Measuring time on array operations

Malte Berg

HT 2024

Introduction

This report covers the first introductory assignment assigned in the course ID1021. The assignment is to measure the time taken to perform three different operations on arrays, those being random access, searching for an item and searching for duplicates, as well as determining the time complexity of these operations.

Clock Accuracy

The time taken for the operations will be measured using the built in method `System.nanoTime()`. Accessing the method takes time, meaning the accuracy of the clock first has to be determined.

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

Running this code gives ten values for the amount of time taken to access the clock.

Run #	1	2	3	4	5	6	7	8	9	10
Runtime	163	222	180	153	193	125	179	121	150	153

Table 1: Runtime for 10 accesses to `System.nanoTime`, runtime in nanoseconds

As seen in Table 1, the runtime is quite uniform with a minimum runtime of 121 ns and a maximum runtime of 193 ns. This is the approximate time it takes to access the method and give back a time.

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

With the above code, 10 values for the time it takes to do a simple array access are printed.

Run #	1	2	3	4	5	6	7	8	9	10
Runtime	301	227	233	240	226	194	222	200	205	207

Table 2: Runtime for 10 different array accesses, runtime in nanoseconds

Comparing the runtime for an array access and just the clock, the array access is most of the time less than half the runtime of just the clock.

Random Access

The first main task of the assignment was to find how the runtime varies for a random array access on different sized arrays.

```
int[] sizes = {100, 200, 400, 800, 1600, 3200};
// JIT warmup
bench(1000,10000000);

int loop = 1000;
int k = 10;
for(int n : sizes) {
    long min = Long.MAX_VALUE;
    for (int i = 0; i < k; i++) {
        long t = bench(n, loop);
        if(t<min) min=t; }
    System.out.println(n + "\t" + (double)min/loop);
}
```

The above code runs a method `bench` that creates arrays of size n, fills them with numbers and returns the time it takes to access 1000 values randomly picked from the n sized array. As it accesses the array 1000 times, the printed value is divided by 1000. It also only prints the minimum

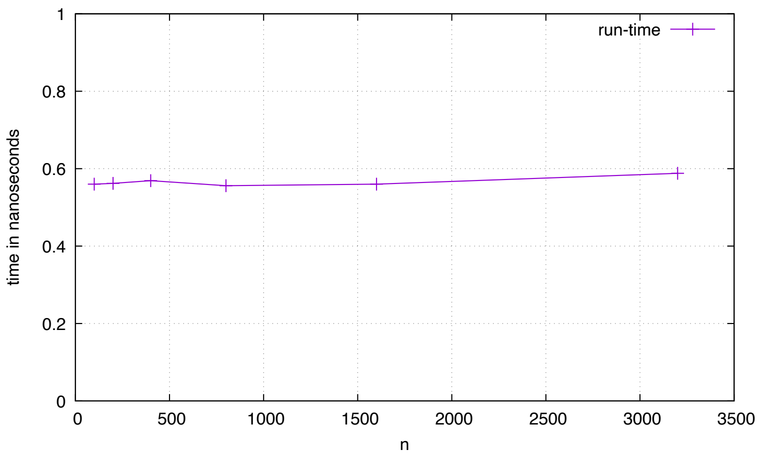

Figure 1: Runtime of random access operation in arrays of varying size

runtime, as the maximum runtime varies a lot and therefor also the average runtime, but the minimum stays almost constant through all runs.

As clearly shown in Figure 1, the runtime is consistent across different array sizes. This means that the time complexity must be $O(1)$.

Search For an Item

The next task in the assignment was to see the runtimes of a search algorithm on the arrays of different size.

```
int sum = 0;
long t0 = System.nanoTime();
for (int i = 0; i < loop; i++) {
    int key = keys[i];
    for (int j = 0; j < n; j++) {
        if (key == array[j]) {
            sum++;
            break; }
    }
}
long t1 = System.nanoTime();
return (t1 - t0);
```

Inside a method `search`, this is the search algorithm used, looking for a specific item in an array of size n.

Looking at Figure 2, except for the first point, a clear linear function is seen. The access time complexity for the search can then be concluded to be $O(n)$

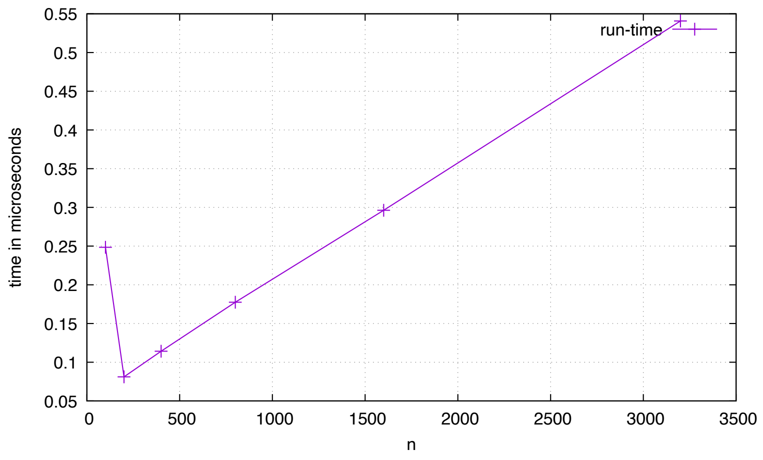

Figure 2: Runtime of search algorithm on arrays of varying size

Search for Duplicates

In the third and final task, the runtime for searching duplicates is explored where two arrays of size n are searched to find duplicated numbers. For every number in the first array, a number of the second array is searched for. This takes a long time and it will not be repeated with loop as in the previous two tasks.

```
long t0 = System.nanoTime();
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        int key = array_a[i];
        for (int j = 0; j < n; j++) {
            if (key == array_b[j]) {
                sum++;
                break; }
        }
    }
}
long t1 = System.nanoTime();
return t1 - t0;
```

This is the code used for finding the duplicates, quite similar to the regular search alogrithm but modified to search both arrays.

In Figure 3, a clear exponential expression is shown. As the size of the arrays double, the runtime is multiplied by four. This gives that the time complexity for the searching for duplicates algorithm must be $O(n^2)$. This comes due to there being a search in a search.

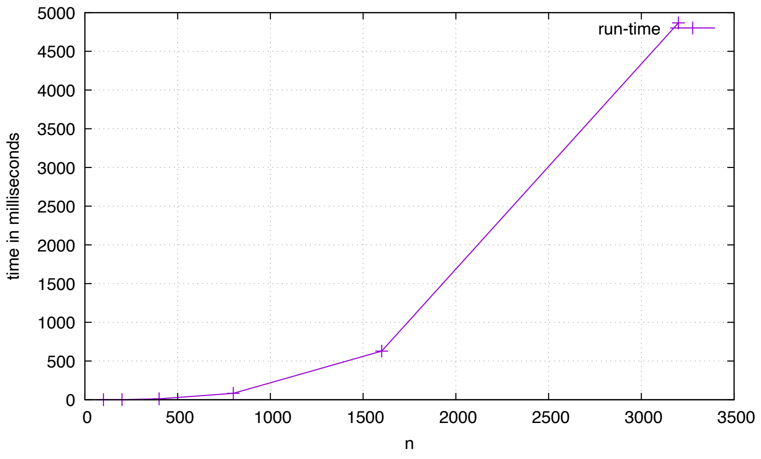

Figure 3: Runtime of searching for duplicates in arrays of varying size