Computer Science Correspondence School

discere domi

Lecture 24

CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Magic Square

Due: **Firday, May 15, 2020, at start of class.**
Late assignments will not be accepted, so start early
Points: **20**

Solution to Problems from Lecture 23

Problem 1:

Write a function with this function header:

```
function findLastElement (twoDimenstionArray)
```

The function is passed a reference to a two dimension array. You can assume that the reference is never *null*. The function returns the row number that contains the last element of the array. If the array contains no elements, the function returns *NaN*. You may only use one loop and may not call any other functions that you have written.

```
function findLastElement ( array2D )
{
 // return row number containing last element or NaN
 var i;

 for(i=array2D.length-1; i>=0 && array2D[i].length<1; i--) {}
 if ( i < 0 ) i = NaN;
 return i;
}
```

Problem 2:

Write a function with this function header:

```
function sameShape ( array2Da, array2Db )
```

The function is passed a reference to two two-dimension arrays. You can assume that the references are never *null*. The function returns Boolean *true* if the two arrays have the same shape and Boolean *false* if they do not have the same shape. The shape is the same if the two arrays have the same number of rows and there are the same number of columns in the i_{th} row of each array for every valid subscript *i*.

```
function sameShape ( array2Da, array2Db )
{
 // return true if the two parameter arrays have the same shape
 var i;
 var result;

 result = (array2Da.length === array2Db.length);
 for(i=0; i<array2Da.length && result; i++)
 {
 result = array2Da[i].length === array2Db[i].length;
 }
 return result;
}
```

Problem 3:
Write a function with this function header:

```
function function copy ( array2D )
```

The function is passed a reference to a two dimension array. You can assume that the referenceis never *null*. The function returns a copy of the array. The copy must have the same shape and values in the same locations as the parameter array.

```
function copy ( array2D )
{
 // return a copy of the parameter array
 var i;
 var j;
 var result;

 result= new Array ( array2D.length );
 for ( i=0; i<result.length; i++ )
 {
  result[i] = new Array ( array2D[i].length );
  for ( j=0; j<result[i].length; j++ )
  {
   result[i][j] = array2D[i][j];
  } // for ( j
 }  // for ( i

 return result;
}
```

Problem 4:
Write aanother version of the two dimension array *copy* function. In this version, make use of the single dimension array *copy* function you wrote when single dimension arrays were introduced.

```
function copy ( array2D )
{
 // return a copy of the parameter array
 var i;
 var j;
 var result;

 result= new Array ( array2D.length );
 for ( i=0; i<result.length; i++ )
 {
  result[i] = copySingleDimensionArray ( array2D[i] );
 }  // for ( i

 return result;
}
```

Start of Today's Lecture/Assignment

In this assignment you will write a system that will produce magic squares. A magic square consists of *n* rows and *n* columns, where *n* is an integer greater than zero and $n \neq 2$. The value of *n* is called the *order* of the square. The intersection of each row and column contains an unique integer between one and n^2. The integers are placed in such a way that the sum of the integers in any row or column or major diagonal is the same value. This sum is called the *magic sum*. Here is an example of an *order* three square with *magic sum* of fifteen.

8	1	6
3	5	7
4	9	2

There can be several magic squares that have the same *magic sum*. Here is a different *order* three square:

2	7	6
9	5	1
4	3	8

The system you will write will ask the user to enter an odd integer that is greater than two[1]. The user entered value must be validated to ensure it is an integer and is greater than two. If the user makes an incorrect entry, an error message should be produced and the program should end. If the entry is correct, the program will produce a magic square using the entered value as the *order* of the square.

Your implementation must check subscript values for the correct range. You must never test to see if an array entry is *undefined*.

Step 0

You must name your *html* and *js* files using these names:

```
MagicSquare.html
MagicSquare.js
```

I need to have the files named consistently in order to run them after you send them to me.

Step 1

Since you will eventually be generating the *html* of the magic square in your Javascript program, it is important to determine exactly the content of the *html* statements the program will need to produce. To that end, create, by hand, the *html* and *css* necessary to produce an *order* five square. While each square must be assigned the same width and height, you must not assign a height or width to the element containing the magic square. This is because the program can generate squares of any size and you will not be able to modify your *css* each time the program produces a square. Use the techniques we have used before. You may use either *divs*, *tables*, or both in your implementation. Do what you think works best.

Step 2

[1] Magic squares of order one are not very interesting, so the program will only create squares of an odd *order* that is three or greater.

In the *window.onload* function, ask the user to enter an odd integer greater than two. If the value entered does not meet these criteria, then display an error message otherwise display a message saying that the value was correct. This latter message is a temporary statement that must be removed after you have tested this part of the program. Test the program with various numeric and non-numeric values including text, non-integers, integers less than one, and integers equal to two.

You can test the value of *x* to determine if it is an integer, by using the functions you have previously written. Once everything is working, remove the *value is correct* message and replace it with a call to the function you will write in Step 3.

Step 3

Write a function called:

```
function createMagicSquare ( containerElement, order )
```

that is passed two values: a reference to the *html* element that will contain the magic square and an integer that is the *order* of the square that your program will produce. The function does not return anything. In order to create the *html* for the magic square, it first creates a *two dimension array* that will be used to hold the integer value of each square.

Recall that single dimension arrays can be thought of as consecutive values that are accessed via an integer offset from the first element in the array. They are enormously useful and occur in many programs. Rectangular two dimension arrays form a grid of elements. The grid contains *n* rows and *m* columns so there are *n* multiplied by *m* elements in the array. A two dimension array with three rows and four columns might look like this:

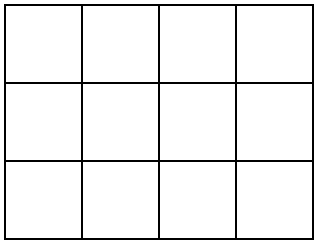

Figure 1

Quick Review of Two Dimension Array Concepts

Recall that you access an element of a two dimension arrays by specifying a subscript value for the row and a subscript value for the column. Here is how the element at row two and column one would be accessed if the array were name *theArray*.

```
theArray[2][1]
```

Other than requiring two subscripts to access an element, all other aspects of the two dimension array are the same as those for a single dimension array.

Javascript, like many modern languages, does not directly implement two dimension arrays. Instead, like other modern languages, a two dimension array is simulated by having each element of a one dimension array contain a reference to another one dimension array.

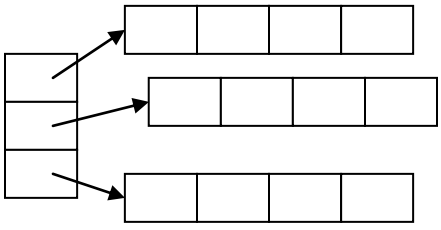

Figure 2

Figure two shows how the array in Figure one is implemented in Javascript. Each box in the leftmost column represents a row element. The row element points to a single dimension array that has the same number of elements as there are columns in the two dimension array. These "column" arrays need not be the same length, although they frequently are. So, creating a two dimension array has two steps: first create the array of rows and then, for each row, create an array of columns.

When accessing the *length* property of a two dimension array, *arrayName.length* is the number of rows; *arrayName[i].length* is the number of columns in row *i*. Since the number of columns is not necessarily the same for each row of the array, it is important to access the number of columns in a row by using the *arrayName[i].length* approach whenever you need to determine the number of columns in a row *i*. Lets say that you wanted to create a two dimension array that had four rows each of which had seven columns. The array will contain twenty-eight elements. Here is the Javascript that would do that:

```
var i;
var sampleArray;

sampleArray = new Array(4);
for(i=0; i< sampleArray.length; i=i+1) { sampleArray [i] = new Array(7); }
```

How many elements are in the array created by the following statements?

```
var i;
var sampleArray;

sampleArray = new Array(4);
for(i=0; i< sampleArray.length; i=i+1) { sampleArray [i] = new Array(i); }
```

Row zero contains zero columns. Row one contains one column. Row two contains two columns. Row three contains three columns. That is a total of six elements in the array.

Initializing the elements of a two dimension array is usually done using nested loops. In languages that support *typed* variables (e.g., Java, C++), the elements of arrays are often initialized to some default value such as zero or *null*. Since Javascript does not have *typed* variables, the elements of all Javascript arrays (single dimension or multi-dimension) are initialized to *undefined*. Thus, it is the programmer's responsibility to initialize the elements of every array.

Here is how to initialize each element of the above *sampleArray* to zero:

```
var i;
var j;
for(i=0; i< sampleArray.length; i=i+1)
{
 for(j=0; j< sampleArray[i].length; j=j+1){ sampleArray[i][j] = 0; }
}
```

The first subscript is the row number and the second subscript is the column number.

Applying the Two Dimension Array Concepts

Back to the requirements for writing the *createMagicSquare* function. The function creates a square shaped (each row has the same number of columns), two dimension array using the integer passed via the order parameter. The value of each element of this array is then initialized

to zero. The values of the array elements are then modified to be those of a magic square by following this algorithm developed by *de la Loubère*[2]:

In this algorithm, the *current square* is the one in which you placed the most recent number. Use the algorithm to draw some magic squares on paper before trying to program it. This will help you understand the algorithm.

> Place 1 in middle of the first row.
>
> Write successive integers in an upward-right diagonal path, with the following special cases:
>
> > If this upward-right movement would result in a location outside the boundaries of the array, place the new number at the opposite end of the row or column that would have contained the new number if the rows and columns were not bounded. (In the case of the *current square* being in the topmost, rightmost corner, we would try to place the next number in the lowest, leftmost corner.)
> >
> > If the upward-right square is already occupied, place the new number directly below the *current square*.

After the array has been filled, the function builds the *html* for required rows and columns for the square using the values in the array as the data for the squares. Finally this *html* is used to set the *innerHTML* of the *containerElement* parameter. It might be a good idea to simply display the contents of the array until you are sure your algorithm works and then remove the display of the array and use the array to generate the *html*.

The function should build this square when three is entered by the user:

8	1	6
3	5	7
4	9	2

This square will be built when five is entered by the user:

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

You should be able to create magic squares of any odd integer *order* although, at some point, the size of the number inside each box will exceed the width you have specified in your *css*. There is no need to correct this. Make sure you can demonstrate squares of order three, five, seven, nine, and eleven.

[2] The Wikipedia site has examples showing squares being built using this algorithm.

What You Must Hand In

- Send me an email with this subject line:

  CISC131-MagicSquare-*YourLastName*
- Attach the **MagicSquare.html** file and the **MagicSquare.js** file to the email message and send it to me.
- Do NOT attach any other files.
- Do NOT put things in the cloud or one-drive.
- Do not zip up the Javascript file.