3. Sorting

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[*].

Implementation

There weren't any particular difficulties in implementing the search algorithms in Go. The implementations are very similar to the given Java examples. The only difficulty I faced was when I forgot to make a copy of the pre-sorted array, as the arrays are sorted in-place. This oversight created nonsensical results, which showed insertion sort being significantly faster than merge sort. The reason for this is that insertion sort has less overhead and is significantly faster on already sorted arrays than merge sort.

Benchmarking

I benchmarked the sorting algorithms by running them 10 times on 100 different, randomly generated integer arrays for array sizes 100, 1000, 2500, 5000, 10000, 25000, 50000, 75000[†], 100000[†], 250000[†]. The benchmark function implementation is given in appendix A. All benchmarks were run on a Ryzen 5 5600H CPU with 16GB of RAM.

The benchmark results show that, generally, selection sort is slower than insertion sort, which in turn is slower than merge sort.

[*] https://github.com/Phanty133/id1021/tree/master/3-sorting

[†] Only insertion, merge, and merge2 sort were tested. The tests were done on 10 different arrays with 10 times.

Size	$t_{\text{Selection}}$, ms	$t_{\text{Insertion}}$, ms	t_{Merge}, ms	$t_{\text{Merge 2}}$, ms
100	0.006	0.003	0.003	0.003
1000	0.470	0.130	0.046	0.040
5000	9.60	3.11	0.293	0.257
50000	930	330	3.65	3.29
100000	N/A	1320	7.78	5.11
250000	N/A	8330	20.4	18.4

Figure 1: Subset of median times for sorting algorithms. Full results available on the GitHub repository.

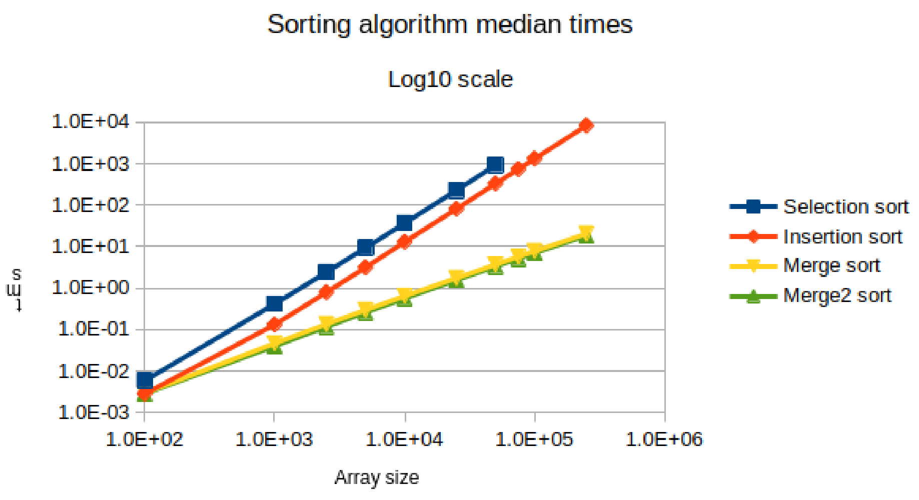

Figure 2: Benchmark results for sorting algorithms.

As expected, as the array size increases, both merge sort implementation median execution times grow slower than for insertion or selection sort. Using the selection sort times as a reference, we can see that insertion sort has a consistent ratio vs selection sort, while the ratio for merge sort significantly increases over array size. This is because selection and insertion sort is $O(n^2)$, while merge sort is $O(n \log n)$ on average.

Size	Insertion	Merge	Merge 2
100	2.1	2.0	2.1
1000	3.1	8.9	10.4
5000	3.0	31.7	36.2
50000	2.8	251.7	279.9

Figure 3: Ratio of median times for sorting algorithms vs selection sort $\frac{t_{\text{Selection}}}{t_{\{\text{Insertion,Merge,Merge 2}\}}}$

The following figures (Fig. 4 and 5) confirm the $O(n^2)$ and $O(n \log n)$ time complexities for selection, insertion and merge sort, respectively.

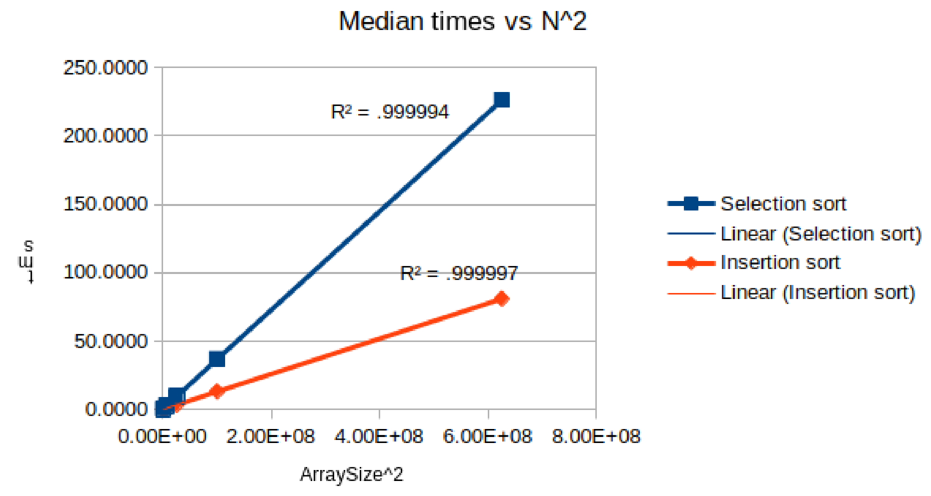

Figure 4: Selection and insertion sort median times vs ArraySize^2

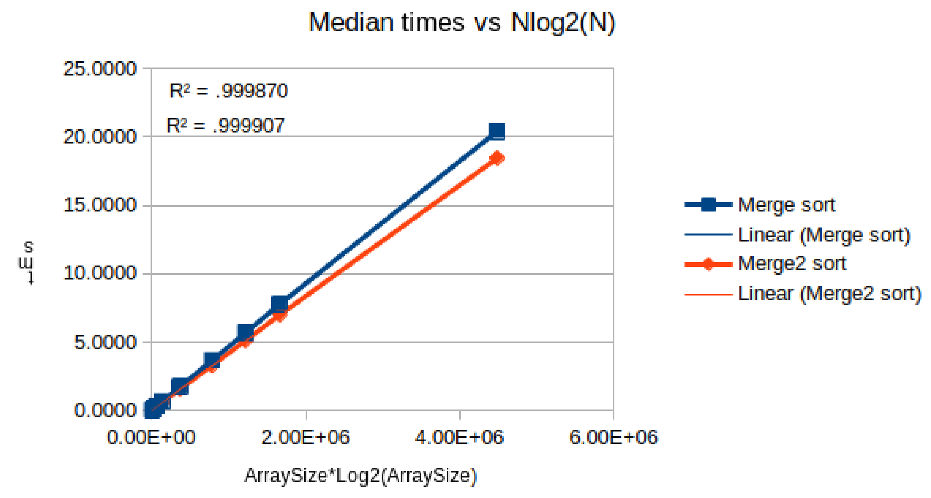

Figure 5: Merge sort median times vs ArraySize $\cdot \log_2$ (ArraySize)

The benchmarks also show a slight difference between the two merge sort implementations. Both implementations are still $O(n \log n)$, but the second implementation has slightly lower constant factors due to it doing less copying. The performance improvement is around 12% overall, but it appears to reduce to around 11% as the array size increases.

Size	Ratio
100	1.05
1000	1.16
5000	1.14
50000	1.11
100000	1.11
250000	1.11

Figure 6: Ratio $\frac{t_{\text{Merge}}}{t_{\text{Merge 2}}}$ for median times

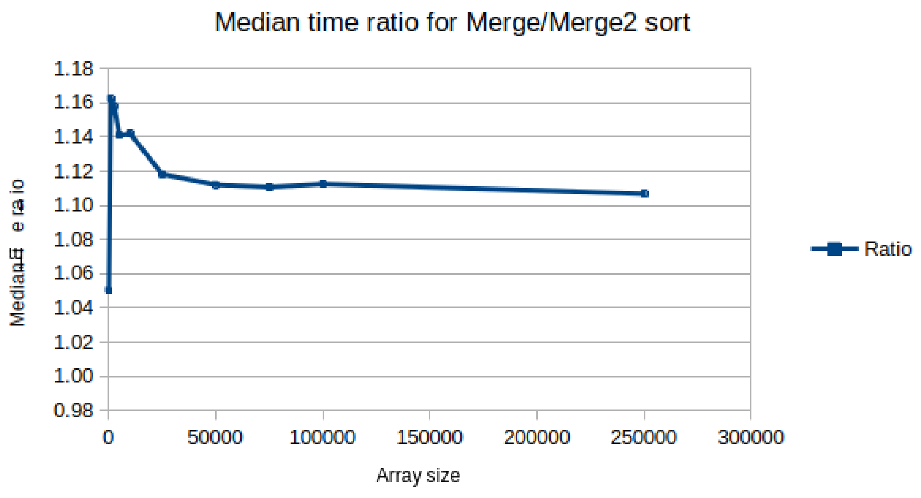

Figure 7: Ratio $\frac{t_{\text{Merge}}}{t_{\text{Merge 2}}}$ for median times

Appendices

A Benchmark

```go
func BenchSort(
    sort func([]int),
    benchArrays [][]int,
    n, max, repeats int
) [][]float64 {
    times := make([][]float64, len(benchArrays))

    for i, arr := range benchArrays {
        times[i] = make([]float64, repeats)

        for r := 0; r < repeats; r++ {
            arrCopy := make([]int, len(arr))
            copy(arrCopy, arr)

            start := time.Now()
            sort(arrCopy)
            elapsed := time.Since(start)

            times[i][r] = float64(elapsed.Nanoseconds()) / 1000 / 1000
        }
    }

    return times
}
```