CS 201

Homework 2

Sec03

Kutay Tire

22001787

04.12.2021

TA: Aydamir Mirzayev

Instructor: Selim Aksoy

Question 1

Study the algorithms and understand their upper bounds. Describe how the complexity of each algorithm is calculated in your own words.

Algorithm 1: The upper bound of this algorithm is O(m * n) as stated. In other words, it can be written as $O(N^2)$ with big O notation. This algorithm uses linear search in order to compare the values between two arrays. In the algorithm, there are 2 nested for loops which are the main sources of calculations for time complexity. For the first loop, total running time is the running time of the statement inside the loop multiplied by n iterations. This is because the statement must be repeated n times to exit the loop which means that running time of the first loop is linearly correlated with size "n". Because the statement inside the first loop is another for loop with m iterations, total run time becomes n * m. For each value of n, the algorithm makes m iterations inside the second loop giving the final running time. Note that checking equality statements are all run in constant time O(1) and are much less than O(m * n).

Algorithm 2: The upper bound of this algorithm is O(m*logn) or O(N* logN) written in big O notation with single variable "N". In this algorithm, there is again a for loop with m iterations. This means that the total running time is going to be the number of iterations of the for loop, which is m, multiplied by the running time of the statement inside the loop. The statement inside the loop calls the binary search algorithm that uses recursion to find the specific value needed. For each call, it checks whether the value is less than the value in the middle index, equals to it or more than it. As a result of it, binary search algorithm has to be called on a sorted array. The binary search algorithm halves the indexes that it has to check in each recursive call. For instance, if the wanted value is more than the value at the middle index, it disregards the bottom half of the sorted array and checks the upper half. As a result, with each call, the array sized that needs to be checked is reduced to half. Therefore, the complexity becomes $\log_2 N$. Of course the best case scenario is O(1) meaning that the wanted value is at index middle. However, the generalization is logN as stated. Combined with

the for loop, total time complexity becomes O(m * logn) since for each unique m value, the binary search algorithm is called.

Algorithm 3: The upper bound of this algorithm is O(n+ m) or directly O(N) in single variable. For this algorithm, there is not any nested for loop, instead for loops follow each other when the other is finished. At first, the algorithm makes n iterations in order to find the maximum value in the first array. With this value, another array is constructed with the size of maximum value. Inside this first loop, there is an assignment statement which takes constant O(1) time and multiplying it with O(n) still gives O(n) as the running time. After that, each value in the newly constructed array is initialized to zero. However, the frequency of each unique element must be found to create a frequency table which takes another O(n) time. The run time for this is O(n) because the for loop iterates through n values and increases the frequency of each value present in the first array. After that, the algorithm makes m iterations for the second array and checks whether its elements are present in frequency array. If it is present, it reduces the frequency by one and again these simple instructions like subtraction take constant time O(1). Because these are consecutive statements, their run times must be added giving O(n) + O(m) or O(m+n).

Question 2

Report parameters of the computer that you used. Report RAM and Processor specifications in particular

The computer that was used to run the algorithms is ASUS VivaBook 15. Installed Physical Memory or RAM is 8 GB. However, the available RAM was 1.09 GB when the algorithms were run.

The CPU of the computer is Intel® Core ™ i5- 10300 CPU @2.50 GHz. Base speed is 2.50 GHz. It has 1 socket, 4 cores and 8 logical processors.

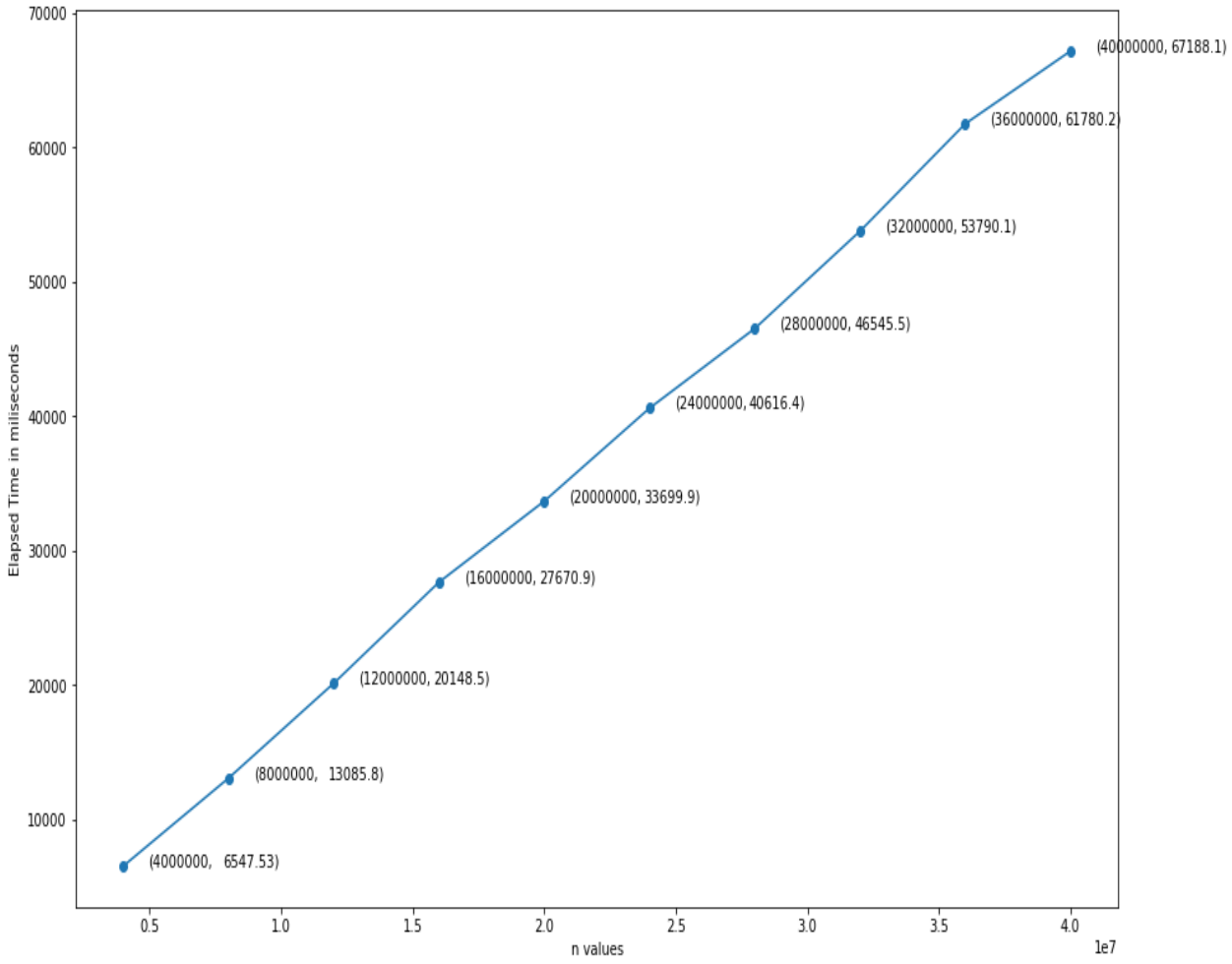
Runtime Table of Algorithms in Miliseconds

n	Algorithm 1		Algorithm 2		Algorithm 3	
	m = 1000	m = 10000	m = 1000	m = 10000	m = 1000	m = 10000
4 * 10^6	6547.53	68211.4	0.00224403	0.0223572	23.977	24.002
8 * 10^6	13085.8	134824	0.00225602	0.0225965	45.622	45.714
12 * 10^6	20148.5	203173	0.00225868	0.0225358	65.688	66.778
16 * 10^6	27670.9	268455	0.00225675	0.0225819	88.994	88.647
20 * 10^6	33699.9	341607	0.00225889	0.0226250	110.237	111.881
24 * 10^6	40616.4	405587	0.00226073	0.0225607	132.895	133.396
28 * 10^6	46545.5	472734	0.00225824	0.0226601	153.038	153.166
32 * 10 ^6	53790.1	538577	0.00226046	0.0225973	176.422	176.745
36 * 10^6	61780.2	593523	0.00225682	0.0226856	195.702	195.136
40 * 10^6	67188.1	659654	0.00225893	0.0226349	215.195	215.875

*The run-times are calculated in miliseconds.

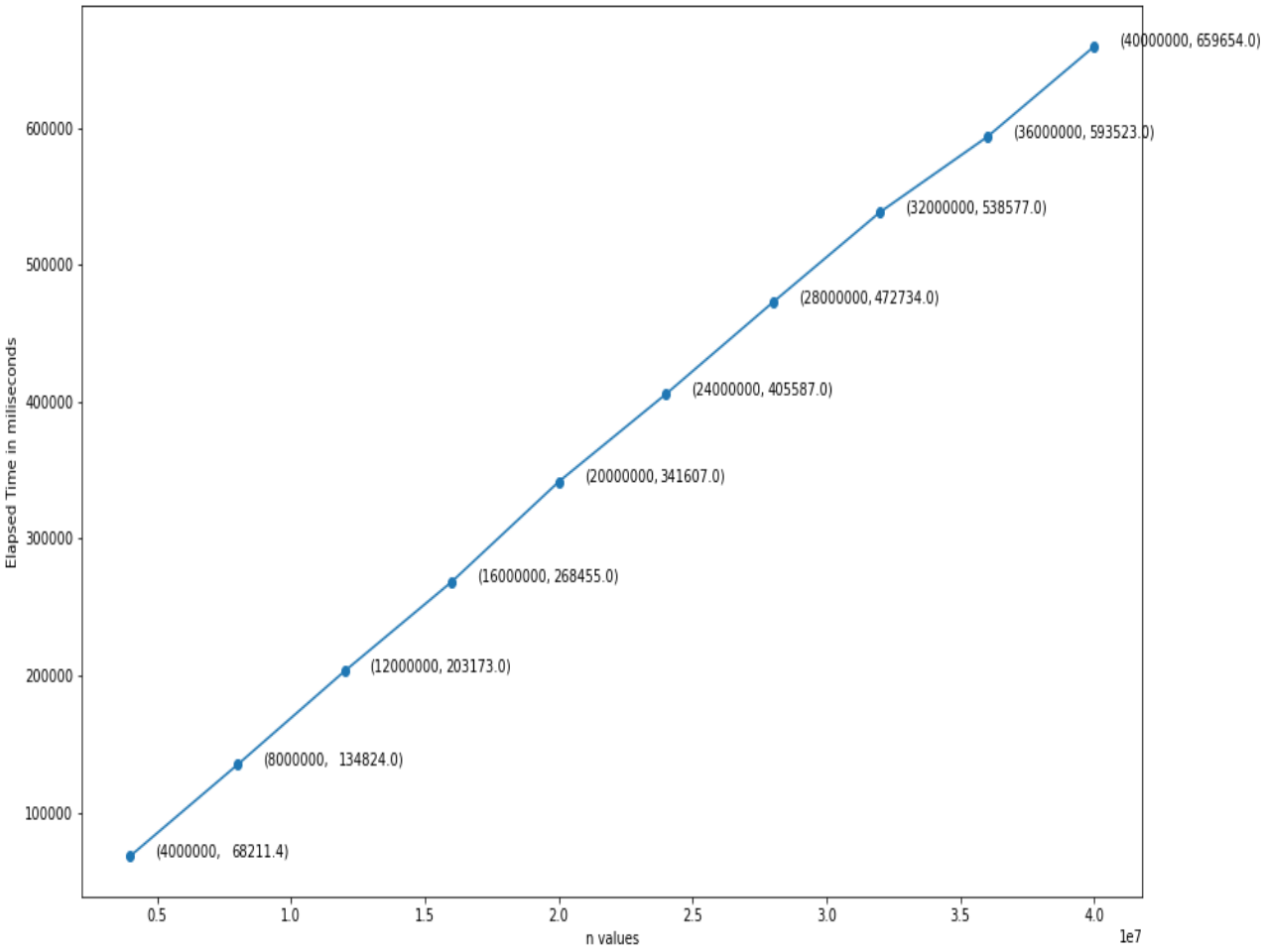
Plots of Time Complexities of Algorithms

Plot 1: Graph of Algorithm 1 with m = 1000



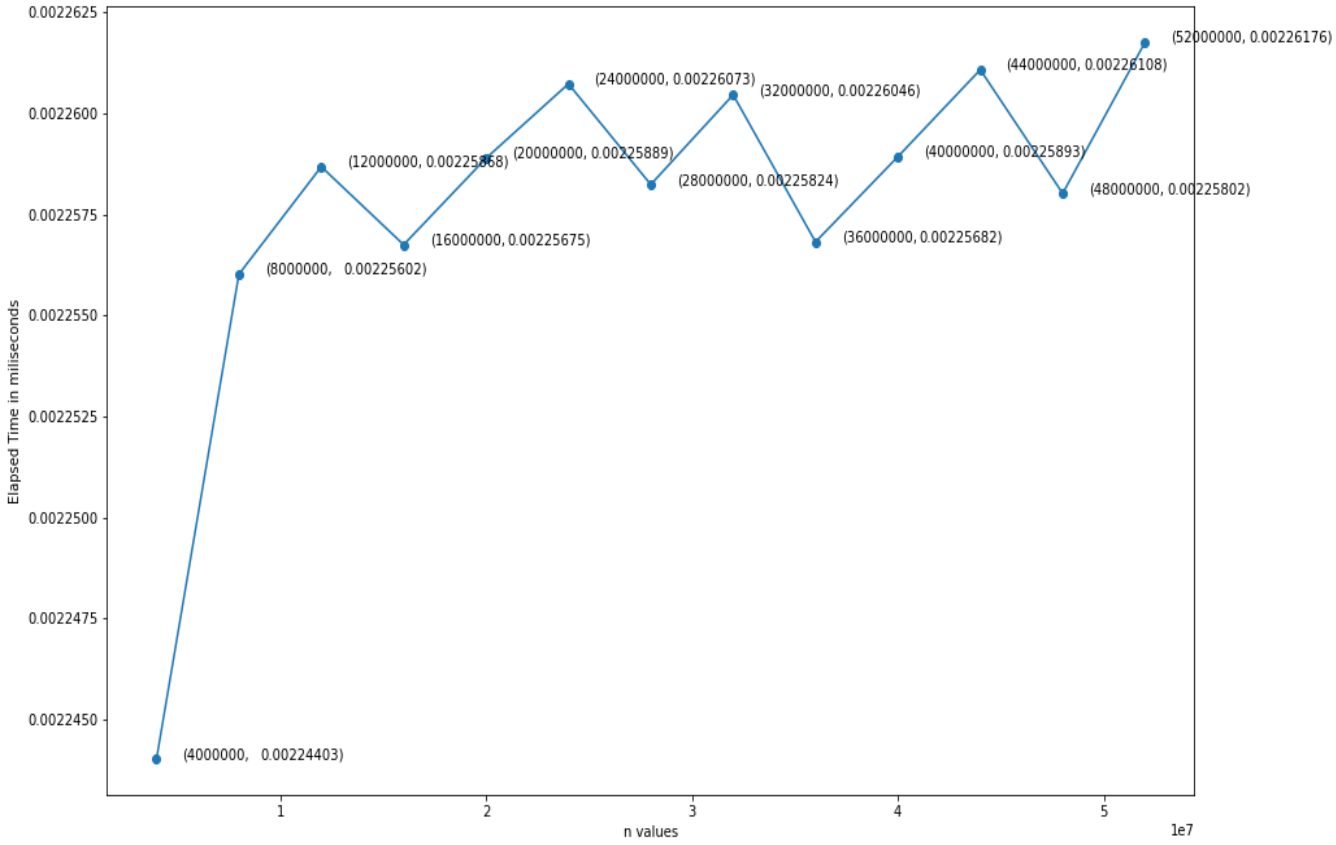
The run-time of the first algorithm is O(n*m). This means that with the m value 1000 acting as a constant, the graph becomes dependent on n. With very large n values, run time increases linearly as above. There are some possible fluctuations as the speed of computer may vary in each run. However, to minimize the error, the average of ten values was taken for small n values.

Plot 2: Graph of Algorithm 1 with m = 10000



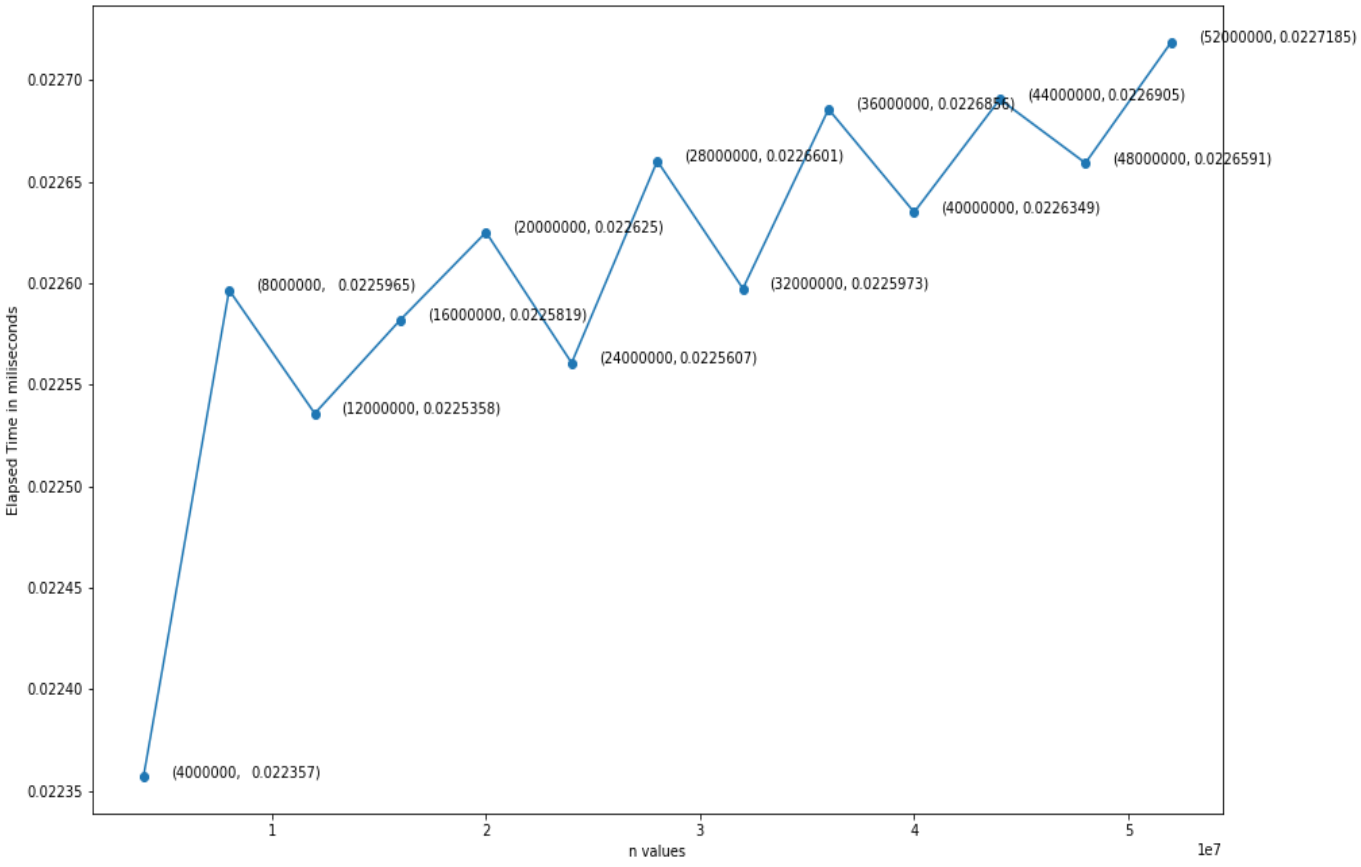
Again, since the firs algorithm is used, the run-time is O(n*m). However, the difference from the first plot is that the m value is 10000. This means that the run-time must be approximately 10 times bigger than the run-time value with m being 1000. As in the second plot, the elapsed time of each value is about ten times bigger than the elapsed times in the first plot.

Plot 3: Graph of Algorithm 2 with m = 1000



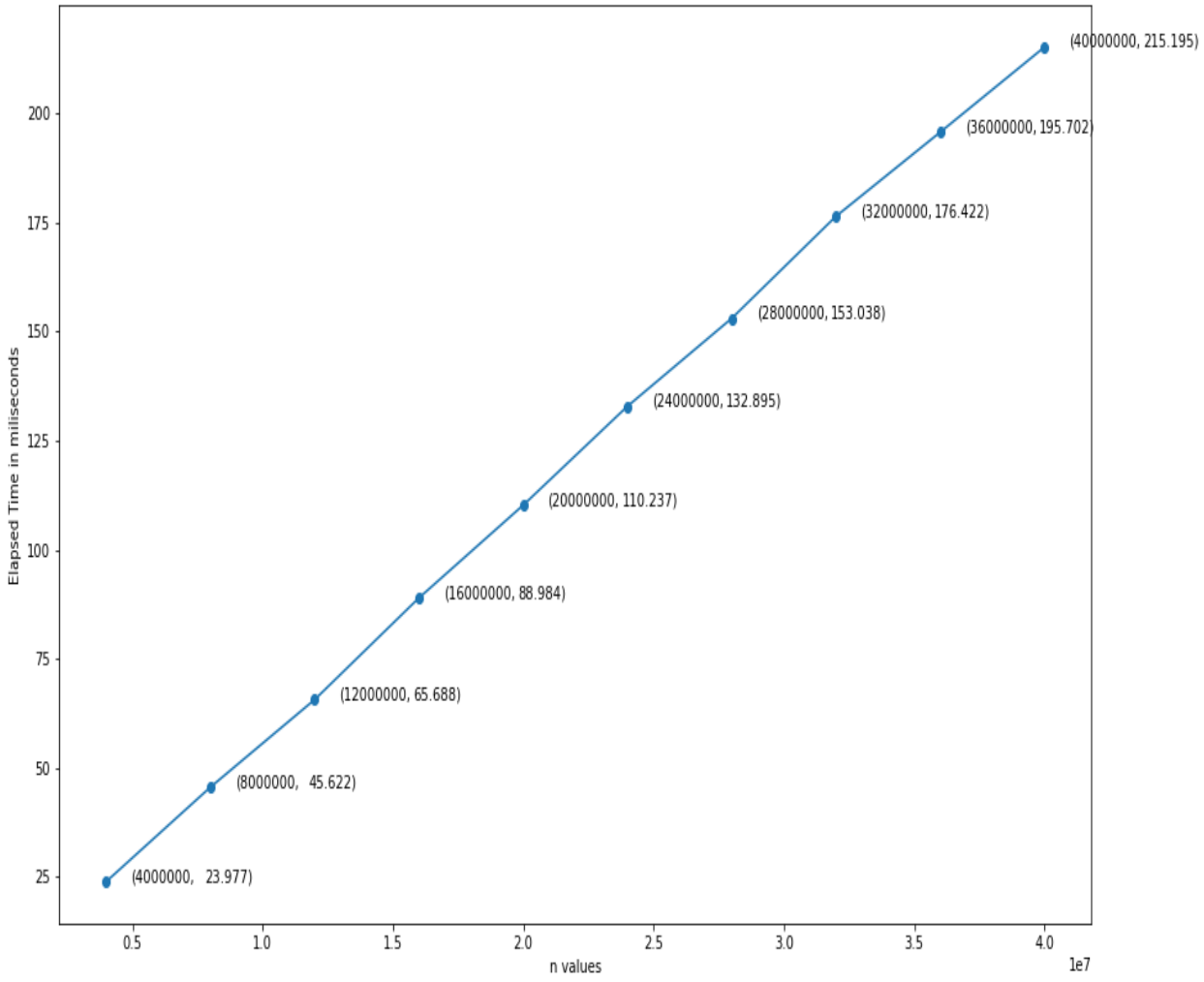
This graph belongs to the second algorithm with the run-time of O(m * logn). With fixed m value being 1000, the graph acts as a simple logN graph. Therefore, as it is seen, there is a sharp increase at first, but the rate of increase slows down gradually. Because the run-time of the second algorithm is too small, the algorithm had to be called over a million times in a for loop and the average was taken for a proper result. Again, although the graph isn't a directly logN graph, it shows a logarithmic trend. However, if more and more sample points are used, the graph becomes more logarithmic.

Plot 4: Graph of Algorithm 2 with m = 10000



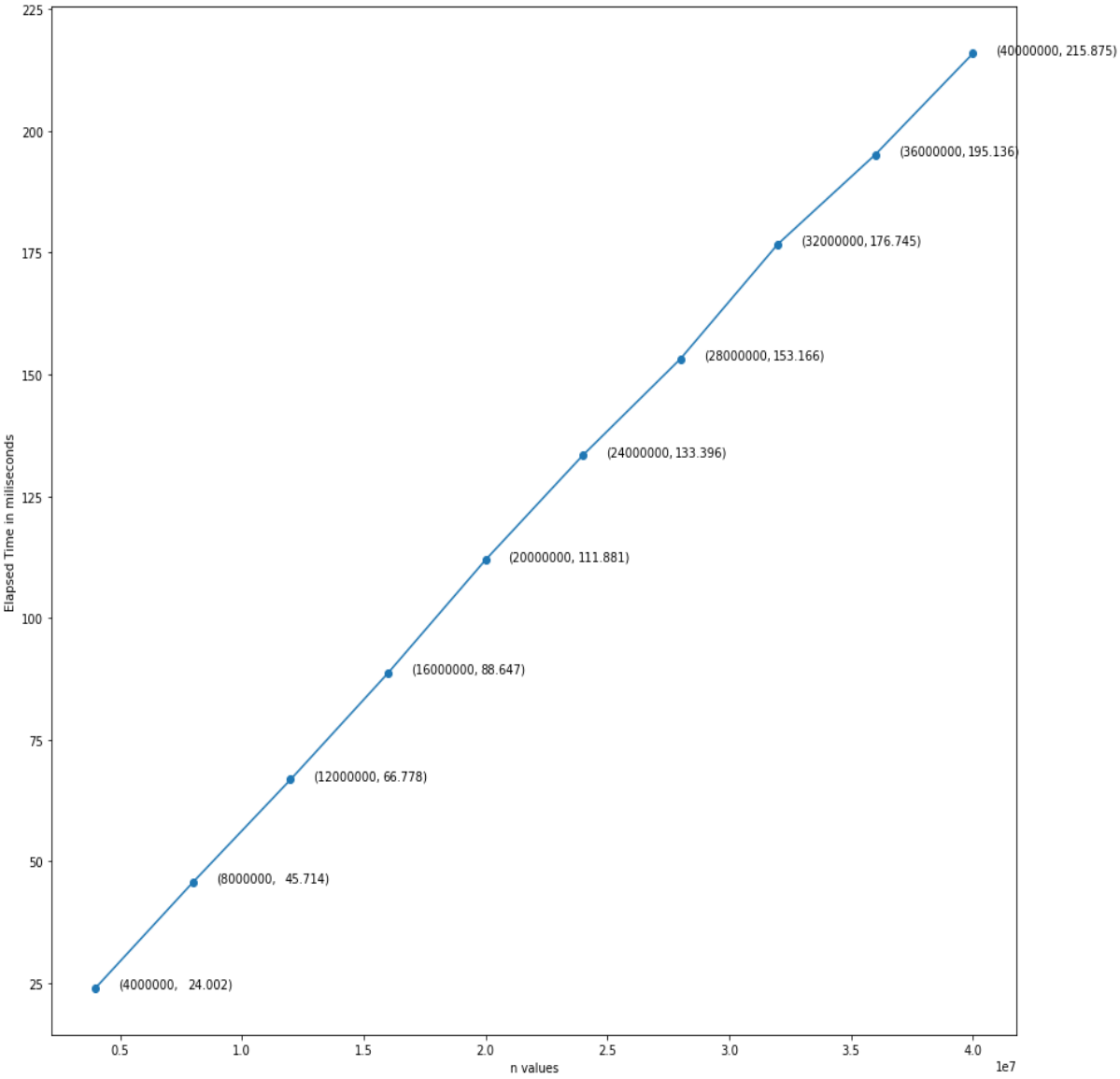
The graph of the second algorithm also resembles the third graph because they have the same time complexities. The only difference is that since the complexity is O(m* logn), with m being ten times bigger, the values are also ten times larger. Similar to third plot, it shows a logarithmic trend, but with small sample points, the graph has some ups and downs.

Plot 5: Graph of Algorithm 3 with m = 1000



This graph is for the algorithm 3 with m value being 1000. The time complexity was O(m + n) as explained in the first question. This means that with a constant m value, the graph has to show a linear trend similar to first algorithm. Since the run-times weren't especially large, the algorithm is called 10.000 times to get a correct result. Then, the average of these values were taken.

Plot 6: Graph of Algorithm 3 with m = 10000



Again the graph is linear because the time complexity was O(n + m). However, the important thing is that this time, m isn't multiplied by added instead. This is significant because for very large n values like 10^6, m is so small that it has little effect on the run-time. This is the reason why the values for graph 5 and graph 6 are so similar with the values with m being 10.000 a little bit larger.

Graph Codes:

Below is the Python code that was used to create the six plots above.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([4 * 10**6, 8 * 10**6, 12 * 10**6, 16 * 10**6, 20 * 10**6, 24 * 10**6, 28 * 10**6, 32 * 10**6, 36 * 10**6, 40 * 10**6, 44 * 10**6, 48 * 10**6, 52 * 10**6])

#y = np.array([6547.53, 13085.8, 20148.5, 27670.9,33699.9, 40616.4,
#46545.5, 53790.1, 61780.2, 67188.1 ])

#y = np.array([68211.4, 134824, 203173, 268455,341607, 405587,
#472734, 538577, 593523, 659654])

#y = np.array([0.00224403, 0.00225602, 0.00225868, 0.00225675, #0.00225889, 0.00226073,
# 0.00225824, 0.00226046, 0.00225682, 0.00225893, 0.00226108, #0.00225802, 0.00226176])

y = np.array([0.022357, 0.0225965, 0.0225358, 0.0225819, 0.022625, 0.0225607,
0.0226601, 0.0225973, 0.0226856, 0.0226349, 0.0226905, 0.0226591, 0.0227185] )
```

```
#y = np.array([23.977, 45.622, 65.688, 88.984, 110.237,  132.895,
 #153.038, 176.422, 195.702, 215.195 ])

#y = np.array([24.002, 45.714, 66.778, 88.647, 111.881,  133.396,
#153.166, 176.745, 195.136, 215.875 ])

fig = plt.figure()
ax = fig.add_subplot(111)

plt.rcParams["figure.figsize"] = (15, 10)

plt.plot(x,y,  marker = 'o')

plt.xlabel('n values')
plt.ylabel('Elapsed Time in miliseconds')

for i,j in zip(x,y):
    ax.annotate('%s)' %j, xy=(i,j), xytext=(80,0), textcoords='offset points')
    ax.annotate('(%s,' %i, xy=(i,j),xytext=(20,0), textcoords='offset points')

plt.show()
```