DS & Algorithms Final Project
Problem 4 Report

1) Problem Statement

There is an undirected connected graph consisting of n nodes. All the nodes have distinct numbers from 1 to n. There are no self-loops in the graph.
The cost of removing an edge is |node1−node2| where node1 and node2 are the vertex numbers of node1 and node2 respectively, and the edge being removed has node1 and node2
as its endpoints.
By performing the above-mentioned operation as many times as wanted, it is required to convert the given graph into a single tree by using minimum possible cost. The output should be the minimum cost required to achieve that task.

2) Algorithm Used & Program's Pseudo Code

I have implemented **Kruskal's Algorithm** in order to convert the graph to the required minimum spanning tree, and have used the **Quicksort Algorithm** to sort the edges descendingly according to the cost of removing the edge.

Program's Pseudo Code:

1) Read number of nodes "n" and number of edges "m".
2) Create a list of size "n" containing a list of each node. "forest"
3) Read "m" edges, creating an object of type "Edge" (a user defined class with the following data fields: "from", "to", and "cost") for each one, and storing them in an array of size m. "edges"
4) Sort "edges" descendingly according to "cost" using quickSort algorithm.
5) Create an empty list that will represent the edges of the MST. "safeEdges"
6) While "forest" has more than one list do the following:
   1- Get "edge" from "edges".
   2- If "from" and "to" belong to different lists in "forest" do the following:
      1- Add "edge" to "safeEdges".
      2- Merge the lists containing "to" and "from" in "forest".
7) Get "totalMinCost" by summing the costs of the edges that are in "edges" but not in "safeEdges".

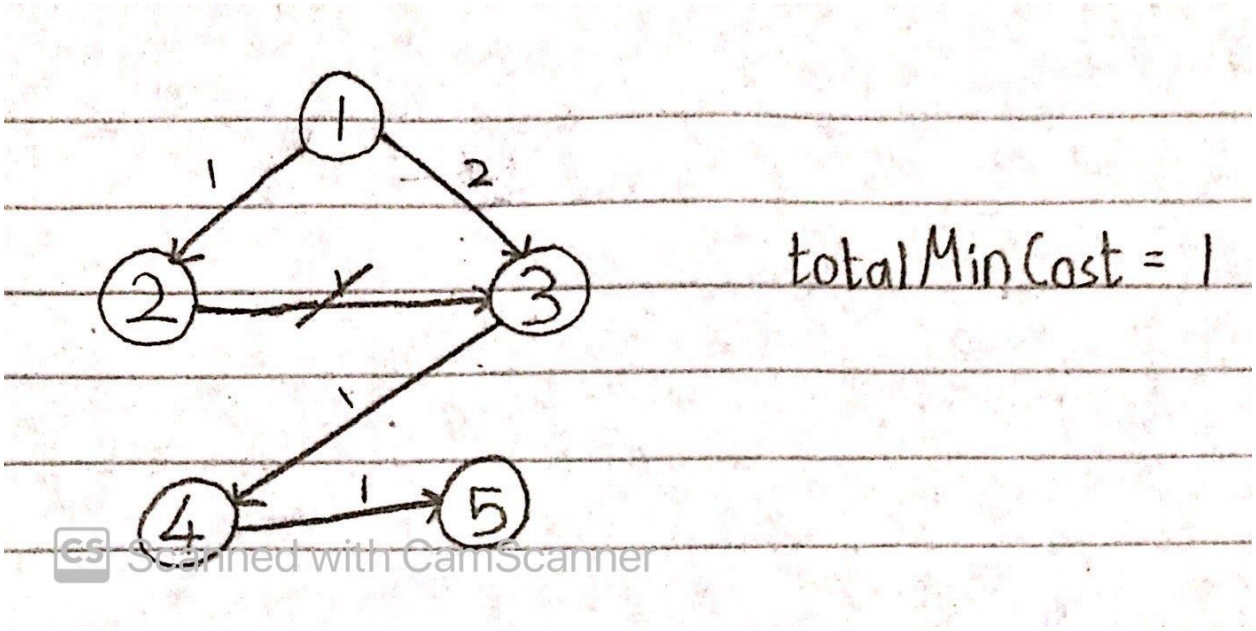
8) Print “totalMInCost”.

3) **Time & Memory Analysis**

Since the sample input given in the project’s file represents a sparse graph, I have chosen Kruskal’s algorithm instead of Prim’s algorithm to generate the MST due to the fact that Kruskal’s algorithm performs better in situations where the graph is sparse because it uses simpler data structures and therefore saving memory, and leading to a running time of O(|E| log |E|), where “|E|” represents the number of edges.
For sorting the edges I have used the quicksort algorithm due to it being the fastest sorting algorithm (time complexity: O(|E| log|E|)), as well as being an in-place sorting algorithm and therefore saving a lot of memory.

4) **Sample Runs**

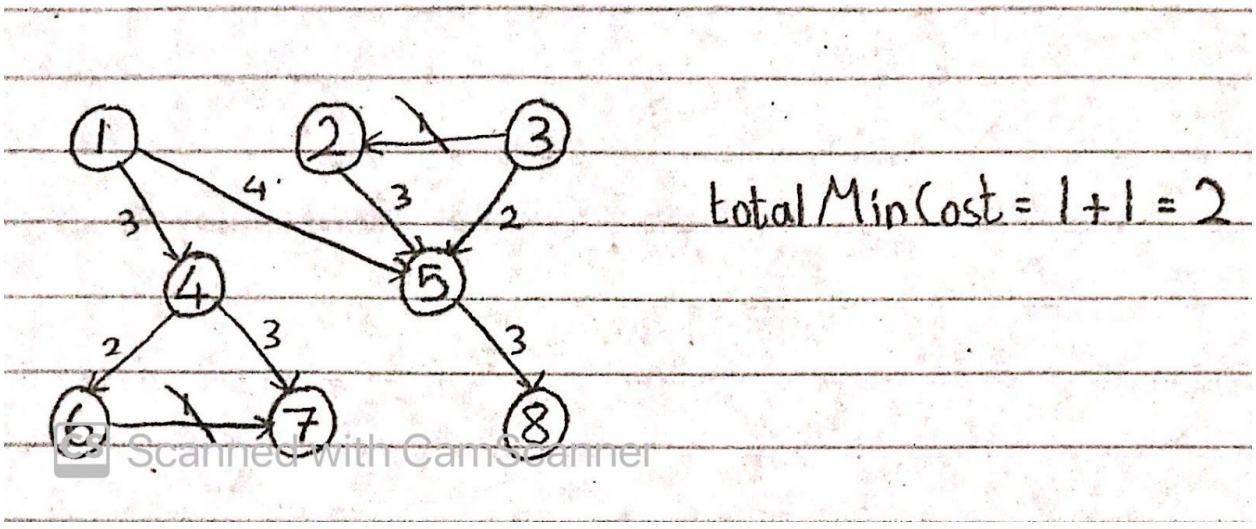
1)



```
5 5
1 2
2 3
3 4
4 5
1 3
1

Process finished with exit code 0
```

2)



```
8 9
1 4
1 5
2 5
3 2
3 5
4 6
4 7
5 8
6 7
2

Process finished with exit code 0
```