Arrays and Performance in Java

Abdinasir Abdi

August 2024

1 Introduction

This report addresses assignment 1 in the course ID1021. The report goes over and explores how different operations on an array effects the efficiency and time complexity of the code. It goes over three different operations and for each explores how the size of an array impacts the execution time of the program. The three operations are the following:

- Random access: Writing to or reading from a random value at a random location in the array.
- Search: Looking for a specified item in an array.
- Duplicates: Finding common values in two arrays.

2 Clock

Before moving to testing we studied methods on how to accurately measure time. The following code was provided by the assignment:

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

The execution time for this code is not dependent on any variable and have therefore a constant time complexity ($O(1)$). However if we look at table 1, the results contradicts the expected outcome. Even after multiple tests the outcome remains the same, with small variations that are still not constant.

Iteration (n)	**Runtime**
1	0.3
2	0.2
3	0.2
4	0.1
5	0.2
6	0.1
7	0.3
8	0.1
9	0.1
10	0.1

Table 1: Run time in μs

The same happens for when we measure the performance for a single array search in a sorted array. On average it takes a little longer to run, but still a negligible difference. For more accurate measurements the access has to be totally random and not sorted. If sorted things like caching(read from the same or consecutive locations) would make the runtime faster. The main problem with this is that it's not measuring the right thing, it's measuring two thing. The time it takes to create a random number and the time it takes to access that number.

3 Random Access

```
  public static void int run(int[] array, int[] indx) {
  int sum = 0;
  for (int i = 0; i < indx.length() ; i++) {
  sum = sum + arr[indx[i]];
  }
  return sum;
}

  public static long bench(int n, int loop) {
  int[] array = new int[n];
  for (int i = 0; i < n; i++) array[i] = i;
  Random rnd = new Random();
  int[] indx = new int[loop];
  for (int i = 0; i < loop; i++) indx[i] = rnd.nextInt(n);
  int sum = 0;
  long t0 = System.nanoTime();
  run(array, indx);
  long t1 = System.nanoTime();
  return (t1 - t0);
}
```

With the following code, the loop now selects a random number from the indx array and then uses the index to access the target array. Now that we are measuring the runtime correctly and with better accuracy, how does the size of the array impact the runtime?

3.1 Task 1

The first task of the assignment was about generating random indices and then measure how long the runtime would be for a random access.

3.2 Benchmarks

```
public static void main(String[] arg) {
for (int i = 0; i < 10; i++) {
long t = bench(1000, 1000);
System.out.println(" access: " + t + " ns");
  }
}
```

We started out with following benchmark and realized from the results seen in table 2 that results vary quite a lot. Even thought the difference is nanoseconds apart, percentage wise the difference in performance is noteworthy. Sometimes, two to three times difference.

Access	**Runtime**
1	21
2	15
3	29
4	30
5	22
6	24
7	16
8	11
9	19
10	22

Table 2: Run time in μs

We then moved on to the core of task 1 and tested how the size of the array effects the performance of code. The size of the array doubled after every test and we initialized new variables to now also be able to measure the minimal,highest, and average. This way we could get a better understanding of the runtime and more accurately study the fluctuations. The results can be seen in table 3.

Size	**Min**	**Max**	**Average**
100	11.1	16.5	12.2
200	10.9	20.2	14.1
400	10.9	14.1	12.9
800	11.2	28.2	13.5
1600	10.8	18.2	11.4
3200	11.1	19.7	11.8

Table 3: The min, max, and average run time in μs

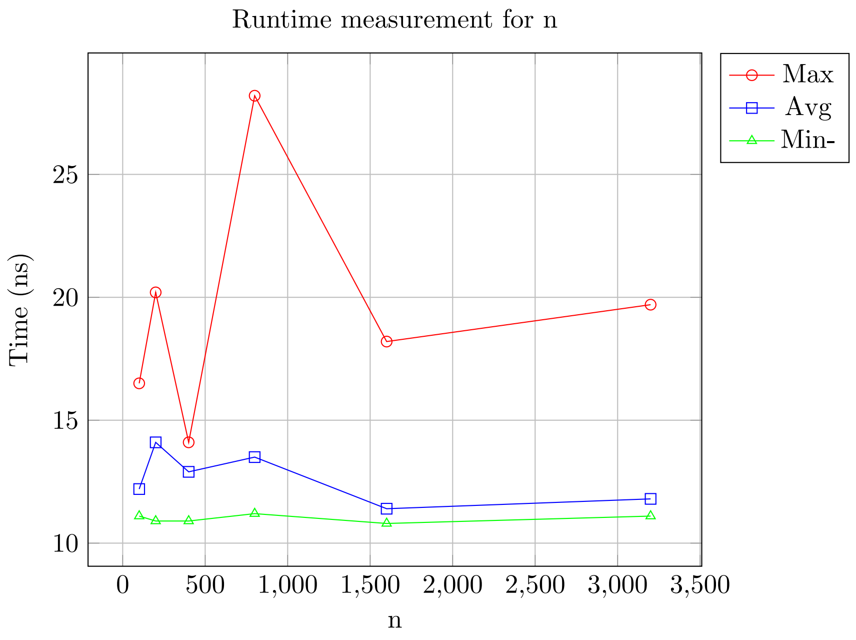

Figure 1: Graph over min, max, and average runtime for different n.

3.3 Conclusion

The minimal runtime seems to be constant independent of the size, but the maximal varies quite much even as the array gets bigger. The average seems to also follow the trend of being somewhat constant with smaller variation, and does not increase proportional to the size.

4 Search

4.1 Task 2

The second task was about measuring the time it would take to search up a random key in an unsorted array. As previous we also tested and compared if the size of the array affected the time.

n	loops	Time
10	10000	1.8
100	10000	2.8
1000	10000	6.2
10000	10000	27.4

Table 4: The runtime as n get larger, in ms

4.2 Conclusion

From what we could make out of the data the relationship between the size of the array (n) and amount of loops had a direct impact on the runtime. Depending if we increased or decreased the size and the loops, the runtime increased and decreased proportionally. In table 4 the results show how the time changes as we increase the size of the array. We chose to keep the amount of loops set, because the tests showed that even if we changed the loops the change in runtime would still be proportional as the size of the array grows. In worst case scenario the key is not found in the array and all loops have to be run, which gives it a linear time complexity of $O(n)$.

5 Duplicates

5.1 Task 3

The final task of the assignment is to search and find any duplicates in two arrays. We used the approach as in the task 2 and got the following results.

n	loops	Time
10	100	0.14
100	100	3.8
1000	100	25
10000	100	138

Table 5: The runtime as n gets larger, in ms

5.2 Conclusion

As presented in the in table 5, the runtime increases greatly as we increase the size for both arrays. We expected this outcome as there are now also conditional factors that have to be considered and that takes time, which makes each iteration longer. Looking at the results , finding duplicates in two arrays have the time complexity of $O(n^2)$.