputs form a domain
- **Input size**: Input size is the length of the input. Often, logarithm is also used to represent the size of the input to keep the number small as the logarithm of a number is often very small compared to the number itself. For example, the input size of the sorting program is N, if sorting involves N numbers. It can be noted that complexity is thus directly dependent on input size as sorting of 10 numbers is easy. But on the other hand, sorting of million numbers is a complex task.

Thus complexity is expressed as a function of input size, T(n) , where 'n' is the input size. T(n) can be estimated using two types of analysis.

- Apriori (or Mathematical analysis)
- Posteriori (or Empirical analysis).

1.3 Apriori analysis (or Mathematical analysis)

Apriori analysis (or Mathematical analysis) is done before the algorithm is translated to a program. The step count or the number of executed count of the dominant operations is used to estimate the running time. On the other hand, posteriori analysis is done by executing the program using standard datasets to estimate time and space.

The apriori (or Mathematical) algorithm analysis is framework involves following tasks:

- Measuring the input size of the algorithm.
- Measuring the running time using either step count or count of basic operations.
- Find the worst-case, best-case and average case efficiency of the algorithm as it is often naïveto think that for all kinds of inputs, the algorithm work uniformly.
- Identify the rate of growth. This step determines the performance of the algorithm when theinput size is increased.

Let us discuss now about step count and operations count.

1.3.1 Step count

T(n) is obtained using the step count of the instructions. The idea is based on the fact that the algorithm is basically a set of instructions. So the run time is dependent of the count of elementary instructions or program steps. Step count is the count of syntactically and semantically meaningful instruction and thus time complexity is expressed as a measure of step count.

1.3.2 Operation Count

Another popular school represented by Donald E Knuth prefer the determination of running time based on the count of basic operations. The idea is to use dominant operator and expressing the complexity as the number of times the basic (or dominant) operator is executed.

Let us discuss about some examples of step count. The idea is to count the instructions that are used by the given algorithm to perform the given task. The idea is to find the step count (called steps per execution s/e) of each instruction. Frequency is the number of times the instruction is executed. The total count can be obtained by multiplying the frequency and steps per execution. The s/e is not always 1. If a function is invoked, then s/e is dependent on

the size of the function.

Let us discuss about some examples.

Example 1

Consider the following program segment as shown in Table 1.

Table 1: Step count of swap of two variables

S.No.	Program	s/e	Frequency	Total
1	Algorithm Interchange(x,y)	0	-	-
2	Begin	0	-	-
3	temp = x;	1	1	1
4	y = x;	1	1	1
5	y = temp	1	1	1
6	End	0	-	-
	Total			3

The step count of instructions 1, 2 and 6 are zero as these are not executable statements. By this it is meant that the computer system need not execute these statements. The executable statements are 3, 4 and 5. It can be observed that the total is calculated based on the multiplication of s/e and frequency. So T(n) = 3. This means the algorithm computes a fixed number of computations irrespective of the algorithm input size and hence the behavior of this algorithm is constant.

Example 2

Consider the following algorithm segment as shown in Table 3.3:

Table 3: Step count of doubling of numbers

SNo.	Program	s/e	Frequency	Total
1	Algorithm Sum(A,n)	0	-	0
2	Begin	0	-	0
3	Sum = 0.0	1	1	1
4	For i = 1 to n do	1	n+1	n+1
5	Sum = sum + A[i]	1	N	n
6	End for	0	-	0
7	Return Sum	1	1	1
8	End	0	-	-
	Total			2n+3

Here, the instructions 1, 2 and 6 are non-executable. Statement 4 has frequency of n+1. This is because additional iteration is required for control to come out of the for-loop. The statements 5 and 6 are within the for-loop. So they are executed 'n' times. So the count of these instructions is n. It can be observed that the total count is obtained by multiplying the s/e and the frequency. Thus it can be seen that $T(n) = 3n + 3$ is the time complexity function of this algorithm segment.

1.3.3 Second Philosophy: Count of Basic Operations

As said earlier, the idea is to count all the operations like add, sub, multiplication and division. Some of the operations that are typically used are assignment operations, Comparison operations, Arithmetic operations and logical operations. The time complexity is then given as the number ofrepetitions of the basic operation as a function of input size. Thus the procedure for function op count is given as follows:

1. Count the basic operations of the program and express it as a formula

2. Simplify it

3. Represent the complexity as a function of count.

The rules for finding the basic operation count for some of the control constructs are given as follows:

- **Sequence**

Let us consider the following algorithm segment

```
Begin
S1
S2
End
```

If statement s1 requires m operations and statement s2 requires n operations. Then the whole algorithm segment requires m+n operations. This is called **addition principle**. For example for the algorithm segment

```
index = index + 1
sum = sum +
index
```

By addition principle, if C1 is the count of first instruction and C2 is the count of second instruction, the count of the algorithm segment is C1 + C2.

- **Decision**

For the decision structure, the rule is given below:

```
IF (Condition C) Then Statement P
Else
Statement Q
```

The time complexity of this program segment is given as T(n) = Maximum { T_P, T_Q}. For example, the following algorithm segment

```
if (n < 0)
    then
    absn = -
    n
```

else

absn = n

The op count is C1 + max (C2, C3) as C1 is the count of the condition. C2 and C3 are the op counts of the remaining instruction. Thus, the maximum operations of either if part or else part is taken as the time complexity.

- **Repetition**

The time complexity of the algorithm for loops is given as

$$T(n) \approx n \times T_R$$

Thus a statement s1 that requires m operations, is executed 'n' times in a loop, and then the program segment requires m X n operations. This is called **multiplication principle.**

Thus for the following algorithm segment

Instruction	cost	Frequency
i = 0	**c1**	**1**
sum = 0	**c2**	**1**
while (i <=n)	**c3**	**n+1**
i = i + 1	c4	n
sum = sum + i	c5	n

The total cost is given as c1 + c2 + n c3 + c3 + n c4 + n c5

= c1 + c2 + c3 + n(c3+c4+c5).

This is the complexity analysis of the above algorithm.

1.4 Summary

In short, one can conclude as part of this unit 1 that

5- Computer Science's core is algorithm study and Algorithmic thinking is necessary to solve problems.
6- Algorithms are the blueprint of how to solve problems and Efficiency is a necessity for algorithms.
7- Classification of Algorithms is based on implementation, design, problem type and tractability.

8- Step wise refinement leads to a better algorithm.

9- Control structures are sequence, Decision or conditional branching and repetition.

10- Pseudocode is better way of writing algorithms.

11- Algorithm efficiency is important for algorithm analysis.

12- Two types of analysis are – apriori (or Mathematical analysis) and Posterior (or Empiricalanalysis)

13- Time complexity can be estimated using step count or operation count.

Check Your Progress

Fill in the Blanks.

Q.1: The _______ or ________of the dominant operations is used to estimate the running time.

Q.2: If the statement s1 that requires m operations, is executed 'n' times in a loop, then the program segment requires _______ operations.

Q.3: __________ can be written using natural language or pseudocode.

Q.4: It is observed that _______ algorithms are compact and it is easier to formulate the algorithms.

1.5 Answer to Check Your Progress

Ans 1. step count, the number of executed count
Ans 2. m X n
Ans 3. Algorithms
Ans 4. recursive

1.6 References

8. S.Sridhar – Design and Analysis of Algorithms, Oxford University Press, 2014.
9. Cormen, T.H., C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press,Cambridge, MA 1992.
10. Gibbs, N.E., Tucker, A.B. "A model Curriculum for a Liberal Arts Degree in ComputerScience", Communications of ACM, Vol. 29, No. 3 (1986).
11. *S.Sridhar – Design and Analysis of Algorithms , Oxford University Press, 2014.*
12. Cormen, T.H., C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press,Cambridge, MA 1992.

13. *S.Sridhar – Design and Analysis of Algorithms, Oxford University Press, 2014.*
14. Cormen, T.H., C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge,MA 1992.

1.7 Model Questions

9. What is Algorithms?
10. What do you mean by Classification of Algorithm?
11. Explain recursion with example.
12. How to calculate time complexity of the algorithm? Explain with an example.

BLOCK 1
UNIT 4: Analysis of Recursive Algorithm

1.1 Learning Objectives
1.2 What Is Recursive Algorithm
1.2.1 What is Recurrence Equation?
1.3 Formulation of Recurrence Equation
1.4 Solution of Recurrence Equation
1.4.1 Methods to solve Recurrence Equation
1.5 Master Theorem
1.6 Summery
1.7 Answer to Check Your Progress
1.8 References

1.9 Model Questions

1.1 LEARNING OBJECTIVES

After the completion of this unit, you will be able to:

- Understand Recurrence equations.
- Know about the formulation of recurrence equations.
- Know the methods to solve recurrence equations.
- Understand and apply Master theorem.

1.2 WHAT IS RECURSIVE ALGORITHM?

In computer science, recursion is a computational problem-solving technique where the answer relies on finding solutions to smaller instances of the same problem. Recursion uses functions that call themselves from within their own code to address such recursive difficulties. Recursion is one of the fundamental concepts of computer science, and the method can be used to solve a wide range of issues.

Recursion is supported by the majority of computer programming languages, which permit a function to call itself from within another function. One typical technique in algorithm design is to break a problem down into smaller problems that are similar to the original, solve those smaller problems, and then combine the solutions, this is commonly known as the "divide and conquer" strategy [1].

1.2.1 What is Recurrence Equation?

A Recurrence equation for the sequence $\{a_n\}$ is a compact equation that expresses a_n and is terms of one or more of the previous terms of the sequence, namely a_0, a_1, a…, a_{n-1}, for all integers n with $n \geq n_0$, where n_0 is nonnegative integer.

1.3 FORMULATION OF RECURRENCE EQUATION

In algorithm analysis, a recurrence relation is a function relating the amount of work needed to solve a problem of size n to that needed to solve smaller problems [1].

No general automatic procedure for solving recurrence relation is known, but once this equation is formulated then we can easily solve the problem.

Example 1:

Mr. X deposits Rs.1000 in a saving account at a bank yielding 5% per year with interest compounded annually. How much money will be in the account after 20 years?

Let T(n) denote the amount in the account after n years. How can we determine T(n) on the basis of T(n-1)?

$T(n) = T(n-1) + 0.05 (T)(n-1) = 1.05 (T)(n-1)$

The initial condition $T(0) = 1000$

$T(1) = 1.05\, T(0)$

$T(2) = 1.05\, T(1) = (1.05)^2 T(0)$

$T(3) = 1.05\, T(2) = (1.05)^3 T(0)$

.

.

$T(n) = 1.05\, T(n-1) = (1.05)^n T(0)$

The formula to calculate T(n) for any natural number n can be guessed now as $(1.05)^n T(0)$.
For 20 years (Avoid repetition)
$T(20) = (1.05)^{20} . 1000$
Solving a recurrence relation employ finding a closed form solution for the recurrence relation.

Example 2:

Consider Towers of Hanoi Problem: There are number of Disks and corresponding numbers of moves –

No. of Disks	No. of moves
1	1
2	3
3	7
4	15
5	31

Here $2^n - 1$is the recurrence formula which satisfy all values of n. Let's verify for n=5, $2^5 - 1 = 31$.

Example 3:

Let's see how the recurrence equation is formulated for Fibonacci Series.

Fibonacci sequence begins as follows:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,……..

Each subsequent number is the sum of the two preceding numbers. Therefore,

Fin(n) = Fib(n-1) + Fib(n-2) or

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

Example 4:

Let's see one more example Stair Case Problem, where you can walk up steps by going up one at a time or two at a time. Like three steps can be walked in three ways: 1 or 1+2 or 2+1.

Steps number	Numbers of possibilities
1	1
2	2
3	3
4	5

Descends to 1, 2, 3, 5, 8, 13, 21,…..

So, T(n) = T(n-1) + T(n-2)

Therefore, we can conclude that Staircase problem is also having same recurrence equation as Fibonacci series T(n) = T(n-1) + T(n-2)

In other words, we can say that, the most difficult part in recurrence equation analysis is the formulation of recurrence equation. Let's us review some of the key terminologies of recurrence equation:

- A linear relation is one where all factors of T(n) have power 1.

-The term linear means that each term of the sequence is defined as a linear function of the preceding terms.

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

- A nonlinear relation is the one where factor may have powers other than 1.

-Example: $T(n+1) = 2nT(n-1)(1-T(n-1))$
-$T(n) = T(n/2) + 1$

- A recurrence relation is said to have constant coefficient if the factors of T(n) have constant coefficients.

-Fibonacci relation is homogeneous and linear:

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

-Non-constant coefficients:

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

- Order of a relation is defined by the number of previous terms used in computing n^{th} terms.

- ***First order:*** $T(n) = 2T(n-1)$ - n^{th} term depends only on terms n-1.

- ***Second order:*** $T(n) = T(n-1) + T(n-2)$

n^{th} term depends only on term n-1 and n-2.

- ***Third order:*** $T(n) = 3nT(n-2) + 2T(n-1) + T(n-3)$

n^{th} term depends on three terms.

1.4 SOLUTION OF A RECURRENCE EQUATION

Find the recurrence equation, having a lot of recursive terms and we have to find closed-form formula. The closed-form formulas that do not having any recursion is the calling a solution.

For example:

Consider the recurrence relation

$T(n) = 2T(n-1) – T(n-2)$ for n=2, 3, 4, ……

Let $a_n= 3n$ be a closed-form formula to solve this recurrence equation. One can verify that for n>=2

$2a_{n-1} - a_{n-2} = 2(3(n-1)) – 3(n-2) = 3n = a_n$

Therefore, $a_n = 3n$ is a solution of the recurrence relation.

1.4.1 Methods to solve Recurrence Equation

- Guess and verify method
- Substitution method
- Recurrence Tree method

Guess and verify method:

The Guess and verify method for solving recurrence, entails two steps:

1. ***Guess*** the form of the solution.
2. Use ***mathematical Induction to verify the solution***.

For example, we will take Tower of Hanoi problem. The recurrence equation already we have found out it is:

$T_n = 2T_{n-1} + 1; T_0 = 0$

n =	0	1	2	3	4	5………
T_n=	0	1	3	7	15	31……..

Guess- $T_n = 2^n - 1$

$T_n = 2T_{n-1} + 1; T_0 = 0$

Prove: $T_n = 2^n - 1$ by induction;

1. **Base case**; n=0; $T_0 = 2^0 - 1 = 0$
2. **Inductive Hypothesis (IH);** $T_n = 2^n - 1$ for $n \geq 0$
3. **Inductive step; show** $T_{n+1} = 2^{n+1} - 1$ for $n \geq 0$

   $T_{n+1} = 2T_n + 1$
   $= 2 (2^n - 1) + 1$ (applying IH)
   $= 2^{n+1} - 1$

Substitution method:

1. ***Plug-*** Substitute repeatedly
2. ***Chug-*** Simplify the expressions

Solve T(n) = T(n-1) + 3, T(1) = 3
Substitute the value repeatedly

T(n-1) = (T (n-2) + 3) + 3 (plug)
= T(n-2) + 3 + 3
= T(n-2) + 2 * 3

Repeatedly Substitute the value of T(n-2)
Substitute the value of T(n-3)

T(n-3) = (T (n-4) + 3) + 3 * 3 (plug)
= T(n-3) + 3 + 3 * 3
= T(n-2) + 4 * 3

Repeat again!
The pattern is ………..

Result at i^{th} unwinding	I
T(n) = T(n-1) + 3	1
T(n) = T(n-2) + 2*3	2
T(n) = T(n-3) + 3*3	3
T(n) = T(n-4) + 4*3	4

The pattern is at step i
T(i) = T(n-i) + i*3
When i = n-1, the equation would become
= T(n-(n-1)) + (n-1) * 3
= T(1) + 3n – 3
= 4 + 3n – 3 = 3n + 1

So, this method is also highly popular for solving linear recurrence equation and using this method we can solve problem very well.

Recurrence Tree method:

1. ***Visualize the recurrence tree***
2. Find the following information:

Level of the tree- Level is the longest path from root to leaf
Cost per level
Total cost
Complexity is the total cost.

Here while solving recurrence, we divide the problem into sub-problems of equal size.
For e.g., T(n) = aT(n/b) + f(n) where a>=1, b>1 and f(n) is a given function.

F(n) is the cost of splitting or combining the sub problems. Fig.1 shows how it works.

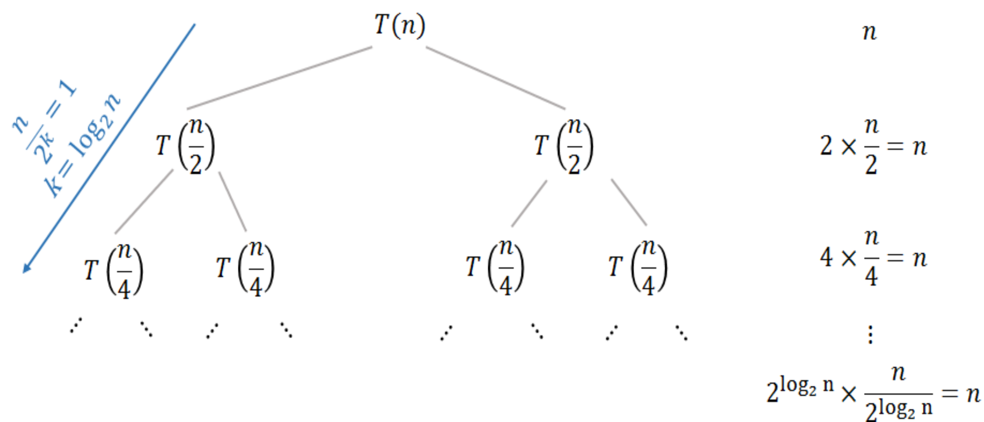


Fig.1 T(n) = 2T(n/2) + n

When we add the values across the levels of the recursion tree, we get a value of n for every level.

we have n + n + n +…….. log n times

= n(1+1+1+……..log n times)

= n ($\log_2$ n)

= Θ (n log n)

T(n) = Θ (n log n)

So, the recurrence tree has certain advantage that it is more visible in comparison to all other methods.

1.5 MASTER THEOREM

If the non-linear recurrence is given then we can use Master Theorem. This theorem mentioned as:

Suppose recurrence equation given-

T(n) = a T(n/b) + f(n)

- If $f(n) = O(n^{\log_b a-\omega})$ for constant ω > 0,

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

- If $f(n) = \Theta(n^{\log_b a})$,

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

- If $f(n) = \Omega(n^{\log_b a+\omega})$ for constant ω > 0,

and if a f(n/b) ≤ c f(n) for some constant c < 1and all sufficiently large n, T(n) = Θ (f(n)).

- Key idea ***compare $n^{\log} b^a$ with f(n).***

Example:

T(n) = 9T (n/3) + n

a = 9, b = 3, f(n) = n

$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

since f(n) = O ($n^{\log_3 9-\omega}$), where ω = 1, case 1 applies.

T(n) = $\Theta(n^{\log_b a})$ when f(n) = $O(n^{\log_b a-\omega})$

Thus, the solution is T(n) = $\Theta(n^2)$

When Master's Theorem cannot be applied:

- If 'a' is not a constant or less than 1

Example: T(n) = 2^nT(n/2) + n or

T(n) = 0.3T(n/2) – n

- When f(n) is negative

T(n) = T (n/2) – n

- When growth is not polynomial

T(n) = 2T (n/2) + n/log n

There are some additional problems of Master's Theorem

- T(n) = 3T (n/5) + n
- T(n) = 2T (n/2) + n
- T(n) = 2T (n/2) + 1
- T(n) = T (n/2) + n
- T(n) = T (n/2) + 1

1.6 SUMMARY

In this unit we discussed:

1. Formulation of Recurrence equation is the key for finding complexity of recursive algorithms.
2. Guess and verify, Substitution method, and Recurrence Tree methods are useful to solve recurrence equations.
3. Master theorem is useful in solving divide and conquer equations.

Check Your Progress

Multiple choice questions

1. What is a recurrence equation?
   a) An equation that only has constants
   b) An equation that defines a sequence in terms of its previous terms
   c) An equation that involves trigonometric functions
   d) An equation that represents a straight-line graph Symmetric key cryptography uses the same key .

2. Which of the following is an example of a linear recurrence relation?
   a) n! = n x (n-1)!
   b) F(n) = F (n-1) + F (n-2)
   c) 2^n
   d) sin(n)

3. What is the purpose of formulating a recurrence relation in algorithms and computer science?
   a) To simplify mathematical expressions
   b) To solve linear equations
   c) To express a problem's solution in terms of smaller instances of the same problem
   d) To create recursive functions
4. Which of the following is a common technique for formulating recurrence relations based on recursive algorithms?
   a) Iterative loops
   b) Dynamic programming
   c) Divide-and-conquer
   d) Hashing

5. What role does the "initial condition" play in solving a recurrence relation?
   a) It helps identify the characteristic equation
   b) It simplifies the calculation of constant coefficients
   c) It determines the order of the recurrence relation
   d) It serves as the base case for induction

1.7 ANSWER TO CHECK YOUR PROGRESS

1. b   2. b   3. c   4. c   5. d

1.8 REFERENCES

[1] https://stackoverflow.com

1.9 MODEL QUESTIONS

2. What is Recursive algorithm?
3. What is Recurrence equation?
4. What are the 3 methods to solve recurrence equations?
5. What is Recurrence Tree method? Describe it.

6. Describe Master's Theorem.

BLOCK 2

UNIT 5- Brute Force Technique and Unintelligent Search

1.1 Learning Objectives

1.2 Brute Force Technique

1.2.1 The advantages of the brute force approach

1.2.2 The advantages of the brute force approach

1.3 Optimization Problem and Exhaustive Searching

1.4 Solution Space and Unintelligent Search techniques

- 1.4.1 DFS Search
- 1.4.2 BFS Search
- 1.4.3 15-Puzzle Problem
- 1.4.4 8-Queen Problem
- 1.4.5 Knapsack problem
- 1.4.6 Assignment Problem

1.5 Answer to Check Your Progress

2.1 Introduction to Divide and Conquer Technique

- 2.1.1 Advantages of Divide and Conquer Technique
- 2.1.2 Disadvantages of Divide and Conquer Technique

2.2 Marge Sort

2.3 Quicksort

2.4 Partitioning Algorithms

2.5 Hoare Algorithm

2.6 Answer to Check Your Progress

3.1 What is divide and Conquer design paradigm?

3.2 Multiplication of Long Integers
3.3 Strassen Matrix multiplication Algorithm
3.4 Strassen Matrix Multiplication
3.5 Summary
3.6 Answer to Check Your Progress
3.7 References
3.8 Model Questions

1.1 Learning Objectives

This unit focuses on the basics of the combinatorial optimization problems. The Learning objectives of this module are as follows:

- To introduce Combinatorial Optimization Problems and brute force approach
- To understand the concept of Exhaustive Searching
- To know about 15-Puzzle game, 8-Queen Problem, Knapsack Problem and assignment Problem
- To understand the concept of Divide and Conquer
- To understand applications of Divide and Conquer technique
- To know about Merge Sort algorithm and Quicksort Algorithm
- To understand how long integer multiplication can be done using divide and conquer designparadigm
- To know about Matrix Multiplication
- To understand Strassen Multiplication Algorithm for multiplying matrices faster.

1.2 Brute Force Techniques

What is a brute force approach?

A brute force approach is a straight forward approach based on the problem statement and definitions of the concepts involved [2]. The general framework of brute force approach of brute force approach is as follows:

- Generate all combinatorial structures that represent all feasible solutions. The combinatorial structures are sets, trees, graphs, permutations, and Catalan families.
- Search for the best solution among the generated feasibility solutions.

1.2.1 The advantages of the brute force approach is as follows:

Wide applicability of brute force approaches. Brute force approach can be applied for all variety of problems.

1. Simplicity of the brute force approach.
2. Brute force approach can yield reasonable algorithms for all important problems.
3. Brute force approach can yield algorithms that can be used as a benchmark for comparingalgorithms of other design approaches.

1.2.2 The disadvantage of brute force approaches are

1. Often brute force algorithms are inefficient.
2. There is not much constructive or creativity involved in brute force algorithms.

1.3 Optimization Problem and Exhaustive Searching

Exhaustive searching is a strategy usually employed for combinatorial problems. What is anoptimization problem? An optimization problem has

1. Objective function: This is in the form of maximization or minimization of some constraint.
2. Predicate P that specifies the feasibility criteria
3. Solution space U with all possible solutions and extremum requirements
4. The aim is to find solution that satisfies the feasibility criteria.

Some of the examples of optimization problems that are discussed in module are 15-puzzle problem, 8-Queen problem, Knapsack problem and Assignment problem. Combinatorial optimization problem uses combinatorial structures such as sets, trees, graphs, permutations, and Catalan families.

Exhaustive search is a strategy for solving combinatorial optimization problems is to use exhaustive search. Exhaustive search involves these steps:

1. List all solutions of the problem
2. Often state space tree is used to represent all possible solutions.
3. Then the solution is searched in the state space tree that has least cost or optimalusing a search technique.

This approach is using a brute force to search for finding solutions. The advantage is the guaranteed solution. But the disadvantage of this approach is a problem called "Combinatorial explosion" where the increase in input is associated with the rapid increase in output.

Let us discuss about applying brute force approach for some of the important problems now:

1.4 Solution Space and Unintelligent Search techniques

The solution space is in the form of a graph. A sample solution space is shown in Fig. 2. Some of the important terminologies used in association with the solution space is given below:

1. **Root:** Root of a graph has no predecessor. In solution space, it is the initialconfiguration.
2. **Solution Space:** All the configurations in the form of a state space tree is calledsolution space. The target configuration is somewhere in the solution space.
3. **Search techniques**: Once a solution space is available, then the tree can besearched using search techniques like DFS and BFS.

Let us review about DFS and BFS now:

1.4.1 DFS Search

Depth First Search (DFS) is an unintelligent search that is used to search a graph. DFS uses a stack for traversing a graph. Initially, the root node is pushed onto a stack. Then, it is checked forgoal node. If it is a goal node, then DFS reports success. Otherwise, its children is generated and pushed onto a stack. This procedure is repeated till stack is empty.

The procedure of DFS is

given below: Put the

```
root node on a stack S;
    while (S is not
    empty)          {
    remove a node
    from the S;
if (node = goal node)
return success; put all
children of node onto the
S;
 }
return failure;
```

The following example illustrates usage of DFS.

Example 1: Assume goal nodes are N and J. Show the DFS search of the following graph shown in Fig. 1 for finding the goal nodes?

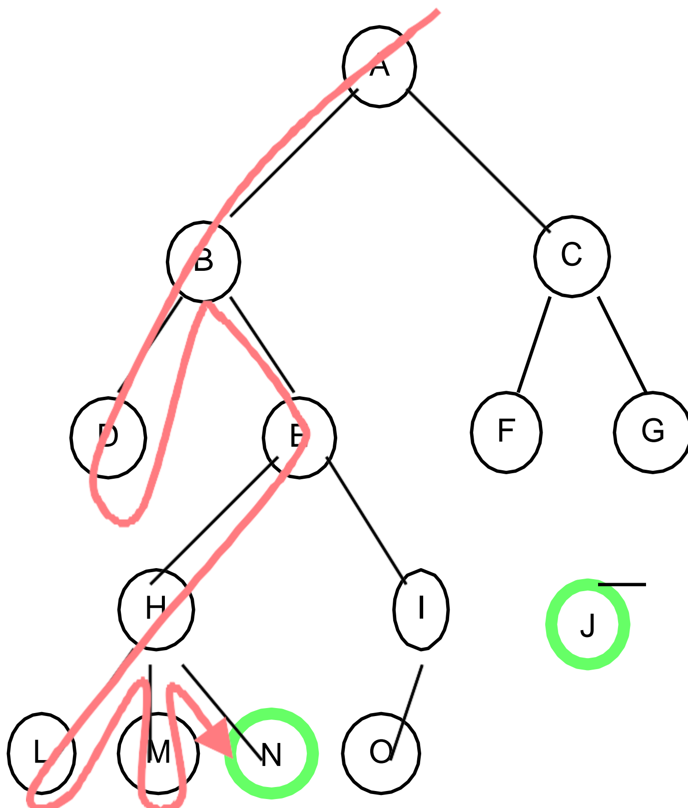


Fig 1: A sample Graph

Nodes that are explored in the order are ; A B D E L M N I

O C F

1.4.2 BFS Search

Breadth First Search (BFS) is also an unintelligent search that is used to search a graph. BFS traverse a graph level-by-level. It uses a queue data structure for traversing a graph. Initially, the root node is added onto a queue. Then, it is checked for goal node. If it is a goal node, then BFS reports success. Otherwise, its children is generated and pushed onto a rear end of the queue Q. This procedure is repeated till Queue is empty.

The procedure of BFS is given below:

```
Put the root node on
a queue Q;while (Q is
not empty) {
    remove a node from the queue Q;
    if (node = goal node) return success;
    put all children of node onto the queue Q;
}
return failure;
```

The following example illustrates usage of BFS.

Example 2: Assume goal nodes are N and J. Show the BFS search of the following graphshown in Fig. 2 for finding the goal nodes?

Example 2: Assume goal nodes are N and J. Show the BFS search of the following graphshown in Fig. 2 for finding the goal nodes?

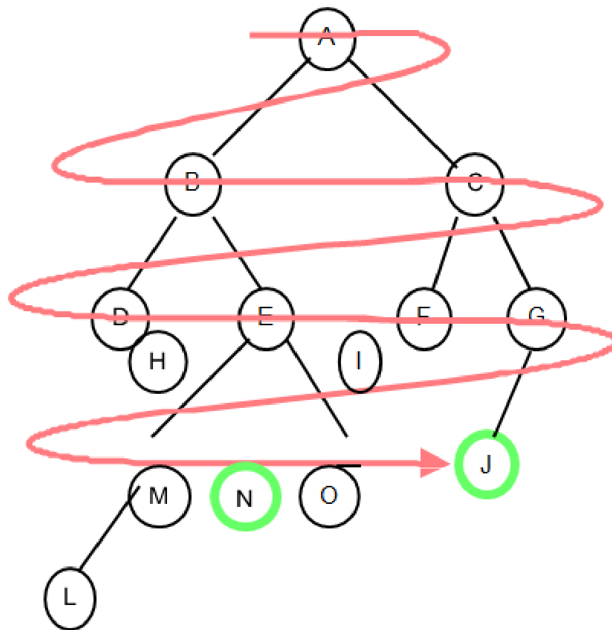


Fig 2: A sample Graph

The nodes that are explored in BFS order is : A B C D E F G H I J L M N O

1.4.3 15-Puzzle Problem

In 1878, Sam Lloyd designed a puzzle called 15-puzzle game. It is a game where an initial and target configuration is given. The game is about moving tile so that the target configuration is reached from the initial configuration. In other words, the objective of the game is to change initial state to goal state. The possible configurations of moving the up, down, right or left using the empty tile. So a node can have at most possible four moves. A sample initial and target configuration is given below in Fig. 3.

$$\begin{bmatrix} 1 & 5 & 12 & 13 \\ 2 & 6 & 11 & 14 \\ 3 & 7 & - & 15 \\ 4 & 8 & 9 & 10 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 5 & 12 & 13 \\ 2 & 6 & 11 & 14 \\ 3 & 7 & 15 & - \\ 4 & 8 & 9 & 10 \end{bmatrix}$$

Initial Configuration   Target Configuration

Fig. 3. : Initial and Target configuration

A brute force algorithm can be written to generate the solution space. The repeated application of the legal moves, top, bottom, right and left results in a graph called state space or **solution space**. This is shown in Fig. 4.

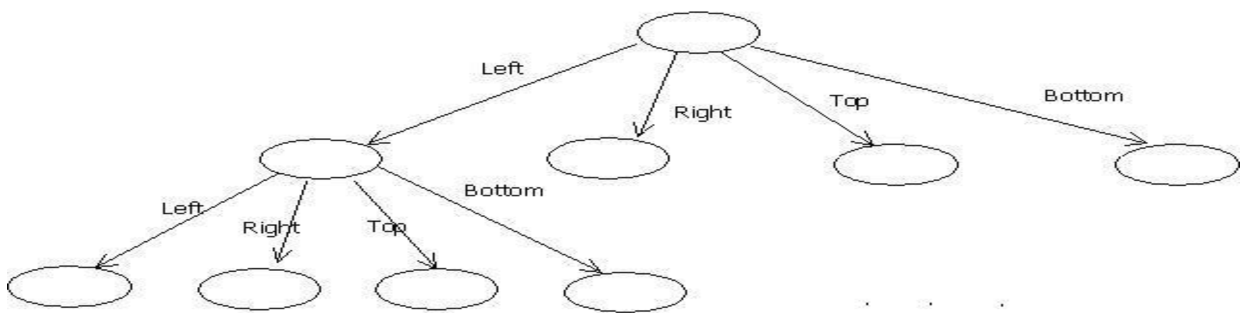


Fig. 4: **Portion of a Graph**

The goal state may be present somewhere in the solution space. Now, Exhaustive search technique is about finding the path from the starting node to the goal state. This may be done using DFS or BFS.

- **Formal Algorithm**

The formal algorithm based on [1] for 15-puzzle is given as follows:

Algorithm 15puzzle (G, root, goal)

%% Input: State space tree with root and target goal state

%% Path from root to

goal nodeBegin

```
 for all nodes make visit = 0

%%  Let  root  be  the
     starting       node
     visited[root] = 1
     generate children w for root for all w do
     If (visited[w] = 0) and (w is not goal
     node) thencall DFS(G,w)
     e
n
d

i
f
e
n
d

f
o
r

E
n
d
```

- **Complexity Analysis**

This brute force algorithm for 15-Puzzle game is inefficient and unrealistic. For 15-puzzle problem, there will be 16! ($\approx 20.9 \times 10^{12}$) different arrangements of tiles. Hence searching for the solution takes exponential time.

1.4.4 8-Queen Problem

The problem of eight queens is that of placing 8 queens in a non-attacking position. The problem of 8-queen problem is to generate a board configuration where the queens are in non-attacking positions.

The movement of queen is shown in Fig. 5. It can be observed that a queen can move in horizontal, vertical and diagonal ways.

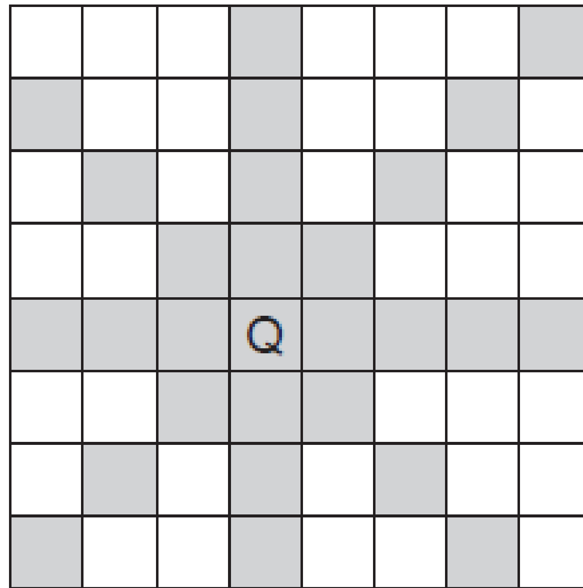


Fig. 5: The movement of a Queen

The brute force algorithm would be to try all possible combinations to check out the placement of the queens such that they are in a non-attacking position.

The informal algorithm is as follows:

1. Try all combinations
2. Check whether the queens are in attacking position
3. Repeat steps 1 and 2 until a valid configuration is possible.

The formal algorithm based on [1] is given as follows:

Algorithm 8queen(n)

%% Input: A 8 x 8 chess board and '8' queens

%% **Output**: Solution of 8-queen problemBegin

```
n = 8
    for i1 = 1 to n do for i2 = 1 to n do
        for i3 = 1 to n do for i4 = 1 to n do
            for i5 = 1 to n do for i6 = 1 to n do
                for i7 = 1 to n do for i8 =
                  1 to n do sol = [i1, i2,
```

```
                    i3, i4, i5, i6, i7, i8]
                    %% check the solution with respect to constraints of
                    %% non-attacking queen
                      if   sol   is   correct
                    then print solEnd if
                  E
             n
             d

             f
             o
             r
             E
             n
             d

             f
             o
             r
         E
        n
        d

        f
        o
        r
        E
        n
        d

        f
        o
        r
      E
     n
     d

     f
     o
     r
     E
     n
     d

     f
     o
     r
   E
```

n
d

f
o
r
E
n
d

f
o
r

E
n
d
.

- **Complexity Analysis:**

Even though, the solution for 8-queen problem looks simple, in reality solution involves a combination of $\frac{64!}{56!}$ positions (56 = 64-8). This makes exhaustive search a difficult process.

There are many solutions for the 8-queen problem.

1.4.5 Knapsack problem

Knapsack problem is one of the most popular algorithms that we often encounter in our daily life. It was designed by Denzig in 1950. A Knapsack problem has a knapsack of capacity K.

There are n different items, each of which is associated with W_i and profit Pi (It is also called as value). The objective of knapsack problem is to load knapsack to maximize profit subjected to the capacity of knapsack.

Mathematical Formulation:

Mathematically, the problem can be stated as

$$Maximize \sum_{i=1}^{n} p_i x_i$$

Subjected to the constraint that

$$\sum_{i=1}^{n} w_i x_i \leq K$$

$$0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

There are two types pf knapsack problem. It is easy to solve **fractional knapsack problem. This problem allows loading of knapsack with fractional items. This is possible if the items to be loaded are like** cloth, liquid, gold dust, etc. On the other hand, Integer knapsack problem, also known as 0/1 knapsack, is hard to solve. This problem is applicable for items which can either be loaded full or not at all. The items here are like electronic items, machineries or any items that can be broken.

Let us consider a scenario where there are three bottles of Salt with their weights and profits as shown in Fig. 6.

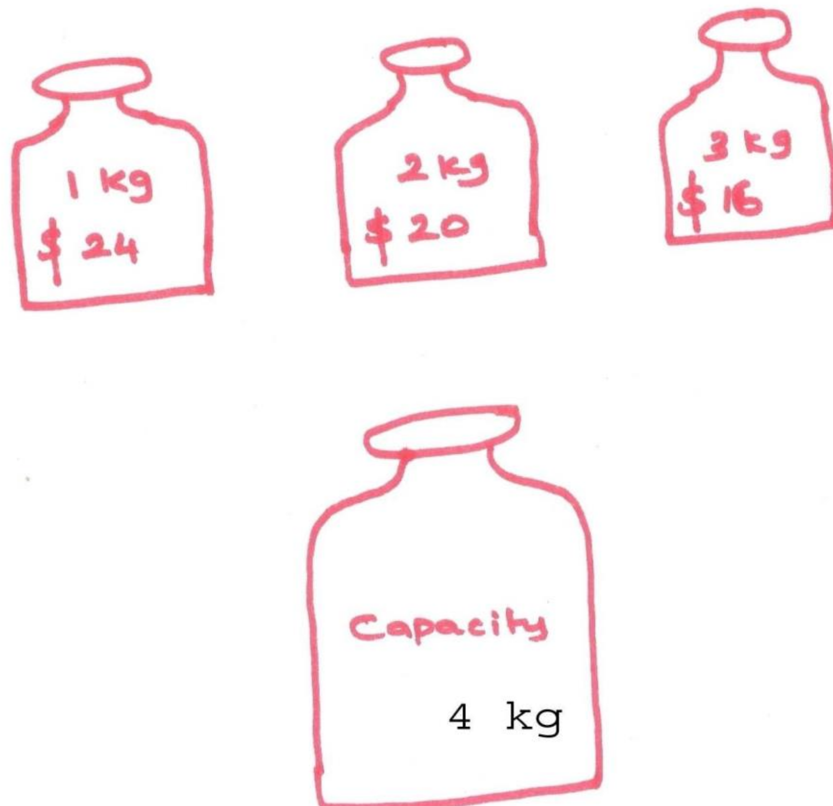


Fig.6 : Knapsack problem

The bigger bottle is knapsack. The capacities and profits are are shown in Fig. 6. So the question here is how to fill the knapsack effectively so that profit is higher subjected to the capacity of the knapsack.

- **Informal Algorithm:**

1. Generate all possible sets. There will be 2^n possible subsets for 'n' items.
2. Generate a binary string of length
3. If the binary bit is 1, then include, otherwise exclude the item subjectedto the constraints.
4. Compute the maximum profit and Weight for all combinations andreport maximum profit.
5. Exit.

- **Formal Algorithm:**

Formally, the knapsack problem based on [1] is given as follows:

Algorithm knapsack(K, w[1..n],p[1..n])

%% Input: Knapsack with capacity K , weights and profits

%% Output: Knapsack items and maximum weight and profit

Begin

```
 globalprofit = 0
choice = 0 index = 1
while (index <= 2^n ) do                %% for all 2^n subsets
   profit = 0
   weight = 0
     for i = 1 to n do               %% for all n items
        if (binary(index, i) == 0)  then        %% Item is excluded if binary bit is
           0profit = profit + 0
           weight = weight +0                %% generate binary bit i for item index
          elseprofit = profit + p[i]        %% Update if Binary bit is 1
           weight = weight +w[i] End
     ifEnd for
        if ((weight <= K) and (profit > globalprofit)) then %% Check the constraintsglobalprofit =
        profit
        choice = i                        %%       Note       the
     combinationEnd if
  index = index
  +1End while
End
```

This is illustrated in Example 3.

Example 3: Consider three items. Generate all the possibilities:

The possibilities are

ϕ

{1}

{2}

{3}

{1, 2}

{1, 3}

{2, 3}

{1,2,3}

Then all combinations are tried out subjected to the constraints. The profit is computed for every case and maximum profit is reported.

- **Complexity Analysis:**

This involves generation of 2^n combinations. Hence, the complexity of the algorithm leads to a Ώ (2^n) algorithm. The implication of NP-hard is that it would be difficult to solve the problem.

1.4.6 Assignment Problem

Assignment problem is to assign m persons to n jobs such that the cost is minimized. This problem can be solved using brute force approach. All possible combinations are tried out and cost is computed. Finally, the least cost assignment is reported.

- **Informal Algorithm**

The informal algorithm is stated as below;

1. Generate all permutation of assignments
2. Compute the cost for all assignments
3. Choose the minimum cost assignment and report.
4. End.

- **Formal Algorithm**

The formal algorithm based on [1] is given below:

```
Algorithm Assign(person[1..n],job[1..n])
%% Input; person and jobs
%% Output: Optimal assignment of person to
jobBegin
    permutate all legitimate assignments ai
    compute the assignment [a1, a2,.., an] whose cost is
    minimumassign as per minimum cost and return min_cost
End
```

The following Example 4 illustrates the application of this algorithm:

Example 4: A sample assignment Table in given in table 1.

Table 1: Sample assignment Table

	J1	J2	J3
person 1	3	4	5
person 2	2	3	7
person 3			

Use assignment algorithm and find optimal cost?

There are 3 persons and 3 jobs. The question is how to assign the jobs to persons in an optimal manner based on the cost matrix given above. For example, if a random assignmentis made like assign the job 1 to person 1 (C(1,1)), job 2 to person 2 (C(2,2)) and job 3 to person 3(C(3,3)), then the cost of the assignment would be

< 1 2 3> = 3 + 3 + 2 = 8

For the above problem, all the possible cost assignments are given below in Table 2.

Table 2: Possible assignments

J1	J2	J3	Cost
1	2	3	3 + 3 + 2 = 8
1	3	2	3 + 7 + 9 = 19
2	1	3	4 + 2 + 2 = 8
2	3	1	4 + 7 + 8 = 19
3	1	2	5 + 2 + 9 = 16
3	2	1	5 + 3 + 8 = 16

It can be observed that the optimal order is < 1 2 3> and <2 1 3> as they are associated least cost.

- **Complexity Analysis**

Again, the complexity analysis shows that the number of permutations are n!. Therefore, the complexity of the problem is O(n!).

Check Your Progress

Fill in the Blanks.

Q.1: Exhaustive search is a strategy for solving ____________________________.

Q.2: Breadth First Search (BFS) is an ______________that is used to search a graph.

Multiple Choice Questions.

Q.3: How many queens can be placed on an 8x8 chessboard without attacking each other?

A) 4

B) 6
C) 8
D) 12
Answer: C) 8
Q.4: In the Assignment Problem, what is the objective?
A) To find the maximum profit
B) To find the minimum cost
C) To maximize resource utilization
D) To minimize the number of assignments
Q.5: In the 0/1 Knapsack Problem, what type of items can be either included or excluded in the knapsack?
A) Fractional items
B) Weightless items
C) Discrete items
D) Continuous items
Q.6: What is the primary objective of the 15-Puzzle game?
A) To arrange numbers from 1 to 15 in ascending order
B) To create a pattern on the board
C) To eliminate tiles from the board
D) To rearrange the tiles in any order

1.5 Answer to Check Your Progress

Ans 1. **Combinatorial optimization problems**
Ans 2. **Unintelligent search**
Ans 3. **8**
Ans 4. **To find the minimum cost**
Ans 5. **Discrete items**
Ans 6. **To arrange numbers from 1 to 15 in ascending order**

2.1 Introduction to Divide and Conquer Technique

Divide and conquer is an effective algorithm design technique. This design technique is used to solve variety of problems. In this module, we will discuss about applying divide and conquer technique for sorting problems. In this design paradigm, the problem is divided into subproblems. The subproblems are divided further if necessary. Then the subproblems are solved recursively or iteratively and the results of the subproblems are combined to get the final solutionof the given problem.

These are the important components of Divide and Conquer strategy:

1. Divide: In this stage the given problem is divided into small problems. The smaller problems are similar to the original problem. But these smaller problems have reduced size, i.e., with less number of instances compared to original problem. If the subproblems are big, then the subproblems are divided further. This division process is continued till the obtained subproblemsare smaller that can be solved in a straight forward manner.

2. Conquer: The subproblems can be solved either recursively or non-recursively in a straightforward manner.

3. Combine: The solutions of the sub-problems can be combined to get the global result of theproblems.

2.1.1 Advantages of Divide and Conquer Paradigm

1. The advantages of divide and conquer approach is that it is perhaps most commonly applied design technique and its application always leads to effective algorithms.
2. It can be used to solve general problems.
3. Divide and conquer paradigm is suitable for problems that are inherently parallel in nature.

2.1.2 Disadvantages of Divide and Conquer

The disadvantage of divide and conquer paradigm is that if division process is not carried in a proper manner, the unequal division of problem instances can result in inefficient implementation.

Let us discuss about one of the most popular algorithm that is based on divide and conquer, i.e., merge sort.

2.2 Merge Sort

Divide and conquer is the strategy used in merge sort. Merge sort was designed by the popular Hungarian mathematician John van Neumann. The procedure for merge sort is given informally as follows:

1. Divide: Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elementseach.

2. Conquer: Sort the two subsequences recursively using merge sort in a recursive or non-recursive manner.

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

- **Informal algorithm**:

Informally merge sort procedure is as follows:

1. Divide the array A into subarrays L and R of size n/2.

2. Recursively sort the subarray L gives L sorted subarray

3. Recursively sort the subarray R gives R sorted subarray

4. Combine L and R sorted subarrays give final sorted array A

The formal algorithm based on [3] is given as follows:

MergeSort **(*A, p, r*) //** sort *A*[*p..r*] by divide & conquer

1 **if** $p < r$

2 **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$

3 *MergeSort* (*A*, *p*, *q*)

4 *MergeSort* (*A*, *q*+1, *r*)

5 *Merge* (*A*, *p*, *q*, *r*) // merges *A*[*p..q*] with *A*[*q+1..r*]

It can be observed that given array A has p and r as lowest and highest indices. The mid-point p is computed so that the given array is divided into two subarrays. Then the merge sort procedure is called so that the array is recursively divided. Then the procedure uses merge to combine the sorted subarrays,

The formal algorithm based on [3] for merging the subarray is given as follows:

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

1 $n_1 \leftarrow q - p + 12$

$n_2 \leftarrow r - q$

3 **for** $i \leftarrow 1$ **to** n_1

```
4      do L[i] ← A[p + i – 1]
5   for j ← 1 to n2
6   do R[j] ← A[q + j] 7
    L[n1+1] ← ∞
8   R[n2+1] ← ∞
9   i ← 1
10  j ← 1
11  for k ←p to r
12     do if L[i] ≤ R[j]
13         then A[k] ← L[i]
14              i ← i + 1
15         else A[k] ← R[j]
16              j ← j + 1
```

It can be observed that the elements of A is divided into the subarray L and R. Then the elements of L and R are compared and the smaller element is copied to the array A. If the subarray is exhausted, then the remaining elements of the other subarray is copied to array A. ∞ is given as sentinel so that comparison is not done for each and every time forthe end of subarray.

The following Example 1 illustrates the function of merge sort:

Example 1

Use merge sort and sort the array of numbers {18,26,32,6,43,15,9,1,22,26,19,55,37,43,99,2}

Solution

As said earlier, the first phase of merge sort is to divide the array into two parts using the middle element. The sub-arrays are divided further till one gets an array that cannot be divided further. This division process is shown as below in Fig. 1.

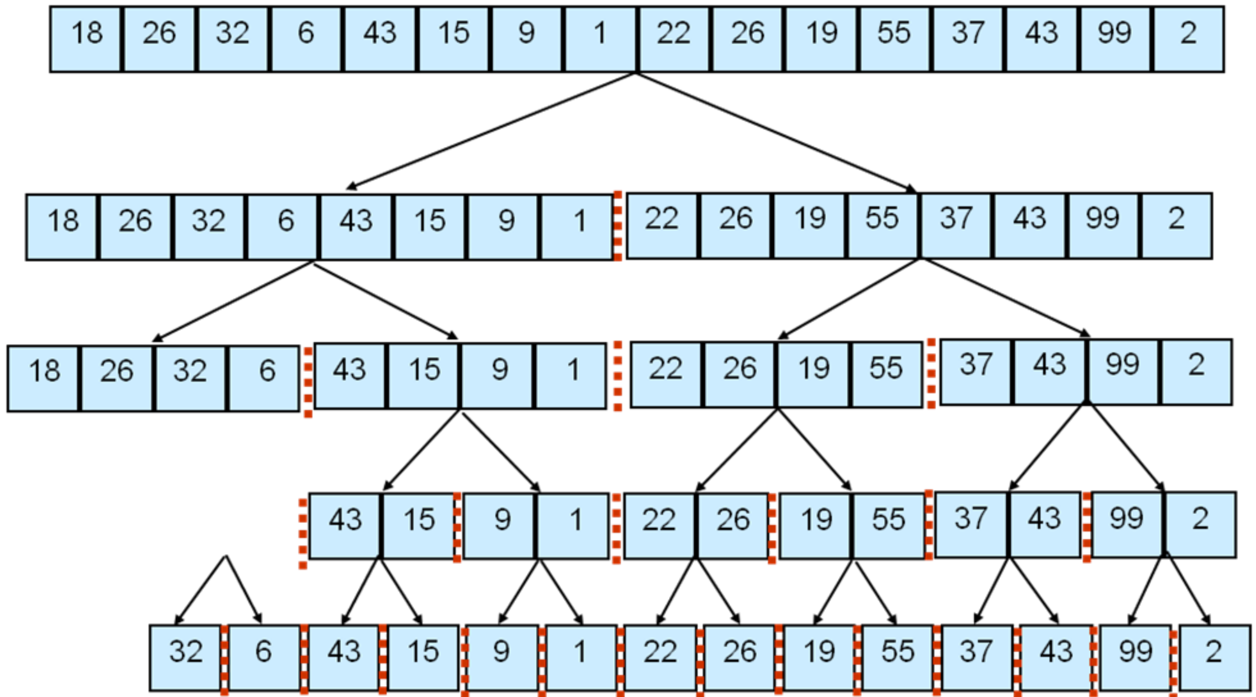


Fig 1: Division process

Then, the elements are merged is illustrated for L shown below in Fig. 2.

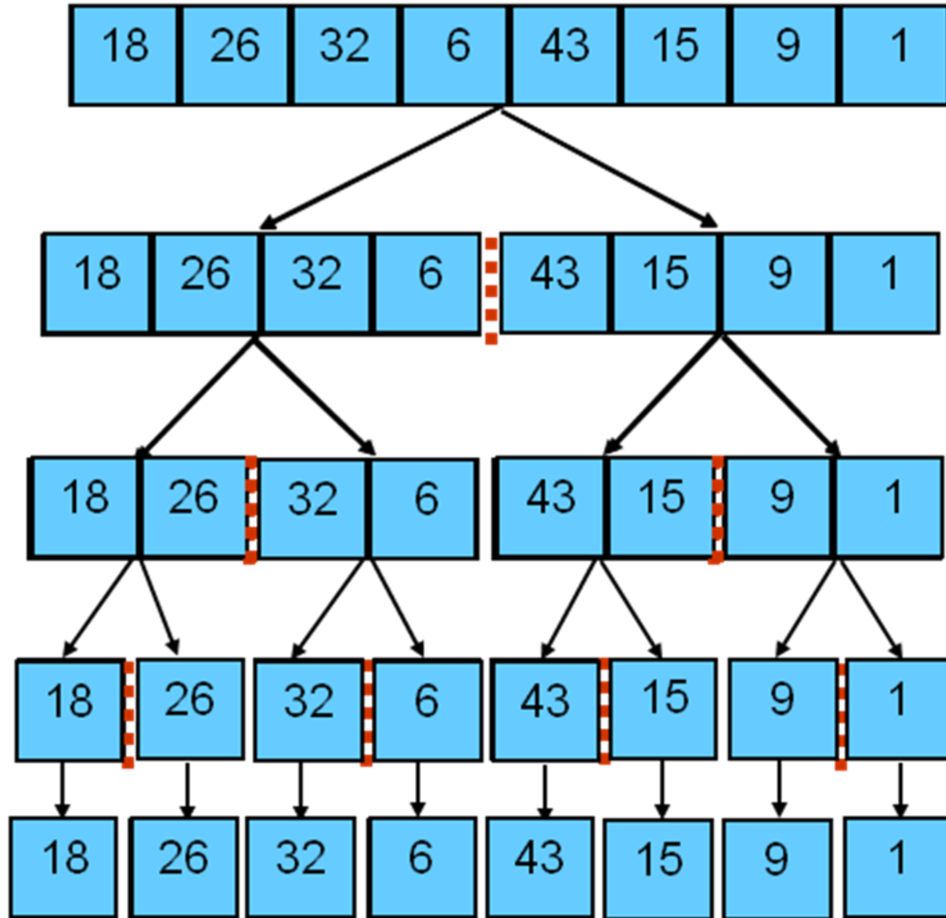


Fig.2: Division Process of left Subarray

The merge process of left subarray is shown below in Fig. 3.

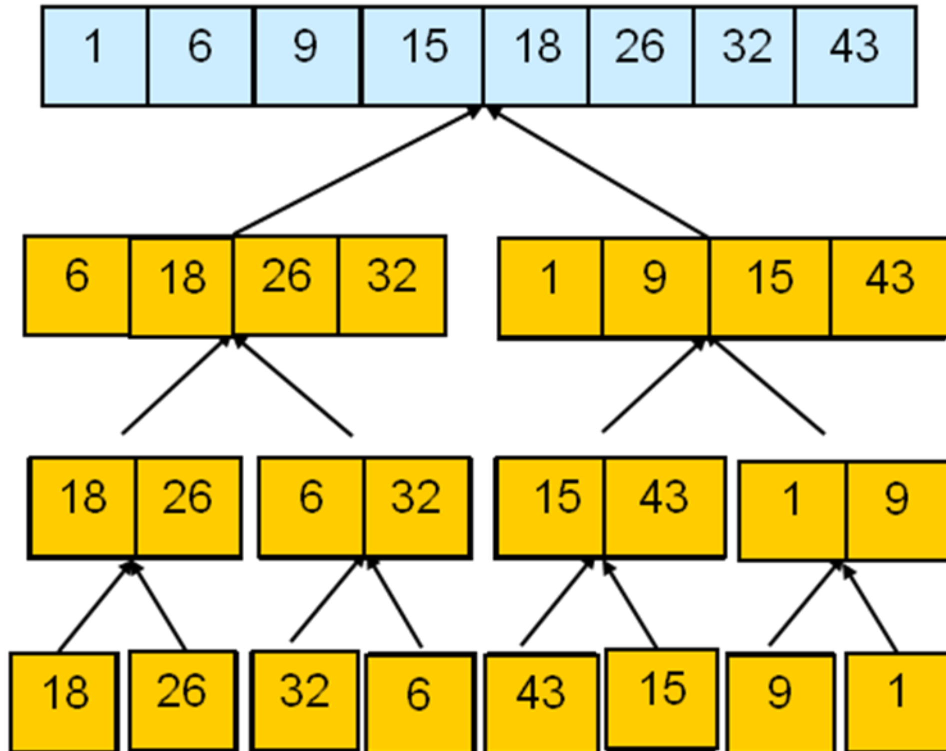


Fig.3: Division Process of left Subarray

Similarly this is repeated for right subarray as well.

- **Complexity analysis:**

If T(n) is the running time ***T*(*n*)** of Merge Sort, then division process for computing the middle takes

$\Theta(1)$, the conquering step , i.e, solving 2 subproblems takes $2T(n/2)$ and combining step, i.e., merging n elements takes $\Theta(n)$. In short, the recurrence equation for merge sort is given as follows:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n - 1 & \text{for n} \geq 2 \\ 1 & \text{for n} = 2 \\ 0 & \text{when n} < 2 \end{cases}$$

Or in short,

$$T(n) = \Theta(1) \;\; if\; n = 1$$

$$T(n) = 2T(n/2) + \Theta(n) \text{ if } n > 1$$

Solving this yields, the complexity of merge sort can be derived as :

$$\Rightarrow T(n) = \Theta(n \lg n)$$

2.3 Quicksort

C.A.R. Hoare in 1962 designed Quicksort in 1962. Quicksort uses divide and conquer as a strategy for sorting elements of an array. Merge sort divides the array into two equal parts. But Quick sort unlike merge sort does not divide the array into equal parts. Instead, uses a pivot element to divide the array into two equal parts.

The steps of Quicksort is given below:

1. ***Divide step:***

*Pick any element (**pivot**) p in S.* This is done using a partitioning algorithm. Then, the partitioning element, p*artition S – {p} into two disjoint groups*

$$S1 = \{x \in S - \{p\} \mid x <= \mathbf{p}\}S2$$

$$= \{x \in S - \{p\} \mid x \geq p\}$$

2. ***Conquer step:*** *recursively sort S1 and S2*
3. ***Combine step:*** *the sorted S1 (by the time returned from recursion), followed by p, followed by the sorted S2 (i.e., nothing extra needs to be done)*

The informal Quicksort algorithm is informally given as follows:

1. if left < right:

1.1. Partition a[left...right] such that:

all a[left...p-1] are less than a[p], and all

a[p+1...right] are >= a[p]

1.2. Quicksort a[left...p-1]

1.3. Quicksort a[p+1...right]

2. Combine the subarrays and Terminate

The formal algorithm of quicksort based on [1] is given as follows:

Algorithm quicksort(A, first, last)

```
%% Input: Unsorted array A [first..last]
][
%%  Output:  Sorted  array  A
    Begin
        if (first < last) then
            v  =  partition(A,first,last)   %%  find  the  pivot  element
            quicksort([A, first,v-1])
            quicksort([A,v+1,      last])
        end if
    end
```

It can be observed that the important phase of a quicksort algorithm is the partitioning stage where the given array is divided into two parts using a 'partition' procedure. While in mergesort, the middle element can be found directly. In quicksort, finding the middle element is

not a straight forward process. It is done using partition algorithms.

2.4 Partitioning Algorithms

Partitioning algorithms are used to divide the given array into two subarrays. It is complicated process in quicksort compared to the division process of merge sort. There are two partitioning algorithms. One is by Lomuto partitioning algorithm and another by Hoare.

Let us discuss about Lomuto algorithm.

2.4.1 Lomuto Algorithm

Lomuto is a one directional partition algorithm. It scans from left to right and checks for the elements. If the number is less than the pivotal elements, the numbers are swapped. The following example illustrates Lomuto algorithm.

Example 2:

Use Lomuto procedure to partition the following given array:

p i → i

60	93	57	90	10

s

A[i] **not less than** p

p i

60	93	57	90	10

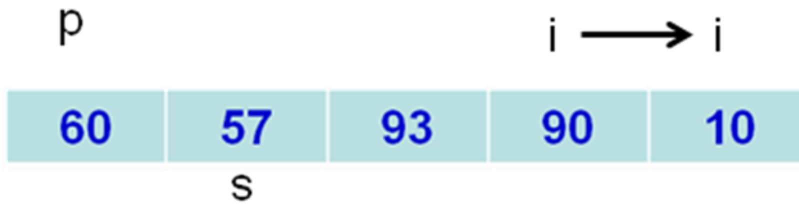
s → s

A[i] **less than** p

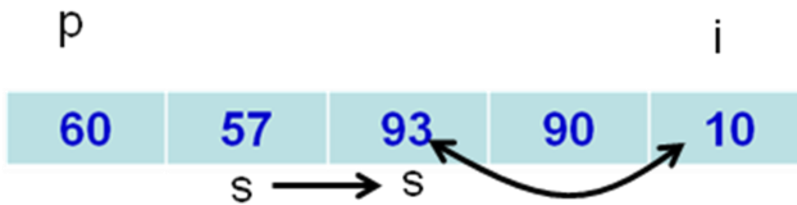
p i → i

60	57	93	90	10

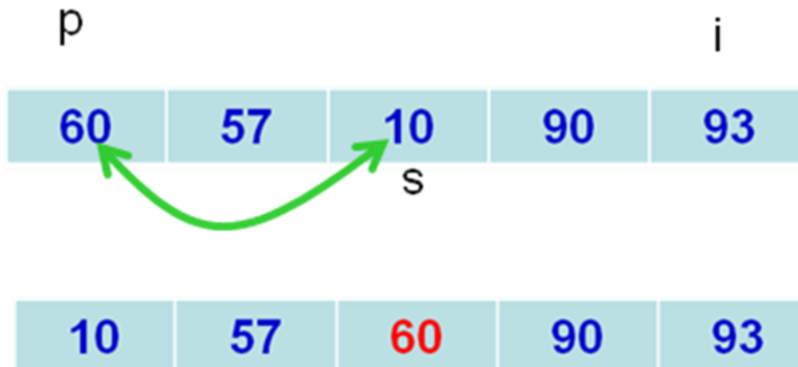
s



A[i] **not less than** p



A[i] **less than** p



It can be observed that all the elements of the left of 60 are less than 60 and all the elementson

the right hand side is greater than 60.

The formal algorithm based on [1] is given as follows:

ALGORITHM LomutoPartition(A[l..r])

//Partition subarray by Lomuto's algo using first element as pivot

//Input: A subarray A[l..r] of array A[0..n-1], defined by its //left and right indices l and r (l ≤r)

//Output: Partition of A[l..r] and the new position of the pivotp <-

A[l]

s <- l

for i <- l+1 to r do

```
                if A[i] < p s

                <- s+1

                swap(A[s], A[i])

    swap(A[l],  A[s])

    return s
```

2.5 Hoare Algorithm

Another useful partition algorithm is called Hoare partition algorithm. This algorithm has two scans. one scan is from left-to-right and another scan is from right-to-left. The left to-right scan (using pointer *i*) aims to skip the smaller elements compared to the pivot and stop when an element is ≥ the pivot. Then right-to left scan (using pointer *j*) starts with the last element and skips over the elements that are larger than of equal to the pivot element. If *i < j,* in that case *A[i]* and *A[j]* are swapped and the process is continued with

the increment of i and decrement of j pointers. If one encounters the situation *i > j,* then the pivot element is swapped with *A[j].*

The Hoare partition algorithm is given informally as follows:

- Choose pivot element from the array A, generally the first element
- Search from left to right looking for elements <u>greater than</u> pivot.
- Search from right to left looking for elements <u>smaller than</u> pivot.
- When two elements are found, exchange them.
- When two elements cross, exchange pivot element such that it is in final place.
- Return the pivot element

Formally, the Hoare partition algorithm is given as follows:

Algorithm Hoare_partition (A,first,last)

%% Input: Array A with elements 1 to n. First = 1 and last = n

%%Output: Sorted array A

```
Begin

%% First Element is the initial pivot

pivot = A [first]

%% Initialize the pointersi =

first+1

j =last

flag    =    false

predicate = true

While (predicate) do

    while (i ≤ j ) and (A[i] ≤ pivot) doi = i

          + 1

    End while

   while (j ≥ pivot and j ≥ i) doj =

         j-1

   End while

  if (j < i)

    break

  else

    A[i]   ↔A[j]

  End if

End while A[first]

↔ A[j]return j

End
```

It can be observed that the algorithm initializes two pointers i and j and initial pivot. The pointers are updated based on the conditions that are discussed above as an informal procedure.

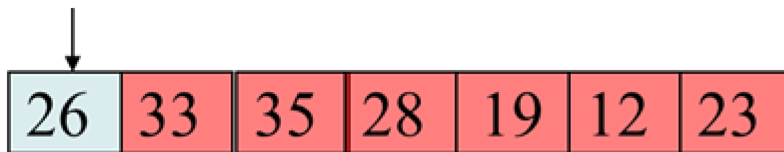
The following example illustrates the application of Hoare partition to a given array.

Example 3 : **Apply** the Hoare portioning algorithm for the following array:

26,33,35,28,19,12,23.

To apply Hoare partition algorithm, the following steps are used:

Step 1: Start with all data in an array, and consider it unsorted



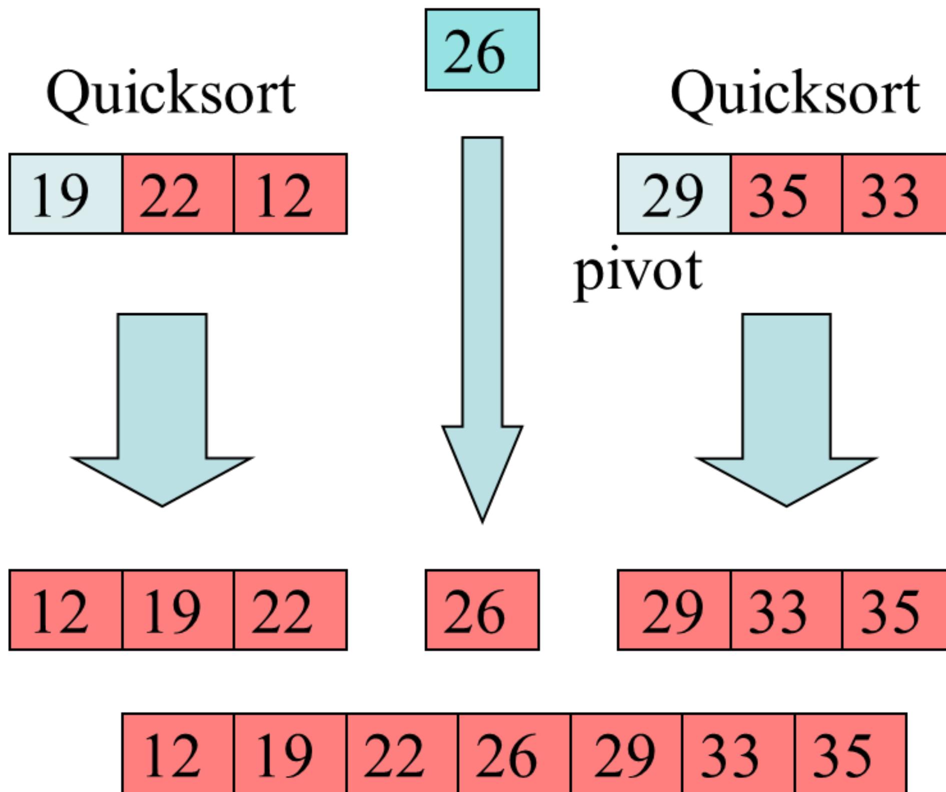
Step 2: Step 1, select a pivot (it is arbitrary), Let it be first element

Step 2, start process of dividing data into LEFT and RIGHT groups. The LEFT group will have elements less than the pivot and the RIGHT group will have elements greater that the pivot.

Step 3: If left element belongs to LEFT group, then left = left + 1. If right index element, belongs to

RIGHT, then right = right – 1. Exchange the elements if they belong to the other group.

The final steps are shown below:



- **Complexity analysis of Quicksort**

Quicksort is an effective and popular algorithm and its complexity analysis is given below:

- **Best Case Analysis:** The best case quicksort is a scenario where the partition element is exactly in the middle of the array. The best case quicksort is when the pivot partitions the list evenly. The resulting partitions of a best case are well balanced. Thus, the recurrence equation is given

below:

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

Using the master's theorem, the complexity of the best case turns out as $T(n) \in \theta(n \log n)$

It can also be derived as follows:

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c\log N \\
T(N) &= cN\log N + N = O(N\log N)
\end{aligned}
$$

Worst Case Analysis: In the worst case, it can be observed that the partitions are no longerbetter than a linear list. This happens because the first element is always the pivot element. Hence, there is no element in the left hand side.

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

∴ The overall size of the tree is given as

$1 + 2 + \cdots + (n-1) + n$

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

$\theta\left(n^2\right)$.

Thus, the worst case complexity of quicksort is $\theta\left(n^2\right)$.

Check Your Progress

Fill in the Blanks.

Q.1: Merge sort divides the array into ______equal parts.

Q.2: The worst case complexity of quicksort is_______________.

Multiple Choice Questions.

Q.3: Question: What is the primary advantage of Merge Sort over other sorting algorithms like Quick Sort?

A) It has a shorter average-case time complexity.

B) It is an in-place sorting algorithm.

C) It guarantees a worst-case time complexity of O(n log n).

D) It has a smaller memory footprint.

Answer: C.

Q.4: Which data structure is typically used to implement Quicksort?

A) Linked List

B) Binary Tree

C) Stack

D) Array

Q.5: In the context of sorting algorithms, what is the primary purpose of a partitioning algorithm?

A) To combine multiple sorted arrays into a single sorted array.

B) To divide an array into two or more subarrays based on a pivot element.

C) To identify and remove duplicates from an array.

D) To count the number of elements in an array.

Q.6: What is the time complexity of the Merge Sort algorithm for sorting n elements in the worst case?

A) O(n)

B) O(n log n)

C) O(n^2)

D) O(log n)

2.6 Answer to Check Your Progress

Ans 1. **Two**

Ans 2. $\theta\left(n^2\right)$

Ans 3. **It guarantees a worst-case time complexity of O(n log n)**

Ans 4. **Array**

Ans 5. **To divide an array into two or more subarrays based on a pivot element.**

Ans 6. **O(n log n)**

3.1 What is divide and Conquer design paradigm?

Divide and conquer is a design paradigm. It involves the following three components:

Divide: In this step, the problem is divided into sub-problems. It must be noted that the sub-problemsare similar to the original problem but smaller in size.

Conquer: In this step, the sub-problems are solved iteratively or recursively. If the problems are smallenough, then they are solved in a straightforward manner.

Combine: In this step, the solutions of the subproblems are combined to create a solution to the originalproblem.

3.2 Multiplication of Long Integers

One study the concept of multiplication in early school. One can note that, the multiplication of an *n*-digit number *X* by a single digit is called short multiplication. The concept of multiplying n-digit *X* with another *n*-digit number *Y* is called long multiplication.

To illustrate this, consider the simple example of multiplication of two digit numbers:

$$
\begin{array}{c}
(10)_2 \times (11)_2 \\
\hline
00 \\
11 \\
\hline
110 \\
\hline
\end{array}
$$

This is a traditional multiplication. How much effort is required to multiply, in general, two n- digit numbers? Andrey Kolmogorov is one of the brightest Russian mathematicians of the 20th century. He stated in 1960 that two n-digit numbers can't be multiplied with less than n^2 multiplications! One can go further and observe that if an *n*-digit number is to be multiplied with a single digit, then $2n$ multiplications are required. Similarly, a long multiplication involving two *n*-digit

numbers requires $n(2n) = 2n^2$ basic multiplication operations. The number of additions would be

$(n - 1) \times 2n =$ $2n^2 - 2n$ number of basic operations. Put together, one requires a total $4n^2 - 2n$

basic operations to carry out multiplication. In general, It can be summarized that

- Addition of two n-digit numbers, *a* and *b*, requires $\Theta(n)$ bit operations.

Multiplication of two *n*-bit integers *a* and *b*, *requires* $\Theta(n^2)$ bit operations.

In short, brute force multiplication of two n-digit numbers requires $\Theta(n^2)$ time. Most of the scientific applications require multiplication of long integers. So, there is a need for faster multiplication. In 1962, two Russian mathematicians Anatoly Alexeevitch Karatsuba and Yu Ofman published a paper titled "*Multiplication of multi-digit numbers on automata' that describes a* method for long multiplication using the divide-and-conquer approach that works faster. How is it done?

- **Multiplication of Long integers using Divide and Conquer paradigm:**
  - **Divide:** Divide the *n*-element into two subsequences of *n/2* elements each.
  - ***Conquer:*** Multiply the two subsequences recursively
  - ***Combine*:** Combine the two multiplied subsequences appropriately to produce the finalanswer.

Let us illustrate it as follows:

To perform integer multiplication faster, the distributive law is used to divide the sequence of n-element into two subsequence of n/2 elements each. For example, consider the following two numbers of two digits: $u = 78$ and $v = 33$.

One can split the numbers as follows:

$$u = 78 = 7 \times 10 + 8$$
$$v = 33 = 3 \times 10 + 3$$

In general, one can generalize this as follows:

$$u = x \times 10 + y$$
$$v = w \times 10 + z$$

It can be observed that for the given problem x = 7, y = 8, w = 3, and z = 3. It can be observed that, the multiplication of two numbers u and v can be done as follows using the distributive law:

$$u \times v = (x \times 10 + y) \times (w \times 10 + z)$$
$$= \text{xw} \times 10^2 + 10 \times yw + 10 \times xz + yz$$
$$= \text{xw} \times 10^2 + (yw + xz) \times 10 + yz$$

One can verify the result by substituting the values of x, y, w, and z in this equation to get theresult:

$$u \times v = \text{xw} \times 10^2 + (yw + xz) \times 10 + yz$$
$$= 7 \times 3 \times 10^2 + (8 \times 3 + 7 \times 3) \times 10 + 8 \times 3$$
$$= 2574$$

It can be verified that the answer is correct by the conventional multiplication.

- **Informal Algorithm**

The informal Karatsuba algorithm is given as follows:

Step 1: Divide the long digits x and y recursively.

Step 2: Express long integers as polynomials.

Step 3: Reuse and combine the terms to compute the product of long integers.

Step 4: Return the result.

- **Computational complexity**

The algorithm involves 4 multiplications of n/2-bit numbers plus 3 additions. Hence , thecomplexity of this approach is given as follows:

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \qquad \text{for } n > 1,\ n \text{ is a power of } 2$$

$$T(1) = 0$$

Here, c is a constant and n indicates all the linear-time operations such as multiplication, addition and taking power operations of the algorithm. The solution of this recurrence leads to $O(n^2)$, which amounts to the same complexity as that of the conventional approach.

Gauss idea: of Carl Friedrich Gauss who proved that the product of two complex numbers $(a + bi)(c + di) = ac - bd - (bc + ad)i$ is equivalent to $(a + b)(c + d) - ac - bd$ or $ac + bd - ((a-b)(c-d))$. It can be observed that the ac and ad need not be computed and can be reused. Karatsuba used this idea independently to prove that three multiplications are enough to solve long-integer multiplications.

Thus, Karatsuba algorithm effectively reduces four multiplications to three multiplications for performing multiplication. This reduction of multiplication reduces computational complexity. Thus Karatsuba algorithm is thus a generalization of the aforementioned idea. Thus, the multiplication of two n-digits numbers as follows:

$$u = x \times 10^m + y, \text{ where } m = n/2.$$

$$v = w \times 10^m + z$$

Therefore, the product of the numbers u and v is as follows:

$$u \times v = \left(x \times 10^m + y\right)\left(w \times 10^m + z\right)$$

$$= xw \times 10^{2m} + \left(xz + yw\right) \times 10^m + yz$$

Using the idea of Gauss, say, the product p, $p = xz + yw$ can be written as follows: $p = (x + y)(z + w) - xw - yz$. It can be observed that xw and yz are already known and need not be computed. Instead, the values of xw and yz can be reused. If $p_1 = x \times w$, $p_2 = y \times z$; hence, $p_3 = [(x + y)(z + w) - p_1 - p_2]$. This gives the formal algorithm of Karatsuba.

- **Formal algorithm:**

The formal Karatsuba algorithm is given as follows:

Algorithm multiply(x, y)

%% Input: x and y are long integers

%%Output: Product of x and y

Begin

Split x, y into halves as follows:

$$u = 10^{\frac{n}{2}} x + y$$

$$v = 10^{\frac{n}{2}} w + z$$

p_1 = multiply(x, w)p_2

= multiply(y, z)

p_3 = multiply($x + y$, $z + w$)z

$= p_3 - p_1 - p_2$

$$T = 10^n p_1 + z \times 10^{\frac{n}{2}} + p_2$$

return(T)

End if

End

The preceding algorithm is illustrated with the help of the following numerical example:

Example 1: **M**ultiply two four-digit numbers u = 2345 and v = 5678 using the Karatsubamethod.

Solution

Let us apply divide and conquer. The digits are of length 4 (i.e., n = 4). Let us divide this into $n/2$ problems. The digits u and v can now be expressed as follows:

$p_1 = x \times w = 23 \times 56 = 1288$

$$u = x \times 10^2 + y = 23 \times 10^2 + 45$$

$$v = w \times 10^2 + z = 56 \times 10^2 + 78$$

Now, one can observe that $p_1 = x \times w$ and $p_2 = y \times z$. In other words, $p_1 = x \times w = 23 \times 56 = 1288$

$p_2 = y \times z = 45 \times 78 = 3510$. Using the karatsuba idea, one can rewrite $yw + xz$ as follows:

$$P_1 = [(x + y)(z + w) - p_1 - p_2]$$

$$= (23 + 45)(56 + 78) - p_1 - p_2$$

$$= 68 \times 134 - 1288 - 3510$$

$$= 4314$$

Substituting all the values into this equation, one gets the following set of equations:

$$\therefore u \times v = p_1 \times 10^n + (p_3 - p_1 - p_3) \times 10^{\frac{n}{2}} + p_2$$

$$= 1288 \times 10^4 + (4314) \times 10^2 + 3510$$

$$= 13314910$$

One can verify the correctness of the result by comparing this value with the product of 2345 and 5678 computed using the conventional multiplication method.

- **Complexity Analysis:**

The Karatsuba method reduces the multiplication operations by 1. This leads to the following recurrence equation:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 3T(n/2) + cn & \text{otherwise} \end{cases}$$

Here c is constant and cn represents the linear-time operations such as multiplication, addition and taking power operations of the algorithm. By solving this, one can observe the complexity of this reduces to $3^{\log n} = n^{\log 3} = n^{1.58}$, compared to the traditional algorithm complexity of O(n^2).

The improvement in performance of Karatsuba [4] algorithm over the standard conventional algorithm for a large value of 'n is shown in Fig 1.

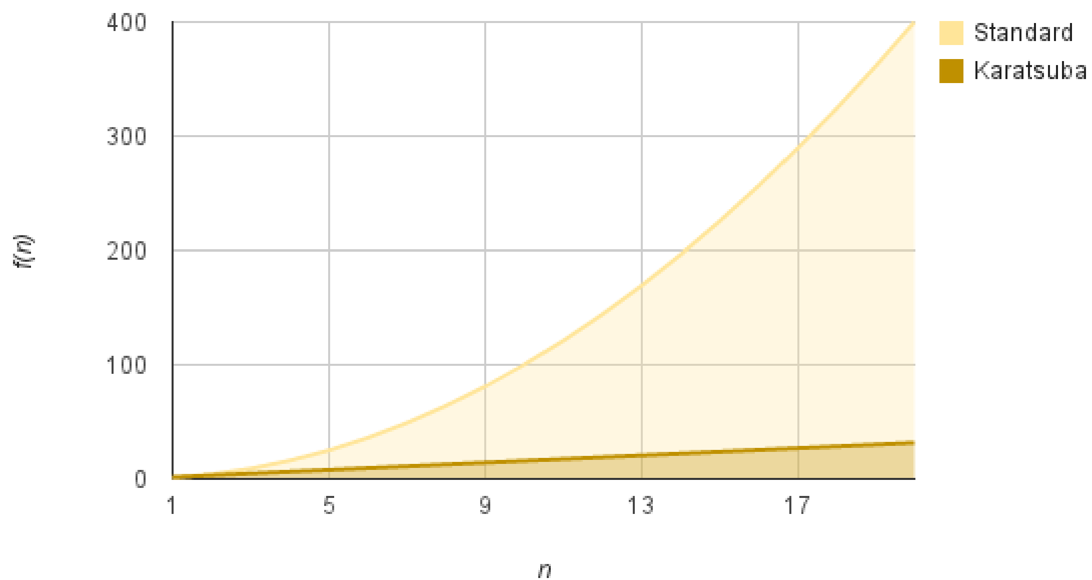


Fig 1: performance of Karatsuba algorithm over Standard algorithm

3.3 Strassen Matrix multiplication Algorithm

Most of the scientific applications use matrix multiplication. If there are two matrices of *A* and *B* of dimensions $n \times n$, the matrix multiplication of these two matrices, $C = A \times B$, then the elements of the resultant matrix C_{ij} are given as follows:

$$C_{ij} = \sum^{n} A_{ik} \times B_{kj}$$

$k = 1$

The variables *i* and *j* represent the rows and columns of the given matrix.

In other words, if two matrices *A* and *B* of order 2×2, then the resultant matrix *C* obtained by

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00} * B_{00} + A_{01} * B_{10} & A_{00} * B_{01} + A_{01} * B_{11} \\ A_{10} * B_{00} + A_{11} * B_{10} & A_{10} * B_{01} + A_{11} * B_{11} \end{bmatrix}$$

The conventional algorithm is given as follows:

```
Algorithm MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i, j] ← 0.0
        for k ← 0 to n-1 do
            C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

- **Complexity analysis:**

It can be observed that eight multiplication and four addition/subtraction operations are required. The time complexity of the traditional matrix multiplication algorithm is $O(n^3)$.

- **Divide and Conquer algorithm:**

The general recurrence equation is given as follows:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \ldots \left(\frac{n}{-}\right) + \Theta \ldots^{2} & \text{if } n \geq 2 \end{cases}$$

$if\ n = 1$

Here, $\Theta(n^2)$

indicates the matrix additions/subtractions required to be performed on n/2 x n/2

matrices. Therefore, the complexity of the algorithm is given as $\Theta(n^3)$.

3.4 Strassen Matrix Multiplication

Volker Strassen is a German mathematician born in 1936. He is well known for his works onmatrix

multiplication that outperforms the general matrix multiplication algorithm. He reduced

the number of multiplications from 8 to 7 using algebraic techniques. The reduction of onemultiplication

provides a faster matrix multiplication algorithm.

Using the Strassen algorithm, the multiplication of 2 $\times$ 2 matrices *A* and *B* to yield matrix *C* can be

carried out using seven multiplications, with the help of the following formulas:

- $d_1=(A_{00}+A_{11})*(B_{00}+B_{11})$
- $d_2=(A_{10}+A_{11})*B_{00}$
- $d_3=A_{00}*(B_{01}-B_{11})$
- $d_4=A_{11}*(B_{10}-B_{00})$
- $d_5=(A_{00}+A_{01})*B_{11}$
- $d_6=(A_{10}-A_{00})*(B_{00}+B_{01})$
- $d_7=(A_{01}-A_{11})*(B_{10}+B_{11})$

Then the elements of the resulting product matrix *C* are given as follows:

$C_{11} = d_1 + d_4 - d_5 + d_7$

$C_{12} = d_3 + d_5$

$C_{21} = d_2 + d_4$

$C_{22} = d_1 + d_3 - d_2 - d_6$

The resultant matrix *C* is given as follows:

$$C = \begin{pmatrix} d_1 + d_4 - d_5 + d_7 & d_3 + d_5 \\ d_2 + d_4 & d_1 + d_3 - d_2 - d_6 \end{pmatrix}$$

The idea of Strassen is given as follows: for example, the element $c_{21} = d_2 + d_4$ can be computed.

Here, $d_4 = a_{22} * (b_{21} - b_{11})$ and $d4 = (a_{21} + a_{22}) * b_{11}$. This can be seen as equal toof the $c_{21} = a_{21}b_{11} + a_{22}b_{21}$ traditional matrix multiplication.

Strassen matrix multiplication is illustrated in the following numerical example:

Example 2: Multiply the following two matrices using the Strassen method:

$$A = \begin{pmatrix} 2 & 5 \\ 5 & 2 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Solution

Here $a_{11} = 2$, $a_{12} = 5$, $a_{21} = 5$, $a_{22} = 2$, $b_{11} = 1$, $b_{12} = 0$, $b_{21} = 0$, and $b_{22} = 1$. These values can besubstituted in the following set of equations:

substituted in the following set of equations:

$$d_1 = (A_{11} + A_{22})(B_{11} + B_{22}) = (2+2)(1+1) = 8$$
$$d_2 = (A_{21} + A_{22})B_{11} = (5+2)(1) = 7$$
$$d_3 = A_{11}(B_{12} - B_{22}) = 2 \times (0-1) = -2$$
$$d_4 = A_{22}(B_{21} - B_{11}) = 2 \times (0-1) = -2$$
$$d_5 = (A_{11} + A_{12})B_{22} = (2+5)1 = 7$$
$$d_6 = (A_{21} - A_{11})(B_{11} + B_{12}) = (5-2)(1+0) = 3$$
$$d_7 = (A_{12} - A_{22})(B_{21} + B_{22}) = (5-2)(0+1) = 3$$

On substituting these values we get the following equations:

$$C_{11} = d_1 + d_4 - d_5 + d_7 = 8-2-7+3=2$$
$$C_{12} = d_3 + d_5 = -2+7=5$$
$$C_{21} = d_2 + d_4 = 7-2=5$$
$$C_{22} = d_1 + d_3 - d_2 + d_6 = 8 - 2 - 7 + 3 = 2$$

Therefore, the resultant matrix would be as follows:

$$C = \begin{pmatrix} 2 & 5 \\ 5 & 2 \end{pmatrix}$$

Divide and Conquer method:

Thus, the informal algorithm for Strassen matrix multiplication is given as follows:

Step 1: Divide a matrix of order $n \times n$ recursively till matrices of 2×2 order are obtained.

Step 2: Use the previous set of formulas to carry out 2×2 matrix multiplication.

Step 3: Combine the results to get the final product matrix.

- **Formal Algorithm**

The formal Strassen algorithm is given as follows:

Algorithm Strassen(n, A, B, k)

%%Input: Matrices *A* and *B*; *n* is the order and *k* is a temporary matrix

%%Output: Matrix *C*, which is the product of matrices *A* and *B*

Begin

if n = threshold, then compute

$C = A \times B$ in a conventional manner

else

partition A into four submatrices A_{11}, A_{12}, A_{21}, A_{22}

partition B into four submatrices B_{11}, B_{12}, B_{21}, B_{22}

strassen($n/2$, $A_{11} + A_{22}$, $B_{11} + B_{22}$, d_1)

strassen($n/2$, $A_{21} + A_{22}$, B_{11}, d_2) strassen($n/2$, A_{11}, $B_{12} - B_{22}$, d_3) strassen($n/2$, A_{22}, $B_{21} - B_{11}$, d_4) strassen($n/2$, $A_{11} + A_{12}$, B_{22}, d_5) strassen($n/2$, $A_{21} - A_{11}$, $B_{11} + B_{12}$, d_6) strassen($n/2$, $A_{12} - A_{22}$, $B_{21} + B_{22}$, d_7)

end if %% Combine to get matrix C

$$C = \begin{bmatrix} d_1 + d_4 - d_5 + d_3 & d_3 + d_5 \\ d_2 + d_4 & d_1 + d_3 - d_2 + d_6 \end{bmatrix}$$

return(C)

End

It can be observed that in the division part, the given matrices are divided recursively. Then, the set of Strassen matrix multiplication formulas are used to compute the product. Next, they are combined to get the final result.

Example 3 Perform conventional and Strassen multiplication for the following two matrices:

$$A = \begin{pmatrix} 2 & 4 & 6 & 3 \\ 1 & 2 & 2 & 1 \\ 3 & 1 & 1 & 3 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix}$$

$$\begin{matrix} 2 & 1 & 1 & 2 \\ 3 & 1 & 1 & 3 \end{matrix}\Bigg)$$

Solution

By the conventional method, the multiplication of these two matrices yields the followingresults:

$$A \times B = \begin{pmatrix} 2 & 4 & 6 & 3 \\ 1 & 2 & 2 & 1 \\ 3 & 1 & 1 & 3 \\ 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 2 \\ 3 & 1 & 1 & 3 \end{pmatrix}$$

$$= \begin{bmatrix} 2+4+12+9 & 2+4+6+3 & 2+4+6+3 & 2+8+12+9 \\ 1+2+4+3 & 1+2+2+1 & 1+2+2+1 & 1+4+4+3 \\ 3+1+2+9 & 3+1+1+3 & 3+1+1+3 & 3+2+2+9 \\ 1+1+2+3 & 1+1+1+1 & 1+1+1+1 & 1+2+2+3 \end{bmatrix}$$

$$= \begin{bmatrix} 27 & 15 & 15 & 31 \\ 10 & 06 & 06 & 12 \\ 15 & 08 & 08 & 16 \\ 07 & 04 & 04 & 08 \end{bmatrix}$$

By the Strassen method, a 4 × 4 matrix can be divided into four halves as follows:

$$A = \left(\begin{array}{cc|cc} 2 & 4 & 6 & 3 \\ 1 & 2 & 2 & 1 \\ \hline 3 & 1 & 1 & 3 \\ 1 & 1 & 1 & 1 \end{array}\right) \text{ with } A_{11} = \begin{pmatrix} 2 & 4 \\ 1 & 2 \end{pmatrix} A_{12} = \begin{pmatrix} 3 & 1 \\ 1 & 1 \end{pmatrix} \quad A_{21} = \begin{pmatrix} 6 & 3 \\ 2 & 1 \end{pmatrix} A_{22} = \begin{pmatrix} 1 & 3 \\ & \end{pmatrix}$$

$$B = \left(\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ \hline 2 & 1 & 1 & 2 \\ 3 & 1 & 1 & 3 \end{array}\right) \text{ with } B_{11} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} B_{12} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \quad B_{21} = \begin{pmatrix} 2 & 1 \\ 3 & 1 \end{pmatrix} b_{22} = \begin{pmatrix} 1 & 2 \\ & \end{pmatrix}$$

It can be observed that the elements are matrices themselves. Substitute these values in the following set of equations:

$$d_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

It can be observed that the elements are matrices themselves. Substitute these values in thefollowing set

of equations:

$$= \begin{bmatrix} 20 & 37 \\ 10 & 18 \end{bmatrix}$$

$$d_2 = (A_{21} + A_{22}) \times b_{11}$$

$$= \begin{bmatrix} 8 & 8 \\ 4 & 4 \end{bmatrix}$$

$$d_3 = A_{11} \times [\mathrm{B}_{12} - \mathrm{B}_{22}]$$

$$= \begin{bmatrix} 0 & -6 \\ 0 & -3 \end{bmatrix}$$

$$d_4 = A_{22} \times [B_{21} - B_{11}]$$

$$= \begin{bmatrix} 7 & 0 \\ 3 & 0 \end{bmatrix}$$

$$d_5 = [A_{11} + A_{12}] \times B_{22}$$

$$= \begin{bmatrix} 15 & 37 \\ 6 & 15 \end{bmatrix}$$

$$d_6 = [A_{21} - A_{11}] \times [B_{11} + B_{12}]$$

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

$d_7 = [A_{12} - A_{22}] \times [B_{21} + B_{22}]$

$$= \begin{bmatrix} 15 & 15 \\ 3 & \end{bmatrix}$$

Therefore, the product matrix is given as follows:

$$AB = \begin{bmatrix} d_1 + d_4 - d_5 + d_7 & d_3 + d_5 \\ d_2 + d_4 & d_1 + d_3 - d_2 + d_6 \end{bmatrix}$$

One can verify that the resultant matrix is $\begin{bmatrix} 27 & 15 & 15 & 31 \\ 10 & 06 & 06 & 12 \\ 15 & 08 & 08 & 16 \\ 07 & 04 & 04 & 08 \end{bmatrix}$ by comparing it with the resultant matrix obtained using the conventional method.

- **Complexity Analysis**

Thus, it can be observed that Strassen used only seven multiplications instead of eight but incurred more additions/subtractions. This reduction in multiplication helps multiply the matrices faster. The Strassen technique can also be combined effectively with the divide-and-conquer strategy. The following is the informal procedure of the Strassen matrix multiplication:

The recurrence equation for Strassen matrix is given as follows:

$$T = \begin{cases} 0 & \text{if } n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Here, $\Theta(n^2)$ indicates all the matrix additions/subtractions involved on n/2 x n/2 matrices. One

can solve this recurrence equation using the master theorem as follows:

$$T(n) = 7^{\log n} = n^{\log 7} \approx n^{2.807}$$ Thus,

the complexity of Strassen matrix multiplication is $\Theta\left(n^{2.81}\right)$.

Performance:

It can be observed that the performance of Strassen matrix multiplication algorithm as shown inFig. 2.

Fig 2: Performance of Strassen Matrix multiplication over Standard o(n^3) algorithm

3.5 *Summary*

- Brute force guarantee solutions but it is inefficient.
- It is difficult to solve combinatorial optimization problems.
- 15 Puzzle, 8-Queen, Knapsack and assignment problems are optimization problem that can be solved using brute force method.
- Divide and Conquer often leads to a better solution.
- Merge sort uses divide and conquer technique and sorts the elements in O(nlogn) time.
- Quicksort uses divide and conquer strategy and sorts the elements in o(nlogn) time.
- Master Theorem is helpful in solving recurrence equations.
- In short, one can conclude as part of this module 11 that
- Divide and Conquer often leads to a better solution.
- Karatsuba and Strassen methods use divide and conquer technique.
- Master Theorem is helpful in solving recurrence equations.

Check Your Progress
Multiple Choice Questions.

Q.1: When multiplying long integers using the standard multiplication algorithm, what is the time complexity for multiplying two n-digit numbers?
A) O(n)
B) O(n^2)
C) O(n^3)
D) O(2^n)

Q.2: What is the primary advantage of the Strassen Matrix Multiplication Algorithm over traditional matrix multiplication methods?

A) It has a lower space complexity.
B) It always has a faster time complexity.
C) It reduces the number of multiplicative operations.
D) It is easier to implement.

Q.3: In the Strassen Matrix Multiplication Algorithm, how are two matrices divided in each recursive step?
A) They are divided into equal-sized submatrices.
B) They are divided into submatrices of different sizes.
C) They are divided into triangular matrices.
D) They are divided into diagonal matrices.

3.6 Answer to Check Your Progress

Ans 1. O(n^2)
Ans 2. It reduces the number of multiplicative operations.
Ans 3. They are divided into submatrices of different sizes.

3.7 References

1. *S.Sridhar , Design and Analysis of Algorithms , Oxford University Press, 2014.*
2. *A.Levitin, Introduction to the Design and Analysis of Algorithms, Pearson Education, New Delhi, 2012.*
3. T.H.Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA 1992.
4. http://www.stoimen.com/blog/2012/05/15/computer-algorithms-karatsuba-fast- multiplication/
5. http://www.stoimen.com/blog/2012/11/26/computer-algorithms-strassens-matrix- multiplication/
6. URL: http://www.hbmeyer.de/backtrack/achtdamen/eight.htm#up

3.8 Model Questions

1. What is 15-Puzzle game and 8-Queen Problem?
2. Explain Knapsack Problem and assignment Problem with example.
3. What do you understand by the concept of Divide and Conquer technique with example of Merge Sort algorithm or Quicksort Algorithm.
4. How long integer multiplication can be done using divide and conquer design paradigm?
5. Explain Matrix Multiplication.

6. Explain Strassen Multiplication Algorithm for multiplying matrices faster.

BLOCK 2
UNIT 6: Divide and Conquer Technique

1.1 Learning Objective

1.2 Closest pair and Convex Hull Problems using Divide and Conquer

1.2.1 Closest Pair problem

1.2.2 Convex Hull

1.2.3 Quick Hull

1.2.4 Merge Hull

1.3 Answer to Check Your Progress

2.1 Applications of Divide and Conquer

2.1.1 Finding Maximum and Minimum Elements

2.2 Tiling Problem

2.2.1 Fourier Transform

2.2.2 Polynomial Multiplication

2.3 Answer to Check Your Progress

3.1 Introduction to Decrease and Conquer Design paradigm

3.2 Categories of Decrease and Conquer Design paradigm

3.1.1 Decrease by a constant

3.1.1.1 Insertion Sort

3.1.1.2 Topological Sorting

3.1.1.3 Permutations

3.1.1.4 Johnston–Trotter Algorithm

3.1.2 Decrease by a constant factor

3.1.3 Decrease by a variable factor

3.3 Summary

3.4 Answer to Check Your Progress

3.5 References

3.6 Model Questions

1.1 Learning Objectives

The Learning objectives of this unit are as follows:

- To understand closest pair problem using divide and conquer strategy
- To know about Convex, Quick and Merge Hull algorithm
- To find minimum and maximum in an array using divide and conquer
- To understand the use of divide and conquer for Tiling problem
- To implement Fast Fourier Transform and polynomial multiplication problem
- To understand decrease and conquer paradigm
- To understand Insertion and topological Sort
- To understand Permutations and Subsets
- To know the algorithms for generation of Permutations and Subsets

1.2 Closest pair and Convex Hull Problems usingDivide and Conquer

What is divide and Conquer design paradigm?

Divide and conquer is a design paradigm. It involves the following three components:

Step 1: (Divide) The problem is divided into subproblems. It must be noted that the subproblems are similar to the original problem but smaller in size.

Step 2: (Conquer) after division of the original problem into subproblems, the sub-problems are solved iteratively or recursively. If the problems are small enough, then they are solved in a straightforward manner.

Step 3: (Combine) Then, the solutions of the subproblems are combined to create a solution to the original problem

1.2.1 Closest Pair problem

What is a closest pair problem? The problem can be stated as follows: Given a set of points in the plane, find closest pair of points. The points can be a city, transistors on a circuit board, or computers ina network.

So the aim of the closest pair problem is to find the closest pair among *n* points in 2-dimensional space. This requires finding the distance between each pair of points and identifies the pair that gives the shortest distance.

Formally stated, the problem is given a set P of 'N' points, find points p and q, such that the d(p,q), distance between points p and q , is minimum.

- **Brute force method:**

The simplest brute force algorithm approach is to find distances between all points and finding the pair where the distance is minimum. What is a distance? A distance is a measure of closeness. There are many types of distances. A Euclidean distance between two points p(x, y) and q(s,t) is given as follows:

$$D_e(p,q) = [(x - s)^2 + (y - t)^2]^{1/2}$$

The distance measure should satisfy the following criteria to qualify as a metric. They are listedas follows:

1. D(A,B) = D(B,A)

   This property is called *symmetry* property

2. D(A,A) = 0

   This property is called Constancy of Self-Similarity

3. D(A,B) >= 0

   This property is called positivity

4. $D(A,B) \leq D(A,C) + D(B,C)$

   This property is called Triangular Inequality

- **Informal Algorithm**

The informal brute force algorithm [1] is given as follows:

For each point i∈S and

another point j∈S , {points i and j are distinct}

compute distance of i, j

if distance of i, j < minum distance then update

min_dist = distance i, j

return min_dist

- **Complexity analysis:**

The computational complexity of brute force approach is given based on [2,3] as

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2\sum_{i=1}^{n-1}(n-i)$$

$$= 2((n-1)+(n-2)+ \ldots +1] = 2\times \frac{n(n-1)}{2}$$

$$= n(n-1) \in \Theta(n^2)$$

Therefore, the complexity of the algorithm is $\Theta(n^2)$. Now, the objective is to solve the sameproblem with reduced number of computations using the divide-and-conquer strategy.

- ## Divide and Conquer strategy:

The divide and conquer can be given informally as follows:

– If trivial (small), solve it "brute force"
– Else

  - **divide** into a number of sub-problems
  - **solve** each sub-problem recursively
  - **combine** solutions to sub-problems

Based on divide and conquer, closest pair problem can be solved. The informal algorithm isgiven as follows:

- ## Informal Algorithm

When n is small, use simple solution.

When n is large

- Divide the point set into two roughly equal parts A and B.
- Determine the closest pair of points in A.
- Determine the closest pair of points in B.
- Determine the closest pair of points such that one point is in A and the other in B.
- From the three closest pairs computed, select the one with least distance.

The division of the points is given as follows:
Initially, n points of a set S are sorted based on x-coordinate. Then the set S is divided into two

subsets, S_{left} and S_{right}, using a vertical line L. This is shown based on [2] in Fig. 1.

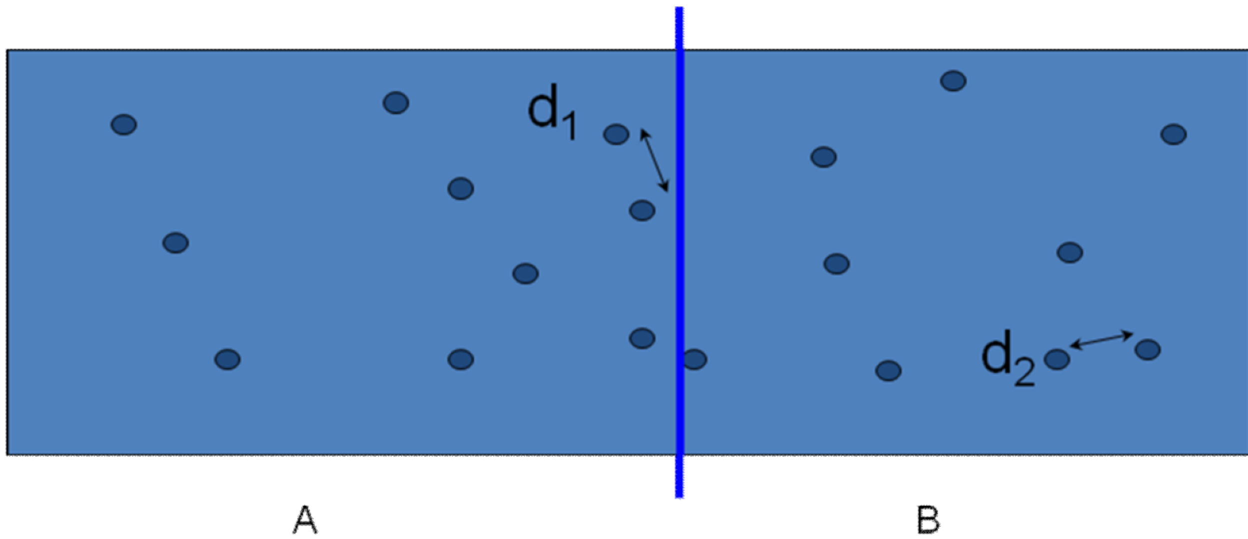


Fig 1: Division of points

It can be verified that the two subsets S_{left} and S_{right} are $\lfloor |S|/2 \rfloor$ and $\lceil |S|/2 \rceil$, respectively.

Recursively, the closest points of S_{left} and S_{right} are computed. Let *d1* and d2 be the distances of the closest points of the sets S_{left} and S_{right}, respectively. Then the minimum distance d_{min} is calculated as follows:

- $d = \min\{d_1, d_2\}$.

Therefore, the closest distance may be either d_l or d_r. The only problem here is that the closest- pair points may spread across S_{left} and S_{right}, that is, one point may belong to S_{left} and another to S_{right}. This needs to be computed. This would take $\vartheta(n^2)$. However, fortunately, one does not require to perform these comparisons as the main objective is to find only the closest pair. Therefore, we examine only a strip of d_{min} from the vertical line based on [2,3], as shown in Fig. 2.

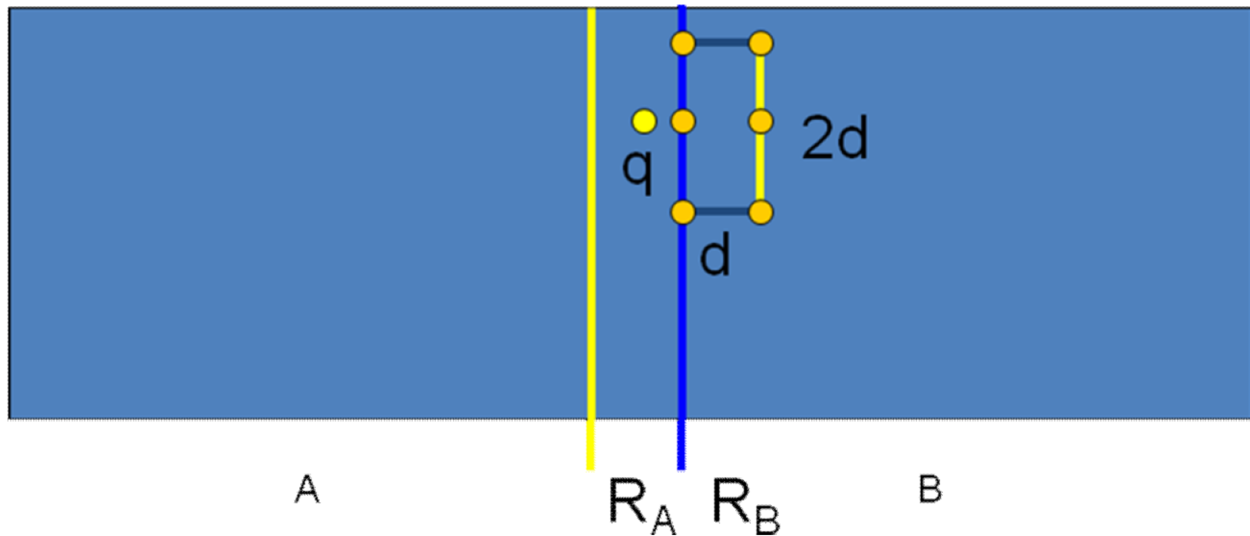


Fig. 2: Grid Formation

It can be observed that the zone around the strip *L* cannot be larger than d_{min}. Hence, the comparison between these points $p \in S_{left}$ and $q \in S_{right}$ is now reduced to this strip only. It can be observed that each point in this strip needs to be compared with at most six points. To find these six points, the points on this strip can be sorted based on *y*-coordinate. Then every point can be compared with at most six points. Next, the minimum distance (denoted as d_{across}) can be computed. It represents the closest distance between the points in the strip.

- **Informal Algorithm:**

The algorithm based on [1] is given as follows:

Step 1: Sort points in S according to their y-values and x-values. **Step 2:** If S contains only two points, return infinity as their distance.

Step 3: Find a median line L perpendicular to the X-axis to divide S into two subsets, with equalsizes, S_L and S_R.

Step 4: Recursively apply Step 2 and Step 3 to solve the closest pair problems of S_L and S_R. Letd_1, d_2 denote the distance between the closest pair in S_L (S_R). Let d = min(d_1, d_2).

Step 5: For a point P in the half-slab bounded by L-d and L, let its y-value by denoted as y_P

For each such P, find all points in the half-slab bounded by L and L+d whose y-value fall within y_P +d and y_P -d. If the distance d′ between P and a point in the other half-slab is less than d, let d=d′ . The final value of d is the answer. Formally, the algorithm can be stated [1] as follows:

Algorithm closestpair *A*[*A*[1 … *n*]]

%% Input: A set of *n* points

%% Output: Two closest points and distance

Begin

If *n* < threshold, then solve the problem by conventional algorithm

Else

$$mid = \left\lfloor \frac{(i+j)}{2} \right\rfloor$$

d_l = closestpair(*A*[1 ... mid])

d_r = closestpair(*A*[mid + 1 … *n*])

d = min(d_l, d_r)

End if

```
for index = 1 to n do        %% collect all points around L

  if (A[index] >= A(mid).x − d) or A[index] <= A(mid + 1).x + d) then
     Append A[i] to array V
  End if
End for

Sort list V based on y-coordinates

%% Find closes points of the strip and distance

Let d_across = minimum of distance among six points of array V

return (min(d_min, d_across))          %% Send minimum distance
End
```

- **Complexity Analysis**

The division of S into S_{left} and S_{right} takes $\vartheta(1)$ time. Combining the two would take $\theta\left(n \log n\right)$time. The important task here is to sort the points. This would take $\theta\left(n \log n\right)$ time using a quicksort algorithm. Therefore, the recurrence equation would be as follows:

To reduce the complexity of the merging process further, one can presort the points based on y-coordinate. This improves the performance as in the combining step, instead of sorting; one needs to extract only the elements in $\vartheta(n)$ time. Total running time: $O(n \log^2 n)$.

1.2.2 Convex Hull

What is a convex Hull?

Let S be a set of points in the plane. Imagine the points of S as being pegs; the convex hull of Sis the shape of a rubber-band stretched around the pegs. Formally stated, the convex hull of S isthe smallest convex polygon that contains all the points of S. Convex hull is useful in many applications such as *collision detection in Robotics and Games design.*

Brute force Method:

Extreme points of the convex polygon form the vertex of convex hull. If all the points in the polygon as a set, then *extreme point* is a point of the set that is not a middle point of any line segment with end points in the set. A line segment connecting two points P_i and P_j of a set of npoints is a part of its convex hull's boundary if and only if all the other points of the set lies onthe same side of the straight line through these two points.

- **Informal Algorithm:**

The informal algorithm based on [1] is given as follows:

Determine extreme edges

for each pair of points $p,q \in P$ do

if all other points lie on one side of line passing thru *p* and *q* then keep edge *(p, q)*

Convex hull can be solved effectively using divide and conquer approach. Quickhull andMergehull are two such algorithms for constructing convex hull.

1.2.3 Quick Hull

Two algorithms, namely, quickhull and merge hull, are available for constructing a convex hull.The common approach for both these algorithms is given informally as follows:

Step 1: Divide the *n* points into two halves.

Step 2: Construct hulls for the two halves.

Step 3: Combine the two hulls to form a convex hull.

Quickhull is an algorithm that is designed to construct a convex hull; it is called quickhull as its logic is closer to that of finding the pivot element in a quicksort algorithm. This algorithm is dubbed the "Quickhull" algorithm by Preparata and Shamos (1985) because of similarity to QuickSort and quickhull uses the divide-and-conquer strategy to divide the *n* points of a set *S* in the plane.

This approach is as follows:

- Identify extreme points a and b (part of hull)
- Compute upper hull

- find point c that is farthest away from line a,b
  - Connect the points ac
  - Connect the points cb

This is shown based on [2] in Fig. 3.

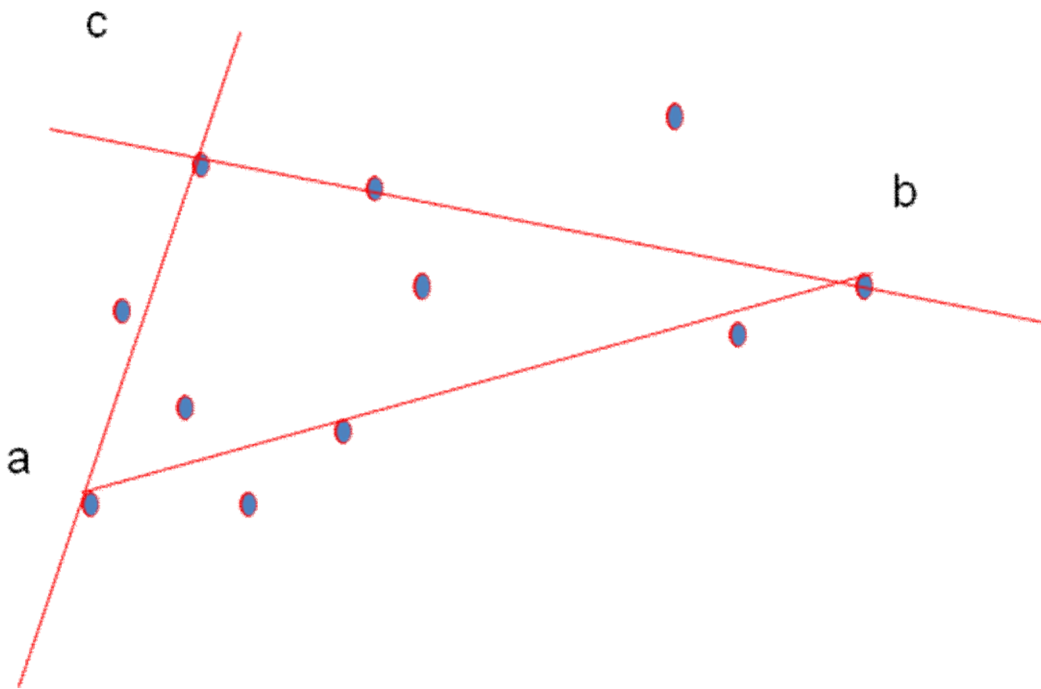


Fig. 3: Quickhull Formation

Thus the idea of quickhull is given [2,3] as follows:

function *QuickHull*(*a*, *b*, *S*)

if *S* = {*p1*, *pn*} then return {p1,pn}else
pmax ← point furthest from edge (*p1,pn*)

A ← points right of (*p1,pmax*)

B ← points right of (*pmax,p2)*

return *QuickHull*(p1,pmax,*A*) concatenate with QuickHull(*pmax,p2,B)*

- ## Informal Algorithm

Informally, the algorithm [1] for Quickhull can be written as follows:**Step**

1: Sort the points based on *x*-coordinates.

Step 2: Identify the first point p_1 and last point p_n.

Step 3: Use $\overline{p_1 p_n}$ to divide the set into S_{left} and S_{right}.

Step 4: For S_{left}, find a point p_{max} that is far from the line $\overline{p_1 p_n}$. This line divides the set of points

of S_{left} into two sets S_{11} and S_{12}.

Step 5: Ignore all the points inside the triangle $p_1\, p_{max}\, p_n$.

Step 6: Form the left convex hull as $p_1 \cup S_{11} \cup p_{max}$ and $p_{max} \cup S_{12} \cup p_n$.

Step 7: Form the right convex hull using the steps similar to those used for the formation of theleft convex hull.

Step 8: Combine the left and right convex hulls to get the final convex hull.

- **Complexity Analysis**

Let n be the number of points of a set S, which are evenly divided into sets S_1 and S_2 in a quickhull algorithm. Let the sets consist of points n_1 and n_2, then the recurrence equation for quickhull algorithm is given as follows:

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

The solution of this leads to $O(n \log n)$.

1.2.4 Merge Hull

Merge hull is another algorithm that is based on merge sort for constructing a convex hull [3]. It uses the divide-and-conquer strategy. This algorithm is as effective as quickhull.

Idea of Merge Hull

Sort the points from left to right
Let A be the leftmost $\lceil n/2 \rceil$
points Let B be the rightmost
$\lfloor n/2 \rfloor$ points

Compute convex hulls $H(A)$ and $H(B)$
Compute $H(A \cup B)$ by merging $H(A)$ and
$H(B)$

Initially, the points are sorted based on x-coordinates. Then partition is made based on the median of x- axis coordinates. Imagine a line that passes through the median dividing the set of points. This process divides the set of n points into two sets S_1 and S_2. Then convex hulls are constructed recursively from the sets of points S_1 and S_2. The main focus shifts to merging the convex hulls that are constructed.

Two convex hulls are joined by a lower tangent and an upper tangent. A tangent is also known as a bridge. It connects a vertex on the left convex hull with a vertex on the right convex hull. It is obtained by keeping one end fixed and changing another end rapidly to find whether it is a potential tangent. Then the convex hulls are merged using the upper and lower tangents while ignoring all the points between the tangents.

The following is the informal algorithm for merge hull:

Step 1: If the number of points involved is less, say less than 3, solve the problem conventionally using abrute force algorithm.

Step 2: Sort the points based on x-axis coordinates.

Step 3: Partition the set S into two sets S_{left} and S_{right} such that

$$Set\ S_{left} = \{1, 2, 3, \cdots S_{mid}\}$$

$$Set\ S_{right} = \{S_{mid}, S_{mid+1}, \cdots S_n\}$$

where mid is the median of x-axis coordinates. S_{left} now has all the points that are less than the median and S_{right} has all the points that are higher than the median.

Step 4: Recursively construct the convex hull.

Step 5: Find the lower and upper tangents between convex hulls

Step 6: Form the convex hull by merging the lower and upper convex hulls using the lower and upper tangents and ignoring all the points that fall between them.

Finding Tangent Lines

The most important aspect of merge hull is finding the upper and lower tangents based on [1] , which are shown in Fig. 4.

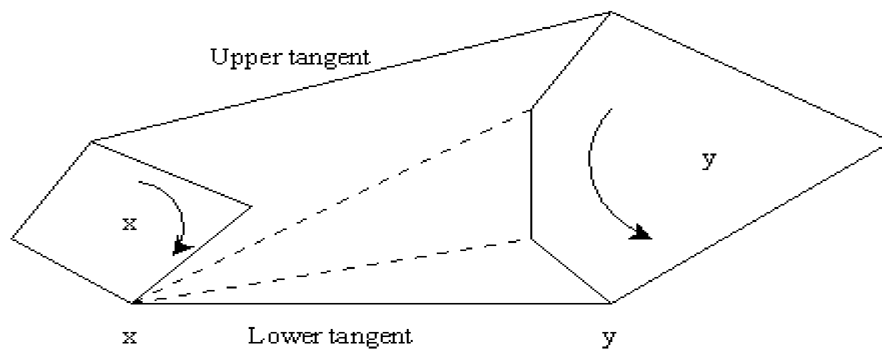


Fig. 4 Lower and upper tangents

Let x be the rightmost point of the S_{left} convex hull and y the leftmost point of the S_{right} convex hull. Connect x and y. If xy is not a lower tangent of S_{left}, then perform a clockwise rotation and pick the next vertex of the convex hull. Similarly, if xy is not a lower tangent for S_{left}, perform a counter-clockwise rotation and pick the next vertex of the convex hull. Using the right and left turns between points, one can decide whether the points lie on the tangent or not. In the same manner, the upper tangent is also formed.

- **Complexity Analysis**

The recurrence equation of merge hull is given as follows:

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

Therefore, the solution of this equation leads to $O(n \log n)$.

Check Your Progress

Fill in the Blanks.

Q.1: Convex hull is useful in many applications such as collision detection in ______________________.

Q.2: Closest pair problem is to find the closest pair among _______in 2-dimensional space.

Q.3: Quickhull is an algorithm that is designed to construct a ___________.

Q.4: QuickSort and quickhull uses the _____________strategy to divide the n points of a set S in the plane.

1.3 Answer to Check Your Progress

Ans 1. Robotics and Game design

Ans 2. n points

Ans 3. Convex hull

Ans 4. divide-and-conquer

2.1 Applications of Divide and Conquer

What is divide and Conquer design paradigm?

Divide and conquer is a design paradigm. It involves the following three components:

Step 1: (Divide) The problem is divided into subproblems. It must be noted that the subproblems are similar to the original problem but smaller in size.

Step 2: (Conquer) after division of the original problem into subproblems, the subproblems are solved iteratively or recursively. If the problems are small enough, then they are solved in a straightforward manner.

Step 3: (Combine) Then, the solutions of the subproblems are combined to create a solution to the original problem

2.1.1 Finding Maximum and Minimum Elements

Finding maximum and minimum of an array is one of the most commonly used routine in manyapplications. Maximum and minimum are called order statistics.

The conventional algorithm for finding the maximum and minimum elements in a given array isgiven as follows:

1. Set largest = A[1]

2. Set index = 2 and N = length(A)

3. While (index <= N) do

    if A[index] > largest then
      largest = A[index]

4. Print the largest

5. End.

- **Complexity Analysis:**

It can be observed that the conventional algorithm requires 2n-2 comparisons for finding maximum andminimum in an array. Therefore, the complexity of the algorithm is O(n).

Idea of Divide and Conquer Approach

One can use the divide-and-conquer strategy to improve the performance of the algorithm. The idea is to divide the array into subarrays and to find recursively the maximum and minimum elements of the subarrays. Then, the results can be combined by comparing the maximum and minimum of the subarrayto find the global maximum and minimum of an array.

To illustrate this concept, let us assume that the given problem is to find the maximum and minimum of an array that has 100 elements. The idea of divide and conquer is to divide the array into two subarrays of fifty elements each. Then the maximum element in each group is obtained recursively or iteratively. Then, the maximum of each group can be computed to determine the overall maximum.

This logic can be repeated for find minimum also.

- **Informal Algorithm**

This idea can be generalized to an informal algorithm as follows:

1. Divide the n elements into 2 groups A and B with floor(n/2) and ceil(n/2) elements, respectively.

2. Find the min and max of each group recursively.

3. Overall min is min{min(A), min(B)}.

4. Overall max is max{max(A), max(B)}.

This idea is illustrated in the following Example 1 and Example 2.

Example 1: Find the maximum of an array {2,5,8,1,3,10,6,7} using the idea of divide and conquer.

Solution: The idea is to split the above array into subarrays A and B
such thatA = {2,5,8,1} and

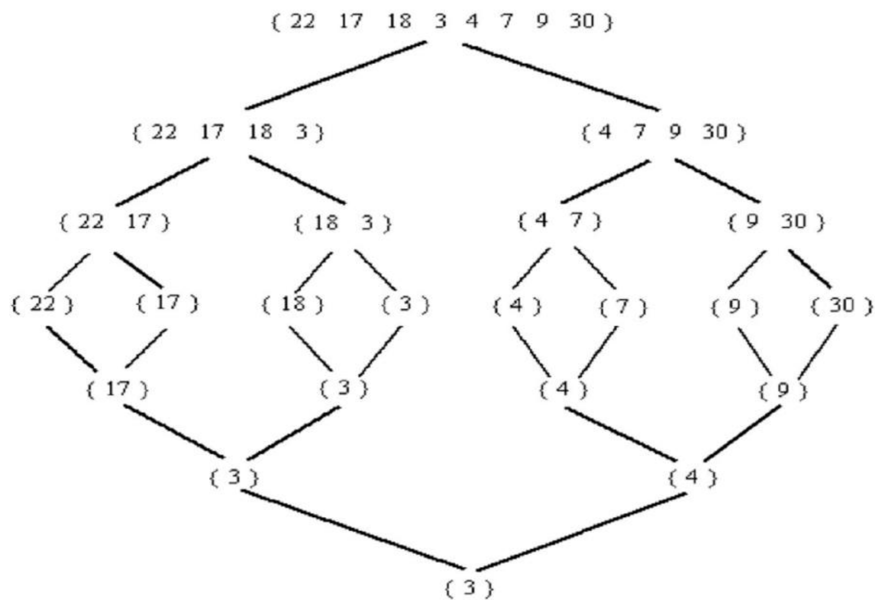
B = {3,10,6,7}

The idea can be repeated to split subarrays A and B further. Then, it can be
found thatmax(A) = 8, max(B) = 10.

Therefore, the maximum of the array is - max{max(A), max(B)} = 10.

Example 2: Find the minimum of the array {22,17,18,3,4,7,9,30} using divide and conquer idea?

Solution: The idea of the previous problem can be repeated. This results in the following Fig. 1.



∴ The minimum element is 3.

Fig 1: Finding Minimum in an array

- **Formal Algorithm**

The formal algorithm based on [1] is given as follows:

```
Algorithm minimummaximum A(i,j)

Begin

  mid = floor of (i + j) / 2

  [max, min] = minimummaximum(A[i,mid])
  [max1,min1] = minumummaximum(A[mid+1,j])
  globalmax = max(max,max1)

  globalmin = min(min,min1)
End
```

It can be observed that, this algorithm formally divides the given array into two subarrays. The subarrays are subdivided further if necessary. It can be observed that only the maximum and minimum elements of the subarrays are compared to get the maximum/minimum element of the parent list.

- ## Complexity Analysis of Finding Maximum/Minimum

The recurrence equation for the max/min algorithm [1,2] can be given as follows:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

Assume that $n = 2^k$. By repeated substitution, one can obtain that the following relations:

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

$$= 2\left[\left\{2T\left(\frac{n}{4}\right)\right\} + 2\right] + 2$$

$$= 4T\left(\frac{n}{4}\right) + 4 + 2$$

$$\vdots$$

$$= 2^{k-1} \cdot T(2) + \sum_{i=1}^{k-1} 2^i$$

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

$$= \frac{2^k}{2} + 2^k - 2$$

$$= \frac{n}{2} + n - 2 \quad \text{, since } n = 2^k.$$

$$= \frac{3n}{2} - 2$$

The solution of the recurrence equations gives $(3n/2) - 2$ comparisons.

2.2 *Tiling Problem*

Another important problem is Tiling problem [3]. The problem can be stated as follows: Given a Region and a Tile T, Is it possible to tile R with T? A defective chessboard is a chessboard that has one unavailable (defective) position. A tromino is an L shaped object as shown in Fig. 2.

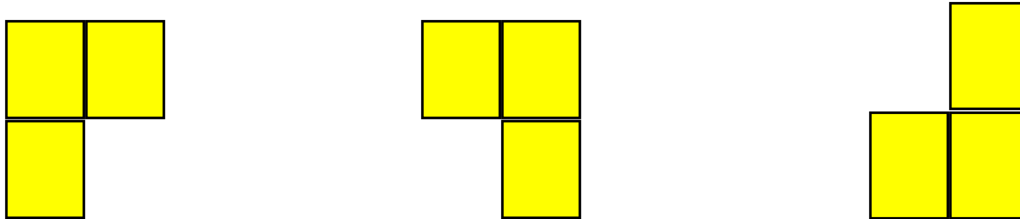


Fig. 2: Examples of Tromino

The idea is to tile the defective chess board with a tromino. The divide and conquer paradigm can be applied to this problem. The procedure for applying divide and conquer paradigm is givenbelow:

If board is small, Directly tile,
else

- Divide the board into four smaller boards
- Conquer using the tiling algorithm recursively
- Combine it

It can be understood as follows. If the board is small, then the tromino can be applied manually and checked. Else, divide and conquer paradigm can be applied. The board configurations can be divided further, then the subboards can be tiled, finally the results can be combined to findsolution of the given larger board.

The formal Algorithm based on [1] for Tiling problem is given as follows: *INPUT*: *n* – the board size (2^nx2^n board), *L* – location of the defective hole. *OUTPUT*: tiling of the board

Algorithm Tile(n, L)

Input : n – order of the board, L

– Tromino

Begin

if n = 1 **then**

Tile with one tromino directly

return

Else

Divide the board into four equal-sized boards

Place one tromino at the centre to cover the defective hole by assuming the extra 3 additionalholes, L1, L2, L3, L4 denote the positions of the 4 holes

Tile(n-1, L1)

Tile(n-1, L2)

Tile(n-1, L3)

Tile(n-1, L4)

End

The complexity analysis of this is given based on [1] below.

Complexity Analysis

The recurrence equation of the defective chess board problem is given as follows:

$$T(n) = \begin{cases} c & \text{for } n = 0 \\ 4T\left(\frac{n}{2}\right) + c & \text{for } n > 0 \end{cases}$$

Therefore, the complexity analysis is O(n^2).

2.2.1 Fourier Transform

Fourier transform [4] is used for polynomial multiplication because it helps to convert one representation of a polynomial (coefficient representation) to another representation (value representation). Thus, computation using Fourier transform can be carried out in the following two ways:

Evaluate Fourier transforms help in converting a coefficient representation to a value representation. This is given as follows:

<values> = Fourier transform((coefficients), ω).

Interpolation After computation, conversion of a value representation to acoefficient form can be performed using inverse Fourier transform. This is given as

follows:

<coefficients> = Fourier transform((values), ω^{-1})

One may view Fourier transform as a method of changing the representation or coding of a polynomial.

Fourier transform has many applications. One of its important applications is polynomial multiplication.

A polynomial is used to represent a function in terms of variables. A

polynomial can be represented as follows:

$$A = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$$

Here n is referred to as the degree bound of the polynomial and $a_0, a_1, \ldots, a_{n-1}$ are called its

coefficients. The polynomial is said to be of degree k, if the highest coefficient of the polynomial

is a_k.

Fourier transform is then given as follows:

$$A_i = \sum_{k=0}^{n-1} a_k e^{-j\frac{2\pi ik}{n}}, \ 0 \le i \le n-1,$$

a_k, where k ranges from 0 to $n - 1$, represents the set of coefficients of a polynomial

$\{a_0, a_1, \cdots a_{n-1}\}$; n is the length of the coefficient vector that represents the degree of the given

polynomial. In other words, Fourier transform also represents a polynomial as the nth root of

unity [3]. The nth root is the solution of the polynomial $x^n - 1 = 0$, which is given as $\omega_n = e^{-j2\pi}$.

The nth root at all points of input x is given as $e^{2\pi jx/n}$. Using Euler's formula, the evaluation of

$e^{2\pi jx/n}$ yields $\cos(\frac{2\pi}{n}x) + j\sin(\frac{2\pi}{n}x)$, where $j = \sqrt{-1}$. The output of a Fourier transform thus

can also be a complex number.

One can also design inverse Fourier transform to convert a value form to a coefficient form.Inverse Fourier transform can be given as follows:

$$a_i = \frac{1}{n}\sum_{k=0}^{n-1} A_k e^{j\frac{2\pi ik}{n}}, \quad 0 \le i \le n-1$$

To multiply faster and effectively, it is better to use matrix representation for implementing Fourier transform. The matrix representation is given as follows:

$$A = Va$$

where V is an $n \times n$ matrix, called the Vandermonde matrix, and a is the vector of coefficients given as $\{a_0, a_1, \cdots, a_{n-1}\}$. Here n represents the number of coefficients of the given polynomial. The resultant vector A is a set of values given as $\{A_0, A_1, \cdots, A_{n-1}\}$, which represent the transformed coefficients of Fourier transform. The matrix V can be given as follows:

$$V = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix}$$

Thus, the resultant matrix A of Fourier transform can be given as follows:

$$A=\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \times a$$

Inverse Fourier transform can be obtained as follows:

As $A = Va$, the coefficients a can be retrieved as follows:

$$a = V^{-1} \times A$$

Here, the matrix V^{-1} can be obtained by taking the complex conjugate of the matrix V by replacing ω by ϖ, as $\varpi = \frac{1}{\omega}$ or ω^{-1}. Complex conjugate means the sign of the imaginary component of a complex number is changed. Therefore, substituting this in the matrix, one getsthe inverse matrix V (V^{-1}), which is as follows:

$$V^{-1} = 1/n \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \varpi^1 & \varpi^2 & \varpi^3 & \cdots & \varpi^{n-1} \\ 1 & \varpi^2 & \varpi^4 & \varpi^6 & \cdots & \varpi^{2(n-1)} \\ 1 & \varpi^3 & \varpi^6 & \varpi^9 & \cdots & \varpi^{3(n-1)} \end{pmatrix}$$

$$\begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \varpi^{(n-1)} & \varpi^{2(n-1)} & \varpi^{3(n-1)} & \cdots & \varpi^{(n-1)^2} \end{pmatrix}$$

Thus, the resultant matrix a of inverse Fourier transform can be given as follows:

$$a = 1/n \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \times A$$

Let us try to design the matrix V and V^{-1} for four sample points. Therefore, $n = 4$ and let $a = \{a_0, a_1, a_2, a_3\}$. Then one can find ω by substituting $e^{\frac{2\pi jx}{n}}$ as follows:

$$\omega = e^{-j\frac{2\pi}{n}} = e^{-j\frac{2\pi}{4}} = e^{-j\frac{\pi}{2}} = \cos\frac{\pi}{2} - j\sin\frac{\pi}{2} = -j \text{ (as } n = 4\text{)}$$

On substituting this value of ω in the matrix V of order 4×4, one gets the following matrix:

$$V = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & (-j)^2 & (-j)^3 \\ 1 & (-j)^2 & (-j)^4 & (-j)^6 \\ 1 & (-j)^3 & (-j)^6 & (-j)^9 \end{pmatrix}$$

Here, j is a complex number and is equal to -1. Therefore, one can observe that the resultant matrix V that involves complex numbers is as follows:

$$V = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix}$$

Thus, the resultant matrix A can be given as follows:

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The coefficients can be retrieved using inverse Fourier transform. For this case where $n=4$ and $A_k = \{A_0, A_1, A_2, A_3\}$, the matrix V^{-1} of order 4×4 can be obtained by substituting $\varpi = \frac{1}{\omega} = e^{j\frac{2\pi}{n}} = e^{j\frac{\pi}{2}} = -j$ in the general matrix. This is the complex conjugate of the matrix V. For finding the complex conjugate, one has to change the sign of the imaginary component of the complex number. For $n = 4$, the inverse matrix V (V^{-1}) is given as follows:

$$V^{-1} = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & +j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{pmatrix}$$

Thus, the resultant matrix for finding coefficients from values is given as follows:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \frac{1}{n} \times \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \times \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix}$$

One can verify that the product of V and V^{-1}
is a unit matrix as they are complex conjugates of

each other. In addition, one can check that the original coefficients are obtained using inverse Fourier

transform and there is no information loss. This is demonstrated in the following numerical example 1.

Example 1

Find Fourier transforms of the following four coefficients and also verify that inverse Fouriertransform

gives the original coefficients without any loss.

$$x = \{1, 3,5,7\}$$

Solution

As there are four samples, n = 4. Fourier transform can be given as follows:

$$A = V \times a$$

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 5 \\ 7 \end{pmatrix}$$

$$= \begin{pmatrix} 16 \\ -4-4j \\ -4 \\ -4-4j \end{pmatrix}$$

One can verify that the inverse of this gives back the original coefficients. Therefore, take theinverse

kernel and multiply the Fourier coefficients:

$$a = \frac{1}{n} \times (V^{-1} \times A)$$

$$a = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & +j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{pmatrix} \begin{pmatrix} 16 \\ -4-4j \\ -4 \\ -4-4j \end{pmatrix}$$

$$= a = \frac{1}{4}\begin{pmatrix} 4 \\ 12 \\ 20 \\ 28 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 5 \\ 7 \end{pmatrix}$$

It can be observed that one is able to get back the original coefficients.

- **Idea of FFT**

One can implement a faster Fourier transform using an algorithm called an FFT algorithm. FFT is implemented using the divide-and-conquer strategy. The input array of points is divided into odd and even arrays of points. Individually, FFT is applied to the subarrays. Finally, the subarrays are merged. Informally, an FFT algorithm can be stated as follows:

Step 1: If n = 1, then solve it directly as a_0.

Step 2: Divide the input array into two arrays B and C such that B has all odd samples and C hasall even samples. Continue division if the subproblems are large.

Step 3: Apply FFT recursively to arrays B and C to get subarrays B' and C'.**Step 4**: Combine the results of subarrays B' and C' and return the final list.

Complexity Analysis of FFT algorithms

The recurrence equation of FFT is given as follows:

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

One can use the master theorem and solve this recurrence equation. One can observe that the complexity analysis of this algorithm turns out to be $O(n \log n)$.

2.2.2 Polynomial Multiplication

Many polynomial operations are required in scientific applications. One of the most important applications is multiplying two polynomials. Let $a(x)$ and $b(x)$ be two polynomials; their product $C(x) = a(x) \times b(x)$ can be expressed [3] as follows:

Compute C(x) = A(x)B(x), where degree(A(x)) = m, and degree(B(x)) = n. Degree(C(x)) = m+n, and C(x) is uniquely determined by its value at m+n+1 distinct points.

The informal algorithm is given as follows:

Step 1: Let the polynomials A and B be of degree n. Then find $m = 2 \times n - 1$.

Step 2: Pick m points ranging from 0 to $m - 1$.

Step 3: Evaluate polynomials A and B at m points.

Step 4: Compute $C(x) = A(x) \times B(x)$ using ordinary multiplication.

Step 5: Interpolate $C(x)$ to get the coefficients of the polynomial $C(x)$.

It can be observed that point-wise multiplication is enough to multiply polynomials. One can combine the idea of divide and conquer with this concept. The idea of division is that any function at sample points x can be divided into function samples at odd points and those at even

points. Thus, a polynomial $A(x)$ can also be represented as $A \quad (x^2) + xA \quad (x^2)$, where A_{odd}

odd even

represents a set of odd sample points and A_{even} a set of even sample points of the given polynomials. Therefore, the advantage of using a divide-and-conquer algorithm is that only one- half of the resultant polynomial is calculated and the other half is a negative of the first half (i.e., $A_{odd}(x^2) - xA_{even}(x^2)$).

Check Your Progress

Fill in the Blanks.

Q.1: One of its important applications of Fourier transformation is__________________.

Q.2: The complexity of the algorithm finding maximum and minimum in an array is________.

Q.3: The complexity of the algorithm faster Fourier transform is___________.

Q.4: The complexity of tiling problem is _____________.

2.3 Answer to Check Your Progress

Ans 1. polynomial multiplication

Ans 2. O(n)

Ans 3. O(nlogn)

Ans 4. O(n^2)

3.1 Introduction to Decrease and Conquer Design paradigm

The decrease and conquer paradigm is based on problem reduction strategy. Problem reduction is a design strategy that aims to reduce a given problem to another problem with a reduced problem with smaller size. Then, attempts are made to solve the problem. Decrease and conquer is a design paradigm that uses the problem reduction strategy. It is also known as the incremental or inductive approach. This paradigm is useful for solving a variety of problems in the computer science domain.

The steps of decrease and conquer is given as follows:

1. Reduce problem instance to same problem with smaller Instance
2. Solve problem of smaller instance
3. Extend solution of smaller instance to obtain solution to original problem with largerinstance

For example, consider the following problem, of computation of a^n. The problem can be solved by reducing it another problem of $a^n = a \times a^{n-1}$ if n>0, If n =0, then its value is n. the problem can further be reduced. It can be observed that this design paradigm reduces a given problem sizeby a certain decreasing factor. Then it establishes a relationship between the solution to a given instance of the problem and that to a smaller instance of it. Once the relationship is established, itis exploited using the top-down (recursive) or bottom-up (iterative) approach to derive the final solution.

3.2 Categorization of Decrease and Conquer Design paradigm

Based on the decreasing factor, the decrease-and-conquer strategy can further be categorized intothe following types:

3.1.4 Decrease by a constant

3.1.5 Decrease by a constant factor

3.1.6 Decrease by a variable factor

3.1.1 Decrease and Conquer by a constant

In decrease by a constant variation, the problem size is reduced by a constant (mostly one) at every iteration. In this category, a problem of size *n* is divided into a subproblem of size '*n* − 1' and an individual element *n*. This design paradigm then incorporates the individual element into the subproblem solution to obtain the final solution. The examples of this category are Insertion sort, Topological sort, generation of permutations and subsets.

The steps of the decrease by a constant are as follows:

Step 1: Reduce a problem *A* of size *n* into a problem of size '*n* − 1' and an individual element *n*.

Step 2: Solve the subproblem recursively or iteratively.

Step 3: Incorporate the individual element into the solution of the subproblem to obtain the solution of the given problem.

In decrease by a constant factor, a problem instance is reduced by a constant factor, which is 2 inmost of the cases. The examples of this category are binary search, faster exponentiation, and Russian Peasant method for multiplying two numbers. In decrease by a variable factor, the reduction size varies from one iteration of the algorithm to another. The number of subproblems may also vary. The examples of this category are Euclid algorithm, selection by partition and Nim type games.

Decrease and conquer by a constant approach is discussed in the following sections.

3.1.1.1 Insertion Sort

Insertion sort is based on decrease and conquer design approach. Its essential approach is to take an array A[0..n-1] and reduces its instance by a factor of 1, Then the instance A[0..n-1] is reduced to A[0..n-2]. This process is repeated till the problem is reduced to a small problem enough to get solved.

The first element is initially considered to be a sorted element; therefore, the second element needs to be compared with one element, requiring only one comparison. The third element needs to be compared with the previous two elements. Thus, the logic of insertion sort is to take an element and copy it to a temporary location. Then the position is looked after for insertion. Once, a position is located, then the array is moved right and the element is inserted. This process is repeated till the entire array is sorted.
Informally the procedure is as follows:

- Finding the element’s proper place
- Making room for the inserted element (by shifting over other elements)
- Inserting the element

- **Formal Algorithm**

The formal insertion sort algorithm [2,3] is given below:

ALGORITHM InsertionSort(A[0..N-1])for i =

1 to N-1 do

temp = A[i]j

= i-1

while j ≥ 0 and A[j] > temp do

A[j+1] < A[j]

j = j-1

A[j+1] = temp

This procedure is illustrated in the following numerical example.

Example 1: Apply quicksort to the following set of numbers and show the intermediate result.Solution:

As discussed earlier, the first number is considered as a sorted number. Then one by one elements are inserted into its appropriate position, and the length of the sorted list is increased.This process is continued till all the elements are sorted.

- 88 | **43** 68 92 23 34 11
- 43 88 | **68** 92 23 34 11
- 43 68 88 | **92** 23 34 11
- 43 68 88 92 | **23** 34 11
- 23 43 68 88 92 | **34** 11
- 23 34 43 68 88 92 | **11**

The final sorted list is : 11 23 34 43 68 88 92

- **Complexity Analysis:**

The basic operation of this algorithm is a comparison operation. The number of comparisons depends on the nature of inputs. As said earlier, The first element is initially considered to be a sorted element; therefore, the second element needs to be compared with one element, requiring only one comparison.

The third element needs to be compared with the previous two elements.

- **Worst case analysis:** The worst-case complexity analysis of insertion sort can be determined as follows: Hence, this requires two comparisons. Thus, in general, for *n* elements, the number of

comparisons would be as follows:

$$t(n) = 1 + 2 + \cdots + (n-1)$$

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

$$= O\left(n^2\right)$$

- **Best-case complexity analysis** The best-case complexity of insertion sort occurs when the listis in a sorted order. Even in this case one comparison is required, to compare an item with its previous element. Thus, at least $n - 1$ comparisons are required. Therefore, the complexity analysis of insertion sort in best case would be

$$T(n) = \underbrace{1 + 1 + \cdots + 1}_{(n-1) times}$$

$$= n-1$$

Therefore, the complexity of the algorithm is $O(n)$. In addition, no shifting of data is requiredand space requirement for the sort is n. Similarly, the average case complexity of insertion sort is

$O\left(n^2\right)$

3.1.1.2 Topological Sort

Topological sort is one of the most important sorting used in variety of applications such as course prerequisites and project management. Thus, the objective of topological sort is to produce an ordering that implies a partial ordering on the elements. Thus, the ordering of the elements satisfies the given constraints. First, given a set of constraints graph is constructed. Every vertex represents an item. Every constraint is represented as an edge. For example, the constraint where item A must finish before B, then a directed edge is created from A to B. If the edge is represented as <A,B>, then the vertex A appears before vertex B in the linear ordering list.

Topological sort is performed for directed acyclic graphs (DAGs), and it aims to arrange thevertices of a given DAG in a linear order. Thus, a DAG is an essential and necessary conditionfor topological sort. What is a DAG? A DAG has no path that starts and ends at the same vertex. A sample DAG is shown below in Fig. 1.0

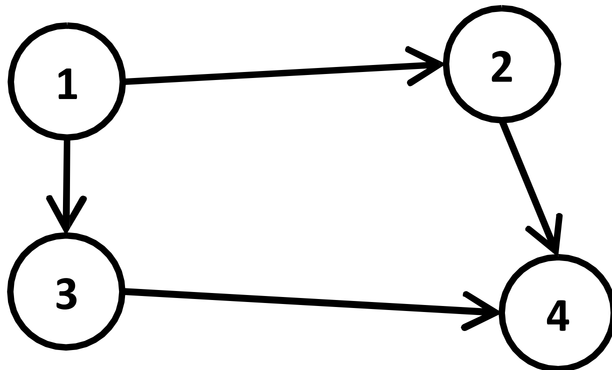


Fig. 1.0: A sample DAG

Recollect that a node of a graph that has no incoming edge is called a source and a vertex that has no outgoing edge is called a sink. A DAG has only one source and one sink. If a graph has many sources, then one can create a super source by creating a node and connecting it to all source nodes.

- **Topological Sort using DFS**

Topological sorting can be performed using BFS and DFS algorithms. The following is the informal algorithm for performing topological sort using DFS [2,3]:

1. Perform DFS traversal, noting the order of the vertices that are popped off stack
2. Reverse order solves topological sorting problem

In other words, the finish time F(u) of all the vertices of a graph is obtained. Then a queue Q is created and all the vertices are inserted on to the queue Q based on finish time. Then the contentsof the queue is printed as the sorted sequence.

- **Formal Algorithm**

The formal algorithm for Topological sort is given as follows:

Topological-Sort()

Begin Run

DFS

When a vertex is finished, output it and assign a number

Vertices are output in reverse topological order

End.

It can be used on the following problem. Consider the following graph shown in Fig 2.

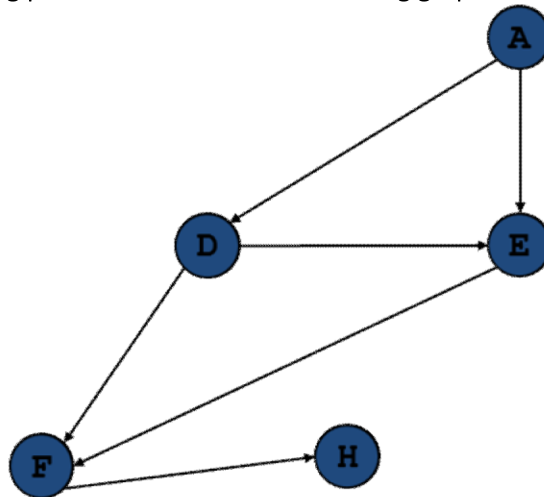


Fig 2: Initial Graph

Initially A is visited. Then the verted D is visited. Then, vertex E is visited. Then, vertex F is visited. Finally, vertex H is visited. This is numbered as 5. By reversing the DFS traversal, it can be observed that F is numbered as 4, vertex E is 3, vertex D is numbered as 2 and finally vertexA as 1. By reversing this one get the topological order which is given as A D E F H.

- **Topological ordering using source node removal algorithm**

In source node removal algorithm, one has to identify the source repeatedly and it is removed. Simultaneously, all the edges incident to it are removed. This process is repeated till either no vertex is left or there is no source among remaining vertices left.

As an example, consider the following graph shown in Fig. 3. This represents the prerequisites ofa set of courses that needs to be taken.

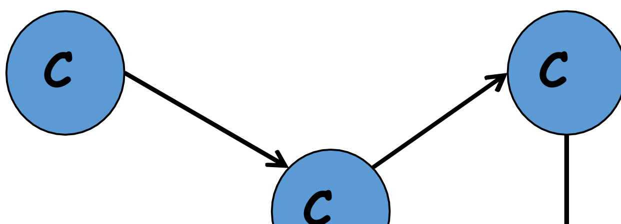


Fig. 3. : Example course graph

It can be observed, the node that has no incoming edges is C1. Therefore, it is removed along with its incident edges. Then, C3 is the vertex that is source. Then, it is removed. Then C4 is the vertex , that is source and hence removed. Finally, the node c5 is selected. Therefore, the sorting order is given as follows: C1, C3, C4 and C5.

- **Complexity Analysis**

Let there be m vertices and n edges of a graph. Then the topological algorithm takes $O(m)$ time for picking the vertex that has no incoming edges. This is done by examining the corresponding adjacency matrix or adjacency list. Picking the vertex after identification takes a constant time. Deleting the vertex along with its edges takes $O(n)$ time. Putting together, the algorithm takes $O(m + n)$ time.

3.1.1.3 Permutations

Permutation is an arrangement of objects in a linear order. A permutation of a set of objects is a particular ordering of those objects For example, for three objects A, B and C, the first element

A can be arranged in three ways, the second object B in two ways, and the third object C in one way. Thus, the three objects can be arranged in 3! Ways. Therefore, the permutations are {ABC, ACB, BAC, BCA, CAB, CBA}. Each arrangement is called a *permutation*. In general, there are n! ways of arranging a set of n elements.

Generating a permutation may seem to be a trivial task. However, in reality, it is not so. For example, if there are 100 elements, then there are 100! ways of arranging the elements. Therefore, there is a need for generating permutations in an effective manner.

- **Decrease and Conquer Approach:**

Decrease-and-conquer paradigm is used for permutation generation. In order to generate n! permutations using the decrease-and-conquer paradigm. For example, consider the problem of generating permutations for the set {A,B,C}.

The solution using the decrease-and-conquer paradigm can be given as follows:

The problem of permutation of three elements *P* is reduced to the subproblem(*P*′) of generating permutation of two elements {A,B}. Then, this problem is reduced to the problem of generating permutation of subproblem {A}.The permutation of {A} is {A}. Then, the element {B} is introduced to enlarge the solution. Then the element {A} is added to get the final answer.

- **Informal Algorithm**

To find all permutations of n objects:

Find all permutations of n-1 of those objects

Insert the remaining object into all possible positions of each permutation of n-1 objects

This can be illustrated as follows:

This process is described as follows:

Given the empty set { }, the only possible permutation is { }Given

the set {A}, the only possible permutation is {A} Given the set {A,

B}, the possible permutations are

{AB, BA}

This idea can be extended for three objects. This is illustrated in the following Example 2.

Example 2 : Find all permutations of 3 objects {A, B, C}

Solution:

Find all permutations of 2 of the objects, say B and C: The possible permutations areB C

and C B

Insert the remaining object, A, into all possible positions (marked by ^) in each of thepermutations of

B and C:

^ B ^ C ^ and ^ C ^ B ^ . This results in the following combinationsABC

BAC BCA ACB CAB CBA

- **Complexity analysis** The complexity analysis for generating permutations is as follows:

$$T(k) = \begin{cases} 0 & \text{if } k=1 \\ kT(k-1)+2 & \text{for } k>1 \end{cases}$$

when *k* = *n*, the worst-case complexity for generating permutations is $\Omega(n!)$.

3.1.1.4 Johnston–Trotter Algorithm

Another way of generating permutations is by using the Johnston–Trotter algorithm that uses the decrease-and-conquer strategy. This algorithm generates permutations in a non-lexicographical order. In this algorithm, every integer is associated with a direction. For example, <3 means the integer 3 is assigned a direction left and >3 means it is assigned a direction right. The core idea of this algorithm is that, a integer is called a mobile integer if it points to a neighbouring integer than is lesser than it. For example, in the generation of a permutation like <2 <3 <1, 3 is a mobile integer as it is pointing to an integer that is lesser than it. The right- and left-most columns of a list are called its boundaries. If any mobile integer in a boundary does not point to any integer, then the number is not a mobile number. In the generated sequence <3 <2 <1, 3 is no longer a mobile number as it is in the left-most column (or boundary).

Informally, the Johnston–Trotter algorithm is given as follows:

- **ALGORITHM Johnson Trotter (n)**

```
Initialize the first permutation with 1 2 … n
while there exists a mobile integer k do
  find k – the largest mobile integer
    swap k and the adjacent integer pointed by arrow reverse
    the direction of all integers that are larger than k
```

This algorithm for generating permutations for three elements {1,2,3} is given below in Table 1.:

Table 1. Generating permutations for three elements using the Johnston–Trotter algorithm

Permutations	Descriptions
<1 <2 <3	3 is the mobile integer as it points to '2' that is smaller. Therefore, move the mobile integer.
<1 <3 <2	Now the mobile integer is pointing to '1', which is smaller. Therefore, move it
<3 <1 <2	The mobile integer 3 has reached the boundary and does not point to any element. Therefore, look for the next largest integer, which in this case is '2'. Move it to get the next permutation and also change the direction of the integers that are larger than the current mobile integer.
3> <2 <1	It can be observed that the direction of 3 is changed. In other words, 3 has become a mobile integer again. Now move it again.
<2 3> <1	Now the mobile integer 3 points to a smaller number '1'. Therefore, move it again.
<2 <1 3>	The numbers 2 and 3 have reached the boundaries and do not point to any
	integers that are lesser than these. As there are no mobile integers, exit.

Generating Subsets

The following are the important terminologies related to this problem:

Set : A collection of distinct elements. For example = colour = { red, Blue, Green}

Subset : A set B is a subset of A , if its all elements are in A. For example, A = {1,2}, then the subsets of A are { }. {1}, {2}, {1,2}

Power set: The set of all subsets is called power set

Generating Subsets Using Decrease-and-conquer Strategy

To generate subsets $A = \{a_1, a_2, a_3, \ldots, a_n,\}$, one has to divide this into two groups: $S_1 = \{a_1, a_2, a_3, \ldots, a_{n-1}\}$ and $S_2 = \{a_n\}$. Add a_n to each subset of S_1 to get the final solution.

Example:

{1,2}

reduce this problem to

{1}

reduce this problem to

{}

So the solution is

{}

{1} {} after {1} is inserted

{1, 2} {2} {1} {}

Another easy approach to generate subsets is to use a binary string for *n* digits. The idea is to have a 1:1 correspondence between a binary string and the generation of subsets. Informal algorithm based on [1] is given as follows:

1. Initialize the counter i to $2^n - 1$
2. Initialize the item counter j = number of items
3. Extract the jth bit from the counter. If binary digit is 1, then include the item , otherwise,exclude it
4. Print the subsets.

The following Table 2 summarizes the generation of the subsets.

Table 2: Generation of Subsets

Binary Pattern for n = 3	Elements to be added
000	{}

001	{1}
010	{2)
011	{2,3}
100	{1}
101	{1,3}
110	{1,2}
111	{1,2,3}

Table 1: Generation of Subsets

Informally, the algorithm for generation of subsets [1,2] is given asfollows:**Step 1**: For *n* elements represent sets with an *n*-bit string.

Step 2: For each bit of the *n*-bit string, perform the following operation:

$$\begin{cases} 1 & \textit{include the element} \\ 0 & \textit{exclude the element} \end{cases}$$

Step 3: Print the resulting subsets $0 \rightarrow 2^{n-1}$, which represents the power set of *n* elements. Thus, the algorithm for generating subsets becomes a sort of a counting algorithm. Informally, thecounting can be said as follows:

- **Complexity Analysis**

The complexity analysis is $O(n\ 2^{n})$.

3.2 Summary

In short, one can conclude as part of this unit that

- Brute force guarantee solution but inefficient. Divide and conquer is effective.

- Closest Pair problem is important problem and can be solved by divide and conquerstrategy.
- Convex hull algorithm can be solved effectively using divide and conquer strategy.
- Divide and conquer is effective in implementing Fourier Transform.
- DFT is important problem and can be solved by divide and conquer strategy.
- Polynomial multiplication and convolution are some of the examples of applications ofFFT.
- Decrease and conquer guarantee solution and effective.
- Insertion sort and Topological sort are important problems and can be solved by decreaseand conquer strategy.
- Generating Permutations and subsets are important problems that can be solvedeffectively using decrease and conquer strategy.

Check Your Progress

Fill in the Blanks.

Q.1: Permutation is an arrangement of objects in a ________________.

Q.2: Worst-case complexity for generating permutations is

Q.3: The objective of topological sort is to produce an ordering that implies a ______________on the elements.

3.3 Answer to Check Your Progress

Ans 1. linear order

Ans 2. $\Omega(n!)$

Ans 3. partial ordering

3.4 References

1. *S.Sridhar , Design and Analysis of Algorithms , Oxford University Press, 2014.*
2. *A.Levitin, Introduction to the Design and Analysis of Algorithms, Pearson Education,New Delhi, 2012.*
3. T.H.Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press,Cambridge, MA 1992.
4. URL: https://en.wikipedia.org/wiki/Joseph_Fourier

3.5 Model Questions

1. What do you understand by divide and conquer technique with example?
2. Explain Closest Pair problem in brief.

3. What do you understand by convex hull and quick hull?
4. What do you understand by merge hull?
5. What do you understand by tiling problem?
6. Explain Fourier Transform in brief.
7. What do you understand by insertion and topological sorting?

BLOCK 2

UNIT-7 Transform and Conquer Design paradigm

1.1 Learning Objectives
1.2 Transform and Conquer Design paradigm
1.3 Variations of Transform and Conquer
1.4 Presorting
1.4.1 Finding unique elements in an array
1.4.2 Search using Presorting
1.4.3 Mode
1.5 Transform and conquer approach
1.6 Matrix Operations
1.6.1 Gaussian elimination method
1.7 Answer to Check Your Progress
2.1 Transform and Conquer Design paradigm
2.1.1 Variations of Transform and Conquer
2.2 Gaussian Elimination method for LU Decomposition
2.2.1 Recursive procedure
2.2.2 LUP decomposition
2.3 Crout's Method of Decomposition
2.4 Finding Inverse of a matrix
2.5 Finding Determinant of a matrix
2.6 Answer to Check Your Progress
3.1 More Transform and Conquer Design problems
3.1.1 Variations of Transform and Conquer
3.2 Polynomial Evaluation
3.3 Faster Exponentiation
3.4 Right – to – Left Computation
3.5 Summary
3.6 Answer to Check Your Progress
3.7 References
3.8 Model Questions

1.1 *Learning Objectives*

- To explain basics of Transform and Conquer
- To illustrate simple examples like presorting and unique elements
- To understand matrix operations and Gaussian Elimination
- To explain Matrix Decomposition
- To explain Gaussian Elimination for matrix decomposition
- To explain Crout Procedure for matrix decomposition
- To explain polynomial evaluation using Horner's method and faster exponentiation
- To understand problem reduction method

1.2 *Transform and Conquer Design paradigm*

Transform and conquer is a design paradigm where a given problem is transformed to another domain. This can be done for familiarity of simplicity. The problem is solved in the new domain. Then, the solutions are converted back to the original domain. In short, transform and conquer proposes two stage solution

1. First stage involves the transformation to another problem that is more amenable for solution
2. Second stage involves solving the new problem where the transformed new problem is solved.Then the solutions are converted back to the original problem.

This is illustrated through the following simple examples.

Consider the problem of multiplying two simple numbers XII and IV. These numbers are in Roman number system. As many are not comfortable with Roman number system, this gets transformed to another problem where Arabic numerals are used instead of Roman system.

1. In the first stage, the numbers XII and IV is transformed to another problem of 12 X 4.
2. In the second stage, the actual multiplication is done as 48, then the result is converted toRoman number as XLVIII.

The advantage is the familiarity of the Arabic numeral system over Roman system

Let us consider another problem of convolution of two signals in spatial domain. Convolution involves shifting and adding which is complex. This is equivalent to simple multiplication in frequency domain. The two stage solution is given as follows:

1. In the first stage, the problem is transformed to another domain where spatial data is convertedto frequency domain. This is done using FFT transform.
2. In the second stage, the transformed problem is solved by multiplication and transformed backto spatial domain using Inverse transform.

Another good example of transform and conquer technique is finding LCM using GCD. For example, if GCD of two numbers is available, then LCM can be obtained as follows:

$$lcm(m,n) = \frac{m \times n}{GCD(m,n)}$$

1.3 Variations of Transform and Conquer

Three variants of transform and conquer are as follows:

- Instance Simplification
- Representation Change
- Problem Reduction

- **Instance simplification** is one variant where the problem transformed to the same problem of simpler or convenient instance. The illustrated example of roman number to Arabic number system is an example of instance simplification.
- **Representation Change** is another variety where the strategy involves the transformation of an instance to a different representation. But this is done without affecting the instance. Theillustrated example of roman number to Arabic number system is an example of instance simplification.

- **Problem reduction** is a strategy that involves a transformation of a problem A to another type of problem B. It is assumed that the solution of problem B already exists. The illustrated example of reduction of computing LCM (Last Common Multiple) in terms of GCD is an example of problem reduction. s GCD. Hypothetically, let us assume that an algorithm exists only for GCD.

1.4 Presorting

Sorting an array before processing is called presorting. Presorting is helpful in various applications such as search and in finding element uniqueness.

1.4.1 Finding unique elements in an array

Consider a problem of finding element uniqueness in an array. The problem can be defined as follows: Example: *Given a random list of numbers, determine if there are any duplicates.*

- **Brute force Algorithm**

A brute force algorithm involves checking ever pair of elements for duplicated. This means comparing an element with all other elements of an array. The informal brute force algorithm for element uniqueness is given as follows:

- **Algorithm Uniqueness**

```
for each x  A
    for each y  {A-x}
        if x=y then
            return not unique
        endif
return unique
```

The complexity of this approach is $\theta(n^2)$.

- **Transform and Conquer approach**

One can apply the principle of instance simplification here. One can sort the array instead. The advantage of sorting here is that only the adjacent elements to be checked. So the algorithm can be stated informally as follows:

1. Sort the numbers.

2. Check the adjacent numbers. If the numbers are same then return uniqueness as false.
3. End.

The formal algorithm is given as follows:

Algorithm Elementuniqueness-Presorting(A[1..n])

%% Input: Array A

%% Output: Unique or not

Begin

Sort A

for i = 1 to n-1

  if A[i] = A[i+1] return not unique

End for

return unique

End

- **Complexity Analysis**

Sorting requires $\theta\left(n\log n\right)$ time. The second step requires at most *n-1* comparisons. Therefore, the total complexity is $\theta\left(n\right)+\theta\left(n\log n\right)$

$$=\theta\left(n\log n\right)$$

1.4.2 Search using Presorting

Presorting can be done for search also. The order in the array allows the usage of binary search. The informal algorithm for binary search is given as follows:

Stage 1 Sort the array by Merge sort

Stage 2 Apply binary search

- **Complexity Analysis**

Sorting requires $\theta(n \log n)$ time. The second step requires at most log(n) time. Therefore the total complexity is $\theta(n) + \theta(n \log n)$.

Therefore, the efficiency of the procedure is $\Theta(n\log n) + O(\log n) = \Theta(n\log n)$

1.4.3 Mode

Mode is defined as the element that occurs most often in a given array. For example, consider the following array of elements, A = [5, 1, 5, 5,5,7, 6, 5, 7,5] . The mode of array A is 5 as 5 is the most common element that appear most. If several and different values occur most often any of them can be considered the mode.

- **Brute force Algorithm**

The brute force algorithm is to scan the array repeatedly to find the frequencies of all elements. Informally, the frequency based approach is given below:

Step 1: Find length of List A

max ← max(A)

Step 2: Set freq[1..max] ← 0

Step3: for each x ∈ A

freq[x] = freq[x] + 1

Step4: mode ← freq[1]

Step 5: for i ← 2 to max

if freq[i] > freq[mode] mode ← i

Step6: return mode

- **Complexity Analysis:**

The complexity analysis of frequency based mode finding algorithm is $\theta(n^2)$.

1.5 Transform and conquer approach

Instead, the array can be sorted and run length of an element can be calculated on the sorted array. Informally, mode finding using presorting is given as follows:

1) Sort the element of an array.
2) Calculate the run length of an element
3) Print the element having longest length
4) Exit.

Formal algorithm based on [2] is given as follows:

```
Sort A
i ← 0

frequency ← 0
while i ≤ n-1

  runlength ← 1;  runvalue ← A[i]
  while i+runlength ≤ n-1  and A[i+runlength] = runvalue
    runlength = runlength + 1

  if runlength > frequency
    frequency ← runlength
    modevalue ← runvalue

  i = i + runlength
return modevalue
```

- **Complexity Analysis:**

Sorting requires $\theta\left(n \log n\right)$ time. The finding of a run length requires only linear time as the array is already sorted. Therefore the worst case performance of the algorithm would be less than the brute-force method.

1.6 Matrix Operations

A matrix is a rectangular table of numbers. In scientific domain, matrices have many uses. Matrix addition and subtractions are relatively easy.

Most of the scientific applications use matrix operations like matrix inverse and matrix determinant. So there is a need to solve these problems. As the computational complexity of these algorithms are high, transform and conquer approach can be used. Gaussian elimination uses transform and conquer approach to solve set of equations. Additionally, it can be used to decompose matrices also. Matrix decomposition is useful for find inverse of a matrix and matrix determinant.

First let us discuss the method of Gaussian elimination in the subsequent section.

1.6.1 Gaussian elimination method

Solving an equation means finding the values of the unknown. A simplest equation is of the

form: $Ax = y$

A solution of this equation is $x = y/A$. but this is true only when (y □ 0 and A is not zero). All values of x

satisfies the equation when y=0. This logic can be extended for two unknowns. Let us consider the following set of equations:

$$A_{11}x + A_{12}y = B_1$$

$$A_{21}x + A_{22}y = B_2$$

The equations can be solved first by finding x as

$$x = (B_1 – A_{12}y) / A_{11}$$

Substituting this in the second equation gives y.

In general, if one plots these two equations as lines, then the intersection of two lines in a single point, then the system of linear equations has a unique solution. If the lines are parallel, then there is no solution. If the lines coincide, then there would be infinite number of solutions.

In many applications, one may have to solve ‘n’ equations with ‘n’ unknowns. The set of linear equations are as below:

$$A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1$$

$$A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2$$

...

$$A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n$$

In matrix form, the above can be represented as

$$Ax = B$$

Gaussian elimination, names after Gauss, uses transform and conquer approach to transform this set ofequations with 'n' unknowns

$$\begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \ldots + a'_{1n}x_n &= B'_1 \\ a'_{22}x_2 + \ldots + a'_{2n}x_n &= B'_2 \\ &\vdots \\ a'_{nn}x_n &= B'_n \end{aligned}$$

The transformation can be represented as follows:

$$Ax = B \rightarrow A'x = B'$$

Here ,

$$A = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \ldots & & & \\ A & A & \ldots & A \end{bmatrix} \quad B = \begin{bmatrix} B_1 \\ B_2 \\ \ldots \\ B \end{bmatrix}$$

$$\lfloor \quad {}_{n1} \qquad {}_{n2} \qquad\qquad {}_{nn} \rfloor \qquad\qquad \lfloor \; {}_{n} \rfloor$$

Gaussian elimination aims to create a matrix with zeros in the lower triangle. This is called upper triangular matrix. This is done by elimination a variable at every stage in all equations. The advantage is that One can solve the last equation first, substitute into the second to last, and can proceed to solve the first one.

To do the manipulations, Gaussian elimination uses some elementary steps. The following elementary steps are used by Gaussian elimination.

1. Exchanging two equations of the system
   In this, two equations are exchanged. For example, consider the following equations.

   x + y = 3

   2x + y = 4

   These equations can be exchanged as

   2x + y =

   4X + y

   = 3

2. Exchanging an equation with non-zero multiples

   Say,

   x + y = 2 can be replaced

   as2x + 2y = 4.

3. Replacing the multiple of one equation to another equation. For example, row 2, R2, can beexpressed as a multiple of row 1, R1, and row 3, R3.

- The **informal algorithm** for Gaussian elimination is given as follows:

Step 1: Write matrix A for a given set of equations

Step 2: Append a vector b to matrix A; this is called an augmentation matrix

Step 3: Apply the elementary operation to reduce the augmented matrix to a triangular form, called an echelon matrix. This is done as follows:

- **3a:** Keep a_{11} as a pivot (reference point) and eliminate all a_{11}'s in other rows of matrix A. A pivot element is the coefficient in a set of equations that is used as a reference point to make all other coefficients in the column equal to zero. For example, in the second row, a_{11} can be eliminated by the factor $\text{row}_2 - (a_{21}/a_{11})$. Here, (a_{21}/a_{11}) is called a multiplier. This operation is applied throughout the row. Using the same logic, a_{11} is eliminated in all other equations.
- **3b:** Similarly, using the multiples of (a_{31}/a_{11}), (a_{41}/a_{11}), …, (a_{n1}/a_{11}), matrix A can be reduced to an upper-triangular matrix A'.

This illustrated in Example 1 [1].

Example 1

- **Solve the set of equations using Gaussian Elimination method**

$$3x_1 + x_2 + x_3 = 11$$

$$6x_1 + 4x_2 + x_3 = 29$$

$$x_1 + x_2 + x_3 = 7$$

In matrix notation, this can be written it as $Ax = b$. *Augment the equation as below and apply the elementary operations as shown below:*

$$\left|\begin{array}{cccc} 3 & 1 & 1 & 11 \\ 6 & 4 & 1 & 2 \\ 1 & 1 & 1 & 7 \end{array}\right| \begin{array}{l} \\ row2 - \frac{6}{3} row1 \\ row3 - \frac{1}{3} row1 \end{array}$$

$$\left|\begin{array}{cccc} 3 & 1 & 1 & 11 \\ 0 & 2 & -1 & 7 \\ 0 & 2|3 & 2|3 & 2|3 \end{array}\right| \begin{array}{l} \\ row2 - 2row1 \\ row3 - \frac{1}{3} row1 \end{array}$$

$$\left|\begin{array}{cccc} 3 & 1 & 1 & 11 \\ 0 & 2 & -1 & 7 \\ & & & \end{array}\right|$$

$$\begin{matrix} 0 & 0 & 1 & 1 \end{matrix} \quad row3 - \frac{1}{3} row2$$

$\therefore x_3 = {}^1$ from the last equation. Using this , one can substitute in the second equation to get,

$$2x_2 - x_3 = 7$$

$$2x_2 = 8$$

$$x_2 = 4$$

Using these two variables, the remaining variable can be obtained as follows:

$$3x_1 + x_2 + x_3 = 11$$

$$3x_2 + 5 = 11$$

$$x_2 = 11\text{-}5/3 = 6/3 = 2$$

Apply the elementary operation to reduce the augmented matrix to a triangular form called echelon matrix. So the idea is to keep a_{11} as pivot and eliminate all a_{11} in other equations. For example, in the second equation, a_{11} can be eliminated by the factor R2-(a_{21}/a_{11}). This operation is applied throughout the equation. Using the same logic, a_{11} is eliminated in all other equations. Similarly, using the multiple of (a_{31}/a_{11}), (a_{41}/a_{11}), …, (a_{n1}/a_{11}), the matrix A can be reduced to a upper triangular matrix A'. Then, the solution can be obtained by back substitution. The algorithm for forward elimination is given as follows:

Thus the reduction is done [2,3] is shown formally as follows:

```
for i ← 1 to n do A[i,n+1] ← B[i]
for i ← 1 to n – 1
  for j ← i+1 to n do
    for k ← i to n+1 do
      A[j,k] ← A[j,k] – A[i,k]*A[j,i] / A[i,i]
```

The backward substitution is given as follows:

```
for j ← n to 1 step -1 do
    t ← 0
    for k ← j +1 to n do
        t ← t + A[j, k] * x[k]
    x[j] ← (A[j, n+1] - t) / A[j, j]
```

- **Complexity Analysis**

How many operations are required for Gaussian elimination? One division and n multiplication/division is required. So (n+1) operations for (n-1) rows, requires $(n-1)(n+1) = n^2 - 1$ operations to eliminate the first column of matrix A. Similarly the second row involves $(n-1)^2 - 1$operations. So all n rows, the numbers of operations are

$$\sum_{k=1}^{n} k^2 - 1 = \frac{n(n+1)(2n+1)}{6} - 1$$

$$= \frac{n(n-1)(2n+5)}{6}$$

$$\approx O(n^3)$$

So Gaussian elimination method time complexity is $O(n^3)$.

Check Your Progress

Fill in the Blanks.

Q.1: Sorting an array before processing is called____________.

Q.2: Gaussian elimination method time complexity is_________.

Q.3: Mode is defined as the element that occurs _________in a given array.

Q.4: A ______________involves checking ever pair of elements for duplicated.

1.6 Answer to Check Your Progress

Ans 1: presorting

Ans 2: $O(n^3)$

Ans 3: most often

Ans 4: brute force algorithm

2.1 Transform and Conquer Design paradigm

Transform and conquer is a design paradigm where a given problem is transformed to another domain. This can be done for familiarity of simplicity [2,3]. The problem is solved in the new domain. Then, the solutions are converted back to the original domain. In short, transform and conquer proposes two stage solution

1. First stage involves the transformation to another problem that is more amenable for solution
2. Second stage involves solving the new problem where the transformed new problem is solved.Then the solutions are converted back to the original problem.

2.1.1 Variations of Transform and Conquer

Three variants of transform and conquer are as follows:

- Instance Simplification
- Representation Change
- Problem Reduction

- **Instance simplification** is one variant where the problem transformed to the same problem ofsimpler or convenient instance.

LU decomposition is an example of instance simplification. In LU decomposition, the given matrix is split or decomposed to two matrices L and U. Why? This splitting helps to solve a set of simultaneous equations faster. In other words, LU Decomposition is another method to solve a set of simultaneous linear equations effectively. Thus, the non-singular matrix $[A]$, can be written as $[A] = [L][U]$, Here,

$[L]$ = lower triangular matrix

$[U]$ = upper triangular matrix

Let the set of simultaneous equations are represented as follows as discussed in the last module in matrix form as:

$$Ax = b$$

Substituting A = LU gives,

$$LUx = b$$

First, let y = Ux. So by keeping

$$Ly = b$$

One can solve for y. Then, by solving $Ux = y$,

One can solve for the unknown x in the set of linear equations. By matrix decomposition, theprocess of computing becomes faster.

2.2 *Gaussian Elimination method for LU Decomposition*

Gaussian elimination method was discussed in the last module. It can be noted that the matrix U is the same as the coefficient matrix at the end of the forward elimination step and the matrix L isobtained using the multipliers that were used in the forward elimination process.

One has to know the limitations of LU decomposition. They are listed below [1] :

1. Not all the matrices have LU decomposition
2. Rows or columns can be swapped to make LU decomposition feasible
3. LU decomposition is guaranteed if the leading submatrices have non-zero determinants. to make LU decomposition. A matrix Ak is called a leading submatrix of matrix A, if it isk x k matrix whose elements are top k rows and k left-most columns.

The following Example 1 illustrates the Gaussian elimination method for solving a set of equations and to find LU decomposition.

Example 1: Solve the set of equations using Gaussian Elimination method and find LU decomposition.

$$3x_1 + x_2 + x_3 = 11$$

$$6x_1 + 4x_2 + x_3 = 29$$

$$x_1 + x_2 + x_3 = 7$$

Solution:

The first step is to augment the matrix. The above matrix can be augmented as follows:

$$A = \begin{bmatrix} 3 & 1 & 1 & 11 \\ 6 & 4 & 1 & 29 \\ 1 & 1 & 1 & 7 \end{bmatrix}$$

Gaussian elimination can be applied now to get upper triangular matrix.

3 l 11 |

–

$$\begin{matrix} 6 & 1 & 2 \\ 1 & 1 & 7 \end{matrix} \left| \begin{matrix} row2 - \frac{6}{3} row1 \\ row3 - \frac{1}{3} row1 \end{matrix} \right.$$

$$\left| \begin{matrix} 3 & 1 & 1 & 11 \\ 0 & 2 & -1 & 7 \\ 0 & 2/3 & 2/3 & 2/3 \end{matrix} \right| \begin{matrix} \\ row2 - 2 \times row1 \\ row3 - \frac{1}{3} \times row1 \end{matrix}$$

$$\left| \begin{matrix} 3 & 1 & 1 & 11 \\ 0 & 2 & -1 & 7 \\ 0 & 0 & 1 & 1 \end{matrix} \right| \begin{matrix} \\ \\ row3 - \frac{1}{3} row2 \end{matrix}$$

The lower-triangular matrix is obtained *L* is made up of *1*'s in the diagonal and the multipliers used for row reduction in the Gaussian elimination. It can be observed that the multipliers usedin the above Gaussian elimination process is used to give matrix L.

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/3 & 1/3 & 1 \end{bmatrix}$$

The upper triangular matrix is made up of elements that are the resultant of the Gaussian elimination process. It can be seen that, the resultant of the Gaussian elimination matrix with thediagonal 1's give the following matrix U.

$$U = \begin{bmatrix} 3 & 1 & 1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

Using the matrices L and U, one can easily solve the simultaneous linear set of equations.The equation $Ax = b$ is equivalent to

$$LUx = b$$

Let $y = Ux$

$$\therefore Ly = b$$

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/3 & 1/3 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 11 \\ 29 \\ 7 \end{bmatrix}$$

This yield the solutions as follows:

$$y_1 = 11, \quad 2y_1 + y_2 = 29 \Rightarrow y_2 = 29 - 2y_1 = 7 \text{ and finally,}$$

$$\tfrac{1}{3} y_1 + \tfrac{1}{3} y_2 + y_3 = 7$$

$$\tfrac{11}{3} + \tfrac{7}{3} + y_3 = 7$$

$$y_3 = 7 - \tfrac{11}{3} - \tfrac{7}{3} = 1$$

having obtained y's, now the unknown x's can be obtained as follows:Solving $Ux = y$ implies

$$\begin{bmatrix} 3 & 1 & 1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 11 \\ 7 \\ 1 \end{bmatrix}$$

$\therefore$ The final solution is

$$x_3 = 1 \text{ ; } 2x_2 - x_3 = 7, \quad 2x_2 = 7 + 1 \Rightarrow x_2 = 4 \text{ ; and finally,}$$

$$3x_1 + x_2 + x_3 = 11$$

$$3x_1 + 4 + 1 = 11$$

$$3x_1 = 6 \Rightarrow x_1 = 2$$

Therefore, the unknowns x_1, x_2 and x_3 are respectively 2, 4 and 1. It can be observed that the LU decomposition simplifies the computational effort.

2.2.1 Recursive procedure

One can automate the above using a recursive procedure. The idea is to automate the above said procedure using matrix decomposition as shown below:

Let us assume that $A = LU$; matrices A, L, and U can be partitioned as follows:

$$\begin{pmatrix} \alpha_{11} & a_{12}^T \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \ell_{21} & L_{22} \end{pmatrix} \times \begin{pmatrix} v_{11} & u_{12}^T \\ 0 & v_{22} \end{pmatrix} = \begin{pmatrix} v_{11} & u_{12}^T \\ \ell_{21} \times v_{11} & \ell_{21} \times u_{12}^T + L_{22}U_{22} \end{pmatrix}$$

The informal algorithm is

1. Find α_{11}
2. Update the value for a_{21} by dividing it with pivot value
3. $a_{12}^T = u_{12}^T$
4. Update the value of A_{22} recursively.

This is illustrated in the following example.

Example 2: Find Gaussian Elimination for the given matrix using recursive procedure

$$A = \begin{pmatrix} 3 & 1 & 1 \\ 6 & 4 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Solution:

Using $a_{11} = 3$, divide the entire column by 3. This gives the column $\begin{pmatrix} 2 \\ \frac{1}{3} \end{pmatrix}$. The matrix $A_{22} = \ell_{21} \times u_{12}^T + L_{22} U_{22}$ can be updated as $\begin{pmatrix} 4 & 1 \\ 1 & 1 \end{pmatrix} - \begin{pmatrix} 2 \\ \frac{1}{3} \end{pmatrix} \times \begin{pmatrix} 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ \frac{2}{3} & \frac{2}{3} \end{pmatrix}$. This results in a matrix

$$\begin{pmatrix} 3 & 1 & 1 \\ 2 & 2 & -1 \\ \frac{1}{3} & \frac{2}{3} & \frac{2}{3} \end{pmatrix}$$

Now, by choosing 2 as the pivot element and diving that by the entire column and computing A_{22}, $A_{22} = \left(\frac{2}{3}\right) - \left(\frac{1}{3}\right) \times (-1) = 3$, gives the matrix

$$\begin{pmatrix} 3 & 1 & 1 \\ 2 & 2 & -1 \\ \frac{1}{3} & \frac{1}{3} & 1 \end{pmatrix}$$

This yields, the following matrices L and U,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{3} & \frac{1}{3} & 1 \end{pmatrix} \text{ and } U = \begin{pmatrix} 3 & 1 & 1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix}.$$

One can easily verify the LU decomposition as follows:

$$\begin{matrix} A & = & L & \times & U \end{matrix}$$

$$\begin{pmatrix} 3 & 1 & 1 \\ 6 & 4 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{3} & \frac{1}{3} & 1 \end{pmatrix} \times \begin{pmatrix} 3 & 1 & 1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

It can be observed that the product of L and U gives the original matrix as it is. Therefore, theLU decomposition is correct.

2.2.2 LUP decomposition

LUP is an extension of LU decomposition. Here the matrix A is decomposed to three matrices L,U and P. P is a permutation matrix. So,

$$P A = L U$$

P is a permutation matrix. In case, if rows or columns needs to be swapped, then this can be avoided by the usage of permutation matrix. In that case, the orderings can be recorded using matrix P. Hence,

$$Ax = b$$

can be written as

$$P Ax = Pb$$

A can be decomposed as LU. P is just a permutation matrix used for swapping rows and columns. So, one can write

$$LUx= Pb$$

Now the above equation is solved as previous methods

1. Define y=Ux and solve for Solve Ly = Pb for unknown y. This process is called forward substation.
2. Solve the unknown x by using the equation Ux = y. This is called backward substitution.

In the absence of partial pivoting, LUP decomposition is almost similar to LU decomposition.

2.3 Crout's Method of Decomposition

Crout's method [1] is another method of LU decomposition similar to Gaussian elimination based LU decomposition. The formula for crout procedure can directly be derived. Let us consider a following 3×3 matrices as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

It can be observed that 3 X 3 matrix is expressed as a product of a lower triangular and upper triangular matrix. Perform the normal multiplication of LU. This yields the following expression.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & (l_{11}u_{12}) & (l_{11}u_{13}) \\ l_{21} & (l_{21}u_{12} + l_{22}) & (l_{21}u_{13} + l_{22}u_{23}) \\ l_{31} & (l_{31}u_{12} + l_{32}) & (l_{31}u_{13} + l_{32}u_{23} + l_{33}) \end{bmatrix}$$

Equating these two matrices, one can directly compute the first columns.

$$l_{11} = a_{11}; l_{21} = a_{21}; l_{31} = a_{31};$$

Then using these values, one can compute the second column as follows;

$$l_{11}u_{12} = a_{12},$$

$$\therefore u_{12} = \frac{a_{12}}{l_{11}} = \frac{a_{12}}{a_{11}};$$

$l_{21}u_{12} + l_{22} = a_{22},$

$\therefore l_{22} = a_{22} - l_{21}u_{12}$

$l_{31}u_{12} + l_{32} = a_{32},$

$\therefore l_{32} = a_{32} - l_{31}u_{12}$

From the above values, one can easily compute the third column

$l_{11}u_{13} = a_{13},$

$\therefore u_{13} = \frac{a_{13}}{l_{11}} = \frac{a_{13}}{a_{11}}$

$l_{21}u_{13} + l_{22}u_{23} = a_{23},$

$\therefore u_{23} = \frac{a_{23} - l_{21}u_{13}}{l_{22}}$

$l_{31}u_{13} + l_{32}u_{23} + l_{33} = a_{33},$

$\therefore l_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23}$

Using these values, one can find matrices L and U respectively. This is illustrated in the following Example 3.

Example 3.

Perform LU decomposition of the following matrix using Crout method.

$$\begin{pmatrix} 3 & 1 & 1 \\ 6 & 4 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Solution:

This is a 3 X 3 matrix. So comparing this matrix with

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

Once can easily derive the values as below:

$l_{11} = a_{11} = 3$, $l_{21} = a_{21} = 6$ **and** $l_{31} = a_{31} = 1$

$$u_{12} = \frac{a_{12}}{a_{11}} = \frac{1}{3};\ l_{22} = a_{22} - l_{21}u_{12} = 4 - 6\times\frac{1}{3} = 2;\ l_{32} = a_{32} - l_{31}u_{12} = 1 - \frac{1}{3} = \frac{2}{3}$$

$$u_{13} = \frac{a_{13}}{a_{11}} = \frac{1}{3};\ u_{23} = \frac{a_{23} - l_{21}u_{13}}{l_{22}} = \frac{1 - 6\times\frac{1}{3}}{2} = -\frac{1}{2};\ l_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23} = 1 - \frac{1}{3} + \frac{1}{3} = 1$$

So arranging this in lower and upper triangular matrix, one gets

$$L=\begin{pmatrix} 3 & 0 & 0 \\ 6 & 2 & 0 \\ 1 & \frac{2}{3} & 1 \end{pmatrix} \text{ and } U=\begin{pmatrix} 1 & \frac{1}{3} & \frac{1}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}.$$

One can check that the above decomposition is true as by multiplying L and U, one gets theoriginal matrix as it is.

2.4 Finding Inverse of a matrix

What is an inverse matrix? A matrix A^{-1} is said to the inverse of matrix A, if the following condition holds good.

$$AA^{-1}=I$$

Here, the matrix I is identity matrix. In other words, the inverse [B] of a square matrix [*A*] isdefined as

$$[A][B] = [I] = [B][A]$$

There is no guarantee that the inverse matrix exists for all matrices [2,3]. If the inverse matrix does not exist for a matrix A, then A is called singular. It must be noted that matrix inverse does not exist for all matrices and not all square matrices have matrix inverse. Matrix inverse is usefulfor finding the unknown x of the equation Ax=y as

$$x=A^{-1}\times b \text{ if } A^{-1} \text{ exists.}$$

To find the inverse of a matrix, one can write

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{n1} & a_{n2} & & a_{nn} \end{pmatrix}\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ x_{n1} & x_{n2} & & x_{nn} \end{pmatrix}=\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \\ 0 & 0 & & 1 \end{pmatrix}$$

This can be written as

$$Ax_j = e_j$$

Where x_j and e_j are the vectors of unknowns in the jth column of the matrix inverse.

One can also use Gaussian process for decomposing the matrix A as A = LU and solving the equations. This will be useful for finding unknowns faster. The following Example 4 illustrates the use of Gaussian elimination in the process of finding the determinant.

Example 4. Find the inverse matrix for the following matrix using Gaussian eliminationmethod.

$$\begin{pmatrix} 3 & 2 \\ 2 & 7 \end{pmatrix}$$

Solution:

Let $A = \begin{pmatrix} 3 & 2 \\ 2 & 7 \end{pmatrix}$,

Augment this matrix with

$$[A|\ I]= \left[\begin{array}{cc|cc} 3 & 2 & 1 & 0 \\ 2 & 7 & 0 & 1 \end{array}\right]$$

Apply Gaussian Elimination method as follows:

$$[A|\ I]= \left[\begin{array}{cc|cc} 3 & 2 & 1 & 0 \\ 0 & \frac{17}{3} & \frac{-2}{3} & 1 \end{array}\right] R_2 \to R_2 - \frac{2R_1}{3}$$

Since the matrix is of order 2 x 2, the matrix inverse would be

$$A^{-1} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}$$

The first column of the matrix inverse is obtaine as follows:

$$\begin{bmatrix} 3 & 2 \\ 0 & \frac{17}{3} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{-2}{3} \end{bmatrix}$$

$$\frac{17}{3} x_{21} = \frac{-2}{3}$$

$$x_{21} = \frac{-2}{17}$$

and

$$3x_1 + 2x_{21} = 1$$

$$3x_{11} + \frac{2 \times -2}{17} = 1$$

$$3x_{11} = 1 + \frac{4}{17} = \frac{21}{17}$$

$$x_{11} = \frac{21}{17} \times \frac{1}{3} = \frac{7}{17}$$

Similarly the second column of the matrix inverse is obtained as follows:

$$\begin{bmatrix} 3 & 2 \\ 0 & \frac{17}{3} \end{bmatrix} \begin{bmatrix} x_{12} \\ x_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\frac{17}{3} x_{22} = 1$$

$$x_{22} = \frac{3}{17}$$

$$3x_{12} + 2x_{22} = 0$$

$$3x_{12} = \frac{-2 \times 3}{17}$$

$$x_{12} = \frac{-2}{17}$$

By substituting these values in the matrix inverse, one can get the inverse matrix as follows:

$$\therefore \quad A^{-1} = \begin{bmatrix} \frac{7}{17} & \frac{-2}{17} \\ \frac{-2}{17} & \frac{3}{17} \end{bmatrix} = \frac{1}{17}\begin{bmatrix} 7 & -2 \\ -2 & 3 \end{bmatrix}$$

One can verify that the matrix inverse is correct by multiplying the matrix and the matrix inverseobtained.

It is equal to matrix I.

2.5 Finding Determinant of a matrix

Finding determinant is one of the most frequently encountered operation in scientific

applications. For a matrix A, such as this

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The determinant D is denoted as

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

and is computed as follows:

$$D = a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

As order of the matrix increases, finding determinant is very difficult. The method of finding determinant can be converted to another problem of finding the determinant using Gaussian elimination. The process of finding the determinant using Gaussian elimination is as follows:

1. Decompose A = LU
2. Find the determinant,

$$As,\ A = LU$$
$$|A| = |L||U|$$

3. The determinant of L is 1. The determinant of the matrix U is the product of its diagonals.

This is illustrated in the following Example 5.

Example 5. Find determinant of the following matrix

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

Solution:

Using Example 1, one can find L and U as follows:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1/3 & 1/3 & 1 \end{bmatrix}$$

and

$$U = \begin{bmatrix} 3 & 1 & 1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, Determinant of matrix A is given as follows:

$$|A| = |LU| = |L||U|$$

$$|L| = 1$$

$|U|$= product of diagonal elements

= 3 X 2 X 1 = 6

Therefore, the determinant of the matrix is 6.

Check Your Progress

Fill in the Blanks.

Q.1: What design paradigm involves breaking down a problem into smaller, more manageable subproblems, solving them, and then combining the solutions to solve the original problem?
a) Divide and Conquer
b) Transform and Conquer
c) Dynamic Programming
d) Greedy Algorithms

Q.2: In the Gaussian Elimination method for LU decomposition, which of the following is NOT a part of the process?
a) Forward elimination
b) Backward substitution
c) Row pivoting
d) Matrix transposition

Q.3: What is the recursive procedure involved in Gaussian Elimination for LU decomposition mainly used for?
a) Solving linear systems of equations
b) Finding eigenvalues of a matrix
c) Matrix multiplication
d) Calculating matrix determinants

Q.4: In LUP decomposition, what does the 'P' matrix represent?
a) The lower triangular matrix
b) The upper triangular matrix
c) A permutation matrix
d) The original input matrix

Q.5: Crout's Method of Decomposition is used for:
a) Finding the eigenvalues of a matrix
b) Solving systems of linear equations
c) Matrix factorization
d) Calculating matrix determinants

2.6 Answer to Check Your Progress

Ans 1: Transform and Conquer
Ans 2: Matrix transposition
Ans 3: Solving linear systems of equations

Ans 4: A permutation matrix
Ans 5: Matrix factorization

3.1 More Transform and Conquer Design problems

Transform and conquer is a design paradigm where a given problem is transformed to another domain. This can be done for familiarity of simplicity [2,3]. The problem is solved in the new domain. Then, the solutions are converted back to the original domain. In short, transform and conquer proposes two stage solution

1. First stage involves the transformation to another problem that is more amenable for solution
2. Second stage involves solving the new problem where the transformed new problem is solved. Then the solutions are converted back to the original problem.

3.1.1 Variations of Transform and Conquer

Three variants of transform and conquer are as follows:

- Instance Simplification
- Representation Change
- Problem Reduction

Instance simplification is one variant where the problem transformed to the same problem ofsimpler or convenient instance.

- **Change of Representation**

Change of representation is another useful variant of transform and conquer design paradigm. Aspart of this technique, two examples are discussed as part of the problem.

3.2 Polynomial Evaluation

One of the common problem is scientific domain is the evaluation of polynomial. A polynomialis given as follows:

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Often, for given x, the polynomial needs to be evaluated. The brute force method of evaluation isgiven as follows:

```
Algorithm evaluation( a[0..n], x)
%% Input: A polynomial A with n+1 coefficients
%% Output: Result of the polynomial
begin
        first = a[0];
        second = a[1];
```

```
result = first + second
for i = 2 to n do
    xpower = xpower * x;
    result = result + a[i] *  xpower
end for
End.
```

An alternative way of solving this problem is to use transform and conquer strategy using Horner method. Horner method uses representation change for faster computation. The polynomial can be represented as follows:

$$a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$$

$$= a_0 + \left(a_1 + \left(a_2 x + \dots + a_n x^{n-1}\right) x\right)$$

$$= a_0 + \left(a_1 + \left(a_2 + \dots + a_n x^{n-2}\right) x\right) x$$

$$\vdots$$

$$= a_0 + \left(a_1 + \left(a_2 + \dots + \left(a_{n-1} + a_n x\right) x\right) \dots\right) x$$

Hence, the informal algorithm is given as follows:

1. Change the polynomial representation
2. Compute the list of powers of x by taking the advantage of the change of presentation.
3. Compute the list of powers and evaluate the polynomial
4. Print the result and exit.

The formal Horner's algorithm is given as follows:

Algorithm Horner(a[0..n), x)

```
Begin
    s = a_n
    for i = 1 to n
      s = a_{n-i} + s × x
    end for
  return s
End.
```

The following Example 1 illustrates Horner's method.

Example 1 : Let $f(x) = 2 + 3x + 4x^2 + 7x^3$

Let $x = 2$ and apply Horner's method and show the intermediate results.

Solution

Here $a_0 = 2$, $a_1 = 3$, $a_2 = 4$ and $a_3 = 7$. *On applying the conventional method, the polynomialcan be evaluated as follow*:

Initialize s = a_3

Let i = 1,

$s = a_{3-1} \times x$

$= a_2 + s \times x$

= 4 + 7×2=18

Let i = 2,

$s = a_{3-2} \times x$

$= a_1 + s \times x$

= 3 + 18×2=39

Let i = 3,

$s = a_{3-3} \times x$

$= a_0 + s \times x$

= 2 + 39×2=80

As i = 3, the algorithm terminates.

One can verify this by substituting this in the equation

$$f\,(x) = 2 + 3x + 4x^2 + 7x^3$$

$$\therefore f\,(2) = 2 + 3 \times 2 + 4 \times \left(2^2\right) + 7\left(2^3\right)$$

= 2 + 6 + 16 + 56

= 80

- **Complexity Analysis**

As Horner method involve only one loop, and inside the loop, only addition and multiplication isinvolved. Therefore, the complexity of the algorithm is O(n).

3.3 Faster Exponentiation

Exponentiation is the problem of finding x^n. The traditional algorithm for solving this is givenbelow:

```
Algorithm Exp(x, n)if
    n = 1 then
        return x
    else
        return x × Exp(x,n-1)
    end if
end
```

- **Complexity Analysis**

What is the complexity of this algorithm? The complexity is O(n) as the algorithm requires (*n-1*) multiplications. If the exponent is large, the number of multiplications required would increase considerably. Therefore, a better solution can be provided for finding exponentiation using transform and conquer method.

- **Idea of Faster Exponentiation:**

The concept of fast exponentiation was given in a Hindu literature Chandah Sutra in 200 BC. x^2 can be computed with one multiplication. If x^4 needs to be computed, it can be computed as $(x^2)^2$. In this case, only two multiplications are required instead of traditional four multiplications. Similarly, x^8 can be computed as $((x^2)^2)^2$ with three multiplications. In general, the faster exponentiation can be done using log n computations.The

informal algorithm for computing is given as follows:

1. Represent the exponent as a binary pattern P. Let the maximum number of bits be p.

2. Scan the binary bits of pattern P.

$$x_n = \begin{cases} (x^p)^2 & \text{if } b_i = 0 \\ (x^p)^2 \times x & \text{if } b_i = 1 \end{cases}$$

3. Compute and print the result.

This is illustrated in the following Example 2.

Example 2:

Compute x^{19} using binary exponentiation left-to right technique.

Solution

The binary representation of 19 is $(1\ 0\ 0\ 1\ 1)$. So x^{19} would be computed as shown in Table 1.

Table 1: Fast Exponentiation Table

Binary Digits	1	0	0	1	1
Product accumulator	x	$(x)^2$	$((x)^2)^2 = x^4$	$x \times (x^4)^2$	$x \times (x^9)^2 = x^{19}$

Therefore, it can be verified that $x^{19} = x^{18} \times x$. Thus, the problem is solved faster.

The formal algorithm for left-to-right binary exponentiation is given as follows;

```
Algorithm left-to-right_binary-exponentiation(x ,n)
%% Input: x and n
%% Output; Result of x^n
Begin
    Product = x
    Find binary of n as vector b For i
    =  length(b) downto 0 do
      Case (b_i) of b
        b_i = 0: product = product × product
        b_i = 1: product =  x  × product
      End case
    End for
    Return Product
 End.
```

3.4 *Right – to – Left Computation*

The binary exponentiation method can be done right-to-left also. This idea was proposed by anArab mathematician Al-Kashi in 1427. In this case, the procedure would be as follows:

$$x^n = \begin{cases} x^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$$

The informal algorithm is given as follows:

1. Represent n as a binary pattern b.
2. Scan the binary pattern from right to left and perform

$$x^{b_i 2^i} = \begin{cases} x^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$$

3. Compute terms ×accumulator if $b_i = 1$
4. Print the result
5. End.

This is illustrated using Example 3.

Example 3:

Compute x^{19} using binary exponentiation left-to right technique.

Solution

The binary representation of 19 is (1 0 0 1 1) . So x^{19} computation is shown below in table 2.

Table 1: Binary Exponentiation Table

Binary Digits	1	0	0	1	1
Terms	x^{16}	x^8	x^4	x^2	x
Product accumulator	$x^3 \times x^{16} = x^{19}$	-	-	$x \times x^2 = x^3$	x

The formal algorithm is given as follows:

Algorithm fast-exponentiation (x,n)

Begin

term = x

Find binary of n as vector bfor

i= 1 to length(b)

term = term * termcase

(b_i) of b

$b_i = 0$: term = term * 1

$b_i = 1$: term = x * termEnd

case

End for

Return Product

End.

- **Complexity analysis:**

The complexity of the algorithm depends on the binary representations of the given number. Therefore, the number of multiplications would be between $\lfloor \log_2 n \rfloor$ and 2 $\lfloor \log_2 n \rfloor$. Hence, the complexity of the algorithm is O(log n).

- **Problem Reduction:**

Problem reduction is another variant of this paradigm. Assume that there are two problems A and B. The problem A is unsolvable and problem B has an algorithm. Hence, the logic of problem reduction is the conversion of instance of problem A into instance of problem B using an algorithm. Then the problem A can be solved by invoking the problem solver of B as a subroutine. Then, the result can be converted back. Some of the examples of problem reduction are computation of LCM in terms of GCD, reducing game problems to graph search problems.

3.5 Summary

- Transform and Conquer is an effective design paradigm
- Matrix operations are computationally very intensive
- Gaussian elimination is an effective technique that uses transform and conquer method
- Transform and Conquer is an effective design paradigm
- Matrix Inverse and Matrix Determinant operations are computationally very intensive
- Gaussian elimination can be used to find matrix inverse and matrix determinant easily.
- Representation Change is an effective way to solve problems.
- Horner method uses representation Change.
- Faster exponentiation can be done using transform and conquer method.

Check Your Progress

Multiple Choice Questions.

Q.1: Which design paradigm involves solving a problem by transforming it into a different problem and then conquering the transformed problem?
a) Divide and Conquer
b) Greedy Algorithms
c) Transform and Conquer
d) Dynamic Programming

Q.2: What problem does the "Polynomial Evaluation" algorithm aim to solve?
a) Sorting a list of integers
b) Finding the greatest common divisor of two numbers
c) Evaluating a polynomial at a given value
d) Calculating the determinant of a matrix

Q.3: "Faster Exponentiation" is a technique used to efficiently compute:
a) Factorial of a number
b) Square root of a number
c) Exponentiation of a number to a power
d) Multiplication of two matrices

Q.4: In "Right-to-Left Computation," which arithmetic operation is typically performed from right to left?

a) Addition
b) Subtraction
c) Multiplication
d) Division

Q.5: What is one of the advantages of using the "Transform and Conquer" design paradigm?
a) It always provides the most optimal solution.
b) It simplifies complex problems by breaking them into smaller, more manageable subproblems.
c) It eliminates the need for recursion in algorithms.
d) It is mainly suitable for sorting algorithms

3.6 Answer to Check Your Progress

Ans 1: Transform and Conquer

Ans 2: Evaluating a polynomial at a given value

Ans 3: Exponentiation of a number to a power

Ans 4: Multiplication

Ans 5: It simplifies complex problems by breaking them into smaller, more manageable subproblems.

3.7 References

1. *S.Sridhar , Design and Analysis of Algorithms , Oxford University Press, 2014.*
2. *A.Levitin, Introduction to the Design and Analysis of Algorithms, Pearson Education, New Delhi, 2012.*
3. T.H.Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA 1992.

3.8 Model Questions

1. Explain Transform and Conquer Design paradigm.
2. What do you understand by matrix operations and Gaussian Elimination method?
3. Explain Matrix Decomposition in detail.
4. What do you mean by Gaussian Elimination method?
5. Explain Crout Procedure for matrix decomposition.
6. What do you mean by Horner's method and faster exponentiation?

BLOCK- 2

UNIT- 8

GREEDY ALGORITHM

1.1 Learning Objectives
1.2 Greedy Algorithm
1.3 Applications of Greedy Algorithm
Check Your Progress
1.4 Answers to check your progress
2.1Coin Change Problem
2.1.1 Coin Change Problem using Dynamic Programming Approach
2.1.2 Failure of coin change problem Check Your Progress
2.2 Answers to check your progress
2.3 Scheduling problem
2.3.1 Types of Scheduling problem
2.3.2 Scheduling problem without deadline
Check Your Progress
2.4Answers to check your progress
2.5 References

1.1 Learning Objectives

- To explain Greedy algorithms.
- To explain coin change problem.
- To explain scheduling problems

1.2 Greedy algorithms

In an attempt to obtain a global optimum, a greedy algorithm is a straightforward, understandable computational paradigm that makes locally optimal decisions at each stage. While a greedy heuristic may produce locally optimum solutions that resemble a global optimal solution in a reasonable amount of time, it is not always guaranteed to produce an optimal solution in many problems.

A greedy algorithm's primary notion is to solve the problem as best it can at each step, taking into account only its present state and not caring about how that decision may affect subsequent steps. In optimization issues, where the objective is to identify the best answer from a set of viable solutions, greedy algorithms are frequently employed.

An overview of a greedy algorithm's operation is provided here:

Initialization: Begin with a partially solved problem that has some initial components in it, or start with an empty solution.

Greedy Decision: Choose the best element or alternative accessible at the current step out of greed. A precise criterion that specifies what constitutes the "best" alternative given the circumstances of the problem should be used to guide this decision.

Verify the selected element's feasibility by making sure it complies with the problem's requirements. Add it to the solution if it does. If not, throw it away and think of another choice.

Optimality Check (if applicable): Determine whether the current solution is optimal if the problem calls for an optimal solution. In that case, stop the algorithm. If not, carry out the previous procedures again. If not, carry out steps 2 and 3 again to iteratively improve the answer.

Termination: Until a termination condition is satisfied, keep making avaricious decisions and determining if anything is feasible or ideal. This could mean examining every avenue, arriving at a particular solution size, or meeting particular requirements unique to the situation.

It's crucial to remember that not every issue can be resolved by a greedy algorithm. Greedy algorithms perform effectively in situations where selecting a solution that is locally optimum also results in a globally optimal solution. Although greedy algorithms don't always yield the best results, they can occasionally be utilized to approximate the ideal answer.

Prim's algorithm and the Greedy method for Minimum Spanning Trees are two classic examples of problems handled by greedy algorithms.

The Greedy Algorithm for Huffman Coding, which is used to compress data, is another example. To build the overall best solution in each scenario, the algorithm selects the minimum edge or minimum frequency character, respectively, at each step in a locally optimal manner.

A greedy algorithm is a type of algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. While it may not find the global optimum, greedy algorithms are often simpler and faster while being not too far from the global optimum.

Greedy algorithms are being used in many areas of computer science. They're also used in operations research, engineering, finance, machine learning, bioinformatics, game theory, etc. They've become a well-established technique for solving optimization problems.

The key to developing an effective greedy algorithm is to identify the problem's specific structure and design a heuristic that makes the optimal local choice at each step. While they do have some limitations, there's on-going research to address these limitations.

Greedy algorithms with a simple example:

Suppose you are given an array of positive integers and you want to find the subset of those integers whose sum is as large as possible, but does not exceed a given value. For instance, if you were given the array [3, 4, 5, 6, 8] and the limit 13, the optimal subset would be [8, 5], with a sum of 13.

A greedy algorithm for this problem would be to sort the array in decreasing order, then start selecting the largest numbers that don't exceed the limit, until the limit is reached. So for the example above, the steps of the algorithm would be:

Sort the array in decreasing order: [8, 6, 5, 4, 3]

Initialize an empty subset and a variable to track the current sum: subset = [], sum = 0

Start the iterations: (i) subset = [8], sum = 8; (ii) subset = [8, 5], sum = 13.

Return the subset: [8, 5]

Why should I use greedy algorithms when the solution is not guaranteed to be optimal?

While greedy algorithms don't guarantee an optimal solution, they have their benefits:

Simplicity: Greedy algorithms are often simple to implement and understand, making them a good choice for problems with large inputs or when efficiency is a concern.

Speed: Greedy algorithms are often very fast, especially when compared to more complex algorithms. This makes them a good choice for problems with large inputs.

Approximation: Even though the greedy algorithm does not always guarantee the optimal solution, it can often give a very good approximation of the optimal solution. In many cases, the difference between the optimal solution and the solution found by the greedy algorithm is not significant.

Starting Point: Greedy algorithms can be a good starting point for more complex algorithms. By using a greedy algorithm to quickly find a good solution, you can then refine the solution using other techniques.

1.3 APPLICATIONS OF GREEDY ALGORITHMS

An optimal solution to the coin changing problem is the minimum number of coins whose total value equals a specified amount. For example what is the minimum number of coins (current U.S. mint) needed to total 83 cents.

25 10 5 1

83 - 25 = 58
58 - 25 = 33
33 - 25 = 8
8 - 5 = 3
3 - 1 = 2
2 - 1 = 1
1 - 1 = 0

25 25 25 5 1 1 1

```
A Greedy Algorithm for
     Coin Changing

1. Set remval=initial_value

2. Choose largest coin that
   is less than remval.

3. Add coin to set of coins
   and set
  remval:=revamal-coin_value

4. repeat Steps 2 and 3
   until remval = 0;
```

Here are some examples of greedy algorithms:

Minimum Spanning Tree (MST): MST is useful in network design and transportation planning. Kruskal's and Prim's algorithms are greedy approaches to this problem.

Huffman Coding: This is applied in data compression and transmission. It assigns shorter codes to more frequently occurring characters.

Knapsack Problem: This is considered as a classic optimization problem. It deals with what is the best way to fill a bag of fixed capacity with items of different sizes and values. A greedy algorithm selects items based on their value-to-size ratio.

Activity Selection: In scheduling problems, there's a need to select the maximum number of non-overlapping activities. A simple greedy algorithm solves this by selecting activities based on their finish time.

Shortest Path Algorithms: Dijkstra's algorithm is an example of shortest path algorithm. It selects the shortest path from a given vertex to all other vertices in a graph.

Coin Changing Problem: deals with the minimum number of coins needed to make change for a given amount of money. This is solved by selecting the largest coin possible at each step.

HOW ARE GREEDY ALGORITHMS DIFFERENT FROM DYNAMIC PROGRAMMING?

Greedy algorithms and dynamic programming are two popular techniques for solving optimization problems, but they differ in several key ways.

Greedy algorithms make the locally optimal choice at each step in the hope of finding a global optimal solution. Dynamic programming breaks down a problem into smaller sub problems and then solves them in a bottom-up fashion. It stores the solutions to the sub problems and reuses them to solve the larger problem.

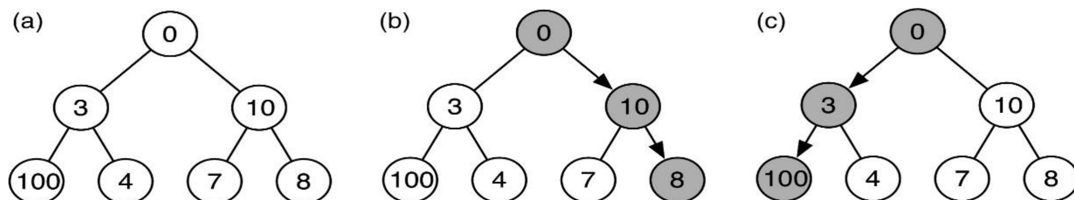
Greedy algorithms may give a sub-optimal solution, whereas dynamic programming always finds the optimal solution. Greedy algorithms typically make choices based only on the current state of the problem, while dynamic programming considers all possible sub problems and their solutions.

Greedy algorithms typically require less memory because they don't need to store the solutions to all possible sub problems.

Greedy algorithms are typically faster due to fewer calculations. However, the time complexity of greedy algorithms can be higher in certain cases.

Greedy algorithms are useful when there is a unique optimal solution. Dynamic programming can solve a wider range of problems, including problems with multiple optimal solutions or overlapping sub problems.

What are some practical limitations of greedy algorithms?



Greedy algorithm (c) is suboptimal and misses the optimal solution (b). Source: Simmons et al. 2019, fig. 1.

Because greedy algorithms make the locally optimal choice at each step, this may not lead to the global optimal solution. In some cases, making a suboptimal choice at an early stage can lead to a better global solution. The figure shows an example in which the objective is to maximize the sum of the nodes on a top-to-bottom path. Greedy algorithm leads to a sub-optimal solution.

Where the objective function has multiple conflicting objectives, or changes non-monotonically over time, greedy algorithms will not work well. The same can be said of problems with complex constraints or a large search space. For these cases, greedy algorithms would incur a larger time complexity as well.

Many optimization problems are NP-hard, which means that finding the optimal solution is computationally intractable. Greedy algorithms are not suitable for solving them, as they can't guarantee the optimal solution in a reasonable amount of time.

What are some recent advances with greedy algorithms?

Adaptive greedy algorithms adjust their choices dynamically based on the problem's structure and input data. They have outperformed traditional greedy algorithms in many real-world optimization problems.

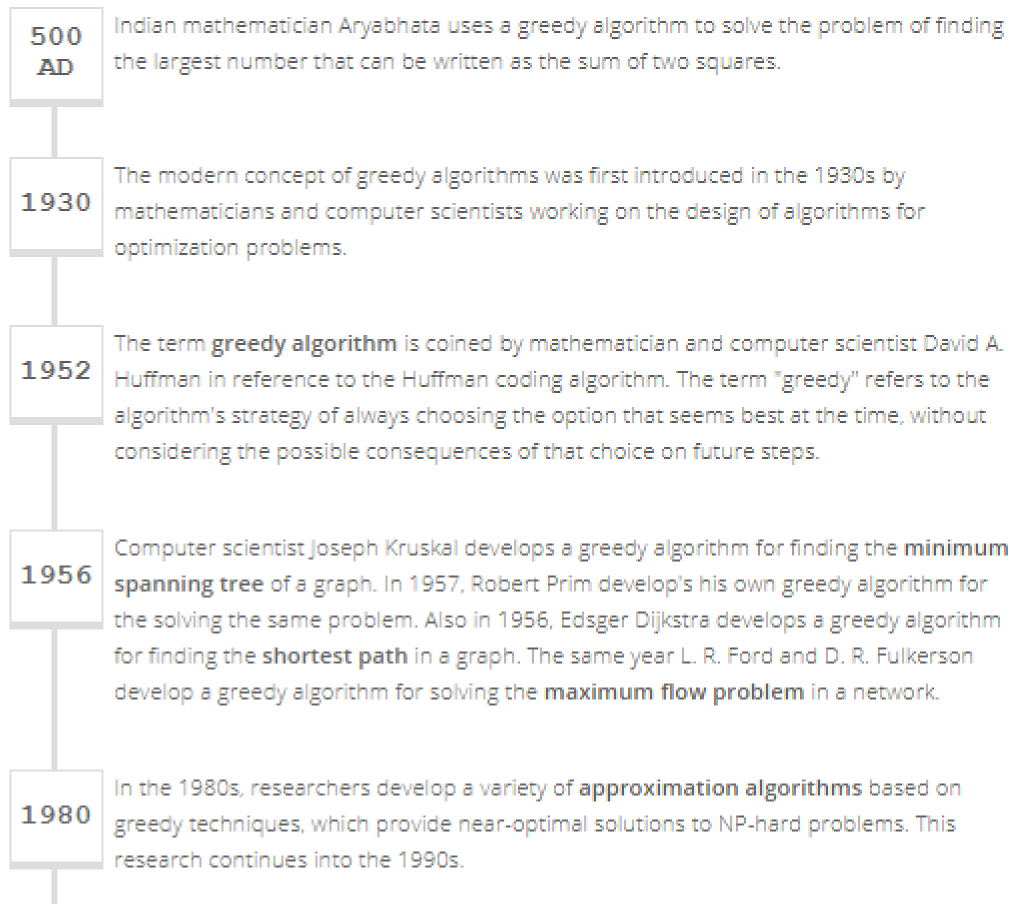
Hybrid algorithms combine greedy techniques with other optimization techniques, such as dynamic programming or local search. These hybrid algorithms can often provide better results than either technique used alone.

There are greedy algorithms capable of multi-objective optimization. They can be used to find a Pareto-optimal set of solutions.

In many real-world applications, input data is received incrementally over time. Online optimization algorithms must make decisions in real-time, without having access to all the input data in advance. Greedy algorithms have been shown to be effective in this setting because they can make quick decisions based on the available data.

Researchers have also been working on developing new methods for analysing the performance of greedy algorithms. There are new theoretical frameworks for understanding the behaviour of greedy algorithms in different types of optimization problems.

Milestones



Check your Progress

1. Which of the following is a characteristic of greedy algorithms?

a. Always guaranteeing an optimal solution

b. Exploring all possible solutions exhaustively

c. Memorizing and reusing sub problem solutions

d. Considering the best immediate choice at each stage

2. In the context of greedy algorithms, what does the "Greedy Choice Property" refer to?

a. Choosing the option that looks best at the moment

b. Exhaustively searching through all possibilities

c. Memorizing previously computed solutions

d. Backtracking to explore alternative choices

3. Which of the following problems can be solved using a greedy algorithm?

a. Travelling Salesman Problem

b. Longest Common Subsequence

c. Fractional Knapsack Problem

d. 0/1 Knapsack Problem

4. What is the main advantage of greedy algorithms?

a. Guaranteed to find the global optimum

b. Suitable for all types of optimization problems

c. Require less computational resources compared to other approaches

d. Can handle problems with overlapping sub problems

1.4 Answers to check your progress

1. d
2. a
3. c
4. c

2.1 The Coin Change Problem

Introduction

The Coin Change Problem is a classic algorithmic problem in computer science and dynamic programming. The objective is to find the number of ways to make change for a given amount using a set of coin denominations. This problem has practical applications in finance, vending machines, and various other domains.

Problem Statement

Given a set of coin denominations [c1, c2, ..., ck] and a target amount A, the task is to determine the number of ways to make change for' A' using the provided coins. Each coin can be used an unlimited number of times.

2.1.1 Coin Change Problem using Dynamic Programming Approach

The Coin Change Problem can be efficiently solved using dynamic programming. The idea is to build a table 'dp' where 'dp[i]' represents the number of ways to make change for amount i. The recurrence relation is:

$dp[i]=\sum^{k}_{j=0} dp[i-cj]$

This means that the number of ways to make change for amount i is the sum of the ways to make change for the remaining amount after subtracting each coin denomination.

Example

Problem Instance

Consider the following:

Coin denominations: [1, 2, 5]

Target amount: A=5

Dynamic Programming Table

Amount	0	1	2	3	4	5
Ways	1	1	2	2	3	4

The table is filled using the recurrence relation described earlier. For example, to fill **dp[5]**, we sum the values of **dp[5-1]**, **dp[5-2]**, and **dp[5-5]** which are 1, 2, and 1, respectively.

Conclusion

The number of ways to make change for A=5 using coins [1, 2, 5] is 4.

The Coin Change Problem is a versatile and widely studied problem in computer science. Its dynamic programming solution provides an efficient way to calculate the number of ways to make change for a given amount using a set of coin denominations.

What are applications of Coin Change problem?

Coin change problem is actually a very good example to illustrate the difference between greedy strategy and dynamic programming. For example, this problem with certain inputs can be solved using greedy algorithm and with certain inputs cannot be solved (optimally) using the greedy algorithm. However, dynamic programming version can solve all cases.

A simple example can be as follows. Let's say that you have N tons stuff, to be delivered from one place to another place. You can use airplane (capacity 100 tons), big truck (capacity 15 tons), medium truck (capacity 10 tons), etc. How do you manage to send your N tons of stuff with the minimum number of facilities? There are an infinite number of applications that you can find.

2.1.2 Failure of coin change problem

Try solving the coin change problem where you use the fewest amounts of coins possible to make an amount. Using the Greedy Approach - the algorithm sorts the coins array, starts with the biggest coin and uses it as many times as possible before moving on the next coin that will divide the remainder.

This worked for the initial test case:

coins = [1,2,5], amount = 11

But failed for this one:

coins = [186,419,83,408], amount = 6249

```
class Solution {
        public int coinChange(int[] coins, int amount) {
            int count = 0;

            if(coins.length == 1 && amount % coins[0] != 0) {
            return -1;
            }

Arrays.sort(coins);

            int i = coins.length - 1;
            while(amount >= 0 && i >= 0) {
            if(coins[i] <= amount) {
            int remainder = amount / coins[i];
            count = count + remainder;
            amount -= (remainder * coins[i]);
              }
                    i--;
              }

              return count;
        }
    }
```

Greedy approach to coin change problem doesn't work on the general case (arbitrary coin values).
Example:
Coins = [2, 3, 6, 7] and Amount = 12,
Greedy takes [2, 3, 7] and the optimal choice is [6, 6]

There is no guarantee of optimal solution in case of coin change problem.

Check your Progress

1. In the context of the coin change problem, what is the objective?
    a. Maximizing the number of coins used.
    b. Minimizing the total value of coins used.
    c. Achieving the target sum using the fewest number of coins.

d. Using all available coin denominations equally.

2. Which algorithmic paradigm is commonly used to solve the coin change problem?
   a. Dynamic Programming
   b. Greedy Algorithm
   c. Divide and Conquer
   d. Backtracking

3. What is the key principle behind the greedy approach for the coin change problem?
   a. Optimal Substructure
   b. Greedy Choice Property
   c. Overlapping Sub problems
   d. Backtracking

4. In the dynamic programming solution for the coin change problem, what does the entry **dp[i]** represent?
   a. The minimum number of coins needed to make change for amount **i**
   b. The maximum number of coins needed to make change for amount **i**
   c. The total value of coins used to make change for amount **i**
   d. The number of ways to make change for amount **i**

5. Which of the following scenarios is the coin change problem well-suited for a greedy algorithm?
   a. Arbitrary coin denominations
   b. Limited coin denominations
   c. Unlimited supply of each coin denomination
   d. Complex and irregular coin denominations

2.2 Answers to check your Progress

1. c
2. a
3. b
4. a
5. c

2.3 Scheduling Problems

Scheduling Problem can be defined be defined as a problem of scheduling recourses effectively for a given request.

Examples:

Processor Management

Scheduling machine Jobs

Ticket Counter

2.3.1 Types of Scheduling Problems

- Scheduling problem without deadline
- Scheduling problem with deadline
- Scheduling for sub interval

2.3.2 Scheduling problem without deadline

- We have to run three jobs with running time 2,7 and 4
- We have one processor on which can run these jobs.
- The problem is how to schedule these jobs?

Scheduling Problem

Schedule	Total time in system	Average time
[1, 2, 3]	2 + (2 + 7) + (2 + 7 + 4) = 24	8
[1, 3, 2]	2 + (2 + 4) + (2 + 4 + 7) = 21	7
[2, 1, 3]	7 + (7 + 2) + (7 + 2 + 4) = 29	9.6
[2, 3, 1]	7 + (7 + 4) + (7 + 4 + 2) = 31	10.3
[3, 1, 2]	4 + (4 + 2) + (4 + 2 + 7) = 23	7.9
[3, 2, 1]	4 + (4 + 7) + (4 + 7 + 2) = 28	9.3

Informal Algorithm

- Sort all the jobs by service time in non-decreasing order.
- Schedule next job in the sorted list.
- If all the instances are sorted, then return the solution list.

FORMAL ALGORITHM

Formal Algorithm

Algorithm schedule(J)

```
%% Input: An array J with a set of jobs 1 to n along with service time
%% Output: An optimal schedule
Begin
    S = {sorted array of jobs in J based on service time}
    i = 1
    schedule = φ
    while (i <= n) do                %% for all jobs do
        select the next job i from S   %% Selection procedure selects from sorted list
        solution = schedule U job i
        i = i + 1
    End while
    return(solution)
End
```

Complexity Analysis

- Sorting of jobs O(n log n)
- Scheduling of jobs O(n)
- Final Complexity analysis O(n log n)

Scheduling problem with deadline

- To maximize the profit by scheduling the jobs taking into account deadline.
- A deadline is a time limit before which the job has to be completed to get profit.
- A sequence is feasible if all the jobs end by the deadline.

Example

Table 11.2 Jobs with deadline and profit

Job	Deadline	Profit (₹)
1	2	60
2	1	30
3	2	40
4	1	80

Example

Table 11.3 Jobs with feasible schedules

Job sequence	Total profit
2, 1	90
3, 1 or 1, 3	100
2, 3	70
4, 1	140
4, 3	120

Informal Algorithm

- Sort all the jobs by profit.
- Solution=Null
- Select the next job
- If job is feasible then add the job
- If all the jobs are considered then exit.

Formal Algorithm

Algorithm schedule_with_deadline

```
%% Input: A set of jobs 1 to n with service time
%% Output: An optimal schedule
Begin
    S = sorted array of jobs based on profit in non-decreasing order
    i = 1
    schedule = ϕ
    while (i <= n) do                                %% for all jobs do
            select the next job i from S             %% Selection procedure
            if (scheduling job is feasible) then     %% Feasibility check
            solution = schedule U job i of S
            i = i + 1
    End while
    return(solution)
End
```

Complexity Analysis

- Sorting - O(n log n)
- Scheduling- O(n)
- Complexity Time- O(n log n)

Check your Progress

1. In the scheduling problem without a deadline, the informal algorithm suggests sorting jobs by:

a. Profit

b. Deadline

c. Service time

d. Complexity

2. What is the time complexity for scheduling jobs in the scheduling problem without a deadline using the formal algorithm?

a. O(n)

b. O(n log n)

c. O(n^2)

d. O(log n)

3. In the scheduling problem with a deadline, what is the primary objective?

a. Minimizing the service time

b. Maximizing the profit

c. Scheduling without any constraints

d. Sorting jobs in non-decreasing order

4. What does the feasibility of a job in the scheduling problem with a deadline depend on?

a. Job complexity

b. Profit

c. Service time

d. Deadline

5. According to the complexity analysis, what is the overall time complexity for scheduling jobs in the problem with a deadline?

a. O(n)

b. O(n log n)

c. O(n^2)

d. O(log n)

2.4 Answers to check your progress

1. c

2. b
3. b
4. d
5. b

2.5 References

Devopedia. 2023. "Greedy Algorithms." Version 3, February 17. Accessed 2023-05-02. https://devopedia.org/greedy-algorithms

https://epgp.inflibnet.ac.in/

Block **3**
Unit **9**: **Knapsack problem**

1.0 Learning Objectives
1.1 Knapsack problem
1.1.1 0 – 1 Knapsack problem
1.1.2 Fractional Knapsack Problem
1.2 Huffman Algorithm

Check your Progress
Answers to check your Progress
1.3 Minimum Spanning Tree
1.3.1 Kruskal Algorithm
1.3.2 Prim's Algorithm

Check your Progress
Answers to check your Progress
1.4 Optimal merge Short
1.5 Single-source Shortest Path Problems

Check your Progress
Answers to check your Progress
Model Questions

1.0 Learning Objectives

After completing this unit, the learner will be able-

- To Understand the Knapsack problem
- To Understand the Huffman algorithm

- Minimum Spanning Tree
- Kruskal Algorithm
- Prim's Algorithm
- Optimal merge problem
- Shortest Path Algorithm

1.1 Knapsack problem

Let us begin with a small story. Let us assume that a thief has entered into the museum the museum has got fabulous paintings sculptures and wells and let us assume that every item is having certain weight as well as the profit and unfortunately the thief has brought only one single knapsack. Knapsack means a large bag. So the issue is the thief can not take all the items so he will look for the items that can fetch the maximum profit for that particular person. So in other words the problem of knapsack is about how to maximize the whole this is called a loading problem. A knapsack problem where you have an objective function the objective function is to maximize and this maximization of profit is possible by the optimal packing of all the items .So let us discuss about how to apply the greedy approach. For so the important gist of that particular story is like this-

A knapsack is given, so the knapsack maximum capacity is w that means at no point of time the capacity of the knapsack can go beyond this w and we are given a set of items there are n items that are available.

Each item is having a weight W_i and it is having a profit or value called P_i . So let us assume that the weight, profit and W are all integer values. So the problem of knapsack is very simple we have to pack the items into the knapsack so that the profit is going to be the maximum. So this is what the knapsack problem is all about that is how to achieve the maximum profit. So this problem is so popular and in fact in computer science we study about two variants of the knapsack problem one is called.

- 0 – 1 Knapsack problem
- Fractional Knapsack problem

1.1.1 0 – 1 Knapsack problem

Where the items cannot be divided .In other words the thief has to take the entire item or he has to leave it in other words. If there are electronic goods available we can't break an item so that means either we have to take the item into the knapsack.

1.1.2 Fractional Knapsack Problem

Where we can take partial content of the given item. Suppose if the item is a powder or liquid then we can take a small part of the item into the knapsack. It is conceivable that fractional knapsack problem is an easy problem that can be solved using the greedy approach and zero bar one knapsack problem is considered to be a toughest problem in computer science.
Fractional knapsack problem can be explained in a mathematical formula is given below-

$$\text{Maximize} \sum_{i=1}^{n} p_i x_i$$

Here Xi represents the fraction of the item loaded into the knapsack subjected to the constraint that is, the capacity of the knapsack cannot be overloaded is capacity W, that is,

$$\sum_{i=1}^{n} w_i x_i \leq W$$

with the constraints $0 \leq x_i \leq 1$ and $1 \leq i \leq n$

So X_i represents the fraction of the item that are loaded into the knapsack and the corresponding profit is going to be Pi and every item is having a weight w_i so that means now you can see that the mathematical formulation of this problem is maximization of p_i and x_i where the aim is to maximize the profit subjected to the constraint that the weight of the items that are loaded into the knapsack should be less than or equal to w so X_i ranges from 0 to 1 and i ranges from 1 to n that means there are n items that are available. This is a mathematical formulation of the problem.

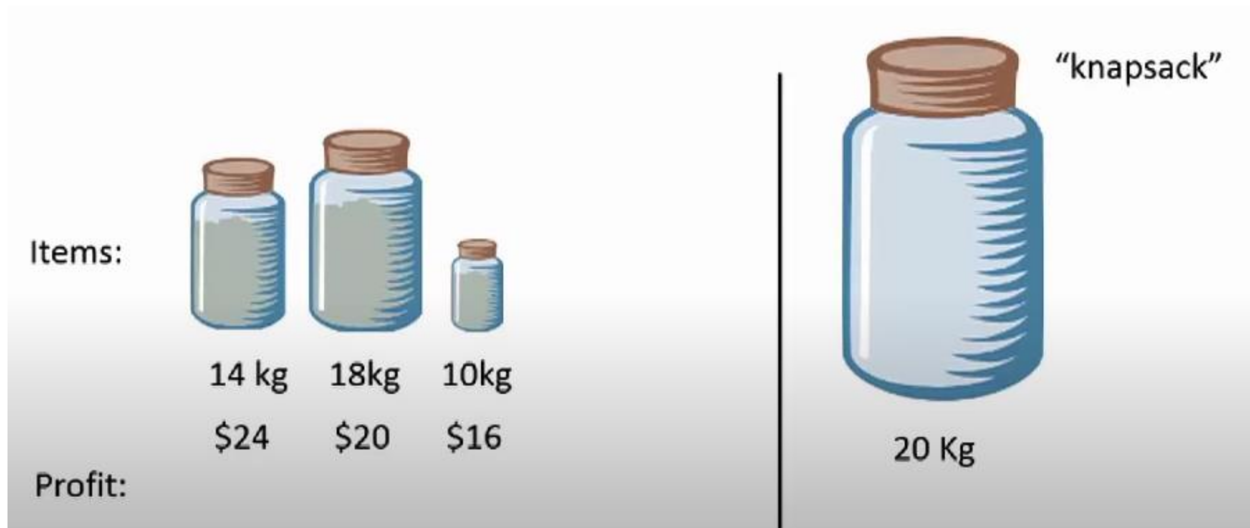


Figure -1

Figure–1 is the graphical illustration of how it really works. So we assume that the bigger bottle is going to be the knapsack whose weight is given as 20 kg and we are given three items and items weight as well as the profit are given in the figure .So the knapsack problem is we have to load the items of the smaller things into the larger thing so that we can get maximum profit this is the fractional knapsack problem so that means we can take a part of the item onto the knapsack .According to the profit so just to see this particular diagram so you can see that the profit is given so for item 1 the profit is 24 $, item 2 profit is 20$ and item 3 profit is 16 $ and the corresponding weights are given as 14, 18, and 10. According to the profit we can see that item 1 is having the maximum profit so-

- Load item 1 as it has maximum profit remaining capacity =20 - 16 = 6 kg
- Load the remain with item 2 i.e. , 6/18 remaining capacity = 0
- Loading third item is not possible
- Total profit = 24 + (1/3) * 20 + 0 = 30.66

Arrange using Weight-

- Load item 3 as it has least weight. Remaining capacity = 20 – 10 = 10 Kg
- Load the remaining with item 2 i.e., 10 /18. Remaining capacity = 0

- Total profit = 0 + (10 /18) * 20 + 16 = 27.11

Ratio of Profit and weight –

P_i divided by W_i ,so calculate the ratio of all the item it is coming as 1.71, 1.11 and 1.6 so let put it in the ascending order so that means the item that is having the maximum profit and weight ratio is item 1 therefore loading the item 1 so that mean 20 - 14 = 6.

So the next item where profit to weight ratio is very high is item 3 so therefore for the remaining capacity of the knapsack filling it with 3/5 so loading of the item 2 is not possible. So in this case the total profit is going to be 33.6, So this is a comparison table.

x_1	x_2	x_3	$\sum x_i w_i$	$\sum p_i x_i$	Comments
1	1/3	0	20	30.66	According to profit
0	5/9	1	20	27.11	By least weight
1	0	3/5	20	33.6	By profit to weight ratio

So you can see that according to the profit means it is coming as 30.66

By weight it is coming as 27.11 and the ratio of profit to weight it is coming as 33.6 .So now you can come to conclusion that best criteria for selection of item is going to be the ratio of the profit to weight so the greedy components are-

Selection procedure- It is based on the ratio of P_i /W_i

Feasibility check- Whether the total weight is less than the capacity of the knapsack.

Solution check- Whether all the instances are checked or not.

Informal Algorithm-

So the informal algorithm is going to be like this. So sort the items in the decreasing order of profit to weight ratio, while there is still room in knapsack .Consider the next item and take as much as possible. The formal algorithm is going to be like-

```
Algorithm greedy_knapsack(W, n)

%% Input: n items with profit p[1 .. n] → Profit and w[1 .. n] → weight, W is the
%% capacity of the knapsack. It is also assumed that the items are ranked by pi/wi
```

so we have algorithm greedy knapsack w then n. n are items with the profit which is given in the array p and it is given the weight is given in the array w and the ratio is P_i by W_i.

```
%% Output: Optimal packing order of items stored in solution vector x[1 .. n]
Begin
    // Initialize the solution vector
    for i ← 1 to n do
        x(i) := 0.0;
    End for
    load = 0                    %% Initial weight of knapsack i
    i = 1                       %% start with the first site,
    while ((load < W) and (i <= n)) do
    if (wi + load ≤ W) then
        load = load + wi        %% Load item fully
        x[i] = 1                %% Mark in the solution vector that the item is
                                %% loaded fully
    else
        r = W - load            %% Compute the space left out
        load = load + r/wi              %% Fill knapsack with the fraction of item
        x[i] = r/wi             %% Record the amount of item in solution vector
    End if
    End while
    return(x)                   %% Return solution vector
End
```

Initially the solution vector is initialized to 0. So the ratio whenever there is a possibility is there and so putting that into the knapsack so taking the remaining item based on that taking the fractions and keep on filling the knapsack so whenever filling the knapsack with the particular item the corresponding vector is going to be 1. This is the formal algorithm for that.
This is a small example of how it really works –

Items	Weight	Profit
1	8	24
2	9	18
3	5	20

there are three items so you have weight profit ratio so as we discussed earlier the best criteria is profit to weight so-Compute this so we can see that it is coming as 3, 2 and 4. So you can see that for p3 by w3 it's going to be 4 so that means going to fill. Then going to fill with the next one and filling the remaining thing with the next item so the profit is coming as 48 this is the answer for this particular problem.

Complexity analysis-

Sorting the items based on their value-to-weight ratios dominates the time complexity. The dominant operation is the sorting step, which takes O(n log n) time.

1.2 Huffman Algorithm

Huffman algorithm which uses the greedy approach. This algorithm is very much useful for data compression so compression is a very important topic where we are talking about reduction of size of data so by that we mean that the number of bits that is required to represent the data is reduced and the advantage of this data compression is that we can reduce the storage space that is necessary for storing the files as well as we can reduce the transmission cost latency and bandwidth so because of this particular reason compression is very important and Huffman code is considered to be one of the most important algorithm in data compression so the logic is very simple optimal coding .

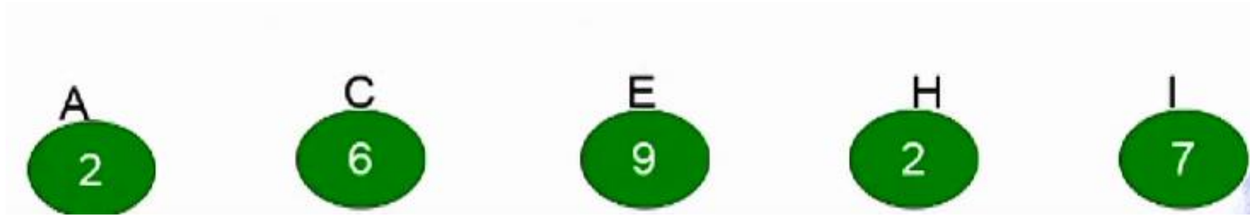
Example: So let us assume that to transmit 3 symbols {A, B, C} .So transmission requires encoding of these symbols as a code by giving some address so that means three symbols are there so that means let us assume that if we go for fixed length(2 bits) code so that means we can go for 0 0, 0 1, 1 0 and 1 1 so even though there are only three symbols so we require at least two bits to represent that so that means in fixed coding if we go for this and if we want to transmit a file that consists of 1000 characters then 2000 bits are required in order to transmit this .So the main issue here is fixed length code which is sub optimal so that means it is not very optimal so the logic is very simple why can't we go for a variable length code where every symbol will not have a fixed address but rather a variable address. So we require some sort of a methodology for that so that is given by Huffman so the idea is very simple we will use a short codes for more frequent characters and long codes for less frequent .So in other words so we are looking for some information content and if a symbol comes very frequently means so that means it is not very important so we will what you say use very short code to represent that and if a symbol comes very rarely that means information content is very high in that case we will go for the longer code. So we will just give some important definitions.

A code- code is the mapping of character of the alphabet to a binary code word.

A prefix code - Is a binary code such that no code word is the prefix of another code word.

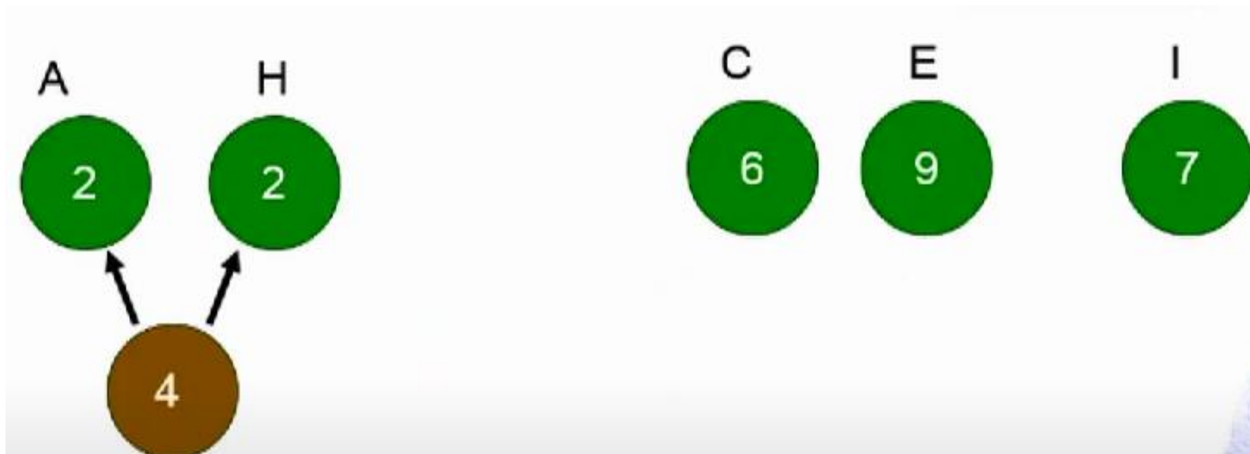
Encoding tree- Encoding tree represents the prefix code where every external node stores a character and the code is given by the path from the root to the external node where every edge is given 0 on the left hand side 1 on the right hand side.

Example-So we will take a very simple example of how to do this –

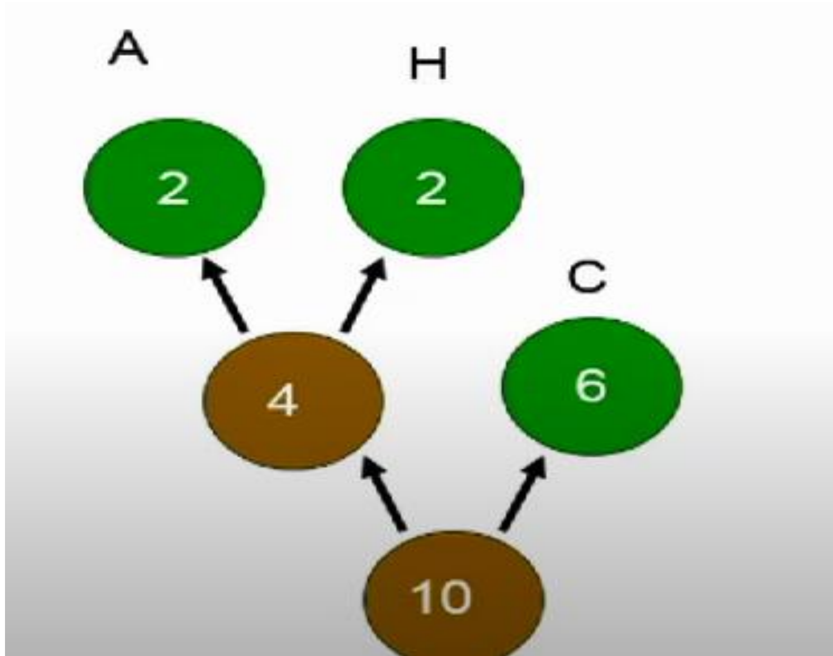


So taking 5 symbols and every symbol is associated with a frequency 2, 6, 9, 2and 7. Say for example if take a string like a, a, b we can say a occurs two times and b occurs one time so we can calculate frequency .we can calculate the probability. Huffman code works with both frequency as well as the probability.

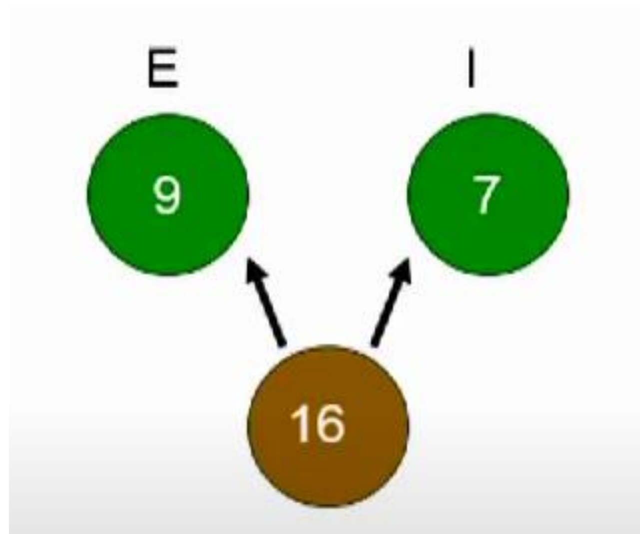
Solution: This algorithm uses the greedy approach, the greedy approach is like this so we have to go for the best local decisions so the logic is very simple like a greedy man we are looking for the smallest symbol and we are trying to combine. So given your option we go for the least frequency symbols the least frequency is a and h so trying to combine this so this is 4 so –



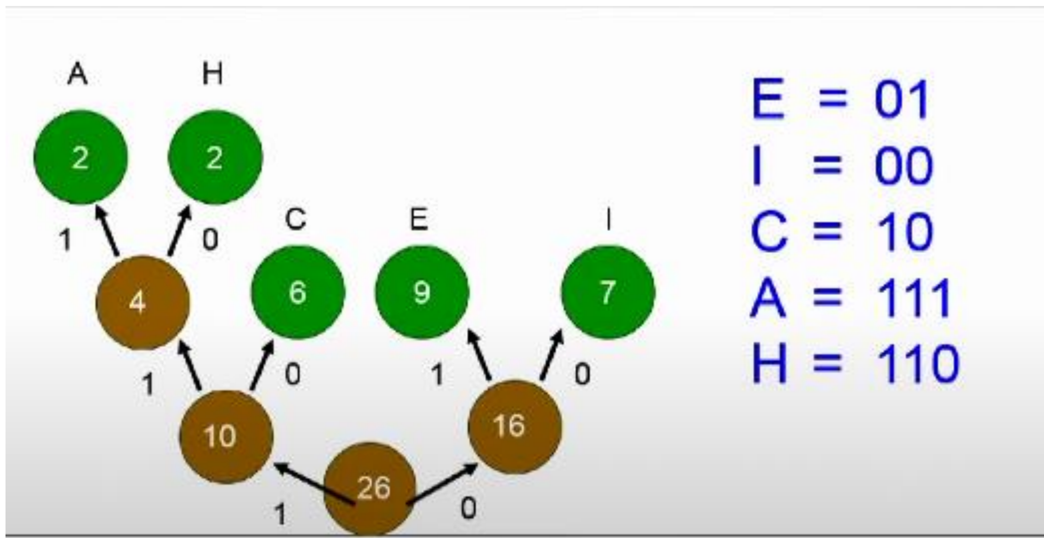
Now consider the best possibility for 4, 6, 9 and 7 so that means you can see that 4 and 6 are the best possibilities is getting 10.



So next consider 9, 7 so the best possibility is 9 and 7 is 16.

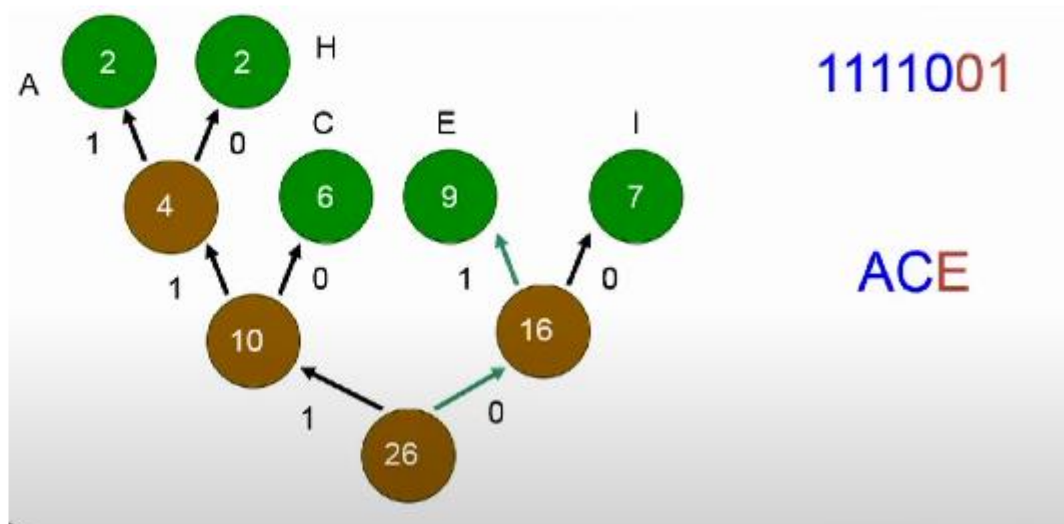


So when combine that getting 26.



So this is called an encoding tree where you can see that every external node that is given in green color is representing the symbol and the binary code of every symbol is obtained by tracing the path from the root to that particular external node. so now you can see that A is nothing but 1 1 1 H is 1 1 0 and C is 1 0 then E is 0 1 and I is 0 0 .This is a variable code so that means you can see that all the symbols are not having the same length of code but rather it varies according to the frequency so this is the logic so the symbol a occurs frequently therefore it is having more length compared to symbol let us say E or something, so this is the Huffman code that we have.

So imagine we want to transmit the message ACE across the channel so that means for A it is 1 11 and C it is 1 0 and for E it is 0 1 so it means we will be sending the binary stream 1 1 1 1001 across the channel. So across the channel this bit stream will be received .So the question is how to interpret this. The logic is very simple the same encoding tree that we used as part of the sender will be used to what you say decode so in decoding we read the bit stream and we will use the encoding tree to decode the file content so let us see what happens .



So first getting A then getting A again then getting 1 so that means that 1 1 1 is nothing but A so next getting 1 so its 1 then getting 0 so that is going to be your C so next is 0 so 0 and 1 that is going to be E so that means now you can see that the ACE is received exactly as we have transmitted and there is no

information loss this is the beauty of the Huffman code .But there is only one problem that is the prefix property needs to be satisfied.

prefix property- prefix property says that no code is should be the prefix of another code for example let us say if for one symbol it is 0 0 0 hypothetically then this should not be the prefix for any other code like 0 0 0 1 . otherwise there will be a lot of confusions so it the message will be misinterpreted for example if 0 represents character A 0 1 represents let us say character B then how you will interpret 0 0 1. so this is where problem comes so this leads to the multiple way of interpretation that may cause problems fortunately the Huffman code is satisfying this property so that means there is no confusion .

Informal algorithm-

Greedy approach choose two lowest frequencies and combine them it will produce a new node where the frequency of these two symbols are added then this process is getting repeated till we get the root so this is the informal algorithms so you can see that initially all characters are assigned as a single node then we will keep on adding all the things and so we will continue this procedure till we get the entire tree so this is a formal algorithm so you can see that the priority queue is used so you can take the character c so what we are trying to do is we are putting everything into the priority queue with f of c as the key then what we are trying to do is we are trying to extract min extract mean from the priority queue so two least frequency items are sent as the output then we are creating a new node then we are adding the frequency we are putting that left hand side we are assigning a 0 and right hand side we are assigning as 1 then we are adding this set on to the cube and we are repeating this process from 1 to n minus 1 therefore at the end of all the iterations you have the encoding tree

Complexity analysis

Complexity analysis is the complexity of handling the heap .so it requires order of n (log n). So that means you have n items so that means the total complexity time is order of n (log n).

Check your Progress

1. What is the Knapsack Problem?

A. Sorting items in a backpack

B. Finding the heaviest item in a set

C. Maximizing the value of items in a limited-capacity knapsack

D. Packing items into a suitcase

2. Which variant of the Knapsack Problem considers fractional parts of items?

A. Fractional Knapsack Problem

B. 0/1 Knapsack Problem

C. Bounded Knapsack Problem

D. Unbounded Knapsack Problem

3. What is the main objective of the Knapsack Problem?

A. Minimize the weight of items

B. Maximize the profit or value of selected items
C. Minimize the number of items in the knapsack
D. Maximize the total number of items selected

4. What is the Huffman algorithm used for?
A. Sorting a list of numbers
B. Data compression
C. Searching in a sorted array
D. Encrypting messages

5. What type of coding does Huffman algorithm provide?
A. Fixed-length coding
B. Variable-length coding
C. Run-length coding
D. Arithmetic coding

Answer to check your progress

1. C
2. A
3. B
4. B
5. B

1.3 Minimum Spanning Tree

Greedy algorithms are much faster therefore we consider this as a useful strategy for solving the optimization problems so the optimization problems are supposed to satisfy these two properties-

Greedy choice property: we take lots of locally optimal decisions and when we try to combine all these locally optimal decisions we are getting the global optimal solutions.

Optimal solutions: Optimal solutions are considered to be the sequence of solutions that are considered to be the optimal for solving the sub problems so the main objective is to introduce the concepts of spanning tree.

Connected graph- Every node is reachable from every other node.

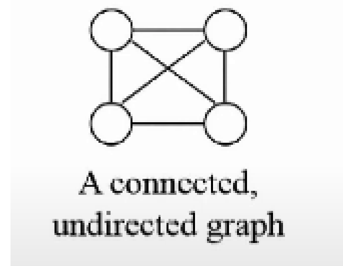
Undirected graph- Edges do not have any associated direction.

Spanning tree- A tree that is connected a cyclic graph which contains all the vertices of the given graph.

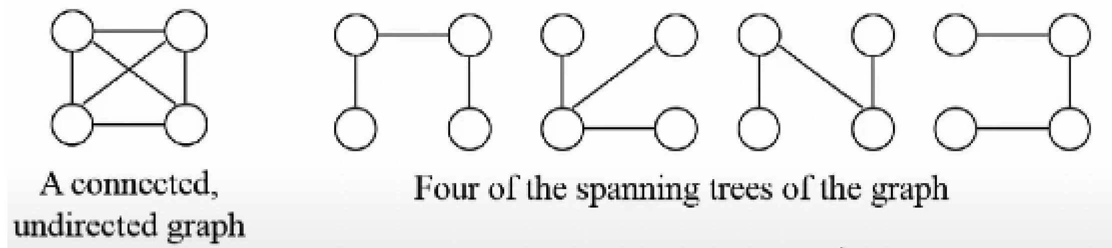
Minimum Spanning Tree – Spanning tree with the minimum sum of weights. The objective of the minimum spanning tree is very simple we will find a spanning tree with a minimum sum of weights. In other words the input for the spanning tree algorithm is a graph so what we are trying to do is we are trying to generate a tree where all the vertices are available but we ensure that cycles are not present in the spanning tree and also we ensure that no node is isolated so the objective in short is to generate a connected acyclic

graph that contains all the vertices of the given graph so the minimum spanning tree is the sum of the cost of the edges should be the minima if the graph is not connected then there is a spanning tree for every component of the graph.

Example of spanning Tree-
We will take one simple example


A connected, undirected graph

Let us take this graph as the input so you can see that there are four vertices. This is a connected undirected graph because as we have seen the edges are not associated with any directions so as per our definition we should get a spanning tree where all vertices should be the present but there should not be any cycle.

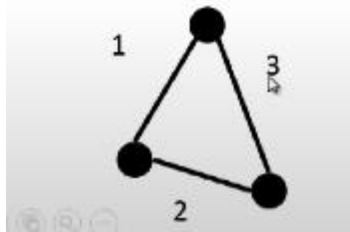

A connected, undirected graph Four of the spanning trees of the graph

So these are some of the spanning trees of the graph so you can verify every spanning tree consists of all the vertices and you can see that there are no cycles and you can see that all the nodes are connected so this is the spanning tree.
we can find the spanning tree by the brute force but it is very complicated as the number of nodes increases the number of spanning tree also grow enormously therefore the application of the brute force technique is not feasible.

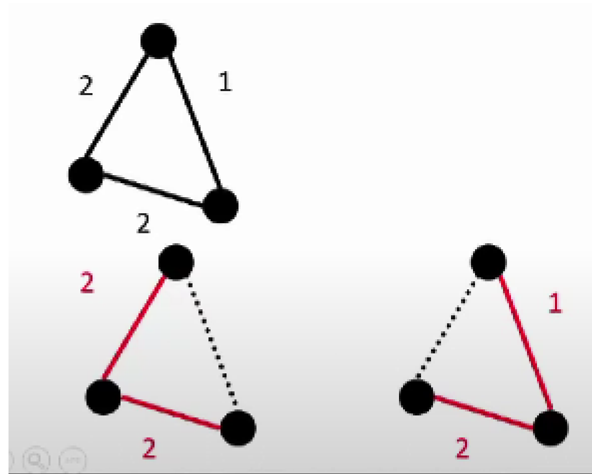
Need for Spanning Trees-

In computer science spanning tree is one of the most important problem which has got a wide variety of applications.
Imagine like we are trying to construct a telephone network then the pre requirements is that all the nodes must be connected but there should not be any cycles so that means the number of edges should be minimized translated to a telephone problem it means that the number of edges reduced means the number of telephone cables are reduced so when the telephone cable is reduced it leads to saving of money therefore not only the telephone network any network that are related to power telephone etc. can

use this minimum spanning tree algorithm to get a network where there is a connectivity but at the same time there is no redundancy of the edges so.



An undirected graph and you have cost that is associated with each edge that is available so what is the minimum spanning tree so these are some of the possibilities-



The cost of the spanning tree is the sum of the weights of the edges that are present so that means out of these three possibilities the second and third one is very effective because they are associated with the minimum cost so any spanning tree where we are getting the minimum cost is what we are calling as the minimum spanning tree in short mst .

Now you can say that a minimum spanning tree is a sub graph of an undirected graph such that the mst encodes all the nodes. Spans all the nodes connected acyclic that means there should not be any cycles and it should also have the minimal total edge weight.

- **Kruskal's Algorithm**
- **Prim's Algorithm**

Both these algorithms work for both weighted as well as the un-weighted graphs both this algorithm works for both directed as well as the undirected graphs.

Generic Algorithm

For each and every individual algorithms the generic algorithm is given connected, undirected, weighted graph G, the aim of this algorithm is to find the spanning tree with minimum weight that is going to be T, so initially it is null then what we are trying to do is we are trying to find the safe edge and we are trying to add that into the set so repeatedly doing so results in the spanning tree.

1.3.1 Kruskal Algorithm

Joseph B. Kruskal

Joseph Bernard Kruskal, Jr. (born January 29, 1928) is an American mathematician, statistician, and computer scientist.

Joseph Bernard kruskal is an American mathematician statistician and computer scientist who proposed this algorithm for finding the spanning tree.

Start with no nodes or edges in the spanning tree then we are going to repeatedly look for the low cost edge or the cheapest edge and we will try to add that into the spanning tree at the same time ensuring that the addition of the edge doesn't create any cycles in other words the kruskal algorithm considers only the edges.

This algorithm works with edges rather than the nodes, so the algorithm is -

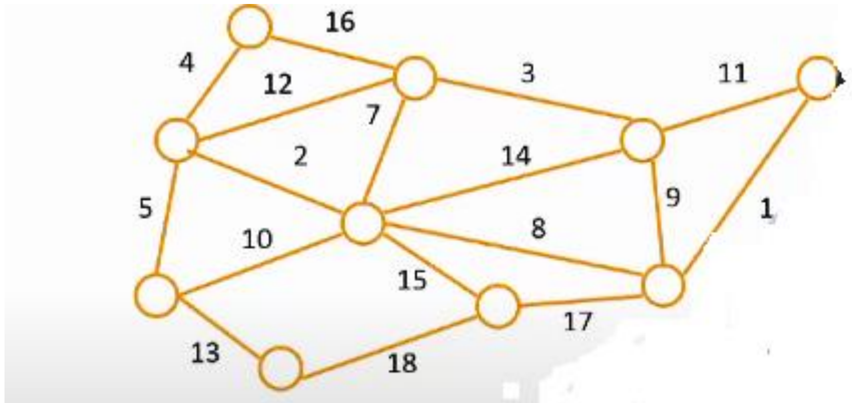
Sorting of all the edges based on the weight because we are going therefore the first step is we are going to sort all to apply the greedy approach where we are going to look for the least cost edge so

```
T = empty spanning tree;
E = set of edges;
N = number of nodes in graph;
while T has fewer than N - 1 edges {
    remove an edge (v, w) of lowest cost
    from E
    if adding (v, w) to T would not create a
    cycle
        then add (v, w) to T
        else Reject it
    }
```

The edges based on the edge weight then we are selecting the first |V|- 1 edges that do not create any cycle .This is an informal algorithm-

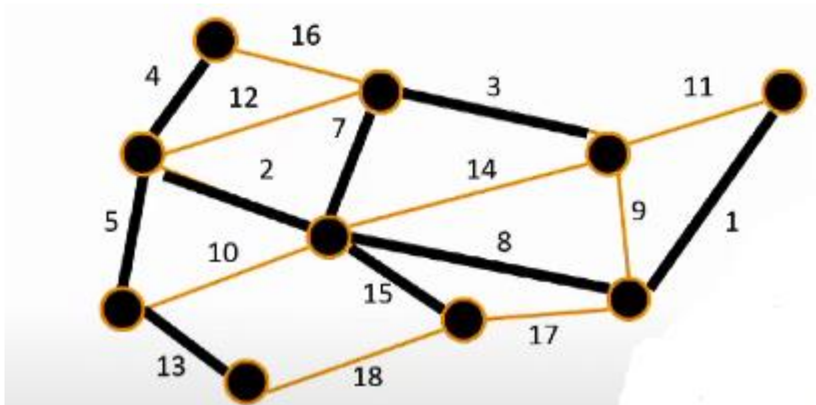
Example of kruskal algorithm-

Find the MST with the help of kruskal algorithm



Solutions:

1 is the minimum cost edge that is present then we have 2 then we have 3 then we have 4 then we have 5 .so the logic is very simple just.
Sort all the edges in the ascending order.
The selection criteria of the greedy algorithm is the least cost and we have to add that into the spanning tree provided if it doesn't create any cycle .
So let us try , 1 is a minimum then look for the minimum 2 is a minimum so we can add this because there is no cycle created then next is 3 then 4 so no problem 5 is also not a problem at all so you can see that 7 is also not a problem 8 we can add because it doesn't create any cycle but 9 we can't add because it can create cycle actually so 10 we can't add so 12 we can't add so 13 is possible so now this is the spanning tree.



Kruskal Algorithm-

```
1.  A ← ∅
2.  for each vertex v ∈ V
3.          do MAKE-SET(v)
4.  sort E into non-decreasing order by w
5.  for each (u, v) taken from the sorted list
6.      do if FIND-SET(u) ≠ FIND-SET(v)
7.             then A ← A ∪ {(u, v)}
8.                  UNION(u, v)
9.  return A
```

Complexity of kruskal Algorithm-

Running time: $O(V+ElgE+ElgV)=O(ElgE)$ –
dependent on the implementation of the disjoint-set data structure

Running time: $O(V+ElgE+ElgV)=O(ElgE)$
- Since $E=O(V^2)$, we have $lgE=O(2lgV)=O(lgV)$

1.3.2 Prim Algorithm

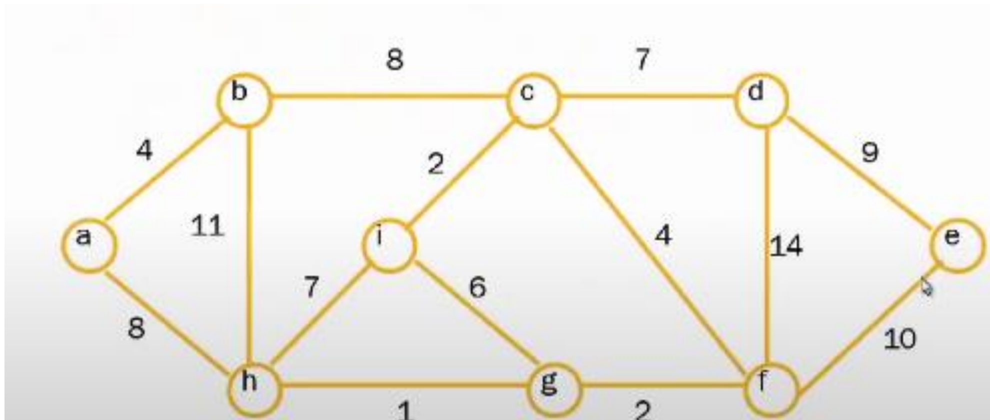
Robert Clay Prim (born 1921 in Sweetwater, Texas) is an American mathematician and computer scientist.

Robert clay prime is an American mathematician computer scientist .This is an informal algorithm. It is exactly similar to the kruskal algorithm**.** We start with any node in the spanning tree and again we are looking for the cheapest edge the criteria is that the new node should be an unexplored node so that means the node that is not belonging to the same subset where the source vertex is present so if any such

notes are there then we are adding that so in this thing we not only look for the edges but also the notes so that means both nodes and edges are the important criteria for primes algorithms.

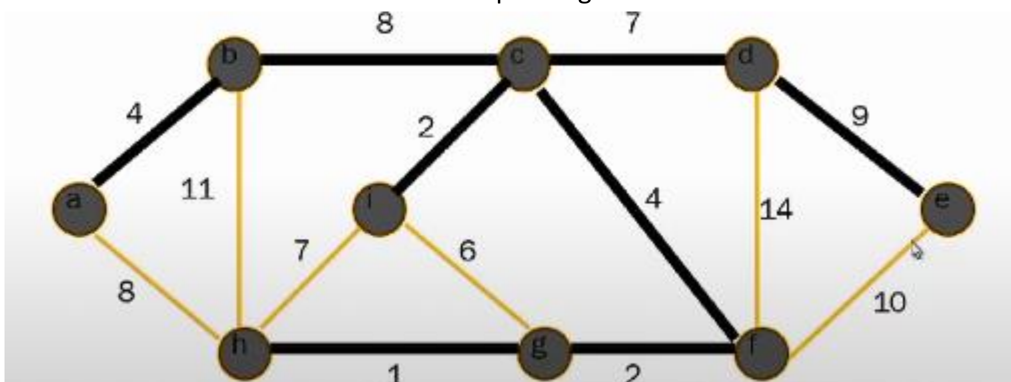
Initially t is going to be having a single node so that we have to start somewhere we can start anywhere in prime's algorithm and e is nothing but the set of edges that are adjacent to s, taking the edge and checking whether it is already there if so going to discard otherwise going to add that so this is the informal algorithm of prime.

Example: Find MST with the help of prims algorithm.



Solutions:

so we will start from "a" so look for the minimum cost ,so 4 is the minimum, now have two possibilities check all edges that starts from "a" as well as b so just branches out so it's c, so now have the possibility of a, b and c . Find the minimum so it's 2 now then again can check the minimum cost so that means you can see it's going to be the 4. so now again can check it's going to be 2 and again it's going to be 1 so we can see that i can't add from h because it may create cycle so that means it's going to be 7 then we can see it's 9 so we can see that this is a minimum spanning tree.



Informal Algorithm

```
Q := V[G];
for each u ∈ Q do
    key[u] := ∞
key[r] := 0;
π[r] := NIL;
while Q ≠ ∅ do
    u := Extract - Min(Q);
    for each v ∈ Adj[u] do
        if v ∈ Q ∧ w(u, v) < key[v] then
            π[v] := u;
            key[v] := w(u, v)
```

Complexity of Prim Algorithm

The time complexity of the Prim's Algorithm is $O((V + E) \log V)$ and total space complexity is O(V+E).

Check Your Progress

1. What is the primary goal of finding a Minimum Spanning Tree in a graph?

a. Maximizing the total weight of edges

b. Minimizing the total weight of edges

c. Counting the number of vertices

d. Finding the shortest path between two vertices.

2. In Prim's algorithm, how is the starting vertex chosen?

A. Randomly

B. It doesn't matter; any vertex can be chosen.

C. The vertex with the maximum degree

D. The vertex with the minimum degree

3. Which condition must be satisfied for a graph to have a unique Minimum Spanning Tree?

A. The graph must be connected.

B. The graph must be undirected.

C. The graph must have distinct edge weights.

D. The graph must be acyclic.

4. Which algorithm builds the Minimum Spanning Tree by always connecting the nearest vertex?

A. Dijkstra's algorithm

B. Prim's algorithm

C. Kruskal's algorithm

D. Boruvka's algorithm

5. What is the time complexity of Kruskal's algorithm for finding a Minimum Spanning Tree with V vertices and E edges?
A. O (V log V)
B. O (E log E)
C. O (V^2)
D. O (E + V)

Answers to check your progress

1. B
2. B
3. C
4. B
5. B

1.4 Optimal merge Short

Linear merge

It is often important to merge two sorted lists. The sorted lists can be merged used linear merge. The linear merge can be stated as follows: Given two sorted lists L_1 and L_2, $L_1 = (a_1, a_2, \dots, a_{n1})$ and $L_2 = (b_1, b_2, \dots, b_{n2})$, the problem of optimal merge is to lists L_1 and L_2 into one sortedlist L.
It can be observed that the complexity of merging two lists require a merging cost of $O(n_1+n_2 - 1)$.
In general, the problem can be stated as below: Given n sorted lists, each of length m_i, what is the optimal sequence of merging process to merge these n lists into one sorted list ? It can be observed that the merging of ‘n’ lists should be done only by merging two lists at a time. Thus, the complexity of merging is multiplied by n-1 times.
To avoid this, optimal merge is preferred. It is based on the idea of optimal weighted tree.

1. **Binary Merge Tree**:

A binary merge tree is a binary tree with external nodes representing entities and internal nodes representing merges of these entities.

2. **Optimal binary merge tree:**

An optimal binary merge tree is a binary merge tree where the sum of paths from root toexternal nodes is optimal (e.g. minimum).
Assuming that the node (i) contributes to the cost by P_i and the path from root to such node has length L_i , then problem is to minimize L, where L is

$$L = \sum_{i-1}^{n} P_i L_i$$

Example 1: Find the weight of the following merge tree shown in Fig. 1.

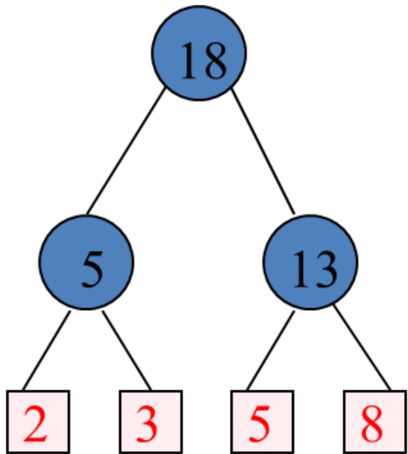


Fig. 1: Sample Merge Tree

Solution:
The weighted external path length can be obtained as follows:
WEPL (T) = S(weight of external node i) * (distance of node i from root of T)Hence, for the WEPL(T) = 2 * 2 + 3*2 + 5*2 + 8*2 = 36.
Example 2: Find the weight of the following merge tree shown in Fig. 2.

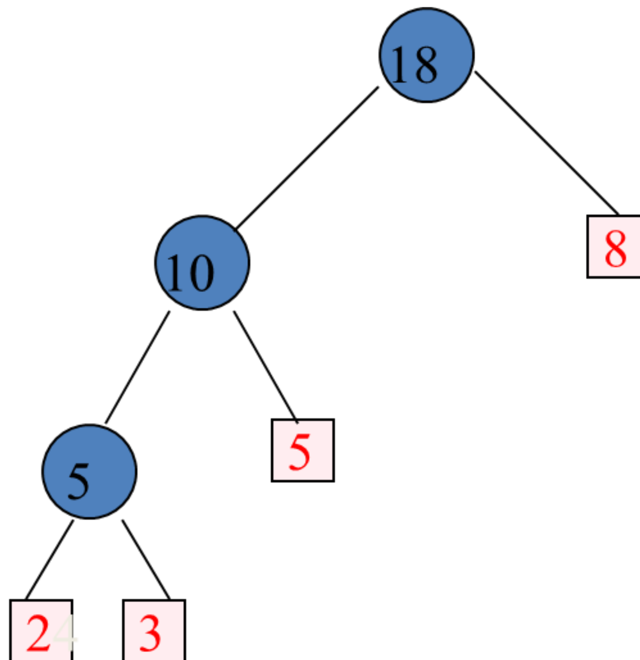


Fig. 2: Sample Merge Tree

Solution: Using the above formula, the weighted external path length can be computed as follows: WEPL(T) = 2 * 3 + 3*3 + 5*2 + 8*1 = 33.

Optimal merge tree-

An optimal weighted tree is a tree that is associated with the minimum weight. Let us illustratethis through a numerical example.

Example 3

Let us consider four items with the weights as follows: 30, 10, 8. Show the two ways of constructing the merge tree.

Solution:

One way of constructing an optimal merge tree is by merging L1 with L2. Then merging the resultant with L3. The merge tree is shown in Fig. 3. The merge cost is given as Cost = 30*2 +10*2 + 8*1 = 88

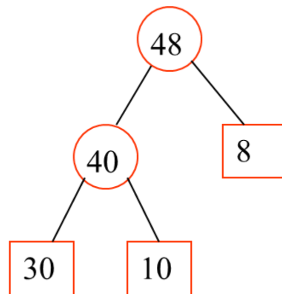


Fig. 3: Merge tree

Another way is to merge tree L2 and L3 and finally merging it with L1. This is shown in Fig. 4.

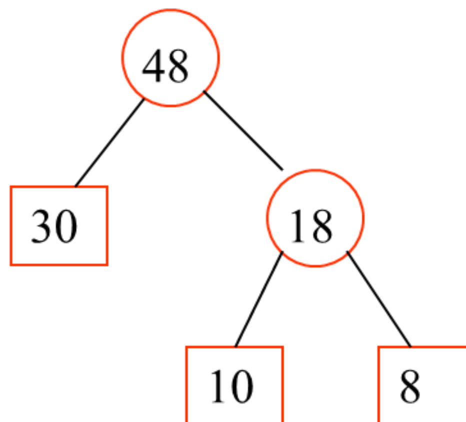


Fig. 4 : Merge tree

The cost of the merged tree is given as cost = 30 * 1 + 10 *2 + 8*2 = 66.It can be observed that the second tree shown in Fig. 4 is optimal as its cost is lesser.

Applications of optimal tree-

Optimal merge tree is useful in many applications such as message coding and decoding. It is also useful in lossless compression algorithms. For example, the messages can be approximated by list length and in that case the weighted external path length equals the transmission time.
The optimal merge tree can be given as follows:

Input: m sorted lists, L_i, $i = 1, 2, \dots, m$, each L_i consisting of n_i elements.
Output: An optimal 2-way merge tree.
Step 1. Generate m trees, where each tree has exactly one node (external node) with weight n_i.
Step 2. Choose two trees T_1 and T_2 greedily with minimal weights.
Step 3. Create a new tree T whose root has T_1 and T_2 as its sub trees andweight are equal to the sum of weights of T_1 and T_2.
Step 4. Let T Replace T_1 and T_2.
Step 5. If there is only one tree left, stop and return; otherwise, go to Step 2.
The optimal merge tree algorithm can be implemented using priority lists. The algorithm can beformally written as follows:
Algorithm optimal merge (m)
Problem: Finding Optimal Merge
Input: m files
Output: optimal merge tree and its cost
Begin
1. Store the files in priority Queue based on the length
2. For index = 1 to m-1 do
   a. Choose two nodes extracted from the priority queue with minimal weights
   b. Merge the nodes and insert it as a new node in the priority queue
3. Calculate weight W (T) of the entire tree

and return T.End.

As said earlier, this algorithm is useful for optimum merging. The use of heap data structure eliminates the requirement of sorting and hence results in an effective algorithm.
Example 4: Consider 6 sorted lists with lengths 2, 3, 5, 7, 11 and 13. Apply the optimal mergealgorithm and construct optimal merge tree.

Solution:

The algorithm uses greedy approach. It finds two lists of least length and merges. This is show inFig. 5.

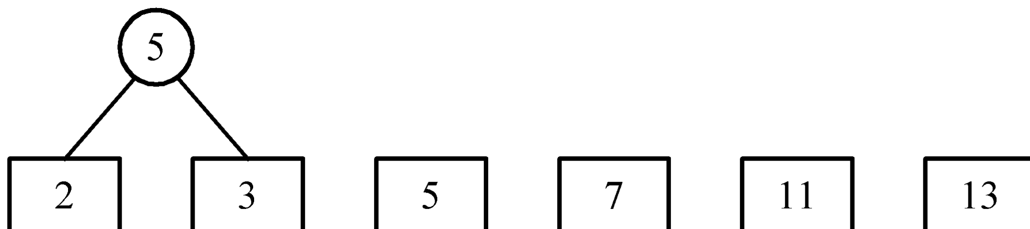


Fig. 5 : Initial merge

The second level merging is shown in Fig.6

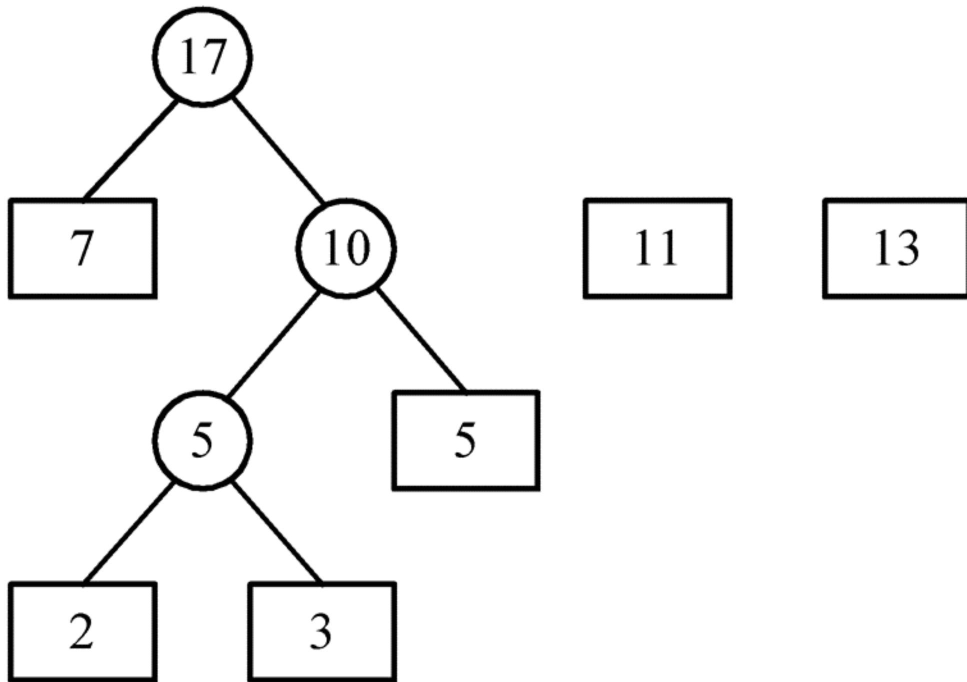


Fig. 6 : Next Level merge

Then, the next two least cost nodes are merged and shown in Fig. 7.

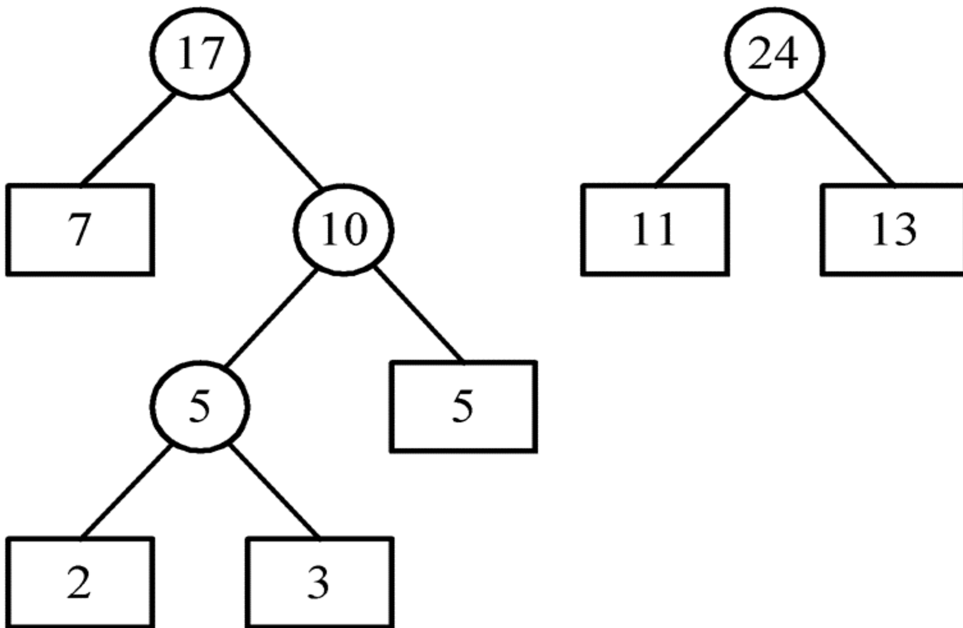


Fig. 7: Next Level merge

The final merge is shown in Fig. 8.

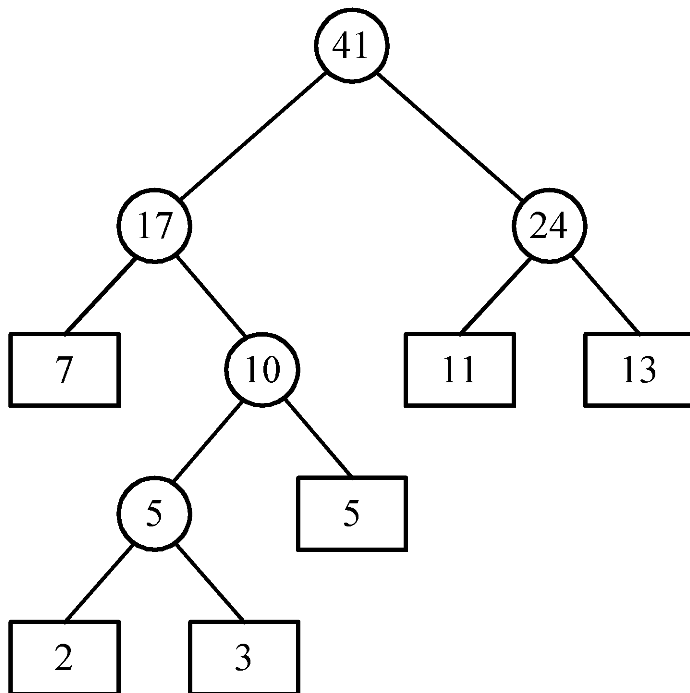


Fig. 8: Final Level Merges

Complexity Analysis:

Insertion of an node in a tree would take θ(n) time. If min – heap data structure is used, then the root would contain minimum element and hence finding the least cost in heap would take θ (1) time .The loop takes at most n-1 times. Therefore, the complexity of the algorithm is θ (n).Then removing the root would take θ (log n) time. The loop takes at most n-1 steps. Therefore, for generating optimal extended binary tree would be θ(n log n).

1.5 Single-source Shortest Path Problems

There is a necessity for finding shortest path. Let G be the given graph and the edge cost now represent the length. The length of a path is defined to be the sum of the weights of the edges. The shortest path problem is one of finding the shortest distance between the source and all othervertices. The algorithm for finding shortest path was proposed by Edsger W. Djikstra.
There are many applications of the shortest path algorithm. Some of the applications are internet protocol, Flight reservations and directions seeking in driving.
Certain limitations of Djikstra algorithm are listed below:

1. Edge length should not be negative
2. The graph G should be acyclic – that is no cycles are permitted.

The idea of finding shortest path is to find the shortest among all shortest paths (from the source), then find the second shortest, etc., breaking ties arbitrarily, until all shortest paths between source and destination are found.
Thus, the problem of finding shortest path is modeled as a graph, where the vertices represent cities and edge weight models the distances between cities. The weights of the edges can represent telecommunication costs also. The initial node is called a source. For each of the remaining nodes, find a shortest path connected from the source
Informally, the algorithm based on [2 , 3] can be written as follows:

Input: *W* [1..*n*][1..*n*] with *W*[*i*, *j*] = weight of edge (*i*, *j*); set *W*[*i*, *j*] = ∞ if no edge
Output: an array *Dist* [2..*n*] of distances of shortest paths to each node in [2..*n*]

C = {2,3,…,*n*} // the set of remaining nodes
for *i* = 2 to *n* do *Dist*[*i*] = *W*[1,*i*] // initialize distance from node 1 to node *i*
// delete node *v* from set *C*
For each node *w* in *C* do

if (*Dist*[v] + *W*[*v*, *w*] < *Dist*[w]) then

Dist[w] = *Dist*[v] + *W*[*v*, *w*] // update *Dist*[w] if found shorter path to *w*

Example 5
Consider the following graph (Refer Fig. 9) and find the shortest path from the source node.

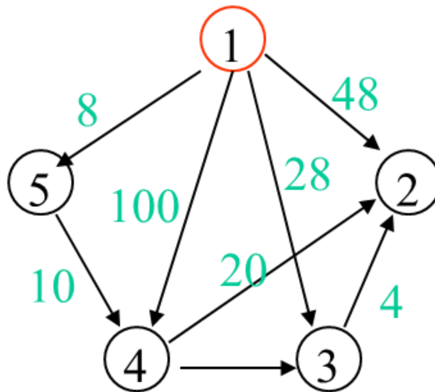
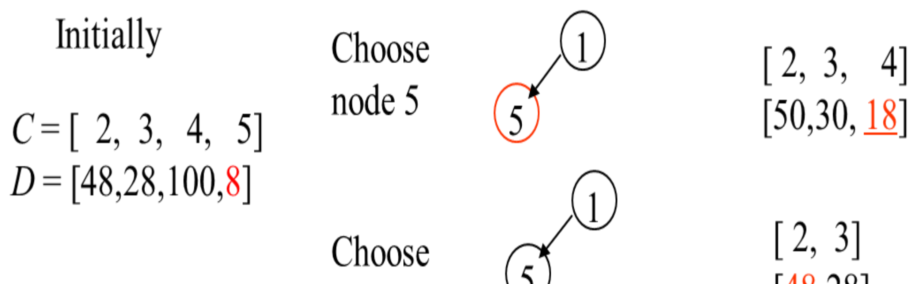


Fig. 9: Sample Graph

Solution:

The idea is chose node 1 and subsequently chose the next node based on the least distance.This It can be observed that the next node chosen is 5. Then node 4 is chosen



But it can be observed that all the shortest paths are from source only.

Complexity Analysis

The total time required for the execution of statements inside the *while* loop is $O(n^2)$. Therefore,the time required to compute the SSSP algorithm is $O(n^2)$.

Check Your Progress

1. What is the Optimal Merge Problem?
A. Finding the best way to combine two sorted arrays into a single sorted array.
B. Determining the most efficient way to merge unsorted arrays.
C. Optimizing the concatenation of two arrays without any sorting.
D. Minimizing the number of comparisons in a sorting algorithm.

2. In the context of the Optimal Merge Problem, what does "merge" refer to?
A. Combining two sorted arrays into a single sorted array.
B. Concatenating two arrays without any specific order.
C. Sorting two arrays independently.
D. Dividing an array into two halves.

3. In the context of the Optimal Merge Problem, what does "merge" refer to?
A. Combining two sorted arrays into a single sorted array.
B. Concatenating two arrays without any specific order.
C. Sorting two arrays independently.
D. Dividing an array into two halves.

4. What is the time complexity of the optimal solution to the Optimal Merge Problem?
A. O (n^2)
B. O (log n)
C. O (n log n)
D. O (n)

5. What is the Shortest Path Problem?
A. Determining the longest path between two nodes in a graph.
B. Finding the most direct route between two nodes in a graph.
C. Identifying the path with the maximum number of edges in a graph.
D. Calculating the average distance between all nodes in a graph.

6. In the context of the Shortest Path Problem, what does "weight" refer to?
A. The physical weight of the edges in the graph.
B. The distance or cost associated with traversing an edge.
C. The number of nodes in a path.
D. The density of the graph.

7. In the Shortest Path Problem, what does a negative edge weight represent?
A. A forbidden path.
B. An undefined weight.
C. A shortcut or faster route.
D. A detour or longer route.

Answers to check your progress

1. A
2. A
3. C
4. C
5. B
6. B

Model Questions

1. What is Huffman algorithm for file compression?
2. What is Huffman coding used for?
3. What is minimum spanning tree with example?
4. How do you solve Kruskals algorithm?
5. What is the time complexity of Prims algorithm?

6. What is Prim's algorithm with example?
7. Which is easiest algorithm for shortest path?
8. Explain the Huffman algorithm and its type in data compression?

Block-3
Unit- 10

1.0 Learning objectives

1.1 Dynamic Programming

Check Your Progress

Answers to check your Progress

1.2 Fibonacci Sequence

Check your Progress

Answers to check your Progress

1.3 Binomial Coefficient

Check your Progress

Answers to check your Progress

1.4 Transitive Closure

Check your Progress

Answers to check your Progress

1.5 Shortest path algorithm

Check your Progress

Answers to check your Progress

1.6 Multistage Graph problem

Check your Progress

Answers to check your Progress

1.7 Traveling Salesman Problem

Check your Progress

Answers to check your Progress

1.8 Chained matrix multiplication

Check your Progress

Answers to check your Progress

1.9 Bellman –ford algorithm

Check your Progress

Answers to check your Progress
Model Questions

1.0 Learning objectives-

After completing this unit, the learner will be able-

- To Understand the Dynamic Programming
- To Understand the Fibonacci Sequence and Binomial Coefficient
- To Understand the Transitive Closure and all pair of shortest path algorithm
- To Understand the Multistage Graph problem and Traveling Salesman Problem
- To Understand the Chained matrix multiplication and for finding shortest path

1.1 Dynamic Programming

Dynamic programming is useful for solving multistage optimization problems, especially sequentialdecision problems. Richard Bellman is widely considered as the father of dynamic programming. He was an American mathematician. Richard Bellman is also credited with the coining of the word "Dynamic programming". Here, the word "dynamic" refers to some sort of time reference and "programming" is interpreted as planning or tabulation rather than programming that is encountered in computer programs.
Dynamic programming is used in variety of applications. Dynamic programming (DP) is used to solve discrete optimization problems such as scheduling, string-editing, packaging, and inventory management.
Dynamic programming employs the following steps as shown below-

Step 1: The given problem is divided into a number of sub problems as in "divide and conquer" strategy. But in divide and conquer, the sub problems are independent of each other but in dynamic programming case, there are all overlapping sub problems. A recursive formulation is formed of the given problem.

Step 2: The problem, usually solved in the bottom-up manner. To avoid, repeated computation of multiple overlapping sub problems, a table is created. Whenever a sub problem is solved, then its solution is stored in the table so that in future its solutions can be reused. Then the solutions are combined to solve the overall problem.

There are certain rules that govern dynamic programming. One is called Principle of optimality.**Rule 1**
Bellman's principle of optimality states that at a current state, the optimal policy depends only on the current state and independent of the decisions that are taken in the previous stages. This is called the principle of optimality.
In simple words, the optimal solution to the given problem has optimal solution for all the sub problems that are contained in the given problem.

Rule 2

Dynamic programming problems have overlapping sub problems. So, the idea is to solve smaller instances once and records solutions in a table. This is called memorization, a corrupted word of memorization.

Rule 3

Dynamic programming computes in bottom-up fashion. Thus, the solutions of the sub problems are extracted and combined to give solution to the original problem.

Check your Progress –

1. What is dynamic programming in data structures?
   A. A technique for designing efficient algorithms by breaking down a problem into smaller sub problems.
   B. A way to store and organize data in a computer program
   C. A process of optimizing memory usage in a program
   D. A method for creating algorithms that use only constant space.

2. Which of the following is not a characteristic of dynamic programming?
   A. Overlapping sub problems
   B. Optimal substructure
   C. Recursion
   D. Divide and conquer

3. Which of the following is an example of a problem that can be solved using dynamic programming?
   A. Sorting a list of integers in ascending order
   B. Finding the shortest path between two nodes in a graph
   C. Computing the nth Fibonacci number
   D. Calculating the greatest common divisor of two numbers

4. What is memorization in dynamic programming?
   A. The process of storing solutions to sub problems in memory
   B. A way to optimize the use of memory in a program
   C. A technique for creating algorithms that use only constant space
   D. A method for breaking down a problem into smaller sub problems

Answer to check your progress-

1- A
2- D
3- C
4- A

1.2 Fibonacci Sequence

The Fibonacci sequence is given as (0, 1, 2, 3, 5, 8, 13. . .). It was given by Leonardo of Pisa.The Fibonacci recurrence equation is given below:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

Conventional Algorithm

The conventional pseudo code for implementing the recursive equation is given below:Fib1 (N)

```
{          if (N =< 1)
                   return  1;
           else
                   return Fib(N-1) + Fib(N-2)
}
```

This straight forward implementation is inefficient. This is illustrated in the following Fig. 1.

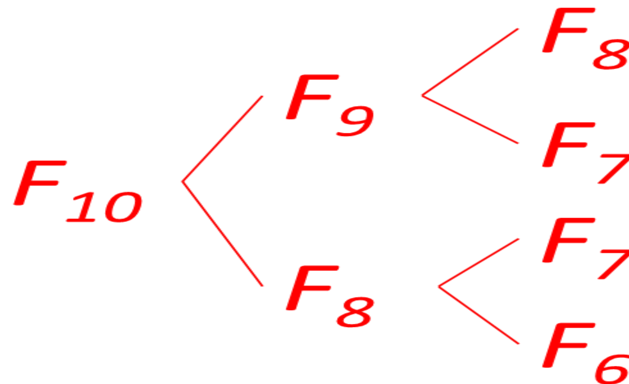


Fig. 1: A portion of the Recurrence Tree

Complexity Analysis:

It can be observed, based on Fig. 1, that there are multiple overlapping sub problems. As 'n' becomes large, the number of sub problems also would increase exponentially. This leads to repeated computation and thus the algorithm becomes ineffective. The complexity analysis of this algorithm T (n) =$(\emptyset^n)$.

Here $\emptyset$ is called golden ratio whose value is given.

Dynamic Programming approach:

The best way to solve this problem is to use dynamic programming approach. The dynamic programming approach uses an approach of storing the results of the intermediate problems. Hence the key is to reuse the results of the previously computed sub problems rather than re computing the sub problems repeatedly. As a sub problem is computed only once, this exponential algorithm is reduced to a polynomial algorithm. To store the intermediate results, a table is created and its values are reused. This table is shown in Table 1.0.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}

0	1	1	2	3	5	8	13	21	34	55

Table 1.0: Fibonacci Table

The tabular computation can avoid computation as the intermediate results can be used instead of recomputed. The algorithm based on this approach is given below:
An iterative version of this algorithm can be given as follows:

```
 Fib2 (n)
{        int Fn = 1, Fn1 = 1,
         Fn2 = 1 for(I = 2; I
         <= n; I++)
         {        Fn       =
                  Fn1      +
                  Fn2  Fn2
                  = Fn1
                  Fn1 = Fn
         }
         return Fn
}
```

Complexity Analysis:

It can be observed that two variables 'Fn1' and 'Fn2' track Fibonacci (n-1) and Fibonacci (n) to compute Fibonacci (n+1). As repeat condition spans only n-1 times, the complexity analysis of this algorithm is ≅ O(n). This is far better than exponential conventional algorithm.

Check your progress-

1. Suppose the first Fibonacci number is 0 and the second is 1. What is the sixth Fibonacci number?
   A. 5
   B. 4
   C. 2
   D. 8

2. Which of the following option is wrong?
   A. Fibonacci number can be calculated by using Dynamic programming
   B. Fibonacci number can be calculated by using Recursion method
   C. Fibonacci number can be calculated by using Iteration method
   D. No method is defined to calculate Fibonacci number

3. Which of the following recurrence relations can be used to find the nth Fibonacci number?
   A. F(n) = F(n) + F(n – 1)
   B. F(n) = F(n) + F(n + 1)
   C. F(n) = F(n – 1)
   D. d) F(n) = F(n – 1) + F(n – 2)

4. Which of the following is the correct Fibonacci number sequence?
   A. 0, 1, 2, 3, 4, 5, etc.
   B. 1,8,27,64,125,etc.
   C. 1,1,2,2,4,8,32,256,etc.
   D. 1,1,2,3,5,8,13,etc.

Answer to check your progress-

1- A
2- D
3- D
4- D

1.3 Binomial Coefficient

Binomial coefficient can be obtained using this formula

$$\begin{bmatrix} n \\ k \end{bmatrix} = \frac{n!}{k!(n-k)!} \text{ for } 0 \leq k \leq n.$$

The conventional algorithm to implement the above formula is given as below:

```
Int bin (int n, int k)

    {
if (k = 0 or n = k )
return 1;
            else

                    return (bin(n-1, k-1) + bin (n-1, k))
}
```

But, the difficulty with this formula is that the factorial of a number can be very large. For example, the factorial

Of 49! = 608,281,864,034,267,560,872,252,163,321,295,376,887,552,831,379,210,240,000,000,000.

Therefore, the application of conventional formula is difficult for large value of ‘n’. Dynamic Programming approach:

The dynamic programming approach can be applied for this problem. The recursive formulation of binomial coefficient is given as follows:

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{cases} \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + \begin{bmatrix} n-1 \\ k \end{bmatrix} & 0 < \mathrm{k} < \mathrm{n} \\ 1 & \mathrm{k} = 0 \text{ or } \mathrm{k} = \mathrm{n} \end{cases}$$

The Dynamic Programming approach for this problem would be

1. Divide the problem $\binom{n}{k}$ into many sub problems like $\binom{0}{0}, \binom{1}{0}, \binom{1}{1}, \ldots, \binom{n}{n}$. A bottom-up approach can be used to solve the problem from scratch.

2. To avoid recomputation of the sub problems, a table can be used where the results of the sub problems can be stored.

This would be similar to Pascal triangle as given below in Fig. 2.

c(0,0)
c(1,0) c(1,1)
c(2,0) c(2,1) c(2,2)
c(3,0) c(3,1) c(3,2) c(3,3)
c(4,0) c(4,1) c(4,2) c(4,3) c(4,4)

Fig. 2: Pascal Triangle

It can be observed that, each row depends only on the preceding row. Therefore, only linear space and quadratic time are needed.

The Table for computation is given below in Table 2.0.

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
.						
.						
.						
n-1					$C(n-1,k-1)$	$C(n-1,k)$
n						$C(n,k)$

This algorithm is known as Pascal's Triangle
The formal algorithm is given below:

```
Int bin (int n, int k)

{

        int i, j;
        int B[0..n,
        0..k]; for i
        = 0 to n
                for j = 0 to minimum (i, k)
                        if ( j = 0 or j = i)
                                B[i, j] = 1;
                        else
                                B[i, j] = B[i-1, j-1] + B[i-1, j];
        return B[n, k]
}
```

Complexity Analysis:

The complexity analysis of this algorithm can be observed as O (nk) as the algorithm has two loops that spans from 1 to n. Hence the algorithm body gets executed at most n^2 times. ∴ The complexity of the algorithm is O(nk) and the space complexity is also O (nk).

Check your progress-

Choose the correct one-

1. What does the binomial coefficient C(n, k) represent?

A. The product of n and k.

B. The number of ways to choose k elements from a set of n elements.

C. The exponentiation of n to the power of k.

D. The quotient of n divided by k.

2. How is the binomial coefficient C (n, k) calculated?

A. C (n,k)=n+k

B. C (n,k)= k!n!

C. C (n,k)=n×k

D. C (n,k)= n/k

3.What is the symmetry property of binomial coefficients?

A. C(n, k) = C(k, n)

B. C (n, k) = C (n - k)

C. C (n, k) = C (n - 1, k - 1)

D. C (n, k) = C (n + k)

4. Which mathematical function is closely related to binomial coefficients?

A. Exponential function.

B. Logarithmic function.

C. Trigonometric function.

D. Factorial function.

Answer to check your progress-

1- B

2- D

2- A

3- D

1.4 Transitive Closure

Let us consider some important aspects of binary relations first:

A binary relation R is said exists between two vertices, say u and v, can be mathematically representedas u R v. A binary relation that is considered here is a path relation and hence a relation u R v indicates that, there is a path from u to v

What is a transitive relation? A transitive relation states that if there is a binary relation between u to v and v to w, then there exists a relation from u to w. This is illustrated in Fig. 1.

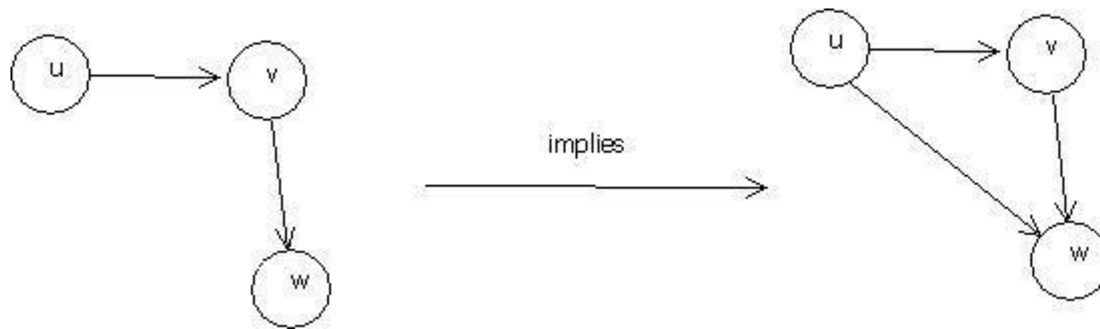


Fig. 1: Illustration of Transitive Closure [1]

If R is a set of transitive relations, then the adjacency matrix of R is called reachability matrix,connectivity matrix pr path matrix.

In other words, the matrix B can be said as

$$B = A + A^2 + A^3 + \ldots\ldots\ldots + A^n$$

And path matrix P is denoted as

P_{ij} = 1 if ij^{th} entry of matrix B is not zero

0 otherwise

If the entry value of Path matrix is 1, then it indicates that there exists a path and if the value is zero, it indicates that the path is absent. This is given mathematically as follows:

$$P_{ij} = \begin{cases} 1 \text{ if there is an dges between} V_i \text{ and } V_j \\ 0 \text{ if there is no edge between vertices } V_i \text{ and } V_j \end{cases}$$

Warshall Algorithm

Warshall algorithm is used to construct transitive closure of a matrix. This is done as transitive closure *T* as the last matrix in the sequence of *n*-by-*n* matrices

$P^{(0)}, \ldots, P^{(k)}, \ldots, P^{(n)}$, where $P^{(0)} = A$. *Here A is* adjacency matrix*.*

The key idea of this algorithm is , on the *k*-th iteration, the algorithm determines for every pair of vertices *i, j* if a path exists from *i* and *j* with just vertices 1,…,*k* allowed as intermediate vertices.

The recurrence equation of this algorithm is given as below:

$$P^{(k)}[i,j] = P^{(k-1)}[i,j] \text{ or } (P^{(k-1)}[i,k] \text{ and } P^{(k-1)}[k,j])$$

The rules of constructing $P^{(k)}$ from $P^{(k-1)}$ is given below:

Rule 1 If an element in row *i* and column *j* is 1 in $P^{(k-1)}$,
it remains 1 in $P^{(k)}$

Rule 2 If an element in row *i* and column *j* is 0 in $P^{(k-1)}$,it has to be changed to 1 in $P^{(k)}$ if and only if the element in its row *i* and column *k* and the elementin its column *j* and row *k* are both 1's in $P^{(k-1)}$

This algorithm is illustrated in the following Example.

Example 1: Find the transitive closure of the following graph shown in Fig. 2.

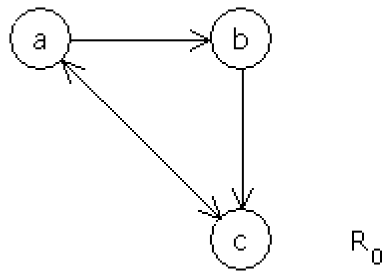


Fig. 2: Sample Graph

Solution:

As discussed earlier, the initial adjacency matrix is given as follows:

	A	B	C
A	0	1	0
B	0	0	1
C	1	0	0

It can be observed that, the entry 1 indicates the presence of edge and 0 indicates the absence ofedge. With R1 node as intermediate node, the adjacency matrix is changed as follows:

	A	B	c
A	0	1	0
B	0	0	1
C	1	1	0

It can be observed that now path between c to b is possible with the availability of node ‘a’. Now node R2 is made available, that is nodes 1 and 2, the adjacency matrix is changed as follows:

	A	B	c
A	0	1	1

B	0	0	1
C	1	0	0

It can be seen, the path between nodes a and c is possible,

Now node R3 is made available. This means, all the nodes are available. The adjacency matrix ischanged as follows:

	A	B	c
A	1	1	1
B	1	1	1
C	1	1	1

Thus one can observe that there is connectivity between all the nodes.

The formal Warshall algorithm is given as follows:

Algorithm warshall (G,A)

```
Begin
 for i = 1 to n
      for j = 1 to n        %%    Initialize
      P[i,j] = A[i,j]
     End   for
 End for
  for  k  =  1  to  n
   for i = 1 to n
      For j =1 to n         %% Initialize
        P[i,j,k] = P[i,j,k-1] Λ (P[i,k,k-1]  ΛP[k,j,k-1])  %%End for
     End   for
   End for
return   P
End.
```

Complexity Analysis

What is the complexity analysis of Warshall algorithm? It can be seen, the algorithm is reducedto filling the table. If both number of rows and columns are same, the algorithm complexity is reduced to $\Theta(n^3)$ as there are three loops involved. The space complexity is given as $\Theta(n^2)$.

Check your progress-

1. What is the key advantage of the Warshall algorithm?

A. It works efficiently on graphs with negative weights.
B. It guarantees finding the globally optimal solution.
C. It has a lower time complexity than Dijkstra's algorithm.
D. It is particularly suited for sparse graphs.

2. Which of the following is a suitable application of the Warshall algorithm?
A. Shortest path in a weighted graph.
B. Minimum spanning tree construction.
C. Determining reachability in a directed graph.
D. Maximum flow in a network.

3. In the Warshall algorithm, what does the recursive formula express?
A. The shortest path between two nodes.
B. The maximum flow in a graph.
C. The transitive closure of a graph.
D. The shortest paths between all pairs of nodes.

4. What is the time complexity of the Warshall algorithm for finding the transitive closure of a graph with n vertices?
A. O (n)
B. O(n log n)
C. O(n^2)
D. O(n^3)

Answer to check your progress-

1- A
2- C
3- C
4- D

1.5 Shortest path

- **Floyd Algorithm**

Floyd algorithm finds shortest paths in a graph. It is known as all pair shortest path algorithm as the algorithm finds shortest path from any vertex to any other vertex. The problem can be formulated as follows: Given a graph and edges with weights, compute the weight of the shortest path between pairs of vertices.

Can the transitive closure algorithm be applied here? Yes. Floyd algorithm is an variant of this algorithm. In a weighted digraph, Floyd algorithm finds shortest paths between every pair of vertices. There is only one restriction as no negative edge allowed in Floyd algorithm. This is illustrated in the following Fig 3 .

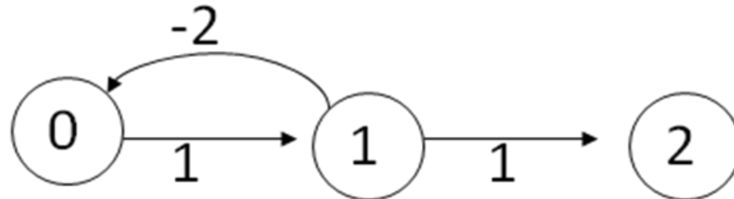


Fig. 3: Sample Graph

It can be observed that the shortest path from vertex 0 o vertex 2 tends to be -∞. In reality, the distance can not be negative and there exists a positive path of 2 from vertex 0 to vertex 2. Thisis the reason why Floyd algorithm fails when edge weight is negative.

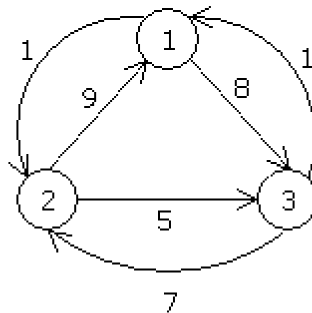
One way to solve this problem is to apply shortest path algorithms like Djikstra's algorithm 'n' times between all possible combinations of vertices with every path takes O(n^3).

Instead, Floyd algorithm can be tried. Floyd algorithm have the following initial conditions:

It represent the graph G by its cost adjacency matrix with cost[i][j]. If the edge <i,j> is not in G,the cost[i][j] is set to some sufficiently large number. D[i][j] is the cost of the shortest path form ito j, using only those intermediate vertices with an index <= k.

The recursive relation of the algorithm is given as follows:

$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i,k) + D^{k-1}(k, j)\}, \qquad k \geq 1$



Example 2: Find the shortest path for the following graph as shown in Fig. 4.

Fig. 4: Sample graph.

The edge weights are given in the following adjacency matrix

	1	2	3
1	**0**	**1**	**8**

2	9	0	5
3	1	7	0

Solution:

D[0] is same as the adjacency matrix. Therefore, The given adjacency matrix is D[0]. Vertex 1 is made intermediate node. With the availability of this node 1, the path between vertex 3 and vertex 2 is possible. This results in the modified path matrix.

$$D[1] = \begin{bmatrix} 0 & 1 & 8 \\ 9 & 0 & 5 \\ 1 & 2 & 0 \end{bmatrix}$$

In the next iteration, nodes 1 and 2 are made temporary. Therefore, the modified path matrix is given as follows:

$$D[2] = \begin{bmatrix} 0 & 1 & 6 \\ 9 & 0 & 5 \\ 1 & 2 & 0 \end{bmatrix}$$

In the next iteration, all the three nodes are available. This results in the modified path matrix asgiven below:

$$D[3] = \begin{bmatrix} 0 & 1 & 6 \\ 6 & 0 & 5 \\ 1 & 2 & 0 \end{bmatrix}$$

This is the final path matrix.

The formal Floyd algorithm is given as follows:

Algorithm Floyd-Marshall (G,s,t)

```
begin
  for i = 1 to n
      for j = 1 to n
        D[i,j]  =  A[i,j]  %%  Initialize
      endfor
    Endfor
D[0] = aij
    for k = 1 to n for i
      = 1 to n
        for j = 1 to n
          D[i,j] = min{D[i,j] , D[i,k] + D[k,j]}endfor
      endfor
    endfor
  end
```

Complexity Analysis

The complexity analysis of Floyd algorithm is same as Warshall algorithm. Time complexity of this algorithm is Θ (n^3) as there are three loops involved. The space complexity is given as Θ(n^2) similar to Warshall algorithm.

Check your progress-

1. What type of algorithm is the Floyd-Warshall algorithm?
A. Greedy Algorithm.
B. Dynamic Programming Algorithm.
C. Divide and Conquer Algorithm.
D. Backtracking Algorithm.

2. What problem does the Floyd-Warshall algorithm solve efficiently?
A. Shortest Path Problem.
B. Minimum Spanning Tree Problem.
C. Maximum Flow Problem.
D. Traveling Salesman Problem.

3. In the context of the Floyd-Warshall algorithm, what does the term "transitive closure" refer to?
A. Determining if a graph is connected.
B. Finding the shortest paths between all pairs of nodes.
C. Identifying strongly connected components.
D. Determining reachability between all pairs of nodes.

4. What is the time complexity of the Floyd-Warshall algorithm for finding the shortest paths between all pairs of nodes in a graph with n vertices?
A. O(n)
B. O(n log n)
C. O(n^2)
D. O(n^3)

5. Which data structure is typically used to implement the Floyd-Warshall algorithm efficiently?
A. Priority Queue.
B. Stack.
C. Adjacency Matrix.
D. Hash Table.

Answer to check your progress-

1- B
2- A
3- D
4- D
5- C

1.6 The Multistage Graph Problem

The idea for Stage coach problem is that a salesman is travelling from one town to another town, in the old west. His means of travel is a stagecoach. Each leg of his trip cost a certain amount and he wants to find the minimum cost of his trip, given multiple paths. A sample multistage graph is shown in Fig. 2. And different stage transitions are shown in Fig. 3.

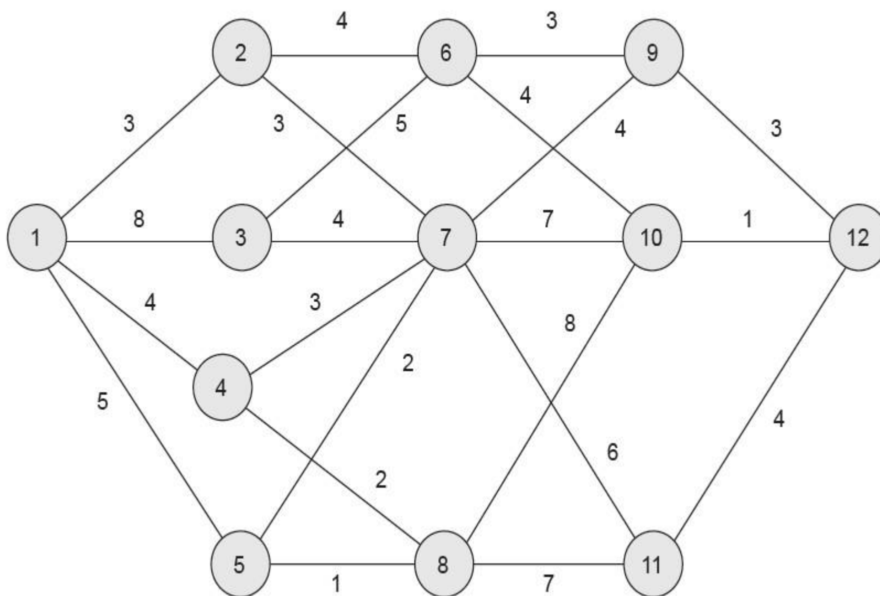


Fig. 2 Sample Multistage Graph

	2	3	4	5
i=1	3	8	4	5

Stage 1-2

	6	7	8
i=2	4	3	2
i=3	5	4	-
i=4	-	3	2
i=5	-	2	1

Stage 2-3

	9	10	11
i=6	3	4	-
i=7	4	7	6
i=8	-	8	7

Stage 3-4

	12
i=9	3
i=10	1
i=11	4

Stage 4-5

Fig. 3: Different Stages of Multistage Graph

Solution:

Stage 4-5: There are three possibilities for going to destination given that one is at points 9, 10 or 11.

cost (4,9) = 3

cost (4,10) = 1

cost (4,11) = 4

cost (3,6) = min $\begin{cases} 3 + cost(4,9) = 3 + 3 = 6 \\ 4 + cost(4,10) = 4 + 1 = 5^* \end{cases}$

Cost(3,7) = min $\begin{cases} 4+cost(4,10)= 4+3=7^* \\ 7+cost(4,10)=7+1=8 \\ 6+cost(4,11)=6+4=10 \end{cases}$

Cost(3,8) = min $\begin{cases} 8 + cost(4,10) = 8 + 1 = 9^* \\ 7 + cost(4,11) = 7 + 4 = 11 \end{cases}$

Stage 2-3: The optimal choices at stages 2 can be given as follows:

Cost(2,2) = min $\begin{cases} 4 + cost(3,6) = 4 + 5 = 9^* \\ 3 + cost(3,7) = 3 + 7 = 10 \end{cases}$

Cost (2,3) = min $\begin{cases} 5 + cost(3,6) = 5 + 5 = 10 \\ 4 + cost(3,7) = 4 + 7 = 11 \end{cases}$

Cost(2,4) = min $\begin{cases} 3 + cost(3,7) = 3 + 7 = 10^* \\ 4 + cost(3,8) = 4 + 9 = 13 \end{cases}$

Cost (2,5) =min $\begin{cases} 2 + cost(3,7) = 2 + 7 = 9 \\ 2 + cost(3.8) = 2 + 9 = 11 \end{cases}$

Cost(1,5) = min $\begin{cases} \min 3 + cost(2,2) = 3 + 9 = 12^{*} \\ 8 + cost(2,3) = 8 + 10 = 18 \\ 4 + cost(2,4) = 4 + 10 = 14 \\ 5 + cost(2,5) = 5 + 9 = 14 \end{cases}$

The formal algorithm is given as follows:

Algorithm F graph(G)

Begin cost
= 0
n = |V|

stage = n-1

while (j <= stage) do

Choose a vertex k such that C[j,k] + cost k is minimum.cost[j]
= c[j,k]+cost(k)
j = j-1

Add the cost of C(j,r) to record

d[j] = k End
while

return cost[j]
end

The path recovery is done as follows:

Algorithm path (G,d,n,k)

Begin

n = |V|

stage = n-1

for j = 2 to stage
path[j] = d[path [j-1]]
End for

End.

Complexity Analysis:

Time efficiency: $\Theta(n^3)$
Space efficiency: $\Theta(n^2)$.

Backward Reasoning

A sample graph is shown in Fig. 4. Let us solve this problem using backward reasoning.

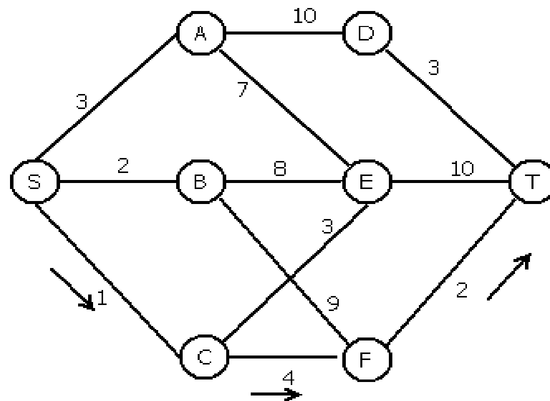


Fig. 4: Sample Graph

Backward approach starts like this

C(S , A) = 3

C(S , B) = 2

C(S , C) = 1

C(S , D) = min{ 10 + C(S , A) }

= min { 10 + 3 } = 13.

C(S , E) = min{ 7 + C(S , A) , 8 + C(S , B) , 3 + C(S , C) }

= min { 7 + 3 , 8 + 2 , 3 + 1 }

= min { 10 , 10 , 4 } = 4

C(S , F) = min { 9 + Cost(S , B) , 4 + Cost(S , C) }

= min{ 9 + 3 , 4 + 1} = 5

D(S , T) = min{ 3 + C(S ,D) , 10 + C(S , E) , 2 + C(S , F) }

=min{ 3 + 13 , 10 + 4 , 2 + 5 } = 7.

The path can be recovered as follows:

T → F →C →

Check your progress-

1. Identify the correct problem for multistage graph from the list given below.
   - A. Resource allocation problem
   - B. Traveling salesperson problem
   - C. Producer consumer problem
   - D. Barber’s problem

2. Which of the following statements is true about the multistage graph problem?

A. It can be solved using any graph traversal algorithm.

B. It is a special case of the Traveling Salesman Problem.

C. The optimal solution can be found using greedy algorithms.

D. Dynamic programming is a common approach to solve it.

3. What does the term "topological sorting" refer to in the context of multistage graphs?
A. Arranging nodes in a way that avoids cycles.
B. Sorting nodes based on their weights.
C. Sorting nodes based on their stages.
D. Arranging nodes in alphabetical order.

4. Which of the following is not a step in solving the multistage graph problem using dynamic programming?
A. Initialization.
B. Forward Pass.
C. Backward Pass.
D. Topological Sorting.

5. In a multistage graph, what is a sink node?
A. A node with no outgoing edges.
B. A node with no incoming edges.
C. The last stage node.
D. The first stage node.

Answer to check your progress-

1- A
2- D
3- C
4- D
5- B

1.7 Travelling Salesperson Problem (TSP)

Travelling salesman problem (TSP) is an interesting problem. It can be stated as follows: Given a set of n cities and distances between the cities in the form a graph. TSP finds a tour that start and terminate in the source city. The restriction is that, every other cities should be visited exactly once and the focus isto find the tour of shortest length
It can be said as a function, f(i, s) , shortest sub-tour given that we are at city i and still have to visit thecities in s (and return to home city). In other words, we move from city I to city j and focus is that all cities needs to be visited except city j and should be back to city i.
Let g (i, S) be the length of a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1.
This can be stated formulated as follows:

$$g(1, V-\{1\}) = MIN_{2 \le k \le n} \{c_{1k} + g(k, V-\{1, k\})\}$$

This represents the optimal tour. In general, the recursive function is given as follows:

$$g(i, S) = \min_{j \in S}\{c_{ij} + g(j, S-\{j\})\}$$

This concept is illustrated through this numerical example.

Example 1: Apply dynamic programming for the following graph as shown in Fig. 1. and find the optimal TSP tour.

The traveling salesman problem involves visiting each city how many times?

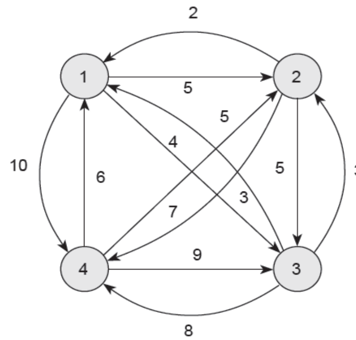


Fig. 1: Sample Graph.

Solution:

Assuming that the starting city 1, let us find the optimal tour. Let us initialize s = null and find the cost as follows:

$g(2,\phi) = C_{21} = 2$
$g(3,\phi) = C_{31} = 4$
$g(4,\phi) = C_{41} = 6$

Let us consider size =1 , then the possible values are {1}, {2}, {3}.

$g(2,\{3\}) = C23 + g(3,\phi)$

$= 5 + 4 = 9$ $g(2,\{4\}) =$

$C24 + g(4,\phi)$

$= 7 + 6 = 13$ $g(3,\{2\}) =$

$C32 + g(2,\phi)$

$= 3 + 2 = 5$ $g(3,\{4\}) =$

$C34 + g(4,\phi)$

$= 8 + 6 = 14$ $g(4,\{2\}) =$

$C42 + g(2,\phi)$

$= 5 + 2 = 7$ $g(4,\{3\}) =$

$C43 + g(3,\phi)$

$= 9 + 4 = 13$

Let the size s increased by 1, that is, |s| = 2

Now, *g(1,s)* is computed $|s|=2,\ i \neq 1,\ 1 \notin s,\ i \notin n$, i.e, involves two intermediatenodes.

g(2,{3,4}) = min{ C23 + g(3,{4}) , C24 + g(4,{3}) }

= min{ 5 + 14 , 7 + 13 }

= min{ 19 , 20 } = 19.

g(3,{2,4}) = min{ C32 + g(2,{4}) , C34 + g(4,{2}) }

= min{ 3 + 13 , 8 + 7 }

= min{ 16 , 15 } = 15.

g(4,{2,3}) = min{ C42 + g(2,{3}) , C43 + g(3,{2}) }

= min{ 5 + 9 , 9 + 5 }

= min{ 14 , 14 } = 14.

Let the size is increased by 1, that is, |s| = 3

Finally, the total cost is calculated involving three intermediate nodes. That is $|s| = 3$.As $|s| = n-1$, where n is the number of nodes, the process terminates.

g(1,{2,3,4}) = min{ C12 + g(2,{3,4}) , C13 + g(3,{2,4}) , C14 + g(4,{2,3}) }

= min{ 5 + 19 , 3 + 15 , 10 + 14 }

= min {24 ,18 , 24 }

= 18.

Hence, the minimum cost tour is 18. The path can be constructed by noting down the *'k'* thatyielded the minimum value. It can be seen that the minimum was possible via route 3.

∴ *P(1,{2,3,4}) = 3.* Thus tour goes from *1→3.* It can be seen that *C(3,{2,4})* ^

Minimum.

∴ *P(3,{2,4}) = 2*

Hence the path goes like *1 → 3 → 2 and the final TSP tour is given as 1 →3 → 2 →1*

Complexity Analysis:

N+ $\sum_{k=2}^{2}(n-1)\binom{n-2}{n-k}(n-k)$

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

Check your progress-

---

1. The travelling salesman problem involves visiting each city how many times?

A. 0

B. 1

C. 2

D. 3

2. A weighted graph has what associated with each edge in travelling salesmen problem?

A. A cost

B. Nothing

C. Direction

D. Size

3. What is the travelling salesman problem equivalent to in graph theory?
A. Any circuit.
B. A Hamilton circuit in a non-weighted graph.
C. A round trip airfare.
D. A Hamilton circuit in a weighted graph.

4. Travelling salesman problem is an example of-
A. Dynamic Algorithm
B. Greedy Algorithm
C. Recursive Approach
D. Divide & Conquer.

Answer to check your progress-

1- B
2- A
3- D
4- B

1.8 The_Chained matrix multiplication-

Chained matrix multiplication is an interesting problem. It can be formally stated as follows:
Given a sequence or chain $A_1, A_2, \ldots, A_n$ of *n* matrices to be multiplied, then How to compute the product $A_1A_2\ldots A_n$

It must be observed that Matrix Multiplication is not commutative. That is AB ≠ BA. On the other hand, matrix multiplication follows associative law, i.e., (AB) C = A (BC). Hence, given 'n' matrices, there are many orders in which matrix multiplication can be carried out. In other words, there are many possible ways of placing parenthesis Chained matrix multiplication is a problem of multiplying matrix multiplications such that it is low cost.

Example 1: Consider the chain A_1, A_2, A_3, A_4 of 4 matrices. Show some of the ways in which the matrix multiplication can be carried out?

Solution: Some of the ways the matrices can be multiplied are as follows:

1. $(A_1(A_2(A_3A_4)))$ **2.** $(A_1((A_2A_3)A_4))$ **3.** $((A_1A_2)(A_3A_4))$, **4.** $((A_1(A_2A_3))A_4)$ **5.** $(((A_1A_2)A_3)A_4)$

The ordering seems to follow Catalan sequence. It follows the following recursive relationship.

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^{n} C_i\, C_{n-i} \quad \text{for } n \geq 0;$$

The first few Catalan numbers for n = 0, 1, 2, 3, … are **1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …**

Brute force algorithm

One way to solve this problem is to use brute force method by listing out all the possible ways and choosing the best possible way.

The following algorithm segment shows how the matrix multiplication is carried out in atraditional manner.
Input: Matrices $A_{m\times n}$ and $B_{n\times r}$ (with dimensions m×n and n×r)
Output: Matrix $C_{m\times r}$ resulting from the product $A\cdot B$

```
for i ← 1 to m
        for j ← 1 to r
              C [i, j+ ← 0
              For k ← 1 to n
    C[i, j+ ← C[i, j] + A[i, k] · B[k, j]
                          return c
```

Complexity analysis-
The cost of multiplication is mnr where m,n and rare dimensions of matrices A and B.
By changing the order of the matrices, the optimization can be carried out. The following example illustrates the advantages of changing the order of the matrices.
Example 2: Consider three matrices $A_{2\times 3}$, $B_{3\times 4}$, and $C_{4\times 5}$. Show two different orders and find the optimal way of multiplying these matrices.
Solution:
There are 2 ways to parenthesize. One way is to multiply these three matrices as - ((AB)C) = $D_{2\times 4} \cdot C_{4\times 5}$. This is done as follows:

- AB → 2 x 3 x4 = 24 scalar multiplications
- DC → 2 x 4 x 5 = 40 scalar multiplications
- Total = 24 + 40 = 64 multiplications.

Another way is to multiply this as follows; (A(BC)) = $A_{2\times 4} \cdot E_{3\times 5}$

- BC → 3 x 4 x 5 = 60 scalar multiplications
- AE → 2 x 3 x 5 = 30 scalar multiplications
- Total = 60 + 30 = 90 scalar multiplications.

So, it can be observed that the optimal way of matrix multiplication is ((AB) C).
From this example, it can be noted that cost and order matters. The optimal matrix multiplication reduces the cost. In other words, Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i-1}\times p_i$
It is necessary to parenthesize the product $A_1A_2\ldots A_n$ such that the total number of scalar multiplicationsis minimized.

Dynamic programming idea:
The idea is to apply dynamic programming to find chained matrix multiplication. An optimal parenthesization of the product $A_1A_2\ldots A_n$ splits the product between A_1 and A_k for some integer k where$1 \le k < n$
First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$.
Dynamic programming solution requires formulation of the recursive formula. The recursive formula can be formulated as follows:

Let $C[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
Minimum cost to compute $A_{1..n}$ is $C[1, n]$

– Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} forsome integer *k* where $i \leq k < j$

The recursive formulation is given as follows:

$C[i, j] = C[i, k] + C[k+1, j] + p_{i-1} p_k p_j$ *for* $i \leq k < j$ *With* $C[i, i] = 0$ for $i=1,2,\ldots,n$ *(Initial Condition)*

The informal algorithm for chained matrix multiplication is given as follows:

1. Read n chain of matrices
2. Compute C[i,j] recursively and fill the table
3. Compute R[i,j] to keep track of k that yields minimum cost
4. Return C[1,n] as minimum cost.

The formal algorithm is given as follows:

```
Algorithm dp_chainmult(p,n)Begin
    for i = 1 to n  do
        C[ i , j ] = 0
    end for
    for diagonal = 1 to n-1
      for  i = 1 to n-
        diagonal j = i +
        diagonal
            C[ i , j ] = ∞
            for k = 1 to j-1 do

              if C[ i , j ] < C[ i , k ]+C[ k+1 , j]+
                                    pi-1  ×pk  ×  pj   the
```

```
C[i,j]= C[ i , k ]+C[ k+1 , j]+
                              R[i,j] = k
                         else
                              C[i,j]      =
                              C[i,j] R[i,j]
                              = k
                         End
                         if
                         End
                         for
                       End
                     for end
                     for

           return C[1,n]
```

Complexity Analysis

The algorithm has three loops ranging from 1 to n. Therefore, the complexity analysis of this algorithm is $O(n^3)$ and the algorithm just needs to fill up the table. If the rows and columns are assumed to be equal, the space complexity of this algorithm is $O(n^2)$.

Example 3: Apply dynamic programming algorithm and apply for this for the following four matrices withthe dimension given as below as in Table 1.

Table 1: Initial matrices with dimensions given

A	B	C	D
4×5	5×3	3×2	2×7
P_0 P_1	P_1 P_2	P_2 P_3	P_3 P_4

Solution

Based on the recurrence equation, one can observe that C[1,1] = 0 ; C[2,2] = 0 ; C[3,3] = 0; C[4,4] = 0 This is shown in Table 2.

Table 2: Initial Table

0			
	0		
		0	
			0

Using the recursive formula, the table entries can be computed as follows:

$C[1,2] = C[1,1] + C[2,2] + P_0 \cdot P_1 \cdot P_2$
$= 0 + 0 + 4 \times 5 \times 3 = 60$

$C[2,3] = C[2,2] + C[3,3] + P_1 \cdot P_2 \cdot P_3$
$= 0 + 0 + 5 \times 3 \times 2 = 30$

$C[3,4] = C[3,3] + C[4,4] + P_2 \cdot P_3 \cdot P_4$
$= 0 + 0 + 3 \times 2 \times 7 = 42$

This is shown in Table 3.

Table 3: **After First Diagonal**

0	60		
	0	30	
		0	42
			0

$C[1,3] = C[1,1] + C[2,3] + P_0 \cdot P_1 \cdot P_3$
$= 0 + 30 + 4 \times 5 \times 2$
$= 30 + 40 = 70$ *(k = 1)*

$C[1,3] = C[1,2] + C[3,3] + P_0 \cdot P_2 \cdot P_3$
$= 60 + 0 + 4 \times 3 \times 2$

$= 84$ $\quad\quad (k = 2)$

The minimum is 70 when $k = 1$

$$C[2,4] = C[2,2] + C[3,4] + P_1{\cdot}P_2{\cdot}P_4$$
$$= 0 + 42 + 5 \times 3 \times 7$$
$$= 42 + 105 = 147 \quad\quad (k = 2)$$
$$C[2,4] = C[2,3] + C[4,4] + P_1{\cdot}P_3{\cdot}P_4$$
$$= 30 + 0 + 5 \times 2 \times 7$$
$$= 30 + 70 = 100 \quad\quad (k = 3)$$

The minimum is 100 when k =3. The matrix is shown in table 4.

Table 4: After second diagonal computation

0	60	70	
	0	30	100
		0	42
			0

Now, C[1,4] is computed.

$$C[1,4] = C[1,1] + C[2,4] + P_0{\cdot}P_1{\cdot}P_4$$
$$= 0 + 10\,0 + 4 \times 5 \times 7$$
$$= 100 + 140 = 200 \quad\quad (k = 1)$$

$$C[1,4] = C[1,2] + C[3,4] + P_0{\cdot}P_2{\cdot}P_4$$
$$= 60 + 42 + 4 \times 3 \times 7$$
$$= 60 + 42 + 84 = 186 \quad\quad (k = 2)$$
$$C[1,4] = C[1,3] + C[4,4] + P_0{\cdot}P_3{\cdot}P_4$$
$$= 70 + 0 + 4 \times 2 \times 7$$
$$= 70 + 56 = 126 \quad\quad (k = 3)$$

The minimum is 126 and this happens when k = 3.
The resulting matrix is shown in table 5.

Table 5: Final Table

0	60	70	126
	0	30	100

		0	42
			0

The order can be obtained by finding minimum k that yielded the least cost. A table can be created and filled up with this minimum k. this is given as shown in Table 6.

Table 6: Table of minimum k

0	1	1	3
	0	2	3
		0	3
			0

From the table, one can reconstruct the matrix order as follows;

[A (B C) D]

Check your Progress-

1. Which of the following methods can be used to solve the matrix chain multiplication problem?

A. Dynamic Programming
B. Recursion
C. Brute force
D. Dynamic Programming, Brute force, Recursion

2. Consider the two matrices P and Q which are 10 x 20 and 20 x 30 matrices respectively. What is the number of multiplications required to multiply the two matrices?

A. 10*20
B. 20*30
C. 10*30
D. 10*20*30

3. Consider the matrices P, Q, R and S which are 20 x 15, 15 x 30, 30 x 5 and 5 x 40 matrices respectively. What is the minimum number of multiplications required to multiply the four matrices?
   - A. 6050
   - B. 7500
   - C. 7750
   - D. 12000

4. Consider the brute force implementation in which we find all the possible ways of multiplying the given set of n matrices. What is the time complexity of this implementation?
   - A. O(n!)
   - B. $O(n^3)$
   - C. $O(n^2)$
   - D. Exponential

Answer to check your progress-

1- D
2- D
3- A
4- D

1.9 Bellman-Ford Algorithm-

Bellman-ford algorithm is another interesting algorithm that can be used to find shortest path. Bellman-Ford algorithm is more general algorithm than Dijkstra's algorithm and it removes one of the majpr limitations of Djikstra algorithm as the edge-weights can be negative. The algorithm detects the existence of negative-weight cycle(s) reachable from source vertex s.
The major differences between Bellman-Ford algorithm and Djikstra algorithm is given below in Table 7.

Bellman-Ford Algorithm	Dijkstra Algorithm

1. Allow negative length edges but does not allow negative cycle. 2. Slower than Djkstra algorithm if more edges are present.	1. Does not allow both negative weights and negative cycle. 2. Faster than Bellman-Fordalgorithm

Table 7: Differences between bellman-Ford and Djikstra algorithms.

The concept of Bellman-Ford algorithm is based on the concept of relaxing. The principle of relaxing is the estimation of actual distance using vertex cost and edge weight so that the distance becomes gradually optimal. This is illustrated below in Fig. 1.

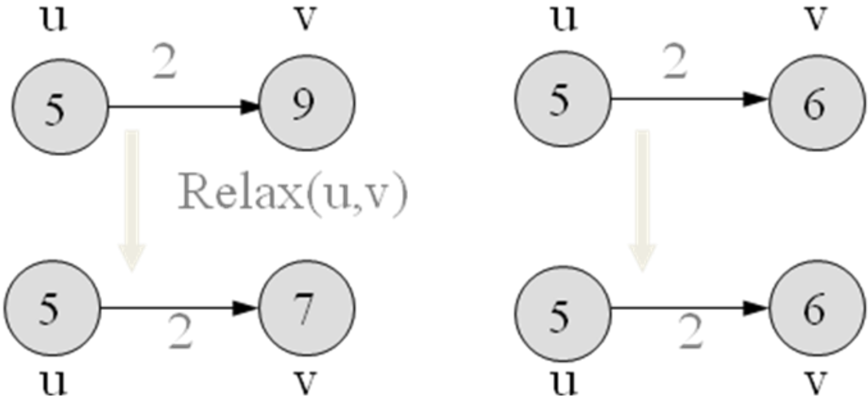


Fig.1: Example of Relaxation

The logic of this relaxing is given below:

RELAX(u, v)
if d[v] > d[u]+w (u,v)
then d*v+ ← d*u++w(u,v)

It can be seen, the vertex v weight is 9 greater that weight of u , i.e., 5 and edge weight is 2. Therefore, it can be noted that it is relaxed to 7. On the other hand, if the weight is less than weight of u and ege weight, it is left undisturbed.

The informal algorithm is given as follows:

1. Choose the source vertex and label it as '0'
2. Label all vertices except the source as ∞
3. Repeat n-1 times where n is the number of vertices

   3a. If label of v is larger than label of u + cost of the edge (u,v) then
   Relax the edge

   3b. Update the label of v as label of v as label of u + cost of the edge (u,v)

4. Check the presence of negative edge cycle by repeating the iteration and

   Carry out the procedure if still any edges relax. If so, report the presence of negative weight cycle

The formal algorithm is given as follows

Algorithm Bellman (G,w,s)

Begin

$$d(s) = 0$$

$$\text{for all other vertices } v,\ v \neq s$$

$$d[v] = \infty$$

%% Initialization

N = | V(G) |

Repeat N-1 times

for each edge (u,v)∈E(G) do

%% relax

if dist(u) +cost(u,v) < dist(v) then

dist(v) = dist(u) +cost(u,v);

end if

end for

end

for each edge (u,v) ∈Es (G) do

%% Check for negative cycle

if d(v) > d[u]+ cost(u,v) then

Output "negative edge cycle is present" ;

end if

End

The algorithm is illustrated with the following example.

Example 4:

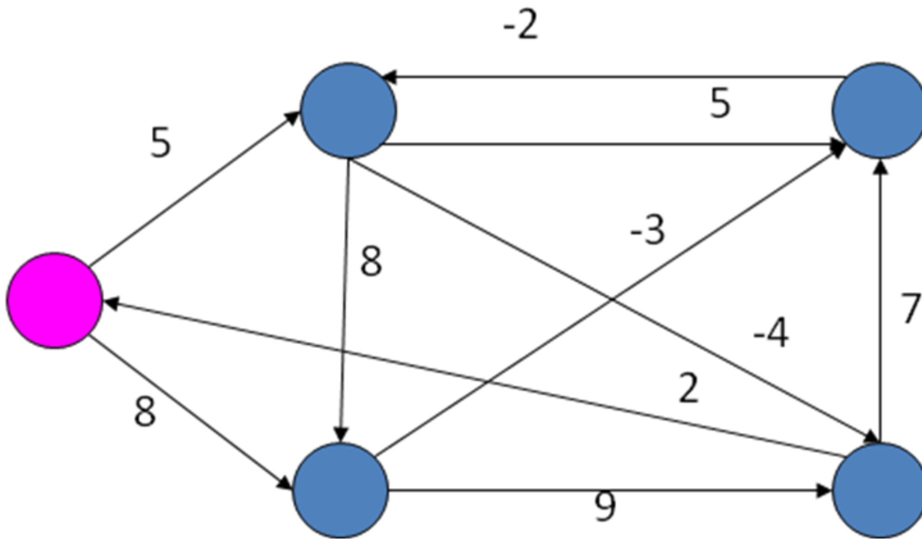


Fig. 2: Sample Graph

Apply Bellman-Ford algorithm and find the shortest path from source node.

Solution

After initial relaxing, this is given in Fig. 3.

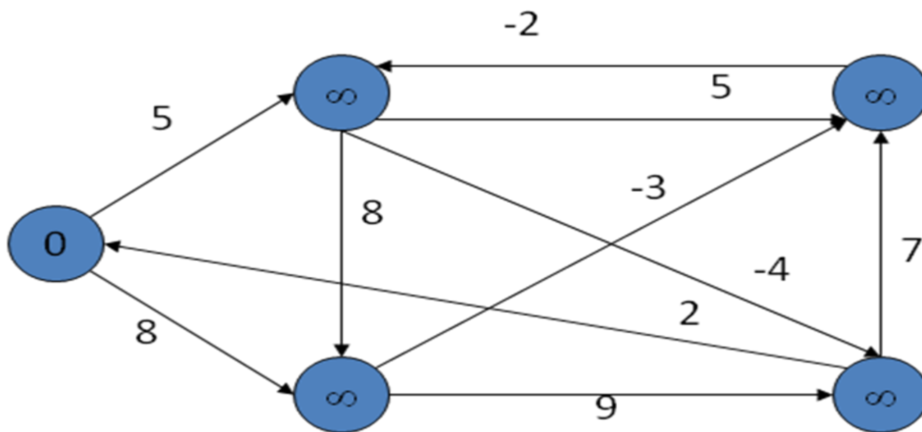


Fig. 3: After initial Relaxing

Further relaxation gives the graph as shown below in Fig. 4.

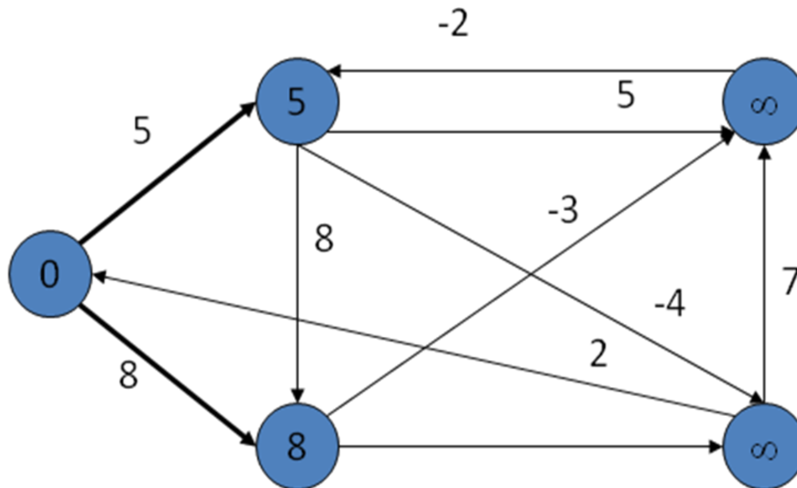


Fig. 4: After further Relaxing – with relaxed edges given in dark lines.

Further relaxation is given in Fig. 5.

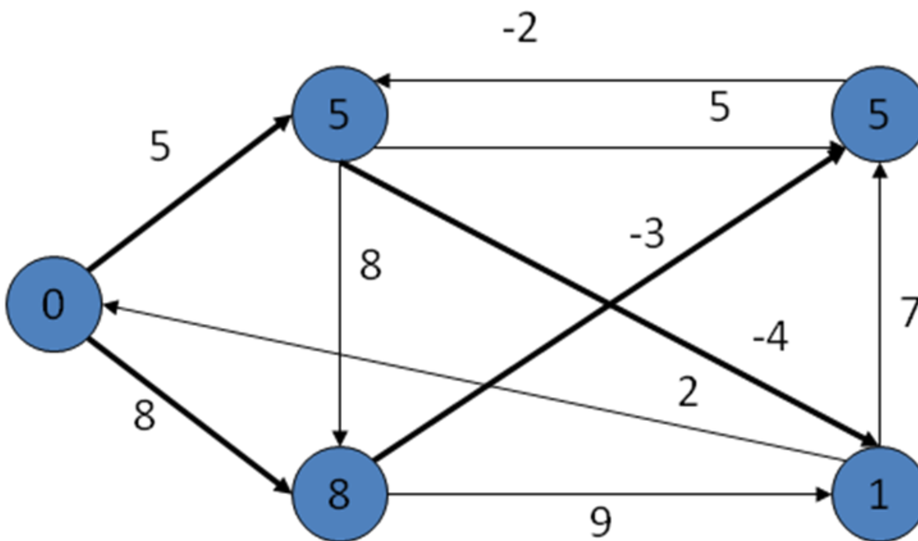


Fig. 5: After further Relaxing – with relaxed edges given in dark lines.

Further relaxation results in Fig. 6.

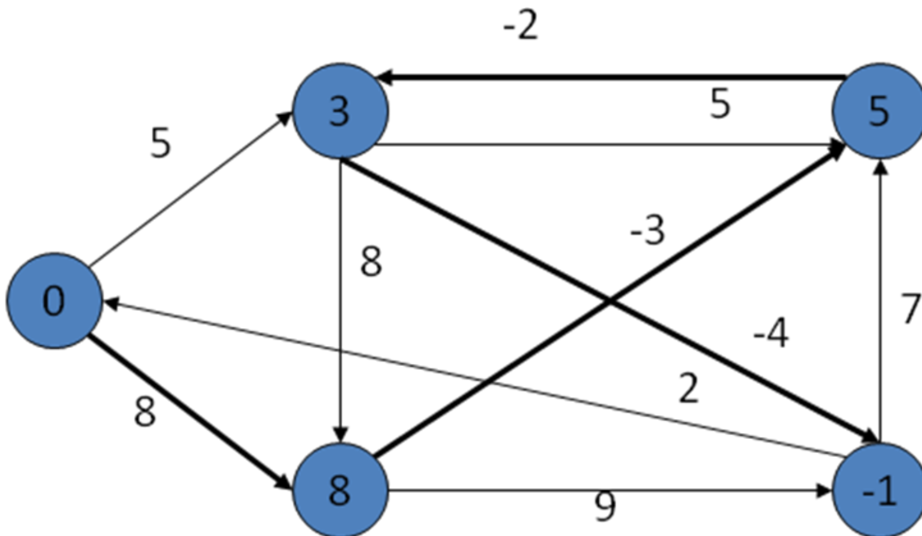


Fig. 6: After further Relaxing – with relaxed edges given in dark lines.

After the final relaxation, the final graph is shown in 7.

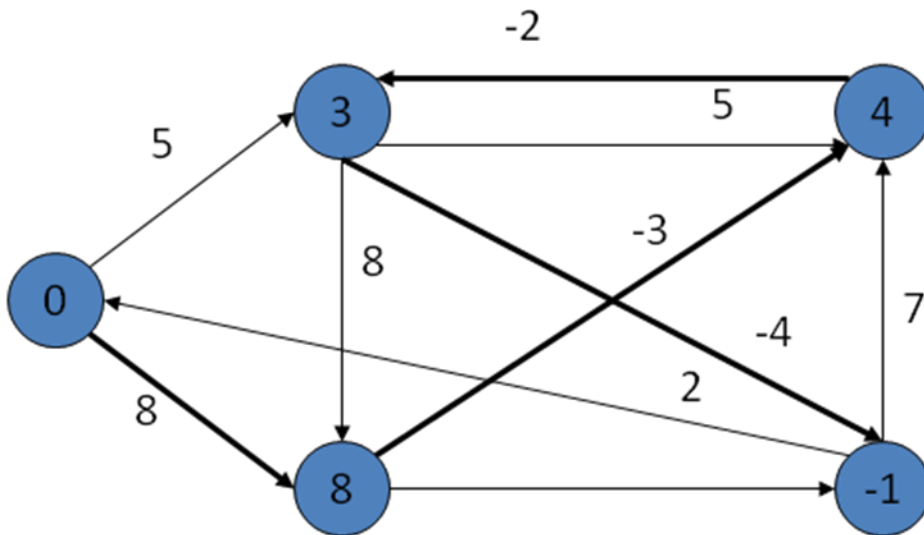


Fig. 6: After final Relaxing – with relaxed edges given in dark lines.

From this, one can find the shortest path from any node to the source node.

Complexity Analysis:

The algorithm takes O(n^2), if the graph has 'n' vertices and 'n' edges.

Check your progress-

1. The Bellmann Ford algorithm returns _______ value.
A. Boolean
B. Integer
C. String
D. Double

2. Bellmann ford algorithm provides solution for ____________ problems.
   A. All pair shortest path
   B. Sorting
   C. Network flow
   D. Single source Shortest path

3. Bellmann Ford Algorithm can be applied for _____________
   A. Undirected and weighted graphs
   B. Undirected and unweighted graphs
   C. Directed and weighted graphs
   D. All directed graphs

4. Bellmann Ford Algorithm is an example for ____________
   A. Dynamic Programming
   B. Greedy Algorithms
   C. Linear Programming
   D. Branch and Bound

Answer to check your progress-

1- A
2- D
3- C
4- A

Model Questions

1. What is Dynamic programming?
2. What is Fibonacci sequence?
3. How do you calculate the binomial coefficient?
4. How do you find the transitive closure of a set?

5. How do you calculate the transitive closure?
6. What is the problem of shortest path algorithm?
7. What are the features of a multi stages graph problem?
8. Which algorithm is used in travelling salesmen problem?
9. What is the good solution to the travelling salesmen problem?
10. What is the basic principle of Bellman – Ford algorithm?

Block -3
Unit 11: Longest Common Subsequence

1.0 Learning objectives

1.1 Longest Common Subsequence

Check your Progress
Answers to check your Progress

1.2 String Edit Problem

Check your Progress
Answers to check your Progress

1.3 Binary Search Tree

Check your Progress
Answers to check your Progress

1.4 Optimal Binary Search Tree

Check your Progress
Answers to check your Progress

1.5 Knapsack Problem

Check your Progress
Answers to check your Progress

1.6 Flow Shop Scheduling

Check your Progress
Answers to check your Progress

1.7 Concept of computational complexity

Check your Progress
Answers to check your
Model Questions

1.0 Learning objectives

After completing this unit, the learner will be able-

- To Understand the Longest Common Sub sequence
- To Understand the String Edit Problem and Binary Search Tree
- To Understand the Optimal Binary search Tree and Knapsack Problem
- To Understand the Flow shop scheduling and Concept of computational complexity
- To Understand the upper bound theory and Lower Bound Theory and Proof of lower bound theory

1.1 Longest common problem

Line numbers, strings also can be manipulated. In many applications, string manipulation is necessary. One of the important algorithms in string manipulation is finding longest common subsequence. Let us discuss about them now:
What is a subsequence? A subsequence of a string is simply some subset of the letters in the whole string in the order they appear in the string. For example, FOO is a subsequence of FOOD. A subsequence of a sequence can be obtained by deleting zero or more characters keeping the remaining characters of the sequence in its original order.
The longest Common sequence between string A and B is the longest (i.e., not shorter than) the other common subsequence that are derivable from strings A and B.
Some of the examples of valid and invalid subsequences are given below:

Example1: Show whether the following subsequences shown in Figs1-2 are valid or not?

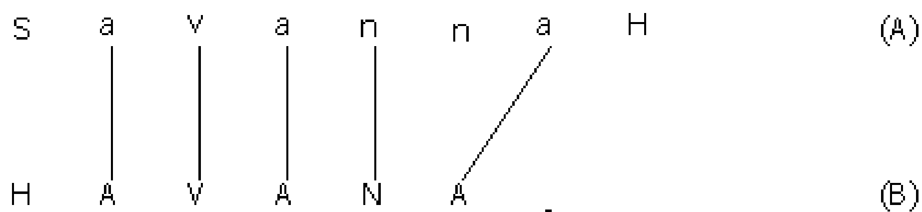


Fig.1: A sample String A and B

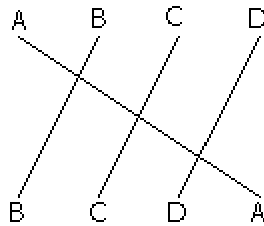


Fig.2: A sample String A and B

Solution-

The mapping between the strings is given in Fig. 1. The longest subsequence is AVANA. It is valid as per the mapping.
In Fig. 2, BCD is the longest common subsequence. One cannot cross match character 'A' of string B with string 'A' as shown in Fig. 2. In short, the direction of finding should be from left to right and mappings should not cross each other. So the second mapping is invalid.
Brute Force method:
One easiest way to find common subsequence is to generate all possible subsequences of string A and B and finding the length of the subsequences. From this information, one can find longest common subsequence between strings A and B. But unfortunately, this strategy will not work as the computational complexity of this algorithm is $\Theta\ (2^k)$. Clearly, this algorithm is exponential.
Dynamic Programming Approach
Dynamic programming can be applied to solve this problem. The first step of dynamic programming approach is to formulate recursive relation that guide the problem. Let us assume that LCS (i,j) is the longest common subsequence of strings x and y. Let us formulate there cursive relation.

- The common subsequence between empty string and any other string is zero. In other words,
  $LCS\ (i,j) = 0$
- If the last characters of both strings s1 and s2 match, then the LCS = 1 + the LCS of both of the strings with their last characters removed. For example, if the strings x and y are FOOD and MOOD, then one can observe that the last character of strings x and y are matching. Therefore, LCS is equal to1+LCSof 'FOO' of sequence FOOD and 'MOO' of sequence MOOD.
- If the initial character so both strings do NOT match ,then the LCS will be one of two options:
  1) The LCS of x and y without its last character.
  2) The LCS of y and x without its last character.

For example, if the strings are "BIR" and "FIN", then longest common subsequence is Max (LCS ("BI" , "FIN"). In other words, one will then take the maximum of the 2 values.
Putting all these conditions, the recursive formulation is given as-

$$\text{LCS}\ (i|j) = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ \max\{LCS(i, j-1)\} & if\ i > 0, j > 0\ and\ x \neq y \\ 1 + LCS\ (i-1, j-1) & if\ i > 0, and\ x_i = y_j \end{cases}$$

The informal algorithm for finding longest common subsequence based on given as follows:

1. Check the last character of the string x and y.

   a. If the last character matches, then delete the last character of both strings. The LCS would be the LCS of 1+ Strings x and y without the last character.

   b. If the last character does not match, then the LCS would be the maximum of LCS of (First word, second word minus the last character) and LCS of (Second word, First word minus the last character).

2. ConstructthesolutionofLCSofxandyusingtheLCSofsubstringsusingstep1.

The formal algorithm based is given as follows:

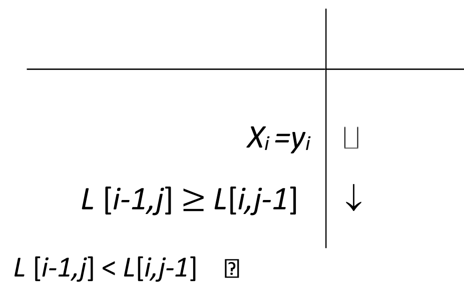
```
Algorithm LCS ( x , y )
Begin
        for j = 0 to n do
            L( 0 , j ) = 0
        End for

        for i = 0 to m do
            L( i , 0 ) = 0
        End for
```

```
for j = 1 to n do
        Case (x(i) , y(j)) of
        x(i) ≠ y(j) and [ L( i-1 , j ) ≥ L( i , j-1 )] : L( i , j ) = L( i-1 , j );
        x(i) ≠ y(j) and [ L( i-1 , j ) < L( i , j-1 )] : L( i , j ) = L( i , j-1 );
        x(i) = y(j): L( i , j ) = 1 + L( i-1 , j-1 );
        End Case
    End for
  Return L(m,n)
  End.
```

L(j,j)=max(L(i-1,j),L(I,j-1)];

One can find the longest common subsequence length using the above algorithm. But in order to find out the actual subsequence, one has to trace the sequence. For this, the arrow marks are put for the sake of tracing based on these conditions.



Example2: Find the Longest common subsequence between two strings "SAVANNAH" and "HAVANNAH" using dynamic programming approach.

Solution: One can apply dynamic programming approach to the sequence and the resultant of the algorithm is shown below:

	0	S	✓A	✓V	✓A	✓N	N	✓A	H
0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	1
✓A	0	0	1	1	1	1	1	1	1
✓V	0	0	1	2	2	2	2	2	2
✓A	0	0	1	2	3	3	3	3	3
✓N	0	0	1	2	3	4	4	4	4
✓A	0	0	1	2	3	4	4	5	5

A V A N A (5)

It can be observed that the longest common subsequence is AVANA.

Check your Progress-

1.What is the Longest Common Subsequence (LCS) of two strings?

A. The longest substring that appears in both strings in the same order

B. The longest substring that appears in both strings in any order

C. The longest substring that appears in only one of the strings

D. The shortest substring that appears in both strings

2. Which dynamic programming technique is commonly used to solve the LCS problem efficiently?

A. Divide and Conquer

B. Greedy Algorithm

C. Backtracking

D. Dynamic Programming

3. What is the time complexity of the dynamic programming approach for solving the LCS problem?

A. O (n)

B. O (n^2)

C. O (2^n)

D. O (n!)

4. LCS is used in which of the following applications?

A. DNA sequence analysis

B. Spell checking

C. Version control systems

D. All of the above

Answer to check your progress-

1. A
2. D
3. B
4. D

1. 2 String Edit Problem

String Edit is another interesting problem in computer science domain .It has many applications, such as spellcheckers, natural language translation, and bioinformatics.

What is a string edit problem? Given an **initial string s**, and a **target string t**, what is the minimum number of changes that have to be applied to ***s*** to turn it into ***t?*** Edit Distance is defined as the minimum number of edits needed to transform one string into the other. This distance is called Edit distance or Levenshtein distance.

Let us discuss about the edit operations that are used to find edit distance. The rules are given below: The list of valid changes for string edit is

1) Inserting a character
2) Deleting a character
3) Changing a character to another character.

Substitution is one of the operations. It mentions the number of substitutions necessary to transform one string to another.

Substitution

Consider the following example:

A B C D

A B Y D

It can be observed that at least one substitutions are required (A, B and D are common) to transform the first string to second string i .e. , C → Y.

Insertion is another operation.

Insertion

Consider the following example:

A B _ _

A B Y D

The blanks are given as "-". It can be observed that at least two insertions are required to convert the first string to the second string. Always for insertion, the blank is given in the firststring.

Similarly, the deletion is also another operation.
Consider the following example:

A B Y D

A B _ D

The blanks are given "-" in the second string .It can be observed that at least one deletions required to convert the first string to the second string.
Based on these, the informal algorithm for finding edit distance is given below:
If either string is empty, return the length of the other string.

1) If the last characters of both strings match, recursively find the edit distance between each of the strings without that last character.

2) If they don't match then return1 +minimum value of the following three choices:
   a) Recursive call with the string s w/o its last character and the string t
   b) Recursive call with the string s and the string t w/o its last character
   c) Recursive call with the string s w/o its last character and the string t w/o its last character.

Thus, the recursive relationship is given below:
Thus the edit distance is the minimum of above factors. Thus, it can be said as

E [i-1,j]+1

E[i,j-1]+1

E [i,j]=min *E[i-1,j-1]+1* *ifA[i]≠B[j]*

E[i-1,j-1] *ifA[i]=B[j]*

The formal algorithm is given below:

Algorithm *Edit (*
A,B)Begin

for i = 1 to n
do Edit(0
, j) = 0

End for

for j = 0 to m do

Edit(i , 0
) = 0End for

for j = 1 to n do

same

Case (x(i) , y(j)) of

x(i) = y(j): Edit(i , j) = 1 + Edit(i-1 , j-1); %% All characters are

x(i) ≠ y(j) :

Edit(i,j) =

E [i-1 , j] + 1

E [i , j-1] + 1

min E [i-1 , j-1] + 1 if A[i] ≠ B[j]

End Case

End for

Return L(m,n)

End

The algorithm is illustrated through the following example [1].**Example 3:** Find the edit distance between strings “XYZ” and “ABC” **Solution:**

The initial table is given below inTable1.

Table 1; Initial Table

		A	B	C
	0	1	2	3
X	1			
Y	2			
Z	3			

It can be observe that the ε is an empty string. For example changing X to Ɛ requires one operation. One can apply the recursive relation to this table and one can check the final table is given in Table 2 as shown below:

Table2: Final Table

		A	B	C
	0	1	2	3

X	1	1	2	3
Y	2	2	2	3
Z	3	3	3	3

It can be checked that the minimum distanceis3. In other words, minimum three operations are required to transform the string “XYZ” to “ABC”.

Complexity Analysis:

What is the complexity analysis of this algorithm? There are n^2(i.e., if m=n) entries in the table, and each entry would take θ*(1)* time. In general, the complexity of time and space would be θ *(mn)* respectively i.e., the product of rows and columns. Therefore, the total running time is θ(n^2).

Check your Progress-

1. Which of the following operations are allowed in the string edit problem?
A. Insertion
B. Deletion
C. Substitution
D. All of the above

2. What is the goal of the string edit problem?
A. Maximizing the length of the strings
B. Minimizing the edit distance between two strings
C. Sorting the characters in a string
D. Concatenating two strings

3. Which of the following is a practical application of the string edit problem?
A. DNA sequence alignment
B. Sorting algorithms
C. Graph traversal
D. File compression

4. What is the space complexity of the dynamic programming approach for the string edit problem?
A. O(n)
B. O(n^2)
C. O(mn)
D. O(2^n)

Answer to check your progress -

1. D

2. B
3. A
4. C

1.3 Binary Search Tree –

A binary search tree is a special kind of binary tree. In binary search tree, the elements in the left and right sub-trees of each node are respectively lesser and greater than the element of that node. Fig. 1showsa binary search tree.

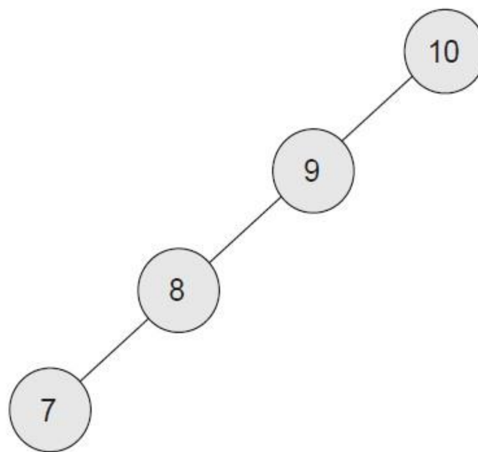


Fig.1: Skewed Binary Search Tree

Fig.1 is a binary search tree but is not balanced tree. On the other hand , this is a skewed tree where all the branches on one side.
The advantage of binary search tree is that it facilitate search of a key easily. It takes O (n) to search for a key in a list .Whereas, search tree helps to find an element in logarithmic time.

How an element is searched in binary search tree?

Let us assume the given element is x. Compare x with the root element of the binary tree, if the binary tree is non-empty. If it matches, the element is in the root and the algorithm terminates successfully by returning the address of the root node. If the binary tree is empty, it returns a NULL value .If x is less than the element in the root, the search continues in the left sub –tree.
If x is greater than the element in the root, the search continues in the right sub-tree.

This is by exploiting the binary search property. There are many applications of binary search trees. One application is construction of dictionary.

There are many ways of constructing the binary search tree. Brute force algorithm is to construct many binary search trees and finding the cost of the tree. How to find cost of the tree? The cost of the tree is obtained by multiplying the probability of the item and the level of the tree. The following example illustrates the way of find this cost of the tree.

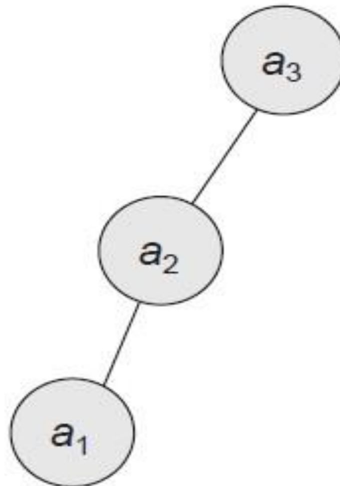


Fig.2: Sample Binary Search Tree

Example 1: Find the cost of the tree shown in Fig. 2 where the items probability is given as follows:
a1 =0.4, a2=0.3, a3=0.3

Solution

As discussed earlier, the cost of the tree is obtained by multiplying the item probability and the level of the tree .The cost of the tree is computed as follows;

$$\text{Cost of BST} = 3(0.4) + 2(0.3) + 1(0.3) = 2.1$$

It can be observed that the cost of the tree is2.1.

Check your progress-

---

1. What is a Binary Search Tree (BST)?
A. A tree structure with three child nodes
B. A tree structure with two child nodes per parent
C. A tree structure with arbitrary child nodes
D. A tree structure with no child nodes

2. In a Binary Search Tree, which property holds true for every node's left sub tree?
A. All nodes have greater values than the node itself.
B. All nodes have smaller values than the node itself.
C. All nodes have equal values to the node itself.

D. There is no specific relationship.

3. What is the purpose of a Binary Search Tree?
A. To organize data in a random manner
B. To store data in a sequential linked list
C. To facilitate fast searching, insertion, and deletion of data
D. To ensure that data is stored in descending order

4. What is the time complexity for searching in a balanced Binary Search Tree with n nodes?
A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)

5. Which operation in Binary Search Trees is used for adding a new element?
A. Insertion
B. Deletion
C. Search
D. Traversal

Answer to check your progress-

1. B
2. B
3. C
4. C
5. A

1.4 Optimal Binary search Tree

What is an optimal binary search tree? An optimal binary search tree is a tree of optimal cost. This is illustrated in the following example.

Example2: Construct optimal binary search tree for the three items a1=0.4, a2=0.3, a3=0.3?

Solution

There are many ways one can construct binary search trees .Some of the constructed binary search trees and its cost are shown in Fig.3.

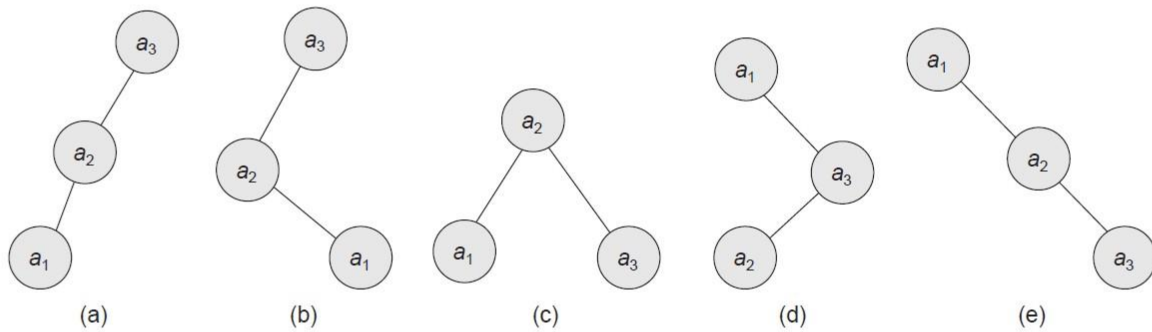


Fig.3. some of the binary search trees

It can be seen the cost of the trees are respectively, 2.1, 1.3, 1.6, 1.9and1.9.So the minimum cost is

1.3. Hence, the optimal binary search tree is (b) Fig.3.

How to construct optimal binary search tree? The problem of optimal binary search tree is given as follows: Given sequence $K = k_1<k_2<\cdots <k_n$ of n sorted keys, with a search probability p_i for each key k_i. The aim is to build a binary search tree with minimum expected cost.

One way is to use brute force method, by exploring all possible ways and finding the expected cost. But the method is not practical as the number of trees possible is Catalan sequence. The Catalan number is given as follows:

$$c(n) = \binom{2n}{n}\frac{1}{n+1} \text{ for } n > 0, \quad c(0) = 1.$$

If the nodes are 3, then the Catalan number is

$$C_3 = \frac{1}{3+1}\binom{6}{3} = 5 \text{ .}$$

Hence, five search trees are possible. In general, $\Omega\ (4^n/n^{3/2})$ different BSTs are possible with n nodes. Hence, alternative way is to explore dynamic programming approach

Dynamic programming Approach

The idea is to One of the keys in a1, …,an, say ak, where 1 ≤ k ≤ n, must be the root .Then , as per binary search rule, Left sub tree of a_k contains $a_1,\ldots,a_{k-1}$ and right sub tree of a_k contains $a_{k+1},\ldots,a_n$.

So, the idea is to examine all candidate roots a_k,for$1 \le k \le n$ and determining all optimal BSTs containing$a_1,\ldots,a_{k-1}$and containing $a_{k+1},\ldots,a_n$

The informal algorithm for constructing optimal BST is given as follows:

Step 1: Read n symbols with probability p_i.

Step 2: Create the table $C[i, j]$, $1 \le i \le j + 1 \le n$.

Step 3: Set $C[i, i] = p_i$ and $C[i-1, j] = 0$ for all $i \in [n]$.

Step 4: Recursively compute the following relation:

$$C[i, j] = C[1 \ldots k+1] + C[k+1 \ldots j] + \sum_{m=1}^{n} p_m, \quad \text{for all i and j}$$

Step 5: Return $C[1 \ldots n]$ as the maximum cost of constructing a BST.

Step 6: End.

The idea is to create a table as shown in below

Table2: Constructed Table for building Optimal BST

	0	1					j	n
1	0	p_1						*
		0	p_2					
i			↑	↑	↑	↑	C[i,j]	
							→	
							→	
							→	
							→	p_n
n+1								0

The aim of the dynamic programming approach is to fill this table for constructing optimal BST. What should be entry of this table? For example, to compute C of two items, say key 2 and key 3,two possible trees are constructed as shown belowinFig.4 and filling the table with minimum cost.

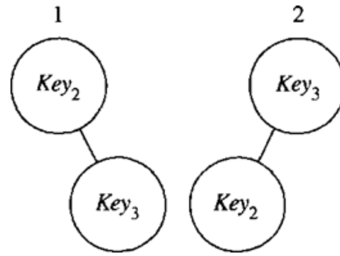


Fig.4: Two possible ways of BST for key2 andkey3.

Example3: Let there be four items *A(Danny),B(Ian), C(Radha) ,and D(zee)*with probability 2/7 1/7 3/7 1/7. Apply dynamic programming approach and construct optimal binary search trees?

Solution

The initial Table isgivenbelowinTable1,

Table1: Initial table

i \ j	0	1	2	3	4
1	0	2/7			
2		0	1/7		
3			0	3/7	
4				0	1/7
5					0

It can be observed that the table entries are initial probabilities given. Then, using the recursive formula, the remaining entries are calculated.

$$C[2,3] = \min\begin{cases} C[2,1] + C[3,3] + p_1 + p_3 & \text{when } k = 2 \\ C[2,2] + C[4,3] + p_1 + p_3 & \text{when } k = 3 \end{cases}$$

$$C[1,2] = \min\begin{cases} C[1,0] + C[2,2] + p_1 + p_2 & \text{when } k = 1 \\ C[1,1] + C[3,2] + p_1 + p_2 & \text{when } k = 2 \end{cases}$$

$$C[3,4] = \min\begin{cases} C[3,2] + C[4,4] + p_3 + p_4 & \text{when } k = 3 \\ C[3,3] + C[5,4] + p_3 + p_4 & \text{when } k = 4 \end{cases}$$

The updated entries are shown below inTable2.

Table 2: Updated table

	0	1	2	3	4
1	**0**	**2/7**	**4/7**		
2		**0**	**1/7**	**6/7**	
3			**0**	**3/7**	**5/7**
4				**0**	**1/7**
5					**0**

Similarly, the other entries are obtained as follows:

$$C[1,3] = \min \begin{cases} C[1,0] + C[2,3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 1 \\ C[1,1] + C[3,3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 2 \\ C[1,2] + C[4,3] + p_1 + p_2 + p_3, \\ \text{when } i = 1, j = 3, k = 3 \end{cases}$$

$$= \min\{{}^{12}/_{7}, {}^{11}/_{7}, {}^{10}/_{7}\} = {}^{10}/_{7}$$

$$C[2,4] = \min \begin{cases} C[2,3] + C[5,4] + p_2 + p_3 + p_4, \text{when } k = 2 \\ C[2,2] + C[4,4] + p_2 + p_3 + p_4, \text{when } k = 3 \\ C[2,3] + C[5,4] + p_2 + p_3 + p_4 \text{ when } k = 4 \end{cases}$$

$$= \min\{{}^{11}/_{7}, {}^{7}/_{7}, {}^{11}/_{7}\} = {}^{7}/_{7}$$

The updated table is given inTable3.

Table 3: Updated table

	0	1	2	3	4
1	0	2/7	4/7	10/7	
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

The procedure is continued as

$$C[1,4] = \min\begin{cases} C[1,0] + C[2,4] + P_1 + P_2 + P_3 \; + P_4, \text{ when } k = 1 \\ C[1,1] + C[3,4] + P_1 + P_2 + P_3 + P_4, \text{ when } k = 2 \\ C[1,2] + C[4,4] + P1 + P_2 + P_3 \; + P_4, \text{ when } k = 3 \\ C[1,3] + C[4,4] + P_1 + P_2 + P_3 \; + P_4\}\text{when } k = 4 \end{cases}$$

$$= \min\{{}^{14}/_{7},\ {}^{14}/_{7},\ {}^{12}/_{7},\ {}^{18}/_{7}\} = {}^{12}/_{7}$$

The updated final table is given as shown inTable4.

Table4: Final table

	0	1	2	3	4
1	0	2/7	4/7	10/7	12/7
2		0	1/7	6/7	7/7
3			0	3/7	5/7
4				0	1/7
5					0

It can be observed that minimum cost is 12/7. What about the tree structure? This can be reconstructed by noting the minimum k in another table as shown inTable5.

Table5: Minimum k

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

It can be seen from the table 5 that C(1,4) is 3. So the item 3 is root of the tree. Continuing this fashion,one can find the binary search tree as shown in Fig. 5.

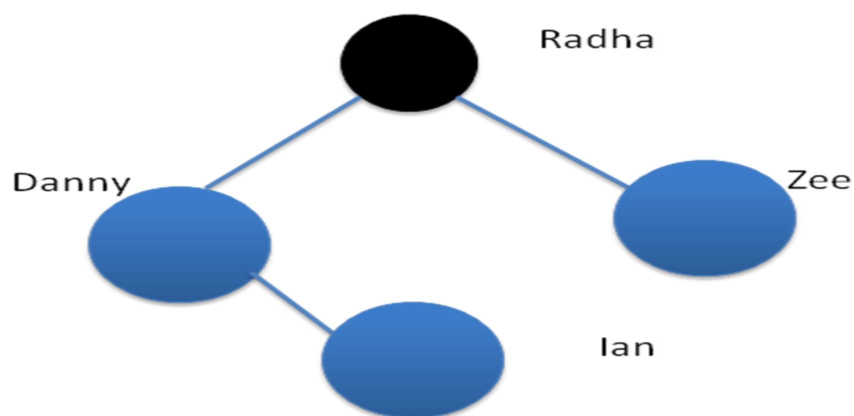


Fig.5: Constructed Optimal BST

It can be seen that the constructed binary tree is optimal and balanced. The formal algorithm for constructing optimal BST is given as follows:

```
%% Initialize the table for C
for i = 1 to n do
    C[i, i - 1] = 0
    C[i, i] = pi
End for
C[n + 1,n]= 0
```

```
%% Initialize the table for R
for i = 1 to n do
    R[i, i - 1] = 0
    R[i, i] = i
End for
C[n + 1, n] = 0
```

```
for diag = 1 to n - 1 do
    for i = 1 to n - diag do
        j = i + diag
```

$$C[i,j] = \min_{i<k\leq j} C[1 \ldots k-1] + C[k+1 \ldots j] + \sum_{m=1}^{n} p_m$$

```
        R[i,j] = k
    End for
End for
Return C[1,n]
```

Complexity Analysis

The time efficiency is Θ (n^3) but can be reduced to Θ (n^2) by taking advantage of monotonic property of the entries. The monotonic property is that the entry *R*[*i,j*] is always in the range between *R*[*i,j*-1] and R[*i*+1,j]. The space complexity is Θ (n^2) as the algorithm is reduced to filling the table.

Check your progress -

1. What is the primary goal of constructing an Optimal Binary Search Tree?

A. Minimizing the number of nodes
B. Maximizing the height of the tree
C. Minimizing the cost of searching
D. Maximizing the number of leaf nodes

2. In an Optimal Binary Search Tree, where are frequently accessed items typically placed?

A. Near the root
B. In the left sub tree
C. In the right sub tree
D. At the leaves

3. What is the time complexity of constructing an Optimal Binary Search Tree with dynamic programming?

A. O(n)
B. O(n log n)
C. O(n^2)
D. O(2^n)

4. In the context of Optimal Binary Search Trees, what does the term "cost" refer to?

A. The number of nodes in the tree
B. The height of the tree
C. The sum of search frequencies for all nodes
D. The depth of the tree

5. Which dynamic programming technique is commonly used to construct Optimal Binary Search Trees?

A. Divide and Conquer
B. Greedy Algorithm
C. Backtracking
D. Dynamic Programming

Answer to check your progress-

1. C
2. A
3. C
4. C
5. D

1.5 Knapsack Problem

Let us assume that there are '*n*' items with weights w_i and a knapsack of capacity j. The idea is to applydynamic programming approach for this problem.
Let *V*[*i*,*j*] be optimal value of such instance. Consider instance defined by first *i* items and capacity *j* (*j* ≤ W).
The dynamic programming can be applied now:

In first case, the item I cannot be added into knapsack. In that case, the capacity of the knapsack is unchanged. On the other hand , if the item I is include , $j-w_i$ >= 0 , and the optimal subset is made up of this item and an optimal subset of the first i- 1 items that fit into the knapsack capacity j – w_i. Therefore ,the optimal subset is v_i + V[i= 1 , j – w_i] .

These discussions can be consolidated and formulated as a recursive relationship as

$$V[i,j] = \begin{cases} \max\{V[i-1,j], v_i + V[i-1, j-w_i]\}, & \text{if } j - w_i \geq 0 \\ V[i-1,j], & \text{otherwise} \end{cases}$$

with the initial condition,

$$V[0,j] = 0, \text{ for } j \geq 0$$

Let us apply this to the following example-

Example 1: Let there be four items with weight and value as given below. Let the Knapsack of capacity *W* = 5

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Apply dynamic programming approach and find maximum profit?

Solution

As per the recursive relation, the table will be filled up and the table is shown below in Table 1.

	capacity j						
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
w_1=2, v_1 = 12	1	0	0	12	12	12	12
w_2=1, v_2 = 10	2	0	10	12	22	22	22
w_3=3, v_3 = 20	3	0	10	12	22	30	32
w_4=2, v_4 = 15	4	0	10	15	25	30	37

Table 1: Knapsack Problem

It can be observed that the maximum profit is 37.

The items can be traced using this following logic -

The tracing can be done like this, Check V [4,5}, it is not equal to V [3,5], so item 4 is included. This reduces the capacity of knapsack to 3. So Check V[3,3]. It is equal to V [2,3], so item 3 is not included. V[3,3] is not equal to V[1,3], so item 2 is included. Now, Knapsack is reduced to capacity of 2. Check V[1,2] is not equal to V[0,2], so item is added to knapsack.

The formal algorithm is given as below:

Algorithm DP Knapsack (*w*[*1..n*], *v*[*1..n*], *W*)

var *V*[*0..n,0..W*], *P*[*1..n,1..W*]: int

```
for j := 0 to W do
        V[0,j] := 0
for i := 0 to n do
    V[i,0] := 0
for i := 1 to n do
        for j := 1 to W do

        if  w[i] ≤ j and v[i] + V[i-1,j-w[i]] > V[i-1,j]            then


        V[i,j] := v[i] + V[i-1,j-w[i]]; P[i,j] := j-
                w[i]else

                        V[i,j] := V[i-1,j]; P[i,j] := j


return V[n,W]
```

Complexity Analysis

The running time complexity of the algorithm is items multiplied by capacity of the knapsack. If the capacity of the knapsack is j and the number of items are ‘n’, then the complexity analysis of the algorithm is O(nj). The space complexity amounts to the filling of the table. Therefore, the complexity of the algorithm is O(nj).

This algorithm is a pseudo-polynomial algorithm as it works well for smaller instances. But for larger instances, knapsack problem is a NP-Complete problem.

The above problem can be solved using top-down also . This technique is called memorization. Thisword is a corrupted word of memorization. In this technique, the table is initialized as null and filled onlyif necessary.

The formal algorithm is given as follows:

Algorithm 1.6: MFKnapsack(i,j)

begin
- if $V[i,j] < 0$ then
  - if $j < W[i]$ then $value \leftarrow MFKnapsack(i-1,j)$;
  - else
    - $value \leftarrow$
    - $\max\{MFKnapsack(i-1,j), Val[i] + MFKnapsack(i-1, j-W[i])\}$
  - end
  - $V[i,j] \leftarrow value$;
- end
- Return $V[i,j]$;

end

Complexity analysis

The running time complexity of the algorithm using memorization approach is items multiplied by capacity of the knapsack. If the capacity of the knapsack is j and the number of items are 'n', then the complexity analysis of the algorithm is O(nj). The space complexity amounts to the filling of the table. Therefore, the complexity of algorithm based on memorization approach is O(nj).

Check your progress-

1. What type of problem is the Knapsack Problem?

A. Sorting Problem
B. Search Problem
C. Combinatorial Optimization Problem
D. Graph Traversal Problem

2. What is the primary goal of the Knapsack Problem?

A. Maximizing the number of items selected
B. B. Maximizing the total weight of selected items
C. C. Maximizing the total value of selected items
D. D. Minimizing the size of the knapsack

3. What is the time complexity of the dynamic programming approach for the 0/1 Knapsack Problem with n items?

A. O(n)
B. O(n log n)
C. O(n^2)
D. O(2^n)

4. In the Knapsack Problem, what does the "knapsack constraint" refer to?

A. The total number of items available
B. The maximum weight the knapsack can hold
C. The total value of selected items
D. The value-to-weight ratio of each item

Answer to check your progress-

1. C
2. C
3. C
4. B

1.6 Flow shop Scheduling

In this problem, the task is to schedule *n* jobs. Every job has *m* tasks. There are *j* processors p_j. The time required to compute the task T_{ji} *is* t_{ji} and should be assigned to processor p_j.

Let us understand some of the important terms now:

Finish time:
The finish time *F(s)* of *s* is defined as the maximum finish time of the jobs.

$$F(s) = \max_{1 \le i \le n} \{f|(s)\ i\}$$

Mean Flow Time:
The mean flow time is defined as follows-

$$\text{mft}(s) = \frac{1}{\sum_{1 \le i \le n} f_i(s)}$$

The aim of flow scheduling is to optimize finish time. Optimal finish time (OFT) is a non-pre-emptive schedule s , for which the *F(s)* is minimum.
The mathematical formulation is given as follows:
Let there be two machines *A* and *B*. Let there be *N* jobs J_1, J_2, J_3, ……. , J_n. In flow shop problem, each job has to be processed in the order *AB*. Total elapsed time: This is the time elapsed for processing of all jobs.
Some of the assumptions are given below:
A job once started must be completed (This is called non-preemptive assignment.
1. Operation time is fixed. Similarly the processing order is fixed.
2. Processing time of every job is independent of each other and does not change.
This problem can be solved using brute force approach. This is done by trying out all possible orders and finding out the order for which F(s) is minimum. But this is not practical, as in general, if there are '*n*' jobs and '*m*' machines, the possible sequences are $(n!)^m$.

The informal Bellman-Ford algorithm based on [1,2] is given as follows
1. Find the smallest value of (A_1 , B_1).
2. If the smallest job is *Ai* , then process the job on machine *A* <u>first</u> . If the smallest job is B_i , then

process the job on machine *B* <u>last</u> .

In case of a tie, $A_i = B_i$, choose any of the jobs.

3. Repeat Step 2, till all jobs are scheduled.
4. Calculate minimum elapsed time and exit.

Example 2: Let there be 5 jobs. All these jobs have to go through the machines A and B in order A followed by B. The processing times of the jobs are given below what is the optimal order for ordering these tasks? The details are given in Table 2.

Table 2: Job details

*Job*1	1	2	3	4	5
Machine A	8	2	12	6	3
Machine B	4	10	14	9	5

Solution:

Let us apply the dynamic algorithm now.

Step 1: Consider the minimum time of machines *A* and *B*.

(Ai, B_i) = 2 for job 2. Therefore the job 2 is scheduled first as shown in Table 3.

Table 3: After First Job is scheduled

2				

Step 2: Delete the job 2. The resulting job table is shown in Table 4.

Table 4: After Deletion of Job 2

Job	1	3	4	5
A	8	12	6	3
B	4	14	9	5
Job	1	3	4	5
A	8	12	6	3
B	4	14	9	5

The minimum now is *min* $(A_1, B_1) = 4$. Therefore the job 1 is scheduled for last as shown inTable 5.

Table 5: After Second Job is scheduled

2				1

Step 3: Delete job 1. The job table is as shown in Table 6.

Table 6: Job Table Status

Job	3	4	5
A	12	6	3
B	14	9	5

The minimum now is . 3 Therefore job 5 is scheduled for machine A and the resultant table is shown in Table 7.

Table 7: After Third Job is scheduled

2	5			1

Step 4: Delete the job 5. Now the job table is as shown in Table 8.

Table 8: Job Table Status

Job	3	4
A	12	6
B	14	9

The minimum is 6 this is for machine *A*. So schedule the job for *A.* Now the task table looks like Table 9.

Table 9: After Fourth Job is scheduled

2	5	4		1

Allot the job 3 to the vacant. ∴The final table become like Table 10.

Table 10: After Final Job is scheduled

2	5	4	3	1

Let us calculate the elapsed time as shown in Table 11.

Table 11 Final Elapsed Time

Job	Machine A		Machine B			
	Time		Time		Idle time	
	In	Out	In	Out	A	B
2	0	2	2	12	-	2
5	2	5	12	17	-	-
4	5	11	17	26	-	-
3	11	23	26	40	-	-
1	23	31	40	44	-	-
					0	2

The total optimal time required to process all the jobs is 44 hours. The idle time for the machine *A* and *B* are 0 and 2 hours respectively.

Check your progress-

---

1. What is Flow Shop Scheduling?
A. Scheduling tasks in a linear sequence
B. Scheduling tasks on parallel machines
C. Scheduling tasks in a circular sequence
D. Scheduling tasks based on priority

2. In Flow Shop Scheduling, what is the characteristic of tasks processed on each machine?
A. They are processed independently
B. They are processed in parallel
C. They are processed sequentially
D. They are processed simultaneously

3. Which of the following is a key objective of Flow Shop Scheduling?
A. Minimizing the number of tasks

B. Minimizing the makespan
C. Maximizing the processing time
D. Maximizing the number of machines

4. What is the primary challenge in solving Flow Shop Scheduling problems optimally?
A. Limited number of machines
B. Limited processing time
C. NP-hard nature of the problem
D. Unlimited resources

Answer to check your progress-

1. B
2. C
3. B
4. C

1.7 Concept of Computational Complexity

There are two types of complexity theory. One is related to algorithm complexity analysis called algorithmic complexity theory and another related to problems called computational complexity theory.

Algorithmic complexity theory aims to analyze the algorithms in terms of the size of the problem. In modules 3 and 4, we had discussed about these methods. The size is the length of the input. It can be recollected from module 3 that the **size of a number** *n* is defined to be the number of binary bits needed to write *n. For example,* Example: $b(5) = b(101_2) = 3$. In other words, the complexity of the algorithm is stated in terms of the size of the algorithm.

Asymptotic Analysis refers to the study of an algorithm as the input size reaches a limit and analyzing the behavior of the algorithms. The asymptotic analysis is also science of approximation where the behavior of the algorithm is expressed in terms of notations such as big-oh, Big-Omega and Big- Theta.

Computational complexity theory is different. Computational complexity aims to determine lower bound on the efficiency of all the algorithms for a given problem and Computational complexity is measured independently of the implementation. In simple words, computational complexity is about the problem and not the algorithm.

In computational complexity theory, two things are important. One is the upper bound and another is the lower bound of the algorithm. Lower and upper bounds defines the limits of the algorithm. Upper bound indicates the worst case performance e of the algorithm and lower bound indicates the best case performance of the given algorithm.

- ### 3.7.1 Upper bound of the Algorithm

Upper bound of the algorithm is the pessimistic view of the algorithm. It can be used to indicate worst, average and best case analysis of the algorithm and often expressed as a function. It can be viewed as a proof that the given problem can be solved using at most 'n' operations, even in the worst case.

The upper bound for an algorithm is used to indicate the upper or highest growth rate. Normally the algorithm is measured with respect to best, worst and average case analysis. It can be said based on literature that "the running time grows at most this much, but it could grow more slowly".

In other words, the cost also is represented as a function. How to determine the upper bound of an algorithm? Let A be the algorithm to be analyzed and if I_n is set of all possible inputs to Algorithm A and $f_A(I)$ is the resource cost of the algorithm when given input I ranges over I_n

Then, the following costs can be defined:

Worst cost (A) = $\max_{I \in I_n} f_A(I)$

Best cost (A) = $\min_{I \in I_n} f_A(I)$

One can say the upper bound of the algorithm succinctly using Big-oh notation [4]. It can be described for run time T(n) of the given algorithm as follows: Let T(n) a non-negatively valued function, then T(n) is in set O(f(n)) if there exists two positive constants c and n_0 such that T(n)

$\leq$ cf (n) for all n > n_0. Here, $c \in \Re$ and $and\ n_0 \in N$

For example, for towers of Hanoi, the upper bound is is given as f(n) = 2^n-1. One can verify that by substituting different values for 'n' and verify it matches with the number of disks movement to move dist from source peg to destination peg. This is shown in table 1.

Table 1: Disks and Number of moves

n	T_n
1	1
2	3
3	7
4	15
5	31
6	63

From this one can guess, that the upper bound is 2^n-1.

One can also recollect from modules 3, 4 and 5, we called this as mathematical Induction and using which the upper bound is correctly established.

- ### 3.7.2 Lower Bound Theory

Lower bound is for problems and finding it is difficult compared to upper bound of the algorithm. Lower bound is the smallest number of operations necessary to solve a problem over all inputs of size n. In short, it is "At least this much work to be done".

Lower bound is an indication of how hard the algorithm is! It is done for problems and not algorithms. In other words, it is obtained the "best' algorithms that is required to solve the given problem. Let M be the model, and if Am is the set of all algorithms for problem P, then the Lower bound on the worst cost of P is given as follows:

$$\min_{A \text{ belongs } Am} \{ \max f_A(I)\}$$

Some of the examples of the lower bound are given as follows:

1. Number of comparisons needed to find the largest element in a set of *n* numbers
2. Number of comparisons needed to sort an array of size *n*
3. Number of comparisons necessary for searching in a sorted array
4. Number of multiplications needed to multiply two *n*-by-*n* matrices

Lower bound for an algorithm with run time T(n) is given formally as follows:

T(n) a non-negatively valued function, T(n) is in set Ω(ƒ(n)) if there exists two positive constants c and n_0 such that T(n) ≥ cƒ (n) for all n > n_0.

Here, $c \in \Re$ and $and\ n_0 \in N$.

Lower bounds can be of two types.

- Worst Case Lower bound
- Average case Lower bound

It must be observed that the actual cost is in between Upper and Lower bound and lower bound indicates how optimal the algorithm is! A best scenario is lower bound = upper bound or nearly equal and if not, then better search for solution continues.

Let us discuss about sorting problems where these theories can be applied.

Sorting problems

There are two models for finding bounds for sorting problems. They are

- Exchange Model
- Comparison model

Let us discuss about them now.

- **Exchange Model**

In exchange model, the input consists of 'n' items and the only operation allowed is exchange at a cost of 1 step. All other operations like comparison, examining item is considered as cost free operations. In exchange

model, n-1 exchanges are required. Therefore, the upper Bound is n- 1 exchanges are sufficient and for lower bound, n-1 Exchanges are necessary in the worst case.
In short, n-1 represents the cost required for lower and upper bound.

Comparison Model

In comparison model, the important operation is comparison operations .In this model, the input consists of 'n' items and the only operation allowed is comparison information as yes / No. Apart from comparison operation, all other operations such as exchange is cost free

Let us apply comparison model for finding lower and upper bounds.

Finding Maximum in an array

Given an array, the problem is to find maximum element. One can easily find that, the upper bound is n-1 as at most n-1 comparisons are sufficient to find maximum of n elements. Similarly, the lower bound is n-1 as n-1 comparisons are necessary in worst case to find the maximum of n elements. In short, finding the maximum element in an array is a linear algorithm of complexity O(n).

Finding the second largest element in Array

Finding the second largest element in an array is an interesting problem. This problem could be solved by sorting the elements using a sorting algorithm. This requires a time of O(n log n). Instead, a better algorithm can be found. This is done by a method called tournament method and is shown Fig. 1 for 8 elements.

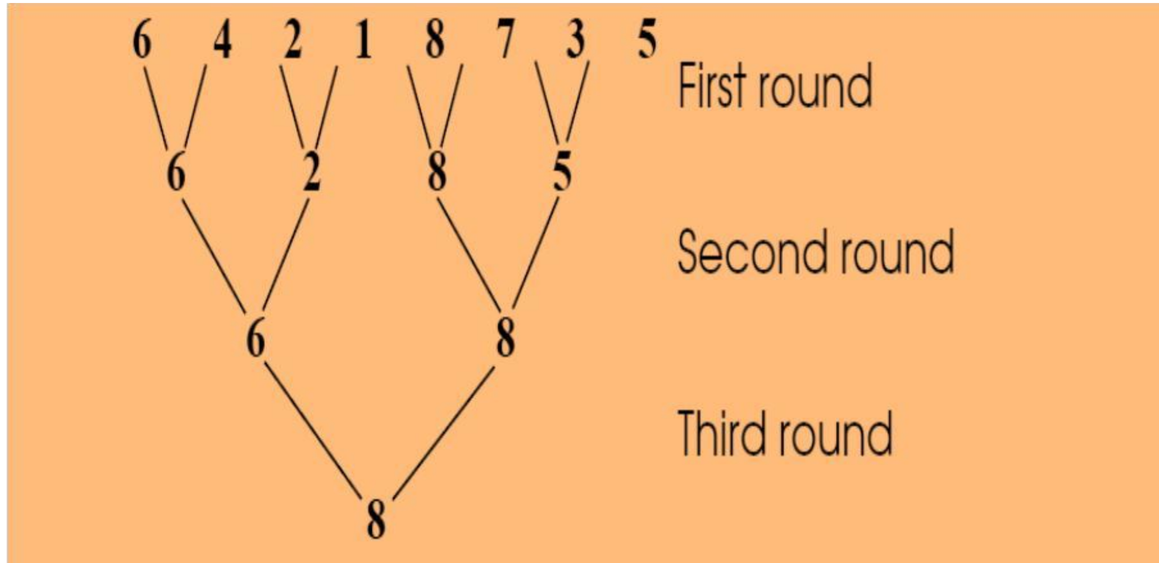


Fig.1 : Comparison model using tree

How many comparisons are required?

Finding upper bound is easy as n-1 comparisons are sufficient to find maximum of n elements. What about lower bound? It can be recollected from divide and conquer discussion for this problem, 2n-3 comparisons are necessary in worst case to find the maximum of n elements. For example, the largest element requires

n-1 comparison, the next largest of the remainder requires n-2 comparisons. In total, 2n-3 comparisons are required.

Lower Bound finding methods:

There are various methods for finding the lower bound. In this module, two techniques are discussed. They are-

1. Trivial lower bounds
2. Information-theoretic arguments (decision trees)

Let us discuss about them now

Trivial Lower Bounds

Based on counting the number of items that must be processed in input and generated as output [2,3]. This is a very primitive method and can be applied to a small set of problems. For example, consider the problem of finding maximum element in array. It requires n steps or n/2 comparisons at most. This is the trivial bound.

Decision Tree

Decision tree is an information theoretic method. It is very popular [1,3,4]. Decision Tree is a convenient model of algorithms involving comparisons in which the internal nodes represent comparisons and the leaves represent outcomes (or input cases).

Let us consider a sorting problem of six elements. This is given in Fig. 2.

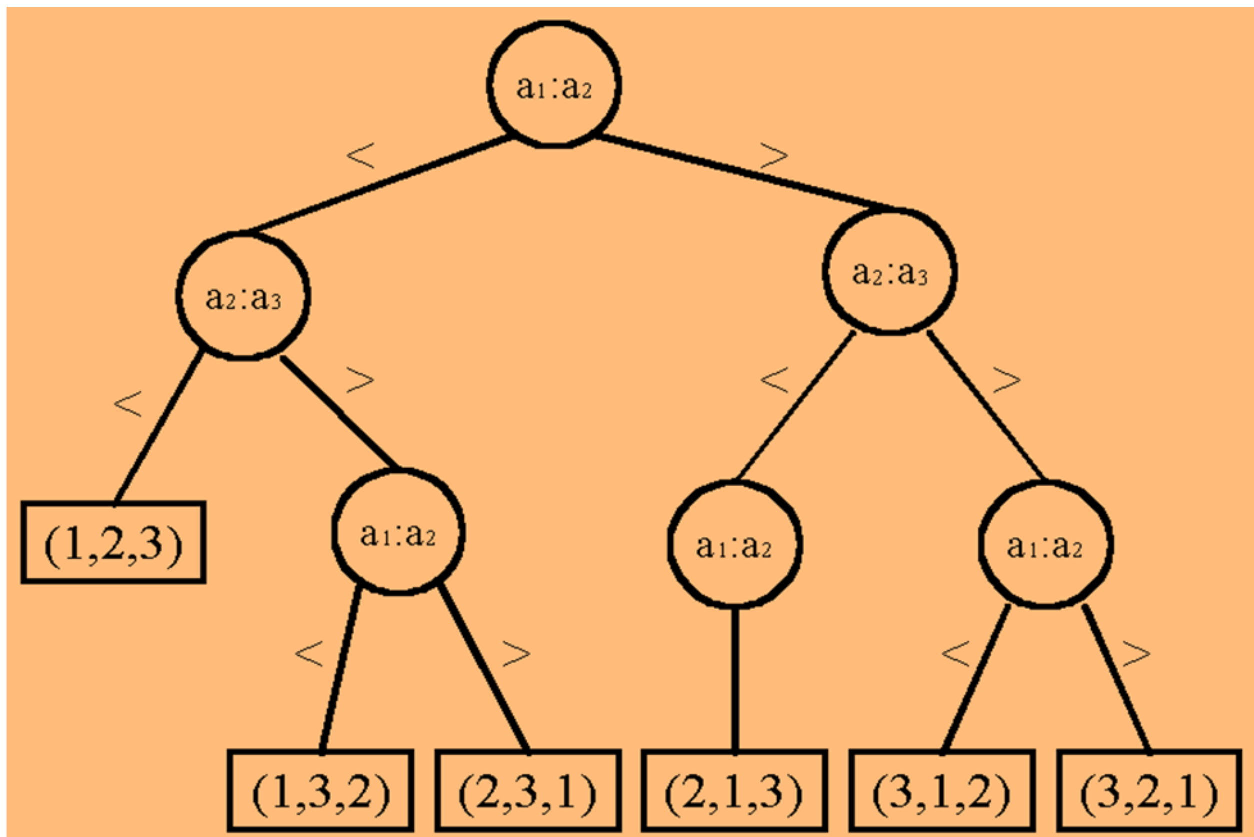


Fig. 2: Decision Tree for Six elements

In decision tree model, to find the lower bound, we have to find the smallest depth of a binary tree. Based on Fig 2 for six elements, one can easily verify that for 'n' elements, there will be n! distinct permutations and there would be n! leaf nodes in the binary decision tree.

In a balanced tree has the smallest depth:

$$\lceil \log(n)! \rceil = \Omega(n \log n)$$

This is because by Stirling approximation, n! ≈ $\sqrt{2\Pi n} \left(\frac{n}{e}\right)$ n

log n ! ≈ log n! ≈ log $\sqrt{2\Pi}$ + $\frac{1}{2}$ log n + n log $\frac{n}{e}$

≈ n log n

≈ $\Omega(n \log n)$

In other words, the lower bound of the sorting algorithm is Ω (n log n). Using decision trees, one can find lower bound for any given problem.

Model Questions

1. What is an example of edit distance?
2. What is an example of Levenshtein algorithm?
3. What are the problem of binary search tree?
4. What is knapsack problem with example?
5. What is the problem of flow shop scheduling?
6. What are the example of computational complexity?

Block 4
Unit 12: Solvability of Problems

1.0 Learning Objective
1.1 Solvability of Problems

1.1.1 Polynomial Time Algorithms
1.1.2 Unsolvable problems
1.1.3 Hard Problems
1.1.4 Turing machine
1.1.5 NP Problems

Check your progress
Answers to check your progress
1.2 Class P problem

1.2.1. What is NP Class?
1.2.2 Co – NP Problem
1.2.3 NP – I Problem
1.2.4 What is NP Hard Problem?
1.2.5 NP complete problem?

Check your progress
Answers to check your progress
1.3 Class P and NP class Problem

1.3.1 Reduction
1.3.2 Turing Reduction
1.3.3 Karp Reduction
1.3.4 NP Complete Proof Out- line

Check your progress
Answers to check your progress
Model Questions

1.0 Learning Objective

After completing this unit, the learner will be able-

- To understand the concept of Polynomial Algorithms and Non deterministic polynomial
- To understand NP- Hard Problem and P and NP class
- To understand NP-Complete problems and some NP-Complete problems
- To understand the concept of Reductions among problems and proof of NP-Complete problems
- some Important NP-Complete problems

1.1 Solvability of Problems

A problem is solvable if there exists a program that always terminate and gives the answer. Solvability of the problems is related to the tractability of the problem.
A problem is tractable if it is solvable and we can say Time(x) ≤ (some polynomial). The problems that can be solved using run time less than polynomial is called polynomial time algorithms.

1.1.1 Polynomial Time Algorithms

A problem is feasible if it has a solution cost as a polynomial. All problems that can be solved in polynomial time is called polynomial time or class P problems. Why is it called polynomial time algorithms? The reason is that Jack Edmonds and Alan Cobham proposed this terminology. It can be recollected from module 3 and module 4 that the run time of an algorithm is represented as a polynomial. The polynomials have the following characteristics.

- Polynomials are closed under composition
- Polynomials are closed under addition
- All sequential digital computers are related

So, no matter what machines are used, T (n) would remain a polynomial whose coefficients vary as per the machine. The constants ate immaterial as in asymptotic analysis, the polynomials are in any way approximated.

Hence polynomial or class P problems are a class of decision problems that are solvable in O(p(n)) time, where p(n) is a polynomial of problem's input size n

Some of the examples of Class P problems are

- Searching
- Element uniqueness
- graph connectivity

Polynomial time algorithms also give a notion of efficient algorithms. All polynomial time algorithms are efficient algorithms as the problem can be solved. How about O (n log n)? or O(n2) ? yes. These algorithms are polynomial algorithms and hence solvable. On the other hand, the algorithms having complexity like O(2n) or O(n!) are not polynomial algorithms and instead are called exponential algorithms as these functions are not polynomial and whose growth is exponential and hence cannot be solved for larger instances.

What about the algorithms like N100? Or algorithms whose degree is large, say 100 . Still, these algorithms are classified as polynomial algorithms as these kinds of algorithms are never encountered in algorithms. In

fact, most of the algorithms may never have degree more than and hence theoretically this classification of algorithms is still valid.

Solvability leads to another question. Are there any problems that can't be solved? Yes. There are many problems that cannot be solved. Let us discuss about them now.

1.1.2 Unsolvable problems

One such problem was introduced by Alan Turing 1912-1954, who is widely regarded as "Father" of modern computing science. In 1936, he introduced a problem called "halting problem". He is also credited with other accomplishments like Turing Machine, Church- Turing thesis, and Turing test.
What is a Turing problem? It can be formulated as
Can we write a program that will look at any computer program and its input and decide if the program will halt (not run infinitely)?
Turing proved that writing such algorithm is not possible. His argument is like using a Program prog and using it as parameter to itself! The general formal of the program is given as follows:

```
if halts(prog, prog):while True:
          print "looping“
    else:
        print "done"
```

The kind of argument given by Turing is that if the segment, If halts(prog,prog) returns True, that means the program will halt when given itself as input. However, in this case the program would go into an infinite loop (symbolically represented as Print "looping"). . Therefore the program doesn't halt. If halts (prog ,prog) returns False, that means that it wouldn't halt, but in that case the program does halt.

1.1.3 Hard Problems

Solvable and Unsolvable problems represent two extreme ends. In between, there are many problems that are hard.
Hard problems are problems whose solution is not guaranteed with limited computer resources such as time and space. In order to analyze these algorithms let us discuss some issues so that some framework can be developed.

1.1.4 Turing machine

Since algorithm analysis should be independent of machines, theoretical machines like Turing machines are useful. A Turing machine is a very simple theoretical "computer" with a couple basic elements such as infinitely long tape broken up into cells that can each store a single symbol from a finite set of symbols, a head that can read or write one symbol at a time from the tape and a state diagram that tells the head what to do, move left/right, print a symbol.

The advantage of Turing machine is that it can theoretically all problems if it is solvable. This is given as Turing – Church theorem given by Alanzo Church in his thesis. This thesis proves that
a Turing Machine could theoretically be created that can do anything any modern digital computer can do.
To use Turing machine, the problem should be encoded in a suitable form. So, these terminologies are important.
Alphabet: An alphabet is a finite set of symbols. For example A = {0,1}String: A finite sequence of symbols
Empty String; A string of zero length Language: A set of strings is called a Language
Complementary Language: if the strings are not in L, then it is called complementary Language.
The given problem should be posed as a decision problem. What is a decision problem? A decision problem is a problem whose output is a single Boolean value: YES or NO. Developing decision problems are useful as NP problems are a set of decision problems with the following property: If the answer is YES, then there is a proof of this fact that can be checked in polynomial time.

Once the given problem is encoded as a decision problem, the problem should be encoded. The encoded decision problem is given as input for Turing machine. The Turing machine does as follows:

If input string is invalid, then Reject

If input string is valid, but output No, then rejectIf
String is valid and output is Yes, then Accept.
With this framework, the concept of hard problems can be discussed.

1.1.5 NP Problems

NP problems stand for Non-Polynomial deterministic algorithms. A NP-Hard problem can be defined as follows:
A problem Π is NP-hard if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for every problem in NP.
Some of the examples of NP-Hard problems are given as follows:

- Traveling Salesman
- N-Queens
- Classroom Scheduling
- Packing
- Scheduling

These problems are hard problems as it is difficult to solve these problems for larger instances. Also, for most of these problems, no polynomial time algorithm is known and also it can be observed that most of these problems are combinatorial optimization problems. Most of the combinatorial problems are hard.
Related to NP problems are co-NP problems. Co-NP problems are the opposite of NP. If the answer to a problem in co-NP is NO, then there is a proof of this fact in polynomial time.

NP-I is called NP- Intermediate problems that are said to be between P and NP. Some of the examples of these problems are

- Factoring problem
- Graph isomorphism

All these point to a important issue whether P = NP? In fact, Clay Institute constituted one million dollar prize for solving this problem. This problem is ranked with other problems as shown below:

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier-Stokes Equations
4. P vs NP
5. Poincaré Conjecture
6. Riemann Hypothesis
7. Yang-Mills Theory

The heart of P Vs NP problem is that many problems do not have any polynomial algorithms. At the same time, there is no proof that these problems can't be solved in polynomial time. Also, these problems are linked with each other. So, if one problem is solved, then all related problems can be solved. In that case, the problem complexity is reduced to P.

Computational complexity is an exciting branch of algorithm analysis that discuss about these issues. The next module discusses about one important class of problems called NP-Complete problems.

Check your progress

1. What is a Polynomial Time Algorithm?

A. An algorithm with polynomial complexity
B. An algorithm that runs in constant time
C. An algorithm with exponential complexity
D. An algorithm that runs in logarithmic time

2 Which of the following time complexities represents a polynomial time algorithm?

A. O (2^n)
B. O (n!)
C. O (n^2)
D. O (log n)

.

3. A problem is NP-Hard if:

A. It has no solution
B. It is at least as hard as the hardest problems in NP
C. It can be solved in polynomial time
D. It is only hard for deterministic algorithms

4. The NP-Hard class includes problems that are:

A. Easier than problems in NP

B. At least as hard as the hardest problems in NP
C. Solvable in logarithmic time
D. Deterministic polynomial-time problems

5. In a Turing Machine, what is the role of the transition function?
A. It determines the initial state
B. It defines the tape alphabet
C. It specifies the next action based on the current state and symbol
D. It controls the movement of the tape

Answer to check your progress-

1. A
2. C
3. B
4. B
5. C

1.2 Class P Problems

P is the set of all decision problems which can be solved in polynomial time by a deterministic Turing machine. A problem is feasible if it has a solution cost as a polynomial. All problems that can be solved in polynomial time is called polynomial time or class P problems. Some of the examples of Class P problems are

- Searching
- Element uniqueness
- graph connectivity

Polynomial time algorithms also give a notion of efficient algorithms. All polynomial time algorithms are efficient algorithms as the problem can be solved. How about O(n log n)? Or O(n^2) ? Yes. These algorithms are polynomial algorithms and hence solvable. On the other hand, the algorithms having complexity like O (2^n) or O (n!) are not polynomial algorithms and instead are called exponential algorithms as these functions are not polynomial and whose growth is exponential and hence cannot be solved for larger instances.

1.2.1 What is NP Class?

The class of decision problems that can be solved by a non-deterministic polynomial algorithm iscalled class NP problem.
What is a Non-deterministic algorithm? Non-deterministic algorithms produce an answer by a sequence "Guesses" while deterministic algorithms (like those that a computer executes) make decisions based on information.
Some of the examples of NP problems are given as follows:

- Traveling Salesman
- N-Queens
- Classroom Scheduling
- Packing
- Scheduling

These problems are hard problems as it is difficult to solve these problems for larger instances. Also, for most of these problems, no polynomial time algorithm is known and also it can be observed that most of these problems are combinatorial optimization problems. Most of the combinatorial problems are hard.
What is a decision Problem?

A decision problem is a problem whose output is a single Boolean value: YES or NO and NP is the set of decision problems with the following property: If the answer is YES, then there is a proof of this fact that can be checked in polynomial time.

One can conclude that from this discussion that some problems are hard to solve with the following characteristics:

- No polynomial time algorithm is known
- If answer is YES, then it can be checked in polynomial time
- Most combinatorial optimization problems are hard

Computational complexity is an exciting branch of algorithm analysis that discuss about these issues. The next module discusses about one important class of problems called NP-Complete problems.

1.2.2 Co-NP Problems

Co-NP is the opposite of NP. If the answer to a problem in co-NP is NO, then there is a proof of this fact in polynomial time.

1.2.3 NP-I Problems

NP-I is called NP- Intermediate problems that are said to be between P and NP. Examples of NP-I problems are factoring problem and graph isomorphism problem.

1.2.4 What is NP-Hard problem?

NP-Hard are problems that are at least as hard as the hardest problems in NP. A problem A is NP-hard, if there is a polynomial algorithm exists, It implies polynomial algorithms for every problem in NP. In that case

P = NP whose proof is difficult. Clay Institute announced one million dollar prize in its web site for anyone who gives a proof. It is shown below with the other kinds of problems that are considered difficult

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier-Stokes Equations
4. P vs NP
5. Poincaré Conjecture
6. Riemann Hypothesis
7. Yang-Mills Theory

1.2.5 NP-Complete Problems

NP-Complete (or NPC) problems are a set of problems that are well connected. A problem x that is in NP, if any one finds an polynomial time algorithm even for one problem, it implies that polynomial time algorithm for all NP-Complete problems. In other words: Problem x is in NP, then every problem in NP is reducible to problem x. Let us present the overview of NP-CompleteProblems

1. Decision Problems

A decision problem is a problem whose output is a single Boolean value: YES or NO. NP is the set of decision problems with the following property: If the answer is YES, then there is a proof of this fact that can be checked in polynomial time

2. Language Framework

Let us review some of the important jargons based on [3] now. Alphabet: An alphabet is a finite set of symbols. For example A = {0,1}String: A finite sequence of symbols

Empty String: A string of zero length Language: A set of strings is called a Language

Complementary Language: if the strings are not in L, then it is called complementary Language.

3. Problem Encoding

Then the decision problem is encoded using Turing machine. Turing machine takes the encoding of the given problem and performs the following actions.

- If input string is invalid, then Reject
- If input string is valid, but output No, then reject
- If String is valid and output is Yes, then Accept.

One can summarize this for language L as follows:

P = { L | L is accepted by a deterministic Turing Machine in polynomial time }

NP = { L | L is accepted by a non-deterministic Turing Machine in polynomial time }

NP-Complete problem can be formally defined as follows:

Q is an NP-Complete problem iff

1) Q is in NP

2) every other NP problem polynomial time reducible to Q

So, to prove a problem A is NP-Complete, then reduce a known NP-C problem to A. Hence, reductions are basis of NP-Complete problems.

NP-Complete problems are solved by non-deterministic algorithms. A non-deterministic algorithm consists of two phases. The first phase is guessing of solutions and the second phase is the verification of the guesses using an algorithm or certificate. If the verification_stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP(nondeterministic polynomial) algorithm.

The verification algorithm A has two components. One is input string x and another string called certificate. Certificate is another string y. It takes the input string x, and if A(x,y) = 1 implies the language is verified by the verification algorithm

One of the famous NP-Complete problems is Circuit Satisfiability problem. It can be formalized as follows: Given a Boolean circuit, consisting of gates such as, NOT, OR, AND, Is there any set of inputs that makes the circuit output TRUE. Simultaneously, one can check for circuit output NO also.

Cook-Levin theorem states that: Circuit- SAT is NP-Complete.

The circuit Satisfiability is formulated mathematically as follows: Given a Boolean formula,determine whether this formula is satisfiable or not.

A literal: x_i or $-x_i$

A clause: $x_1 \vee x_2 \vee -x_3 \equiv C_i$

A formula: conjunctive normal form $C_1 \& C_2 \& \ldots \& C_m$

Example 1: Is there any one assignment that makes the expression true : $x_1 \vee x_2$

$\vee x_3$

$\& -x_1$

$\& -x_2$

Solution:

It can be observed that the following assignment $x1 \leftarrow F$, $x_2 \leftarrow F$, $x_3 \leftarrow T$ will make the above formula true

A related problem is called Formula Satisfiability problem. A Boolean formula is in CNF (Conjunctive Normal Form) if it is a conjunction (AND) of clauses. All clauses are disjunction (OR) of many literals. Every literal is a variable or its negation.

Another related problem is 3-SAT. In 3-SAT or 3SAT, there must be exactly 3 literals per clause. 3-SAT problem is : Given 3-CNF formula, Is there an assignment that makes the formula to evaluate to TRUE.

Examples of NP Hard Problems

Some of the known NP Hard problems are listed here. One of the major NP hard problem is vertex cover.

(i) Vertex Cover

Vertex cover is an important problem. It is formally stated as follows: Given a graph G=(V, E), S is the node cover if $S \subseteq V$ and for every edge $(u, v) \in E$, either $u \in S$ or $v \in S$ or both.

Example 1: Consider the graph shown based on [1] in Fig. 1.

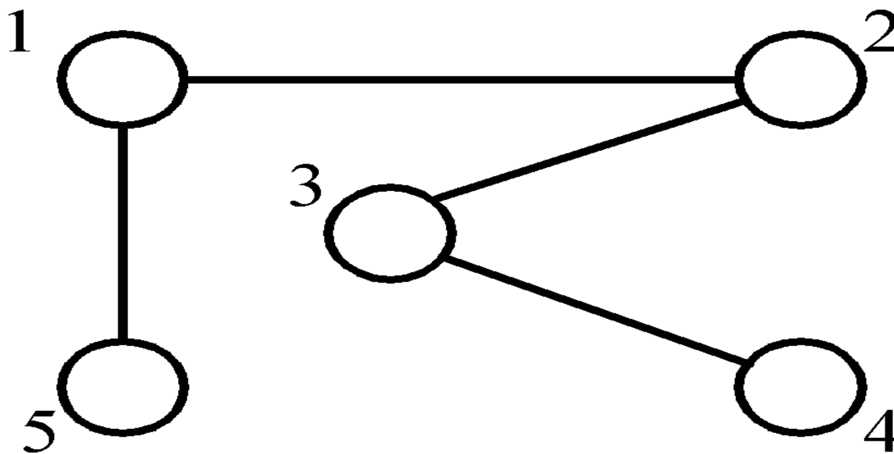


Fig 1: Sample Graph

Solution

The vertex cover finds minimal vertices that cover the entire graph. Some of the possible solutions node covers of this problem are {1, 3} and {5, 2, 4}.

Vertex cover is a NP hard problem as the solution involves guessing a subset of vertices, count them, and show that each edge is covered. This is simple if the number of vertices is smaller. But if the number of vertices becomes large, the number of possibilities becomes larger. Therefore, the algorithm becomes exponential algorithm.

(ii)Set Cover Problem

The set cover decision problem is to determine if F has a cover T containing no more than *c* sets.

Example 2: Consider the following sets :

F = {(a_1, a_3), (a_2, a_4), (a_2, a_3), (a_4), (a_1, a_3 , a_4)}

s_1 s_2 s_3 s_4

s_5Find set cover of the above problem

Solution

Like vertex cover, set cover finds the minimum number of sets that covers all the elements. Some of the possible solutions are T = {s1, s_3, s_4} and T = {s1, s_2}.

Set cover is a NP hard problem as the solution involves guessing a subset of vertices, count them, and show that universal set is well covered. This is simple if the number of elements is smaller. But if the number of elements becomes large, the number of possible set covers becomes larger. Therefore, the algorithm for set cover becomes exponential algorithm. X Therefore, set cover problem is NP hard.

(iii)Sum of Subsets

Given a set of positive numbers A = { a_1, a_2, …, a_n } and constant C, find the set of elements whose sum equals C.

Example 3: Consider the following set, A = {7, 5, 19, 1, 12, 8, 14} and C = 21, Find onesolution.

Solution

One solution is A′ = {7, 14} for C = 21. Some other solutions are {12,8,1}, {7,5,8,1}.If C = 11, then there would be no solution at all.

Sum of subsets is a NP hard problem. The solution of sum of subsets involves generating a a subset of numbers. If the sum of elements equals C, then the subset is a solution. Finding a solution is simple if the number of elements is smaller. But if the number of elements becomes large, the number of possible subsets becomes larger and algorithm becomes exponential as there are n! possible subsets for a set of 'n' elements. Therefore, the algorithm for sum of subsets becomes exponential algorithm. Therefore, sum of subsets is NP hard.

(iv) Hamiltonian Cycle

A Hamiltonian cycle is a closed path along n edges of G which visits every vertex once and returns to its starting vertex.

Example 4: Consider the graph based on [1] shown in Fig. 2.

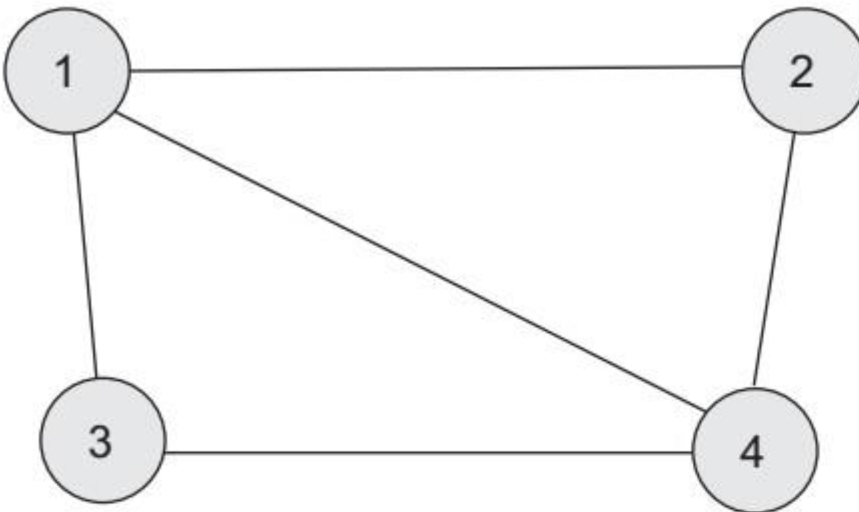


Fig. 2: Sample Graph

Solution

One possible Hamiltonian cycle is given as 4, 3, 1, 2, 4.

The concept of NP – Completeness and proofs of some of these problems are NP-Complete isdiscussed in next module.

Check your progress

1. What is the complement class of NP?
A. NP-Hard
B. P
C. Co-NP
D. EXP

2. The co-NP class is primarily concerned with:
A. Problems with non-deterministic polynomial-time solutions
B. Problems with deterministic polynomial-time solutions
C. Problems with non-polynomial time solutions
D. Problems with non-deterministic exponential time solutions

3. The concept of NP-Completeness was introduced by:
A. Stephen Cook
B. Richard Karp
C. Alan Turing
D. John von Neumann

4. Which class of problems is considered to be a subset of NP-Complete problems?
A. P
B. NP
C. NP-Hard
D. PSPACE

5. What is the complexity class of the decision problem for Hamiltonian Cycle?
A. P
B. NP
C. NP-Complete
D. NP-Hard

Answers to check your progress-

1. C
2. C
3. A
4. C

5. C

1.3 Class P and NP class of problems

P is the set of all decision problems which can be solved in polynomial time by a deterministic Turing machine. A problem is feasible if it has a solution cost as a polynomial. All problems that can be solved in polynomial time is called polynomial time or class P problems. The class of decision problems that can be solved by a non-deterministic polynomial algorithm is called class NP problem.
Class NP problems are hard problems. 'Hard' in the sense, these problems require more computer resources such as CPU time and memory to solve these problems. A lso, for most of these problems, no polynomial time algorithms exist for these problems. Also, it can be observed that most of these problems are combinatorial optimization problems and most of the combinatorial problems are hard problems. NP-Complete (or NPC) problems are a set of problems that are well connected. A problem x that is in NP, if known to have a polynomial time algorithm, it implies that polynomial time algorithm exist for all NP-Complete problems.

1.3.1 Reductions

Reduction algorithm reduces a problem to another problem. The input of a problem is called instance. Problem A is reduced to another problem B, if any instance of A "can be rephrased" as instance of B, the solution of which provides a solution to the instance of A.
There are two types of reduction. One is called Turing reduction and another is called Karp reduction.

1.3.2 Turing reduction

Let us assume two problems A and B. Problem B has a solution while problem A does not have any solution. Then reduction reduces attempts to solve problem A using procedure of solving problem B. In Turing reduction, problem A is solved using a procedure that solves B. Thus, efficient procedure for Problem A would be the efficient procedure for problem B. It can be shown mathematically denoted as follows:

$$A \leq_P B$$

This implies that problem B is at least as hard as problem A and also in other words, problem A cannot be harder than problem B.

1.3.3 Karp Reduction

Karp reduction is another important concept in NP-Complete theory. It can be mathematicallyrepresented as follows:

$$A \leq_P B$$

It illustrates that problem B is as hard as problem A and solving problem A cannot be harder thanproblem B. Karp reduction algorithm reduces a problem to another problem. It can be denoted as follows:

$$A \leq_P B$$

The input of a problem is called instance. Mathematically, if there exists a polynomial time computable function f, such that for instance w,

$$\forall w,\ w \in A \Leftrightarrow f(w) \in B$$

Then, the reduction is called Karp reduction.
So, the steps of Karp reduction can be said follows:

1. Construct f.
2. Show f is polynomial time computable.
3. Prove f is a reduction, i.e show for instance w,
    1. If $w \in A$ then $f(w) \in B$
    2. If $f(w) \in B$ then $w \in A$

Polynomial time Turing reduction is also called Cook Reduction. It must be observed that all Karp reductions are Cook Reductions but vice versa is not true and Turing Reduction and Oracle Reduction are synonymous.

Karp reduction is suitable for NP-C proof, Cook reduction is general and Karp reduction issuitable for decision problems. Cook reduction is applicable for all problems in general

Example of Reduction

Consider the example of reducing Hamiltonian path problem to Hamiltonian cycle problem. Consider the following graph based on [1] shown in Fig. 1.

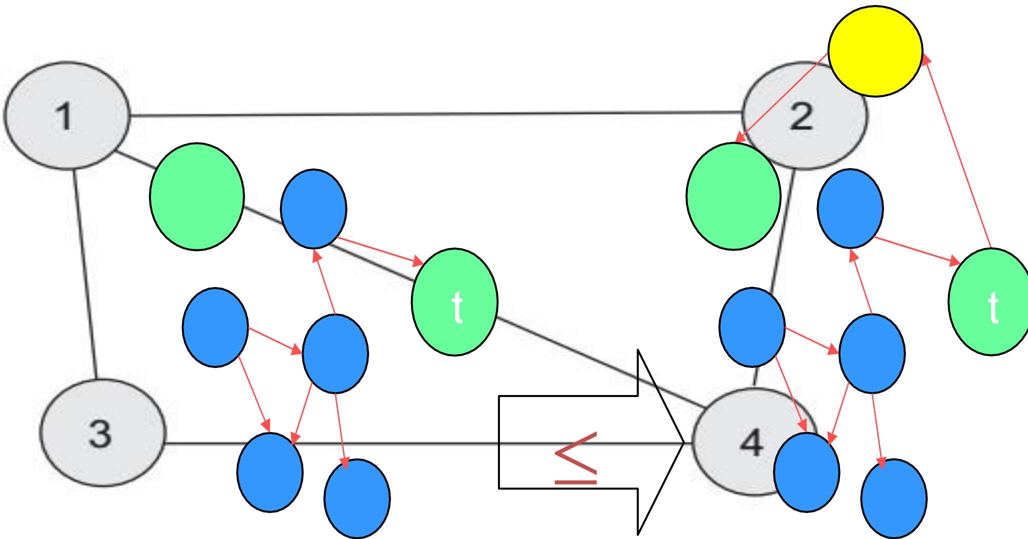


Fig. 1: Sample Graph

In Hamiltonian cycle, the aim is to find Hamiltonian cycle. In Fig 1, the Hamiltonian cycle is given as 4-3-1-2-4. The aim of reduction is to reduce a Hamiltonian cycle to Hamiltonian path and vice versa. The Hamiltonian path problem can be stated as follows:

Instance: a directed graph G=(V,E) and two vertices $s \neq t \in V$.

Hamiltonian Path Problem: To decide if there exists a path from s to t, which goesthrough each node once.

To illustrate reduction, let us first restate the problem as follows:

Fig 2: Illustration

The reduction can now be illustrated as follows:

1. If there exists a Hamiltonian path (v_0 =s,v_1 ,…,v_n =t) in the original graph, then

(u,v_0 =s,v_1 ,…,v_n =t,u) is a Hamiltonian cycle in the new graph. This is called completeness

Property.

2. (u,s) and (t,u) must be in any Hamiltonian cycle in the constructed graph, thus removingu yields a Hamiltonian path from s to t. This is called soundness property.

In other words, Hamiltonian path is converted to a cycle by adding a source vertex to form acycle and by removing it the cycle to a path.
So, the proof can be shown as follows:

1. Construct f.
2. Show f is polynomial time computable.
3. Prove f is a reduction, i.e show:
    1. If w$\in$ HAMPATH then f(w) $\in$ HAMCYCLE
    2. If f(w) $\in$ HAMCYCLE then w$\in$ HAMPATH

1.3.4 NP-Complete proof outline

Using the concept of reduction, the NP-Complete proof can be done. The proof outline is givenas follows:

1. Take one existing well known NP-Complete Problem
2. Reduce the NP-Complete problem to the given problem to be proved.
3. Argue that the given problem is as hard as the well-known NP-Complete problem.

Proof of SAT is NP- Complete

Let us consider a simple case of proving Formula Satisfiability problem as NP-Complete. Let us use the outline of NP-Complete proof outline.

1. Let us take a well-known NP-Complete problem called Circuit Satisfiability problem., Circuit Satisfiability problem is given as follows:

Given a Boolean circuit, consisting of gates such as, NOT, OR, AND, is there any set of inputs that makes the circuit output TRUE. Simultaneously, one can check for circuit output NO also.

Circuit satisfiability is a NP-Complete problem as per Cook-Levin theorem.

2. Now take the problem of Formula Satisfiability problem, SAT, for which the NP-Complete proof is required. The formula Satisfiability problem is given as follows:

   Given a Boolean formula, determine whether this formula is satisfiable or not.

   SAT problem formula consists of following components:

A literal : x_1 or x_1**A clause C** : $x_1 \vee x_2 \vee x_3$

A formula : conjunctive normal form

C_1& C_2 & … & C_m

So, the proof outline based on [1,2,3] is given as follows:

1. Circuit-SAT is NP-Complete. Given an assignment, we can just check that each clause is covered in polynomial time. It is also possible to reduce a circuit for a formula in a polynomial time. Let us consider the circuit shown in Fig. 3.

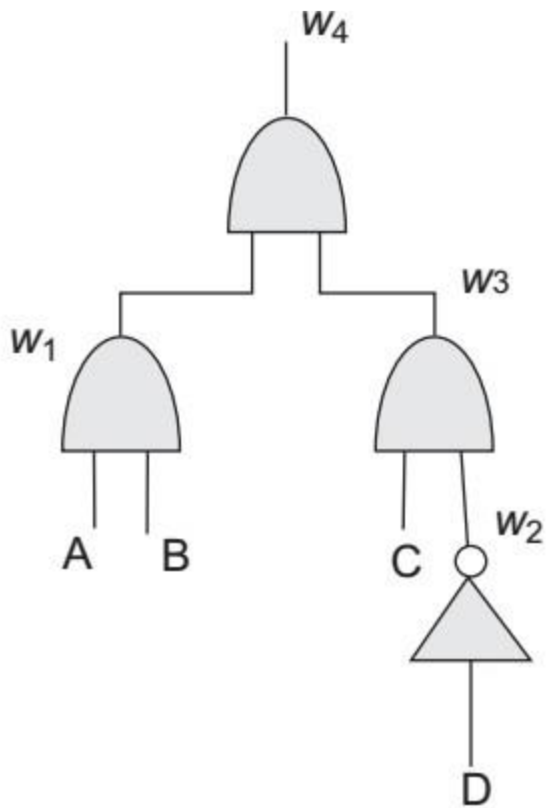


Fig. 3: Sample Circuit

An equivalent formula of the circuit can be written easily as follows:

$$\begin{aligned} & (w_1 \leftrightarrow A \wedge B) \\ \mathrm{C} = & (w_2 \leftrightarrow \neg D) \\ & (w_3 \leftrightarrow C \wedge w_2) \end{aligned}$$

$$(w_1 \leftrightarrow A \wedge B)$$
$$(w_2 \leftrightarrow \neg D)$$
$$(w_3 \leftrightarrow C \wedge w_2)$$
$$(w_4 \leftrightarrow w_1 \wedge w_3)$$

Where the conditions are given as follow -

3. In the third step, one can conclude that as any Circuit-SAT solution will satisfy the formula SAT instance and a Circuit-SAT solution can set variables giving a SAT solution, the problems are equivalent. Therefore, one can conclude that formula SAT is NP-Complete.

3-SAT is NP-Complete

One can extend the above proof for showing that 3-SAT is also NP-Complete. In 3-SAT or 3SAT, there must be exactly 3 literals per clause. The problem can be formulated as follows:
Given 3-CNF formula, Is there an assignment that makes the formula to evaluate to TRUE.
SAT is NP. Given an assignment, we can just check that each clause is covered. Based on theoutline, one can say 3-SAT is hard.
To give a proof of 3-SAT, a well-known NP-Complete problem SAT can be taken. SAT can beconverted to 3-SAT in a polynomial time as shown below:
We will transform each clause independently based on its length. Suppose a clause contains k literals:

1. if $k = 1$ (meaning $C_i = \{z_1\}$), we can add in two new variables v_1 and v_2, and transformthis into 4 clauses:

$\{v_1, v_2, z_1\}$ $\{v_1, \neg v_2, z_1\}$ $\{\neg v_1, v_2, z_1\}$ $\{\neg v_1, \neg v_2, z_1\}$

2. if $k = 2$ ($C_i = \{z_1, z_2\}$), we can add in one variable v_1 and 2 new clauses: $\{v_1, z_1, z_2\}$

   $\{\neg v_1, z_1, z_2\}$

3. if $k = 3$ ($C_i = \{z_1, z_2, z_3\}$), we move this clause as-is

4. if $k > 3$ ($C_i = \{z_1, z_2, \ldots, z_k\}$) we can add in k - 3 new variables ($v_1, \ldots, v_{k-3}$) and k - 2clauses:
   $\{z_1, z_2, v_1\}$ $\{\neg v_1, z_3, v_2\}$ $\{\neg v_2, z_4, v_3\}$ ... $\{\neg v_{k-3}, z_{k-1}, z_k\}$

To prove 3-SAT is hard, a reduction from SAT to 3-SAT must be provided. This is done usingthe above said rules.
Then in third step, the NPC of 3-SAT can be argued like this: Since any SAT solution will satisfy the *3*-SAT instance and a *3*-SAT solution can set variables giving a SAT solution, the problems are equivalent. In other words, If there were n clauses and m distinct literals in the SAT instance, this transform takes O(nm) time. Therefore, SAT = 3-SAT.

Check your progress-

1. What is the primary purpose of Turing reduction in computational theory?
A. To design efficient algorithms
B. To compare the time complexities of different algorithms
C. To define the concept of reducibility between decision problems
D. To optimize space complexity in algorithms

2. Which of the following is a common technique used to prove Turing reductions?
A. Dynamic programming
B. Divide and conquer
C. Backtracking
D. Reduction by construction

3. Which of the following statements is true regarding the relationship between Karp reductions and NP-Completeness?
A. Karp reductions are used to prove problems are NP-Hard.
B. NP-Completeness is proven through Cook reductions, not Karp reductions.
C. Karp reductions are used to prove problems are in P.
D. NP-Completeness is equivalent to Karp reductions.

4.Which complexity class is typically preserved under many-one polynomial-time reductions?
A. P
B. NP
C. NP-Complete
D. EXP

5. What is a Turing reduction also known as in the context of complexity theory?
A. Polynomial-time reduction
B. Space reduction
C. Logarithmic reduction
D. Exponential-time reduction

Answers to check your progress-

1. C
2. D
3. A

4. C
5. A

Model Questions

1. What are the conditions for a problem to be NP?
2. What are the some example of NP Problem?
3. What is the basic example of Turing machine?
4. What is P class problem with example?
5. What is the difference between NP and CO – NP?
6. What is an example of CO- NP problem?
7. What is most famous NP – complete problem?
8. What are NP – Hard problem examples?
9. What are the properties of Turing reductions?
10. What is the difference between Turing reduction and Karp reduction?

Block -4
Unit 13: NP – Complete Problem

1.0 Learning Objective
1.1 NP – Complete Problem
    1.1.1 Reduction
    1.1.2 Travelling Salesmen Problem
    1.1.3 Clique Problem
    1.1.4 Vertex Cover
Check Your Progress
Answers to Check your Progress
1.2 NP – Complete Problem
    1.2.1 Backtracking Technique
    1.2.2 Sum of sub – set
    1.2.3 Branch and bound Technique
    1.2.4 Assignment Problem
Check Your Progress
Answers to check your Progress
1.3 Randomized Algorithm
    1.3.1 Concept of witness
    1.3.2 Randomized sampling and ordering
    1.3.3 Foiling Adversary
    1.3.4 Types of Randomized Algorithms
    1.3.5 Complexity Class
    1.3.6 Random Number
    1.3.7 Hiring Problem
Check Your Progress
Answers to check your Progress
Model Questions

1.0 Learning Objective

After completing this unit, the learner will be able-

- To understand the proof of NP-Complete and TSP problem
- To understand proof of Clique, Vertex cover problem and Sum of Subsets
- To Understand Backtracking
- To Understand the overview of Branch and Bound techniques
- To understand the Randomized algorithms
- To understand Pseudorandom numbers and its generation
- To understand basic Randomized algorithms

1.1 NP-Complete Problems

NP-Complete (or NPC) problems are a set of problems that are well connected. A problem x that is in NP, if any one finds an polynomial time algorithm even for one problem, it implies that polynomial time algorithm for all NP-Complete problems. In other words: Problem x is in NP,
then every problem in NP is reducible to problem x. Let us present the overview of reductions first.

1.1.1 Reductions

Reduction algorithm reduces a problem to another problem. There are two types of reduction. One is called Turing reduction and another is called Karp reduction. Reductions form the basis of NP-Complete proofs. The proof of NP-Complete problems starts with reducing a well-known NP-Complete problem to a problem for which NP-
Complete proof is sought. This reduction should take place in polynomial time. Then theequivalence of the problems is justified. Let us discuss some basic NP-Complete proofs now:

1.1.2 Traveling Salesman Problem

Let us consider giving NP-Complete proof for travelling salesman (TSP) problem. One can consider NP-Complete proof outline.

Take a well-known NP-Complete problem and reduce that to the given problem for which proof is sought and prove that the reduction is done in polynomial time.

3. Argue that the given problem is as hard as the well-known NP-Compete problem.

To prove TSP as NP-Complete, take a well-known problem Hamiltonian Cycle Problem. It is a

Well known NP-Complete problem

The first step is to prove that Hamiltonian circuit is a NP-Complete problem. One can guess many sequences of cities as certificates. The verification algorithm is possible if atour can be guessed which is of length of k. This verification can be done in polynomial time. Therefore, Hamiltonian cycle is a NP-Complete problem.

In step 2, Hamiltonian cycle can be reduced to Traveling Salesman problem in polynomialtime. This is done by taking arbitrary G instance of Hamiltonian cycle and construct G' and

bound k such that G has Hamilton cycle iff G' has a tour of length k

Let G' = G (V,E) be the complete graph of vertices V and E' such that

E' = { (u,v) | u,v $\in$ E}

Assign length to each edge as follows:

$$\text{--------} \quad \left\{ \begin{matrix} 0 \ if \ e \ \in E \\ 1 \ if \ \notin E \end{matrix} \right\}$$

2. In step 3, It can be observed that

$$|V| = n$$

Assign k =n. the graph G has a Hamiltonian cycle iff G' has a tour of cost at most 0. If the graph G has a Hamiltonian cycle h, then all the edges belongs to E and thus has a cost of at most 0 in G'. Thus h is a tour in G' with cost of 0.

Conversely, suppose graph G' has a tour h' of cost 0. Since the cost of the tour is 0, each edge of the tour must have cost of 0. Therefore, h' contains only edges in E. Therefore, one can conclude that h' is a Hamiltonian cycle in graph G. Therefore, one can conclude that TSP is as hard as Hamiltonian cycle.

1.1.3 Clique problem

The NP-Complete proof for Clique problem can be give as per the proof outline. Let us formally,state the clique problem as follows:

Given a Graph G = (V,E) and a positive integer k, Does G contain a clique of size k?

Clique k is a complete sub graph of graph G on k vertices.

For example, for the sample graph shown in Fig. 1, the clique is shown in Fig. 2.

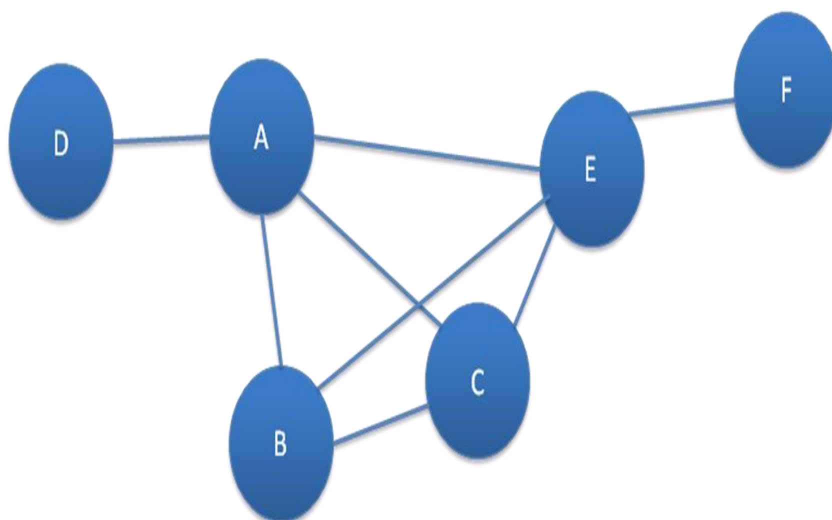


Fig. 1: Sample Graph

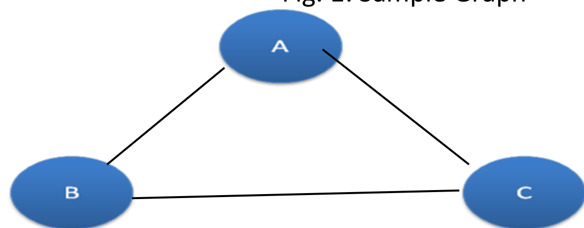


Fig 2: Sample Clique

It can be observed the sub graph of three vertices shown in Fig. 2. Is the clique for the sample graph shown in Fig. 1.

1. In step 1, a well-known NP – complete problem, SAT (Formula Satisfiability problem) is chosen. Satisfiability problem can be given as follow:

Given a Boolean formula determine whether this formula is satisfiable or not. The Boolean formula may have literal : x1 or x1, a clause C like x1 v x2 v x3 and formula be in the conjunctive normal form as C_1 & C_2 &………………& $C_{m.}$. It can be observed that there would be m – clauses.

The reduction from SAT problem to clique problem is given as follows:Construct a graph G = (V,E) where V = 2n literals and edge as

$$E = \{(x_i, x_j) \mid x_i \text{ and } x_j \text{ are in two different clauses and } x_i \neq \overline{x_j}\}.$$

For example, based on [4], a sample expression is given as follows:

$$f = (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y} \vee z).$$

A graph can be constructed based on [4] as follows as shown in Fig. 3

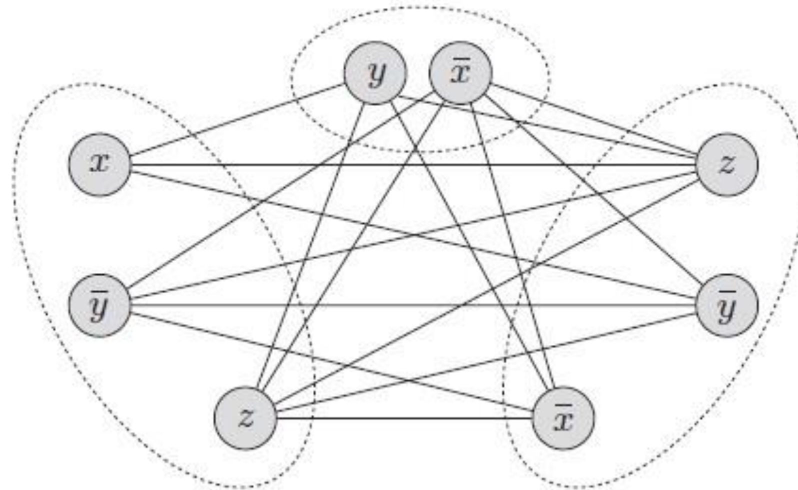


Fig. 3: Constructed Graph for the Expression

2. In step 2, one can observe that F is satisfiable iff G has a clique of size m

3. In step 3, one can argue that a clique of m corresponds to assignment of true to m literals in mdifferent clauses. It should also be observed that

- An edge is between only non-contradictory nodes. Hence, f is satisfiable iff there is non-contradictory assignment of true to m literals.
- This is possible iff G has a clique of size m.

1.1.4 Vertex Cover

Given a graph G = (V,E) and a positive integer k, Find a subset C of size k such that each edge inE is incident to at least one vertex in C

Example 1: Given the graph sown in Fig. 4, are the red vertices {1,2} form a vertex-cover?

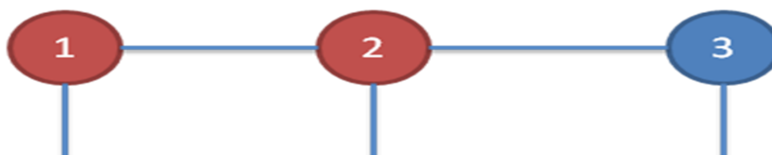


Fig. 4: Sample Graph

Solution

No. Because, Edges (4, 5), (5, 6) and (3, 6) are not covered by it
So, the idea is to cover all edges of the given graph, by picking the extra vertices 4 and 3 so that all edges are covered. The idea is thus to pick a minimum set of vertices so as to cover the graph
The proof for vertex cover can be given based on [4] as follows: First a Formula SAT is chosen and the graph is constructed using the following rules.

(1) For each Boolean variable X_i in f, G contains a pair of vertices X_i and $\overline{x_i}$ joined by an edge.
(2) For each clause C_j containing n_j literals. G contains a clique C_j of size n_j.
(3) For each vertex w in c_j, there is an edge connecting w to its corresponding literal in the vertex pairs (X_i , $\overline{x_i}$) constructed in part (1) . Call these edges connection edges.
(4) Let $K = n + \sum_{j=1}^{m}(n_j - 1)$

For, example, for the given graph, the constructed graph would be shown in Fig. 5 as follows:

$$f = (x \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee y).$$

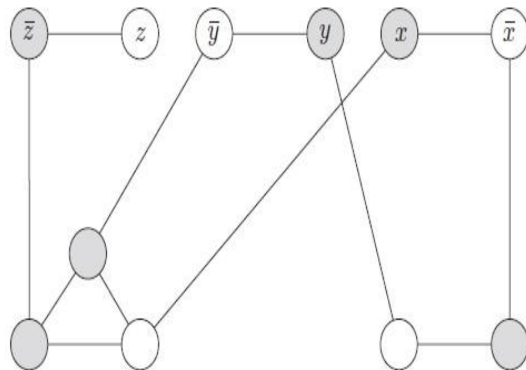


Fig 5: Constructed

Then in step 3 , the argument can be made [3] as follow :
The aim is to show that F is satisfiable iff graph has vertex cover of size k based on the proof based on [4].

Proof. $\Rightarrow$: If x_i is assigned *true*, add vertex x_i to the vertex cover; otherwise add $\overline{x_i}$ to the vertex cover. Since f is satisfiable, in each clique C_j there is a vertex u whose corresponding literal v has been assigned the value *true*, and thus the connection edge (u, v) is covered. Therefore, add the other $n_j - 1$ vertices in each clique C_j to the vertex cover. Clearly, the size of the vertex cover is $k = n + \sum_{j=1}^{m}(n_j - 1)$.

And

$\Leftarrow$: Suppose that the graph can be covered with k vertices. At least one vertex of each edge $(x_i, \overline{x_i})$ must be in the cover. We are left with $k - n = \sum_{j=1}^{m}(n_j - 1)$ vertices. It is not hard to see that any cover of a clique of size n_j must have exactly $n_j - 1$ vertices. So, in each clique, the cover must include all its vertices except the one that is incident to a connection edge that is covered by a vertex in some vertex pair $(x_i, \overline{x_i})$. To see that f is satisfiable, for each vertex x_i, if it is in the cover then let $x_i = true$; otherwise (if $\overline{x_i}$ is in the cover), let $x_i = false$. Thus, in each clique, there

Thus one can conclude that vertex cover problem is NP-Complete.

Check your progress

1. The concept of NP-completeness was introduced by:
A. Alan Turing
B. Richard Karp
C. Stephen Cook
D. Donald Knuth

2. The first known NP-complete problem is:
A. Traveling Salesman Problem (TSP)
B. Boolean Satisfiability Problem (SAT)
C. Knapsack Problem
D. Hamiltonian Cycle Problem

3. Which of the following is a common strategy for proving a problem is NP-complete?
A. Dynamic Programming
B. Divide and Conquer
C. Reduction from a known NP-complete problem

D. Greedy Algorithms

4. Which algorithmic technique is commonly used to approximate solutions for the Vertex Cover problem?
A. Greedy Algorithms
B. Divide and Conquer
C. Dynamic Programming
D. Backtracking

5. In a weighted graph, where each vertex has a positive weight, what variant of the Vertex Cover problem is considered?
A. Minimum Weighted Vertex Cover
B. Maximum Weighted Vertex Cover
C. Weighted Independent Set
D. Maximum Cardinality Matching with Weights

6. The decision version of the Clique Problem belongs to which complexity class?
A. P
B. NP
C. NP-complete
D. EXPTIME

Answers to check your progress

1. C
2. B
3. C
4. A
5. A
6. C

1.2 NP-Complete Problems

NP-Complete (or NPC) problems are a set of problems that are well connected. A problem x thatis in NP, if any one finds an polynomial time algorithm even for one problem, it implies that polynomial time algorithm for all NP-Complete problems. In other words: Problem x is in NP, then every problem in NP is reducible to problem x.
Most of the real world problems are NP-Hard . that it is difficult to solve NP-hard problem. But, most of the practical applications like weather forecasting have problems that are NP-Hard. So, it is necessary to find some ways to deal with those problems. Approximation algorithms, Randomized algorithms are some ways to tackle these problems. Backtracking and Branch-and-bound techniques are also useful techniques. Let us discuss about them now.

1.2.1 Backtracking Technique

Backtracking technique can be viewed as a systematic method of searching and it can solve many Enumeration type, decision and optimization problems

Backtracking is a depth first search with bounding functions. What is a bound function? A bounding or criterion function is a promising function and a bounding function represents the constraints of the problem.

There may be many types of constraints on the problem. Two types of constrains are internal constraints and external constrains. These constrains arise as part of the problem itself. For example, in a 4-Queen problem, two queens cannot be place in the same row, column or diagonal. This is a constraint.

Backtracking constructs a state space tree as part of the problem solving. A state space tree is a collection of problem states represented in the form of a tree data structure. A sample state space tree is shown in Fig. 1.

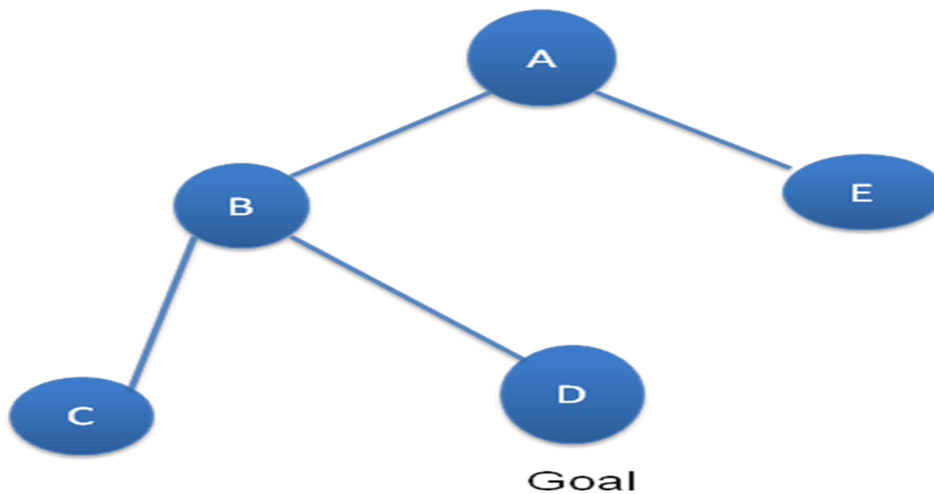


Fig. 1: A State Space Tree

Every node of the state space tree represents a state of a problem. There are different types ofnodes. A answer state is a node that is associate with the goal or target. A E-node or Extended Node is a node that is currently expanded. All nodes that are not dead and whose status active is called live node. An already explored node is called a dead node.

The approach of backtracking is as follows: It takes root and expands it based on the constraints and uses Depth First Search (DFS) to search for the goals. It goal is available in state space tree, then the search terminates else if dead end is reached, then algorithm backtracks and search process is continued on.

1.2.2 Sum of Subsets

Sum of subsets is an interesting algorithm. The problem can be stated as follows:

Given N items with weights, and a positive Integer W, sum of subsets problem is to findsubsets whose sum equals W

For example, consider a set of numbers w = {5, 10, 15, 20, 25} and W = 30,the possible solution of sum of subsets problem is {5,10,15} , {25,5} , {20,10}.

Backtracking constructs a state apace tree. A binary fixed tree can be construct for the sum of subsets problem as follows. For example, a decision can be taken whether to add an item or not. A sample state space tree is shown below in Fig. 2 where at every node a decision is taken whether an element can be added or not.

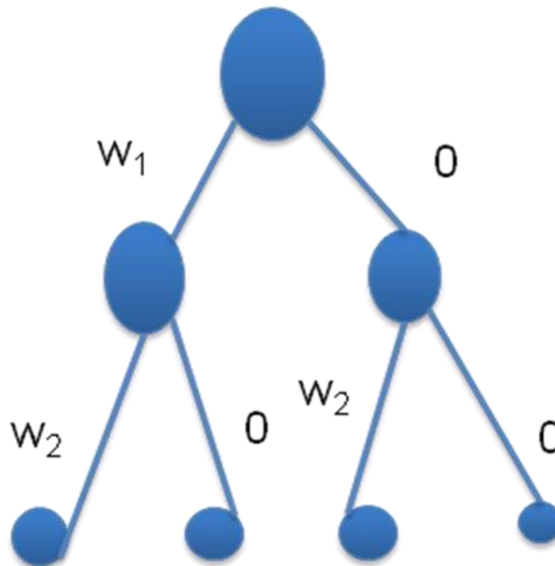


Fig 2: Fixed Binary State Space Tree

The informal algorithm can be written based on as follows:

1. Let weight be the sum of all weights of all items
2. If the weight of all the items equals W, then print solution
3. Try the following actions:
   - Add the item to the next level and update the weight
   - Exclude the item and update the weight
   - Go to step 2
4. Exit

Complexity Analysis

It can be observed that at every stage, two nodes are generated. Therefore, the number ofnodes generated is powers of 2. Therefore, time complexity is $O(2^n)$.

1.2.3 Branch and Bound Technique

Branch and Bound technique is another useful technique for an attempt to tackle NP-hard problems. In this technique, a set of feasible solution is generated and the subsets that do not have optimal solutions are deleted from further consideration.
The technique has two stages - Branching and Bounding. In the first phase, the state space tree is generated and in the second phase, bounds are used to prune the tree so that the search is focused.
The advantage of branch and bound technique is that, it is not limited to any search techniques unlike backtracking. In backtracking, DFS is used. Such restrictions are not there in branch and bound techniques.

Also branch and bound technique checks state space tree for optimal value. Branch and bound techniques are used to solve optimization problems.

Branch and bound technique can use least cost search (LC-Search). This is also called as best- First Search. The technique generates a state space tree. At every stage, using a suitable heuristics, bounds are generated. Then, using this as cost, the least cost path is selected and explored. This is repeated till the scenario where the goal is either present or not present at all.To implement this, branch-and-bound uses priority queue.

An informal algorithm for least-cost search is given as follows:

1. Initialize priority queue Q

Let v be the root of state space tree.

2. Let v be the "best" node.

- Insert v onto queue Q

Repeat the following steps till completion

- Remove node v
- if v is solution, then print
- if v is not solution and Q is empty, there is no solution
- Else, generate v and add the children to Q.

1.2.4 Assignment Problem

Assignment problem is an interesting problem in computer science. It can be stated asfollows:

Let there be N tasks and N workers and assignment problem is to assign tasks to workersoptimally based on cost matrix. Least cost solution is preferred.

The informal algorithm to solve assignment problem using branch-and-bound technique is givenas follows:

1. Begin search from start node and enqueue on to priority queue Q
2. Assign tasks to workers and compute bounds
3. Enqueue all children nodes onto Q
4. If goal is achieved, then report success else go back step 2
5. Exit

Let us apply this algorithm for this example.

Example 1: Solve assignment problem to assign three jobs 1, 2, and 3 to workers A , B and Cusing the cost table [1] given below:

Table 1: Cost Table for Assignment Problem

Workers / Tasks	1	2	3
A	18	3	15
B	4	7	14
C	13	12	7

Solution

The first step of the problem is to construct the state space tree. The bounds of this problem can be computed as follows: Random assignment of tasks 1, 2 and 3 to workers A, B and C gives a cost of 18+7+7 = 32. This is the upper bound. Whenever the bound crosses this, the tree is not generated.

The initial state space tree is shown in Fig. 3.

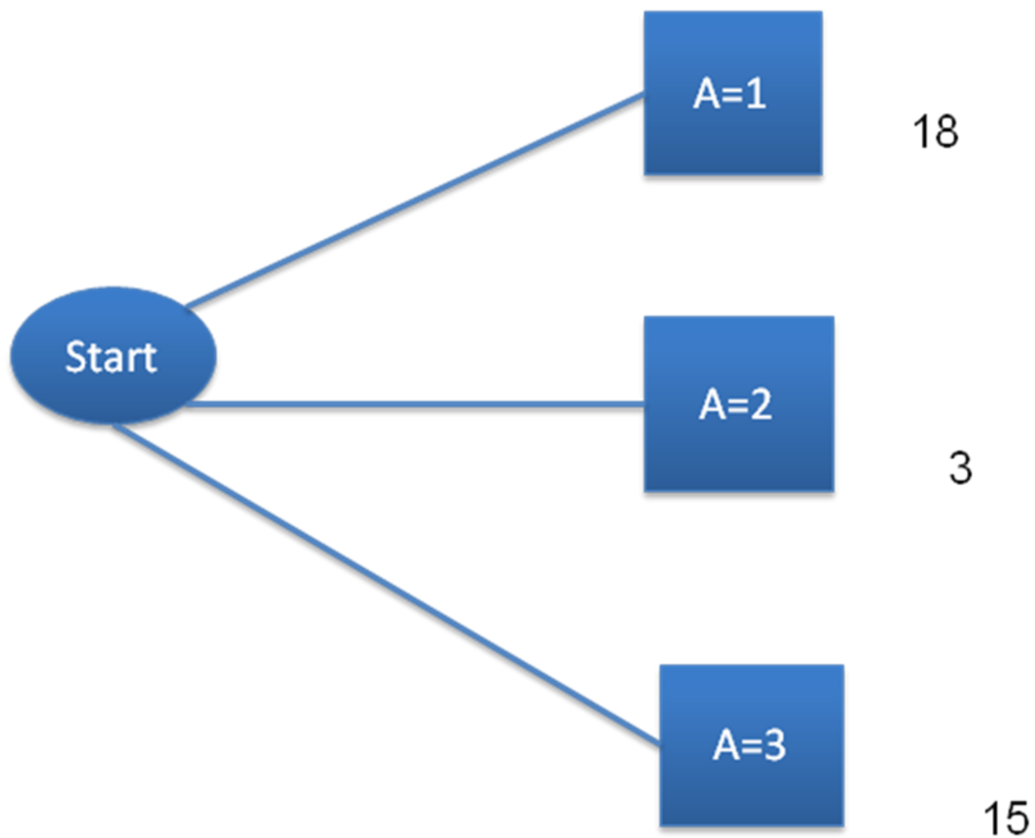


Fig 3: Initial State space Tree

The minimum cost is 3 for assigning task 2 to worker A. So, the task 2 is allotted for worker A. Exploration begins for assigning task for worker B. This is shown in Fig. 4.

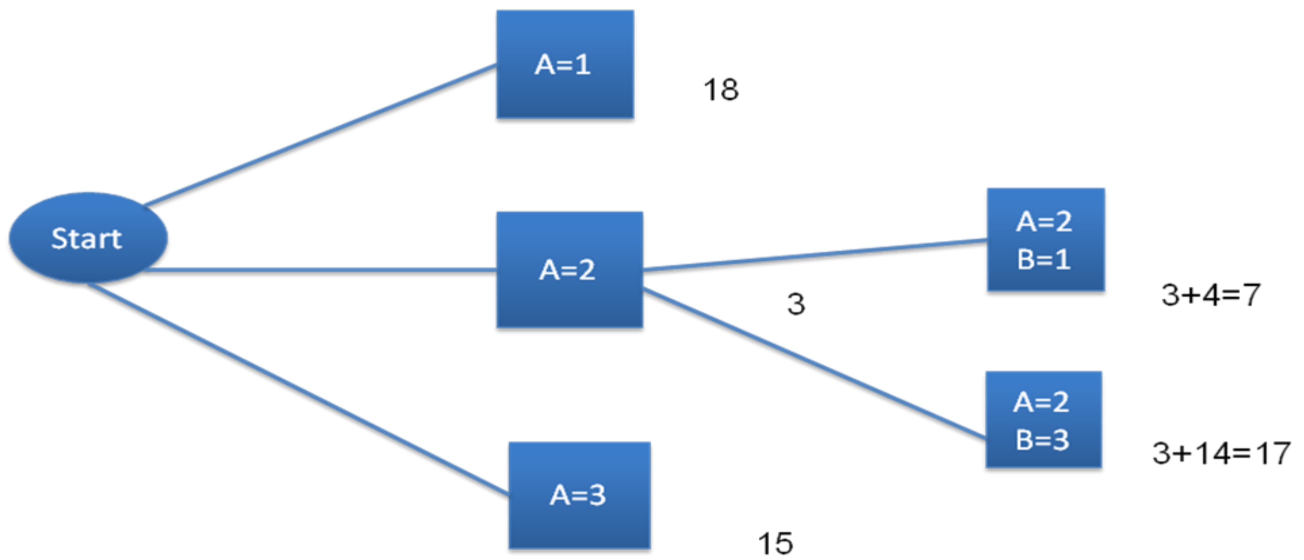


Fig. 4: Allocation of tasks 1 and 2

The cost minimum is 7. So it is explored further. The final state space tree is shown in Fig. 5.

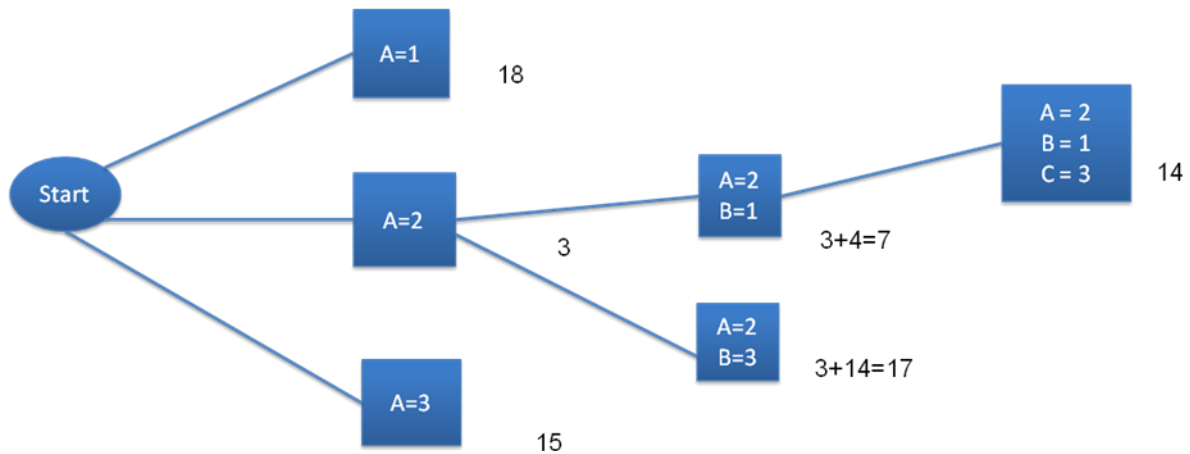


Fig. 5: Final State Space Tree

Hence, minimum cost required is 14 and Tasks 1 is allotted to worker B, Task 2 to worker A andTask 3 to worker C.

Check your progress

1. In the context of NP-complete problems, why is Backtracking often used?
A. It guarantees an optimal solution.
B. It ensures polynomial-time complexity.
C. It efficiently handles problems with exponential solution spaces.
D. It is only applicable to P problems.

2. In NP-complete problems, Backtracking may be combined with other techniques such as:
A. Divide and Conquer
B. Greedy Algorithms
C. Dynamic Programming
D. All of the above

3. What is the goal in the Subset Sum problem?
A. To find all possible subsets
B. To find the smallest subset
C. To find any subset with a given sum
D. To find the largest subset

4. Which of the following can be used to optimize the solution of the Subset Sum problem?
A. Greedy algorithms

B. Backtracking
C. Dynamic programming
D. All of the above

5. What is the Assignment Problem in optimization about?
A. Finding the optimal assignment of tasks to workers
B. Calculating the total cost of a project
C. Determining the maximum profit in a project
D. Allocating resources to tasks

Answers to check your progress

1. C
2. D
3. C
4. D
5. A

1.3 Randomized Algorithms

What is randomness? Randomness is a state of the system whose behaviour follows no deterministic or predictable pattern. Some of the daily encounters like gambling, puzzles, decision making process and heuristics are examples of randomness.
Randomness is used as a computing tool by randomized algorithms for algorithm design. Randomized algorithms are also called probabilistic algorithms.
It can be recollected from module 1 that an algorithm takes an input, process it and generates an output. This is shown in Fig. 1.

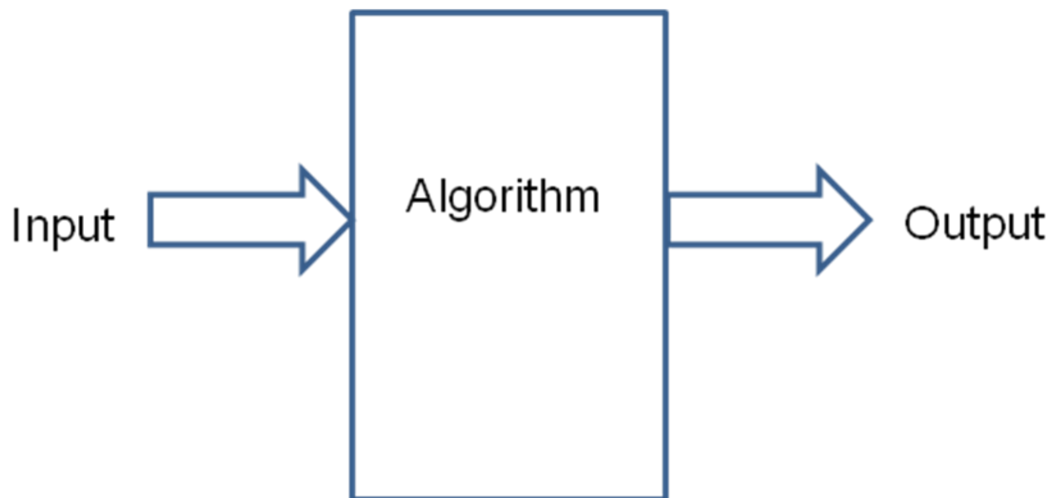


Fig. 1: Algorithm Environment

Algorithms can be classified into deterministic algorithms and randomized algorithms. Theoutput is always fixed for deterministic algorithms.
Randomized algorithms are on the other hand [1,2,3] is as shown in Fig. 2.

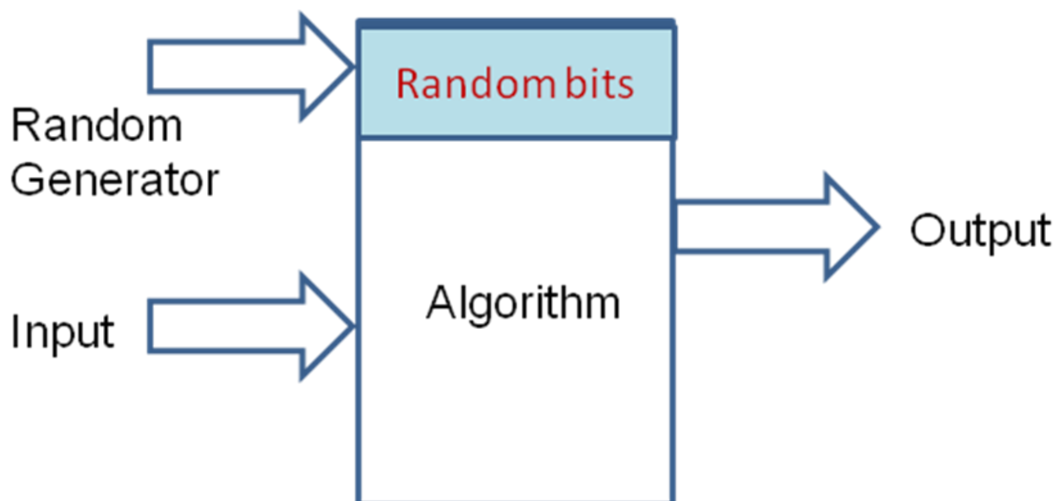


Fig. 2: A Randomized algorithm

It should be noted that randomized algorithms output is based on random decisions and its outputis based on probability. There would be negligible errors on the long run.
In short, Randomized Algorithms are dependent on inputs and use random choices as part of thelogic itself.
What are the advantages of randomized algorithms? Some of the advantages of randomizedalgorithms are given below:

- Known for its simplicity
- very Efficient
- Computational complexity is better than deterministic algorithms

Some of the disadvantages of randomized algorithms are as given below:

- Reliability is an issue
- Quality is dependent on quality of random number generator used as part of the algorithm

But randomized algorithms are very popular and are useful to solve many problems in computerscience domain effectively. Let us discuss some of the design principles that are useful for randomized algorithm design.

1.3.1 Concept of witness

This is one of the important design principles for randomized algorithms. The concept of witness is about checking whether given input X has property Y or not. The core idea is the concept of witness that gives a guarantee. Some of the problems like random trials and Primality testing canbe solved using this concept of witness.

Fingerprinting

Fingerprinting is the concept of using a shorter message representative of a larger object. This representative is called fingerprinting. If two large strings need to be checked, then instead of comparing two larger strings, two fingerprints can be compared. The problem of comparing larger strings can be done using this design principle.

1.3.2 Randomized Sampling and Ordering

Some problems can be solved by random sampling and ordering. This is done by randomizingthe input distribution or order or by partitioning or sampling randomly. Some of the problemsthat use this principle are hiring problem and randomized quicksort.

1.3.3 Foiling adversary

This is another useful principle. This can be viewed as a game between a person and an adversary with both are attempting to maximize their gains. This can be view as a selection of algorithm from a large set of algorithms.

1.3.4 Types of Randomized Algorithms

There are two types of algorithms. One is called Las Vegas Algorithms and another is called Monte Carlo Algorithms.

Las Vegas Algorithms have the following characteristics

- always correct

- “probably fast”

Randomized quicksort is an example of Las Vegas algorithm. It is faster than the traditional quicksort algorithm and its results are always correct.
Monte Carlo algorithms were designed by Nicholas Metropolis in 1949. Unlike Las Vegas algorithms, Monte Carlo algorithms give results that are mostly or probably correct. These algorithms have guaranteed running time unlike Las Vegas algorithms. Primality testing problemcan be solved using Monte Carlo algorithms.

1.3.5 Complexity Class

Like P and NP classes for deterministic algorithms, randomized algorithms also can be groupedtogether as class of problems. Some of the classes are given below:

RP Class

RP class is a set of decision problems solvable with one-sided error in polynomial time. It is an abbreviation of Random Polynomial algorithms. What is a one sided error? If the correct answer is ‘NO’, then the algorithm always returns ‘NO’ as the answer. But, if the ccorrect answer is ‘YES’, return algorithm result is associated with a probability. In other words, the algorithm output would be ‘YES’ with probability $\geq \frac{1}{2}$.
Monte Carlo algorithms belong to class RP.

ZPP Class

ZPP is a class of decision problems that are solvable in expected polynomial time. Las Vegasalgorithms are examples of ZPP class.
There is a theorem that defines the hierarchies as follows:

$$P \subseteq ZPP \subseteq RP \subseteq NP$$

1.3.6 Random Numbers

One of the primary requirements of good quality randomized algorithms is the quality of its random number generator. The quality of the random number generator determines the reliability of the randomized algorithm. Let us discuss about them now.
A random number generator generates a random number. The true random numbers are based on radioactive decay; flip of coins, shot noise, radiations. One of the characteristic of the “true” random number generator is that the generated number should not appear again. But, based on the memory and processor limitations, generation of such number is often difficult. So, Pseudo random numbers are generated. Pseudo-random numbers are as good as true random numbers in most of the situations.
Pseudo-random numbers are generated using software applications and can be recreated if formula is known. But, Pseudo-random numbers are sufficient for most of the purposes.
Some of the characteristics of “Good” random numbers are

- Efficiency

- Deterministic Property
- Uniformity
- Independence
- Long cycle

There are many algorithms are available for generating pseudorandom numbers. One simplest algorithm is called Linear Congruential Generator (LCG). The formula of LCG is given below:

$X_{i+1} = (a^{*} + b) \% m$;

Here, a and b are large prime numbers, and m is 2^{32} or 2^{64}. The initial value of x_i is called a seed.
Often a permutation array is created by generating random numbers and storing it as an array. The array of random numbers is called permutation array. The steps of creating a permutationarray are given as follows:

1. Let index = 1
2. Generate random number
3. Swap A[index] and random number
4. Fill the array

The formal algorithm based on [1,2,3] is given as follows:
Algorithm random-array(A)
Begin

for index = 1 to Nk =
random();
Exchange A[index] and A[k]End
for

End

1.3.7 Hiring problem

Hiring problem is a problem of hiring a secretary among a group of secretaries. This problemcan be solved using deterministic and randomized algorithm.
Informally, the steps of hiring problem is given as follows:

1. Initial candidate is best candidate
2. Interview new candidate
3. If new candidate is better, then hire new candidate and old candidate is fired.

What is the complexity analysis? Conducting interview and hiring costs something. If
n candidates interviewed and m candidates hired. In that case, the total cost of the algorithm would be $O(m \times C_{hire} + n \times C_{Interview})$.
The computational complexity can be improved by randomized hiring algorithm. The improvement comes because of shuffling the input. By random order of the input, the algorithm becomes randomized algorithm. The steps of the randomized hiring problem are given as follows:

1. Randomly permute array A
2. Let Initial candidate is best candidate

3. Interview new candidate
4. If new candidate is better, then hire new candidate and old candidate is fired.

Randomized analysis of this algorithm is discussed in the next module.

Check your progress

1. What is a randomized algorithm?

A. An algorithm that generates random outputs
B. An algorithm that uses random numbers in its execution
C. An algorithm that is executed in a random order
D. An algorithm that solves random problems

2. What is the primary advantage of using randomization in algorithms?

A. Deterministic outcomes
B. Simplicity in design
C. Improved efficiency
D. Increased likelihood of correctness

3. Which randomized algorithm is commonly used to find an approximate solution to the traveling salesman problem (TSP)?

A. Quick Sort
B. Prim's Algorithm
C. Monte Carlo algorithm
D. Simulated Annealing

4. What is the objective of the Hiring Problem?

A. To minimize the cost of hiring
B. To maximize the quality of the hired candidate
C. To find the most qualified candidate with the fewest interviews
D. To find the average cost of hiring

5. The Hiring Problem is often used to illustrate the concept of:

A. Greedy Algorithms
B. Divide and Conquer
C. Dynamic Programming
D. Randomized Algorithms

Answers to check your progress

1. B
2. C
3. D
4. B
5. A

Model Questions

1. Which are the three major concepts used to show that a problem is an NP-complete problem?
2. What is reduction in NP completeness?
3. How do you solve a clique problem?
4. What is vertex cover problem with example?
5. What is the complexity of the vertex cover problem?
6. What is the Travelling salesman problem?
7. What is the main objective of assignment problem?
8. What is an example of an assignment problem?
9. What are the two most common objectives for the assignment problem?
10. What is the randomness of an algorithm?
11. What is advantages and disadvantages of randomized algorithm?

Block 4
Unit 14: Randomized Algorithm

1.0 Learning Objective

1.1 Randomized Algorithm

1.1.1 Randomized Analysis
1.1.2 Primality Testing
1.1.3 Randomized large string Comparison
1.1.4 Randomized quick sort

Check Your Progress
Answers to check your progress
1.2 Approximation algorithm
1.2.1 Vertex Cover Problem
1.2.2 Travelling Salesmen Problems
Check Your Progress
Answers to check your progress
1.3 Heuristic
1.3.1 Greedy Approach
1.3.2 Linear Programming
1.3.3 Dynamic Programming
Check Your Progress
Answers to check your progress
Model Questions

1.0 Learning Objective

After completing this unit, the learner will be able-

- To understand Randomized algorithm for Primarily Testing
- To understand comparison of larger strings using Randomized algorithms
- To know about randomized quicksort
- To understand the Approximation algorithms
- To understand Need for Approximation algorithms
- To understand Basic Approximation algorithms
- To understand Heuristics, Greedy and Dynamic programming approaches for approximation algorithms

1.1 Randomized Algorithms

NP-Hard problems can be tackled by Randomized algorithms. Randomized algorithms are algorithms that uses randomness as part of its logic. What is randomness? Randomness is a state of the system whose behavior follows no deterministic or predictable pattern. Some of the daily encounters like gambling, puzzles, decision making process and heuristics are examples of randomness.

Randomness is used as a computing tool by randomized algorithms for algorithm design. Randomized algorithms are also called probabilistic algorithms.

It can be recollected from module 1 that an algorithm takes an input, process it and generates an output. This is shown in Fig. 1.

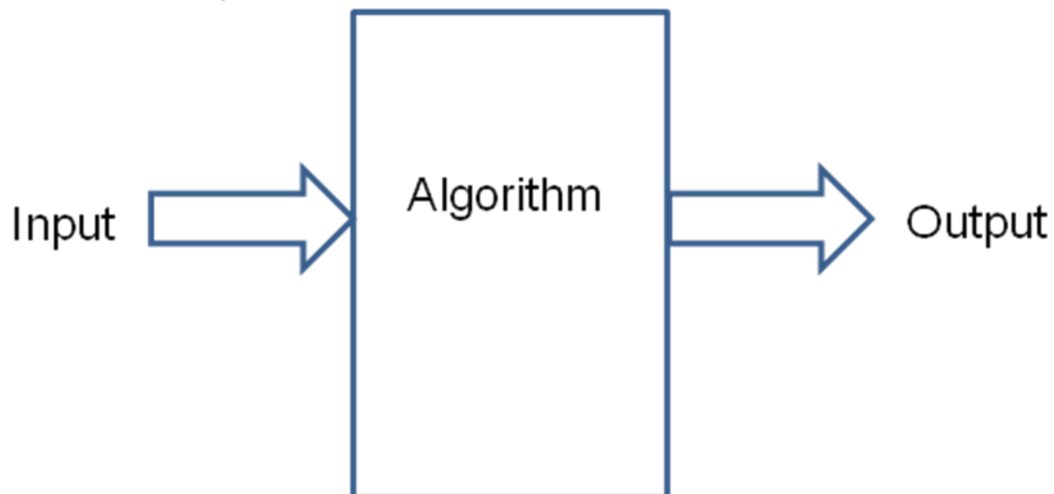


Fig. 1: Algorithm Environment

Algorithms can be classified into deterministic algorithms and randomized algorithms. The output is always fixed for deterministic algorithms.

Randomized algorithms are on the other hand is as shown in Fig. 2.

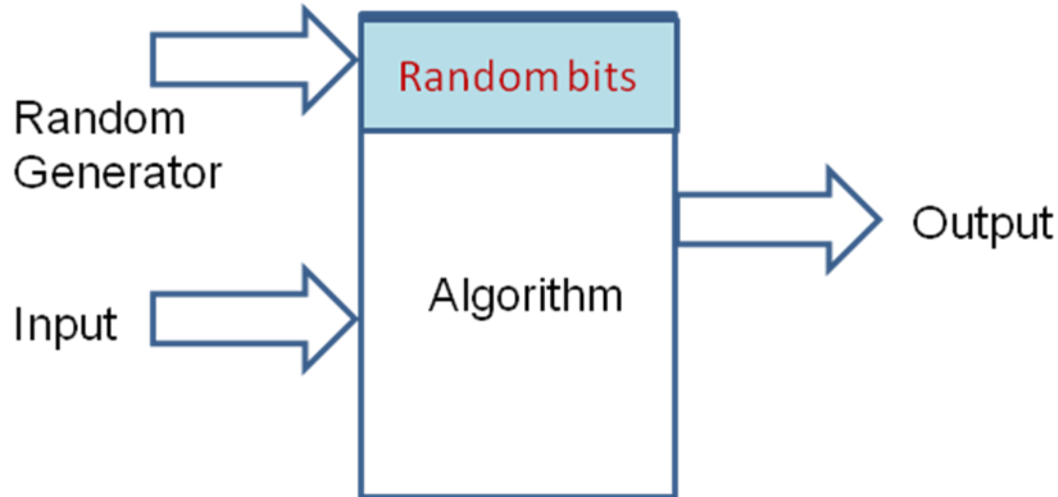


Fig. 2: A Randomized algorithm

It should be noted that randomized algorithms output is based on random decisions and its output is based on probability. There would be negligible errors on the long run.

In short, Randomized Algorithms are dependent on inputs and use random choices as part of the logic itself.

What are the advantages of randomized algorithms? Some of the advantages of randomized algorithms are given below:

- Known for its simplicity
- very Efficient
- Computational complexity is better than deterministic algorithms

Some of the disadvantages of randomized algorithms are as given below:

- Reliability is an issue
- Quality is dependent on quality of random number generator used as part of the algorithm

But randomized algorithms are very popular and are useful to solve many problems in computerscience domain effectively.

1.1.1 Randomized analysis

Hiring problem is a problem of hiring a secretary among a group of secretaries. This problem can be solved using deterministic and randomized algorithm. The informal algorithm based on randomized hiring is given as follows:

- Randomly permute array A
- Initial candidate is best candidate
- Interview new candidate
- If new candidate is better than old candidate, then new candidate is hired and old candidate is fired.

What is the complexity analysis? Conducting interview and hiring costs something. If n candidates interviewed and m candidates hired. In that case, the total cost of the algorithm would be $O(m \times C_{hire} + n \times C_{Interview})$.

Randomized analysis should be performed for randomized algorithms. It is done using a concept of indicator variables. What is an indicator random variable? Indicator random variables convert probabilities and Expectations to a number.

For an event E, the indicator variable is given as

$$I\{A\} = \begin{cases} 1 \text{ if } A \text{ occurs} \\ 0 \text{ if } A \text{ does not occurs} \end{cases}$$

Given a discrete random variable X, the Expectation of a random variable E[X] is defined by:

$$E[X] = \sum_{j=0}^{\infty} j \, P_r \, [x=J]$$

The concept of indicator random variable can be applied to randomized hiring problem.

$$X_i = \begin{cases} 1 \text{ if candidate is hired} \\ 0 \text{ if candidate is not hired} \end{cases}$$

And $x_i = x_1 + x_2 + x_3 + \dots\dots\dots\dots\dots\dots + x_n$

It can be observed that the indicator variable for a person "i" is 1 if he is hired and zero if he is not hired. So, the sum of all the indicator variables represents the Expectation. The chance of aperson getting hired is equally likely. Therefore, the expectation is given as

$$E[x] = E[\sum_{i=1}^{n} x_i]$$

$$= [\sum_{i=1}^{n} E\,[x_i]]$$

$$= \left[\sum_{i=1}^{n} \frac{1}{i}\right]$$

$$= \ln n + O\,(1)$$

It can be observed that, the complexity analysis of randomized algorithm is logarithmic compared to the linear time of conventional algorithm. This leads to a conclusion based on the randomized hiring algorithm is better compared to the convention algorithm.

1.1.2 Primality Testing

Primality testing is one of the most important randomized algorithms. Primality testing problem can be formally given as follows:

Given a number, how to check whether it is prime or not?

a number whether it is prime or not is to check divisibility from 2 to sqrt(N) . Obviously, the problem becomes hard when N is very large.

A randomized algorithm can be written for Primality testing using the principles of concept of witness. Recollect from module 37 that, concept of witness is checking whether given input X has property Y or not. Obviously, the idea is to select a witness to guarantee the prime of the given number. If the reliability is an issue, then the number of random trials can be increased.

Fermat stated that a number n is prime if and only if the congruence $x^{n-1} \equiv 1 \pmod{n}$ is satisfied for every integer x between 0 and n. based on this, a randomized algorithm can be given as follows:

- Read number n
- Pick a witness x uniformly in the range 1 to n
- Check for Fermat Criteria
- If number is composite, it is always true. If number is prime, it is probably correct

The formal algorithm based on given as follows:

Algorithm Fermat Test (n)

Begin

Choose $x \in \{1, 2, \ldots, n-1\}$ uniformly at random

If $x^{n-1} \not\equiv 1 \pmod{n}$, return composite

Else return probably prime

End if

End

Complexity Analysis

The algorithm involves k trials for picking x randomly. The algorithm involves squaring/ multiplication and modulo operations. Therefore, the algorithm has at most O(log n) steps.

1.1.3 Randomized Large String Comparison

This is another useful algorithm. The aim of this algorithm is to compare very large strings. A brute force approach is to check every bit of the message. Obviously, the conventional algorithm is tedious for larger strings.

A better randomized algorithm can be designed for this problem. The idea is to use the concept of fingerprinting. Fingerprint is a representative of larger message. So, the problem of comparing larger strings is reduced to the comparison of fingerprints of larger strings.
The informal algorithm based on for comparing two larger strings a and b is given as follows:

- Choose a prime uniformly from n to n^k

  Where k is a constant > 2

- Find fingerprint for message a and b

- Check for equality using fingerprint

The formal algorithm is given based on given as follows:

Algorithm Random-Equal (a,b)

Begin

Choose $p \in \{2, \ldots, n^k\}$ uniformly at random.
m = Fingerprint of message a
n = Fingerprint of message b
if m = n then
return true
else
return false
End

Complexity Analysis

It can be observed that the fingerprints of two larger strings a and b, m, n respectively and p requires only O(log n) bits. Therefore, the algorithm requires at most O(log n) steps. The reliability if required can be extended to k trials. Even then, the complexity of this algorithm is better than quadratic complexity of the traditional algorithm.

1.1.4 Randomized quicksort

The concept of randomized quicksort is based on randomize input distribution or order. Randomized quicksort is based on this concept.
Traditional quicksort uses a partitioning algorithm to pick pivotal element. A randomized algorithm can be written by randomly picking the pivot element. The informal algorithm can be given as follows:

- pick a pivot element randomly

- Recursively perform sort on sub arrays.

The formal algorithm for choosing the pivotal element is given as follows

Algorithm Randomized-Partition (*A, p, r*)

Begin

i ← *Random*(*p*, *r*)

Exchange *A*[r] ↔ *A*[i]

Return Partition (*A*, *p*, r)

End

The complete algorithm is given as follows:

Algorithm Randomized-Quicksort (*A, p,* r)

Begin

if *p* < *r*

then *q* ← **Randomized-Partition**(*A*, p, *r*)

Randomized-Quicksort(*A*, p , *q*-1)

Randomized-Quicksort(*A*, *q*+1, *r*)

En

d if

En

d

Complexity Analysis

The behavior of Randomized Quick Sort is determined not only by the input but also by the random choices of the pivot. The randomized analysis for randomized quicksort is done as follows:

Rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the ith smallest element (Rank "i").Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ be the set of elements between z_i and z_j, inclusive.

The indicator variable for randomized quicksort is given as follows:

Let $X_{ij} = I\{z_i \text{ is compared to } z_j\}$

Let X be the total number of comparisons performed by the algorithm. Then

$$\left[x = \sum_{i=1}^{n-1} \sum_{j=i=1}^{n} x_{ij}\right]$$

Therefore, the expected number of comparisons performed by the algorithm is

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

by linearity of expectation

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

Then, one can observe that there are only two cases as shown below:

Case 1: pivot chosen such as: $z_i < x < z_j$. In this case, z_i and z_j will never be compared

Case 2: z_i or z_j is the pivot. In that case, z_i and z_j will be compared only if one of them is chosenas pivot before any other element in range z_i to z_j

Therefore,

Pr {Z_i is compared with Z_j} = Pr{Z_i or Z_j is chosen as pivot before other elements in $Z_{i,j}$} = 2 /(j-i+1)

Finally, the Expectation of the random variable is given as

$$E[x] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} p_r \{z_i \text{ is compared to } Z_j\}$$

Therefore,

$$E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-I+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{K+1} < \sum_{i=1}^{n-1}\sum_{k=1}^{n-1} \frac{2}{K} = = \sum_{i=1}^{n-1} O(\lg n)$$

$$= O(n \lg n)$$

Therefore, the complexity analysis of randomized quicksort is same as the traditional quicksort algorithm.

Check your progress -

1. What is the main goal of primality testing algorithms?
   A. To factorize large numbers

B. To factorize large numbers
C. To find the least common multiple of two numbers
D. To solve linear equations modulo a prime number

2. Which of the following is a deterministic primality testing algorithm?
A. Miller-Rabin
B. Fermat's Little Theorem
C. Solovay-Strassen
D. AKS algorithm

3. What is the main advantage of using randomized algorithms for large string comparison?
A. Deterministic outcomes
B. Improved accuracy
C. Enhanced efficiency
D. Lower memory requirements

4. In randomized large string comparison, which technique is commonly employed to quickly identify non-matching substrings?
A. Dynamic Programming
B. Rabin-Karp algorithm
C. Knuth-Morris-Pratt algorithm
D. Boyer-Moore algorithm

5. What is the primary drawback of randomized algorithms for large string comparison?
A. Lack of accuracy
B. Deterministic outcomes
C. Higher time complexity
D. Inability to handle large strings

Answer to check your progress

1. B
2. D
3. C
4. B

5. A

1.2 Approximation Algorithms

For NP-complete problems, there is possibly no polynomial-time algorithm to find an optimal solution. But, it has been observed that most of the real world problems are hard problems. Therefore, a technique is required to tackle hard problems. Randomized algorithms are one approach for tackling NP-Hard problems.

The idea of approximation algorithms is to develop polynomial-time algorithms to get near optimal solution. In short, approximation algorithms produce near optimal solutions. This is acceptable for hard problems which are difficult to solve. So , the idea of approximation algorithm is to find approximate instead of exact solutions.

The quality of the approximation algorithm is determined by comparing the generated feasible solution with the optimal solutions. This is called approximation ratio or "Goodness factor" .

Thus, approximation ratio is a metric for quality of approximation algorithms. If the cost of approximation solution is C* and actual optimal solution is C, then the approximation ratio is given as max {C*/C, C / C*). If the problem is of minimization type, then the approximation ratio is given as C*/C and if the problem is of maximization type, then the approximation ratio is given as C/C*.

If the approximation ratio is $\rho(n)$, then the algorithm is called $\rho(n)$ -approximation ratio. For example, let us consider the problem of minimum spanning tree. Let us assume that, the cost of the approximation solution is 15 and the optimal solution is 10, then the approximation ratio is given as 15/10 = 1.5. Then, the approximation algorithm is called 1.5-approximation algorithm.

What is the range of approximation ratio? If the approximation ratio, $\rho(n)$, is one, then the approximation algorithm is same as the exact algorithm. Otherwise, this will be in the range of zero to one.

1.2.1 Vertex Cover Problem

The input for vertex cover problem is a Graph G = (V,E). The aim of vertex cover problem is to find a vertex cover of smallest number of vertices such that every edge of G is incident of at leastone vertex in C. What is a cover? A vertex is said to cover all the edges that are incident on it. If V' is the vertex cover, then

$V' \subseteq V$ and for each edge (u,v0 in E, wither u $\in$ V' or v $\in$V' or both.

The concept of vertex cover was discussed in introduction of computational complexity.

These sorts of problems can be solved using greedy approach. The approximation algorithm for solving vertex cover is given as follows:

APPROX-VERTEX-COVER(G)

```
C = ∅;
E'=G.E;
while(E' ≠ ∅ ){
    Randomly choose a edge (u,v) in E', put u and
    v into C;
    Remove all the edges that covered by u or v
    from E'
}
Return C;
```

It can be observed that the cover C is initially null. Then, an edge (u,v) is picked randomly and the edge u or v is put into cover C. Then, all the edges that are covered by u and v are removed. This process is repeated with there are no edges left uncovered. Finally, the cover C is returned as output.

1.2.2 Traveling Salesperson Problem (TSP)

Traveling Salesman or Traveling Salesperson (TSP) is another candidate for approximation algorithms. It can be recollected that, TSP can be formulated as follows:
Given a weighted, undirected graph, start from certain vertex, find a **minimum** route visit each vertex once, and return to the original vertex.
TSP is a NP-complete problem and there is no polynomial- time algorithm exist. It is possible to produce an approximation algorithm with a constant approximation ratio. How? The approximation algorithm can be designed using the principle of restriction. As per this principle, certain conditions of the problem are relaxed and then later reinforced or restricted. By this principle, approximation algorithms are designed.
An approximation algorithm based designed for this problem. It is given as follows:

APPROX-TSP-TOUR (G)

Find a MST x;
Choose a vertex as root r;
return preorderTreeWalk(x, r)

Example 1 : Apply the approximation to the following graph shown in Fig. 1 and obtain TSP tour?

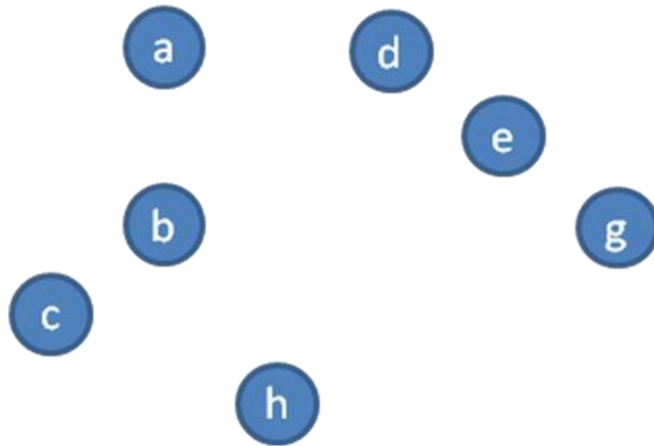


Fig. 1: A set of cities

Solution:

One can choose vertex “a” and can visit the neighbor city and return the list of cities. This isshown in Fig. 2.

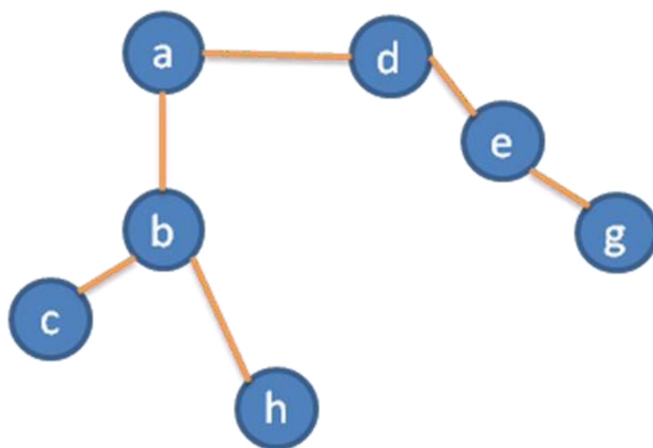


Fig. 2: Preorder walk

The preorder walk is given as

As TSP does not allowing the visiting of the same city twice, the repeating cities are removed and a new link is created to create a tour. This is shown in Fig. 3.

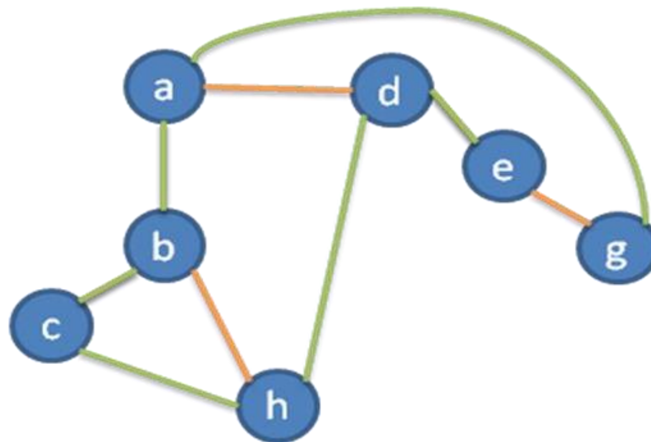


Fig. 3: A shortcut is created to avoid duplicate cities

It can be seen, a new edge is created to avoid repetition of cities. This gives a TSP tour of a,b,c,h,d,e,g and a.

Set cover problem

Set cover problem is another interesting problem. The problem can be formulated as follows:
Given a set X, and a family F of Subsets of X, the problem is to select set cover F, that covers all the elements of the subsets. In other words, the union of all elements of the set cover F, should give X. The set cover problem is to find F that covers X and it should be of minimum size.
The approximation algorithm using greedy procedure is given as follows:

```
GREEDY-SET-COVER(X, F)
    U=X;
    C=∅;
    While(U≠ ∅){
        Select S∈F that maximizes |S∩U|;
        U=U-S;
        C=C∪{S};
    }
    return C;
```

Example: The set X is shown in Fig. 4.

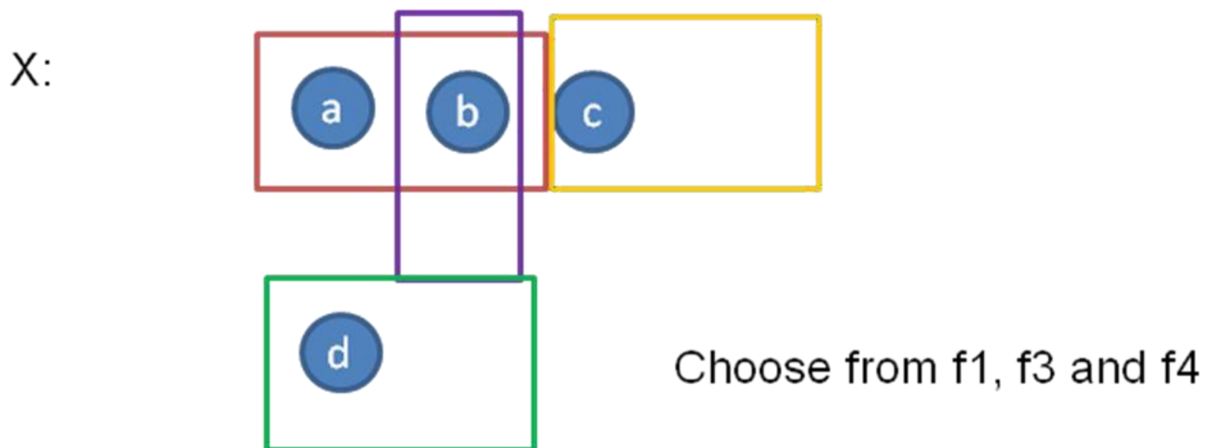


Fig. 4: Set X

The set F given as follows in Fig. 5.

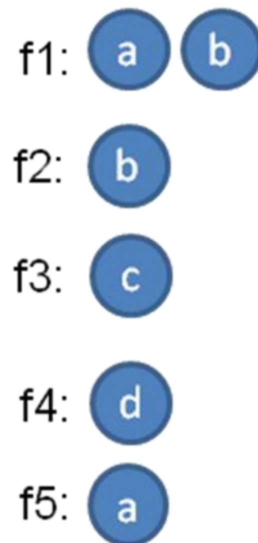


Fig. 5: Set F

What is the solution of set cover problem?

Solution:

The set cover problem is as discussed aims to find subsets of F whose union results in X. the purpose is to find minimum cover. Obviously, the elements are a,b,c and d. The sets that can bechosen to cover are given below in Fig. 6.

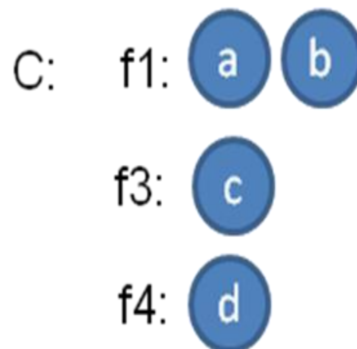


Fig. 6: Selected Sets to cover X

This covers all the elements of the set X.

Check your Progress

1. What is the primary goal of approximation algorithms?
A. To find exact solutions to optimization problems
B. To find solutions with guaranteed quality close to the optimal
C. To minimize the time complexity of algorithms
D. To maximize the accuracy of algorithms

2. Which class of problems is often targeted by approximation algorithms?
A. P
B. NP
C. NP-complete
D. EXP

3. The Traveling Salesman Problem (TSP) is an example of an optimization problem. What is a common approach for its approximation?
A. Dynamic Programming
B. Greedy Algorithms
C. Divide and Conquer
D. Backtracking

4. What is the primary disadvantage of approximation algorithms?
A. They are computationally expensive.
B. They often lead to suboptimal solutions.
C. They can only handle small instances of problems.
D. They are limited to specific problem classes.

5. The decision version of the Vertex Cover problem belongs to which complexity class?
A. P
B. NP
C. NP-complete
D. EXPTIME

Answers to check your progress

1. B
2. C
3. B
4. B
5. C

1.3 Heuristics

Heuristics are commonsense based rules that are used to design an algorithm. Heuristics do not guarantee a feasible solution. But mostly, heuristics can be used to solve hard problems. Some of the heuristics based algorithms may take possible exponential time also. But heuristics based algorithms can be tested experimentally.

Let us discuss about some heuristics based algorithm for solving Traveling salesperson (TSP) problem.

- **Nearest Neighbor heuristic Algorithm**

The heuristic is called nearest neighbor heuristics. The approach of nearest neighbor heuristics isgiven as follows:

- Choose an arbitrary node as starting vertex
- Visit all the nodes using nearest neighbor rule and return to starting vertex
- Return the tour and exit.

- **Multi-fragment Heuristic Method**

This is another heuristic method. The approach of multi- fragment heuristic method based on given as follows:

- Start the edges in ascending order based on the weights of edges
- Chose the next edge and add if feasible
- Repeat steps 1-2 till a tour of given list of V vertices are obtained.

- **Christofides Heuristic Algorithm**

Christofides heuristic algorithm is another heuristics based algorithm .The algorithm based on informally given as follows:

1. Initially a MST is constructed using Prim algorithm
2. Add edges of minimum matching to all odd-degree vertices of MST.
3. Form multi-graph by adding extra edges and find Eulerian circuit.
4. Find equivalent Hamiltonian circuit and output it as a TSP tour.

1.3.1 Greedy approach

Greedy approach is useful for designing approximation algorithms . The approach of greedy approach is given as follows:

1. Find a set S which is most cost-effective.
2. Add S to the solution set if feasible.
3. Repeat until all the elements are considered.

Some of the problems that can use greedy procedure is given as follows.

1. Knapsack Problem

One can recollect, the problem of knapsack problem is

Give "n" objects, each with a weight $w_i > 0$, with profit $p_i > 0$ and capacity of knapsack: B, find the way to make objects so that profit is maximum.

The problem can be formulated as follows:

$$\text{Maximize} \quad \sum p_i x_i$$

$$\text{Subject to} \quad \sum w_i x_i \le B$$

$$0 \le x_i \le 1, \; 1 \le i \le n$$

1. Bin Packing

Bin packing is another problem for which approximation algorithm can be designed using greedy procedure. The bin packing problem is given as follows:

Given a set of items $S = \{x1 \dots x_n\}$ each with some weight wi, pack maximum number of items into a collection of finite number of bins each with some capacity Bi using minimum number of bins.

2. **Sum of Subsets**

Sum of subsets is another problem for which approximation algorithm can be designed using greedy approach. The sum of subsets problem is given as follows:

Given a set of items $S = \{x1 \dots x_n\}$ each with some weight wi, pack maximum number of items into a collection of finite number of bins each with some capacity Bi using minimum number of bins.

1.3.2 Linear Programming

Linear programming is another approach used for designing approximation algorithms. What is linear programming? A Linear programming is the problem of optimizing a linear function subject to linear inequality constraints. The function being optimized is called the *objective function*.
The function with the constraints is called the Linear Program. Any assignment of variables that satisfies the constraints is called a feasible solution.
A sample linear programming problem is given as follows:

Minimize $7x_1 + x_2 + 5x_3$
Constraints:

$$x_1 - x_2 + 3x_3 \geq 10$$
$$5x_1 + 2x_2 - x_3 \geq 6$$
$$x_1 \geq 0$$
$$x_2 \geq 0$$
$$x_3 \geq 0$$

All the constraints involved in the above problem involve inequalities. All constraints are of type greater or equal in minimization LP, and less or equal in maximization LP. It should be noted in linear programming that all variables are constrained to be non negative.
By a simple transformation any linear program can be written as a standard minimization or maximization Linear Programming.
A variation of linear programming is integer programming. Integer Programming is simply Linear Programming - All variables must be integers. Many problems can be stated as Integer Programs.
Some of the problems have the approximation algorithms using this approach are vertex cover and set cover. These problems are discussed in earlier modules. The approach of linear programming for designing approximation algorithms based on given below:

1. Reduce an NP-hard problem to an integer programming problem.
2. Relax the integer programming problem to linear programming.
3. Find optimal solutions using linear programming.
4. Round-off optimal fractional solution deterministically or approximately to get approximate solutions.

1.3.3 Dynamic programming

Dynamic programming is another way to design approximation algorithms. The steps for designing approximation algorithms using dynamic programming approach is given as follows:

1. Find Pseudo-polynomial algorithm for solving given problem
2. Trim or scale down input by rounding it off
3. Use dynamic programming to compute approximate solutions for the modified instances.

PTAS (Polynomial Time Approximation Time) is one scheme where a family of algorithms can be designed within a certain range . The formal definition of PTAS is given as follows:

PTAS (Polynomial Time Approximation Scheme): A (1 + ε)-approximation algorithm for a NP- hard optimization *Π* where its running time is bounded by a polynomial in the size of instance.
FPTAS (Fully PTAS): The same as above + time is bounded by a polynomial in both the size ofinstance *I* and 1/ε
Dynamic programming based approximation algorithm can be designed for knapsack problem. The knapsack problem, as discussed earlier, can be formulated for dynamic programming as follows:
Given a set S = {a_1, …, a_n} of objects, with specified sizes and profits, size(a_i) and profit(a_i),and a knapsack capacity B, find a subset of objects whose total size is bounded by B.
The aim is to maximize the total profit. If A(i,p) denote the minimize size to achieve profit p using objects from 1 to i, then If we do not choose object i+1: then A(i+1,p) = A(i,p). If we choose object i+1: then A(i+1,p) = size(a_{i+1}) + A(i,p-profit(a_{i+1})) if p > profit(a_{i+1}). In other words, A(i+1,p) is the minimum of these two values.
So, for the dynamic programming algorithm, the aim is to design a table where there are n rows and at most nP columns. Each entry can be computed in constant time (look up two entries). So the total time complexity is O(nP).
PTAS based approximation algorithm can be designed for knapsack program. The informal approach is given as follows:
The idea is to scale down the values, so that a polynomial time algorithm can be designed for it. Based on this principle, an approximation algorithm based on given as follows:

1. Given ϵ > 0, let K = ϵP/n, where P is the largest profit of an object.
2. For each object a_i, define profit*(a_i) = floor(profit(a_i)/K)
3. With these as profits of objects, using the dynamic programming algorithm, find the most profitable set S'.
4. Output S' as the approximate solution.

So the approach of designing an approximation algorithm using dynamic programming approachis as follows:

1. Modify the instance by rounding the numbers.
2. Use dynamic programming to compute an optimal solution S in the modified instance.
3. Output S as the approximate solution.

Complexity Analysis

For the dynamic programming approach approximation algorithm, the complexity analysis can be done as follows: There are n rows and at most “n” floor(P/K) columns. Each entry can be computed in constant time (look up two entries). So the total time complexity is $O(n^2$ floor(P/K)) = $O(n^3/ \epsilon)$.

Check Your Progress

1. What is a heuristic algorithm primarily designed for?
A. Finding optimal solutions to problems
B. Guaranteeing the best possible outcomes
C. Quickly finding feasible solutions in a reasonable amount of time
D. Solving problems deterministically

2. What is the main trade-off associated with heuristic algorithms?
A. Accuracy vs. Precision
B. Time complexity vs. Space complexity
C. Completeness vs. Optimality
D. Exploration vs. Exploitation

3. In heuristic algorithms, what is the role of a "fitness function"?
A. To measure the complexity of the algorithm
B. To evaluate the quality of a solution
C. To determine the probability of convergence
D. To generate random solutions

4. What type of problem does Christofides' Algorithm aim to solve?
A. Knapsack Problem
B. Traveling Salesman Problem
C. Shortest Path Problem
D. Maximum Flow Problem

5. What is the time complexity of Christofides' Algorithm?
A. O(n log n)
B. O(n^2)

C. O(n^2 log n)
D. O(n^3)

Answers to check your progress

1. C
2. C
3. B
4. B
5. C

Model Questions

1. What is a randomized quick sort?
2. What is the difference between quick sort and random quick sort?
3. What is the complexity of randomized quick sort?
4. What is an approximation algorithm for NP?
5. What is an approximation algorithm?
6. What is the vertex cover problem?
7. What is an example of a heuristic search algorithm?
8. What is an example of a linear program?