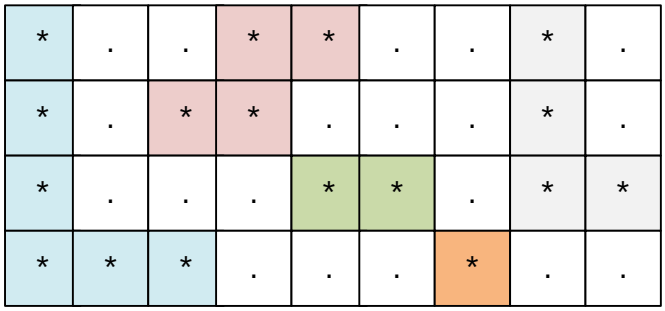
Depth-First Search on Grids

Look at the picture of the space that has a form of a grid *n* * *m*:



We have a grid with *n* = 4 rows and *m* = 9 columns. Stars ('*') are stones in the space, points ('.') are empty places. If two stones have *common side* (not corner!), they belong to one asteroid. According to this picture, we can ask the questions:

- How many asteroids are there is the space?
- Find the biggest (smallest) asteroid

How to store grid in memory? Like two-dimentional char array:

C style	C++ style
char m[101][101]	**string** m[101]

The grid is given in input like a set of strings. First line usually contains the size of a grid – values of *n* and *m*.

```
4 9
*..**..*.
*.**...*.
*...**.**
***...*..
```

Here is the way of reading input:

C style

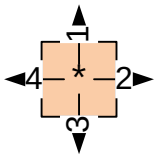
```
scanf("%d %d\n", &n, &m);
for (i = 0; i < n; i++)
{
  for (j = 0; j < m; j++)
    scanf("%c", &m[i][j]);
  scanf("\n");
}
```

C++ style

```
cin >> n >> m;
for (i = 0; i < n; i++)
  cin >> m[i];
```

Imagine that you stand in a cell with a stone and want to pass all the cells of one asteroid. You can move in four directions: *up*, *right*, *down* and *left*. It does not matter

the direction which we go first. Algorithm must be deterministic, so let's define the order in which the directions will be passed:



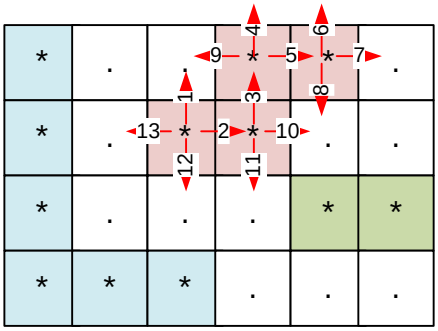
Let's move through the cells with stones in the next way:

- If there is a neighbour cell (that has a common side) with stone that is not visited yet – go there;
- If there is no way to go (there is no unvisited neighbour cell) – return back.

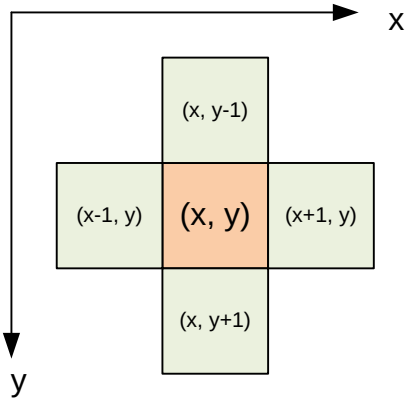
We have two cases where its forbidden to move:

- You can't move otside the border;
- You can't move to the empty cell.

If we arrive to one of forbidden cells – we must return back.

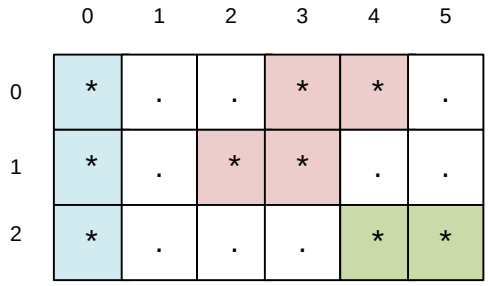


Current position in depth first search algorithm is given with (x, y) coordinates. Trying to move into four neighbouring positions means to move to the cells with coordinates:



dfs(x, y)
if (x,y) outside the grid, **return**;
if (x,y) is empty space, **return**;
dfs(x,y-1)
dfs(x+1,y)
dfs(x,y+1)
dfs(x-1,y)

Grid n * m means that rows are numbered from 0 to n – 1, columns are numbered from 0 to m – 1. Below given the sample of 3 * 6 grid:



Current position (x, y) is out of bounds means satisfying the next condition:

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

Initially all the cells of the grid are unvisited. When we start to pass them, we need to mark them used. We can use additional two-dimentional array `int used[101][101]` for it. But there also exists another way. When we enter the cell (x, y) with stone, we have: m[x][y] = '*'. When we pass this cell, we mark it as stone disappeared, like now this position is empty: m[x][y] = '.'. So when depth first search is run on one asteroid, this asteroid *disappears* from the map.

Each asteroid on a grid can be considered like a separate connected component. Number of asteroids on a grid equals to the number of *connected components*.

E-OLYMP 2590. Space Exploration Grid n * n contains stones and empty cells. Count the number of asteroids in the grid.

► Read the grid. Find the number of connected components.

```
#include <iostream>
#include <string>
using namespace std;

int n, i, j, c;
string m[1001]; // declare the grid - two dimentional char array

// start depth first search at position (x, y)
void dfs(int i, int j)
{
  // if we out of bounds of the grid - return back
  if ((i < 0) || (i >= n) || (j < 0) || (j >= n)) return;

  // if we come to empty cell - return back
  if (m[i][j] == '.') return;

  // mark current position as used (passed)
  m[i][j] = '.';

  // run depth first search in four neighbouring directions
  dfs(i - 1, j); dfs(i + 1, j);
  dfs(i, j - 1); dfs(i, j + 1);
}

int main(void)
{
  //freopen("grid.in", "r", stdin);
```

```
  // read input grid
  cin >> n;
  for (i = 0; i < n; i++)
    cin >> m[i];

  // count the number of asteroids - number of connected components
  c = 0;
  for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    // if position (i, j) contains a stone, run dfs from it
    if (m[i][j] == '*')
    {
      dfs(i, j);
      c++; // increase the number of connected components
    }

  // print the answer
  cout << c << endl;
  return 0;
}
```

E-OLYMP 2168. Square punch Grid *n* * *m* represents a badge. Some cells are punched. '1' is a punched cell, '0' is not a punched. Count the number of little badges that will remain after using a punch.

0	0	1	0
0	1	0	0
1	1	1	1
0	0	0	0
1	1	0	0

► Read the grid. Find the number of connected components.
In this problem grid is represented like two dimentional array of integers:

```
int mas[101][101];
```

Reading a grid looks like reading two dimentional array:

```
scanf("%d %d", &n, &m);
for (i = 0; i < n; i++)
for (j = 0; j < m; j++)
  scanf("%d", &mas[i][j]);
```

E-OLYMP 1063. Cell removal Grid *m* * *n* represents a sheet of paper. If the cell of the paper is cut, it is represented with '.' (point). If it is not cut, cell contains '#' sign. Find the number of pieces remained from the sheet.

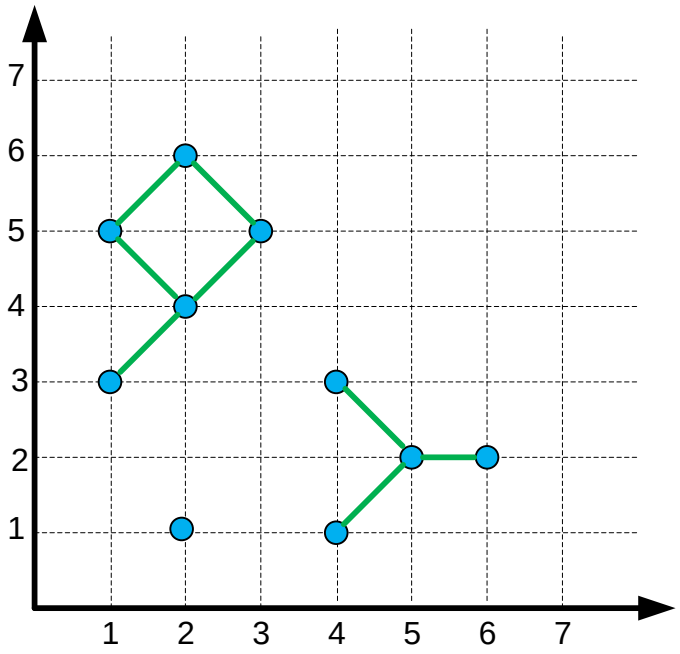
#	.	#	#	.	#	.	#
.	.	.	.	.	.	#	#
#	.	#	#	#	.	#	#
#	#	.	#	#	.	#	#

► Read the grid. Find the number of connected components.

E-OLYMP **1065. Garden beds** Grid *n* * *m* represents a garden bed. Find the number of connected components.

► Read the grid. Find the number of connected components.

E-OLYMP **1685. Lands of Antarctica** *n* polar stations are located on the map. You can move from one station to another only if the distance between them is no more than 2 unit segments. Find the number of connected components.



► Declare two dimentional array – this is our grid of size 101 * 101. Fill it with 0.

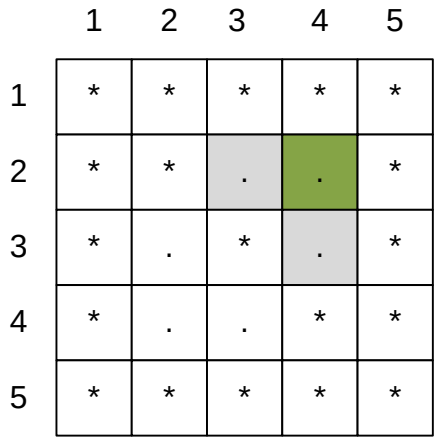
```
int m[101][101];
```

Read the coordinates of polar stations, mark points on the grid with 1.

```
scanf("%d", &n);
for (i = 0; i < n; i++)
{
  scanf("%d %d", &a, &b);
  m[a][b] = 1;
}
```

According to problem statement, during dfs on a grid, it is allowed to move *diagonally* too.

E-OLYMP 4001. Area of the room The person is located in a room on a grid at position (x, y). Find the size of the room.



► Run dfs(x, y). Use counter *cnt* in dfs function to find the number of visited cells.