Course Code	**:**	**MCS-211**
Course Title	**:**	**Design and Analysis of Algorithms**
Assignment Number	**:**	**MCAOL(I)/211/Assign/2025**
Maximum Marks	**:**	**100**
Weightage	**:**	**30%**
Last Dates for Submission	**:**	**30th April 2025 (for January Session)**
		31st October 2025 (for July Session)

This assignment has four questions (80 Marks). Answer all questions. The remaining 20 marks are for viva voce. You may use illustrations and diagrams to enhance the explanations. Please go through the guidelines regarding assignments given in the Programme guide for the presentation format.

Q1: a) Design and develop an efficient algorithm to find the list of prime numbers in the range 501 to 2000. What is the complexity of this algorithm? **(2 Marks)**

b) Differentiate between Cubic-time and Factorial-time algorithms. Give example of one algorithm each for these two running times. **(2 Marks)**

c) Write an algorithm to multiply two square matrices of order n*n. Also explain the time complexity of this algorithm. **(2 Marks)**

d) What are asymptotic bounds for analysis of efficiency of algorithms? Why are asymptotic bounds used? What are their shortcomings? Explain the Big O and Big Θ notation with the help of a diagram. Find the Big O-notation and Θ-notation for the function: **(4 Marks)**

$$f(n)= 100n^4 +1000n^3 +100000$$

e) Write and explain the Left to Right binary exponentiation algorithm. Demonstrate the use of this algorithm to compute the value of 3^{29} (Show the steps of computation). Explain the worst-case complexity of this algorithm. **(4 Marks)**

f) Write and explain the Bubble sort algorithm. Discuss its best and worst-case time complexity. **(3 Marks)**

g) What are the uses of recurrence relations? Solve the following recurrence relations using the Master's method **(3 Marks)**

a. $T(n) = 4T\left(\frac{n}{4}\right) + n^1$

b. $T(n) = 4T\left(\frac{3n}{4}\right) + n^1$

Q2: a) What is an Optimisation Problem? Explain with the help of an example. When would you use a Greedy Approach to solve optimisation problem? Formulate the Task Scheduling Problem as an optimisation problem and write a greedy algorithm to solve this problem. Also, solve the following fractional Knapsack problem using greedy approach. Show all the steps. **(4 Marks)**

Suppose there is a knapsack of capacity 20 Kg and the following 6 items are to packed in it. The weight and profit of the items are as under:

$(p_1, p_2, \ldots, p_6) = (30,16,18,20,10, 7)$

$(w_1, w_2, \ldots, w_6) = (5, 4, 6, 4, 5, 7)$

Select a subset of the items that maximises the profit while keeping the total weight below or equal to the given capacity.

b) Assuming that data to be transmitted consists of only characters 'a' to 'g', design the Huffman code for the following frequencies of character data. Show all the steps of building a huffman tree. Also, show how a coded sequence using Huffman code can be decoded. **(4 Marks)**

a:5, b:25, c:10, d:15, e:8, f:7, g:30

c) Explain the Merge procedure of the Merge Sort algorithm. Demonstrate the use of recursive Merge sort algorithm for sorting the following data of size 8: [19, 18, 16, 12, 11, 10, 9, 8]. Compute the complexity of Merge Sort algorithm. **(4 Marks)**

d) Explain the divide and conquer approach of multiplying two large integers. Compute the time complexity of this approach. Also, explain the binary search algorithm and find its time complexity. **(4 Marks)**

e) Explain the Topological sorting with the help of an example. Also, explain the algorithm of finding strongly connected components in a directed Graph. **(4 Marks)**

Q3: Consider the following Graph:

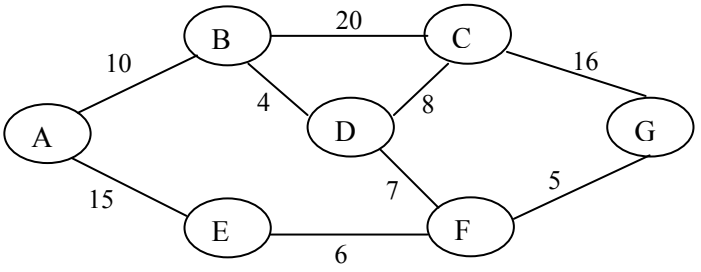

Figure 1: A sample weighted Graph

a) Write the Prim's algorithm to find the minimum cost spanning tree of a graph. Also, find the time complexity of Prim's algorithm. Demonstrate the use of Kruskal's algorithm and Prim's algorithm to find the minimum cost spanning tree for the Graph given in Figure 1. Show all the steps. **(4 Marks)**

b) Write the Dijkstra's shortest path algorithm. Also, find the time complexity of this shortest path algorithm. Find the shortest paths from the vertex 'A' using Dijkstra's shortest path algorithm for the graph given in Figure 1. Show all the steps of computation. **(4 Marks)**

c) Explain the algorithm to find the optimal Binary Search Tree. Demonstrate this algorithm to find the Optimal Binary Search Tree for the following probability data (where p_i represents the probability that the search will be for the key node k_i, whereas q_i represents that the search is for dummy node d_i. Make suitable assumptions, if any) **(6 Marks)**

i	**0**	**1**	**2**	**3**	**4**
p_i		**0.10**	**0.15**	**0.20**	**0.10**
q_i	**0.05**	**0.10**	**0.10**	**0.10**	**0.10**

d) Given the following sequence of chain multiplication of the matrices. Find the optimal way of multiplying these matrices: **(2 Marks)**

Matrix	Dimension
A1	10 × 15
A2	15 × 5
A3	5 × 20
A4	20 × 10

e) Explain the Rabin Karp algorithm for string matching with the help of an example. Find the time complexity of this algorithm. **(4 Marks)**

Q4: a) Explain the term Decision problem with the help of an example. Define the following problems and identify if they are decision problem or optimisation problem? Give reasons in support of your answer. **(4 Marks)**

(i) Travelling Salesman Problem
(ii) Graph Colouring Problem
(iii) 0-1 Knapsack Problem

b) What are P and NP class of Problems? Explain each class with the help of at least two examples. **(4 Marks)**

c) Define the NP-Hard and NP-Complete problem. How are they different from each other. Explain the use of polynomial time reduction with the help of an example. **(4 Marks)**

d) Define the following Problems: **(8 Marks)**

(i) SAT Problem
(ii) Clique problem
(iii) Hamiltonian Cycle Problem
(iv) Subset Sum Problem

MCS-211

SOLVED ASSIGNMENT 2024-25

Q.1 -

(a)- Design and develop an efficient algorithm to find the list of prime numbers in the range 501 to 2000. What is the complexity of this algorithm?

ANS.- A highly efficient algorithm to find prime numbers in the range **501 to 2000** is the **Sieve of Eratosthenes**. Here's how it works:

1. Create a boolean array is_prime of size **2001**, initializing all values as true.
2. Mark is_prime[0] and is_prime[1] as false (0 and 1 are not prime).
3. Iterate from **2 to sqrt(2000)**, marking multiples of each prime as false.
4. Collect numbers in the range **501 to 2000** where is_prime[i] is true.

Complexity:

- **Time Complexity: O(n log log n)** (efficient for large ranges).
- **Space Complexity: O(n)** due to the boolean array.

This method ensures fast computation compared to checking divisibility for each number separately.

(b)- Differentiate between Cubic-time and Factorial-time algorithms. Give example of one algorithm each for these two running times.

ANS.- Cubic-time algorithms have a time complexity of **$O(n^3)$**, meaning their execution time grows polynomially with input size. They are feasible for moderate input sizes. An example is the **Floyd-Warshall algorithm** for finding all-pairs shortest paths in a graph.

Factorial-time algorithms have a time complexity of **O(n!)**, meaning execution time grows exponentially, making them impractical for large inputs. An example is the **Brute-force Traveling Salesman Problem (TSP) solver**, which checks all possible permutations of cities to find the shortest route.

(c)- Write an algorithm to multiply two square matrices of order n*n. Also explain the time complexity of this algorithm.

ANS.-

Algorithm to Multiply Two $n \times n$ Matrices:

1. Initialize a result matrix C of size $n \times n$ with all elements as 0.
2. For each row i in matrix A:
   - For each column j in matrix B:
     - Compute $C[i][j] = \sum_{k=0}^{n-1} A[i][k] \times B[k][j]$.
3. Return matrix C.

Time Complexity:

The algorithm has three nested loops, each running $O(n)$ times, leading to a total time complexity of $O(n^3)$.

(d)-

What are asymptotic bounds for analysis of efficiency of algorithms? Why are asymptotic bounds used? What are their shortcomings? Explain the Big O and Big Θ notation with the help of a diagram. Find the Big O-notation and Θ-notation for the function:

$$f(n)= 100n^4 +1000n^3 +100000$$

ANS.- **Asymptotic Bounds and Their Significance**

Asymptotic bounds are mathematical tools used to analyze the efficiency of algorithms, focusing on how their resource usage (time or memory) grows as the input size increases. They provide a high-level understanding of an algorithm's performance without getting bogged down in implementation details or specific hardware.

Why Use Asymptotic Bounds?

- **Platform and Implementation Agnostic:** Asymptotic analysis abstracts away hardware-specific factors, making it easier to compare algorithms across different machines.

- **Focus on Growth:** It highlights the algorithm's behavior for large inputs, which is often more critical than its performance on small datasets.
- **Simplified Analysis:** Asymptotic bounds simplify complex algorithms by focusing on the dominant terms in their resource usage.

Shortcomings of Asymptotic Bounds:

- **Hidden Constants:**

Asymptotic notation ignores constant factors, which can sometimes be significant in practice.

- **Worst-Case vs. Average-Case:** Asymptotic bounds often represent the worst-case scenario, which might not reflect the typical behavior of the algorithm.
- **Oversimplification:** Asymptotic analysis can oversimplify real-world performance by neglecting factors like cache behavior or hardware optimizations.

Big O (O) and Big Theta (Θ) Notation

- **Big O (O):** Represents the upper bound of an algorithm's growth rate. It provides an asymptotic guarantee that the resource usage will not exceed a certain function.
- **Big Theta (Θ):** Represents both the upper and lower bound of an algorithm's growth rate. It indicates that the resource usage grows at the same rate as a particular function.

Diagram:

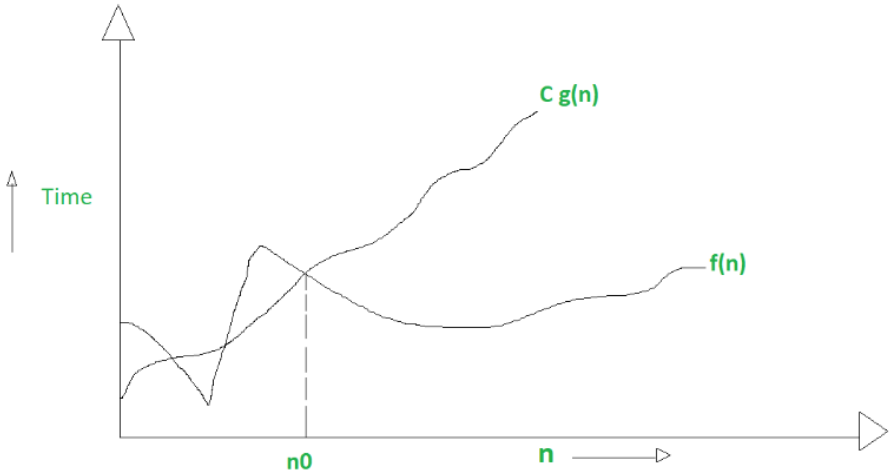

diagram showing Big O and Big Theta notation with curves representing different growth rates

Big O and Big Theta Notation for f(n) = 100 n^4 + 1000 n^3 + 100000

- **Big O:** O(n^4) (since the n^4term dominates as n grows)
- **Big Theta:** Θ(n^4) (because the n^4 term is the dominant term)

In summary, asymptotic bounds provide valuable insights into algorithm efficiency but should be used with awareness of their limitations.

(e)-

Write and explain the Left to Right binary exponentiation algorithm. Demonstrate the use of this algorithm to compute the value of 3^{29} (Show the steps of computation). Explain the worst-case complexity of this algorithm.

ANS.-

Left-to-Right Binary Exponentiation Algorithm:

1. Convert the exponent b into its binary form.
2. Initialize $result = 1$.
3. For each bit in the binary form of b (from left to right):
   - Square the current result: $result = result^2$.
   - If the current bit is 1, multiply $result$ by the base a: $result = result \times a$.
4. Return $result$.

Example: Compute 3^{29}

Binary of 29: 11101.

Steps:

- Start with $result = 1$.
- Process bits from left to right:
  $1 : 3^1 = 3, 11 : (3^2)^1 = 27, 111 : (27^2)^1 = 729, 1110 : 729^2 = 531441, 11101 : 531441 \times 3 = 1594323$.

Complexity:

- **Time Complexity:** $O(\log b)$ (for squaring and multiplications).
- **Space Complexity:** $O(1)$.

(f)- Write and explain the Bubble sort algorithm. Discuss its best and worst-case time complexity.

ANS.- **Bubble Sort Algorithm**

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted.

Algorithm Steps:

1. Start from the first element.
2. Compare it with the next element. If they are in the wrong order, swap them.
3. Move to the next pair and repeat step 2.
4. Repeat the process for all elements until no swaps are needed.

Time Complexity:

- **Best Case (Already Sorted):** $O(n)$
- **Worst Case (Reversed Order):** $O(n^2)$

  Bubble Sort is inefficient for large datasets but useful for small lists.

(g)-

What are the uses of recurrence relations? Solve the following recurrence relations using the Master's method

a. $T(n) = 4T\left(\frac{n}{4}\right) + n^1$

b. $T(n) = 4T\left(\frac{3n}{4}\right) + n^1$

ANS.- **Uses of Recurrence Relations**

Recurrence relations express the runtime of recursive algorithms. They are used to determine the time complexity of algorithms by defining the relation between the input size and subproblems.

Solving with Master's Method

a. $T(n) = 4T\ \frac{n}{4}\ + n^1$

Here, $a = 4, b = 4, f(n) = n^1$.

- $\log_b a = \log_4 4 = 1.$
- $f(n) = O(n^1).$

  Since $p = \log_b a, T(n) = \Theta(n \log n).$

b. $T(n) = 4T\left(\frac{3n}{4}\right) + n^1$

Here, $a = 4, b = \frac{4}{3}, f(n) = n^1$.

- $\log_b a = \log_{4/3} 4 \approx 1.26 > 1.$
  Since $p < \log_b a$, $T(n) = \Theta(n^{\log_b a})$.

Q.2 -

(a)- What is an Optimisation Problem? Explain with the help of an example. When would you use a Greedy Approach to solve optimisation problem? Formulate the Task Scheduling Problem as an optimisation problem and write a greedy algorithm to solve this problem. Also, solve the following fractional Knapsack problem using greedy approach. Show all the steps.

Suppose there is a knapsack of capacity 20 Kg and the following 6 items are to packed in it. The weight and profit of the items are as under:

$$(p_1, p_2, \ldots, p_6) = (30, 16, 18, 20, 10, 7)$$
$$(w_1, w_2, \ldots, w_6) = (5, 4, 6, 4, 5, 7)$$

Select a subset of the items that maximises the profit while keeping the total weight below or equal to the given capacity.

ANS.- Optimisation Problem

An optimisation problem aims to find the best solution from a set of possible solutions, where "best" is defined according to a specific objective function. It involves maximizing or minimizing a value (e.g., profit, cost, time).

Example:

- **Knapsack Problem:** Given a set of items with weights and values, select a subset of items to fit in a knapsack of limited capacity while maximizing the total value.

Greedy Approach

The greedy approach makes locally optimal choices at each step, hoping to reach the global optimum. It's suitable when:

- **Optimal Substructure:** The problem can be broken down into subproblems, and the optimal solution to the subproblems contributes to the optimal solution of the overall problem.

- **Greedy Choice Property:** Making the locally optimal choice at each step leads to the globally optimal solution.

Task Scheduling Problem as Optimisation

Objective: Schedule tasks with deadlines and durations to minimize the total weighted completion time.

Greedy Algorithm:

1. Sort tasks in ascending order of deadlines.
2. Schedule tasks in the sorted order.

Fractional Knapsack Problem

Objective: Select items to maximize profit while staying within the knapsack's capacity.

Greedy Algorithm:

1. Calculate the profit-to-weight ratio for each item.
2. Sort items in descending order of the ratio.
3. Add items to the knapsack in the sorted order, taking fractions of items if necessary to fill the remaining capacity.

Example:

1. Calculate ratios: (30/5, 16/4, 18/6, 20/4, 10/5, 7/7) = (6, 4, 3, 5, 2, 1)
2. Sort: (30/5, 20/4, 16/4, 18/6, 10/5, 7/7)
3. Take all of items 1 and 4 (profit 30 + 20 = 50)
4. Take 1/2 of item 2 (profit 16/2 = 8)
5. Total profit: 50 + 8 = 58

(b)- Assuming that data to be transmitted consists of only characters ‘a’ to ‘g’, design the Huffman code for the following frequencies of character data. Show all the steps of building a huffman tree. Also, show how a coded sequence using Huffman code can be decoded.

a:5, b:25, c:10, d:15, e:8, f:7, g:30

ANS.- To build the Huffman Tree for the given frequencies:

1. **Sort characters by frequency:**
   {a:5, f:7, e:8, c:10, d:15, b:25, g:30}
2. **Combine the two lowest:** (a+f) = 12
   {(af):12, e:8, c:10, d:15, b:25, g:30}
3. **Repeat:** (e+c) = 18
   {(af):12, (ec):18, d:15, b:25, g:30}

4. (af + d) = 27
   {(afd):27, (ec):18, b:25, g:30}
5. (ec + b) = 43
   {(afd):27, (ecb):43, g:30}
6. (afd + g) = 57
   {(ecb):43, (afdg):57}
7. **Final step:** (ecb + afdg) = 100 (Root)

Huffman Codes:

- g: 0
- ecb: 10, e: 100, c: 101, b: 11
- afdg: 11, a: 1100, f: 1101, d: 111

Decoding Example:

For 11001110111:

- 1100 → a
- 1101 → f
- 111 → d
  Decoded sequence: **"afd"**.

(c)- Explain the Merge procedure of the Merge Sort algorithm. Demonstrate the use of recursive Merge sort algorithm for sorting the following data of size 8: [19, 18, 16, 12, 11, 10, 9, 8]. Compute the complexity of Merge Sort algorithm.

ANS.- The **Merge procedure** in Merge Sort combines two sorted subarrays into a single sorted array. It compares elements from both subarrays and places the smallest element into the merged array, repeating until all elements are merged.

Recursive Merge Sort on [19, 18, 16, 12, 11, 10, 9, 8]

1. **Divide:** [19, 18, 16, 12] and [11, 10, 9, 8]
2. Further divide until single elements remain.
3. **Merge:**
   - Merge [18, 19] → [16, 18, 19] → [12, 16, 18, 19]
   - Merge [10, 11] → [9, 10, 11] → [8, 9, 10, 11]
   - Merge both halves: [8, 9, 10, 11, 12, 16, 18, 19]

Complexity:

- **Best, Worst, Average: O(n log n)**

(d)- Explain the divide and conquer approach of multiplying two large integers. Compute the time complexity of this approach. Also, explain the binary search algorithm and find its time complexity.

ANS.-

The divide-and-conquer approach for multiplying two large integers involves breaking them into smaller parts, solving subproblems recursively, and combining results. Karatsuba's algorithm improves upon the naive $O(n^2)$ method by reducing the number of multiplications. It splits numbers into halves and recursively computes three multiplications instead of four, achieving a time complexity of $O(n^{\log_2 3}) \approx O(n^{1.585})$.

Binary search efficiently finds an element in a sorted array by repeatedly halving the search space, comparing the middle element, and discarding half. Its time complexity is $O(\log n)$, making it much faster than linear search ($O(n)$).

(e)- Explain the Topological sorting with the help of an example. Also, explain the algorithm of finding strongly connected components in a directed Graph.

ANS.-

Topological Sorting & Strongly Connected Components

Topological sorting of a Directed Acyclic Graph (DAG) is a linear ordering of its vertices such that for every directed edge (u, v), vertex u appears before v in the ordering. It is used in scheduling problems like task dependency resolution.

Example: Consider a graph with edges: $A \to B$, $A \to C$, $B \to D$, $C \to D$. A valid topological order is: **A, B, C, D**.

Finding Strongly Connected Components (SCCs)

An SCC is a maximal subgraph where every vertex is reachable from every other vertex.

Kosaraju's Algorithm:

1. Perform DFS to get the finish order of nodes.
2. Reverse all edges of the graph.
3. Perform DFS on the reversed graph, processing nodes in decreasing finish order to identify SCCs.

Q.3 - Consider the following Graph:

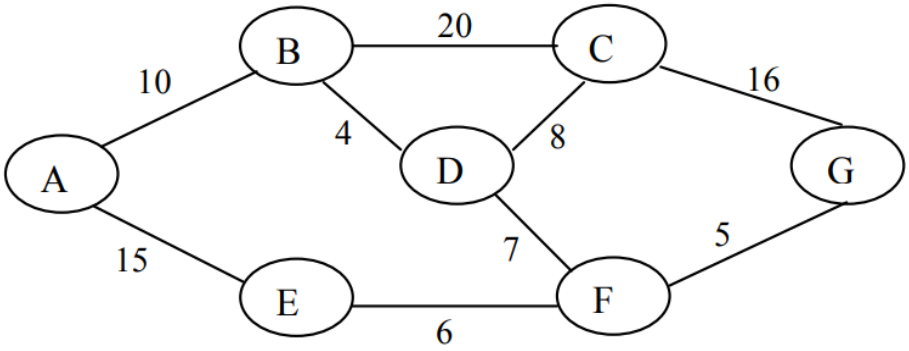

Figure 1: A sample weighted Graph

(a)- Write the Prim's algorithm to find the minimum cost spanning tree of a graph. Also, find the time complexity of Prim's algorithm. Demonstrate the use of Kruskal's algorithm and Prim's algorithm to find the minimum cost spanning tree for the Graph given in Figure 1. Show all the steps.

ANS.- Prim's Algorithm for Minimum Spanning Tree (MST)

Prim's algorithm starts with an arbitrary node and grows the MST by adding the smallest edge connecting a visited node to an unvisited node.

Steps for the given graph:

1. Start from **A**, pick the minimum edge **A–B (10)**.
2. Pick the smallest edge from visited nodes: **B–D (4)**.
3. Next, choose **D–F (7)**.
4. Select **F–G (5)**.
5. Pick **D–C (8)**.
6. Finally, add **F–E (6)**.

Total MST cost = 40.

Time Complexity: With a priority queue, Prim's algorithm runs in **O(E log V)**.

Kruskal's Algorithm

Sort edges, then add the smallest edge without forming a cycle:

1. **B–D (4), F–G (5), F–E (6), D–F (7), D–C (8), A–B (10)** → MST Cost = **40**.

Both methods yield the same MST cost.

(b)- Write the Dijkstra's shortest path algorithm. Also, find the time complexity of this shortest path algorithm. Find the shortest

paths from the vertex 'A' using Dijkstra's shortest path algorithm for the graph given in Figure 1. Show all the steps of computation.

ANS.- **Dijkstra's Shortest Path Algorithm**

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative weights.

Algorithm:

1. Initialize distances from source A to all vertices as ∞, except A (0).
2. Use a priority queue (min-heap) to process the smallest known distance vertex.
3. Update distances to adjacent vertices if a shorter path is found.
4. Repeat until all vertices are visited.

Steps for Graph (Starting from A):

1. Start at $A(0)$, pick **A–B (10)**, **A–E (15)**.
2. Pick **B (10)**, update **B–D (14)**.
3. Pick **D (14)**, update **D–F (21)**, **D–C (22)**.
4. Pick **E (15)**, update **E–F (21)**.
5. Pick **F (21)**, update **F–G (26)**.
6. Pick **C (22)**, then **G (26)**.

Final Shortest Paths:

A → B (10), A → D (14), A → E (15), A → C (22), A → F (21), A → G (26).

Time Complexity: Using a priority queue, $O((V + E) \log V)$.

(c)-

Explain the algorithm to find the optimal Binary Search Tree. Demonstrate this algorithm to find the Optimal Binary Search Tree for the following probability data (where p_i represents the probability that the search will be for the key node k_i, whereas q_i represents that the search is for dummy node d_i. Make suitable assumptions, if any)

i	0	1	2	3	4
p_i		0.10	0.15	0.20	0.10
q_i	0.05	0.10	0.10	0.10	0.10

ANS.- The **Optimal Binary Search Tree (OBST)** minimizes the expected search cost based on given probabilities. The **Dynamic Programming** approach is used to construct it.

Algorithm

1. Define `e[i][j]` as the expected cost of searching in subtree `T(i, j)`.
2. Define `w[i][j]` as the total probability sum in `T(i, j)`.
3. Initialize base cases:
   - `e[i][i-1] = q[i-1]` (cost of empty subtrees).
   - `w[i][i-1] = q[i-1]`.
4. Compute `e[i][j]` using:

$$e[i][j] = \min_{r \in [i,j]} (e[i][r-1] + e[r+1][j] + w[i][j])$$

5. Compute `w[i][j] = w[i][j-1] + p[j] + q[j]`.
6. Fill the table iteratively and select the optimal root.

Using the given data, construct the table and derive the optimal tree structure.

(d)-

Given the following sequence of chain multiplication of the matrices. Find the optimal way of multiplying these matrices:

Matrix	Dimension
A1	10×15
A2	15×5
A3	5×20
A4	20×10

ANS.- The **Matrix Chain Multiplication** problem minimizes the number of scalar multiplications needed to compute the product of matrices. The **Dynamic Programming** approach is used to solve it efficiently.

Algorithm

1. Let matrices have dimensions **p = {10, 15, 5, 20, 10}**.
2. Define `m[i][j]` as the minimum cost of multiplying matrices from `A_i` to `A_j`.
3. Initialize `m[i][i] = 0` for all `i`.
4. Compute `m[i][j]` using:

$$m[i][j] = \min_k (m[i][k] + m[k+1][j] + p[i-1] \times p[k] \times p[j])$$

5. Fill the DP table iteratively and determine the optimal parenthesization.

By computing, the optimal order is **((A1 × A2) × (A3 × A4))** with minimal cost.

(e)- Explain the Rabin Karp algorithm for string matching with the help of an example. Find the time complexity of this algorithm.

ANS.- **Rabin-Karp Algorithm for String Matching**

The **Rabin-Karp algorithm** is a hashing-based approach used for **pattern matching** in a given text. It works by computing a hash value for the pattern and comparing it with hash values of substrings in the text. If a match is found, a character-by-character comparison is performed to confirm the match.

Algorithm Steps

1. Compute the hash value of the pattern (P) and the first substring of text (T) of the same length.

2. Slide the pattern over the text one character at a time and update the hash using a rolling hash function.
3. If the hash values match, perform a character-by-character comparison to avoid hash collisions.

Example

Pattern: "abc", Text: "abdababc"

- Compute hash for "abc", check against substrings like "abd", "bda", etc.

Time Complexity

- Best/Average Case: $O(n)$
- Worst Case (hash collisions): $O(nm)$

Q.4 -

(a)- Explain the term Decision problem with the help of an example. Define the following problems and identify if they are decision problem or optimisation problem? Give reasons in support of your answer.

(i) Travelling Salesman Problem

(ii) Graph Colouring Problem

(iii) 0-1 Knapsack Problem

ANS.- A **decision problem** is a problem with a yes/no answer. It involves determining whether a solution exists that satisfies given constraints.

Example: The **Hamiltonian Cycle Problem** asks if a given graph contains a cycle that visits every vertex exactly once. The answer is either "Yes" or "No," making it a decision problem.

Problem Classification:

(i) **Travelling Salesman Problem (TSP)** – **Optimisation Problem**

- It seeks the shortest possible route visiting all cities and returning to the start. Since it requires finding the best solution, it is an optimisation problem.

(ii) **Graph Colouring Problem** – **Decision Problem**

- Given a graph and a number kkk, it asks whether the graph can be colored using at most kkk colors so that no adjacent vertices share the same color. This requires a yes/no answer.

(iii) **0-1 Knapsack Problem** – **Optimisation Problem**

- It aims to maximize the total value of selected items while staying within weight constraints, making it an optimisation problem.

(b)- What are P and NP class of Problems? Explain each class with the help of at least two examples.

ANS.- In computational complexity theory, **P and NP are classes of decision problems** that describe their solvability and efficiency.

- **P (Polynomial Time)**: Problems in P can be solved efficiently by a deterministic Turing machine in polynomial time. These problems have algorithms that run in O(n^k) time for some constant *k*.
  Examples:
  1. **Sorting (Merge Sort, Quick Sort)** – Runs in O(n log n) time.
  2. **Finding the Shortest Path (Dijkstra's Algorithm)** – Solves in $O(V^2)$ or better.
- **NP (Nondeterministic Polynomial Time)**: Problems in NP have solutions that can be *verified* in polynomial time but may not have known polynomial-time algorithms for solving them.
  Examples:
  1. **Traveling Salesman Problem (TSP)** – Checking a given path is easy, but finding the shortest one is hard.
  2. **Boolean Satisfiability (SAT)** – Verifying a satisfying assignment is quick, but finding one is difficult.

(c)- Define the NP-Hard and NP-Complete problem. How are they different from each other. Explain the use of polynomial time reduction with the help of an example.

ANS.- An **NP-Hard** problem is at least as hard as the hardest problems in **NP (Nondeterministic Polynomial time)** but may not be in NP itself. It does not necessarily have a solution that can be verified in polynomial time.
An **NP-Complete** problem is both in NP and NP-Hard, meaning:

1. It belongs to NP (its solution can be verified in polynomial time).
2. Any NP problem can be reduced to it in polynomial time.

Difference: Every NP-Complete problem is NP-Hard, but not all NP-Hard problems are NP-Complete (e.g., optimization problems like the Halting Problem).
Polynomial Time Reduction Example:
Consider **3-SAT → Clique Problem**. If we can transform any 3-SAT instance into an equivalent Clique problem instance in polynomial time, solving the Clique problem would also solve 3-SAT efficiently. This helps in proving NP-completeness.

(d)- Define the following Problems:

(i) SAT Problem

ANS.- The SAT (Boolean Satisfiability) problem is a fundamental decision problem in computer science and logic. It asks whether there exists an assignment of truth values (true/false) to a set of Boolean variables such that a given Boolean formula evaluates to true. The formula is usually expressed in **conjunctive normal form (CNF)**, meaning it consists of multiple clauses joined by ANDs, with literals in each clause connected by ORs. SAT was the first problem proven to be **NP-complete**, meaning that if an efficient algorithm exists for solving SAT, it could solve all problems in NP efficiently.

(ii) Clique problem

ANS.- The Clique problem is a well-known problem in graph theory. A **clique** in a graph is a subset of vertices such that every two vertices in the subset are connected by an edge. The **decision version** of the problem asks whether a graph contains a clique of at least a given size kkk. The problem is **NP-complete**, meaning no polynomial-time solution is known. The **optimization version** involves finding the largest possible clique in a graph, which is **NP-hard**.

(iii)Hamiltonian Cycle Problem

ANS.- A **Hamiltonian cycle** in a graph is a cycle that visits each vertex exactly once and returns to the starting vertex. The **Hamiltonian Cycle Problem** asks whether such a cycle exists in a given graph. This problem is **NP-complete**, making it computationally difficult to solve for large graphs. It has applications in routing, scheduling, and network topology.

(iv) Subset Sum Problem

ANS.- The **Subset Sum Problem** asks whether a subset of a given set of integers exists such that the sum of the subset equals a specific target value. It is a fundamental **NP-complete** problem, commonly used in cryptography and combinatorial optimization. The problem can be solved in exponential time using brute force or more efficiently with dynamic programming for small inputs.