Gabriel Brehm

CS361

April 21, 2019

Lab 1 Report

Part 1

For this project, it was necessary to read in data from a text file. The format of the file was known; a single number on each line with white space at the end, C# makes it very easy to read in such text files through the use of the System.IO library, so this method was straight forward.

```
public static string[] ReadFile(string pathToFile)
{
        string[] listOfStrings = File.ReadAllLines(pathToFile);

        return listOfStrings;
}
```

It simply reads each line into a single element in a list of stings. In the end, we need integers to sort, so the entire list needed to be converted into a list of equivalent integers.

```
public static int[] ConvertToInts(string[] listOfStrings)
{
        int[] listOfInts = new int[listOfStrings.Length];

        for (int i = 0; i < listOfStrings.Length; i++)
        {
                listOfInts[i] = int.Parse(listOfStrings[i]);
        }
        return listOfInts;
}
```

This method does just that. It walks through the list an applies the int.Parse method to each element, which turns the integer into its 32-bit signed integer equivalent.

As a verification that these methods functioned as desired, the following method was written.

```
public static long SumOfInts(int[] listOfInts)
{
        long sum = 0;

        for (int i = 0; i < listOfInts.Length; i++)
        {
                sum = sum + listOfInts[i];
        }
        return sum;
        }
```

We were only given that the sum of all the numbers was 49999995000000, thus, this function confirmed that the data was being appropriately read in.

Part 2

For part two, we needed to write the merge-sort method. Essentially. this method works by splitting the list up into smaller and smaller lists until each list contains just one element. They are then merged back together in the right order. Before we can explain the merge sort method, we need to define and explain the two helper methods. The first of which is getMiddleIndex. Note: the inline comments are omitted from this report because it is unnecessary to verbally explain what is happening twice.

```
        public static int getMiddleIndex(int startIndex, int endIndex)
        {
                double temp = (startIndex + endIndex) / 2;

                int middleIndex = Convert.ToInt32(Math.Floor(temp));

                return middleIndex;
        }
```

This method takes in the start and end indices and returns the index that is almost exactly between them. First, we get the average of the two values as a double, then we cast that to an int and floor it, so our number can still be used as an index. The other helper method is the merge method, which is really the meat and potatoes of merge sort.

```
        public static void Merge(int[] list, int startIndex, int middleIndex, int endIndex)
        {
                int leftHalfLength = middleIndex - startIndex + 1;
                int rightHalfLength = endIndex - middleIndex
                int[] leftList = new int[leftHalfLength + 1];
                int[] rightList = new int[rightHalfLength + 1];
                for(int i = 0; i < leftHalfLength; i++)
                {
                        leftList[i] = list[startIndex + i];
                }for(int i = 0; i < rightHalfLength; i++)
                {
                        rightList[i] = list[middleIndex + 1 + i];
                }
                leftList[leftHalfLength] = Int32.MaxValue;
                rightList[rightHalfLength] = Int32.MaxValue;
                int leftIndex = 0;
                int rightIndex = 0;
        for(int k = startIndex; k < endIndex + 1; k++)
        {
                if(leftList[leftIndex] <= rightList[rightIndex])
                {
                        list[k] = leftList[leftIndex];
                        leftIndex++;
                }
                else
                {
                        list[k] = rightList[rightIndex];
                        rightIndex++;
                }
        }
}
```

This method takes in the list, along with the first, last, and middle indices. Then we calculate the length of each sub-list, and create two new lists on the heap of each length plus one. The left and right portions of the original list are copied into the two new sub-lists. Then, in those extra two spots we added, we insert the value int.MaxValue, which is equal to $2^{31} - 1$. Then an index is created for each sub-list, and we iterate through the two lists comparing their values at respective indices. If the left value is less than or equal to the right one, then it is added back into the original list, and the index is incremented. Otherwise, the right value is added back to the list, and the right index is incremented. Now we can get to merge-sort.

```
public static void MergeSort(int[] list, int startIndex, int endIndex)
{
        if (startIndex < endIndex)
        {
                int middleIndex = GenFunctions.getMiddleIndex(startIndex, endIndex);
                MergeSort(list, startIndex, middleIndex);
                MergeSort(list, middleIndex + 1, endIndex);
                Merge(list, startIndex, middleIndex, endIndex);
        }
}
```

This recursive method takes in a list of integers, the starting index of that list, and the last index of that list. It then checks if the starting index is smaller than the ending index. If it is, then it finds the middle index using the getMiddleIndex method. Then it recursively calls itself on the left and right halves of the list. Finally, it merges them back together.

These methods were straight forward to write, and were based almost entirely on the pseudocode form the textbook.

Part 3

For this step we basically did the same thing as part 2, except not we used quicksort instead of merge-sort. Just as merge is the core of merge-sort, partition is the core of quicksort.

```
public static int Partition(int[] list, int startIndex, int endIndex)
{
        int pivot = list[endIndex];
        int i = startIndex - 1;
        for (int j = startIndex; j < endIndex; j++)
        {
                if (list[j] <= pivot)
                {
                        i++;
                        int temp1 = list[j];
                        list[j] = list[i];
                        list[i] = temp1;
                }
        }
        int temp2 = list[endIndex];
        list[endIndex] = list[i + 1];
        list[i + 1] = temp2;
        return i + 1;
}
```

As the name implies, partition breaks the list up into two smaller sub-lists, but unlike merge, these are not stored in additional lists. Partition is done in place. It takes in the list, the start index, and the end index as parameters, sets the pivot equal to the last element in the list, and sets and indexer i equal to the start index minus 1. We then enter a for loop where we check if each value is less than or equal to the pivot. If it is, then we increment i, and swap the element at i with the element at j. This is basically separating the list into two parts; one part contains everything less than or equal to the pivot, and the other part is greater than it. After the loop, we switch the first element of the larger partition with the pivot.

It is a good time to note that the above pivot method does not meet the requirements in the lab. We were asked to do something like the following method.

```
public static int AveragePartition(int[] list, int startIndex, int endIndex)
{
        int pivotIndex = (list[startIndex] / 3) + (list[endIndex] / 3) +
        (list[GenFunctions.getMiddleIndex(startIndex, endIndex)] / 3);
        pivotIndex = pivotIndex % (endIndex - startIndex);
        int temp = list[endIndex];
        list[endIndex] = list[pivotIndex];
        list[pivotIndex] = temp;
        return Partition(list, startIndex, endIndex);
    }
```

Instead of using whatever value was at the end of the list as the pivot, this method uses the average of the values at the beginning, middle, and end of the list. We first set the pivot to be the average, but to avoid potentially getting a value larger than $2^{31} - 1$, we divided each term by 3 individually. Then, to guarantee that the value is within the bounds of the section of the list we are partitioning, we mod it by the difference between the end index and the start index. Once that is finished, we swap the last value in the list with the value at the index we just found, and then call partition, defined above.

The only issue is that this method does not work. I wrote an almost identical method using the random index approach, and it worked find for some reason. It is given here.

```
    public static int RandomPartition(int[] list, int startIndex, int endIndex)
    {
        Random rand = new Random();
        int randomIndex = rand.Next(startIndex, endIndex);
        int temp = list[endIndex];
        list[endIndex] = list[randomIndex];
        list[randomIndex] = temp;
        return Partition(list, startIndex, endIndex);
    }
```

It was consistently slower than the original partition method. This can be seen in the second graph at the end of this report. Hours upon hours were spent trying to fix this method, but to no avail. As a result, the data collected for this project is based on the original partition method.

Part 4

The next part of the lab was about making sure the sorting methods defined above sorted the integers. So, we needed a method to check if a list was sorted. Two constraints were that the method could only take in one parameter and it had to be recursive. Before we can explain how IsSorted works, we need to look at two helper methods.

```
    public static int[] getFirstHalf(int[] list, int startIndex, int endIndex)
    {
        int middleIndex = getMiddleIndex(startIndex, endIndex);
        int[] result = new int[middleIndex];
        for (int i = 0; i < middleIndex; i++)
        {
            result[i] = list[i];
        }
```

```
		return result;
	}
```

This first method returns the first half of a list of integers. It takes in a list, and the starting and ending indices of that list as parameters. It calculates the middle, using the getMiddleIndex method defined earlier, and then creates a new list on the heap with a length of that index. It has half the length of the original list. It then copies the values of the first half of the original list into the new list, and returns it. The second method follows.

```
	public static int[] getSecondHalf(int[] list, int startIndex, int endIndex)
	{
		int middleIndex = getMiddleIndex(startIndex, endIndex);
		int[] result = new int[endIndex - middleIndex];
		int j = 0;
		for (int i = middleIndex; i < endIndex; i++)
		{
			result[j] = list[i];
			j++;
		}
		return result;
	}
```

In a similar fashion to the previous method, this one gets the second half of the list. It gets the middle index in much the same way, but this time it makes a new list on the heap with a length equal to the end index minus the middle index. We then create a counter j, and set it equal to zero. After that, we enter a for loop where we set the jth value of the new list to the ith value of the original list. Since i started at the middle index of the original list, and j started at zero, we successfully copy the second half of the original list into the new list.

Now, why did we do that? Well, there is a somewhat clever way to determine if a list is sorted without walking through the entire thing and checking if each value is greater than or equal to the previous one. The IsSorted method is given below.

```
	public static bool IsSorted(int[] list)
	{
		if (list.Length == 1)
		{
			return true;
		}
		else if (IsSorted(getFirstHalf(list, 0, list.Length)) == true)
		{
			if (IsSorted(getSecondHalf(list, 0, list.Length)) == true)
			{
```

```
                        if (getSecondHalf(list, 0, list.Length)[0] >= getFirstHalf(list, 0,
                list.Length)[getMiddleIndex(0, list.Length) - 1])
                        {
                                return true;
                        }
                        else
                        {
                                return false;
                        }
                }
                else
                {
                        return false;
                }
        }
        else
        {
        return false;
        }
}
```

This method takes in only one parameter, which is the list, and it is a Boolean method, so it returns either true or false. First, we establish our base case, and that is if the list is only one value, it is sorted. Otherwise, we recursively call IsSorted to see if the first half of the list is sorted. If it is, then we check if the second half of the list is sorted, also recursively. If that is also true, then we check if the first value of the second half is greater than or equal to the last value of the first half. Finally, if that is also true, then the list is indeed sorted, and true is returned. Otherwise, false is returned.

Part 5

For part five, we wanted to time how long it takes to perform these sorting algorithms. First, we needed data of varying sizes, so a method was written to create data files of sizes 1000, 5000, 10000, et cetera. It is good to note that the machine this was written on was not capable of sorting a list of 100000000 integers, so additional data points were added for precision. That code, however, is omitted because it is long and is not a requirement for this lab.

C# does not normally measure time in nanoseconds, so the following method was used.

```
public static long NanoTime()
{
        long nano = 10000L * Stopwatch.GetTimestamp();
        nano = nano / TimeSpan.TicksPerMillisecond;
        nano = nano * 100L;
        return nano;
}
```

This method creates a long and sets it equal to 10000, casted as a long, multiplied by the current time in milliseconds. It is then divided by the number of ticks per millisecond, which in this case is 1000. The result is then multiplied by 100, again, casted as a long, and is then returned. This method is used in the timing methods as follows.

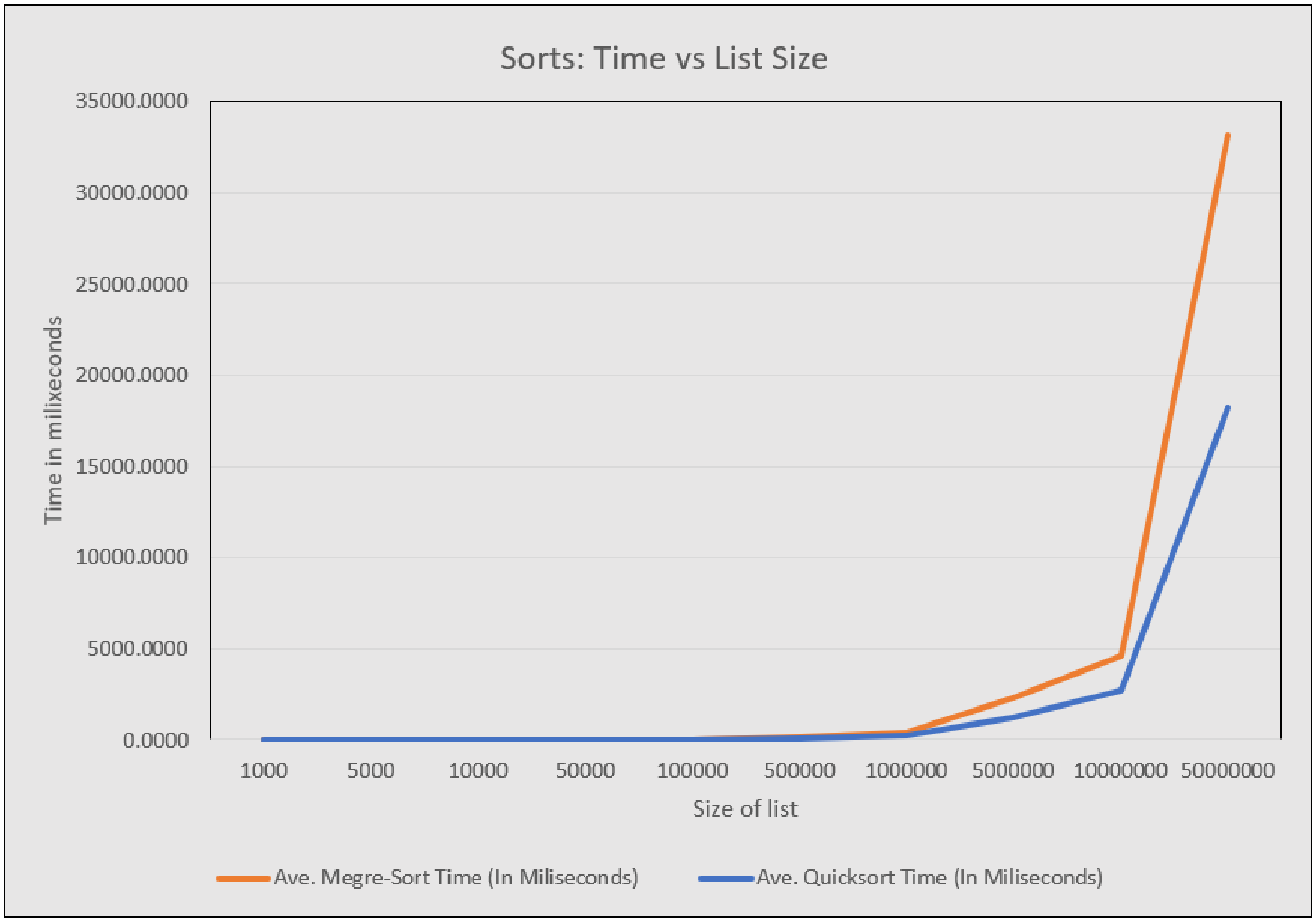
```
public static long TimeMergeSort(int[] list)
{
        long startTime = GenFunctions.NanoTime();
        MergeSort(list, 0, list.Length - 1);
        long endTime = GenFunctions.NanoTime();
        long timeElapsed = endTime - startTime;
        return timeElapsed;
}
public static long TimeQuicksort(int[] list)
{
        long startTime = GenFunctions.NanoTime();
        Quicksort(list, 0, list.Length - 1);
        long endTime = GenFunctions.NanoTime();
        long timeElapsed = endTime - startTime;
        return timeElapsed;
}
```

These methods are almost identical, except for the sorting method that is used. These methods simply get the current time in nanoseconds, run the sort on the list that is taken in as a parameter, and then get the current times again. They then calculate the difference between the start and end times and return the times elapsed.

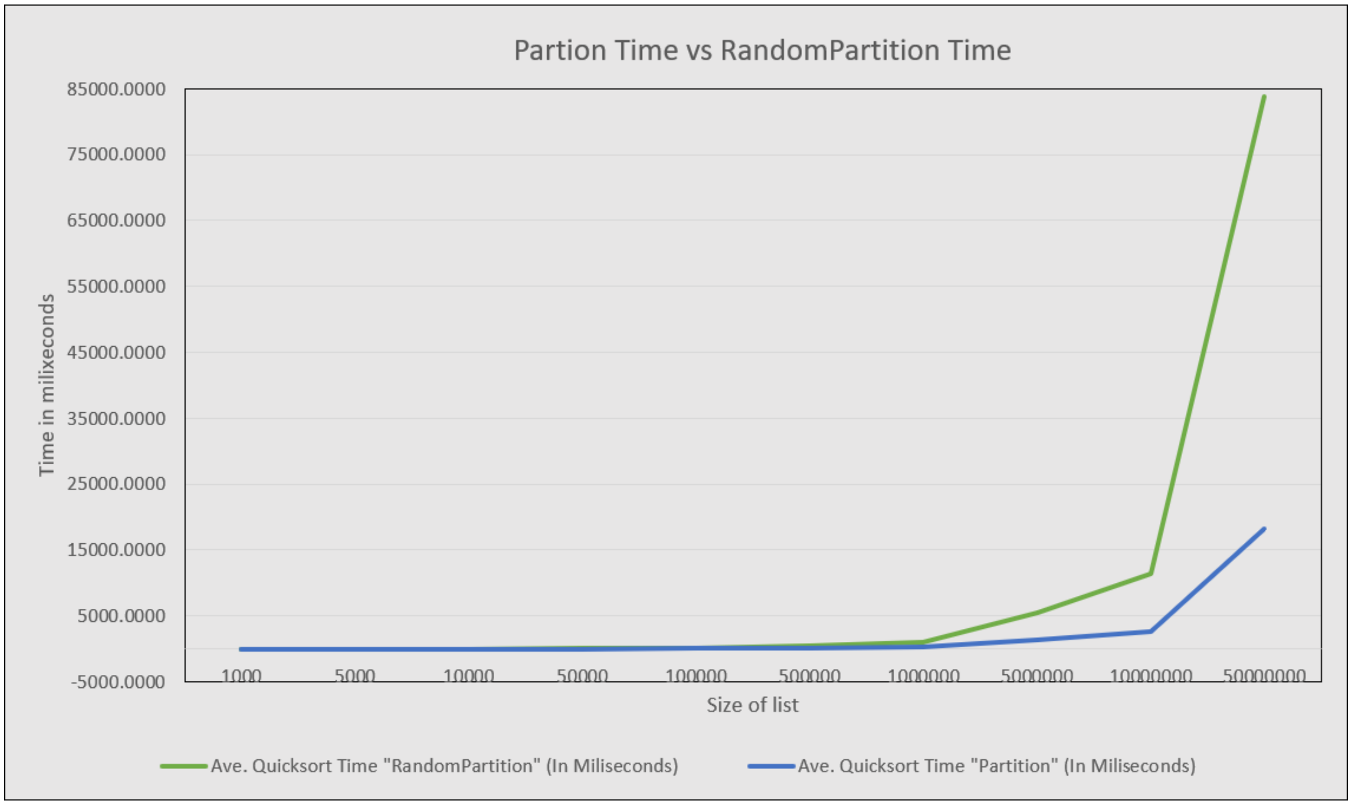
Following are images of the data tables and charts recording the sorting methods trends as the size of the list increases. As one can see, based on the data, quicksort is steadily faster than merge-sort, but both algorithms have a similar trend. It appears to be $O(n \log n)$, but it is somewhat difficult to tell. If I had more time, a lot more time, I would collect data for every 10,000 or so sizes. I think that would demonstrate the complexity better.

The second graph just shows the difference between quicksort with RandomPartition and quicksort with partition. I am not quite sure what happened there, but my assumption is the randomization process actually took a substantial amount of time to perform.

Size	Ave. Megre-Sort Time (In Miliseconds)	Ave. Quicksort Time (In Miliseconds)
1000	0.4942	0.2273
5000	2.0949	0.9548
10000	5.1134	1.8428
50000	18.3236	9.3878
100000	37.0481	19.7916
500000	194.5111	109.0247
1000000	415.9392	259.9464
5000000	2316.8979	1271.8550
10000000	4607.0562	2698.2814
50000000	33180.7843	18258.8311

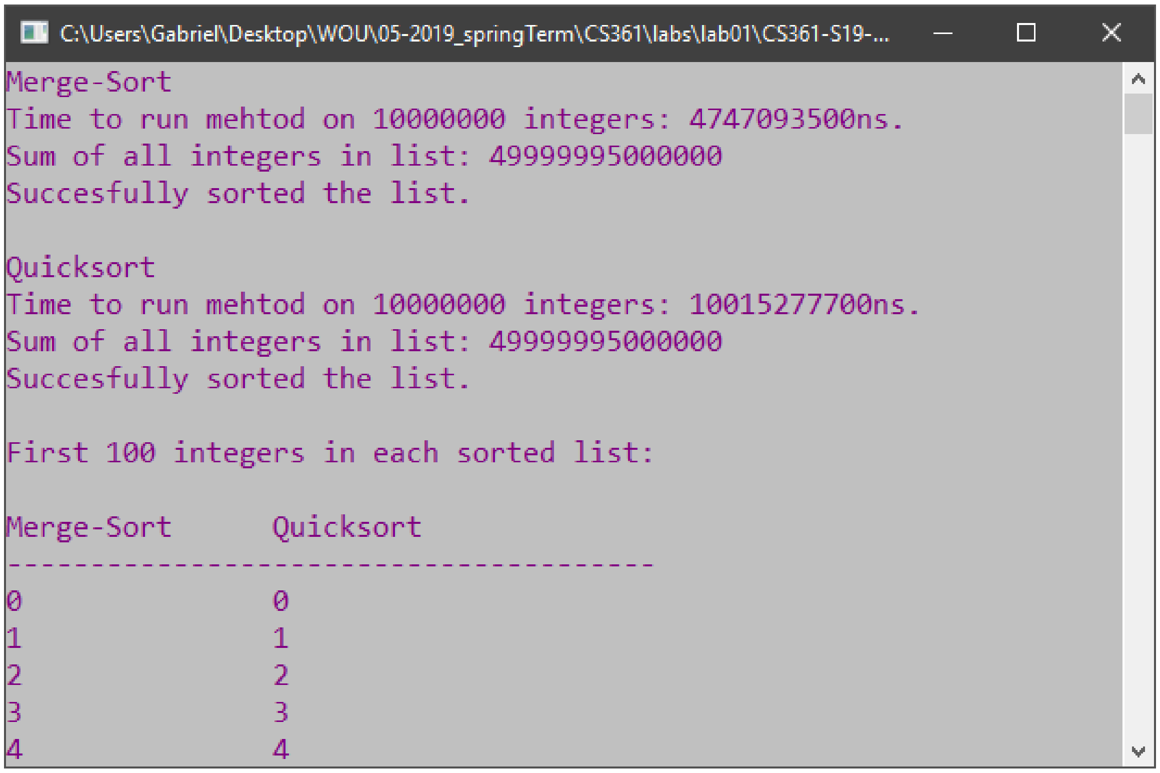


Size	Ave. Quicksort Time "Partition" (In Miliseconds)	Ave. Quicksort Time "RandomPartition" (In Miliseconds)
1000	0.2273	1.1428
5000	0.9548	5.7000
10000	1.8428	10.7515
50000	9.3878	51.3466
100000	19.7916	107.5384
500000	109.0247	532.0156
1000000	259.9464	1075.9366
5000000	1271.8550	5541.7748
10000000	2698.2814	11473.3486
50000000	18258.8311	83738.9187

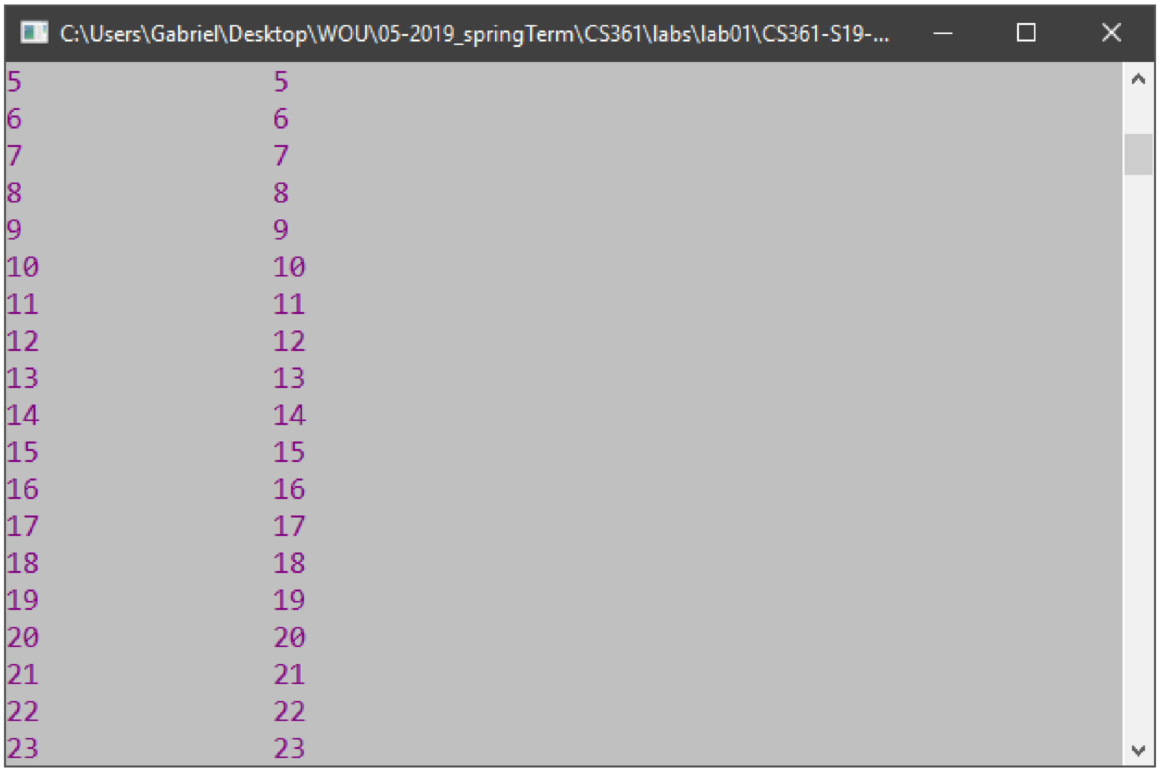


Screen Dumps

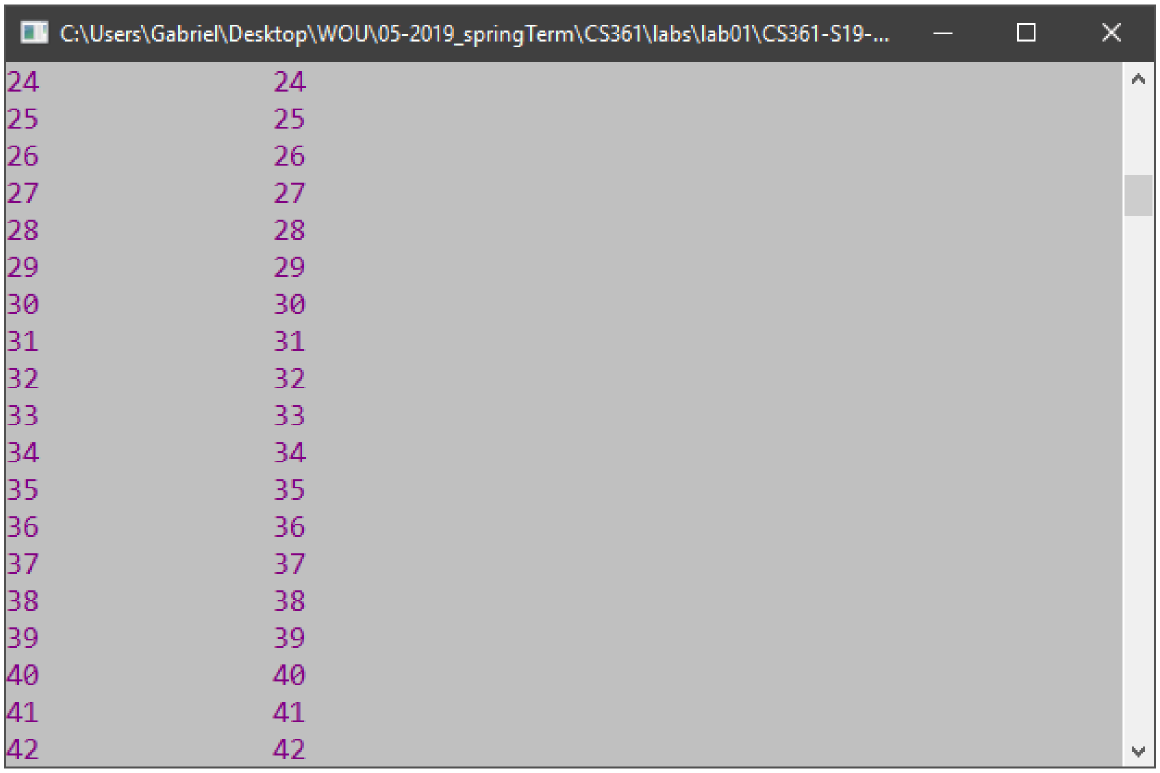
These first six images show the general functionality of the methods written in this project. As one can see, the time it took to sort the given list of integers was about 4.75 seconds and 10.02 seconds for Merge-sort and Quicksort, respectively. Note: this data was recorded using the RandomPartition method. Additionally, the sum of all the integers is 49999995000000, as it should be, and the Boolean value for the IsSorted method returns true. Finally, the first hundred integers can be seen in order after the sorting process finished.



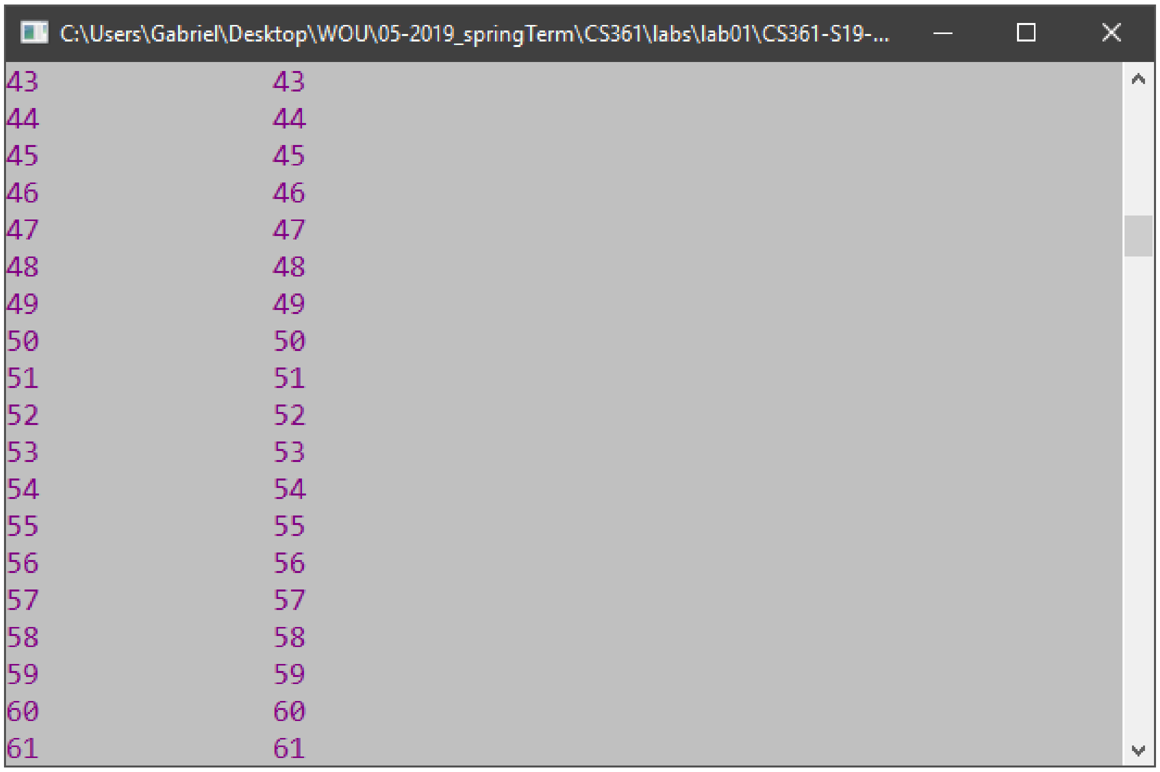
```
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...
Merge-Sort
Time to run mehtod on 10000000 integers: 4747093500ns.
Sum of all integers in list: 49999995000000
Succesfully sorted the list.

Quicksort
Time to run mehtod on 10000000 integers: 10015277700ns.
Sum of all integers in list: 49999995000000
Succesfully sorted the list.

First 100 integers in each sorted list:

Merge-Sort      Quicksort
---------------------------------------
0               0
1               1
2               2
3               3
4               4
```


C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

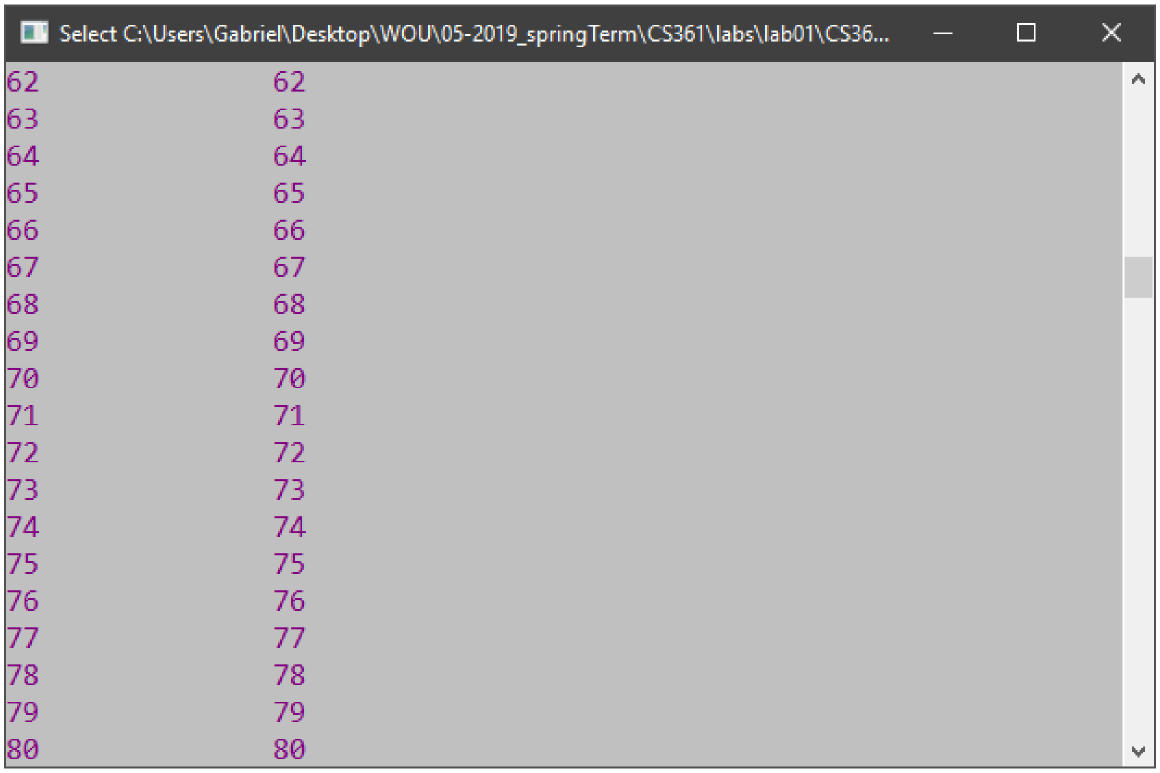
```
5               5
6               6
7               7
8               8
9               9
10              10
11              11
12              12
13              13
14              14
15              15
16              16
17              17
18              18
19              19
20              20
21              21
22              22
23              23
```


C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

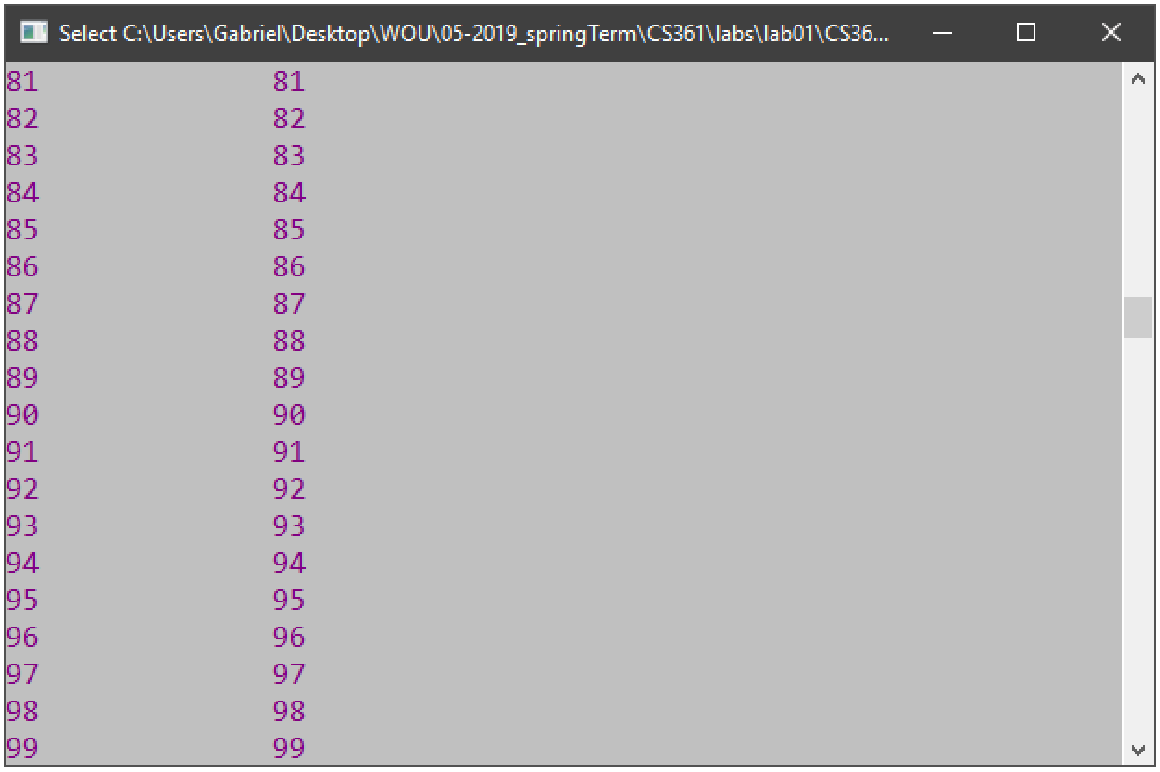
```
24              24
25              25
26              26
27              27
28              28
29              29
30              30
31              31
32              32
33              33
34              34
35              35
36              36
37              37
38              38
39              39
40              40
41              41
42              42
```


C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
43                43
44                44
45                45
46                46
47                47
48                48
49                49
50                50
51                51
52                52
53                53
54                54
55                55
56                56
57                57
58                58
59                59
60                60
61                61
```

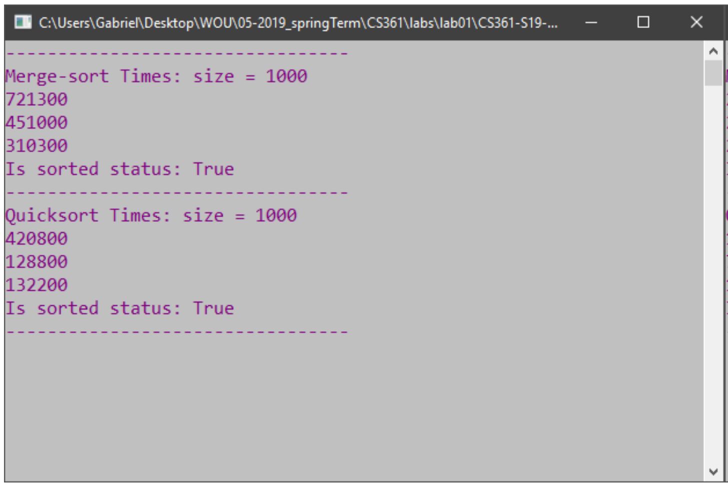

Select C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS36...

```
62                62
63                63
64                64
65                65
66                66
67                67
68                68
69                69
70                70
71                71
72                72
73                73
74                74
75                75
76                76
77                77
78                78
79                79
80                80
```

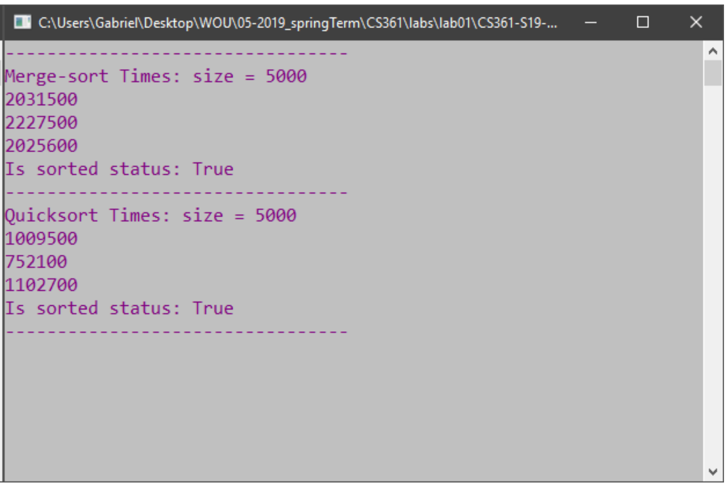


```
Select C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS36...
81              81
82              82
83              83
84              84
85              85
86              86
87              87
88              88
89              89
90              90
91              91
92              92
93              93
94              94
95              95
96              96
97              97
98              98
99              99
```

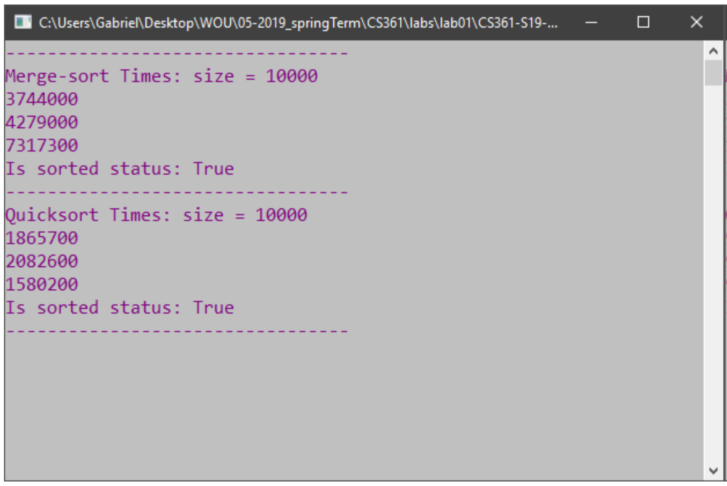
The following ten images show the results of running the main program code ten times. This is the data that is stored in the Excel sheet, and thus, makes up the chart. In each image, the size of the list being sorted is printed out, followed by the time it took for each of three iterations. Then the result of the IsSorted method is then displayed.



```
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...
---------------------------------
Merge-sort Times: size = 1000
721300
451000
310300
Is sorted status: True
---------------------------------
Quicksort Times: size = 1000
420800
128800
132200
Is sorted status: True
---------------------------------
```

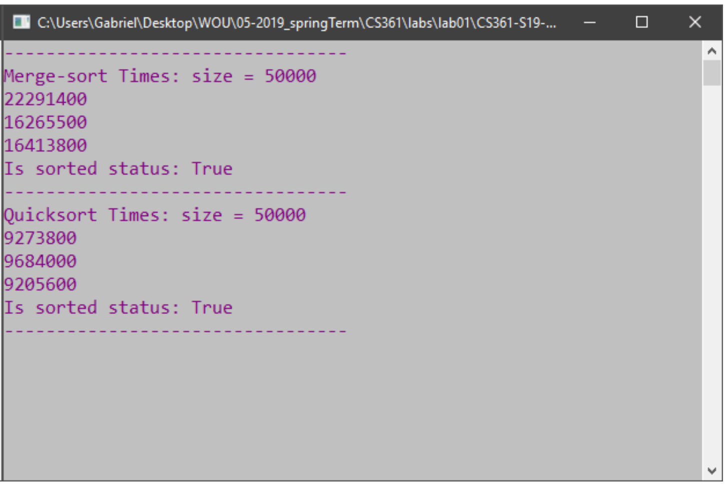


```
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...
---------------------------------
Merge-sort Times: size = 5000
2031500
2227500
2025600
Is sorted status: True
---------------------------------
Quicksort Times: size = 5000
1009500
752100
1102700
Is sorted status: True
---------------------------------
```



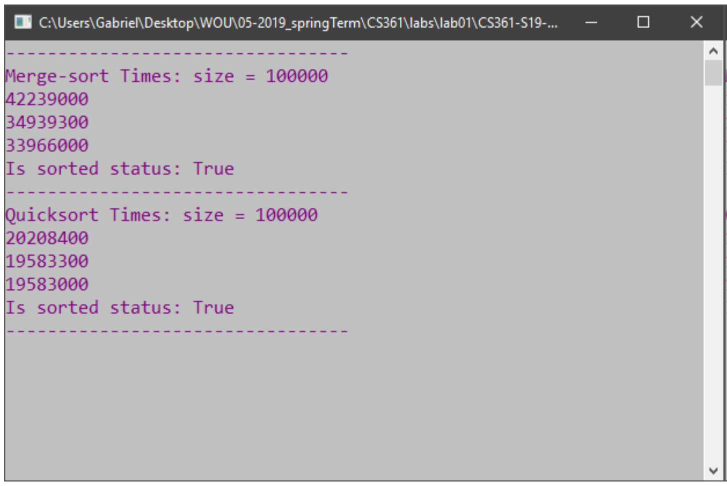
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 10000
3744000
4279000
7317300
Is sorted status: True
---------------------------------
Quicksort Times: size = 10000
1865700
2082600
1580200
Is sorted status: True
---------------------------------
```



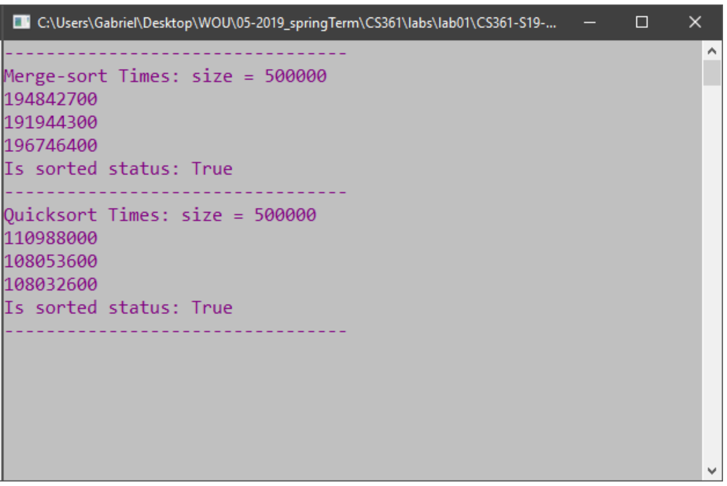
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 50000
22291400
16265500
16413800
Is sorted status: True
---------------------------------
Quicksort Times: size = 50000
9273800
9684000
9205600
Is sorted status: True
---------------------------------
```



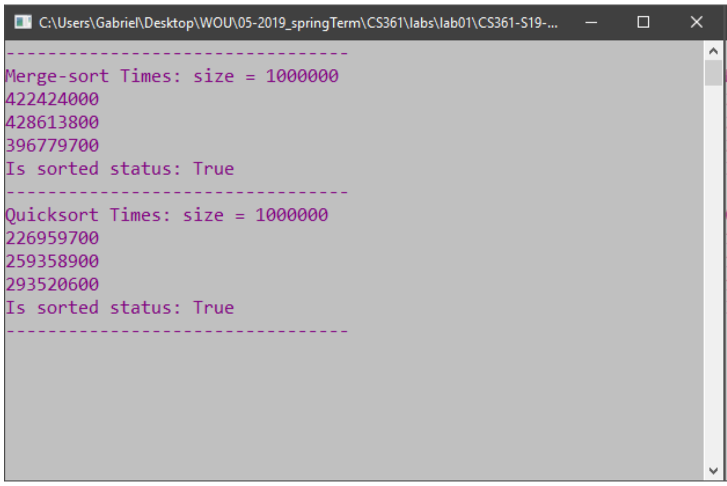
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 100000
42239000
34939300
33966000
Is sorted status: True
---------------------------------
Quicksort Times: size = 100000
20208400
19583300
19583000
Is sorted status: True
---------------------------------
```



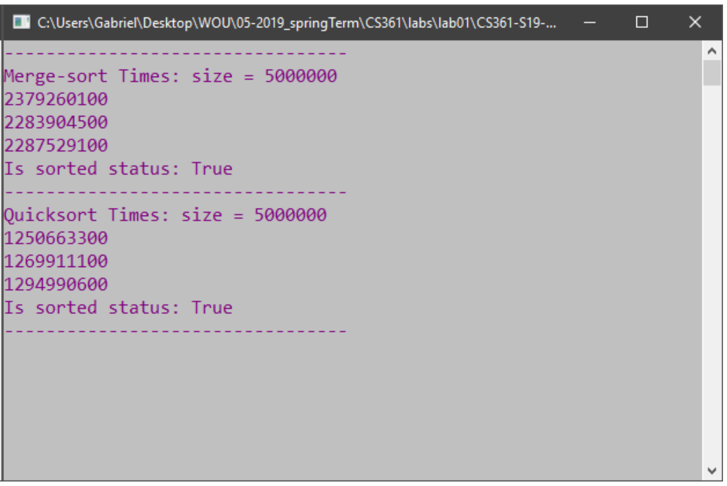
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 500000
194842700
191944300
196746400
Is sorted status: True
---------------------------------
Quicksort Times: size = 500000
110988000
108053600
108032600
Is sorted status: True
---------------------------------
```



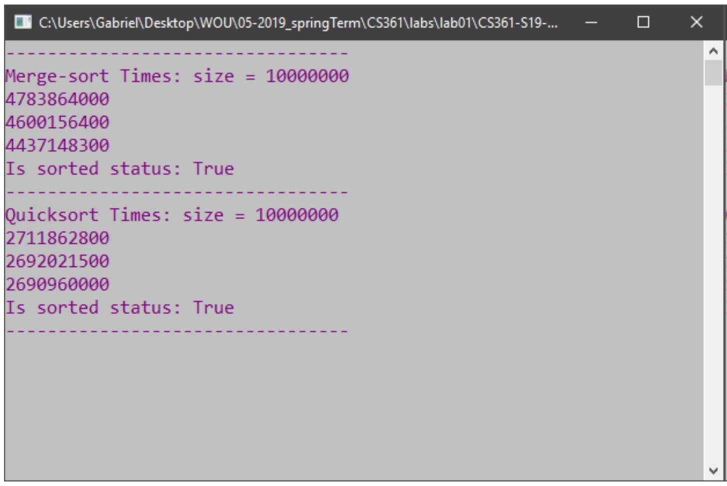
C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 1000000
422424000
428613800
396779700
Is sorted status: True
---------------------------------
Quicksort Times: size = 1000000
226959700
259358900
293520600
Is sorted status: True
---------------------------------
```

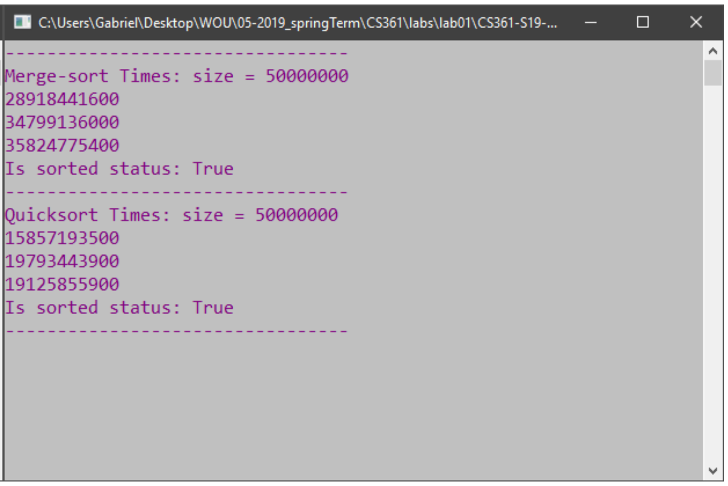


C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 5000000
2379260100
2283904500
2287529100
Is sorted status: True
---------------------------------
Quicksort Times: size = 5000000
1250663300
1269911100
1294990600
Is sorted status: True
---------------------------------
```


C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 10000000
4783864000
4600156400
4437148300
Is sorted status: True
---------------------------------
Quicksort Times: size = 10000000
2711862800
2692021500
2690960000
Is sorted status: True
---------------------------------
```


C:\Users\Gabriel\Desktop\WOU\05-2019_springTerm\CS361\labs\lab01\CS361-S19-...

```
---------------------------------
Merge-sort Times: size = 50000000
28918441600
34799136000
35824775400
Is sorted status: True
---------------------------------
Quicksort Times: size = 50000000
15857193500
19793443900
19125855900
Is sorted status: True
---------------------------------
```

References

I worked with Jacob Malmstadt on this project.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). Cambridge, MA: The MIT Press.

C# Examples. (n.d.). Retrieved April 14, 2019, from http://csharpexamples.com/c-merge-sort-algorithm-implementation/.

LazloLazlo 3, GuffaGuffa 565k79576886, AubinAubin 11.4k64666, SACOSACO 1, Scott M.Scott M. 6, Mike PayneMike Payne 113, & Overstoodoverstood 94567. (n.d.). What is the equivalent to System.nanoTime() in .NET? Retrieved April 14, 2019, from https://stackoverflow.com/questions/1551742/what-is-the-equivalent-to-system-nanotime-in-net.

Bospear. (2017, April 29). Retrieved April 22, 2019, from https://www.youtube.com/watch?v=lsv7rAsvYuA.