DATA STRUCTURES AND ALGORITHMS

K.NIRANJANA

CSBS-B

2ND YEAR

Greedy Algorithms

Definition:

- A Greedy Algorithm is an algorithmic technique where we solve a problem by selecting the best available option at each step without considering whether this choice will lead to an overall optimal result. A Greedy Algorithm never reverses its decisions once made.

- It typically works using a Top-Down approach.

- **Optimality:** Doesn't always guarantee an optimal solution, depends on the problem.

Use cases:

- When there is logical reason where we can get the optimal solution by choosing the best option

Examples:

Greedy algorithms make a series of choices, each of which looks the best at the moment, without considering the global optimum. They are efficient for problems where local optimum decisions lead to a global optimum.

Dijkstra’s Algorithm

- **Description:** Dijkstra's Algorithm is a graph search algorithm that finds the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights. It maintains a set of nodes whose shortest distance from the source is known and iteratively explores the nearest unvisited node, updating the shortest path estimates for its neighbors.
- **Explanation**: Dijkstra’s algorithm finds the shortest path from a source node to all other nodes in a weighted graph. It starts with the source node and explores all neighboring nodes with the smallest tentative distance, updating the shortest path until all nodes are processed.
- **Type**: Greedy Algorithm.
- **Use Case**: Finding the shortest path in road maps, networks, and navigation systems.
- **Time Complexity**: O(V^2)with an adjacency matrix, O(E log V) with a priority queue.
- **Space Complexity**: O(V) for distance storage.
- **Example**: Finding the shortest distance between cities on a map.

Prim's Algorithm

- **Description:** This algorithm finds the Minimum Spanning Tree (MST) of a weighted undirected graph by growing the MST one edge at a time, always choosing the smallest edge that connects a vertex in the MST to a vertex outside the MST
- **Explanation**: Prim's algorithm builds a minimum spanning tree (MST) by starting from a random node and repeatedly adding the shortest edge that connects a node in the MST to a node outside of it.
- **Type**: Greedy Algorithm.
- **Use Case**: Network design, such as laying cables or optimizing a communication network.
- **Time Complexity**: O(E log V) with a binary heap and adjacency list.
- **Space Complexity**: O(V) for MST set and key array.
- **Example**: Laying out a telecommunications network.

Kruskal’s Algorithm

- **Description:** This algorithm finds the MST of a graph by sorting all edges and adding them one by one to the MST, provided they do not form a cycle**.**
- **Explanation**: Kruskal's algorithm builds a minimum spanning tree by sorting all edges and adding the smallest edge to the tree, ensuring no cycle is formed.
- **Type**: Greedy Algorithm.
- **Use Case**: Finding MST in networks, clustering algorithms.
- **Time Complexity**: O(E log E) due to sorting of edges.
- **Space Complexity**: O(V) for storing the parent array for the disjoint-set.
- **Example**: Finding the least-cost network of roads between cities.

Fractional Knapsack Problem

- **Description:** In this problem, we can take fractions of an item. The goal is to maximize the total value of items in the knapsack by taking as much as possible of the highest value-to-weight ratio items first.
- **Explanation**: The fractional knapsack problem allows taking fractions of items, aiming to maximize the total value in a knapsack without exceeding the weight limit. The greedy approach picks items based on the highest value/weight ratio.
- **Type**: Greedy Algorithm.
- **Use Case**: Resource allocation problems.
- **Time Complexity**: O(n log n) due to sorting items by value/weight.
- **Space Complexity**: O(1).
- **Example**: Maximizing profits with divisible goods.

Huffman Tree (Huffman Coding)

- **Description:** algorithm creates an optimal prefix code for data compression. It builds a tree based on the frequency of each character, combining the least frequent characters until only one tree remains.
- **Explanation**: Huffman coding is used to compress data by building a tree with the least frequent elements merged first, resulting in an optimal prefix code.
- **Type**: Greedy Algorithm.
- **Use Case**: File compression, data transmission.
- **Time Complexity**: O(n log n), where n is the number of distinct characters.
- **Space Complexity**: O(n) for the tree.
- **Example**: Compressing text files for storage.

Dynamic Programming (DP)

- **Definition:**
  Dynamic Programming is an algorithmic technique for solving optimization problems by breaking them down into simpler subproblems and leveraging the optimal solutions to these subproblems to obtain the overall optimal solution. Dynamic Programming always contains overlapping subproblems in addition to the previous two values.
- **Types of Dynamic Programming**

1. Top-Down Approach (Memoization):

Description: This approach uses recursion and stores the results of previously solved subproblems. When the same subproblem is encountered again, the stored result is returned from memory instead of being recalculated.

1. Bottom-Up Approach (Tabulation):

Description: This approach iteratively solves all subproblems starting from the smallest ones and builds up to the overall solution. Results are stored in a table (array or matrix) to avoid recalculating.

- **Optimality:** Always guarantees an optimal solution.
- Dynamic programming solves problems by breaking them down into subproblems and storing their solutions to avoid redundant work. It is most useful for optimization problems.

0/1 Knapsack Problem

- **Explanation**: The 0/1 knapsack problem involves selecting items to maximize the total value without exceeding the knapsack's capacity. Unlike the fractional version, items cannot be divided.
- **Type:** DP Algorithm.
- **Use Case:** Budget allocation, resource management.
- **Time Complexity**: O(n·W), where n is the number of items, and W is the capacity.
- **Space Complexity:** O(n·W).
- **Example:** Packing a suitcase without exceeding weight limits.

Bellman-Ford Algorithm

- **Explanation:** The Bellman-Ford algorithm computes the shortest paths from a single source to all vertices in a graph with negative weight edges. It repeatedly relaxes edges and updates distances.
- **Type:** DP Algorithm.
- **Use Case:** Shortest path problems with negative weights.
- **Time Complexity:** O(V·E), where V is the number of vertices, and E is the number of edges.
- **Space Complexity**: O(V) for storing distances.
- **Example:** Finding the cheapest path with discounts or penalties in a network.

Floyd-Warshall Algorithm

- **Explanation:** The Floyd-Warshall algorithm finds the shortest paths between all pairs of nodes in a weighted graph. It updates the shortest path matrix by considering each node as an intermediate node**.**
- **Type**: DP Algorithm**.**
- **Use Case:** Shortest path problems between all pairs, used in network routing**.**
- **Time Complexity:** O(V^3), where V is the number of vertices.
- **Space Complexity**: O(V^2).
- **Example**: Calculating the shortest distances between all cities in a network.

Optimal Merge Pattern for Merging of Files

- **Explanation:** This problem seeks to minimize the total computational cost when merging multiple files. The approach is similar to Huffman coding, where the smallest files are merged first to minimize the total merge cost.
- **Type**: Greedy Algorithm.
- **Use Case**: File merging, combining datasets.
- **Time Complexity:** O(n log n) due to sorting.
- **Space Complexity**: O(n) for storing intermediate results.
- **Example:** Merging sorted log files.

Mixed Approach Algorithms (Both Greedy and DP Elements)

Relaxed Knapsack Problem

- **Explanation:** The relaxed knapsack problem allows fractions of items to be taken and is solved using a greedy approach. However, if additional constraints are added (such as item dependencies), a dynamic programming approach may be used.
- **Type:** Mixed (Greedy and DP depending on constraints).
- **Use Case:** Resource allocation with complex constraints.
- **Time Complexity**: O(n log n) for fractional, O(n·W) for 0/1.
- **Space Complexity**: O(1) for fractional, O(n·W) for 0/1.
- **Example:** Allocating bandwidth with restrictions.

Algorithm Classification with Complexity Table

Algorithm	Type	Time Complexity	Space Complexity	Use Case
Dijkstra's Algorithm	Greedy	O(E log V)	O(V)	Shortest path in graphs
Prim's Algorithm	Greedy	O(E log V)	O(V)	Minimum Spanning Tree (MST)
Kruskal's Algorithm	Greedy	O(E log E)	O(V)	MST using edge sorting
Fractional Knapsack	Greedy	O(n log n)	O(1)	Maximizing profit by dividing items
Huffman Tree	Greedy	O(n log n)	O(n)	File compression
0/1 Knapsack	DP	O(n·W)	O(n·W)	Resource allocation with whole items
Bellman-Ford Algorithm	DP	O(V·E)	O(V)	Shortest path with negative weights
Floyd-Warshall Algorithm	DP	O(V^3)	O(V^2)	Shortest paths between all pairs
Optimal Merge Pattern	Greedy	O(n log n)		

Coin Change Problem

Explanation

The Coin Change Problem involves finding the minimum number of coins required to make a certain amount of money using a given set of denominations. It can also involve finding the number of different ways to make that amount.

Objective: Minimize the number of coins or count the combinations to form a specific amount.

Type

- Dynamic Programming (DP) or Greedy (depending on the denominations).

Use Case

- Making change in cash transactions, financial applications, and resource allocation.

Time Complexity

- **DP Approach**: O(n·m), where n is the amount and mmm is the number of coin denominations.
- **Greedy Approach**: O(m), but only applicable for certain denominations.

Space Complexity

- **DP Approach:** O(n) for storing the number of ways to make change for each amount.
- **Greedy Approach :**O(1) as it uses constant space for tracking coins.

Example

- Given coin denominations of [1,2,5] and a target amount of 11:
  - **Minimum Coins:** 333 (e.g., 5+5+1)
  - **Combinations:** Multiple ways to reach 11 using the denominations.

Bike Petrol Problem

Explanation

The Bike Petrol Problem involves calculating the minimum amount of petrol required to complete a journey between multiple checkpoints, given constraints like fuel tank capacity and petrol available at each checkpoint.

Objective: Optimize the petrol usage while ensuring the journey is completed without running out of fuel.

Type:

Dynamic Programming (DP) or Greedy

(depending on the constraints and problem formulation).

Use Case

Route planning for logistics, managing fuel resources in transportation, and optimizing travel costs.

Time Complexity

- **DP Approach**: O(n·d), where n is the number of checkpoints and d is the maximum distance that can be travelled with the available petrol**.**
- **Greedy Approach:** O(n) to make local optimal choices for refuelling at checkpoints.

Space Complexity

- **DP Approach:** O(n) for storing the minimum petrol needed at each checkpoint.
- **Greedy Approach:** O(1) as it tracks minimal state without additional structures.

Example

- Given checkpoints with petrol availability: [5,10,15] litres and a total distance of 20 km:
- Calculate the minimum petrol needed at each checkpoint to ensure the bike can reach the destination.