HW2 Report CS 201 Sec.3
Ömer Can Baykara 22302436 16 Nov. 2024

Introduction

This report compares the runtimes of various search algorithms on sorted lists. It looks at the algorithms of binary search, jump search, linear search (both iterative and recursive), and random search. It considers different cases for keys searched for including when it is near the start of the list, near the middle of the list, near the end of the list, and not in the list. All of these cases are timed and will be analyzed.

The program running the search algorithms was written in C++, and the plots were drawn using Matplotlib. All the code can be found at https://github.com/omercanb/cs201hw2. All lists used in the program are int arrays.

Notes on Jump Search and Random Search: The block size for jump search is taken as $\sqrt{n}$. Random search is implemented by first creating a sorted list of indexes, shuffling it, and then searching through those indexes.

Methodology

Creating Sorted Lists

For a list of size n, the numbers in the list were chosen from a range of $[0, 10n]$. To create the lists, n numbers in the range of $[0, 10n]$ were chosen and sorted. These numbers were chosen using a random number generater which was seeded by a constant so all algorithms would be run on the same list. Different constants were tested to see that there would be no difference in outcome.

Choosing Keys

To choose keys that were near the start, middle, and end, an index was chosen uniformly from a range of size $\lfloor \frac{n}{10} \rfloor$, around the required point. The ranges were specifically chosen as follows:

Index near the start: $[0, \lfloor \frac{n}{10} \rfloor]$,

Index near the middle: $[\lfloor \frac{9n}{20} \rfloor, \lfloor \frac{11n}{20} \rfloor]$,

Index near the end: $[\lfloor \frac{9n}{10} \rfloor, n-1]$.

For the key not in the list, a random number in the range of $[0, 10n]$ until a number not in the list was found.

Tables of Runtimes

The different cases of runtimes are marked by letters a through d, they represent, in order the runtimes for the key being, near the beginning, near the middle, near the end, and not in the list. The runtimes given are in nanoseconds. n is the size of the list. For rows with an n value but no runtimes, either the runtime was too long or a stack overflow occurred (Recursive Linear Search). These values were obtained by running the algorithm multiple times, anywhere from 1000 to 100000, and taking the mean.

Table 1. Table of All Runtimes

n	Binary Search				Jump Search				Linear Search				Recursive Linear Search				Random Search			
	a	b	c	d	a	b	c	d	a	b	c	d	a	b	c	d	a	b	c	d
1	3.2x10^{1}	3.0x10^{1}	2.9x10^{1}	2.8x10^{1}	5.6x10^{1}	5.6x10^{1}	6.8x10^{1}	7.7x10^{1}	2.8x10^{1}	2.8x10^{1}	2.8x10^{1}	2.5x10^{1}	2.8x10^{1}	2.7x10^{1}	2.8x10^{1}	2.5x10^{1}	3.1x10^{2}	3.1x10^{2}	3.1x10^{2}	3.8x10^{2}
5	3.9x10^{1}	3.3x10^{1}	4.8x10^{1}	4.5x10^{1}	2.9x10^{1}	5.1x10^{1}	4.6x10^{1}	6.4x10^{1}	2.5x10^{1}	3.0x10^{1}	3.5x10^{1}	3.6x10^{1}	2.6x10^{1}	3.3x10^{1}	3.8x10^{1}	4.0x10^{1}	1.0x10^{3}	1.0x10^{3}	1.0x10^{3}	1.0x10^{3}
10	5.0x10^{1}	5.9x10^{1}	5.9x10^{1}	5.1x10^{1}	5.0x10^{1}	7.0x10^{1}	6.8x10^{1}	7.1x10^{1}	2.7x10^{1}	4.1x10^{1}	5.1x10^{1}	4.3x10^{1}	2.7x10^{1}	4.8x10^{1}	7.1x10^{1}	6.2x10^{1}	1.9x10^{3}	1.8x10^{3}	1.8x10^{3}	1.9x10^{3}
50	7.2x10^{1}	7.3x10^{1}	7.6x10^{1}	7.9x10^{1}	8.0x10^{1}	1.0x10^{2}	1.0x10^{2}	1.1x10^{2}	3.2x10^{1}	1.1x10^{2}	1.8x10^{2}	1.9x10^{2}	3.4x10^{1}	1.4x10^{2}	4.8x10^{2}	5.0x10^{2}	8.3x10^{3}	8.3x10^{3}	8.2x10^{3}	8.4x10^{3}
100	8.3x10^{1}	8.2x10^{1}	8.1x10^{1}	9.7x10^{1}	8.5x10^{1}	1.1x10^{2}	1.3x10^{2}	1.1x10^{2}	3.9x10^{1}	1.9x10^{2}	3.4x10^{2}	3.6x10^{2}	4.8x10^{1}	4.9x10^{2}	9.3x10^{2}	9.6x10^{2}	1.6x10^{4}	1.6x10^{4}	1.6x10^{4}	1.6x10^{4}
500	1.1x10^{2}	1.1x10^{2}	1.1x10^{2}	1.3x10^{2}	1.3x10^{2}	1.7x10^{2}	2.2x10^{2}	1.8x10^{2}	1.1x10^{2}	8.8x10^{2}	1.6x10^{3}	1.7x10^{3}	2.3x10^{2}	2.7x10^{3}	5.1x10^{3}	5.3x10^{3}	7.3x10^{4}	7.3x10^{4}	7.3x10^{4}	7.4x10^{4}
1000	1.2x10^{2}	1.3x10^{2}	1.2x10^{2}	1.5x10^{2}	1.5x10^{2}	2.2x10^{2}	2.8x10^{2}	2.4x10^{2}	1.9x10^{2}	1.7x10^{3}	3.3x10^{3}	3.4x10^{3}	4.9x10^{2}	5.3x10^{3}	1.0x10^{4}	1.1x10^{4}	1.4x10^{5}	1.4x10^{5}	1.4x10^{5}	1.5x10^{5}
5000	1.6x10^{2}	1.6x10^{2}	1.5x10^{2}	1.8x10^{2}	2.7x10^{2}	4.1x10^{2}	5.6x10^{2}	4.2x10^{2}	8.7x10^{2}	8.5x10^{3}	1.6x10^{4}	1.7x10^{4}	2.8x10^{3}	2.8x10^{4}	5.6x10^{4}	5.9x10^{4}	7.2x10^{5}	7.2x10^{5}	7.2x10^{5}	7.4x10^{5}
10000	1.7x10^{2}	1.8x10^{2}	1.7x10^{2}	1.9x10^{2}	3.6x10^{2}	5.5x10^{2}	7.6x10^{2}	5.6x10^{2}	1.7x10^{3}	1.7x10^{4}	3.2x10^{4}	3.4x10^{4}	5.4x10^{3}	5.7x10^{4}	1.1x10^{5}	1.2x10^{5}	1.4x10^{6}	1.4x10^{6}	1.4x10^{6}	1.5x10^{6}
50000	2.0x10^{2}	2.1x10^{2}	2.0x10^{2}	2.2x10^{2}	7.7x10^{2}	1.2x10^{3}	1.7x10^{3}	1.3x10^{3}	8.5x10^{3}	8.4x10^{4}	1.6x10^{5}	1.7x10^{5}	2.8x10^{4}	3.0x10^{5}	5.6x10^{5}	5.9x10^{5}	7.2x10^{6}	7.2x10^{6}	7.2x10^{6}	7.4x10^{6}
100000	2.2x10^{2}	2.3x10^{2}	2.2x10^{2}	2.2x10^{2}	9.9x10^{2}	1.7x10^{3}	2.3x10^{3}	1.7x10^{3}	1.7x10^{4}	1.7x10^{5}	3.2x10^{5}	3.4x10^{5}	5.6x10^{4}	5.9x10^{5}	1.1x10^{6}	1.2x10^{6}	1.4x10^{7}	1.4x10^{7}	1.4x10^{7}	1.5x10^{7}
500000	2.6x10^{2}	2.7x10^{2}	2.7x10^{2}	2.4x10^{2}	2.3x10^{3}	3.8x10^{3}	5.2x10^{3}	3.8x10^{3}	8.1x10^{4}	8.4x10^{5}	1.6x10^{6}	1.7x10^{6}								
1000000	2.9x10^{2}	3.1x10^{2}	3.0x10^{2}	2.6x10^{2}	3.2x10^{3}	5.3x10^{3}	7.3x10^{3}	5.3x10^{3}	1.7x10^{5}	1.8x10^{6}	3.4x10^{6}	3.6x10^{6}								
5000000	3.6x10^{2}	3.7x10^{2}	3.4x10^{2}	2.8x10^{2}	7.0x10^{3}	1.2x10^{4}	1.7x10^{4}	1.2x10^{4}												

Table. 2. Runtimes of Binary Search

n	Binary Search			
	a	b	c	d
1	$3.2x10^1$	$3.0x10^1$	$2.9x10^1$	$2.8x10^1$
5	$3.9x10^1$	$3.3x10^1$	$4.8x10^1$	$4.5x10^1$
10	$5.0x10^1$	$5.9x10^1$	$5.9x10^1$	$5.1x10^1$
50	$7.2x10^1$	$7.3x10^1$	$7.6x10^1$	$7.9x10^1$
100	$8.3x10^1$	$8.2x10^1$	$8.1x10^1$	$9.7x10^1$
500	$1.1x10^2$	$1.1x10^2$	$1.1x10^2$	$1.3x10^2$
1000	$1.2x10^2$	$1.3x10^2$	$1.2x10^2$	$1.5x10^2$
5000	$1.6x10^2$	$1.6x10^2$	$1.5x10^2$	$1.8x10^2$
10000	$1.7x10^2$	$1.8x10^2$	$1.7x10^2$	$1.9x10^2$
50000	$2.0x10^2$	$2.1x10^2$	$2.0x10^2$	$2.2x10^2$
100000	$2.2x10^2$	$2.3x10^2$	$2.2x10^2$	$2.2x10^2$
500000	$2.6x10^2$	$2.7x10^2$	$2.7x10^2$	$2.4x10^2$
1000000	$2.9x10^2$	$3.1x10^2$	$3.0x10^2$	$2.6x10^2$
5000000	$3.6x10^2$	$3.7x10^2$	$3.4x10^2$	$2.8x10^2$

Table. 3. Runtimes of Jump Search

n	Jump Search			
	a	b	c	d
1	$5.6x10^1$	$5.6x10^1$	$6.8x10^1$	$7.7x10^1$
5	$2.9x10^1$	$5.1x10^1$	$4.6x10^1$	$6.4x10^1$
10	$5.0x10^1$	$7.0x10^1$	$6.8x10^1$	$7.1x10^1$
50	$8.0x10^1$	$1.0x10^2$	$1.0x10^2$	$1.1x10^2$
100	$8.5x10^1$	$1.1x10^2$	$1.3x10^2$	$1.1x10^2$
500	$1.3x10^2$	$1.7x10^2$	$2.2x10^2$	$1.8x10^2$
1000	$1.5x10^2$	$2.2x10^2$	$2.8x10^2$	$2.4x10^2$
5000	$2.7x10^2$	$4.1x10^2$	$5.6x10^2$	$4.2x10^2$
10000	$3.6x10^2$	$5.5x10^2$	$7.6x10^2$	$5.6x10^2$
50000	$7.7x10^2$	$1.2x10^3$	$1.7x10^3$	$1.3x10^3$
100000	$9.9x10^2$	$1.7x10^3$	$2.3x10^3$	$1.7x10^3$
500000	$2.3x10^3$	$3.8x10^3$	$5.2x10^3$	$3.8x10^3$
1000000	$3.2x10^3$	$5.3x10^3$	$7.3x10^3$	$5.3x10^3$
5000000	$7.0x10^3$	$1.2x10^4$	$1.7x10^4$	$1.2x10^4$

Table. 4. Runtimes of Linear Search

n	Linear Search			
	a	b	c	d
1	$2.8x10^1$	$2.8x10^1$	$2.8x10^1$	$2.5x10^1$
5	$2.5x10^1$	$3.0x10^1$	$3.5x10^1$	$3.6x10^1$
10	$2.7x10^1$	$4.1x10^1$	$5.1x10^1$	$4.3x10^1$
50	$3.2x10^1$	$1.1x10^2$	$1.8x10^2$	$1.9x10^2$
100	$3.9x10^1$	$1.9x10^2$	$3.4x10^2$	$3.6x10^2$
500	$1.1x10^2$	$8.8x10^2$	$1.6x10^3$	$1.7x10^3$
1000	$1.9x10^2$	$1.7x10^3$	$3.3x10^3$	$3.4x10^3$
5000	$8.7x10^2$	$8.5x10^3$	$1.6x10^4$	$1.7x10^4$
10000	$1.7x10^3$	$1.7x10^4$	$3.2x10^4$	$3.4x10^4$
50000	$8.5x10^3$	$8.4x10^4$	$1.6x10^5$	$1.7x10^5$
100000	$1.7x10^4$	$1.7x10^5$	$3.2x10^5$	$3.4x10^5$
500000	$8.1x10^4$	$8.4x10^5$	$1.6x10^6$	$1.7x10^6$
1000000	$1.7x10^5$	$1.8x10^6$	$3.4x10^6$	$3.6x10^6$
5000000				

Table. 5. Runtimes of Recursive Linear Search

n	Recursive Linear Search			
	a	b	c	d
1	$2.8x10^1$	$2.7x10^1$	$2.8x10^1$	$2.5x10^1$
5	$2.6x10^1$	$3.3x10^1$	$3.8x10^1$	$4.0x10^1$
10	$2.7x10^1$	$4.8x10^1$	$7.1x10^1$	$6.2x10^1$
50	$3.4x10^1$	$1.4x10^2$	$4.8x10^2$	$5.0x10^2$
100	$4.8x10^1$	$4.9x10^2$	$9.3x10^2$	$9.6x10^2$
500	$2.3x10^2$	$2.7x10^3$	$5.1x10^3$	$5.3x10^3$
1000	$4.9x10^2$	$5.3x10^3$	$1.0x10^4$	$1.1x10^4$
5000	$2.8x10^3$	$2.8x10^4$	$5.6x10^4$	$5.9x10^4$
10000	$5.4x10^3$	$5.7x10^4$	$1.1x10^5$	$1.2x10^5$
50000	$2.8x10^4$	$3.0x10^5$	$5.6x10^5$	$5.9x10^5$
100000	$5.6x10^4$	$5.9x10^5$	$1.1x10^6$	$1.2x10^6$
500000				
1000000				
5000000				

Table. 6. Runtimes of Random Search

n	Random Search			
	a	b	c	d
1	$3.1\text{x}10^2$	$3.1\text{x}10^2$	$3.1\text{x}10^2$	$3.8\text{x}10^2$
5	$1.0\text{x}10^3$	$1.0\text{x}10^3$	$1.0\text{x}10^3$	$1.0\text{x}10^3$
10	$1.9\text{x}10^3$	$1.8\text{x}10^3$	$1.8\text{x}10^3$	$1.9\text{x}10^3$
50	$8.3\text{x}10^3$	$8.3\text{x}10^3$	$8.2\text{x}10^3$	$8.4\text{x}10^3$
100	$1.6\text{x}10^4$	$1.6\text{x}10^4$	$1.6\text{x}10^4$	$1.6\text{x}10^4$
500	$7.3\text{x}10^4$	$7.3\text{x}10^4$	$7.3\text{x}10^4$	$7.4\text{x}10^4$
1000	$1.4\text{x}10^5$	$1.4\text{x}10^5$	$1.4\text{x}10^5$	$1.5\text{x}10^5$
5000	$7.2\text{x}10^5$	$7.2\text{x}10^5$	$7.2\text{x}10^5$	$7.4\text{x}10^5$
10000	$1.4\text{x}10^6$	$1.4\text{x}10^6$	$1.4\text{x}10^6$	$1.5\text{x}10^6$
50000	$7.2\text{x}10^6$	$7.2\text{x}10^6$	$7.2\text{x}10^6$	$7.4\text{x}10^6$
100000	$1.4\text{x}10^7$	$1.4\text{x}10^7$	$1.4\text{x}10^7$	$1.5\text{x}10^7$
500000				
1000000				
5000000				

Plots

Now the data of each table is plotted. Both the X and Y scales are logarithmic. They were plotted this way because consecutive values of n increased logarithmically. The range of the X-axis is $[0, 1.5\times10^7]$ and the range of the Y-axis is $[0, 5.0\times10^7]$. An exception is for binary search where to observe the shape there is a plot with a smaller Y-axis range.

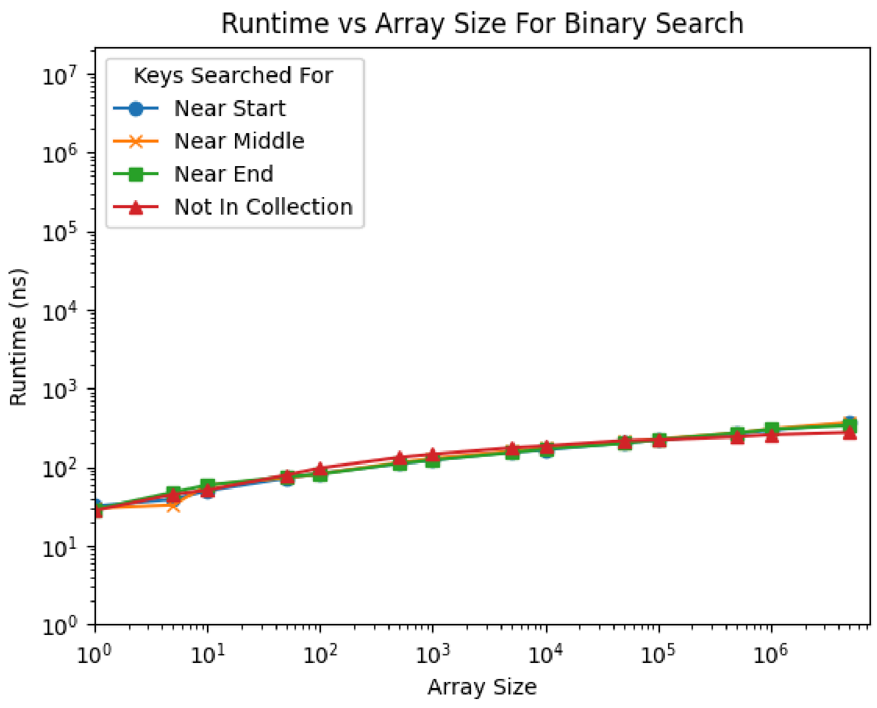


Fig. 1. Plot for Binary Search

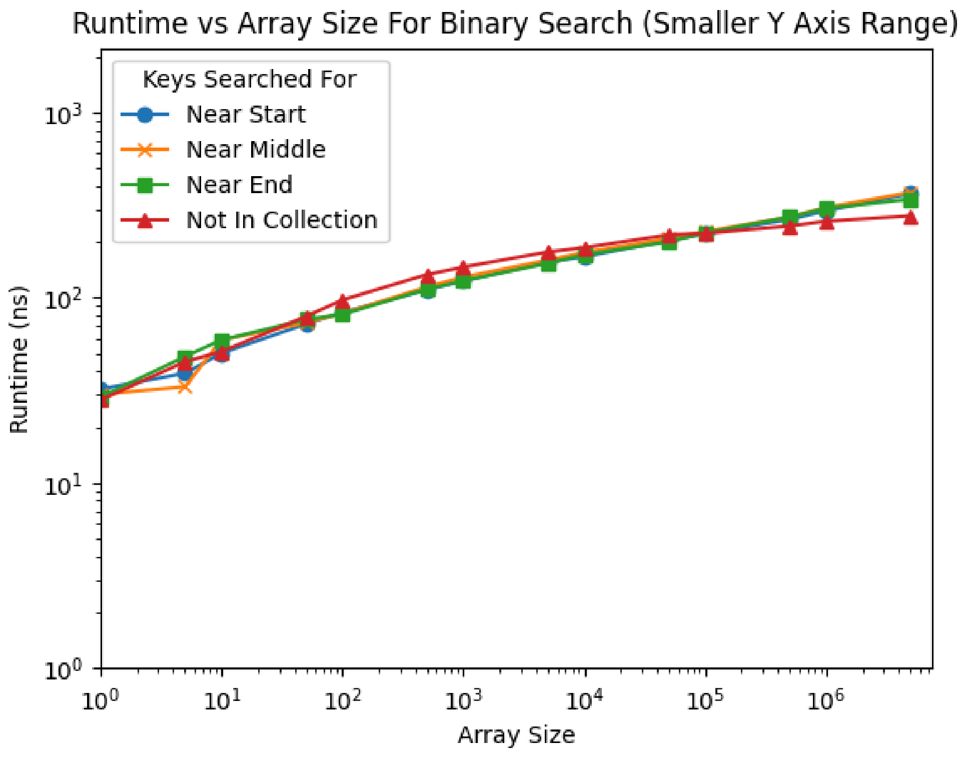


Fig. 2. Plot for Binary Search With a Smaller Y-Axis Range

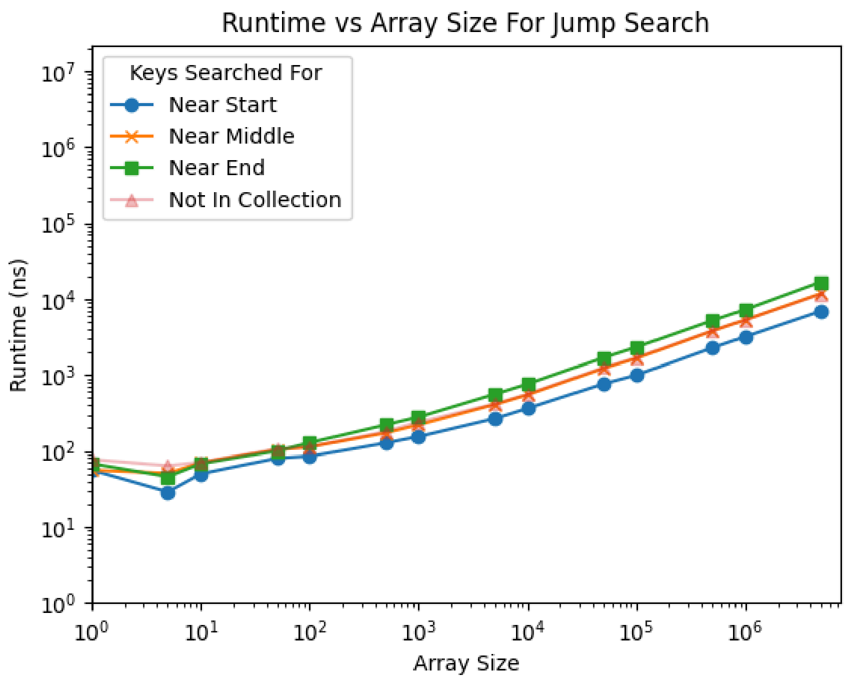


Fig. 3. Plot for Jump Search
(Near Middle and Not In Collection overlap)

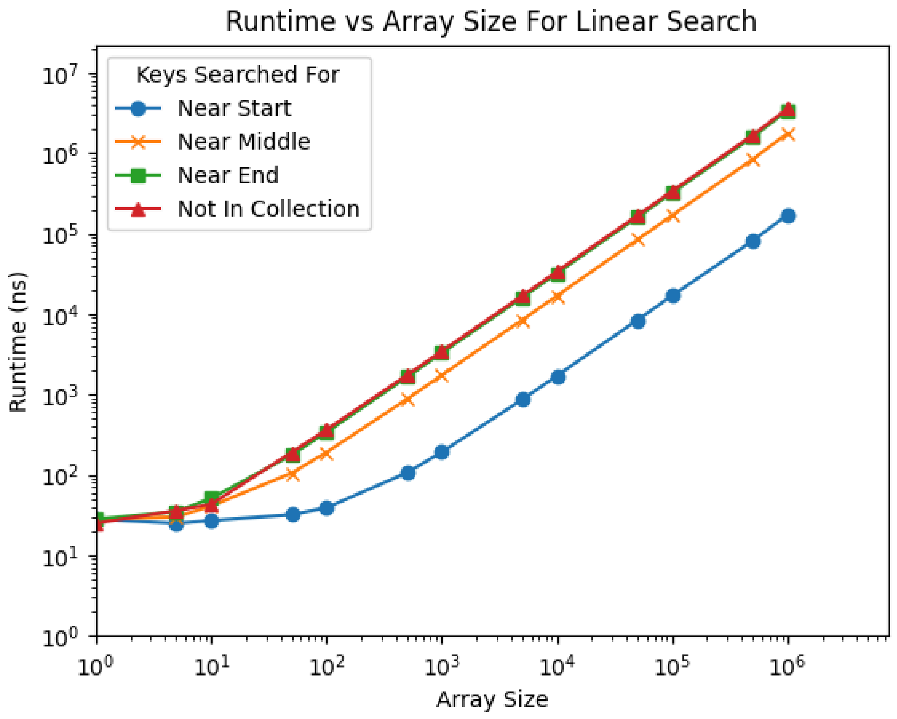


Fig. 4. Plot for Linear Search

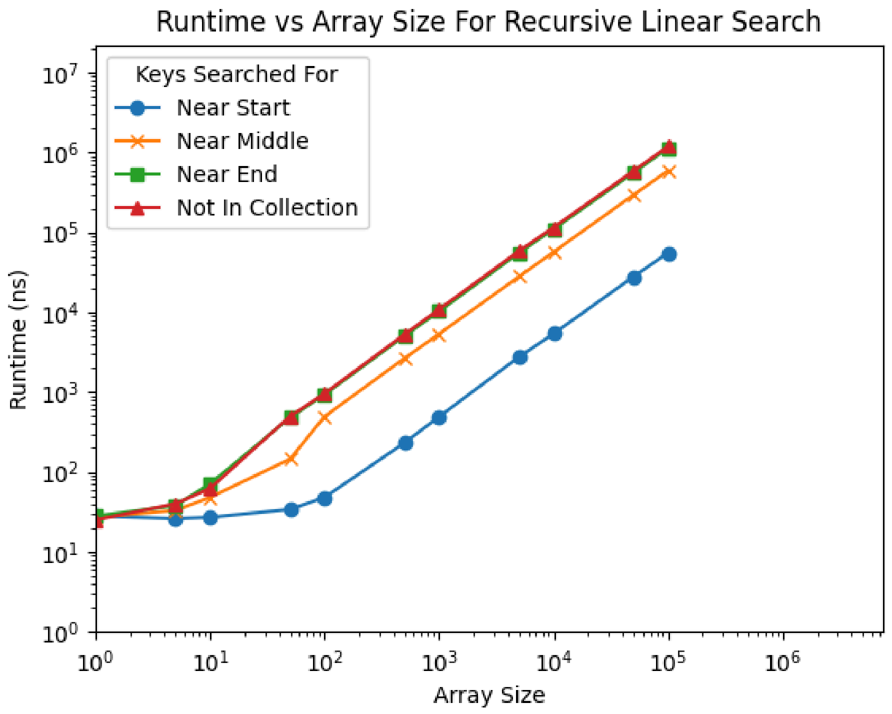


Fig. 5. Plot for Linear Search (Recursive)

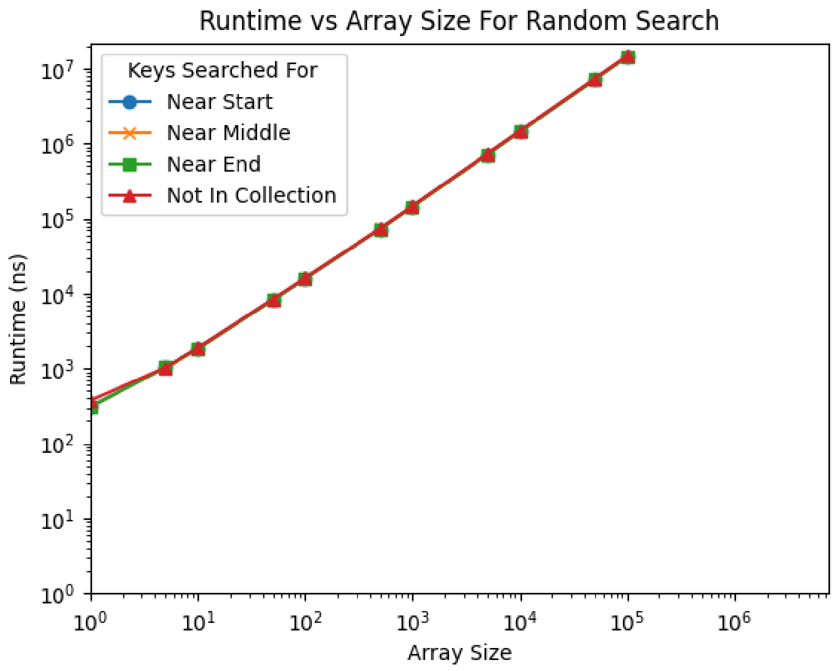


Fig. 6. Plot for Random Search

Comments on Results

Computer Specification

The program was run on a Macbook M2 Pro 2023, with a 10-core Apple M2 Pro chip and 16 GB of ram.

Theoretical Analysis of Each Algorithm

This section will look at the best, average, and worst-case runtimes of the algorithms.

Binary Search

Best Case: When the key is in the middle of the list, the algorithm finds it in one operation, thus, it is $O(1)$ time.

Average Case: For any key, in or out of the lists the algorithm is $O(logn)$ time.

Worst Case: For a key not in the list, the algorithm needs to do the maximum number of comparisons, but still runs in $O(logn)$ time.

Jump Search

For block size = $\sqrt{n}$*.*

Best Case: When the key is at the end of the first block the algorithm checks, it is $O(1)$ time.

Average Case: When the key is in the middle of a block, the algorithm does $O(\sqrt{n})$ operations to get to the block and $O(\sqrt{n})$ operations to find the key, for a total of $O(\sqrt{n})$ time.

Worst Case: Same as the average case.

Linear Search

Best Case: When the key is at the start of the lists, the algorithm finds it in one operation, thus, it is $O(1)$ time.

Average Case: $\frac{n}{2}$ indices are checked before finding the key, so it is $O(n)$ time.

Worst Case: The key is at the end or not in the list, so n indices are checked, therefore it is $O(n)$ time.

Recursive Version: It is expected that the time complexity will be $cO(n)$, where $O(n)$ is the time complexity of the iterative version and c is a constant greater than 1, caused by the overhead time cost of calling functions.

Random Search

For this algorithm, the time taken to populate a sorted array of indices and shuffle it is included, so there will be + $O(n)$ time for each case. (The exact number of steps taken for random number generation is not known, so it will be kept as $O(n)$)

Best Case: The first index checked is correct, so it is $O(1) + O(n) = O(n)$ time.

Average Case: It takes $\frac{n}{2}$ tries to guess the correct index, thus it is $O(n)$ time.

Worst Case: The key is not in the list so n tries are made, running in $O(n)$ time.

Comparison of Plots With the Theoretical Cases

A Note on Best and Worst Cases

In the theoretical best and worst cases, the situation where a key is at the start or end of a list comes up often. For the case of the key choices in this report, it should be considered that the expected position of a key near the beginning also scales with n. A key near the beginning is on average at position $\frac{n}{20}$, and the key near the end is at $\frac{19n}{20}$.

Binary Search

According to the plots, the choice of key does not change the runtime. For the key near the middle, it is rarely in the exact middle, thus it is expected for the plot to not be $O(1)$. It can be observed that all choices of keys give a $O(logn)$ time complexity. (It should be noted that this is the shape of a logarithmic function when both scales are logarithmic). Thus, between the choices of key used here, there is no best or worst case for binary search, which matches the theoretical results.

Jump Search

The choices of keys can be ordered from the fastest runtime to the slowest: near the start, near the middle, and near the end/not in the list. For these plots, the slope of the line is identical, and there is a vertical shift, creating the difference. This is expected as the number of keys to be checked differs by a constant factor, they are as follows:

(The first term is the average number of blocks to look through to find the key, and the second term, is the average number of keys to compare in the block, These are equal to the time complexity.)

For the key near the start: $\frac{n/20}{\sqrt{n}} + \frac{\sqrt{n}}{2} = \frac{11\sqrt{n}}{20}$,

The key near the middle:$\frac{\sqrt{n}}{2} + \frac{\sqrt{n}}{2} = \sqrt{n}$,

The key near the end: $\frac{19n/20}{\sqrt{n}} + \frac{\sqrt{n}}{2} = \frac{29\sqrt{n}}{20}$,

The key not in the list: $\frac{\sqrt{n}}{2} + \frac{\sqrt{n}}{2} = \sqrt{n}$, for a key in the range of the list ($[0, 10n]$).

It should be noted that for a plot where both scales are logarithmic, the multiplication of the function by a constant creates a vertical shift. Positive numbers less than one create a downward shift and the shift gets larger the closer to zero the number gets.

The shapes of the plots match the theoretical value because they all have an $O(\sqrt{n})$ shape, this can be understood by observing that their slope is half that of the slope of the plot of linear search. Then, their difference in vertical shift can be explained by their respective constant factors.

Linear Search

The choices of key from fastest runtime to slowest are near the start, near the middle, and near the end/not in the list. The number of keys expected for the algorithm to check based on the choice of key, and thus the expected time complexity is as follows.

For the key near the start: $\frac{n}{20}$,

The key near the middle: $\frac{n}{2}$,

The key near the end: $\frac{19n}{20}$,

The key not in the list: n.

The shape of the plots matches the theoretical value, as all choices of keys have the same slope of about 1 in the plot, meaning that they are all in $O(n)$ time. Furthermore, their vertical shifts can be explained by the constant factors found above.

The plot for the recursive version goes to a smaller value of X, as a stack overflow occurs above the value of n=500,000 on the computer the program was run on. Additionally, it can be observed that the lines of the plot are similar to the lines of iterative linear search, but shifted up, which matches the theoretical prediction of $cO(n)$, where $O(n)$ is the time complexity of the iterative version and c is a constant greater than 1. To calculate an approximate value for c we can equate a ratio of the time complexities of recursive and iterative linear sort, with their runtimes. Let us take the case where the key is not in the list for example. This will look like:

$$\frac{cn}{n} = \frac{T_{n\,recursive}}{T_{n\,iterative}}.$$

Using a large value for n like value $n = 5000$, we get: $c = 5.9/1.7 = 3.3$.

Random Search

In talking about the time complexity, the term cn will be used to represent the time spent creating a shuffled index list. It is expected that for the key near the start, near the middle, and near the end, the time complexity would be on average $\frac{n}{2} + cn$ as it takes on average $\frac{n}{2}$ tries to locate the correct key. For the key not in the list, it's time complexity should be $n + cn$. But the plots overlap and in the tables the values are also similar. This means that the constant $c >> \frac{1}{2}$, as the increase created by c overpowers the difference of n and $\frac{n}{2}$. To calculate an approximate value for c we can take the points where $n = 1$ from the tables of linear search and random search. For both, we use the time for when the key was not in the list, and we write a ratio of time complexity with runtime, getting:

$$\frac{1+c}{1} = \frac{400}{25},$$

$$c = 15.$$

From Looking at the starting points of the graphs of random search and linear search, we caThis is an $O(n)$ function, explaining the near 1 slope, and the added cn term explains the upward shift.

Plots of the Calculated Time Complexity for Different Keys

Now the functions found in terms of n above will be plotted, in the same format as the runtime plots above, considering the steps taken by the algorithm when the keys are near the start, near the middle, near the end, and not in the list. For these plots, the constants for recursive linear search and random search were chosen as 5.5 and 5.1.

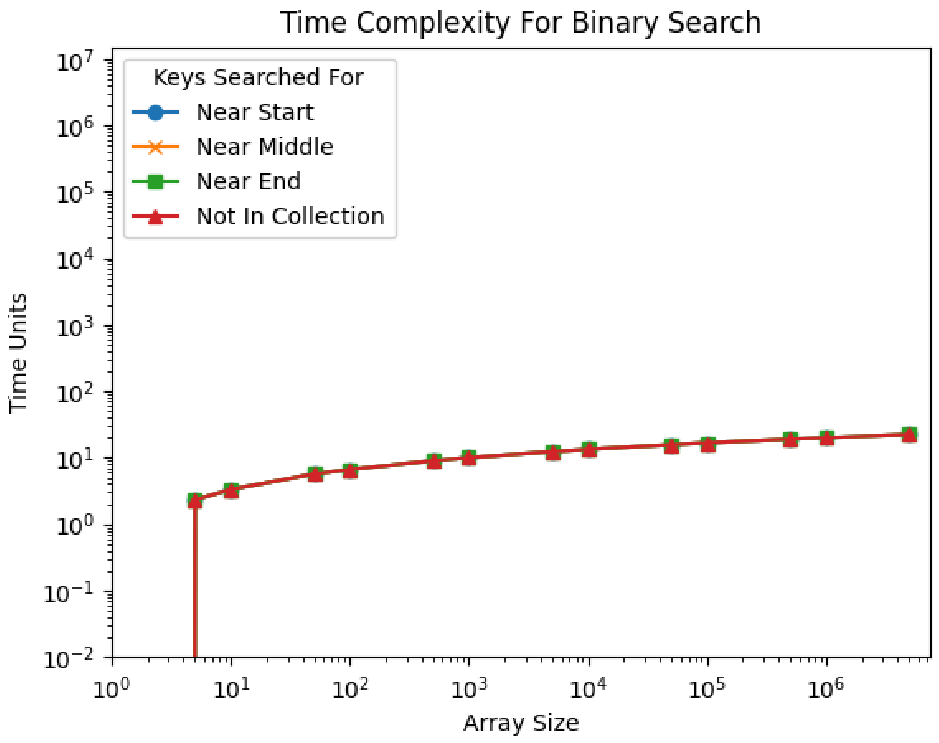


Fig. 7. Time Complexity Plot For Binary Search

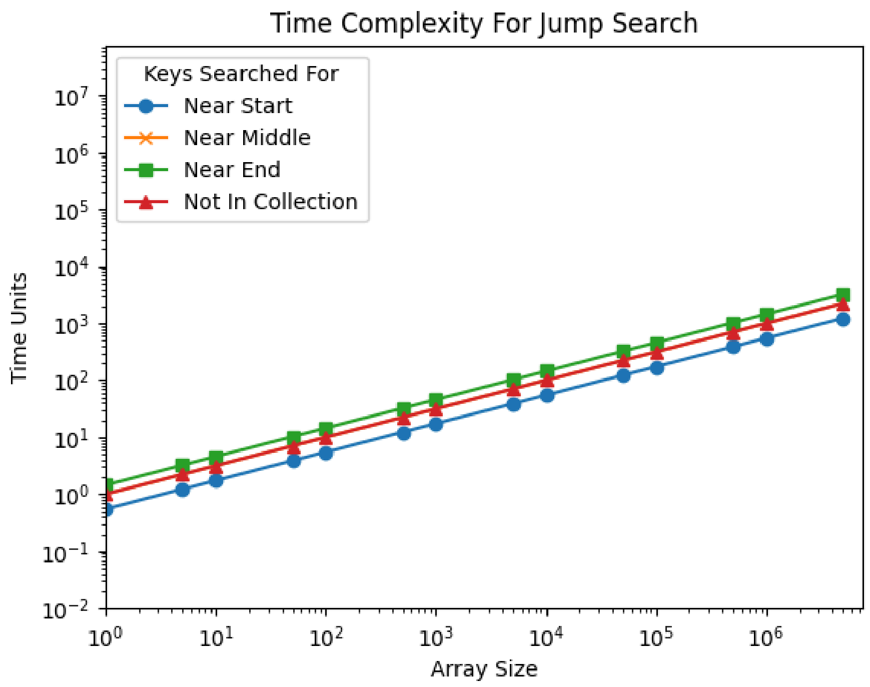


Fig. 8. Time Complexity Plot For Jump Search
(Near Middle and Not In Collection overlap)

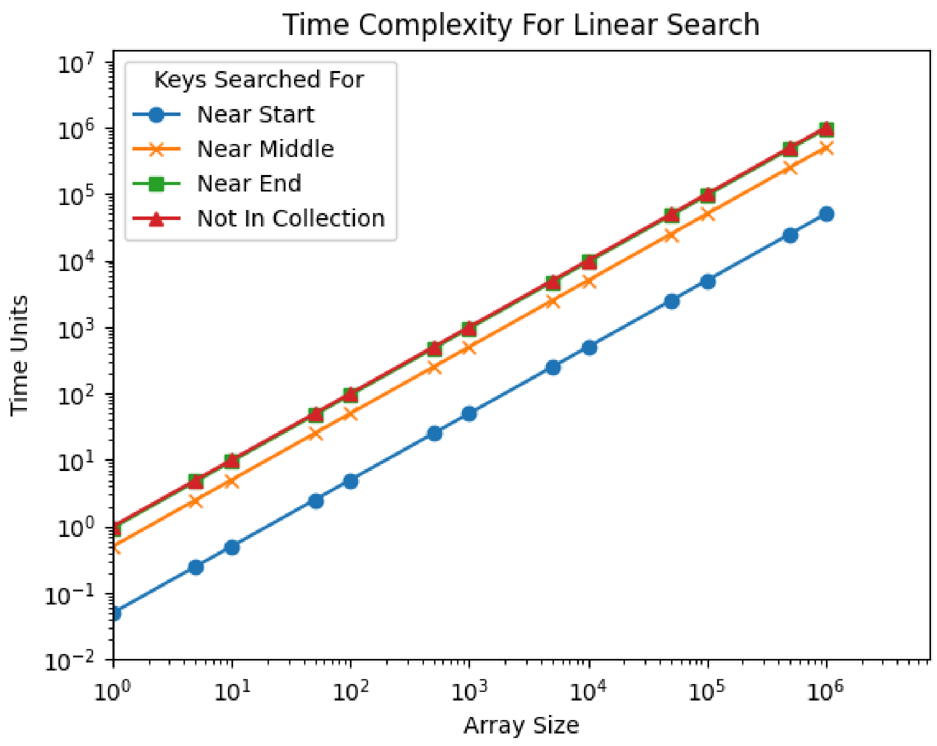


Fig. 9. Time Complexity Plot For Linear Search

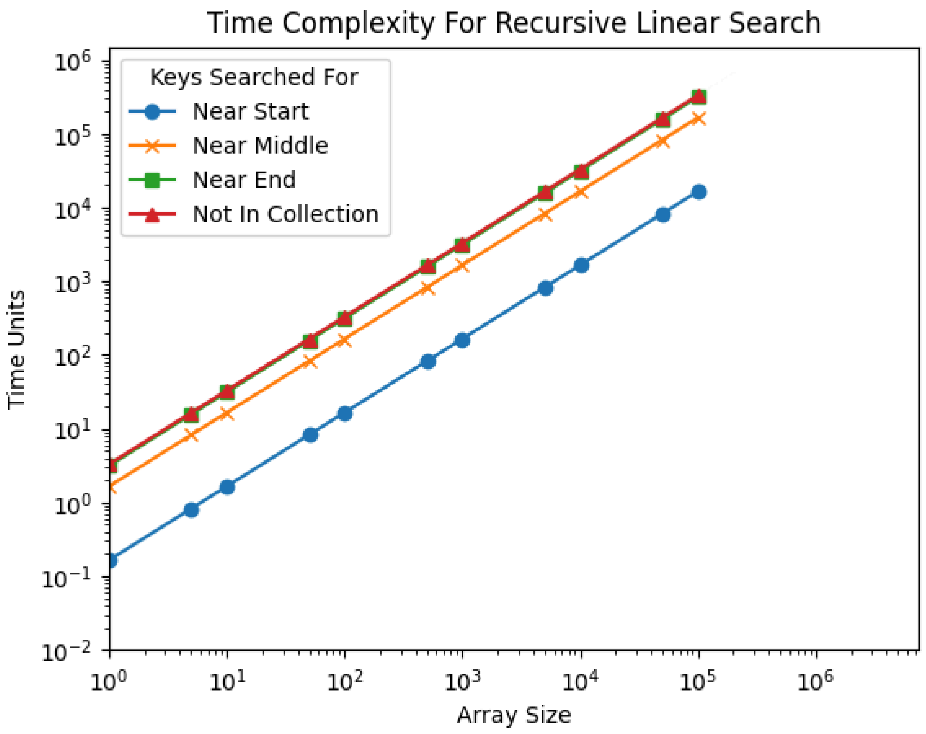
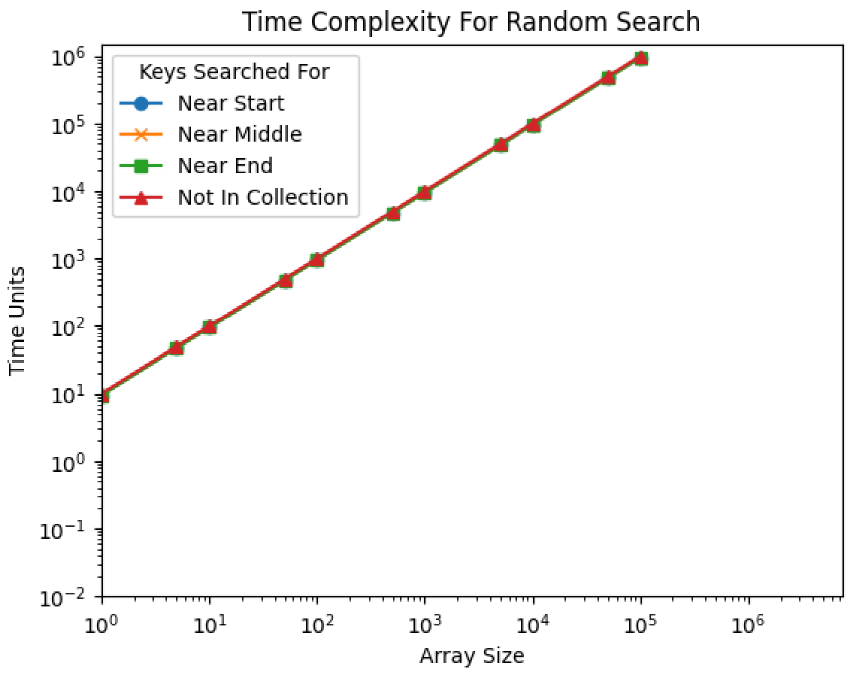


Fig. 10. Time Complexity Plot



For Recursive Linear Search

Fig. 11. Time Complexity Plot For Random Search

Comments on the Time Complexity Plots

The shapes of all plots match exactly. An interesting thing to note is that all plots with real-time are about a factor of 20 larger than their time complexity counterparts, which could show that each unit of time in the context of this experiment is about 20ns. The reason why the binary search plot seems to have a vertical asymptote at n = 1 is because the logarithm of the value is 0, which is a vertical asymptote on a logarithmic plot. There are no inconsistencies in the graphs, showing that all the calculations done for the time complexity of the algorithms in each case are true. The constant chosen for random and recursive linear search are also values which makes the time complexity plots resemble their runtime plots, which is evidence that the values chosen were close to the real values.

Bibliography

Hunter, D. John. “Matplotlib: Visualization with Python.” *Matplotlib*. Version 3.7.1, 2024, https://matplotlib.org.

Wikipedia contributors. "Jump search." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 19 Jul. 2024. Web. 18 Nov. 2024.