Arrays

Axel Månson Lokrantz

Spring Fall 2022

Introduction

In the 'Array' assignment the students were asked to measure the efficiency of operations on different arrays. The assignment was divided into three separate tasks where each of the tasks had a specific operation that had to be bench marked by the students.

Task 1.1: nanoTime

The students were given the following code:

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

With the help of the Java Run time Environment it is possible measure how long an operation takes. As seen in the above example, two primitive variables are declared n0 and n1. Each of them are given the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds. By subtracting n1 with n0, we can see the accuracy of the clock. Ten iterations of the loop yielded the following results:

iteration	ns
#1	273
#2	137
#3	87
#4	69
#5	87
#6	96
#7	85
#8	85
#9	86
#10	78

As the table suggest, even though there are no operations in between the two timings of the program, a time difference can be seen. It is extremely hard to explain why this happens but a simplified answer would be that the nanoseconds represent the time it takes for the program to respond to the question "what time is it?". Even though the time difference is in mere nanoseconds, any test measuring with a precision of nanoseconds would be useless considering our results. Another notably problem with the test is that it only iterates 10 times which range all the way from 273 to as low as 69 nanoseconds.

Task 1.2: Random Access

`Set up a benchmark where you call the access function with a larger and larger n. Present your conclusions in a nice table and pay attention to the number of significant figures that you use.`

To get a better understanding of the performance we measured the timings of accessing an array. In very large arrays accessing the array in sequential order could cause problems due to caching - therefor we had to create a benchmark where the access was random. The test was at first done using the following code.

```
long t0 = System.nanoTime();
for (int i = 0; i < n; i++){
    sum += array[rnd.nextInt(n)];
}
long t1 = System.nanoTime();
```

Which made it confusing, since generating random numbers within our test could give us misleading results. Therefor the test was rewritten and the random sequence was implemented before doing any array access.

```
for (int i = 0; i < n; i++){
            indx[i] = rnd.nextInt(n);
}
int[] array = new int [n];
for (int i = 0; i < n; i++){
    array[i] = 1;
}
long t0 = System.nanoTime();
for (int j = 0; j < k; j++){
    for (int i = 0; i < n; i++){
        sum += array[indx[i]];
    }
}
long t_access = (System.nanoTime() - t0);
```

n	ns
10	0.74
100	0.46
1000	0.36
10000	0.35
100000	0.37

From the different tests made with an increasing n it is obvious that the amount of data does not influence the computation time. From this observation we can also conclude that the program has an efficiency of O(1). To combat the built in errors with the clock the result was adjusted by decreasing t_access with a dummy add operation.

Task 2: Search

`You now have to choose k (number of rounds) and m (number of search operation in each round) to something that gives you predictable results. You then do the benchmark for a growing size of the array n and examine the result. Find a polynomial that roughly describes the execution time as a function of n.`

The example code had to be slightly rewritten since it originally gave out of bounds errors.

```
 int[] keys = new int[m];
        int[] array = new int[n];

        for (int j = 0; j < k; j++){
            for (int a = 0; a < m; a++){
```

```
        keys[a] = rnd.nextInt(n*10);
    }
    for (int b = 0; b < n; b++){
        array[b] = rnd.nextInt(n*10);
    }
```

We set our m value (length of keys array) to 50 and kept the loop count at 1.000.000. To ensure our results were correct, both the m and n values were adjusted to higher numbers but the results were unaffected. Through the test we can see that increasing n with 5 each iteration gives an almost linear increase in time. Plotting the numbers gave the following polynomial `y = 51,517x + 54,75.` Which shows that the program has an efficiency of O(n).

n	ns
5	284
10	544
15	830
20	1058
25	1453

Task 3: Duplication

The third task was very similar to the search task, but this time we were asked to set both lengths of the arrays to n and search for duplicates.

```
long t0 = System.nanoTime();
for (int ki = 0; ki < n; ki++){
    int key = keys[ki];
    for (int i = 0; i < n ; i++){
        if (array[i] == key){
            sum++;
        }
    }
}
t_total += (System.nanoTime() - t0);
```

Several tests were made, but increasing our n to higher numbers than 1280 took too long to be time efficient.

n	ns
80	2169
160	8660
320	27530
640	104205
1280	394474

Every time we doubled our n the time it took to compute the program was about four times longer. This finding makes a lot of sense, considering we are working with a nested for-loop (n*n) which gives us an algorithm with an efficiency of O(n^2). Plotting the numbers gave the following polynomial `y = 0,2282`x^2` + 16,366x - 242,54.` By calculating y = 1h (in nanoseconds) we found that in order for the computation to take one hour the array would need to be approximately 3.9 million elements long. It may or may not be accurate considering it took almost 7 minutes to compute an array with the length of 1280 elements.