DSA ASSIGNMENT

1.Describe Greedy algorithms with example.

A Greedy Algorithm is a problem-solving approach where, at each step, the algorithm makes the locally optimal choice, aiming to find the global optimum. It is called "greedy" because it makes decisions based on the current situation without considering the overall problem. The hope is that these locally optimal solutions will lead to a globally optimal solution.

Greedy algorithms are generally faster and simpler to implement but do not always produce the globally optimal solution. For certain types of problems, such as Minimum Spanning Trees or Huffman Coding, greedy algorithms always provide the correct solution.

Example Problems Solved by Greedy Algorithms:

- Fractional Knapsack Problem
- Minimum Spanning Tree (MST) Problems (Prim's and Kruskal's)
- Huffman Coding
- Dijkstra's Algorithm for Shortest Paths

Steps to Implement a Greedy Algorithm:

1. Identify the Problem: First, ensure that the problem can be solved using a greedy approach (i.e., it exhibits the greedy choice property and optimal substructure).
2. Make a Greedy Choice: At each step, select the option that looks best at that moment. The choice should be based on a criterion that leads to an optimal solution.
3. Check Feasibility: After making the greedy choice, verify that the solution is still feasible (e.g., constraints are satisfied).
4. Iterate: Repeat the process until you reach a complete solution (either all elements are covered, or the objective is met).

Example : Fractional Knapsack Problem

In the Fractional Knapsack Problem, you are given a set of items, each with a weight and value. The goal is to fill a knapsack of capacity WWW such that the total value is maximized. Unlike the 0/1 Knapsack, you can take fractions of an item.

Problem:

- Input: Items with weights and values, and a knapsack of capacity WWW.

- Output: Maximum value that can be obtained by taking whole or fractional parts of the items.

Approach:

1. Compute the value/weight ratio for each item.
2. Sort items based on the value/weight ratio in descending order.
3. Take as much as possible from the item with the highest value/weight ratio until the knapsack is full.

Greedy Algorithm (Pseudocode):

python

Copy code

```
def fractional_knapsack(values, weights, capacity):
    # Step 1: Compute value/weight ratio and sort items by it
    items = sorted(zip(values, weights), key=lambda x: x[0]/x[1], reverse=True)
        total_value = 0
    current_weight = 0
    for value, weight in items:
        if current_weight + weight <= capacity:
            total_value += value
            current_weight += weight
        else:
            remaining_capacity = capacity - current_weight
            total_value += (value / weight) * remaining_capacity
            break

    return total_value
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50
result = fractional_knapsack(values, weights, capacity)
```

print(result)

Time Complexity:

O(nlogn)

Advantages of Greedy Algorithms:

- Simplicity: Easy to understand and implement.
- Efficiency: Often faster than other algorithms (such as dynamic programming) for certain problems.
- Space Efficiency: Typically requires minimal memory.

Disadvantages:

- Not Always Optimal: Greedy algorithms don't always guarantee an optimal solution, especially for problems like the 0/1 Knapsack, Traveling Salesman Problem (TSP), etc.
- Problem-Specific: They work well only for problems with the greedy-choice property and optimal substructure.

2.What is the Time complexity, Space complexity and data structures used in different types of greedy algorithms?

1. Fractional Knapsack Algorithm

- **Problem**: Given a set of items with weights and values, maximize the total value by selecting whole or fractional parts of the items within a given capacity.

2. Dijkstra's Algorithm (Single-Source Shortest Path)

- **Problem**: Find the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

3. Prim's Algorithm (Minimum Spanning Tree)

- **Problem**: Find the minimum spanning tree of a graph, which connects all vertices with the minimum possible total edge weight.

4. Kruskal's Algorithm (Minimum Spanning Tree)

- **Problem**: Find the minimum spanning tree of a graph by adding edges in increasing order of weight, ensuring no cycles are formed.

Algorithm	Time Complexity	Space Complexity	Data Structures
Fractional Knapsack	O(nlogn)	O(n)	Array/List for storing items and ratios
Dijkstra's Algorithm	O((V+E)logV)	O(V+E)	Priority Queue (Min-Heap), Adjacency List
Prim's Algorithm	O((V+E)logV)	O(V+E)	Priority Queue (Min-Heap), Adjacency List
Kruskal's Algorithm	O(ElogE	O(V+E)	Union-Find (Disjoint Set), List of edges

- **Fractional Knapsack** has the complexity dominated by sorting, making it O(nlogn).
- Both **Dijkstra's** and **Prim's** algorithms have similar time complexities because of the use of priority queues for efficient vertex extraction.
- **Kruskal's Algorithm** relies on sorting the edges and efficient union-find operations for managing the sets, giving it a complexity of O(ElogE).

3.What is Dynamic programming? What are the types of Different algothms in DP?

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when a problem has overlapping subproblems (subproblems that recur multiple times) and optimal substructure (the optimal solution to the problem can be constructed from the optimal solutions of its subproblems). DP is used to avoid redundant calculations by storing the results of subproblems and reusing them when needed, thus improving efficiency.

Characteristics of Dynamic Programming:

1. Overlapping Subproblems: The problem can be broken into smaller subproblems, which are solved repeatedly.
2. Optimal Substructure: The optimal solution to the problem is built from the optimal solutions to its subproblems.
3. Memoization or Tabulation: Solutions to subproblems are stored for future use to avoid redundant computation.

Types of Dynamic Programming Algorithms
There are two main strategies in DP based on how subproblem results are stored and reused:

1. Memoization (Top-Down DP)

    - In memoization, we solve the problem in a top-down manner.
    - The main problem is solved recursively, and the results of subproblems are stored in a cache (usually an array or dictionary) for future use.
    - When a subproblem is encountered again, its value is retrieved from the cache, avoiding redundant calculations.
    - Example: Fibonacci series using recursive DP with memoization.

    Advantages:
    - Only computes necessary subproblems.
    - Space complexity depends on the depth of recursion.

    Disadvantages:
    - Recursion depth can be a limiting factor in some languages that have recursion depth limits.

2. Tabulation (Bottom-Up DP)

    - In tabulation, we solve the problem in a bottom-up manner by first solving the smallest subproblems, and then combining them to solve larger subproblems.
    - Instead of recursion, we use iteration and store the results in a table (usually an array).
    - Example: Fibonacci series using iterative DP (tabulation).

    Advantages:
    - No recursive overhead (so, more efficient for languages with recursion depth limitations).
    - Often easier to understand and implement.

    Disadvantages:
    - May compute subproblems that are not needed to solve the problem.

4.What is the Time complexity, Space complexity and data structures used in different algorithms in dynamic programming?

Summary Table of Common Algorithms:

Algorithm	Time Complexity	Space Complexity	Data Structures
Fibonacci Sequence	O(n)	O(n) or O(1)	Array
0/1 Knapsack	O(nW)	O(nW)	2D Array/Matrix
Longest Common Subsequence	O(m×n)	O(m×n)	2D Array/Matrix
Matrix Chain Multiplication	O(n^3)	O(n^2)	2D Array/Matrix
Edit Distance	O(m×n)	O(m×n)	2D Array/Matrix
Coin Change Problem	O(n×m)	O(n)	Array
Longest Increasing Subsequence	O(n^2) or O(nlogn)	O(n)	Array
Rod Cutting	O(n^2)	O(n)	Array
Subset Sum Problem	O(n×S)	O(n×S) or O(S)	2D Array/Matrix
Travelling Salesman Problem	O(n^2 *2^n)	O(n^2 *2^n)	2D Array with Bitmasking

- **Time complexity** in DP depends on the number of states and how each state is computed from previous states.
- **Space complexity** can often be reduced by using optimized techniques (e.g., by only keeping track of necessary rows/columns in memory).
- **Data structures** such as arrays, matrices, and bitmasking are commonly used to store intermediate results.