MICROSOFT PREPARATION

DAY : 15

Minimum Cost Path :

Problem Link :
https://practice.geeksforgeeks.org/problems/minimum-cost-path3833/1?page=1&company%5B%5D=Microsoft&category%5B%5D=Graph&sortBy=submissions

Test Cases Passed : 90 / 90

Time Used : 35.10

Difficulty Level : HARD

Approach Used :

- Calculating the dimensions of the grid
- Creating a distance vector and initialize with infinity
- Create a set to store distance , [row, col]
- Mark the source as distance with grid distance
- Insert the source with grid distance into the set
- Traverse until the set becomes empty :
    - Extract the first element from the set
    - Check if we have reached the last element :
        - Return the distance to reach the last element
    - Traverse for the adjacent elements of the node :
        - Check for the validity of the indexes :
            - Check if we can reach the adjacent element with a better distance :
                - Update the distance in the distance vector
                - Push the updated distance and adjacent indexes in the set
- Return -1 // no need unreachable code piece

Solution :

```
int minimumCostPath(vector<vector<int>>& grid)
    {
        // We will be using Dijkstra to find the distance of every element
from the source element and return the last element stored
        // getting the dimensions of the grid
        int n = grid.size();
        int m = grid[0].size();
        // creating a distance vector to store the distances from the
source
        vector<vector<int>> distance(n,vector<int>(m,1e9));
        // marking the distance of the first element as distance
        distance[0][0] = grid[0][0];
        // creating a set to store the distance, row, col
        set<pair<int, pair<int, int>>> s;
        // inserting the first element with distance grid into the set
        s.insert({distance[0][0],{0,0}});
        // traversing until the set becomes empty
        while(!s.empty())
        {
            // extracting the first element from the set
            auto it = *(s.begin());
            // popping the first element of the set
            s.erase(it);
            // getting the element from set
            int dist = it.first;
            int row = it.second.first;
            int col = it.second.second;

            // if we reached the last node then return the distance to
reach
            if(row==n-1 && col==m-1)
            {
                return dist;
            }
            // traversing for the adjacent elements
            int delRow[] = {-1,0,1,0};
            int delCol[] = {0,1,0,-1};

            for(int i=0;i<4;i++)
            {
                // calculating the indexes of adjacent elements
                int nrow = row+delRow[i];
```

```
                int ncol = col+delCol[i];

                // checking for validity of dimensions
                if(nrow<n && ncol<m && nrow>=0 && ncol>=0)
                {
                    // checking if we can reach the node with a better
distance
                    if(distance[nrow][ncol]>dist+grid[nrow][ncol])
                    {
                        // updating the distance
                        distance[nrow][ncol] = dist+grid[nrow][ncol];
                        s.insert({distance[nrow][ncol],{nrow,ncol}});
                    }
                }
            }
        }
        // unreachable
        return -1;
    }
```