Sep 7, 2025

GRAPHS - BFS USAGE

* In DFS, using some order we were going to every node exactly once.
BFS also allows you to do the same thing plus let us solve this additional problem:
Shortest Path Problem

- Reachability
- Component Numbering
- Bi-Partite
- Cycles

(Use of DFS)

- Topological
- Shortest Path (only using BFS)

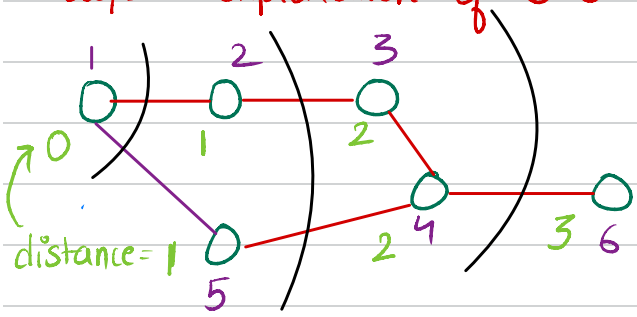
(Use of BFS)

(can be solved using both DFS & BFS)

```
BFS(x):
    Q.push(x)
    While (!Q.empty())
    {
        cur = Q.front(); Q.pop();
        if (visited[cur]) continue;
        visited[cur] = 1;
        for (v : neighbour[cur])
        {   if(!visited[v]) {
                Q.push[v];
            }
        }
    }
```

Instead of Recursion, we are using Queue.

* Layered Exploration of BFS



Initially the queue is empty

1

Now, when 1 comes out, it pushes 2 and 5

1 { 2 5

Then 2 comes out, it pushes 3 (not visited neighbour)
Then 5 comes out, it pushes 4

1 { 2 5 { 3 4

Then 3 comes out, it might put 4 again (since 4 is not yet visited) [Only when something gets out of the queue it gets marked as visited]

1 { 2 5 { 3 4 4

Then, 4 goes out, 6 comes in.

1 { 2 5 { 3 4 { 4 6

Node at; 0 1 2 3
distance

Then, we take out 4 again, but this gets ignored as 4 is already visited

1 2 5 3 4 4 6

Then 6 will come out and mark itself visited & exit since no new node left to explore.

BFS is better to find distance because of this layered exploration of nodes.

0 distance nodes are pushing other nodes at distance 1.
The 1 distance nodes are coming out one by one and they are pushing back the nodes at distance 2 .. and so on.

What is distance?

Edges between 2 nodes.
In this example Node 3 has distance = 2 because there are 2 edges b/w Node 3 and Node 1.

Ques 1. Labyrinth

https://cses.fi/problemset/task/1193

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

- $1 \le n, m \le 1000$

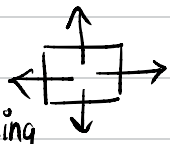
Example

Input:
```
5 8
########
#.A#...#
#.##.#B#
#......#
########
```

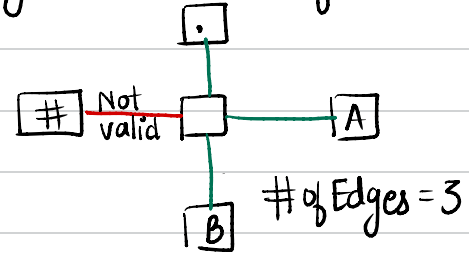
Output:
```
YES
9
LDDRRRRRU
```

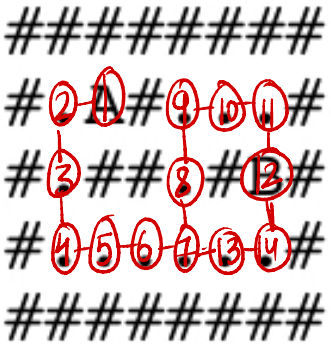
- Every character is a graph node in the input. Since, we can't go into the cell/nodes with '#' Hence, not keeping as a graph node. Only valid nodes are '.', 'A' & 'B'.

- Edges?



Nodes surrounding are edges here. Every cell has potential 4 neighbours. So, 4 edges But if the neighbouring node is a '#' then we shall make that edge invalid. Ex of valid edges:





← Implicit Graph

Here, we take each node as (x,y) and change our code.

As we can see above, this is a complex mapping. So, we will not create graph here.

When we have edge list like in DFS ques, those are explicit graph and can easily be mapped.

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

const int INF = 100;
using state = pair<int,int>;
#define F first
#define S second

int n,m;
vector<string> arr;

vector<vector<int>> vis,dist;

int dx[] = {0,1,0,-1};
int dy[] = {1,0,-1,0};

bool inside(int x,int y){
    if(0<=x&&x<n&&0<=y&&y<m)return 1;
    else return 0;
}
vector<state> valid_neighbours(state cur){
    vector<state> res;
    for(int dir = 0;dir<4;dir++){
        int nx = cur.F + dx[dir];
        int ny = cur.S + dy[dir];
        if(inside(nx,ny) && arr[nx][ny]!='#'){
            res.push_back(make_pair(nx,ny));
        }
    }
    return res;
}

void printer(){
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            cout<<dist[i][j]<<"\t";
        }
        cout<<endl;
    }
    cout<<endl;
}
```

```
void bfs(state st){
    vis = vector<vector<int>>(n, vector<int>(m,0));
    dist = vector<vector<int>>(n, vector<int>(m,INF));

    queue<state> q;
    // Add source
    q.push(st);
    dist[st.F][st.S]=0;
    // Start BFS
    while(!q.empty()){
        state cur = q.front(); q.pop();
        if(vis[cur.F][cur.S])continue;

        // Explore
        vis[cur.F][cur.S]=1;
        for(state v:valid_neighbours(cur)){
            if(!vis[v.F][v.S] && dist[v.F][v.S] > dist[cur.F][cur.S]+1){
                q.push(v);
                dist[v.F][v.S] = dist[cur.F][cur.S]+1;
            }
        }

        // cout<<cur.F<<" "<<cur.S<<endl;
        // printer();
    }
}
```

```
void solve(){
    cin>>n>>m;
    arr.resize(n);
    for(int i=0;i<n;i++){
        cin >> arr[i];
    }
    state st,en;
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            if(arr[i][j]=='A')st={i,j};
            else if(arr[i][j]=='B')en={i,j};
        }
    }
    bfs(st);
    printer();
    if(dist[en.F][en.S]==INF){
        cout<<"NO\n";
    }else{
        cout<<"YES\n";
        cout<<dist[en.F][en.S]<<endl;
    }
}
```

* Path Printing

How do we tweak above algorithm so it covers path also (print)

Define a parent node for each node except root

Changes in bfs fxn:

```
void bfs(state st){
    vis = vector<vector<int>>(n, vector<int>(m,0));
    dist = vector<vector<int>>(n, vector<int>(m,INF));
    par = vector<vector<state>>(n, vector<state>(m,{-1,-1}));
    queue<state> q;
    // Add source
    q.push(st);
    dist[st.F][st.S]=0;
    // Start BFS
    while(!q.empty()){
        state cur = q.front(); q.pop();
        if(vis[cur.F][cur.S])continue;

        // Explore
        vis[cur.F][cur.S]=1;
        for(state v:valid_neighbours(cur)){
            if(!vis[v.F][v.S] && dist[v.F][v.S] > dist[cur.F][cur.
            S]+1){
                q.push(v);
                dist[v.F][v.S] = dist[cur.F][cur.S]+1;
                par[v.F][v.S] = cur;
            }
```

Update in main()

```
    bfs(st);
    if(dist[en.F][en.S]==INF){
        cout<<"NO\n";
    }else{
        cout<<"YES\n";
        cout<<dist[en.F][en.S]<<endl;

        state cur = en;
        vector<state> path;
        while(cur!=make_pair(-1,-1)){
            path.push_back(cur);
            cur = par[cur.F][cur.S];
        }

        for(auto v:path){
            cout<<v.F<<" "<<v.S<<endl;
        }
    }
}
```

* Printing Direction:

Addition to above code:

```
        vector<state> path;
        while(cur!=make_pair(-1,-1)){
            path.push_back(cur);
            cur = par[cur.F][cur.S];
        }
        reverse(path.begin(),path.end());
        for(int i=1;i<path.size();i++){
            int cx = path[i].F-path[i-1].F;
            int cy = path[i].S-path[i-1].S;
            if(cx==1)cout<<"D";
            else if(cx==-1)cout<<"U";
            else if(cy==1)cout<<"R";
            else cout<<"L";
        }
    }
}
```

T.C = O(N+M)

* Homework Problems:

1. https://cses.fi/problemset/task/1667

2. https://cses.fi/problemset/task/1668

AlgoZenith