Problem Statement

You are given an m x n grid. Each cell of the grid has a direction pointing to the next cell you should visit if you are currently in this cell. The direction in grid[i][j] can be:

- 1 which means go to the cell to the right. (i.e go from grid[i][j] to grid[i][j + 1])
- 2 which means go to the cell to the left. (i.e go from grid[i][j] to grid[i][j - 1])
- 3 which means go to the lower cell. (i.e go from grid[i][j] to grid[i + 1][j])
- 4 which means go to the upper cell. (i.e go from grid[i][j] to grid[i - 1][j])

Notice that there could be some signs on the cells of the grid that point outside the grid.

You will initially start at the upper left cell (0, 0). A valid path in the grid is a path that starts from the upper left cell (0, 0) and ends at the bottom-right cell (m - 1, n - 1) following the signs on the grid. The valid path does not have to be the shortest.

You can modify the sign on a cell with cost = 1. You can modify the sign on a cell one time only.

Return the minimum cost to make the grid have at least one valid path.

Input Format

The first line contains two integers n and m, where m is the columns of the mat matrix and n is the process of the mat matrix.

The second line contains the input of the matrix.

Output Format

Return the minimum cost to make the grid have at least one valid path.

Constraints

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 100
- 1 <= grid[i][j] <= 4

Sample Testcase 0

Testcase Input

```
4 4
1 1 1 1
2 2 2 2
1 1 1 1
2 2 2 2
```

Testcase Output

```
3
```

Explanation

You will start at point (0, 0).
The path to (3, 3) is as follows. (0, 0) --> (0, 1) --> (0, 2) --> (0, 3) change the arrow to down with cost = 1 --> (1, 3) --> (1, 2) --> (1, 1) --> (1, 0) change the arrow to down with cost = 1 --> (2, 0) --> (2, 1) --> (2, 2) --> (2, 3) change the arrow to down with cost = 1 --> (3, 3)
The total cost = 3.

Sample Testcase 1

Testcase Input

```
2 2
1 2
2 4
```

Testcase Output

```
1
```

CODE

```cpp
#include<bits/stdc++.h>
using namespace std;

vector<vector<int>> dirs={{0,1,1},{0,-1,2},{1,0,3},{-1,0,4}};

int minCost(vector<vector<int>>& grid) {
    int n=grid.size(),m=grid[0].size();
    vector<vector<int>> dp(n,vector<int>(m,99999));
    //source distance will be 0
    dp[0][0]=0;
    deque<pair<int,int>> dq;
    dq.push_back({0,0});
    while(!dq.empty()){
        pair<int,int> p=dq.front();
        dq.pop_front();
        //if we reach the destination
        if(p.first==n-1 && p.second==m-1) return dp[p.first][p.second];

        for(vector<int> dir:dirs){
            int x=dir[0]+p.first;
            int y=dir[1]+p.second;
            if(x<0 || y<0 || x>=n || y>=m) continue;
            int cost;
            //checking the ddirection indicated by grid[p.first][p.second]
            if(grid[p.first][p.second]==dir[2])
                cost=0;
            else
                cost=1;
            //if cost of adjacent cell is less insert it in the queue
            if(dp[x][y]>dp[p.first][p.second]+cost){
                dp[x][y]=cost+dp[p.first][p.second];
                //if cost is 0 insert it in front of deque
                if(cost==0)
```

```cpp
                        dq.push_front({x,y});
                    //else insert it at back of deque
                    else
                        dq.push_back({x,y});
                }
            
        }
    }
    return dp[n-1][m-1];
}

int main(){
    int n,m;
    cin>>n>>m;
    vector<vector<int>> grid(n,vector<int>(m));
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            cin>>grid[i][j];
        }
    }
    cout<<minCost(grid);
  return 0;
}
```

CODE EXPLANATION

#include<bits/stdc++.h>
using namespace std;

vector<vector<int>> dirs = {{0,1,1},{0,-1,2},{1,0,3},{-1,0,4}};

This part of the code includes the necessary C++ standard libraries and defines a 2D vector `dirs`. Each subvector within `dirs` represents a direction and its associated information. The three values in each subvector are as follows:

- The first value is the change in the row index (i.e., delta_x) when moving in this direction.

- The second value is the change in the column index (i.e., delta_y) when moving in this direction.
- The third value represents the direction itself, where 1 means go right, 2 means go left, 3 means go down, and 4 means go up.

```
int minCost(vector<vector<int>>& grid) {
    int n = grid.size(), m = grid[0].size();
    vector<vector<int>> dp(n, vector<int>(m, 99999));

    // Source distance will be 0
    dp[0][0] = 0;
    deque<pair<int, int>> dq;
    dq.push_back({0, 0});
```

- The `minCost` function takes a 2D vector `grid` as input, which represents the grid with directions.
- `n` and `m` are the number of rows and columns in the grid, respectively.
- `dp` is a 2D vector used for dynamic programming, initialized with a large value (99999) to represent infinity.
- The distance to the source cell (0, 0) is set to 0, indicating that it costs nothing to start from there.
- `deque<pair<int, int>> dq` is a double-ended queue (deque) of pairs of integers. It is used for BFS (Breadth-First Search) to explore the grid.

```
    while (!dq.empty()) {
        pair<int, int> p = dq.front();
        dq.pop_front();

        // If we reach the destination
        if (p.first == n - 1 && p.second == m - 1)
            return dp[p.first][p.second];

        for (vector<int> dir : dirs) {
            int x = dir[0] + p.first;
            int y = dir[1] + p.second;

            if (x < 0 || y < 0 || x >= n || y >= m)
                continue;

            int cost;
```

```
            // Checking the direction indicated by grid[p.first][p.second]
            if (grid[p.first][p.second] == dir[2])
                cost = 0;
            else
                cost = 1;

            // If cost of adjacent cell is less, insert it in the queue
            if (dp[x][y] > dp[p.first][p.second] + cost) {
                dp[x][y] = cost + dp[p.first][p.second];

                // If cost is 0, insert it in front of deque
                if (cost == 0)
                    dq.push_front({x, y});
                // Else insert it at back of deque
                else
                    dq.push_back({x, y});
            }
        }
    }
    return dp[n - 1][m - 1];
}
```

- This part of the code implements the BFS algorithm to find the minimum cost.
- The `while (!dq.empty())` loop continues as long as there are cells in the deque to explore.
- For each cell `(p.first, p.second)` extracted from the deque, the code checks if it has reached the destination `(n - 1, m - 1)`. If it has, it returns the minimum cost, indicating that a valid path has been found.
- The code then iterates through the `dirs` vector, which defines the four possible directions to move (right, left, down, up).
- For each direction, it calculates the new row index `x` and column index `y` based on the current cell `(p.first, p.second)`.
- It checks if the new indices `x` and `y` are within the valid bounds of the grid (not out of bounds).
- The `cost` variable is determined based on whether the current cell's direction matches the direction defined by `grid[p.first][p.second]`. If they match, the cost is set to 0; otherwise, the cost is set to 1, indicating that a direction change is needed.
- If the cost of reaching the adjacent cell `(x, y)` is less than the current cost stored in `dp[x][y]`, it updates the minimum cost in `dp[x][y]`.

- Depending on whether the cost is 0 (no direction change) or 1 (direction change), it adds the cell `(x, y)` to the front or back of the deque, respectively. This ensures that cells with cost 0 are explored first, helping find shorter paths more efficiently.
- Finally, the function returns the minimum cost to reach the destination cell `(n - 1, m - 1)`.

```
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> grid(n, vector<int>(m));

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            cin >> grid[i][j];
        }
    }

    cout << minCost(grid);
    return 0;
}
```

- In the `main` function, the code reads the dimensions of the grid (`n` and `m`) and the grid elements from the input.
- It then calls the `minCost` function to calculate the minimum cost to make the grid have at least one valid path.
- Finally, it prints the result.

This C++ code correctly solves the problem of finding the minimum cost to ensure a valid path through the grid and efficiently handles direction changes and path exploration using BFS