Searching in a sorted array

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the third assignment "Searching in a sorted array" on the course "ID2021" TIDAB, KTH. The assignment is to try different algorithms for searching in a sorted array and compare them. As an introduction we also compare searching in an unsorted array to a sorted one. The different cases are then compared to each other in form of bench-marking with different array sizes to be able to analyze and form a conclusion form the result.

Unsorted vs sorted

At first an unsorted array is searched through for a specific key number. Since the array is unsorted, we have to search through the whole array. We then search through a sorted array where we add the optimization to stop if we end up on an array element larger than our key.

Code for filling the sorted array with values was given in the assignment. Below is code for filling the unsorted array:

```
private static int[] sorted(int n) {
    Random rnd = new Random();
    int[] array = new int[n];
    for (int i = 0; i < n ; i++) {
        array[i] = rnd.nextInt(10);
    }
    return array;
}
```

Optimized search:

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

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

Figure 1 shows the mean value from 5 different measurements on both unsorted and sorted. Every measurement is a mean from one million runs repeated five times. All measurements in the report are done this way. What we can see from the result is that the optimization in the sorted array slightly decreased the mean execution time. However both methods have a linear time complexity on O(n) with regards to the input data.

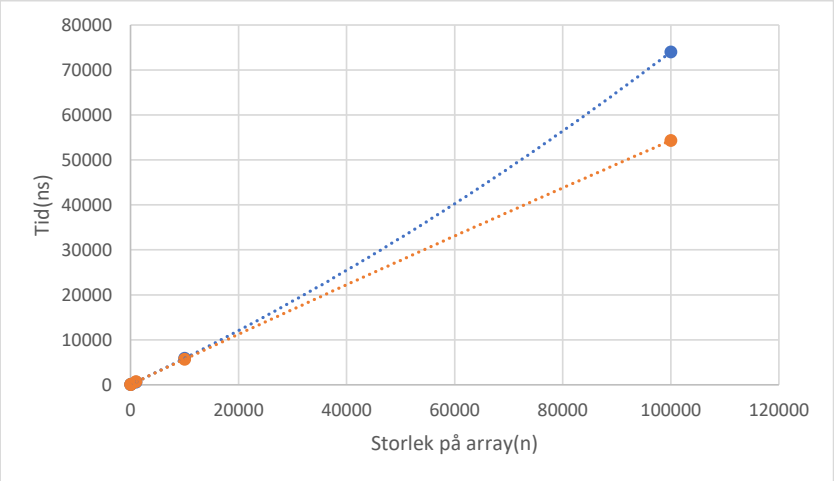

Figure 1: Measurements of unsorted(blue) and sorted(orange) compared. Run-time in ns.

Binary search

The next task is to implement an algorithm called *Binary search* where you start in the middle of the array, look if your key is there, if not you move

your index in the correct direction in reference to the middle value either a quarter forwards or backwards. This process is repeated until the correct key is found. This algorithm was incredibly superior to the previously benchmarked version of just going through the list. Searching through an array of one million elements to find a key took on average 270 ns. The results for the mean value of five similar measurements is shown in figre 2. The code for the algorithm is show below.

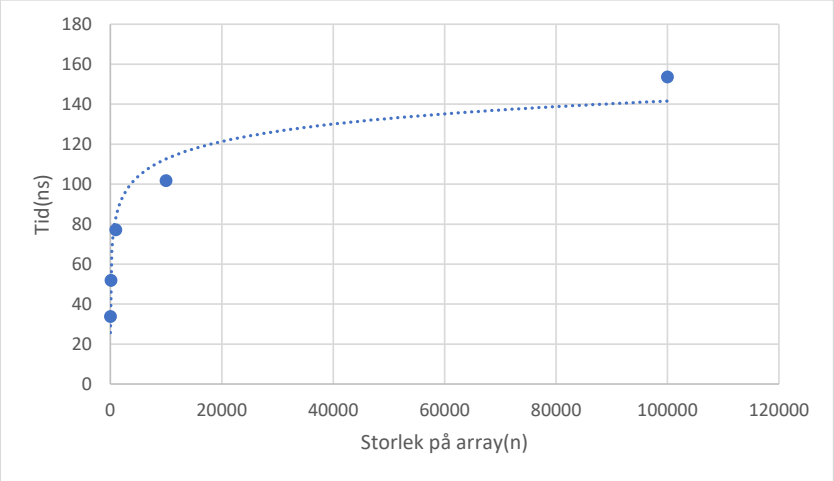

Figure 2: Measurements of searching for a key using *Binary search*. Runtime in ns.

```
while (true) {
    int index = first + ((last-first)/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 -1 ;
            continue;
        }
        break;
        }
        return false;
    }
```

The results show that the execution time of the algorithm based on input gives a logarithmic function. From these measurements we get the function $y = 12,6 * ln(x) - 3,3$, as an estimate then 64 million runs should take 223 ns. The actual result of this measurement was 1200 ns which proves that the any conclusion from data collected is in the right direction but not very precise. To improve on this more measurements could have been taken to be able to form a more precise function.

Duplicates

In assignment 1 of this course we did a *Duplicates search* where we have a list of key elements and a list of the same length with other elements and then search through to see if the two lists have any matching elements. This gave us a time complexity of $O(n^2)$ since it was dependent on the input of both lists. In this task the lists will instead be sorted and we will be using *binary search* which will give us the time complexity of $O(n * log(n))$. The bench-marking from assignment 1 gave a mean execution time of 127 ns for an array size of 10 elements and 6500 ns for an array size of 100 elements. The results from the new improved duplicates search is shown in table 1.

Array size(n)	**10**	**100**	**1000**	**10000**	**100000**
Run-time(ns)	34	52	77	102	154

Table 1: Measurements of the improved duplicates search. Run-time in ns.

The results show, as expected, a great improvement when using the *binary search* algorithm for finding the duplicate values. The execution time now grows closer with the input data instead of faster which was the case with $O(n^2)$.

Even better

To try and make the duplicates search even better the task is to try to make an algorithm that keeps track of the elements in both lists and uses the fact that the lists are sorted and logic to advance through the lists. Given that

an element in the second list is smaller than the key element we are looking for we can move on. The code for this new algorithm is show below:

```
        while (ArrayIndex < array.length && KeyIndex < key.length ) {
            if (array[ArrayIndex] == key[KeyIndex]) {
                KeyIndex++;
                sum++;
                continue;
            }
            if (array[ArrayIndex] < key[KeyIndex]) {
                ArrayIndex++;
                continue;
            }
            if (array[ArrayIndex] > key[KeyIndex]) {
                KeyIndex++;
                continue;
            }
        }
        return sum;
    }
```

Figure 3 shows a comparison between the first search for duplicates and this new improved one.

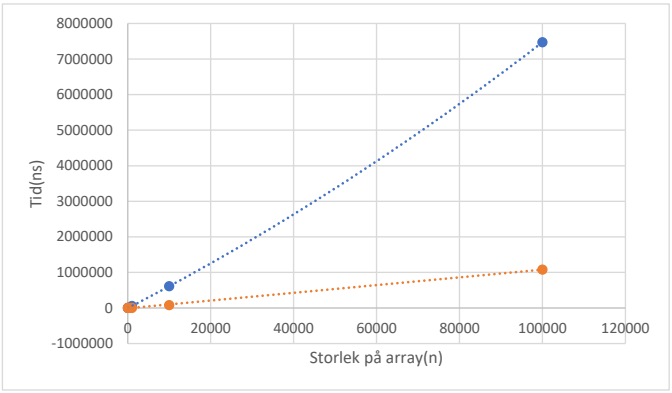

Figure 3: Measurements of duplicates searches *binary search*(blue) and the new even better algorithm(orange). Run-time in ns

We can see a great improvement, but how great? Since the algorithm is able to use logic to not compare all elements similarly as the improvement in the first sorted list at the start of this assignment we get closer to a linear function. This gives the new algorithm a time complexity of O(n).