Searching in a sorted array

Mahdi Nazari

September 2022

Introduction

In this assignment the efficiency of different algorithms will be benchmarked. To make it clear the all results will be the average of the execution time. Also the benefits of using sorted array and its impact on the execution time will be discussed. The benchmark for every algorithm is constructed to contain a for-loop of 10000 iterations and the execution time for every iteration will be summed up in a variable and at the end the average time will be calculated by dividing the total time by number of loops.

Searching in an unsorted array

In the first section a number of unsorted arrays were sent to the benchmark for the method of "search-unsorted". The result from measurements was as the first column in the Table 1 below. To see the relation between the results easier the measurement results were rounded off.(OBS! to save some spaces in the report, the benchmarking results for just 6 different sizes of array are presented in the Table 1, 2, but the benchmarking was performed with 16 different sizes of arrays which are used in the Graph 1, 2)

number	**search-unsorted**	**search-sorted**	**binary-search**
100	400	200	180
200	530	270	270
300	700	340	360
400	900	400	450
500	1200	490	550
600	1500	570	640

Table 1: The runtime for benchmarking of search-unsorted, search-sorted and binary-search with different sizes of arrays.

As the result shows the method "search-unsorted" is likely to be a linear function.(look at blue line in Graph 1 in the next section)This means that the runtime will increase proportional to the number of elements in the array. The time complexity of this function will be an O(n) type.

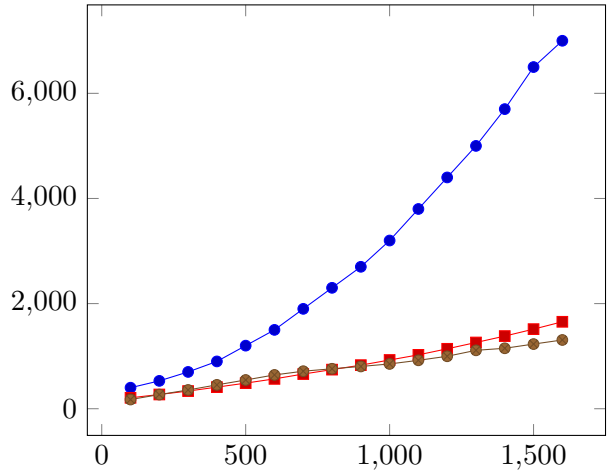
Searching in a sorted array

In the next step a new method was created to search in a sorted array. This method was enhanced and would stop searching for the key as soon as the next element in the array was larger than the key. The key was chosen randomly every time from 0 up to the size of the array.To generate a sorted array without duplicates, a function was given in the assignment and this function will be used in the benchmark. The new created method was called "search-sorted" and the function is as follows:

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

The method "search-sorted" was benchmarked and the result is presented in the second column in the Table 1, above: (look at the second column in Table 1 in the previous section)

As the result demonstrates, also in this case the runtime is proportional to the number of entries in the array. This means that the time complexity for the function is an O(n) type. Even if the time complexity of this method is the same type as the method "search-unsorted" but there are some differences between these two methods regard to runtime. To demonstrate the differences, the results from benchmarks are shown in a graph below. (Graph 1)



Graph 1: Results from benchmarking of search-unsorted(blue line) , search-sorted(red line) and binary-search(brown line). x-axis, size of arrays, y-axis the average execution time in ns.

As the graph shows it clearly, the slope of the red line (search-sorted) is not as steep as the slope of the blue line(search-unsorted). This means that the runtime for the "search-unsorted" will be much longer than the "search-sorted" when the number of the elements in the array grows. (The result for search-unsorted is not a perfect linear function, but regard to its trend it can be seen as a linear function)
To estimate how long time it takes to run the function "search-sorted" with an array of size 1 million: a formula for the function can be as: y(n) = c*n where (y) is the runtime in nanoseconds, (c) is a constant and (n) is the size of the array. Points (100,208) and (200,271) are located on the red line, then c= (271-208) / (200 - 100) = 0.63
y(1000000) = 0.63 * 1000000 = 630000 ns.
Now the function was run with an array of size 1 M and the execution time was calculated: 79253 ns. (quite close to the estimated time)

Binary search

The function "binary-search" was completed to generate the result as it was described in the assignment. A benchmark was constructed to send a sorted array of different sizes and a key which was chosen randomly every time. To make sure that the runtime calculated by javas clock system is reliable a for-loop with 10000 iterations was created so that in every iteration the runtime was calculated and summed up in a variable which was called "t-total". At the end the average time was calculated by dividing "t-total" / 10000. After benchmarking the "binary-search" algorithm, the results was as in the third column in the Table 1.(look at the Table 1 in the first section)

The results in the Table 1 and Graph 1 (brown line) shows that the binary-search algorithm is likely to be an O(log n) type. This means that this algorithm is more efficient regard to execution time than the two previous algorithms which were O(n) type. (This fact is obvious in the Graph 1 in the previous section)

Finding duplicates in two sorted arrays

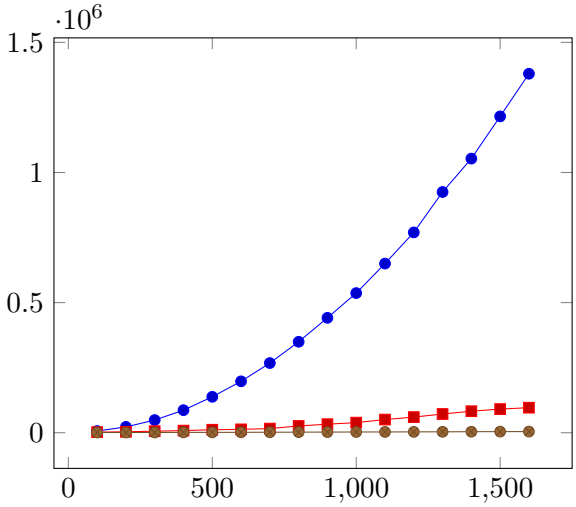
In the previous assignment it was proved that the time complexity for finding duplicates in two unsorted arrays with length of n will be an $O(n^2)$ type. In this assignment, for finding duplicates in two arrays, two sorted arrays and the binary-search will be used. A new method which will be called "findDup-binary" was created and the result for benchmarking of this method will be presented in the column 2, Table 2, and Graph 2(red line) below.
Now a new method will be created which is optimized even more. This new method was called "optimized-findDup" and in this method the function will keep track of the next element and will increment and decrement the index of these two arrays if the elements of these arrays are not equal to each other. A part of the implemented function looks like as follows:

```
for(int k=0; k<loop; k++){
    int i = 0;
    long t0 = System.nanoTime();
    for(int j=0; j<keys.length; j++){
        if(keys[j] == array[i]){
            sum++;
            continue;
        }
        if(keys[j] < array[i] && j<keys.length-1){
            continue;
        }
        if(keys[j] > array[i] && i<array.length-1){
            i++;
            j--;
            continue;
        }
    }
    t_total += (System.nanoTime() - t0);
}
```

The result from benchmarking of these three algorithms are presented in the Table 2 and Graph 2 below. To see the relation between the results easier the measurement results were rounded off.

number	findDup-unsorted	findDup-binary	optimized-findDup
100	6600	2300	940
200	2200	3700	870
300	49000	6000	970
400	87000	8200	1520
500	138000	11500	1440
600	198000	13000	1700

Table 2: The runtime for benchmarking of findDup-unsorted, findDup-binary and optimized-findDup with different sizes of arrays.



Graph 2: Results from benchmarking of findDup-unsorted(blue line) , findDup-binary(red line) and optimized-findDup(brown line). x-axis, size of arrays, y-axis the average execution time in ns.

The result from benchmarking which are presented in the Table 2 and Graph 2 shows that findDup-unsorted(blue line) is an $O(n^2)$ type as it was proved in the previous assignment.
The method findDup-binary (red line) is an O(n log n) type because the binary-search is an O(log n) type as it was discussed in the last section and the binary-search will be used (n) times. For a large (n) the O(n) type will be dominant therefor the graph of the function is likely to be a linear.
The method optimized-findDup(brown line) is a linear function and is an O(n) type. But as it sees in the Graph 2 the slope of findDup-binary(red line) is steeper than the slope of optimized-findDup (brown line). This means the execution time for findDup-binary will increase faster than optimized-findDup when the size of array grows.