Assignment 1 Report

Dean Tsankov

August 31, 2024

Contents

Introduction

In this report I will present my work on the first "Introduction" assignment. Here I should explore the efficiency of different operations over an array of elements. I should focus on performance measurement, presenting numbers and realizing that there is a fundamental difference between what is called O(1), O(n)and O(n^2)

I should benchmark three different operations and determine how the execution time differs with the size of the array. The three operations 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.

1 Random access

When trying to measure how much time an operation takes, one is limited by the resolution of the clock. If we have a clock that measures microseconds then it won't be possible to accurately measure an operation that only takes a few nanoseconds.

The Clock

In order to measure the performance we need a method to measure time. My first task is to figure out the accuracy of the provided clock code.

This proves to be fairly difficult as with each execution of the code the returned values differ greatly. This difference is too big and as such would not allow us to measure the required tasks accurately.

	test run 1	**test run 2**	**test run 3**	**test run 4**
outputs	200	200.0	200	100
	200	101	101	100
	99	101	101	100
	101	101	101	101
	0	100	100	0
	101	101	0	0
	100	0	100	0
	101	0	100	100
	100	101	101	99
	0	100	0	100

Table 1: Output of the clock code ran multiple times, output in nanoseconds

Conclusion

After constructing the final version of our random access test code, suited to the most degree for measuring the access time over an array at a random location, we can test run it a few times and view the results.

array size	test 1	test 2	test 3	test 4
100	11.4 μs	5.1 μs	11.4 μs	11.4 μs
200	5.6 μs	2.1 μs	2.1 μs	2.1 μs
400	7.7 μs	2.1 μs	2.1 μs	2.5 μs
800	2.1 μs	2.1 μs	2.1 μs	2.8 μs
1600	2.2 μs	2.2 μs	2.1 μs	2.1 μs
3200	2.1 μs	2.2 μs	2.1 μs	2.1 μs

Table 2: Output of the random access code ran multiple times

Except the first row all other values seam to be in right about the same ball park. This is, however, not surprising as the complexity of the code is constant and as such does not depend on the size of the array.

2 Search

Once we know how to set up a nice benchmark we can explore a simple search algorithm and see how the execution time varies with the size of the array. When we setup the benchmark we want to capture the estimated time it would take to search for a random element in an array of unsorted elements.

After plugging in the provided code segment and fixing an error regarding the declaration of the keys array in the search method:

```
int[] keys = new int[loop]; //originally it was new int[n];
```

We can run the code a couple of times to check the results.

array size	test 1	test 2	test 3	test 4
100	208.6 μs	209.1 μs	209.5 μs	208.8 μs
200	78.2 μs	78.2 μs	77.7 μs	78.1 μs
400	112.7 μs	113.7 μs	111.7 μs	112.4 μs
800	182.1 μs	180.6 μs	177.2 μs	174.8 μs
1600	321.9 μs	309.8 μs	311.2 μs	310.0 μs
3200	461.5 μs	458.7 μs	462.1 μs	452.4 μs

Table 3: Output of the search code ran multiple times

Conclusion

Except the longer time it takes for all tests with array size 100, the other results show a formulaic increase in execution time as the array size becomes larger. This is of course expected as its is logical that when the sample size increases so does the time to find a specific element.

This increase in execution time related to the increase of the array size can be represented with a polynomial of the type $O(n)$ since the complexity of the execution linearly changes as n (size of array) changes.

3 Duplicates

For the final task, we need to find duplicated numbers in two arrays of size n i.e. an element in the first array that also is present in the second array. This task is very similar to the search exercise - for a given element in the first array we will find it in the second array. The difference is that now both arrays will grow in size. This seemingly small change makes a big difference in execution times. For a large enough array size the loop parameter's influence could be neglected. And the time to find the duplicates will be long enough to hide the uncertainty of the clock making our results a bit more showing.

I plug in the provided code for the *duplicates* method and fix one of the for loops:

```
for (int k = 0; k < n; k++) {...} //originally it was k < loop;
```

Due to how much time a single benchmark takes and the fact that this negates the inaccuracies in our clock, I will present only a single test of the method, but nonetheless this result should give us the right understanding of the behaviour of the function.

array size	**test output**
100	141.6 μs
200	1.12 ms
400	9.72 ms
800	75.36 ms
1600	583.65 ms
3200	4.65 s

Table 4: Output of the duplicates code ran a single time

Conclusion

Again here we see a formulaic, this time exponential, increase in the execution times as the array size n becomes larger. This is as expected, because

to find duplicates we traverse both of the n-sized arrays and as such when we double n a quadratic time complexity should follow. Due to this we can conclude that a polynomial of type $O(n^2)$ is a representation of the benchmark.