The Advantage of Sorted Data

William Frid

Fall 2023

Introduction

Arrays can be sorted and unsorted. Both states have their strengths and weaknesses: While the sorted array is quicker to search in, the unsorted one doesn't require the action of being sorted (which will be the focus of an upcoming report), which itself can require a lot of computations. When it comes to searching through an array, there are multiple different algorithms to choose from, but only when working with sorted ones as an unsorted array is a cluster of random data and is unpredictable.

This report focuses on the time advantage of using a sorted array alongside different search algorithms and exploring how and when each of these should be used. Comparisons will be made between benchmarks covering different scenarios to give a proper understanding of when to choose which algorithm.

Generating Arrays

To generate arrays to perform benchmarks on, two methods were created: The first one being `generateArray()` which generates sorted arrays with a given size with each next number being 1-10 greater in value than the previous one. This was done so that the benchmarks wouldn't get any "shortcuts" due to an uneven distribution of numbers upon search, and to add holes in the data so that the benchmarks take missing data into consideration. The main logic is seen below.

```java
int intToAdd = 0;
for (int i = 0; i < size; i++) {
    intToAdd += (int) (Math.random() * 10 + 1);
    arr[i] = intToAdd;
}
```

The second method created was called `shuffleArray()` and was used for shuffling a sorted array to get an unsorted one. Based on a for loop and swapping two elements at a time, it makes sure each and every element gets shuffled thus giving a randomly shuffled array (limited by `Math.random()`).

```
for (int i = 0; i < newArray.length; i++) {
    int tmp = newArray[i];
    int indexTwo = (int) (Math.random() * newArray.length);
    newArray[i] = newArray[indexTwo];
    newArray[indexTwo] = tmp;
}
```

This approach was chosen as it was a relatively simple way to achieve both sorted and unsorted arrays without having somewhat identical methods. It also allowed for running benchmarks on both sorted and unsorted arrays that contained the exact same data.

Searching In An Array

Unsorted Array - Linear Search

Searching through unsorted arrays is only possible by iterating each element and comparing until the correct data is, or isn't found. The best-case scenario would be finding the data instantly (located at index 0), and the worst case would be not finding the data. Benchmarks on this were run as well as the average time where it would search for an integer in the span of zero and the max value held in the array.

Size(k)	**Best(µs)**	**Avg.(µs)**	**Avg.(µs)/Size(k)**	**Worst (µs)**
10	0.10	1.63	0.16	1.80
20	0.10	3.21	0.16	3.60
30	0.10	4.53	0.15	5.40
40	0.10	6.07	0.15	7.20
50	0.10	7.51	0.15	9.00
60	0.10	8.82	0.15	9.70
70	0.10	10.52	0.15	12.50
80	0.10	11.94	0.15	14.30
90	0.10	13.22	0.15	16.10
100	0.10	14.75	0.15	17.90

Table 1: Benchmark results showing the best, average, and worst case of searching an unsorted array based on size.

The benchmark results in Table **??** prove the theory regarding a constant best-case result, no matter the array size. It's also visible that the average time based on array size remains constant (columns *Avg.(μs)/Size(k)*). To analyze the other columns, *Avg.(μs)* and *Worst (μs)*, a graph will be used:

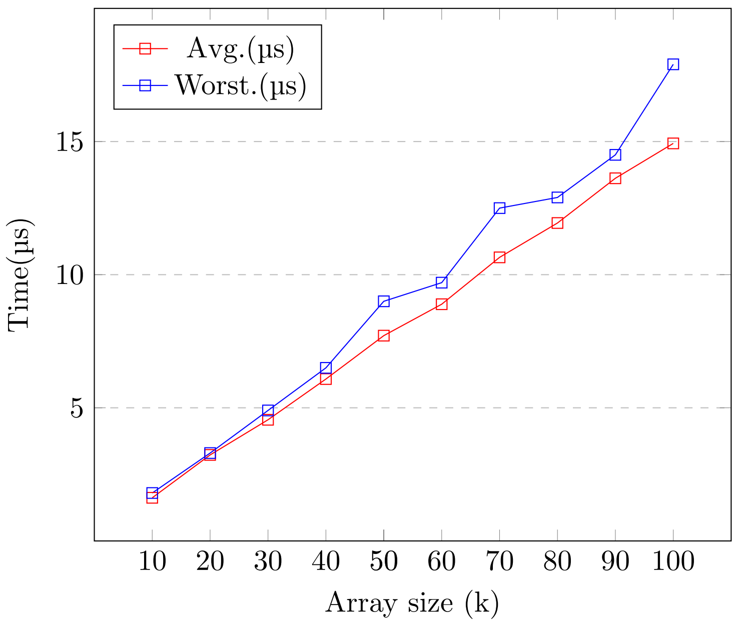

Figure 1: Average and worst time plotted to illustrate the linear time complexity.

Graph 1 proves that both average time and worst case increase linearly depending on the array size. Note how it's not very effective as soon as the array becomes too large but can still be useful when searching smaller amounts of data.

Sorted Array - Linear Search

To search through a sorted array using a linear search, one can modify the algorithm to stop early if it detects that the key is missing. It's based on the same principle as the one used for the unsorted array, but with the ability to stop if it detects that the key is missing (the key is what it's searching for). This can in the best case save $n - 1$ comparisons (where $n = size$), or simply just slow down the search due to extra logic if the element does exist. This algorithm can look something like the following snippet:

```
for (int i = 0; i < arr.length ; i++) {
    if (arr[i] == key) {
        return true;
    } else if (arr[i] > key) {
        return false;
    }
}
return false;
```

Size(k)	**Best(µs)**	**Avg.(µs)**	**Avg.(µs)/Size(k)**	**Worst (µs)**
10	0.10	0.04	0.0041	3.90
20	0.10	0.03	0.0017	6.90
30	0.10	0.03	0.0011	10.30
40	0.10	0.03	0.0008	15.30
50	0.10	0.03	0.0007	19.10
60	0.10	0.03	0.0006	20.60
70	0.10	0.03	0.0005	26.70
80	0.10	0.03	0.0004	27.40
90	0.10	0.03	0.0003	30.80
100	0.10	0.03	0.0003	34.30

Table 2: Benchmark results showing the best, average, and worst case of searching a sorted array based on size.

In Table 2 it shows that the best-case scenario is indeed constant. However, there's a drastic improvement in the columns Best(µs) and Avg.(µs) compared to searching in the unsorted array. This is because the algorithm in some cases searches for an element that doesn't exist since the key is based on a random index number in the range of zero and the max value held in the array. However since the same targeting technique was used in the unsorted array, this is still valid data.

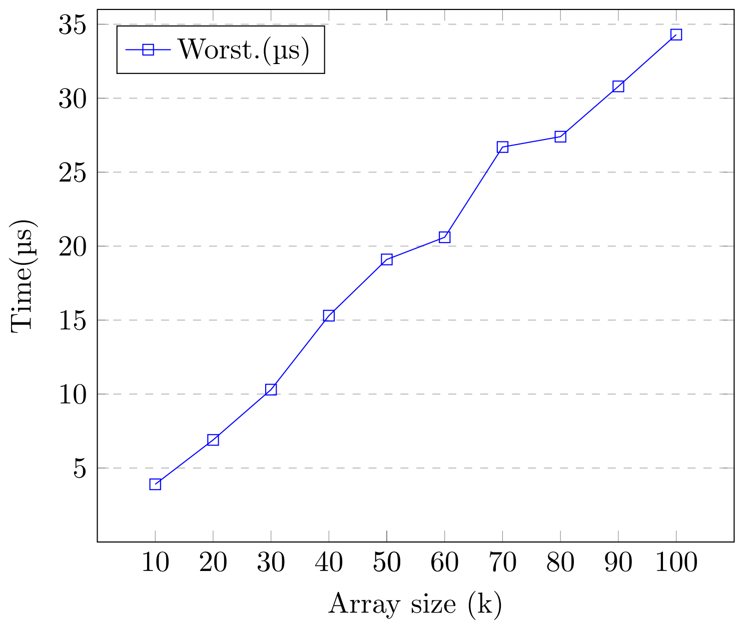

Figure 2: Average and worst time plotted to illustrate the linear time complexity.

Looking at Graph 2, it shows that the worst case remains linear but slower, while the average time is closer to a constant as seen in Table 2. Note that the average time wasn't plotted because it was too close to a constant number and because of that wouldn't show anything.

Sorted Array - Binary Search

Binary search is another more efficient algorithm for searching in a sorted array. Thanks to its time complexity being $O(log2(n))$, it only takes one more action to search for each doubling in the array size. Writing this algorithm in Java can be accomplished in code like so:

```
int min = 0;
int max = arr.length - 1;

    while (min <= max) {

        int mid = (min + max) / 2 ;

        if (arr[mid] == key) {
           return true;
        } else if (arr[mid] < key) {
                min = mid + 1;
                continue;
        } else if (arr[mid] > key) {
                max = mid - 1;
                continue;
        }

    }
    return false;
}
```

Size(k)	Best(ns)	Avg.(ns)	Avg.(ns)/Size(k)	Worst (ns)
1	1.30	2.80	2.80	13.30
2	1.30	2.70	1.35	15.10
3	1.10	5.10	1.70	19.10
4	1.00	1.60	0.40	17.80
5	0.90	8.40	1.68	17.70
6	0.90	4.30	0.72	17.80
7	0.90	8.00	1.14	17.80
8	0.90	2.90	0.36	17.70
9	0.90	5.90	0.66	19.20
10	0.90	6.00	0.60	19.20

Table 3: Benchmark results showing the best, average, and worst case of "binary searching" a sorted array using based on size.

Notice in Table 3 how the data is somewhat inconsistent. This is because of the difficulty of measuring the speed of the binary search alongside the OS itself, hardware, and background optimization. The important conclusion to draw from the table is that the binary search's time constant is logarithmic and a more effective algorithm than a linear one.

Find Commons In Arrays

When searching for elements two arrays have in common, you can use the linear algorithm for unsorted/sorted arrays, the binary search, and finally, an algorithm made for just this purpose. This algorithm works with two sorted arrays and takes steps to find the common elements.

```
int indexOne = 0, indexTwo = 0;
while (indexOne < arrOne.length && indexTwo < arrTwo.length) {
    if (arrOne[indexOne] > arrTwo[indexTwo]) {
        indexTwo++;
        continue;
    } else if (arrOne[indexOne] < arrTwo[indexTwo]) {
        indexOne++;
        continue;
    } else if (arrOne[indexOne] == arrTwo[indexTwo]) {
        // Match found.
        indexOne++;
    indexTwo++;
        continue;
    }
}
```

Since the time complexity for this algorithm is $O(n)$, it performs better than binary search as that algorithm would have a time complexity of $(O(log2(n) * n))$ in this case.

Size()	**Unsorted(lin.)**	**Sorted(bin.)**	**Sorted(opt.)**	**Bin./Opt.**
100	3.00	2.80	1.20	0.43
200	13.80	6.40	1.60	0.25
300	20.70	8.50	2.10	0.25
400	35.00	11.90	2.90	0.24
500	60.00	15.80	3.90	0.25
600	85.40	20.20	4.80	0.24
700	115.80	24.40	5.60	0.23
800	145.70	28.50	6.30	0.22
900	179.50	32.50	7.20	0.22
1000	232.60	34.30	8.00	0.23

Table 4: Benchmark results comparing the difference in speed between algorithms used for comparing arrays.

Table 4 proves the theory that the customized algorithm, made for comparing arrays, outperforms both the linear, and binary search algorithms. The column Bin./Opt. shows the ratio between these two top-performing algorithms, and that the optimized algorithm becomes an even better choice when dealing with larger arrays.

Discussion

The benchmarks performed for this lab proved hard to perform due to underlying reasons such as background optimization and OS. To avoid this, running the benchmarks on a Linux system could prove to give more precise benchmark results since the Java method `System.nanoTime()` (that being used) acts in a higher resolution in Linux than in Windows. More work for cleared linear graphs would also be preferred to visualize the differences in the algorithms more clearly.

Worth mentioning is that the benchmark results are highly affected by the generation of arrays. Since the method `generateArray()` skips numbers to create gaps, this affects the amount of advantage the different algorithms get. Let's say that the array only contains small gaps, then an algorithm that might stop early while searching a sorted array won't stop beforehand as often as it would if the arrays had larger gaps (only goes for linear search).

Conclusion

It's essential to choose the proper algorithm for both the task and data size. This becomes more important when the algorithms are used repeatedly. One option could be to first check the data size and wherever it's sorted, then choose the proper algorithm based on these criteria. This can be done by performing benchmarks and/or calculations on time complexity.