CSE431 HW3

Siqi Cheng, 50388579

**Problem 1:**

I was recently planning a driving route from Buffalo to Los Angeles and I chose to use the shortest path algorithm from graph theory. This problem can be translated into the problem of finding the shortest path from a start point to a goal point in a graph. Where the vertices of the graph represent the cities, the edges represent the roads, and the weights of the edges represent the distance or driving time from one city to another.

G(V, E), where V is the set of cities and E is the set of roads connecting the cities. Each edge e has an associated cost c[e], which is a function of both distance and traffic between two cities.

Start at the city of Buffalo.

While we are not in Los Angeles:

For each neighboring city v, calculate the travel cost to v considering traffic. This cost can be represented as a function of distance and current traffic conditions. Choose the city with the lowest cost and move there. Mark the chosen city as visited to avoid going in circles. Repeat until we reach Los Angeles.

```
Function greedy(G, start, end):
    Input: G(V, E), a weighted graph where V is cities and E is roads.
          start is the starting city and end is the destination city.
    visited = set()  // Keeps track of visited cities
    current_city = start
    path = [start]
    While current_city is not end:
        visited.add(current_city)
        next_city = None
        min_cost = infinity
        For each city v connected to current_city:
            If v is not in visited:
                cost = calculate_cost(current_city, v)
                If cost < min_cost:
```

```
                min_cost = cost
                next_city = v
        current_city = next_city
        path.append(next_city)
    return path
```

In the worst case, the algorithm visits all cities once. Therefore, the time complexity is O(V), where V is the number of cities. The space complexity is O(V) as well, since we need to store the visited cities and the path.

**Problem2:**

```
function subword (s, sub):
    n = length(s)
    m = length(sub)
    initialize dp[n+1][m+1] to 0
    for i from 0 to n:
        dp[i][0] = 1
    for i from 1 to n:
        for j from 1 to m:
            if s[i-1] == sub[j-1]:
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
            else:
                dp[i][j] = dp[i-1][j]
    return dp[n][m]
```

I define a DP table, dp[i][j], where dp[i][j] is the number of occurrences of the first j characters of sub in the first i characters of s. If s[i-1] == sub[j-1], then dp[i][j] = dp[i-1][j-1] + dp[i-1][j]. Otherwise, dp[i][j] = dp[i-1][j]. If the ith character in the binary string is the same as the j-th character in the subword, we can either include it in count (dp[i-1][j-1]) or exclude it (dp[i-1][j]). If the current character doesn't match, so only have the option to exclude it. until it computed dp[n][m].

The time complexity of the algorithm is O(nm) because we need to fill a DP table of size n+1 by m+1 and each cell can be filled in constant time. The space complexity is also O(nm) because the DP table takes up space.

**Problem 3**

Given a n × n 2D grid of positive integer numbers. You place yourself at the top left corner. The bottom right corner is your destination. You can only move right or down. Your score is the sum of the numbers you get along the way. Design an algorithm to compute the path you take that maximizes this score.

The solve method of this Dynamic Programming algorithm is to iteratively compute the maximum path sum to each cell in the grid.

It starts from the top left corner and can only move down or right. At any cell, the maximum path sum is the cell's value plus the maximum of the path sums to the cell above and to the left of it.

Initialize:

**n** is the number of rows (or columns) in the grid. dp is an array of size n, initialized to 0, that stores the maximum sum of paths for each cell in the current row of the grid. It is updated and replaced for each row of the grid.

Firstly we at the top left corner, so the maximum path sum to it is just its value:

dp[0] = grid[0][0]

And then we traverse the rest of the dp table and compute the maximum score for each cell.

Consider the cell at position (i,j). We can reach this cell from the cell above it (i-1, j) or from the cell to its left (i, j-1). Therefore, to maximize the score at (i,j), we should start from the cell with the highest score. We denote this as dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + grid[i][j].

This is repeated for each cell in the dp table. The final dp table is obtained with each cell having the maximum score that can be obtained when that cell is reached.

After the dp table has been computed, the cell in the lower right corner (dp[n-1][n-1]) will receive the highest score because it represents the end of all possible paths from the starting point.

Finally, we return the maximum score, which is the value of the destination cell in the dp table, and the path, which is the list of cells we passed through during the backtracking process. The paths are in reverse order, starting at the destination and ending at the start. We reverse the list to get the correct order of the paths from the start to the destination.

```
function maxScore(grid)
    Let n = length of grid
    dp = new n x n array filled with 0
    Create an array dp of size n initialized to 0
    // Start at the top left corner
       dp[0] = grid[0][0]
    for i from 1 to n-1
            for j from 1 to n-1
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + grid[i][j]
    max_score = dp[n-1][n-1]  //max score already find
    //Flowing is finding optimum path
    i = n-1
       j = n-1
       while i > 0 or j > 0:
            if i > 0 and j > 0:
              if dp[i-1][j] > dp[i][j-1]:
                 i = i - 1
              else:
                 j = j - 1
           elif i > 0:
              i = i - 1
           else:
              j = j – 1
     return max_score path
```