Arrays and performance in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment we explore the efficiency of different operations over an array of elements.

Random access

In this first task we try and measure the time it takes to perform a single memory access. We are using java's `nanoTime()` method which returns the current value of the most precise available system timer, in nanoseconds. To be able to use this, we need to figure out it's accuracy. The following table contains the measurements for 5 iterations.

iteration	execution time [ns]
1	100
2	200
3	200
4	300
5	100

This shows that it's probably not accurate enough to measure operations with runtime of just a couple nanoseconds, so we concluded that due to the resolution of the clock, we would need to look at a couple hundred of read/write operations to get a better idea of the time it takes to perform a single one. There may also be inaccuracies due to caching, so in the next section we will use random read operations.
Next we looked at the average, minimum and maximum value we measured for a single memory access, and noticed that the average and maximum time varied quite a lot, whereas the minimum time was a constant value of around 6 ns. If then we added a run of our method to the beginning of our code, we could see all three drop dramatically. This is simply because of the Java compiler.

	without Java optimization	**with optimization**
avg	6.4	1.8
min	5.8	1.6
max	7.5	3.0

Average, minimum and maximum execution time

We decided to only look at the minimum value for the different array sizes, measurements shown in table below.

n (size of array)	**execution time** [µs]
100	1.6
200	1.6
400	1.7
800	1.5
1600	1.5
3200	1.6

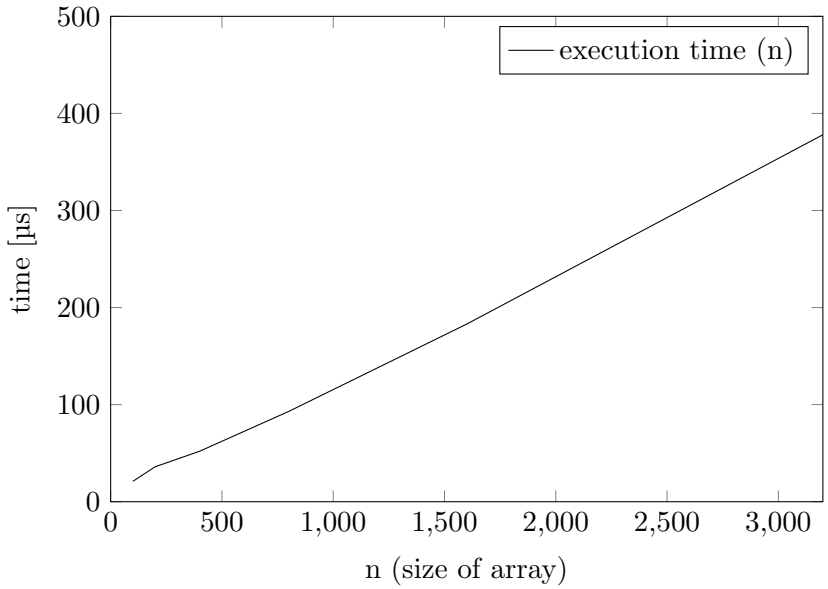
This table clearly shows that the method we were looking at is of O(1), meaning that its execution time does not depend on its size.

Search for an item

In order to test the time complexity of searching for an arbitrary element in an array of size `n`, we once again fill up an array with random elements and another one with the random keys that we will be searching for. I tested searching for 1000 keys in an array of increasing size using a similar structure as before, and got the results shown in the table below.

n (size of array)	**execution time** [µs]
100	21
200	36
400	52
800	93
1600	183
3200	378

We can see that every time the problem size doubles, the execution time is roughly doubled too. This searching algorithm appears linear, so it is O(n). I did also notice that sometimes there were some anomalies, e.g. the execution of the method on 100 elements or 300 elements would have an unreason-



Graph showing the linear growth in execution time

ably high execution time most likely due to the operation system favoring some other running program, which I assumed may be minimised by increasing `k` from 10 to say 200, so that we are more likely to catch at least one optimal run. After doing so, I compared the results from these two programs in the table below.

n	**exec time prog 1**[µs]	**exec time prog 2** [µs]	**conclusion** [ms]
100	21	15	0.01
200	36	29	0.03
400	52	49	0.05
800	93	87	0.09
1600	183	163	0.17
3200	378	315	0.34

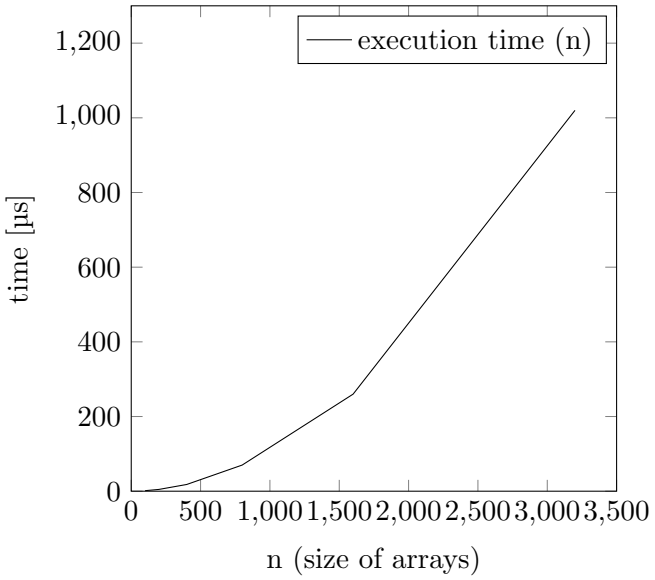
And lastly, I also noticed that adding a similar JIT warmup as before did not significantly improve the execution time, however, plugging my PC in definitely makes a difference when it comes to execution times.

Searching for duplicates

For this final task we were looking at finding duplicates in the two arrays from before. We kept the same algorithm for the search, but now we also increase the number of keys. For the sake of simplicity, we are only dealing with the case where both arrays have the same size, and there are no duplicates in either of the arrays. Below are my measurements.

n (size of arrays)	**execution time** [µs]
100	1.4
200	4.8
400	19
800	70
1600	260
3200	1020

We can see that as the size of the arrays double, the execution times grows by a factor of 4, so it is $O(n^2)$.



Graph showing execution time for the increasing array sizes