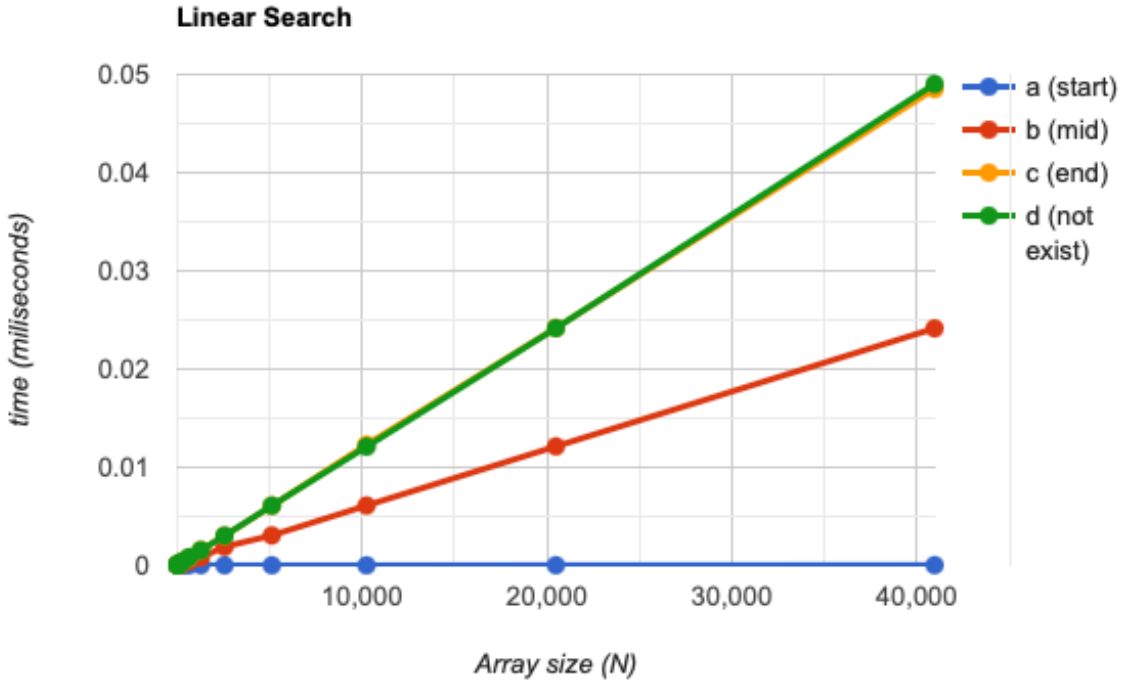
CS201 Homework-2:

Analyzing different algorithms and their execution time

Ece ŞEŞEN
22201637
22.11.2024

Algorithm 1: Linear Search

Array Size (N)	Linear (Iterative) Search			
	a (start)	b (mid)	c (end)	d (not)
10	0.000050	0.000040	0.000040	0.000040
20	0.000020	0.000030	0.000050	0.000040
40	0.000030	0.000050	0.000070	0.000090
80	0.000010	0.000090	0.000160	0.000160
160	0.000030	0.000140	0.000260	0.000260
320	0.000020	0.000210	0.000450	0.000420
640	0.000030	0.000420	0.000790	0.000850
1280	0.000040	0.000830	0.001580	0.001550
2560	0.000020	0.001910	0.003010	0.003040
5120	0.000020	0.003070	0.006090	0.006100
10240	0.000020	0.006110	0.012320	0.012100
20480	0.000030	0.012130	0.024200	0.024170
40960	0.000040	0.024150	0.048520	0.049030



Algorithm 2: Linear Recursive Search

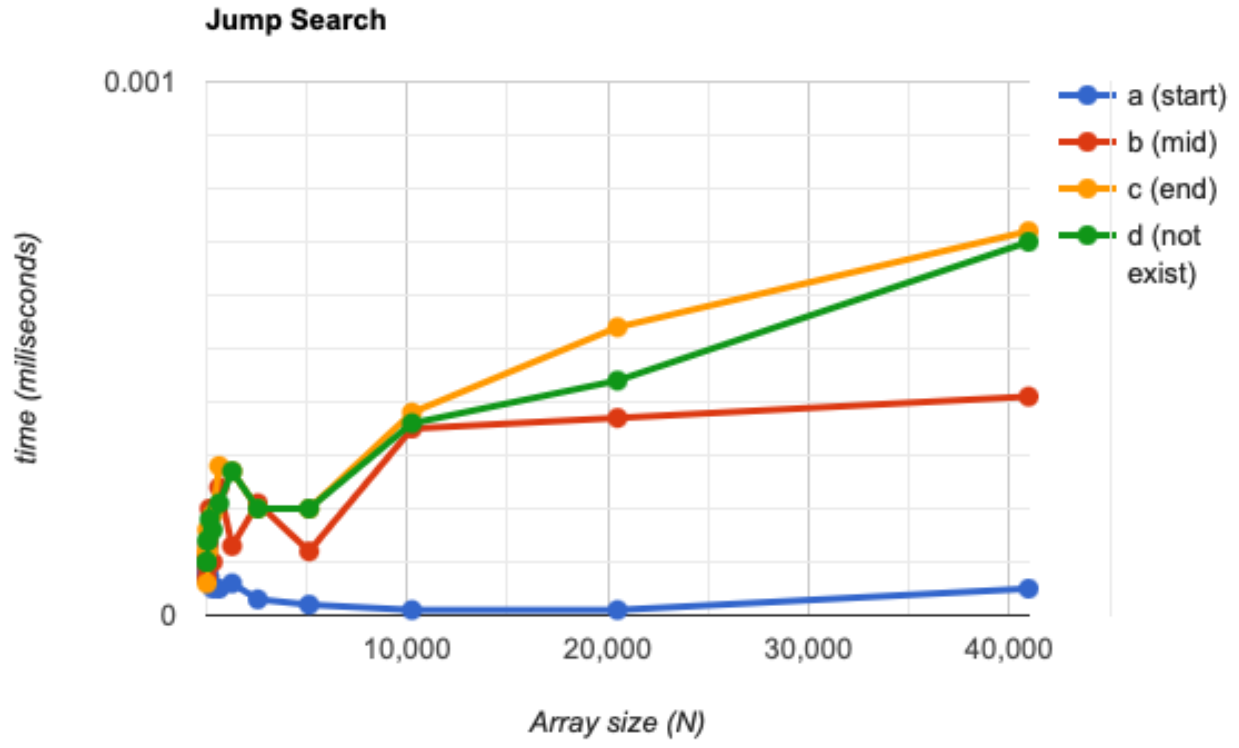
Array Size (N)	Linear (Recursive) Search			
	a (start)	b (mid)	c (end)	d (not)
10	0.000020	0.000020	0.000040	0.000040
20	0.000020	0.000080	0.000060	0.000070
40	0.000030	0.000060	0.000410	0.000430
80	0.000010	0.000500	0.000560	0.000610
160	0.000010	0.000570	0.001400	0.001300
320	0.000020	0.001240	0.002380	0.002290
640	0.000020	0.002250	0.004490	0.004540
1280	0.000020	0.004520	0.008900	0.008850
2560	0.000030	0.010300	0.017590	0.017490
5120	0.000040	0.018040	0.037800	0.037900
10240	0.000020	0.037340	0.076420	0.074930
20480	0.000030	0.076120	0.153540	0.152680
40960	0.000010	0.154560	0.309610	0.310880

Algorithm 3: Binary Search

Array Size (N)	Binary Search			
	a (start)	b (mid)	c (end)	d (not)
10	0.000020	0.000021	0.000019	0.000027
20	0.000022	0.000025	0.000023	0.000020
40	0.000033	0.000025	0.000030	0.000020
80	0.000028	0.000026	0.000029	0.000031
160	0.000035	0.000029	0.000028	0.000020
320	0.000025	0.000032	0.000031	0.000020
640	0.000033	0.000039	0.000030	0.000018
1280	0.000035	0.000036	0.000039	0.000033
2560	0.000033	0.000038	0.000032	0.000029
5120	0.000035	0.000032	0.000039	0.000042
10240	0.000044	0.000033	0.000041	0.000048
20480	0.000046	0.000040	0.000047	0.000051
40960	0.000050	0.000041	0.000045	0.000054

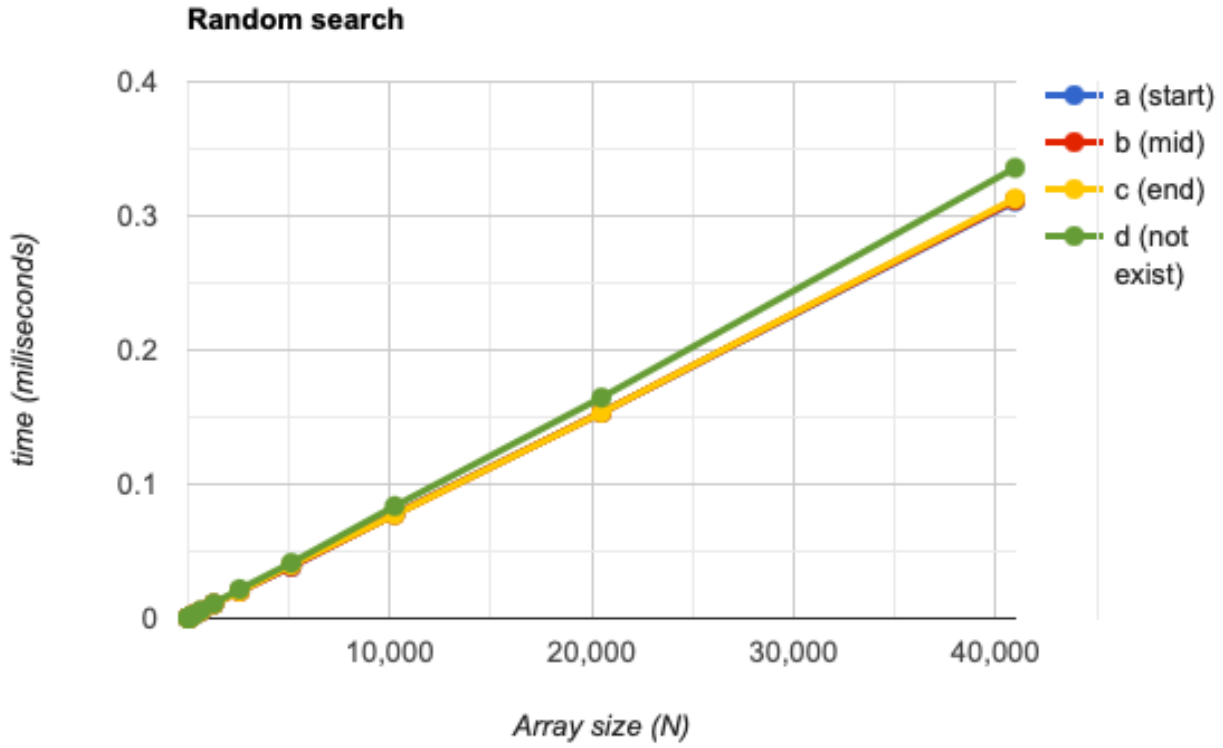
Algorithm 4: Jump Search

Array Size (N)	Jump Search			
	a (start)	b (mid)	c (end)	d (not)
10	0.000080	0.000110	0.000120	0.000120
20	0.000070	0.000070	0.000060	0.000060
40	0.000090	0.000090	0.000160	0.000160
80	0.000080	0.000130	0.000120	0.000120
160	0.000070	0.000200	0.000160	0.000160
320	0.000050	0.000100	0.000190	0.000190
640	0.000050	0.000240	0.000280	0.000280
1280	0.000060	0.000130	0.000270	0.000270
2560	0.000030	0.000210	0.000200	0.000200
5120	0.000020	0.000120	0.000200	0.000200
10240	0.000010	0.000350	0.000380	0.000380
20480	0.000010	0.000370	0.000540	0.000540
40960	0.000050	0.000410	0.000720	0.000720



Algorithm 5: Random Search

Array Size (N)	Random Search			
	a (start)	b (mid)	(end)	d (not)
10	0.000172	0.000207	0.000189	0.000165
20	0.000246	0.000263	0.000261	0.000258
40	0.000456	0.000438	0.000426	0.000483
80	0.000828	0.000776	0.000766	0.000934
160	0.001565	0.001441	0.001617	0.001697
320	0.002627	0.003019	0.002811	0.002830
640	0.005026	0.005371	0.005344	0.005969
1280	0.010643	0.010840	0.010693	0.011082
2560	0.020347	0.020072	0.020042	0.021783
5120	0.038490	0.038700	0.039714	0.041414
10240	0.077750	0.077422	0.077226	0.083717
20480	0.153389	0.153365	0.153295	0.164627
40960	0.310776	0.311760	0.312921	0.335838



5.1. What are the specifications of your computer? (Processor, RAM, operating system and etc)

I use MacBook M2 Air. Features of the computer are as follows: Firstly, considering processor, it has *Apple M2 chip*. Also, it has *8 core CPU* and *10 core GPU*. Moreover, it has *16 core Neural Engine*. Its *ram* is *16GB unified memory* (shared between CPU, GPU and neural engine). Its operating system is *macOS Ventura* operating system. The storage of the computer is *512GB SSD*.

5.2. What are the theoretical worst, average, and best cases for each algorithm? What are the corresponding running times for each scenario for each algorithm?

1. Linear Search:

a. Best case -> O(1)

The best case of the linear search occurs when the key element is located in the index zero which means first element of the array. It has only **O(1)** time complexity. Since, computer finds the index in the first comparison and directly returns the result.

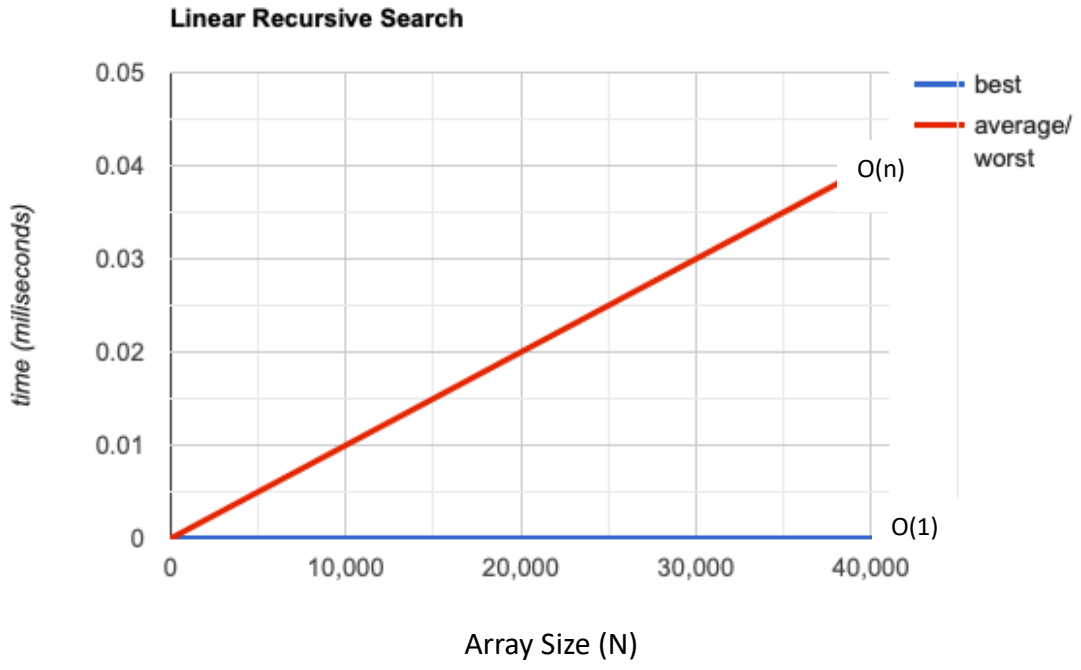
b. *Average case -> O(n)*

Traversing the elements and finding the key element in the middle of the array. In other words, this algorithm needs to go over half of the elements which correspond to (size / 2). However, since the constants are ignored while calculating time complexity, this algorithm has **O(n)** time complexity in average case.

c. Worst case -> O(n)

The worst case occurs both when the key element does not include in an array or located in the last index of the array. This algorithm has to traverse all along to array elements and compare all elements with the key element in order to find its index in both situations. It returns (-1) when it could not find key element in the array or return (size – 1) which means the key element is the last element of the array. Therefore, comparing all the elements with the key elements gives **O(n)** time complexity.

2. Linear Recursive Search:



a. Best case -> O(1)

This case occurs when the key element is in the first because recursive function checks first element and calls itself again by ignoring the one more index from the beginning. *(index + 1)* Since base cases depends on either empty array *(index >= size)* or *(arr[index] == key),* if key element is stored in the first index of the array, function could directly return 0 by doing only one comparison which makes time complexity **O(1)**.

b. Average case -> O(n)

This case corresponds to when computer finds the key element on half of the way. Which means since array start index increases one in each call, computer finds desired value when it calls itself with the parameter (nearly index -> size / 2). Nevertheless, since result depends on N (size of the array), time complexity is equals to **O(n)**.

c. Worst case -> O(n)

It occurs when computer could *not find the key value* or find it in the last index of the array. It requires to compare key element with all elements according to function call interval. Time complexity is equals to **O(n)**.

3. Binary Search:

a. Best case -> O(1)

The best case occurs when the key element is located in the middle of the array. Therefore, computer can find its index directly with just comparing once. *(arr[mid] == target)* which makes time complexity is **O(1).**

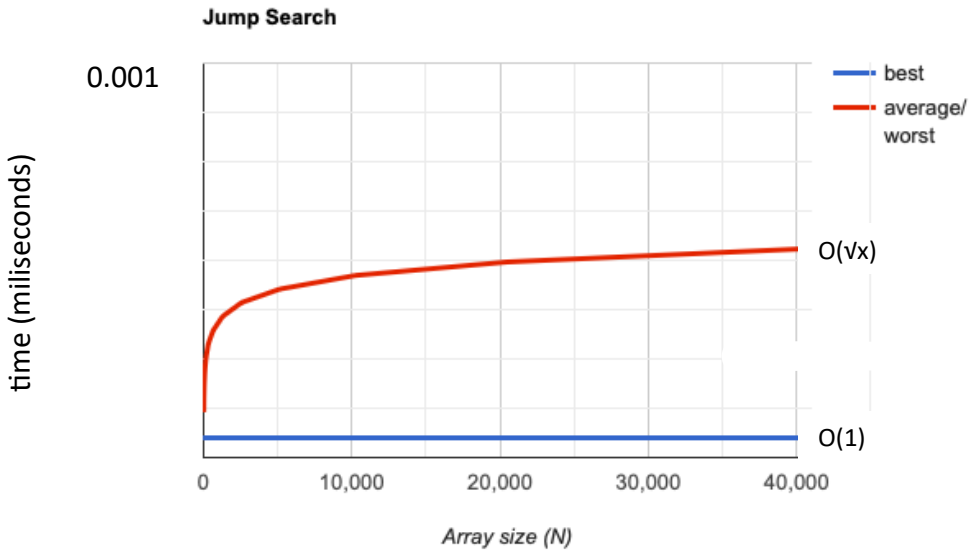
b. Average case -> O(logn)

This algorithm tries to find the key elements by updating borders and checking the middle value in each loop *(while (low < high)).* In other words, comparing the key element with the middle *((low + high) / 2)* value in the updated interval continues until computer finds the index of wanted number or return (-1) meaning that number does not exist in the array. In the average case, computer goes half of the several intervals which is equals to n times ignoring constant values. Therefore, time complexity of the algorithm is **O(logn)**.

c. Worst case -> O(logn)

The worst case occurs when the computer could not find the value after traversing all element. This process includes comparison between array elements and key element which costs **O(logn)** time complexity.

4. Jump Search:



a. Best case -> O(1)

Best case occurs when the key element is located in the first index. Therefore, being the first element of the array enables computer to find key element by just doing one truncating interval containing only (index, index + interval of jump) relevant part.

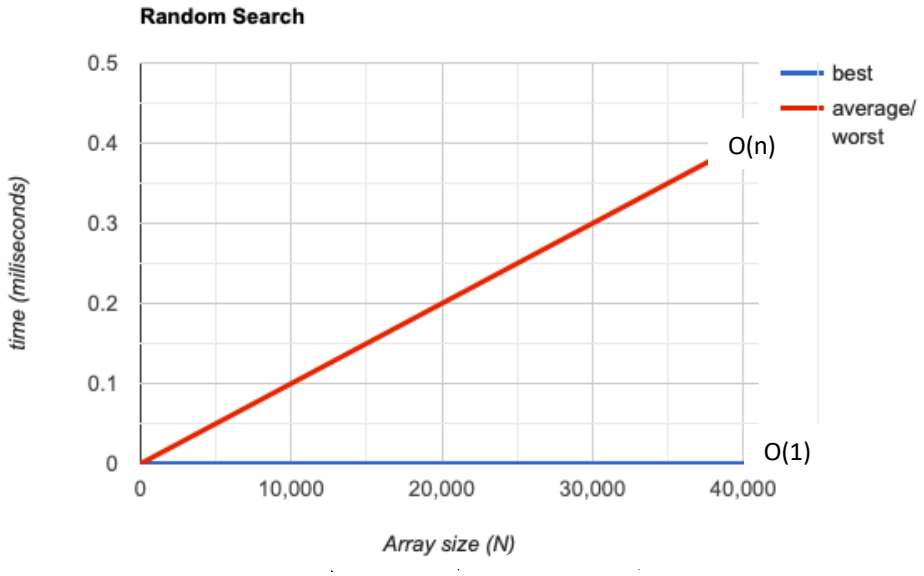
b. Average case -> O(√n)

Average case occurs when the key elements stay in the middle. In order to find this element, computer checks multiple of jump intervals. When the value, stored in one of the jump multiples, greater than the key element, computer takes an interval starting from previous jump multiple indexes. Finally, it makes a linear search in the determined interval and return index of key value ultimately. Time complexity of the algorithm is **O(n)**.

c. Worst case -> O(√n)

Traversing all the elements and could not find the key element in the array is the worst case. Going over all elements make time complexity is **O(n)**.

5. Random Search:



a. Best case -> O(1)

Best case corresponds to when random index is same with the index of key element. Therefore, in the first attempt computer finds the searched index. Therefore, since there is a only one comparison *(arr[random] == key),* time complexity is **O(1).**

b. Average case -> O(n)

When computer finds the key element in its half off the maximum attempt, it corresponds too average case. Since it requires nearly n/2 comparisons and when computer finds the jump interval, it makes linear search which makes time complexity **O(n)**.

c. Worst case -> *O(n)*

Finding the key element in the last attempt or never finding it corresponds to the worst case. This method is implemented as follows: Firstly, random array is mixed and then linear search is used. Since computer traverse from all elements in order to find the key element its time complexity is **O(n)**.

➔ **In conclusion:**

Algorithm	Best	Average	Worst
Linear Search	O(1)	O(n)	O(n)
Linear Recursive Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(logn)	O(logn)
Jump Search	O(1)	O(√n)	O(√n)
Random Search	O(1)	O(n)	O(n)

5.3. What are the worst, average, and best cases for each algorithm based on your table and your plots? Do theoretical and observed results agree? If not, give your best guess as to why that might have happened.

Linear Search -> Best Case: beginning
Average Case: middle
Worst Case: end/no exist

Theoretical results and my results agreed. The graph is **linear** and while the best case is when the element at the beginning and the worst case when the element is at the end.

Linear Recursive Search -> Best Case: beginning
Average Case: middle
Worst Case: end/no exist

Theoretical results and my results agreed. The graph is **linear** and while the best case is when the element at the beginning and the worst case when the element is at the end.

Binary Search -> Best Case: middle
Average Case: beginning/end
Worst Case: does not exist

Theoretical results and my results are agreed which means it shows **f(x) = log(x)**. Because comparing scenarios of first and end locations, there should not be a distinct difference which means since binary search works as updating low and high limits of the interval, finding element at the end or at the beginning may be resulted in equal time.

However, finding element which does not exist is the array is the worst case and it agreed with my observation. Moreover, since I chose exactly the middle element in the second scenario, binary search gives the best case.

Jump Search -> Best Case: beginning
Average Case: middle
Worst Case: end/no exist

Theoretical results and my results are nearly agreed. The graph should represent the function of **f(x) = sqrt(x)** since I chose jump interval as "sqrt(x)" in order to be more efficient. However, even if there are some oscillations in small sized arrays, function takes its expected shape in grater sized arrays.

Moreover, best case which occurs when the key element at the beginning and the worst case which is when the key element is at the end or does not exist, both are agreed with theoretical results.

The time different between third and fourth scenarios may stem form linear search implementation of third scenario. Even if traversing from all blocks (except the last one) is common in both scenarios, third scenario makes linear search in order to find key element in determined interval. However, forth scenario is directly return (-1) without doing any linear search since the "endBlok" variable exceed the "*size" of the array*.

Random Search -> Best Case: beginning/middle/end
Average Case: beginning/middle/end
Worst Case: does not exist

In random search, the time should not depend on the element's initial position in the array. In other words, element in the middle or at the beginning or end does not make any sense when we interpret the behavior of the graph because since I shuffle the array before finding, position of key element is changed which means that end element may become first element or vice versa. Therefore, each iteration could give different results, however the wort case is always the same which is when key element does not exist on the array.

Using my observations, my plot represents the worst case. However, since each iteration gives different results, beginning/middle/end scenarios overlapped.

5.4. Plot theoretical running times for each algorithm across different scenarios and compare them to the plots at step 4. Comment on consistencies and inconsistencies. (These plots should have the same organization as plots in step 4.)

In conclusion, as I mentioned before, I compared and make comments on their differences. Nevertheless, briefly, my observations and plots are mostly agreed with theoretical results.

However, initial part of the plots is not agreed at all with the expected ones. In other words, while my plots generally oscillate in small sized arrays, they reach expected shape in larger sized arrays.

One reason of the decreasing oscillation towards to end could be the more data entered; the more accurate plot gained.