Runtime Complexity of Array Operations

Dominik Gelland

Fall 2024

Introduction

Arrays are perhaps the most fundamental collection type in programming. Each element is accessed via an index which represents the offset in memory from the first array element. This report explores the runtime complexity of array operations using the Kotlin programming language. Kotlin runs on the Java Virtual Machine (JVM) and is therefore equivalent to Java in many ways. To mitigate some of the inconsistencies of benchmarking on the JVM, tests will be performed with large array sizes and a JIT warmup phase is.

Measuring time

The execution time of arbitrary code can be measured by recording timestamps before and after the code runs, followed by calculating the difference between these two timestamps. In Java and Kotlin `System.nanoTime()` can be used as a high resolution time source for this purpose. However, measuring the time difference for a single array access is unreliable due to the inconsistent resolution of the timer. This inconsistency is demonstrated by calculating the difference between two consecutive calls to `System.nanoTime()` without any other code in between:

```
val t0 = System.nanoTime()
val t1 = System.nanoTime()
println("resolution: ${t1 - t0} nanoseconds")
```

Running this a few times reveals that the apparent resolution of the timer is either `0 ns`, `100 ns` or rarely `200 ns`. This suggests that the timer will not be precise enough to measure the time difference of small operations such as a single array access. Instead, the time required to perform multiple iterations of each operation is recorded. Then, the average time per iteration is calculated by dividing the total elapsed time by the number of iterations.

Random access

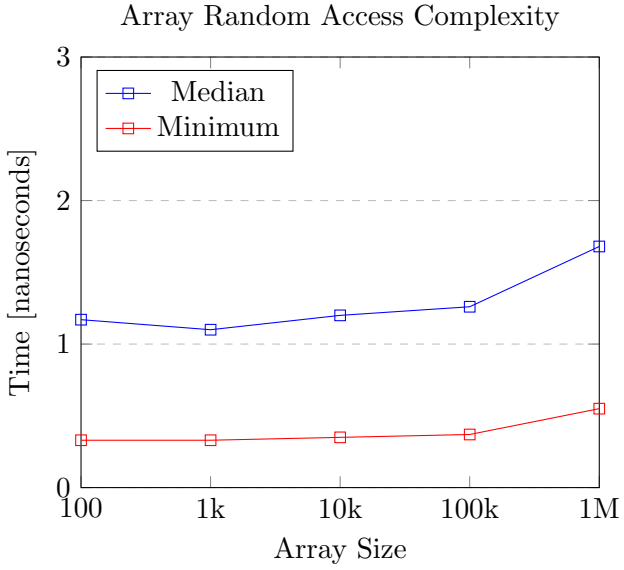
As accessing any array element via its index should be the same as accessing any data from memory at a specific address, we expect the performance of random access to be independent of the array's size, i.e., $O(1)$.

To verify this, we can measure the approximate time it takes to access elements at random indices using the following Kotlin function:

```
fun benchmarkAccess(n: Int, numAccesses: Int): Double {
    val array = IntArray(n)
    val randomIndices = IntArray(numAccesses) { Random.nextInt(n) }

    val t0 = System.nanoTime()
    for (i in 0..<numAccesses) {
        val index = randomIndices[i]
        array[index] = index
    }
    val t1 = System.nanoTime()
    return (t1 - t0).toDouble() / numAccesses
}
```

In each iteration of the benchmark, two arrays are created: `array` with size `arraySize`, this is the array that is being tested, and `randomIndices` which holds random indices to access the first array with. The function measures the total time required to perform `numAccesses` random access operations on `array` using the indices from `randomIndices`. The function was was run 10,000 times with 1000 accesses for each value of `n`. The minimum and median results are shown here:



The results show that for varying array sizes, the minimum and median access times remain similar, confirming that array access has a constant time complexity. Both the median and minimum lines have an overall upward trend however, this suggests that there is some additional overhead for larger arrays that could not be accounted for in this test.

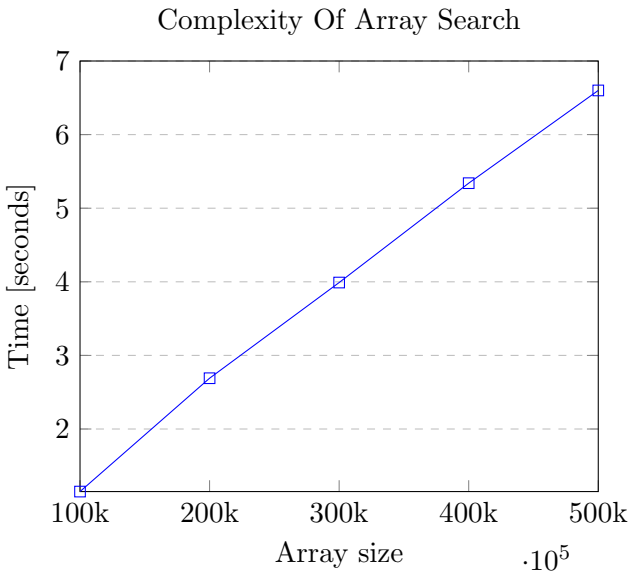
Search for an item

Searching for an item in an unsorted array involves going through all elements until an element is equal to the target, then stopping the search. This has a worst case time complexity of $O(n)$. The worst-case scenario occurs when when the target was not found, or the target was the last element in the search array, meaning every element had to be searched. To measure the complexity of this task we can use the following Kotlin function:

```
fun benchmarkSearch(n: Int, numSearches: Int): Long {
    val array = IntArray(n) { Random.nextInt(n * 2) }
    val keys = IntArray(numSearches) { Random.nextInt(n * 2) }

    val t0 = System.nanoTime()
    for (i in 0..<numSearches) {
        val key = keys[i]
        for (j in 0..<n) {
            if (key == array[j]) {
                break
            }
        }
    }
    val t1 = System.nanoTime()
    return (t1 - t0)
}
```

It creates two arrays, `array` of size `n` and `keys` of size `numSearches`. Both arrays are populated with random numbers between `0` and `n * 2` to simulate a realistic distribution of searches that are successful and unsuccessful. Then, the function returns the time it takes to perform `numSearches`. The graph below shows the total time to perform 10,000 searches for varying values of `n`:



Here the time to array size ratio follows a linear path which aligns with the initial theory that the complexity is $O(1)$.

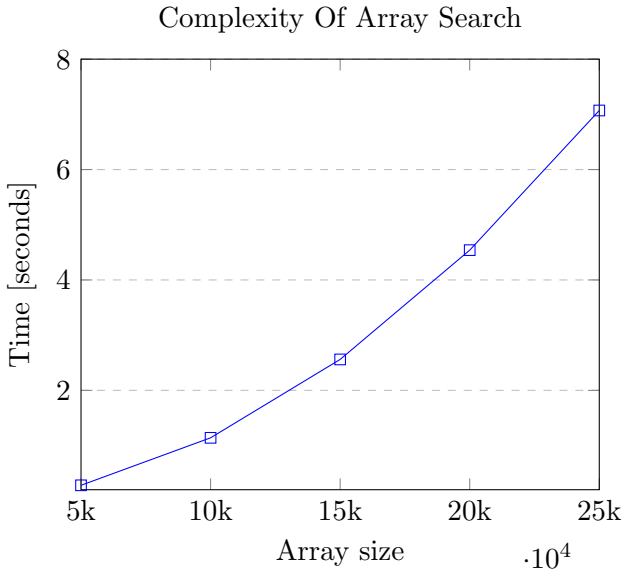
Search for duplicates

Finding duplicates across two unsorted arrays of equal length has an $O(n^2)$ runtime complexity. This is because, in the worst case, it necessitates checking every entry of the first array against every entry of the second array. This is demonstrated by the following Kotlin function:

```
fun benchmarkDuplicates(n: Int, num: Int): Long {
    val a = IntArray(n) { Random.nextInt(n * 2) }
    val b = IntArray(n) { Random.nextInt(n * 2) }

    val t0 = System.nanoTime()
    repeat(num) {
        for (i in 0..<n) {
            for (j in 0..<n) {
                if (a[i] == b[j]) {
                    break
                }
            }
        }
    }
    val t1 = System.nanoTime()
    return t1 - t0
}
```

The function creates two arrays of equal length and fill them with random numbers between `0` and `n * 2`. This range ensures a mix of values that will overlap and those that won't. Then, the time it takes to perform `num` duplicate searches returned. The graph below shows the benchmarked times for varying values of `n` with `num = 100`:



As expected, the time taken to perform the duplicates search grows quadratically with the array size. This confirms the expected $O(n^2)$ complexity of the operation.

Conclusion

The results of these tests underscore both the strengths and weaknesses arrays in various scenarios. For example, finding duplicates across two large arrays scales poorly. In such a case it's best to use another collection type, such as a `Set`, or to sort the arrays beforehand if the operation will be performed multiple times. However, it should be noted that, for small arrays or less critical sections of code, these poorly scaling operations may still be preferable to the potential memory and performance overhead or complexity of instantiating and using other collection types.