Sorted Data

Jeffrey Chang

8th September 2022

Introduction

For the course ID1021 - Algorithms and Data Structures, we are given tasks to submit each week for us to learn about the course content. This report concerns the third assignment which was about *Searching* in sorted and unsorted arrays as well as the *Binary Search.* We explore and analyse the speeds/efficiency of search and binary search on unsorted and sorted arrays and determine which ones are better to use in order to make the actual search less expensive.

Background

This assignment requires us to understand the fundamentals of searching algorithms which check for elements in a data structure. We explore one sequential, "*normal* search and an interval search in the form of Binary Search later on.

It is of course fairly important to understand and use the correct sorting algorithms to avoid wasting time or memory. For instance, searching in an unsorted array is more expensive than searching in a sorted array. The question then follows if it may be worth it to first sort an array and then search or if one should search regardless - something which we will start to explore already in this task.

The Tasks

Unsorted and Sorted Array

The first task of this assignment was to simply see how efficient the system is at searching through an unsorted array by benchmarking using the inbuilt Java runtime system, **System.nanoTime**. Following the code given in the instructions, to benchmark an unsorted array, I found the average time taken to find the *key* in an array of varying size by running a simple loop.

```
int[] arrayLengths = {100, 1000, 10000, 100000, 1000000} //array lengths
Random rndm = new Random()
int runs = 10000 //runs runs for average

for (int i : arrayLengths)
        for (int k = 0; k < key; k++)
            keys[k] = rndm.nextInt(test[i-1])
            long t0 = System.nanoTime()
            searchUnsorted(test, keys[k])
            long t1 = System.nanoTime()
```

For generating the unsorted array, I used the same code snippet as the one given in the assignment paper for generating a sorted array, but I edited the following line to make it unsorted rather than sorted, it is possible that we might get duplicates, but for this task it does not matter since we are simply testing its search speed.

```
next = rndm.nextInt(100) + 1;
```

After benchmarking, the conclusion can be drawn that as the amount of elements in an array increases, the time taken to search also increases - meaning that searching in an unsorted array is in the O(n) complexity. Following is a table presenting my results after running the program 1000 times:

Array Size	**Search Time Average**
100	91
1000	123
10000	522
100000	4652
1000000	73973

Table 1: The array size vs search time in an unsorted array

The second part of this task was about **Sorted** arrays. Because the fundamental searching algorithm for both sorted and unsorted arrays are the same - O(n), I believe that even in a sorted array, searching through a million elements would take some time - especially if the key lies towards the back end of the array.

Whilst for smaller arrays, a sorted array would definitely be more efficient, I am not so sure about larger arrays, especially one with 1 000 000 elements. By looking at the time taken to perform searches in an unsorted array of 1 000 000 elements I would guess that it might take just over 100 000 nanoseconds. Following is the table presenting my results after running and benchmarking a sorted array:

Array Size	Search Time Average
100	47
1000	169
10000	1405
100000	14369
1000000	138908

Table 2: The array size vs search time in a sorted array

As expected, searching in a sorted array has the same complexity as an unsorted array, O(n) with a linear x=y increase. The unexpected part was that searching in a sorted array is already slower by the 1000th array element (and possibly even before that - which we can check exactly using smaller incremental changes in array sizes) so it is not as efficient for larger arrays. The benchmarked result for searching 1 000 000 elements in the array was close to my guess at 138 908 ns.

Binary Search

For the second part of this assignment, we were tasked with implementing an interval search that is much more efficient than a linear search, except for smaller arrays called *Binary Search*. The basic idea of Binary Search is that if he have a sorted array, we can start in the middle and divide the search size in half, then we hone in on the region that could contain that which we are looking for and continue halving the search sizes. This way, we should have a time complexity of O(lnn).

The important parts of a Binary Search can be summarised in the following code snippet given in the assignment:

```
int first = 0 //init the first and last of the array
int last = array.length - 1
while(true)
    int index = first + (last - first) / 2
    if (array[index] == key)
    //going to the middle, if true = found
    if (array[index] < key && index < last)
        first = index + 1
    //if less, go to the "lower" half of array
    if (array[index] > key && index > first)
        last = index - 1
    //if more, go to the "higher" half of array
```

This division of the array continues until the program finds the key, unless it does not exist in the array. As shown in the code, the important parts are initialising the first and last parts of the array, going to the middle of the array, checking if its larger or smaller than the indexed array and the choosing to either index + 1 or index - 1 for the first or last respectively.

This way we divide the array into two, and then limit the array into the correct half. Doing this repeatedly creates a loop which halves continuously, one noteworthy part is that if the key lies in the first or last index of the array, we will use the *else* part of this code.

Array Size	**nanoTime**
100	204
1000	213
10000	261
100000	336
1000000	670
16000000	1942

Table 3: Array Size vs Binary Search Runtime

As we can see from the data table above, the runtime of Binary Search follows a natural logarithmic curve where the larger the array size, the smaller the change in search time. Using this to estimate the runtime for 64 million elements then gives us roughly 3000 nanoseconds - for my estimation.

I tried running the program for 64 million entities 1000 times which resulted in an average of 2356 nanoseconds, slightly faster than expected. This may be due to the fact that I underestimated the growth of a logarithmic function but it should be noted that 64 million and 16 million are in the same order of 10 whilst my other trials always increased by an order of 10. This may have altered my prediction to be slightly higher than it should have been.

Even Better

For the last part of this assignment, we are trying to further improve on the search algorithm. This time, the task algorithm is to search through two sorted arrays instead by using binary search to find duplicates or an element either larger or smaller than the previous one.

The basic idea of this algorithm is very similar to binary search - we use binary search to efficiently find keys in two arrays instead by going "upwards".

```
while (j < arrayI.length && k < arrayII.length) //checking both array 1 & 2
    if (arrayI[j] < arrayII[k]) //if key in array 1 is smaller, continue checking
        j++
    else if (arrayI[j] == arrayII[k]) //if equal, we have duplicate
```

```
        j, k++
    else if (arrayI[j] > arrayII[k]) //if array 1 is larger, check array 2
        k++
```

As can be seen in the code snippet above, the concept is fairly simple and similar to Binary Search, and can be simplified into three different steps. Once we find something equal, its a duplicate in the arrays and we continue searching with the next index in both arrays, if either one of array 1 or 2 are smaller, continue searching in that array for the next index.

This way, we check everything in both arrays but only *really* go through one array size once. The complexity of this algorithm lies in O(n) as it just checks arrays of size n, therefore, it is still an improved algorithm over binary search when looking through two arrays, and just like binary search, it is more efficient over larger and larger arrays.

Array Size	**Original Duplicate**	**New Duplicate**	**Binary Search**
100	89	8	67
1000	1268	54	253
10000	31625	472	5783
100000	985362	3221	64436

Table 4: Array Size vs Original, Improved Duplicate and Binary Search

From the table above, we can compare the execution time in nanoseconds for the final streamlined version with binary search, and the previous duplicate search algorithm. It is obvious from first glance that our final version is the most efficient and best at searching, especially for larger array sizes. But we can also see this if we compare their respective complexities where the final one was in O(n), binary search was in O(nlogn) and O(n^2) for the previous duplicate search algorithm.

It is obvious that, as we keep growing larger array sizes, O(n) will be the most efficient, followed by O(nlogn) and the worst case would be O(n^2) (because it obviously checks through *both* arrays of size n, meaning n*n). The conclusion can be drawn that the improved binary search is much better than binary search, but both are significantly more efficient than that of our old duplicate algorithm.