7. Quicksort

Kristiāns Vinters

Fall 2023

Introduction

I solved the 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 quicksort algorithms in Go. For the linked list quicksort, I reused the linked list implementation from a previous assignment. The implementations of the quicksort algorithms in Go look very similar to implementations in any other C-like language, with the exception of some Go-specific syntactic sugar, for example, being able to swap variable values in a single line.

```
func partitionLinkedList(
    min, max *llist.LinkedListItem[int]
) *llist.LinkedListItem[int] {
    // ...
    // Single-line swap example:
    min.Head, max.Head = max.Head, min.Head
    // ...
}
```

The quicksort implementations for both data types are very similar except for the partition functions. The array partition function has more cases to handle, whereas the linked list function was simpler to implement.

```
func partitionArray(array []int, min, max int) int {
    pivot := array[min]
    i := min
    j := max
```

*https://github.com/Phanty133/id1021/tree/master/7-quicksort

```
    for i < j {
        for array[i] <= pivot && i < max {
            i++
        }

        for array[j] > pivot && j > min {
            j--
        }

        if i < j {
            array[i], array[j] = array[j], array[i]
        }
    }

    array[min], array[j] = array[j], array[min]

    return j
}

func partitionLinkedList(
    min, max *llist.LinkedListItem[int]
) *llist.LinkedListItem[int] {
        pivot := min
        i := min

        for i != max && i != nil {
                if i.Head < max.Head {
                        pivot = min
                        i.Head, min.Head = min.Head, i.Head
                        min = min.Next()
                }

                i = i.Next()
        }

        min.Head, max.Head = max.Head, min.Head
        return pivot
}
```

Benchmarking

I benchmarked the sorting algorithms by running them 500 times on different, randomly generated integer arrays for array sizes 10, 100, 1000, 5000,

10000, 50000, 100000, 500000, 1000000. Both implementations were run on the same arrays. I also benchmarked a linked list with a pointer to the last element, but the performance was near identical to the standard linked list, as I access the last element only once in my implementation. For analyzing run times, I measured the raw execution time for every repeat in nanoseconds, which I stored in an array and then wrote to a `.csv` file. I used LibreOffice Calc to calculate the mean, median, min, and max times and plot the graphs.

As it turned out, the run times for both the array and linked list implementations were near identical, except that the linked list started to become marginally slower with larger datasets (fig. 3).

Size	t_{Array}, μs	t_{LL}	$\frac{t_{\text{LL}}}{t_{\text{Array}}}$
10	1.40	1.40	1.00
100	3.77	3.77	1.00
1000	35.8	39.1	1.09
5000	214	239	1.12
10000	461	519	1.13
50000	2670	3060	1.15
100000	5570	6460	1.16
500000	30700	36800	1.20
1000000	64800	78400	1.21

Figure 1: Median times for array vs linked list quicksort

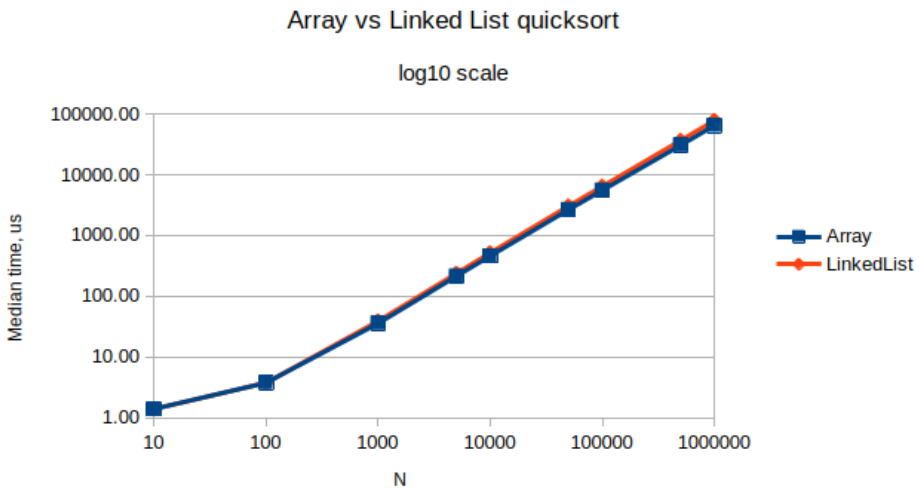

Figure 2: Median quick sort times

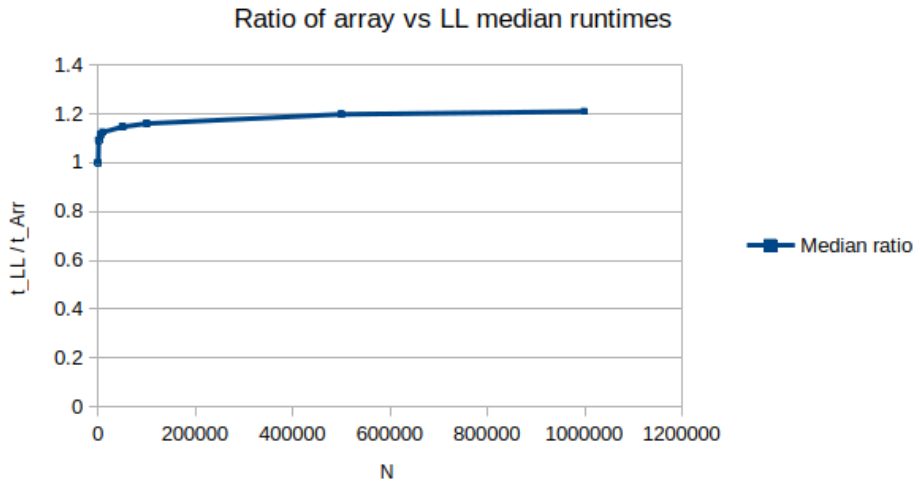

Figure 3: Median time ratios vs array size

The expected complexity of quicksort for both the array and linked list (with `last` pointer) implementation is expected to be $O(n \log_2 n)$. To determine whether my implementations had the same complexity, I plotted the median times against Array size $*$ $\log_2$ (Array size) and determined the coefficient of determination R^2. R^2 was 1.000 for both (fig. 4).

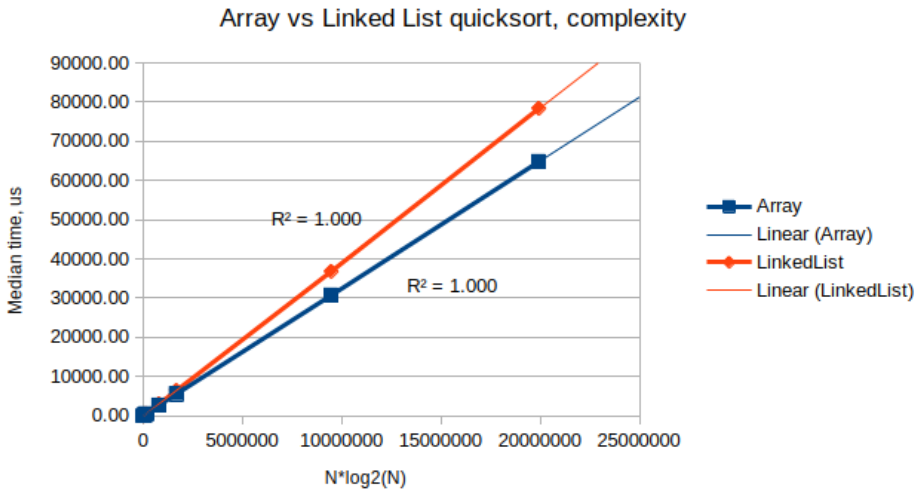

Figure 4: Implementation time complexity analysis