Sorted or Unsorted: That is the question

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to explain and showcase why it is important to use sorted data collections when running different operations. In other words, in this assignment, unsorted and sorted arrays of different sizes have been generated and used as benchmarks in order to present the contrast between searching a given element or elements in the sets of data. What I have particularly found most interesting is that in ceratin situations, not all code optimisations are necessarily always faster. This topic will, of course, be further explored in the upcoming sections.

Searching in an unsorted array

In order to be able to analyse how much time it would take to search through an array of 1 million elements, my goal was to first find a relationship between *the size of the array* (given by 'n') and *the time* (given by 't') it takes to search for the key in it. The result, namely the average value of the ratio t/n would therefore be the coefficient I would use in order to estimate the time it takes to search in an unsorted array, regardless of its length.

```
int[] sizes = {100,200,300,400 ... 1300,1400,1500,1600, 1000000};
for(int n:sizes) {

    int key = rnd.nextInt(n*7);
    int array [] = unsorted(n);
    double t = time (array, loops, tries, key);

    System.out.printf("%8d %8.0f %8.2f\n ", n, t, t/n );
    avg += t/n;

}
    System.out.printf("%s", "ratio on average : ");
    System.out.printf("%.2f", avg/sizes.lenght);
```

As seen in the graph below and after having executed the code several times, I noticed that from time to time, it turns out there occur spikes in the value of the execution time, which I highly assume rely on luck. And what do I mean by that is, since I randomly generate both the array and the key I am looking for, there are high chances that regardless of the data set size, the element is sometimes found sooner or later. I, therefore, decided to run the code below multiple times, namely in an outer for loop and calculate the average execution time (I divided the previous obtained result by the numbers of iterations in the extra for loop).

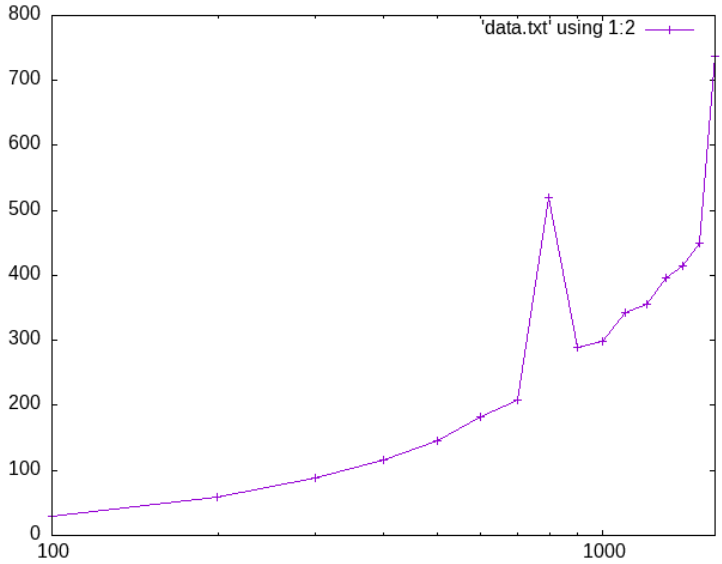

Figure 1: An example of a spike in the execution time.

As a result, my estimation was extremely close to the actual obtained result, which can be seen in the next section where it is also compared to sorted array's one. Given this situation, the sorted array is *1.1* times slower than the unsorted one.

Searching in a sorted array

Here's the section where the interesting part I was talking about in the introduction comes in. Before creating the benchmark and running any code, my wild guess was that: *an extra condition to stop when the key is larger = significantly less time.* Well, is that the case all the time?

Moreover, as it can be noticed from the table below, the answer to the posed question is, therefore, no - an optimisation doesn't always make the algorithm faster, especially in certain situations: for instance if the element

is not found at all or found, but close to the end of the sorted array. The **CoeffDiff** column refers to the absolute value as a result of the difference between the obtained and estimated coefficient. I decided to include this piece of information in the table as well since after having run the code, I was impressed of how close the estimation actually was.

Type	ArraySize	EstimatedTime	ObtainedTime	CoeffDiff
Unsorted	10^6	0.44	0.48	0.04
Sorted	10^6	0.49	0.4	0.09

Table 1: Comparison between the estimated and obtained coefficient.

```
if(array[index] > key) {
    return false;
}
```

And apparently, the unexpected slowliness is because of the time that is lost, checking for each element the above new added condition in order to optimise the algorithm. However, it should be noted that if the element is found, this improvement does pay off timewise.

The Binary Search

The binary search is an extremely efficient algorithm, since given a sorted array, the space where the search for a specific key goes on is always diminished by half. In other words, in terms of how efficient it is, the numbers speak for themselves as follows:

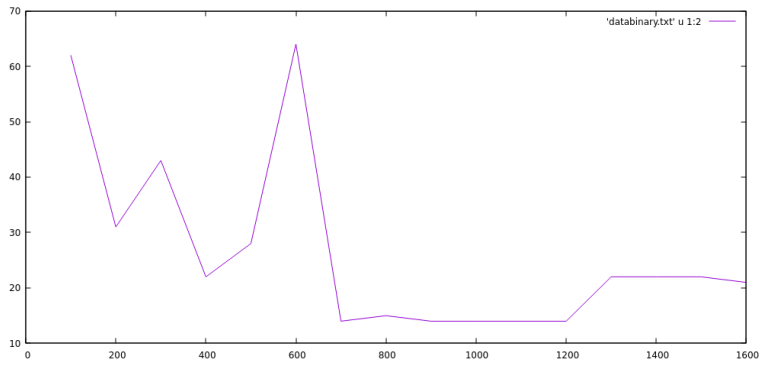

Figure 2: The function describing the binary search execution time.

As far as the graph above is concerned, its relevant part is precisely where the almost linear parts are *(starting on the x-axis with an array of 700 elements)*, since it's there where we can notice the almost constant change

when increasing the number of elements in an array and in other words, observe the logarithmic behaviour of the binary search.

On the other hand, the spikes are not of particular interest simply because we can not draw any conclusion. And that is because of the high contrast between their peak value and the other ones. This contrast is most probably due to the fact that in the beginning, those values are not cached.

n	LinBinRatio
100	6.4
200	4.3
400	9.1
600	5.7
800	16.4
1000	21.6
1200	25.3
1400	19.0
1600	23.4

Table 2: The evolution of the ratio between the linear and binary search depending on the array's size.

To put it briefly, the linear search in *a sorted array* is much less efficient than the binary one and the obtained ratio between the two is proportional with the array's size. In other words, as the data set grows, so does it. To illustrate this, for an array of 1 million elements, the linear search is **11363.7596** times slower than binary one. However, that is not it. Turns out, the binary search becomes even more efficient as the array grows. For example, my timewise estimation for a 64 million long array was $58 * 10^3$ *greater* than the actual result, which was highly impressive to say the least.

Looking for duplicates

As discussed above, looking for duplicates in an unsorted array, will undoubtedly be the most inefficient case since it is an algorithm that goes under the category of $O(n^2)$. That is precisely why, in this subsection, I decided to focus on the sorted arrays related experiments, which we assume do not contain duplicates themselves.

After having set up my code for the final, improved version described in the PDF, I noticed that, in order to look for duplicates, the new algorithm, which uses two sorted arrays is **213** times better than the normal binary search for the given benchmark, therefore for an array of maximum 1600 elements.

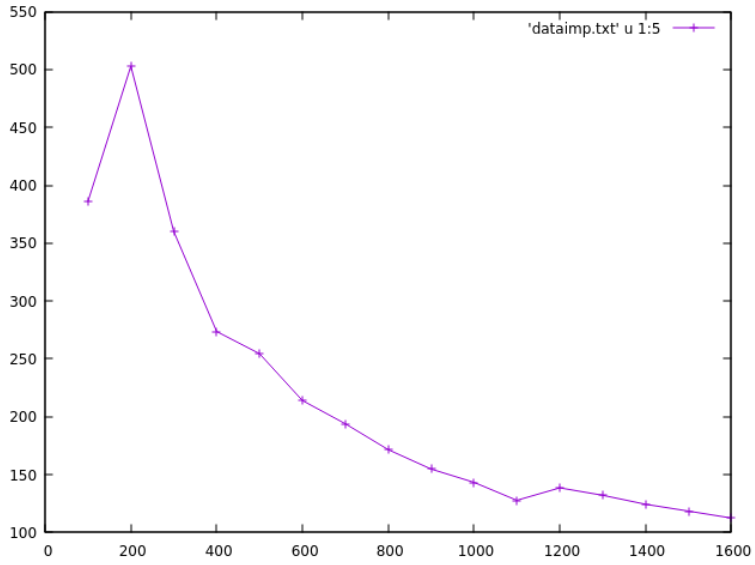

Figure 3: The binary and improved version ratio evolution.

What's interesting about the evolution of the ratio between the binary search and the improved alternative is that the ratio between the two decreases, eventually *tending towards 0* as the array increases. Take the case of an array of 1 million elements for example:

Size	BinarySearchTime [ns]	ImprovedAlgTime [ns]	Ratio
10^6	151	787	0.1

Table 3: Time analysis for an array of 1 million elements.

As a conclusion, in spite of the first impression, for large data sets, the binary search still remains the most efficient!

```
public static void better (int [] array, int [] index) {
    int j=-1, i=-1, count = 0;
    while (i < array.length -1 && j < index.length -1) {

        if(index[j+1] < array[i+1])
            j++;
        else {
            if (index[j+1] == array[i+1])
                count++;
            i++;
        }
}}
```

Note: I have included above, my code set up for the algorithm described in the assignment PDF.

Difficulties

The main difficulty I have encountered in this assignment was related on how large should the set of the interval from where I generated the random arrays and keys be. Had it been too small, the probability to have duplicates and therefore, not so relevant results for the time analysis would increase. In other words, I was not able to come up with a reliable depencency between time and the data set. After having drawn this conclusion, I, therefore, randomly populated an array and picked a key from an interval equal to the array's size.

Conclusion

All in all, after having reached the end of this paper, numbers undoubtedly speak for temsselves. To put this in other words and in order to answer the last question of this experiment, operating on sorted data sets is much more efficient than working on unsorted ones. However, how much does sorting cost in terms of resources will be elaborated in the next report!