Report Assignment 3: Sorted data

Thomas Gustafsson — thomg@kth.se

September 2022

Introduction

The purpose of this assignment is to understand how time-efficient different ways of finding elements in an array can be. As this report will uncover; a sorted array will be a lot easier to find an element in, comparing to searching through an unsorted array. All code is written in Java.

A first try

search: unsorted

The first thing to do is to set up a benchmark where an unsorted array is searched through. This is easily done by creating a loop through all the elements in said array. I'm searching for a specific number, the key. Therefore, if in one of the loops array[index] == key, the method returns true. If there aren't any matching numbers, the method returns false.

It is not very interesting to measure the time of searching an array the same size several times. That is why the benchmark is built to create bigger arrays for each loop and copy over contents of the previous array. To get more accurate numbers, the average of 1M rounds is taken for every test where the array is doubled from the previous one. Following are the results for this method:

Array Size	**Key**	**Time (ns)**
20	4	300
40	8	300
80	16	400
160	32	600
320	64	900
640	128	1200
1280	256	1400
2560	512	6000
5120	1024	9000
10240	2048	18000

The table shows that the execution time for an unsorted search grows with the size of an array. But regarding time complexity, I am sure that this method should be O(n) as the each index is searched regardless. The results saying otherwise can may be due to some issues in the code or IDE. Could be more accurate at larger array seeing as System.nanoTime() can be unreliable at times.

search: sorted

Searching through a whole array index for index is very time consuming. There are many other ways of doing it, and this report will go through some of these ways. We can follow the same format, but give the method a sorted array. When the key is found, we stop searching. That can cut time down substantially. See the following table:

Array Size	**Time (ns)**
20	5
40	15
80	35
160	70
320	150
640	300
1280	650
2560	1300
5120	2600
10240	5200

Seeing the results of this method, it is clear that the time complexity of searching through a sorted array and exiting as soon as the key is found, is O(n). Doubling the array size, also doubles the time it takes to complete. Seeing as the complexity is O(n); for an array size of 1M it would take 500k ns or 0.5 ms. A lot more time efficient than the previous method. There is an even more efficient searching algorithm, and it's called binary search.

Binary Search

Binary search works the same way one would search through a large book to find a specific page. Jump to the middle of the book, and then jump either one quarter forwards or backwards. And the same again until it isn't possible anymore, then you've either found what you're looking for, or it isn't there. Finding the specific page is now easy, and time efficient. But this requires that the pages actually are sorted. This algorithm works the same with arrays in Java as well.

For this to be possible it is needed to keep track of the first possible page and the last possible page. These being known, the middle of can be found. A sorted array the size of *n* is given. The method for binary searching this array with a wanted value *key* is structured as following. A variable *first* is set to 0, and another *last* is equals to the length of the array - 1. The algorithm can be built:

```
while (first <= last) {
                int index = (first + last)/2;
                if (array[index] == key) {
                    return true;
                }
                else if(array[index] < key)
                    first = index + 1;
                else
                    last = index - 1;
            }
```

It works as described. The middle is found. If the key is larger than the middle, the first possible index for it to be is middle + 1. Or if the key is smaller than the middle, the last possible index for it to be is middle - 1. Now the loop runs once again, with an updated *first* or *last*. As the loop runs, the bound of potential positions halve for each time. This makes it incredibly time efficient. Once the middle *index* is equal to the key, the algorithm is done and returns *true*. To truly test the capability of this algorithm, a benchmark is created for an array size up to 5M. The array size begins at 20, and doubles for each round. As the following graph shows, the time complexity for searching a sorted array for a specific number using binary search is O(log(n)).

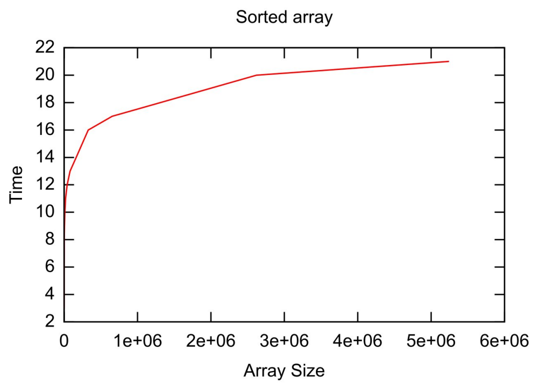

Figure 1: Searching sorted array using binary search. (ns)

Duplicates - Binary Search

The binary search method can be used for more than finding a number in a single array. In this section, a merge of a previous assignment and binary search will be discussed. The goal is to use binary search for two sorted arrays to find duplicates in these. A requirement is that numbers are unique in respective arrays. Meaning that there are no repeating numbers in one array. The algorithm above is used as a base but is further developed to support two arrays.

```
while (index2 < Arr2.length && index1 < Arr1.length) {
                int first = 0, last = (Arr1.length) - 1;
                while (first <= last) {
                    int mid = (first + last) / 2;
                    if (Arr2[index2] == Arr1[mid]) {
                        break;
                    } else if (Arr2[index2] > Arr1[mid])
                        first = mid + 1;
                    else last = mid - 1;
                }
                index2++;
}
```

Comparing the time of this algorithm to the method in the previous assignment for searching duplicates, it is clear that binary search outperforms the linear one by far. The difference is x100 faster for binary in lower array sizes, and x10-x25 for large arrays. Now also consistent time measurements. As for analyzing the time complexity, larger arrays should be searched to give a more definite answer. Judging by the measurements, an estimation would be O(nlog(n)). Plotting the results gives a relatively linear slope. But, the theory is that the time begins to greatly rise as larger arrays. See the following graph for binary search:

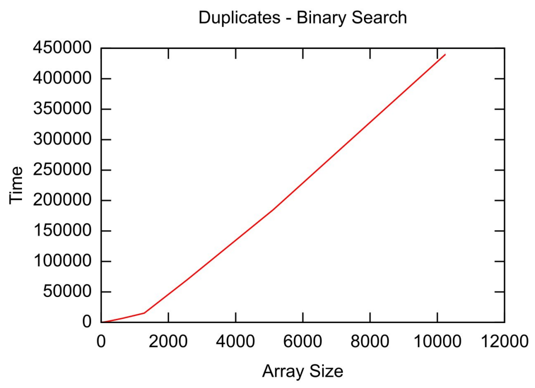

Figure 2: Searching duplicates in two sorted arrays using binary search. (ns)

Optimization

Using binary search for finding duplicates is already relatively quick. But an optimization can be done. Keeping track of the next element in the first list enables us to move forward more efficiently. If the next element in the second list is smaller than the next in the first, we move forward in the second list. If an equal or a greater number is found, we move forward in the first list.

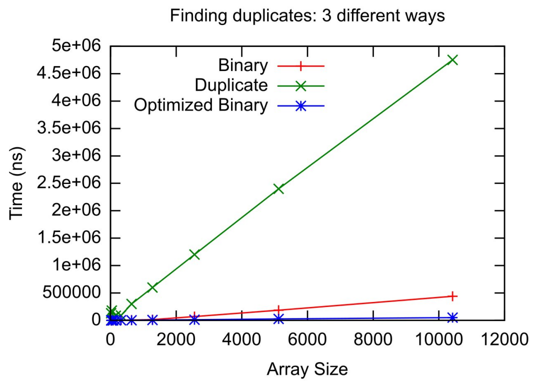

Figure 3: Searching duplicates comparing three methods

As discussed and the benchmarks in the report show, sorting two arrays before doing the search for duplicates is definitely worth it. Reason being that it makes it possible to use efficient algorithms such as binary search. This algorithm has shown greatly improved times comparing to less efficient methods. But this of course raises the question, how expensive is it to sort an array?