UNIVERSITY OF CALGARY | Schulich School of Engineering

Lab Report 10: ENSF 593/594 Principles of Software Engineering Development

Prepared for:

Mohammad Moshirpur

November 29, 2019

Prepared By:

Arsalan Fardi
Asif Bux

Exercise 1

Experimental Method

To test the efficiency of the algorithms implemented in Exercise 1, we measured the processing time to sort an array of varying sizes (10, 1000, 10000, 100000 and 1000000) and type (random, ascending and descending) for each algorithm. Three trials were performed for each test combination and the average time was recorded. The results were then compared to the expected time complexity (O(n)) of the algorithm for average, best and worst case.

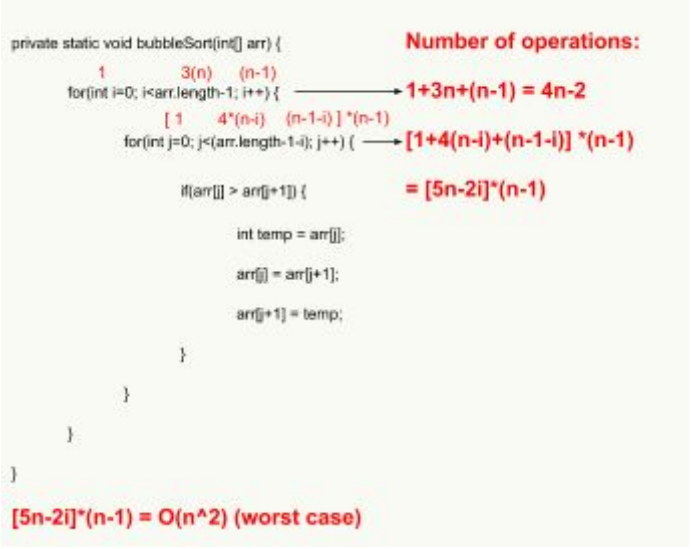
Analysis and Interpretations

A table with data from each trial run can be found in the appendix. Additionally, to allow for comparison, each graph of an algorithm's experimental results includes a plot of the expected algorithmic time complexity.

The experiments for every tested algorithm showed the unintuitive result where the random sorting case was significantly slower than the ascending and descending cases. To avoid repetitiveness, a possible explanation for this result (i.e. branch prediction) will be discussed only in the conclusion of this section.

Bubble Sort

Bubble sort is a simple sorting algorithm that compares successive elements within a list, and if necessary, swaps them into order. The largest elements are essentially "bubbled out" to the right of the list during each pass. In bubble sort, n-1 comparisons are performed on the first pass, n-2 on the second, and so on. This results in time complexity of bubble sort of **$O(n^2)$** for all cases assuming we do not keep track of swaps, as shown below in the implementation below. If a variable is kept to track whether a swap was performed on a pass, then bubble sort can have a best case of O(n) if the array is already sorted.



The experimental results for bubble sort are shown in the figure below. The processing time was by far the slowest of any sorting algorithm. For a size of one million numbers, the ascending and descending finished around the 130 second mark, and the random sort was unable to finish in a reasonable amount of time (<1 hour). The ascending and descending patterns seem to follow an $O(n^2)$ trend. An interesting observation is that the expected best (ascending) and expected worst (descending) cases finish in the same amount of time, and even the expected worst case is significantly better than the random case. This finding can likely be related back to branch prediction, which is discussed in-depth in the conclusion.

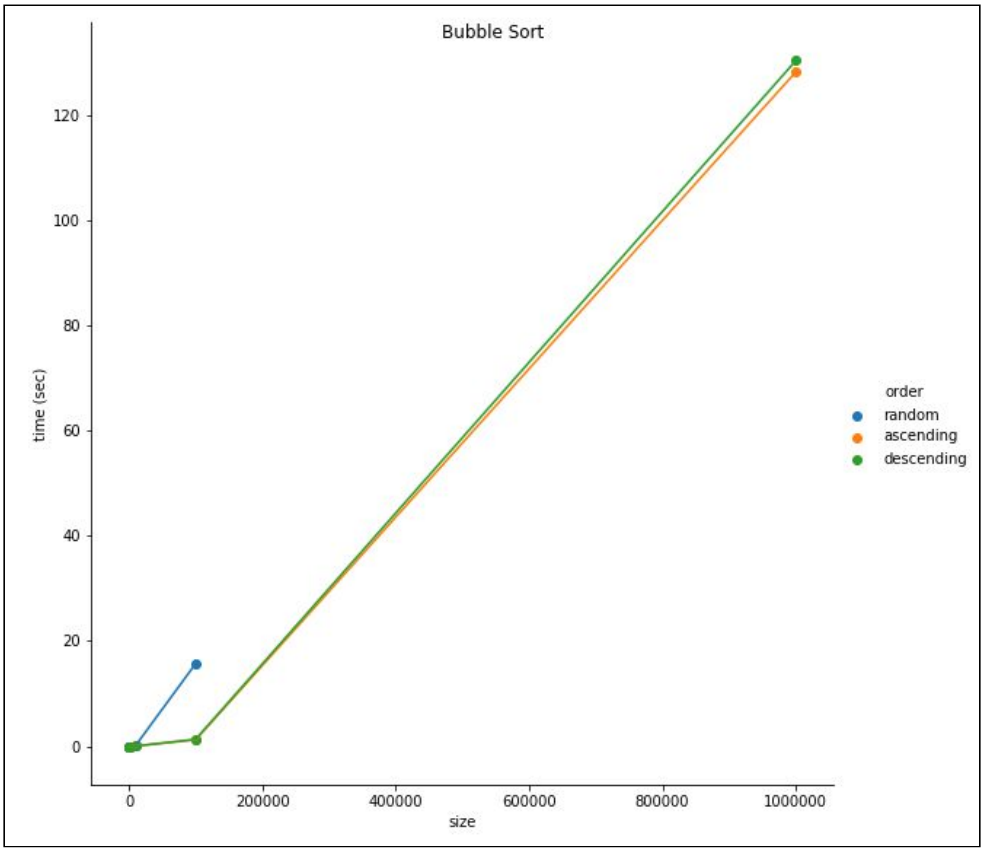


Figure 1: Bubble sort experimental results for sorting time

Insertion Sort:

Insertion sort is another sorting algorithm in which an element is a list is compared to prior elements and is only inserted when its correct position is found, similar to sorting a deck of cards. The worst-case time complexity for insertion sort is **$O(n^2)$** occurring when the list is in descending order, and needs to traverse every previous element before being inserted. A more detailed complexity analysis is shown below. Insertion sort has a best case of O(n) for an already ordered list because the inner loop would never be triggered.

```
private static void insertionSort(int[] arr) {          Number of operations:
            1       2(n+1)    n
        for(int i=0; i<arr.length; i++) {  ⟶  3n+2

                int temp = arr[i];

                int j = i;
                    [ 1n          2(n-1)] *(n)
                while(j>0 && temp<arr[j-1]) {  ⟶  [3n-2]*(n)

                        arr[j] = arr[j-1];

                        j -= 1;

                }

                arr[j] = temp;

        }
}
```

[3n-2]*(n) = O(n^2)

From the experimental results below, insertion sort is much quicker than bubble sort, finishing ascending and descending order in 0.0056 seconds for one million values. This increased efficiency relative to bubble sort can be attributed to the fact that insertion sort swaps a value only once per pass. Although the random case was able to finish, clocking in at 112 seconds for one million values, it was again much slower than the other cases. Interestingly, both the ascending *and* descending order both showed O(n) trends, signaling some optimization during the processing by the computer.

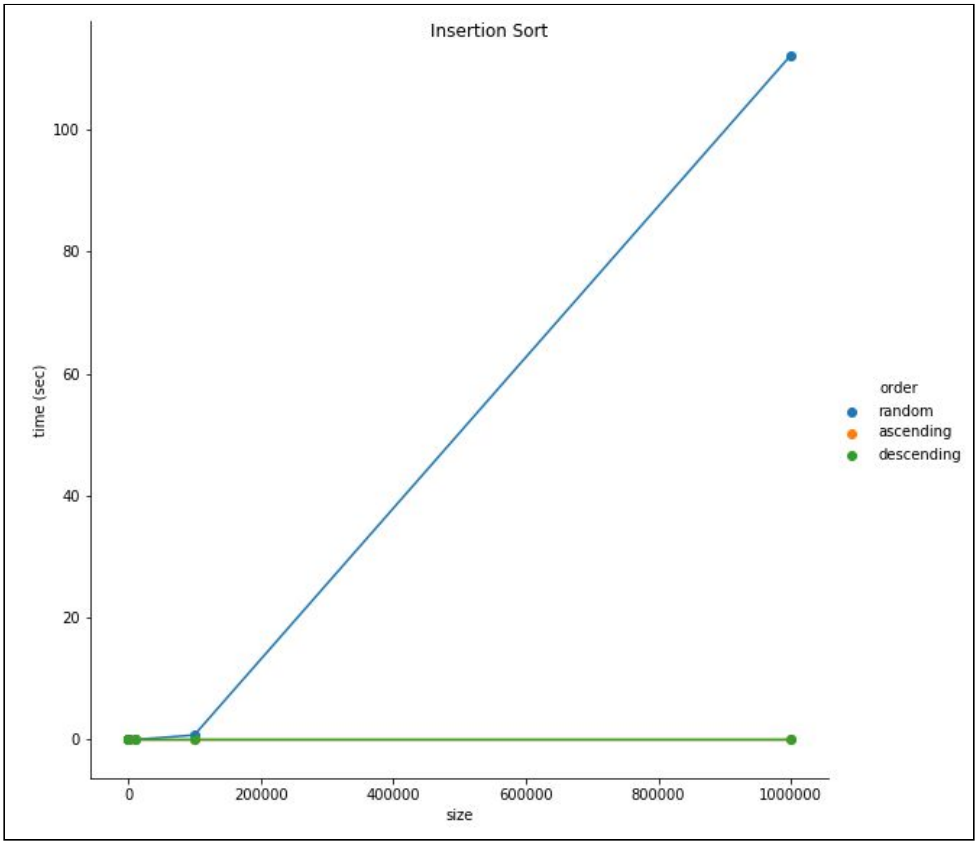
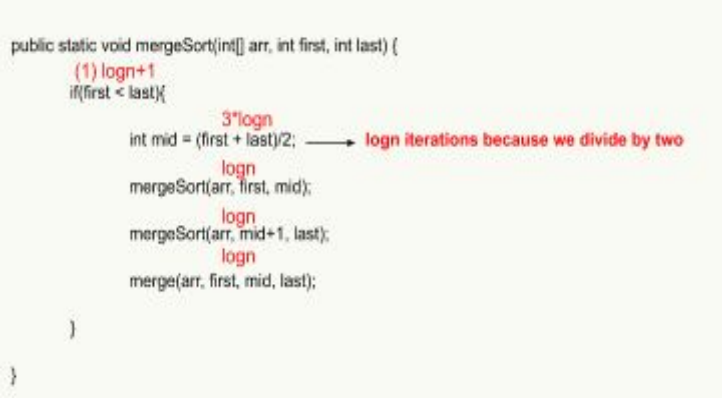
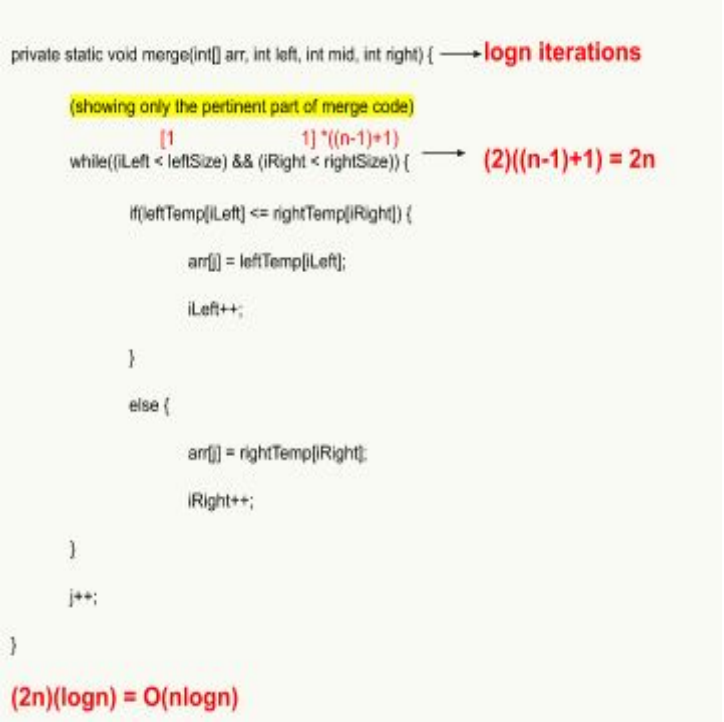


Figure 2: Insertion sort experimental results for sorting time

Merge Sort

Merge sort is a divide and conquer algorithm where the original list is divided in half into sub-lists which are then sorted and finally merged back together in order. Due to the recursive division of the list, the merge sort has a time complexity of **O(nlogn)** for every case, indiscriminate of the input. A detailed complexity analysis is shown below. The main disadvantage of merge sort is the need for temporary arrays, therefore increasing the space complexity.





Merge sorts time complexity can also be seen in a branching structure, with logn divisions and n leaves, as shown below.

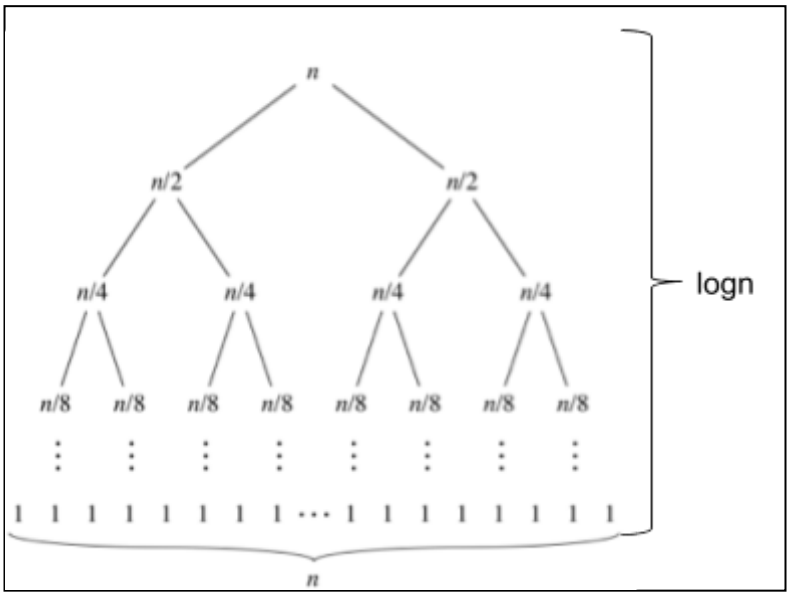


Figure 3: Recursion tree for merge sort showing O(nlogn) complexity

Experimental results for merge short showed an O(nlogn) trend for all of the cases, which was to be expected. The algorithm was much faster than its simple predecessors when sorting the random case, finishing one million elements in approximately 0.150 seconds. The ascending and descending cases at approximately the 0.077 second mark for one million elements. The similarity in processing time for ascending and descending cases was to be expected, because as previously mentioned, the merge sort is indiscriminate of the input.

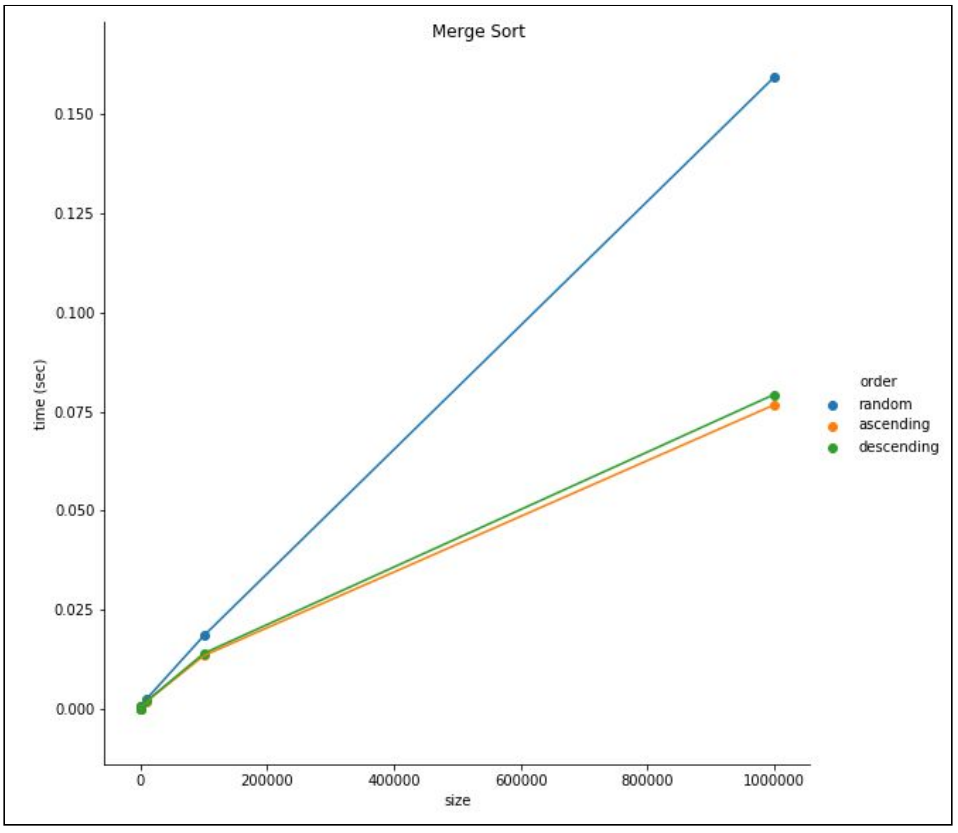


Figure 4: Merge sort experimental results for sorting time

Quicksort:

Quicksort is another divide and conquer algorithm, however, as opposed to merge sort, it occurs in place. The array is partitioned around a pivot value, with everything to the left of the pivot being smaller, and everything to the right larger. This is applied recursively to the original list. Quicksort has an average and best case time complexity of **O(nlogn)**. The best-case occurs when partitions are evenly balanced, and the recursion tree would look very similar to that of the merge sort (refer to Figure 3). The worst case for quicksort occurs when the highest or lowest value is chosen for the pivot, meaning the array is partitioned one index at a time, and gives a complexity of **$O(n^2)$.** To avoid the worst-case complexity in our descending and ascending cases, we chose the pivot value to be the middle element. Other options to avoid the worst-case are to pick a random pivot or use the median value as the pivot.

```
private static void quickSort(int arr[], int begin, int end) {   Number of ops is dependent on
        (1) logn+1 OR n+1                                          partitioning, worst case n
          if (begin < end) {
                                          logn OR n
            int partitionIndex = partition(arr, begin, end);
                                  Logn OR n
            quickSort(arr, begin, partitionIndex-1);
                                  Logn OR n
            quickSort(arr, partitionIndex+1, end);

          }

        }
}
```

```
private static int partition(int[] arr, int start, int end) {      logn OR n iterations

        int pivot = arr[end];

        int pivotIndex = start; //tracks index which pivot will eventually go
            1          n+1  n
        for(int i = start; i<end; i++) { //i              1+(n+1)+n = 2n+2

                if(arr[i] < pivot) {

                        /swap with pivot index position, increment pivot index

                        int temp = arr[i];

                        arr[i] = arr[pivotIndex];

                        arr[pivotIndex] = temp;

                        pivotIndex++;

                }

        }
}
(2n+2)(logn) = O(n^2) OR (2n+2)(n) = O(n^2)
```

The experimental results showed quick sort to be the faster sorting algorithm than merge sort and displayed a trend similar to O(nlogn). The ascending and descending cases were processed within 0.035 seconds for one million elements, and random case at 0.12 seconds. The implementation of quicksort used for this lab involved using the middle value as the pivot, therefore we were effectively able to avoid the worst case of quicksort in the ascending and descending cases. This implementation resulted in our quick sort algorithm performing similar to merge sort for ascending and descending cases. Previous trials with using the end value of the pivot had resulted in overflows

at sizes of one hundred thousand and one million in the ascending and descending cases, as they were effectively the worst case for quicksort.

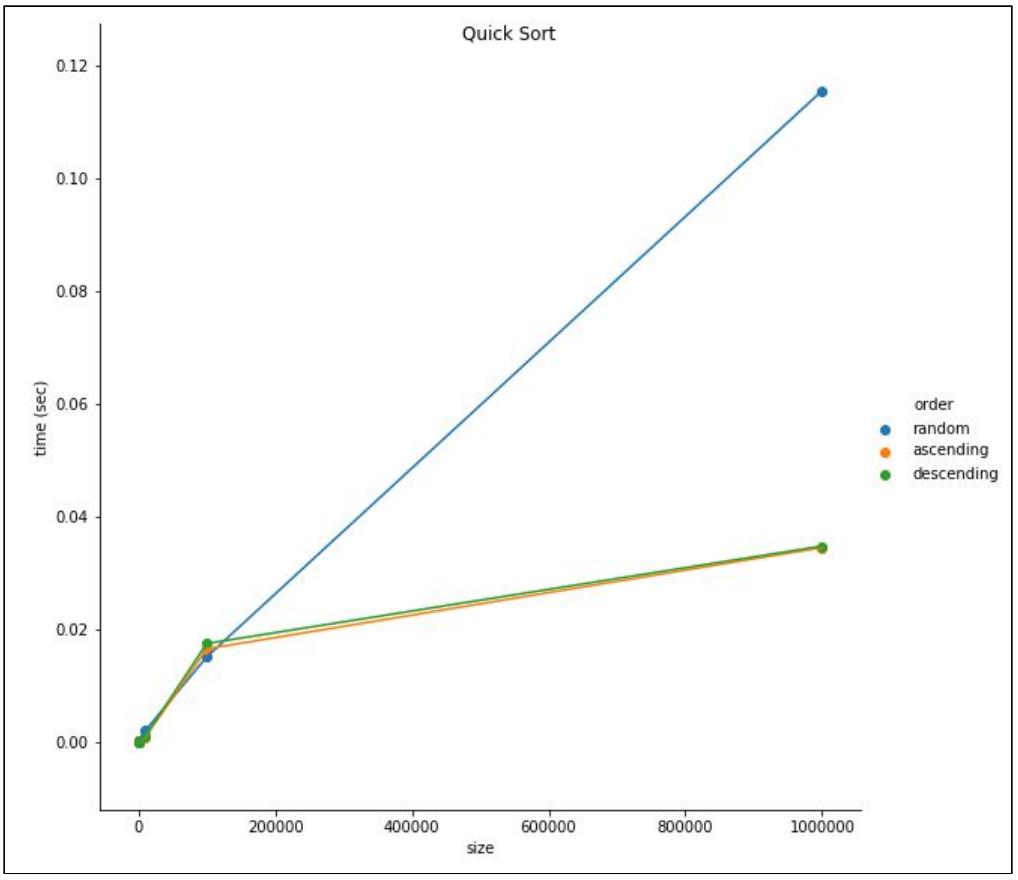


Figure 4: Quicksortt experimental results for sorting time

Conclusion

Our experimental results show bubble sort to be by far the slowest sorting algorithm, as was expected by its $O(n^2)$. Saying that, when comparing the random cases to avoid favoring a certain algorithm for array sizes of up to 100, there is not a significant discrepancy in processing time between the simple and complex sorting algorithms. Therefore, we can extend that for small inputs, the sorting algorithm does not make a significant difference on the processing time.

The divide and conquer algorithms showed their strength when sorting large sizes of 100000 and 1000000 elements, processing in fractions of a second. We were able to see trends consistent with O(nlogn) for both merge and quick sort regardless of the order of inputs, particularly because we chose the middle value each time as the index, effectively nullifying the worst case for quicksort. A precaution with merge sort is not the time, but actually the space complexity. Due to the creation of temporary arrays, merge sort has a space complexity of O(n), which could become an issue with extremely large inputs.

Throughout each case, we observed the peculiar result where the random case, which is supposed to represent the average, perform worse than other cases by a significant margin. An attempt was made to resolve the issue by removing the frequency of duplicates in the random array, with the theory that it was possibly the duplicates causing the algorithm's inefficiency. However, there was no difference in the results. After performing more research, a possible theory explaining this discrepancy is something that may be outside the scope of our analysis or algorithms, branch prediction.

Branch prediction is a technique used by modern processors to guess the outcome of branches (i.e. if-else statements) before they are definitively known. The branch predictor finds patterns within the branches to get a head start on processing the pipeline of instructions. In the case of our sorting algorithms, it could be theorized that when the input order is ascending or descending, the branch prediction is effective in guessing the results of each if-else statement. Therefore, it is able to process the array much faster than an input of random order, where the processor would frequently make incorrect predictions and need to start over. This could also be why we observed a similar processing time for both the best and worst case of insertion sort, as both inputs were in a predictable manner.

Exercise 2

1. What is the worst-case complexity of your algorithm when checking if two words are anagrams of each other? Express this using big-O notation, and use the variable k to represent the number of letters in each word. Support this with a theoretical analysis of your code.

```
public boolean isAnagram(Anagram a1, Anagram a2) {
        if((sortWord(a1)).equalsIgnoreCase(sortWord(a2))) { // n ——→ n
                return true;
        }
        return false;
}
public String sortWord(Anagram a1) {
        char tempArray[] = a1.getWord().toCharArray();  // k     k
                for(int i = 0; i < tempArray.length; i++) { //k
                        char value = tempArray[i]; // k-1
                        int hole = i;            // k-1
                        while((hole > 0 && tempArray[hole-1] > value)) { //k-1
                                tempArray[hole] = tempArray[hole -1]; // k-1
                                hole = hole -1; // k-1
                        }
                        tempArray[hole] = value; // k-1
                        }
  return new String(tempArray);
```

To check if two words are an anagram of each other, an insertion was used to derive the comparison method.

First, the two words are sorted using the insertion sort and then compared using in the string .equalsIgnorecase method which has worst-case complexity of O(k) due to single-use of while loop in

the String's character matching algorithm. Where k is the number of letters in each word from the Node. Now, the worst-case complexity analysis for insertion sort is as following:

Index	Swap	Comparison	Complexity
1	1	1	1(1)
2	2	2	2(2)
3	3	3	3(3)
....	...	...	..
n	(n-1)	(n-1)	2(n-1)

$2(1) + 2(2) + 2(3) + \ldots + 2(n-1) = 2(n-1)(n)/2 = O(n^2)$

Therefore, the worst-case complexity of checking if two words are an anagram of each is a higher order of the two complexities which is $O(n^2)$

2. Let N be the number of words in the input word list, and L be the maximum length of any word. What is the big-O running time of your program? Justify your answer using both a theoretical analysis and experimental data (i.e. timing data).

To determine the overall complexity of the program, it's important to understand the sequence of the algorithm which is executed upon the call. Following is the order of the logic:

1. **insertInOrder()** method is executed when the program creates the ArrayList and adds the LinkedList to each array's elements. Additionally, the order uses the comparable interface method compareTo, which compares the order of the Nodes by calling the String compareTo method for each word in the node. The complexity is first calculated from the compareTo which is O(L Log L) and multiplied by N for number words due to the operation of the outer while loop in this InsertInOrder() method. Therefore, the overall complexity for this method is O(N) times O(L log L) or **O(N* L log L)**.
2. **quicksort()** method is executed when the program has the correct order of anagram words in all the LinkedList which is then sorted using the quicksort algorithm. The complexity of this method is **O(nlogn).**
3. **sortWord()** method is executed using the Insertion sort so that two words can be compared when the words are sorted in ascending order. The complexity of this method is **O(n)**.
4. **equalsIgnoreCase()** method is used to compare if the words in the two Nodes are equal when the words are sorted. The complexity of this method is **$O(n^2)$**.

Therefore, the overall worst-case complexity of the program is **$O(n^2)$**.

Figure 5 below shows the time complexity for the overall program using the word list of 1000, 10000, 20000, and 40000 words. The plot fits the trendline of approximately $\mathbf{n^2}$ with some minor deviation. Therefore, we can conclude that theoretical analysis and experimental data provide approximately the same complexity of $\mathbf{O(n^2)}$ for average or worst case of the overall program**.**

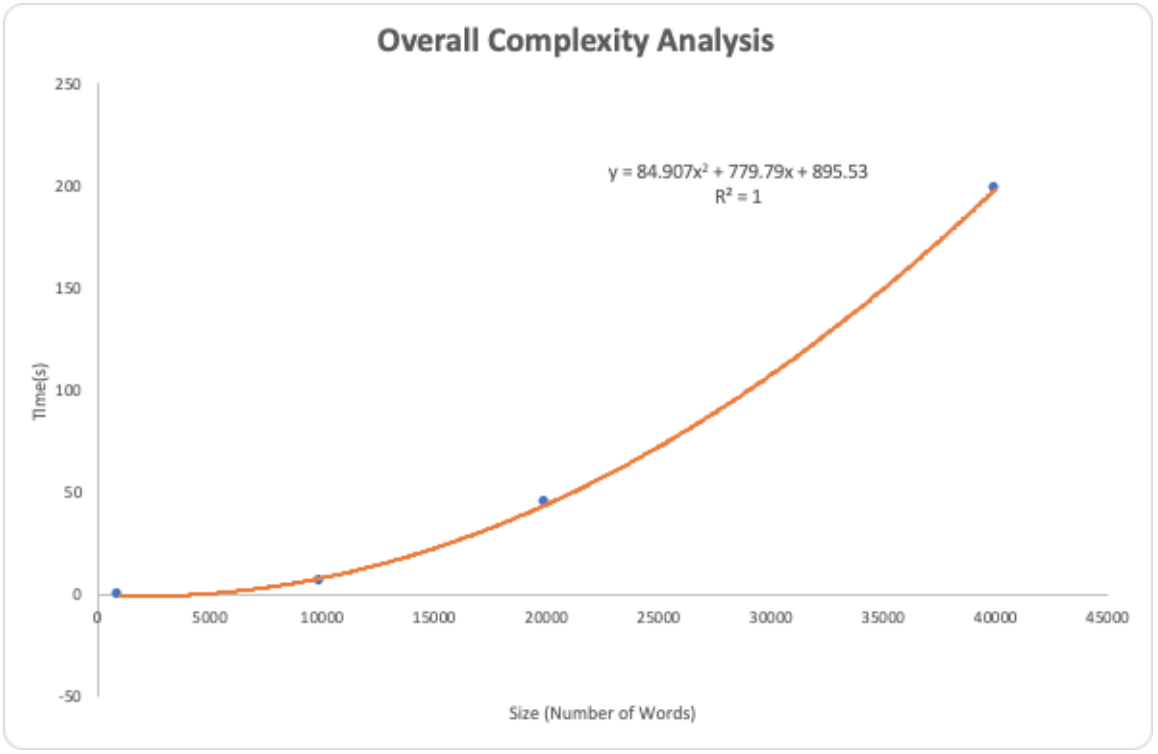


Figure 5: Overall Complexity Analysis

Appendix: Results Tables for Exercise 1

Table 1: Bubble sort results

Order	Time (sec)					
	10	**100**	**1000**	**10000**	**100000**	**1000000**
random	3.39E-05	0.000225	0.005107	0.125445	15.68164	-
ascending	3.52E-05	0.000136	0.002909	0.017146	1.213506	128.142
descending	3.52E-05	0.00015	0.002945	0.016942	1.303709	130.2812

Table 2: Insertion sort results

Order	Time (sec)					
	10	**100**	**1000**	**10000**	**100000**	**1000000**
random	3.15E-05	0.000079	0.00288	0.01591	0.772201	112.2105
ascending	3.47E-05	4.14E-05	7.76E-05	0.00035	0.002106	0.005673
descending	0.00004	5.35E-05	7.05E-05	0.000362	0.002049	0.005648

Table 3: Merge sort results

Order	Time (sec)					
	10	**100**	**1000**	**10000**	**100000**	**1000000**
random	2.84E-05	7.76E-05	0.000713	0.002519	0.018417	0.159349
ascending	4.07E-05	9.21E-05	0.000618	0.00186	0.013385	0.07666
descending	4.56E-05	0.000111	0.000712	0.001959	0.013961	0.079283

Table 4: Quick sort results

Order	Time (sec)					
	10	**100**	**1000**	**10000**	**100000**	**1000000**
random	3.07E-05	6.37E-05	0.000377	0.002155	0.015221	0.115462

ascending	4.15E-05	5.19E-05	0.000315	0.00102	0.016636	0.034563
descending	3.73E-05	6.45E-05	0.000305	0.001194	0.017583	0.034831