Introduction-report

Emanuel Paraschiv

Fall 2024

Introduction

In this report we had to benchmark different operations for array data structures for observing their time execution (Random access, Searching and finding Duplicates).

Random access

The tool we are using for measuring the time execution is the `nanoTime()` method in java. In general, as we don't depend on a variable, the memory access should take the same time, having a constant time complexity, $O(1)$.

```
for (int i = 0; i < 10; i++)
  long n0 = System.nanoTime();
  long n1 = System.nanoTime();
```

After running this piece of code however, we find that the clock measurements seems to differ:

resolution	**microsec**
resolution 1	0.2 μs
resolution 2	0.1 μs
resolution 3	0.1 μs
resolution 4	0.1 μs
resolution 5	0.1 μs
resolution 6	0.1 μs
resolution 7	0.1 μs
resolution 8	0.1 μs
resolution 9	0.09 μs
resolution 10	0.09 μs

Table 1: Time in μs

We use this clock now to measure the access time of an element. From the results we can notice that it is not accurate enough for an access operation:

```
long t0 = System.nanoTime();
sum += given[i];
long t1 = System.nanoTime();
```

iteration	**microsec**
operation 1	0.3 μs
operation 2	0.2 μs
operation 3	0.1 μs
operation 4	0.1 μs
operation 5	0.1 μs

Table 2: Time of accessing elements in an array

One reason for the inconsistency could also be that we are accessing the array elements consecutively and thus spatial locality is used. This way we can see that it takes shorter to access elements after the cache fetches the first few elements.

First Benchmark-Random Access

In order to avoid that, we are making the access random:

```
long t0 = System.nanoTime();
  sum += given[rnd.nextInt(10)];
long t1 = System.nanoTime();
```

The result is 0.6 μs

However, as the time to create a random number is much longer than the time to access a an element in memory the time elapsed bigger than usual.

In order to avoid this, we develop an enhanced program consisting of 2 methods. One is `bench()`, which creates 2 arrays with the user's input, one containing random numbers. This method calls the `run()` method which does the actual reading of the elements.

```
for (int i = 0; i < 10; i++) {
  long t = bench(1000, 1000);
}
```

After running the measurements for the min, max and avg of the execution, I was able to see a min time of about 18 ns. However the max time will fluctuate a lot `(in terms of 100's of ns)`, and the avg time was affected accordingly.This fluctuation is given to the operating system which will sometimes pause the program to execute other tasks.

However, given the behavior of the JIT, I managed to get down the min time to about 7 ns after adding this line of code:

```
bench(n, 1000000);
```

Basically this is happening because we warm up/ optimise the "bench" method and its execution becomes thus faster.

```
int[] sizes = {100, 200, 400, 800, 1600, 3200};
// the results in microseconds
int[] results = {24, 13, 3, 3, 3, 3};
```

When inputing different sizes of arrays, the results seems to be related: Although counter-intuitive, the smallest array has the largest access time. This is happening as the JIT is "warming up".

After that, the next arrays, although bigger seem to have a similar access time, almost equal and smaller than the first one.

However, as mentioned before, the execution time should be constant with complexity $O(1)$.

Second Benchmark-Search

In this task we try to figure out the relation between the size of an array and the execution time.

The procedure is similar to the previous task: we loop several times to get the min time, and we use 2 variables: n (holds the array size) and k which I kept as a constant. The execution time is presented below:

Array Size	**Execution Time (ms)**
100	0.5
200	0.5
400	1
800	2
1600	4
3200	8

Table 3: Execution Time vs. Array Size

It is clear that as we increase the size of the array, the execution time increases accordingly in a linear manner, complexity being $O(n)$.

The equation I was able to come up with is:

$$y = 20n + 25 \tag{1}$$

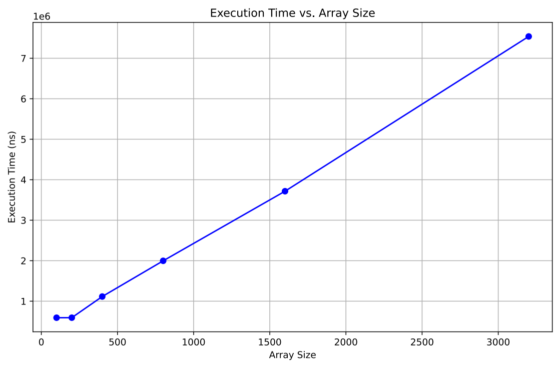

Figure 1: Execution time

Explanation: As we can see, the execution time is dependent on the size of the array (n), and increases in a linear manner.

Third Benchmark-Duplicates

In this task we are given 2 arrays ,having to measure the time for finding duplicates. This is similar to the previous task with one difference: the second array, previously of constant size k, has now same size as the first (n).

Array Size	**Execution Time (ms)**
100	0.05
2000	0.5
5000	3.5
10000	15.5
20000	64
30000	147

Table 4: Execution Time vs. Array Size

The idea is that only by looking at the code, after noticing the 2 nested for() loops we can figure out that the time complexity would be quadratic $O(n^2)$, apart from the single for() loop from the previous task.

An approximation for a polynomial to fit the points well is:

$$y = 2n^2 - 9n + 70 \tag{2}$$

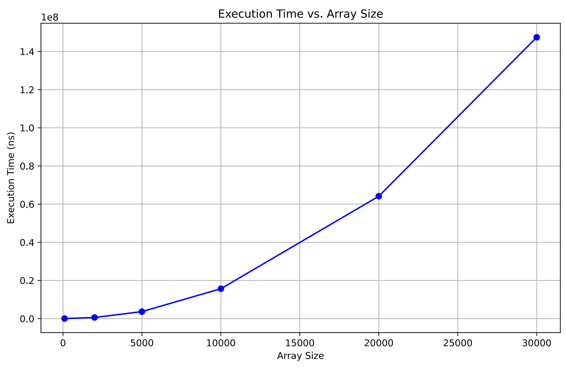

Figure 2: Execution time

Explanation:
As we can see, the execution time increases significantly with the size-growth of the array in a quadratic manner.