Hayden Walpole
U1210998
6/4/2024

Analysis Document — Assignment 4

1. My Partner for this assignment was Zijia (Harry) Xie, and they submitted our assignment on the GradeScope group.

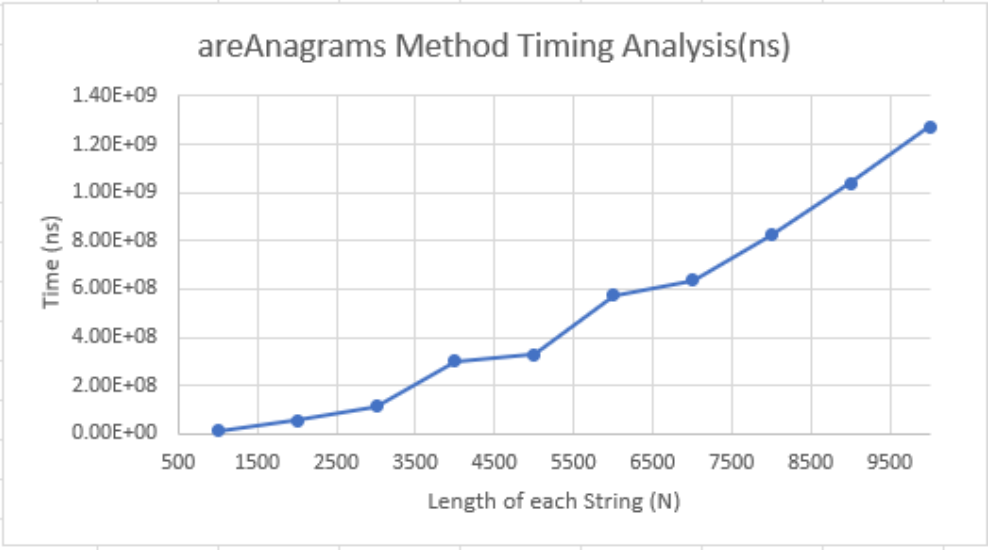
2. The pair programming experience went better than last time on this assignment. We wrote all the code together over the course of about 4 hours for the initial code, and 7 additional hours debugging and timing. We would try to switch roles every 40 minutes, but occasionally lost track of time. This time when we met I made sure I started as the driver on my machine and that helped split the coding roles more equally. I enjoy working with Harry, He is very good at java syntax and is very detail oriented, but we have very different approaches to problem solving. I have found he has a hard time being open to suggestions of ways to fix our code when I have an idea, but if I can write a simple code snippet or write my approach on paper so he can convince himself it will work, he is more willing to try it when he is the driver.

3. The sort method should not be void, and return a string instead because we don't want to modify the original string. By returning a string in sorted order we can compare two strings to see if they are anagrams, while keeping the original order of characters and the letter case in each original string. This allows us to show the strings in there original form without losing the format.

   The insertionSort method does the opposite, and is void instead of returning a new array of type T. This is because the only thing we are modifying about the array is the order of the strings in the array, without modifying any of the strings themselves. By having the method void we also are reducing memory by not creating another array every time we need to sort an array. We can then pass the original array which is now sorted into our other methods that require a sorted array.

4. **areAnagrams method:** For this timing experiment we used a value of 1,000 times to loop, starting at N=10,000 to N=100,000 increasing N in steps of 10,000 for 10 times. After looking at the code I expect this method to have a Big-O = O(N^2) because it has 2 calls to the sort method which has a call to insertionSort method. Insertion sort has two nested for loops that form a O(N^2) as they loop over length N, with N being the length of each string we are comparing in our timing analysis. Shown below in Graph 1 is the data collected from timing analysis and the accompanying graph. Upon inspection using an analysis check, we see the data is best represented by an O(N^2) which confirms what our hypothesis after looking at the code. It looks similar to O(N) in our sample, but we can see rapid growth of the R value in T(N)/O(N). We also see some jumps of higher values in our data at N=4,000 and N=6,000, these could be examined more in depth by increasing our times to loop, or continuing onto higher values of size N.

size(N)	time(ns)
1000	1.39E+07
2000	5.31E+07
3000	1.15E+08
4000	3.00E+08
5000	3.29E+08
6000	5.72E+08
7000	6.37E+08
8000	8.23E+08
9000	1.04E+09
10000	1.27E+09

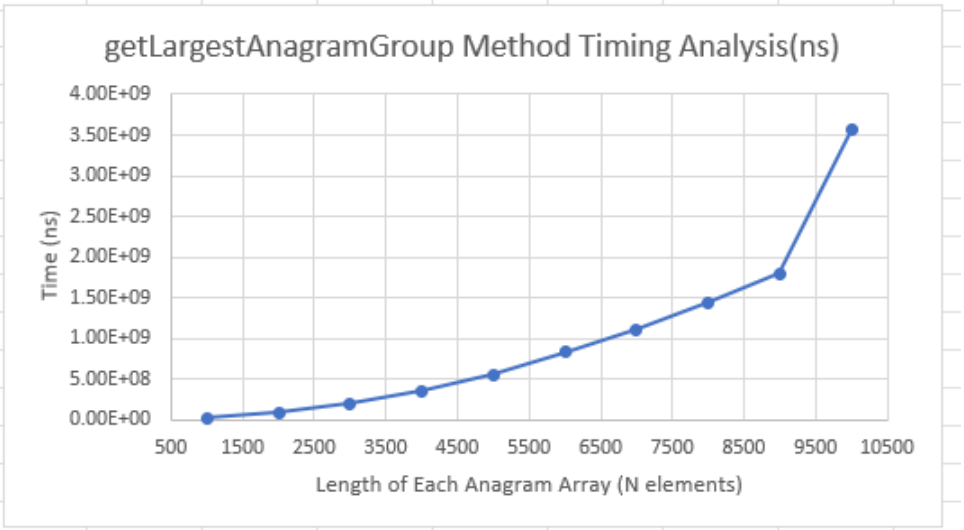


Graph 1

size(N)	T(N)/O(N)	T(N)/O(N^2)	T(N)/O(LogN)
1000	1.39E+04	1.39E+01	4.62E+06
2000	2.66E+04	1.33E+01	1.61E+07
3000	3.82E+04	1.27E+01	3.30E+07
4000	7.50E+04	1.87E+01	8.32E+07
5000	6.59E+04	1.32E+01	8.91E+07
6000	9.54E+04	1.59E+01	1.52E+08
7000	9.11E+04	1.30E+01	1.66E+08
8000	1.03E+05	1.29E+01	2.11E+08
9000	1.15E+05	1.28E+01	2.63E+08
10000	1.27E+05	1.27E+01	3.18E+08

5. **getLargestAnagramGroup method:** For this timing experiment we used a value of 1,000 times to loop starting at N=1,000 to N=10,000 increasing N in steps of 1,000 for 10 times. After looking at the code I expect this method to have a Big-O = O(N^2) because its longest component calls insertionSort to sort the array, and we know insertion sort typically has a timing of 0(N^2). The other loops in this method both are expected to be 0(N). Shown below in Graph 2 is the data collected from timing analysis and the accompanying graph. Upon inspection using an analysis check, we see the data is best represented by an T(N)/O(N^2) which confirms what our hypothesis is accurate prediction.

size(N)	time(ns)
1000	2.34E+07
2000	9.14E+07
3000	2.00E+08
4000	3.52E+08
5000	5.56E+08
6000	8.36E+08
7000	1.11E+09
8000	1.45E+09
9000	1.80E+09
10000	3.56E+09

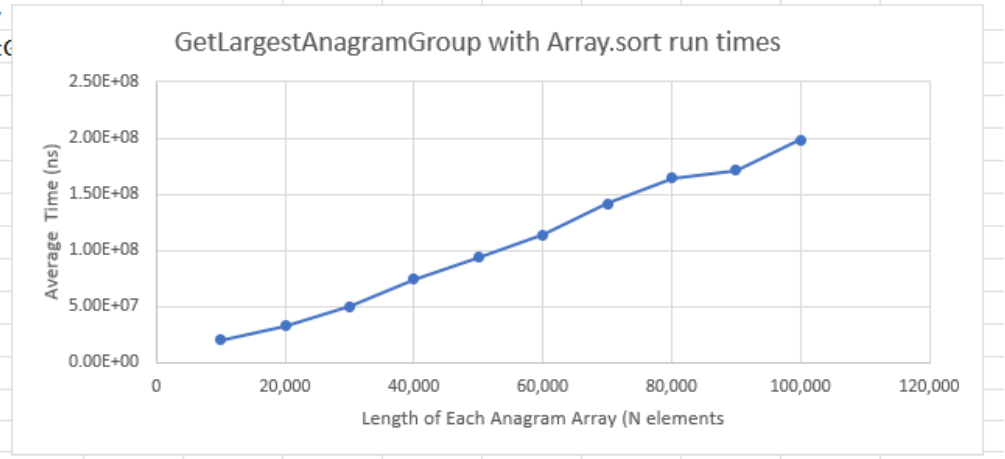


Graph 2

size(N)	T(N)/O(N)	T(N)/O(N^2)	T(N)/O(LogN	T(N)/O(N^3)
1000	2.34E+04	2.34E+01	7.80E+06	2.34E-02
2000	4.57E+04	2.28E+01	2.77E+07	1.14E-02
3000	6.67E+04	2.22E+01	5.76E+07	7.42E-03
4000	8.79E+04	2.20E+01	9.77E+07	5.50E-03
5000	1.11E+05	2.23E+01	1.50E+08	4.45E-03
6000	1.39E+05	2.32E+01	2.21E+08	3.87E-03
7000	1.58E+05	2.26E+01	2.88E+08	3.23E-03
8000	1.81E+05	2.26E+01	3.71E+08	2.83E-03
9000	2.00E+05	2.22E+01	4.55E+08	2.47E-03
10000	3.56E+05	3.56E+01	8.90E+08	3.56E-03

6. **getLargestAnagramGroups with Array.sort:** For this timing experiment we replace out the call of insertionSort with Array.Sort(inputArray), and we used a value of 1,000 times to loop starting at N=10,000 to N=1,000,000 increasing N in steps of 10,000 for 10 times. N in our experiment was the number of strings in the array. We can see from timing analysis that T(N)/O(N) is the best fit for the graph. By using javas sort method, the run-time performance decreased to O(N) instead, which is much faster than using insertion sort which from our previous experiment was found to have a run time of O(N^2). Shown below in Graph 3 is the data collected from timing analysis and the accompanying graph.

Get Largest Array	
size(N)	GetLargestG
10,000	2.04E+07
20,000	3.28E+07
30,000	5.00E+07
40,000	7.44E+07
50,000	9.36E+07
60,000	1.14E+08
70,000	1.41E+08
80,000	1.65E+08
90,000	1.71E+08
100,000	1.98E+08



Graph 3

size(N)	T(N)/O(N)	T(N)/O(N^2)	T(N)/O(LogN)
1000	2.04E+03	2.04E-01	5.09E+06
2000	1.64E+03	8.19E-02	7.62E+06
3000	1.67E+03	5.55E-02	1.12E+07
4000	1.86E+03	4.65E-02	1.62E+07
5000	1.87E+03	3.74E-02	1.99E+07
6000	1.89E+03	3.16E-02	2.38E+07
7000	2.02E+03	2.89E-02	2.92E+07
8000	2.06E+03	2.57E-02	3.36E+07
9000	1.90E+03	2.11E-02	3.45E+07
10000	1.98E+03	1.98E-02	3.96E+07