Lab Report

Locating duplicates in an array though various sorting algorithms

By: Matthew Montoya

1 Introduction

In this lab, the program will take a user's integer input of length *n*, that will create an array of said length, and input *m*, which will generate a random integer between 0 and *m-1* for each array position. The program will then and use various sorting methods (including a given method, which must be improved) to organize the array in ascending order, then identify any duplicates that may exist in the array.

2 Proposed Solution & Design Implementation

To achieve the end goal, the program will consist of serval different methods. Some methods within the program may be *helper methods* for methods called by the main method. This is done to isolate, evaluate, and solve each problem. For best practices, the only object or variable manipulation done in the main method will consist of a scanner and integer variables for user input.

2.1— The Main Method

Within the main method, the program is going to ask the user for an input of two integers (*n* and *m*). These integers will be used to define the length of an integer array (size *n*) and the range of available numbers from which the program will select for the value of each element in the array (the range between *0* and *m-1*). After given the input values, the program will then call a return method (*buildArray*) to construct the array using the given parameters. After receiving the constructed array, the program will then call another return method (*copyArray*) to build and return an exact copy of the given array. Prior to calling any methods which may alter the array, this method will be called, as this program is sending arrays using *pass-by-value* and not *pass-by-reference*. After creating a copy of the randomized array, the program will then send that array to its respective sorting method, where the array will be reorganized in ascending order, have each one of its elements printed for the user to see, evaluated for the existence of duplicate integers, and return the sorted array. For each method, the program will return a boolean, which will be printed on screen, informing the user whether duplicate integers were found. Because all the arrays contain the same values, the Boolean values from all the methods should return the same (*true* or *false*).

2.2— The *buildArray* Method

The buildArray method generates an integer array of size *n*, filling each element with a randomized integer between 0 and *m*-1. Recall that integer variables *n* and *m* were passed from the *Main* method. To efficiently build and populate the array, the program will construct the array using two different for-loops, each with the run time of O(*n*). This method will be utilized once during the entire program, and will return an integer array, with the appropriate parameters (as specified by the user in the main method).

2.3— The *arrayCopy* Method

The *arrayCopy* method generates a duplicate copy of the array created in the *buildArray* method. The input for this method will be the size of the array (integer *n*) and a one-dimensional array from the main method. This method will return a one-dimensional array, populated by the same lengths and values as the array it received. This method will be called every time an array needs to be sorted by a sorting method.

2.4— The *hasDuplicates* Method

The *hasDuplicates* method determines is a given method in the lab. In this method, if a given array contains any duplicate integers, the array will return a Boolean value of "true." By default, the Boolean value is set to false. This method will be run once, and is executed so the user can compare the output of this method to the output of the improved version of the code (the *hasDuplicatesRevised* method). Because they search for the same thing, the Boolean values of the both the given code and revised code should be the same. This method will return the Boolean value to the main method.

2.2— The *hasDuplicatesRevised* Method

The *hasDuplicatesRevised* method will take a copy of the randomized array from the main method and, implementing a more efficient version of the code, determine if any duplicates exist in the array. To implement a more efficient version of the code, the method will start with a counter at the array's *position 1*, and compare that element to the one before it, moving across the array in a single pass, and identifying if there are any duplicates. A Boolean variable created in this method will be the returned to the main method, returning true if duplicates are found or returning false if no duplicates exist. The run time of this method will be O(*n*).

2.4— The *sortWithBubbleSort* Method

The *sortWithBubbleSort* method will take in a copy of the randomized array from the main method, sort the array via Bubble Sort, print the elements of the array, and determine if any duplicates exist. This method will be broken down into two parts. The first part of the method will be sorting the method. This will come in the form of a nested loop. Because bubble sort will keep iterating through the array until it is sorted, a Boolean variable (of value *false* [because the array is presumably not sorted]) will be created to trigger the outer loop (a while-loop) to execute. This while loop will keep executing if the array is not sorted. Within the while loop, the Boolean will be changed to *true*. This is done to execute an inner loop, a for-loop. This loop will compare one element to the next, and swap the values of the elements if they're not in ascending order. If the elements need to be exchanged at any point during the pass, the Boolean will be triggered to *false*, causing the outer while-loop to execute again, until the inner for loop passes the entire array without the Boolean triggering to *true*.

The second part of the *sortWithBubbleSort* method will contain the same code from section 2.2 (the *hasDuplicatesRevised* method) and evaluate if the now (presumably sorted) method contains any duplicates. If duplicates are found, a Boolean with the value *true* will be returned to the main method otherwise, a value of *false* will be returned.

2.5— The *sortWithMergeSort* Method

The *sortWithMergeSort* method will take in a copy of the randomized array from the main method, sort the array via Merge Sort, print the elements of the array, and determine if any duplicates exist. Despite its name, method will only break the array into individual elements, then call a helper method (the *mergerTogether* method) to reconstruct the array in ascending order. Within the *sortWithMergeSort* method, the sorting will be done by utilizing recursive calls. With each iteration, the given array will be sorted into two separate arrays (split half way down the middle), and information from the randomized array will be copied into its respective halves (the left half and right half of the array). Each half will then be sent to a helper method (*mergeTogether*), which will organize the elements, and reconstruct the array in ascending order. After the array from the *mergeTogether* method is returned to this method, it will be returned to the *main* method. The run time of this method will be O(log *n*)

2.6— The *mergeTogether* Method

The *mergeTogether* method will take in an array from the *sortWithMergeSort* method, compare each element of the given array, and reconstruct the array one merge at a time, in ascending order. This is going to be done using three counters, one for the left half of the array, another for the right, and a general counter, which for the size of the array (known as *n*). If both counters from each half of the array are less than the size of the array, there will be a loop which executes, comparing which element at the given counter's position is less than the other. The element which is less will be placed into the general counter's position in the array, and the general counter will increment, moving onto the next element. When all the elements have been sorted, the array will be return to the *sortWithMergeSort* method.

2.7— The *printMergeSort* Method

The *printWithMerge* method will take in *sortWithMergeSort*'s array (which was returned to the main method) and determine if there are any duplicate integers within the (now) sorted array. This non-return method will be done using a Boolean (set to false) and a for-loop, which will scan each element and compare it to the next one. If a pair is found to be true, the Boolean will be set to true, and *break* from the loop. There will be no need to continue scanning, as a match has already been found. The program will then print *true* if there are any duplicates (*false* otherwise), followed by each element in the sorted array. The run time of this method will be O(*n*).

2.8— The *singlePassDuplicates* Method

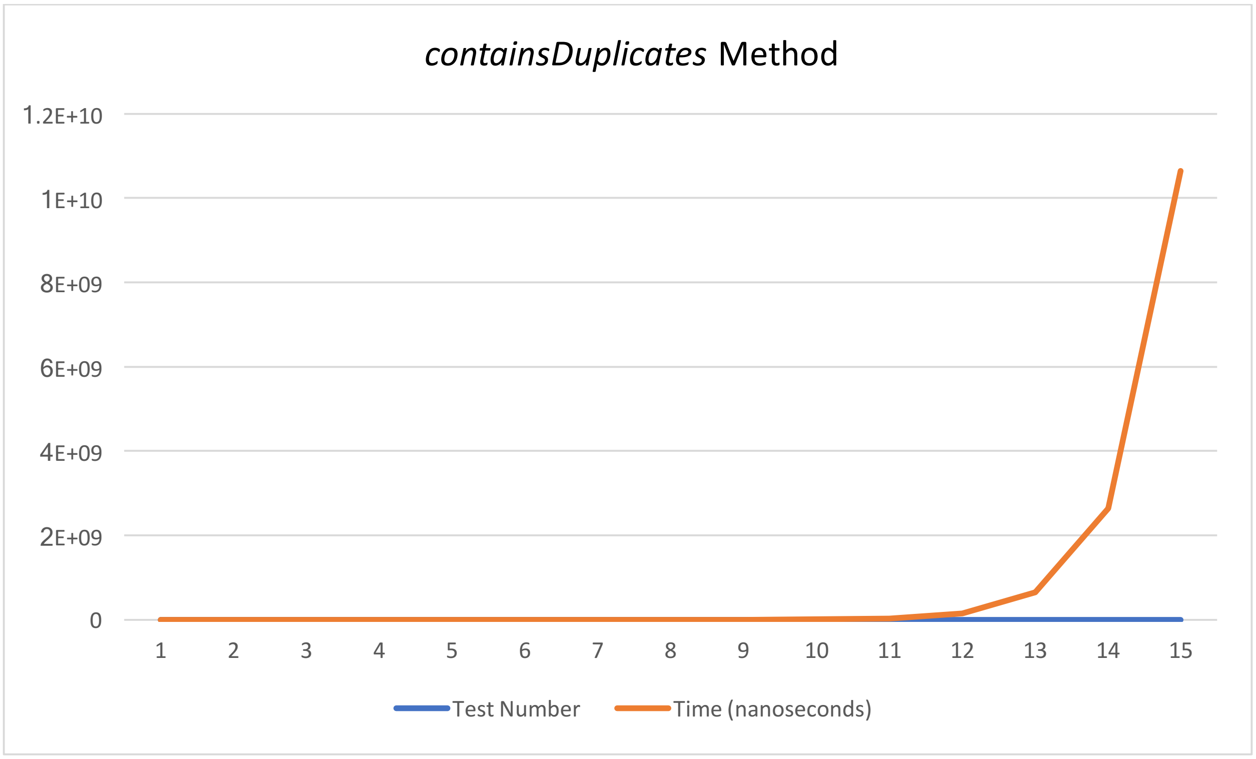
The final method of the program will be the *singlePassDuplicates* method. This method will take a copy of the randomized array and determine if any duplicate integer values exist, by only passing through the array once. The program will create a Boolean variable (set to *false*) and a Boolean array using the *length* value of the integer array. By default, all the elements will be set to false. This will be done using a for-loop. Using a separate for-loop (as well as a counter), the program will scan through the integer array and at every stop, check if that integer in the respective position in the Boolean array is true or false. If it is false, that position will be set to true (e.g. if the integer array at position *i* equals 1 and the Boolean array at position 1 is *false*, it will be set to *true*). If the position is already set to *true*, this means that the program has already seen that number in the array, and thus, will set the Boolean variable to *true*. The program will print out each element in the Boolean array and return the Boolean variable to the main method. The run time of this method will be O(*n*).

3 Experimental Results

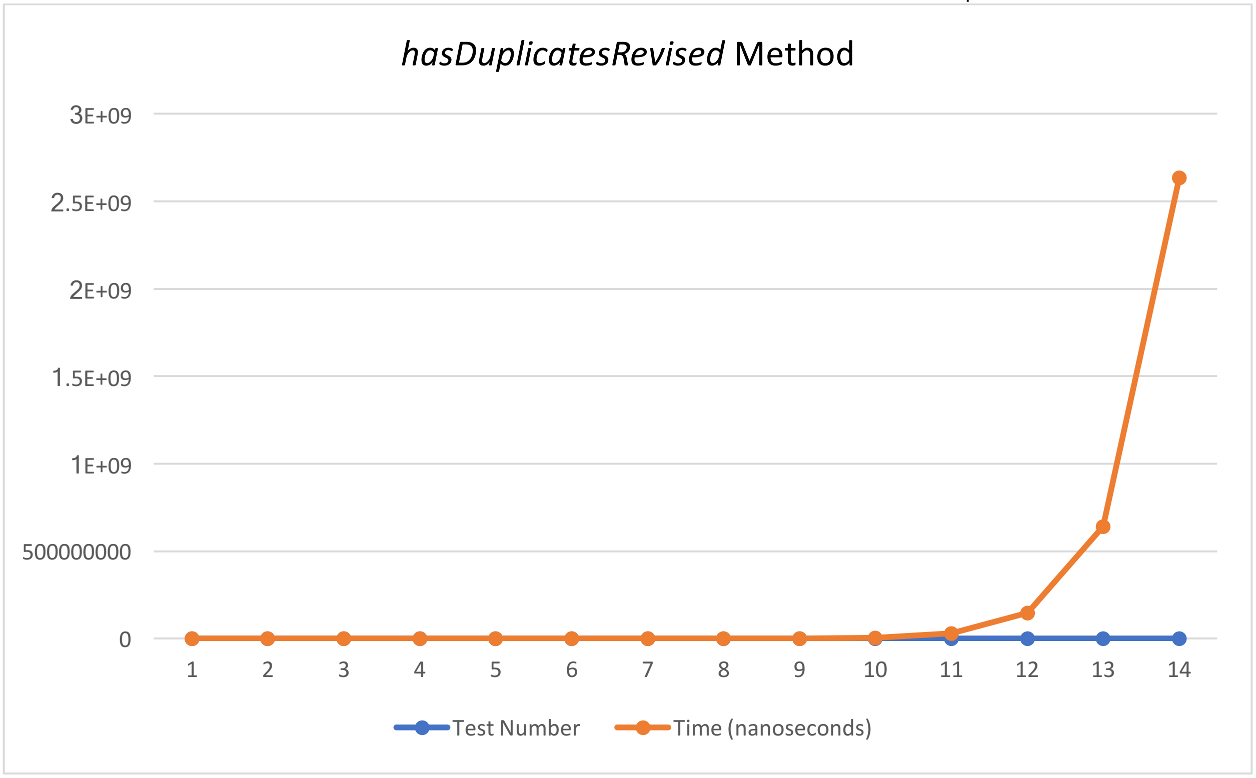
3.1— O(n) runtimes

Disclaimer: The computer used to run the program was running a pre-release version of the macOS operating system which, during programming and execution in two different compilers, caused lags and system hangs not present prior to the software's installation.

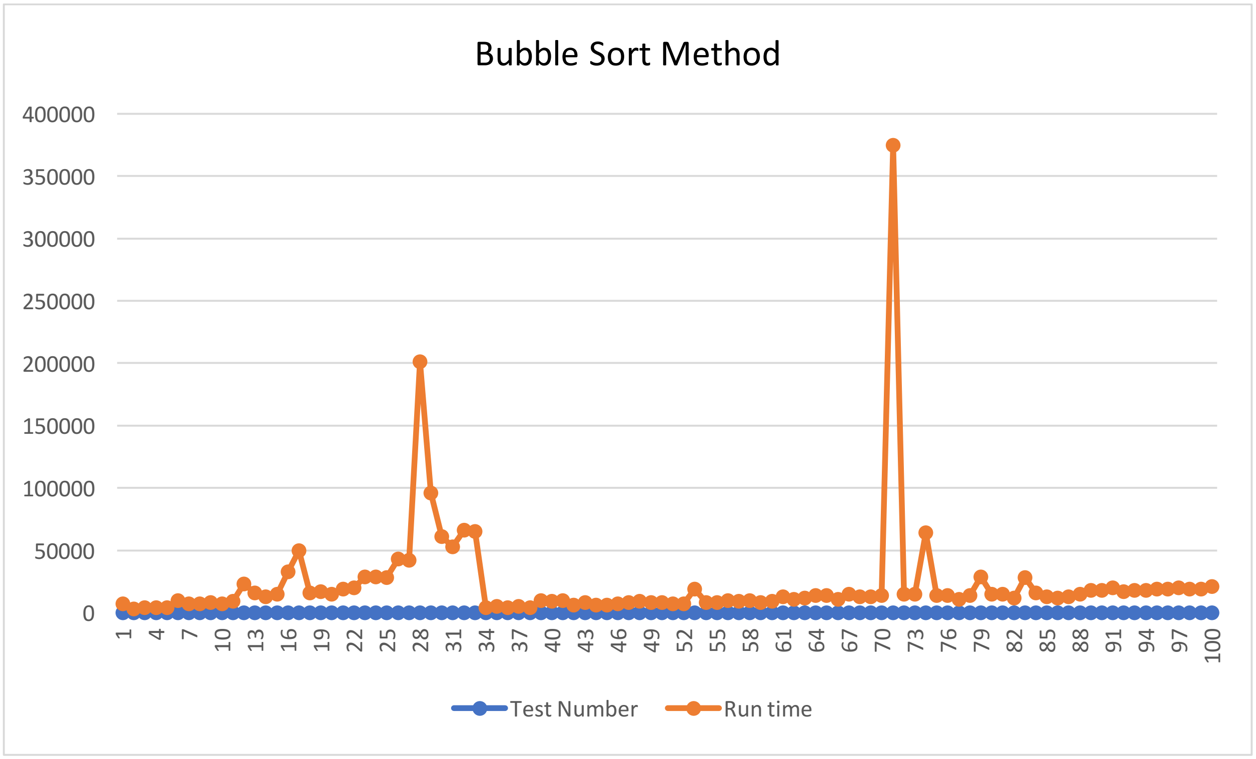
The big-oh runtimes for each of the required methods are shown below. The method names are listed at the top, followed by a brief description below the graph and/or any discrepancies at the bottom. Note: All times are measured in nanoseconds.



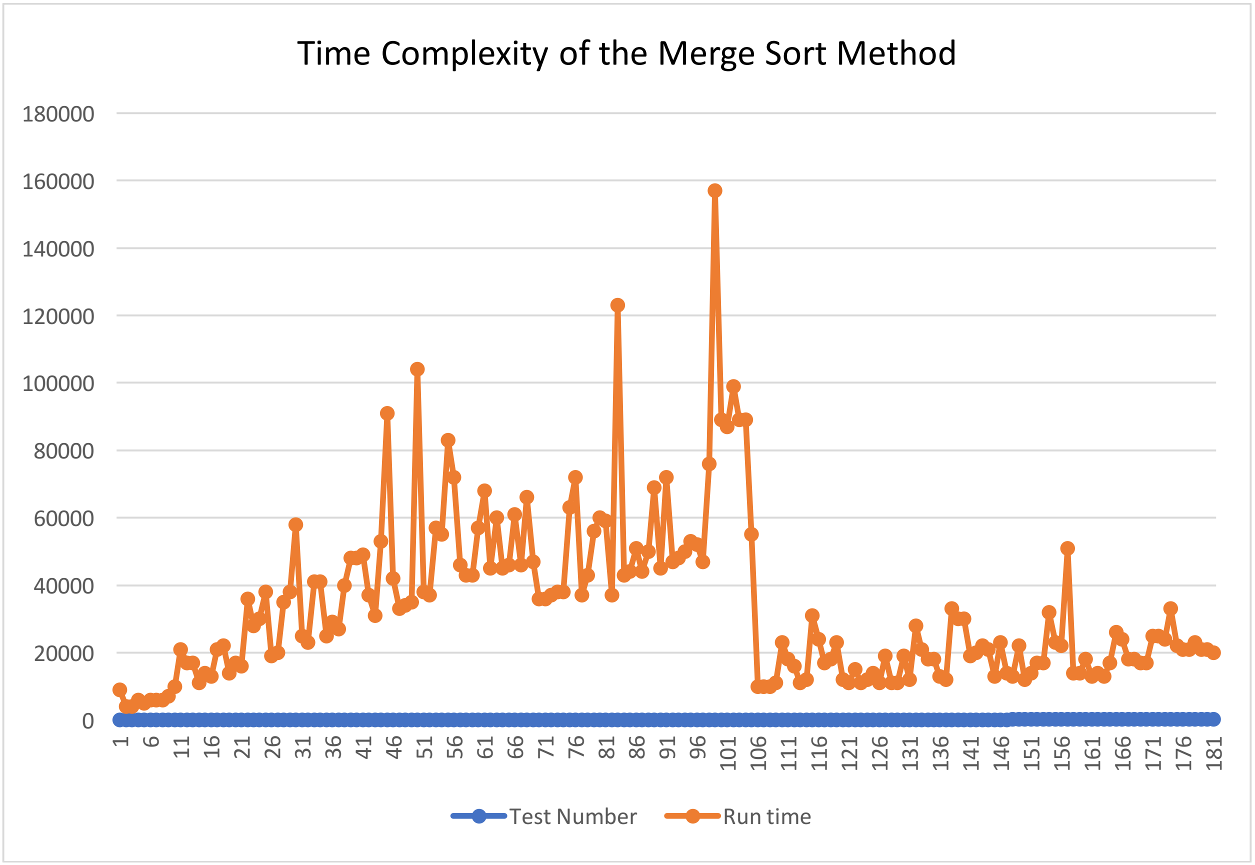
The *containsDuplicates* method, as the size of the array increases. Note: With the growing amount of data, the compiler could not process the 16th test.



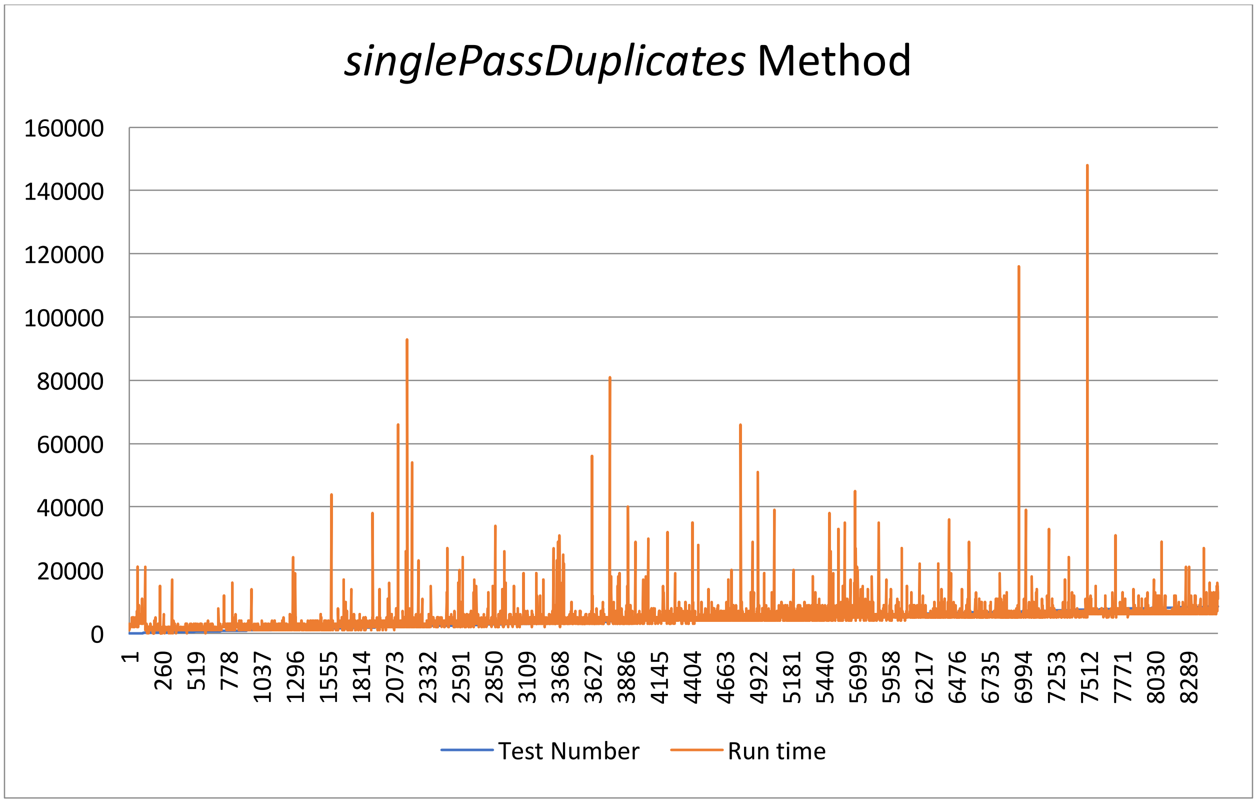
The *hasDuplicatesRevised* method, which looks to eliminate the redundancy in the given method, as the size of the array increases. Note: With the growing influx of data, the compiler could not process the 15th test. As of the writing of this report, the data is concerning, as the expected runtime of this method was O(n).



The results from running the bubble sort method 100 times. Apart from a few anomalies, notably test 28 and test 71, the data shown is generally indicative of what was expected to be produced, with the time complexity being linear (O(*n*)).



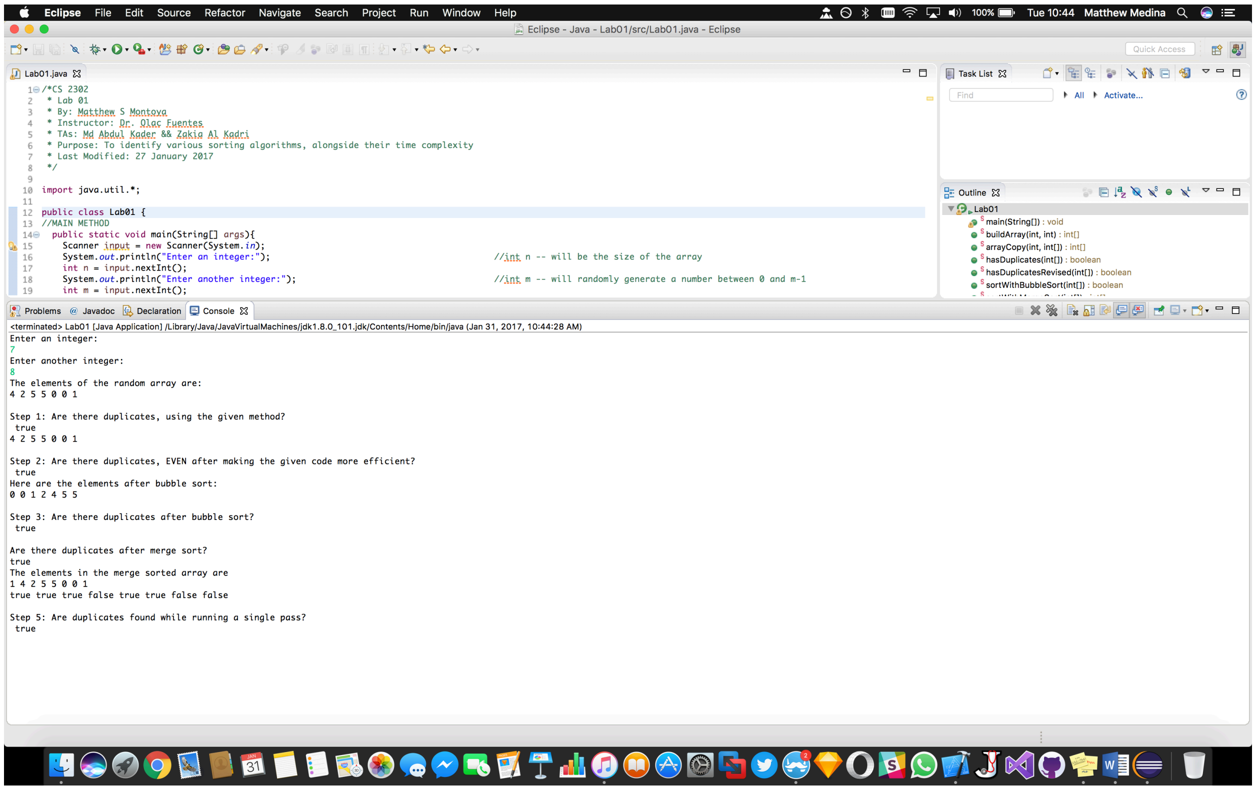
The results from running the merge sort method, ran 181 times. Like the previous method, there are a few anomalies, notably between tests 96 and 106; however, apart from those (concerning) areas, the time complexity is what was expected, that is O(log *n*).



The results from running the singlePassDuplicates method 8522 times. Like the previous graphs, there are a few anomalies, however, nothing which throws the program from its expect time complexity O(n). Because the time was measured in nanoseconds, the data on the graph is difficult to distinguish. For examples tests 1 through 1000 ran at approximately 1000 nanoseconds, and tests in-between 4000 and 5000 ran at approximately 4000 nanoseconds, showing a linear growth.

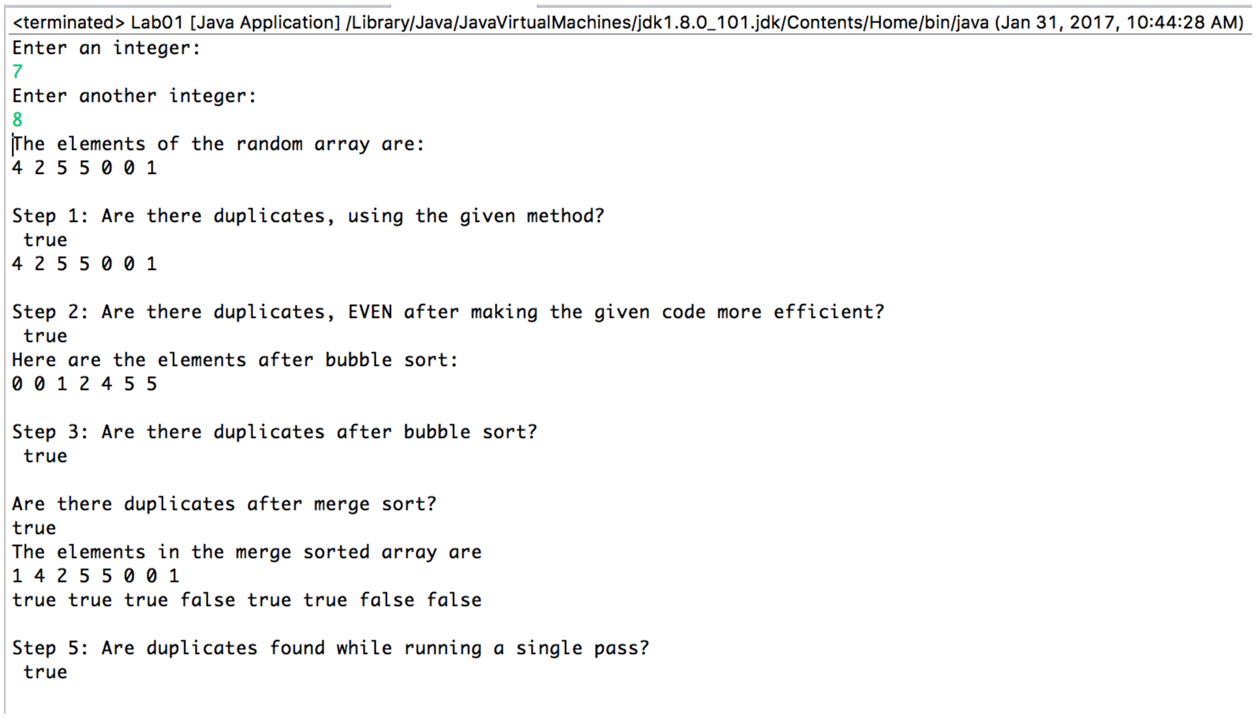
3.2— Program Output

The output of the code (found in the appendix) is shown below. The two images are from the same test. Note: The first image includes the beginning of the code (which can also be found in the *Appendix* section), as proof the executed code is not the work of others. The second image is an enhanced version of the program output.


Eclipse - Java - Lab01/src/Lab01.java - Eclipse
Lab01.java
Task List
Outline
Lab01
main(String[]) : void
buildArray(int, int) : int[]
arrayCopy(int, int[]) : int[]
hasDuplicates(int[]) : boolean
hasDuplicatesRevised(int[]) : boolean
sortWithBubbleSort(int[]) : boolean
Problems
Javadoc
Declaration
Console

```java
/*CS 2302
 * Lab 01
 * By: Matthew S Montoya
 * Instructor: Dr. Olac Fuentes
 * TAs: Md Abdul Kader && Zakia Al Kadri
 * Purpose: To identify various sorting algorithms, alongside their time complexity
 * Last Modified: 27 January 2017
 */

import java.util.*;

public class Lab01 {
//MAIN METHOD
  public static void main(String[] args){
    Scanner input = new Scanner(System.in);
    System.out.println("Enter an integer:");                        //int n -- will be the size of the array
    int n = input.nextInt();
    System.out.println("Enter another integer:");                   //int m -- will randomly generate a number between 0 and m-1
    int m = input.nextInt();
```

```
<terminated> Lab01 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (Jan 31, 2017, 10:44:28 AM)
Enter an integer:
7
Enter another integer:
8
The elements of the random array are:
4 2 5 5 0 0 1

Step 1: Are there duplicates, using the given method?
 true
4 2 5 5 0 0 1

Step 2: Are there duplicates, EVEN after making the given code more efficient?
 true
Here are the elements after bubble sort:
0 0 1 2 4 5 5

Step 3: Are there duplicates after bubble sort?
 true

Are there duplicates after merge sort?
true
The elements in the merge sorted array are
1 4 2 5 5 0 0 1
true true true false true true false false

Step 5: Are duplicates found while running a single pass?
 true
```



```
<terminated> Lab01 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (Jan 31, 2017, 10:44:28 AM)
Enter an integer:
7
Enter another integer:
8
The elements of the random array are:
4 2 5 5 0 0 1

Step 1: Are there duplicates, using the given method?
 true
4 2 5 5 0 0 1

Step 2: Are there duplicates, EVEN after making the given code more efficient?
 true
Here are the elements after bubble sort:
0 0 1 2 4 5 5

Step 3: Are there duplicates after bubble sort?
 true

Are there duplicates after merge sort?
true
The elements in the merge sorted array are
1 4 2 5 5 0 0 1
true true true false true true false false

Step 5: Are duplicates found while running a single pass?
 true
```

It was noted that the data from the merge sort method did not merge properly. The images shown are from the exact copy of the program submitted for grading. Despite this error, the program was still able to determine if duplicates existed.

4 Conclusions

For the most part, the expectations and results of the data were to be expected. While testing the lab, it became apparent that regardless of the sorting algorithm used, there were a few anomalies. As these were not occurring with the same test number every time, it leads me to believe this may be normal for the program, or normal in this case, given the use of two separate IDEs on a pre-release operating system. Watching the program autonomously increase the size of data, I learned the impact time complexity has program run times, especially when working with larger amounts of data. Historically I have been accustomed to working with smaller data sets (less than 20 elements); however, when dealing with data sets of various lengths, it's important to know which algorithm is the most efficient to analyze and possibly, reorganize the data. However, there are some instances where the programmer has the option of choosing an algorithm of their choice, because multiple algorithms may have the most efficient same time complexity.

Additionally, I learned the importance of coming up with different naming conventions for both variables and methods, as having similar names, though descriptive, can confuse the program, and may cause the user to suffer the consequences.

Lastly I learned that, unlike my transition from CS 1 to CS 2, my transition from CS 2 to CS 3 has been smoother in terms of programming skills. Although I used a plethora of visuals to help me design both my methods and my code, I remembered how to program in Java, unlike

5 Appendix

5.1— Documentation & Disclosure

- Methods are documented using the following format—
```
//METHOD NUMBER: BRIEF MENTION OF METHOD'S PURPOSE
public static void methodName(parameters){
}
```
- Non-obvious code is documented using the following format—
```
for(int i=0; i<x.length-1; i++){          //summarizes the function of the loop
        x[i] = i;          //may include brief logical reasoning
}
```
- Disclosure: Though keeping the original digital format of the code (indention lengths, spaces, etc.) the use of Microsoft Word has reformatted the code in the document to be of different space and tab lengths. The font size has been reduced to help alleviate some visual misrepresentations.

5.2— Student Code

```
/*CS 2302
 * Lab 01
 * By: Matthew S Montoya
 * Instructor: Dr. Olac Fuentes
 * TAs: Md Abdul Kader && Zakia Al Kadri
 * Purpose: To identify various sorting algorithms, alongside their time complexity
 * Last Modified: 27 January 2017
 */

import java.util.*;

public class Lab01 {
//MAIN METHOD
  public static void main(String[] args){
    Scanner input = new Scanner(System.in);
    System.out.println("Enter an integer:");          //int n -- will be the size of the array
    int n = input.nextInt();
    System.out.println("Enter another integer:");   //int m -- will randomly generate a number
between 0 and m-1
    int m = input.nextInt();
    int [] randomizedArray = buildArray(n, m);    //Calls method 1 to build the array

    int[] arrayForGivenMethod = arrayCopy(n, randomizedArray);
    System.out.println();
    boolean containsDuplicates = hasDuplicates(arrayForGivenMethod);
    System.out.println("Step 1: Are there duplicates, using the given method?\n
"+containsDuplicates);

    int[] arrayForEfficiency = arrayCopy(n, randomizedArray);
    boolean duplicateRevised = hasDuplicatesRevised(arrayForEfficiency);          //Calls method
3 for more effient way of doing it
    System.out.println("\n\nStep 2: Are there duplicates, EVEN after making the given code more
efficient?\n "+duplicateRevised);

    int[] bubbleSortArray = arrayCopy(n, randomizedArray);
    boolean duplicatesInBubbleSort = sortWithBubbleSort(bubbleSortArray);       //sends array to
be sorted via Bubble Sort and identifies if duplicate elements exist
    System.out.println("\n\nStep 3: Are there duplicates after bubble sort?\n
"+duplicatesInBubbleSort);

    int[] mergeSortArray = arrayCopy(n, randomizedArray);
    int[] mergedArray = sortWithMergeSort(mergeSortArray);      //sort with Merge Sort; do
duplicates exist?
    printMergeSort(mergedArray);

    int[] unsortedArray = arrayCopy(n, randomizedArray);
    boolean duplicatesInUnsorted = singlePassDuplicates(unsortedArray, m);
    System.out.println("\n\nStep 5: Are duplicates found while running a single pass?\n
"+duplicatesInUnsorted);
  }

//METHOD A: RECEIVES 2 INTERGERS N AND M. BUILDS & RETURNS AN ARRAY OF
OF LENGTH N AND CONTAINS RANDOM INTEGERS IN THE 0 ---> M-1 RANGE
  public static int[] buildArray(int n, int m){
    int[] randomizedArray = new int[n];                  //builds an array of size n
    for(int i=0; i<randomizedArray.length; i++)
      randomizedArray[i] = ((int)(Math.random()*(m-1)));    //assigns a value of between (0, m-1)
to the element

    System.out.println("The elements of the random array are:");

    for(int i=0; i<randomizedArray.length; i++){
      randomizedArray[i] = randomizedArray[i];    //builds a series of elements into an array
      System.out.print(randomizedArray[i]+" ");
    }
    System.out.println();
    return randomizedArray;
  }

//METHOD B: ARRAYCOPY
  public static int[] arrayCopy(int n, int[] randomizedArray){
    int[] randomArrayCopy = new int [randomizedArray.length];
    for(int i=0; i<randomArrayCopy.length; i++){  //create an exact copy of the array to aid with
calling fresh, unsorted, array copies
      randomArrayCopy[i] = randomizedArray[i];
    }
    return randomArrayCopy;
  }

//METHOD 1: COMPARE EVERY ELEMENT IN THE ARRAY WITH EVERY OTHER
ELEMENT USING A GIVEN NESTED LOOP
  public static boolean hasDuplicates(int[] randomizedArray){
    if(randomizedArray.length <= 1)                  //base case: only one element
      return false;

    boolean duplicates = false;       //given method to analyze
    for(int i=0;i<randomizedArray.length;i++){
      for(int j=0;j<randomizedArray.length;j++){
        if(randomizedArray[i]==randomizedArray[j] && i!=j)
          duplicates = true;
      }
    }
    return duplicates;
  }

//METHOD 2: IMPROVE THE CODE
  public static boolean hasDuplicatesRevised(int[] randomizedArray){
    if(randomizedArray.length <= 1)                      //base case: not enough duplicates
      return false;

    boolean duplicates = false;
    for(int i=1;i<randomizedArray.length; i++){
      if(randomizedArray[i] == randomizedArray[i-1]){               //start at position 1 and compare
to the previous position
        duplicates = true;                        //traverse one-by-one for duplicates
        break;                               //break if a match is found
      }
    }
    for(int i=0; i<randomizedArray.length; i++){
      System.out.print(randomizedArray[i]+" ");
    }
    return duplicates;
  }

//METHOD 3: BUBBLE SORT
  public static boolean sortWithBubbleSort(int[] randomizedArray){
    if(randomizedArray.length<=1)
      return true;                              //base case: sorted by default

    boolean isSorted = false;                    //assume nothing is sorted
    int tempElement;                             //to hold a temp value if we need to swap elements

    while(isSorted == false){
      isSorted = true;                           //resets the trigger
      for(int i=0; i<randomizedArray.length-1; i++){
        if(randomizedArray[i] > randomizedArray[i+1]){
          tempElement=randomizedArray[i];   //use the temp element as a 3-way handshake to move
elements around
          randomizedArray[i]=randomizedArray[i+1];
          randomizedArray[i+1] = tempElement;
          isSorted = false;                      //triggers at any point in the array
        }
      }
    }
    boolean duplicatesExist = false;         //assume no duplicates exist
    int position=0;
    while(position<randomizedArray.length-1){
      if(randomizedArray[position] == randomizedArray[position+1]){
        duplicatesExist=true;                    //triggers at any point in traversal
      }
      position++;
    }

    System.out.println("Here are the elements after bubble sort:");
    for(int i=0; i<randomizedArray.length; i++){ //prove elements have been sorted
      System.out.print(randomizedArray[i]+" ");
    }
    return duplicatesExist;
  }


//METHOD 4a: MERGESORT (PRIMARY METHOD: SORTS INTS)
  public static int[] sortWithMergeSort(int[] randomizedArray){

    if(randomizedArray.length <= 1)
      return randomizedArray;
      int[] leftHalf = new int [randomizedArray.length/2];
      int[] rightHalf = new int [randomizedArray.length-leftHalf.length];      //Mahdokht: "Explore
'arraycopy' via java.lang.system"

      System.arraycopy(randomizedArray, 0, leftHalf, 0, leftHalf.length);                  //copies
leftHalf.length-elements of arrayCopy to the new leftHalf array
      System.arraycopy(randomizedArray, leftHalf.length, rightHalf, 0, rightHalf.length);    //copies
rightHalf.length-elements of arrayCopy to the leftHalf array

      sortWithMergeSort(leftHalf);
      sortWithMergeSort(rightHalf);

    int[] merged = mergeTogether(leftHalf, rightHalf, randomizedArray);      //call the helper
method

    return merged;
  }

//METHOD 4b: MERGE EVERYTHING BACK TOGETHER
  public static int[] mergeTogether(int[] leftHalf, int[] rightHalf, int[] randomizedArray){

    int leftCounter=0;
    int rightCounter=0;     //counters for left and right side
    int i=0;    //general counter

    while(leftCounter<leftHalf.length && rightCounter<rightHalf.length){ //analyze the smaller
arrays to construct the larger array
      if(leftHalf[leftCounter] < rightHalf[rightCounter]){
        randomizedArray[i] = leftHalf[leftCounter];
        leftCounter++;
      }
      else{
        randomizedArray[i] = rightHalf[rightCounter];
        rightCounter++;
      }
      i++;        //increment the position of the larger array as you move each element into place
    }

    return randomizedArray;
  }

//METHOD 4c: PRINT MERGE SORT ELEMENTS & RETURN IF DUPLICATES EXIST
  public static void printMergeSort(int[] merged){
    if (merged.length <= 1)
      System.out.println("\nStep 4: The elements are sorted; there is <= 1 elements.");

    //Do duplicates exist?
    boolean mergedDuplicates = false;
    for(int i=1; i<merged.length; i++){ //Assuming duplicates are in order, start at 1 and compare
backwards
      if(merged[i] == merged[i-1]){
        mergedDuplicates = true;
        break;
      }
    }

    System.out.println("\nAre there duplicates after merge sort?");
    if(mergedDuplicates == true)          //Prove there are duplicates after merge sort
      System.out.println(mergedDuplicates);

    System.out.println("The elements in the merge sorted array are ");
    for(int i=6; i<merged.length; i++)         //Prove the elements are sorted
      System.out.print(merged[i]+" ");
  }


//METHOD 5: DETERMINE IF THERE ARE DUPLICATES BY PERFORMING A SINGLE
PASS
  public static boolean singlePassDuplicates(int[] randomizedArray, int m){
    if(randomizedArray.length <= 1)
      return true;

    boolean duplicates = false;                //assume no duplicates exist
    boolean[] boolArray = new boolean[m];       //create boolean array of size m
    for(int c=0; c<boolArray.length; c++){
      boolArray[c]=false;                      //set all the booleans to false (by default)
    }
    int c=0;

    for(int i=0; i<randomizedArray.length; i++){
      c=randomizedArray[i];
      System.out.print(c+ " ");

      if(boolArray[c] == false)            //Set the numbers (position [c+1]) to true as you scan them
        boolArray[c] = true;
      else{
        duplicates = true;                 //If you come across a number again, it will recognize it as a
duplicate and trigger
      }
    }

    System.out.println();
    for(int d=0; d<boolArray.length; d++){  //Prove that the boolean array[x] is triggered correctly
      System.out.print(boolArray[d]+" ");
    }
    return duplicates;
  }
}
```