Assignment 3 Report - Sorted array search in Java

Dean Tsankov

September 14, 2024

Introduction

Searching for an item in an unsorted array is quite time expensive. If there is no order to the elements, the only way to find a specific one is to go through the whole data structure. This report will present how things will become much easier and potentially more time efficient if the array is sorted.

Base Case

Let us get an idea of what a benchmark on the standard case of traversing an unsorted array to find an element, looks like. Using the provided method `unsorted_search()` and plugging it into a benchmark code very reminiscent of the one in the first assignment, we can conclude a few things. A plot of the benchmark could also help us get an understanding.

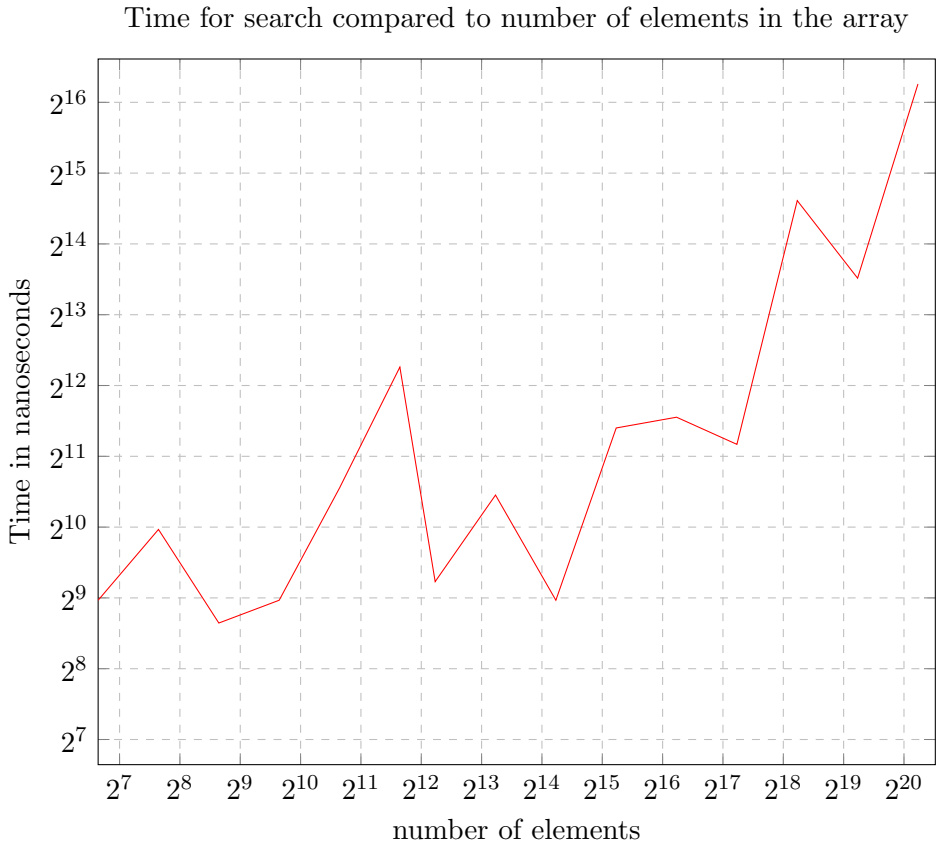

Figure 1: Over unsorted array(samples are average of 3 measurements)

As we might expect the points are a bit sporadic due to the unordered nature of the array. Due to this I decided to take the average time for each array size test of three different executions of the search algorithm. This gives a slightly more general appearance to the plot and allows us to still see a somewhat linear increase in the time. We get this from the fact that the graph axes are mapped logarithmically.

In this case our last sample is the time over an array with 1228800 and the measured time is `78300ns`.

Sorted Array

We now try to benchmark the code using the given snippet for generating an already sorted array of elements. As well as implementing the single optimisation step of stopping our search if the current element is already larger than the one we are searching for. Here is the search method:

```
(...)
 public static boolean sorted_search(int[] array, int key) {
    for (int index = 0; index < array.length; index++) {
        if (array[index] == key) {
            return true;
        }
        if(array[index] > key){
            return false;
        }
    }
    return false;
}
(...)
```

Here I present the new samples over the sorted array:

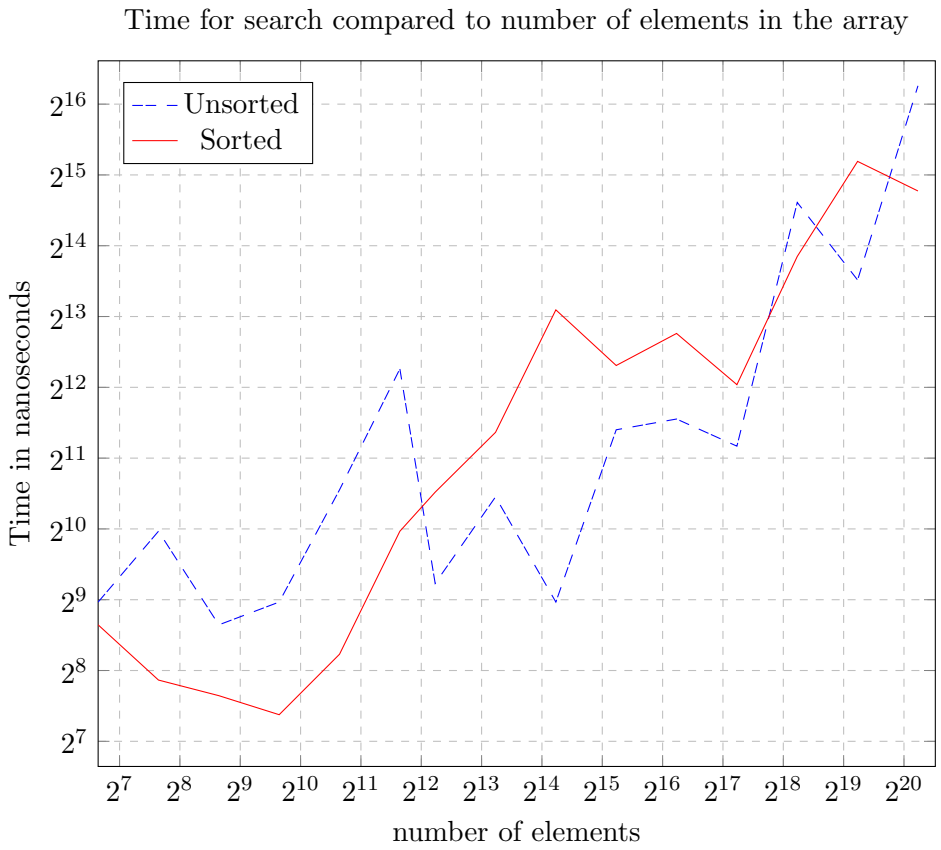

Figure 2: Over sorted array(samples are average of 3 measurements)

As we can see there is some improvement up to around 4000 entries in

the array. After that point it seems it is not guaranteed that the sorted array yields faster times as compared to the unsorted.

Binary Search

In this section we will apply a well-known search algorithm, which is in fact fairly similar to how most people would instinctively search for a page in a book, for example. We could open the book at about the middle page and from there decide that the page we are searching for is either before or after the middle, we than open the middle of let us say the first half and apply the same process. Continuing with this strategy, we will find our page in a fairly small amount of time, as opposed to going through each page and checking if it is the one. Programming this algorithm is simple enough and requires us to only keep track of two additional parameters: the beginning and end of the sub set of the array where we are currently searching.

Here is the filled in code skeleton:

```
(...)
 public static boolean binary_search(int[] array, int key) {
    int first = 0;
    int last = array.length - 1;

    while (true) {
        int index = (first + last) / 2;

        if (array[index] == key) {
            return true;
        }

        if (array[index] < key && index < last) {
            first = index + 1;
            continue;
        }

        if (array[index] > key && index > first) {
            last = index;
            continue;
        }

        return false;
    }
}
(...)
```

Below is the plotted data from benchmarking the binary search.

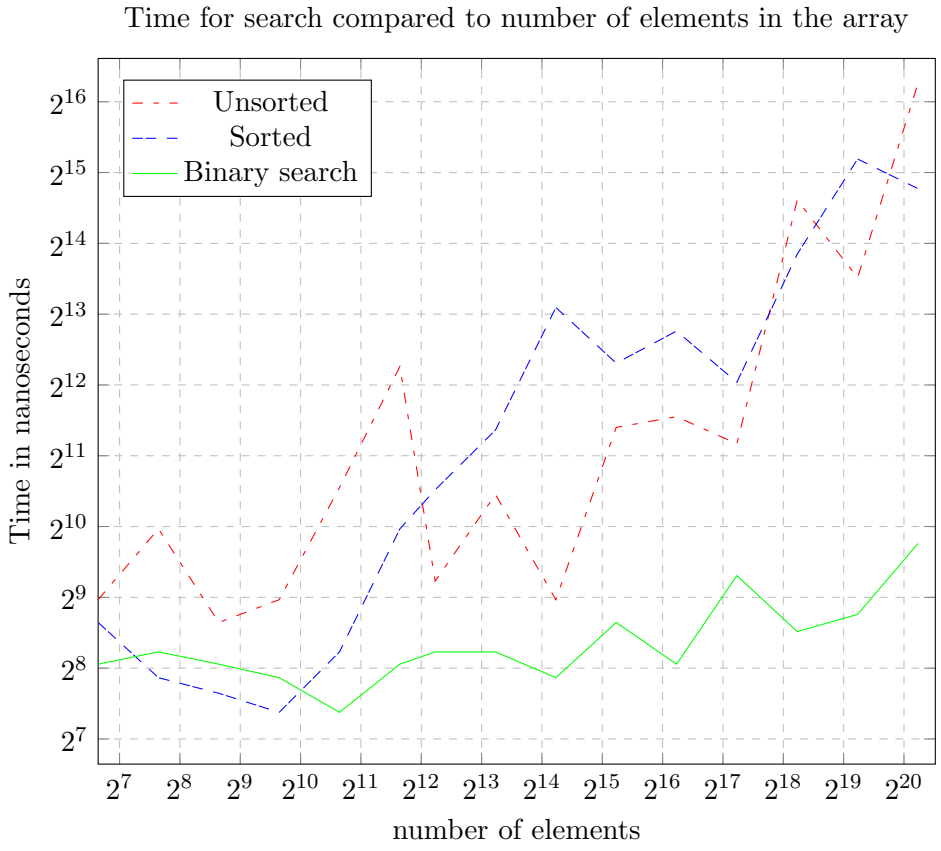

Figure 3: Using binary search(samples are average of 3 measurements)

This now shows us how much more efficient this algorithm is as compared the the simple traversal of elements. From this graph we can generally equate the time complexity as logarithmic i.e. `O(log(n))`. Knowing this we can estimate that an array with 64M elements would only take a comparatively small amount of additional time then 1M elements.

Recursive Binary Search

When we're at it, we might as well introduce an alternative way of implementing the binary search procedure. We will define a procedure recursive() that will do a binary search in an array from a minimum index to a maximum index. The maximum index has to be at least as large as the minimum index. We could for example search for a key between index 10 and 30 but instead of just modifying our existing implementation we will do a trick; we will call the method recursively.

```
(...)
 private static boolean recursive(int[] arr, int key,
    int min, int max) {

    int mid = min + ((max - min) / 2);
    if (arr[mid] == key) {
        return true;
    }
    if ((arr[mid] > key) && (min < mid)) {
        return recursive(arr, key, min, mid);
    }
    if ((arr[mid] < key) && (mid < max)) {
        return recursive(arr, key, mid + 1, max);
    }
    return false;
}
(...)
```

In order for this recursive method to be interchangeable with the `binary_search` method I encapsulate it in a wrapper function which takes only the two parameters of the array and key and passes the additional two to the recursive function.

```
(...)
public static boolean recursive_search(int[] array, int key) {
    return recursive(array, key, 0, array.length - 1);
}
(...)
```

And now we can compare the performance of all the search implementations discussed in this report:

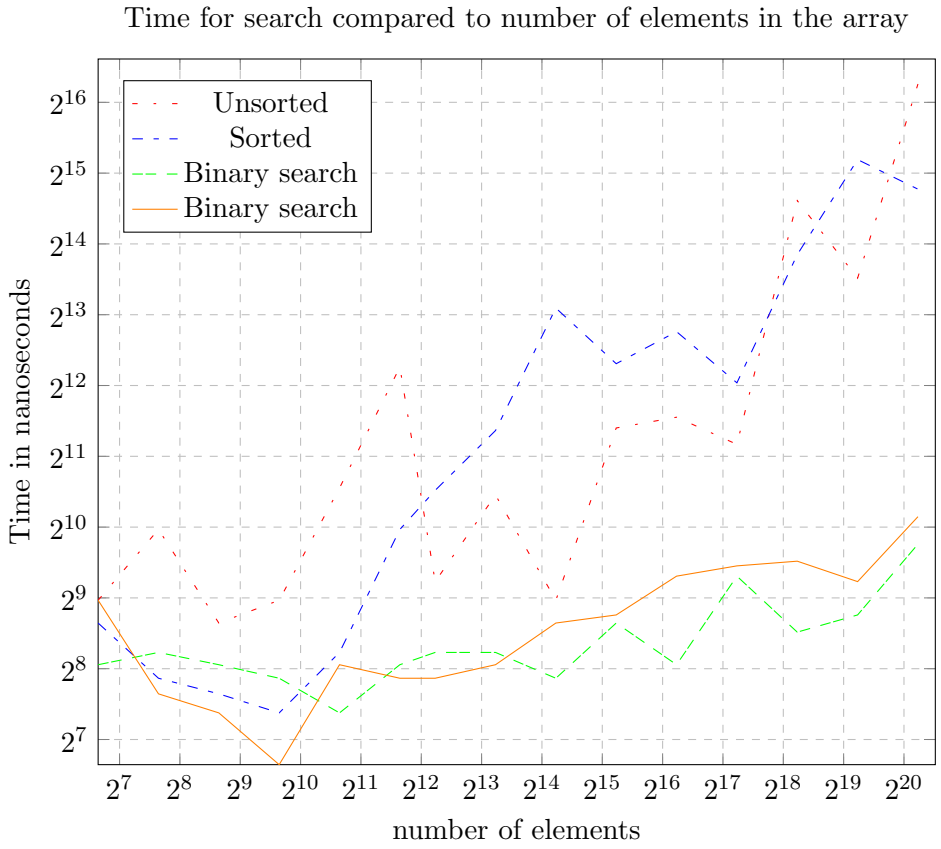

Figure 4: Recursive binary search(samples are average of 3 measurements)

Contrary to what I would think the recursive implementation does not deviate as much from the iterative binary search. We can be sure, though, that this tendency would not follow for larger array sizes.