|  |
| --- |
|  |
| DSA ASSIGMENT |
| -NANDA KISHORE S |
|  |
| **2117230080069** |
|  |

|  |
| --- |
|  |

***Greedy Algorithms***

**Greedy algorithms solve problems by making a series of locally optimal choices, with the goal of finding a globally optimal solution. They are often used in optimization problems, where the best choice at each step eventually leads to an overall optimal solution.**

***Characteristics of Greedy Algorithms:***

1. **Optimal Substructure: The problem can be broken down into smaller sub-problems that contribute to the overall solution.**
2. **Greedy Choice Property: The best local choice is made at each step with the hope that this leads to the global optimum.**
3. **No Backtracking: Decisions are final, and there’s no revisiting or backtracking on previous choices.**

**Example: Coin Change Problem**

**Find the minimum number of coins required to make a given amount.**

**Available Coins: 1¢, 5¢, 10¢, 25¢  
Target Amount: 36¢**

**Greedy Solution:**

1. **Choose the largest coin until the remaining amount is less than the coin value:**
   * **36¢ - 25¢ = 11¢ (1st coin)**
   * **11¢ - 10¢ = 1¢ (2nd coin)**
   * **1¢ - 1¢ = 0¢ (3rd coin)**
2. **Result: 3 coins (25¢, 10¢, 1¢)**

**Python Code Example**

**def coin\_change(amount, coins):**

**coins.sort(reverse=True)**

**num\_coins = 0**

**for coin in coins:**

**while amount >= coin:**

**amount -= coin**

**num\_coins += 1**

**return num\_coins**

**coins = [1, 5, 10, 25]**

**amount = 36**

**print(coin\_change(amount, coins)) # Output: 3**

**Data Structures Used in Greedy Algorithms**

1. **Huffman Tree: Used in Huffman coding for efficient data compression.**
2. **Knapsack Matrix: Often utilized in the 0/1 Knapsack Problem.**
3. **Coin Change Table: Manages solutions for the coin change problem.**
4. **Activity Selection Table: Helps optimize selection processes in scheduling problems.**

**Applications of Greedy Algorithms**

1. **Resource Allocation: Optimizing resource use, e.g., CPU time.**
2. **Scheduling: Assigning tasks to resources over time.**
3. **Network Routing: Optimizing packet routing to minimize travel time.**
4. **Data Compression: Efficiently encoding information, e.g., Huffman coding.**
5. **Cryptography: Encrypting data through algorithms like RSA.**

**Advantages**

1. **Efficient Computation: Fast and direct approach.**
2. **Simple Implementation: Easy to code and understand.**
3. **Optimal Solution: Often finds the best solution quickly, especially in unconstrained problems.**

**Disadvantages**

1. **Local Optima: May find only a locally optimal solution rather than the global optimum.**
2. **Sensitive to Problem Constraints: Some problems are not well-suited to greedy solutions due to specific constraints.**

**Complexity Analysis**

**Time Complexity**

1. **Best Case: O(n)O(n)O(n), where nnn is the number of inputs.**
2. **Average Case: Typically O(nlog⁡n)O(n \log n)O(nlogn) or O(n2)O(n^2)O(n2).**
3. **Worst Case: Complex problems may require O(2n)O(2^n)O(2n).**

**Space Complexity**

1. **Best Case: O(1)O(1)O(1), for simpler problems.**
2. **Average Case: O(n)O(n)O(n).**
3. **Worst Case: O(n2)O(n^2)O(n2), for complex problem structures.**

**Example Complexities**

1. **Coin Changing Problem:**
   * **Time Complexity: O(n)O(n)O(n), where nnn is the amount.**
   * **Space Complexity: O(1)O(1)O(1) in a simple greedy solution.**
2. **Huffman Coding:**
   * **Time Complexity: O(nlog n)O(n \log n)O(nlogn), where nnn is the number of symbols.**
   * **Space Complexity: O(n)O(n)O(n) for storing the Huffman Tree.**
3. **Knapsack Problem:**
   * **Time Complexity: O(n2)O(n^2)O(n2), where nnn is the number of items.**
   * **Space Complexity: O(n)O(n)O(n) for storing matrix data.**

**Optimizations**

1. **Memoization: Store intermediate results to avoid redundant calculations.**
2. **Dynamic Programming: Useful for breaking down problems into overlapping sub-problems.**
3. **Efficient Data Structures: Selection of optimal structures can significantly reduce time and space complexity.**

**Comparison with Other Approaches**

1. **Dynamic Programming: Greedy algorithms are faster but may not always reach an optimal solution, unlike dynamic programming which considers all possibilities.**
2. **Brute Force: Greedy approaches are more efficient but may miss some potential solutions that brute-force methods explore.**
3. **Backtracking: Greedy methods are quicker but may not handle constraints as effectively as backtracking algorithms.**

**4o**

Dynamic Programming (DP) Algorithm

**Definition:**

**Dynamic Programming is a problem-solving strategy that breaks down complex problems into smaller sub-problems, solving each only once and storing the results to sub-problems to avoid redundant computation**.

**Key Characteristics**:

**1. Optimal Substructure: Problem can be broken into smaller sub-problems.**

**2. Overlapping Sub-Problems: Sub-problems may have some overlap.**

**3. Memoization: Store solutions to sub-problems.**

**Steps:**

**1. Define the Problem: Identify the problem and constraints.**

**2. Break Down: Divide the problem into smaller sub-problems.**

**3. Create a Table: Store solutions to sub-problems.**

**4. Fill the Table: Solve each sub-problem and store the result.**

**5. Combine Solutions: Combine solutions to sub-problems**.

**Types of DP:**

**1. Top-Down: Start with the original problem and break it down.**

**2. Bottom-Up: Start with smallest sub-problems and combine.**

**Examples:**

**1. Fibonacci Series: Calculate the nth Fibonacci number.**

**2. Longest Common Subsequence (LCS): Find the longest common subsequence between two strings.**

**3. Shortest Path Problems: Find the shortest path in a graph.**

**4. Knapsack Problem: Optimize item selection.**

**Advantages**:

**1. Efficient Computation: Avoid redundant calculations.**

**2. Optimal Solution: Guaranteed optimal solution.**

**3. Scalability: Suitable for large problems.**

**Disadvantages:**

**1. Complexity: Difficult to implement.**

**2. Memory Usage: Requires extra memory.**

**DP vs Greedy Algorithm:**

**1. Optimality: DP guarantees optimality, while Greedy Algorithms may not.**

**2. Complexity: DP is more complex.**

**Code (Python)**

**def fibonacci(n):**

**dp = [0] \* (n + 1)**

**dp[1] = 1**

**for i in range(2, n + 1):**

**dp[i] = dp[i - 1] + dp[i - 2]**

**return dp[n]**

**def lcs(str1, str2):**

**m, n = len(str1), len(str2)**

**dp = [[0] \* (n + 1) for \_ in range(m + 1)]**

**for i in range(1, m + 1):**

**for j in range(1, n + 1):**

**if str1[i - 1] == str2[j - 1]:**

**dp[i][j] = dp[i - 1][j - 1] + 1**

**else:**

**dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])**

**return dp[m][n]**

**Time Complexity**

**1. Best-case scenario: O(n), where n is the number of inputs.**

**2. Average-case scenario: O(n^2) or O(n^3), depending on the problem.**

**3. Worst-case scenario: O(2^n) or O(n!), for complex problems.**

**Space Complexity**

**1. Best-case scenario: O(1), for simple problems.**

**2. Average-case scenario: O(n), for most problems.**

**3. Worst-case scenario: O(n^2), for complex problems.**

**Examples**

**1. Fibonacci Series:**

**1. Time complexity: O(n)**

**2. Space complexity: O(1)**

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

**1. Time complexity: O(m \* n)**

**2. Space complexity: O(m \* n)**

**3. Shortest Path Problems:**

**1. Time complexity: O(n^2)**

**2. Space complexity: O(n)**

**4. Knapsack Problem:**

**1. Time complexity: O(n \* capacity)**

**2. Space complexity: O(n \* capacity)**