CS201 - 02

FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE

2021-2022 FALL SEMESTER

HOMEWORK ASSIGNMENT 2

ARDA TAVUSBAY

21902722

05.12.2021

DISCUSSION

1- Algorithm Using Linear Search

Linear search algorithm iterates through both arrays' elements and checks whether each element in the smaller array also exists in the bigger array. By going through each element in both arrays, time complexity scales dependent on both arrays' sizes. Thus, the general time complexity for this algorithm is O($n * m$). However, when comparing both arrays, if the first element in the smaller array is not existent in the bigger array, thus the smaller array is not a subset of the bigger array, the time complexity would be O(1). Similarly, if only the last element of the smaller array is not existent in the bigger array, program would iterate through every element of both arrays, thus making the complexity O($n * m$). Considering the average case, where a value from the smaller array is not existent in the bigger array is in the middle of the smaller array, $\frac{n*m}{2}$ amount of iterations would be needed and the time complexity would be O($n * m$).

2- Algorithm Using Sorted Arrays and Binary Search

Binary search algorithm repeatedly divides the array from its center until the expected value is found or there are no more elements left to check in the array. If the expected value is less than the element in the middle of the array, then the array is narrowed down to only its center to its left-most value. As the exact opposite, if the expected value is bigger than the element in the center of the array, then the array gets narrowed down from its center to its right-most value. Since with each iteration the size of the array gets halved, after k amount of iterations, the new size of the array would consist of only one element. Since the initial size of the array was n and after k iterations the new size became 1, we see that $\frac{n}{2^k} = 1$. Thus, $k = log_2 n$, and the time complexity is O($logn$). In addition, since we use this algorithm for each element in the array with size m to check whether the elements also exist in the bigger array, total time complexity of the program would be O($m * logn$). Considering the best case, if the target value is not found in the first iteration, and thus the smaller array is not a subset of the bigger array, time complexity would be O(1). Similarly, if the last element in the smaller array is not

existent in the bigger array, $m * logn$ amount of iterations would be made and for both average and worst cases, the time complexity would be O($m * logn$).

3- Algorithm Using Frequency Tables

By using a frequency table for storing the occurrences of each element in the first array, we can compare the elements of the second array with the frequency table. In order to create the frequency table, we need to use an array with the size of the largest number in the first array. Then by using linear iteration, we increment the values at specific indexes in the frequency table according to the values in the first array. Time complexity of linear iteration for initializing the frequency table with the values taken from the first array is O(n) since n (size of the first array) amount of iteration for n amount of increment will be made. Afterwards, by traversing the second array and comparing every element to their corresponding index in the frequency table, we decrement the value if it's larger than 1, meaning that the element in the second array is also existent in the first array. If the value in the frequency table that corresponds to an element in the second array is 0, it means that for the element in the second array, there are no occurrences in the first array. Therefore, by traversing the second array, m amount of iterations are made and the time complexity is O(m). Combining both time complexities, O(n) from iterating through the first array and O(m) from traversing the second array, we see that the total time complexity of the algorithm is O(m + n).

COMPUTER SPECIFICATIONS

Processor: 1.4 GHz Quad Core Intel Core i5

RAM: 8 GB 2133 MHz LPDDR3

Operating System: macOS Catalina 10.15.7

RESULTS

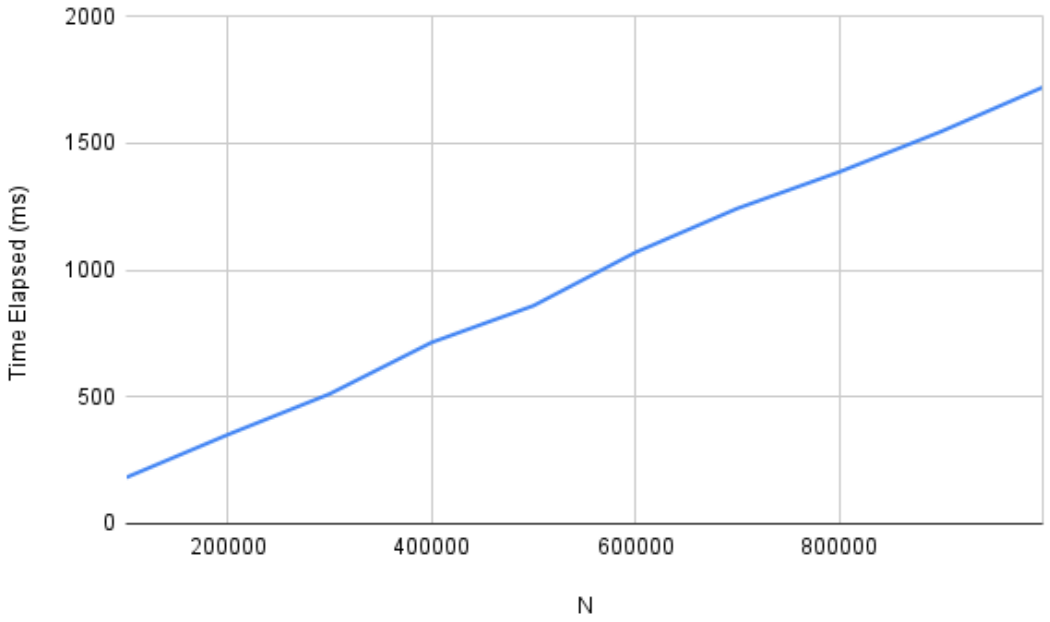
Execution Time vs Array Sizes & Algorithms Table

n	Algorithm 1		Algorithm 2		Algorithm 3	
	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$
10^5	181.826	1710.89	0.3796	4.057	1.604	1.698
$2 * 10^5$	351.046	3585.18	0.4073	4.254	3.212	3.632
$3 * 10^5$	511.602	5274.61	0.4206	4.39	4.935	4.805
$4 * 10^5$	714.319	7045.86	0.4293	4.47	5.885	6.162
$5 * 10^5$	859.064	8799.81	0.4365	4.537	6.845	6.892
$6 * 10^5$	1068.87	10583.9	0.4404	4.582	8.208	7.515
$7 * 10^5$	1242.03	12222.8	0.4452	4.636	9.232	8.549
$8 * 10^5$	1385.68	13948.4	0.4538	4.671	10.76	9.358
$9 * 10^5$	1545.02	15643.5	0.4584	4.708	11.648	10.578
10^6	1719.8	17069.2	0.456	4.738	12.766	11.695

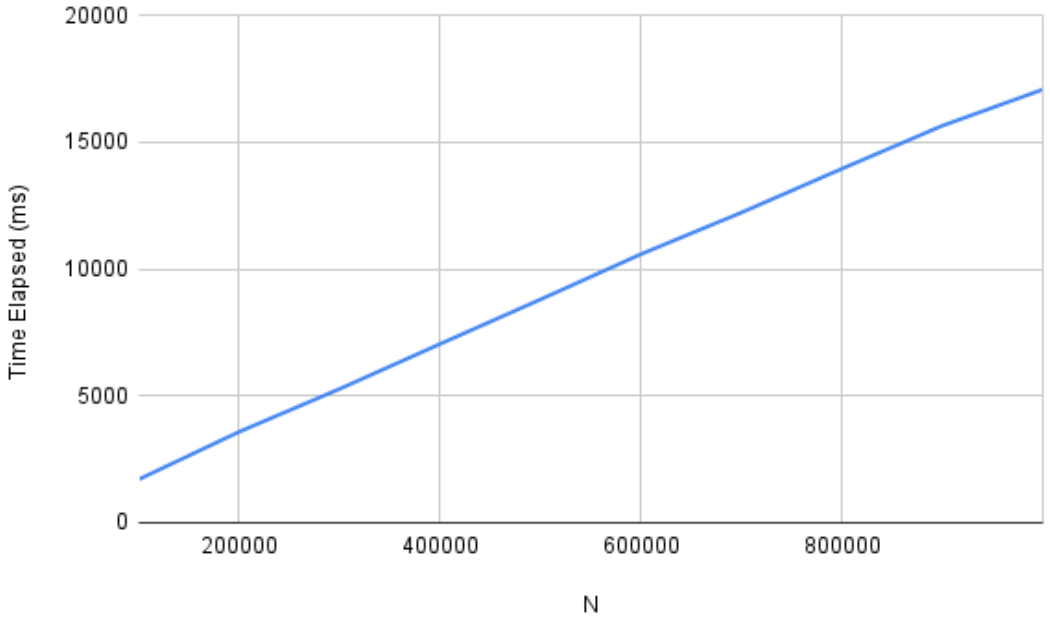
(Table of comparing different sizes and algorithms with their corresponding runtime in milliseconds)

1- Algorithm Using Linear Search

First, the value m is constant and equals to 10^3. Since m is constant and the time complexity is described as O($n * m$), we can see that the total time complexity scales with value n. Thus, the graph increases linearly according to the value n. (Graph below shows the first algorithm with $m = 10^3$).

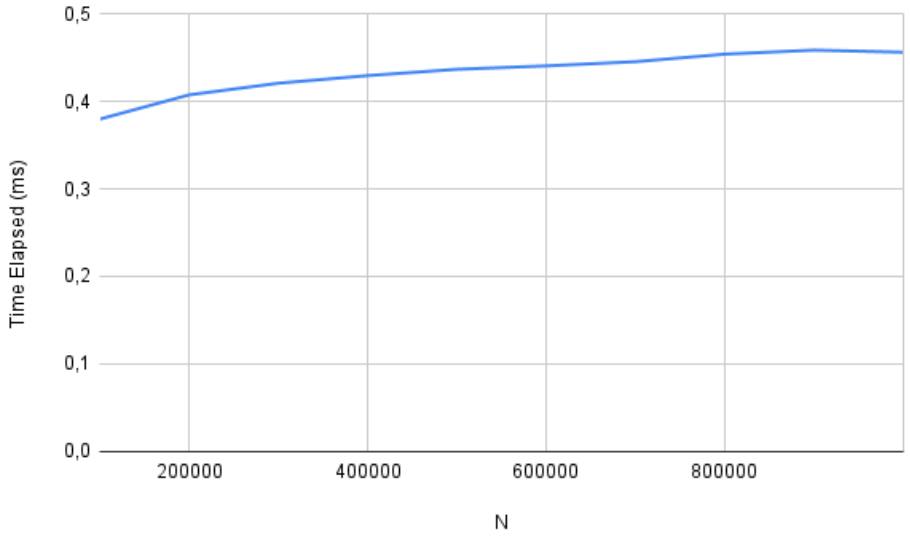


We then change the value m to 10^4 but the value m is still constant. Even though we changed the value m, it is still constant and the total time complexity still depends on the value n. (Graph below shows the first algorithm with $m = 10^4$).

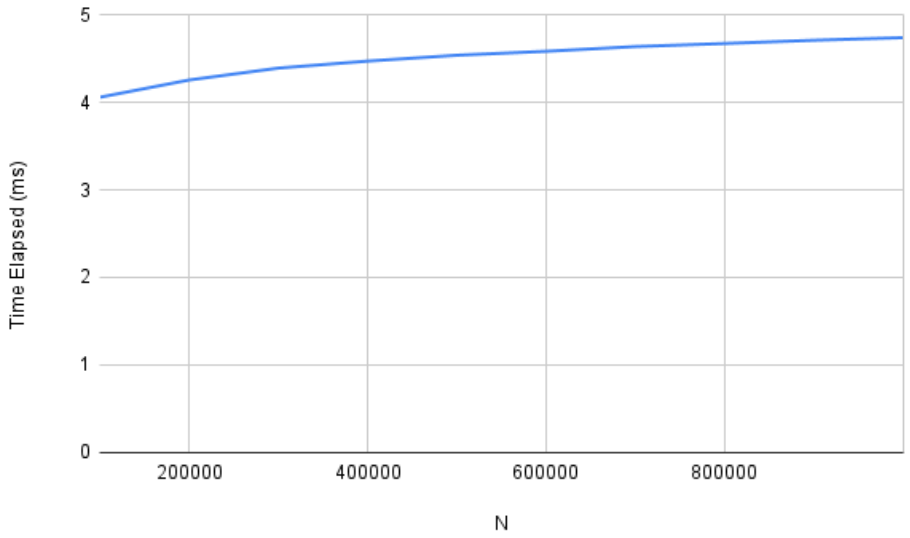


2- Algorithm Using Sorted Arrays and Binary Search

Time complexity for this algorithm is O($m * logn$) as mentioned before.By setting m to a constant value 10^3, total time complexity of the algorithm becomes dependent on the $logn$. Thus the graph will increase in the form of a $logn$ graph. Since the initial values are chosen as already big values, the interval is similar to the $logn$ graph with large x values (Graph below shows the second algorithm with $m = 10^3$).

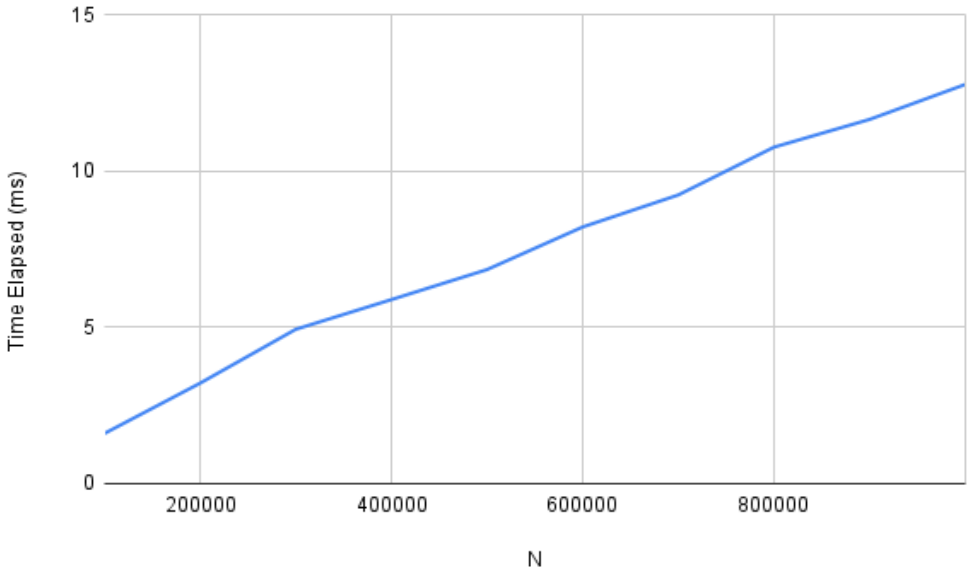


We then change the value m to 10^4 but the value m is still constant. Even though we changed the value m, it is still constant and the total time complexity still depends on the value n. Only difference now is that since m is increased, $m * logn$ and total time elapsed will also increase. (Graph below shows the second algorithm with $m = 10^4$).



3- Algorithm Using Frequency Tables

By using frequency tables, we use more memory in exchange for shortening the total time elapsed for running the program. By iterating through both the first and second array for once, we see that the time complexity depends on the sizes of these arrays which are values m and n. Thus, the total time complexity of the program is O(m + n). Therefore, the time graph should be dependent on the values m and n. Since m is constant, the graph will increase linearly as n increases. (Graph below shows the third algorithm with $m = 10^3$).



Since time complexity for this algorithm is O(m + n), by increasing the value m to 10^4, total time elapsed until the start of the program will also increase. (Graph below shows the third algorithm with $m = 10^4$).