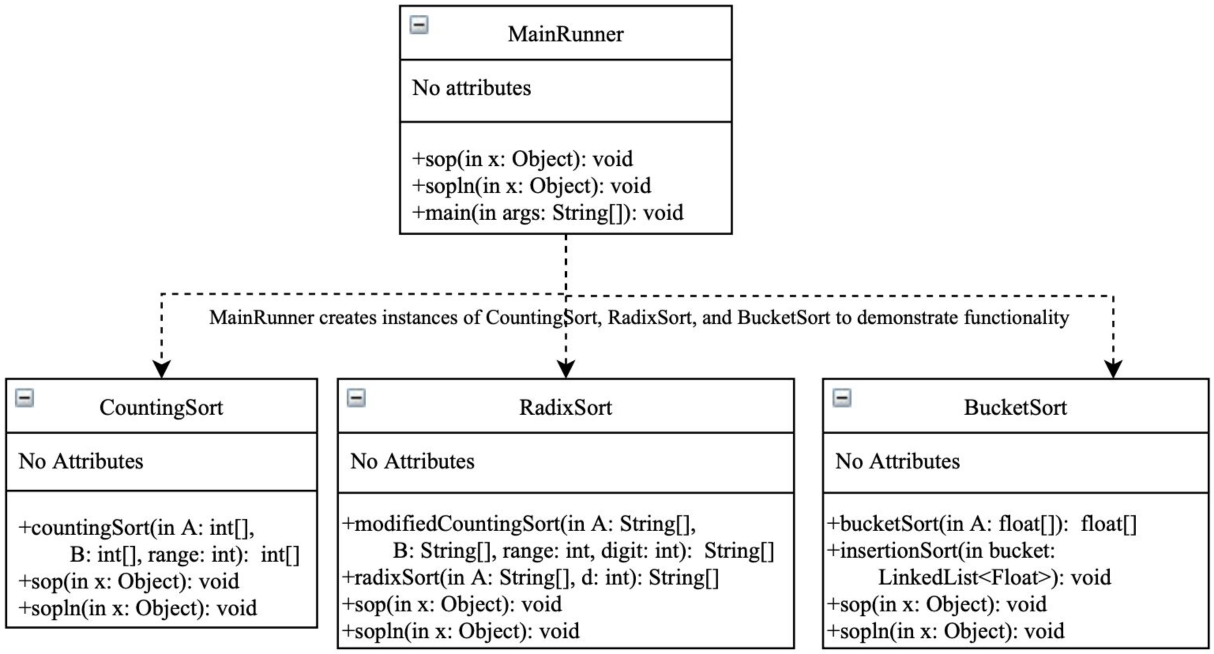
Programming Assignment #2
Sorting Algorithms in Linear Time

By: Ayush Gaggar
CS 146, Section 4

Table Of Contents

Design and Implementation

Below is a UML diagram detailing the 4 classes, as well as the variables and methods contained within each class. It's important to note that although MainRunner is a class, its only purpose is to create, initialize, and run methods to demonstrate the functionality of Counting Sort, Radix Sort, and Bucket Sort sorting algorithms.


MainRunner
No attributes
+sop(in x: Object): void
+sopln(in x: Object): void
+main(in args: String[]): void
MainRunner creates instances of CountingSort, RadixSort, and BucketSort to demonstrate functionality
CountingSort
No Attributes
+countingSort(in A: int[], B: int[], range: int): int[]
+sop(in x: Object): void
+sopln(in x: Object): void
RadixSort
No Attributes
+modifiedCountingSort(in A: String[], B: String[], range: int, digit: int): String[]
+radixSort(in A: String[], d: int): String[]
+sop(in x: Object): void
+sopln(in x: Object): void
BucketSort
No Attributes
+bucketSort(in A: float[]): float[]
+insertionSort(in bucket: LinkedList<Float>): void
+sop(in x: Object): void
+sopln(in x: Object): void

The Counting Sort algorithm follows the pseudocode exactly in sorting an array of integer values. Although an array is a more rigid data structure, versus for example, an ArrayList, for the purposes to sort a given array of 15 integer values, using an array gets the job done.

The Radix Sort algorithm also follows the pseudocode exactly in sorting a given input array in the radixSort method. The type of each array used in this class is now of type String, because this Radix Sort algorithm will be sorting hexadecimal numbers, which can include the letters a-f as well as the numbers 0-9. Again, for the functionality of this programming assignment, an array is a sufficiently robust data structure to use to sort the hex numbers.

The BucketSort algorithm works best for numbers of type float, as it requires the range to be within [0, 1) in order to sort elements into their proper subinterval "buckets". The data structure holding this series of buckets needs to be one that can aggregate other data structures; therefore, an ArrayList<LinkedList<Float>> is used, where each element is a LinkedList linking float numbers from the input array. The ArrayList should be of length n, where n is the length of the input array, and each should each be a LinkedList with subinterval equal to 1/n, which will eventually contain all elements within the bucket's subinterval. It’s important that each bucket be a LinkedList, as it is necessary to be able to use the add method to create a “has-next” relationship between elements within the LinkedList. Additionally, after sorting each interval, all LinkedLists need to be linked together to form one LinkedList. The concatenation of these LinkedLists is linked to the first LinkedList, which is then passed back into A and returned.

<u>Description of Classes and Methods</u>

Class 1: CountingSort

- No local variables or explicit constructors
- Methods: all methods are public unless otherwise specified
    - countingSort: Returns int[]
        - This method in the Counting Sort algorithm will be sorting the input array. The method takes in 2 arrays of type int as parameters, one as the input and one as the output, as well as an int variable called range. The range of the numbers will determine the size of Array C. The method first creates a hardcoded array of ints called index, which will be used to show the relative position of each element in Array A for this assignment. An array C of elements from [0...range] will be initialized to all zeros. Then, looping through the elements in A, it assigns each element to its corresponding index in C. We then loop through C, adding each element in C to the element before it to determine how many input elements are less than or equal to the value of the index of C. Finally, each element in A will be placed into its correct sorted position in the output array B using array C. The method returns array B, as a sorted and stable version of array A. There are various print statements dispersed throughout the method (using sop and sopln) to show the intermediate results for array B and C, as well as the relative index of each element as its being placed in B to show that the algorithm is stable.
    - sop: void, no return type
        - A helper method, which simply calls System.out.print() in order to print the input object
    - sopln: void, no return type
        - A helper method, which simply calls System.out.println() in order to print the input object

Class 2: RadixSort

- No local variables or explicit constructors
- Methods: all methods are public unless otherwise specified
    - modifiedCountSort: Returns String[]
        - Radix Sort requires a stable, linear sorting algorithm for each digit of a given Array of numbers. This assignment calls for the sorting of hexadecimal numbers which may contain letters as well as numbers. Thus, this class uses a modified Counting Sort algorithm, different from the one in Exercise #1 in 2 ways: 1) input and output arrays are of

        - type String, and 2) there is an additional parameter of type int called "digit", which tells the modified Counting Sort Algorithm which digit of the number it will be sorting. radixSort method calls this method d times, where d is the number of digits.
    - radixSort: Returns String[]
        - This method is the central method in this class, and is relatively simple. Its input parameters are an Array A of type String, which contains the randomly generated hex numbers, and an int d which tells the method how many digits long each hex number in A is. It creates a new empty String[] B for every digit in the hexadecimal number, because the modifiedCountingSort algorithm requires an empty Array as one of its parameters. It then calls the modifiedCountingSort method above for a given digit, starting from the Least Significant Digit, until the most significant digit is sorted. It then returns A, which is now sorted.
    - sop: void, no return type
        - A helper method, which simply calls System.out.print() in order to print the input object
    - sopln: void, no return type
        - A helper method, which simply calls System.out.println() in order to print the input object

Class 3: BucketSort

- No local variables or explicit constructors
- Methods: all methods are public unless otherwise specified
    - bucketSort: Returns float[]
        - This method is the critical method which will sort the input Array using Bucket Sort. Bucket Sort works best if the input is of type float, as one of the first steps is to convert the input elements to a range between [0,1) by dividing by n, the length of Array A. There is also an array called index which is hardcoded, and contains the relative index of each element for this particular exercise. Next, it creates an ArrayList called B of size 2*n, where each element in B is a LinkedList<Float>. The elements in B from [0...n) will be used to store each element in A to its correct subinterval location in the "bucket" of B. The elements in B from [n...2n) will be used to store the relative index of each element in A. The relative indices do not have a "subinterval location" in B, but rather get moved proportionally with their respective element in A, to demonstrate that the elements in A maintain their relative position. After being added to buckets, each

LinkedList in B calls insertionSort method below to sort the elements within it. This step is unnecessary in our case, as each bucket contains the same number. Then, each LinkedList element in B is concatenated to the LinkedList stored at index 0. Using the LinkedList at 0, the now sorted position is added back to the index A and index respectively, in a sorted and stable fashion.

    - insertionSort: void, no return type
        - A simple insertionSort method, modified such that its input parameter is of type LinkedList<Float>. It compares numbers in the LinkedList with the numbers before it, moving them to their correct location.
    - sop: void, no return type
        - A helper method, which simply calls System.out.print() in order to print the input object
    - sopln: void, no return type
        - A helper method, which simply calls System.out.println() in order to print the input object

Class 4: MainRunner

- The MainRunner class is used as a runner class to test and run the necessary methods from all three above classes, and demonstrates for this assignment the linear sorting algorithms.
- No local variables or explicit constructors
- Methods: all methods are public unless otherwise specified
    - sop: void, no return type
        - A helper method, which simply calls System.out.print() in order to print the input object
    - sopln: void, no return type
        - A helper method, which simply calls System.out.println() in order to print the input object
    - main: void, no return type
        - The main method is where all objects are created and where these objects call the necessary methods.
        - First, main creates the array specified by Exercise #1, as well as an empty array B, the same size as A. It then creates an instance of the CountingSort class, and calls the countingSort method. Through this call, necessary intermediate steps are printed to the terminal.
        - It then creates an object of class Random, which is used to randomly generate each digit of a 5-digit hexadecimal number, which is then added to an array of type String called toSort. It generates 30 random

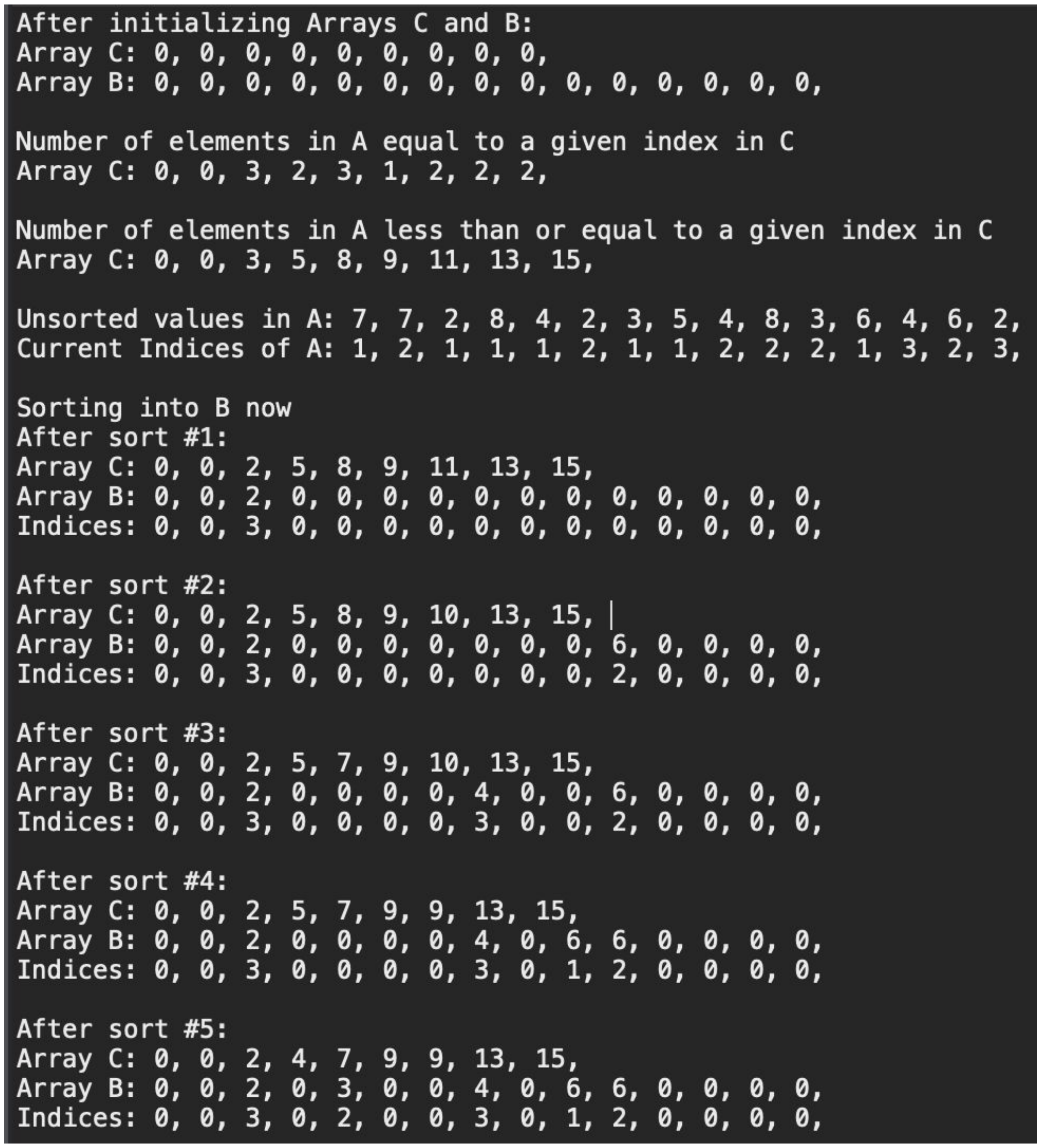
hex numbers by looping and adding each randomly generated digit to the previous digit to make a 5-digit hex number, a process which is repeated 30 times for each hex number. It then creates an instance of the RadixSort class, and calls the radixSort method. Through this call, necessary intermediate steps are printed to the terminal.

- Finally, main creates a new array with elements as specified by exercise 1, this time of type float. It then creates an instance of the BucketSort class, and calls the bucketSort method. Through this call, necessary intermediate steps are printed to the terminal.
- After this call, the main method terminates.

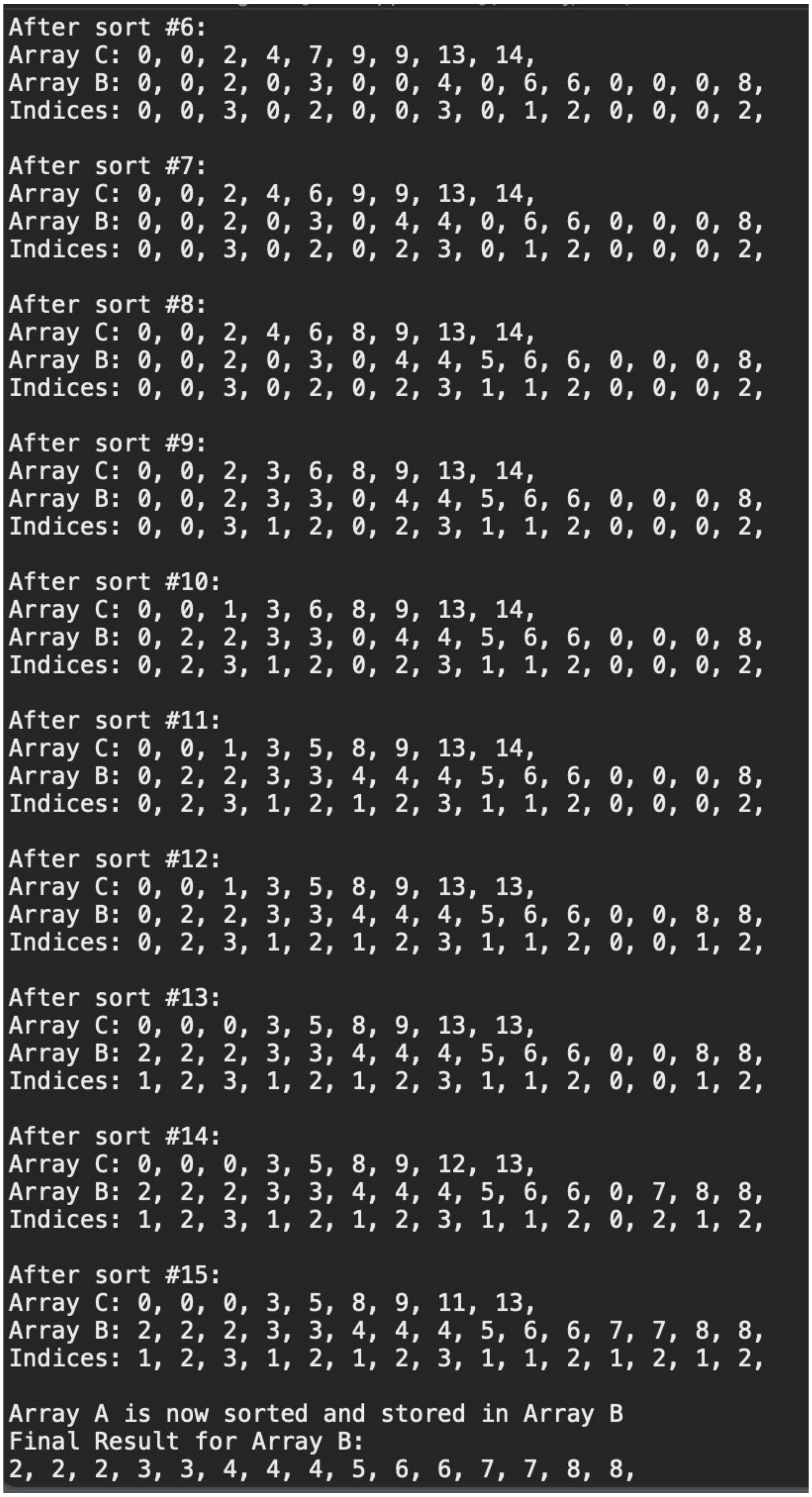
Demonstration of Meeting Functional Requirements
Functional Requirements as stated in Programming Assignment #2:

Exercise #1: Demonstrate Counting Sort Algorithm functionality, and prove that it is a stable algorithm.

Counting Sort is a stable sorting algorithm which sorts an input array of integers in a given range from low to high. The code implemented in Exercise #1 of this Programming Assignment demonstrates the intermediate steps done by the Counting Sort algorithm when sorting the input array A into the output array B using the auxiliary array C. Because the input array A and its indices do not contain a value of 0, the arrays of B and C are initialized by default to have values of 0.



```
After initializing Arrays C and B:
Array C: 0, 0, 0, 0, 0, 0, 0, 0, 0,
Array B: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

Number of elements in A equal to a given index in C
Array C: 0, 0, 3, 2, 3, 1, 2, 2, 2,

Number of elements in A less than or equal to a given index in C
Array C: 0, 0, 3, 5, 8, 9, 11, 13, 15,

Unsorted values in A: 7, 7, 2, 8, 4, 2, 3, 5, 4, 8, 3, 6, 4, 6, 2,
Current Indices of A: 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 2, 1, 3, 2, 3,

Sorting into B now
After sort #1:
Array C: 0, 0, 2, 5, 8, 9, 11, 13, 15,
Array B: 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Indices: 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

After sort #2:
Array C: 0, 0, 2, 5, 8, 9, 10, 13, 15,
Array B: 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0, 0,
Indices: 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0,

After sort #3:
Array C: 0, 0, 2, 5, 7, 9, 10, 13, 15,
Array B: 0, 0, 2, 0, 0, 0, 0, 4, 0, 0, 6, 0, 0, 0, 0,
Indices: 0, 0, 3, 0, 0, 0, 0, 3, 0, 0, 2, 0, 0, 0, 0,

After sort #4:
Array C: 0, 0, 2, 5, 7, 9, 9, 13, 15,
Array B: 0, 0, 2, 0, 0, 0, 0, 4, 0, 6, 6, 0, 0, 0, 0,
Indices: 0, 0, 3, 0, 0, 0, 0, 3, 0, 1, 2, 0, 0, 0, 0,

After sort #5:
Array C: 0, 0, 2, 4, 7, 9, 9, 13, 15,
Array B: 0, 0, 2, 0, 3, 0, 0, 4, 0, 6, 6, 0, 0, 0, 0,
Indices: 0, 0, 3, 0, 2, 0, 0, 3, 0, 1, 2, 0, 0, 0, 0,
```

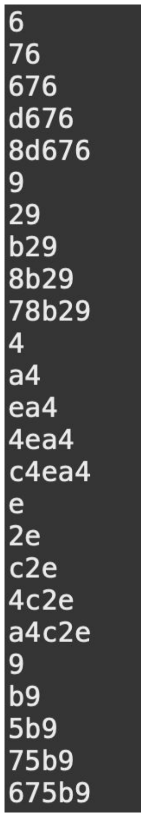


```
After sort #6:
Array C: 0, 0, 2, 4, 7, 9, 9, 13, 14,
Array B: 0, 0, 2, 0, 3, 0, 0, 4, 0, 6, 6, 0, 0, 0, 8,
Indices: 0, 0, 3, 0, 2, 0, 0, 3, 0, 1, 2, 0, 0, 0, 2,

After sort #7:
Array C: 0, 0, 2, 4, 6, 9, 9, 13, 14,
Array B: 0, 0, 2, 0, 3, 0, 4, 4, 0, 6, 6, 0, 0, 0, 8,
Indices: 0, 0, 3, 0, 2, 0, 2, 3, 0, 1, 2, 0, 0, 0, 2,

After sort #8:
Array C: 0, 0, 2, 4, 6, 8, 9, 13, 14,
Array B: 0, 0, 2, 0, 3, 0, 4, 4, 5, 6, 6, 0, 0, 0, 8,
Indices: 0, 0, 3, 0, 2, 0, 2, 3, 1, 1, 2, 0, 0, 0, 2,

After sort #9:
Array C: 0, 0, 2, 3, 6, 8, 9, 13, 14,
Array B: 0, 0, 2, 3, 3, 0, 4, 4, 5, 6, 6, 0, 0, 0, 8,
Indices: 0, 0, 3, 1, 2, 0, 2, 3, 1, 1, 2, 0, 0, 0, 2,

After sort #10:
Array C: 0, 0, 1, 3, 6, 8, 9, 13, 14,
Array B: 0, 2, 2, 3, 3, 0, 4, 4, 5, 6, 6, 0, 0, 0, 8,
Indices: 0, 2, 3, 1, 2, 0, 2, 3, 1, 1, 2, 0, 0, 0, 2,

After sort #11:
Array C: 0, 0, 1, 3, 5, 8, 9, 13, 14,
Array B: 0, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 0, 0, 0, 8,
Indices: 0, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 0, 0, 0, 2,

After sort #12:
Array C: 0, 0, 1, 3, 5, 8, 9, 13, 13,
Array B: 0, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 0, 0, 8, 8,
Indices: 0, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 0, 0, 1, 2,

After sort #13:
Array C: 0, 0, 0, 3, 5, 8, 9, 13, 13,
Array B: 2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 0, 0, 8, 8,
Indices: 1, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 0, 0, 1, 2,

After sort #14:
Array C: 0, 0, 0, 3, 5, 8, 9, 12, 13,
Array B: 2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 0, 7, 8, 8,
Indices: 1, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 0, 2, 1, 2,

After sort #15:
Array C: 0, 0, 0, 3, 5, 8, 9, 11, 13,
Array B: 2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 7, 7, 8, 8,
Indices: 1, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 1, 2, 1, 2,

Array A is now sorted and stored in Array B
Final Result for Array B:
2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 7, 7, 8, 8,
```

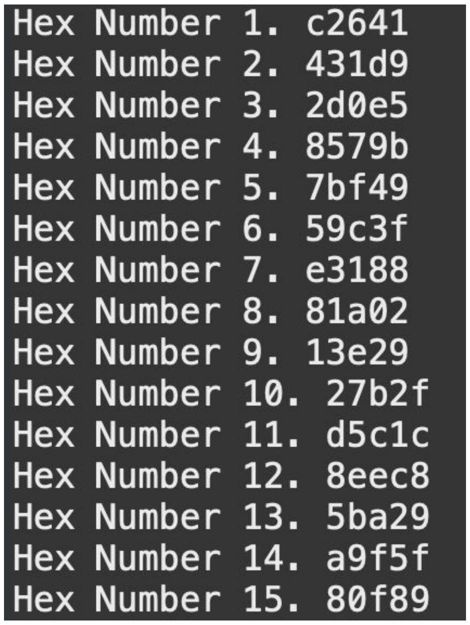
As the above output screenshot shows, the relative order of the indices in A are preserved as the algorithm runs, thus proving that Counting Sort runs in linear time and is a stable sorting algorithm.

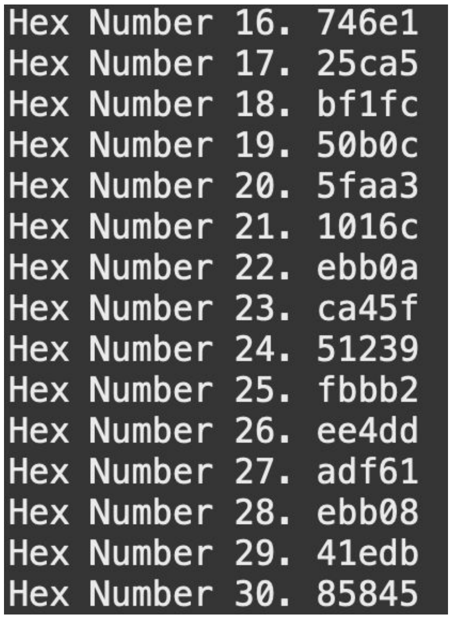
<u>Exercise #2:</u> Demonstrate and explain Radix Sort Algorithm functionality on 30 randomly generated hexadecimal numbers.

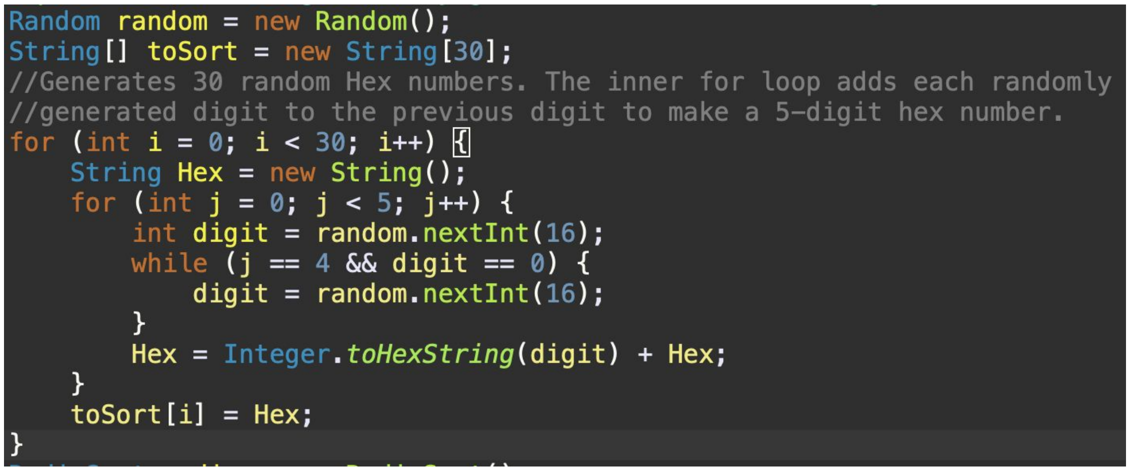
The first step is to first randomly generate 30 different hexadecimal numbers, each with 5 digits. In order to accomplish this, I first generated a random hex digit, and then appended this to the beginning of the previously generated digits. There is a while loop in the code which checks to make sure that the most significant digit is not a 0; this is to ensure that each number will be 5 digits long. An example of 5 randomly generated hex numbers is shown to the right, with print statements showing each digit being added to the beginning of the hex number string.


```
6
76
676
d676
8d676
9
29
b29
8b29
78b29
4
a4
ea4
4ea4
c4ea4
e
2e
c2e
4c2e
a4c2e
9
b9
5b9
75b9
675b9
```

Next, the method created 30 randomly generated hex digits, and added it to the input array which would then be sorted using Radix Sort. These 30 numbers are shown below, as well as the code which generated them:

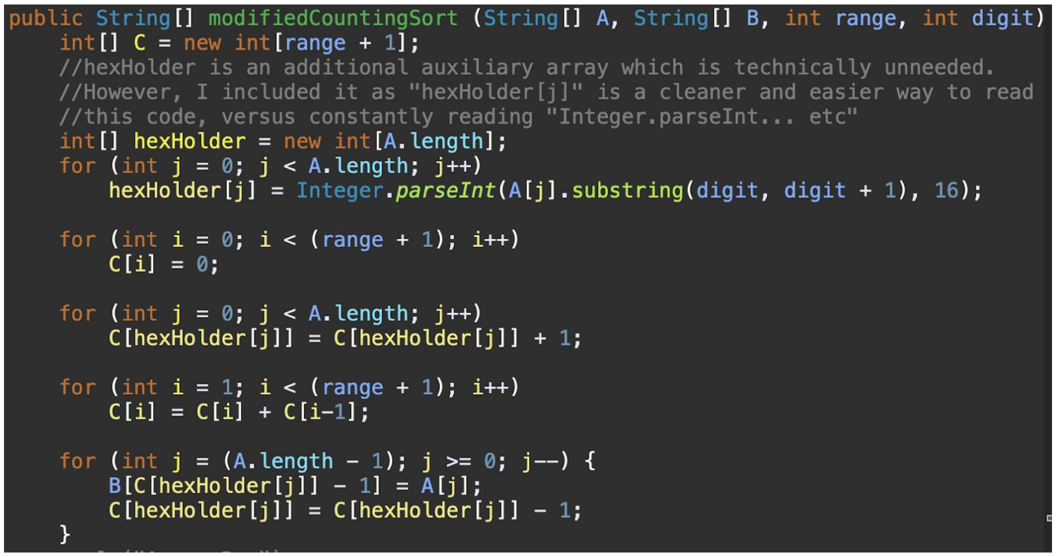

```
Hex Number 1. c2641
Hex Number 2. 431d9
Hex Number 3. 2d0e5
Hex Number 4. 8579b
Hex Number 5. 7bf49
Hex Number 6. 59c3f
Hex Number 7. e3188
Hex Number 8. 81a02
Hex Number 9. 13e29
Hex Number 10. 27b2f
Hex Number 11. d5c1c
Hex Number 12. 8eec8
Hex Number 13. 5ba29
Hex Number 14. a9f5f
Hex Number 15. 80f89
```


```
Hex Number 16. 746e1
Hex Number 17. 25ca5
Hex Number 18. bf1fc
Hex Number 19. 50b0c
Hex Number 20. 5faa3
Hex Number 21. 1016c
Hex Number 22. ebb0a
Hex Number 23. ca45f
Hex Number 24. 51239
Hex Number 25. fbbb2
Hex Number 26. ee4dd
Hex Number 27. adf61
Hex Number 28. ebb08
Hex Number 29. 41edb
Hex Number 30. 85845
```



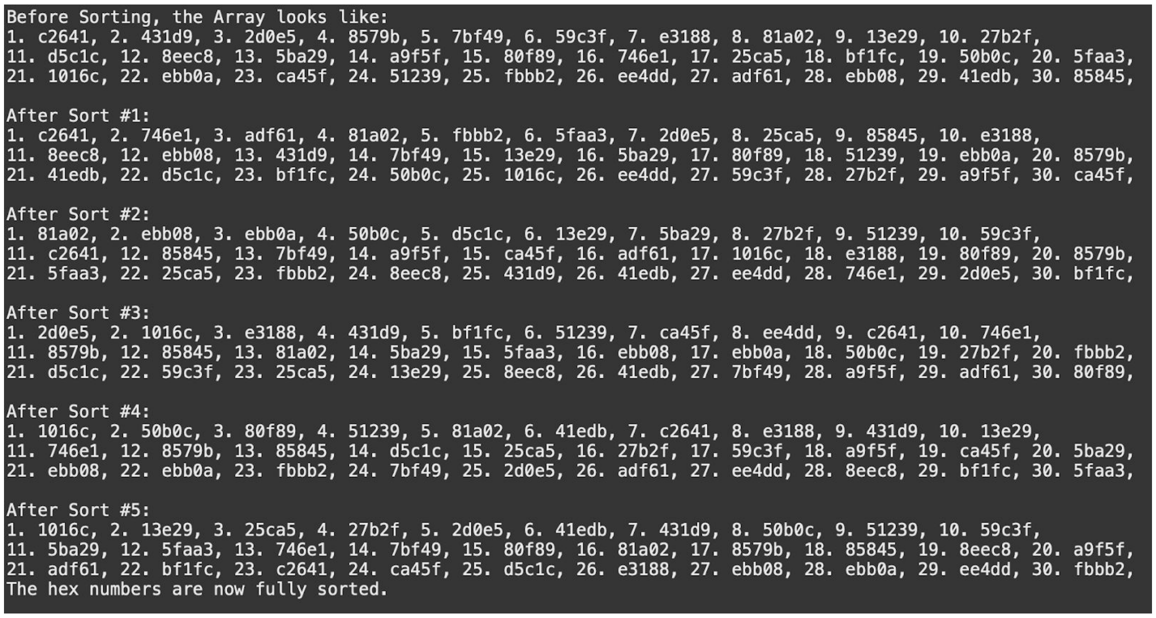
```java
Random random = new Random();
String[] toSort = new String[30];
//Generates 30 random Hex numbers. The inner for loop adds each randomly
//generated digit to the previous digit to make a 5-digit hex number.
for (int i = 0; i < 30; i++) {
    String Hex = new String();
    for (int j = 0; j < 5; j++) {
        int digit = random.nextInt(16);
        while (j == 4 && digit == 0) {
            digit = random.nextInt(16);
        }
        Hex = Integer.toHexString(digit) + Hex;
    }
    toSort[i] = Hex;
}
```

After these numbers were generated, the class calls the radixSort method on the toSort array of Strings. The radixSort method then called the modifiedCountingSort method for each of the 5 digits in each number in the toSort array. The radixSort method is shown in the Problems Encountered section of this report, which goes into further detail about which parts of programming this algorithm were challenging. Screenshots of the modifiedCountingSort method are shown below:



```java
public String[] modifiedCountingSort (String[] A, String[] B, int range, int digit)
    int[] C = new int[range + 1];
    //hexHolder is an additional auxiliary array which is technically unneeded.
    //However, I included it as "hexHolder[j]" is a cleaner and easier way to read
    //this code, versus constantly reading "Integer.parseInt... etc"
    int[] hexHolder = new int[A.length];
    for (int j = 0; j < A.length; j++)
        hexHolder[j] = Integer.parseInt(A[j].substring(digit, digit + 1), 16);

    for (int i = 0; i < (range + 1); i++)
        C[i] = 0;

    for (int j = 0; j < A.length; j++)
        C[hexHolder[j]] = C[hexHolder[j]] + 1;

    for (int i = 1; i < (range + 1); i++)
        C[i] = C[i] + C[i-1];

    for (int j = (A.length - 1); j >= 0; j--) {
        B[C[hexHolder[j]] - 1] = A[j];
        C[hexHolder[j]] = C[hexHolder[j]] - 1;
    }
```

The results after each call to modifiedCountingSort are shown below. The RadixSort algorithm is shown in the Problems Encountered Section as well.



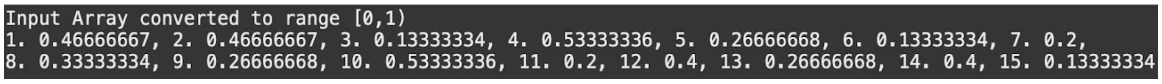
```
Before Sorting, the Array looks like:
1. c2641, 2. 431d9, 3. 2d0e5, 4. 8579b, 5. 7bf49, 6. 59c3f, 7. e3188, 8. 81a02, 9. 13e29, 10. 27b2f,
11. d5c1c, 12. 8eec8, 13. 5ba29, 14. a9f5f, 15. 80f89, 16. 746e1, 17. 25ca5, 18. bf1fc, 19. 50b0c, 20. 5faa3,
21. 1016c, 22. ebb0a, 23. ca45f, 24. 51239, 25. fbbb2, 26. ee4dd, 27. adf61, 28. ebb08, 29. 41edb, 30. 85845,

After Sort #1:
1. c2641, 2. 746e1, 3. adf61, 4. 81a02, 5. fbbb2, 6. 5faa3, 7. 2d0e5, 8. 25ca5, 9. 85845, 10. e3188,
11. 8eec8, 12. ebb08, 13. 431d9, 14. 7bf49, 15. 13e29, 16. 5ba29, 17. 80f89, 18. 51239, 19. ebb0a, 20. 8579b,
21. 41edb, 22. d5c1c, 23. bf1fc, 24. 50b0c, 25. 1016c, 26. ee4dd, 27. 59c3f, 28. 27b2f, 29. a9f5f, 30. ca45f,

After Sort #2:
1. 81a02, 2. ebb08, 3. ebb0a, 4. 50b0c, 5. d5c1c, 6. 13e29, 7. 5ba29, 8. 27b2f, 9. 51239, 10. 59c3f,
11. c2641, 12. 85845, 13. 7bf49, 14. a9f5f, 15. ca45f, 16. adf61, 17. 1016c, 18. e3188, 19. 80f89, 20. 8579b,
21. 5faa3, 22. 25ca5, 23. fbbb2, 24. 8eec8, 25. 431d9, 26. 41edb, 27. ee4dd, 28. 746e1, 29. 2d0e5, 30. bf1fc,

After Sort #3:
1. 2d0e5, 2. 1016c, 3. e3188, 4. 431d9, 5. bf1fc, 6. 51239, 7. ca45f, 8. ee4dd, 9. c2641, 10. 746e1,
11. 8579b, 12. 85845, 13. 81a02, 14. 5ba29, 15. 5faa3, 16. ebb08, 17. ebb0a, 18. 50b0c, 19. 27b2f, 20. fbbb2,
21. d5c1c, 22. 59c3f, 23. 25ca5, 24. 13e29, 25. 8eec8, 26. 41edb, 27. 7bf49, 28. a9f5f, 29. adf61, 30. 80f89,

After Sort #4:
1. 1016c, 2. 50b0c, 3. 80f89, 4. 51239, 5. 81a02, 6. 41edb, 7. c2641, 8. e3188, 9. 431d9, 10. 13e29,
11. 746e1, 12. 8579b, 13. 85845, 14. d5c1c, 15. 25ca5, 16. 27b2f, 17. 59c3f, 18. a9f5f, 19. ca45f, 20. 5ba29,
21. ebb08, 22. ebb0a, 23. fbbb2, 24. 7bf49, 25. 2d0e5, 26. adf61, 27. ee4dd, 28. 8eec8, 29. bf1fc, 30. 5faa3,

After Sort #5:
1. 1016c, 2. 13e29, 3. 25ca5, 4. 27b2f, 5. 2d0e5, 6. 41edb, 7. 431d9, 8. 50b0c, 9. 51239, 10. 59c3f,
11. 5ba29, 12. 5faa3, 13. 746e1, 14. 7bf49, 15. 80f89, 16. 81a02, 17. 8579b, 18. 85845, 19. 8eec8, 20. a9f5f,
21. adf61, 22. bf1fc, 23. c2641, 24. ca45f, 25. d5c1c, 26. e3188, 27. ebb08, 28. ebb0a, 29. ee4dd, 30. fbbb2,
The hex numbers are now fully sorted.
```

Upon a closer inspection to the above screenshot, it is evident that each pass through modifiedCountingSort targets and sorts a different digit. After the first sort, the least significant digit in each of the 30 hex numbers is sorted from low to high. The number 0 is the lowest value and the letter f (which represents 15) is the highest value. After the second sort, digit #4 is sorted from low to high. This process continues until after the fifth sort, when the most significant digit has been sorted. After this sort, the input array of 30 hexadecimal numbers is clearly sorted from lowest value to highest value.

<u>Exercise #3:</u> Demonstrate Bucket Sort Algorithm functionality for the same array as Exercise #1, and prove that it is a stable algorithm.

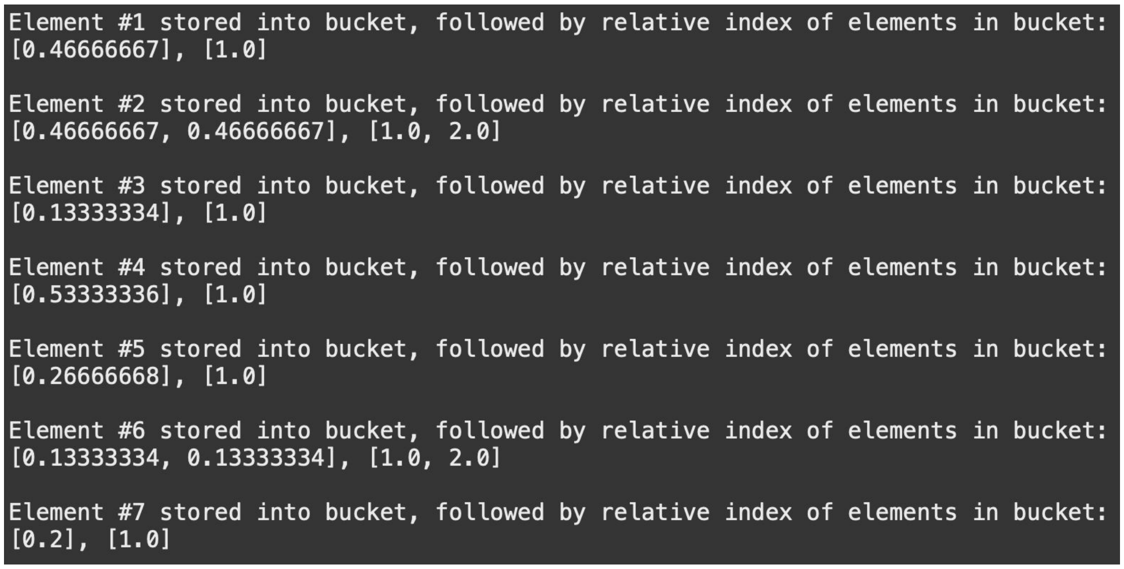
Bucket Sort is a stable sorting algorithm which sorts an input array of integers in the range [0,1) from low to high. The code implemented in Exercise #3 of this Programming Assignment demonstrates the intermediate steps done by the Bucket Sort algorithm when sorting the input array A using an auxiliary ArrayList composed of LinkedList<Float> to store elements in a given subinterval. Because the ArrayList aggregates LinkedList<Float>, the indices are also represented as type Float.

The first step the Bucket Sort algorithm does is convert the input array into a valid range, from [0,1), by dividing each element by the length of the input Array.

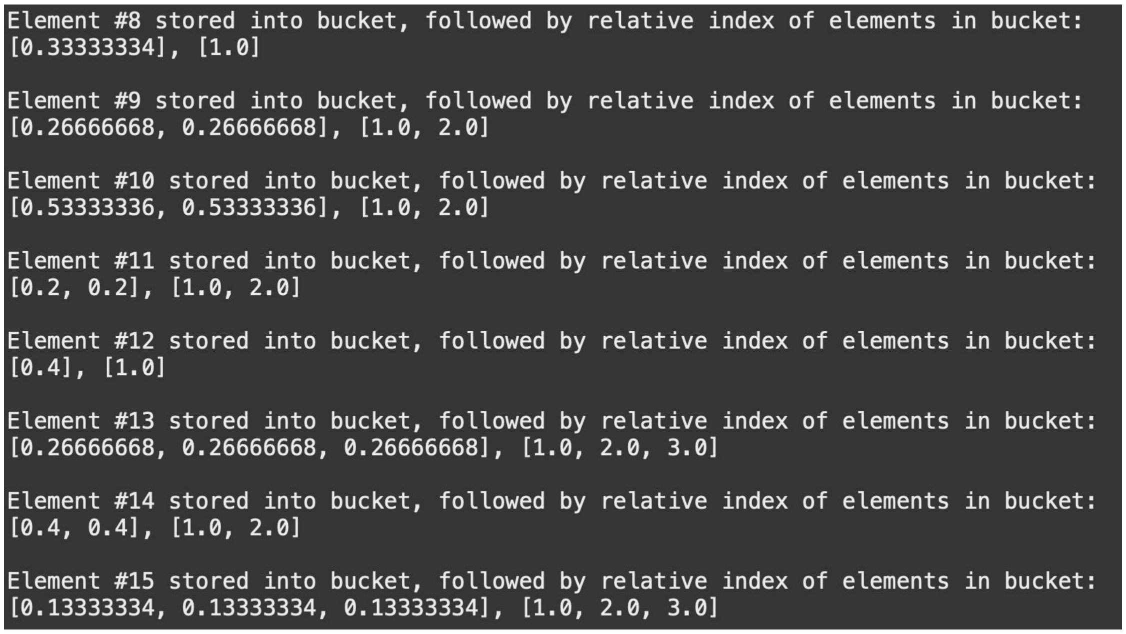


```
Input Array converted to range [0,1)
1. 0.46666667, 2. 0.46666667, 3. 0.13333334, 4. 0.53333336, 5. 0.26666668, 6. 0.13333334, 7. 0.2,
8. 0.33333334, 9. 0.26666668, 10. 0.53333336, 11. 0.2, 12. 0.4, 13. 0.26666668, 14. 0.4, 15. 0.13333334
```

Next, the algorithm adds the values in the algorithm one by one to their respective buckets.



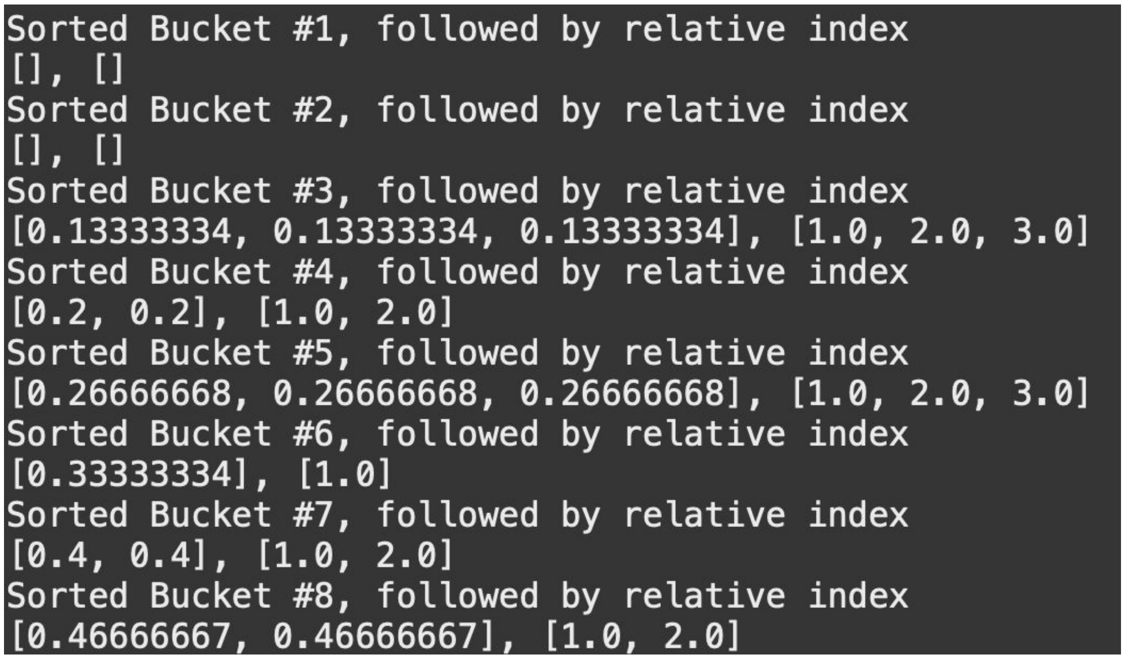
```
Element #1 stored into bucket, followed by relative index of elements in bucket:
[0.46666667], [1.0]

Element #2 stored into bucket, followed by relative index of elements in bucket:
[0.46666667, 0.46666667], [1.0, 2.0]

Element #3 stored into bucket, followed by relative index of elements in bucket:
[0.13333334], [1.0]

Element #4 stored into bucket, followed by relative index of elements in bucket:
[0.53333336], [1.0]

Element #5 stored into bucket, followed by relative index of elements in bucket:
[0.26666668], [1.0]

Element #6 stored into bucket, followed by relative index of elements in bucket:
[0.13333334, 0.13333334], [1.0, 2.0]

Element #7 stored into bucket, followed by relative index of elements in bucket:
[0.2], [1.0]
```



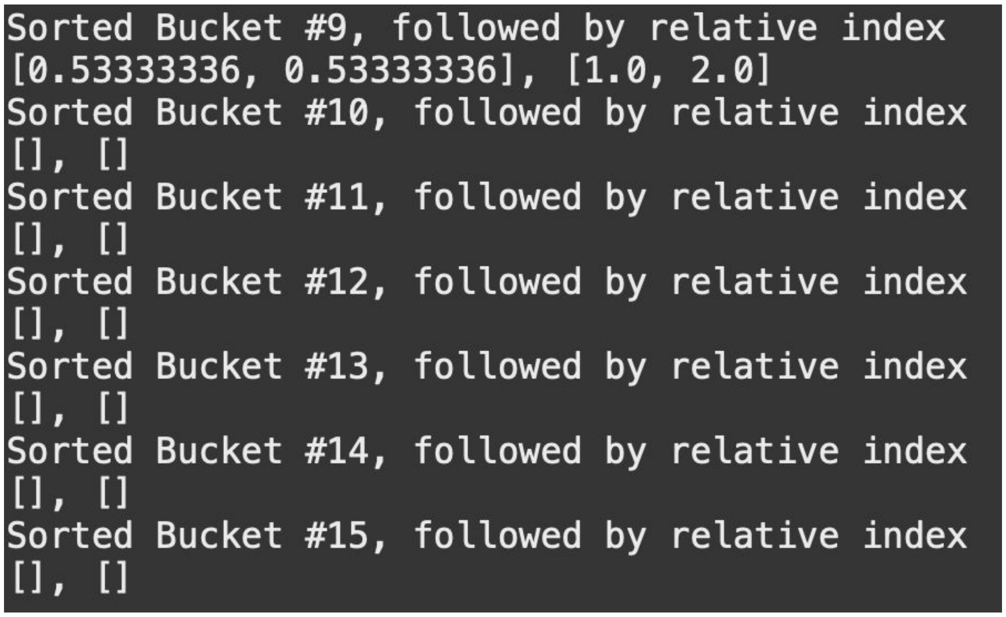
```
Element #8 stored into bucket, followed by relative index of elements in bucket:
[0.33333334], [1.0]

Element #9 stored into bucket, followed by relative index of elements in bucket:
[0.26666668, 0.26666668], [1.0, 2.0]

Element #10 stored into bucket, followed by relative index of elements in bucket:
[0.53333336, 0.53333336], [1.0, 2.0]

Element #11 stored into bucket, followed by relative index of elements in bucket:
[0.2, 0.2], [1.0, 2.0]

Element #12 stored into bucket, followed by relative index of elements in bucket:
[0.4], [1.0]

Element #13 stored into bucket, followed by relative index of elements in bucket:
[0.26666668, 0.26666668, 0.26666668], [1.0, 2.0, 3.0]

Element #14 stored into bucket, followed by relative index of elements in bucket:
[0.4, 0.4], [1.0, 2.0]

Element #15 stored into bucket, followed by relative index of elements in bucket:
[0.13333334, 0.13333334, 0.13333334], [1.0, 2.0, 3.0]
```

Note, the first two steps are adding the value "7" to the same bucket; it is quite evident that the first occurrence of 7 is added first, with the second 7 (with relative index "2.0") beiing added second.

The next step is to then run Insertion Sort on each of the 15 buckets. Note that several of the buckets are empty, which is okay.

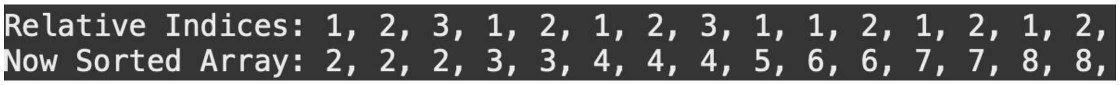


```
Sorted Bucket #1, followed by relative index
[], []
Sorted Bucket #2, followed by relative index
[], []
Sorted Bucket #3, followed by relative index
[0.13333334, 0.13333334, 0.13333334], [1.0, 2.0, 3.0]
Sorted Bucket #4, followed by relative index
[0.2, 0.2], [1.0, 2.0]
Sorted Bucket #5, followed by relative index
[0.26666668, 0.26666668, 0.26666668], [1.0, 2.0, 3.0]
Sorted Bucket #6, followed by relative index
[0.33333334], [1.0]
Sorted Bucket #7, followed by relative index
[0.4, 0.4], [1.0, 2.0]
Sorted Bucket #8, followed by relative index
[0.46666667, 0.46666667], [1.0, 2.0]
```



```
Sorted Bucket #9, followed by relative index
[0.53333336, 0.53333336], [1.0, 2.0]
Sorted Bucket #10, followed by relative index
[], []
Sorted Bucket #11, followed by relative index
[], []
Sorted Bucket #12, followed by relative index
[], []
Sorted Bucket #13, followed by relative index
[], []
Sorted Bucket #14, followed by relative index
[], []
Sorted Bucket #15, followed by relative index
[], []
```

Finally, the last step is to concatenate all the buckets containing elements together, as well as all the buckets containing elements. The final output is shown below:



```
Relative Indices: 1, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 1, 2, 1, 2,
Now Sorted Array: 2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 7, 7, 8, 8,
```

As the above output screenshot shows, the relative order of the indices in A are preserved as the algorithm runs, thus proving that Counting Sort runs in linear time and is a stable sorting algorithm.

Installation Procedure

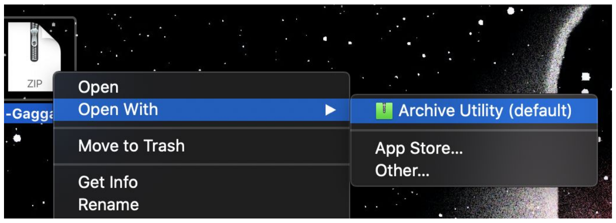
The following installation procedure is for Mac OSX. The procedure is probably very similar in Windows or Linux, but each step might be a little different. Note: A JDK must be pre-installed on the computer.

Step 1: Download the file PA2-Gaggar_Ayush.zip

The process will be easier if you save the zip file to a convenient file location, i.e. Desktop, Downloads, etc. In this procedure, I downloaded and saved the zip file on my Desktop.

Step 2: Unzip PA2-Gaggar_Ayush.zip

Double click to unzip the file. If that doesn't work, right click, click open with, and click the default Archive Utility Opener. Once the folder is unzipped, the zip file can be deleted.



Step 3: Navigating the folder PA2-Gaggar_Ayush

Within the folder will be multiple documents, including this formal report, a folder called pa2_CS146 which contains the java classes themselves, and a jar executable file called pa2.jar

Step 4: Running the jar file on your computer

On Mac, open up Terminal. (If you cannot find it, go to the top right of your screen and click the search icon (the magnifying glass). Type Terminal and open the application).

Note: The below commands might be different if you saved the file at a different location.

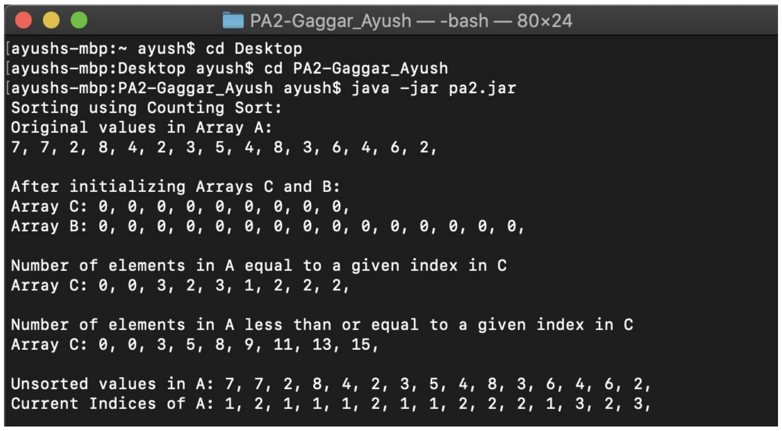
Type the following into the terminal screen, and hit enter in between commands:

cd Desktop (Optional follow-up Command to see what files are on your Desktop: ls)

cd PA2-Gaggar_Ayush

java -jar pa2.jar

Upon hitting enter on that last command, the following should begin printing out:



```
PA2-Gaggar_Ayush — -bash — 80×24
ayushs-mbp:~ ayush$ cd Desktop
ayushs-mbp:Desktop ayush$ cd PA2-Gaggar_Ayush
ayushs-mbp:PA2-Gaggar_Ayush ayush$ java -jar pa2.jar
Sorting using Counting Sort:
Original values in Array A:
7, 7, 2, 8, 4, 2, 3, 5, 4, 8, 3, 6, 4, 6, 2,

After initializing Arrays C and B:
Array C: 0, 0, 0, 0, 0, 0, 0, 0, 0,
Array B: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

Number of elements in A equal to a given index in C
Array C: 0, 0, 3, 2, 3, 1, 2, 2, 2,

Number of elements in A less than or equal to a given index in C
Array C: 0, 0, 3, 5, 8, 9, 11, 13, 15,

Unsorted values in A: 7, 7, 2, 8, 4, 2, 3, 5, 4, 8, 3, 6, 4, 6, 2,
Current Indices of A: 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 2, 1, 3, 2, 3,
```

At this point, the program will run as required, and will terminate upon completion. To re-run the program, simply input "java -jar pa2.jar" into Terminal again!

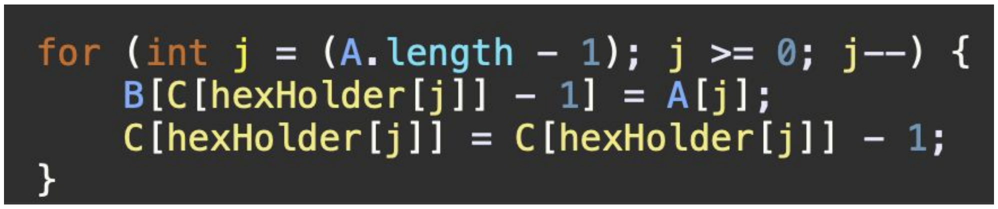
Problems Encountered During Implementation

- Moving Numbers along with Sorting in Radix Sort

  The largest obstacle I encountered during this assignment was when sorting a given digit in the number, how do I ensure the rest of the number moves positions with this digit when it is being sorted? Radix Sort works by sorting the input numbers from their least significant digit to their most significant digit. As each digit gets sorted, the entire number should move with the digit being sorted. As the algorithm loops through each digit, the number should become sorted.

  It was relatively straightforward to follow the pseudocode for the radixSort method; however, I soon realized that although each digit was being sorted, the position of the numbers themselves wasn't being rearranged. How can I "float" the rest of the hex number along with that digit up the column to its correct location? My solution to this was to create a custom, stable sorting algorithm that the radixSort method would use for each of the 5 digits in the hexadecimal number.

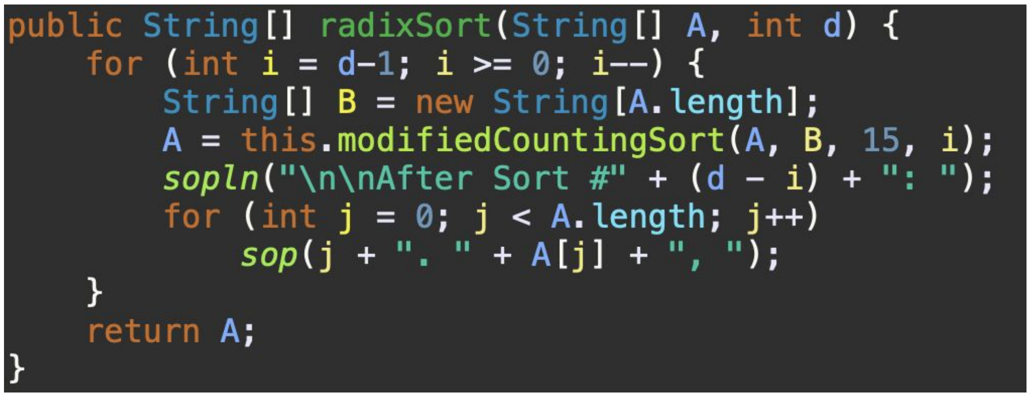
  This stable sorting algorithm was modified from the Counting Sort algorithm in Exercise #1, with several key differences. The first difference was that the input and output arrays were all of type String, as the hex number could also contain a letter between a-f along with the 0-9 digits. Second, there was an added input parameter called digit, which directed the Counting Sort algorithm to run on the given digit. Within this method, I created an additional auxiliary array called hexHolder, which contained an integer representation of the given digit of the hex number. The countingSort method continued as normal, but instead used the hexHolder array as the one to sort. The final and most critical difference is in the final loop. Below is a screenshot from the method which demonstrates how B results in a String array of the full hex number, now sorted based on the input digit.



```
for (int j = (A.length - 1); j >= 0; j--) {
    B[C[hexHolder[j]] - 1] = A[j];
    C[hexHolder[j]] = C[hexHolder[j]] - 1;
}
```

  Instead of assigning to B the appropriate value from hexHolder, the algorithm uses the sorted index of the element in hexHolder to assign the full String representation of the hex number to the value in B.

  Now, the radixSort method will call the modifiedCountingSort method d times on the input array, where d represents the number of digits in each number of the input array. After each call, the result of modifiedCountingSort is again assigned to the input array A, so that each subsequent call to modifiedCountingSort will be based off of the sorting of the previous digit. The implementation of radixSort is shown below, with several print statements to show what the array would look like after each digit is sorted, from the least significant digit, to the most significant one.



```
public String[] radixSort(String[] A, int d) {
    for (int i = d-1; i >= 0; i--) {
        String[] B = new String[A.length];
        A = this.modifiedCountingSort(A, B, 15, i);
        sopln("\n\nAfter Sort #" + (d - i) + ": ");
        for (int j = 0; j < A.length; j++)
            sop(j + ". " + A[j] + ", ");
    }
    return A;
}
```

- Data Structure Usage for Bucket Sort
  The pseudocode for Bucket Sort says to use an array B to hold a series of LinkedLists, each of which would hold a value between [0,1). The first challenge here is to convert all input values such that they fall within the acceptable range. Because it is not possible to have integer values in that range, I changed the input to an array containing float values for the values given in this assignment. It was then just a simple step to divide each element by the length of the array to ensure that each element was within the range.

  BucketSort requires an auxiliary data structure where each element in the structure can contain a LinkedList. Therefore, I created an ArrayList which aggregated LinkedList<Float>; it was necessary for LinkedList to aggregate type Float because the values inside each bucket would be the float value found in the first paragraph of this sub-bullet above. Using an ArrayList to hold these LinkedLists allowed for an easy way to add empty LinkedLists to each element in the ArrayList, and later adding an element from the input float array to the correct sub-interval "bucket" (LinkedList). Additionally, it is easy to set a relationship between elements in an ArrayList, a feature necessary for this algorithm as it would be "linking" together (concatenating) each of the individual LinkedLists to output a series of floats now sorted in order. The relevant code for this part is shown below, with the next sub-bullet depicting an explanation as to how the challenge of showing relative indices was resolved.

```
ArrayList<LinkedList<Float>> B = new ArrayList<LinkedList<Float>>(2*n);
for (int i = 0; i < (2*n); i++)
    B.add(new LinkedList<Float>());

for (int i = 0; i < n; i++) {
    B.get((int) (n*A[i])).add(A[i]);
    B.get((int) (n*A[i] + n)).add((float) index[i]);
    sopln("Element #" + (i+1) + " stored into bucket, "
            + "followed by relative index of elements in bucket: ");
    sop(B.get((int) (n*A[i])) + ", ");
    sopln(B.get((int) (n*A[i] + n)));
}
```

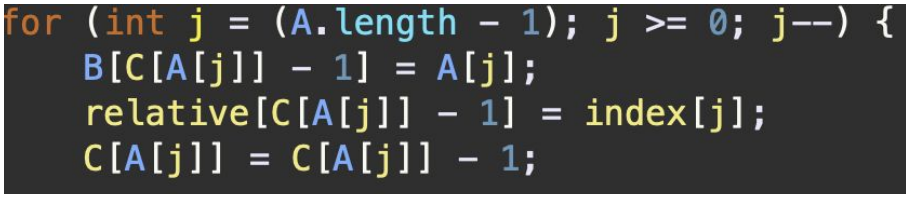
- Showing Relative Index for Counting and Bucket Sort
  Another challenge in this programming assignment was coming up with a way to show the relative index/order for elements in A. My first idea was to change the entire input to doubles, with values like 7.1 to indicate that it was the first occurrence for 7, 3.2 to indicate the second occurrence for 3, etc. I quickly realized that this solution would be too troublesome to implement in CountingSort, which expects an input of integers. Additionally, "7.1" is not an accurate representation for the first occurrence of the number 7, as 7.1 is a completely different number than 7.

  The next attempted solution consisted of making a two-dimensional array and sorting both consecutively, with one dimension for the input array and the second dimension for containing the relative indices for the array. This solution clearly proved to be too much work in organizing the code to sort through the given index for both dimensions. However, this idea did lead down the path to what the final implementation looks like. I *did* need a way to sort the relative indices in the same way the input array was being sorted.

  The final solution was to hard code an array of ints called index, which contained the relative order of occurrences for the input array. This solution was common to both Counting Sort and Bucket Sort; however, both classes then used the below array in different ways to track relative indices

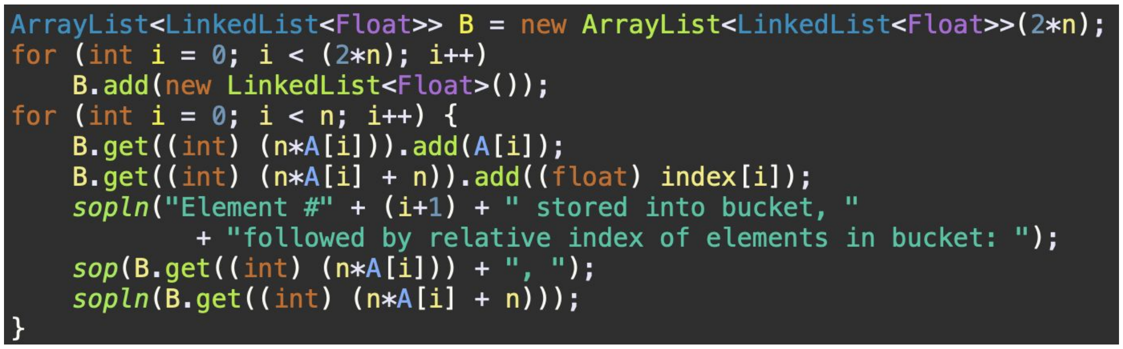
```
int[] index = new int[] {1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 2, 1, 3, 2, 3};
    //index contains the relative order of occurrences for the elements in A
```

  Counting Sort proceeded to treat index similarly to an input array into Counting Sort, where it also needed an auxiliary array where it would store the final location of the relative indices; this array was called relative. The algorithm took the same sorted location of the element in A to move the element in index to its correct location in relative. Below is a snippet of the code that depicts the process of sorting the index into the correct location in relative along with sorting A into its correct location in B, both using Array C as an intermediary.



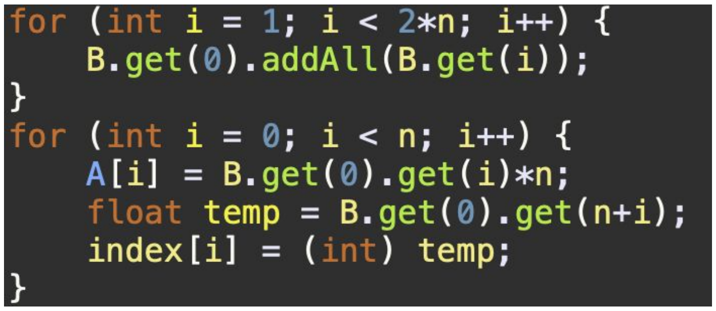
```
for (int j = (A.length - 1); j >= 0; j--) {
    B[C[A[j]] - 1] = A[j];
    relative[C[A[j]] - 1] = index[j];
    C[A[j]] = C[A[j]] - 1;
```

Bucket Sort took a slightly different approach to store the relative indices. Because Bucket Sort created an ArrayList whose elements contained LinkedLists, it was easy to modify the length of the ArrayList to include all the elements in the array index as well. The first n elements in the ArrayList (where n represents the length of the input array) contain the elements of the input Array in their sorted position; the next n elements will then contain the elements in the array index, moved to the same location as the input elements, simply shifted n elements down the ArrayList. In this way, whatever sorting was occurring in the first n elements, the same process would be mirrored for the next n elements; this was a roundabout way to “link” the elements with their index such that whatever motion happened to the input element, the same sorting motion would move the index element. The relevant code is shown below:



```
ArrayList<LinkedList<Float>> B = new ArrayList<LinkedList<Float>>(2*n);
for (int i = 0; i < (2*n); i++)
    B.add(new LinkedList<Float>());
for (int i = 0; i < n; i++) {
    B.get((int) (n*A[i])).add(A[i]);
    B.get((int) (n*A[i] + n)).add((float) index[i]);
    sopln("Element #" + (i+1) + " stored into bucket, "
            + "followed by relative index of elements in bucket: ");
    sop(B.get((int) (n*A[i])) + ", ");
    sopln(B.get((int) (n*A[i] + n)));
}
```

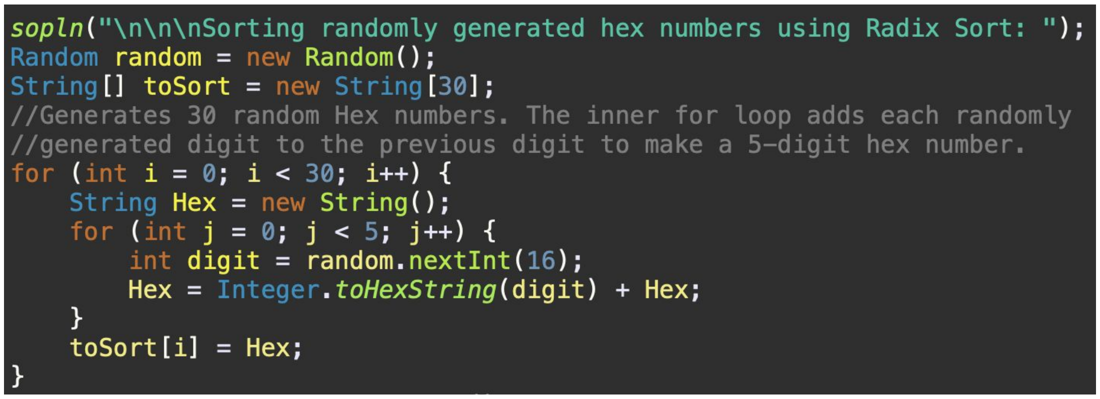
After concatenating the ArrayList into one LinkedList, it's fairly simple to assign the first n elements back into the array A, and then next n elements back into the array index. In this way, the result would be a sorted version of the input array, along with each element’s respective index as well. The relevant code is shown below:



```
for (int i = 1; i < 2*n; i++) {
    B.get(0).addAll(B.get(i));
}
for (int i = 0; i < n; i++) {
    A[i] = B.get(0).get(i)*n;
    float temp = B.get(0).get(n+i);
    index[i] = (int) temp;
}
```

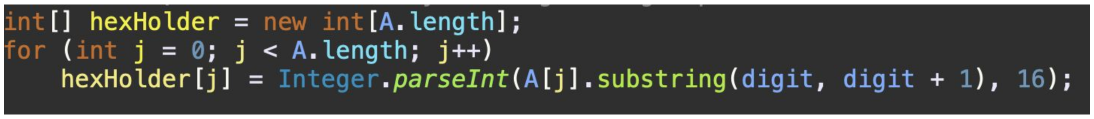
- Converting Input and Output Elements Appropriately for Radix Sort
  Exercize #2 in this Assignment called for using Radix Sort on 30 randomly generated hexadecimal numbers. Before this assignment, I had only a vague idea of what a hexadecimal number was, and no clue as to how to implement and use hexadecimal numbers in actual programming. The first step was to what the hexadecimal notation was, and then learn appropriate methods which allowed for me to convert a given integer to its hexadecimal notation using the Integer.toHexString(int) method. Because hex numbers are a way to represent integers with a 16-base system, I needed to create a random integer between 0 and 15, for a total of 16 possibilities, and then convert that integer to its hex variation.

  That entire procedure was just to create one "digit", which was actually a String. I needed to repeat this process 4 more times, each time appending the randomly created digit to the beginning of the others in order to create a String with 5 digits, each in a hexadecimal format. The code is shown below.



```
sopln("\n\n\nSorting randomly generated hex numbers using Radix Sort: ");
Random random = new Random();
String[] toSort = new String[30];
//Generates 30 random Hex numbers. The inner for loop adds each randomly
//generated digit to the previous digit to make a 5-digit hex number.
for (int i = 0; i < 30; i++) {
    String Hex = new String();
    for (int j = 0; j < 5; j++) {
        int digit = random.nextInt(16);
        Hex = Integer.toHexString(digit) + Hex;
    }
    toSort[i] = Hex;
}
```

  Now, the challenge was implementing modifiedCountingSort such that it would be able to read the various String inputs and be able to sort it. My solution here was to create the auxiliary array hexHolder as mentioned above. This array would contain the letter that Counting Sort would be sorting. In order to accomplish this, I used the substring(startIndex, endIndex) method in the String class combined with the method Integer.parseInt(String, base) in order to convert the specified letter to a number between 0 and 15. CountingSort can now continue as specified by the pseudocode, as it is the best linear and stable sorting algorithm to sort integers between a given range. The code for the hexHolder array is shown below:

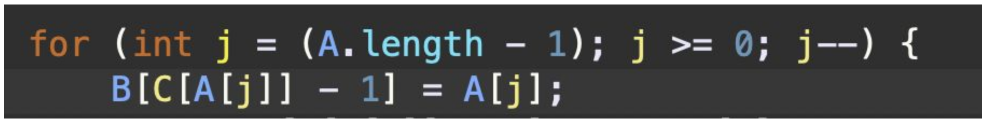


```
int[] hexHolder = new int[A.length];
for (int j = 0; j < A.length; j++)
    hexHolder[j] = Integer.parseInt(A[j].substring(digit, digit + 1), 16);
```

The final challenge for the Radix Sort algorithm was to ensure that the sorting started from the least significant digit (LSD) to the most significant digit (MSD). When initially following the pseudocode, the method started at digit 1 and went up to digit d. At first, I was puzzled. Why was the output result not sorted? Upon analyzing what the output actually looked like after each digit was sorted, I realized that the algorithm was actually starting from the MSD and sorted the LSD last.

The solution to this was quite simple, and involved just a simple change in the bounds of the for loop in the radixSort method. Instead of starting at the first digit, we simply need to start at the last digit, and loop up to the MSD. The screenshot of the method is also shown above, where the important line of code is line 54, which is: for (int i = d-1; i >= 0; i--).

- Using Arrays in Counting and Radix Sort
  The decision to use arrays versus a different data structure for these two classes was dependent on the implementation specified by the pseudocode. Quite simply, there was no cause to use a more robust data structure, i.e. ArrayList, because an Array was able to accomplish the same task easily and cleanly. The only problem encountered in these two classes was modifying the code such that the starting index was 0 instead of 1. Especially in CountingSort, the pseudocode states that arrays A and B start at index 1, and array C starts at index 0. This issue often caused a NullPointerException or an ArrayOutOfBoundsException; however, by writing print statements in the code and walking through the logic on paper, it was simple to find and correct the programming flaws. The most troublesome array to track placement of elements by index was array B, but the highlighted line of code below did the trick ("-1") to place elements in A to their correct location in B:



```
for (int j = (A.length - 1); j >= 0; j--) {
    B[C[A[j]] - 1] = A[j];
```

Lesson Learned

This programming assignment has been more in line with the level of experience and comfort I have in programming. I am still an amateur in Java Programming, continuing to strengthen my understanding of the fundamentals of Java and object oriented programming, and developing the logic required to tackle larger projects such as this one. Thus far, the largest projects I had worked on were from the Intro to Data Structures class offered at SJSU, CS 46B, many of which were basic homework assignments or lab work which could be completed in a matter of hours. This project was more complex than those, and took a week to fully develop and test three robust linear sorting algorithms. In order to do so, it required a thorough understanding of the Linear Sorting Algorithms as well as their implementation in order to truly be successful. Furthermore, it expanded and strengthened the programming skills I had, and gave me exposure to what a sample project in the workplace might look like, in customizing sorting algorithms to work for input data types the company might be working with.

By developing and debugging each of the sorting algorithms, I was truly able to grasp the inner workings of what makes each line of code in the algorithm function as it does. An understanding of each line of code results in an understanding of the code; it is not possible to have one without the other. Although the code was a direct translation of the pseudocode from the textbook, actually coding it for myself and debugging it helped me understand what exactly was occurring behind the scenes at every line of code. As mentioned in the Problems Encountered section, the biggest learning point was figuring out how to "float" and sort the numbers in radix sort when sorting by digit. In principle and drawing out diagrams, it is easy to draw the entire number moving along with the digit; however, the actual implementation in practice is more challenging. Through the process of debugging, I was able to concretely understand how Radix Sort worked in sorting a number by each digit, and how Bucket Sort adds elements to their correct buckets..

Some of the other errors I encountered helped strengthen my understanding of the basics of Java, and developed my logical thinking skills. For me, Programming Assignment #1 was more about learning a ton of new concepts, such as simulation, Input/Output Streams, the HeapSort algorithm, etc; on the other hand, Programming Assignment #2 was a much more straightforward assignment. The challenge in this one was not so much how much new information can you learn, but rather how well can you think and logically implement each one of these sorting algorithms. For example, there was a lot of flexibility in deciding which data structures to use for a given part of a sorting algorithm. Deciding between them and then implementing them correctly, accounting for starting index, valid ranges, incorporating relative indices, etc was a challenging series of logical problem solving.

This programming assignment has really inspired me to learn more about the purpose of various sorting algorithms, and how best to streamline efficiency for each one. I truly love logical problem solving, which is at the heart of Computer Science in general; logically approaching this assignment and tackling it in what I see as the best way possible really appealed to my logical nature. Developing these algorithms inspired me to come up with innovative ways to prove stability of algorithms and show the intermediate steps and inner workings of an algorithm in ways not possible through a textbook, classroom format; I believe that with more time, I would learn how to make each sorting algorithm take in generic types such that it would be more applicable in a broader scope of applications. Further functionality could easily be added into this program by allowing users to add in values to sort, and perhaps even combining these sorting algorithms into an exercise like Programming Assignment #1, in sorting a series of objects by a given key value.

In the future, I'd love to work in the industry with the data structures knowledge I have right now. Obviously, over the next couple years at SJSU, I want to continue learning in-depth coding practices related to Bioinformatics and Artificial Intelligence along with mastering data structures; I hope to be able to combine my knowledge from these three fields in Computer Science with the pursuit of my Mechanical Engineering degree as well to work with robotics in the future, ideally in the healthcare industry. The first stepping stone to that would be landing an internship to truly test and grow my CS knowledge and establish an educational foundation in robotics. I hope to someday take my passion and expertise towards founding a startup focused on improving current healthcare practices using advanced, efficient algorithms controlling robotics. I'm extremely fortunate that this programming assignment was able to strengthen my grasp and confidence of programming data structures while deepening my knowledge of how sorting algorithms work!