Programming Assignment #2: Bonfire

COP 3330, Spring 2023

Due: Sunday, February 5, *before* 11:59 PM

Abstract

This assignment is designed to give you practice with some of the Java syntax we've learned so far this semester. More than that, however, the methods you have to write in this assignment will serve as exercises in critical thinking and creative problem solving while also encouraging you to adopt some of the coding principles covered in class, such as functional decomposition and DRY ("don't repeat yourself," also referred to in class as "never write the same code twice"). By completing this assignment, you'll also get some hands-on experience with test-driven development. You're encouraged to generate additional test cases for all the methods in this assignment.

Historical Note: When I started drafting this assignment, I was originally calling it "Warmup," but I ended up renaming it to "Bonfire" in order to instill more of a sense of urgency and danger (because I expect that many people will find some of these methods fairly challenging, so I want everyone to start early).

Deliverables

Bonfire.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Method and Class Requirements

Implement the following methods in a public class named *Bonfire*. Please note that they are all ***public*** and ***static***. You may implement helper methods as you see fit (unless explicitly forbidden for certain methods). Please include your name, the course number, the current semester, and your NID in a header comment at the very top of your source file.

```
public static boolean containsValueAfterIndex(int needle, int [] haystack, int index)
```

Description: Determine whether *needle* occurs in *haystack* at any cell with an index greater than *index*. If so, return *true*. Otherwise, return *false*. For example, if our *needle* is 9, our *haystack* is {3, 5, 3, 9, 6}, and our *index* is 3, this method should return *false*, since there are no occurrences of the value 9 at any index greater than 3 in that array. However, if our *needle* is 6 with the same *haystack* and *index*, we should return *true* because there is an occurrence of the value 6 at an index greater than 3 (index 4).

Special Restrictions: For this particular method, you must abide by all the following restrictions:

- You cannot create any new arrays in this method.
- You cannot sort the array, and you cannot change the contents of the array passed to this method at any point (even if you change that array back before returning from this method).
- You cannot call any helper methods (whether built in to Java or hand-crafted by you); all the work for this problem must be contained within this method, with no additional method calls.

The TAs will manually inspect your code to ensure that you are abiding by the above restrictions. Failure to abide by these restrictions will result in catastrophic point loss, even if you are passing test cases.

Parameter Restrictions: We will never pass a *null* parameter to this method. However, the array we pass to this method could be empty (i.e., it could contain zero elements). Furthermore, we make no promises whatsoever regarding the values we might pass to this method for *needle* and *index*, other than the fact that they will indeed be integers.

Output: This method should *not* print anything to the screen. Printing stray characters to the screen (including newline characters) is a leading cause of test case failure.

Related Test Case: *TestCase01.java*

Return Value: This method should return either *true* or *false* as described above.

```
public static int getThirdLargest(int [] array)
```

Description: Return the third largest integer in *array*. Note that if there are duplicate elements within the array, each occurrence should count as a separate element. For example, 4 is the 2nd *and* 3rd largest element in the following array: {2, 4, 9, 4}. Additional examples are given in the test cases released with this assignment, which you should read carefully.

We guarantee that the array passed to this method will not be *null*. However, it may contain fewer than three elements. If this method receives an array with fewer than three elements, it should return *Integer.MIN_VALUE*.

Special Restrictions: For this particular method, you must abide by the same special restrictions articulated for the *containsValueAfterIndex()* method above (pg. 2). The TAs will manually inspect your code to ensure that you are abiding by the above restrictions. Failure to abide by these restrictions will result in catastrophic point loss, even if you are passing test cases.

Parameter Restrictions: We will never pass a *null* parameter to this method. However, the array we pass to this method could be empty (i.e., it could contain zero elements).

Output: This method should not print anything to the screen. Printing stray characters to the screen (including newline characters) is a leading cause of test case failure.

Related Test Case: *TestCase02.java*

Return Value: This method should return a single integer as described above.

```
public static void printThirdLargest(int [] array)
```

Description: This method has the same behavior as *getThirdLargest()*, except it does not return a value; instead, it prints the result to the screen. Your output should contain a single integer, followed by a newline character ('\n').

Special Restrictions: For this particular method, you must abide by all the following restrictions. Please note that this method has one less restriction than the *getThirdLargest()* method:

- You cannot create any new arrays in this method.
- You cannot sort the array, and you cannot change the contents of the array passed to this method at any point (even if you change that array back before returning from this method).

The TAs will manually inspect your code to ensure that you are abiding by the above restrictions. Failure to abide by these restrictions will result in catastrophic point loss, even if you are passing test cases.

Parameter Restrictions: We will never pass a *null* parameter to this method. However, the array we pass to this method could be empty.

Output: See above for a description of this method's output. Also, see the test cases and sample output included with this assignment for further examples of this method's output. Your output must match our sample output files *exactly*.

Related Test Case: *TestCase03.java*

Return Value: This is a *void* method and therefore should not return a value.

```
public static boolean isRotation(int [] array1, int [] array2)
```

Description: We say that an array is a rotation of another if we can attain one array by shifting the elements of the other array by some number of places, wrapping elements back around to the beginning of the array as they fall off the end of that array. For example, let us shift the elements of the array {1, 7, 2, 3} one by one to show all possible arrays that are rotations of this one:

{3, 1, 7, 2}
{2, 3, 1, 7}
{7, 2, 3, 1}
{1, 7, 2, 3}

Notice that the array is a rotation of itself. This can be achieved by shifting all elements 0 places to the right, 4 places to the right, -4 places to the right (which is equivalent to shifting 4 places to the left), and so on. Notice also that this is a symmetric relationship. So, if *array1* is a rotation of *array2*, it necessarily follows that *array2* is a rotation of *array1*, and vice versa.

Given two arbitrary arrays, *array1* and *array2*, determine whether they are rotations of one another. If so, return *true*. Otherwise, return *false*.

Special Restrictions: For this particular method, you must abide by the same special restrictions articulated for the *containsValueAfterIndex()* method above (pg. 2). The TAs will manually inspect your code to ensure that you are abiding by the above restrictions. Failure to abide by these restrictions will result in catastrophic point loss, even if you are passing test cases.

Parameter Restrictions: The arrays we pass to this method will not be *null*, although one or both of them might be empty.

Output: This method should _not_ print anything to the screen. Printing stray characters to the screen (including newline characters) is a leading cause of test case failure.

Related Test Case: *TestCase04.java*

Return Value: Return *true* if the arrays are rotations of one another. If not, return *false*.

```
public static int [] generateNthRotation(int [] array, int n)
```

Description: Return a new integer array that contains the *n*th rotation of the integer array passed to this method. The *n*th rotation of an array is achieved by shifting all elements *n* places to the right and wrapping elements back around to the beginning of the array as they fall off the end of the array. For example, the first rotation of {1, 7, 2, 3} is {3, 1, 7, 2}. Similarly, the second rotation of {1, 7, 2, 3} is {2, 3, 1, 7}, the -2nd (*_negative_* second) rotation of {1, 7, 2, 3} is {2, 3, 1, 7}, and the fourth rotation of {1, 7, 2, 3} is simply {1, 7, 2, 3}.

Special Restrictions: For this particular method, you must abide by all the following restrictions:

- You can only create one new array in this method.
- You cannot directly modify the contents of the array passed to this method at any point (even if you change that array back before returning from this method).
- You cannot call any helper methods (whether built in to Java or hand-crafted by you); all the

work for this problem must be contained within this method, with no additional method calls. (Note that you will need to use the "new" keyword to create a new array. That's fine. That does *not* count as a method call.)

- ***Important!*** You can only write one loop in this method. You cannot write multiple loops, and you cannot have any nested loops.
- ***Important!*** The loop you write can only perform *array.length* number of iterations, even if *n* is greater than *array.length*! So, for example, if *array* contains 5 elements and *n* is 1,000,000, you must find a way to generate the 1,000,000th rotation of *array* using a loop that *only* performs five iterations! This restriction does *not* apply in the other direction, though: if *array.length* is 1,000,000 and *n* is 5, the loop you use to generate the 5th rotation of that array is allowed to perform 1,000,000 iterations.

The TAs will manually inspect your code to ensure that you are abiding by the above restrictions. Failure to abide by these restrictions will result in catastrophic point loss, even if you are passing test cases.

Parameter Restrictions: We will never pass a *null* parameter to this method. However, the array we pass to this method could be empty.

Output: This method should *not* print anything to the screen. Printing stray characters to the screen (including newline characters) is a leading cause of test case failure.

Related Test Case: *TestCase05.java*

Return Value: Return an integer array as described above. Note that you might have to return an empty array (an array of length zero). That shouldn't be problematic; Java will allow you to create arrays of length zero. This method should never return *null*.

```
public static void printRibbon(int n, int width)
```

Note: This is an advanced problem! Don't feel bad if this one takes you a while! This might be easier if you first review the *printDiamond()* method in the Webcourses notes from Jan. 20, as well as the *printChevron()* exercise on that page.

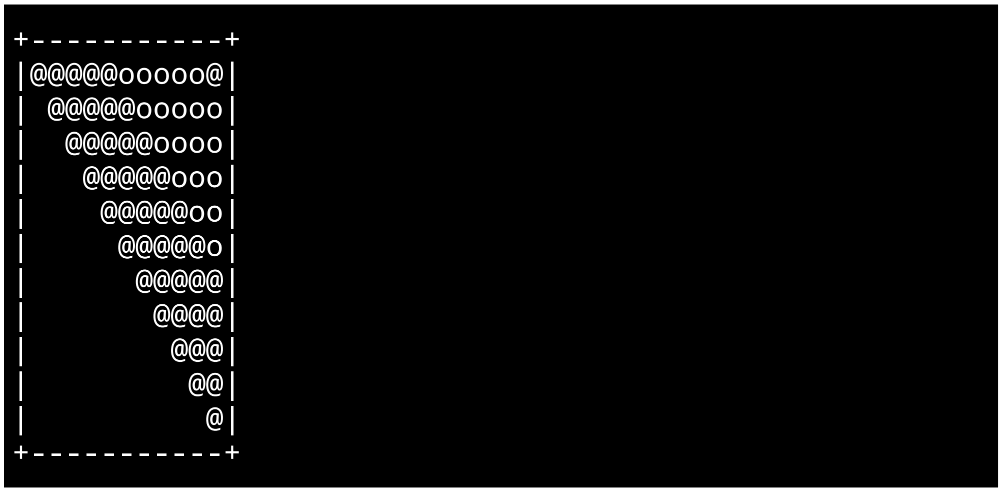
Description: This method takes two integers (*n* and *width*) and produces the following output:

1. The first line contains *n* '@' symbols, followed by *n* 'o' symbols (lowercase letter 'o'), followed by *n* '@' symbols, followed by however many spaces are necessary to cause a total of *width* symbols to have been printed for this line. As we print the characters as described, we should always terminate prematurely as soon as we have printed *width* number of characters for this line.
2. The second line is the same as the first line of output, except it's shifted to the right by one space (i.e., it starts with a single space character and then follows the pattern described for the first line of output); the third line is the same as the second line of output, except it's shifted to the right by yet another extra space (i.e., it starts with two spaces and then follows the pattern described for the first line of output); and so on.

3. This should continue until the last line of output, which will contain a single '@' symbol (possibly preceded by several spaces in order to achieve the full *width* of this line).

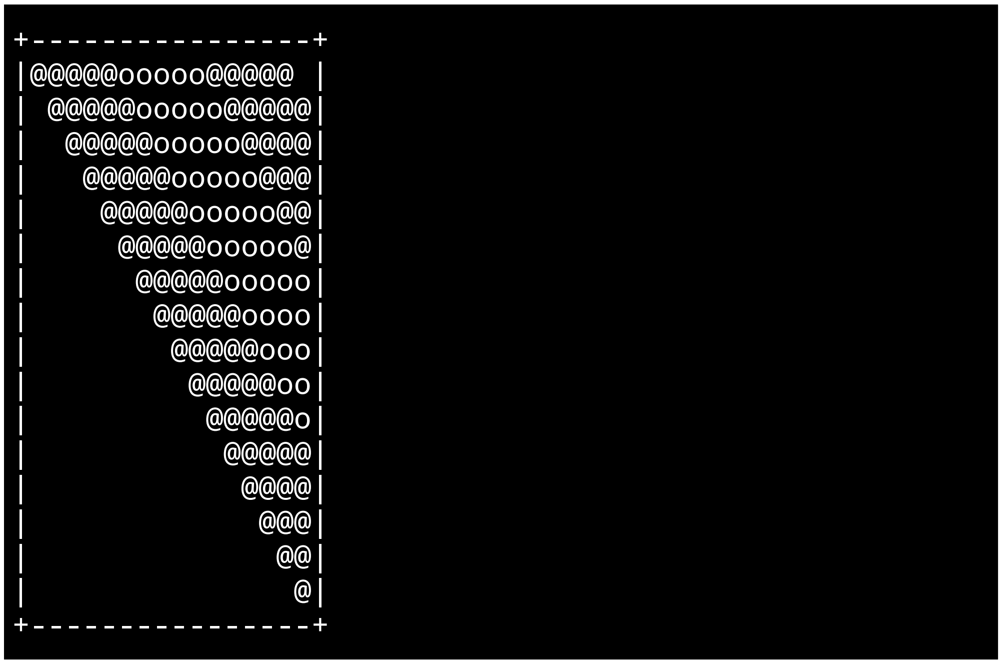
4. In addition to the rules laid out above, the output for this method should be contained within a frame of '+', '-', and '|' symbols as shown below. Note that characters from the frame do *not* count toward the *width* restriction on each line of output for this ribbon. The *width* refers only to the number of characters printed *inside* the bounds of the frame.

For example, *printRibbon(5, 11)* should produce the following output. Notice how the first line of output within the frame gets cut short by the fact that we are only allowed to print 11 characters there:



```
+-----------+
|@@@@@ooooo@|
| @@@@@ooooo|
|  @@@@@oooo|
|   @@@@@ooo|
|    @@@@@oo|
|     @@@@@o|
|      @@@@@|
|       @@@@|
|        @@@|
|         @@|
|          @|
+-----------+
```

Similarly, *printRibbon(5, 16)* should produce the following output:



```
+----------------+
|@@@@@ooooo@@@@@ |
| @@@@@ooooo@@@@@|
|  @@@@@ooooo@@@@|
|   @@@@@ooooo@@@|
|    @@@@@ooooo@@|
|     @@@@@ooooo@|
|      @@@@@ooooo|
|       @@@@@oooo|
|        @@@@@ooo|
|         @@@@@oo|
|          @@@@@o|
|           @@@@@|
|            @@@@|
|             @@@|
|              @@|
|               @|
+----------------+
```

For further examples of the expected output for this method, be sure to see the test cases included with this assignment.

Parameter Restrictions: When passing integers to this method, we guarantee 1 ≤ *n* ≤ 100,000 and 1 ≤ *width* ≤ 100,000. Note that there's no guarantee that *width* will be larger than *n*.

Output: See above for a description of this method's output. Also, see the test cases and sample output included with this assignment for further examples of this method's output. Your output must match our sample output files *exactly*.

Related Test Cases: *TestCase06.java* through *TestCase10.java*

Return Value: This is a *void* method and therefore should not return a value.

Hints: (Highlight and/or copy and paste to reveal.) *Hint #1*: [redacted]
Hint #2: [redacted]

public static double `difficultyRating()`

Description: Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult). Note that this function does not ask you to print anything to the screen or read input from the user, so you should not use *printf()* or *scanf()* here.

Related Test Case: *TestCase11.java*

public static double `hoursSpent()`

Description: Return an estimate (greater than zero) of the number of hours you spent on this assignment. Your return value must be a realistic and reasonable estimate. Unreasonably large values will result in loss of credit. Note that this function does not ask you to print anything to the screen or read input from the user, so you should not use *printf()* or *scanf()* here.

Related Test Case: *TestCase12.java*

2. Compiling and Running All Test Cases (and the *test-all.sh* Script!)

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. At the command line, whether you're working on your own system or on Eustis, you need to use the *cd* command to move to the directory where you have all the files for this assignment. For example:

```
cd Desktop/bonfire_assignment
```

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type, or the entire name needs to be wrapped in double quotes. For example:

```
cd bonfire\ assignment
```

```
cd "bonfire assignment"
```

It's probably easiest to just avoid file and folder names with spaces.

2. To compile your program with one of my test cases:

   ```
   javac Bonfire.java TestCase01.java
   ```

3. To run this test case and redirect the program's output to a text file:

   ```
   java TestCase01 > myoutput.txt
   ```

4. To compare your program's output against the sample output file I've provided for this test case:

   ```
   diff myoutput.txt sample_output/TestCase01-output.txt
   ```

   If the contents of *myoutput.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

   ```
   seansz@eustis:~$ diff myoutput.txt sample_output/TestCase01-output.txt
   seansz@eustis:~$ _
   ```

   Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

5. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *Bonfire.java* and all the test case files and typing:

   ```
   bash test-all.sh
   ```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting. Note that this script might have limited functionality on Mac OS systems or Windows systems that aren't using the Linux-style bash shell.

3. Transferring Files to Eustis

When you're ready to test your project on Eustis, using MobaXTerm to transfer your files to Eustis isn't too hard, but if you want to transfer them using a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*Bonfire.java*, *test-all.sh*, the test case files, and the *sample_output* folder).

2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

```
scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
```

Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the ***entire contents*** of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

4. Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.
- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Do not use block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed ***above*** any import statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed ***above*** the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be indented with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do ***not*** have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.

- ★ Leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)
- ★ Do not use *var* to declare variables.

5. Special Restrictions (*Super Important!*)

You must abide by the following restrictions in this assignment. Failure to abide by certain of these restrictions could result in a catastrophic loss of points.

- ★ For this particular assignment, you are not allowed to have any *import* statements, as you should not need any built-in Java goodies other than what is automatically compiled into your code from *java.lang*. We might automatically detect assignments with *import* statements and refuse to compile them for this particular assignment, resulting in zero credit.
- ★ Your *Bonfire* class cannot have any member variables (i.e., fields). Every variable you create for this assignment must be declared within a method.
- ★ File I/O is forbidden. Please do not read or write to any files.
- ★ Do not write malicious code. (I would hope this would go without saying.)
- ★ No crazy shenanigans.

6. Deliverables (Submitted via Webcourses, Not Eustis)

Submit a single source file, named *Bonfire.java*, via Webcourses. The source file should contain definitions for all the required methods (listed above), as well as any helper methods you've written to make them work.

Be sure to include your name, the course number, the current semester, and your NID in a header comment at the very top of your source file.

7. Grading Criteria and Miscellaneous Requirements

Important Note: When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

80%	Passes test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
20%	Adequate comments and whitespace and sound programming practices. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. For some methods, we might also check that you're using good functional decomposition and/or the DRY principle ("don't repeat yourself," also referred to in class as "never repeat the same code twice"). This portion of the grade might also award credit for including a header comment at the top of your source code with your name and NID.

Your program must be submitted via Webcourses.

Please be sure to submit your *.java* file, not a *.class* file (and certainly not a *.doc* or *.pdf* file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Important! **Programs that do not compile on Eustis will receive zero credit.** When testing your code, you should ensure that you place *Bonfire.java* alone in a directory with the test case files (source files, sample output files, and the input text files associated with the test cases), and no other files. That will help ensure that your *Bonfire.java* is not relying on external support classes that you've written in separate *.java* files but won't be including with your program submission.

8. Final Thoughts

Important! **You might want to remove *main()* and then double check that your program compiles without it before submitting.** Including a *main()* method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. You are strongly encouraged not to have a *main()* method in the code you submit.

Important! **Please do not create a java package.** Articulating a *package* in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! **Name your source file, class(es), and method(s) correctly.** Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to write any of the required methods, or failing to make them *public* and *static*, may cause test case failure. Please double check your work!

Input specifications are a contract. We promise that we will work within the confines of the problem statements above when creating the test cases that we'll use for grading. For example, the writeup for the *printRibbon()* method says, "we guarantee $1 \le n \le 100{,}000$ and $1 \le width \le 100{,}000$." We will keep that promise when grading your assignment: the integers we pass to *printRibbon()* are guaranteed to be in that range. We will never pass a negative integer to that method or an integer greater than 100,000.

However, please be aware that the test cases included with this assignment writeup are by no means comprehensive, and if there are no promises made about a parameter passed to a method, you should not make assumptions. Please be sure to create your own test cases and thoroughly test your code. Sharing test cases with other students is allowed (as long as those test cases don't include any solution code for this assignment), but you should challenge yourself to think of edge cases before reading other students' test cases.

Start early! Work hard! Ask questions! Good luck!