Greedy algorithm:

A greedy algorithm is a method for solving problems by making the most optimal choice at each step with the goal of finding a global optimum. It works by taking the best immediate, or "greedy," choice and does not reconsider its decisions. This approach is simple and efficient but does not always yield the best overall solution. Examples include Dijkstra's shortest path algorithm and Huffman coding. While greedy algorithms can be very fast and easy to implement, they are most effective when a locally optimal choice also leads to a globally optimal solutionAlways choose the largest denomination coin that is less than or equal to the remaining amount.

* This approach works optimally for some denominations but not all.

Examples for this algorithm

* **Dijkstra’s Algorithm**: Used for finding the shortest path in a graph.
* **Prim’s Algorithm**: Used for finding a minimum spanning tree.

**Time complexity for the greedy algorithm :**

* **Different greedy algorithms have different time complexities. For instance, Dijkstra's algorithm has a time complexity of O(V^2) with a simple implementation, and O(E + V log V) with a priority queue.**

**Time complexity for Dijkstra’s algorithm: O(V^2) with an adjacency matrix, O(E + V log V) with a priority**

**Space complexity for Dijkstra’s algorithm is** **O(V + E), where V is the number of vertices and E is the number of edges.**

**Time complexity and space complexity for prims algorithm**

**Space complexity:**

**Storing the graph as an adjacency list requires O(V + E) space. Additional space is needed for the heap and tracking structures.**

**Time complexity:**

**Involves iterating over all vertices (V) and finding the minimum edge, which takes O(V) time for each vertex.**

**Dynamic programming:**

**Dynamic Programming (DP) is an optimization technique used to solve complex problems by breaking them down into simpler subproblems and solving each of these just once, storing their solutions. It is particularly useful for problems with overlapping subproblems and optimal substructure properties.**

1. **Overlapping Subproblems:**
   * **Problems can be divided into subproblems that are reused multiple times.**
2. **Optimal Substructure:**
   * **An optimal solution to the problem can be constructed from optimal solutions to its subproblems.**

**Top-Down (Memoization):**

**Recursive approach that solves subproblems and stores their results to avoid redundant calculations.**

**Bottom-Up (Tabulation):**

**Iterative approach that solves subproblems starting from the smallest and building up to the solution of the main problem.**

**You**

**time and space complexity for dynamic programming and which data structure is used**

**Copilot**

**Time and Space Complexity in Dynamic Programming**

**Time Complexity:**

* **Memoization (Top-Down): Generally O(n) for problems like Fibonacci where n is the input size, because each subproblem is solved once.**
* **Tabulation (Bottom-Up): Similar O(n) complexity, as it iteratively solves all subproblems.**

**Space Complexity:**

* **Memoization: O(n) due to the storage of subproblem results in a cache or dictionary.**
* **Tabulation: Also O(n), as it stores solutions in a table.**

**Common Data Structures Used:**

1. **Arrays/Tables: Used for storing results of subproblems in both memoization and tabulation.**
2. **Hash Maps/Dictionaries: Often used in memoization to store previously computed values for quick lookup.**