CS201 HOMEWORK_2

Anıl Keskin

22201915

CS201-3

1)TABLES FOR SEARCH ALGHORITMS

N / ms		Jump Search		
	a	b	c	d
10	0.000021	0.000024	0.000027	0.000032
25	0.000020	0.000026	0.000036	0.000036
100	0.000032	0.000032	0.000082	0.000049
500	0.000020	0.000053	0.000093	0.000098
1.500	0.000019	0.000100	0.000133	0.000149
3.000	0.000021	0.000129	0.000166	0.000136
5.000	0.000020	0.000151	0.000201	0.000168
10.000	0.000022	0.000123	0.000300	0.000210
15.000	0.000020	0.000200	0.000397	0.000256
30.000	0.000022	0.000339	0.000432	0.000363

N / ms		Iterative Linear Search		
	a	b	c	d
10	0.000005	0.000011	0.000015	0.000016
25	0.000003	0.000020	0.000034	0.000031
100	0.000004	0.000054	0.000096	0.000096
500	0.000005	0.000221	0.000444	0.000428
1.500	0.000004	0.000636	0.001264	0.001274
3.000	0.000004	0.001262	0.002519	0.002514
5.000	0.000004	0.002192	0.004268	0.004202
10.000	0.000004	0.004148	0.008205	0.008324
15.000	0.000003	0.006124	0.012591	0.013267
30.000	0.000003	0.012828	0.025629	0.026193

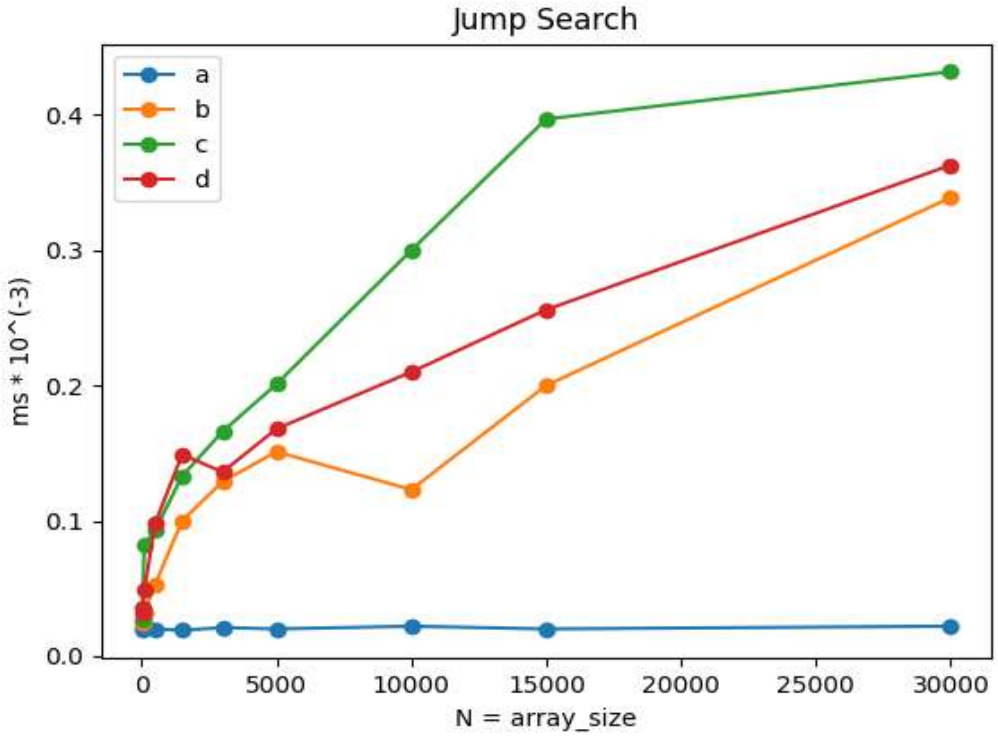
N / ms		Recursive Linear Search		
	a	b	c	d
10	0.000003	0.000014	0.000020	0.000023
25	0.000005	0.000027	0.000074	0.000104
100	0.000004	0.000375	0.000321	0.000322
500	0.000003	0.000884	0.001651	0.001474
1.500	0.000003	0.002594	0.006117	0.005814
3.000	0.000002	0.005838	0.012334	0.012517
5.000	0.000004	0.009773	0.020664	0.020209
10.000	0.000003	0.019565	0.041786	0.041354
15.000	0.000003	0.030098	0.063529	0.062639
30.000	0.000004	0.062681	0.139647	0.138046

N / ms		Binary Search		
	a	b	c	d
10	0.000011	0.000011	0.000071	0.000012
25	0.000013	0.000004	0.000015	0.000021
100	0.000014	0.000019	0.000020	0.000018
500	0.000019	0.000024	0.000026	0.000025
1.500	0.000022	0.000036	0.000032	0.000031
3.000	0.000039	0.000039	0.000042	0.000035
5.000	0.000027	0.000038	0.000039	0.000049
10.000	0.000045	0.000037	0.000041	0.000033
15.000	0.000073	0.000040	0.000041	0.000047
30.000	0.0042	0.000037	0.000045	0.000050

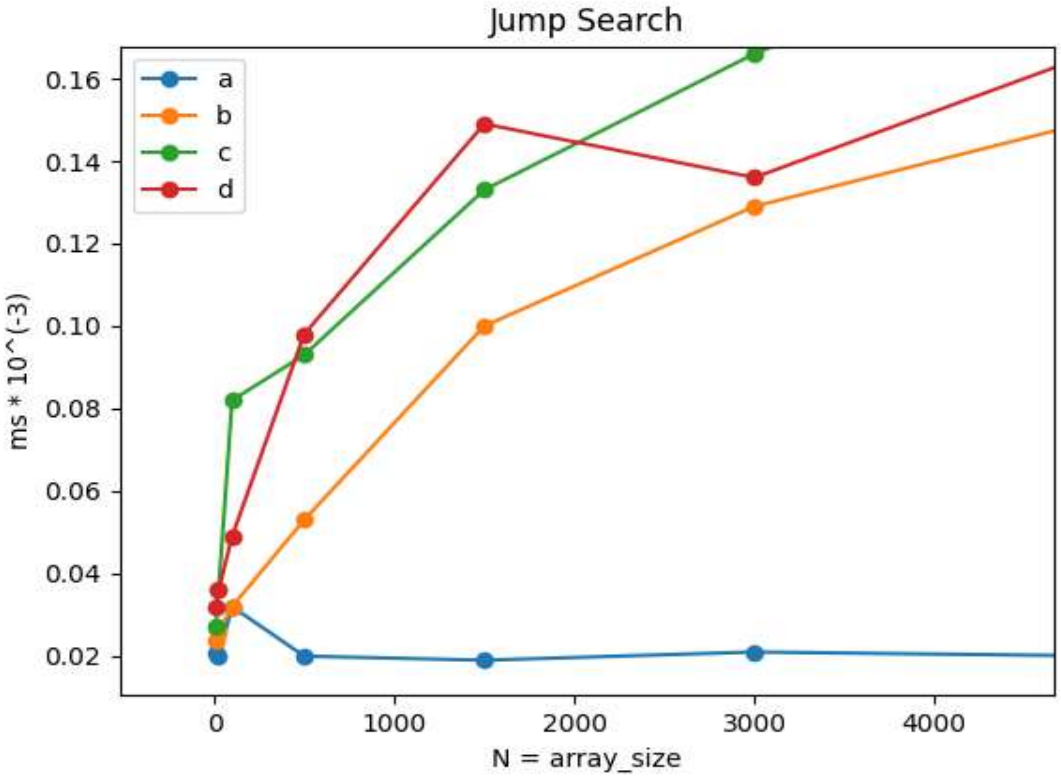
N / ms	Random Search			
	a	b	c	d
10	0.000212	0.000148	0.000447	0.00742
25	0.000668	0.000203	0.000159	0.001163
100	0.002866	0.000723	0.002263	0.008116
500	0.004908	0.002137	0.002244	0.047473
1.500	0.017531	0.002212	0.002305	0.131079
3.000	0.016168	0.021544	0.002861	0.287797
5.000	0.032239	0.083154	0.004018	0.548129
10.000	0.029160	0.043217	0.005267	1.211370
15.000	0.029816	0.061192	0.521508	2.460240
30.000	0.036644	0.050691	1.273130	4.123210

2) PLOTS FOR SEARCH ALGHORITMS

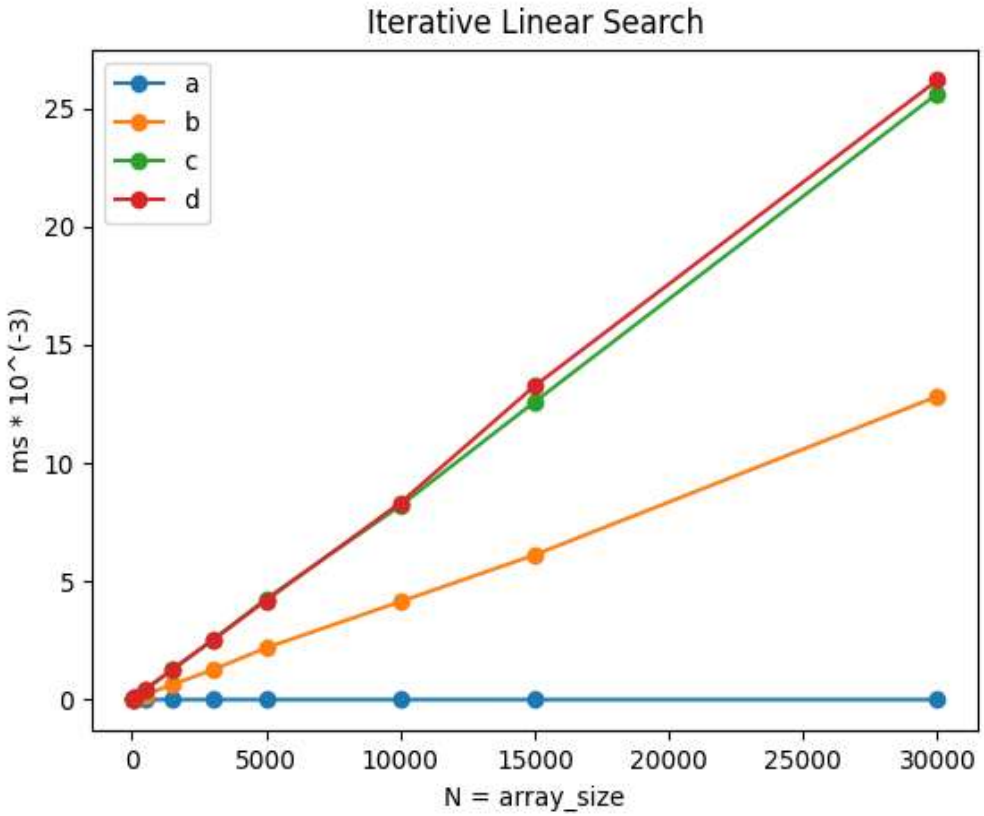
2.1)Jump Search



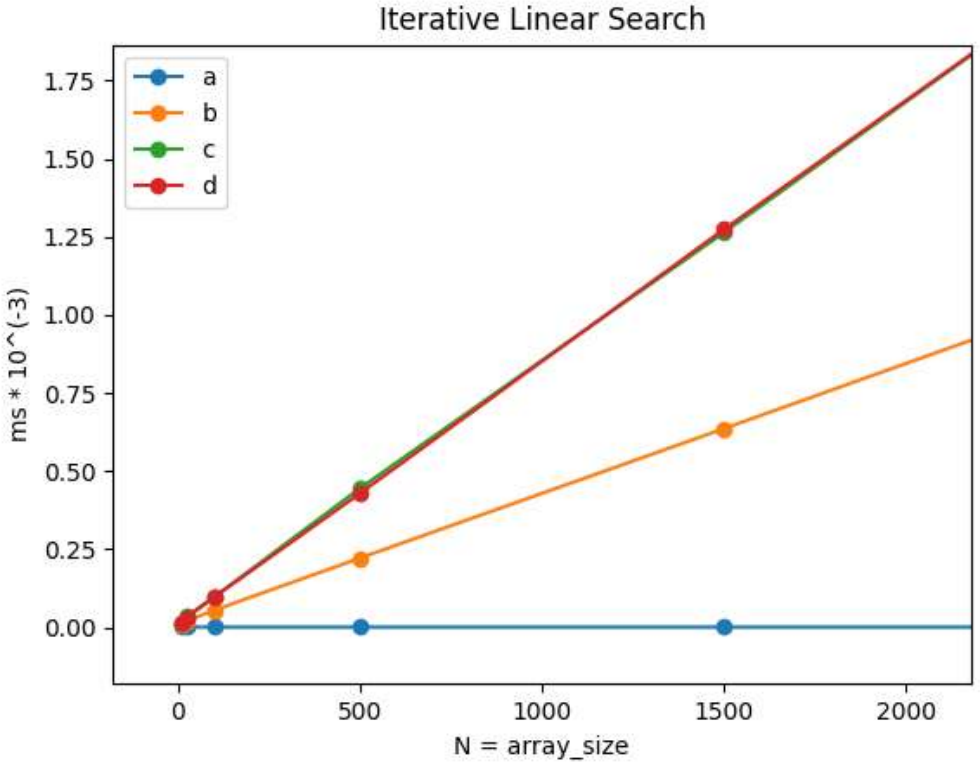
- Plot below encapsulates initial array_sizes for easier observation



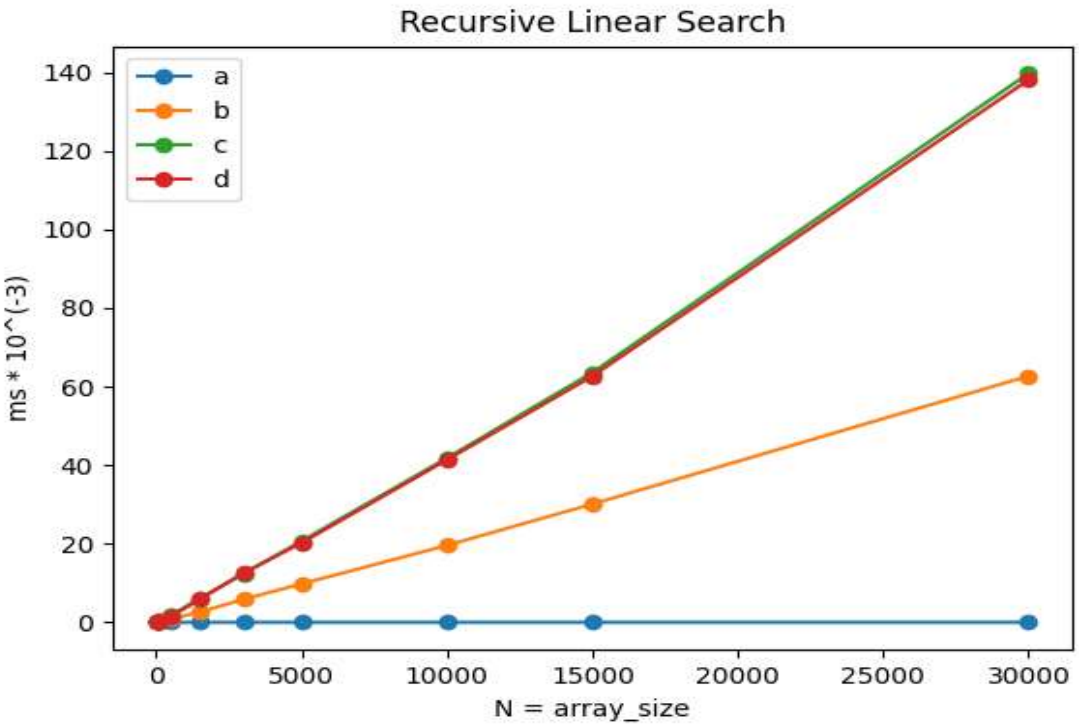
2.2)Iterative Linear Search



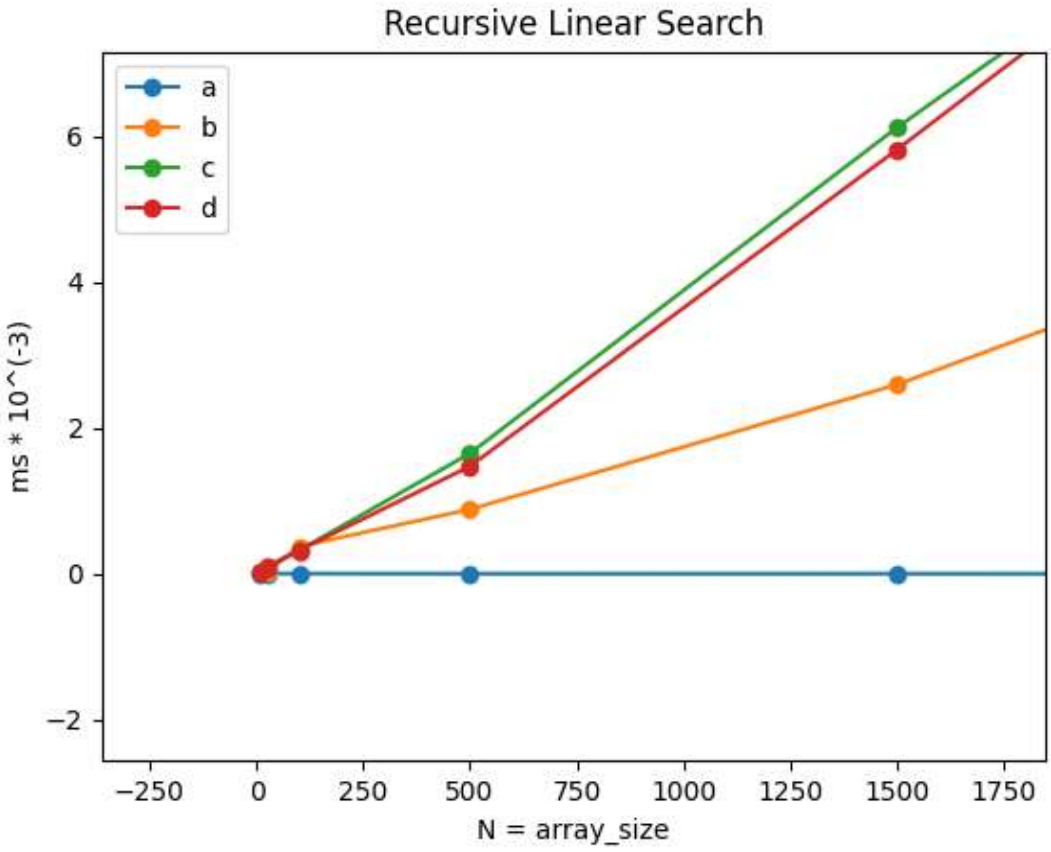
- Plot below encapsulates initial array_sizes for easier observation



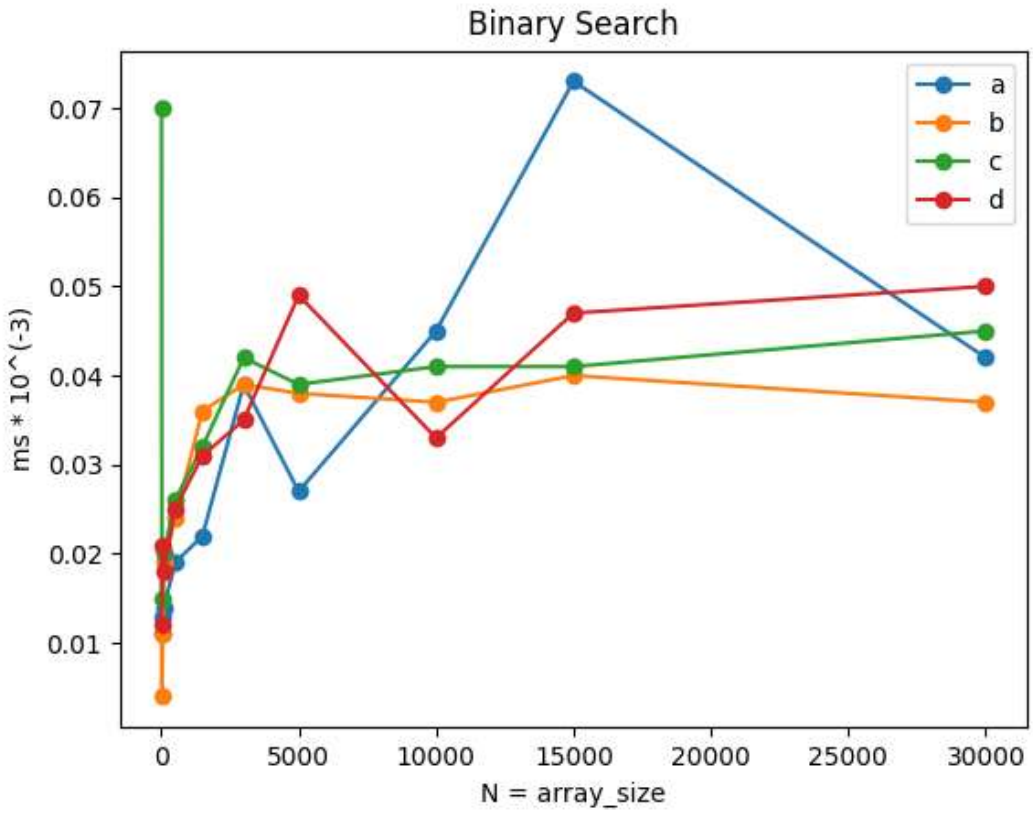
2.3) Recursive Linear Search



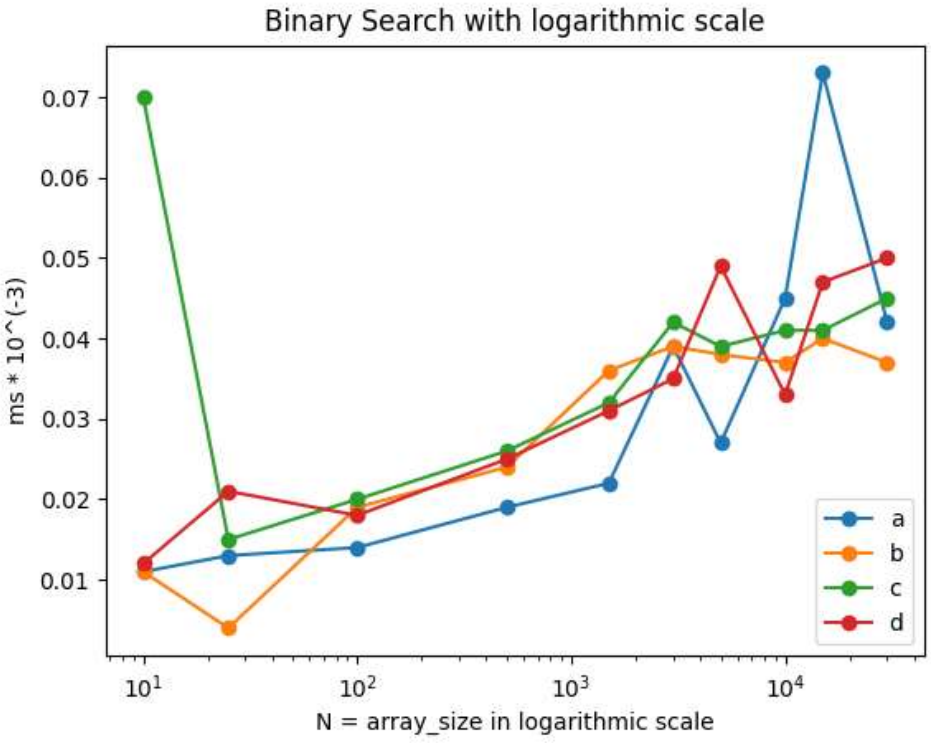
- Plot below encapsulates initial array_sizes for easier observation



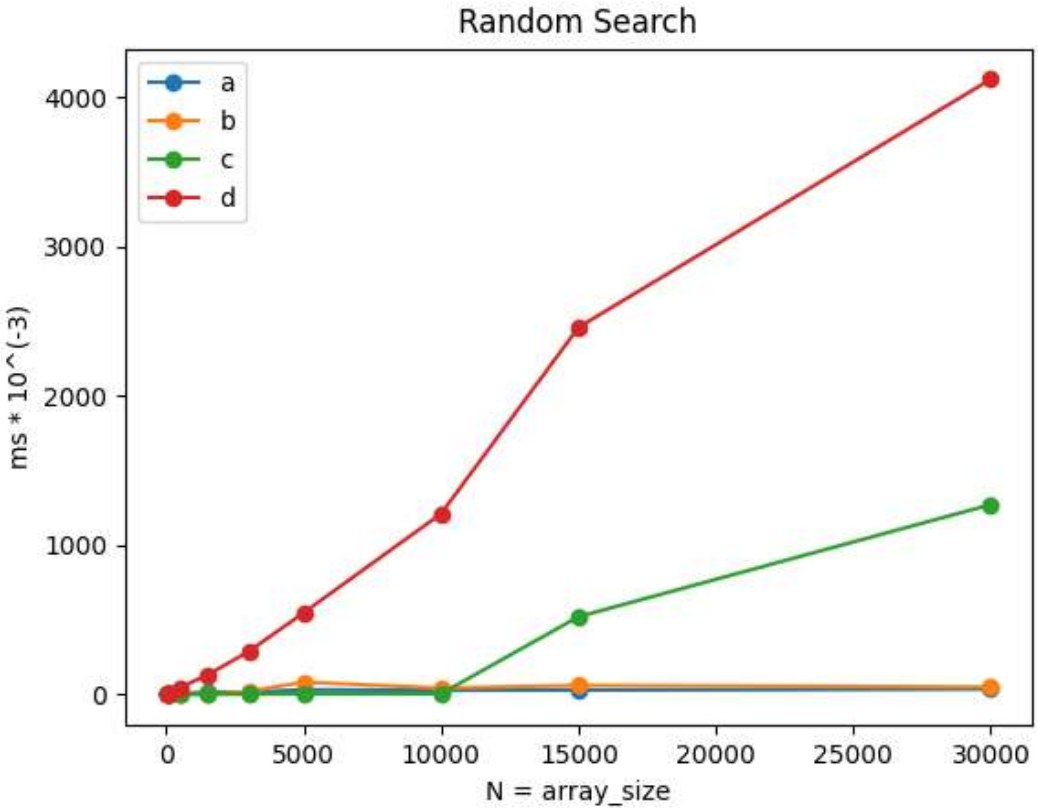
2.4)Binary Search



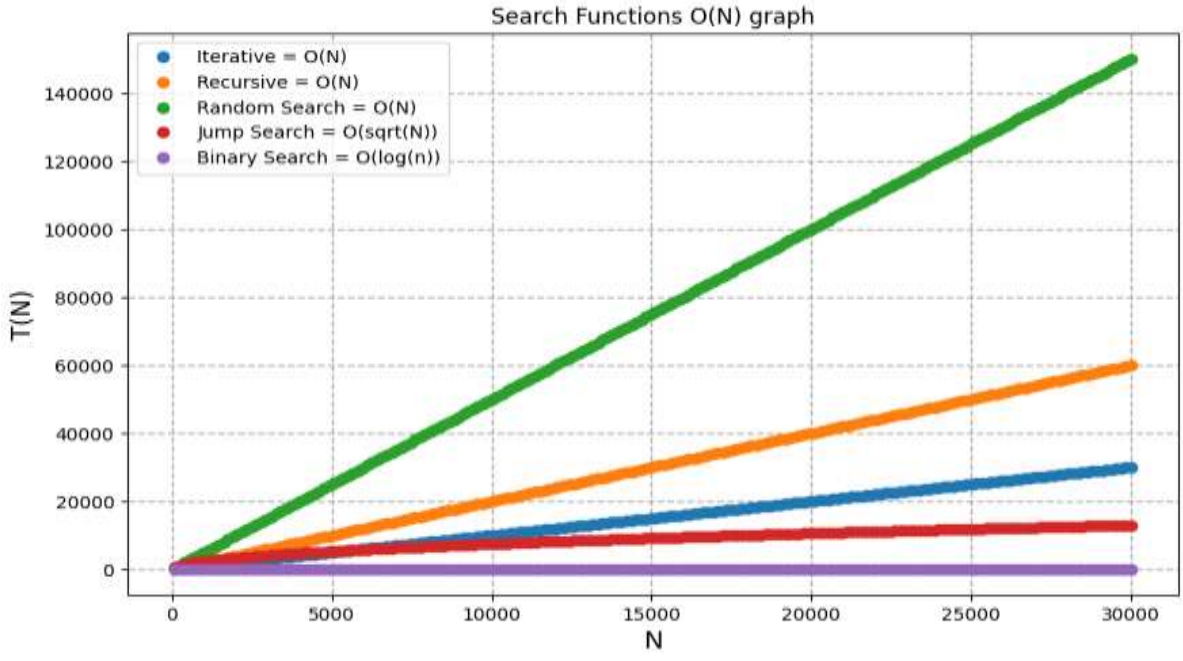
- Plot below puts array_sizes according to logaritmic scale making function look like O(N) if function is O(log(n)), for easier observation



2.5)Random Search



3.Discussion About Alghorithms and Results



During the computations of the algorithms dynamically allocated arrays are used to increment the size of array and search algorithms executed 100 times per test and later average of 100 taken to get more accurate time executions . Array size varied from 10 to 30.000. Larger sizes than 30000 resulted in crash of the program due to only Recursive Linear Search's recursive chain call causing possible stack overflow or heap memory issues(?). On the contrary, the selection of array sizes provided enough observation to determine the behavior of search algorithms.

3.1-The results of Iterative and Recursive Linear Searchs are observed to be fit in the pattern of O(N) time complexity as they should be. -for the sake of making point, empricial datas will be used for once only here- Getting experimental numbers from the table:

For N 5000-10000 => (5000/10000) = 1/2 ≈ (0.002192ms/0.004148ms) and N : 1500-3000 => 1500/3000 = 1/2 ≈ (0.000636ms/0.0012620ms)[For Iterative]

For N-5000-15000 => 1/3 ≈ (0.009773ms)/(0.030098ms) and for N : 15.000-30.000 => 1/2 ≈ (0.030098ms)/(0.0626310ms) [For Recursive];

Discussion of Cases for Both Linear Search Algorithms:For senarios of b,c,d ; that formula holds for any pair of data sets because b is Average [it takes (best+worst) /2 times iteration making it ave**rage case**] and c,d are **worst Case** senarios [due to whole array being iterated] for both algorithm having time complexity of O(N) whereas for senario a, it is the **best case** and gives size-independent results of (O(1)).

However recursive one taking much time than iterative one isn't result of them having different time complexities but a result of recursive one having much work due to recursion, such as memory allocations and new frames added to stack and parameters being copied down due to constant function calls.

3.2-When 'Jump Search' is runned, we expect it to be efficient than O(N) algorithms (such as Linear) and less efficient than Binary Search (on sorted collections it is O(log(N)). Implemantation of Jump Search and its time complexity depends on the smaller block size (m) of array chosen => O(N/(m+m)) and for it to be efficient, it is to be O(sqrt(n)) when m is chosen to be sqrt(n) also.

Deriving comparision from getting pair of values and checking if they are giving constant of sqrt(n) and its multiples would take much space in the report, instead interpreting the previous paragraph would be convenient. Jump search must be between linear search and binary search. The figure at the very beginning of section 3 showing theoric results of functions also states that. When empirical results of stated algorithms are compared we indeed can see jump search takes less time to operate than linear searches and more time than binary search. In addition to that, jump search has shape of curve-like line resembling the function of y=sqrt(x) in the empirical plot.

Jump search's **best case** is when key value is to be found in very first block; only having O(N=sqrt(m)) linear search; or having no key value which is less than values of first-block elements. If key value that does not exist in the array less than 0 (or elements of first block) it would result in best case senario but key value that does not exist in the array held bigger than the last element of array in this experiment. For key values that near last (**scenario c**) is to be **worst case** senario due to blocks being iterated through whole array and later being subject of linear search of that block. For key values "near midde" and "not in collection", they can be **average case** since first basically being near middle having **(best+worst)/2** and latter avoiding time cost of linear search of a block.

3.3-In Binary Search we expect **worst cases** to happen when elements are at extreme points of array [such as near first and near last] and have **best** in middle, **average** in anywhere of the array.

In the empricial plot, we see it not from the very beginning of the plot but after the half; that most efficient being **scenario-b** and **scenario (a-c)** being the **worst cases** [a is worser than c according to plot but behavior of plot states a and c being worst two scenarios]. Scenario d is also candidate of being **worst case** however, due to them being dependent of array elements (so is a and c), points in the plot have some fluctiations. Correct observation would be rather than comparing the a-b-c-d with themselves pairwise, to seperating **scenario b(best senario)** and **(a-c-d).** Scenarios of a-c-d may overcome or suprass each other according to what random array is created (each comparision is element dependent so every random array, one of three scenarios would be less worse than other).

3.4- Random search in theory oblicated to have O(N) time complexity because linear-search like comparision is applied randomly on the collection. The empirical plot shows time taken during operation increases as much increase in size of collection made. In random search, actually there is no **average** and **best case** but the **worst case** which is the case when whole array is searched completely. It is the case in **scenario d** which distinguishes from other 3 scenarios. For other 3 scenarios, a value in array to be selected randomly has same possibility as anothers being selected. For **scenarios of a-b-c**, they randomly (*by luck or possibility*) can have same, less or bigger time duration during operation [*the green line somehow seperates from the other two below is just produce of binomial possibility or by just luck basically*) but never bigger than **scenario d**. Moreover, due to constant produce of random value and validation of if that random being produced ever, it takes more time than other O(N) algorithms. Indeed, this algorithm is efficient for very small and trivial collections where random value that can be generated has smaller range, hence values that would have worst-case scenario in other algorithms would be found in instant by luck in that algorithm.

4. System Specifications

Operating System: Windows 11 Pro 64-bit (10.0, Build 22631) (22621.ni_release.220506-1250)

Language: Turkish (Regional Setting: Turkish)

System Manufacturer: Micro-Star International Co., Ltd.

System Model: MS-7D22

BIOS: 2.20 (type: UEFI)

Processor: 11th Gen Intel(R) Core(TM) i5-11400F @ 2.60GHz (12 CPUs), ~2.6GHz

Memory: 16384MB 3200 MHz DDR4 RAM

Available OS Memory: 16266MB RAM

Page File: 10979MB used, 10150MB available

Windows Dir: C:\WINDOWS

DirectX Version: DirectX 12

DX Setup Parameters: Not found

User DPI Setting: 96 DPI (100 percent)

System DPI Setting: 96 DPI (100 percent)

DWM DPI Scaling: Disabled

Miracast: Available, no HDCP

Microsoft Graphics Hybrid: Not Supported

DirectX Database Version: 1.6.3

DxDiag Version: 10.00.22621.3527 64bit Unicode