Searching in sorted and unsorted arrays in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment the goal is to compare different searching techniques for arrays. First I'm going to look at unsorted arrays and then sorted ones while trying to improve execution times.

Unsorted search

Having the first algorithm given, I just filled up an array of size `n` with random integer values (between 0..2n) and decided to benchmark searching for random keys in them. I started with a warmup and then ran the benchmark k=100 times for each value of n and compared the minimum times for the different array sizes in the table below.

array size (n)	execution time [µs]
100	13.5
200	25
400	48
800	90
1600	178
3200	340
6400	640

As you can see, every time the problem size doubles, the execution time increases roughly by a factor of 2. This means that the algorithm is linear, so it is of O(n). This is fine for smaller arrays like what I worked with here, but when it comes to data, it rarely is in such small quantities. The need to search through large data arrays motivates the existence of many different searching algorithms that strive to improve efficiency as problem size grows. I am going to be looking at a few of them in this weeks assignment.

Sorted search

For the next part, I will assume that the array is sorted, i.e. it is ordered increasingly. This means that we can stop looping through the array when

1. the key is found in the array (`array[i]==key`)
2. the current value is greater than the key (`array[i]>key`)

This second condition is why it may be a bit faster than the previous algorithm, at least in the case when the key is not found in the array. To test this I generated an array of increasing random integers and searched for random keys in them. My measurements confirmed that it is indeed faster (see table below).

array size (n)	execution time before [µs]	execution time now
100	13.5	7
200	25	14
400	48	22
800	90	45
1600	178	82
3200	340	160
6400	640	310

As you can see, we already cut the execution time in half by using a sorted array, but still, it is of O(n) and so it will grow pretty fast with the problem size.

Binary search

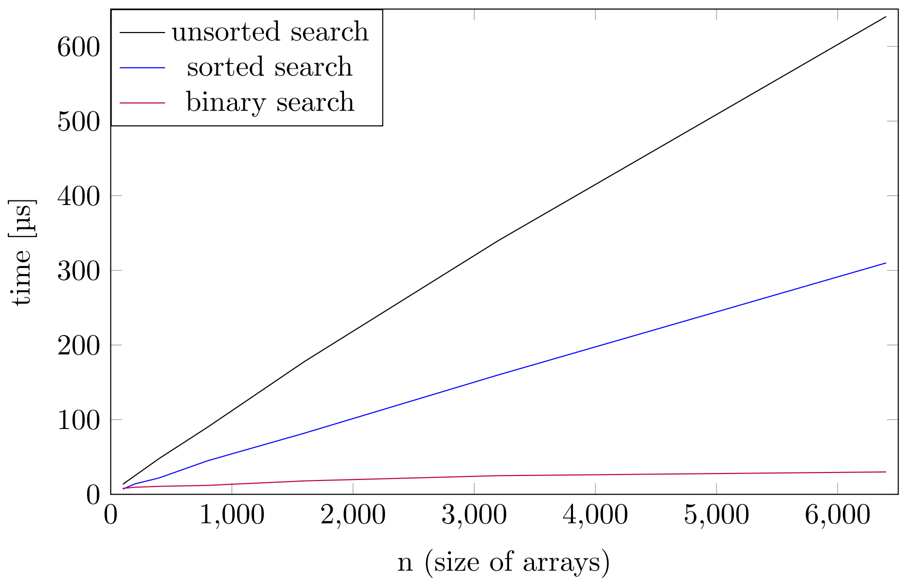
The next and final algorithm I will be looking at also requires the data to be sorted, but it introduces a new concept that aims to cut down on the cost of looping through an entire array. The idea is to jump to the middle of the array and then depending on the value found there, decide which half to continue searching in and then jump to the middle of that section and so on, until it founds the key or it has concluded that the key is not contained in the array.
Assume we're searching for key=5 in the array [1,3,5,8,9,12,17]. First we jump to the middle and see that 8 is greater than our key, so if the array contains the key, it must be to the left of 8, so we will only have to keep searching through the first half, namely [1,3,5]. Again, we check the middle and see that $3 < 5$, so now all we have left to check is [5], so in the next step we return true as the key was found. However, if we were looking for 4, we would still get to the final step of having only [5] to go through, in which case we would not find our key and would return false. Below is my code for binary search:

```java
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;
        }
        else if (array[index] < key && index < last) {
            first = index+1 ;
        }
        else if (array[index] > key && index > first) {
            last = index-1 ;
        }
        else return false;
    }
}
```

We had most of this code given, so I just had to determine the new values for first and last when jumping to the next section of the array.
At first I set both equal to index itself which resulted in getting stuck in an infinite loop. I was tempted to get rid of the `while(true)` loop (which I wouldn't had even started with if it wasn't in the skeleton code) but decided to challenge myself with debugging this code. To do so, I moved it over to a separate file and quickly deducted that the problem arises when the key is not found in the array. Through a few more `print` statements I realised that in such cases it gets to a point where first = k and last = k+1, and due to integer division we have index = (k+k+1)/2 = k+0.5 = k. So when setting first = index, nothing new happens, and that's where the infinite loop comes from. To solve it, I had to set `first` equal to `index+1` and `last` to `index-1`, which I could do safely, because we have already seen that `array[index]` is not equal to our key.
At this point the code was up and running so I started the benchmark. I knew it should be O(log(n)), but my numbers did not seem to agree. After a long while I realised the problem was that I was searching for n elements every time, instead of a fix number of elements, so what I was measuring was O(nlog(n)), and also O(n^2) for the previous two algorithms. By changing it so that I always search for a 1000 random elements in the arrays, I managed to get seemingly correct numbers (see table below), and so I went back to change it in the first two task as well (which I decided to leave out of this report).

array size (n)	execution time [µs]
100	8
200	9.5
400	10.7
800	12
1600	18
3200	25
6400	30

After sorting things out, I benched this version as well and saw that this algorithm does indeed make a huge difference especially on larger arrays. I compare them in the following graph.



To see how well it works for large arrays, I tested it on an array with 1.000.000 elements and got that it takes about 79 µs to search through it. This surprised me, because this implies a coefficient around 4, which is not what I gathered from the previous measurements. I decided to just run with it and guessed that for an array size of 64M, it would take $4 * \log(64000000)$=104 µs, but I measured around 190. It appears that the programs time complexity is still not purely logarithmic, but either way, it does work pretty fast, especially if you compare it to what we started with.

Recursive binary search

Next I implemented the same exact algorithm again - but this time recursively (see code snippet below).

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

It requires two more arguments (a `from` and a `to` index) and instead of looping infinitely, it calls itself again and again on smaller sections of the array. Benchmarking this results in exactly what I expected from the previous implementation too, so we know that the algorithm is correct and has logarithmic time complexity, but something must have went wrong during implementation. I did not manage to figure out the problem, but it does show the importance of testing your programs. My measurements are in the table below.

array size (n)	execution time [µs]
100	7.8
200	9.3
400	11
800	12.8
1600	14.3
3200	16.2
6400	18

Notice that every time the problem size doubles, the execution time increases by about 1.7. This kind of relationship implies O(log(n)).

Summary

Through this assignment we saw the difference in searching in unsorted and sorted data, implemented the binary search algorithm two different ways, and concluded that having sorted data has great advantages - but we must not forget that sorting can be costly as well.