DSA ASSIGNMENT

1.**Describe greedy algos with example**

Greedy Algorithms:

- **Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution.**
- **In these algorithms, decisions are made based on the information available at the current moment without considering the consequences of these decisions in the future.**
- **The key idea is to select the best possible choice at each step, leading to a solution that may not always be the most optimal but is often good enough for many problems.**

STEPS:

1. Define the problem – Clearly state the problem to solve.

2. Identify the greedy choice – Find the best immediate choice based on a specific criterion.

3. Sort data (if needed) – Sort or organize data as per the problem requirement.

4. Make the greedy choice – Pick the option that offers the most immediate benefit.

5. Reduce problem size – Solve the reduced problem by removing the chosen option.

6. Repeat– Continue making greedy choices until no further steps remain.

7. **Verify the solution – Check if the solution is correct or optimal for the problem.**

EXAMPLE:

Problem Statement:

You are given an infinite supply of coins of different denominations and a target amount. Your goal is to find the minimum number of coins that make up the target amount.

DSA ASSIGNMENT

INPUT:

Coins available: 1, 5, 10, 25 (in cents)

Target amount: 36 cents

Steps to Solve Using the Greedy Algorithm:

1. **Define the problem:**
    - **Minimize the number of coins needed to make up 36 cents using the available denominations (1, 5, 10, 25).**
2. **Identify the greedy choice:**
    - **At each step, choose the largest denomination coin that is less than or equal to the remaining amount. This helps minimize the number of coins.**
3. **Sort data (if needed):**
    - **Since the coin denominations are already sorted in descending order, no sorting is required.**
4. **Make the greedy choice:**
    - **Start with the largest coin that fits into the target amount.**
    - **Choose 25 cents (largest coin ≤ 36). Remaining amount: 36 - 25 = 11.**
    - **Choose 10 cents (largest coin ≤ 11). Remaining amount: 11 - 10 = 1.**
    - **Choose 1 cent (largest coin ≤ 1). Remaining amount: 1 - 1 = 0.**
5. **Reduce problem size:**
    - **After each choice, subtract the coin's value from the remaining amount until the target amount is zero.**
6. **Repeat:**
    - **Repeat the process until no remaining amount is left (i.e., the target amount becomes 0).**
7. **Solution:**
    - **The coins selected are: 25 cents, 10 cents, 1 cent.**

OUTPUT:

Selected coins: 25, 10, 1

Total coins used: 3

DSA ASSIGNMENT

Explanation:

- **The greedy approach works here because, at each step, choosing the largest coin minimizes the remaining amount, ultimately leading to a minimal number of coins.**

However, note that the greedy algorithm does not always give an optimal solution for all coin denominations. For example, if the available coins are 1, 3, and 4 cents and the target amount is 6 cents, the greedy algorithm would fail, as it might choose 4 + 1 + 1, while the optimal solution is 3 + 3.

In this case, since 25, 10, 5, and 1 are commonly used denominations, the greedy algorithm finds the optimal solution.

2. For all algorithms what is the time complexity, space complexity and data structure in use

1. Coin Change Problem

- **Objective: Minimize the number of coins needed to make a specific amount.**

Time Complexity:

- **Sorted Coin Denominations: O(1)O(1)O(1) per coin selection (since we use the largest denomination).**
- **Unsorted Denominations: O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of available coin denominations (if sorting is required).**

Space Complexity:

- **O(1): Only a fixed amount of space is needed regardless of input size (for storing remaining amount and coin count).**

Data Structure:

- **Array or List: To store the coin denominations.**

---

2. Activity Selection Problem

- **Objective: Select the maximum number of non-overlapping activities.**

Time Complexity:

- **Sorting activities by finish time: O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of activities.**
- **Selecting activities after sorting: O(n)O(n)O(n) for iterating through the list.**

Space Complexity:

- **O(n): For storing the list of activities.**

Data Structure:

- **Array or List: To store the activities and their start and finish times.**

---

3. Fractional Knapsack Problem

- **Objective: Maximize the total value of items in the knapsack by allowing fractions of items.**

Time Complexity:

- **Sorting items by value-to-weight ratio: O(nlog⁡n)O(n \log n)O(nlogn).**
- **Filling the knapsack: O(n)O(n)O(n) for iterating through the sorted items.**

Space Complexity:

- **O(n): For storing the items' values, weights, and value-to-weight ratios.**

Data Structure:

- **Array or List: To store item weights, values, and their ratios.**
- **Sorting algorithms like Merge Sort or Quick Sort may be used to sort the items by ratio.**

---

4. Huffman Coding

- **Objective: Create an optimal prefix code based on the frequencies of characters.**

Time Complexity:

- **Building the priority queue (min-heap): O(nlog⁡n)O(n \log n)O(nlogn), where nnn is the number of characters.**
- **Building the Huffman tree: O(nlog⁡n)O(n \log n)O(nlogn).**

Space Complexity:

- **O(n): For storing the frequency table and the Huffman tree.**

Data Structure:

- **Priority Queue (Min-Heap): To repeatedly pick the two smallest frequency nodes.**
- **Binary Tree: To construct the Huffman coding tree.**

---

5. Prim's Algorithm (for Minimum Spanning Tree)

- **Objective: Find the Minimum Spanning Tree of a graph.**

Time Complexity:

- **Using a priority queue (min-heap): O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), where VVV is the number of vertices and EEE is the number of edges.**
- **Without priority queue (adjacency matrix): O(V2)O(V^2)O(V2).**

Space Complexity:

- **O(V + E): To store the graph and the Minimum Spanning Tree.**

Data Structure:

- **Priority Queue (Min-Heap): To select the next minimum weight edge.**
- **Adjacency List: To store the graph (if using a more efficient implementation).**
- **Array: To track visited vertices and their corresponding costs.**

---

6. Kruskal's Algorithm (for Minimum Spanning Tree)

- **Objective: Find the Minimum Spanning Tree of a graph by sorting edges.**

Time Complexity:

- **Sorting edges: O(Elog⁡E)O(E \log E)O(ElogE), where EEE is the number of edges.**
- **Union-Find operations: O(Elog⁡V)O(E \log V)O(ElogV), using a Union-Find/Disjoint Set structure, where VVV is the number of vertices.**

Space Complexity:

- **O(V + E): To store the graph and the disjoint-set data structure.**

DSA ASSIGNMENT

Data Structure:

- **Union-Find/Disjoint Set: For cycle detection when adding edges to the MST.**
- **Array: To store the parent and rank for the Union-Find operations.**
- **Edge List: To store and sort the graph's edges.**

---

7. Dijkstra's Algorithm (for Shortest Path)

- **Objective: Find the shortest path from a source vertex to all other vertices in a weighted graph.**

Time Complexity:

- **Using a priority queue (min-heap): O((V+E)log V)O((V + E) \log V)O((V+E)logV), where VVV is the number of vertices and EEE is the number of edges.**
- **Without a priority queue (using arrays): O(V2)O(V^2)O(V2).**

Space Complexity:

- **O(V + E): To store the graph and the shortest paths.**

Data Structure:

- **Priority Queue (Min-Heap): To extract the next vertex with the minimum distance.**
- **Adjacency List: To represent the graph.**
- **Array: To store distances from the source vertex and to track visited vertices.**

3. What is Dynamic Programming? What are the types? Different algorithms

Dynamic Programming (DP) is an algorithmic technique used to solve complex problems by breaking them down into simpler sub-problems. It is especially useful when these sub-problems overlap, meaning the same sub-problems are solved multiple times. In dynamic programming, each sub-problem is solved once, and its result is stored, so it can be reused later to avoid redundant computations. This technique is used to optimize problems with optimal substructure and overlapping subproblems.

Types of Dynamic Programming:

DSA ASSIGNMENT

1. Memoization (Top-Down Approach)

- **In memoization, we solve the problem in a top-down manner. We start solving the main problem and then recursively solve smaller sub-problems. Each sub-problem result is stored (memoized) to avoid redundant calculations.**
- **Approach: Recursive function + Cache (e.g., dictionary or array).**
- **Example:**
    - **Fibonacci sequence: Use a cache to store previously computed Fibonacci values.**

2. Tabulation (Bottom-Up Approach)

- **In tabulation, we solve the problem in a bottom-up manner. We start solving the smallest sub-problems first, and gradually build up to the solution of the main problem.**
- **Approach: Iterative + Table (usually a 1D or 2D array).**
- **Example:**
    - **Fibonacci sequence: Start from the base cases (F(0) and F(1)) and compute the Fibonacci numbers iteratively up to the desired value.**
    -

Common Dynamic Programming Algorithms:

1. Fibonacci Sequence

2. 0/1 Knapsack Problem

3.Matrix Chain Multiplication

4. Types of dynamic programming , time complexity, space complexity and data structure in use

1. Fibonacci Sequence

- **Time Complexity: O(n)O(n)O(n) (with memoization or tabulation)**
- **Space Complexity: O(n)O(n)O(n) (for memoization) or O(1)O(1)O(1) (for iterative approach)**

- **Data Structure: Array (for storing Fibonacci values) or variables (for iterative).**

---

2. 0/1 Knapsack Problem

- **Time Complexity: O(nW)O(nW)O(nW) (where nnn is the number of items and WWW is the maximum capacity)**
- **Space Complexity: O(nW)O(nW)O(nW) (using a 2D table) or O(W)O(W)O(W) (using a 1D table for space optimization)**
- **Data Structure: 2D array or 1D array.**

---

3. Longest Common Subsequence (LCS)

- **Time Complexity: O(mn)O(mn)O(mn) (where mmm and nnn are the lengths of the two strings)**
- **Space Complexity: O(mn)O(mn)O(mn) (for a 2D table)**
- **Data Structure: 2D array.**

---

4. Longest Increasing Subsequence (LIS)

- **Time Complexity: O(n2)O(n^2)O(n2) (using dynamic programming) or O(nlog⁡n)O(n \log n)O(nlogn) (using binary search)**
- **Space Complexity: O(n)O(n)O(n)**
- **Data Structure: 1D array.**

---

5. Matrix Chain Multiplication

- **Time Complexity: O(n3)O(n^3)O(n3) (where nnn is the number of matrices)**
- **Space Complexity: O(n2)O(n^2)O(n2) (for storing minimum costs)**
- **Data Structure: 2D array.**

---

6. Edit Distance (Levenshtein Distance)

- **Time Complexity: O(mn)O(mn)O(mn) (where mmm and nnn are the lengths of the two strings)**

- **Space Complexity: O(mn)O(mn)O(mn) (for a 2D table) or O(n)O(n)O(n) (using only two rows for space optimization)**
- **Data Structure: 2D array or 1D array.**

---

7. Coin Change Problem

- **Time Complexity: O(n×m)O(n \times m)O(n×m) (where nnn is the amount and mmm is the number of coin denominations)**
- **Space Complexity: O(n)O(n)O(n) (for storing ways to make change)**
- **Data Structure: 1D array.**

---

8. Rod Cutting Problem

- **Time Complexity: O(n2)O(n^2)O(n2)**
- **Space Complexity: O(n)O(n)O(n)**
- **Data Structure: 1D array.**

---

9. Travelling Salesman Problem (TSP)

- **Time Complexity: O(n2⋅2n)O(n^2 \cdot 2^n)O(n2⋅2n) (using dynamic programming)**
- **Space Complexity: O(n⋅2n)O(n \cdot 2^n)O(n⋅2n)**
- **Data Structure: 2D array or bitmask.**

---

10. Bellman-Ford Algorithm

- **Time Complexity: O(VE)O(VE)O(VE) (where VVV is the number of vertices and EEE is the number of edges)**
- **Space Complexity: O(V)O(V)O(V)**
- **Data Structure: Array (for distances) or adjacency list/array.**

---

11. Dijkstra's Algorithm

- **Time Complexity: O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV) (using a priority queue)**
- **Space Complexity: O(V)O(V)O(V)**
- **Data Structure: Priority queue (min-heap) and array (for distances).**

---

12. Floyd-Warshall Algorithm

- **Time Complexity: O(V3)O(V^3)O(V3)**
- **Space Complexity: O(V2)O(V^2)O(V2)**
- **Data Structure: 2D array (for distances).**

---

13. Palindromic Subsequence

- **Time Complexity: O(n2)O(n^2)O(n2) (where nnn is the length of the string)**
- **Space Complexity: O(n2)O(n^2)O(n2) (for a 2D table)**
- **Data Structure: 2D array.**

---

14. Wildcard Matching

- **Time Complexity: O(mn)O(mn)O(mn) (where mmm is the length of the string and nnn is the length of the pattern)**
- **Space Complexity: O(mn)O(mn)O(mn) (for a 2D table) or O(n)O(n)O(n) (using a single row)**
- **Data Structure: 2D array or 1D array.**

---

15. Regular Expression Matching

- **Time Complexity: O(mn)O(mn)O(mn) (where mmm is the length of the string and nnn is the length of the pattern)**
- **Space Complexity: O(mn)O(mn)O(mn) (for a 2D table) or O(n)O(n)O(n) (using a single row)**
- **Data Structure: 2D array or 1D array.**