Question:

Maze game is a well-known problem, where we are given a grid of 0's and 1's, 0's corresponds to a place that can be traversed, and 1 corresponds to a place that cannot be traversed (i.e. a wall or barrier); the problem is to find a path from bottom left corner of grid to top right corner; immediate right, immediate left, immediate up and immediate down only are possible (no diagonal moves). We consider a variant of the maze problem where a cost (positive value) or profit (negative value) is attached to visiting each location in the maze, and the problem is to find a path of least cost through the maze.

You may solve the problem after imposing/relaxing other restrictions on the above problem on

- Values of cost/profit (but not same cost/profit for all traversable cells in the grid)
- Moves possible (but you cannot trivialize the problem by making the grid linear or partly linear in any way)
- No. of destinations possible

Choose the most efficient algorithm possible for your specific case.

Hints: Convert the maze to a weighted graph G (V, E). Each location (i, j) in the maze corresponds to a node in the graph. The problem can have multiple solutions. Students can use any design technique such as greedy method, backtracking, dynamic programming. Students can choose their own conditions, positive or negative costs for the graph.

AIM:-To reach the destination in a maze (minimum cost) and moving in 4 possible directions.

Approach:-back tracking

Given a grid of n*n filled with either 0(safe) or 1(un-safe) and a cost matrix with cost to move to respective square in the grid we use 4 recursive calls (up,down,left,right) with two base conditions

1-if the path go's out of bound or it tries to revisit the visited cell or if the cell has 1 which is a wall we simply return the function

2-If the path reaches the destination then we push the path and the cost into a data structure and simply return the function

Once the path is completed we make the visited vertices un-visited

This recursive approach helps us to find the minimum cost of the path from source to destination

Algorithm:-DFS

- Algorithm:
- Create a recursive function that takes the index of node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

SOURCE CODE:-

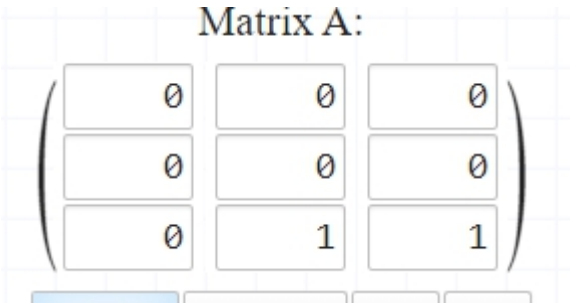
```
#include<bits/stdc++.h>

using namespace std;


  void find_path(int i,int j,vector<vector<int>> &m,string s,int
n,vector<string>&res,vector<vector<int>> &cost,vector<int> &ktk,int sum,
vector<pair<string,int>> &vect)

    {     //BASE CASE TO CHECK IF THE PATH GO'S OUT OF BOUND OR TO CHECK IF IT IS
VISITED      OR IF THERE IS A WALL

          if(i<0 || j<0 || i>=n || j>=n )

          {

                return;

          }

          if( m[i][j]==1 || m[i][j]==-1)

          return;

          //ANOTHER BASE     CONDITION IF REACHES THE DESTINATION WE SIMPLY PUSH
THE COST OD THE PATH INTO THE VECTOR

          else if(i==0 && j==n-1)

          {
```

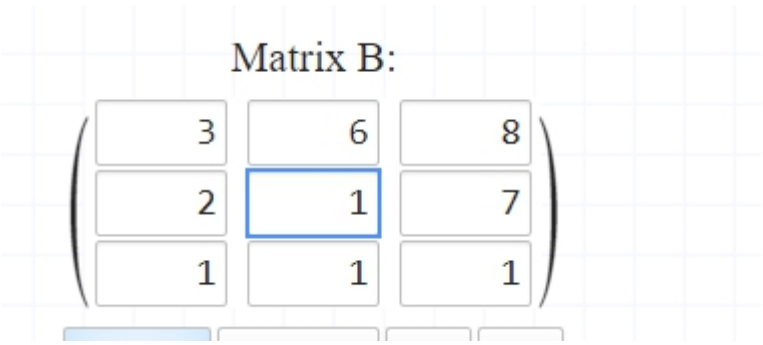
```
        res.push_back(s);
        ktk.push_back(sum);
        vect.push_back( make_pair(s,sum));
        return ;
      }
      //TRAVERSING IN ALL POSSIBLE PATH USING 4 RECURSIVE FUNCTIONS
      else
      {
            m[i][j]=-1;
            find_path(i+1,j,m,s+'D',n,res,cost,ktk,sum+cost[i][j],vect);
            find_path(i-1,j,m,s+'U',n,res,cost,ktk,sum+cost[i][j],vect);
            find_path(i,j+1,m,s+'R',n,res,cost,ktk,sum+cost[i][j],vect);
            find_path(i,j-1,m,s+'L',n,res,cost,ktk,sum+cost[i][j],vect);
            m[i][j]=0;
      }
}
  void findPath(vector<vector<int>> &m, int n,vector<vector<int>> &cost)
  {
    //res vector stores the path from source to destination
    vector<string>res;
    vector<int>ktk;
    vector<pair<string,int>>vect;
    //ktk vector stores the cost of the path
    int sum=0;
   //SUM TO KEEP A TRACK OF THE SUM FOR RESPECTIVE PATH
    find_path(n-1,0,m,"",n,res,cost,ktk,sum,vect);
    if(ktk.size()==0)
    {
    cout<<"NO PATH";
    }
    else
```

```
        {
        sort(ktk.begin(),ktk.end());
        cout<<ktk[0];


  for (auto i : vect )
  {
        cout<<i.first<<" "<<i.second;
  }
        }
       }
int main()
{
  //ENTER THE LENGTH OF THE MATRIX
  int n;
  cin>>n;
  //ENTER THE 1,0 MATRIX
  vector<vector<int>>m(n,vector<int>(n,0));
  for(int i=0;i<n;i++)
  {
        for(int j=0;j<n;j++)
        {
              cin>>m[i][j];
        }
  }
  //ENTER THE COST OF 1,0 MATRIX
  vector<vector<int>>cost(n,vector<int>(n,0));
  for(int i=0;i<n;i++)
  {
        for(int j=0;j<n;j++)
        {
```

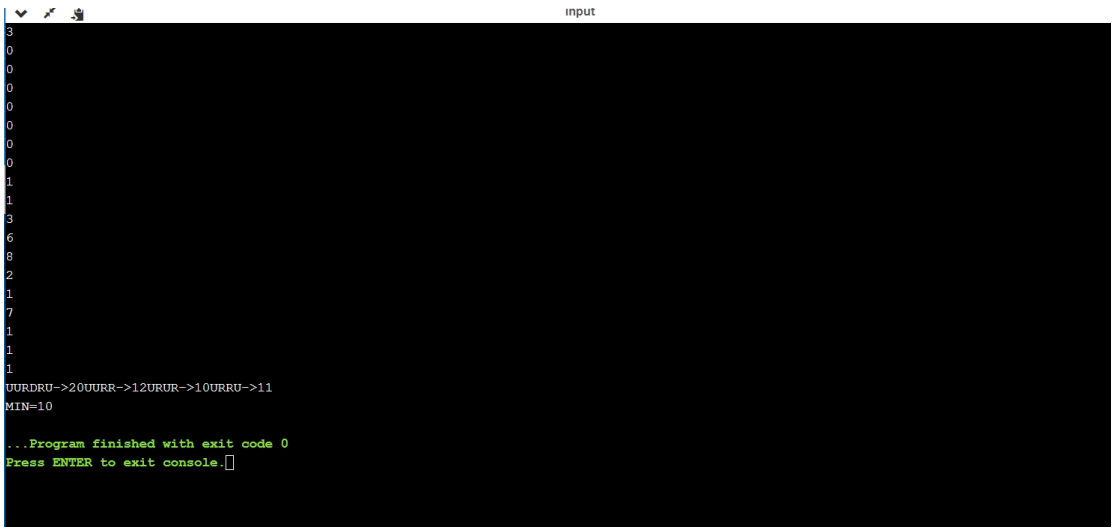
```
            cin>>cost[i][j];
        }
    }
    findPath(m,n,cost);

    return 0;
}
```

TC-1





Output



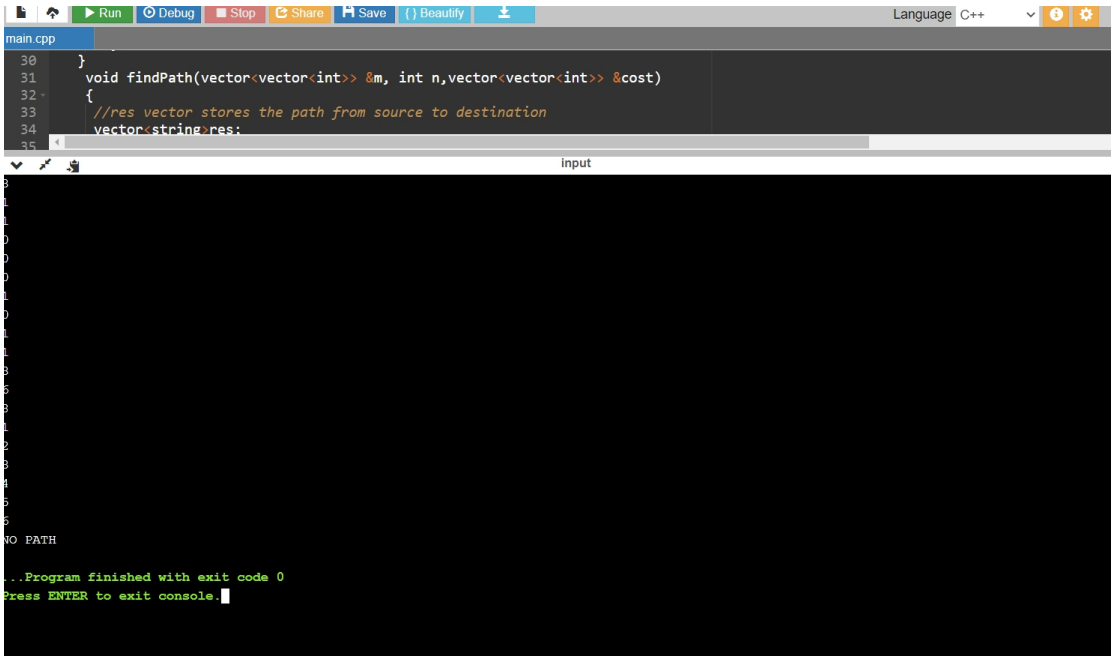
Tc-2

Matrix A:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Matrix B:

$$\begin{pmatrix} 3 & 6 & 8 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



Time complexity:-Time Complexity: O(3^(n^2)).

As there are N^2 cells from each cell there are 3 unvisited neighbouring cells. So the time complexity O(3^(N^2)