Kordel France
Lab 4
Class 605.202 Section 81

Lab 4 Analysis Document

FranceLab4 is focuses on the analysis of different sorting algorithms under different conditions and different distribution of data. Operationally, *FranceLab4* is a software module written in Python that performs automatic data generation, analysis of 4 different sorting algorithms, and logs metrics about the performance of these algorithms over different permutations of data.

Recursion vs. Iteration

Recursion was leveraged throughout each of my sorting algorithms. It was an intuitive choice for me partially since I leveraged recursion heavily in *Lab2* and *Lab3*. The nature of merge sorts already easily lend themselves to recursion due to the repetitive bifurcating-merging behavior. I made sure to highlight the recursive calls for each sorting algorithm in the module. One thing that was obvious during my analysis was how noticeable the time difference is between sorting smaller and larger files. I imagine this is at least partially due to such a large recursion stack being maintained, especially for large, randomized data files. This is where the argument for iteration starts to become more appealing than the argument for recursion since we have a strain on resources with the amount of RAM the recursion stack occupies. *Reference 4* contained some excellent insight on how to implement a *k-way* merge through iterative 2-way partitions. While I didn't implement it in the lab, if the recursion stack became a practical issue I would address it through the implementation of tail recursion. Tail recursion will allow us to maintain recursive behavior while not compromising space complexity. Despite iteration's potential savings, I still believe recursion of some form is best suited for these algorithms. Given more time, it would have been an interesting experiment to perform these sorts recursively in parallel with an iterative algorithm to obtain differences in sorting execution times.

Description of Chosen Data Structures

I elected to use the list primitives in Python as my primary data structures for representing the data. It was mentioned in class that the construction of our own ADT was not required since the goal of this lab was to conduct analysis on different sorting algorithms, so I felt comfortable simply leveraging the built-in list / array primitives.

I constructed my own *Metric* object to hold information about each algorithm's performance as well. Each of these objects contained ten properties that held metrics from the algorithm's performance as well as datasets of both before and after sorting taking place. This *Metric* object proved very useful in being able to categorize the sorting runs. I was able to categorize them by data distribution type (either *sorted*, *reverse-sorted*, or *randomized*) and then again in suborder of

sorting algorithm. This allowed me to display the data cleanly to the console and write it to output csv files.

Complexity and Algorithm Analysis

FranceLab4 graphs the performance of each of the sorting algorithms side-by-side to observe how they all compares in the number of comparisons and exchanges. One of the most difficult tasks of this lab was to figure out what to count as a complexity and what to count as an exchange in each algorithm. Multiple loops and array traversals (especially for 3-way and 4-way merge sorts) make this rather difficult at first glance. As such, the validity of the following discussion is contingent upon whether or I correctly wrote my code to correctly account for these exchanges and comparisons.

Sorted Data

Please refer to Figure 1 for an actual plot from *FranceLab4*. The figure shows the number of comparisons and exchanges for each merge sort analyzed over sorted data of *n = 50, 500, 1000, 2000, 5000.*

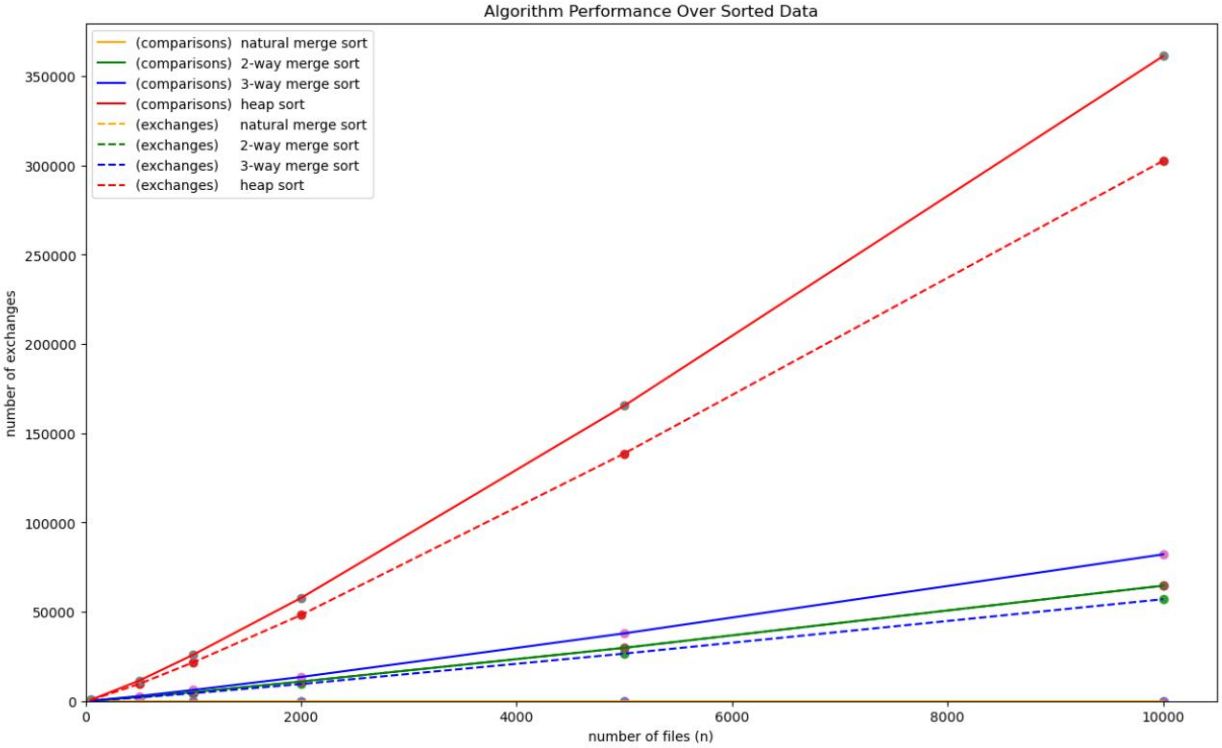

Figure 1 - algorithm performance over sorted data

One can see that the numbers of exchanges and operations required for heap sort (red) are astoundingly higher from the rest of the sorting algorithms. Both numbers for natural merge sort

(yellow) can barely be seen at the bottom because the number of operations required to sort a presorted, ascending order array with the natural merge algorithm are essentially negligent. We see that 2-way and 3-way merge sorts show closer competition and that, while the 3-way merge sort's comparison count is higher, its number of exchanges is lower than its 2-way merge counterpart. This was interesting behavior to me that I saw make more sense over the other two data distributions. Addressing the reverse-ordered data shows why.

Reverse-Ordered Data

Please reference Figure 2 for an identical distribution as that of Figure 1 but over reverse-sorted data instead. Once again, heap sort takes a distant lead with its number of exchanges and comparisons. In fact, the numbers seem to look nearly the same in the forward ordered data above as they do here, a trend that will carry through to the randomized distribution as well. What is particularly interesting is the fact that natural merge sort's metrics have skyrocketed much higher; whereas in the forward ordered dataset its numbers were negligible, here it is ranks as the second highest computational cost. Even more interesting is how 3-way merge sort's comparison and exchange metrics have converged toward each other but moved *below* 2-way merge sort's numbers. The hypothesis that 2-way and 3-way merge sorts' comparison and

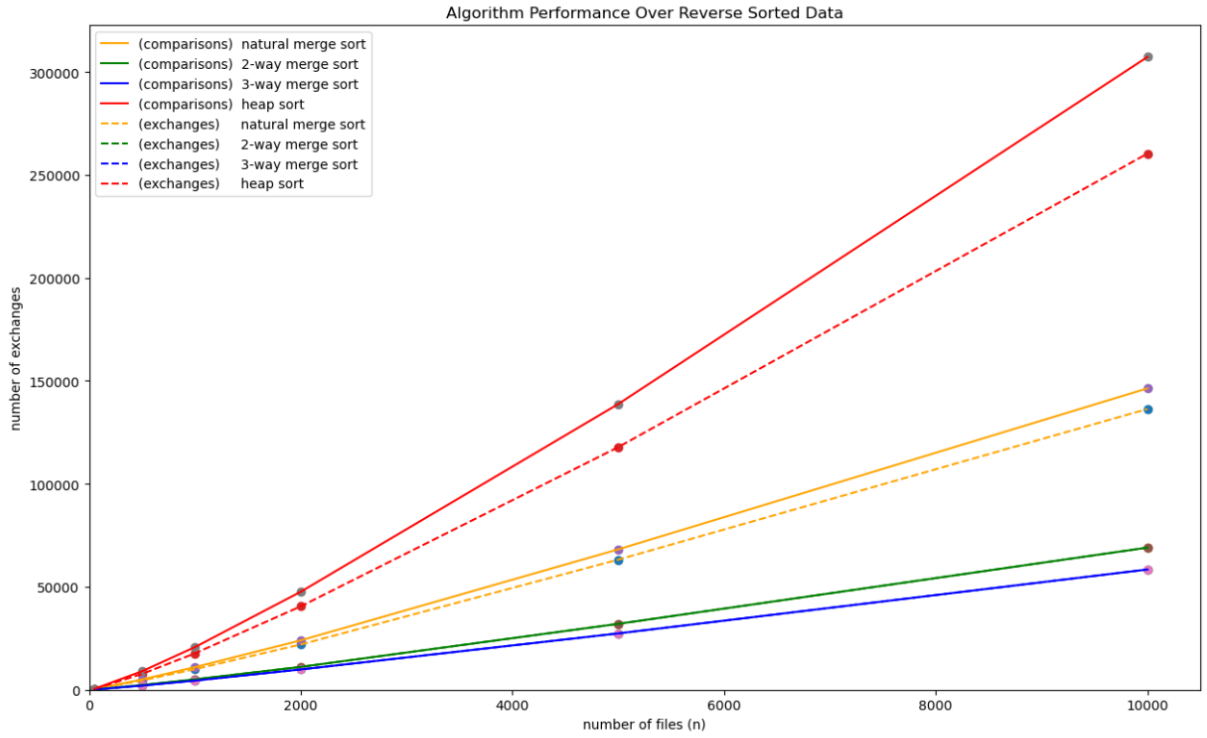

Figure 2 - algorithm performance over reverse-sorted data

exchange counts converge closer together as the dataset becomes more "worst-case" proved true as I completed more runs. This begs the question about a 4-way merge—would metrics from 4-way merge rank below its 3-way counterparts and behave similarly? What about a 5-way—or even *k-way*—merge? Sources 4, 5, and 5 had some great insight into the k-way merge algorithm and asymptotic costs.

Randomized Data

Please reference Figure 3 for the algorithmic performance on randomized data. I experimented with the randomized datasets for quite some time to ensure as much randomness and as little duplicates in data as possible. Though still all fundamentally random, different randomized datasets produced noticeably different results. For the third time in a row, heap sort takes the lead on metrics. It seems that heap sort, while very consistent with its numbers, will quite simply always result in a higher algorithmic cost. We also acknowledge that natural merge sort notches another runner-up on high costs but that the remaining two merges are much closer together than before. Whereas the number of exchanges for the 3-way merge was identical to its number of comparisons in the preceding graph, here the number of exchanges is noticeably smaller. This suggests that for more normalized (random) data, a *k*-way merge with a higher order *k* may prove to provide the best performance.

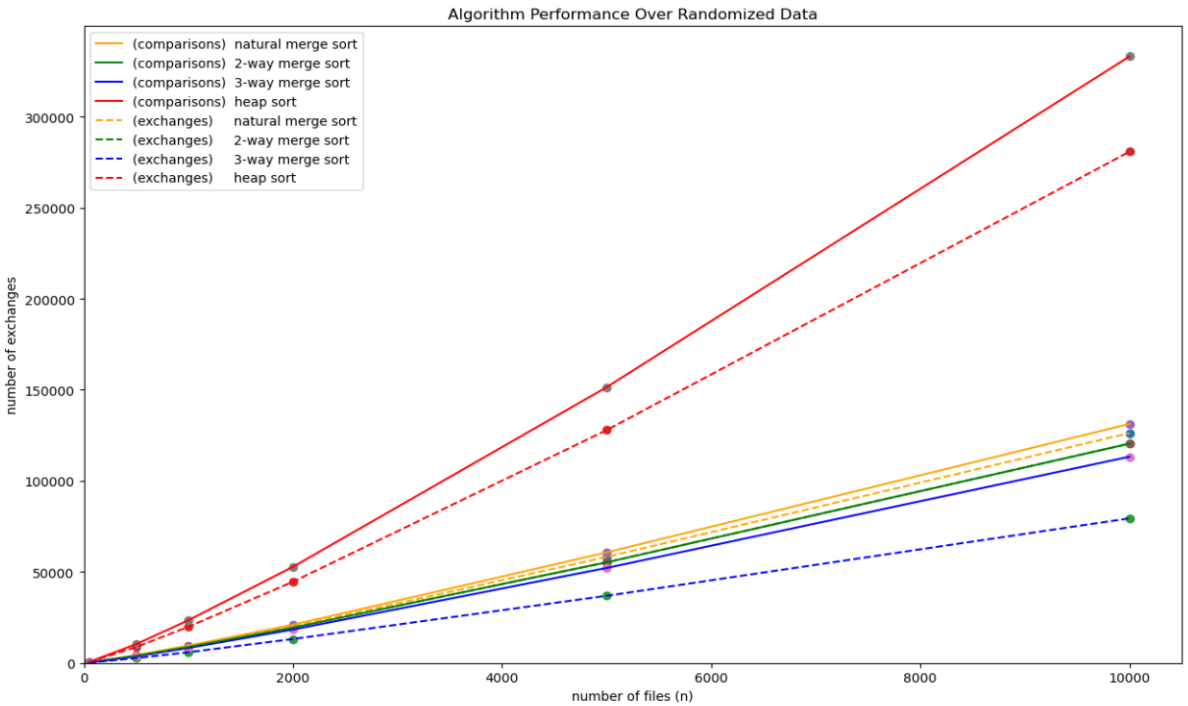

Figure 3 - algorithm performance over randomized data

I thought it prudent to ensure that the results in the above randomized data were not a significant standard deviation from the mean so I performed the operation again over a different randomized dataset. Reference Figure 4 below. The data between both randomized plots look very similar. We are able to validate repeatability between these two plots to gain confidence that a *k*-way merge sort will grant better performance on data that follows a normal distribution (truly random). We can also gather additional insight in that a higher order *k* for a *k-way* merge sort will grant you some of the best computational performances if the data is known to not be sorted before search. *However,* to this point, if the data is known to be ascendingly sorted and we are using a high order *k*, we can simply partition the array in the opposite direction and decrement pointers, effectively performing a reverse-sorted operation on a forward-sorted array to extract better performance from our algorithm.

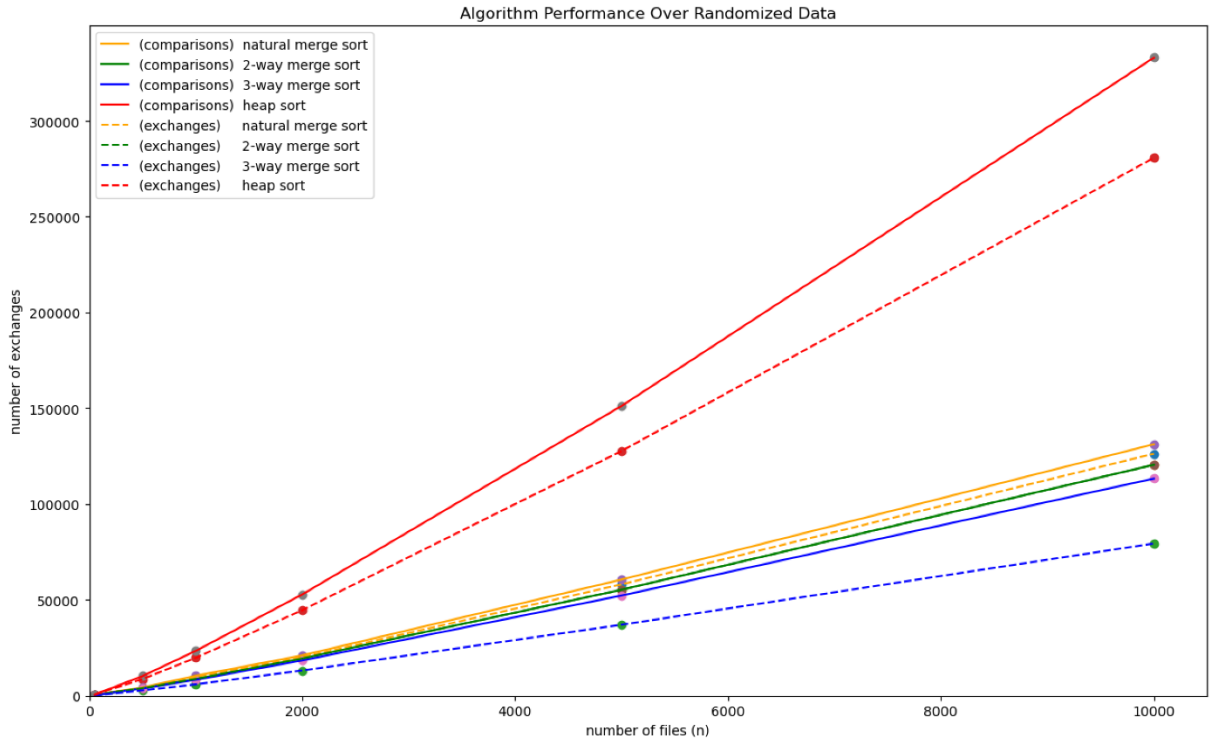

Figure 4 - a second dataset of randomized numbers to validate repeatability.

For a full distribution of the data above, please reference Table 1 below. All 72 runs are included. Consulted sources [4][5][6] suggest that the number of comparisons performed by a sort at a worst-case scenario converge to $n * lg(n) + 2^{\lfloor lg(n) \rfloor} + 1$. However, my metrics show different. At times, the program would hit this number or very close, but for the runs below, they show slightly different.

Table 1 - Raw data for sorted, reverse-sorted, and randomized plots above

#	data_type	n	algorithm	comparisons	exchanges	time
1	random	50	2-way_merge	216	216	time: 0.000000 s
2	random	500	2-way_merge	3859	3859	time: 0.008997 s
3	random	1000	2-way_merge	8700	8700	time: 0.019000 s
4	random	2000	2-way_merge	19400	19400	time: 0.028003 s
5	random	5000	2-way_merge	55305	55305	time: 0.054001 s
6	random	10000	2-way_merge	120455	120455	time: 0.118999 s
7	reverse	50	2-way_merge	153	153	time: 0.000998 s
8	reverse	500	2-way_merge	2272	2272	time: 0.054002 s
9	reverse	1000	2-way_merge	5044	5044	time: 0.085002 s
10	reverse	2000	2-way_merge	11088	11088	time: 0.027999 s
11	reverse	5000	2-way_merge	32004	32004	time: 0.066998 s
12	reverse	10000	2-way_merge	69008	69008	time: 0.183334 s
13	sorted	50	2-way_merge	133	133	time: 0.000000 s
14	sorted	500	2-way_merge	2216	2216	time: 0.002999 s
15	sorted	1000	2-way_merge	4932	4932	time: 0.009999 s
16	sorted	2000	2-way_merge	10864	10864	time: 0.031001 s
17	sorted	5000	2-way_merge	29804	29804	time: 0.143664 s
18	sorted	10000	2-way_merge	64608	64608	time: 0.091000 s
19	random	50	3-way_merge	208	162	time: 0.000999 s
20	random	500	3-way_merge	3635	2634	time: 0.007016 s
21	random	1000	3-way_merge	8176	5856	time: 0.013999 s
22	random	2000	3-way_merge	18335	13069	time: 0.017002 s
23	random	5000	3-way_merge	52169	36956	time: 0.041003 s
24	random	10000	3-way_merge	113151	79367	time: 0.092585 s
25	reverse	50	3-way_merge	133	133	time: 0.000999 s
26	reverse	500	3-way_merge	2021	2021	time: 0.006999 s
27	reverse	1000	3-way_merge	4436	4436	time: 0.012579 s
28	reverse	2000	3-way_merge	9848	9848	time: 0.020999 s
29	reverse	5000	3-way_merge	27345	27345	time: 0.037000 s
30	reverse	10000	3-way_merge	58383	58383	time: 0.096997 s
31	sorted	50	3-way_merge	171	126	time: 0.001001 s
32	sorted	500	3-way_merge	2790	1986	time: 0.003001 s
33	sorted	1000	3-way_merge	6187	4352	time: 0.011000 s
34	sorted	2000	3-way_merge	13441	9476	time: 0.022998 s
35	sorted	5000	3-way_merge	37890	26562	time: 0.061003 s
36	sorted	10000	3-way_merge	82124	56989	time: 0.065124 s
37	random	50	heap_sort	540	464	time: 0.065124 s
38	random	500	heap_sort	10190	8628	time: 0.007985 s

39	random	1000	heap_sort	23240	19678	time: 0.031002 s
40	random	2000	heap_sort	52744	44543	time: 0.024994 s
41	random	5000	heap_sort	151632	127969	time: 0.063996 s
42	random	10000	heap_sort	332604	280348	time: 0.159040 s
43	reverse	50	heap_sort	430	373	time: 0.001000 s
44	reverse	500	heap_sort	8968	7661	time: 0.023002 s
45	reverse	1000	heap_sort	20680	17657	time: 0.021005 s
46	reverse	2000	heap_sort	47568	40493	time: 0.051000 s
47	reverse	5000	heap_sort	138720	117797	time: 0.074998 s
48	reverse	10000	heap_sort	307238	260316	time: 0.125000 s
49	sorted	50	heap_sort	644	543	time: 0.001000 s
50	sorted	500	heap_sort	11424	9567	time: 0.005000 s
51	sorted	1000	heap_sort	25926	21672	time: 0.023998 s
52	sorted	2000	heap_sort	57700	48151	time: 0.046003 s
53	sorted	5000	heap_sort	165380	138623	time: 0.149999 s
54	sorted	10000	heap_sort	361168	302541	time: 0.153601 s
55	random	50	natural_merge	268	242	time: 0.000000 s
56	random	500	natural_merge	4247	3995	time: 0.004742 s
57	random	1000	natural_merge	9413	8933	time: 0.014620 s
58	random	2000	natural_merge	20930	19938	time: 0.020998 s
59	random	5000	natural_merge	60523	57995	time: 0.039999 s
60	random	10000	natural_merge	131214	126178	time: 0.127000 s
61	reverse	50	natural_merge	344	294	time: 0.003000 s
62	reverse	500	natural_merge	4992	4492	time: 0.002998 s
63	reverse	1000	natural_merge	10984	9984	time: 0.015998 s
64	reverse	2000	natural_merge	23968	21968	time: 0.132745 s
65	reverse	5000	natural_merge	68160	63160	time: 0.070589 s
66	reverse	10000	natural_merge	146320	136320	time: 0.073036 s
67	sorted	50	natural_merge	1	0	time: 0.000000 s
68	sorted	500	natural_merge	1	0	time: 0.000000 s
69	sorted	1000	natural_merge	1	0	time: 0.001002 s
70	sorted	2000	natural_merge	1	0	time: 0.001000 s
71	sorted	5000	natural_merge	1	0	time: 0.006000 s
72	sorted	10000	natural_merge	1	0	time: 0.003003 s

Please refer to *Table 2* for a complete analysis of time and space complexities for every function in the *FranceLab4* module.

Table 2 - Complexity values for every algorithm in FranceLab4

FranceLab4 Complexity Analysis				
			Worst Case	
#	File	Function	Time Complexity	Space Complexity
1	__init__.py	*[file]*	N/A	N/A
2	__main__.py	*[file]*	4n^3	n^2
3	constants.py	*[file]*	3	12
4	file_manager	*generate_sorted_file*	n	n
5		*generate_reverse_sorted_file*	2n	n
6		*generate_randomized_file*	n	n
7	graph_data	*stratify_data_sorts*	n	n
8		*stratify_data_algos*	3n	n
9		*graph_exchanges*	8n	n
10		*write_data_to_file*	n	1
11		*present_and_save_summary*	n^2	n
12	heap_sort	*construct_heap*	1	1
13		*heap_sort*	2n	n
14		*reset_counts*	1	1
15	merge_sort_2_way	*merge_sort_two_way*	3n	n
16		*reset_counts*	1	1
17	merge_sort_3_way	*merge_three_way*	7n	n
18		*merge_sort_three_way_recursive_helper*	n	n
19		*merge_sort_three_way*	2n	n
20		*reset_counts*	1	1
21	merge_sort_natural	*merge_sort_natural*	n^2 * 2n	n
22		*merge_natural*	3n	n
23		*reset_counts*	1	1
24	Metric	*init*	1	1
25		*fit_power_regression_curve_comparisons*	7n	n
26		*fit_power_regression_curve_exchanges*	7n	n

The most demanding tasks on time complexity was actually the driver code in *__main__.py*. I have 3 nested for-loops that are generating data files, saving them, and sorting them appropriately. This could definitely be optimized and I found a dynamic solution that works well. If given more time, I would implement it to relieve some time complexity. All other algorithms behaved as expected. The natural merge sort maintained the next highest time complexity value, but it isn't immediately apparent on how to bring that down.

What is concerning is that the program will have a time complexity of $\boldsymbol{4O(n^3)}$ every time it runs due to the way the procedure is built. Luckily, the values of *n* are small at the moment (4, 3, and 6), but it would be good to implement best practices and integrate a better solution in. There are some concerns about recursion depth within a for-loop for a proposed solution of the 4-way merge sort, but the algorithm for this is still in prototype and hasn't been fully determined. However, I hypothesize that a *k*-way merge sort would not surpass a time complexity of $\boldsymbol{O(n^3)}$**.**

Based off of the above analysis and, in consideration of the data presented in *Table 1*, the *FranceLab4* program has a worst-case time complexity of $\boldsymbol{4O(n^3)}$ and a worst-case space complexity of $\boldsymbol{O(n^2)}$.

Lessons Learned

All in all, *Lab4* provided me with another angle of appreciation for recursion. This lab taught me a lot about *k*-way merge sort algorithms and when they should/shouldn't be used. I also found it interesting to formally see how *already* sorted data can provide worst performance than randomized data for some algorithms. This intensified my appreciation for the beauty of sorting and encouraged me to follow-up this lab with a similar comparison against other sorting algorithms and data distributions.

Optimizations for Later Versions

As specified previously, I would choose to implement at least two of my algorithms utilizing tail recursion. Additionally, my code contains a mix of camel case (camelCase) and "python_style" (python_style) syntax which I would like to clean up. Working on the project across multiple periods while working across multiple languages with work is the driver behind the inconsistency. I would also optimize the time complexity of the driver code in *__main__.py*. Finally, I would like to alter *Lab4* to accommodate higher order *k* values for *k*-way merging algorithms.

Addressing Requirements in *Lab 4*

FranceLab4 addresses the requirements as specified in the *Lab 4* handout. Specifically, the program generates 3 different datasets over 6 (the lab requirement was 5, but I added a 6^{th} for curiosity sake) different counts and inputs each of those into the 2-way merge sort, 3-way merge sort, natural merge sort, and heap sort algorithms for a grand total of 72 data runs. The data is sorted, graphed, and output to the user with metrics on algorithmic efficiency for each run. Finally, the user is presented with a summary for all data runs performed and the program is terminated.

Enhancements

The program provides 9 major enhancements (along with many other minor enhancements) on top of the original requirements. These enhancements are also elaborated upon in the .README document associated with the module

1) Time Delays - processing is paused briefly throughout the program to allow the user time to read and interpret the output. This creates for a much better user experience.

2) An additional count of `10,000` was added to provide additional insight into algorithm performance.

3) Execution time is tracked and monitored for each sorting algorithm.

4) Each sorting run is graphed in a *`.csv`* file. Files are named {algorithm_name}-{date_type}-{n} count.csv such as *'3-way_merge_random_1000count.csv'*. Each file is located in the *output_files* directory and contains the following properties:
    - the data type.
    - file count.
    - sorting algorithm name.
    - number of comparisons performed.
    - number of exchanges performed.
    - an equation that plots the trajectory of the number of comparisons made by the algorithm over this data type as `n` scales along with the correlation coefficient between the equation and the observed data.
    - an equation that plots the trajectory of the number of exchanges made by the algorithm over this data type as `n` scales along with the correlation coefficient between the equation and the observed data.
    - the initial dataset for each run before it entered the sorting algorithm.
    - the final dataset for each run after it entered the sorting algorithm and was fully sorted.
    - the execution time (in seconds) for the algorithm to completely sort the data.

5) A `Summary Table` is provided at the very end of the program that shows the performance of each algorithm over different data distributions. A *`FINAL_ANALYSIS.csv`* file is a direct copy of the *`Summary Table`*, but in a *`.csv`* file that shows performance over all 72 sorting runs.

6) Equations for trajectory curves were calculated to extrapolate the number of comparisons \ exchanges that would be theoretically needed for very large *n* as the algorithm scales. This was accomplished by calculating coefficients of power regression to define the trajectory path based on the data gathered for similar sorts. If one opens `Metric.py` where the regression algorithms are located, they will notice the regression curve is computed from scratch with no "packages" - the regression equation is derived from low-level statistics functions. The *`numpy`* package is only used to cast an array type to one of a type easier to manipulate with these statistics functions.

7) Correlation values are calculated (again from scratch and without any use of packages) to show how well the above regression curve fits the empirically gathered data in our analysis.

8) A status is communicated to the user as a % complete in the *`__main__.py`* file while the data is being processed and sorted. This allows for a more appealing user interface and lets the user have an idea of where the program is at in its execution steps.

9) Plots of all of the data runs for each algorithm are shown and allowed for easy comparison against other algorithms. This makes it simple for the user to spot analytical trends and spot which algorithms out-perform others on certain datasets.

More On Recursive vs. Iterative Evaluation

The constraints posed in *Lab 1* and *Lab 2* illuminate the pros and cons of using both recursive and iterative procedures in evaluating mathematical expressions. While I know that iterative methods can occasionally out-perform recursive methods in time and space complexities, both *Lab 1* and *Lab 2* showed consistent algorithmic behavior across my code in this respect. Under both scenarios, the project produced worst-case time complexities of $O(n^3)$ and worst-case space complexities of $O(n^2)$. However, the iterative solution wins my preference due to the facts that it is easily represented the mathematical evaluation rules, resulted in less lines of code, and worked seamlessly with the circular singly linked list ADT.

Conclusion

The *FranceLab4* Python module performs algorithmic analysis across multiple dataset simulations. Datasets are automatically generated without the need for the user to supply them. The program monitors algorithmic efficiency and compares different sorts to each other. In hindsight, there are some optimizations that could be made to enhance the time and space complexities and fine-tune the data preparation algorithm through tail recursion. The module

contains several enhancements that aid in usability, user interface, error handling, and functionality. This lab has reinforced my preference for recursion and allowed me to further acknowledge its beauty in algorithmic analysis, particularly in *k*-way merge sort algorithms.

References

1. Miller, B. N., & Ranum, D. L. (2014). Problem solving with algorithms and data structures using Python (2nd ed.). Decorah, IA: Brad Miller, David Ranum.

2. Rayapati, P. (2019, August 20). 3-Way merge sort. Retrieved April 25, 2021, from https://www.geeksforgeeks.org/3-way-merge-sort/

3. Woltmann, S. (2021, January 13). Merge sort – ALGORITHM, source code, time complexity. Retrieved April 24, 2021, from https://www.happycoders.eu/algorithms/merge-sort/#Natural_Merge_Sort

4. K-way merge algorithm. (2021, April 09). Retrieved April 24-27, 2021, from https://en.wikipedia.org/wiki/K-way_merge_algorithm

5. Artificial Intelligence: A Modern Approach. Third Edition. Russel, Stuart J.; Norvig, Peter. 2015, Pearson India Education Services Pvt. Ltd. p 961-962.

6. Deep Learning. Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. 2016, Massachusetts Institute of Technology. p 147 -149, 525 – 527.

7. Almes, Scott; et al. (2021). Lab 0 Sample Project, JHU Course 605.202 Section 81. Accessed through Blackboard on JHU student account 26 February 2021. Web. URL: https://blackboard.jhu.edu/bbcswebdav/pid-9469422-dt-content-rid-100642966_2/xid-100642966_2