# Friday Code Challenge #4: Implement the Flood Fill Algorithm

A picture containing text, crossword puzzle, fruit

Description automatically generated A picture containing text, crossword puzzle, fruit

Description automatically generated

## Overview

This code challenge is to implement the Flood Fill (aka Seed Fill) algorithm in ***the language of your choice***. This purpose of this algorithm is to determine and alter a sequence of nodes, connected to an arbitrary starting node, in a multi-dimensional array, by examining some matching attribute of the nodes. The most familiar use of this algorithm is in graphics packages (usually the bucket fill function) to fill a bounded area with a colour. This algorithm is described in this Wikipedia page: <https://en.wikipedia.org/wiki/Flood_fill> There is a JS solution here: <https://gpriestner.github.io/Filler/> Use the left mouse button to draw white squares and double-click inside to fill. (Don’t move the mouse too fast when drawing or there will be gaps in the outline that the fill will go through.)

Flood fill can be implemented as either the 4-way or 8-way implementation, as illustrated in the animated gifs, above. Note that the difference is whether a diagonal line can act as a boundary or not. We will concentrate on the 4-way version, but it is easy to modify the 4-way to an 8-way. Also, for simplicity, we will only test the algorithm in a test harness that only uses black and white.

This code challenge will focus on low level programming skills and problem solving rather that learning a framework to develop a high-level solution.

## Instructions

You will need to declare a two-dimensional array of numbers/integers. This data structure shall represent the image or array of pixels. 1 will represent white and 0 will be black. Look in appendix A to see the JavaScript syntax, which is similar to C# and Java.

You will need to implement a ***display*** function that will use the 2d array to provide some form of visual output that will make it easier to see what’s going on in the array. There are 2 options. First, use the logging facility in your environment to log out a representation of the array (e.g. substitute black squares for 0s and white squares for 1s). Second, you can research how to use the graphics capabilities of the environment to produce a higher quality image. This second option may be more difficult to implement. Unless you are already familiar with your environment’s graphics API then it may be easier to use the logging facility. For JS you can use a canvas, which is assumed in the starter code in Appendix A. For C# you can either output to a console window or use WinForms for graphical output. For Java, it is recommended to use the AWT. Code examples in this document are provided in JS.

Note that the JS grid example in appendix A forms a white square on a black background with a smaller square in the top right corner. All the lines are 1 pixel wide. Also note that this JS example implements a 2d array as an array of arrays. Make sure that your ***display*** function accurately shows this image. Also note that you will need to use the array access syntax like this (note the order of the x and y indexes):

grid[y][x]

This order is needed when reading and writing to the grid in JS because of the way the 2d array is accessed. Make sure you research/understand why the [x] and [y] seem to be reversed in the JS implementation and check whether this is needed for C#/Java.

Add this code (this is JS, you will need to work out the equivalent if you are using a different language):

**const** xStart = 10;

**const** yStart = 10;

**var** oldColour = grid[yStart][xStart];

**var** newColour = oldColour === 0 ? 1 : 0;

Next write a function called fill that takes 2 parameters x and y (numbers/ints). Implement this function according to the following pseudo code (based on the Wikipedia article):

Fill (x,y)

1. If x,y is within the bounds of the 2d array and the colour at x,y is the old colour to be replaced
   1. Set x,y to the new colour
   2. Call Fill for the north node
   3. Call Fill for the south node
   4. Call Fill for the east node
   5. Call Fill for the west node

When this function is ready then test it by calling it and passing in xStart and yStart. Experiment with different values of xStart and yStart so that the Fill function is called with initial points in the smaller square and outside the large square. Verify that the Fill function is filling the expected area of the image by calling the ***display*** function.

If the Fill function is not working as expected, then experiment by calling the ***display*** function inside Fill just after step 1a and set a break point so you can step through the code in the debugger so you can see each individual pixel being changed as part of the flood fill process.

The recursive Fill algorithm is the easiest to implement, so that was attempted first. When you get it working successfully, write another Fill function, called FillStack or FillQueue, and reimplement using a stack or queue instead of using recursion. See the Wikipedia article for the pseudo code that describes the algorithm for these variations on the recursive Fill.

If you have time for a final flourish, and you chose one of the graphical output options (JS canvas, WinForms, AWT), you can add functionality that allows you to draw an outline of a shape with the mouse, and then right-clicking inside it to select the start position and colour to be replaced before kicking off you code to perform the fill.

## Appendix A: Starter Files for JS

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta http-equiv="X-UA-Compatible" content="IE=edge" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Fill</title>

  </head>

  <body

    oncontextmenu="return false;"

    style="margin: 0; height: 100vh; background-color: black"

  >

    <canvas style="display: block"></canvas>

    <script type="module" src="index.js"></script>

  </body>

</html>

console.log("Starting. . . .");

**var** canvas = document.querySelector("canvas");

**var** view = canvas.getContext("2d");

resize();

**function** resize() {

  canvas.height = window.innerHeight;

  canvas.width = window.innerWidth;

  view.fillStyle = "white";

}

**var** grid = [

  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

  [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],

  [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],

  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

];