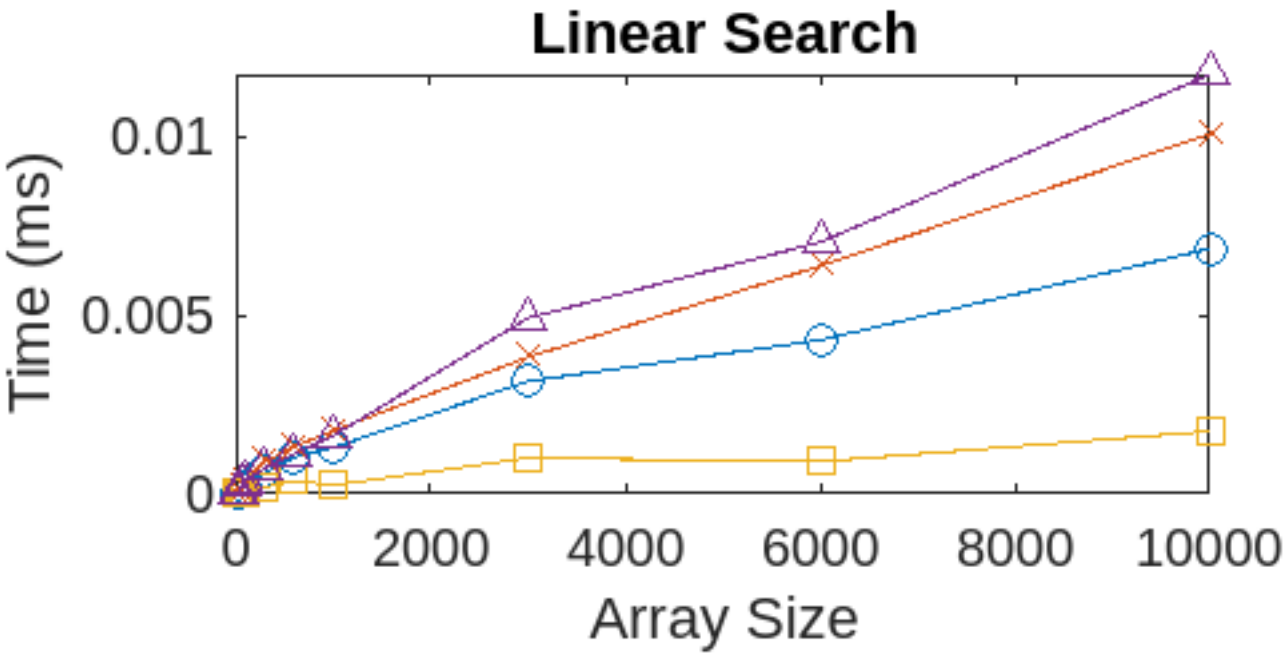
Cs201 HW2

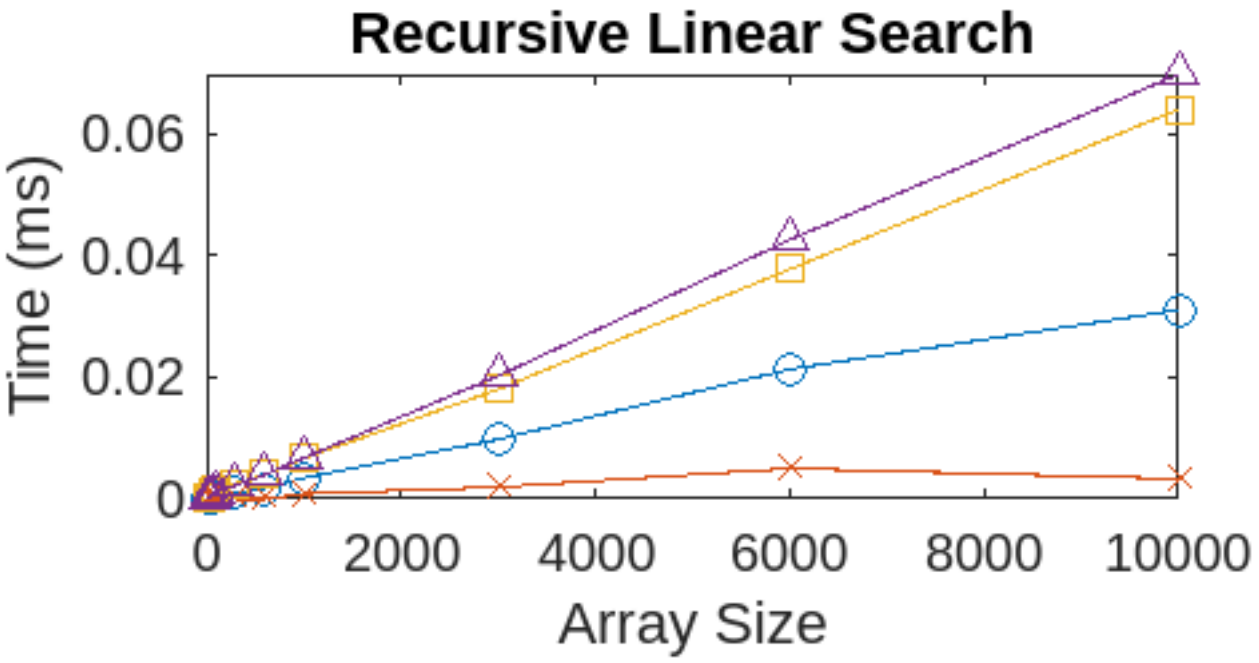
Altay İlker Yiğitel
22203024
Sec-3

Functions use average key assignment(First key(arr(0,size/5) etc.)



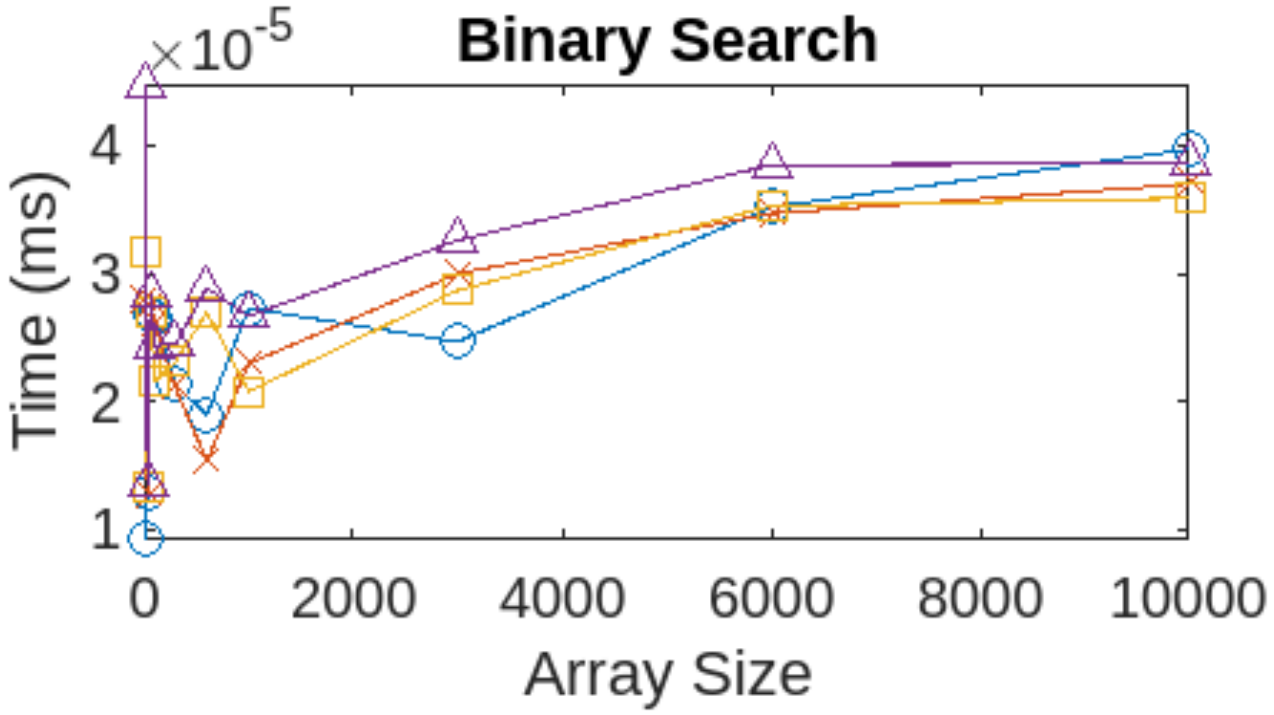
LinearSearch

Key/ Time(ms)	10	30	60	100	300	600	1000	3000	6000	10000
First Key	6.1e-06	1.04e-05	3.13e-05	2.4e-05	0.0001893	0.0003752	0.0002438	0.0010063	0.0009053	0.0017525
Mid Key	2,00E-0	2.65e-05	0.0001379	0.0001813	0.0005989	0.0010259	0.0012955	0.0031492	0.0042977	0.0068533
Last Key	3.53e-05	4.99e-05	0.0002069	0.0003137	0.0008967	0.0013316	0.0017671	0.0038235	0.0063712	0.0100607
Non-existing	4.11e-05	4.89e-05	0.0002242	0.0003618	0.0007306	0.0011187	0.0016305	0.0049134	0.0070364	0.0117186



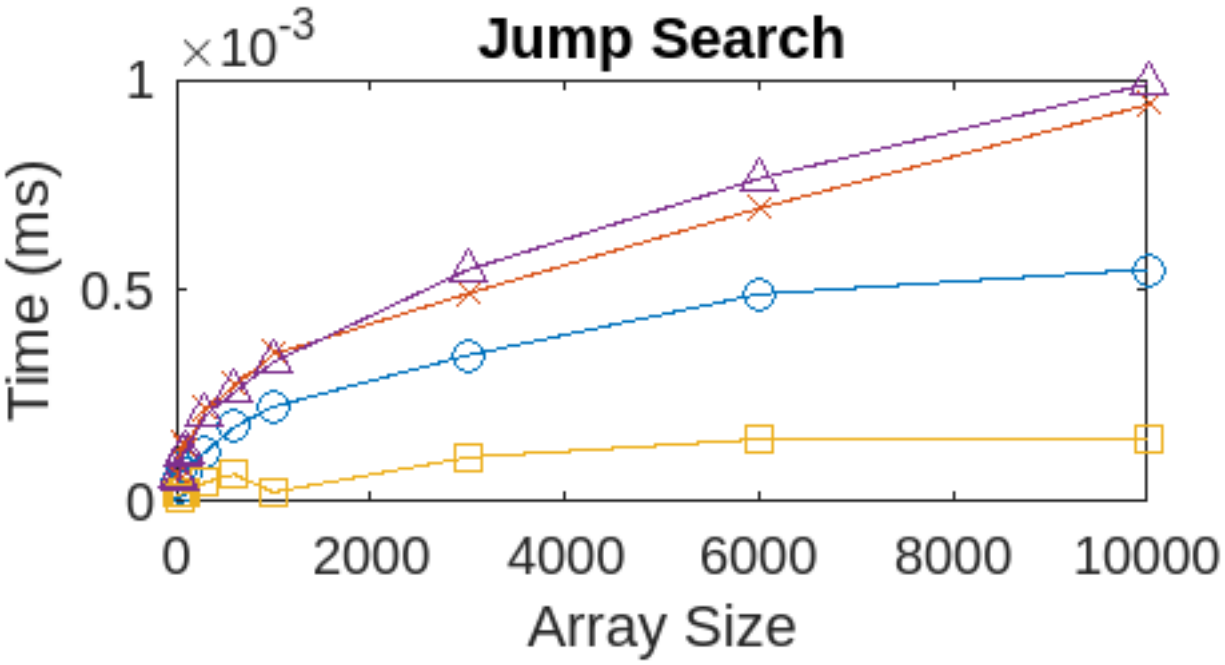
RecursiveLinearSearch

Key/ Time(ms)	10	30	60	100	300	600	1000	3000	6000	10000
First Key	7.73e-05	5.76e-05	0.0009262	0.0014598	0.0025866	0.0039829	0.0066631	0.0180044	0.0377098	0.0639199
Mid Key	4.67e-05	2.63e-05	0.0001264	0.0006236	0.0009573	0.0017232	0.0034976	0.0097955	0.0213068	0.0310417
Last Key	2.8e-05	2.7e-06	2.06e-05	4.38e-05	2.62e-05	3.2e-05	0.0008551	0.0019585	0.0050452	0.0033483
Non-existing Key	7.66e-05	7,00E-0	0.0007694	0.0011937	0.0022556	0.0039792	0.0067624	0.0202183	0.0425336	0.0696595



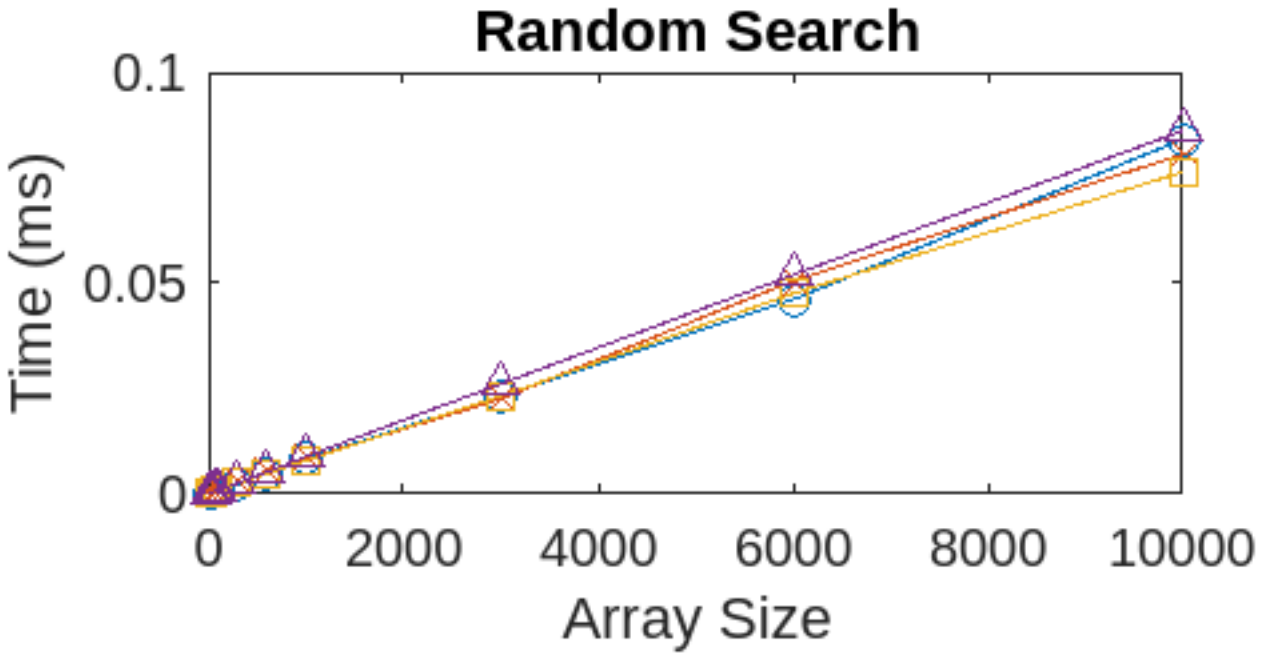
BinarySearch

Key/ Time(ms)	10	30	60	100	300	600	1000	3000	6000	10000
First Key	3.16e-05	1.34e-05	2.71e-05	2.16e-05	2.31e-05	2.7e-05	2.08e-05	2.87e-05	3.53e-05	3.59e-05
Mid Key	9.2e-06	1.28e-05	2.68e-05	2.65e-05	2.13e-05	1.9e-05	2.73e-05	2.47e-05	3.52e-05	3.97e-05
Last Key	2.8e-05	1.26e-05	2.77e-05	2.61e-05	2.11e-05	1.54e-05	2.3e-05	3,00E-05	3.47e-05	3.7e-05
Non-existing Key	4.48e-05	1.35e-05	2.84e-05	2.44e-05	2.45e-05	2.89e-05	2.67e-05	3.26e-05	3.85e-05	3.86e-05



JumpSearch

Key/ Time(ms)	10	30	60	100	300	600	1000	3000	6000	10000
First Key	2.16e-05	1.23e-05	2.16e-05	2.13e-05	4.6e-05	6.53e-05	2.34e-05	0.0001055	0.0001483	0.00014 6
Mid Key	4.46e-05	3.46e-05	8.36e-05	6.93e-05	0.0001184	0.0001774	0.0002258	0.0003472	0.0004919	0.0005506
Last Key	6.44e-05	5.14e-05	0.0001366	0.0001255	0.0002157	0.0002736	0.0003493	0.0004932	0.0006949	0.0009405
Non-existing Key	5.74e-05	5.14e-05	0.0001135	0.0001189	0.0002049	0.0002597	0.0003311	0.0005494	0.0007651	0.0009894



RandomSearch

Key/ Time(ms)	10	30	60	100	300	600	1000	3000	6000	10000
First Key	0.0001797	0.0003051	0.0008671	0.001075	0.0026652	0.0048607	0.0078385	0.0232576	0.0475629	0.0763922
Mid Key	0.0001075	0.0002989	0.0007644	0.0010915	0.0022926	0.0049896	0.0083628	0.0231901	0.0461277	0.0841642
Last Key	0.0001087	0.000275	0.0007619	0.0009817	0.0025411	0.0054022	0.0082686	0.0226143	0.0505299	0.0806171
Non-existing Key	0.0001143	0.0003081	0.0007103	0.0009959	0.0026461	0.0052405	0.008934	0.0261306	0.051981	0.0861673

Processor: Apple M2 Chip
RAM:8gb
Operating System: macOS 15.1

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

Linear Search

Best Case:
For linear search best case is when algorithm finds the key in first try which gives O(1) running time.

Average Case:
Average case is when the key is somewhere in the middle of the array it gives O(n) running time(because it is linearly dependent).

Worst Case:
Target element is at the last looked element or does not exist which gives again O(n) running time(because it is linearly dependent).

Recursive Linear Search

(Additional overhead because of the recursion.)

Best Case:
For linear search best case is when algorithm finds the key in first try which gives O(1) running time.

Average Case:
Average case is when the key is somewhere in the middle of the array it gives O(n) running time(because it is linearly dependent).

Worst Case:
Target element is at the last looked element or does not exist which gives again O(n) running time(because it is linearly dependent).

Binary Search

Best Case:
The target is the middle element and function finds it in first try which gives O(1) running time.

Average Case:

Because the algorithm eliminates half the elements on each step it results in a logarithmic number of steps. Which gives O(log2 (n)) running time.

Worst Case:
For the worst case scenario function still divides the elements needed to search so it again gives a logarithmic time complexity and a running time of O(log2(n)).

Jump Search

Best Case:
The target is found at the first jump so no more loops are needed which gives O(1) running time.

Average Case:
Because both the jumping and searching in the algorithm is proportional to sqrt(n) the running time for the algorithm on average case is O(sqrt(n)).

(For optimized searches m = sqrt(n) It balances the cost of jumps and the linear search after)

Worst Case:
Worst case is when algorithm jumps and searches for sqrt(n) times for last element and last search which gives O(sqrt(n)) running time.

Random Search

Best Case:
The function finds the required element on first try which gives O(1) time complexity.

Average Case:
The function checks for half the elements before finding the key which gives O(n) time complexity.

Worst Case:
Worst case is when the element is not in the array or is at the last looked index which again gives O(n) time complexity.

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.

The searches on average align with the theoretical ones but there are some deflections which may be because of various reasons such as processor architectures, cache misses, pipelining, system interruptions and background tasks working.

Linear Search

For my table and plots best case is when key is around first part, average case is when key is around middle element and worst case is when key does not exist. Just like the theoretical results which they agree on average.

Recursive Linear Search

My tables and plots agree with the theoretical ones again because the worst case is when key does not exist and best case is when key is at the last of the array which is because my recursive search starts from the last element in the array and searches through to the first element.

Binary Search

My plots graph show the non existing as worst case scenario and others are average with slight differences. Theoretical and observed results don't agree at array size 10 and around 300. The reason of this is because The keys are at random places at first middle and last indexes. The function gets inconsistent around small arrays. Of course there are reasons beyond the key location. Which I indicated such as processor architectures, cache misses, pipelining, system interruptions and background tasks working. With additional searches program would work like in the theoretical ones. The situation with larger array shows that my assumption is correct and needs larger sample size.

Jump Search

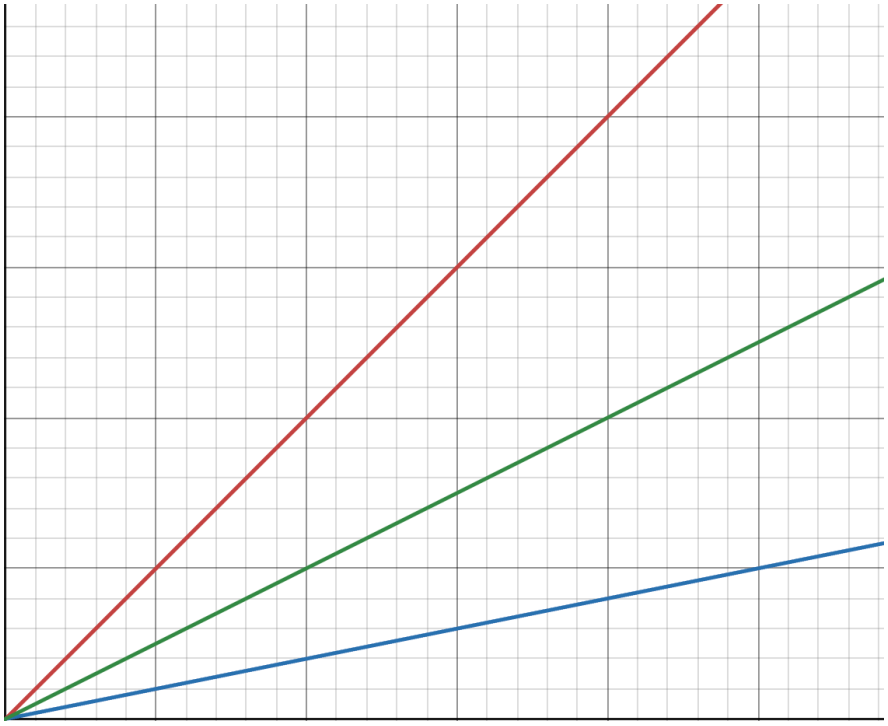
My plots and tables agree with the ones that are theoretical for best average and worst cases. Best case for my code is when element around the first index of the array which gives lower jumps and as the theoretical values are at the first jump point align with my results. For the average case time complexity of O(sqrt(n)) both at my table and the theoretical ones. The worst case scenario is when the element is at a point where it need sqrt(n) jumps and sqrt(n) searches and for my plots and graphs the worst case is when element either does not exist or is around the last element which confirms my plot's correct status. Again the keys are not at the exact best and worst cases but around them which shows a correlation and that is why the search is correct.

Random Search

Random search is at a almost perfect line which correlates with O(n) complexity. Best case scenario cannot be seen as it doesn't depend on the location of the key but the luck of algorithm. All cases show the average ones because of their random position and the random searches.

Plot theoretical running times for each algorithm across different scenarios and compare them to the plots at step 4. Comment on consistencies and inconsistencies.

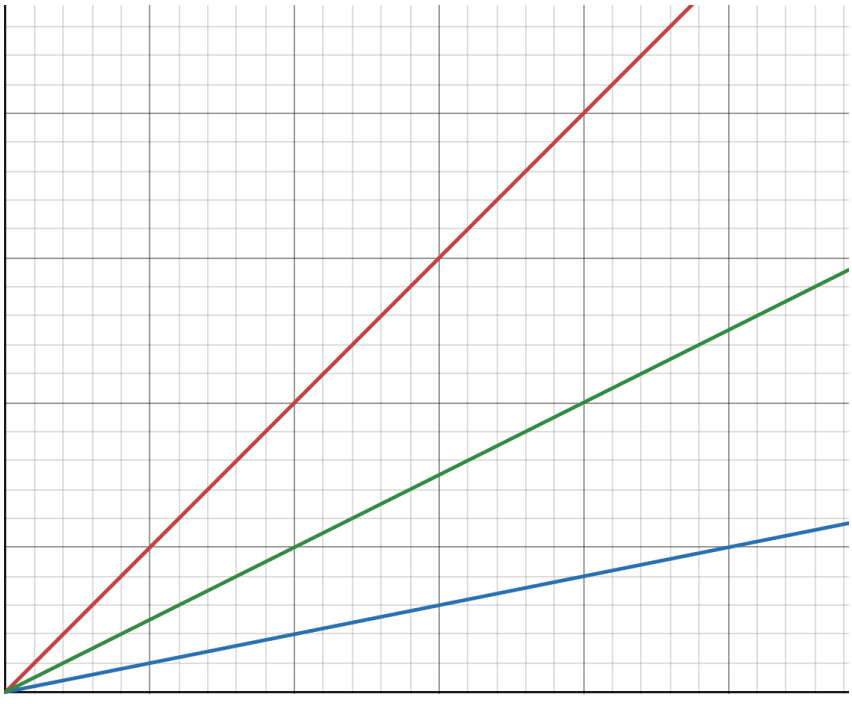
Linear Search



Red key around last or non existent
Green key at middle
Blue Key around first

While theoretical plot shows a perfect linear plot the one I made has some ups and downs even though it is generally the same again these problems can be attributed to processor architectures, cache misses, pipelining, system interruptions and background tasks working.

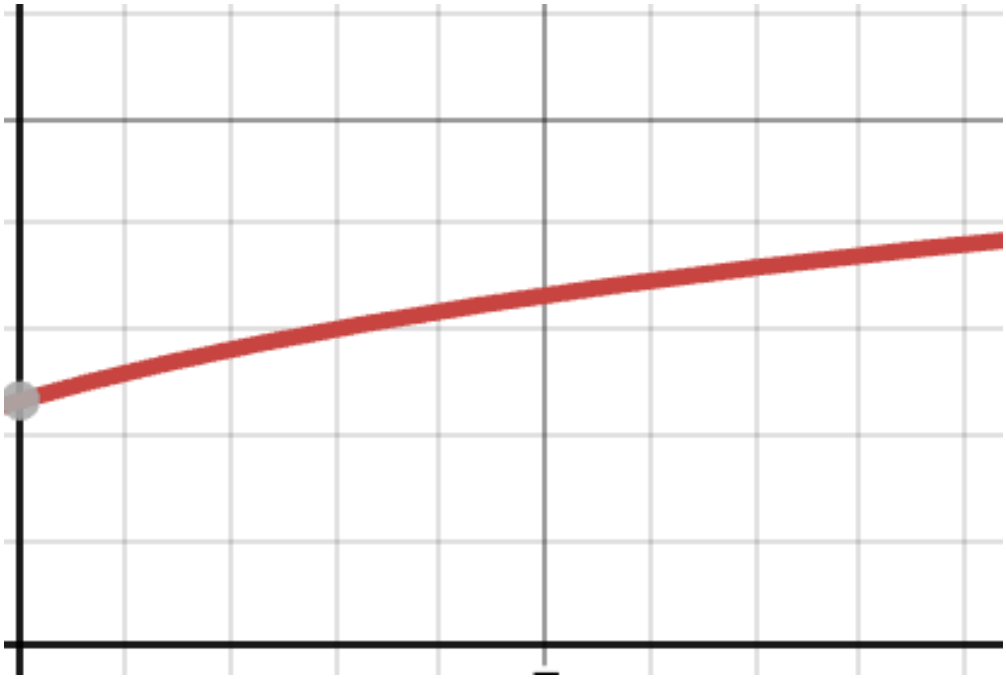
Linear Recursive Search



Red key around last looked element or non existent
Green key at middle
Blue Key around first looked

The theoretical plots and plots made by me are almost same with my plots having some deflection. The key position difference is because of reverse searching of key in my code which shows last element faster and first element slower again it shows consistency because of the search positions and where they start.

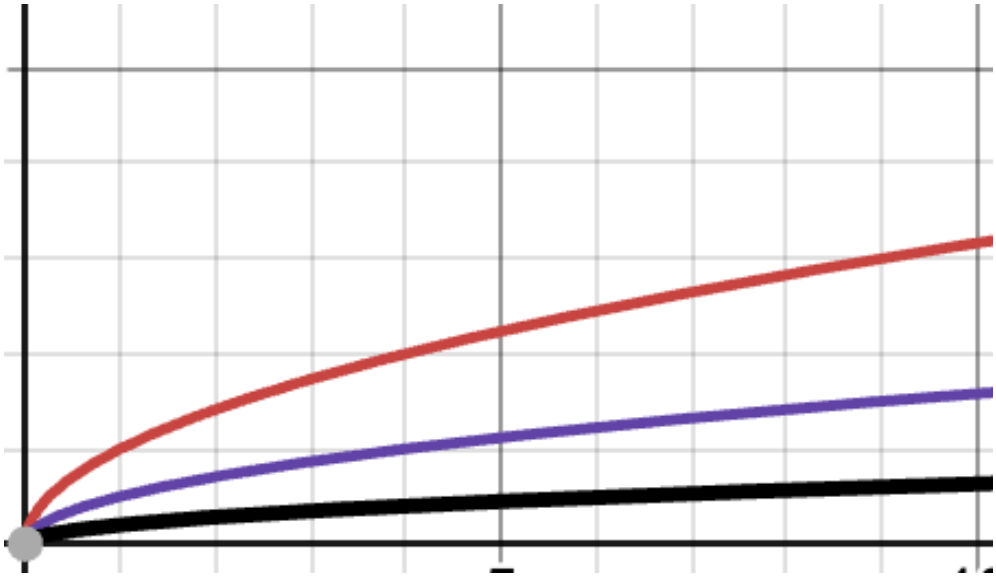
Binary Search



Red as key at all positions as they all give almost same answer(for around solutions)

First elements in my plot are distributed unevenly and create a inconstancy. That is because of the changeability of the function time on small arrays and with different arrays with different key used many times would give something like the theoretical which gives log2(n) time complexity. The reason for the mid and last elements going out of the expected plot at a certain point when array size is around 300 is because of the same reason. Of course there are the problems from outside sources like pc capacity, efficiency problems and others I talked about before.

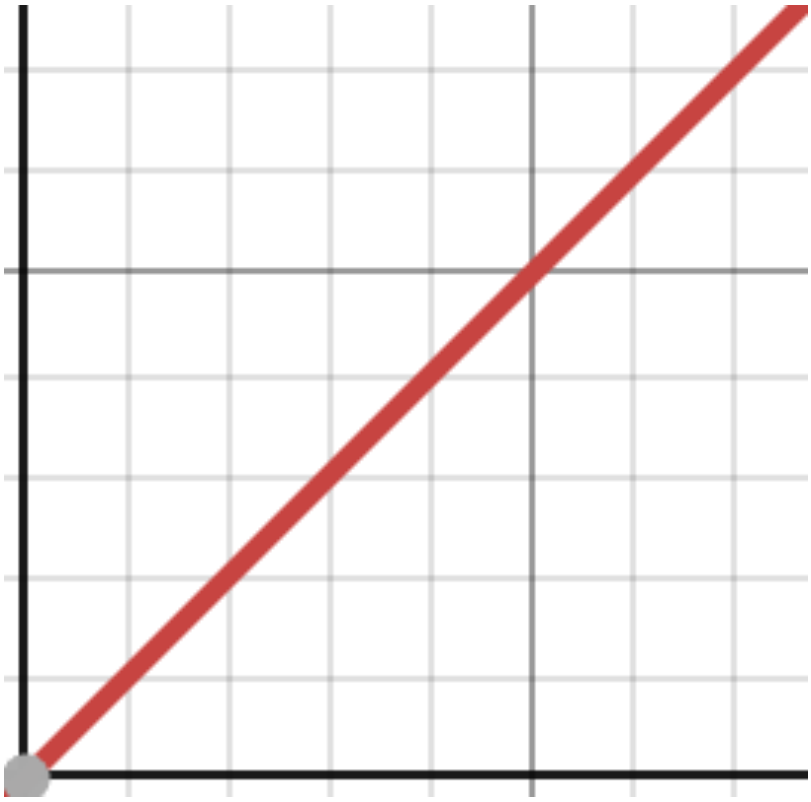
Jump Search



Red for key around the last element
Blue for around middle
Black for around first element

The plots are consistent and give a almost perfect alignment with some changes. There is 2 points where the time it took gets shorter with bigger arrays which is mainly because of the randomness of key choosing. They are at the key around first index and may be because the key is chosen really close to array start or to jump point. The insignificant shifts are because of the elements outside the code implementation, arrays and keys affecting the time it takes to search.

Random Search



Red for all situations because of the randomness

Both plots are almost perfectly aligned. The reason for the closeness of 4 key situations is that search is conducted in random manner and location of the key does not matter