Report 1

VIktor Lado Naess

August 26, 2023

Introduction

We begin with checking system time which can be done with the following code:

```
for (int i = 0; i < 100; i++) {
    long n0 = System.nanoTime();
    long n1 = System.nanoTime();
    System.out.println ( "Trial " + i +" resolution "
    + (n1 - n0) + " nanoseconds");
    val[i] = n1-n0;
    sum += n1-n0;}
```

Following the above we see form the 100 trials that there is some fluctuation in our results. From this we can calculate that the mean is 350 and the standard deviation is roughly 39.45 (2 d.c.). The variation can be attributed to many things, but ultimately every iteration has its own system call, the speed of which can be variable due to caching or compiler optimization. The solution to this is in the following random access part.

Trials	Values
1	300
2	400
3	300
4	300
5	200
6	300
7	300
8	400
9	300
10	400

Table 1: Table of Trials and Values

Task 1: Random Access

This part required that we determine the time value of accessing a random integer in an array. This prevents caching from playing a role. One method to accomplish this is to make two arrays, one that we are going to access and one to give us the random index for said access. For this we wish to find the smallest time value for successive arrays sizes n. Below shows the code in the main for this, as there was little discernible difference in times between the first 6 values of n (seen by the array named "size") a more extensive set of values was of interest.

```
int[] sizes = {100, 200, 400, 800, 1600, 3200, 10000,
20000, 50000, 100000, 2000000, 4000000, 10000000, 20000000};
bench(1000);
for(int size : sizes) {
double t = bench(size);
System.out.println(size + " " + t);}
```

To quickly explain this: The bench method, when given an array size n, it executes the access method (which simulates the process of accessing array elements by generating random indices and using those indices to access array elements) a total of 1000 times and returns the minimum time observed(the best-case array access scenario among those trials).

Below is a table of the Array sizes n and the corresponding time in ns. As we can see there is a clear correlation between array size and minimum access time of random value. Initially, there is little change between access time of an array elements however as we approach larger and larger array sizes the minimum access time increases.

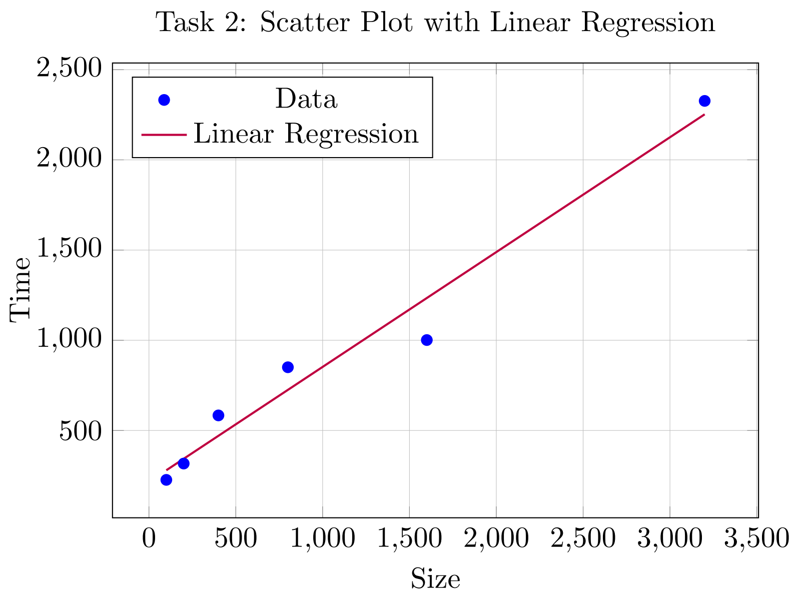
n	Time (ns)
1×10^2	0.3
1×10^4	0.3
2×10^4	0.4
5×10^4	0.4
1×10^5	0.5
2×10^6	0.9
4×10^6	3.1
1×10^7	7.5
2×10^7	12.8

Table 2: Array Access Time Benchmark

On a hardware level, this correlation between array size and minimum access time of random value is primarily because of the effect of cache memory and cache misses. As the array size increases, it becomes more likely that the elements will no longer fit entirely within the cache. When data cannot fit in the cache, the processor needs to fetch it from the slower main memory, resulting in increased access time. Thus as array size increases the likelihood of cache misses increase increasing minimum random access time.

Task2:Search

Looking at the code for this section we see a that the bench to input size variation should be O(n) as there is a linear correlation between size of the array and length of time to sort through it. To test this I tuned the size array as to be the original assessment values only, as the previously chosen values were too high or too low to be truly appreciated on the graph. The corresponding values are also easily seen by the table below.



n	Time (ns)
100	226.2
200	316.7
400	583.6
800	850.5
1600	1001.3
3200	2326.9

Table 3: Array Access Time Benchmark

Search for duplicates

Searching for duplicates was much the like previous section, however, here we have a nested loop as the one seen below. This nested loop implies that we do the function twice in a way that the previous parts did not. This implies that we have O($O(n \times n)$ which in turn implies that we have $O(n^2)$. As such we should have a quadratic function which is what we get when plot our values.

```
        for(int z= 0; z<1000; z++){
                long t0 = System.nanoTime();
                for(int i = 0; i<n; i++){
                    int val =array1[i];
                    for( int k=0; k<n; n++){
                        if (val == array2[k]){
                            break;
                        }
                    }
                }
                long t1 = System.nanoTime();

                if (min >t1-t0){
                    min= t1-t0;
                }
         }
```

n	Time (ns)
10	6.00×10^2
100	1.53×10^4
200	6.24×10^4
400	7.16×10^4
800	2.29×10^5
1600	8.88×10^5
1800	1.18×10^6
3200	3.79×10^6

Table 4: Array Access Time Benchmark

Running a quadratic regression on these values, gives the equation $0.39x^2 - 64.4x + 27004$ and an r correlation coefficient of 0.99. Hence it is likely our previous assertion of the function being of $O(n^2)$ nature, holds.