2. Sorted arrays

Kristiāns Vinters

Fall 2023

Introduction

I solved the sorted array search assignment in Go. I used Go because I want to become more familiar with it. Source code and benchmark data is available on GitHub[1].

Implementation

I implemented all of the search and duplicate functions using generic typing with the constraint `cmp.Ordered`, which constrains the type to anything that implements the `==`, `!=`, `<`, `>`, `>=`, `<=` operators. Frankly, there is no particular reason to do so for this assignment other than to get more familiar with Go's generics.

There weren't any particular difficulties in implementing the search and duplicate functions in Go. The implementations for the unordered, ordered, and binary search closely follow the given example Java implementations.

```
func NaiveSearch[T cmp.Ordered](arr []T, target T) int {
    for i := 0; i < len(arr); i++ {
        if arr[i] == target {
            return i
        }
    }

    return -1
}
```

Figure 1: Naive search implementation

[1] https://github.com/Phanty133/id1021/tree/master/2-sorted

```go
func NaiveSortedSearch[T cmp.Ordered](arr []T, target T) int {
    for i := 0; i < len(arr); i++ {
        if arr[i] > target {
            return -1
        }

        if arr[i] == target {
            return i
        }
    }

    return -1
}
```

Figure 2: Naive sorted search implementation

```go
func BinarySearch[T cmp.Ordered](arr []T, target T) int {
    first := 0
    last := len(arr) - 1

    for {
        idx := (first + last) / 2
        val := arr[idx]

        if val == target {
            return idx
        }

        if val < target && idx < last {
            first = idx + 1
        } else if val > target && idx > first {
            last = idx - 1
        } else {
            return -1
        }
    }
}
```

Figure 3: Binary search implementation

The duplicate finding functions were also straightforward to implement and two of the three utilized the search functions. The `DuplicatesNaive`

and `DuplicatesBinary` were implemented almost exactly the same apart from using different search functions.

```
// or DuplicatesBinary[T cmp.Ordered](a []T, b []T) []T
func DuplicatesNaive[T cmp.Ordered](a []T, b []T) []T {
    res := make([]T, 0, len(a)+len(b))

    for _, v := range a {
        // or if BinarySearch(b, v) != -1 {
        if NaiveSearch(b, v) != -1 {
            res = append(res, v)
        }
    }

    return res
}
```

Figure 4: Naive and binary duplicate finding implementation

The final duplicates function, `SortedDuplicatesSmart`, didn't use any of the search functions, but rather iterated over both array's indices simultaneously.

```go
func SortedDuplicatesSmart[T cmp.Ordered](a []T, b []T) []T {
    res := make([]T, 0, len(a)+len(b))

    i := 0
    j := 0

    for i < len(a) && j < len(b) {
        if a[i] > b[j] {
            j++
        } else if a[i] < b[j] {
            i++
        } else {
            res = append(res, a[i])
            i++
            j++
        }
    }

    return res
}
```

Figure 5: Smart duplicate finding implementation

Benchmarking

To benchmark the functions, I rewrote the given example benchmarking function given in `Bench.java` to Go as the `bench` package. The Go implementation improves upon the example by making the benchmarkable function as a function argument and also determining median and mean times, in addition to min and max. The benchmark function implementations are given in appendix A and B. All benchmarks were run on a Ryzen 5 5600H CPU with 16GB of RAM.

I evaluated the functions for array sizes $N \in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 10000, 100000, 1000000\}$. Each function had 1000 runs for each N. In addition, the search functions were executed on 10000 keys in every run. An exception was made for sizes $N \in \{100000, 1000000\}$, where the search functions had 250 runs and the duplicate functions had 5 runs due to the long execution times of the naive search.

Search

As expected, the naive search implementation performed the worst. The naive search was an order of magnitude slower than sorted, while binary search was the fastest, being another order of magnitude faster. The results for searching 1M elements are given in Fig. 6.

Function	Min, ms	Max, ms	Median, ms	Mean, ms
`NaiveSearch`	252	377	276	282
`NaiveSortedSearch`	25.5	33.1	25.8	26.0
`BinarySearch`	0.72	0.76	0.73	0.73

Figure 6: Results for searching 1M elements

However, when the number of elements to search is small, sorted search is faster than binary search (Fig. 7). This is because binary search has a higher constant factor due to the extra arithmetic operations and the extra conditional branches. Naive search is $O(N)$ against the number of elements to search, while binary search is $O(\log_2 N)$ (Fig. 8). Using the trendline from fig. 8, I estimated the run time for $N = 6.4\text{e}{+7}$:

$$t = 48.716 \log_2 N - 254.512$$
$$t = 48.716 \log_2 \left(6.4 \cdot 10^7\right) - 254.512$$
$$t \approx 48.716 \cdot 25.932 - 254.512$$
$$t \approx 1008.8 \mu s$$

The actual median run time was measured to be $1300 \mu s$, which is somewhat close to the estimated time.

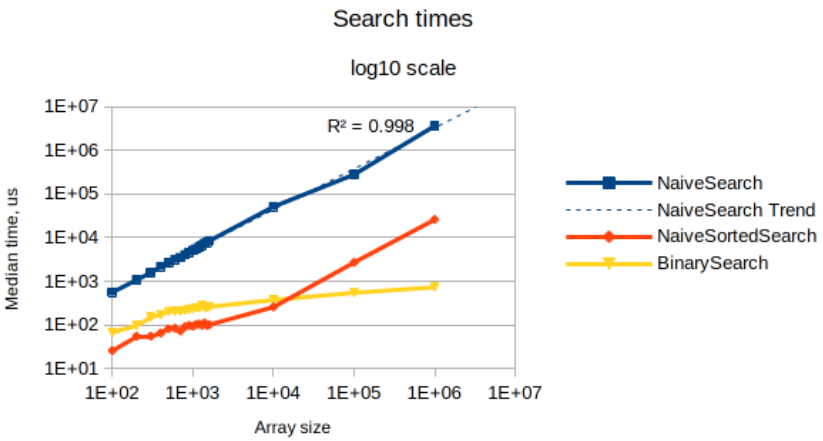

Figure 7: Median time vs Array size for all implementations

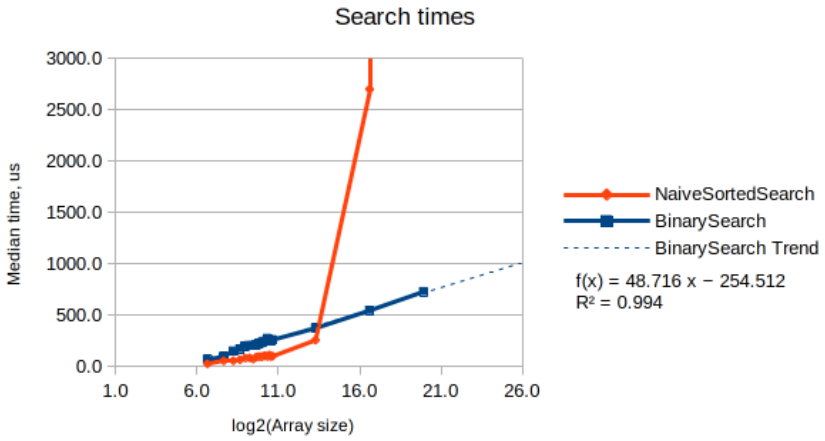

Figure 8: Median time vs $\log_2$ (Array size) for the sorted naive and binary search implementations

Finding duplicates

Unlike in search, the naive duplicate algorithm was four orders of magnitude slower than with the binary search algorithm, while the index-based duplicate algorithm was another order of magnitude faster. The results for finding duplicates in 1M elements are given in Fig. 9.

Function	Min, ms	Max, ms	Median, ms	Mean, ms
`DuplicatesNaive`	273000	351000	340000	315000
`SortedDuplicatesBinary`	44.5	50.3	44.8	46.3
`SortedDuplicatesSmart`	6.77	7.34	7.20	7.11

Figure 9: Results for finding duplicates in 1M elements

When running the duplicate finding functions across all N (Fig. 10), it appears that the binary search-based algorithm is actually faster for small N compared to the index-based algorithm. The naive implementation is still slower than one or the other algorithm for all N. The binary search algorithm is faster for small N compared to the index-based algorithm because of a smaller constant factor. The index-based algorithm iterates over both arrays completely, while the binary search algorithm only iterates over the first array. At high N, the index-based algorithm is faster because it doesn't have to perform any searches.

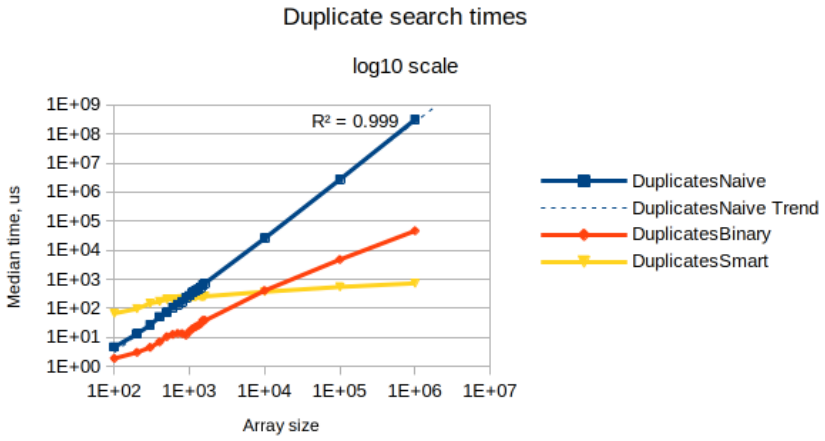

Figure 10: Median time vs Array size for all duplicate finding implementations

Appendices

A Search benchmark implementation

```
func BenchSearch(search func([]int, int) int) []SizeTime {
    sizes := []int{
        100, 200, 300, 400, 500,
        600, 700, 800, 900, 1000,
        1100, 1200, 1300, 1400,
        1500, 1600, 10000,
        100000, 1000000,
    }
    times := make([]SizeTime, 0, len(sizes))

    for _, size := range sizes {
        loop := 10000
        sortedArr := CreatedSortedIntArray(size, 1000)
        indx := keys(loop, size)

        fmt.Printf("Size: %d\n", size)

        iters := 250
        minTime := math.Inf(1)
        maxTime := math.Inf(-1)
        iterTimes := make([]float64, iters)
```

```
        for i := 0; i < iters; i++ {
            start := time.Now()

            for _, i := range indx {
                search(sortedArr, i)
            }

            elapsed := time.Since(start).Nanoseconds()
            minTime = math.Min(minTime, float64(elapsed))
            maxTime = math.Max(maxTime, float64(elapsed))
            iterTimes[i] = float64(elapsed)

            if i%100 == 0 {
                fmt.Printf(
                    "Iteration %d. Last elapsed time: %fus\n",
                    i,
                    float64(elapsed)/1000
                )
            }
        }

        times = append(times, SizeTime{
            size,
            minTime / 1000,
            maxTime / 1000,
            median(iterTimes) / 1000,
            mean(iterTimes) / 1000,
        })
    }

    return times
}
```

B Duplicates benchmark implementation

```
func BenchDuplicates(findDuplicates func([]int, []int) []int) []SizeTime {
    sizes := []int{
        100, 200, 300, 400, 500,
        600, 700, 800, 900, 1000,
        1100, 1200, 1300, 1400,
        1500, 1600, 10000,
        100000, 1000000,
```

```
    }
    times := make([]SizeTime, 0, len(sizes))

    for _, size := range sizes {
        sortedArrA := CreatedSortedIntArray(size, 1000)
        sortedArrB := CreatedSortedIntArray(size, 1000)

        fmt.Printf("Size: %d\n", size)

        iters := 1000
        minTime := math.Inf(1)
        maxTime := math.Inf(-1)
        iterTimes := make([]float64, iters)

        for i := 0; i < iters; i++ {
            start := time.Now()

            findDuplicates(sortedArrA, sortedArrB)

            elapsed := time.Since(start).Nanoseconds()
            minTime = math.Min(minTime, float64(elapsed))
            maxTime = math.Max(maxTime, float64(elapsed))
            iterTimes[i] = float64(elapsed)

            if i%100 == 0 {
                fmt.Printf(
                    "Iteration %d. Last elapsed time: %fus\n",
                    i,
                    float64(elapsed)/1000
                )
            }
        }

        times = append(times, SizeTime{
            size,
            minTime / 1000,
            maxTime / 1000,
            median(iterTimes) / 1000,
            mean(iterTimes) / 1000,
        })
    }

    return times
}
```