CS201 Homework 2

Section: 1

Name: Barkın Saday

ID: 21902967

	Algorithm 1		Algorithm 2		Algorithm 3	
	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$
$n = 10^6$	1812 ms	17566 ms	0.64 ms	3.76 ms	3.1 ms	3.9 ms
$n = 2*10^6$	3661 ms	37226 ms	0.94 ms	4.38 ms	6.3 ms	6.3 ms
$n = 3*10^6$	5551 ms	55435 ms	0.62 ms	3.12 ms	7.8 ms	9.4 ms
$n = 4*10^6$	7563 ms	74460 ms	0.94 ms	2.98 ms	11 ms	14 ms
$n = 5*10^6$	9347 ms	93480 ms	0.62 ms	2.82 ms	12.5 ms	15.7 ms
$n = 6*10^6$	11213 ms	112442 ms	0.92 ms	3.6 ms	18.7 ms	18.6 ms
$n = 7*10^6$	13171 ms	142706 ms	0.64 ms	3.96 ms	20.3 ms	21.2 ms
$n = 8* 10^6$	15029 ms	170563 ms	0.62 ms	2.5 ms	21.9 ms	24. 9 ms
$n = 9*10^6$	16952 ms	174570 ms	0.62 ms	2.82 ms	23.5 ms	23.9 ms
$n = 10^7$	18801 ms	196902 ms	0.62 ms	3.12 ms	26.9 ms	29.7 ms

Parameters Of The Computer:

-For this homework, I executed the program in my personal computer using the IDE CodeBlocks.

- I measured the time by using “ctime”. Since algorithm 3 and especially algorithm 2 is working very fast, in order to have a more precise measurement I called algorithm3 10 times and called algorithm2 50 times. Then I divided the measured time to those numbers.

-Processor: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz

-RAM: 8GB

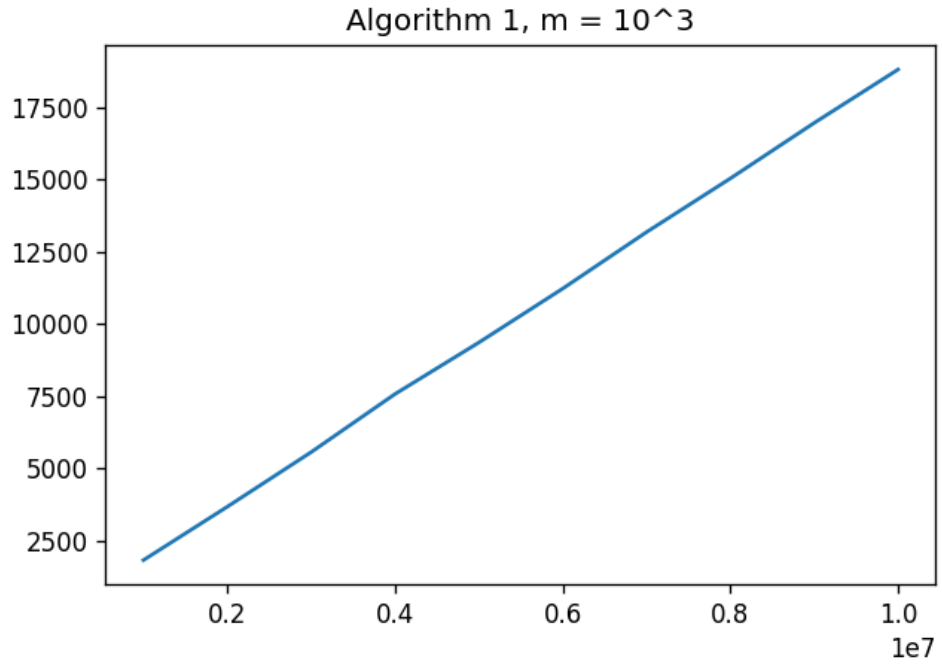
-System Type: x64 based processor

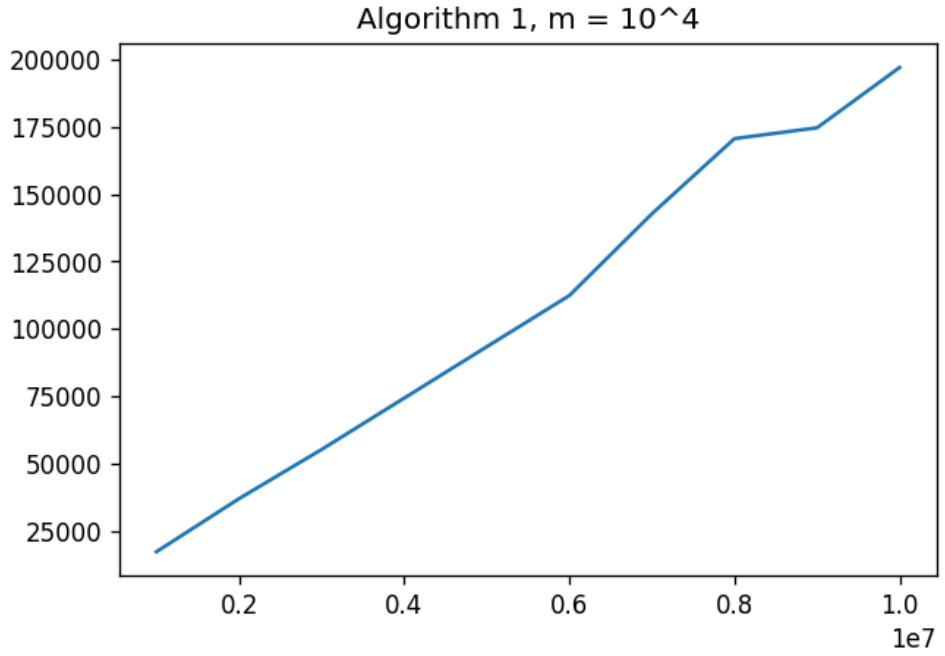
Initializing Arrays:

-For testing the algorithms, I used two arrays of size n and m with both of them containing integers from 1 to its size so both arrays are sorted and contain unique items. (Array 1 must be sorted for algorithm 2 and array 2 must contain unique items for algorithm 1).

Note about graphs: The x-axis represents the input n in 1e7 notation (x * 10^7), the y-axis represents the elapsed time (execution time for that algorithm) in milliseconds.

For Algorithm 1:





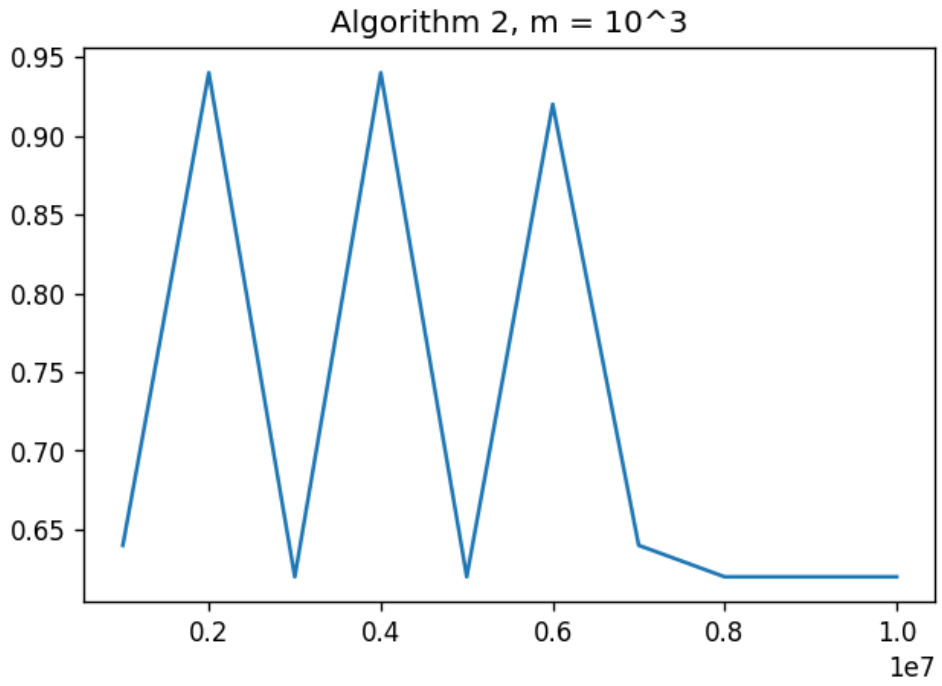
-How the time complexity is calculated?

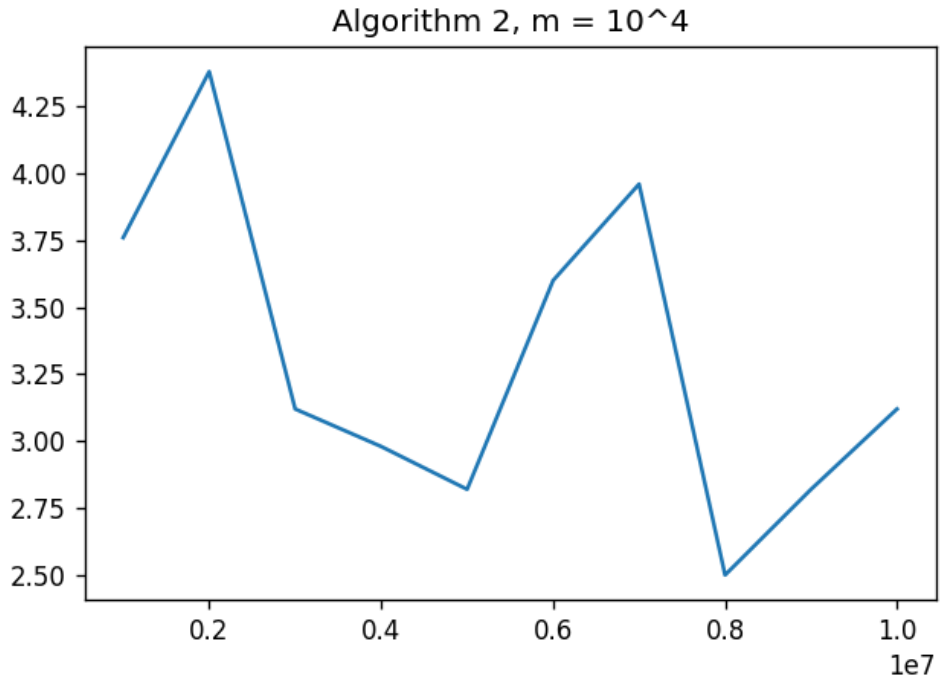
-We compare each item of array 2 with each item of array 1. We use two loops that are nested. The inner loop is executed each time we execute the outer loop. We enter the outer loop m many times and in each of these we enter the inner loop n many times. So basically we enter the inner loop (where we check if the elements of two arrays are equal) m*n many times.

Observing table chart and graphs:

Algorithm 1 performs the slowest in terms of execution time when compared to algorithm 2 and algorithm 3. Since we use a nested loop and iterate through all elements of array 1 (n many items) for each element of array 2 (m many times) the time complexity for algorithm 1 is O(n*m). From the table chart, we can see both when we increase n or m the execution time increases in almost a linear way which proves our theory. When m = 10^3 we increase n by 10^6 at each step and execution time increases about 2 seconds at each step (about 20 seconds when m = 10^4). Also for the same n value if we change m to 10^4 the exectution takes about 10 times longer to perform (1.8 seconds to 17.5 seconds for example). Also, we can see that for the largest array sizes that we use (n = 10^7, m = 10^4) the execution time is longer than 3 minutes. This shows for big inputs algorithm 1 works poorly.

For Algorithm 2:





-How the time complexity is calculated?

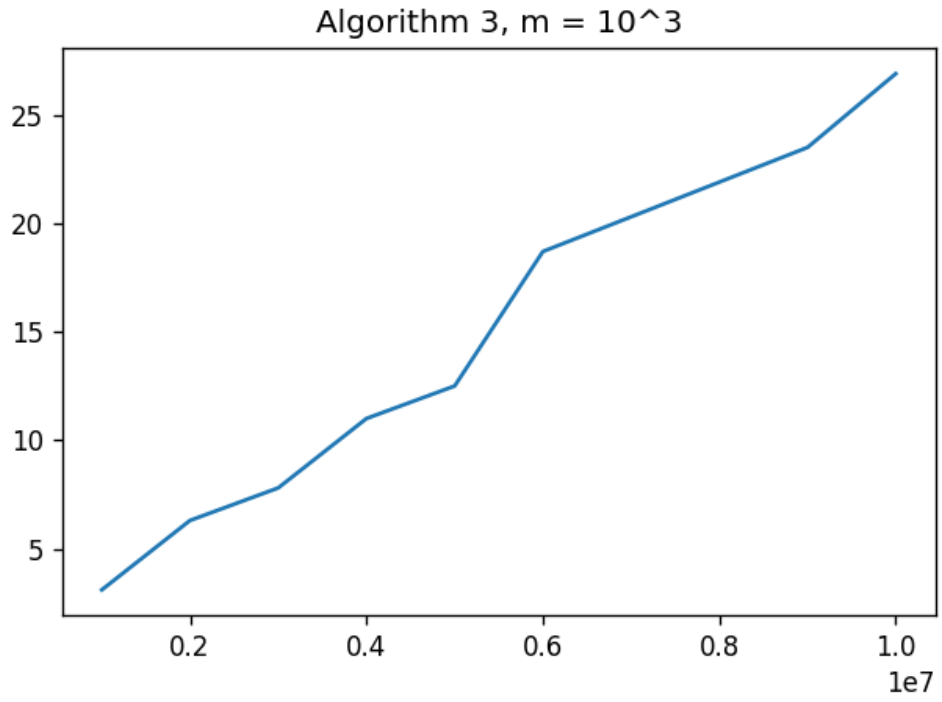
-For each element of array 2, we search for that element in array 1 by binary search. This means we do a binary search m many times. The binary search takes a sorted array (sorting is done outside of the method) and checks if the middle item of the array is the item we are looking for, if it is the middle item of the array then returns that index. If it is not we split the array into two parts and keep working with only one of the parts (the part we keep working with is determined whether the value we are searching for is smaller or greater than the middle item). To work with this new part we recursively call binarySearch function (this time for a part of the array). In this new call, the middle value is updated so we check again. Basically, we split the array into half in each check until we find the element or there is only one element left (which means that element does not exist and it is the worse case). So binary search takes logn time for an array of size n. Since we call binary search m many times time complexity for algorithm 2 is O(m*logn).

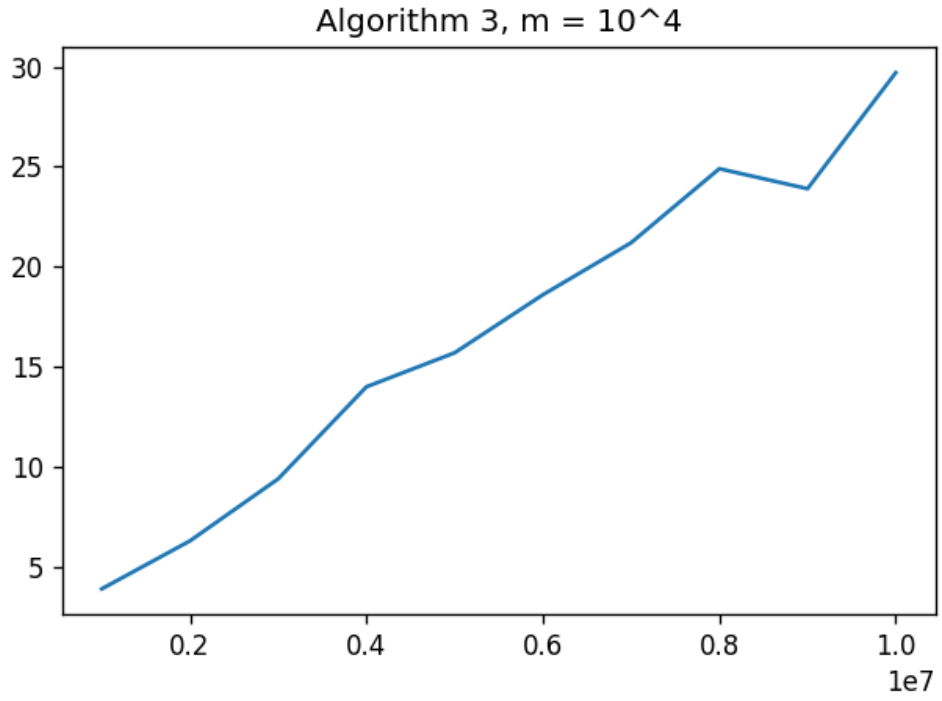
Observing table chart and graphs:

While we increase n in each step algorithm 1 and algorithm 3 have greater execution times. However, when we increase n, algorithm 2 does not increase in each step. In fact, sometimes it takes a longer execution time for a smaller value. For example when $n = 6*10^6$, $m = 10^3$ the execution time for algorithm 2 is 0.92 ms and when we increase the input size n to $7*10^6$ the execution time is 0.64 ms. The reason for that is algorithm has time complexity O(m logn) so increasing n does not affect the execution time as much as algorithm 1 or algorithm 3. Also since the time of execution is very short for each case, the differences in execution time are mostly caused by the CPU's current performance (performance while executing the program) which changes at each execution due to other operations happening (which are unrelated to our cpp file) at execution time. This is probably why algorithm 2 does not take a longer execution duration each time we increase the input n. However, when we increase m to

10^4 the duration definitely increases (from 0.64 ms to 3.76 ms for example while n = 10^6) because increasing m affects the execution time more than increasing n for algorithm 2.

For Algorithm 3:





-How the time complexity is calculated?

- Here we create a new array which is called a frequency table (outside of the method to observe better). Than inside the method; for this new array, each element of array 1 is used as an index. The element with that index of the frequency table is assigned to 1 (since both array 1 and array 2 have only unique items we can just assign the necessary elements to 1 instead of counting them) For this we use a loop which is executed n many times. Then we use another loop which executes m many times. In each execution, it checks whether the elements in the frequency table with the index of each element in array 2 is equal to 0 or not. If any element is equal to 0 it means array 2 is not a subset and returns false. Otherwise (in a successfull search) we iterate through the loop to the end then we return true. Since we perform an execution n many times and than another execution m many times the time complexity for algorithm 3 is O(n+m).

Observing table chart and graphs:

Algorithm 3 has two separate loops (not nested). One of the loops iterates n many times while the other loops iterates m many times so the time complexity is O(n+m). The frequency table is created outside of the method and used as a pass-by-value parameter (to observe the time complexity better). For the same m, if we increase n by 10^6 in each step, we can see that execution time also gets longer at each step. Unlike algorithm 1, since algorithm 3 is still very fast (at most 29.7 ms with our largest input size) the increase rate is not that consistent. The reason for that is similar to the reason we discussed for algorithm 2 (CPU's current performance). So basically, since our measurements are not that precise and are affected by external factors (especially for algorithm 2 and algorithm 3) our table and graphs are different (for some atributes) from what we expected in theory. Still, even if it is not coherent execution time for algorithm 3 increases in each step which we increase n (unlike what we measured for algorithm 2). If we increase m execution time also increases but it is not always observable this time (due to external factors). The reason for that is we increase n by 10^6 in each step while we increase m by $9*10^3$ which is a smaller number so execution time is less affected compared to increasing n. In this case, external factors have a greater effect on execution time so there are some measurement errors (for example execution time goes from 18.7ms to 18.6 ms when we increase m).