```
/*

 Thoughts on how search algorithms work

 Search algorithms play a prominent role in seach engines, path finding, and
             even in AI such as AlphaGo. I self-studied two fundamental
             search algorithms, DFS and BFS, trying to apply them to a
             specific simple case and make analogies to understand them in a
             more significant way.
 */
/*

 "Lake Counting" from USACO 2004 November

 Description:
 Due to recent rains, water has pooled in various places in Farmer John's
             field, which is represented by a rectangle of N x M (1 <= N <=
             100; 1 <= M <= 100) squares. Each square contains either water
             ('W') or dry land ('.'). Farmer John would like to figure out
             how many ponds have formed in his field. A pond is a connected
             set of squares with water in them, where a square is considered
             adjacent to all eight of its neighbors.Given a diagram of Farmer
             John's field, determine how many ponds he has.

 Input:
     * Line 1: Two space-separated integers: N and M
     * Lines 2...N+1: M characters per line representing one row of Farmer
                       John's field. Each character is either 'W' or '.'.
                       The characters do not have spaces between them.

 Output:
     * Line 1: The number of ponds in Farmer John's field.
     Sample input:
     10 12
     W........WW.
     .WWW.....WWW
     ....WW...WW.
     .........WW.
     .........W..
     ..W......W..
     .W.W.....WW.
     W.W.W.....W.
     .W.W......W.
     ..W.......W.
     Sample output:
     3
*/
/*
 The main idea of this code is to firstly find one water square, by simple
             iterating; then use search algorithms to find all water squares
             connected to it. Then set number_of_ponds up by 1, mark all the
             squares involved as visited. Continue to find next one water
             square that is not visited yet. I applied BFS and DFS seprately;
             either gives the correct answer and is accepted by online judge.

 In my understanding, BFS in this kind always works depending on new data
             structure "point" and its queue. It is just like a mouse mother
             sending her baby mice to every possible squares next to the
             square she stays on, at the same time marking the squares that
             have been visited; then baby mice become mothers immediately,
             and the process repeats until one of the mice arrives at the
             destination.

 In comparison, in my understanding, DFS works depending on stack by a
             recursive function. It is just like a mouse goes to one possible
             square next to the square she stays on repeatedly until she has
             nowhere to go; then she returns to the previous one square and
             selects one of the other possible squares, and the process
             repeats until one of the mice arrives at the destination.
*/



//headers:
#include<iostream>
#include<algorithm>
#include<cstring>
#include<cmath>
#include<queue>
using namespace std;



// define the conditions needed to be expressed in bool array G[][] and u[]
          []
#define WATER 1
#define DRY_LAND 0
#define VISITED 1
#define NOT_VISITED 0

// N and M are both less than 101
const int maxn=102;

// these 2 arrays above serves to check all 8 neighbors by iterating through
          them and adding dx[i] to current x-coordinate and dy[i] to current
          y-coordinate simply and effectively.
const int dx[8]={0,1,0,-1,-1,1,-1,1},dy[8]={1,0,-1,0,1,1,-1,-1};

// I use bool arrays to save computer's memory, for only states "water" or
          "dry","visited" or "not visited" are necessarily expressed.

// to save the map of water and dry lands
bool G[maxn][maxn]={DRY_LAND};

// u[x][y] serves to save whether G[x][y] is visited or checked in order to
          simplify the loop.
bool u[maxn][maxn]={NOT_VISITED};

int N=0,M=0;//N and M given in the description

//Once a pond is found by each algorithm, these two variables will increase
          by 1. To represent the number of ponds correctly, their initial
          value will be set to 0.
int ans_of_BFS=0;
int ans_of_DFS=0;

//BFS:
// define a new data structure: point
struct poi
{
    // x and y represent the x and y coordinate of the point
    int x,y;
};
queue<poi> q;



// to check if point(a,b) is within the map
bool is_ok(int x,int y)
{
    return (x>=1 && x<=N) && (y>=1 && y<=M);
}

void BFS(int x,int y)
{
    // the first mouse mother
    poi s;
    s.x=x;
    s.y=y;
    // initialize and push her into the queue
    q.push(s);

    // When the queue is empty, it means all the mice has nowhere to go,
                    which means the input has no solution
    while(!q.empty())
    {
        poi cur,tmp;
        cur=q.front();
        // pop current mother mouse out of the queue
        q.pop();

        // send baby mice to adjacent blocks one by one
        for(int i=0;i<8;i++)
        {
            tmp.x=cur.x+dx[i];
            tmp.y=cur.y+dy[i];

            // Is it within the map? Is it visited? Is it water?
            if(is_ok(tmp.x,tmp.y) && u[tmp.x][tmp.y]==NOT_VISITED && G[tmp.x
                                        ][tmp.y]==WATER)
            {
                // mark as visited
                u[tmp.x][tmp.y]=VISITED;
                // push it to the tail of the queue in order to operate
                q.push(tmp);

            }
        }
    }
}

void DFS(int x,int y)
{
    for(int i=0;i<8;i++)
    {
        int nx=x+dx[i];
        int ny=y+dy[i];

        // Is it within the map? Is it visited? Is it water?
        if(is_ok(nx,ny) && u[nx][ny]==NOT_VISITED && G[nx][ny]==WATER)
        {
            // mark as visited
            u[nx][ny]=VISITED;

            // recursion
            DFS(nx,ny);
        }
    }
}



int main()
{
    char ch;

    // read N and M
    cin>>N>>M;

    // read the whole char-matrix
    for(int i=1;i<=N;i++)
    {
        for(int j=1;j<=M;j++)
        {
            cin>>ch;
            if(ch=='W')
            {
                G[i][j]=WATER;
            }
            else
            {
                G[i][j]=DRY_LAND;
            }
        }
    }

    // compute the answer given by BFS
    // iterate through the map
    for(int i=1;i<=N;i++)
    {
        for(int j=1;j<=M;j++)
        {
            // if a water square that is not visited is found:
            if(u[i][j]==NOT_VISITED && G[i][j]==WATER)
            {
                // mark it as visited
                u[i][j]=VISITED;

                // mark all water squares connected to it as visited
                BFS(i,j);

                // number_of_ponds++;
                ans_of_BFS++;
            }
        }
    }

    // reset for reusage
    for(int i=1;i<=N;i++)
    {
```

```cpp
        for(int j=1;j<=M;j++)
        {
            u[i][j]=NOT_VISITED;
        }
    }

    // compute the answer given by DFS
    for(int i=1;i<=N;i++)
    {
        for(int j=1;j<=M;j++)
        {
            if(u[i][j]==NOT_VISITED && G[i][j]==WATER)
            {
                u[i][j]=VISITED;
                DFS(i,j);
                ans_of_DFS++;
            }
        }
    }

    // output the answer only if both algorithms give the same answer
    if(ans_of_BFS==ans_of_DFS)
    {
        cout<<ans_of_DFS<<endl;
    }
    else
    {
        cout<<"ERROR"<<endl;
    }
    return 0;
}
```