W10 - Data Structure & Algorithms

Problem 1: Find the Shortest Path

Overview: Implement a program to find the shortest path from the first city (index 0) to the last city (index n - 1) given a list of direct distances between cities in a 2D array. If there is no direct path between two cities, the distance is represented as zero.

Algorithm Steps:

1. **Graph Representation**:
    - Represent the cities and distances using a graph where the nodes are the cities, and the edges are the distances between the cities.
2. **Dijkstra's Algorithm**:
    - Use Dijkstra's algorithm to find the shortest path from the first city to the last city. This algorithm is efficient for graphs with non-negative edge weights.
3. **Priority Queue for Efficient Searching**:
    - Use a priority queue to efficiently select the next city with the shortest distance from the start city.
4. **Handling Edge Cases**:
    - Ensure that the graph is properly constructed and that distances are non-negative.
5. **Code**

```
import java.util.Arrays;
import java.util.PriorityQueue;

public class ShortestPathCalculator {
   // Function to find the shortest path from the first city to the last city
   public static int shortestPath(int[][] distances) {
       int n = distances.length;
       int[] dist = new int[n];
       Arrays.fill(dist, Integer.MAX_VALUE);
       dist[0] = 0;

       PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> dist[a] - dist[b]);
       pq.offer(0);

       while (!pq.isEmpty()) {
           int city = pq.poll();

           for (int i = 0; i < n; i++) {
               if (distances[city][i] > 0 && dist[i] > dist[city] + distances[city][i]) {
                   dist[i] = dist[city] + distances[city][i];
                   pq.offer(i);
               }
           }
       }
       return dist[n - 1];
   }

   // Main method to test the shortestPath function
   public static void main(String[] args) {
       int[][] distances = {
           {0, 3, 2, 0},
           {3, 0, 0, 5},
           {2, 0, 0, 9},
           {0, 5, 9, 0}
       };
       System.out.println("The shortest path's length from city 0 to city " + (distances.length - 1) + " is: " -
```

```
    }
}
```

Problem 2, 3

Problem 2: Minimum Road System for Castles

Overview: Determine the minimum total length of a road system required to connect all castles. Given a 2D array representing the direct distances between each pair of castles, find the shortest total road length that connects all castles.

Input: A 2D array `castles`, where `castles[i][j]` represents the distance between castle `i` and castle `j`.

Goal: Build the shortest possible road system that connects all castles.

Approach:

- Use a Minimum Spanning Tree (MST) algorithm, like Prim's or Kruskal's algorithm, to find the minimum road length.
- The MST algorithm will select the shortest roads that connect all castles without forming any cycles.

Algorithm Steps:

1. **Initialize**:
    - Represent the castles and distances using a graph.
    - Choose an arbitrary starting castle.
2. **Apply MST Algorithm**:
    - For each castle, select the road with the minimum distance that does not form a cycle.
    - Keep adding the shortest roads until all castles are connected.
3. **Calculate Total Road Length**:
    - Sum up the lengths of all roads selected by the MST algorithm.
4. **Code**:

```
import java.util.Arrays;

public class CastleRoadSystem {

   // Function to find the minimum total length of the road system
   public static int minTotalRoadLength(int[][] castles) {
       int n = castles.length;
       int[] key = new int[n]; // Key values used to pick minimum weight edge in cut
       boolean[] mstSet = new boolean[n]; // To represent set of vertices included in MST
       Arrays.fill(key, Integer.MAX_VALUE); // Initialize all keys as infinite

       key[0] = 0; // Make key 0 so that this vertex is picked as first vertex
       int totalLength = 0; // Store the total length of roads in MST

       for (int count = 0; count < n - 1; count++) {
           int u = minKey(key, mstSet); // Pick the minimum key vertex from the set of vertices not yet included
           mstSet[u] = true; // Add the picked vertex to the MST set

           for (int v = 0; v < n; v++) {
               // Update the key only if castles[u][v] is smaller than key[v]
               if (!mstSet[v] && castles[u][v] != 0 && castles[u][v] < key[v]) {
                   key[v] = castles[u][v];
               }
           }
       }

       // Calculate total weight of MST
```

```
        for (int i = 0; i < n; i++) {
            if (key[i] != Integer.MAX_VALUE) {
                totalLength += key[i];
            }
        }

        return totalLength;
    }

    // Utility function to find the vertex with minimum key value
    private static int minKey(int[] key, boolean[] mstSet) {
        int min = Integer.MAX_VALUE, minIndex = -1;

        for (int v = 0; v < key.length; v++) {
            if (!mstSet[v] && key[v] < min) {
                min = key[v];
                minIndex = v;
            }
        }

        return minIndex;
    }

    public static void main(String[] args) {
        int[][] castles = {
            {0, 1, 2, 8},
            {1, 0, 3, 5},
            {2, 3, 0, 4},
            {8, 5, 4, 0}
        };

        int totalLength = minTotalRoadLength(castles);
        System.out.println("Minimum Total Road Length: " + totalLength);
    }
}
```

Problem 3: Longest Increasing Subsequence of Items

Overview: Find the longest subsequence of distinct items in increasing order. Given a list of distinct items, identify the longest subsequence where each item is greater than the previous one.

Input: A list of distinct items, e.g., `[5, 2, 3, 9, 6, 7, 8]`.

Goal: Find the longest increasing subsequence of items.

Approach:

- Use dynamic programming to build an array that stores the length of the longest increasing subsequence ending at each index.
- For each item in the list, determine the length of the longest subsequence it can form with previous items.

Algorithm Steps:

1. **Initialize**:
    - Create an array `dp` of the same length as the input list. Each element of `dp` represents the length of the longest increasing subsequence ending at that index.
    - Initialize all elements of `dp` to 1, as the minimum length of any subsequence is 1.
2. **Build the DP Array**:
    - Iterate through the list, for each item `item[i]`, compare it with previous items `item[j]` (where `j < i`).
    - If `item[i] > item[j]`, update `dp[i]` to `max(dp[i], dp[j] + 1)`.
3. **Find the Longest Subsequence**:

    - The length of the longest increasing subsequence is the maximum value in the `dp` array.

4. **Code**:

```
public class LongestIncreasingSubsequence {
   public static int findLongestSubsequenceLength(int[] items) {
       int n = items.length;
       int[] dp = new int[n];
       int maxLength = 1;

       for (int i = 0; i < n; i++) {
           dp[i] = 1; // Initialize all lengths to 1
           for (int j = 0; j < i; j++) {
               if (items[i] > items[j]) {
                   dp[i] = Math.max(dp[i], dp[j] + 1);
               }
           }
           maxLength = Math.max(maxLength, dp[i]);
       }
       return maxLength;
   }

   public static void main(String[] args) {
       int[] items = {5, 2, 3, 9, 6, 7, 8};
       int length = findLongestSubsequenceLength(items);
       System.out.println("Length of the longest increasing subsequence: " + length);
   }
}
```

Problem4 : Unique Paths for a Robot in a Grid

Overview: Calculate the number of unique paths a robot can take to reach the lower-right corner of a rectangle field from the upper-left corner. The robot can only move RIGHT or DOWN.

Input: Dimensions of the rectangle field `[R, C]`, where `R` is the number of rows and `C` is the number of columns.

Goal: Find the total number of unique paths from the cell `(0, 0)` to the cell `(R-1, C-1)`.

Approach:

- Use dynamic programming to build a 2D array where each cell represents the number of ways to reach that cell.
- The robot can only move RIGHT or DOWN, so the number of ways to reach a cell is the sum of the ways to reach the cell to its left and the cell above it.

Algorithm Steps:

1. **Initialize**:
    - Create a 2D array `dp` with dimensions `[R][C]`.
    - Set all cells in the first row and first column to 1 since there is only one way to reach these cells (either all the way right or all the way down).
2. **Fill the DP Array**:
    - Iterate through the grid starting from cell `(1, 1)`.
    - Update each cell with the sum of the cell directly above and the cell to the left: `dp[i][j] = dp[i-1][j] + dp[i][j-1]`.
3. **Find the Total Paths**:
    - The value in the cell `(R-1, C-1)` gives the total number of unique paths.
4. **Code**:

```
public class RobotPaths {

   public static int uniquePaths(int R, int C) {
       int[][] dp = new int[R][C];
```

```
        // Initialize first row and first column
        for (int i = 0; i < R; i++) {
            dp[i][0] = 1;
        }
        for (int j = 0; j < C; j++) {
            dp[0][j] = 1;
        }

        // Fill the rest of the dp array
        for (int i = 1; i < R; i++) {
            for (int j = 1; j < C; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }

        return dp[R - 1][C - 1];
    }

    public static void main(String[] args) {
        int R = 3, C = 3; // Example dimensions
        int totalPaths = uniquePaths(R, C);
        System.out.println("Total unique paths: " + totalPaths);
    }
}
```