Arrays

Sami Al Saati

Fall 2024

1 Random Access

1.1 The clock

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

i	Runtime
0	250
1	208
2	84
3	84
4	83
5	83
6	83
7	84
8	84
9	83

Table 1: the runtime for each iteration, measured in ns

The code works as intended since we're not accessing anything, the time is "constant".

1.2

```
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");
}
```

i	runtime
0	208
1	167
2	125
3	83
4	167
5	125
6	83
7	125
8	83
9	83

Table 2: the runtime for each iteration (i), measured in ns

The code works as intended and we see that the time complexity is of $\mathcal{O}(1)$ since we're accessing a specific element.

1.3 Random

```
int[] given = {0,1,2,3,4,5,6,7,8,9};
Random rnd = new Random();
int sum = 0;
long  t0 = System.nanoTime();
for (int i = 0; i < 1000; i++) {
    sum += given[rnd.nextInt(10)];
}
long t1 = System.nanoTime();
System.out.println(" one read operation in " + (t1 - t0)/1000 +"nanoseconds");
```

iteration	Runtime
1	242
2	207
3	203

Table 3: Runtime is measured in ns, results are taken from running the code 3 times

To give proper measurements we have to access the array at random. If the array gets accessed in order then it will give the Running this code

significantly increases the runtime from the previous code. This is because finding a random number takes time. It is not instant.

1.3.1

To simplify the process of finding a random number, we're now using two methods. In the method `bench` we create an array arr with n elements. Then we create an array indx, with loop amount of elements. Thereafter we call to `run` and then measure the time it takes for `run` to complete `Run` uses indx to randomly select an element in arr.

```
    public static long bench(int n, int loop)
    public static int run(int[] arr, int[] indx)
```

avg	min	max
10	9	11

Table 4: Setting loop,n = 1000 and calling bench 10 times gives the following avg, min and max values in ns

1.3.2

```
bench(n, 1000000);
for (int i = 0; i < k; i++){
    long t = bench(n, loop);
    if(t>max) max=t;
    if(t<min) min=t;

    total += t;
}
```

avg	min	max
9.86	4.33	14.3

Table 5: Caption

1.3.3

```
    int[] sizes = {100, 200, 400, 800, 1600, 3200};
```

Size	Runtime
100	9,4
200	2.1
400	2.2
800	2.1
1600	2.1
3200	2.1

Table 6: Minimum runtime for different array sizes, given in ns

2 Search for an item

n	Runtime
100	1.7
200	0.62
400	0.88
800	1.4
1600	2.5
3200	4.8

Table 7: Average runtime for different array sizes, given in μs

From table 7 we can see that the execution time increases with a linear fashion which indicates that we have: $\mathcal{O}$(n).

3 Search for duplicates

n	Runtime
100	7.14
200	2.04
400	7.56
800	28.7
1600	115.1
3200	440.7

Table 8: Average runtime for different array sizes, given in ms

From table 8 we can see that the execution time is $\mathcal{O}(n^2)$. From the table above we can see that the runtime increases by a multiple of about 4 each time we double the number om n. This is also the indicator for why the order is n^2, as $2^2 = 4$ which validates our assessment of the execution time polynomial.