Name: Maryfrances Umeora
BBID: mumeora
Email: mumeora@u.rochester.edu
TA/Grader: Linan Li

Note to Grader: Because the instructor said on Piazza that we can submit a pdf as a README, I decided to do so so that I can make my README more detailed and readable. Note that all the text in *blue* are not links, but variable names so you can keep track of which is a variable I used in my code.

README Submission Requirements
a description of your work ✓
team members Not Applicable
the choices for the parameters in Task 1 and 2 ✓
comments about the quality of the distribution ✓

In this assignment, there are two main parts. Please scroll all the way to the bottom in order to get everything.

Part I
I was to write Java code to compute the sequence of values $h(k)$ for any sequence of integers given in input, store the sequence computed in an output file, and create a hash table for the output and input, implementing a rehashing of the hash table when the load factor α reaches the value 0.75 ie 75% of the size.

The code is executed with `a b m input_sequence`, so I begin by setting the integers *a*, *b*, and *m* to `args[0], [1]` and `[2]`, and the String *infileArg* to `args[3]`.

Next, I generate my `input_sequence` file, which is attached in my `mumeora_Lab8.zip` folder I submitted. It is a text file with a sequence of numbers divided by blank spaces. The teacher said to use sequences of 300-1000 numbers, so my `input_sequence` contains one thousand random integers between 1 and 1000. You will notice that I have commented out the code that does this. This is because, for the purpose of this assignment, I do not want a new one generated every time, as my responses in Part II must be congruent with the numbers in the input file I submit.

The next step is to generate my output_sequence file using the numbers in `input_sequence` through the hash function $h(k) = (ak + b) \bmod m$. I believe that the way I did it is quite simple but effective.

First, of course, I create the empty text file called `output_sequence`. Then I created a PrintWriter called *ow* (it stands for **o**utput **w**riter) to write to that file. After that, I initialized a Scanner called *fileReader* that will read every integer in `input_sequence` one by one. Every time *fileReader* gets to an integer, the following will happen:

- We know that that integer is the *k* that will be put in the hashing function, so I save it to an int *k*.
- I then put this *k* through the hashing function, saving the answer $h(k)$ in another integer named *ans*. I then write *ans* to output_sequence using *ow*.
- Of course, I also wanted to keep track of what values map to what, so I also update my makeshift hashtable. Earlier, I had created an array of linked lists called *hashArr*. This is what I use to store my hashtable.
- I wrote my array-hashtable to deal with collision by method of chaining. If one value maps to more than one, I just create a linked list of those values. Looking over the code is fairly self-explanatory.
- And finally, I make sure to rehash in case the *loadfactor* has gotten to 0.75. I simply do this by doubling m everytime this happens.

Like I mentioned, this all happens for each and every integer in `input_sequence`, effectively creating a complete and accurate sequence of values in `output_sequence`. I even did some of the math myself to make sure they were right.

This concludes all the necessary work for Part I.

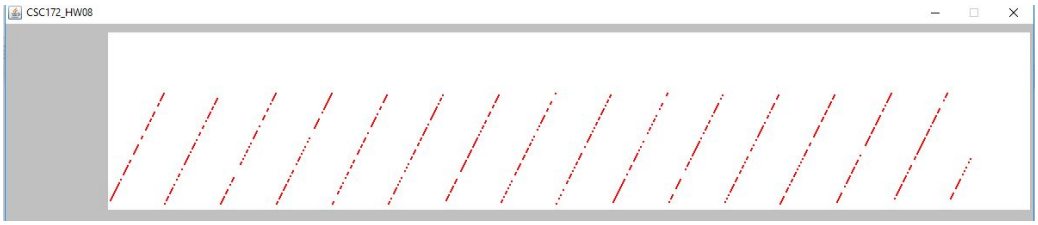
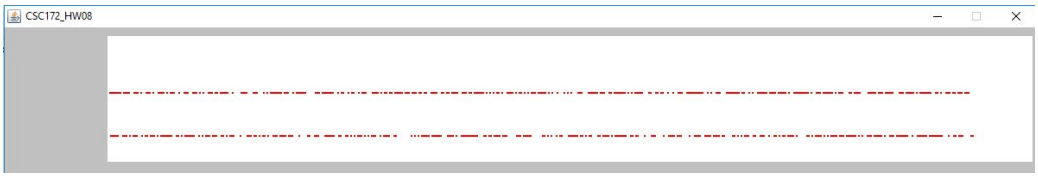
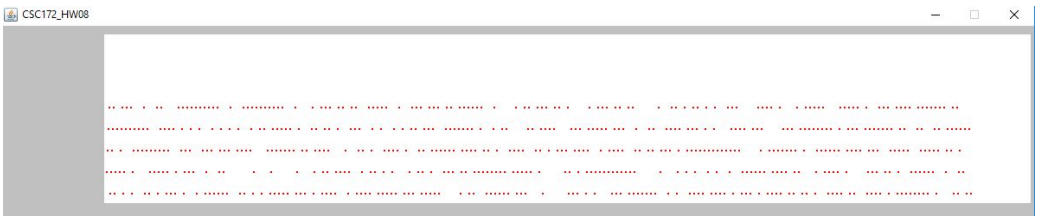
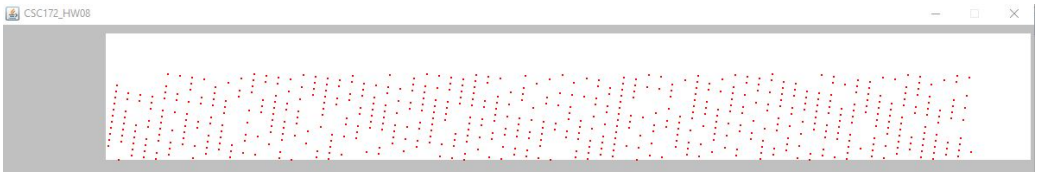
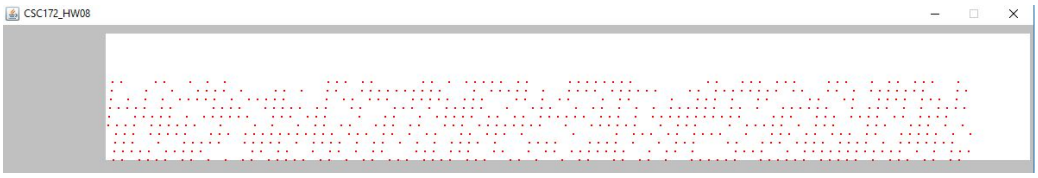
Part II

This part involves plotting the pairs $(k, h(k))$ to see how well/bad elements are distributed. Below, I will identify 3 choices for a, b and m, for which you think the distribution of values is bad **(Task 1)** and 3 for which I think they are good **(Task 2)**. I will also leave short comments about the quality of the distribution for each case in Task 1 and Task 2.

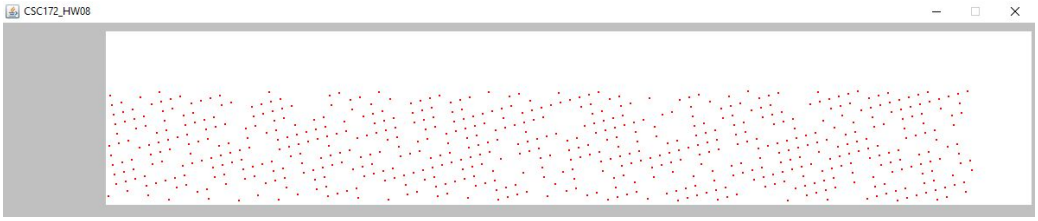
So the first thing of course is to plot the graphs. I wrote a method called drawGraphs (located above my main method) in order to handle this. I used Java GUI as suggested by an instructor on Piazza, so drawGraphs extends JComponent. In the paintComponent method, I loop through my hashtable and draw a small circle for every value. It's a lot of Math to get the dots to work with the dimensions of the table, but it works!

Maryfrances Umeora

Below are my results for Task 1 and 2, complete with the requested comments about the quality of the distribution. Of course, the images are also provided in PNG format, included in my submitted zip folder. I just chose to have them here as well.

Task 1 (Bad Distributions)	**Comments**
	$a = 2$, $b = 3$, $m = 130$ When I choose an m that is divisible by both a and b, I get bad, diagonal distributions like this one.
	$a = 150$, $b = 230$, $m = 100$ I also found that I seem to have even worse distributions should a and b be much bigger than m.
	$a = 25$, $b = 6$, $m = 125$ In the end, I realized that the more divisible a and m are with each other, the worse the distribution. b, if small enough, does not have much of an effect in the long run.
Task 2 (Good Distributions)	
	$a = 7$, $b = 9$, $m = 100$ A good distribution is more spread out, and I've found that this is more easily attainable should a and b be prime to each other.
	$a = 10$, $b = 30$, $m = 100$ Surprisingly however, I managed to get a good distribution with numbers that were multiples of each other. However, this is largely depended on the fact that b is larger than a and thus will greatly vary the results mod m.

Maryfrances Umeora

	$a = 58$, $b = 455$, $m = 127$ Ultimately however, we get the best values when a, b and m are largely different from one another.