ID1021: Searching in a sorted array

Alexander Lundqvist

04-09-2022

Introduction

In a previous assignment we explored how the execution time for different array operations changed with increasing array size and compared the different operations. Two of the operations were based on the concept of searching and as we saw during the testing, searching through unsorted arrays quickly becomes very expensive for a computer.

In this report we will explore how such searching operations gets affected when dealing with sorted arrays and also take a look at the binary search algorithm.

Method

Linear search

The act of searching through an array for every index is called linear search. The instructions gave us some pre-written code which I wrote into a separate class called LinearSearch. The class contains two methods of searching, one being the classig linear search ant the other is a optimized version that is used for pre-sorted arrays. I also created two methods which could produce arrays with randomized integers with one producing a pre-sorted array.

The class also contains benchmark methods described by the code below. All benchmark methods that are used in the assignment follow the same structure apart from the function names. The iterations parameter allows for flexibility in the testing but is pretty much set to 100 000 for every benchmark.

```
public long benchmarkUnsorted(int[] array, int iterations) {
    Random rnd = new Random();
    long t_total = 0;
    for (int i = 0; i < iterations; i++) {
        long t_start = System.nanoTime();
        unsorted(array, rnd.nextInt(array.length - 1));
        t_total += System.nanoTime() - t_start;
```

```java
        }
        return (t_total/iterations);
    }
```

Binary search

To perform binary search there are two things needed. First we need an item or key that we are searching for. The second thing is that the array we are searching through needs to be sorted. The binary search algorithm itself works by looking at the middle element in increasingly smaller segments in an array. If the element in the middle is less than the key we are looking for, then we look to the left segment. if it is higher we look the right. This is repeated until we find the element or not. The code I used for this assignment is described below.

```java
    public boolean search(int[] array, int key) {
        int lo = 0;
        int hi = array.length - 1;
        while (lo <= hi) {
            int mid = lo + ((hi - lo) / 2);
            if (array[mid] == key) return true;
            else if (array[mid] > key && mid > lo) hi = mid - 1;
            else if (array[mid] < key && mid < hi) lo = mid + 1;
            else return false;
        }
        return false;
```

Double search

For the last task we will revisit a previous assignment where we are supposed to find duplicates in unsorted arrays and benchmark the methods to examine their respective time complexities. This time we will use sorted arrays instead. The task is divided into 3 steps. The first is to use the binary search method we constructed earlier to find duplicates, i.e elements from the first array that also appear in the in the second array.

The second task is to re-implement the binary search method so that we also keep track of the keys and thus being able to skip some steps when searching.

Lastly we will take a look at our old search method and do the same benchmark except with sorted arrays.

Result

Linear search

These two graphs shows the execution time versus array size for the two different methods of linear search. Notice how they look quite similar?

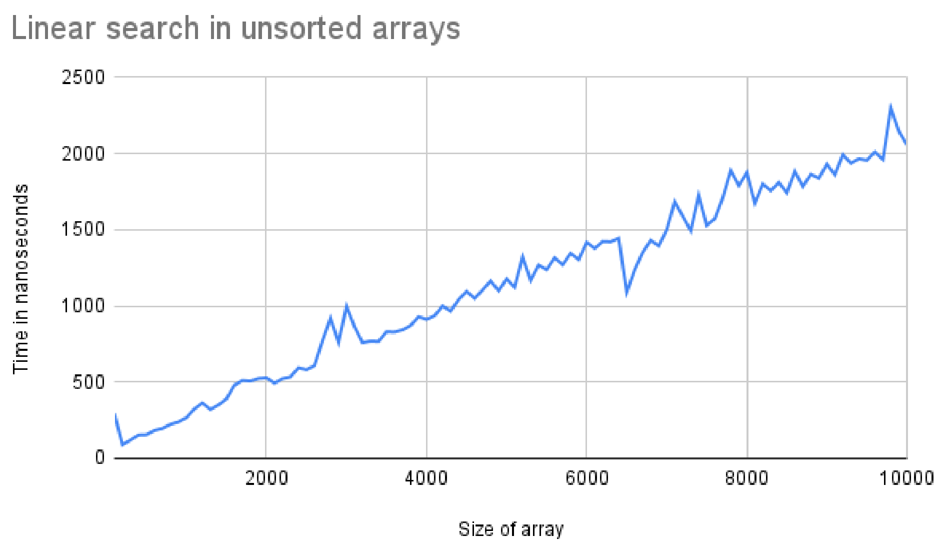

Figure 1: Linear search of unsorted array

The since the graphs can be described as linear we can safely assume that the linear search method follows $O(n)$ time complexity.

By manually calculating the equation of the trend line we can approximate how long it would take to search through an array of a million entries.

For the standard search we choose $x_1 = 0$, $x_2 = 2000$, $y_1 = 0$ and $y_2 = 500$. With this we can calculate $k = (y_2 - y_1)/(x_2 - x_1) = 0.25$ and by using the point $(2000, 500)$ in $y = kx + m$ we get $500 = 0.25 * 2000 + m$ and $m = 0$. If we then use $x = 1000000$ we get that it should take 250000ns to search the array.

The same math was performed on the optimized version with point $(10000, 400)$ and $(1250, 100)$ from the corresponding graph. This calculation resulted in a time of ≈34300ns.

Binary search

If we take a look at the graph produced from the benchmark of the binary search, it is not perfectly clear what the time complexity is but if we would draw a line touching the underside of the graph we can deduct that it follows some kind of logarithmic function. When comparing to known graphs I came

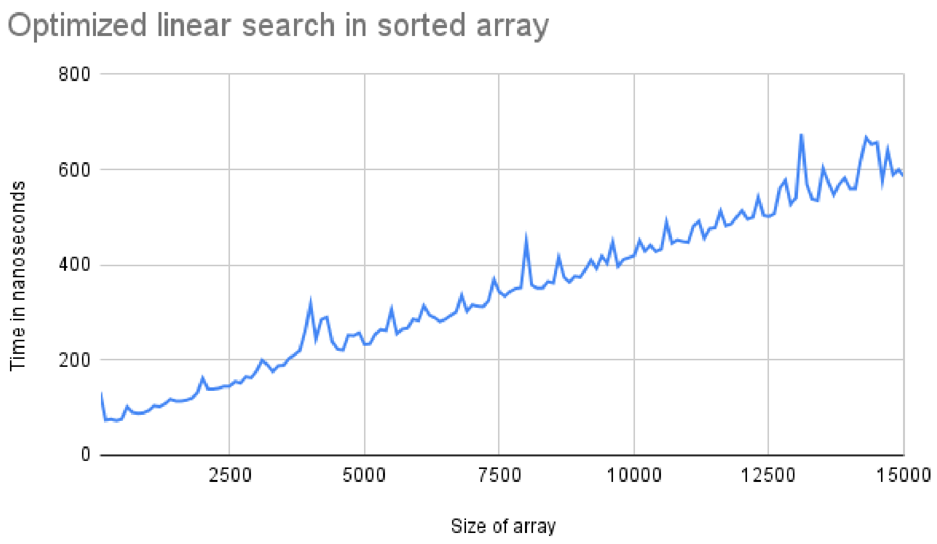

Figure 2: Linear search of sorted array

to the conclusion that it should be $O(log(n))$. Since the graph is shifted upwards we know that the function should be $y = c_y + log_b x * c_x$.

I tried to do a manual approximation by inspecting the graph but this became very tedious and time consuming. I tried some online tools but none allowed for big data sets so lastly I installed Excel and did a scatter plot with logarithmic which yielded a function which I tweaked a bit. The final function was $y = 13 * Ln(x) + 6$. The function was plotted against the data and is displayed in the following figure.

With this formula in mind we can calculate how long it would take to search an array 64 million elements. The assignment suggests running the actual code but due to me only having my old laptop to work with at the moment I will only do the theoretical calculations.

We have $y = 13 * Ln(x) + 6$ where x is the size of the array.

$$y = 13 * Ln(64000000) + 6 = 239.667117337 \approx 240ns$$

.

Double search

To benchmark these methods I had to scale down on the array sizes as my laptop couldn't handle these methods for some reason. In any case, the measurements are sound according to me and we can take a closer look at them below.

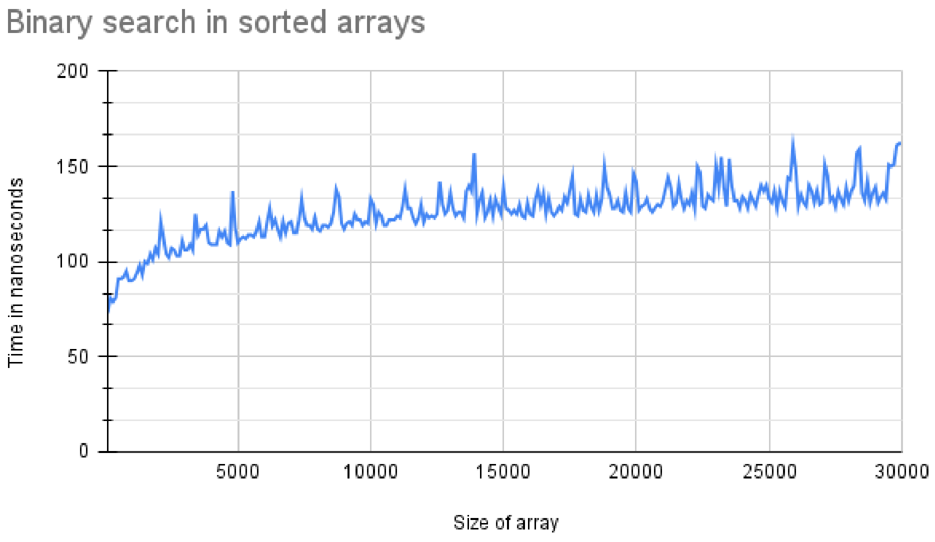

Figure 3: Binary search of sorted array

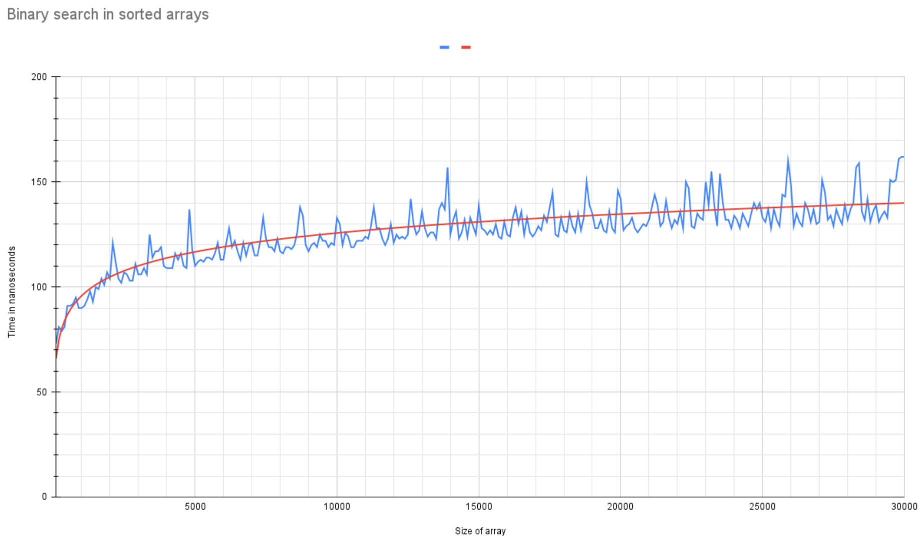

Figure 4: Binary search of sorted array

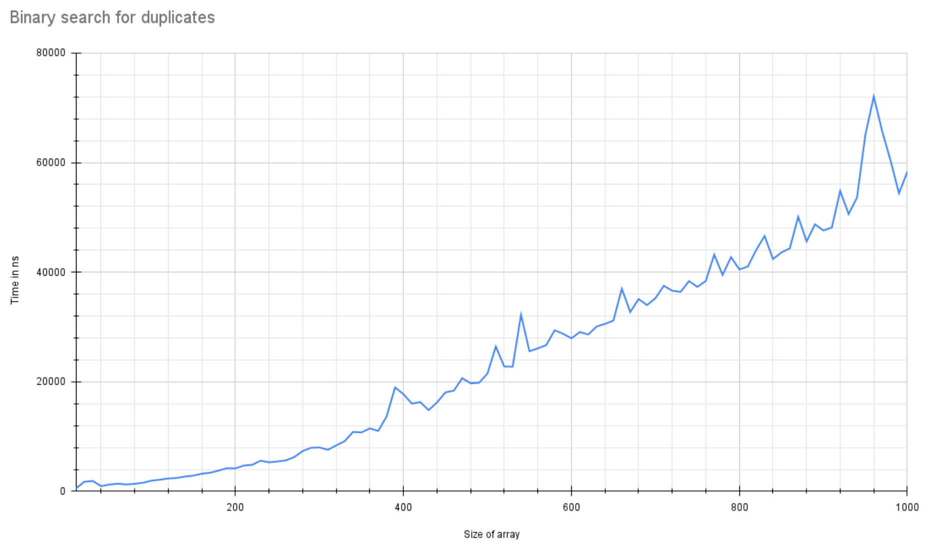

Figure 5: Using binary search to find duplicates

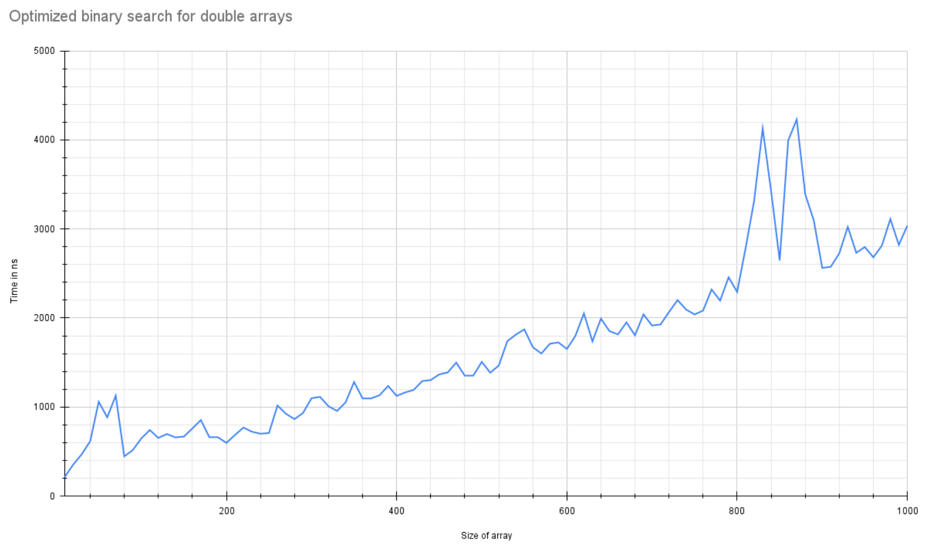

Figure 6: Improved binary search to find duplicates

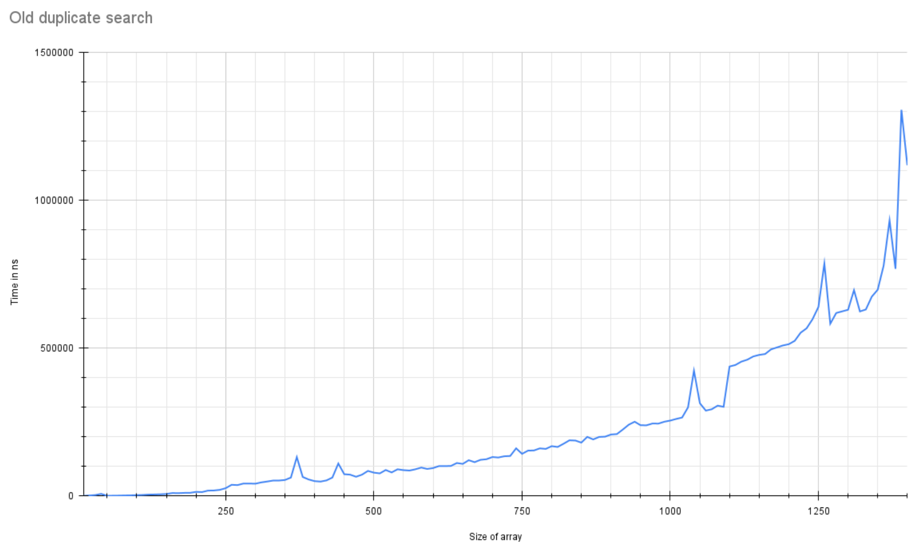

Figure 7: Old duplicate searching method

Figure 5 and 6 looks quite similar. They are pretty much linear and thus we can assume that they both have $O(n)$ in time complexity. The only difference is that the optimized version has a higher k-coefficient, meaning it takes less time per n than the un-optimized version.

If we instead look at figure 7 we can see an exponentially growing curve. This is similar as the experiment in the previous assignment.

Discussion

Linear search

As we saw from the benchmark, both methods are quite similar in performance. If we take a closer look at the data we can see that the optimized search is faster. Since the optimized method have the internal check if a search is pointless, it can exit the search prematurely and thus save time. However, since it always needs to perform the search element by element we still have the same category of time complexity. If we want to describe it more explicitly we could write $c * O(n), c \in \mathbb{R}$. This is relevant for smaller arrays but if we work with really large n we usually do not consider constant factors.

The task in itself was pretty straight forward and no complications did arise during the work.

Binary search

The biggest issue I had with this task was to figure out good benchmark parameters, as the initial graphs that I produced were janky and didn't seem to have any pattern. After some tweaking of the array size and increment I finally got some data that could be properly visualized.

The second issue was to approximate the logarithmic function based on the graph. I used Google sheets and didn't find any built in functions for either trend lines nor approximation and I spent too much time trying to find a solution. Luckily a friend gave a hint that excel is better for approximations.

The thing we can see from both the data, graph and approximated function is that while the time rapidly increases it slows down after a while. This indicates that this algorithm would serve excellent for searching in very large data sets as opposed to linear search. The obvious downside is of course that the data has to be pre-sorted.

Double search

As mentioned in the result section both the standard binary search and the optimized binary search, even though the optimized variant takes less time, have time complexity $O(n)$. But as stated earlier, when dealing with already time efficient algorithms and it is quite hard to reduce the time complexity we do have to consider the constant factors that influence the execution time. It is a shame that the optimized version didn't reach $O(log(n))$ however it was expected since we are dealing with two arrays.

The graph for the old search method is a little weird since I expected it to look more like $O(n^2)$ due to the fact that we loop N*N times in the algorithm. Right now its more akin to $O(n * log(n))$. This could however be because i didn't use high enough benchmark parameters (As mentioned earlier, my laptop has a hard time handling large array sizes + many iterations).