CS 201

Section 3

Homework 2

Ömer Oktay Gültekin

21901413

07/12/2021

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

Note that all algorithms show the worst case when the second array is a subarray of the first array. Other than that:

The 1st Algorithm, for each element on the second array, traverses the first array and compares the elements one by one with the element of the second array. The worst-case (upper bound) is that the elements the Algorithm is looking for in the first array are at the end of the first array. In that case, the Algorithm should do m iteration for the second array. In each iteration, it does n iteration to compare all elements of the first array with the element of the second array one by one. Therefore, the Algorithm works in O(n * m) in the worst case.

The 2nd Algorithm, for each element on the second array, takes the middle value of the sorted first array and compares the middle value with the element of the second array. If the looked element is small, then it takes the mid-value at the left subset; if the looked element is high, it takes the mid-value from the right subset; otherwise, it continues to the next element since it finds the element in the first array. The worst-case (upper bound) is that the elements the Algorithm is looking for in the first array are at the beginning or the end of the first array. Since the Algorithm needs to look at all subsets to find elements in 1st Algorithm, in that case, the Algorithm should do m iteration for the second array. In each iteration, it iterates through the first array. Each iteration eliminates half of the array until it finds the element in the first array. Therefore, the Algorithm works in O(m * log(n)) in the worst case.

The 3rd Algorithm constructs a frequency table by linearly passing over all elements of first array and then uses this frequency table while iterating through the second array. Each element on the second array looks at the frequency table and decrease the value if not 0 at a constant time. The worst and best cases are the same for the 3rd Algorithm since the elements of the second array search through the frequency table independent of the size of the frequency table and first array. The Algorithm should do n iteration for all cases while creating the frequency table. After creating the table, it does m iteration to compare all elements of the first array. Both iterations are independent of each other. Therefore, the Algorithm works in O(n * m) in the worst case.

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

MacBook Pro(2017), macOS Big Sur Version 11.6.1

Processor 2,3 GHz Dual-Core Intel Core i5

Memory 8 GB 2133 MHz LPDDR3

Graphics Intel Iris Plus Graphics 640 1536 MB

3)

n	Algorithm 1		Algorithm 2		Algorithm 3	
	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$
$5 * 10^4$	47.313	485.014	0. 261	2.638	1.776	4.575
10^5	94.605	1027.66	0. 279	2.759	3.512	3.513
$2.5 * 10^5$	245.817	2446.42	0. 302	3.114	8.938	9.516
$5 * 10^5$	514.91	5085.94	0. 316	3.223	16.649	17.369
$7.5 * 10^5$	763.359	7684.51	0. 334	3.261	25.457	27.445
10^6	1014.22	10245.9	0. 332	3.319	34.027	40.504
$2,5 * 10^6$	2478.19	25587.3	0. 354	3.578	82.589	85.782
$5 * 10^6$	5117.32	51982.8	0. 383	3.7	179.68	200.248
$7,5 * 10^6$	7822.8	78933.2	0. 375	3.8	271.38	304.734
10^7	10907.1	105513	0. 384	3.849	379.739	403.587

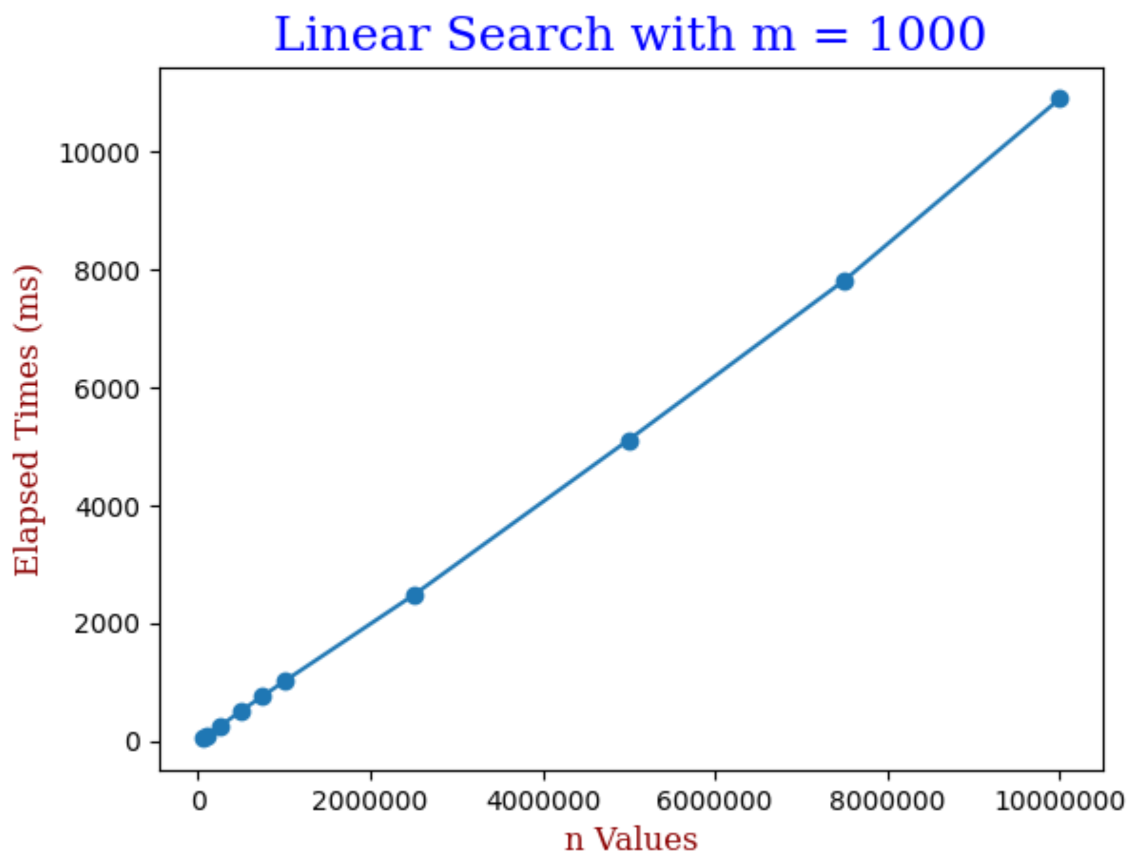
N - Elapsed Time (ms) Table

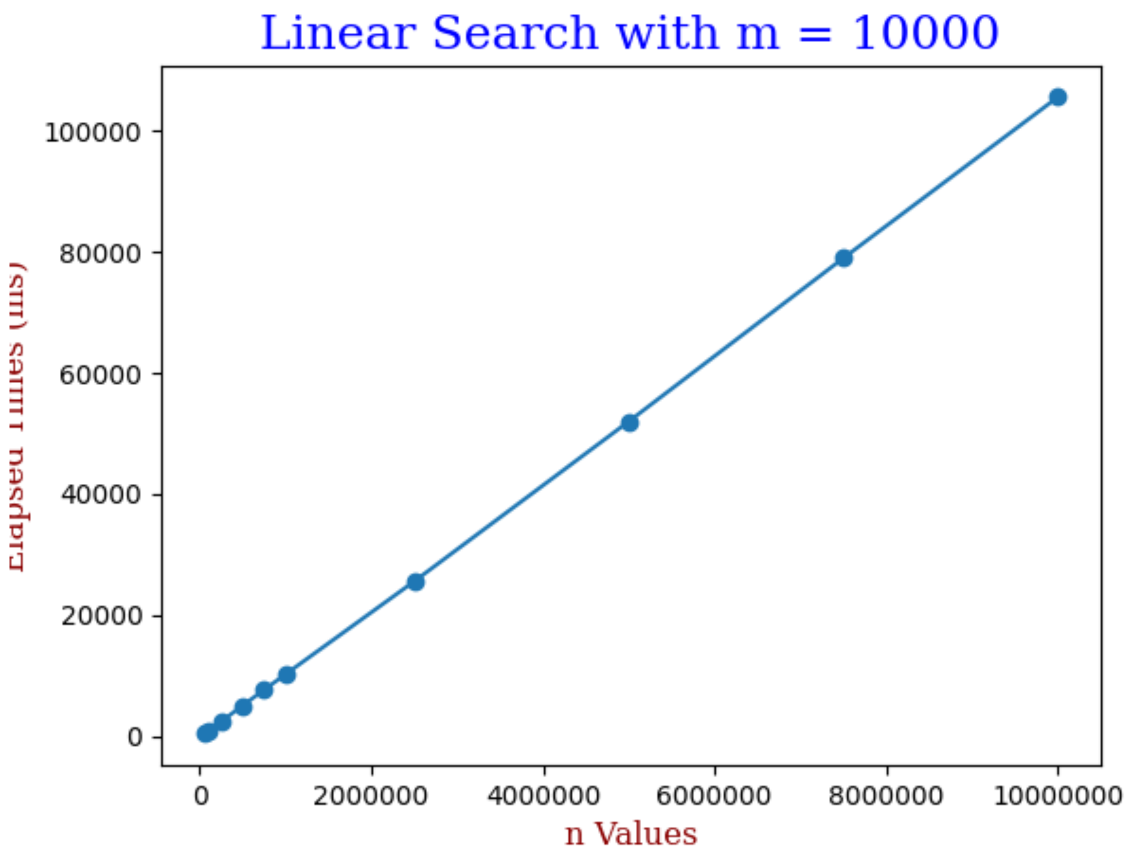
We were expected that algorithm 1 works in O(n*m). Indeed, the table shows the complexity O(n * m) since when m is constant, the elapsed time increases at a rate of increase in n. Also, when n is constant, the ratio of elapsed times is ten which is the ratio of the m values. Another evidence that starting from 6th row 1st column and 1st row 2nd column, the elapsed times are similar since n * m is equal.

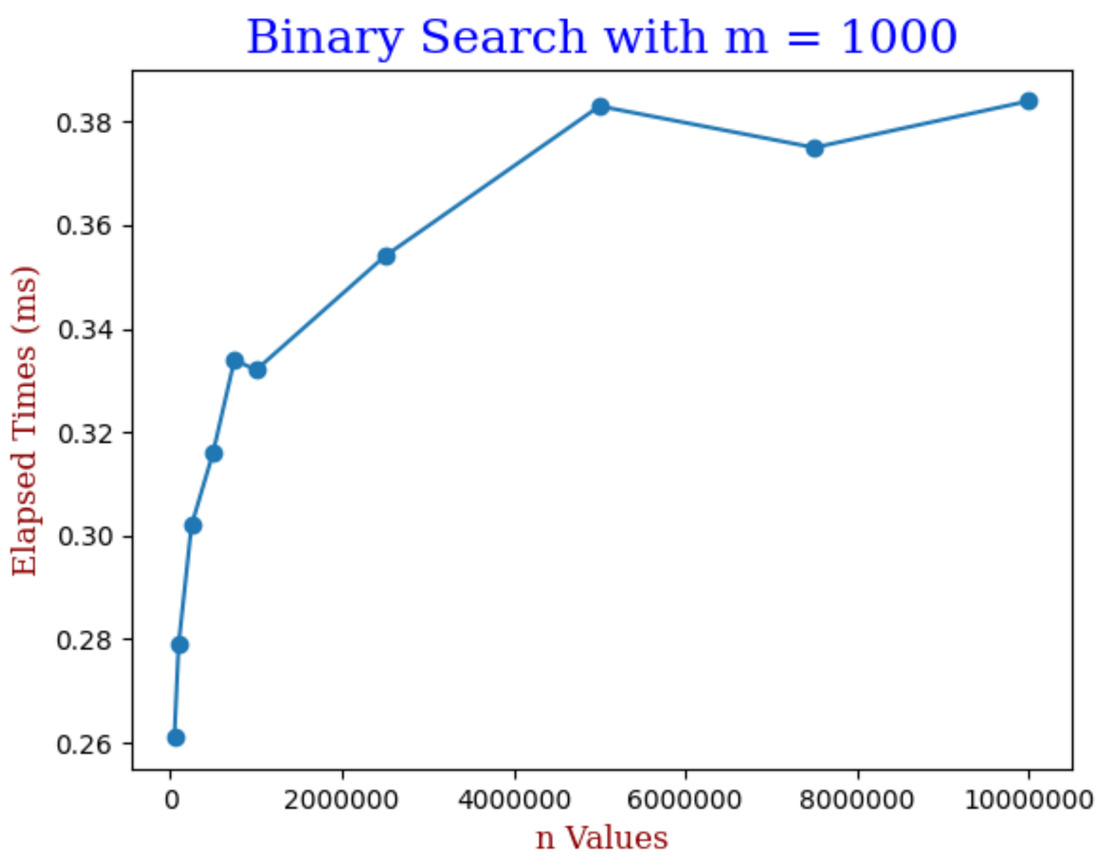
2nd algorithm also shows what we want since it works in O(m*logn). When m is constant, we can see that the algorithm works like a logarithmic function since at the beginning, the increase is fast, and at the end, the increase is slow down (better visible on the plot). When n is constant, we can see the elapsed times differ by the factor of 10. Putting together two behavior, the algorithm works in O(m * logn)

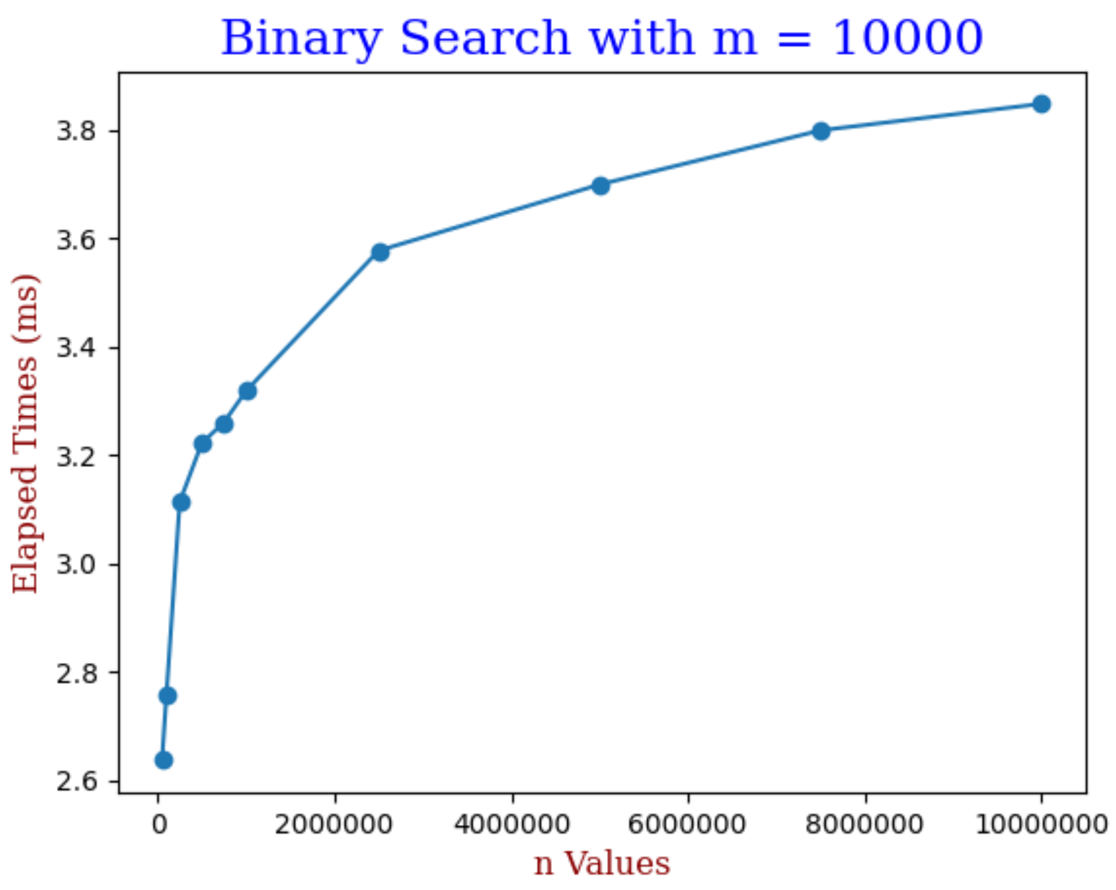
3rd algorithm shows O(n+m). When m is constant, the elapsed times increase at a rate of increase in n with some fluctuations. Therefore, the algorithm works like a linear function. When n is constant, we can see some fluctuations with small values. However, some elapsed times are added to the previous ones for large values. Therefore, the algorithm shows O(n + m) behavior.

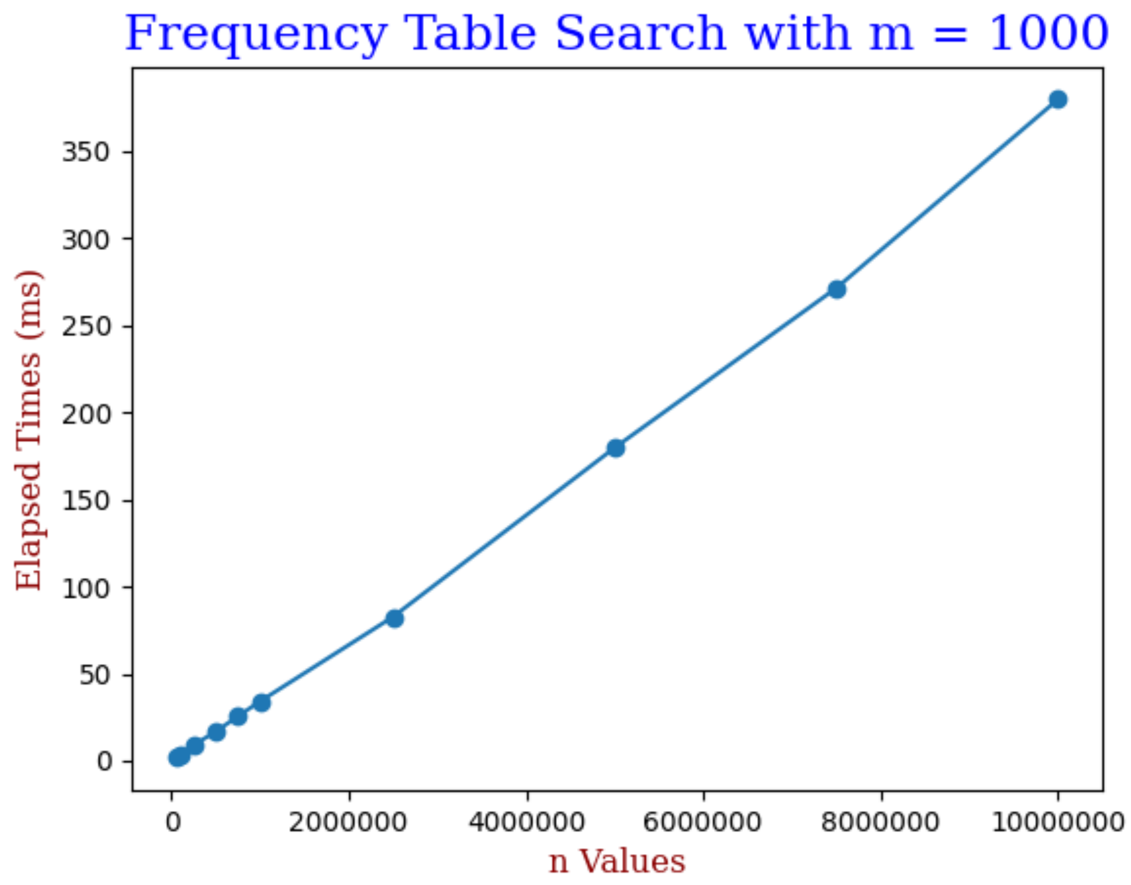
4)

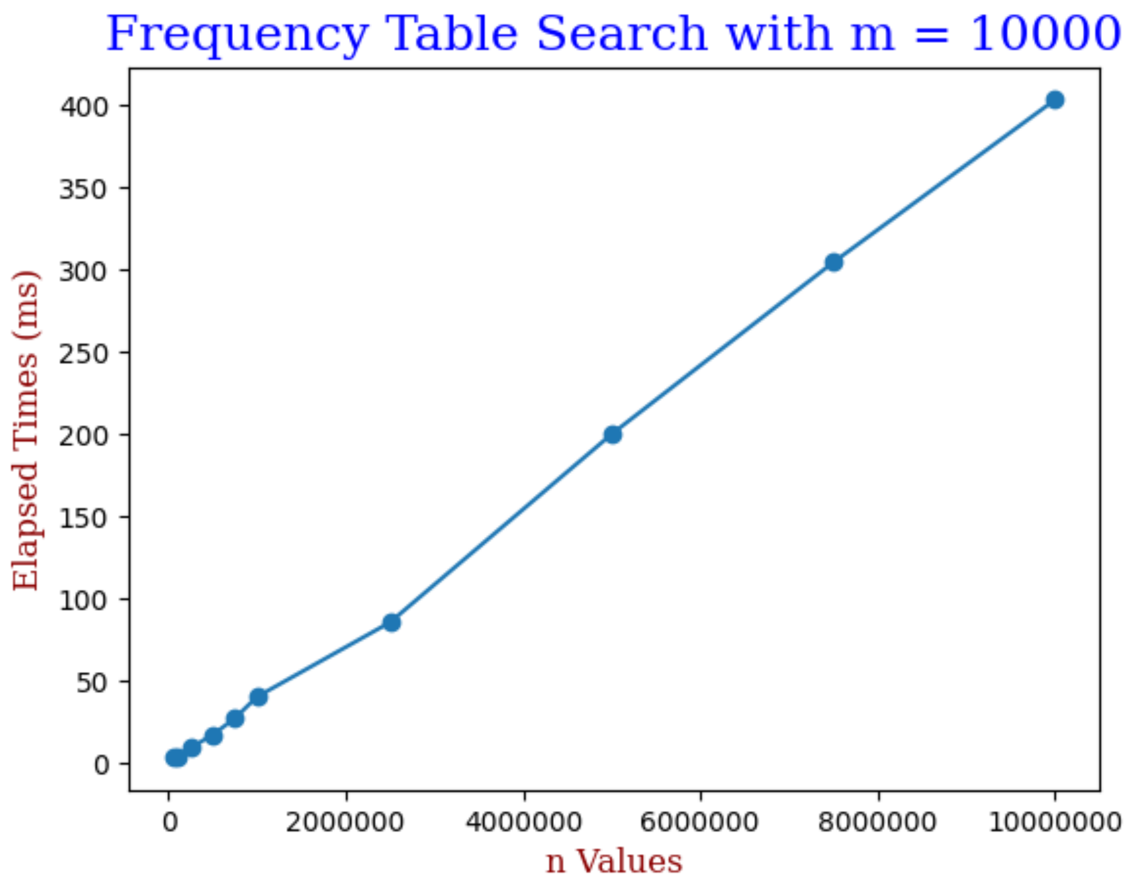

Linear Search with m = 1000
Elapsed Times (ms)
n Values
0
2000
4000
6000
8000
10000
0
2000000
4000000
6000000
8000000
10000000


Linear Search with m = 10000
Elapsed Times (ms)
n Values
0
20000
40000
60000
80000
100000
0
2000000
4000000
6000000
8000000
10000000


Binary Search with m = 1000
Elapsed Times (ms)
0.26
0.28
0.30
0.32
0.34
0.36
0.38
0
2000000
4000000
6000000
8000000
10000000
n Values


Binary Search with m = 10000
Elapsed Times (ms)
2.6
2.8
3.0
3.2
3.4
3.6
3.8
0
2000000
4000000
6000000
8000000
10000000
n Values





Linear Search and Frequency Table Algorithms show linear plots as expected, whereas Binary Search Algorithm shows the logarithmic plot. Since computer performance depends on too many conditions, there are some fluctuations in graphs.