WEEK – 9

. When Optimal Substructure Fails

- When a problem does not exhibit the optimal substructure property, it means that a straightforward application of dynamic programming may not be appropriate or may fail to provide the optimal solution. Here are a few scenarios where the optimal substructure property fails:
- **Scenario 1: Overlapping Subproblems**
- Dynamic programming relies on storing solutions to subproblems to avoid redundant computations.
- If there are overlapping subproblems, meaning the same subproblem is solved multiple times, DP can be inefficient.
- In such cases, memoization (top-down DP) or tabulation (bottom-up DP) can still be used, but the efficiency gains may not be significant.

. When Optimal Substructure Fails

- **Scenario 2: Greedy Choice is Not Always Optimal**
- Some problems require making greedy choices at each step, but these local optimal choices do not lead to a global optimal solution.
- Greedy algorithms, which make the best choice at each step, might not work if the problem doesn't have the property that a global optimum can be reached by selecting local optima.
- In such cases, a different algorithmic approach may be needed.

. When Optimal Substructure Fails

- **Scenario 3: Non-Monotonicity**
- Problems where the optimal solution does not necessarily include the optimal solution to subproblems.
- An example is the "Longest Increasing Subsequence" problem, where simply choosing the longest subsequence at each step does not guarantee the overall longest increasing subsequence.
- This violates the optimal substructure because the optimal solution of the problem cannot be directly constructed from the optimal solutions of its subproblems.

. When Optimal Substructure Fails

- **Scenario 4: Dependency on Future Choices**
- Problems where the optimal solution at a particular step depends on future choices, which dynamic programming cannot anticipate.
- An example is the "0-1 Knapsack Problem" with a constraint on the weight of items.
- Choosing an item in the current step may affect the choices available for future steps, making it challenging to use DP directly.
- **Scenario 5: NP-Hard or NP-Complete Problems**
- Problems that are inherently complex, such as "Traveling Salesman Problem" or "Subset Sum Problem".
- These problems do not have efficient solutions in general and dynamic programming might not be a suitable approach due to the exponential time complexity.

Dynamic Programming: Longest Common Subsequence

- Problem Statement:
- Given two sequences of characters, find the length of the longest subsequence present in both sequences.
- Example:
- Sequence 1: "ABCDGH"
- Sequence 2: "AEDFHR"
- Longest Common Subsequence (LCS): "ADH" (length = 3)

- Approach:
- **Dynamic Programming** is used to solve this problem efficiently.
- We create a 2D table to store the lengths of LCS for different prefixes of the two sequences.
- Dynamic Programming Table:
- Let's use the sequences "ABCBDAB" and "BDCAB" for illustration.

- B D C A B
- 0 0 0 0 0 0
- A 0 0 0 0 1 1
- B 0 0 0 0 1 2
- C 0 0 1 1 1 2
- B 0 0 1 1 1 2
- D 0 1 1 1 1 2
- A 0 1 1 1 2 2
- B 0 1 1 1 2 3

- Algorithm:
- **Initialization**: Fill the first row and column of the table with zeros.
- **Build the Table**:
  - If characters match:
    - Increment the value in the diagonal by 1.
  - If characters don't match:
    - Take the maximum of the value above or left.
- **Result**: The bottom-right cell contains the length of the LCS.

- Traceback:
- To find the actual LCS:
  - Start from the bottom-right cell.
  - Move diagonally up whenever the characters match.
  - Follow the path until reaching the top-left cell.
- Complexity:
- Time Complexity: O(mn) where m and n are the lengths of the two sequences.
- Space Complexity: O(mn) for the 2D table.

Introduction to Greedy Algorithm

- What is a Greedy Algorithm?
- A Greedy Algorithm is an algorithmic paradigm that makes a series of locally optimal choices at each step with the hope of finding a global optimum.
- It makes the best choice at each step without regard for the global structure, with the belief that this will lead to the best possible solution.
- Greedy algorithms are quite efficient and straightforward to implement, making them useful in a variety of problems.

Greedy Interval Scheduling

- **Introduction to Greedy Algorithm**
- What is a Greedy Algorithm?
- A Greedy Algorithm is an algorithmic paradigm that makes a series of locally optimal choices at each step with the hope of finding a global optimum.
- It makes the best choice at each step without regard for the global structure, with the belief that this will lead to the best possible solution.
- Greedy algorithms are quite efficient and straightforward to implement, making them useful in a variety of problems.
- **Greedy Interval Scheduling**
- Problem Statement:
- Given a set of tasks with start and finish times, the goal is to find the maximum number of tasks that can be completed without conflicts.
- Greedy Solution:

- **Sort**: First, sort the tasks by their finish times in ascending order.
- **Select**: Iterate through the sorted tasks. At each step, choose the task that finishes earliest and does not conflict with previously selected tasks.
- Example:
- Tasks:
  - Task 1: Start – 1, Finish – 4
  - Task 2: Start – 3, Finish – 5
  - Task 3: Start – 0, Finish – 6
  - Task 4: Start – 5, Finish – 7
  - Task 5: Start – 3, Finish – 8
  - Task 6: Start – 5, Finish – 9

Prefix Codes (Huffman Code)

- What are Prefix Codes?
- Prefix codes are codes with the property that no code word is a prefix of another.
- Huffman coding is a method to construct such prefix codes efficiently.
- Huffman Coding Steps:
- **Frequency Table**: Create a frequency table of characters based on their occurrence in the given data.

- **Build Huffman Tree**:
  - Start with all characters as individual nodes with their frequencies.
  - Combine two nodes with the lowest frequencies into a new node. Repeat until there's only one node left.
- **Assign Codes**:
  - Traverse the Huffman tree:
    - Left edge -> '0'
    - Right edge -> '1'
  - Assign codes by the path from the root to each character.

- Sorted by Finish Time:
  - Task 3, Task 1, Task 2, Task 5, Task 4, Task 6
- Greedy Selection:
  - Select Task 3, Task 5, Task 6 for a total of 3 tasks.
- Explanation:
- Greedy Interval Scheduling works because selecting the task that finishes earliest allows room for more tasks to fit into the schedule without overlap.
- It doesn't always provide the optimal solution, but in this case, it does.

- Example:
- Data: "ABBCCCDDDDEEEEE"
- Frequency Table:
  - A: 1, B: 2, C: 3, D: 4, E: 5

```
   / \
      /   \
     /       \
   E:5    / \
        A:1 B:2
          / \
        C:3 D:4
```

- Codes:
  - A: 00, B: 01, C: 10, D: 11, E: 1
- Explanation:
- Huffman Coding optimally assigns shorter codes to more frequent characters.
- Prefix property ensures no ambiguity in decoding, as no code is a prefix of another.

Summary:

- Greedy algorithms make locally optimal choices with the hope of finding a global optimum.
- Greedy Interval Scheduling selects tasks based on earliest finish times without conflicts.
- Huffman Coding creates prefix codes efficiently for data compression.
- Feel free to use these points and examples to create a PowerPoint presentation on Greedy Algorithms, Greedy Interval Scheduling, and Huffman Coding!