Our Flounder

Computer Science Correspondence School

discere domi

Lecture 26

CISC 131
Introduction to Programming and Problem Solving
Spring 2020
The Game of Life

Points: None

Solutions to Problems from Lecture 25

Here are my solutions to the functions you were to write. Your implementations may differ and, if you followed the requirements, that is okay. There are usually several ways to implement the logic of an algorithm.

Problem 1

Write a function with this function header:

```
function canBeAdded ( array2D1, array2D2 )
```

The function is passed two two-dimension arrays and returns *Boolean true* when the actual parameters are two dimension arrays for which matrix addition is defined and *Boolean false* if they cannot be added. You may want to refresh your memory on the requirements for doing matrix addition.

```
function canBeAdded ( array2D1, array2D2 )
{
 //  matrics can be added if they each have at least one row,
 //  are rectangular, and have the same shape.
 return isNaN (findFirstElement ( array2D1 )) &&
        isRectangular           ( array2D1 )  &&
        sameShape               ( array2D1, array2D2 );
}
```

Problem 2

Write a function with this function header:

```
function canBeMultiplied ( multiplicand, multiplier )
```

The function is passed two two-dimension arrays and returns *Boolean true* when the actual parameters are two dimension arrays for which matrix multiplication is defined and *Boolean false* if they cannot be multiplied. You may want to refresh your memory on the requirements for doing matrix multiplication.

```
function canBeMultiplied ( multiplicand, multiplier )
{
 // matrices can be multiplied if they each have at least one
 // column, are rectangular and the number of columns of multiplicand
 // is equal to the number of rows of multiplier
 return isNaN (findFirstElement  ( multiplicand )) &&
        isNaN (findFirstElement  ( multiplier )) &&
        isRectangular            ( multiplicand ) &&
        isRectangular            ( multiplier ) &&
        multiplicand[0].length ===  multiplier.length;
}
```

Problem 3
Write a function with this function header:

```
function add ( array2D1, array2D2 )
```

The function is passed two, two dimension arrays. If the arrays represent matrices that can be added, the function returns a two dimension array that represents the result of doing matrix addition on the formal parameters. If the arrays do not represent matrices that can be added, the function returns a zero length array.

```
function add ( array2Da, array2Db )
{
 var col;
 var row;
 var result;

 result = [];
 if ( canBeAdded(array2Da, array2Db ) )
 {
  result = new Array ( array2Da.length );
  for (row=0; row<array2Da.length; row++)
  {
   result[row] = new Array(array2Da[row].length);
   for (col=0; col<array2Da[row].length; col++)
   {
    result[row][col] = array2Da[row][col] + array2Db[row][col];
   }  // for col
  }   // for row
 }
 return result;
}
```

Problem 4
Write a function with this function header:

```
function subtract ( minuend, subtrahend )
```

The function is passed two, two dimension arrays. If the arrays represent matrices that can be subtracted, the function returns a two dimension array that represents the result of doing matrix subraction on the formal parameters. If the arrays do not represent matrices that can be added, the function returns a zero length array.

Implementation is very similar to the add function shown above in Problem 3

Problem 5
Write a function with this function header:

```
function transpose ( array2D )
```

Write a function is passed a two dimension array that represents a matrix. The function returns a two dimension array that represents the transpose of the formal parameter matrix. Return a zero length array if it is not possible to transpose the parameter array.

```
function transpose( array2D )
{
 var i;
 var j;
 var result;

 result = [];
 if ( !isNaN(findFirstElement ( array2D) ) && isRectangular(array2D) )
 {
  result = new Array ( array2D[0].length );
  for ( i=0; i<result.length; i++ )
  {
   result[i] = new Array ( array2D.length );
   for ( j=0; j<result[i].length;j++) {  result[i][j] = array2D[j][i];
}
  }  // for i
 }   // if

 return result;
                                }
```

Start of Today’s Assignment

In this assignment you will implement the *Game of Life*. You will want to read the Wikipedia article:

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

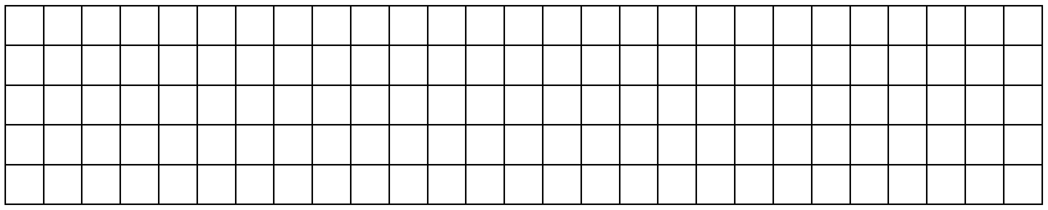
before proceeding with the assignment.

The implementation that you create will use a straightforward approach. Two two-dimensional arrays will be used in your program. They will have the same shape and size. An HTML grid of *div* cells with this same shape and size will be used to display the results of applying the game’s rules. If you want to use and HTML *table* instead of *divs*, you may do so. The examples I give below use *divs* but tables might be less work and the detail work with *divs* about first row and last row described below can be avoid. Whether you use *tables* or *divs*, the HTML for the grid will be generated by your program.

Some of the functions that you will need for this assignment have already been written in previous assignments so you should reuse what you have, if you have it. For example, you will need the *createHTMLElement* function, so just copy that into your program when you reach that step.

Step 1: HTML and CSS

Before you can have your program generate HTML, you first need to know how the page should look and then decide how to go about achieving that. The *Game of Life* is simply a grid of cells on the page:



The grid will be the same size and shape as each of the two arrays you create in your Javascript program. I think it is useful to have a border around the cells to help define the game area. I suggest you use this CSS for the class of each cell:

```
.cell
{
 border-left   : 1px solid black;
 border-bottom : 1px solid black;
 float         : left;
 height        : .5em;
 width         : .5em;
}
```

Each cell is nested inside the *gameBoard* which, in turn, is nested inside the *page*, like this:

```
<body>
<div id="page">
<div id="gameBoard">
</div>   <!--  gameBoard -->
</div>   <!-- page -->
</body>
```

Give the *page* a margin to offset it from the edges of the browser window, but don’t give it a *height* or *width*. Instead, include this class in the cell that starts each row:

```
.newRow
{
 border-left : 1px solid black;
 clear       : left;
}
```

To get a single border around each cell, you will also need to include these two CSS classes and apply them to the appropriate cells:

```
.firstRow   { border-top   : 1px solid black; }
.lastColumn { border-right : 1px solid black; }
```

Lay out a couple of short rows by hand to ensure that your CSS classes give you the results you need. Your program will be generating all the *innerHTML* of the *gameBoard* element, so it is important that you understand which elements get the additional CSS classes.

Step 2: Some Preliminary Javascript Functions

Having the program generate the HTML cannot be done until the program creates the arrays and, before that happens it will be useful to write and test some helper functions that will be used in the game.

Every cell in the game is in one of two states: *alive* or *dead*. Your program will be using an array element to represent the state of each cell and, therefore, each element in the array must contain one of two possible values. It doesn't really matter what these values are as long as the values are not the same. It is a bad programming practice to use values of this kind in multiple places in a program - it makes maintenance and debugging much more difficult. To avoid that, write two simple functions:

```
function getDeadValue()
function getLiveValue()
```

Each function will return the value associated with the state it is implementing. I used zero and one, but you can use any value you want. If a function needs to use these values, that function *must* call one of the above functions.

There is one other function that will be used throughout the program. This function tests to see if a cell is alive. You don't need to test to see if a cell is dead because if it is not alive it's dead. That's pretty much how things work both inside and outside of the game. So, you only need to implement one function to test the cell state:

```
isAlive(cell)
```

The function returns a *Boolean* value – *true* if the parameter contains the *live* value and *false* if it does not. It is important from a program style aspect that this function determines its return value by calling the *getLiveValue* or *getDeadValue* function rather than using whatever actual value you used to indicate *alive* or *dead*.

In addition to the array elements having two states, the cells in the HTML grid must also have two states that indicate visually whether the cell is *alive* or *dead*. The best way to do this is to use two different background colors and write two functions that will return this background colors:

```
function getLiveColor()
function getDeadColor()
```

Again, as long as the two colors are different, it will be OK. My *getDeadColor* function returned *white* but the choice is up to you. The return value is a *String* and can be an HTML named color, an RGB color, or a hex color value. It's up to you.

Step 3: Neighboring Cells

The state of a cell (*alive* or *dead*) is solely dependent on the state of the *neighbors* of that cell. A neighbor cell is one that "touches" the cell. Here are a few examples using the following grid:

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p
q	r	s	t

Cell	**It's Neighbors**
a	b, e, f
f	a, b, c, e, g, i, j, k
k	f, g, h, j, l, n, o, p
m	i, j, n, q, r
t	o, p, s

As you can see, a cell may have as few as three neighbors or as many as eight neighbors. The number of neighbors depends on where the cell is located in the grid. Those located on a border have fewer neighbors that those located in the interior of the grid.

If you study the problem for a bit, I think you will see that the neighbors of a cell can be identified by looking at the cells in three rows and three columns and noting that a cell cannot be a neighbor of itself and a neighbor must be within the grid. To help determine the latter criteria, write a function with this function header:

```
function isInArray(array2d, row, col)
```

The function is passed a reference to a two dimension array, a row number, and a column number. The function returns *true* only if the row number and column number are valid subscripts for the two dimension array. In all other cases, the function returns *false*.

The function can be written as a single statement but be sure you test it in such a way that all possible invalid value ranges are covered in the test. Create a temporary two-dimension array in your *window.onload* function and test this function..

Step 4: Some Two Dimension Array Functions

You may already have written functions that create and copy two dimension arrays. If you have, include them in the program now. If not, then write the following functions:

```
function create2dArray(rows, columns, initialValue)
```

This function creates a rectangular two-dimension array and returns a reference to that array. The array is created with *rows* rows each of which has *columns* columns. Each element of the array is initialized to the value in *initialValue.*

```
function copy2dArray ( array )
```

The function is passed a two-dimension array. The function creates and returns a reference to an identical copy (a duplicate) of the parameter array (same shape, size, and element values)

Test both functions thoroughly.

In the *window.onload* function, create a *gameBoardArray* variable and a *tempArray* variable. Use one of the functions you wrote in this step to initialize the *gameBoardArray* to be a 100 by

100 array whose elements all are set to the *dead* value. Use the other function you wrote in this step to place a copy of the *gameBoardArray* in *tempArray*. Test.

Step 5: Creating the HTML
The *gameBoardArray*, the *tempArray*, and the grid of *div* elements that will become the *innerHTML* of the *HTML gameBoard* element all have the same shape and size. You have written a function that creates a single *HTML* element (*createHTMLElement(elementType, id, classInfo, content*) and you will use that function when implementing the logic to create the *innerHTML* of the gameBoard element.

Write a function with this header:

```
              function createGameBoard(containerElement, array2d)
```

The function will eventually be passed a reference to the *gameBoard* element and will set that element's *innerHTML* to the grid of *div* elements the function creates. The size of the grid is based on the number of rows and columns (assume that each row has the same number of columns) in the parameter array. Each *div* must have an unique *id* based on the row and column in which it appears. For example. *row6col8* or, as I did, *r6c8*. Remember that each cell will be associated with exactly one element of the two-dimension array.

Test this function by placing this statement after the *copy2dArray* statement in the *window.onload* function:

```
     createGameBoard(document.getElementById('gameBoard'), gameBoardArray);
```

It will be easier to test your function if you temporarily change the size of the *gameBoardArray* to be 10 by 10 rather than 100 by 100. The grid that is display in the browser after the *createGameBoard* function returns should have a single border around each cell and there should be the same number of cells in each row.

Test thoroughly. When the function works, restore the 100 by 100 size to the *gameBoardArray*.

Step 5: Initial Configuration
The initial configuration of the game determines what happens in the remainder of the game. This initial configuration is called the first generation. Generation$_i$ is derived solely from the configuration that was produced by generation$_{i-1}$. So, essentially, the entire game is based on the initial configuration. It will be handy to have a function that will set the initial configuration. Here is an implemented version of the function. Once you get the program working, play around with this implementation to see what patterns emerge.

```
function createFirstGeneration( array2d )
{
 var i;
 var j;
 var k;
 var row;
 var col;


 for(i=0; i<array2d.length; i++)
  for(j=0; j<array2d[i].length; j++)
   {
    if(i === j|| i== j || (i+j)%2===0) {array2d[i][j] = getLiveValue();}
   }
}
```

Place a call to this function in the *window.onload* function immediately after the call to the *createGameBoard* function. Once this is done, you need to transfer the game state from the *gameBoardArray* into the *HTML* page. Immediately following the call to *createFirstGeneration*, write the code that will set each of the *divs* in the *HTML* grid to the *living color* or the *dead color* based on the setting of the corresponding element in the *gameBoardArray*. Test it out.

Step 6: Finally – Playing the Game

Finally, it is time to write the code that will play the game. The rules are simple and are based on the state (*alive* or *dead*) of a cell and that cell's living neighbors. The state of a cell is determined at the same time for all the cells. That is, you must identify all cells for which the rules will change a cell's state before you actually change the state of any those cells. This is the reason that there are two arrays. One is used to determine the new state for each cell and the other is used to store that new state.

Here are the rules for changing the state of a cell:

- A cell that is currently alive changes to dead if that cell has fewer than two living neighbors.
- Any cell that is currently alive, stays alive if it has two or three living neighbors.
- Any cell that is currently alive changes to dead if that cell has more than three living neighbors.
- Any cell that is currently dead changes to alive if that cell has exactly three living neighbors.

Since the number of living neighbors is essential in order to determine whether a rule should be applied, it will be useful to write a function with this function header:

```
function countLivingNeighborsOf(array2d, row, col)
```

The function is passed a two-dimension array, a row number, and a column number. The row and column identify a cell in the array. The function returns the number of neighbors of this cell that are *alive*. Remember, that cell references outside of the grid are considered to be cells that are dead.

Make certain that your function is identifying the correct neighbors of the cell. Test this by using cells that are on the edge of the grid as well as those inside the grid. Test thoroughly. This function is essential to have the game rules applied.

Once the function works, it is time to write the last function – the one that implements the game rules. The function should be named:

```
function applyRules(array2d, tmpArray)
```

The function is passed two two-dimension arrays. The arrays are the same shape and contain the same number of elements. The parameter *array2d* contains the current state (*alive* or *dead*) of each cell in the game. The function does the following:

- Store in $tmpArray_{ij}$ the count of the number of living neighbors for $array2d_{ij}$
- Modify the state (*alive* or *dead*) of each cell of the *array2d* array by applying the rules.
- Update each HTML *div* with the color that reflects whether that div is alive or dead.

After you have tested this function, write, as the last statement in the *window.onload* function, the statement that will call this function every second passing as actual parameters the *gameBoardArray* and the *tempArray*. This should animate the display and you should be able to watch the patterns change until they stabilize (or not, depending on the initial configuration).

Once it is working. Investigate other initial configurations. Each will produce a differnt result. The net has many examples.

What You Must Turn In
Nothing