ENSF-594

Final Report

Mike Lasby

8/12/2020

Exercise A: Implementation and Comparison of Sorting Algorithms

Algorithm Complexity Analysis:

For the following complexity analysis the letter “n” is used to denote the input array size (ie., arr.length).

Bubble Sort:

Line	Algorithm	Times (highest order)
1	int temp;	1
2	for (int i = 1; i < arr.length; i++) {	**n-1**
3	for (int j = 0; j < arr.length - 1; j++) {	**n-1**
4	if (arr[j] > arr[j + 1]) {	n-1
5	temp = arr[j];	n-1
6	arr[j] = arr[j + 1];	n-1
7	arr[j + 1] = temp;	n-1

Bubble sort has an average and worst case time complexity of O(n^2) due to the nested for loops which dominate the expression. The assignment expressions within the for loops will be executed n-1 times, but occur in constant time. Bubble sort is not an appropriate algorithm to select in most cases.

Insertion Sort:

Line	Algorithm	Times (highest order)
1	int temp;	1
2	for (int i = 1; i < arr.length; i++) {	**n**
3	int j = i;	1
4	while (j > 0 && arr[j - 1] > arr[j]) {	**n-1**
5	temp = arr[j];	n-1
6	arr[j] = arr[j - 1];	n-1
7	arr[j - 1] = temp;	n-1
8	j--;	n-1

Similar to bubble sort, insertion sort has a worst case complexity of O(n^2) due to the while loop nested within the for loop. Insertion sort is much more effective than bubble sort as it will only perform a maximum of one swap per iteration, whereas bubble sort may swap from 0 to n times. Swapping may have a high computational overhead on very large data sets that cannot be fully loaded into the cache.

MERGE SORT:

Line	Algorithm	Times (highest order)
1	if (l < r) {	
2	int mid = (l + r) / 2;	1
3	mergeSort(arr, l, mid);	**log N**
4	mergeSort(arr, mid + 1, r);	**log N**
5	merge(arr, l, mid, r);	**n** (for func.)
6		
7	**Merge Function:**	
8	int[] left = new int[mid - l + 2];	1
9	int[] right = new int[r - mid + 1];	1
10	for (int i = 0; i < left.length - 1; i++) {	**(n-1)/2**
11	left[i] = arr[l + i];}	1
12	for (int j = 0; j < right.length - 1; j++) {	**(n-1)/2**
13	right[j] = arr[mid + j + 1];}	1
14	int i = 0;	1
15	int j = 0;	1
16	right[right.length - 1] = Integer.MAX_VALUE;	1
17	left[left.length - 1] = Integer.MAX_VALUE;	1
18	for (int k = l; k <= r; k++) {	**n**
19	if (left[i] <= right[j]) {	n
20	arr[k] = left[i];	n
14	i++;	n
15	} else {	n
16	arr[k] = right[j];	n
17	j++;	n

Merge Sort is the first example of a divide and conquer sorting function. The merge sort function is recursive, calling itself until all elements have been divided into trivial single element sub arrays. Once this is completed, the functions return back up the binary sorting tree until all sub arrays have been merged. Merge sort has a worst case complexity of O(n log n) due to the depth of the recursion tree and the required number of operations to split the array into its sub arrays.

QUICK SORT:

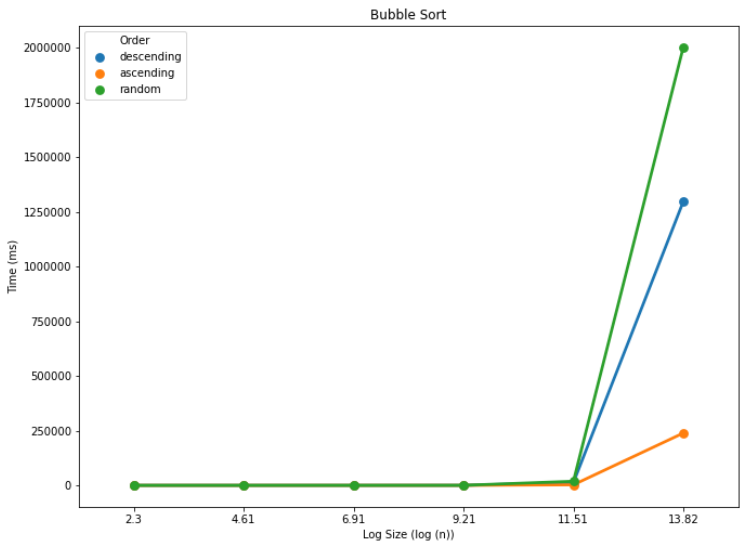
Line	Algorithm	Times (highest order)
1	if (l >= r) {	1
2	return;}	1
3	Random rand = new Random();	1 (more complex in reality, but implementation is hidden
4	int randIdx = l + rand.nextInt(r - l + 1);	1
5	swap(arr, randIdx, r);	1
6	int pivot = arr[l];	1
7	`int pIdx = l;`	1
8	for (int i = l + 1; i <= r; i++) {	**n-1**
9	if (arr[i] <= pivot) {	1
10	pIdx++;	1
11	swap(arr, i, pIdx);	1
12	swap(arr, l, pIdx);	1
13	quickSort(arr, l, pIdx - 1);	**(n-1)/2**
14	quickSort(arr, pIdx + 1, r);	**(n-1)/2**

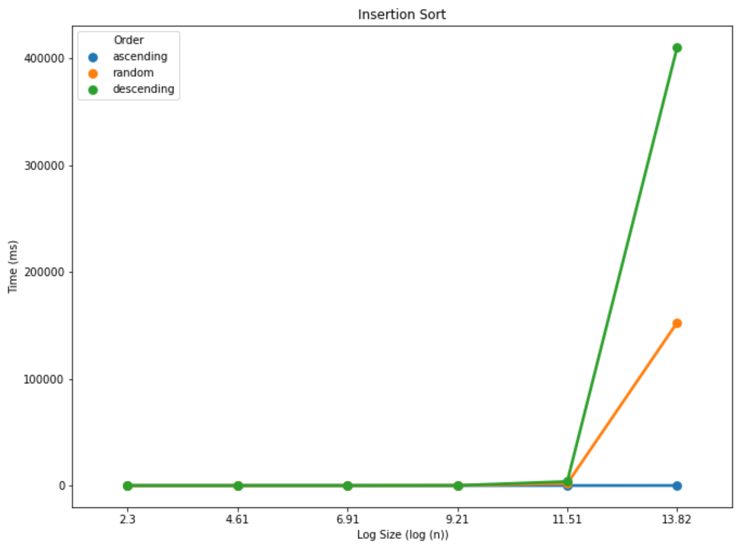
Quick sort's worst case running time is O(n^2); however, the average complexity is only O(n log n). Due to the fact that quick-sort sorts in place, it is a very efficient algorithm with respect to memory space and can often utilize cache more effectively that merge sort.

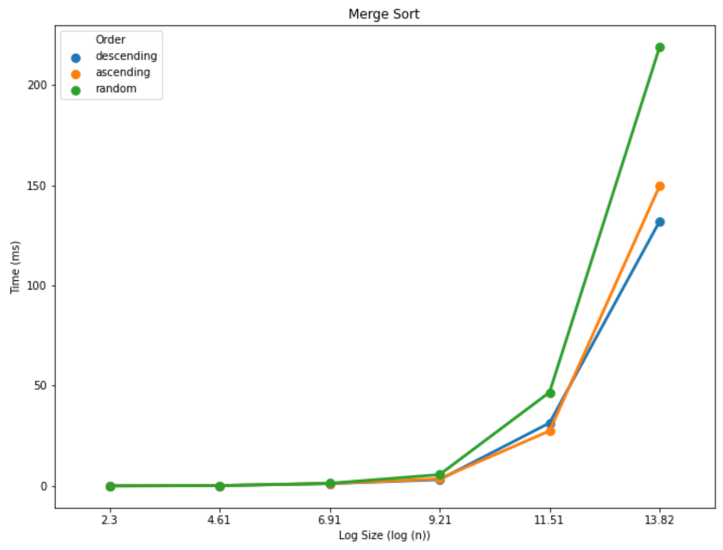
EXPERIMENT METHODOLOGY

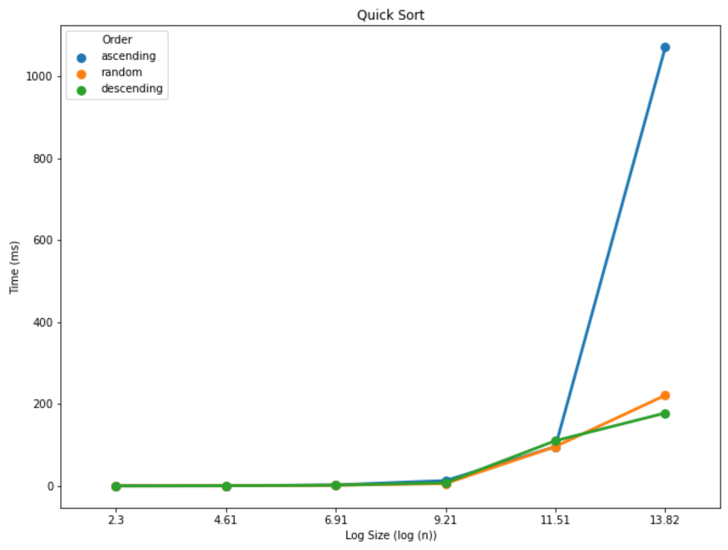
Each of the four sorting algorithms above were tested with ascending, descending and randomised arrays. Additionally, the sorting algorithms were tested with array lengths of 10, 100, 1000, 10 000, 100 000, and 1 000 000 length arrays. In total, 72 trials were run covering each permutation of the above-noted controlled variables.

Due to the length of time required to run the large arrays, particularly with the inefficient bubble and insertion sorts, a bash script was used to automate the program execution. Once all files had been output, a python script was used to pull the relevant data out of the .csv file headers and plot using a jupyter notebook. See the below plot for a summary of observations.









As we can see from the above data, the algorithm performance becomes most clear with very large input sizes. Bubble and merge sort performed the worst with random ordering, whereas the insertion sort and quick sort algorithms performed worst on descending and ascending data, respectfully.

Large variations in performance were noted between algorithms which have the same time complexity. For example, the bubble sort took orders of magnitude more time to compute the large data sets than insertion sort. The random ordered 1,000,000 sized arrays took approximately 30 minutes to compute. Ideally, more trials would have been run to control for random variability within my computer; however, the data obtained in a single run clearly shows the general trends between the various sorting algorithms.

The order of the data had a large effect on the speed of the algorithms. Merge and quick sort both performed very well on most data sets, but it's worth noting that the quick sort had the slowest times on arrays that were already sorted. The quick sort was implemented with a random pivot to attempt to reduce the methods inherit weakness to ascending and descending orders. Further refinement on our pivot selection for large data sets, such as selecting a median value, would further reduce the risk of the worst case complexity occurring for a typical array.

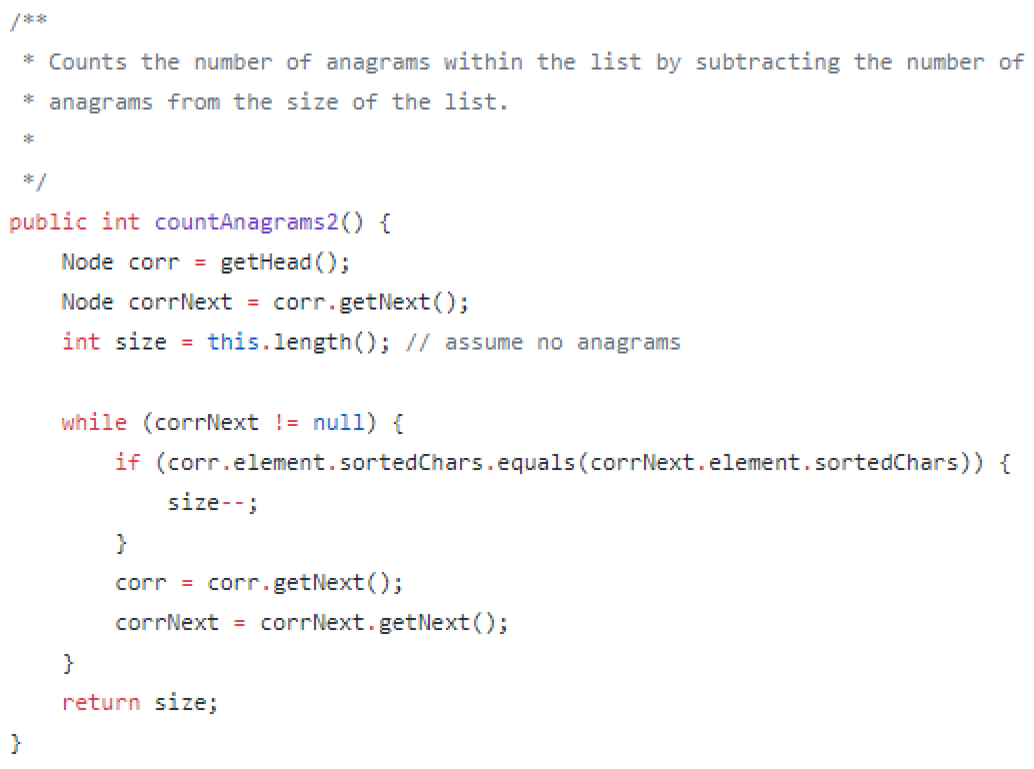
While quick sort was slower than merge sort on large data sets, it sorts in place and therefore does not require additional space in memory for our sorting operation.

If very large datasets are being sorted and memory space is not a concern, I recommend use of merge sort as it scales very well with large data sets. For a good balance between speed and space efficiency, quick sort is the best choice, especially if a more sophisticated method of obtaining the pivot is used.

EXERCISE B: LINKED LISTS AND SORTING

ANAGRAM DETERMINATION

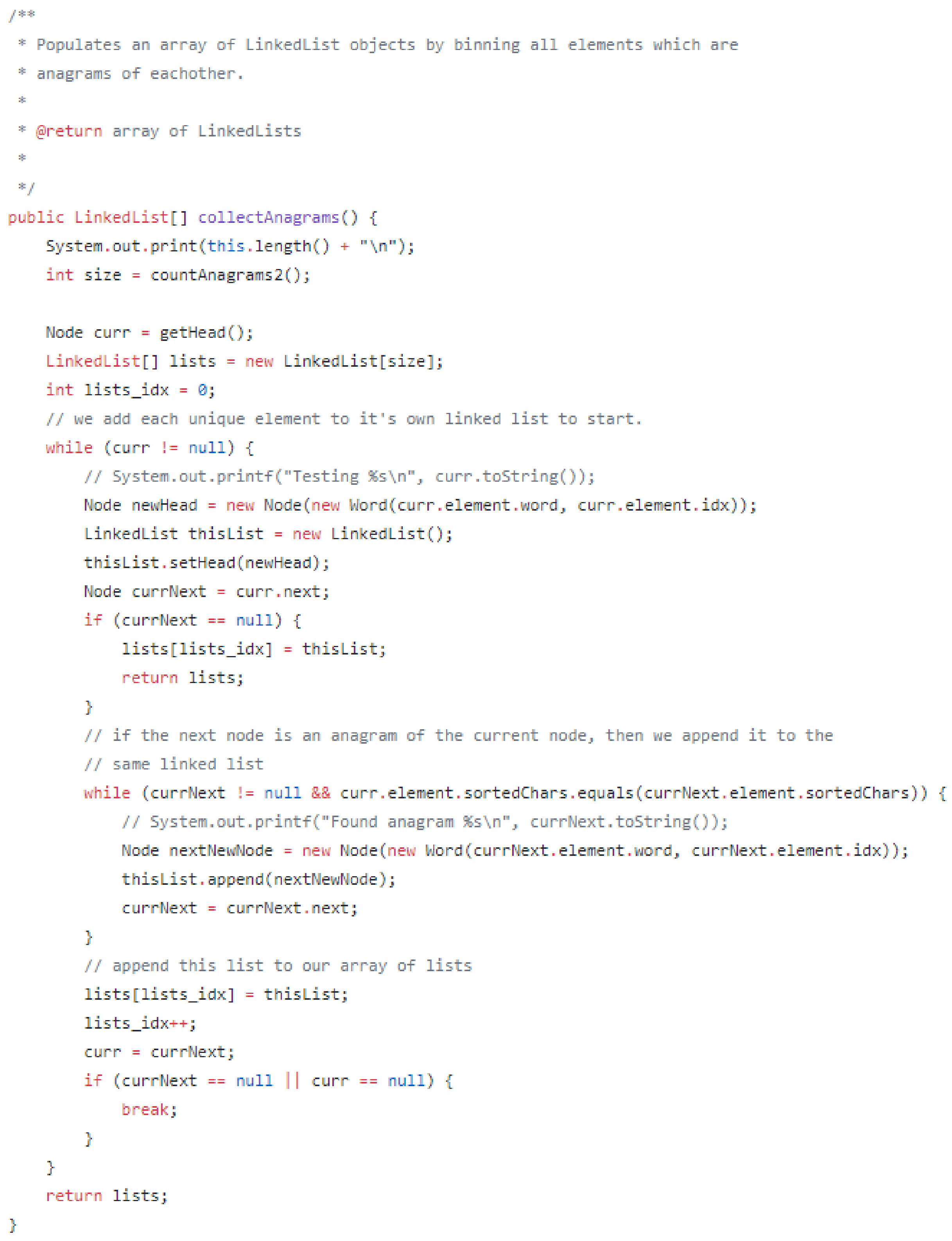
The code snippets below depict the functions used to determine which words are anagrams of each other. The words in the text file are parsed into a linked list structure to allow for dynamically resizing of the data structure. Once we have parsed the file, the linked list is sorted with respect to the alphabetically arranged array of characters within each word. The sorting is performed by an insertion sorting algorithm that has a complexity of $O(n^2)$. However, the benefit of pre-sorting the list is that we may simply traverse the linked list n times to determine the total number of anagrams within the input text. As seen below, two cursors simply increment along the list from the head to tail and since our elements have been sorted, any anagrams must be immediately adjacent to the previous node. Therefore, the countAnagrams function has a complexity of O(n).



```
/**
 * Counts the number of anagrams within the list by subtracting the number of
 * anagrams from the size of the list.
 *
 */
public int countAnagrams2() {
    Node corr = getHead();
    Node corrNext = corr.getNext();
    int size = this.length(); // assume no anagrams

    while (corrNext != null) {
        if (corr.element.sortedChars.equals(corrNext.element.sortedChars)) {
            size--;
        }
        corr = corr.getNext();
        corrNext = corrNext.getNext();
    }
    return size;
}
```

Figure 2: Counting Anagrams



```java
/**
 * Populates an array of LinkedList objects by binning all elements which are
 * anagrams of eachother.
 *
 * @return array of LinkedLists
 *
 */
public LinkedList[] collectAnagrams() {
    System.out.print(this.length() + "\n");
    int size = countAnagrams2();

    Node curr = getHead();
    LinkedList[] lists = new LinkedList[size];
    int lists_idx = 0;
    // we add each unique element to it's own linked list to start.
    while (curr != null) {
        // System.out.printf("Testing %s\n", curr.toString());
        Node newHead = new Node(new Word(curr.element.word, curr.element.idx));
        LinkedList thisList = new LinkedList();
        thisList.setHead(newHead);
        Node currNext = curr.next;
        if (currNext == null) {
            lists[lists_idx] = thisList;
            return lists;
        }
        // if the next node is an anagram of the current node, then we append it to the
        // same linked list
        while (currNext != null && curr.element.sortedChars.equals(currNext.element.sortedChars)) {
            // System.out.printf("Found anagram %s\n", currNext.toString());
            Node nextNewNode = new Node(new Word(currNext.element.word, currNext.element.idx));
            thisList.append(nextNewNode);
            currNext = currNext.next;
        }
        // append this list to our array of lists
        lists[lists_idx] = thisList;
        lists_idx++;
        curr = currNext;
        if (currNext == null || curr == null) {
            break;
        }
    }
    return lists;
}
```

Figure 3: Collecting Anagrams

In a similar fashion, we can see that collecting the anagrams into the new sub linked lists also has a complexity of n as we only need to traverse the list from head to tail once. Should anagrams be found, we add those to our sub linked list before moving the cursor position to the element immediately following the last matching anagram.

In total, we can sum the above as follows:

- Selection Sort: $O(n^2)$
- Counting Anagrams $O(n)$
- Collecting Anagrams $O(n)$

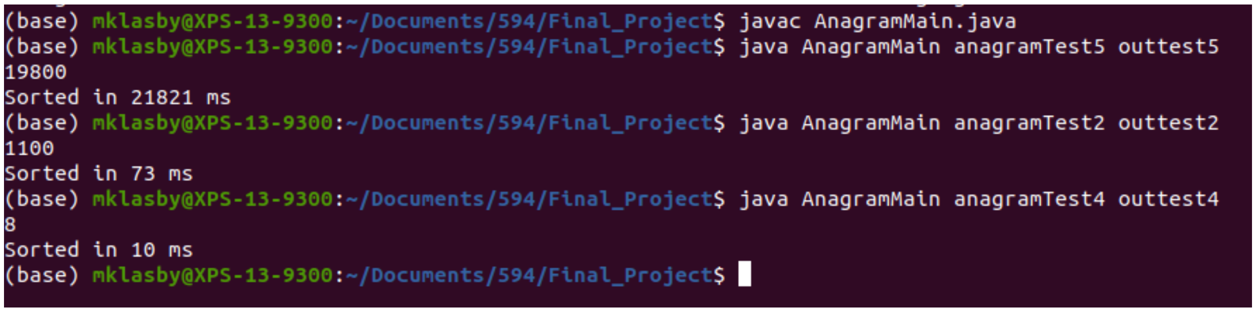
Since the quadratic function will dominate the other terms, we can say that these steps have an overall complexity of $O(n^2)$.

However, we must also consider the maximum length of each word. Larger words have a higher cost to sort into their constitute characters. Since we used an insertion sort algorithm for sorting the characters as well, we can describe the complexity as follows: $O(n*L^2)$ where L is the maximum length of any word in the set. We have to sort each word (1, 2, 3…n) and sorting of the characters will take L^2 iterations.

Therefore, we can describe the overall complexity as $O(n^2 + L^2n)$.

To test this analysis, we generated a large data set of 19800 words with many repeated words. This simulates having many anagrams, which add more complex constant time operations to our algorithms, such as initializing new linked lists. The largest word in our sample was 7 characters.

We can see a sample output of some of our tests below:



```
(base) mklasby@XPS-13-9300:~/Documents/594/Final_Project$ javac AnagramMain.java
(base) mklasby@XPS-13-9300:~/Documents/594/Final_Project$ java AnagramMain anagramTest5 outtest5
19800
Sorted in 21821 ms
(base) mklasby@XPS-13-9300:~/Documents/594/Final_Project$ java AnagramMain anagramTest2 outtest2
1100
Sorted in 73 ms
(base) mklasby@XPS-13-9300:~/Documents/594/Final_Project$ java AnagramMain anagramTest4 outtest4
8
Sorted in 10 ms
(base) mklasby@XPS-13-9300:~/Documents/594/Final_Project$
```

Figure 4: Terminal output of AnagramMain

Table 1: Time (ms) vs n and L

n (elements)	L (char)	Time (ms)
8	7	8
1100	7	73
19800	7	21821

Figure 5: Time (ms) vs n elements

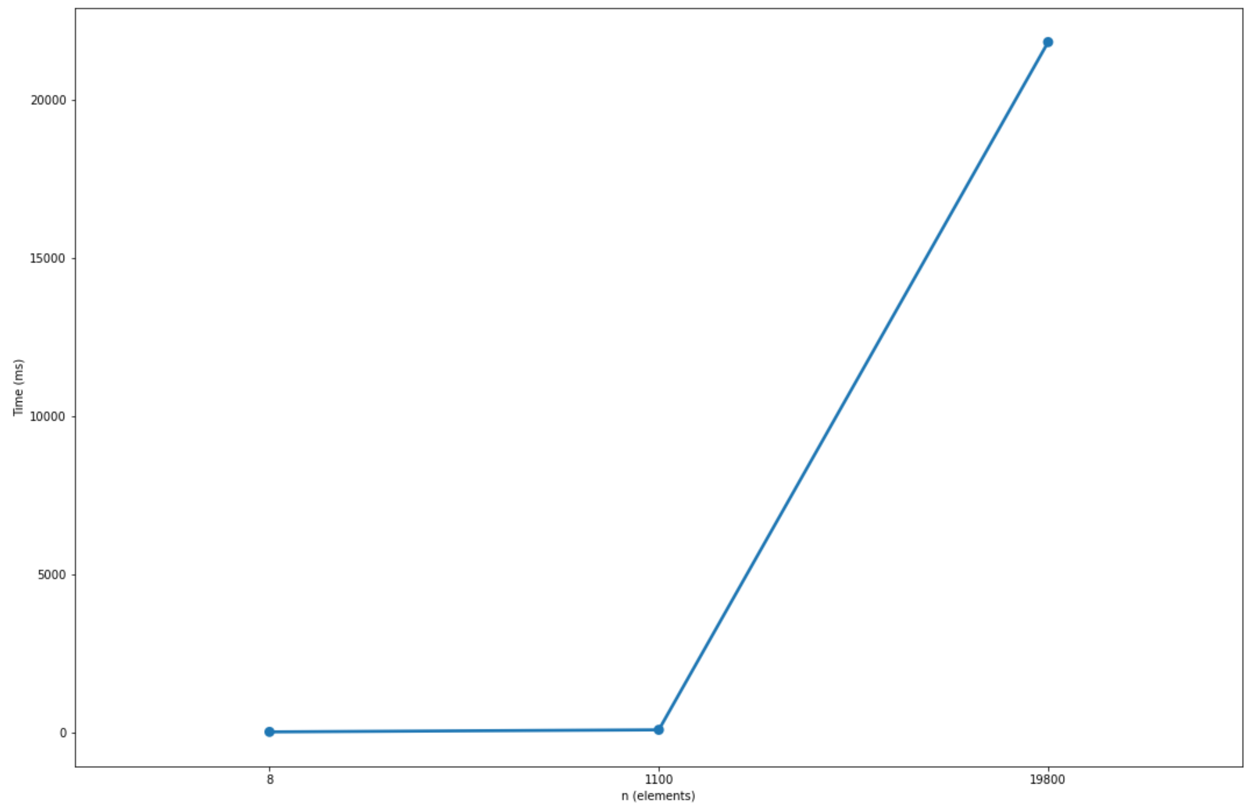


Figure 6: Time(ms) vs Element Size

Even with the few points we have available, we can see that indeed there is a quadratic relationship with respect to n.

Sincerely,

M. Lasby

Mike Lasby
mklasby@gmail.com
T: 587-777-9257