Greedy Algorithms

Greedy Algorithm

A greedy algorithm is a type of algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

It aims to find the optimal solution by making the most advantageous choice at each individual stage.

The key characteristic of greedy algorithms is their greedy choice property.

This property ensures that at each step, the algorithm selects the most immediate, optimal solution without considering the consequences of that choice on future steps.

While this strategy does not always guarantee a globally optimal solution, it often leads to sufficiently good solutions for a wide range of problems.

Steps in Designing Greedy Algorithm

1. **Define the problem:** Clearly understand the problem you're trying to solve and determine if the greedy approach is suitable.
2. **Identify subproblems:** Break down the problem into smaller subproblems or stages where you can make local optimal choices.
3. **Define the objective function:** Clearly define the objective function that needs to be optimized. The objective function guides the algorithm to make decisions that contribute to the overall optimization.
4. **Make greedy choices:** At each step, make the locally optimal choice that contributes to the objective function. This choice is often based on the current state of the problem.
5. **Update the solution:** After making a choice, update the solution and move on to the next stage or subproblem.

Applications - Optimization Problems

- Coin Change
- Knapsack Problem
- Shortest Path - Dijkastra
- Minimum Spanning Tree - Prim, Kruskal
- Activity Selection Problem
- Job Scheduling
- Huffman Coding

Coin Change Problem

Given coin denominations and amount. The problem is to **Find the minimum number of coins** needed to make a certain amount of change using a given set of coin denominations.

Design Coin Change Problem

- **Sort the coin denominations:**
  - Sort the coin denominations in descending order. This step is crucial for the greedy approach to work, as it allows us to always choose the largest coin denomination that is less than or equal to the remaining change.
- **Initialize variables:**
  - Set a variable to represent the total amount of change to be made.
  - Initialize a variable to count the total number of coins used.
- **Greedy approach:**
  - Iterate through the sorted coin denominations.
  - At each step, check if the current coin denomination is less than or equal to the remaining change.
  - If true, subtract the coin value from the remaining change, and increment the coin count.
- **Repeat until the remaining change is zero:**
  - Continue the greedy approach until the remaining change becomes zero.
- **Output the result:**
  - The total number of coins used is the solution to the Coin Change Problem.

Algorithm

```
Greedy_Coin_Change(coins[], amount):
    1. Sort coins[] in descending order.
    2. Initialize total_coins = 0
    3. Initialize remaining_amount = amount
    4. For each coin in coins:
        a. While remaining_amount >= coin:
            i. Subtract coin from remaining_amount
            ii. Increment total_coins
    5. If remaining_amount is 0:
        a. Return total_coins as the minimum number of coins used.
    6. Else:
        a. Return -1 (indicating it is not possible to make exact change).
```

Example

Consider an example of the Coin Change Problem using the greedy approach. Suppose you have the following coin denominations (Rs.):

1. 25, 10, 5, and 1. You want to make change for Rs.63.
2. 25, 20, 10, 5, and 1, and you want to make change for Rs. 40.
3. 10, 7, 5, and 3, and you want to make change for Rs. 18.

Fractional Knapsack Problem

Given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity.

The goal is to determine the maximum value that can be obtained by selecting a fraction of each item to include in the knapsack.

Algorithm

```
Fractional_Knapsack(items[], capacity):
    1. Sort the items
    2. Initialize total_value = 0 and remaining_capacity = capacity.
    3. For each item in sorted items:
        a. If item.weight <= remaining_capacity:
            i. Add the item's value to total_value.
            ii. Reduce remaining_capacity by item.weight.
        b. If the entire item cannot fit:
            i. Calculate the fraction of the item that can fit
                                    (fraction = remaining_capacity / item.weight).
            ii. Add fraction * item.value to total_value.
            iii. Set remaining_capacity to 0, as the knapsack is now full.
            iv. Break the loop.
    4. Return total_value as the maximum value obtainable.
```

Example

Consider 5 items for the Fractional Knapsack Problem. Suppose we have the following items:

Item A: Weight = 2, Value = 10
Item B: Weight = 3, Value = 5
Item C: Weight = 5, Value = 15
Item D: Weight = 7, Value = 7
Item E: Weight = 1, Value = 8

The knapsack has a maximum capacity of 10 units.

Solve

Given (weight, profit) of the items and the knapsack capacity W

1. [(2, 10), (3, 5), (5, 15), (7, 7), (1, 8)], W = 10
2. [(2, 3), (1, 2), (4, 10), (5, 5), (2, 9)], W = 8
3. [(5, 10), (3, 7), (2, 5), (8, 2), (1, 9)] W = 15

Huffman Coding

Huffman Coding is a popular compression algorithm that uses a greedy approach to generate variable-length codes for characters based on their frequencies.

The algorithm constructs a binary tree, known as the Huffman tree, where characters with higher frequencies have shorter codes.

Variable vs Uniform Length Codes

Example

Consider Characters and frequencies: {A: 4, B: 2, C: 1, D: 3}.

Variable length (Huffman) codes: {'A': '0', 'B': '10', 'C': '110', 'D': '111'}

Uniform-Length Codes: {'A': '00', 'B': '01', 'C': '10', 'D': '11'}

message = **'ABACDAD'**

Huffman Encoded Message: **010001100111111**

Uniform-Length Encoded Message: **00010001100111101**

Algorithm

```
Algorithm HuffmanCoding(characters_freq):
    1. Create a priority queue (min-heap) of nodes, each containing a character and its frequency.
    2. Initialize the priority queue with nodes for each character and its frequency.

    3. While there is more than one node in the priority queue:
        a. Remove the two nodes with the lowest frequencies.
        b. Create a new internal node with a frequency equal to the sum of the two nodes' frequencies.
        c. Insert the new internal node back into the priority queue.

    4. The remaining node in the priority queue is the root of the Huffman tree.

    5. Perform a depth-first traversal of the Huffman tree, assigning binary codes to characters:
        a. Assign '0' when moving left in the tree.
        b. Assign '1' when moving right in the tree.

    6. The binary codes assigned to each character represent the Huffman codes.
```

Example

1. **A: 4, B: 2, C: 1, D: 3** & **message = 'ABACDAD'**
   a. Huffman Codes: {'A': '0', 'B': '10', 'C': '110', 'D': '111'}
   b. Huffman Encoded Message: 01000110011111

2. **('A', 15), ('B', 7), ('C', 6), ('D', 6), ('E', 5), ('F', 5), ('G', 4), ('H', 4), ('I', 3), ('J', 1) & message = 'ABBCDIHEAJJFFGG'**
   a. Huffman Codes for 10 characters: {'A': '10', 'B': '01', 'C': '001', 'D': '000', 'E': '1110', 'F': '1111', 'G': '1101', 'H': '1100', 'I': '0000', 'J': '0011'}
   b. Huffman Encoded Message for 10 characters: 100111010001010110111110111001110000000011101111011110111000

Activity Selection Problem

The Activity Selection Problem is a **scheduling problem**

To select a maximum-size set of **non-overlapping activities**, given their start and finish times.

The goal is to find the **maximum** number of activities that can be **scheduled without any conflicts.**

Algorithm

ActivitySelection(activities):

1. Sort the activities based on their finish times in ascending order.
2. Select the first activity and add it to the schedule.
3. For each remaining activity in the sorted list:
    a. If the start time of the current activity is greater than or equal to the finish time of the previously selected activity, select the current activity and add it to the schedule.
4. The selected activities form the maximum-size set of non-overlapping activities.

Example

1. Activities = [(1, 4), (3, 5), (0, 6), (5, 7), (8, 9), (5, 9)]
    a. Selected Activities: [(1, 4), (5, 7), (8, 9)]

2. Activities = [(1, 4), (3, 5), (0, 6), (5, 7), (8, 9), (5, 9), (2, 3), (6, 8), (9, 10), (4, 8)]
    a. Selected Activities: [(2, 3), (3, 5), (5, 7), (8, 9), (9, 10)]

3. Activities = [(1, 3), (2, 5), (5, 8), (4, 6), (7, 9), (8, 10), (11, 13), (12, 14), (14, 15), (15, 17)]
    a. Selected Activities: [(1, 3), (4, 6), (7, 9), (11, 13), (14, 15)]