Introduction

Pontus Hector

Fall 2022

Introduction

This report is conducted as the first assignment in the course ID1021. In this report one explore the efficiency and time complexities of different operations over an array of elements.

Random Access

In the first task we run the following loop to measure the time elapsed for random memory accesses.

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

Just based off of the code, one can suspect that the time complexity of this algorithm is constant (O(1)) since the execution is not dependent on any input variable. Measurements of the above standing code:

iteration	**run time**
1	0.3
2	0.3
3	0.3
4	0.4
5	0.4
6	0.4
7	0.4
8	0.4
9	0.4
10	0.4

Table 1: run time in μs

When reviewing the measurements one cannot be certain whether these results are accurate because of multiple reasons. We access the array in sequential order, which may affect the result due to caching, and adding lines of code barely changes the run time. Even with hundreds of thousands of measurements, these unwanted factors will play a role in the results one obtains which will render the results rather vague.

Benchmark 1

In the first benchmark we run an improved program with the purpose of measuring the time elapsed for random memory accesses where we further ensure **random** memory access.

```
for (int j = 0; j < c; j++) {
    for (int i = 0; i < n; i++) {
        sum = sum + array[indx[i]];
                                }
                            }
```

In this new program the algorithm introduces the variable n which at first glance might seem as the algorithm is dependent on that variable. However the amount of work of making memory accesses in this program should not depend on this variable (n) and therefore should n not prolong the actual run time.

n	**run time**
10	0.33
100	0.40
1000	0.38
10000	0.39

Table 2: run time in nanoseconds

This algorithm is clearly of constant time complexity since we basically obtain the same static run time no matter what value n obtains.

Benchmark 2

In the second benchmark we run a search algorithm where we analyze how the execution time varies with the size of the array. One can draw the conclusion that the code is linearly dependent on (n) since the code is dependent, and only dependent on n, as a varying variable ((m) and (k) are constants).

```
for (int ki = 0; ki < m; ki++) {
        int key = keys[ki];
        for (int i = 0; i < n ; i++) {
```

From the measurements below one can further see that the run time is linearly dependent since when the input grows, the algorithm takes proportionally longer to complete. One can draw the conclusion that the time complexity of this algorithm is O(n).

n	m	k	run time
1	10000	1000	3
10	10000	1000	6
100	10000	1000	38
1000	10000	1000	356

Table 3: run time in nanoseconds

Benchmark 3

In the third benchmark we run a search algorithm similar to the last benchmark where the main difference is that the size of the key array is now also dependent on the variable (n).

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

Theoretically this algorithm should be quadratic due to the construction of the loop in the code above. Here one can see that we walk through the elements n amount of times (from the first loop), n amount of times (from the second loop). Therefore ones theoretical conclusion from the code would be of that the algorithm is of quadratic time complexity (O(n^2)).

n	run time
1	0.025
10	0.05
100	2.6
1000	272

Table 4: run time in μs

The theory that the algorithm is of quadratic time complexity is furthermore supported by the obtained measured values. One hour would be sufficient to run this algorithm with about 1.500.000 - 2.500.000 n´s with this quadratic growth of time consumption.