

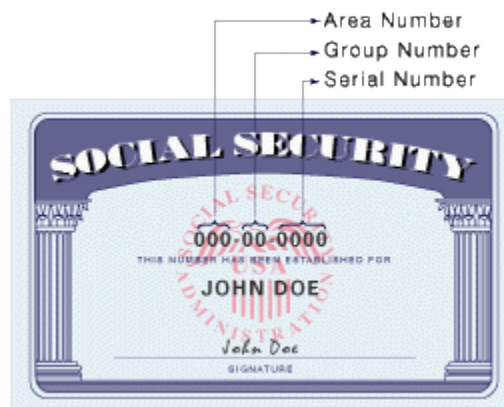
Homework 2

Social Security Number Sorting System

James Nguyen

## I. Introduction

Social Security numbers play a large role within our society as it contains many different and unique information about a specific person. Each citizen across the United States has their own specific social security number which is a combination of nine numbers. These numbers hold many important aspects as the first three digits contain the area number, the next two represents the group number, and the last four digits is the serial number for that specific citizen. The social security numbers are only disclosed to the person with the number, and the number is especially useful to the government or other organizations, like banks, as they are able to keep track information about a specific citizen like their lifetime's earning. With the millions of social security numbers, the government probably also uses a system to sort these numbers in order for them to easily search and find a specific number of a citizen.



*Image: Different aspects of the social security number according to <https://people.howstuffworks.com/social-security-number2.htm>*

This project will show a possibility of how the government or different organizations are able to keep track of these distinct social security numbers from millions of people, and different ways each organization may sort the list. This assignment's objective is to make a social security number sorting system in which users are able to choose a sorting method to sort a list of random social security numbers in order and be able to find out the amount of people that applied for SSN in the different areas from the list of the different social security numbers. The assignment will run in Java as it was originally coded in the language of Java.

## II. Design and Implementation

This Section will go through the design of the Social Security Number Sorting System, and how it will be implemented as Java Code.

### A. Fabricating Social Security Numbers

In order to create a Social Security Sorting System, we must first need different social security numbers to sort in order within the program. Since social security numbers are private and secured for many people, we must use a random number generator to create distinct social security numbers. Since this a smaller scale system, we will be generating three hundred different random social security numbers.

In order to have a very small probability that these fabricated social security numbers will not match when created, we will be generating 9 different numbers from 0 to 9, and then concatenating them together to form one social security number. Furthermore, since the area, group, and serial numbers, cannot be consecutive zeros, if the social security number falls in this case, a new random social security number will be generated again. Thus we will be able to create three hundred random distinct social security numbers to be used.

### *B. Storage for the Sorted and Unsorted Social Security Numbers*

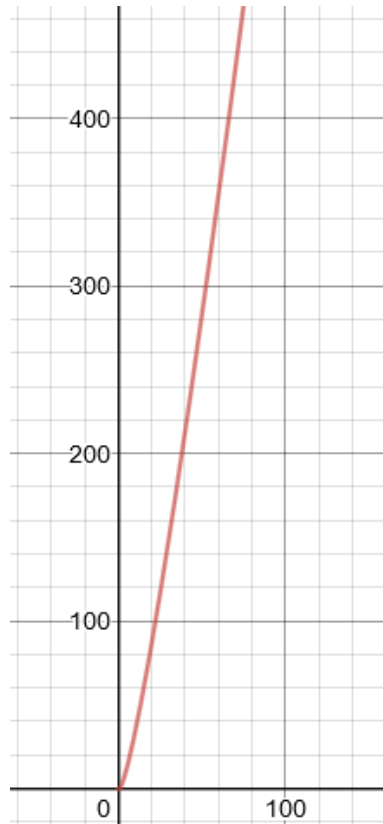
With these sorted and unsorted social security numbers, we will need to have a place to store these different numbers for users to view. In this assignment, we will be using text files to store the unsorted random social security numbers as well as sorted social security numbers. There will be four different text files that will be created on the Desktop location of the user. The first of these text files will be used to store the unsorted random social security numbers when they are created, while the other three files will be utilized to store the sorted random social security numbers. These three different text files will be each for different sorting algorithms but will have the same content in the end.

### *C. Sorting Algorithms*

In this Social Security Number System, users will be able to choose from three different sorting algorithms to sort the three hundred random social security numbers. There are three main distinct sorting algorithms; however, these sorting algorithms also use other sorting algorithms.

- 1) Quick Sort – This sorting algorithm will be one of the options that users are able to choose from. This algorithm will first essentially partition the list based on a specific pivot point in the list such that the elements that are less than or equal to the pivot point will be appearing before the pivot point, while those greater than the pivot point will be appearing after the pivot point on the list. This will allow the pivot point to be in the correct position of the list. The algorithm will further recursively call quick sort on the array of elements that are less than or equal to the pivot point, and the array of elements that are greater than the pivot point, until there is only one element in the input of quick sort. This will allow all the elements to be in the correct position such that all the elements will be from least to greatest.

In this program, every time we call on quicksort, the *pivot point* will be the **last element** in the inputted list or the index inputted for the last element. This algorithm will work fairly fast also as it has an average running time of  $O(n \lg n)$ . Since the sorting algorithm will be used on random numbers, this will be most likely the case for the majority of the time.



**Image:** Graph of  $n^2 \lg n$  provided by Desmos. This will be the majority running time for Quick Sort throughout the application

- 2) Bucket Sort – Bucket Sort will be another algorithm that users can choose to sort the random Social Security Numbers with. This algorithm can only work with lists that have numbers less than one and greater than 0. This algorithm will first sort the elements from the inputted list into different buckets. In this program, to *determine the bucket* for the element, we will **multiply the decimal value with the length of the list** and place it into that bucket that represents that result.

Once the elements are sorted into different buckets. Each bucket is then sorted through another sorting algorithm known as insertion sort. Insertion Sort will basically go through each element within the bucket and compare the element with the element in the previous index, and check if it's greater than the element, or else it would be swapped with the previous elements until its in the correct position. Once all the buckets are sorted, the ordered buckets will be then concatenated together into one list in which all the elements will be sorted.

Since these are just random Social Security Numbers, Bucket Sort will most likely have a linear running time of  $O(n)$  since the numbers will most likely be evenly distributed into buckets and sorted.

- 3) Radix Sort – The last option that users are able to choose from is to radix sort the random Social Security Numbers. This sorting algorithm works by using a stable sort to continually sort the social security number based on the LSD, and then based on the next

digit place until the algorithm finishes sorting based on the MSD. In this program, the stable sorting algorithm that we will be using is counting sort.

The Counting Sort Algorithm will use a separate counting array to count the number of times a digit appears in the specified digit place of each number in the inputted list, and to find the correct position of the number in the sorted array that will be returned.

When counting the number of times a digit appears in the specified digit place of each number in the inputted list, to find the digit at the specific position in a number, we first will divide number by the 10 for the digit position minus one time to move the digit we would like to the ones place. Then, we will modulo the number by 10 to get the digit at the once place which will be the digit we're looking for. Counting sort will count based on those digits, and using its counting array to find the position for the numbers in the sorted array based on the specified digit position

Since Radix Sort will be using Counting Sort, the algorithm will have a linear running time of  $O(d(n+k))$ , where  $d$  is the number of digits which in this case is 9 while  $k$  is 10 which is the range.

#### *D. Finding the Amount of People in a Specific Area*

Another objective of this project is to be able to give users information about the number of people in a specific area that applied for their SSN based on the Social Security Numbers in the text file. Throughout every social security number, the first three digits represent the area code.

The numbers from 001 to 199 means that the person applied in the Northeast Coast States, 200 to 399 means that the South Coast States, 400 to 599 means the Middle States, 600 to 79 means the Northwest Coast States, and 800 to 999 means the West Coast States.

In order to find the number of people in the different areas, we can iterate through the list while its being sorted and add up the number of times a person who applied from a specific area appears.

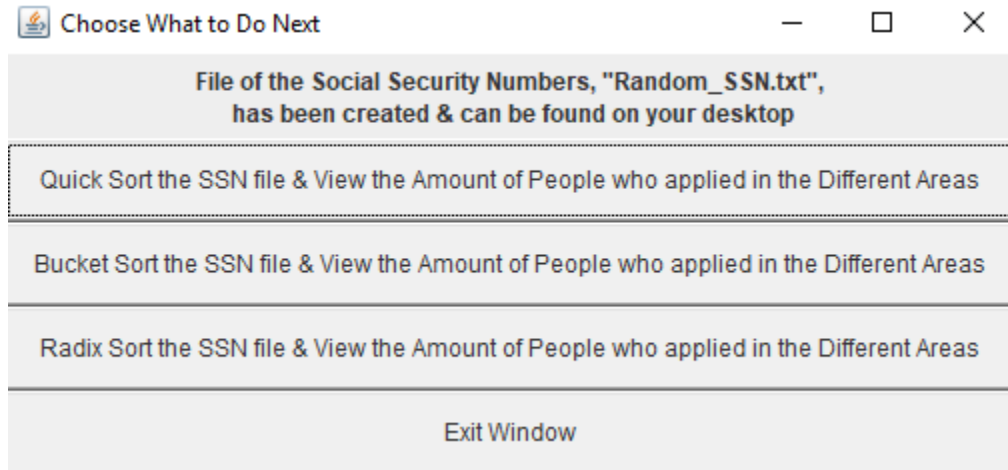
#### *E. Data Structure*

Throughout the assignment, we will also need a data structure to store the random Social Security Numbers and sort them through the different algorithms. In this project, we will be using Array Lists as the primary data structure. These Array List will hold integers for the majority of the time but will also hold doubles and other lists for algorithms such as Bucket Sort. Although Array List will be used for most of the program, in Bucket Sort, we will also be using Linked Lists as our buckets that the Array List in the algorithm will use.

#### *F. User Interface*

Users will be able to choose different options on how they would like to sort the random Social Security Numbers. As a result, a user interface will be required within this program. In order to meet this requirement, we can implement a pop-up menu with four main buttons for the four different options:

- 1) Quick Sort the SSN file & View the Amount of People who applied in the Different Areas
- 2) Bucket Sort the SSN file & “ ”
- 3) Radix Sort the SSN file & “ ”
- 4) Exit Window



**Image:** Implementation of the User Interface with the different options

User will be able to choose different sorting methods and be able to view the amount of people who applied in the distinct areas which essentially matches the requirements of the programming assignment.

### **III. List of classes /subroutines/function calls**

Throughout the program there are many different classes with many different methods that will be called and used. This section will explain these distinct classes as well as the different methods within the classes.

#### **A. Sorter Class**

This class will allow one to be able to sort a list through different sorting algorithms, specifically quick sort, bucket sort, and radix sort. These class also contain other sorting algorithms in which the main sorting algorithms will be using.

- 1) partition- This method will partition an array as it will first set the last index inputted into the method as the pivot point. The method will then compare each element within the list with the pivot point and rearrange the array such that the all elements less than or equal to the pivot point will appear before the pivot point while elements that are greater than the pivot point will appear after the pivot point. After partitioning, the pivot point will remain at that position in the array as it's in the correct position as if the list was sorted.
- 2) quickSort – This method will quickSort by continuously partitioning the list, and then recursively partition the subarrays on the left and right side of the pivot point. The sorting algorithm will first sort all of the left side since and will make its way to the right side.

By continuously calling the partition method, every element will be able to be moved into the correct position to be a sorted array from least to greatest.

```
public void quickSort(ArrayList<Integer> list, int startIndex, int endIndex) {
    if (startIndex < endIndex) { // will continually recursive call until there is only 1 element in the list
        int partitioned = partition(list, startIndex, endIndex); // partition the list
        quickSort(list, startIndex, partitioned - 1); // recursively call and partition both sides of the list,
                                                    // starting with the left side first
        quickSort(list, partitioned + 1, endIndex);
    }
}
```

**Image:** quickSort source code in Java

- 3) getDigitPosition- This method will get the digit at a specific position of a number. The method will have an inputted digit position, and the method will continuously divide the number by ten for the digit position subtracted by one time. This will move the wanted digit to the ones place. Next, we will then modulo the number by ten to get the digit at the ones place.
- 4) countingSort -This method will sort the inputted list based on the inputted digit position. The sorting algorithm will first create a counting array in which it will adding up the index to the number of times the index appears in the specified digit position of each number in the list. Next, method will add the amount in each index with the value that is in the previous index to find the indexes for numbers based on their digit position. The method will then copy the elements from the inputted list into the inputted returned list, but based on the indexes from the counting array, such that the elements will be least to greatest based on the digit position.
- 5) radixSort- This method will continually call counting sort based on the least significant digit to the most significant digit. Since counting sort is a stable sort, after calling counting sort onto the MSD, the numbers should be sorted from least to greatest in the array list.

```
public void radixSort(ArrayList<Integer> list, int digit) {
    ArrayList<Integer> returnedList = new ArrayList<Integer>(); // creates a returned list and makes it empty by
                                                                // filling it with zeroes
    for (int i = 1; i <= list.size(); i++) {
        returnedList.add(0);
    }
    for (int i = 1; i <= digit; i++) { // calls counting sort based on each digit starting from the LSD to the MSD
        countingSort(list, returnedList, i);
        for (int j = 0; j < returnedList.size(); j++) {
            list.set(j, returnedList.get(j));
        }
    }
}
```

**Image:** Radix Sort Algorithm

- 6) insertionSort – This method is another sub-sorting algorithm which will be used in bucket sort to sort the different buckets. The algorithm sorts an inputted list by iterating through the list, starting from the second element in the list, and comparing each element with its previous element, and checks if the element is greater than the previous element. If the condition is false, the method will rearrange the element to the correct position.
- 7) bucketSort – This sorting method will first create a list of linked lists with a length equal to the length of the inputted array list. The method will then copy the elements from the

inputted array into their respective bucket by multiplying their value with the length of the inputted array list and inserting it into the linked list with the index that matches the value. Insertion sort is then called to sort every linked list or bucket. The buckets are then concatenated together and returned.

```
public ArrayList<Double> bucketSort(ArrayList<Double> list) {
    int size = list.size();
    ArrayList<Double> returnedList = new ArrayList<Double>(); // the list that will be returned
    ArrayList<LinkedList<Double>> bucketList = new ArrayList<LinkedList<Double>>(); // the list that will hold the
                                                                    // different buckets

    for (int i = 1; i <= size; i++) {
        LinkedList<Double> bucket = new LinkedList<Double>();
        bucketList.add(bucket); // adding buckets into the bucket list
    }
    for (int i = 0; i < list.size(); i++) {
        bucketList.get((int) (size * (list.get(i)))).add(list.get(i)); // sorts the different elements into their
                                                                    // respective bucket as the method will add
                                                                    // the element to the bucket of its value
                                                                    // multiplied by the size
    }
    for (int i = 0; i < bucketList.size(); i++) { // insertion sort each bucket
        insertionSort(bucketList.get(i));
    }
    for (int i = 0; i < bucketList.size(); i++) { // concatenating all the buckets together into one list
        returnedList.addAll(bucketList.get(i));
    }
    return returnedList;
}
```

*Image: Bucket Sort Algorithm*

## B. Social Security System Class

This class will allow one to be able to create a Social Security Number System in which it will generate three hundred random distinct social security numbers in which users are able choose different options to sort the distinct social security numbers and view the amount of people who applied in a specific area. Furthermore, the class extends JFrame and implements ActionListener in order for the different buttons on the user interface to work. The class also uses different methods of sorter class as it creates a sorter object which will be effectively used throughout the different methods.

- 1) Constructor – This constructor will first generate three hundred different social security numbers by creating nine random digits that doesn't have any consecutive zeroes in the area, group, or social code, for three hundred times. Then it will use a print writer and file writer to create and write to a file called "Random\_SSN.txt" with each random social security number in each line in the form of 'XXX-XX-XXXX'. The Constructor will then pop up a user interface in it tells the user that the file has been created and can be found on the desktop and allows the users to choose from four different options to sort the text file.
- 2) socialToInt – This method turns any string social security number into an integer number by taking out the dashes and using the integer parseInt method.
- 3) intToSocial – This method will turn any integer social security number into a string social security by first turning the integer into a string, and then adding zeroes in front of the integer until the length is 9. Then, the method will add the required dashes to make the string match social security number.



- 4) `getGroup` – This getter method will return a number depending on the inputted social security number. The getter method will get the first three numbers of the social security and return 1 if the SSN represents the northeast coast, 2 if the SSN represents the south coast states, 3 if the SSN represents the middle states, 4 if the SSN represents northwest coast states, and 5 if the SSN represent the west coast states
- 5) `getRanSSNList`- This getter method will return an `ArrayList` of the three hundred integers in the “Random\_SSN.txt” file by reading through each line of the text using a `File Reader` and `Buffered Reader`, turning the string into an `int` using the `socialToInt` method, and adding it to the turned `Array List`.
- 6) `actionPerformed` - This method causes actions to occur if a specific button is pressed since each button on the method passes an action command which this method will detect. There are many different actions that will occur for each different button:
  - a) “Quick Sort the SSN file & View the Amount of People who applied in the Different Areas” – When this button is clicked, the command, “`quickSort`” is sent out. Due to the command sent out, `getRanSSNList` is called to get a list of all the SSN in the file, “Random\_SSN.txt”, and `quickSort` from the instance variable, `ssnSorter`, is used to sort the list. `Print Writer` and `File Writer` is then used to write and create a file called “Quick\_SSN.txt” which will be found on the user’s desktop. While iterating through the sorted list to print to the file, the program calls on `getGroup` on each element in the list to find and add up the local variables that represent each area. Once that is finished, a window will pop up showing that it was successful and will show the amount of people who applied in the different areas.
  - b) “Bucket Sort the SSN file & View the Amount of People who applied in the Different Areas” – When this button is clicked, the command, “`bucketSort`” is sent out. As a result, `getRanSSNList` is called to get a list of all the SSN in the file, “Random\_SSN.txt”. Next since `bucketSort` only take in `arraylist` within numbers greater than or equal to zero and less than 1. The program will divide every number by 1,000,000,000, in order to have every number as a decimal, and store it into an `ArrayList` for `doubles`. Then, `bucketSort` is called to sort the list. After being sorted, the elements within the list are multiplied by 1,000,000,000 and put into the list that would be printed. `Print Writer` and `File Writer` is then used to write and create a file called “Bucket\_SSN.txt” which will be found on the user’s desktop. While iterating through the sorted list to print to the file, the program calls on `getGroup` on each element in the list to find and add up the local variables that represent each area. Once that is finished, a window will pop up showing that it was successful and will show the amount of people who applied in the different areas.
  - c) Radix Sort the SSN file & View the Amount of People who applied in the Different Areas” - When this button is clicked, the command, “`radixSort`” is sent out. Due to the command sent out, `getRanSSNList` is called to get a list of all the SSN in the file, “Random\_SSN.txt”, and `radixSort` from the instance variable, `ssnSorter`, is used to sort the list. `Print Writer` and `File Writer` is then used to write and create a file called “Radix\_SSN.txt” which will be found on the user’s desktop. While iterating through the sorted list to print to the file, the program calls on `getGroup` on each element in the list to find and add up the local variables that represent each area. Once that is

finished, a window will pop up showing that it was successful and will show the amount of people who applied in the different areas.

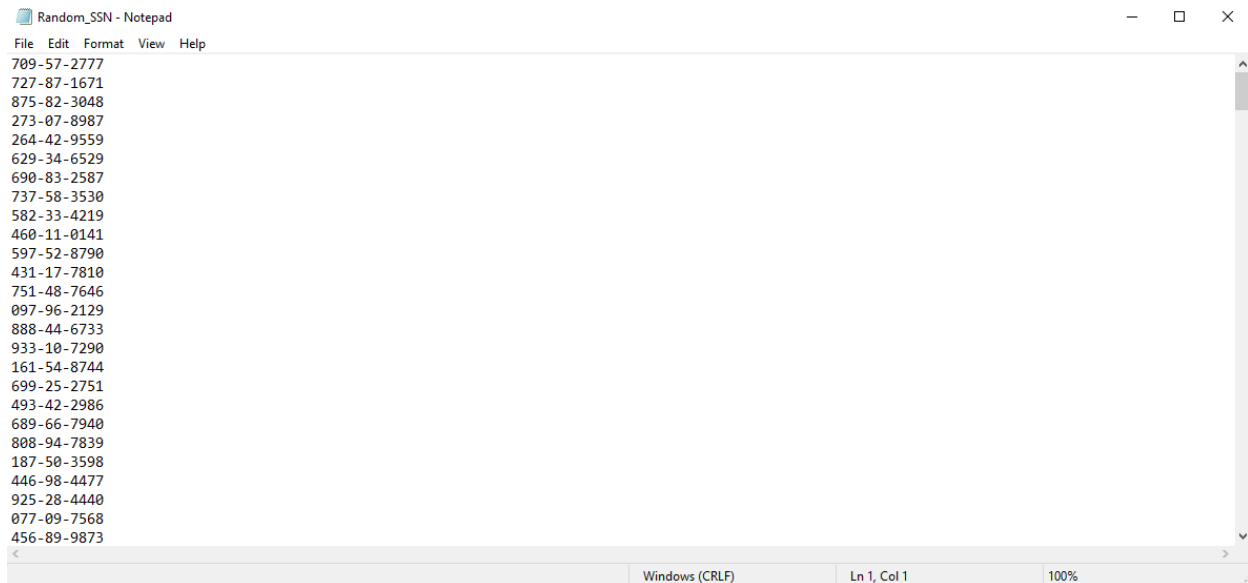
#### IV. Self-Testing Screen Shots

This section will display screenshots of testing the program to show that it has met the Functional Requirements of the programming assignment.

##### A. Using the Random Number Generator to generate 300 SSN and store them into Random\_SSN.txt

```
for (int i = 1; i <= 300; i++) {  
    String ssn = "";  
    for (int j = 1; j <= 9; j++) {  
        if (j == 4 || j == 6) {  
            ssn = ssn + "-"; // adds a dash after the third and fifth numbers to put it in  
        }  
  
        ssn = ssn + (int) (Math.random() * 10);  
        // generating a digit between 0 to 9 and adding it to the ssn  
        // string until there is 9 digits.  
    }  
    if (ssn.substring(0, 3).equals("000") || ssn.substring(4, 6).equals("00")  
        || ssn.substring(7).equals("000")) { // if the ssn create contains consecutive zeroes in the area  
        // number, group number, or serial number, the ssn will not  
        // be used  
        i--;  
    } else {  
        pw.println(ssn);  
    }  
}
```

Using a random number generator from Math.random to generate 9 different digits for the social security number, and printing it using printWriter.



```
Random_SSN - Notepad  
File Edit Format View Help  
709-57-2777  
727-87-1671  
875-82-3048  
273-07-8987  
264-42-9559  
629-34-6529  
690-83-2587  
737-58-3530  
582-33-4219  
460-11-0141  
597-52-8790  
431-17-7810  
751-48-7646  
097-96-2129  
888-44-6733  
933-10-7290  
161-54-8744  
699-25-2751  
493-42-2986  
689-66-7940  
808-94-7839  
187-50-3598  
446-98-4477  
925-28-4440  
077-09-7568  
456-89-9873  
Windows (CRLF) Ln 1, Col 1 100%
```

Random\_SSN file that was found on desktop when ran.

*B. Reading from Random\_SSN file, Quick Sorting the list and storing it into Quick\_SSN.txt, and showing on the user's laptop's screen the number of people from the same area when they applied for a SSN*

```
public ArrayList<Integer> getRanSSNList() throws IOException {
    File randomSSNFile = new File(System.getProperty("user.home") + "/Desktop", "Random_SSN.txt");
    ArrayList<Integer> ranSSNList = new ArrayList<Integer>();
    FileReader fr;
    try {
        fr = new FileReader(randomSSNFile);
        BufferedReader br = new BufferedReader(fr);
        for (int i = 1; i <= 300; i++) {
            ranSSNList.add(socialToInt(br.readLine())); // turns the string into an integer and then adds it into the
                                                         // array list
        }
    } catch (FileNotFoundException e1) {
        System.out.println("file unable to be found");
    }
    return ranSSNList;
}
```

Using a File Reader and Buffered Reader to read through the Random\_SSN.txt file



Success Window that appears after the SSN has been quick sorted and stored into Quick\_SSN.txt



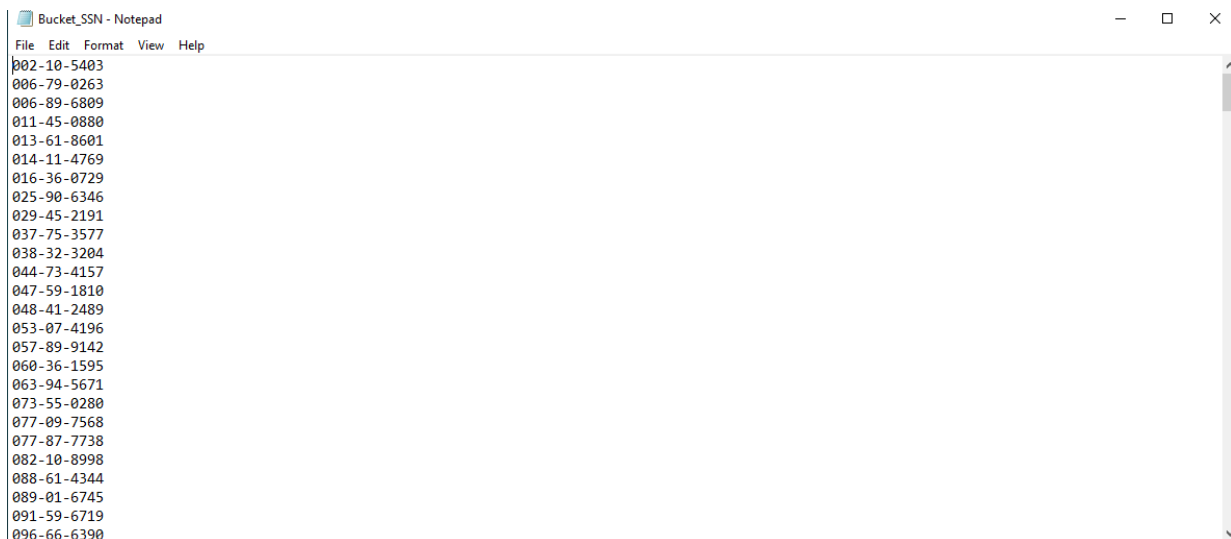
Quick\_SSN.txt file that is created with the sorted Social Security Numbers.

*C. Reading from Random\_SSN file, Bucket Sorting the list and storing it into Bucket\_SSN.txt, and showing on the user's laptop's screen the number of people from the same area when they applied for a SSN*

Will use same method as above to read the file



Success Window that appears after the SSN has bucket sorted.

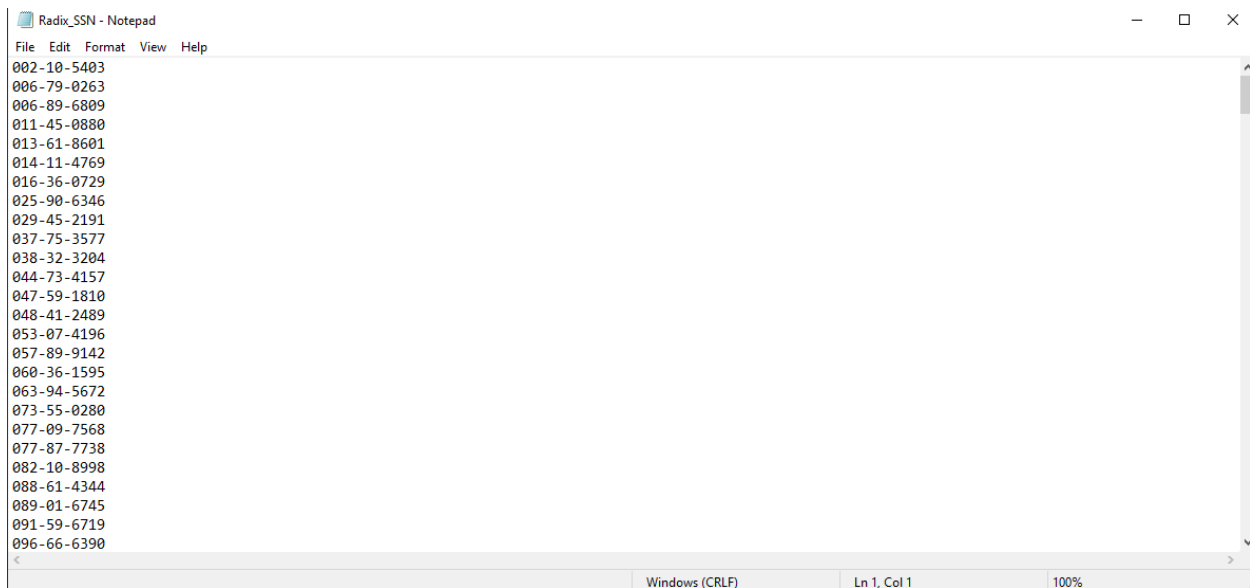


Bucket\_SSN.txt file that is created with the sorted Social Security Numbers. You can see that the first few sorted elements are the exact same as Quick\_SSN.txt.

*D. Reading from Random\_SSN file, Radix Sorting the list and storing it into Radix\_SSN.txt, and showing on the user's laptop's screen the number of people from the same area when they applied for a SSN*



Success Window that appears after the SSN has radix Sorted



Radix\_SSN.txt file that is created with the sorted Social Security Numbers. You can see that the first few sorted elements are the exact same as the other two files.

## V. How to Install the Application and Test the Codes

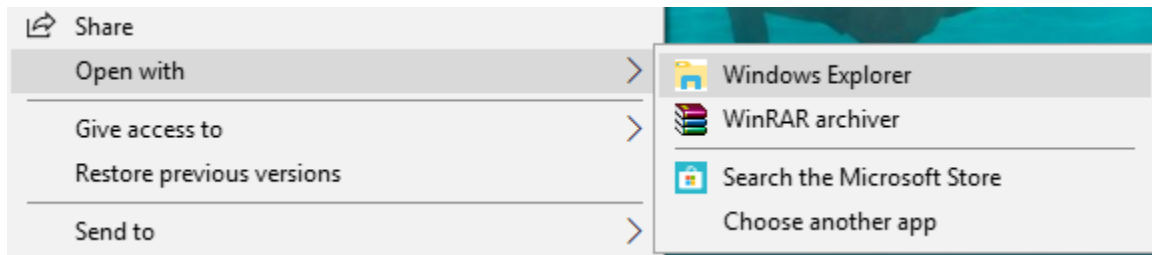
This section will demonstrate how to install the application and test the code of the programming assignment.

### A. Required Programs

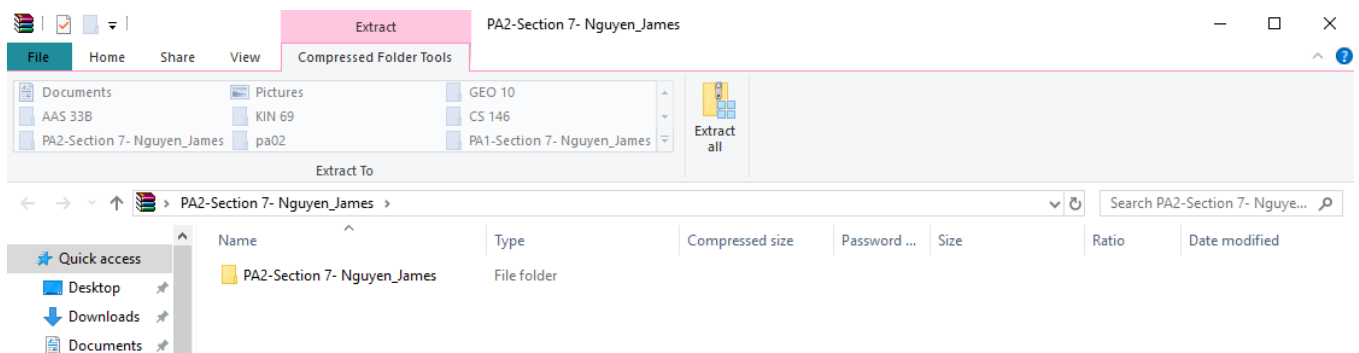
In order to test the application and the code, you will need to download Java which can be found at <https://www.java.com/en/download/>. This will allow you to run the executable jar file that comes in the zip folder. Furthermore, you will also need to have Eclipse downloaded which can be found at <https://www.eclipse.org/downloads/>.

### B. Extracting the Folder

In order to run the application you will need to extract the zipped folder first. In order to do this, you first have to left click on the folder, and then under the drop-down menu, you will select open with, and then Windows Explorer.



Once you have opened it with Windows Explorer, you can then click the extract all button on the top of the window.



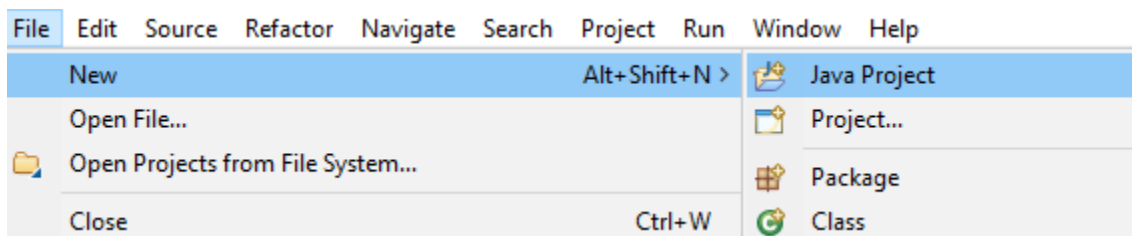
Then a pop-up window will ask you to select a destination window to extract the files to. Once it has been extracted, a folder containing the extracted files will pop up.

### C. Running the Application

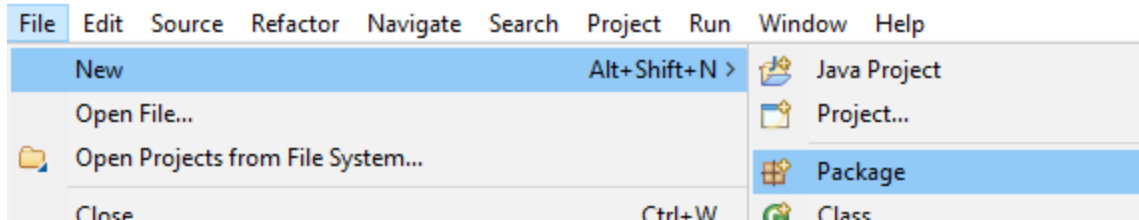
To run on the application, double click on the “Social Security System.jar” application. A text file named “Random\_SSN.txt” will then appear on your desktop, and a pop-up menu will show up in which you can pick a choice in how you would like to sort the file.

### D. Testing the Code in Eclipse

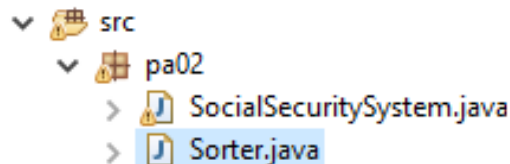
In order to test and view the code, you will have to open Eclipse first. After selecting a workspace in Eclipse, you will first press ‘File’ on the top left corner, then select new, and then Java Project.



You can name this new Java Project ‘Tester’. Next you will then go back to ‘File’, select new, and this time, you will select ‘Package’



You will then name this Package 'pa02'. Once this has finished, you can then drag and drop the \*.java files into the package.



Then, you can double click the files, and the source codes for both the java files will pop up. If you would like to run the code in eclipse, you can then press the green play button to run the main method of the Social Security System java file.

## VI. Problems Encountered during the Implementation

During the implementation of the Social Security Sorting System, there were many different problems that occurred during the coding and execution of the program, and these problems needed to be fixed or debugged.

### A. Implementation of Quick Sort, Radix Sort, and Bucket Sort

Many problems arose as I tried transferring pseudo code for quick sort, radix sort, and bucket sort into Java code. The pseudo code within the textbook for these sorting algorithms had indexes that started at 1 at times, while at other times it started at 0, which caused me to be very confused. This caused me to sometimes miscalculate the indexes as I implemented the code for the different sorting algorithms causing many different bugs such as social security numbers missing in the new texts files, or Null Pointer Exceptions.

In order to handle this bug, I had to reread the pseudo code, and fully understand what exactly is happening during each line to fix my Java code. I also took a visual stance by creating a list on a separate paper and tracing the code as if the list was running through my program to make sure these index values were correct.

### B. Integers into Doubles

Before using Bucket Sort to sort a list, one needs to make sure all the elements of the inputted list are decimal values that are less than one and greater than or equal to zero. As I divided every element by 1,000,000,000 and casted it as a double, all the result returned were zero. As a result, the Bucket Sort document returned was incorrect as it returned 300 SSN that was all zeroes.

In order to fix this, I added a .0 after 1,000,000,000 which allowed the result to be in decimal form and not integer form. This allowed numbers to be able to run through the bucket sort algorithm.

### *C. Using Array List instead of Array*

The pseudo code for the different algorithms used arrays to show the steps for the different sorting methods. Since Array Lists uses methods such as set and get instead of the '=' function, there were many times problems that occurred when I was transferring the pseudo code into Java code causing problems such as incorrect results, and many Exceptions to be thrown.

In order to fix these problems, I had to go through my code, and make sure the set and get methods were correct and had to visualize if they were doing exactly what the pseudo code wanted it to do.

### *D. Use of Java Libraries of JFrame*

This program was the second time in which I used the JFrame library from java. Although I was starting to get the hang of how to use the libraries and methods, there were many problems and bugged that arose. The first of these problems was forgetting to set the close button to end the program. As a result, when I closed the user interface, the program was still running in the background which could've caused problems like slowing my computer. Furthermore, I would forget to set the visibilities of windows, and as a result, there would be nothing showing up when I clicked on a button or ran the application

### *E. Use of File Class, Readers, and Writers*

I haven't used the File Class for a while so using the class this time brought up some conflicts. One of the problems that occurred was the File Not Found Exception that was thrown since inputted the wrong path to the file.

I also used Readers and Writers for the first time in a while, which caused me to use the incorrect methods, or forget to close the readers and writers causing the created files to not contain anything.

## **VII. Lessons Learned**

### *A. How Different Organizations Keep Track and Sort Your Social Security Numbers*

With this program assignment, I was able to learn several of the different possibilities, a organization such as the government is able to sort and keep track of one's social security number. This assignment also opened my eyes about the different aspects of the social security number such as the group, area, and serial number, and how the government may use a system like this to count the different people in each area or group.

### *B. Creations of Files, and Using Readers and Writers*

During the program assignment, I was able to learn how to create a file on a user's system on any type of platform using methods such as user.home and choose the location of the



exact file. Furthermore, I was able to fully understand how File Reader and Writers are used, and the different methods within the classes that I can use to edit my files.

### *C. Confidence in Java User Interface*

Although this was only my second time creating a user interface in Java, I was able to create the interface with much more ease, and less bugs or problems. With the last programming assignment, it took a whole day to just set up a menu, but this time, I was able to set up the menu, and complete the rest of the programming application within a day. This caused me to be much more confident in using the Java User Interface and building these interfaces through the JFrame classes.

### *D. Knowledge of Different Sorting Algorithms*

Through this program, I gained a much better understanding about the different sorting algorithms. By turning these different sorting algorithm pseudo code into Java Code, and applying them to real life applications, I was able to fully grasp more knowledge about the sorting algorithms of Quick Sort, Radix Sort, Counting Sort, Insertion Sort, and Bucket sort, and how they can be applied. I was also able to see that each sorting algorithm was quick as there wasn't that much of a delay when I ran the application.

### *E. Inspiration for Computer Science*

After completing and accomplishing this program, I began to be much more confident in computer science, as I saw how I was able to complete this project all by myself without any help. This was the second programming assignment where I had the freedom of making whatever classes and methods to create the program and doing it without any specific instructions. Even though there were many things that I may have forgotten such as the File Reader and Writer, I was still able to learn on my own and accomplish the assignment. The programming assignment further opened my eyes and showed me that Computer Science is definitely meant for me!