Advantage of Sorted Data Report.

Adrian Jonsson Sjödin

Fall 2022

Introduction

The goal of this assignment is to evaluate how the time complexity changes when we apply different searching algorithms, as opposed to simply searching through an array from start to end.

Task 1

Set up a benchmark for the given method that searches through an unsorted array, and determine the relationship of how the time increases for a growing number of elements in the array.

Method

The method to benchmark was already given and required no further modification. What was left to implement for this task was solely the benchmarking of the given method and a method creating an unsorted array. The implementation of the benchmarking method can be seen in the code bellow.

```
public static double benchmarkUnsortedSearch(int maxArraySize) {
  Random rnd = new Random();
  int[] arrayToSearch = createRandomArray(maxArraySize);
  double sum = 0;
  for (int i = 0; i < 100_000; i++) {
    int key = rnd.nextInt(arrayToSearch.length);
    long timeStart = System.nanoTime();
    searchUnsorted(arrayToSearch, key);
    sum += (double) (System.nanoTime() - timeStart) ;
  }
  return sum / 100_000;
}
```

Result

In fig. 1 we see a plot of how the execution time in nano seconds increases when the array size increase. The data points are the average execution time for an array of size n over $100,000$ searches with a new random element to search for in each search.

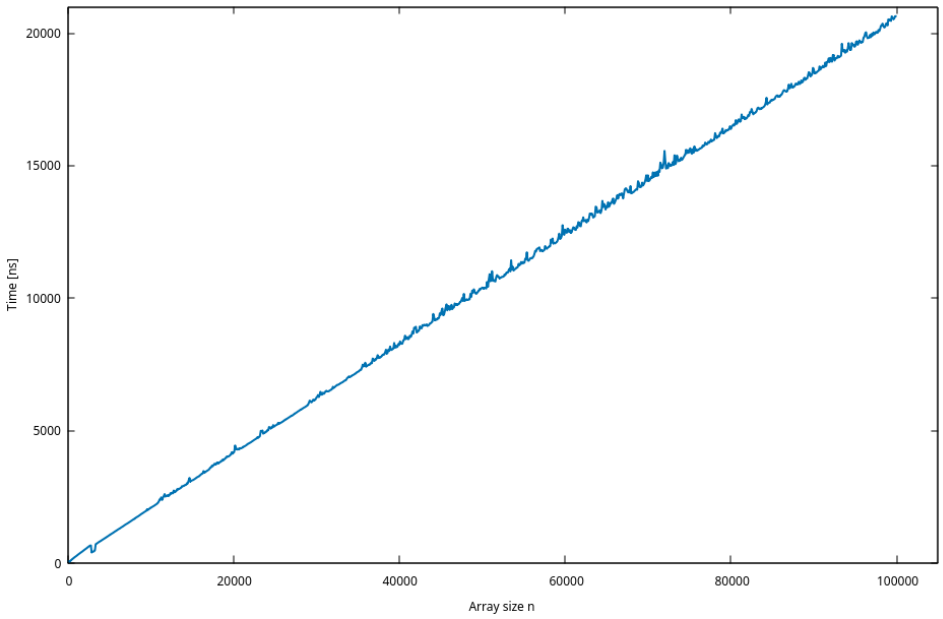

Figure 1: Linear search of unsorted search

Discussion

In plot 1 we can clearly see that the time complexity for the unsorted search was $\mathcal{O}(n)$. This was expected since the searching algorithm will have to go through each element from element 0 up to element n, which is an operation of time complexity $\mathcal{O}(n)$.

The reason why I measured the average time for a search in the benchmark test is because I generated a random key to search for each time, and if I would have taken just the shortest amount of time it took to find said key then the result would be about the same no matter the size of the array. This since it is highly likely that out of the $100,000$ runs of the search algorithm for each array size, a key with a value close to, or the same as, index 0 would be generated. This taking the average will give a result that more accurately reflects reality.

Task 2

Using the provided method that creates a sorted array, examine how long it will take to find a random key in a sorted array. Furthermore estimate how long time it would take to search through an array of one million entries.

Method

For this task we simply need to change one line in the code for the benchmark method and that is the line responsible for calling the method creating the array. Instead of creating an array with random element values we create a sorted array instead. Then we just run this array through the linear search method.

Result

Same kind of plot as in the previous task. However for this one we only gather data for arrays increasing in size by a 1000 instead of a 100 as in the previous section. An approximate function for the graph can be derived

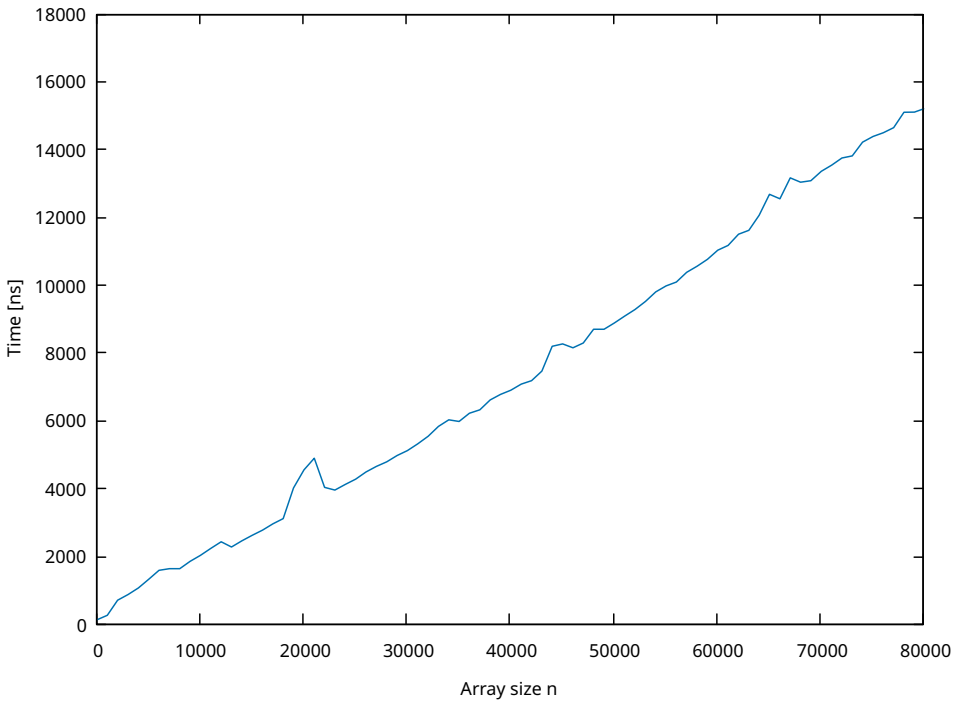

Figure 2: Linear search of sorted array

from the linear equation $y = kx + m$, where $k = \frac{y_2 - y_1}{x_2 - x_1}$. This give us the that the time increases approximately as $t(n) = 0.2n$ and thus it would take $200,000ns = 200\mu s$ to search through an array of size one million.

Discussion

As I expected the time complexity for the linear search of the sorted array is the same as for the unsorted array, that is $\mathcal{O}(n)$. This is expected since we're still searching through the array from start to finish and looking for a random key, and in so doing ignoring and not taking advantage of the fact that the array is sorted. However if we modify the linear search algorithm to stop when the current array element value is larger than the key, it is possible to lower the time usage. It will however still be of the order $\mathcal{O}(n)$

Task 3

Implement the binary search algorithm and benchmark the algorithm. From the benchmark results approximate a function that describe how the execution time increases with the size of the array. Test the approximation by running benchmarks for arrays up to 16 million. Lastly estimate how long it would take to search through an array of size 64 million.

Method

The implementation of the binary search algorithm is straight forward and already described in the assignment and the for the benchmarking we simply use the same method as already described above, but changing which searching method we are calling.

Result

Data from benchmark of the binary search algorithm run on arrays of size $n = 100 \rightarrow 10^6$. Using excel we get the function $t(n) = 10.5 \cdot ln(n)$ describing the time-array size dependency.

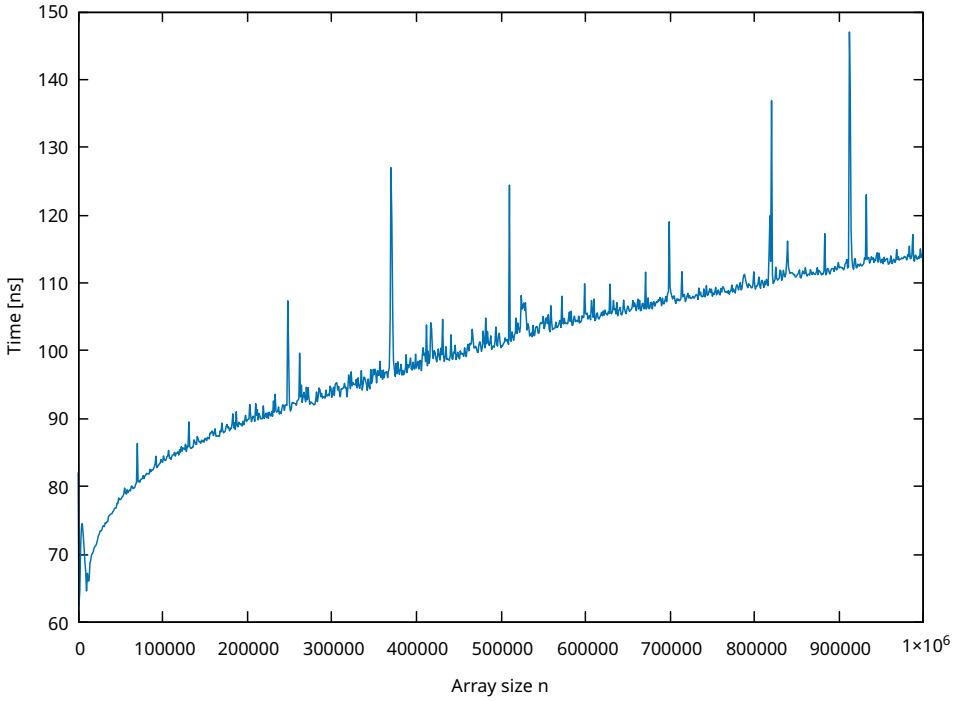

Figure 3: Binary search for $n = 100 \rightarrow 10^6$

Discussion

As seen in the figure above the result clearly shows that the time complexity for the binary search algorithm is $\mathcal{O}(log(n))$. Using the approximated function we get that it will take $10.5 \cdot ln(16 \cdot 10^6) = 174ns$ to search for a random element in an array 16 million long. Running a benchmark on an array of this size we get that it takes on average $152ns$ This seems to match pretty well with the calculated result, but perhaps we should adjust it to be $t(n) = 9.5 \cdot ln(n)$ instead. Using this adjusted function we get that it will take $170ns$ to search through an array of size 64 million.

Task 4

Search through two sorted array for duplicated elements. Do this first by implementing the binary search algorithm to try and get the time complexity down under $\mathcal{O}(n^2)$.

Method

There's two different ways we can achieve this task. The first and easiest way is to simply loop through the first sorted array linearly and use each element as the search parameter for the binary search method from task 3. This should in theory achieve a time complexity of $\mathcal{O}(n \cdot log(n))$.

The second method is to take advantage of the fact that both arrays are sorted from low to high. Instead of searching through one of the arrays linearly and the other one with binary search we can instead look at the elements in both arrays simultaneously and compare them. I've included the code bellow for just the comparing algorithm since it is easier to understand how it works that way rather than me trying to explain in words.

```
while (index1 < (sortedArray1.length) &&
        index2 < sortedArray2.length) {
  if (sortedArray2[index2] < sortedArray1[index1]) {
    index2++;
  } else if (sortedArray1[index1] == sortedArray2[index2]) {
    System.out.println("Duplicate found! " + sortedArray1[index1] +
    " " + sortedArray2[index2]);
    index1++;
  } else if (sortedArray1[index1] < sortedArray2[index2]) {
    index1++;
  }
}
```

This method should in theory have a time complexity of $\mathcal{O}(n)$ since while we're going through two arrays from start to finish, we are going through them at the same time.

Result

The result for the first method can be seen in fig. 4 and the second method in fig. 5

Discussion

The first method seen, in fig. 4, was expected to be of time complexity $\mathcal{O}(n \cdot log(n))$ but seem to be of time complexity $\mathcal{O}(n)$ instead. As in task 2 we get that the equation $t(n) = 0.04 \cdot n$ approximates how the execution time (in μs) increases with the size of the array. The reason for this linear time increase as opposed to the expected time increase is for me unknown, but a guess is that I simply didn't run my measurements on arrays of a sufficient size.

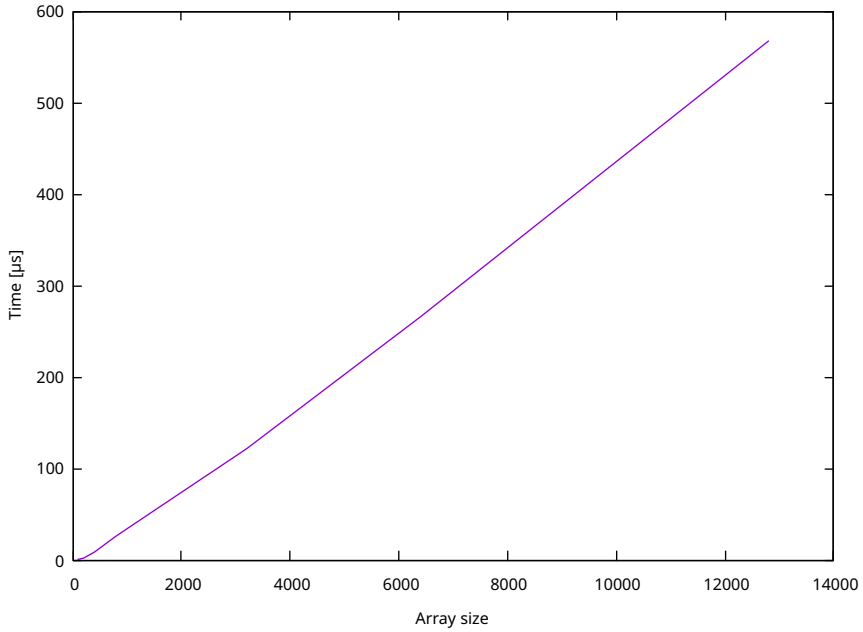

Figure 4: Duplicate search

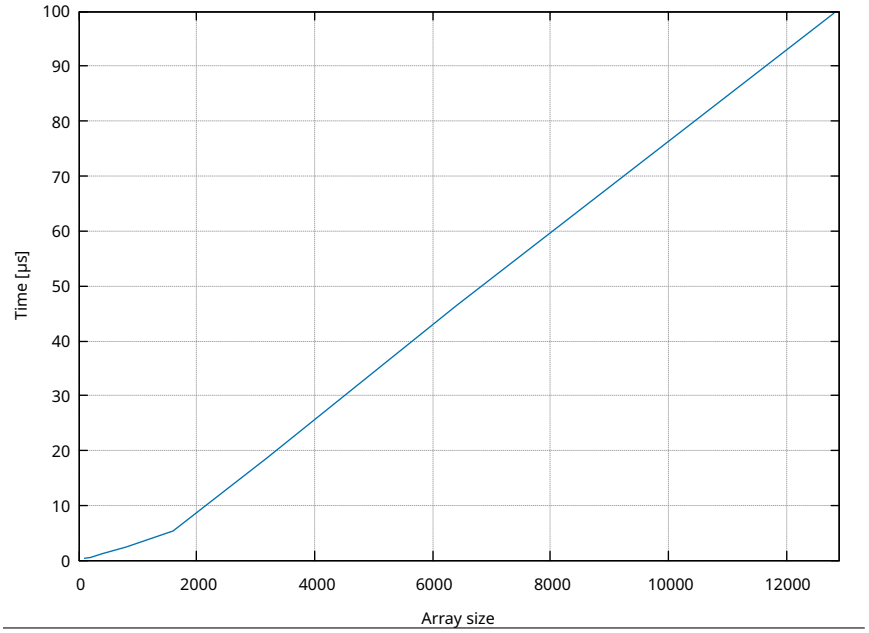

Figure 5: Optimized duplicate search

The results for the second optimized method of searching for duplicates (fig. 5) however seem to better match with what was expected. It can

clearly be seen that the time complexity here is $\mathcal{O}(n)$ with the following equation approximating the time increase for arrays of size $n > 1800$: $t(n) = 8.5 \cdot 10^{-3}n$ For smaller arrays it is still linear but with an even smaller coefficient. This is most likely do to the physical hardware and where the array is stored. For the smaller arrays they might be stored in the first cache which is faster to access and then when the array becomes to large they have to be stored outside this cache, thus changing the time coefficient.

Whatever the case is though it is clear that this optimized version is faster than the first method, and vastly better than doing a liner search through two arrays like we did for the first assignment. Enough so that it is worth considering sorting the arrays before running the searching algorithm.

The code for this assignment can be found here GitHub