605.202: Data Structures

Peter Rasmussen

Lab 4 Analysis

Due Date: April 27, 2021

Dated Turned In: April 27, 2021

Lab 3 Analysis

This analysis examines the 1) use of the five sorting algorithms across four different pre-sorts (i.e., in-order, reverse-ordered, randomly-ordered, and randomly-ordered with 20% duplicates) and 2) code used to do so. I justify the use of recursion to implement each sorting algorithm on the basis that four of the algorithms – that is, the straight merges and natural merge – are more naturally recursive because of the inherently recursive nature of partitioning. If I had taken an iterative approach, I think the heap sort would have been easier to implement as it is more commonly done iteratively. I compare each sort and its performance to its counterpart and explain the basis for differences in performance.

I organized my program as a Python package comprised of modules and submodules, and grouped submodules by theme (e.g., sort submodules were organized under the sorts module). The program also generated complexity statistics, including as a consolidated CSV file, which I used to analyze the program's time and space efficiency.

I analyzed the complexity of each sorting algorithm. Based on information in the lecture notes and other research[1], the worst-case complexity of each method is below:

- Two-way merge: n * log2n
- Three-way merge: n* log3n
- Four-way merge: n * log4n
- Natural merge: Best case O(n) and worst case O(n2)
- Heap sort: n * log2n

The spacetime complexities of each sort are below:

- Two-, three-, four-way and natural merge: O(N)
- Heap sort O(1)

I learned that I should have asked about which Python primitives were available, as I spent a considerable amount of time developing my own list class that was ultimately not necessary.

The rest of this analysis covers the following:

- Justification of the design of the program.
- Description and justification of recursion.
- Consider whether iteration would have been better.
- Discussion of the time and space efficiency of the program.
- Summarization of lessons learned and what I could have done differently.

<u>Justification of the Design of the Program.</u>

Lab 4 organizes Python files into submodules. This approach yielded a more coherent, organized construction of the program. In addition, the design of the program supports up to 26 variables (a-z) and allows the user to efficiently organize, retrieve, and combine polynomial terms because of its use of node data dictionaries.

- Four-way merge complements two-way and three-way merge sorts and heap sort

[1] (link, link, link)

- CSV outputs organized such that, for each input dataset, performance metrics are tabulated at the top of the CSV and beneath that is a side-by-side comparison of the echoed input and the sorted outputs.
- Tested on file sizes of 10,000 integers.
- Option to process one file at a time or in bulk.
- Recursive implementation of each sort algorithm to enable apples-to-apples performance comparisons.
- Using the 25 input files and five sorts, the user can execute 125 runs in one go.

Description and justification of recursion.

The partitioning algorithm of the straight merges is inherently recursive, as the code below shows.

```
def partition(self, unsorted_li, partitioned_li: Union[list, None] = None) -> list:
    """
    Recursively partition a list.
    :param unsorted_li: List to be partitioned
    :param partitioned_li: List to add partitions to
    :return: Fully-partitioned list
    """
    if partitioned_li is None:
        partitioned_li = []
    if len(unsorted_li) == 0:
        return partitioned_li

    if len(unsorted_li) == 1:
        return partitioned_li + [unsorted_li]
    else:
        partitioned_li.append([unsorted_li.pop(0)])
        self.n_partition_calls += 1
        return self.partition(unsorted_li, partitioned_li)
```

Consider whether iteration may have been preferred.

Give the inherently recursive structure of the merge sorts, I'm glad I chose the recursive method. That being said, far fewer implementations of the recursive version of the heap sort exist than iterative ones.

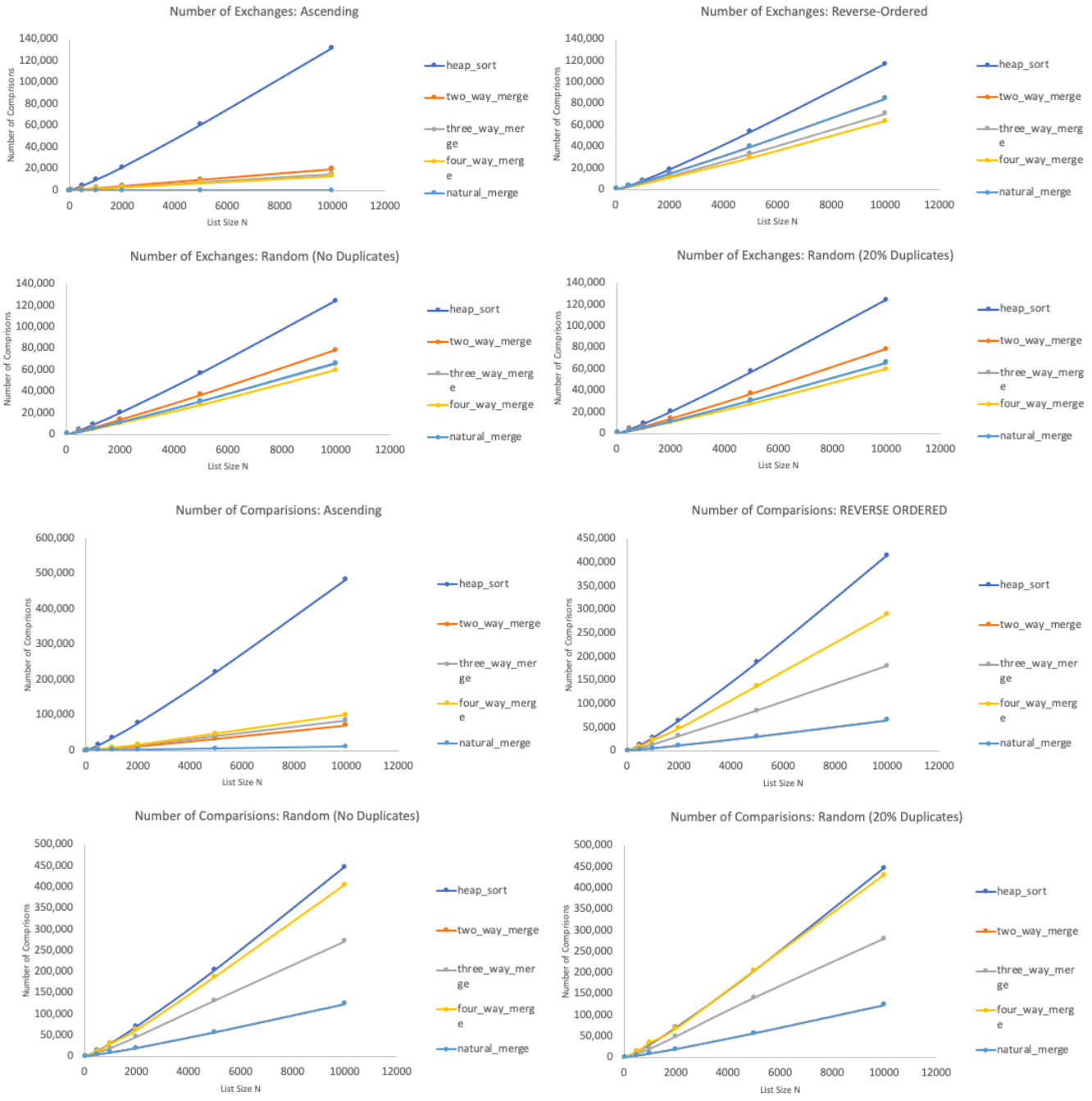
Discussion of the time and space efficiency of the program.

Runtime

I analyzed the complexity of each sorting algorithm. Based on information in the lecture notes and other research[2], the worst-case complexity of each method is below:

- Two-way merge: n * log2n
- Three-way merge: n* log3n
- Four-way merge: n * log4n
- Natural merge: Best case O(n) and worst case O(n2)
- Heap sort: n * log2n

The charts below summarize the performance of each sort.



[2] (link, link, link)

Summarization of Lessons Learned and What I Could Have Done Differently.

Although life is hectic, I should have begun this lab sooner. Life is hectic, and it's sometimes hard to balance.