ID1021: Measuring time variance in different array operations

Alexander Lundqvist

30-08-2022

Introduction

The main purpose of this report is to to measure the execution time for different operations on array elements and observe what happens when we increase the size of the array.

We also discuss how these different operations differ and how they can be described mathematically. The operations described in the instructions are:

- Random access: reading or writing a value at a random location in an array.
- Search: searching through an array looking for an item.
- Duplicates: finding all common values in two arrays.

The secondary purpose is to let the student learn how to produce a well written and organized report in LaTeX.

Method

Array access

In this task we are supposed to measure the time it takes to measure the time it takes to access a random element in an array and how the time is affected when the array grows in size **n**.

To start of we create an array with random numbers ranging from 0 up to but not including n. This will serve as an array with indexes that will later be used to access another array. We continue by creating an array populated with some dummy value, in this case 1s.

To test the array access time we utilize the built in Java function nanoTime which lets us accurately measure the execution time for different code segments.

Searching

For this task we are supposed to create a method for searching after a random key in an array of unsorted keys, where the keys will have values in the interval $[0, n * 10)$. This is because we want the possibility that a given key can't be found during one search.

For this task I mostly used the code from the instructions as due to the time constraint I was very stressed and had a hard time understanding this task.

Finding duplicates

Since the last task was similar to the array searching operation there isn't much to say about the code. The only modifications that were made was making sure both arrays had the same length and since there wasn't much explanation in the instructions, I choose to interpret the task as finding every duplicate in the searched array. This was done by removing one single break statement.

```
for (int ki = 0; ki < n; ki++) {
        int key = keys[ki];
        for (int j = 0; j < n ; j++) {
            if (array[j] == key) {
                sum++;
                //break;
            }
        }
    }
```

Result

Array access

Searching

Finding duplicates

Discussion

The main lesson that I take from this exercise is the method by which we benchmark the different functions. In previous courses I haven't learned to test execution time properly, mostly it has just been measuring time at the start of the whole method then at the end and taking the difference.

Here we have done it in several different ways that eliminate a lot of factors that can influence performance testing such as randomizing array

ns	**n**
0.295	250
0.342	500
0.323	750
0.292	1000
0.295	1250
0.291	1500
0.288	1750
0.290	2000
0.285	2250
0.289	2500

Table 1: Random array access with n being the array size and ns the time in nanoseconds

ns	**n**
57.3	250
102.4	500
148.9	750
252.4	1000
329.5	1250
387.5	1500
442.8	1750
526.3	2000
591.4	2250
651.0	2500

Table 2: Array search with n being the array size and ns the time in nanoseconds

access, using dummy code to subtract time not used for the actual operations and checking for minimum and average times.

Array access

We can see that the time for accessing elements in the array doesn't increase by any substantial amount when we use larger and larger n. As we can see, the second and third iterations are outliers however, when performing the benchmark several times the aberrations appear randomly which leads me to believe it just has something to do with hiccups in the CPU.

If we would perform thousands of benchmarks and plot the data, we would see that the time would be constant (I gave Gnuplot a try but due to lack of time I wasn't able to learn the program). Therefore, the time

ns	**n**
800.0	4
1200.0	8
3300.0	16
11800.0	32
45300.0	64
188700.0	128
713000.0	256
696900.0	512
2183200.0	1024
9828500.0	2048

Table 3: Finding duplicates in array with n being the array size and ns the time in nanoseconds

complexity can be described with big-Oh notation as $O(1)$.

Searching

Although I found this task to be the hardest and didn't fully understand everything I did, I got the idea of the theory behind it, how the search algorithm works and why we measure the execution time as we do. I feel that if I revisit this task later in the course I will understand it better.

The results from the benchmark testing of the array searching method was presented in table 2. We can see that with increasing the size of the array, the execution time also gets increased. If we inspect the data more closely we notice a pattern; when we double the value of n, the execution time gets doubled (apart from outlier measurements). This means that the time is dependent on n and we can say that the code has time complexity of $O(n)$.

Finding duplicates

When inspecting the the measurement data in table 3, we can see how the time does not get doubled when we double n, rather it gets multiplied by 4. This got me thinking that it can't be just $O(n)$ but some other function that describes the time complexity.

We can also take a look at the code to determine the time complexity. A simple way of calculating the time it takes for some code to execute is to look at how many operations are performed. As we have two arrays, A and B, of the same length and we search in array B for every element in A, we can say that we do "length-of-B" number of operations (with big-Oh value of $O(1)$ for elementary operations) times the "length-of-A". Mathematically this can be represented as (with the length of the arrays being n):

$$(O(1)_1+O(1)_2+O(1)_3+...+O(1)_n)*(O(1)_1+O(1)_2+O(1)_3+...+O(1)_n) =$$

$$(O(1*n))*(O(1*n)) = O(n)*O(n) = O(n^2)$$

Hence, the time complexity is quadratic and will grow exponentially when n gets larger.

As for the question of how large n we would be able to handle given one hour of computation time it would largely depend on which computer it would be tested on. We could run multiple tests and then run the data through excel for instance and find a trend line with which we could approximate n at one hour. But this is far too time consuming in my opinion, at least for me. We can however do some calculations to get our approximation. Unfortunately I ran out of time..