Assignment 1 Arrays and performance

Mahad Ahmed

Spring Fall 2022

Introduction

This is the first assignment for this new course ID 1021. In this course we will learn basic algorithms and data structures. In this first assignment we explored the efficiency of different operations over an array of elements. We had two tasks in this first assignment. Task 1 being that we should set up a benchmark where we call an access function with a bigger and bigger n i.e integer. After which we should present them in a table and discuss the outcome. In task 2 we had to choose two integers that would tell our program how many rounds and how many search operations in each round. Then do a benchmark for the growing size of the array n. Also finding a polynomial that was near or roughly describes our execution time. For this report I wrote LaTeXusing overleaf because it was quite simple to get the ball rolling. Much of this introduction assignment was for the students to get acquainted using LaTeXwhile also doing some basic measurements.

Run time

For this task we were given some code:

```
    for (int i = 0; i < 10; i++) {
        long n0 = System.nanoTime();
        long n1 = System.nanoTime();
        System.out.println(" resolution " + (n1 - n0) +
            "nanoseconds");
```

The questions was if this clock within java was accurate enough to measure the run time of one operation. Upon executing this slab of code I received different execution times. 100ns(nanoseconds), 200 ns and 300 ns were the results. But then we noticed that the clock within java takes time to "look" at. Meaning that for example if we are measuring let say in milliseconds the time it takes you to read the clock on your wrist but when you are trying to move your wrist to your face it will add on time. It's the same here.

```
System.nanoTime();
```

Hence why during this small task we could measure more accurately by looping many times.

Interference

I noticed some issues, one of them was a cache problem. If we have a code or a given array that is sequential we run in to the problem that the memory will allocate for the next given integer. Take for example this piece of code:

```
int[] given = {1,2,3,4,5,6,7,8,9,0};
int sum = 0;
    for (int i = 0; i < 10; i++) {
        long t0 = System.nanoTime();
        sum += given[i];
        long t1 = System.nanoTime();
        System.out.println(" resolution " + (t1 - t0) + "
            nanoseconds");
}
```

When we access the memory for the first time, let's say we access the first element in our array 1. The memory will guesstimate that I will access the second element in our array "2" and take it from the memory the same time as the first access. This is what is called as spatial locality when working with memory.

This causes a problem because I want to figure the time it takes for only one memory access i.e one operation. The temporary solution to this is to have random. By that I mean that I want to access the array randomly, by doing so the memory will probably not allocate for the next integer.

Task 1

In this task we were supposed to fill our array with dummy values and do random accesses. Then collect the integers in a variable called "sum". After this we used larger and larger n to create the benchmark below. We ran every chosen n at least 20 times to get a close enough median. Sometimes the values on the run time would spike and give us unpredictable numbers. We thought that is due to some compilation/ windows failure or that something is running in the background and taking up space/time.

Discussion task 1

Before creating our benchmark for task 1 we did some some test to see how well or accurately the built in java clock was measuring run time. In those test the larger the n the greater the chance for us to get a value

n	time (ns)
10	0,7
100	0,6
1000	0,3
10 000	0,3

Table 1: A benchmark where I call the access function with a larger and larger n

of significance. This happened because the java clock took some time to actually run.

The solution to this problem was to measure the time it took to actually do the operation and not the loops. We created thus a dummy operation (doing essentially nothing) running simultaneous as our operation we wanted to measure. An accurate reading of the operation was conducted by taking the total time of the dummy operation was subtracted for the total time of the actual operation (look at the code below). Thus the uncertainty was reduced in the measurement.

```
long t_dummy = (System.nanoTime() - t0);
    return ((double)(t_access - t_dummy))/((double)k*(double)l);
```

Task 2

For this task we were supposed to choose two variables k m that gives us predictable results. We are given two loops where we fill up those arrays with random integers called "keys" and "array". For this task we also test for larger and larger n. Finding a polynomial also called big O notation/ time complexity for our run time. Then also do a benchmark for the results.

n	k	m	time (ms)
10	1000	1000	28
100	1000	1000	200
1000	1000	1000	1000
10000	1000	1000	9000

Table 2: Benchmark for task 2, where n becomes larger and larger

Discussion task 2

For this task we implemented a function because the given code was incomplete. Similar to task 1.

When choosing lower numbers for our m and k we got very unpredictable values. Instead we chose k and m to be constant at 1000, which gave us predictable values.

Looking at our table it is most likely that our polynomial is linear. Time complexity of O(n). We also graphed the values from the benchmark and the results were that the function was indeed linear. The approximated polynomial is:

$$y = 0.9n + 76x$$

Task 3

This task we are find duplicates in a array of size n. We use the same search algorithm but now only with one array. Same as before we want larger and larger n. The polynomial/the time complexity is also part of this final task. We also created a table below.

n	**k**	**time (ms)**
10	1000	3
100	1000	30
1000	1000	3000
10 000	1000	170 000

Table 3: Benchmark of operation finding duplicates in two arrays of n size, run time in milliseconds

Discussion task 3

The execution on this task was similar to Task 2. The difference between them was the two arrays were of the same size. No "break" was done, i.e I didn't break out the loop.

Both arrays change size with different n, we could already assume that the time complexity would be quadratic. Plotting of the points gave us the polynomial:

$$y = 0,002^2 + 1,5n71$$

. When graphing the function we can see that an hour of computing we can handle n around 470000.

Conclusion

After these benchmarks we conclude that it takes a while for a given operation to run. Some operations needs to be run a couple of time before you can get an accurate reading. Time complexity can not just be decided

on how the code looks, if it has nested loops etc. But only with accurate testing of the algorithm.