Labyrinth

You are given a map of a labyrinth, and your task is to find a path from start to end. You can walk left, right, up and down.

Input

The first input line has two integers n and m: the height and width of the map.

Then there are n lines of m characters describing the labyrinth. Each character is . (floor), # (wall), A (start), or B (end). There is exactly one A and one B in the input.

Output

First print "YES", if there is a path, and "NO" otherwise.

If there is a path, print the length of the shortest such path and its description as a string consisting of characters L (left), R (right), U (up), and D (down). You can print any valid solution.

Constraints

1 <= n,m <= 1000

Example

Input:

```
5 8
########
#.A#...#
#.##.#B#
#......#
########
```

Output:

```
YES
9
LDDRRRRRU
```

Problem Breakdown

Lets understand the problem first

Have to go from A -> B

Have to walk in a grid(labyrinth) which also have obstacles

We have to find the path and not just a path but the shortest path

```
########
#.A#...#
#.##.#B#
#......#
########
```

Why is this a graph problem

Any problem where entities are connected and you need to explore reachability between them can usually be represented as a graph.

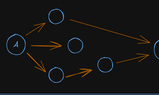
Each cell in the grid becomes a node, and each valid move to an adjacent cell is an edge.

This is called implicit graph construction — we don't store an adjacency list, we use the grid itself to traverse.

Shortest path in graph (unweighted)

This is a classic case for Breadth-First Search (BFS).

BFS explores layer by layer, guaranteeing the first time we reach B, we've used the fewest steps



How do we reconstruct the path?

We use a prev matrix to store the direction taken to enter each cell (U, D, L, R).

Once we reach the destination (B), we backtrack using these directions.

We reverse the path at the end to go from A to B.

Algorithm

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

using pii = pair<int, int>;

const vector<char> DIRECTIONS = {'U', 'R', 'D', 'L'};
const vector<int> dRow = {-1, 0, 1, 0}; // U, R, D, L
const vector<int> dCol = {0, 1, 0, -1}; // U, R, D, L

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m;

    vector<string> grid(n);
    for (int i = 0; i < n; ++i)
        cin >> grid[i];

    vector<vector<bool>> visited(n, vector<bool>(m, false));
    vector<vector<char>> previousDirection(n, vector<char>(m, 0));

    pii start, end;
    for (int row = 0; row < n; ++row) {
        for (int col = 0; col < m; ++col) {
            if (grid[row][col] == 'A')
                start = {row, col};
            else if (grid[row][col] == 'B')
                end = {row, col};
        }
    }

    queue<pii> q;
    visited[start.first][start.second] = true;
    q.push(start);

    bool found = false;

    while (!q.empty() && !found) {
        auto [row, col] = q.front(); q.pop();

        for (int d = 0; d < 4; ++d) {
            int newRow = row + dRow[d];
            int newCol = col + dCol[d];

            if (newRow >= 0 && newRow < n && newCol >= 0 && newCol < m &&
                !visited[newRow][newCol] && grid[newRow][newCol] != '#') {

                visited[newRow][newCol] = true;
                previousDirection[newRow][newCol] = DIRECTIONS[d];
                q.push({newRow, newCol});

                if (make_pair(newRow, newCol) == end) {
                    found = true;
                    break;
                }
            }
        }
    }

    if (!visited[end.first][end.second]) {
        cout << "NO\n";
        return 0;
    }

    // Reconstruct the path
    string path;
    pii current = end;

    while (current != start) {
        char dir = previousDirection[current.first][current.second];
        path.push_back(dir);

        int index = find(DIRECTIONS.begin(), DIRECTIONS.end(), dir) - DIRECTIONS.begin();
        current.first -= dRow[index];
        current.second -= dCol[index];
    }

    reverse(path.begin(), path.end());

    cout << "YES\n";
    cout << path.size() << '\n';
    cout << path << '\n';

    return 0;
}
```

Read n, m, and the grid.

prev_dir array to record the direction taken into each cell.

Locate start A and B.

Start BFS with A

For each cell dequeued

For each 4-direction move:

If valid and unvisited:

Mark visited, save direction, and enqueue.

Stop when reaching B.

If B reached:

Backtrack using prev_dir to reconstruct the path.

Time Complexity

O(N x M)

we visit every cell at most once.

Space Complexity

O(N * M)

for queues and auxiliary arrays