Berk Çakar – 22003021

CS201, Fall 2021

Homework Assignment 2 - Report

Questions

1) **How the complexity of each algorithm is calculated?**

   In a more straightforward and generalized way, the upper bound of an algorithm is defined as the most amount of time required (the worst-case time complexity) which guarantees the function to execute. This concept is represented with the Big-Oh notation. For algorithm 1, in order to detect given `arr2` is a subset of `arr1` or not, each element of `arr2` (size m) will be selected in the outer for loop, and for each selected element, the inner for loop will perform a linear search in `arr1` (size n). These steps show that the upper bound of algorithm 1, and hence the time complexity, will be O(n*m) in the worst-case. Algorithm 2 is similar to the first algorithm in terms of operating logic. However, instead of performing a linear search at `arr1` (size n) in the inner loop for each `arr2` (size m) element selected in the outer for loop, binary search takes the place of the linear search. Hence, the time complexity will turn into an order of logn (time complexity of the binary search algorithm), such that O(m*logn). In Algorithm 3, the values in `arr1` (size n) will be mapped to the indexes of an array named `freqTable` in a for loop (i.e. `arr1[i] = k; freqTable[k]++;`). In another discrete for loop, `arr2` (size m) will be traversed from start to finish, and each value in `arr2` will be searched linearly in the `freqTable`. Since an array of size m and an array of size n are traversed in separated for loops, the upper bound of this algorithm, in other words the worst-case time complexity, will be O(n+m).

2) **Computer Specification:**

   Operating System: macOS Catalina 10.15.7 19H1217 x86_64

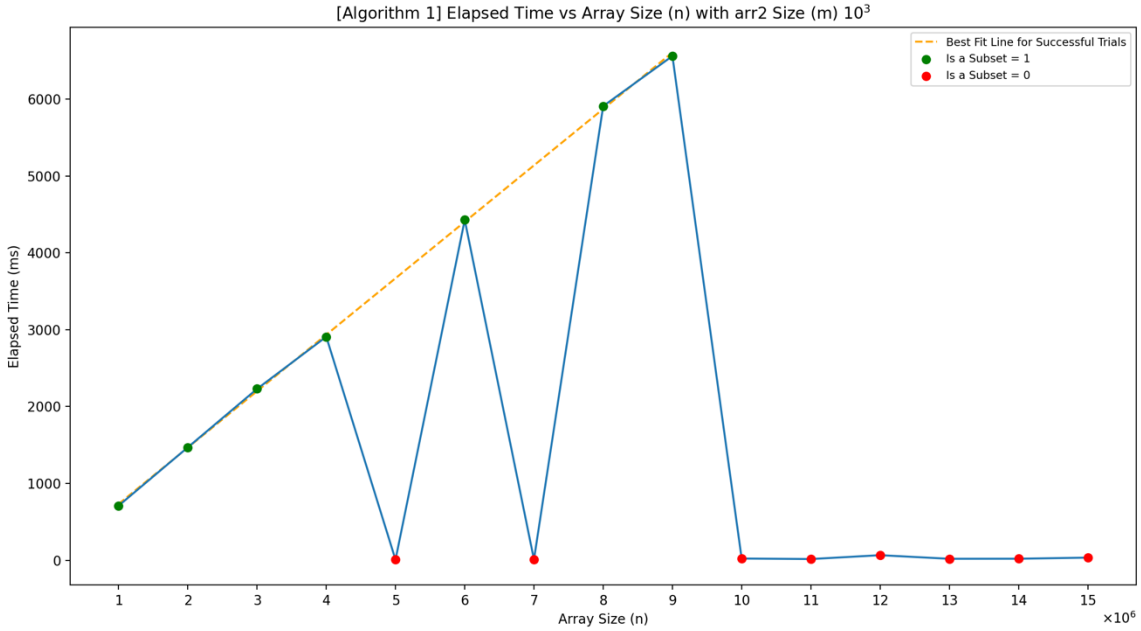
   Processor: Intel i5-1038NG7 (4 Cores/8 Threads) @ 2.00GHz

   RAM: 16 GB 3733 MHz LPDDR4X

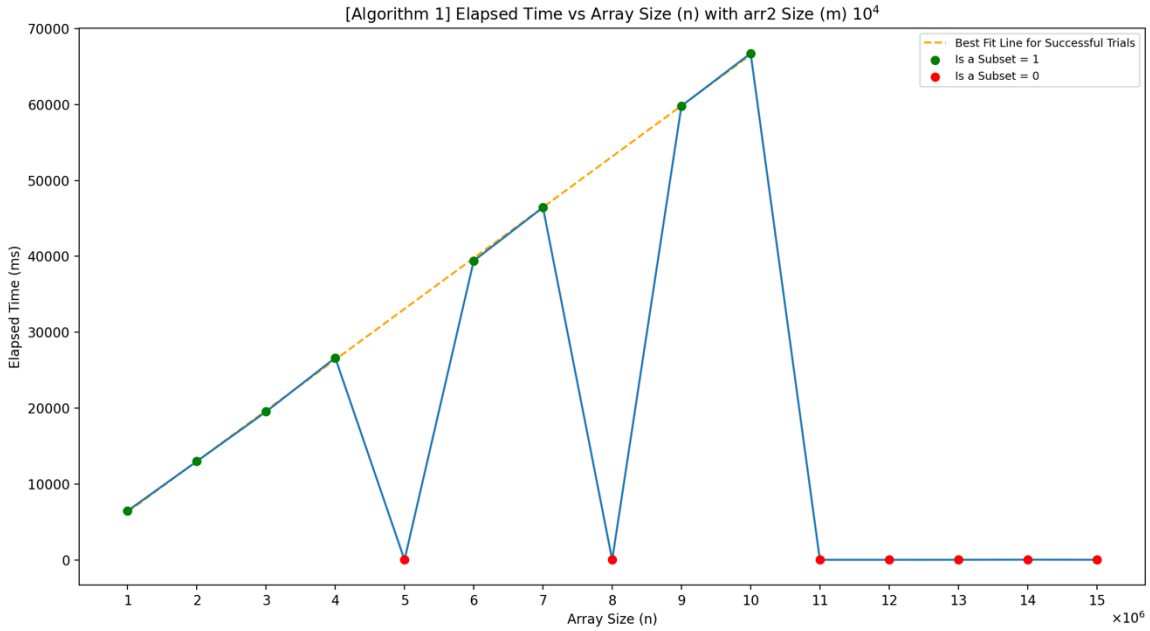
Table

n	Elapsed Time in Algorithm 1 (ms)		Elapsed Time in Algorithm 2 (ms)		Elapsed Time in Algorithm 3 (ms)	
	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$
$1 * 10^6$	709.791	6465.74	0.425	4.424	6.251	6.308
$2 * 10^6$	1469.64	12992.2	0.699	5.624	7.744	8.208
$3 * 10^6$	2229.92	19532.9	0.821	6.351	10.216	10.107
$4 * 10^6$	2908.27	26631.2	0.898	6.885	12.042	11.435
$5 * 10^6$	7.314	8.14	0.002	0.004	13.314	12.9
$6 * 10^6$	4427.7	39408.9	0.972	7.807	16.28	15.087
$7 * 10^6$	10.329	46445.5	0.005	8.203	16.909	16.903
$8 * 10^6$	5908.32	10.365	0.991	0.004	22.061	18.052
$9 * 10^6$	6560	59804.6	0.998	8.487	22.546	20.606
$10 * 10^6$	22.875	66744.6	0.007	8.634	24.217	22.62
$11 * 10^6$	16.27	15.73	0.003	0.003	27.22	26.344
$12 * 10^6$	66.521	15.369	0.013	0.003	29.96	27.245
$13 * 10^6$	19.563	17.59	0.005	0.004	31.684	30.864
$14 * 10^6$	20.357	34.294	0.004	0.008	34.018	32.75
$15 * 10^6$	34.842	20.96	0.008	0.006	36.738	34.12

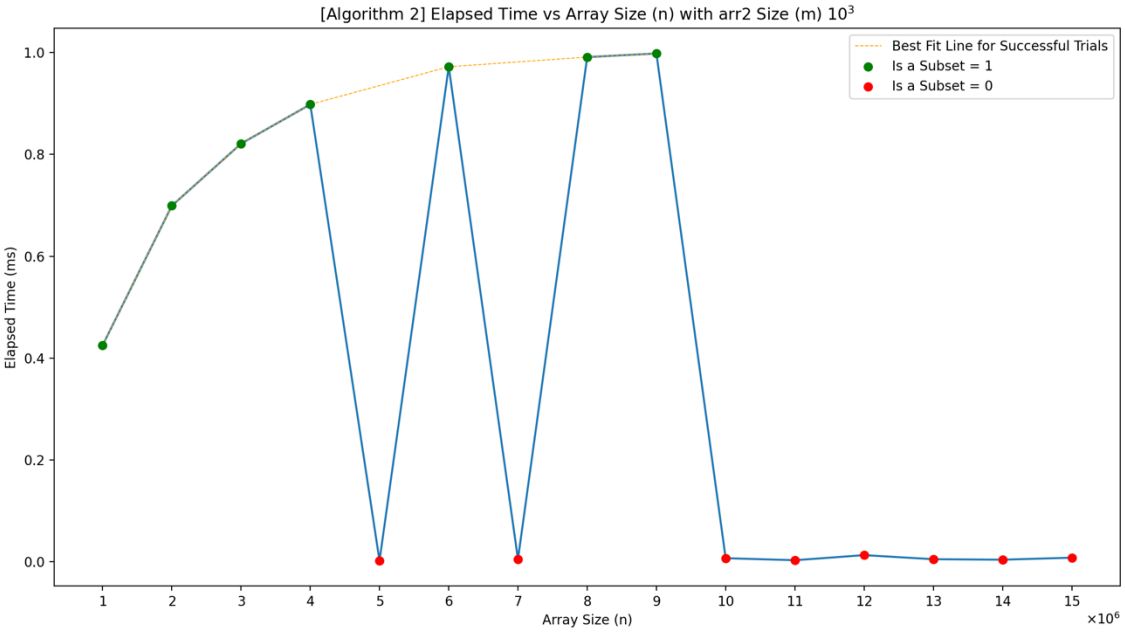
Plots



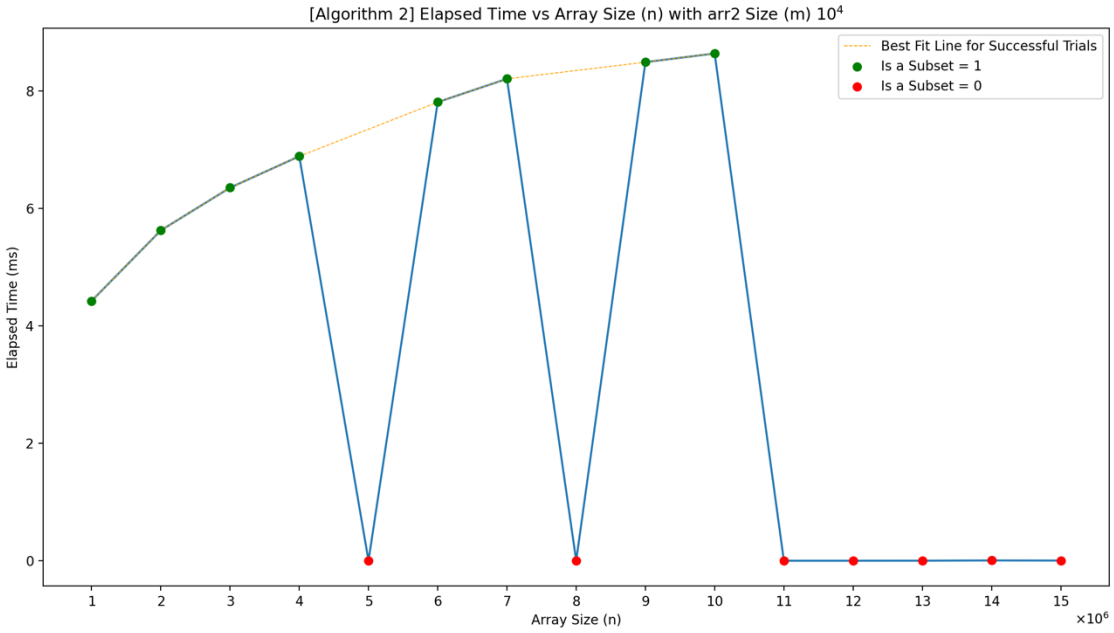
Plot 1



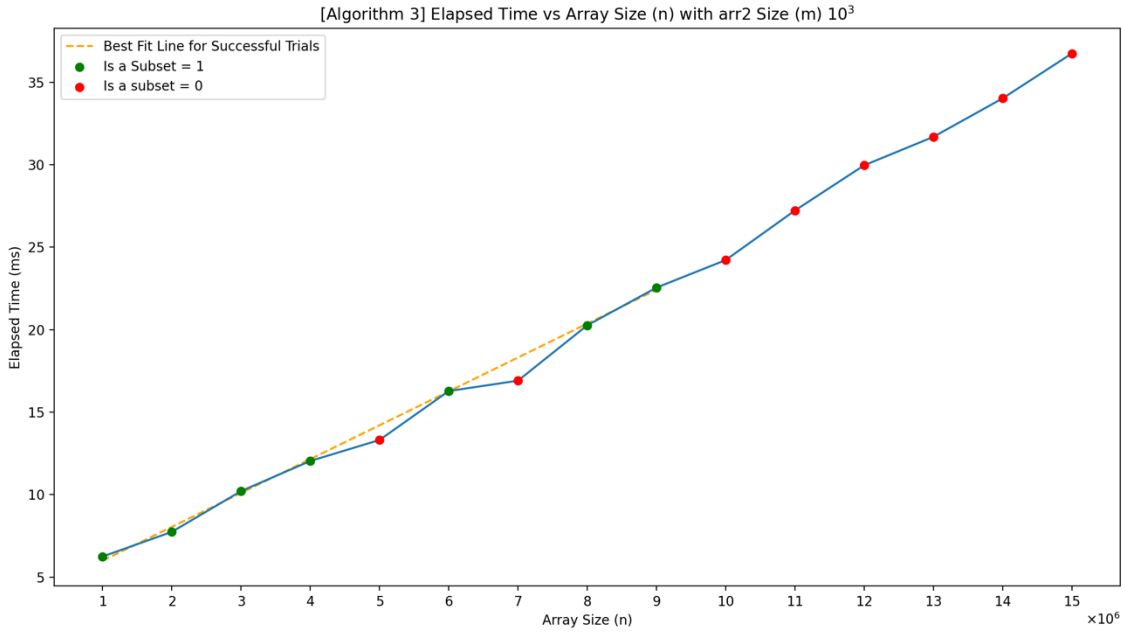
Plot 2


[Algorithm 2] Elapsed Time vs Array Size (n) with arr2 Size (m) 10^3
Best Fit Line for Successful Trials
Is a Subset = 1
Is a Subset = 0
Elapsed Time (ms)
Array Size (n)
×10^6

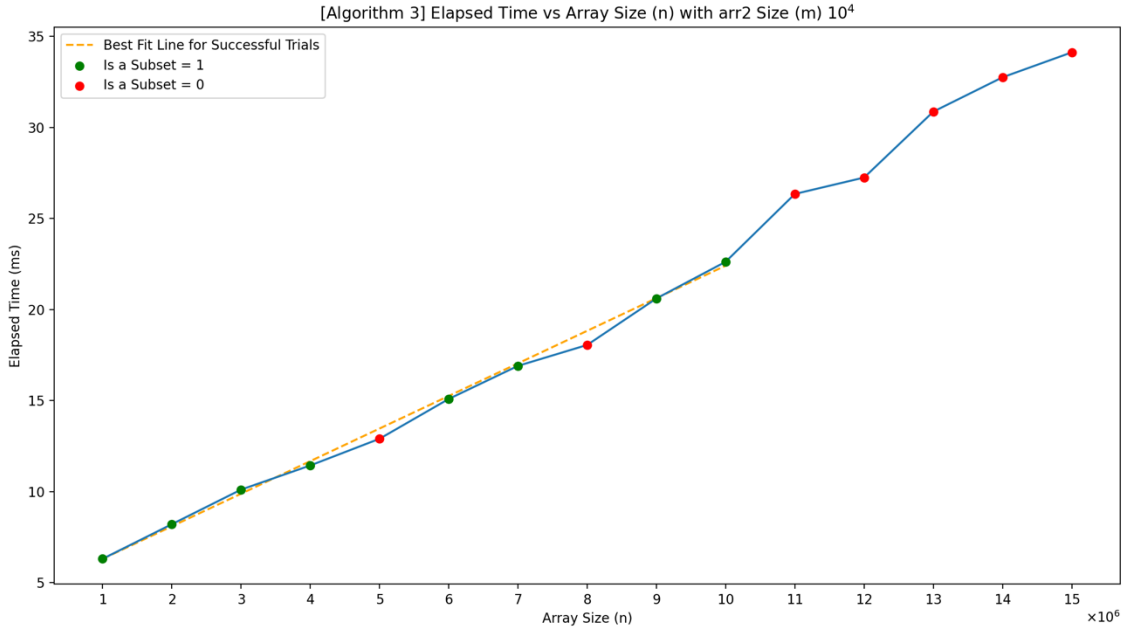
Plot 3


[Algorithm 2] Elapsed Time vs Array Size (n) with arr2 Size (m) 10^4
Best Fit Line for Successful Trials
Is a Subset = 1
Is a Subset = 0
Elapsed Time (ms)
Array Size (n)
×10^6

Plot 4


[Algorithm 3] Elapsed Time vs Array Size (n) with arr2 Size (m) 10^3
Best Fit Line for Successful Trials
Is a Subset = 1
Is a subset = 0
Elapsed Time (ms)
Array Size (n)
×10^6

Plot 5


[Algorithm 3] Elapsed Time vs Array Size (n) with arr2 Size (m) 10^4
Best Fit Line for Successful Trials
Is a Subset = 1
Is a Subset = 0
Elapsed Time (ms)
Array Size (n)
×10^6

Plot 6

Discussion

Each array used in the experiment was filled with random integers ranging from 0 to 15 million using `rand()`. With the help of the `srand()` function, unique seeds are used so that each element in the arrays is different from each other[1]. Interval of the array sizes is determined as to differ from 1 million to 15 million for `arr1` (15 different n sizes in total) and 1 thousand to 10 thousand (as given in the assignment paper) for `arr2` (size m). Every array used in this analysis is dynamically allocated. After every initialization of `arr1`, it was sorted via `std::sort` function and passed to all three algorithms as sorted. Although it is only necessary for algorithm 2 (due to binary search) that the array in which the subset will be searched is sorted, this way is preferred for all three algorithms to ensure consistency. `ctime` header was used for time measurement. In Algorithm 1, each `arr2` element selected in the outer for loop was searched linearly at `arr1` in the inner for loop. When the elapsed time for the algorithm to execute was converted into a graph (Plot 1-2), it was seen that the algorithm created a linear best fit line for the cases where the algorithm detects `arr2` (size m) as a subset of `arr1` (size n). As expected, it was concluded that this situation is consistent with the theoretically determined time complexity O(n*m), which grows according to a fixed slope when m or n is constant. In this case, the slope is $\sim\frac{1}{10^6}$ since the size of `arr2` (m) is fixed for each run. Since the searched array `arr1` is sorted, large values in `arr2` are accessed in `arr1` in a longer time, so it can be said that this increases the time spent in cases where `arr2` is a subset. Because of the early exit behavior of the algorithm, it directly returns 0 (false) as soon as it confirms that `arr2` is not a subset of `arr1`[2]. In such cases, it was noticed that the time spent in the algorithm was very short. Therefore, since it would be difficult to tell in which index the function fails and returns, it would be pointless to evaluate the timing of the cases found to be non-subsets based on the available data. Algorithm 2, unlike Algorithm 1, binary search is employed. In this case, since binary search will be performed instead of linear search in the n sized array, the n multiplier will be replaced by the logn in the worst-case time complexity. So, worst-case time complexity becomes O(m*logn). This situation shortens the time required for subset detection considerably (around

[1] Since advanced random number generation algorithms (e.g., mtt19937, Boost Library's random) are out of this course's scope, the RNG algorithm used in this experiment is less complex; therefore, repeating elements can occur with a bit of chance. Nevertheless, since the array sizes are relatively big, this behavior can be negligible.

[2] If `arr1` was not sorted, the timing of the cases where `arr2` is a subset and the timing of the cases where `arr2` is not a subset would be much closer than in this experiment since the large and small values in the array will show a homogeneous distribution. However, this would not change the upper bound; the previously determined time complexity O(n*m) would still be valid.

99%) compared to algorithm 1. As shown in the plots of Algorithm 2 (Plot 3-4), the best fit line forms a logarithmic line consistent with the theoretically calculated worst-time complexity. Similar to algorithm 1, the early return technique is used in Algorithm 2 too. Thus, in cases where `arr2` is not a subset, it can be suggested that the time spent in the function is very close to 0 milliseconds. For algorithm 3, an array named `freqTable`, whose size is the maximum value in `arr1 + 1`, was created within the main function and passed as an argument to the function which contains algorithm 3[3]. In Algorithm 3, `arr1` values are mapped to `freqTable`'s indexes. Then, by checking whether each element in `arr2` is in the frequency table, the subset status of `arr2` was determined. Since two respectively m and n sized arrays are traversed in distinct for loops in Algorithm 3, the worst-case time complexity of this algorithm is expected to be O(n+m). Likewise, by looking at the plots of Algorithm 3 (Plots 5-6), it can be seen that the best fit line of the successful trials formed a linear line, which validates the time complexity of this algorithm O(n+m). Unlike the rest of the algorithms, elapsed time of non-subset cases is very close to the subset ones. Although algorithm 3 also uses early returns, it early returns in the for loop with m iterations, n iterations in the first for loop is guaranteed to execute whether the `arr2` is a subset or not. Since used m's are much bigger than n's, this causes little change in the total time consumed. Therefore, no matter the output of algorithm 3, elapsed times would be close to each other for every n value. In conclusion, it was seen that the graphs formed by the data obtained as a result of this experiment were compatible with the theoretically calculated worst-case time complexities for all three algorithms. It was observed that the fastest running algorithm was algorithm 2, and the slowest running algorithm was algorithm 1, and algorithm 3 was somewhere between these two.

[3] In the assignment paper, it is unclear where the array containing the frequency table will be created and its elements will be set to zero. If these operations were included in algorithm 3, the upper bound of this algorithm would change to O(n+m+max) as the freqTable array would be traversed "`max+1`" times to set its elements to 0. Contacted the TA of the course about this issue, who confirmed that the algorithm should be implemented with time complexity of O(n+m). Therefore, the largest element in `arr1` is detected in the main function and the `freqTable` array is filled with zeros, again in the main function, and given as parameters to algorithm 3.