**Linear Sorting Algorithms**

**The Second Programming Assignment (PA-2)**

Ahror Abdulhamidov

CS-146, Instructor: Dr. Mike Wu

San Jose State University

November 13th, 2020

**Design and Implementation**

This application’s purpose is to show the effectiveness of the three standard linear sorting algorithms.

The application consists of three runnable main classes, CountingSort, RadixSort, and BucketSort, and each of them has a main method to test certain linear sorting algorithm and methods for that sorting algorithm. In addition, there are two wrapper classes, Number and DoubleNumber, for storing and integer or double number with its relative number (count) in an array or ArrayList.

Counting sort algorithm is similar to the one in the book, with a difference that instead of arrays of integers it uses arrays A and B of Numbers that have both value and count for an integer. The auxiliary array C for counting sort uses just an array of integers.

Radix sort algorithm is similar to the one in the book, with a difference that it is for hexadecimal numbers, which is stored as an array A of Strings. Also, before doing counting sort for the array A by each hex-digit, it creates an empty array B of hex Strings. The counting sort here works similar to the one in the book, but for hex numbers (Strings). When dealing with the auxiliary array C of integers for counting sort in this exercise, the algorithm right there converts hex characters to integer (from 0 to 15), and vice versa. After each counting sort for a digit, it stores the sorted by digit array B back to the array A.

Bucket sort algorithm is similar to the one in the book, with a difference that it accepts an array A of DoubleNumbers (to store both value and count) and an integer n, which represents the number of buckets. Since the original array A is not normalized, in the beginning of bucket sort, it gets normalized (every element’s value divided by n) and then in the end – un-normalized (back to how they were). An array of buckets is created in the process which represents an array of ArrayLists of DoubleNumbers. Instead of doing the full insertion sort for each bucket, it uses the insert method (helper function) that inserts value to a bucket in the right position, by comparing with other elements in that bucket – similar to insertion sort. Once buckets are filled, all of them are concatenated together to one big ArrayList of DoubleNumbers, and its values are then un-normalized and stored back in the original array A.

**List of Classes and Subroutines**

**CountingSort class:**

Has a static array A with initialized integer numbers.

Has three static methods: main, setCounts, and countingSort.

SetCounts method accepts an array of Numbers and sets counts for numbers that repeat in the array. The method is hard coded.

CountingSort method accepts two arrays of Numbers, A and B, and an integer k – the max value of the numbers in the array. The detailed description of this method can be find in the Design and Implementation section of this document.

The main method test the counting sort algorithm. First, it creates an array A of Numbers out of the array of integers that needs to be sorted and an array B of Numbers, filled with zeros. Then, it sorts the array A and stores the result in B. During all of that, it prints the arrays before, during, and after the sorting, and in the end proves the stability of the algorithm.

This class strongly uses the Number class objects and its methods.

**Number class:**

Has two properties – int value and int count.

Has basic constructor, getters and setters.

Has four static methods: 2 overloaded makeNumberArray methods, getValues, and getCounts.

Also, has a toString method that just returns the value of the Number as a String.

The first makeNumber method accepts an array of integers and returns an array of Numbers, created from the array of integers.

The second overloaded makeNumber method accepts an integer length, creates an array of Numbers of that length filled with zeros, and returns that array.

The method getValues accepts an array of Numbers and returns a string of values for all numbers in the array.

The method getCounts accepts an array of Numbers and returns a string of counts for all numbers in the array.

**RadixSort class:**

Has a static array A of Strings.

Has five static methods: main, makeRandomHexArray, hexRadixSort, hexCountingSort, hexToInt.

The makeRandomHexArray method accepts an int length and int maxDigit, creates an array of the specified length, consisting of randomly generated hex-Strings of the specified number of digits, and returns that array.

The hexRadixSort accepts an array A of hex-Strings and int maxDigit – a number of digits in each hex-String. It sorts the array A by each digit using counting sort. The detailed description of this method can be find in the Design and Implementation section of this document.

The hexCountingSort method accepts an array A of hex-Strings, an empty array B of Strings, and an int d – the digit by which it is sorting. The detailed description of this method can be find in the Design and Implementation section of this document.

The hexToInt method simply converts a hex-character to an integer.

The main method tests the radix sort for the hex-String array. It first generates a random hex-String array and then sorts is. Before, during, and after sorting, it displays the array.

**BucketSort class:**

Has a static array A filled with some doubles.

It has six static methods: main method, setCounts, bucketSort, insert, concatenateLists, and getBuckets.

SetCounts method accepts an array of DoubleNumbers and sets counts for numbers that repeat in the array. The method is hard coded.

The bucketSort method accepts an array A of DoubleNumbers and int n – the number of buckets. It sorts the array A using bucket sort. The detailed description of this method can be find in the Design and Implementation section of this document.

The insert method an ArrayList of DoubleNumbers and a DoubleNumber. It inserts the DoubleNumber in the right position in the array list, so that it maintains the order, by comparing with other existing elements (similar to insertion sort).

The concatenateLists method accepts an array B of ArrayLists of DoubleNumbers (array of buckets), concatenates the buckets into one big ArrayList of DoubleNumbers, and returns it.

The getBuckets method accepts an array B of ArrayLists of DoubleNumbers (array of buckets) and returns all the values of the DoubleNumbers in those buckets as a String.

This class strongly uses the DoubleNumber class objects and its methods.

**DoubleNumber class**

Has two properties – double value and int count.

Has basic constructor, getters and setters.

Has four static methods: 2 overloaded makeNumberArray methods, getValues, getCounts, normalize, and unnormalize.

Also, has a toString method that just returns the value of the DoubleNumber as a String.

The first makeNumber method accepts an array of doubles and returns an array of DoubleNumbers, created from the array of doubles.

The second overloaded makeNumber method accepts an integer length, creates an array of DoubleNumbers of that length filled with zeros, and returns that array.

The method getValues accepts an array of DoubleNumbers and returns a string of values for all numbers in the array.

The method getCounts accepts an array of DoubleNumbers and returns a string of counts for all numbers in the array.

The normalize method accepts and array A of DoubleNumbers and an int n – number of buckets. It divides all values of the elements in the array by n.

The unnormalize method accepts and array A of DoubleNumbers and an int n – number of buckets. It multiplies all values of the elements in the array by n.

**Testing Screenshots**

**Output for Counting Sort:**

Array of numbers and their counts before sorting:

[8, 8, 2, 9, 4, 2, 3, 5, 4, 9, 3, 7, 4, 7, 2]

[1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 2, 1, 3, 2, 3]

Begin Counting Sort:

Array B: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Array C: [0, 0, 3, 5, 8, 9, 9, 11, 13, 15]

Array B: [0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Array C: [0, 0, 2, 5, 8, 9, 9, 11, 13, 15]

Array B: [0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0]

Array C: [0, 0, 2, 5, 8, 9, 9, 10, 13, 15]

Array B: [0, 0, 2, 0, 0, 0, 0, 4, 0, 0, 7, 0, 0, 0, 0]

Array C: [0, 0, 2, 5, 7, 9, 9, 10, 13, 15]

Array B: [0, 0, 2, 0, 0, 0, 0, 4, 0, 7, 7, 0, 0, 0, 0]

Array C: [0, 0, 2, 5, 7, 9, 9, 9, 13, 15]

Array B: [0, 0, 2, 0, 3, 0, 0, 4, 0, 7, 7, 0, 0, 0, 0]

Array C: [0, 0, 2, 4, 7, 9, 9, 9, 13, 15]

Array B: [0, 0, 2, 0, 3, 0, 0, 4, 0, 7, 7, 0, 0, 0, 9]

Array C: [0, 0, 2, 4, 7, 9, 9, 9, 13, 14]

Array B: [0, 0, 2, 0, 3, 0, 4, 4, 0, 7, 7, 0, 0, 0, 9]

Array C: [0, 0, 2, 4, 6, 9, 9, 9, 13, 14]

Array B: [0, 0, 2, 0, 3, 0, 4, 4, 5, 7, 7, 0, 0, 0, 9]

Array C: [0, 0, 2, 4, 6, 8, 9, 9, 13, 14]

Array B: [0, 0, 2, 3, 3, 0, 4, 4, 5, 7, 7, 0, 0, 0, 9]

Array C: [0, 0, 2, 3, 6, 8, 9, 9, 13, 14]

Array B: [0, 2, 2, 3, 3, 0, 4, 4, 5, 7, 7, 0, 0, 0, 9]

Array C: [0, 0, 1, 3, 6, 8, 9, 9, 13, 14]

Array B: [0, 2, 2, 3, 3, 4, 4, 4, 5, 7, 7, 0, 0, 0, 9]

Array C: [0, 0, 1, 3, 5, 8, 9, 9, 13, 14]

Array B: [0, 2, 2, 3, 3, 4, 4, 4, 5, 7, 7, 0, 0, 9, 9]

Array C: [0, 0, 1, 3, 5, 8, 9, 9, 13, 13]

Array B: [2, 2, 2, 3, 3, 4, 4, 4, 5, 7, 7, 0, 0, 9, 9]

Array C: [0, 0, 0, 3, 5, 8, 9, 9, 13, 13]

Array B: [2, 2, 2, 3, 3, 4, 4, 4, 5, 7, 7, 0, 8, 9, 9]

Array C: [0, 0, 0, 3, 5, 8, 9, 9, 12, 13]

Array B: [2, 2, 2, 3, 3, 4, 4, 4, 5, 7, 7, 8, 8, 9, 9]

Array C: [0, 0, 0, 3, 5, 8, 9, 9, 11, 13]

End of Counting Sort

Array of numbers and their counts after sorting:

[2, 2, 2, 3, 3, 4, 4, 4, 5, 7, 7, 8, 8, 9, 9]

[1, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 1, 2, 1, 2]

The array of the counts indicates that the Counting Sort is stable.

**Source Code for Counting Sort:**

*// CS-146 - Ahror Abdulhamidov  
// Class for Counting Sort  
  
import* java.util.Arrays;  
  
*public class* CountingSort {  
  
 *// Array of numbers  
 private static int*[] *array* = {8, 8, 2, 9, 4, 2, 3, 5, 4, 9, 3, 7, 4, 7, 2};  
  
 *// Main method  
 public static void* main(String[] args) {  
  
 *// Prepare the array of numbers* Number[] A = Number.*makeNumberArray*(*array*);  
 *setCounts*(A);  
 System.*out*.println("Array of numbers and their counts before sorting:");  
 System.*out*.println(Number.*getValues*(A));  
 System.*out*.println(Number.*getCounts*(A));  
  
 *// Counting Sort* System.*out*.println("\nBegin Counting Sort:");  
 Number[] B = Number.*makeNumberArray*(15);  
 *countingSort*(A, B, 9);  
 System.*out*.println("End of Counting Sort");  
  
 *// Show the sorting results and prove the stability* System.*out*.println("\nArray of numbers and their counts after sorting:");  
 System.*out*.println(Number.*getValues*(B));  
 System.*out*.println(Number.*getCounts*(B));  
 System.*out*.println(  
 "The array of the counts indicates that the Counting Sort is stable."  
 );

}  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Manually sets counts for each Number in the array  
 \*/  
 private static void* setCounts(Number[] A) {  
 A[1].setCount(2);  
 A[5].setCount(2);  
 A[8].setCount(2);  
 A[9].setCount(2);  
 A[10].setCount(2);  
 A[12].setCount(3);  
 A[13].setCount(2);  
 A[14].setCount(3);  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Does Counting Sort for the array of Numbers A[] and stores the sorted array in B[]  
 \* Auxiliary array for the Counting Sort is of specified size k  
 \*/  
 private static void* countingSort(Number[] A, Number[] B, *int* k) {  
 System.*out*.println("Array B: " + Number.*getValues*(B));  
 *int*[] C = *new int*[k + 1];  
 *for* (*int* i = 0; i <= k; i++)  
 C[i] = 0;  
 *for* (Number number : A)  
 C[number.getValue()]++;  
 *for* (*int* i = 1; i <= k; i++)  
 C[i] += C[i - 1];  
 System.*out*.println("Array C: " + Arrays.*toString*(C));  
 *for* (*int* j = A.length - 1; j >= 0; j--) {  
 C[A[j].getValue()]--;  
 B[C[A[j].getValue()]] = A[j];  
 System.*out*.println("\nArray B: " + Number.*getValues*(B));  
 System.*out*.println("Array C: " + Arrays.*toString*(C));  
 }  
 }  
  
}

*// CS-146 - Ahror Abdulhamidov  
// Number class - to store numbers with their count numbers  
  
public class* Number {  
  
 *private int* value; *// Integer value of the number  
 private int* count; *// Relative number of the number  
  
 public* Number(*int* value, *int* count) {  
 *this*.value = value;  
 *this*.count = count;  
 }  
  
 *public int* getValue() {  
 *return* value;  
 }  
  
 *public void* setCount(*int* count) {  
 *this*.count = count;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Creates and returns an array of Numbers from an array of integers  
 \*/  
 public static* Number[] makeNumberArray(*int*[] array) {  
 Number[] numbers = *new* Number[array.length];  
 *for* (*int* i = 0; i < numbers.length; i++) {  
 numbers[i] = *new* Number(array[i], 1);  
 }  
 *return* numbers;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Creates and returns an array of Numbers (zeros) of the specified length  
 \*/  
 public static* Number[] makeNumberArray(*int* length) {  
 Number[] numbers = *new* Number[length];  
 *for* (*int* i = 0; i < numbers.length; i++) {  
 numbers[i] = *new* Number(0, 1);  
 }  
 *return* numbers;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Returns String of values for each Number in the array  
 \*/  
 public static* String getValues(Number[] numbers) {  
 StringBuilder sb = *new* StringBuilder();  
 sb.append("[").append(numbers[0].value);  
 *for* (*int* i = 1; i < numbers.length; i++) {  
 sb.append(", ").append(numbers[i].value);  
 }  
 sb.append("]");  
 *return* sb.toString();  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Returns String of counts for each Number in the array  
 \*/  
 public static* String getCounts(Number[] numbers) {  
 StringBuilder sb = *new* StringBuilder();  
 sb.append("[").append(numbers[0].count);  
 *for* (*int* i = 1; i < numbers.length; i++) {  
 sb.append(", ").append(numbers[i].count);  
 }  
 sb.append("]");  
 *return* sb.toString();  
 }  
  
 *// Returns String value of the Number* @Override  
 *public* String toString() {  
 *return* Integer.*toString*(value);  
 }  
}

**Output for Radix Sort:**

Array of numbers and their counts before sorting:

[80df9, 832ee, c7507, 2c419, 09483, f8944, 55c0a, 13e20, e1778, 98134, 1f144, 6f32b, 76276, 1bd4e, b59d9, ef258, 4fa42, 1e0f9, 7fbbe, ff4f5, 2fce6, 208bd, d64a0, d62ec, 3f051, e17af, 7ffa4, 6252f, d7175, a4c4e]

Begin Radix Sort:

Array, sorted by the digit 5: [13e20, d64a0, 3f051, 4fa42, 09483, f8944, 98134, 1f144, 7ffa4, ff4f5, d7175, 76276, 2fce6, c7507, e1778, ef258, 80df9, 2c419, b59d9, 1e0f9, 55c0a, 6f32b, d62ec, 208bd, 832ee, 1bd4e, 7fbbe, a4c4e, e17af, 6252f]

Array, sorted by the digit 4: [c7507, 55c0a, 2c419, 13e20, 6f32b, 6252f, 98134, 4fa42, f8944, 1f144, 1bd4e, a4c4e, 3f051, ef258, d7175, 76276, e1778, 09483, d64a0, 7ffa4, e17af, 208bd, 7fbbe, b59d9, 2fce6, d62ec, 832ee, ff4f5, 80df9, 1e0f9]

Array, sorted by the digit 3: [3f051, 1e0f9, 98134, 1f144, d7175, ef258, 76276, d62ec, 832ee, 6f32b, 2c419, 09483, d64a0, ff4f5, c7507, 6252f, e1778, e17af, 208bd, f8944, b59d9, 4fa42, 7fbbe, 55c0a, a4c4e, 2fce6, 1bd4e, 80df9, 13e20, 7ffa4]

Array, sorted by the digit 2: [208bd, 80df9, e1778, e17af, 6252f, 832ee, 13e20, a4c4e, b59d9, 55c0a, 76276, d62ec, d64a0, d7175, c7507, 98134, f8944, 09483, 1bd4e, 2c419, 1e0f9, 3f051, 1f144, ef258, 6f32b, ff4f5, 4fa42, 7fbbe, 2fce6, 7ffa4]

Array, sorted by the digit 1: [09483, 13e20, 1bd4e, 1e0f9, 1f144, 208bd, 2c419, 2fce6, 3f051, 4fa42, 55c0a, 6252f, 6f32b, 76276, 7fbbe, 7ffa4, 80df9, 832ee, 98134, a4c4e, b59d9, c7507, d62ec, d64a0, d7175, e1778, e17af, ef258, f8944, ff4f5]

End of Radix Sort

Array of hex numbers after sorting:

[09483, 13e20, 1bd4e, 1e0f9, 1f144, 208bd, 2c419, 2fce6, 3f051, 4fa42, 55c0a, 6252f, 6f32b, 76276, 7fbbe, 7ffa4, 80df9, 832ee, 98134, a4c4e, b59d9, c7507, d62ec, d64a0, d7175, e1778, e17af, ef258, f8944, ff4f5]

**Source Code for Radix Sort**

*// CS-146 - Ahror Abdulhamidov  
// Class for Radix Sort  
  
import* java.util.Arrays;  
*import* java.util.Random;  
  
*public class* RadixSort {  
  
 *// Empty array of hex Strings  
 private static* String[] *A*;  
  
 *// Main method  
 public static void* main(String[] args) {  
  
 *// Prepare the array of hex Strings  
 A* = *makeRandomHexArray*(30, 5);  
 System.*out*.println("Array of numbers and their counts before sorting:");  
 System.*out*.println(Arrays.*toString*(*A*));  
  
 *// Radix Sort* System.*out*.println("\nBegin Radix Sort: ");  
 *hexRadixSort*(*A*, 5);  
 System.*out*.println("End of Radix Sort");  
  
 *// Show the sorting results* System.*out*.println("\nArray of hex numbers after sorting:");  
 System.*out*.println(Arrays.*toString*(*A*));  
  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Creates an array of specified length consisting of random hex strings  
 \* of specified number of digits filled with zeros and returns it  
 \*/  
 private static* String[] makeRandomHexArray(*int* length, *int* maxDigit) {  
 String[] hexArray = *new* String[length];  
 Random r = *new* Random();  
 *for* (*int* i = 0; i < length; i++) {  
 StringBuilder sb = *new* StringBuilder();  
 *for* (*int* d = 0; d < maxDigit; d++) {  
 sb.append(Integer.*toHexString*(r.nextInt(16)));  
 }  
 hexArray[i] = sb.toString();  
 }  
 *return* hexArray;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Does Radix Sort for the Array of hex Strings A[]  
 \*/  
 private static void* hexRadixSort(String[] A, *int* maxDigit) {  
 *for* (*int* d = maxDigit - 1; d >= 0; d--) {  
 String[] B = *new* String[A.length];  
 *hexCountingSort*(A, B, d);  
 *for* (*int* i = 0; i < A.length; i++) {  
 A[i] = B[i];  
 }  
 System.*out*.println("Array, sorted by the digit " +  
 (d + 1) + ": " + Arrays.*toString*(A));  
 }  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Does Counting Sort by the d-th digit for the Array of hex Strings A[]  
 \* and stores it in B[]  
 \*/  
 private static void* hexCountingSort(String[] A, String[] B, *int* d) {  
 *int*[] C = *new int*[16];  
 *for* (*int* i = 0; i < 16; i++)  
 C[i] = 0;  
 *for* (String a : A)  
 C[*hexToInt*(a.charAt(d))]++;  
 *for* (*int* i = 1; i < 16; i++)  
 C[i] += C[i - 1];  
 *for* (*int* j = A.length - 1; j >= 0; j--) {  
 C[*hexToInt*(A[j].charAt(d))]--;  
 B[C[*hexToInt*(A[j].charAt(d))]] = A[j];  
 }  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Converts a hex character into an integer and returns it  
 \*/  
 private static int* hexToInt(*char* c) {  
 *return* Integer.*parseInt*(Character.*toString*(c), 16);  
 }  
  
}

**Output for Bucket Sort:**

Array of numbers and their counts before sorting:

[8.0, 8.0, 2.0, 9.0, 4.0, 2.0, 3.0, 5.0, 4.0, 9.0, 3.0, 7.0, 4.0, 7.0, 2.0]

[1, 2, 1, 1, 1, 2, 1, 1, 2, 2, 2, 1, 3, 2, 3]

Begin Bucket Sort:

Buckets: [[], [], [], [], [], [], [], [], [0.8], []]

Buckets: [[], [], [], [], [], [], [], [], [0.8, 0.8], []]

Buckets: [[], [], [0.2], [], [], [], [], [], [0.8, 0.8], []]

Buckets: [[], [], [0.2], [], [], [], [], [], [0.8, 0.8], [0.9]]

Buckets: [[], [], [0.2], [], [0.4], [], [], [], [0.8, 0.8], [0.9]]

Buckets: [[], [], [0.2, 0.2], [], [0.4], [], [], [], [0.8, 0.8], [0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3], [0.4], [], [], [], [0.8, 0.8], [0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3], [0.4], [0.5], [], [], [0.8, 0.8], [0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3], [0.4, 0.4], [0.5], [], [], [0.8, 0.8], [0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3], [0.4, 0.4], [0.5], [], [], [0.8, 0.8], [0.9, 0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3, 0.3], [0.4, 0.4], [0.5], [], [], [0.8, 0.8], [0.9, 0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3, 0.3], [0.4, 0.4], [0.5], [], [0.7], [0.8, 0.8], [0.9, 0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3, 0.3], [0.4, 0.4, 0.4], [0.5], [], [0.7], [0.8, 0.8], [0.9, 0.9]]

Buckets: [[], [], [0.2, 0.2], [0.3, 0.3], [0.4, 0.4, 0.4], [0.5], [], [0.7, 0.7], [0.8, 0.8], [0.9, 0.9]]

Buckets: [[], [], [0.2, 0.2, 0.2], [0.3, 0.3], [0.4, 0.4, 0.4], [0.5], [], [0.7, 0.7], [0.8, 0.8], [0.9, 0.9]]

End of Bucket Sort

Array of numbers and their counts after sorting:

[2.0, 2.0, 2.0, 3.0, 3.0, 4.0, 4.0, 4.0, 5.0, 7.0, 7.0, 8.0, 8.0, 9.0, 9.0]

[1, 2, 3, 1, 2, 1, 2, 3, 1, 1, 2, 1, 2, 1, 2]

The array of the counts indicates that the Counting Sort is stable.

**Source Code for Bucket Sort:**

*// CS-146 - Ahror Abdulhamidov  
// Class for Bucket Sort  
  
import* java.util.ArrayList;  
  
*public class* BucketSort {  
  
 *// Array of double numbers  
 private static double*[] *array* = {8, 8, 2, 9, 4, 2, 3, 5, 4, 9, 3, 7, 4, 7, 2};  
  
 *// Main method  
 public static void* main(String[] args) {  
  
 *// Prepare the array of doubles* DoubleNumber[] A = DoubleNumber.*makeNumberArray*(*array*);  
 *setCounts*(A);  
 System.*out*.println("Array of numbers and their counts before sorting:");  
 System.*out*.println(DoubleNumber.*getValues*(A));  
 System.*out*.println(DoubleNumber.*getCounts*(A));  
  
 *// Bucket Sort* System.*out*.println("\nBegin Bucket Sort:");  
 *bucketSort*(A, 10);  
 System.*out*.println("End of Bucket Sort");  
  
 *// Show the sorting results and prove the stability* System.*out*.println("\nArray of numbers and their counts after sorting:");  
 System.*out*.println(DoubleNumber.*getValues*(A));  
 System.*out*.println(DoubleNumber.*getCounts*(A));  
 System.*out*.println("The array of the counts indicates that the Counting Sort is stable.");  
  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Manually sets counts for each DoubleNumber in the array  
 \*/  
 private static void* setCounts(DoubleNumber[] A) {  
 A[1].setCount(2);  
 A[5].setCount(2);  
 A[8].setCount(2);  
 A[9].setCount(2);  
 A[10].setCount(2);  
 A[12].setCount(3);  
 A[13].setCount(2);  
 A[14].setCount(3);  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Does Bucket Sort for the specified array by creating n specified buckets  
 \*/  
 private static void* bucketSort(DoubleNumber[] A, *int* n) {  
 DoubleNumber.*normalize*(A, n);  
 ArrayList<DoubleNumber>[] B = *new* ArrayList[n];  
 *for* (*int* i = 0; i < n; i++) {  
 B[i] = *new* ArrayList<>();  
 }  
 *for* (*int* i = 0; i < A.length; i++) {  
 *insert*(B[(*int*) (n \* A[i].getValue())], A[i]);  
 System.*out*.println(*getBuckets*(B));  
 }  
 ArrayList<DoubleNumber> list = *concatenateLists*(B);  
 *for* (*int* i = 0; i < A.length; i++) {  
 A[i] = list.get(i);  
 }  
 DoubleNumber.*unNormalize*(A, n);  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Inserts a DoubleNumber into the bucket in the right position  
 \* Style of inserting the element is similar to Insertion Sort  
 \*/  
 private static void* insert(ArrayList<DoubleNumber> bucket, DoubleNumber number) {  
 *if* (bucket.size() == 0)  
 bucket.add(number);  
 *else* {  
 *int* i = bucket.size();  
 *while* ((bucket.get(i - 1).getValue() > number.getValue()) && (i != 1)) {  
 i--;  
 }  
 bucket.add(i, number);  
 }  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Concatinates ArrayLists of DoubleNumbers into one list and returns it  
 \*/  
 private static* ArrayList<DoubleNumber> concatenateLists(  
 ArrayList<DoubleNumber>[] B) {  
 ArrayList<DoubleNumber> list = *new* ArrayList<>();  
 *for* (ArrayList<DoubleNumber> doubleNumbers : B) {  
 list.addAll(doubleNumbers);  
 }  
 *return* list;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Returns String of buckets of DoubleNumbers  
 \*/  
 private static* String getBuckets(ArrayList<DoubleNumber>[] B) {  
 StringBuilder sb = *new* StringBuilder();  
 sb.append("Buckets: [");  
 sb.append(B[0]);  
 *for* (*int* i = 1; i < B.length; i++) {  
 sb.append(", ").append(B[i]);  
 }  
 sb.append("]");  
 *return* sb.toString();  
 }  
  
}

*// CS-146 - Ahror Abdulhamidov  
// DoubleNumber class - to store double numbers with their count numbers  
  
public class* DoubleNumber {  
  
 *private double* value; *// Double value of the number  
 private int* count; *// Relative number of the number  
  
 public* DoubleNumber(*double* value, *int* count) {  
 *this*.value = value;  
 *this*.count = count;  
 }  
  
 *public double* getValue() {  
 *return* value;  
 }  
  
 *public void* setValue(*double* value) {  
 *this*.value = value;  
 }  
  
 *public int* getCount() {  
 *return* count;  
 }  
  
 *public void* setCount(*int* count) {  
 *this*.count = count;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Creates and returns an array of DoubleNumbers from an array of doubles  
 \*/  
 public static* DoubleNumber[] makeNumberArray(*double*[] array) {  
 DoubleNumber[] numbers = *new* DoubleNumber[array.length];  
 *for* (*int* i = 0; i < numbers.length; i++) {  
 numbers[i] = *new* DoubleNumber(array[i], 1);  
 }  
 *return* numbers;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Creates and returns an array of DoubleNumbers (zeros) of the specified length  
 \*/  
 public static* DoubleNumber[] makeNumberArray(*int* length) {  
 DoubleNumber[] numbers = *new* DoubleNumber[length];  
 *for* (*int* i = 0; i < numbers.length; i++) {  
 numbers[i] = *new* DoubleNumber(0, 1);  
 }  
 *return* numbers;  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Normalizes the array (divides every element by n)  
 \*/  
 public static void* normalize(DoubleNumber[] A, *int* n) {  
 *for* (DoubleNumber number : A) {  
 number.setValue(number.getValue() / n);  
 }  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Unnormalizes the array (mulitplies every element by n)  
 \*/  
 public static void* unNormalize(DoubleNumber[] A, *int* n) {  
 *for* (DoubleNumber number : A) {  
 number.setValue(number.getValue() \* n);  
 }  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Returns String of values for each DoubleNumber in the array  
 \*/  
 public static* String getValues(DoubleNumber[] numbers) {  
 StringBuilder sb = *new* StringBuilder();  
 sb.append("[").append(numbers[0].value);  
 *for* (*int* i = 1; i < numbers.length; i++) {  
 sb.append(", ").append(numbers[i].value);  
 }  
 sb.append("]");  
 *return* sb.toString();  
 }  
  
 */\*\*  
 \** ***@author*** *Ahror Abdulhamidov  
 \* Returns String of counts for each DoubleNumber in the array  
 \*/  
 public static* String getCounts(DoubleNumber[] numbers) {  
 StringBuilder sb = *new* StringBuilder();  
 sb.append("[").append(numbers[0].count);  
 *for* (*int* i = 1; i < numbers.length; i++) {  
 sb.append(", ").append(numbers[i].count);  
 }  
 sb.append("]");  
 *return* sb.toString();  
 }  
  
 *// Returns String value of the DoubleNumber* @Override  
 *public* String toString() {  
 *return* Double.*toString*(value);  
 }  
}

**Setup and Installation**

After unzipping the project, open the command line in the project folder. Then, type these commands (each command is for the specific exercise) to run the program:

For Counting Sort exercise:

java -jar CountingSort.jar

For Radix Sort exercise:

java -jar RadixSort.jar

For Bucket Sort exercise:

java -jar BucketSort.jar

Note: the jar files were created using the Java SDK 11.

**Encountered Problems and Lessons**

The first problem that I encountered was before I even started coding. It was to figure out how to store counts for each number for proving stability of the algorithms. I asked my Professor, Mike Wu, during the class about that, and he suggested to create a class to wrap a number. I created 2 classes: Number and DoubleNumber – classes that contain both value and count of a number.

Another problem that I encountered was when I was doing the radix sort. Everything was working smoothly, especially the counting sort by each hex digit. But it wouldn’t be sorted in the end. I figured out that the stability is the key. According to the book, in the radix sort, you do counting sort for each digit from the 1st to the last, but I changed it so that it is from the last to the 1st, and then it finally worked.

Lastly, I noticed that insertion sort was not needed at all for the array in the bucket sort algorithm. It was a special case array in which every bucket would have numbers with same values, so no sorting was needed. I talked to the Professor about that, and he suggested to insert it similarly to insertion sort, and I did so by making an insert method. Even though the method was not needed for this special case, I still did it.

From this assignment, I learned the how effective the linear sorting methods are. I remember how long was the output for the exercises where I used other comparison based algorithms, and they were much longer than in here. I also thought that stability was something useless, but dealing with the radix sort problem I encountered, I realized that stability may be useful sometimes. Also, I learned how to document my code better, and as you can see, my source code does look more beautiful and methods better explained. If it wasn’t for linear sorting algorithms, I would have probably kept thinking in my head that quicksort or mergesort are the fastest algorithms.