Floodfill Algorithm

A floodfill is a name given to the following basic idea:

In a space (typically 2-D or 3-D) with a initial starting square, fill in all the areas adjacent to that space with some value or item, until some boundary is hit. As an example, imagine an input 2-D array that shows the boundary of a lake (land is designated with * characters.)

```
**********
**  ***  *
*    *   *
*    *   *
*        *
*    *   *
**********
```

Now, imagine that you wanted to fill in a "lake" with the ~ character. We'd like to write a function that takes in one spot in the lake (the coordinates to that spot in the grid), and fills in each contiguous empty location with the ~ character. Our final grid should look like:

```
**********
**~~***~~*
*~~~~*~~~*
*~~~~*~~~*
*~~~~~~~~*
*~~~~*~~~*
**********
```

Of course, in this particular, example, we could just fill in all spaces with ~ characters, but it's easy to imagine a larger grid where we just fill in this one lake and not other areas with spaces.

Depending on how the floodfill should occur (do we just fill in each square above, below, left and right, or do we ALSO fill in diagonals to squares already filled), the basic idea behind a recursive function that carries out this task is as follows (this is just a very rough sketch in pseudocode:

```
public static void FloodFill(char[][] grid,
int x, int y) {

  grid[x][y] = FILL;

  for (each adjacent location i,j to x,y) {

    if (inbounds(i,j) && grid[i][j]!=FILL)
      FloodFill(grid, i, j);

}
```

When we actually write code for a floodfill, we may either choose to use a loop to go through all adjacent locations, or simply spell out the locations, one by one. If there are 8 locations, a loop is usually desirable. If there are 4 or fewer, it might just make sense to write each recursive call out separately.

Minesweeper - Recursive Clear

Minesweeper is the popular game included with Windows where the player has to determine the location of hidden bombs in a rectangular grid. Initially, each square on the grid is covered. When you uncover a square, one of three things can happen:

1) A bomb is at that square and you blow up and lose.
2) A number appears, signifying the number of bombs in the adjacent squares.
3) The square is empty, signifying that there are no adjacent bombs.

In the real minesweeper, when step 3 occurs, all of the adjacent squares are automatically cleared for you, since it's obviously safe to do so. And then, if any of those are clear, all of those adjacent squares are cleared, etc.

Step 3 is recursive, since you apply the same set of steps to clear each adjacent square to a square with a "0" in it.

I am going to simplify the code for this a bit so that we can focus on the recursive part of it. (I will replace some code with comments that simply signify what should be done in that portion of code.) The full example is posted online under the Sample Programs. Comments have been removed so the code takes up less space.

The key here is ONLY if we clear a square and find 0 bombs adjacent to it do we make a recursive call. Furthermore, we make SEVERAL recursive calls, potentially up to 8 of them.

```
final public static int[] DX = {-1,-1,-1,0,0,1,1,1};
final public static int[] DY = {-1,0,1,-1,1,-1,0,1};

public static int domove(char[][] board, char[][] realboard,
                         int listbombs[][2], int row, int column) {
  int i, j, num;
  if (realboard[row][column]=='*') {
    // Hit a bomb, losing move…
  }
  else if (board[row][column]!='_') {
    // This square was previously cleared…
  }

  else {
    // This calculates the # of adjacent bombs.
    num = numberbombs(row, column, listbombs);

    board[row][column]=(char)(num+'0');

    // If there are no adjacent bombs, we can recursively clear.
    if (num == 0) {

      // Traces through all 9 squares in the box around row,col.
      for (i=0; i<dx.length; i++)

        if (valid(row+DX[i],column+DY[i]) &&
                (board[row+DX[i]][column+DY[i]]=='_'))

                  domove(board, realboard, listbombs, row+DX[i],
                                  column+DY[i], totalmoves);

    } // end-if num==0
    return 0;
  } // end else
}
```

In this code, a for loop structure with two constant arrays for DX and DY helps us iterate through all the adjacent squares:

```
          for (i=0; i<dx.length; i++)
```

Basically, DX[i] and DY[i] represent the offsets for row and column, respectively, for all possible adjacent squares.

Thus, the following if statement is critical:

```
                    if (valid(row+i,column+j) &&
                      (board[row+i][column+j]=='_'))
```

The first clause in the if statement prevents array out of bounds errors. The second clause in the if statement prevents from clearing a square that was previously cleared.

Only if these two tests are passed do we recursively clear the square with the location

```
                      row+i, column+j
```

In essence, we are performing a floodfill of all adjacent squares with no adjacent bombs, starting from the initial chosen location by the user.

Programming Contest Example: Golf Fine

The following question is taken from a high school computer programming contest held at University of Florida on January 28, 2010. Its solution involves a floodfill.

The question is included on the following two pages. That is followed by a discussion of how to solve the problem utilizing the floodfill idea, followed by the code that solves the problem.

Note: The input for the problem (for this contest) was supposed to come from the keyboard. I have written my solution so that the input is read in from a file called, “golf.txt”.

To fully test this program, one would have to test it many times with different input files, since the format in which the question is posed only tests one case at a time.

Golf Fine

Land development company Developers-R-Us has been in constant battle with environmentalists for decades. In recent years, the company has been responsible for destroying the habitat of the Michigan monkey flower, and has faced large fines as a consequence. The lawyers for this land development company have thought up a new idea - a sort of loophole in the system. The company has purchased a large plot of land, but will not develop all of it, thus incurring fines only for those areas containing monkey flowers that are adjacent (horizontally, vertically, or diagonally) to developed land. The environmental engineers of Developers-R-Us have provided you, the software engineer, with a series of grid maps representing the area where a new golf course will be built. They would like to determine the area which will be covered by the proposed golf course, as well as the fines the company will have to pay for building it. You will be provided with a 10 x 10 grid, representing the 100 square acres being used to build the golf course. Each acre of monkey flowers along the path costs $50 000.

Input

Input will consist of ten lines, each containing ten characters, where:

- . - represents land not being developed
- s - represents the start of the golf course; there will be exactly one such acre in the whole gridmap
- d - represents an acre of developed land; note that there may be developed land which is not connected (horizontally, vertically, or diagonally) to the golf course, but this is not your problem
- m - represents an acre of land containing Michigan monkey flowers

Output

Provide, on two separate lines, the number of acres being developed for the golf course, and the fine for building the course next to areas containing Michigan monkey flowers. Pay close attention to the output format for the fine: there must be a dollar sign at the beginning of the line, and a space should be used as thousands separator.

Example

Input

```
..........
.sd.......
..d.......
..dm......
..d.......
..d...m...
..ddddm...
..........
...ddd....
...mmm....
```

Output

```
10
$150 000
```

Our goal is to identify how many adjacent squares (directly or indirectly) to ‘s’ are part of the golf course.

This indicates that we ought to do a floodfill starting at character ‘s’.

Furthermore, we can assume that our floodfill should go in all eight directions. (Though this isn’t directly stated, it’s probably the best assumption to make, since this is how we are to tell whether or not the flowers are adjacent to the course.)

There is one more complicating factor for this problem after we do our floodfill:

Counting the adjacent squares that have monkey flowers in them

First, let’s concentrate on the floodfill:

This will be very, very similar to Minesweeper, except that we should fill our squares with ‘s’ and ‘d’ with different characters to mark the golf course. In the following implementation, the character ‘X’ is chosen. This is an arbitrary choice. Any choice of character that isn’t already in the grid would be fine.

Basically, we will change the grid at the location to be filled to the character ‘X’. Then, for each adjacent location, we’ll see if it’s part of the course (with a ‘d’). If so, we’ll continue the floodfill at that location.

Code for Golf Fine Floodfill

```
final public static int[] DX = {-1,-1,-1,0,0,1,1,1};
final public static int[] DY = {-1,0,1,-1,1,-1,0,1};

public static void floodfill(char[][] grid, int x, int y) {

   int i,j;

   // Mark this spot
   grid[x][y] = 'X';

   // Go through all valid adjacent squares with a 'd'.
   for (i=0; i<DX.length; i++)
      if (inbounds(x+DX[i],y+DY[i]) && grid[x+DX[i]][y+DY[i]]=='d')
          floodfill(grid,x+DX[i],y+DY[i]);

}
```

This should look very, very similar to the Minesweeper code. The only reason it's much shorter is the action we need to take to clear a square is very simple (just putting an 'X' in that slot) and there are fewer contingencies for other situations.

Also, note that the initial call to this function has the x-y coordinates of the spot of the 's', so that is why the floodfill doesn't look for the character 's' at all.

The rest of the code is in the attached file, golf.java.