# 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[][SIZE], int x, int y) {

  grid[x][y] = FILL_CHARACTER;

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

        if (i,j is inbounds and not filled)
            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 boolean domove(int r, int c, char[][] grid) {

    // You hit a bomb, you die!
    if (grid[r][c] == '*')
        return false;

    // Change this square to show # of adjacent bombs.
    int ans = numBombsAdj(r,c,grid);
    grid[r][c] = (char)('0'+ans);

    // Recursively clear
    if (ans == 0) {

        for (int i=0; i<DX.length; i++)
            if (inbounds(r+DX[i],c+DY[i]) && grid[r+DX[i]][c+DY[i]] == '_')
                domove(r+DX[i], c+DY[i], grid);
    }

    return true;
}
```

**In this code, the DX and DY arrays show all the valid movements from one square to the next for the floodfill. To utilize these parallel arrays, we loop through each possible index into the direction arrays:**

**for (i=0; i<DX.length; i++)**

**Basically, DX[i] and DY[i] represent the offsets for row and column, respectively, for the $i^{th}$ direction.**

**Due to our if statement that checks for validity, there is no problem in accidentally calling our function again with the exact same row and column value. This call does nothing because the if statement screens it out.**

**Thus, the following if statement is critical:**

```
if (inbounds(r+DX[i],c+DY[i]) && grid[r+DX[i]][c+DY[i]] == '_')
```

**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. These two checks MUST BE done in this order. If we tried to access the array index but it to see if it was an underscore before checking if it was inbounds, we would get an array out of bounds error if that new index was out of bounds. (The computer is smart enough to not check the second condition if the first condition in an and expression is false. This is called short-circuiting and we're taking advantage of it here in this if statement.)**

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

**r + DX[i], c + DY[i]**

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