Introduction

Jeffrey Chang

31st August 2022

Introduction

For the course ID1021 - Algorithms and Data Structures, we are given tasks to submit each week for us to learn about the course content. The first task of this course is the *Introduction* assignment where we are to find the time efficiency of different operations over an array of elements.

Background

This assignment requires us to test three different operations, namely, *Random access, Search, and Duplicates* and how efficient they are, and the main purpose is to find the fundamental difference between O(1), O(n) and O(n^2).

The implementation of benchmarking the efficiency of these operations are done by utilising the clock in Java's runtime system called *nanoTime* - which gives us the total execution time in nanoseconds, or at least that is what its name indicates.

The Tasks

The first task which was given to us was checking the actual clock accuracy of Java's inbuilt runtime **System.nanoTime**. Following the code in the instructions where we find the difference between two identical executions, it can be concluded that the operation **System.nanoTime** is not precisely accurate to the exact nanosecond as the results from the difference in operation time was rounded up to 100's of nanoseconds. The 10 different tests resulted in variations of 100 - 300 nanoseconds throughout which shows that the actual accuracy rounds the inbuilt runtime to one significant figure.

Additionally, precisely because the code runs in a loop, the executions are exactly the same, yet our clock times are not the same for each iteration. This indicates that the actual measurement may be inaccurate from the very beginning which could be cause by other programs running or perhaps a difference in the computer one has.

Task 1

For the first *proper* task we were experimenting with calling the access function with increasing size of arrays to find the time it takes and which big O notation it fits. *Table 1* from subsection **Tables** shows the time taken with different amounts of data.

The clear conclusion one can draw from this data table is that the access function with increasing size of arrays is constant and stagnates to become at 0.4 nanoseconds - so it lies in the O(n) notation. The compiler seems to struggle to optimise the speed to execute with smaller data sets, but it eventually (by data set 1000) manages to reduce the time to 0.04 nanoseconds. My personal computer seems to be able to optimise the execution time to 0.04 nanoseconds after 1000 data sets as it becomes the constant execution time even for 5000 and 10 000 data sets.

Task 2

For the second task, we were tasked with finding the big O notation for the *Search* operation by benchmarking an increasing sized array and keeping the search number constant. The point of the program is to match an amount of keys to a varying size of arrays that are randomised between 0 - 10*n.

To complete this task I decided to keep the amount of times searched at a constant 1000 times whilst the array size increased and the data result output is shown in *Table 2*. The conclusion that comes with this table is that, like in *Task 1*, here the big O notation is also O(n) which means that it is linear as the correlation between the number of data and time is parallel, i.e. it follows a linear graph.

Finding Duplicates

For this final task we had to find the duplicates in two arrays of length n. It is similar to *Task 2* but now we have one array searching through another array. This was done by benchmarking an increasing sized array n to then find a polynomial that describes the execution time and estimate the size of n that could be handled with one hour of computation time and the results and outputs of this task is shown in *Table 3*.

The results are very clearly different from the first two as it is not an exponential increase in the time measurements. This is, in the big O notation an O(n^2) and if we were to use the data from *Table 3*, we can understand that the amount of data we can produce after one hour of work is going to be the square root of 3.6 * 10^{12}, since its in O(n^2). which would give us 1.9 * 10^6 units of data.

Data

Relevant Code

Checking the accuracy of **System.nanoTime**

```
//runtime difference uising nanoTime()
public static void main (String[] args)
    for (int i = 0; i < 10; i++)
        long n0 = System.nanoTime();
        long n1 = System.nanoTime();
        System.out.println("resolution " + (n1 - n0) + " nanoseconds");
```

Task 2 Search and Randomisation

```
//randomiser search loop from 0 - 10 * n
for(int i = 0; i < m; i++)
    Keys[i] = rand.nextInt(10 * n);

for (int b = 0; j < n; j++)
    array[j] = rand.nextInt(10 * n);

//searching and finding
long t0 = System.nanoTime();
    for (int i = 0; i < n ; i++)
        if (array[i] == key)
            sum++;
total += (System.nanoTime() - t0);
```

Tables

Data	nanoTime
1	3.5
10	0.24
50	0.24
100	0.12
1000	0.04
5000	0.04
10000	0.04

Table 1: Amount of data vs Time recorded in the access function

Array size	**Runtime (ms)**
100	6
500	9
1000	16
2000	27
5000	62
10000	102

Table 2: Time taken to search through varying amounts of data

Array size	**Runtime (ms)**
100	6
500	10
1000	21
2000	74
5000	196
10000	899

Table 3: Time taken to search and amount of searches compared to varying amounts of data

Conclusion

In conclusion, this introductory assignment has taught me how to use Overleaf and the fundamental differences of the three different operations *Random access, Search, and Duplicates* and their big O notations. It was slightly more challenging than I had originally thought, but still interesting to see the compiler take its time with certain executions more than others.