|  |
| --- |
|  |
| DSA ASSIGMENT |
| -POORANI B |
|  |
|  |
|  |

|  |
| --- |
|  |

Greedy Algorithms

A greedy algorithm solves problems by making locally optimal choices, hoping to find a global optimum solution.

**Characteristics**

1. Optimal Substructure: Break down the problem into smaller sub-problems.

2. Greedy Choice: Choose the locally optimal solution at each step.

3. No Backtracking: Make irreversible decisions.

**Example: Coin Changing Problem**

Problem Statement: Find the minimum number of coins needed to make change for a given amount.

Available Coins: 1¢, 5¢, 10¢, 25¢

Target Amount: 36¢

**Specific Data Structures for Greedy Algorithms**

1. Huffman Tree (for Huffman Coding)

2. Knapsack Matrix (for 0/1 Knapsack Problem)

3. Coin Changing Table (for Coin Changing Problem)

4. Activity Selection Table (for Activity Selection Problem)

**Greedy Solution**

1. Choose the largest coin (25¢) until the remaining amount is less than 25¢.

- 36¢ - 25¢ = 11¢ (1st coin)

- 11¢ - 10¢ = 1¢ (2nd coin)

- 1¢ - 1¢ = 0¢ (3rd coin)

2. Result: 3 coins (25¢, 10¢, 1¢)

**Code (Python)**

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

**Applications**

1. Resource Allocation

2. Scheduling

3. Network Routing

4. Data Compression

5. Cryptography

**Advantages**

1. Efficient Computation: Fast execution.

2. Simple Implementation: Easy to understand and code.

3. Optimal Solution: Often finds the global optimum.

**Disadvantages**

1. Local Optimum: May not always find the global optimum.

2. Sensitive to Constraints: Problem-specific constraints can affect performance

**Real-World Applications**

1. Resource allocation

2. Scheduling

3. Network routing

4. Data compression

5. Cryptography

**Time Complexity**

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

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

3. Worst-case scenario: O(n^2) or O(2^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. Coin Changing Problem:

1. Time complexity: O(n), where n is the amount.

2. Space complexity: O(1), for a simple greedy solution.

2. Huffman Coding:

1. Time complexity: O(n log n), where n is the number of symbols.

2. Space complexity: O(n), for the Huffman tree.

3. Knapsack Problem:

1. Time complexity: O(n^2), where n is the number of items.

2. Space complexity: O(n), for the knapsack matrix.

**Optimizations**

1. Memoization: Store intermediate results to avoid recalculations.

2. Dynamic programming: Break down problems into smaller sub-problems.

3. Efficient data structures: Choose data structures with optimal time and space complexity.

**Comparison with Other Algorithms**

1. Dynamic programming: Greedy algorithms are generally faster but may not always find the optimal solution.

2. Brute force algorithms: Greedy algorithms are more efficient but may not explore all possible solutions.

3. Backtracking algorithms: Greedy algorithms are faster but may not handle constraints as effectively.

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)