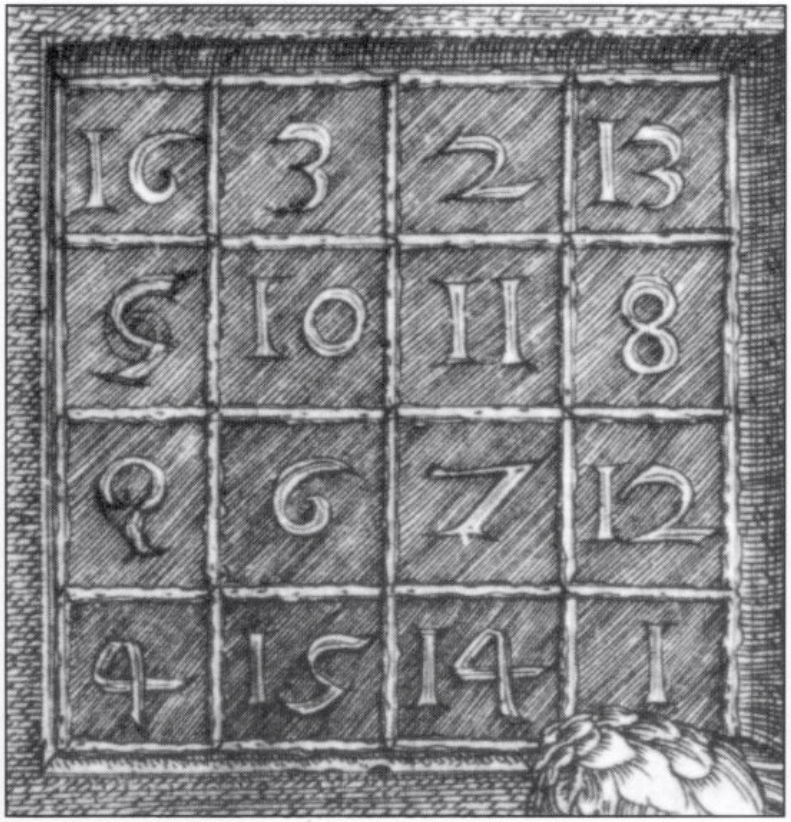
Consider the intricate engraving below by Albert Dürer entitled Melencolia I, created more than 500 years ago in 1514 AD. A print of this engraving hangs in the New York Metropolitan Museum of Art. Much has been written about the individual items depicted in the engraving and the overwhelming sense of melancholy (anger perhaps?) exhibited by the main character.

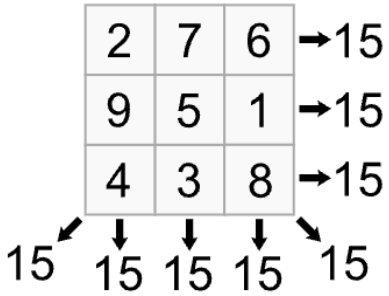
Notice in particular the 4 x 4 grid of numbers in the upper right (under the bell). Here is an enlargement of it.



The author cleverly included the year (and perhaps the month and day) of his work in the bottom row.

Interestingly, the sum of each row, column, and main diagonal add up to the same number: 34. This is an example of a Magic Square, a square array of numbers where the sums of the numbers in each row, column, and both main diagonals are the same. We call this number the **magic constant** of a Magic Square. The magic constant may vary between different Magic Squares.

Here is another example of a Magic Square, where the magic constant is 15.



In this project, you will be working with 2D arrays of integers to determine if a given array is a Magic Square or not. It is not as easy as it sounds; in order to accomplish this, you will be completing a dozen methods that will enable you to finally make a determination. The arrays you will be testing will be of different sizes and shapes (some Rectangular, some Square).

To get started download, extract, and save the Magic Square project files to your computer. Then work through the following instructions carefully to complete each of the required methods. Note that you can test your work by running `MagicSquareTest` to see how you are doing.

When you are finished, please submit a screenshot of the `MagicSquareTest` success message. Good luck. I hope you enjoy this project as much as I did in creating it. I find the topic fascinating.

---

1. Start by inspecting the `MagicSquare` class. Notice (in the comments at the top) the definitions of a ***Square*** array and a ***Rectangular*** array. Observe the dozen methods that you will be completing. All of the methods return something trivial (meaningless) at the moment so that the code will compile. You will be replacing the existing return statement in each method as you complete it.

2. Begin with completing the `isSquareArray` method. This method determines if a given array is square (i.e. having the same number of rows and columns) or not. How can you determine the number of rows in a 2D array? How can you determine the number of columns? Just return whether the two values are the same or not.

   Be sure to check your work by running `MagicSquareTest` before you continue.

3. A non-trivial Magic Square has unique numbers (no duplicates) in the array. The next two methods will help you determine if a given number appears only once in the array. We'll do this by selecting a number from the array, and then searching through the entire array from start to finish to see that the number appears exactly once. We'll do the second part first, counting the number of times a given number appears in the array.

   Complete the method `isNumberUnique` which counts the number of times `number` appears in `array`. Start by setting up a counter, and then use nested loops to go through the entire array from start to finish, looking to see if any individual element in `array` equals `number`. If it does, count it. Then when the loop is finished, return `true` if the count is 1, or `false` otherwise.

   Note that some of the arrays tested are Rectangular where the number of rows and columns are different. Be sure your algorithm tests *all* of the elements of the array.

   Be sure to check your work by running `MagicSquareTest` before you continue.

4. Next, complete the `isEveryNumberUnique` method. Start by setting up nested loops to access each element in `array`, one at a time. Grab an individual `number` and use it to call the method you just wrote, `isNumberUnique(array, number)`. If `isNumberUnique(array, number)` ever returns `false`, then just return `false` for `isEveryNumberUnique`. If you get through the *entire* array, and `isNumberUnique(array, number)` keeps returning `true`, then `isEveryNumberUnique` should return `true`.

   Now we can use the methods `isSquareArray(array)` to make sure `array` is square, and `isEverNumberUnique(array)` to make sure that every number only appears once before we even start to add up the rows and columns.

   Be sure to check your work by running `MagicSquareTest` before you continue.

5. Next, complete the `colSum` method, that adds up a given column of `int` values in a 2D array, and returns the total. Notice that the column does not change, but the row does. (Only one loop is required.) How can you find out how many rows are in a 2D array? Loop through each row, totaling up each element in the given column, and then return the total.

   Be sure to check your work by running `MagicSquareTest` before you continue.

6. Now it's time to complete the `rowSum` method, that (obviously) adds up a given row of `int` values in a 2D array, and returns the total. Notice that the row does not change, but the column does. (Again, only one loop is required.) How can you find out how may columns are in a 2D array? Be careful, here, because the number of columns may be different than the number of rows, yet the number of columns will be the same for each row (e.g. Rectangular array). Loop through each column, totaling up each element in the given row, and return the total.

   Be sure to check your work by running `MagicSquareTest` before you continue.

7. A Semi-Magic Square is *almost* a Magic Square, except that one or both of the main diagonals don't necessarily add up to the magic constant. In other words, a Semi-Magic Square has the following attributes.

   - It is a Square array
   - Every number appears only once
   - Each of the rows and columns adds up to the same number

Now we have the tools in place to determine if a 2D array is a Semi-Magic Square or not. Complete the `isSemiMagicSquare` method by seeing if the given array satisfies all three attributes. ***Do not reinvent the wheel.*** Use the methods you just wrote to determine the proper result.

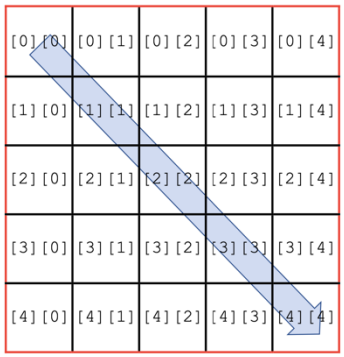
- Start by determining that both `isSquareArray(array)` *and* `isEveryNumberUnique(array)` are `true`.
- Next, declare a local variable and assign it the value of the `rowSum` of the first row. Now you have a possible magic constant to check everything else against. Any time a subsequent call to `rowSum` or `colSum` does not equal that saved value, you know you don't have a Semi-Magic Square, and the method immediately returns `false`.
- Loop through all of the rows (even the first one again), calling `rowSum` for each row. If the sum returned is ever different than the saved value, then return `false`.
- Same thing for the columns, calling `colSum` for each column. If the sum returned is ever different than the saved value, then return `false`.
- If you have gotten through all of the above tests successfully, then `array` must be Semi-Magic Square (return `true`).

Be sure to check your work by running `MagicSquareTest` before you continue.

Congratulations, you have completed the first half of the project.

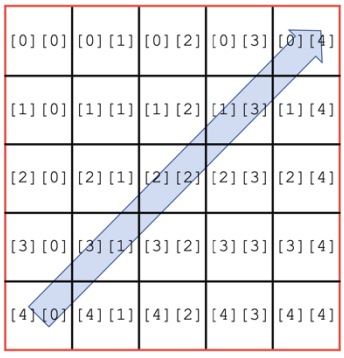
8. Now it is time to sum the diagonals so we can see if we have a true Magic Square. The first diagonal is easy (upper-left to lower-right). It is the one where the row and column indexes are the same (e.g. [0][0], [1][1], [2][2], etc.). Complete the method `sumFirstDiagonal(square)`. (Note the method precondition that `square` is Square.) Loop once through the rows (or columns, it does not matter because they are the same value) totaling the elements on the first diagonal, and return the total.

   Here is a diagram of the elements to be totaled for a 5x5 array, but your algorithm must work for a Square array of any size.



   Be sure to check your work by running `MagicSquareTest` before you continue.

9. At first, it seems like finding the coordinates (in the form of [row][column]) for the second diagonal (lower-left to upper-right) might be challenging. Specifically, for a 5x5 array, the second diagonal coordinates would be [4][0], [3][1], [2][2], [1][3], [0][4]. Notice that as the row index decreases, the column index increases.



   Perhaps you wrote the last method something like the following

```
int sum = 0;
for(int i = 0; i < square.length; i++)
   sum += square[i][i];
```

In `sumSecondDiagonal(square)`, we want the *last* row and the first column, and then the second-to-last row and the second column, etc. What is the index of the last row in a square array?

In other words, loop in the same manner as above, but instead of using `[i]` for the row index (going from top to bottom), work backward (going from bottom to top) by using `[square.length - 1 - i]`.

There, that's it. Easy peasy, lemon squeezy.

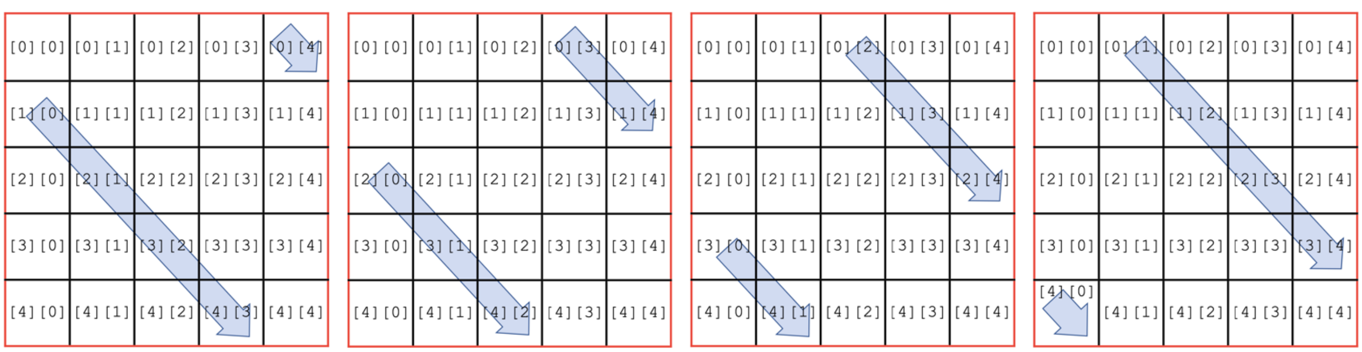
Be sure to check your work by running `MagicSquareTest` before you continue.

10. A Magic Square is just a Semi-Magic Square plus checking the two diagonals. No need to check if the array is Square or if the numbers are unique or if the row sums and columns sums agree. The test for a Semi-Magic Square does all of that already. (Again, do not reinvent the wheel.)

    So, in order to complete the `isMagicSquare` method, first check that array is a Semi-Magic Square. If it is, then save the first row totals (as you did in `isSemiMagicSquare`), and compare that value to the sum of both diagonals. If they agree, then, voilà! You have a Magic Square.

    Be sure to check your work by running `MagicSquareTest` before you continue.

11. But wait, there's more! In a square array, there is more than 1 diagonal in each direction to be considered. Here we introduce the concept of a ***broken diagonal***. In the examples below, the broken diagonals fall off the bottom of the grid and wrap back around to the top of the grid to finish the journey. In a square n x n grid, there are n – 1 broken diagonals *in each direction* ($2n$ – 2 total). For example, here are the broken diagonal in the first direction for a 5 x 5 grid. Note that each one starts in the first column and then goes through each of the remaining columns, always inspecting n elements on the diagonal.



    Perhaps you noticed, but these diagonals are just the first primary diagonal plus a row offset. But what to do when the row index goes off the grid (so to speak) at the bottom? How do we get it to wrap around to the top row? Ah, we use modulus, my friend.

    If you take the row value and 'mod' it by the number of rows, when the row value goes out of bounds, it will reset back to row zero.

    Complete the overloaded method `firstDiagonalSum` that now includes a row offset parameter. As you did for the previous `firstDiagonalSum` method, only one loop is needed. Loop through the number of rows, using the same variable for both the rows and columns. This time, however, add the row offset to the row value, and then 'mod' this by the number of rows.

    In other words, the row value (if you are using `i` as an index) would be `[(i + rowOffset) % square.length]`. This way the row value will never go out of bounds.

    Be sure to check your work by running `MagicSquareTest` before you continue.

12. OK, now time to do the same thing in the other direction (lower-left to upper-right). Put together what you learned from writing the initial version of `secondDiagonalSum` and write the overloaded version of the method. Make sure you 'mod' the row value by the number of rows, just like you did for the previous method, so you won't go out of bounds.

The final calculation for the row value is a little bit convoluted: `[(square.length - 1 - i + rowOffset) % square.length]`.

Be sure to check your work by running `MagicSquareTest` before you continue.

13. We'll be using these last two methods to determine the pièce de résistance in the world of Magic Squares, a **Panmagic Square**, which is a Magic Square where the broken diagonals also add up to the magic constant.

    A Panmagic Square is just a Magic Square plus checking *all* of the broken diagonals. No need to check anything else. The test for a Magic Square does all of that already. (Again, do not reinvent the wheel.)

    So, in order to complete the `isPanMagicSquare` method, first check that array is a Magic Square. If it is, then save the first row totals (as you did before), and compare that value to the sum of all the broken diagonals. You can just loop through each row, providing the row number as the `rowOffset` parameter of both `firstDiagonalSum` and `secondDiagonalSum` methods. If everything agrees, then presto-chango! You have a Panmagic Square.

    Be sure to check your work by running `MagicSquareTest` before you continue.

That's all there is. Your code is now able to determine if a 2D array of integers is a Semi-Magic Square, a Magic Square, or even if it is a Panmagic Square. Pretty amazing, huh?

Congratulations. Please submit a screenshot of the bottom of the tester success message when you are finished.