# Data Structures and Algorithms 1 Presentation

Ethan Hastie

# The Problem

- We need an algorithm that is able to **search** through a collection of data and find out if that value exists.
- We have three collections of arrays with the first array holding **numbers 1 – 100**, the second **1 – 1,000** and the latter containing **1 – 10,000**.
- The arrays are not ordered since they are filled from numbers to whatever the size of the array is.

    - for e.g. if we wanted an array filled with 5 elements, then each of those elements would be hold a number from 1 to 5 (1,2,3,4,5).

# The Problem

- We can use **Linear Searching** to find out if the value exists – just iterate through the array and compare each element, if they are the same return true…
- But what if we are working with a large size collection?
- Then, **Binary Search** would be more viable – just keep splitting the array into smaller arrays until we find our intended value, return the middle value or else return false if it isn't found.
- Binary searching is extremely useful, especially if our value is in some obscure position inside the collection, or in the very middle of the list – just split the array and the value is found!

# Why implement a searching algorithm?

- Searching through a large collection can be extremely difficult as you have to individually look at each item and find out if that is your intended value – this process could take minutes, hours even, especially if you are dealing with a large amount of data.
- A searching algorithm can be much more efficient in finding the data for you in a matter of seconds or less.
- It can save you a lot of time having to do it on your own.

# The Problem

- Binary search has rendered linear search completely obsolete because it takes **less time** to find the required element.
- The only prerequisite for binary search is that the collection must be sorted, while linear search doesn't need to be.
  - It doesn't matter which order (ASC or DESC).
- I want to compare the efficiency of these two searching algorithms
- Given a set of items we want to find, I also want to scale the size of our search (increasing the size of the array from 100 to 1,000 to finally 10,000) and time how long it takes to find the required items within these collections.
- This will therefore increase the amount of items we wish to search for.

# The Problem

- In order to demonstrate which algorithm would be more efficient, I conducted 3 tests and measured how long it took the algorithm to find each of the items we wanted over **101 iterations** and timed how long it takes.

    - the reason I chose 101 instead of 100 was to make it easier to do the **box and whiskers charts** I will show later on (median, etc.) – our data is skewed and therefore it is an accurate indicator of the typical value.

    - I also plotted variance and standard deviation to test the spread and show if there was any significance between the recorded times.

- Therefore we can get an idea of the **typical behaviour** of these algorithms when searching through 3 differently sized arrays, using multiple iterations.
- I wrote all the times it took for each algorithm to find the values to separate files

# Tests

- As mentioned in the previous slide, I wrote my timings to separate csv. files to keep my data organised.
- The test files were appropriately named…

    - e.g. [the name of the test]_[name of algorithm].csv (test1a_linear indicates that linear search was used to search for number 29 inside a 100 size array)

- Test 1: having an array labelled '**small**' with **size 100**

    - test1a: number 29

    - test1b: number 67

# Tests

- Test 2: having an array labelled '**medium**' with **size 1,000**
  - test2a: number 333
  - test2b: number 961
- Test 3: having an array labelled '**large**' with **size 10,000**
  - test3a: number 1191
  - test3b:  number 9979
- In the program, I measured my timings in **nanoseconds** because when I was trying to output the number of milliseconds it displayed 0 ms. – not helpful.
- Therefore, I converted my times in nanoseconds to milliseconds within an excel worksheet that I had the results in for each of my tests.

# Data Structures Implementation

- When I first started writing my code, my original plan was to have the searched array set up as a **vector** because I wanted to separate myself from using c-style arrays that I always hated using in other modules.
- Vectors are good because they can **dynamically resize in memory**, therefore we can store as many elements as we want. The drawback of this data structure is that it occupies **more memory**, which could **slow down** the execution of program.
- Vectors have a **higher time complexity** than arrays, especially in this case because we are wanting to search through the array and it is quite time consuming to access each element based on the position it is in. Using a vector would mean that when I am populating my array, insertion would be **0(N)**.

- We need easy-as-we-can access to each element in the array, which is imperative to our algorithm – efficient access.
- Therefore, I used the **c-style array** which is O(N).

```
buffer[0] = '\0';
// section here was implemented to try to fix annoying warnings
/* Section end */
unsigned int small[smallsize]; // our small array (1 - 100)
unsigned int medium[mediumsize]; // our medium array (1 - 1,000)
unsigned int large[largesize]; // our large array (1 - 10,000)
std::list<int> linear_times; // list for our times for our linear search
std::list<int> binary_times; // list for our times for our binary search
```

Here I am declaring my three arrays where I have specified their data type and the size they are holding. I know how many elements each array will store and this is constant. I am not actually adding any elements because I don't need to. If I were using vectors then I would have to be certain that the storage I need would be a lot. Since I know the arrays are to be populated with 1 to 'size_of_array' then arrays are more appropriate.

Arrays take the same time to traverse as a list in an best case scenario O(N)

# Data Structures Implementation

- Another data structure I used was **lists**.
- I used lists to **store the timings** recorded over the 101 iterations.
- Lists are used in my program because I could do extra functions with them such as '**sort()**' and '**clear()**' rather than having to create two extra functions that would do the same if I were using arrays.
- They are also much better at inserting items compared to arrays. I also don't need direct access to a specific element within the list because it all gets outputted to a csv. file anyway.
- A drawback of lists is that they consume more memory to keep the linking information associated to each element.

```
101             time_l = duration_cast<nanoseconds>(end_l - start_l).count();
102             //cout << time_l << " ns." << endl; // displays the time in na
103             times.push_back(time_l); // add this time to a std::list that
104         }
105     }
106     //cout << "\n";
107     ofstream l_times(file_name); // make a csv. file with a passed in name
108     l_times << "Item, No. (not iterations), Times(ns.)" << endl; // make c
109     times.sort(); // sort the times (needed for median and graphs later)
110     for (auto ltime : times) { // use ltime for syntatic sugar for our tim
111         l_times << value << "," << identifier << "," << ltime << endl; //
112         identifier++; // increment identifier
113     }
114     destroy_array(times); // erase all the elements from the times list sc
115     // continue and move onto the next test
```

All I have to do is use the public member function ‘push_back()’ and that will insert the time into the list. push_back() is O(1) because it is taking the same time regardless to push the time into the list.

I used the ‘sort()’ member function that will sort my list. That prevented me from writing a function that will sort the list for me. Sort() is O(N log N) where N is the size of the list.

I also implemented a ‘destroy_array’ function which clears all the elements from the list and displays the contents (I was a bit paranoid). Inside destroy_array I used ‘clear()’ which is O(1).

# Theoretical Performance

| Algorithm | Best case | Average Case | Worst Case |
|---|---|---|---|
| Linear Search | O(1)<br>Item is in the first in the list | O(N) | O(N)<br>Item could be at the end of the array or N comparisons are required |
| Binary Search | O(1)<br>Item is the very center element | O(log N)<br>Item is in a reasonable position in either the left or right subarray | O(log N)<br>Item could be the very first element or the neighbour element next to the center or can conclude after only $\log_2 N$ comparisons |

# Tests

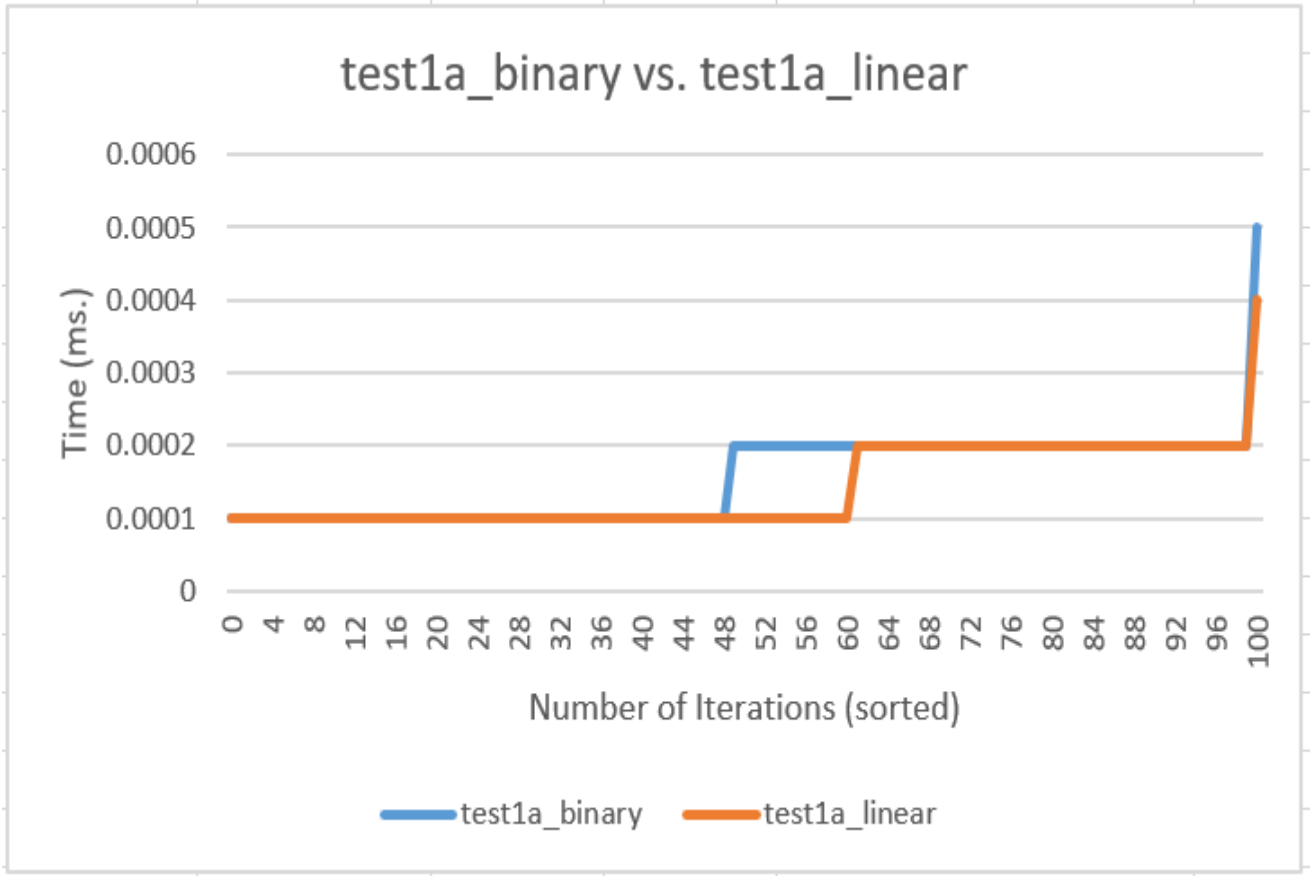
- I used line graphs to display the results of my times in milliseconds. The x-axis line indicates the **number of iterations** in a sorted order with the y-axis showing the **time in milliseconds**.
- I also used box plots to display my results.
- I calculated the mean, variance and standard deviation on all the tests and displayed it in a bar chart.
- To make the results as fair as possible, I had the 2 algorithms search for **2 different values** – one number in the **left side** of the array and another in the **right**.
- The purpose for this was to give linear search a fair chance to demonstrate its efficiency and this is clear in the very first test I did.

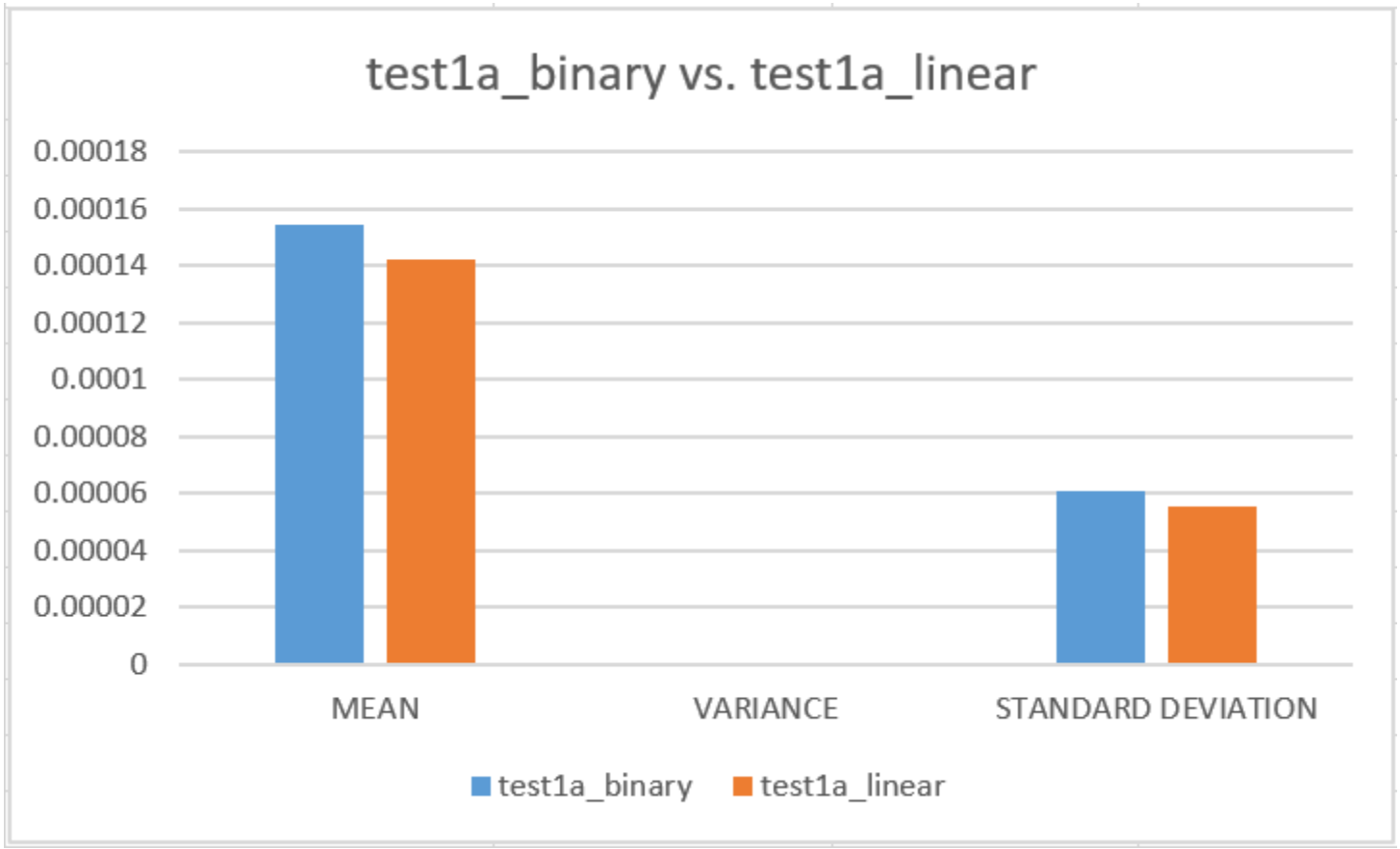
# Sample output for my program

| Item | No. (not iterations) | Time(ms.) |
|---|---|---|
| 333 | 0 | 0.0001 |
| 333 | 1 | 0.0001 |
| 333 | 2 | 0.0001 |
| 333 | 3 | 0.0001 |
| 333 | 4 | 0.0001 |
| 333 | 5 | 0.0001 |
| 333 | 6 | 0.0001 |
| 333 | 7 | 0.0001 |
| 333 | 8 | 0.0001 |
| 333 | 9 | 0.0001 |
| 333 | 10 | 0.0001 |

The time (ms.) was originally ns.

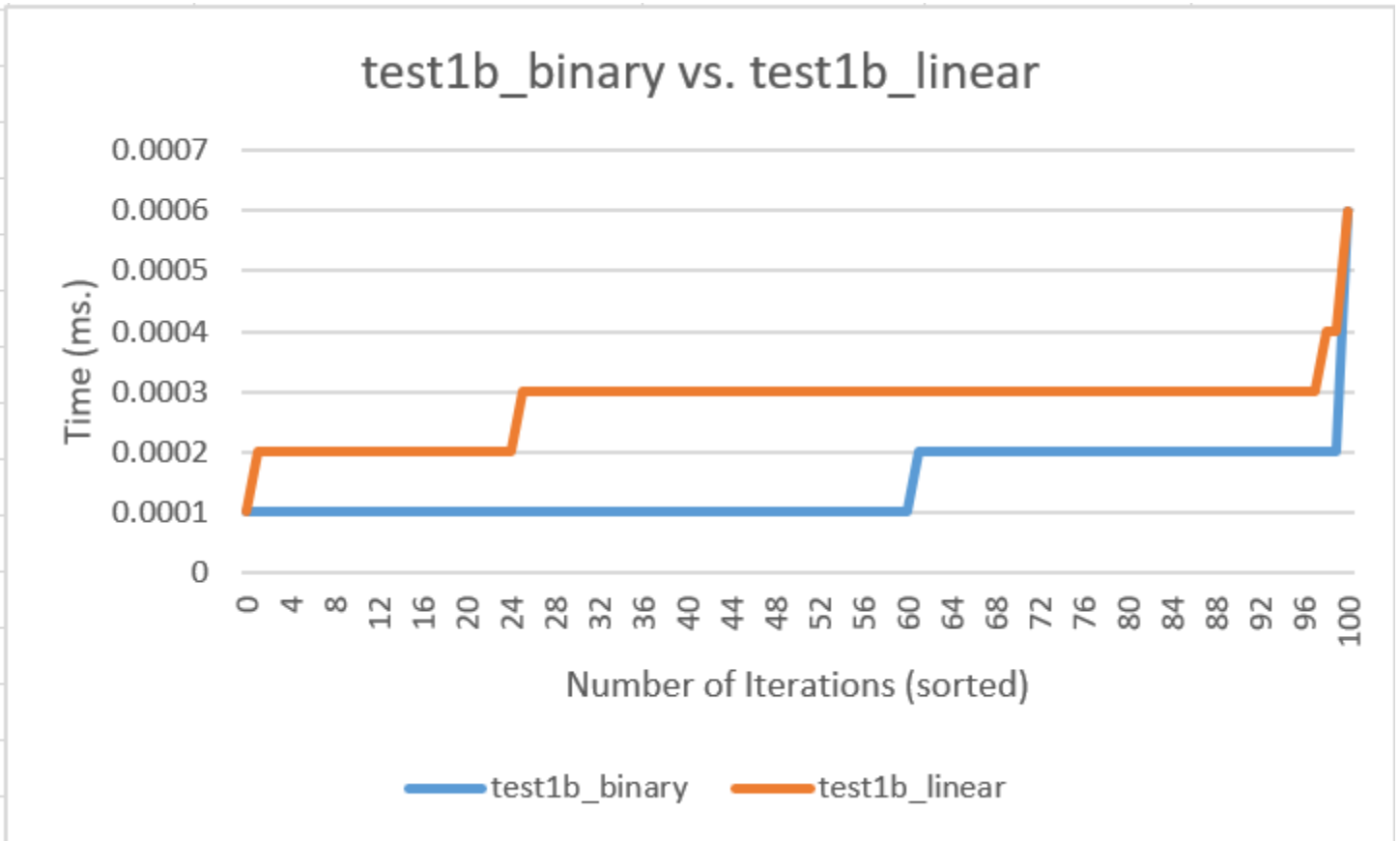
test1a_linear, test1a_binary - search for 29 inside an array of 100 elements.

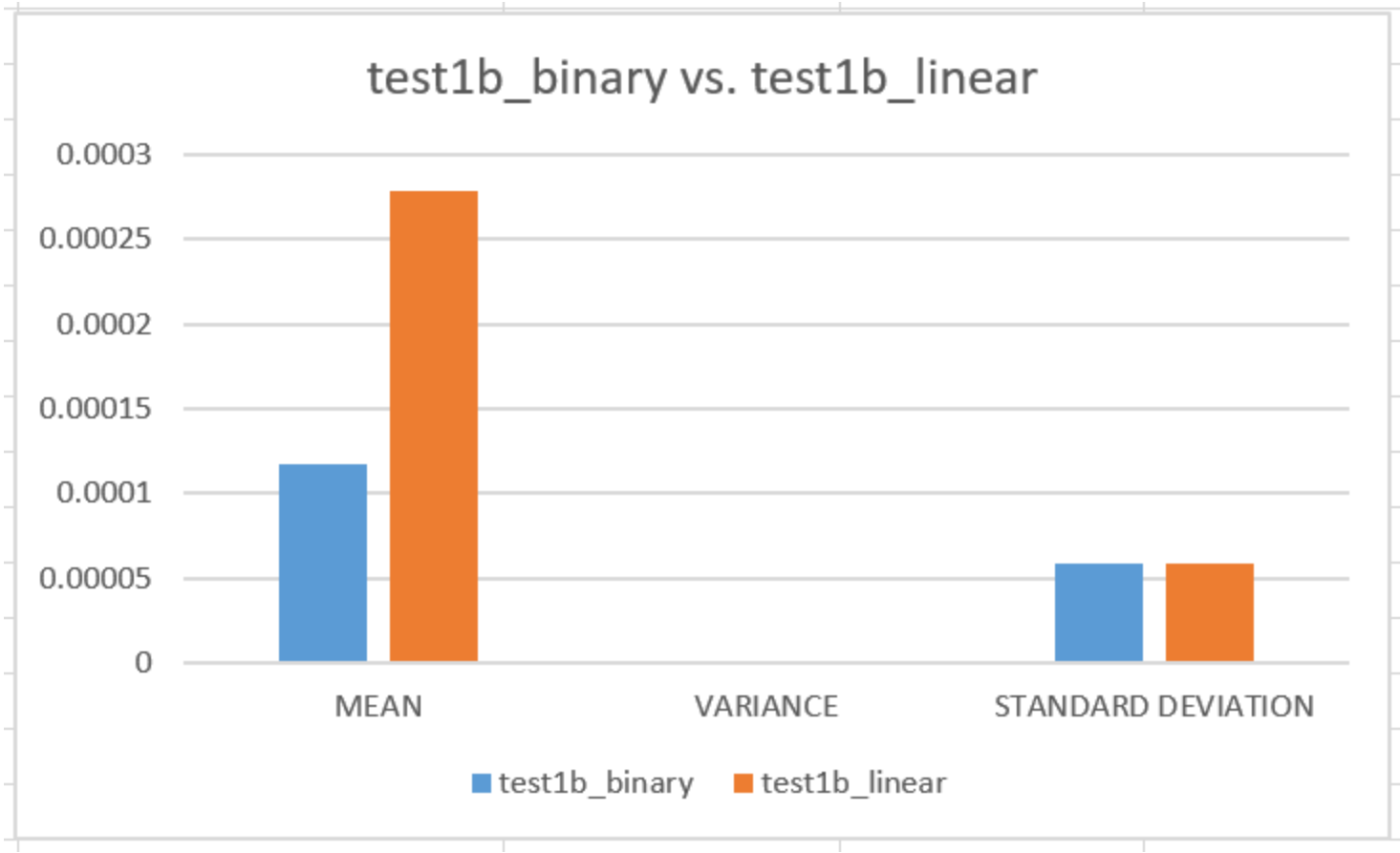




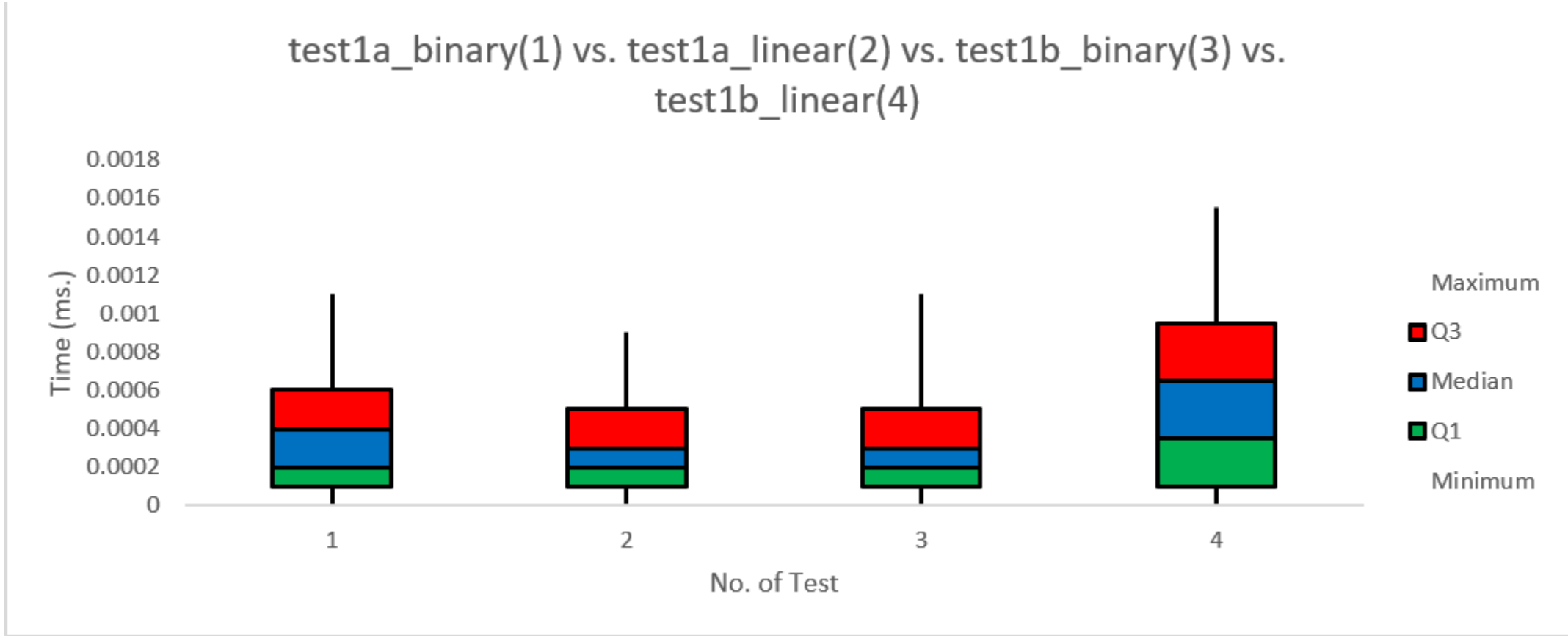

| | test1a_binary | test1a_linear |
|---|---|---|
| MEAN | 0.00015466 | 0.000142 |
| VARIANCE | 0.00000000370495 | 0.00000000305347 |
| STANDARD DEVIATION | 0.0000608683 | 0.0000552582 |

test1b_linear, test1b_binary - search for 67 inside an array of 100 elements.



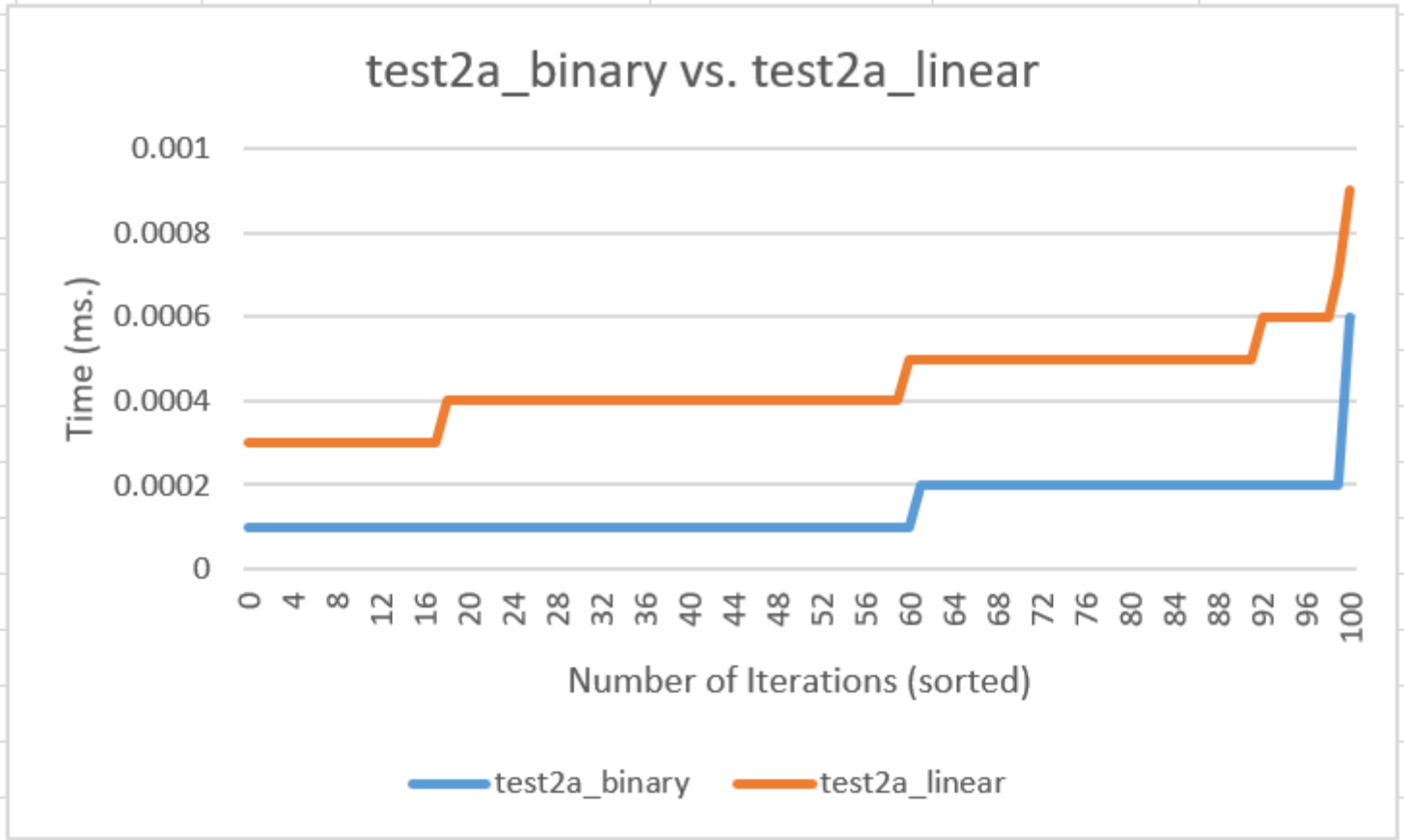


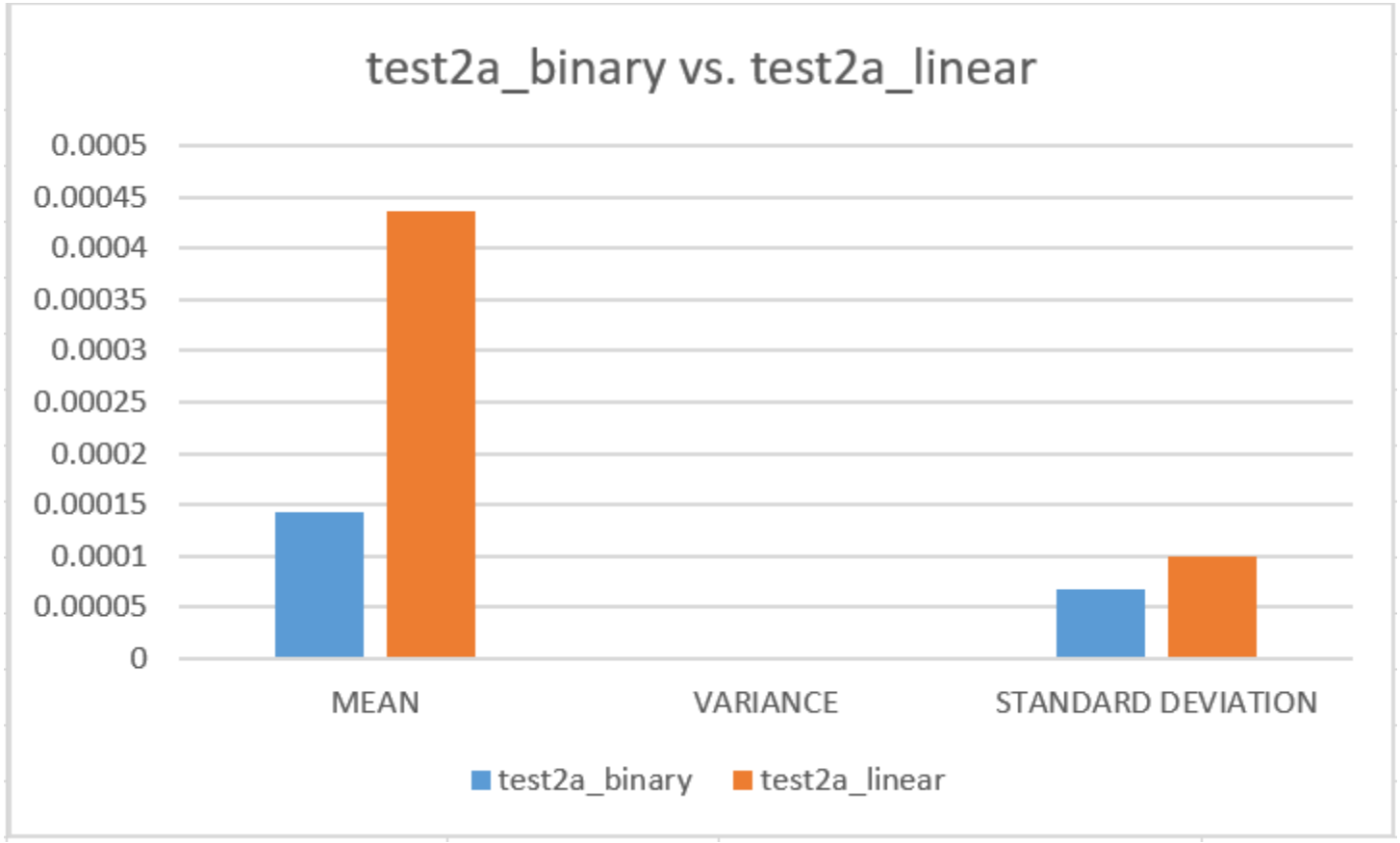

| | test1b_binary | test1b_linear |
|---|---|---|
| MEAN | 0.00011782 | 0.000279 |
| VARIANCE | 0.00000000347921 | 0.00000000346337 |
| STANDARD DEVIATION | 0.0000589848 | 0.0000588504 |




| | test1a_binary | test1a_linear | test1b_binary | test1b_linear |
|---|---|---|---|---|
| | | | | |
| Minimum | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| Q1 | 0.0001 | 0.0001 | 0.0001 | 0.00025 |
| Median | 0.0002 | 0.0001 | 0.0001 | 0.0003 |
| Q3 | 0.0002 | 0.0002 | 0.0002 | 0.0003 |
| Maximum | 0.0005 | 0.0004 | 0.0006 | 0.0006 |
| IQR | 0.0001 | 0.0001 | 0.00001 | 0.00005 |

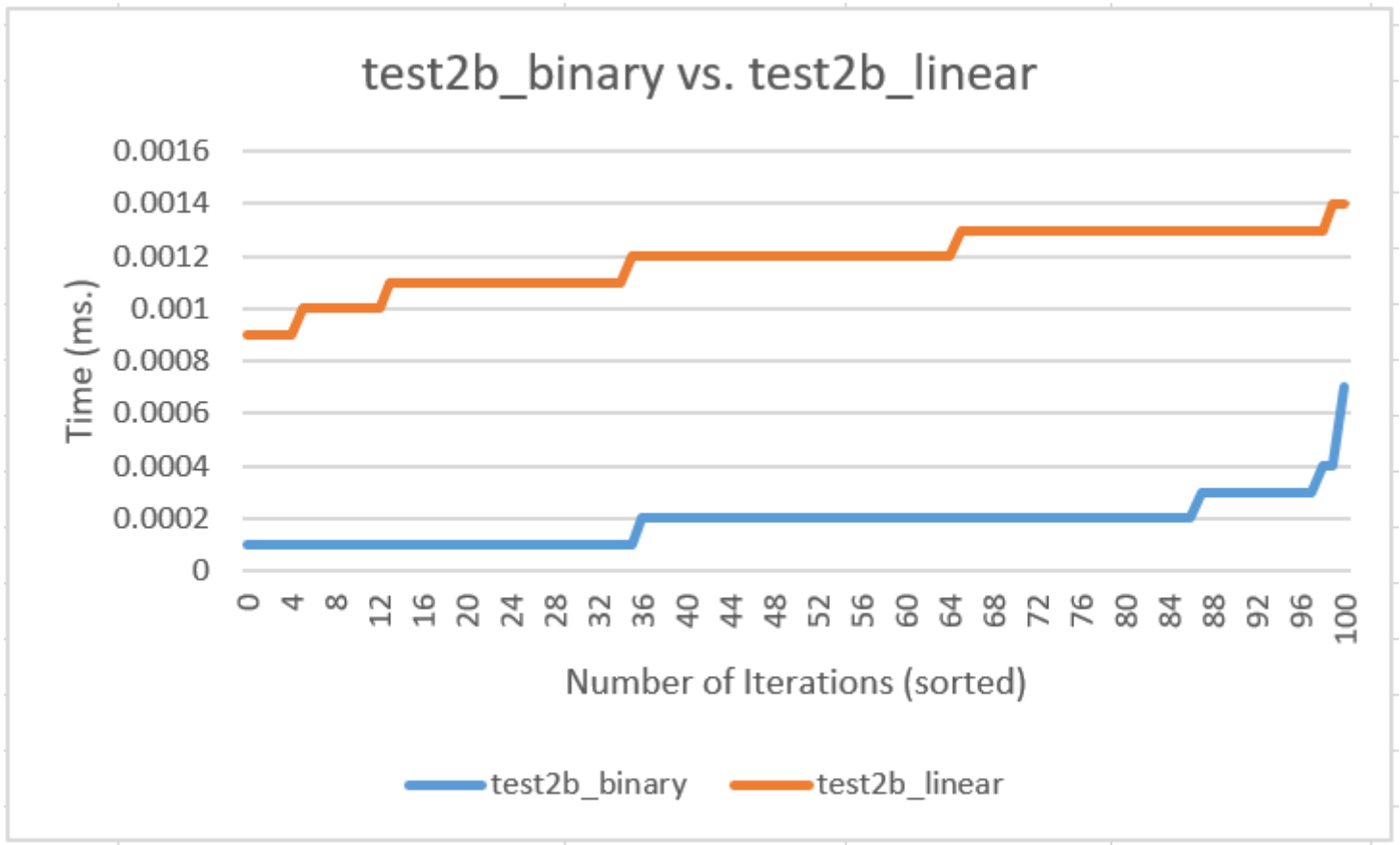
test2a_linear, test2a_binary - search for 333 inside an array of 1000 elements.

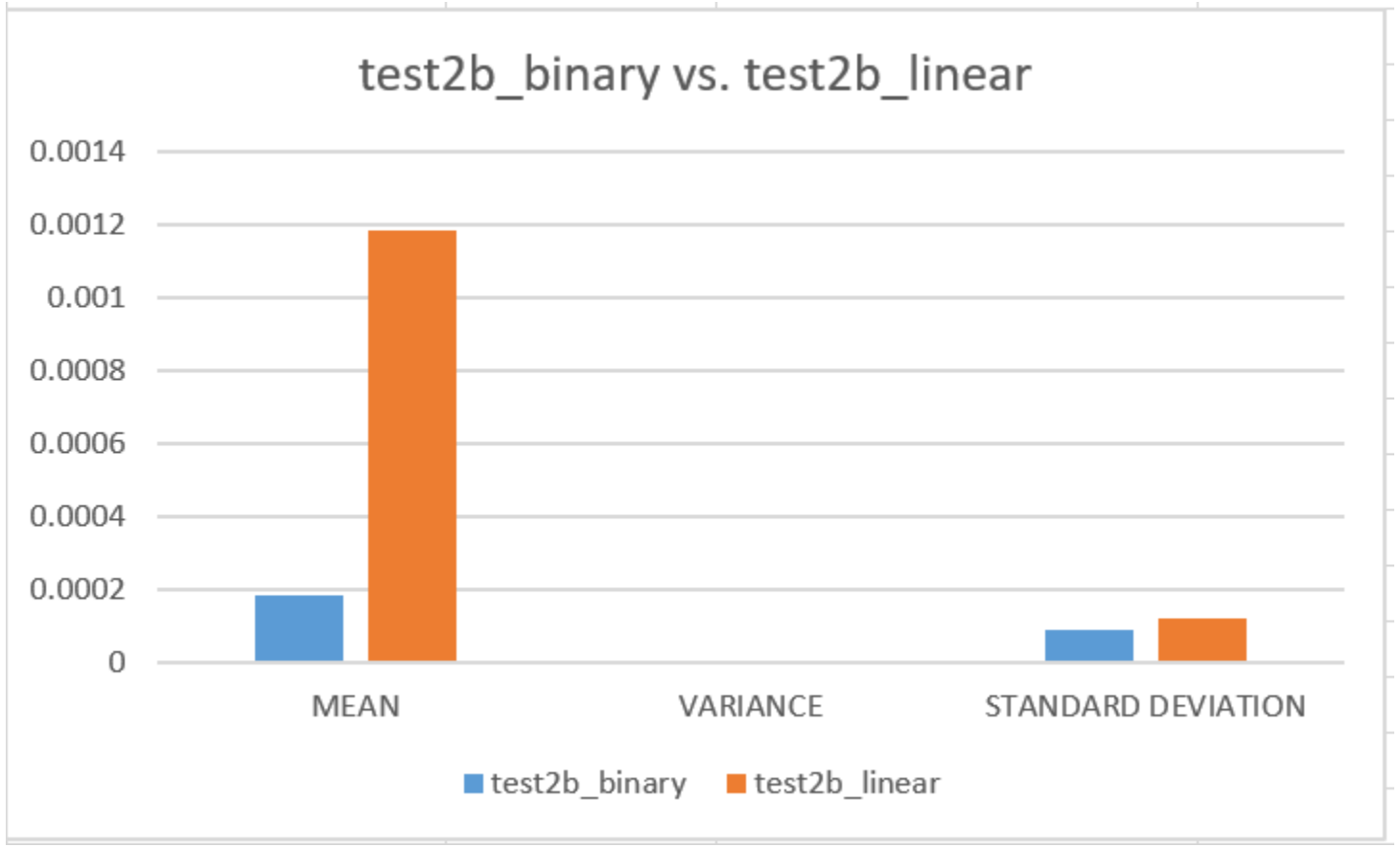




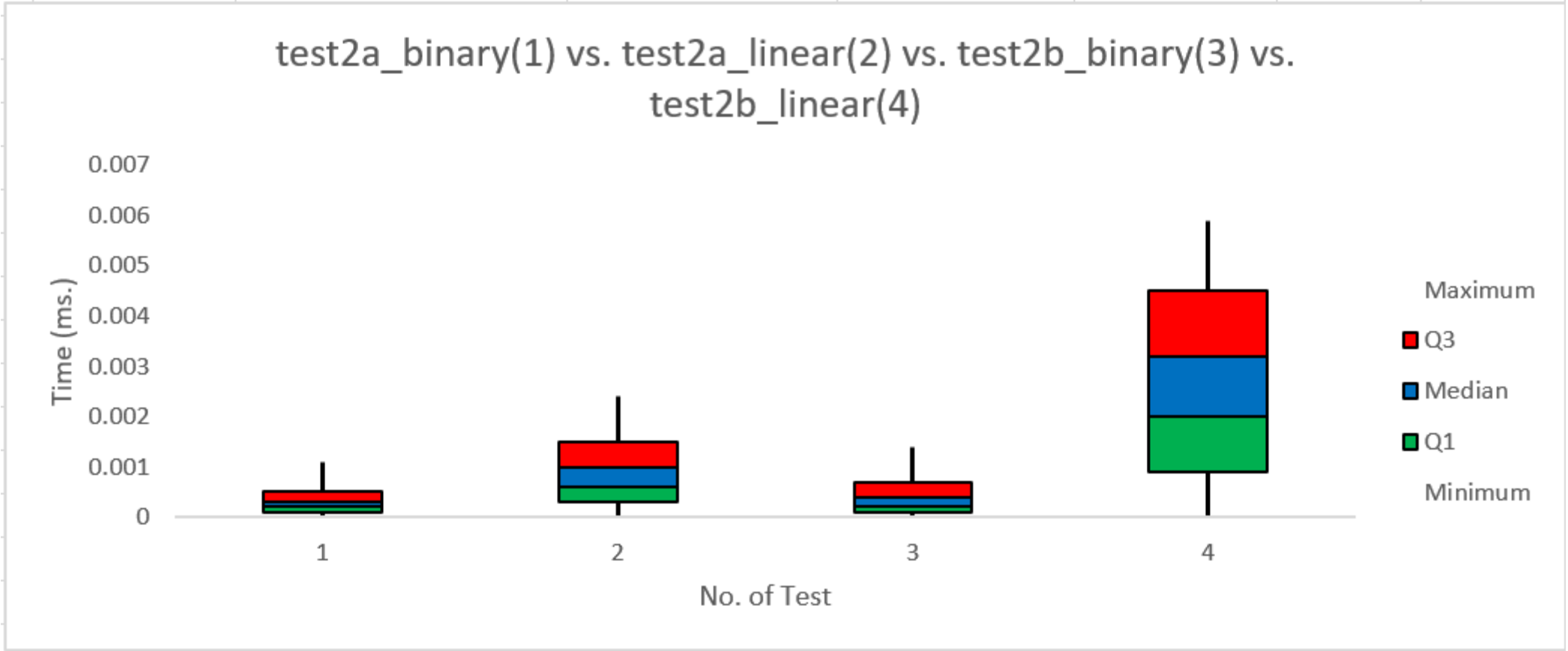

| | test2a_binary | test2a_linear |
|---|---|---|
| MEAN | 0.00014356 | 0.000436 |
| VARIANCE | 0.00000000448162 | 0.00000000991683 |
| STANDARD DEVIATION | 0.0000669449 | 0.0000995833 |

test2b_linear, test2b_binary - search for 961 inside an array of 1000 elements.



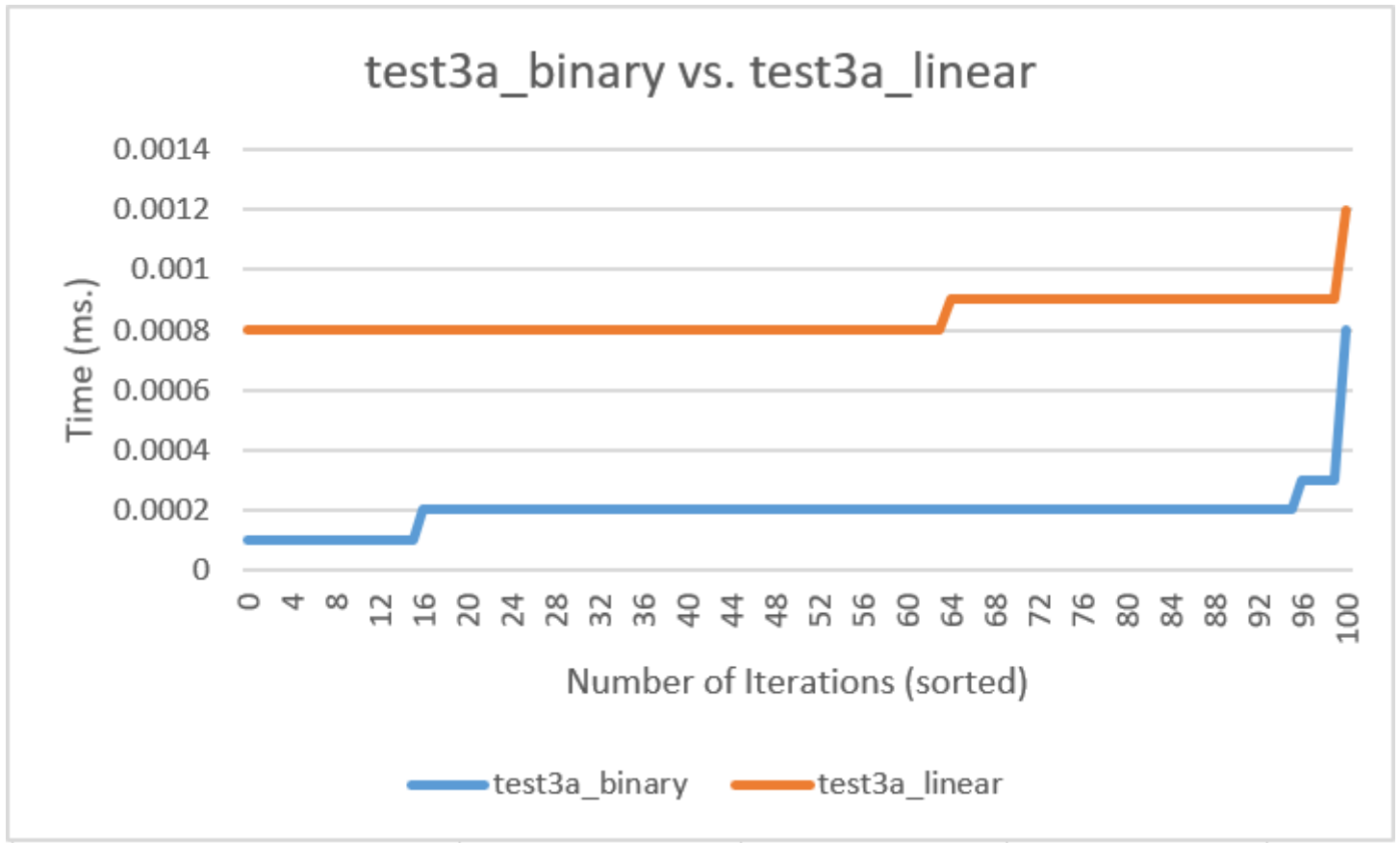


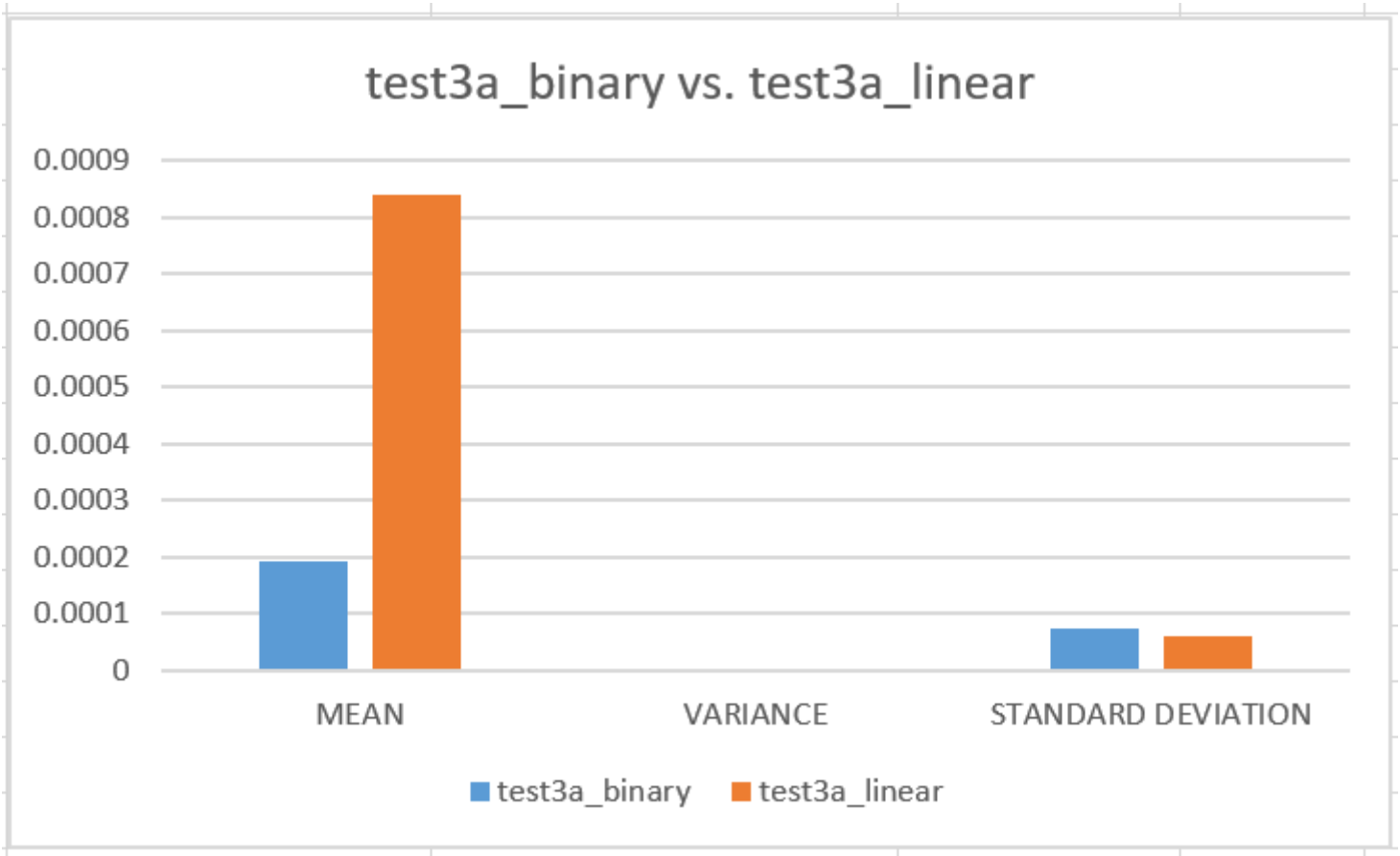

| | test2b_binary | test2b_linear |
|---|---|---|
| MEAN | 0.00018416 | 0.001185 |
| VARIANCE | 0.00000000774653 | 0.0000000138772 |
| STANDARD DEVIATION | 0.0000880144 | 0.000117802 |




| | test2a_binary | test2a_linear | test2b_binary | test2b_linear |
|---|---|---|---|---|
| | | | | |
| Minimum | 0.0001 | 0.0003 | 0.0001 | 0.0009 |
| Q1 | 0.0001 | 0.0003 | 0.0001 | 0.0011 |
| Median | 0.0001 | 0.0004 | 0.0002 | 0.0012 |
| Q3 | 0.0002 | 0.0005 | 0.0003 | 0.0013 |
| Maximum | 0.0006 | 0.0009 | 0.0007 | 0.0014 |
| IQR | 0.0001 | 0.0002 | 0.0002 | 0.0002 |

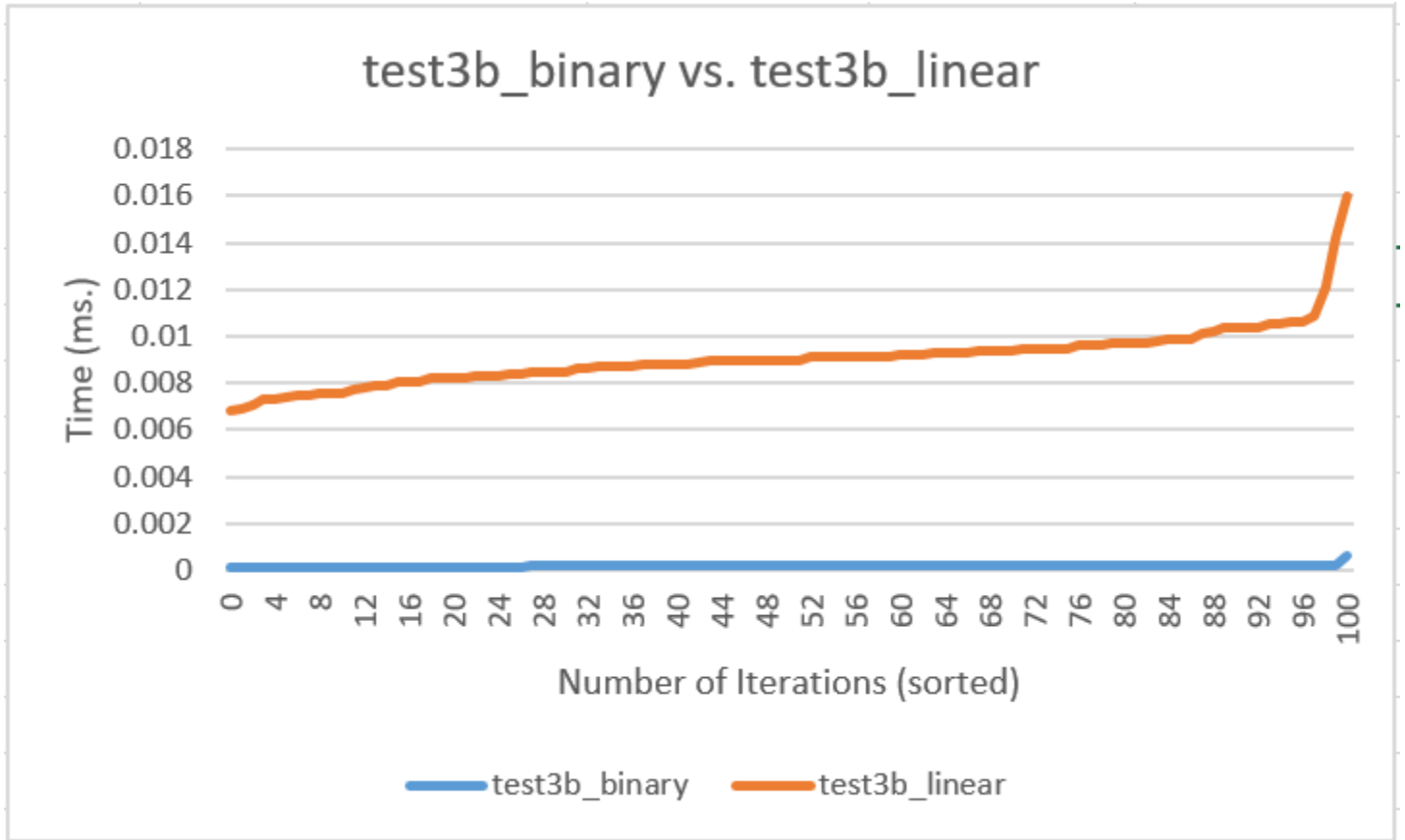
test3a_linear, test3a_binary - search for 1191 inside an array of 10,000 elements.

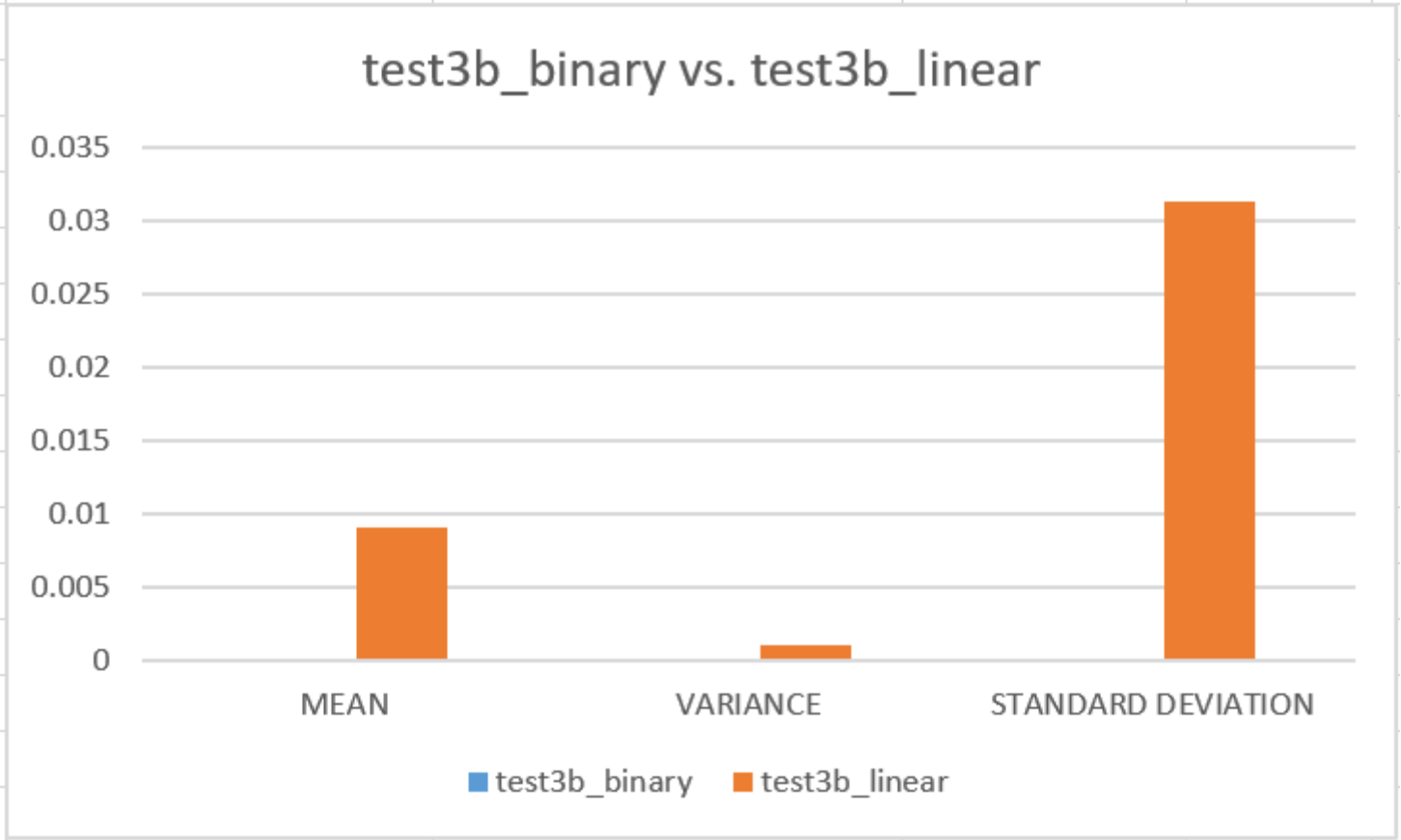




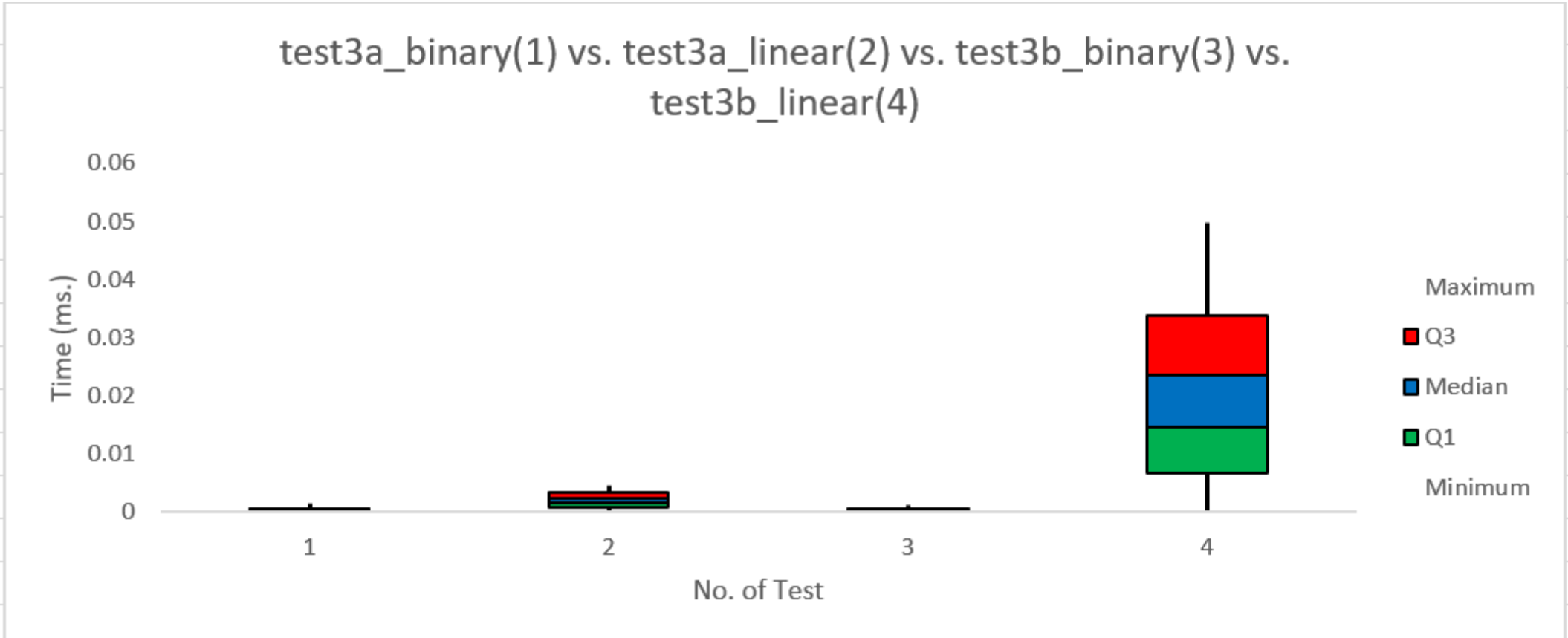

| | test3a_binary | test3a_linear |
|---|---|---|
| MEAN | 0.00019406 | 0.00084 |
| VARIANCE | 0.0000000055643 | 0.00000000361584 |
| STANDARD DEVIATION | 0.0000745946 | 0.0000601319 |

test3b_linear, test3b_binary - search for 9979 inside an array of 10,000 elements.






| | test3b_binary | test3b_linear |
|---|---|---|
| MEAN | 0.00017723 | 0.009086 |
| VARIANCE | 0.00000000377624 | 0.000984723 |
| STANDARD DEVIATION | 0.0000614511 | 0.0313803 |




| | test3a_binary | test3a_linear | test3b_binary | test3b_linear |
|---|---|---|---|---|
| | | | | |
| Minimum | 0.0001 | 0.0008 | 0.0001 | 0.0068 |
| Q1 | 0.0001 | 0.0008 | 0.0001 | 0.007825 |
| Median | 0.0002 | 0.0008 | 0.0002 | 0.009 |
| Q3 | 0.0002 | 0.0009 | 0.0002 | 0.010175 |
| Maximum | 0.0008 | 0.0012 | 0.0006 | 0.016 |
| IQR | 0.0001 | 0.0001 | 0.0001 | 0.00235 |

# Discussion

- It is clear from the results that as the size of my search increases (i.e. the size of my array increases and we are able to search for my values) the **more efficient** Binary Searching is.
- In the very first test (test1a) linear search is a bit faster than binary search because the value it searched for was very close to the start of the array.
- The results showing the efficiency of binary search begin to become much clear after finding the intended values after test1b.
- For test1a, 1b, 2a, 2b and 3a, the variance was barely visible. This could indicate that there was very little spread from the average.
- The standard deviation shown in test3b_linear search was expected as the results were quite far spread out and weren't constant as the other tests I performed.

# Discussion

- My box plot for test 3b_linear was quite significant as compared to the other test3's, this test performed as expected but it was quite poor because test3b_binary did well.
- 9979 existed at the very end of the list, so no wonder!
- The results from test3a_linear vs. test3a_binary were quite significant – test3a_binary performed close to O(1) and O(log N) – which I did not expect as I thought binary search would take slightly longer.
- For most of my results, the standard deviation demonstrated that there was little spread between my times, while for test3b_linear there was a clear significance of spread and for binary search there was little variance and standard deviation.

# Conclusion

- The results from my tests have demonstrated that as we are dealing with larger collections of data we need an efficient method of finding the values we are looking for.
- Binary search O(log N) is the clear choice, but if you are dealing with unordered collections of data then linear search might be appropriate and is still O(N) as there is a chance you may be making less comparisons to find the intended value.
- There are not many cases where you will be dealing with short amounts of data or it is not ordered.

Thank you for listening!

# Sources


- https://www.geeksforgeeks.org/linear-search/
- https://www.geeksforgeeks.org/binary-search/
- https://gist.github.com/sunmeat/c4b4068b83d69c1b40dd75ed0d771b58
- https://techdifferences.com/difference-between-linear-search-and-binary-search.html
- http://www.cplusplus.com/reference/
- https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/
- https://www.geeksforgeeks.org/linear-search-vs-binary-search/