ASSIGNMENT: UNIT - 4

Describe greedy algorithm:

Greedy algorithms are a type of algorithmic paradigm that make the locally optimal choice at each stage with the hope of finding a global optimum. They are often used for optimization problems where a sequence of choices can lead to the best overall solution.

1. **Local Optimization**: Greedy algorithms make the best possible choice at each step based on the current situation without looking ahead.
2. **Feasibility**: Each chosen option must satisfy the problem's constraints and contribute to a valid solution.
3. **Not Always Optimal**: Greedy algorithms do not guarantee an optimal solution for every problem; their effectiveness depends on the specific problem structure.

Examples:

* Dijkstra’s algorithm, Prims algorithms, Fractional Knapsack, Kruskal’s algorithm, Huffman algorithm.

Dijkstra’s Algorithm:

**Dijkstra’s algorithm** is a well-known algorithm used to find the shortest path from a starting vertex to all other vertices in a weighted graph with non-negative edge weights. It is widely used in network routing and geographic mapping applications.

The time complexity of Dijkstra's algorithm depends on the data structures used to implement the priority queue.

* **Time Complexity**: O(V2), This occurs because for each vertex, you might have to scan through all vertices to find the minimum distance vertex.
* **Time Complexity**: O((V+E)log⁡V) , In this case, inserting and extracting from the heap takes O(log⁡V), and for each vertex and edge, you perform operations related to the heap.
* **Space Complexity**: O(V)

This includes space for storing:

* The distance array (to hold the shortest distance from the source to each vertex).
* The previous array (to keep track of the path).
* The priority queue (which can hold all vertices).

Prims Algorithm:

**Prim's algorithm** is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph with weighted edges. The algorithm constructs the MST by starting from an arbitrary vertex and repeatedly adding the smallest edge that connects a vertex in the growing MST to a vertex outside it, ensuring that no cycles are formed. DS in use binary heap(min).

* **Time Complexity**: O(V2) using adjacency matrix .This is due to scanning all vertices to find the minimum edge.

**Time Complexity**: O((V+E)log⁡V) using adjacency list and a binary heap(min). In this case, for each vertex, you extract the minimum edge from the heap (which takes O(log⁡V) and then update adjacent vertices.

* **Space Complexity**: O(V+E) for the priority queue and O(V2) for the adjacency matrix representation.

**Where V is the number of vertices and E is the number of edges in the graph.**

**Fractional Knapsack:**

The Fractional Knapsack Problem involves maximizing total value in a knapsack by taking fractions of items, subject to a weight limit. DS in used Arrays and priority Queue.

* **Time Complexity:O(n log n)**: This is primarily due to the need to sort the items based on their value-to-weight ratios. After sorting, the selection process (iterating through the items) takes linear time, resulting in an overall complexity of O(nlogn).
* **Space Complexity:O(n)**: This space is used for storing the items along with their weights and values. Additional space may be used for the sorting algorithm, depending on the implementation, but the primary space requirement is for the items themselves.

Kruskal’s Algorithm:

Kruskal's Algorithm is a greedy algorithm that finds the minimum spanning tree of a connected, undirected graph by adding the smallest edge that connects two different components, ensuring no cycles are formed. DS used in union find.

* **Time Complexity: O(E log E)**: The dominant factor is the sorting of edges, which takes O(Elog⁡E). Since E (the number of edges) can be at most V2 (where V is the number of vertices), it can also be expressed as O(Elog⁡V)in terms of vertices.
* **Space Complexity: O(V + E)**: This includes space for storing the edges (in an edge list or adjacency list) and the disjoint-set (union-find) data structure used to manage the connected components.

Huffman Coding:

The Huffman Algorithm is a greedy algorithm used for lossless data compression that constructs a binary tree, assigning shorter binary codes to more frequent characters and longer codes to less frequent ones. DS used in priority queue

* **Time Complexity: O(n log n)**: The dominant factor is the need to sort the characters (or nodes) based on their frequencies, which takes O(nlog⁡n). Constructing the Huffman tree itself involves n−1 merging operations, which can be done in O(n)using a priority queue (min-heap), but the sorting step generally dominates.
* **Space Complexity: O(n)**: This includes space for storing the frequency table, the priority queue (or heap) used to build the Huffman tree, and the resulting codes for each character.

Dynamic Programming:

Dynamic Programming (DP) is a method used in algorithms to solve problems by breaking them down into simpler subproblems and solving each subproblem just once, storing the results for future use. This technique **is** particularly useful for optimization problems where the solution can be constructed from solutions to smaller subproblems.

1. **Top-Down Approach (Memoization)**:

* **Description**: This approach involves solving the problem recursively and storing the results of subproblems to avoid redundant computations.
* **Characteristics:**
* Starts with the main problem and breaks it down recursively.
* Uses a cache (usually an array or dictionary) to store the results of subproblems.
* **Example**: Computing Fibonacci numbers using recursion and memoization.

1. **Bottom-Up Approach (Tabulation)**:

* **Description**: This approach involves solving all possible subproblems first, typically using an iterative process, and building up the solution to the main problem.
* **Characteristics:**
* Starts with the simplest subproblems and iteratively combines them to solve larger subproblems.
* Uses a table (usually an array) to store solutions for all subproblems.
* **Example**: Filling a table to find the Longest Common Subsequence.

Example:

* **Fibonacci Numbers:**

 Algorithm: Recursive with memoization or iterative.

 Time Complexity: O(n)

 Space Complexity: O(n) (with memoization) or O(1) (iterative).

**DS in used 2D Arrray**

* **Longest Common Subsequence (LCS):**

 Algorithm: Build a 2D table to store lengths of common subsequences.

 Time Complexity: O(m \* n)

 Space Complexity: O(m \* n)

**DS in used 2D Array**

* **0/1 Knapsack Problem:**

 Algorithm: Use a 2D table to decide whether to include each item.

 Time Complexity: O(n \* W) where n is the number of items and W is the maximum weight.

 Space Complexity: O(n \* W)

**DS in used 2D Array**

* **Coin Change Problem:**

 Algorithm: Use a 1D or 2D table to count combinations.

 Time Complexity: O(n \* m) where n is the amount and m is the number of coin types.

 Space Complexity: O(n) (using a 1D array).

**DS in used Stack or Array**

* **Maximum Subarray Sum (Kadane's Algorithm):**

 Algorithm: Track the maximum sum with two variables.

 Time Complexity: O(n)

 Space Complexity: O(1)