1.DESCRIBE GREEDY ALGORITHM WITH EXAMPLE

- ## Greedy Algorithm :

- A greedy algorithm is a simple, intuitive algorithmic technique used for solving optimization problems. It makes the locally optimal choice at each stage with the hope of finding a global optimum solution.

- ## Example:

- Coin Changing Problem Statement:
- Given a set of coins with denominations {1, 5, 10, 25}, find the minimum number of coins needed to make change for a given amount.

Greedy Algorithm Solution (Python):

```
def coin_change(amount, coins):
    coins.sort(reverse=True)  # Sort coins in descending order
    num_coins = 0
    i = 0
    while amount > 0:
        if coins[i] <= amount:
            amount -= coins[i]
            num_coins += 1
        else:
            i += 1
    return num_coins
# Example usage:
coins = [1, 5, 10, 25]
amount = 37
```

```
print("Minimum number of coins:", coin_change(amount, coins))
```

Output:

```
Minimum number of coins: 4
```

Advantages:

- Efficient (O(n) time complexity)

- Simple to implement

Disadvantages:

- May not always find the global optimum solution

- Requires careful selection of greedy choice

2.For all those algorithms, what is the time complexity, space complexity, what DS is in use

1. Coin Change Problem:

- **Problem**: Given a set of coin denominations and an amount, find the minimum number of coins needed to make that amount.
- **Greedy Approach**: Always pick the largest coin possible. For example, to make ₹49 using denominations {₹1, ₹5, ₹10, ₹20}, you would pick ₹20 twice, then ₹5 once, and finally ₹4 coins of ₹1.

Time complexity :

Sorting the Denominations: O(n logn)

Selecting coins: O(n)

Overall time complexity: O(nlogn+n)=O(nlogn)

If coins are already sorted: O(n)

Space complexity:

To store the list of n coin denominations : O(n)

2.Huffman coding:

- **Problem**: Huffman coding is a **greedy algorithm** used to compress data. It creates a binary tree where characters with lower frequency get longer codes, and those with higher frequency get shorter codes.
- **Greedy Approach**: Build the encoding by combining the least frequent characters into a tree until you have a single tree representing all characters.

Time complexity :

Building a priority queue (min-heap) takes *O(nlogn)*

Merging trees during the construction of the Huffman tree also takes *O(nlogn)*

Space complexity:

To store the tree nodes and frequency table : O(n)

3.Egyptian Fraction Problem:

- **Problem**: It represents a fraction as a sum of distinct unit fractions (fractions with a numerator of 1). The **greedy approach** repeatedly subtracts the largest possible unit fraction until the original fraction is reduced to zero.
- **Greedy Approach**: For a fraction like 4/13, repeatedly subtract the largest possible unit fraction until the remainder is zero.

Time complexity : It depends on how many iterations are required to reduce the fraction to zero. In most cases, the number of steps grows as the size of the numerator and denominator increase, so the complexity is *O(n)*, where O(n) is the size of the denominator.

Space complexity: Number of distinct unit fractions in the result : O(k)

4.Greedy Coloring Algorithm:

- **Problem**: Assign the fewest colors to the vertices of a graph so that no two adjacent vertices share the same color.

Greedy Approach: Color each vertex one by one, assigning the smallest possible color that hasn't been used by its adjacent vertices.

Time complexity :

For each vertex, the algorithm checks the colors of all its adjacent vertices. If there are V *vertices and E* edges:

Checking adjacent vertices for each vertex takes :*O(V+E)*

Therefore, the overall time complexity is O(V^2) in the worst case

Space complexity :

To store the color assignment for each vertex and possibly an array to track used colors : O(V)

5.Djikstra's Algorithm:

- **Problem**: Find the shortest path from a source node to all other nodes in a graph with non-negative edge weights.
- **Greedy Approach**: Start from the source node and repeatedly choose the closest node that hasn't been processed, updating the shortest path for neighboring nodes.

Time complexity :

Priority queue operations (insert, update, extract-min) take *O(logV),*where V *is the number of vertices.*

The algorithm processes every vertex once and checks each edge, so:

In total, the time complexity is O((V+E)log V) , V-> vertices and E-> Edges

Space complexity :

To storage for the adjacency list of the graph, the distance table, and the priority queue : O(V+E)