UECS2083/UECS2453 Problem Solving with Data Structures and Algorithms *[202406]*

PRACTICAL ASSIGNMENT - MARKING REPORT

1. PERSONAL DATA

Group number :				
No	Name	ID	Programme	Total Marks
1.	Lee Kim Lan	2104745	AM	
2.	Tan Wan Xuen	2207214	AM	
3.	Tee Junn Jeh	2105387	SE	
4.	Yeap Huaizhou	2104837	SE	

2. SUBMISSION STATUS

No soft copy/ Upload wrong file(s)	Late submission of softcopy	No hardcopy	Late submission of hardcopy	No issue

3. COMPILATION AND RUNNING

Does not compile/Bytecode & batch file do not work	Compile but no output/ wrong output/ runtime error	Compile and produce output

4. PRESENTATION OF SOURCE CODES(4%)

(a)	Indent Style (2%)	☐ Poor	☐ Inconsistent	☐ Good
(b)	Identifier names (2%)	☐ Poor choice	☐ Meaningful	☐ Good naming convention

5. PROGRAM COMPONENT (46% + 4%)

Program Components	Missing/ Does not work	Major errors	Minor errors	Not robust	No issue/ Excellent design	Max marks	Marks obtained
Presentation of source codes						4	
Generate two datasets, one with 10M numbers, another with 1M English words						6	
Implement FOUR (4) sorting algorithms with TWO (2) data structures.						20	
Ensure the correctness of the implementation.						5	
Perform numerous experimental tests using the four algorithms, and get the elapsed times (average case & worst case) for the algorithms.						15	
					Total	50	

6. REPORT AND OTHER COMPONENTS (50%)

Components	Missing	Poor	Average	Good	Excellent	Max marks	Marks obtained
Design (data structures and algorithms) and discussion (efficiency and complexities)						20	
Sample input and test cases						10	
Screenshots of results						5	
Conclusion						5	
					Total	40	

Individual presentation (Involvement, language, confident, preparedness, attitude)						10	
Student 1:							
Student 2:							
Student 3:							
Student 4:							
					Total	50	

Table of Contents

Introduction

In this assignment, we explore the performance and efficiency of various sorting algorithms, with focus on both comparison-based (**Merge-Sort, Selection-Sort, Tim-Sort**) and non-comparison-based (**Counting-Sort**) algorithms.

Our objective was to comprehensively analyse these algorithms by applying them to datasets of varying sizes and characteristics. We evaluated their behaviour across 11 experimental designs, including random data, sorted data, reversed data, datasets with repeated duplicate, negative data etc.

Time complexity served as the primary metric for assessing computational efficiency, enabling a clear comparison of each algorithm's performance under different conditions. Additionally, we compiled detailed tables of result to compare the efficiency of the four algorithms across these scenarios, providing a comprehensive understanding of their strengths and limitations.

This analysis highlights the computational strengths and weaknesses of each algorithm, offering practical insights for real-world applications. By understanding their performance across different scenarios, we aim to guide the selection of the most suitable sorting techniques for specific data challenges, ultimately enhancing the efficiency of data processing tasks.

Input Data and Test Case

Input Data

The input data used include one English words dataset (Words.txt) and one 10 M numbers dataset (RandomNumbers.txt). The English words are retrieved from online sources (included in references) while the numbers are generated using a self-defined python program DataGeneration.py. The methods listed below are used to generate datasets for each experiment design.

Dataset	**English Words Data**	**Numbers Data**
Random	getStringList(int)	getNumberArrayList(int) getNumberLinkedList(int)
Sorted	getSortedStringList(int)	getSortedNumberArrayList(int) getSortedNumberLinkedList(int)
Reversed	getReversedStringList(int)	getReversedNumberArrayList(int) getReversedNumberLinkedList(int)
Nearly Sorted	getNearlySortedStringList(int)	getNearlySortedNumberArrayList(int) getNearlySortedNumberLinkedList(int)
Duplicates	getDuplicateStringList(int)	getDuplicateNumArrayList(int) getDuplicateNumLinkedList(int)
Upper Case & Lower Case	getUpperCaseStringList(int) getLowerCaseStringList(int)	-
Outliers	-	getOutliersArrayList(int) getOutliersLinkedList(int)
Skewed	-	getNegativeNumberArrayList(int) getNegativeNumberLinkedList(int)

Negative	-	getNegativeNumberArrayList(int) getNegativeNumberLinkedList(int)
Even	-	getEvenNumberArrayList(int) getEvenNumberLinkedList(int)
Odd	-	getOddNumberArrayList(int) getOddNumberLinkedList(int)
Floating Point	-	getDoubleNumberArrayList(int) getDoubleNumberLinkedList(int)

Test Case

Test Case	Test Title	Test Summary	Test Steps	Test Data	Expected Result	Post-condition
TC001	Data size [String + Number]	Compare and analyse the performance of each algorithm on datasets with different data size.	1. Generate datasets (10M numbers & 500k String) 2. Run all sorting algorithms and analyse their performances.	String (10K - 500K), Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: O(n +k) Merge Sort: O(n log n) Tim Sort: O(n log n)	Time elapsed of sorting is recorded
TC002	Type of Lists [ArrayList + LinkedList]	Compare and analyse the performance of each algorithm on datasets using ArrayList and LinkedList.	1. Generate datasets (10M numbers) 2. Run all sorting algorithms and analyse their performances.	Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: O(n +k) Merge Sort: O(n log n) Tim Sort: O(n log n)	Time elapsed of sorting is recorded
TC003	Data sequence [random + sorted + nearly sorted]	Compare and analyse the performance of each algorithm on datasets with different data sequence.	1. Generate datasets (10M numbers & 500k String) 2. Run all sorting algorithms and analyse their performances.	String (10K - 500K), Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: O(n +k) Merge Sort: O(n log n) Tim Sort: O(n log n)	Time elapsed of sorting is recorded
TC004	Data Type [String Vs Number]	Compare and analyse the performance of each algorithm on datasets with different data type.	1. Generate datasets (10M numbers & 500k String) 2. Run all sorting algorithms and analyse their performances.	String (10K - 500K), Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: O(n +k) Merge Sort: O(n log n) Tim Sort: O(n log n)	Time elapsed of sorting is recorded

TC0 05	Duplicates [non-duplicate + duplicate]	Compare and analyse the performance of each algorithm on datasets with duplications.	1. Generate datasets (10M numbers) 2. Run all sorting algorithms and analyse their performances.	Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: $O(n + k)$ Merge Sort: $O(n \log n)$ Tim Sort: $O(n \log n)$	Time elapsed of sorting is recorded
TC0 06	Presence of Outliers [no outliers + contains outliers]	Compare and analyse the performance of each algorithm on datasets with outliers.	1. Generate datasets (10M numbers) 2. Run all sorting algorithms and analyse their performances.	Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: $O(n + k)$ Merge Sort: $O(n \log n)$ Tim Sort: $O(n \log n)$	Time elapsed of sorting is recorded
TC0 07	Data distribution [random + skewed]	Compare and analyse the performance of each algorithm on datasets with different data distribution.	1. Generate datasets (5M numbers) 2. Run all sorting algorithms and analyse their performances.	Number (10K - 5M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: $O(n + k)$ Merge Sort: $O(n \log n)$ Tim Sort: $O(n \log n)$	Time elapsed of sorting is recorded
TC0 08	Floating points data [random + Double]	Compare and analyse the performance of each algorithm on datasets with floating points.	1. Generate datasets (10M numbers) 2. Run all sorting algorithms and analyse their performances.	Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: $O(n + k)$ Merge Sort: $O(n \log n)$ Tim Sort: $O(n \log n)$	Time elapsed of sorting is recorded
TC0 09	Even and Odd number	Compare and analyse the performance of each algorithm on datasets with odd and even numbers.	1. Generate datasets (5M numbers) 2. Run all sorting algorithms and analyse their performances.	Number (10K - 5M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: $O(n + k)$ Merge Sort: $O(n \log n)$ Tim Sort: $O(n \log n)$	Time elapsed of sorting is recorded
TC0 10	Positive and Negative Numbers [all positive + all negative]	Compare and analyse the performance of each algorithm on datasets with negative numbers.	1. Generate datasets (10M numbers) 2. Run all sorting algorithms and analyse their performances.	Number (10K - 10M)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: $O(n + k)$ Merge Sort: $O(n \log n)$ Tim Sort: $O(n \log n)$	Time elapsed of sorting is recorded

TC011	Upper- and Lower-Case Words [String]	Compare and analyse the performance of each algorithm on datasets with different String cases.	1. Generate datasets (500k String) 2. Run all sorting algorithms and analyse their performances.	String (10K - 500K)	Worst case time complexity: Selection Sort: $O(n^2)$ Counting Sort: $O(n + k)$ Merge Sort: $O(n \log n)$ Tim Sort: $O(n \log n)$	Time elapsed of sorting is recorded

Experimental Design and Implementation

Data Structure

Knowing that ArrayLists and LinkedLists are data structures, they are used to test the efficiency and effectiveness of the Selection-Sort, Merge-Sort, Tim-Sort and Counting-Sort algorithms. Table below shows the data structures used to store different types of datasets.

English Words (ArrayList)	Numbers (ArrayList and LinkedList)
• Random • Sorted • Reversed • Nearly Sorted • Presence of Duplicates • Upper Case • Lower Case	• Random • Sorted • Reversed • Nearly Sorted • Presence of Duplicates • Presence of Outliers • Skewed data • Negative Numbers • Even Numbers • Odd Numbers • Floating Point Data

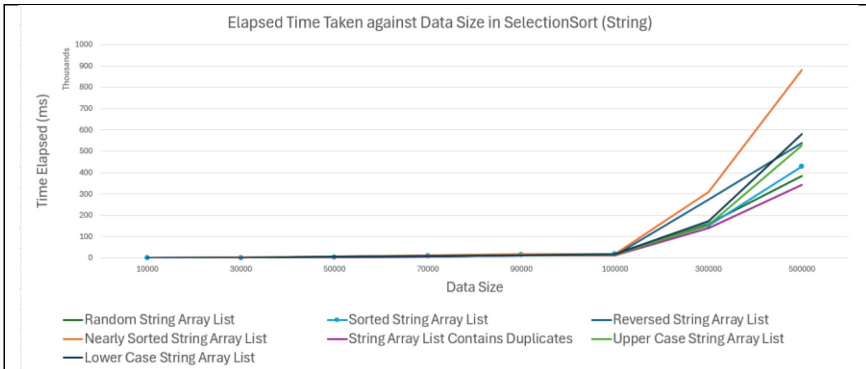
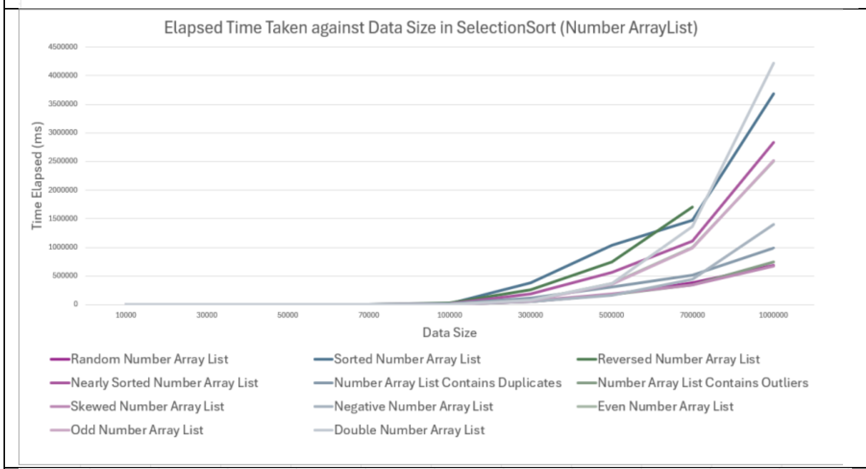
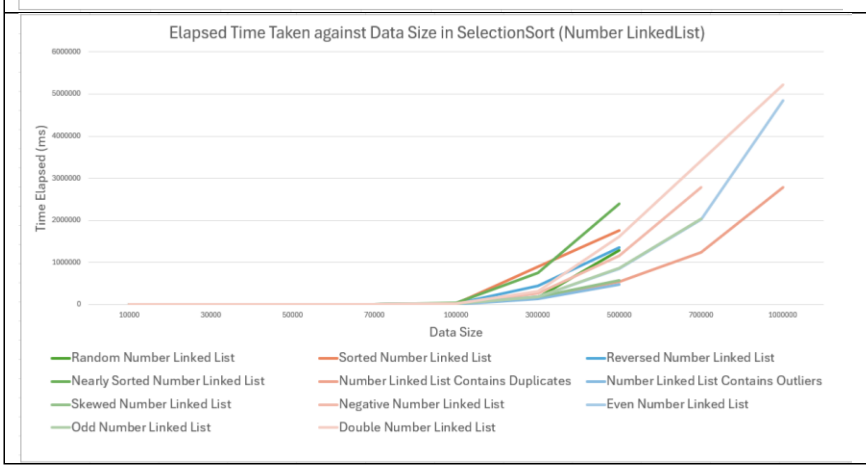
Besides that, noticed that LinkedList is facing the high elapsed time problem during sorting, Node was used to access the LinkedList, overcoming the lengthy sorting time as it was exceeding one hour.

Sorting Algorithms

Selection-Sort Algorithm

The **Selection Sort** algorithm is a comparison-based sorting method that operates by repeatedly selecting the smallest (or largest) element from the unsorted portion of an array and placing it in its correct position. The algorithm proceeds to divide the list into two distinct portions: a sorted portion, located on the left, and an unsorted portion, situated on the right. In its initial state, the sorted portion empty since no element has sorted, and the algorithm proceeds to transfer elements from the unsorted portion to the sorted portion. This is achieved by selecting the minimum element from the unsorted portion, exchanging it with the leftmost unsorted element, and adjusting the boundary between the two portions (GeeksforGeeks, 2024). This process is repeated for each remaining element in the unsorted list until all elements have been sorted and fulfill the sorted portions.

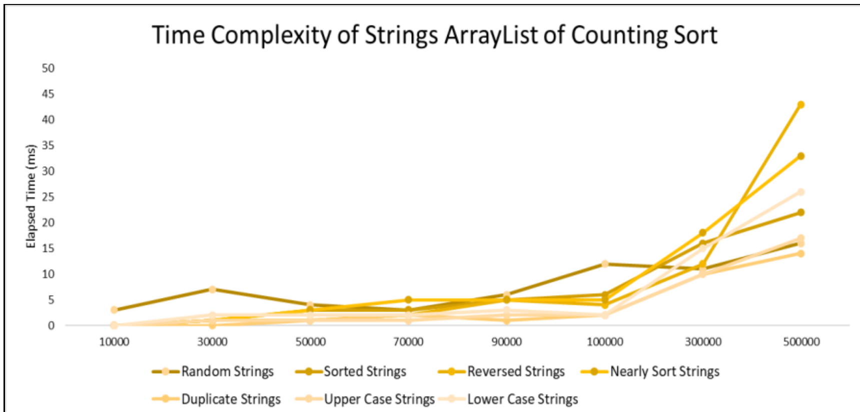
In terms of performance, Selection Sort is **not an efficient algorithm** for large datasets due to its quadratic time complexity of **O(N²)** in all cases, including the best, worst, and average cases. This is due to the fact that, for each element, the algorithm makes comparisons with the remaining unsorted elements. In the worst and average cases, the algorithm performs approximately $\frac{n\,(n-1)}{2}$ comparisons (Baeldung, n.d.), where n is the number of elements in the list. This formula represents the sum of the first n−1 integers and illustrates how the number of comparisons grows quadratically as the size of the list increases.

	The graph of the Selection Sort in String illustrated that the sorting time increases as the data size increases. Once the threshold point is reached, the elapsed time reveals a noticeable growth. This graph shows that, as time progresses, the time needed to sort the String increases dramatically. It obeys with the time complexity of O(N²), no matter it is in which conditions including best, average, and worst cases.
	This graph showcases the performance of Selection Sort in Number when using ArrayList. It behaves consistently across various types of numerical data and performing O(N²) time complexity. The characteristics of the input data might affect the number of swaps and comparison needed, but the fundamental approach is remaining the same. It performs the same number of comparisons for any input type but may require varying numbers of swap based on the nature of the data.
	In contrast with the Number ArrayList approach, most of the designs that perform in LinkedList will reach the threshold point faster than the ArrayList. It requires more time to sort when it comes to big data sets and time complexity will be O(N²) in every design. After finding the minimum node, it adjusts the pointers, placing it at the start and repeat this process for the rest of the unsorted list. This causes manual abortion since every sorting exceeding one hour is inefficient.

Counting-Sort Algorithm

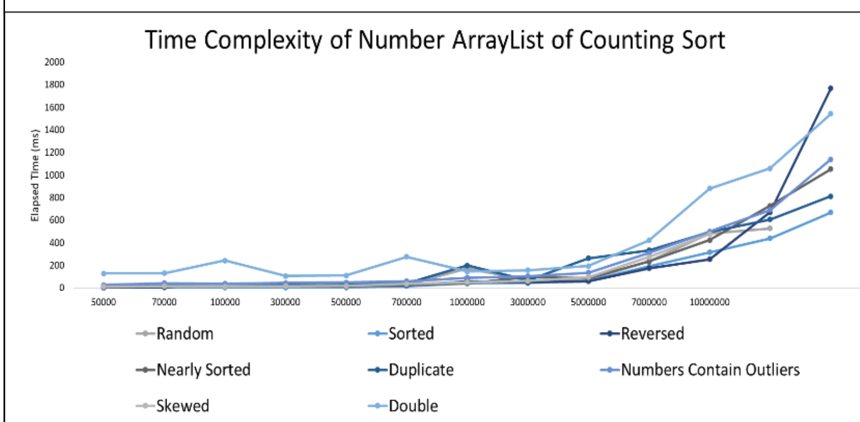
Counting sort, as a **non-comparison-based method**, sorts the keys within a specific range. By arithmetic operations, it identifies each object's index position to sort the elements. The time complexity of counting sort is O(n+k). Unlike comparison-based algorithms, it does not depend on the element-by-element comparisons. Hence, when the datasets are having narrow range of values, counting sort is very efficient. The time complexity of counting sort depends on the value of k, which leads to different scenarios for best, average, and worst cases. However, the time complexity of Counting Sort was always **O(n+k)**, no matter best, average, or worst cases, as its efficiency is determined by the ratio between n and k (*Counting Sort (with Code)*, n.d.). The table below shows the comparison of time complexity within counting sort based on different data structures. Since the program is facing a heap space error, the heap size was increased by setting the initial heap size from 512MB to 6144MB, as the laptop has 8GB of RAM. Overall, all the graphs satisfied O(n+k).

<table>
<tr>
<td>



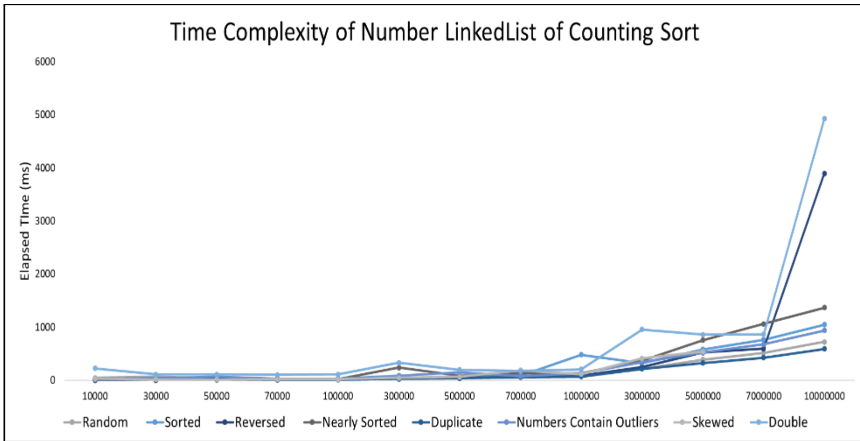
ArrayList (String)</td>
<td>The graph illustrates the sorting time expands with dataset size, particularly for reversed strings. Nearly sorted and random strings perform consistently on smaller datasets but slow down dramatically as size increases. Sorted and duplicate strings are the most productive, with duplicates affecting just greater datasets. Hence, it can be concluded:

- Best case: Duplicate strings
- Worst case: Reversed strings
- Average case: Random strings
</td>
</tr>
<tr>
<td>



ArrayList (Number)</td>
<td>According to the graph, time complexity increases dramatically approximately 1 million components, particularly for reversed, random numbers and those with outliers. Sorted and nearly sorted data are still efficient, however skewed distributions and outliers slow down Counting Sort. Double numbers, like random data, demonstrate the lowest effective results. Therefore, it was summarized as:

- Best case: Sorted numbers
- Worst case: Floating numbers
- Average case: Random numbers
</td>
</tr>
</table>



LinkedList (Number)

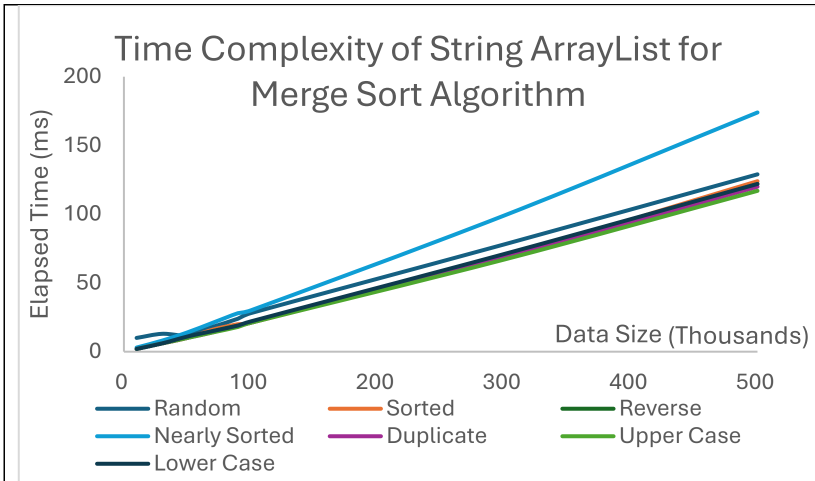
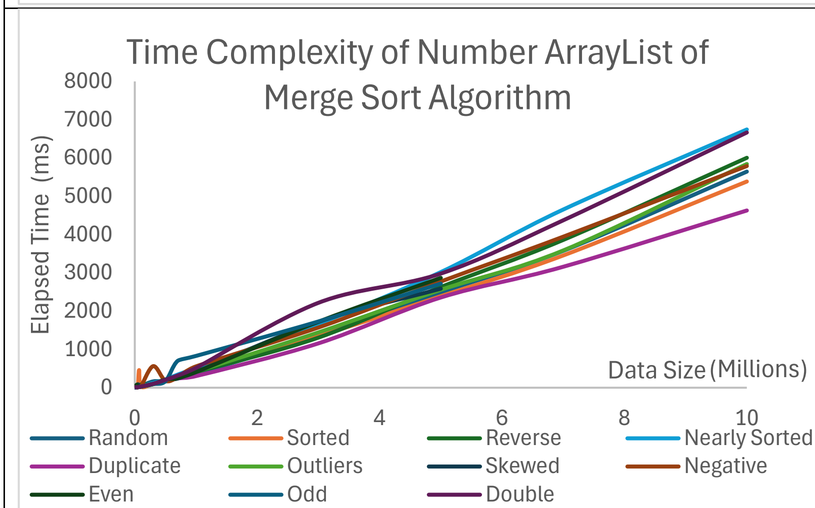
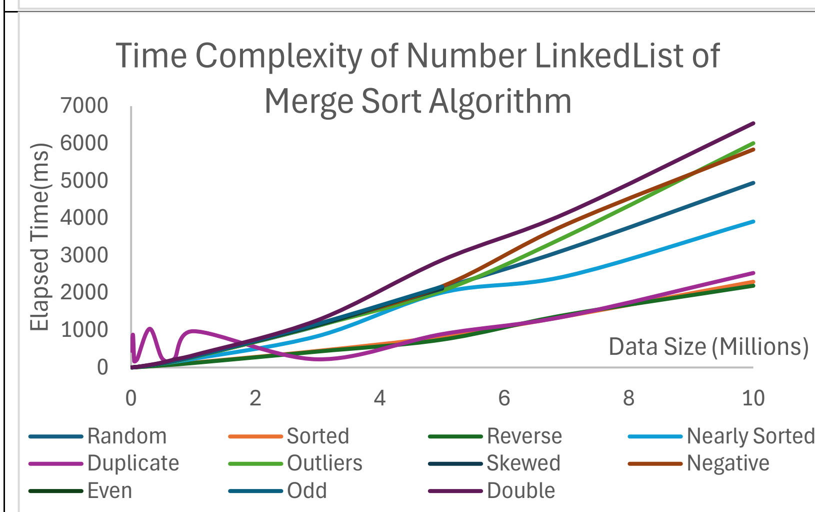
Based on the graph, LinkedLists have greater variability in sorting time than ArrayLists, particularly for huge datasets, with reversed and skewed data incurring the greatest delays. Sorted and nearly sorted numbers operate best, but performance diminishes with increasing size. Floating-point data and outliers also cause severe sorting inefficiencies. Overall, it wrapped up with:

- Best case: Duplicate numbers
- Worst case: Floating numbers
- Average case: Random numbers

Merge-Sort Algorithm

Merge sort is a highly efficient, comparison-based sorting algorithm that follows the divide-and-conquer paradigm (Tyagi & Ahlawat, 2023). It works by recursively dividing an unsorted list into smaller sublists until each sublist contains a single element or is empty. Once the list is sufficiently divided, the algorithm merges these sublists back together in a sorted order. By systematically splitting and merging, merge sort ensures that the final list is sorted. The process involves dividing the list into two halves, sorting each half, and merging the sorted halves in linear time (P. Asha Rani et al., 2023).

The worst-case time complexity of merge sort is O(nlogn), where n represents the number of elements in the list. This time complexity arises because the algorithm must recursively divide the list into smaller sublists, which takes O(logn) levels of recursion. At each level, merging the sublists requires O(n) operations, resulting in an overall complexity of O(nlogn). Hence, merge sort's worst-case performance remains efficient even for large datasets. However, it requires additional space proportional to the input size, which can be a trade-off in memory-limited environments (Lobo & Kuwelkar, 2020).

	As observed from the graph, merge sort performs consistently with the time complexity of O(n log n) for all types of cases. Merge Sort recursively divides the array into two halves, irrespective of the initial order. Whether the dataset is nearly sorted or completely reversed, Merge Sort will still perform the same number of divisions and merges. Hence, the additional time elapsed for nearly sorted dataset could be due to overhead and cache misses.
	The graph illustrates the time complexity of merge sort on various number ArrayList. The best, average, worst case obeys the time complexity of O(n log n) as the same number of divisions and merges are performed. Dataset with duplicates takes shorter sorting time as fewer comparisons and swapping are required. Overall, merge sort takes consistent time complexity regardless of random, sorted, reverse initial order.
	The graph shows the time complexity of merge sort on various number LinkedList. The time complexity for best cases, average cases and worst cases are O(n log n). In contrast to ArrayList, merge sort performs better in sorted, duplicate and reverse LinkedList. However, the sorting time for negative and outliers LinkedList are higher than average case. This might be due to the sign handling and overhead from comparing negative numbers.

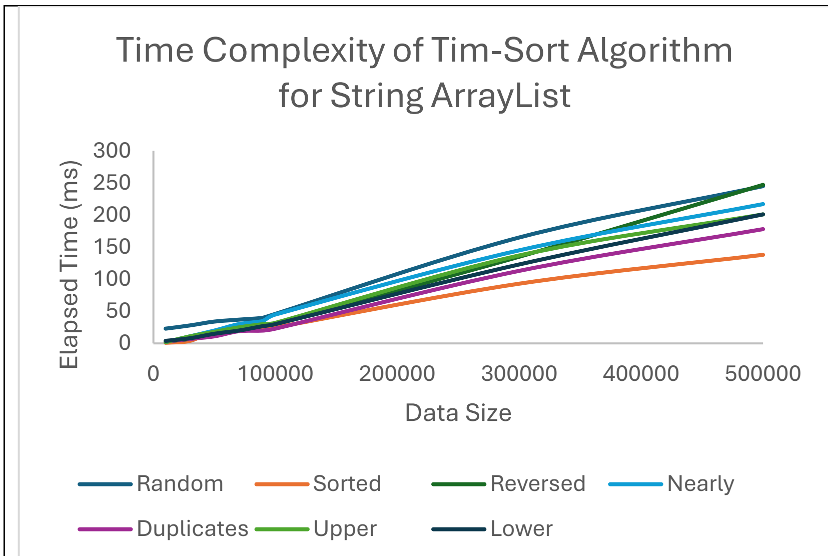
Tim-Sort Algorithm

Tim-Sort algorithm utilizes the primary functions of Merge and Insertion sort, both of which are made to deal with sorting through heaps and stacks of uncategorized data (Abdon, A et al., 2024).

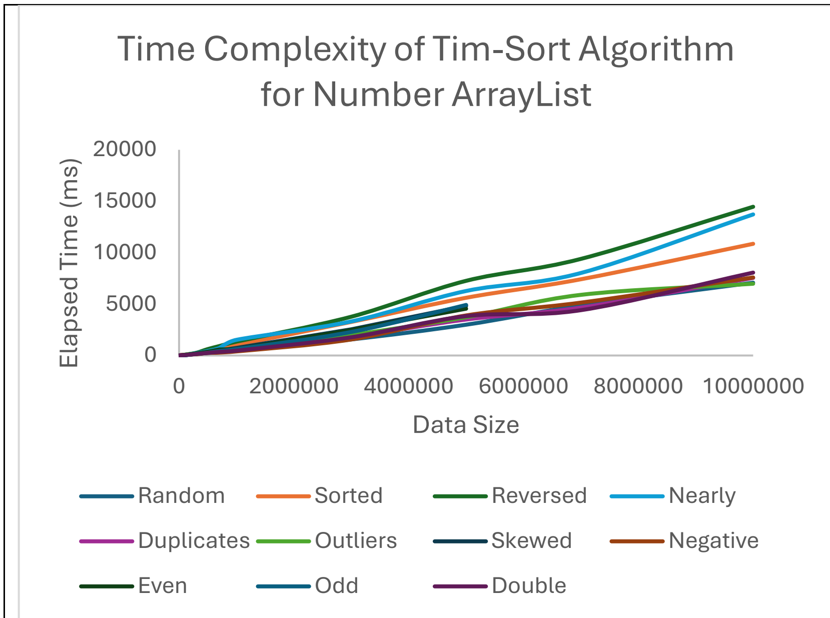
Let's refer sub-array as *run*, the minimal length of runs as *min_run* and the length of input array as *N*. In accordance with Zhang, Y. et al., 2018, Tim-Sort algorithm works by first determining *min_run*. Since insertion sort is only useful for short arrays, it shouldn't be too small. It also should not be too small either, because it will lead to more merge iterations later. When $\frac{N}{min\_run}$ is a power of 2, it is the ideal *min_run* value because merge sort works perfectly on balanced sub-arrays. However, there is not always such an integer *min_run* for every possible value of *N*, thus we choose a value in the range (32, 65) that $\frac{N}{min\_run}$ is strictly less than or equal to a power of 2.

The input array is then divided into *runs*. The algorithm first counts the number of continuous increasing or decreasing elements from current pointer. If the number is greater than *min_run*, then this sorted sub-array will be count as a *run*; else, it will be reversed to make merging easier. Otherwise, it's extended to the length of *min_run* using binary insertion sort to keep it sorted. Lastly, the sorted sub-arrays are then all merged in increasing order of size. Here, we keep doubling the size until the entire list is sorted.

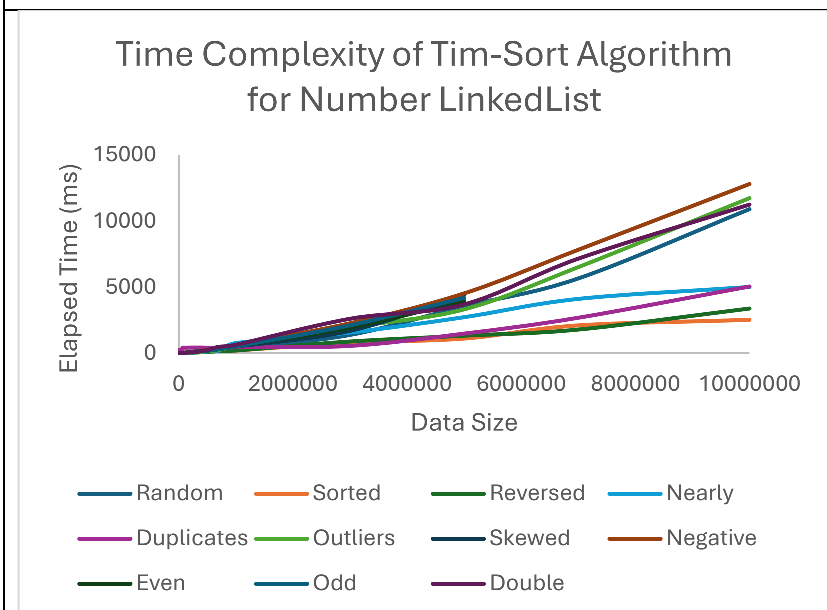
Below are the graphs of time complexity of the Tim-Sort algorithm applied to String ArrayList, Number ArrayList and Number LinkedList respectively with different dataset characteristics.



The graph shows that sorted data consistently takes the least time to sort, with a time complexity that approaches O(n). Both random and reversed data exhibit similar performance, with a time complexity of O(n log n), reflecting that Tim-Sort handles these datasets with a slightly higher elapsed time compared to sorted data. The light blue line indicates that nearly sorted data is handled slightly slower than sorted data but better than random or reversed data. The performances of duplicate, upper case and lower-case datasets are similar to that of random and reversed data, suggesting that Tim-Sort handles these cases reasonably well.



From the graph, we can observe that Tim-Sort algorithm perform particularly well on sorted data, as indicated by the orange line, which represents a time complexity of O(n). This demonstrates that Tim-Sort can efficiently handle sorted number ArrayList with linear time complexity. Next, two lines above the orange line indicates that reversed and nearly sorted data exhibit higher time complexity of O(n log n), as compared to sorted data. The remaining data conditions also display a time complexity of O(n log n) as shown by the graph, consistent with the typical performance of Tim-Sort on less-optimized or more randomly ordered datasets.



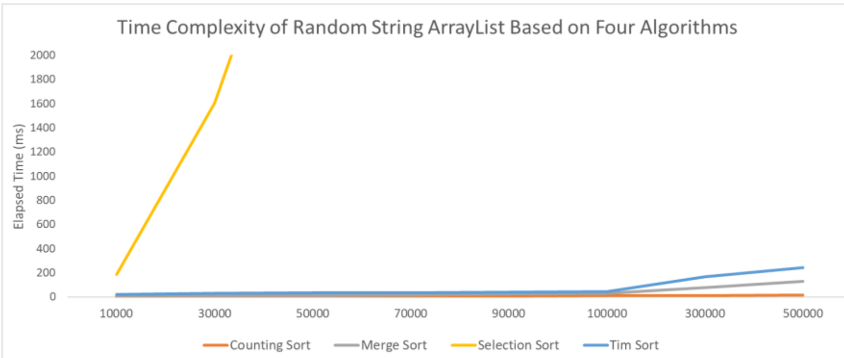
This graph illustrates the time complexity of the Tim-Sort algorithm applied to LinkedList containing numbers with different data characteristics. As shown by the graph, the orange line (sorted data) is relatively low, indicating that Tim-Sort performs efficiently on sorted data, with a time complexity of O(n). Other data conditions especially negative number LinkedList possess time complexity of O(n log n).

The time complexity for Tim-sort algorithm differs in various cases. For the **best case** this algorithm produces **O(n)** which occurs when all the data is already sorted, while for the **average and worst-case** it produces **O(n log(n))**, which occurs when the data is unsorted or reversed (Hanafi et al., 2022).

Result

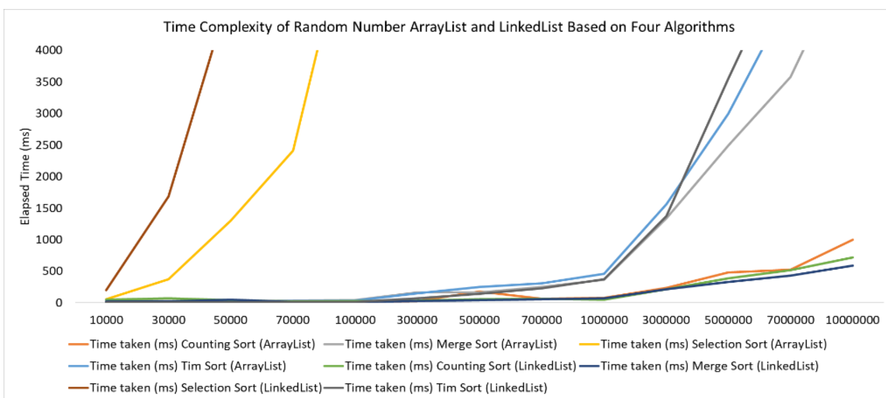
Noted that some of the result exceeded one hour, hence, those that exceeded were recorded as '-' since the program was terminated.

Design 1: Data Size



From the graph, it is evident that as the data size increases, the time complexity also grows longer as more elements required to access. Notably, each sorting algorithm demonstrates a different time complexity. Selection Sort, Tim Sort, Merge Sort and Counting Sort perform as $O(n^2)$, $O(n \log n)$, $O(n \log n)$ and $O(n)$ respectively. Time complexity of the four algorithms based on different input size is ranked as follow: **Counting Sort< Merge Sort< Tim Sort< Selection Sort.**

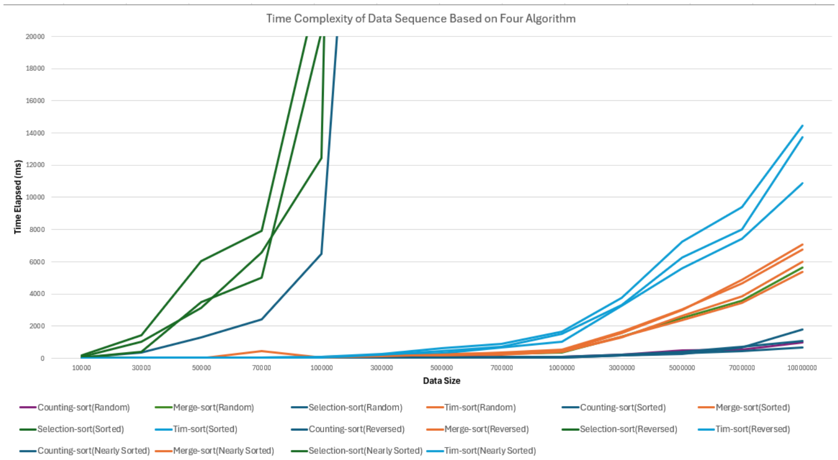
Design 2: Type of Lists



Based on the graph, ArrayList and LinkedList with random number sequences are plotted. It was noticed that the Selection sort has the highest time complexity while the merge sort has the lowest time complexity. The time complexity of the four algorithms based on different input size of **LinkedList** is ranked as follow**: Merge Sort< Counting Sort< Tim Sort< Selection Sort.** In contrast, the time complexity of the four algorithms based on different input size of **ArrayList** is ranked as follow: **Counting Sort< Merge Sort< Tim Sort< Selection Sort.** However, an interesting phenomenon was observed as the curve of Counting Sort of random numbers in ArrayList (red line) displayed small fluctuations, which is an unusual pattern. It can be explained due to the **java enhancement system function** as the enhancement of runtime efficiency could lead to variations in sorting time when handling large datasets. Additionally, it was found that the LinkedList of all four algorithms is more efficient than ArrayList of all four algorithms as the graph shows the time taken for LinkedList to be sorted is shorter. This is due to the **Node-based structure in LinkedList** allowing faster access and manipulation of elements compared to ArrayList. Another interesting perspective was found. The ArrayList of Tim Sort being sorted is performs better when data size is small while LinkedList of Tim

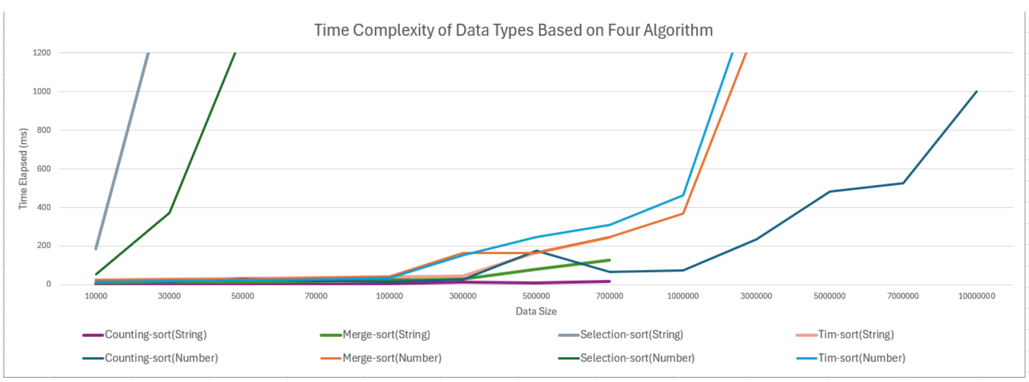
Sort performs better when data size is large. It can be said that the lesser memory overhead when dealing with large datasets in LinkedList is more suitable for Tim Sort.

Design 3: Data Sequence



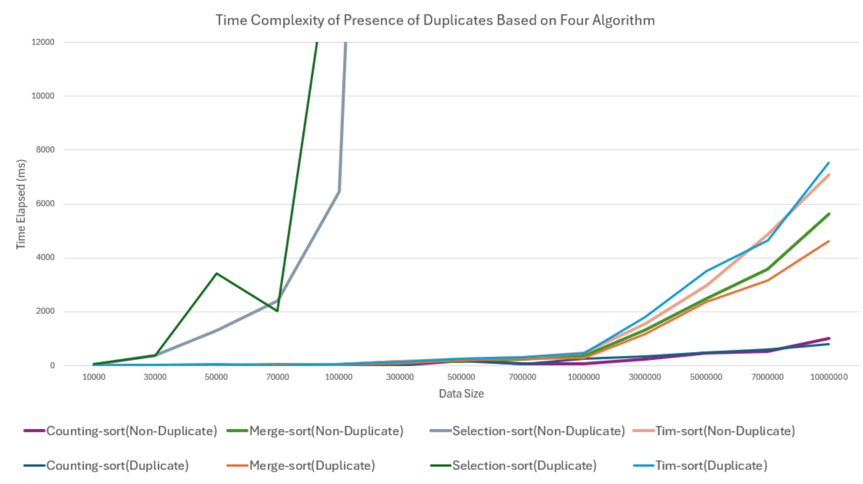
From the graph above, it is evident that the performance of Selection-Sort, Counting-Sort, Merge-Sort, and Tim-Sort algorithms varies with different data arrangements: random, sorted, reversed, and nearly sorted. In this design, Counting-Sort performance is the best compare with the other three algorithms in every data sequence. This shows the good performance with **O(N + K)** which is unaffected by data sequence in the sorting process. In contrast with the Counting-Sort, Selection-Sort has the worst performance in the sorting process. It consistently scans the entire unsorted portion to find the minimum element regardless of the data arrangement. This cause the time complexity of the Selection-Sort will be **O(N²)**. The time complexity of the four algorithms when applied to data sequence: **Counting-Sort < Merge-Sort < Tim-Sort < Selection-Sort.**

Design 4: Data Type



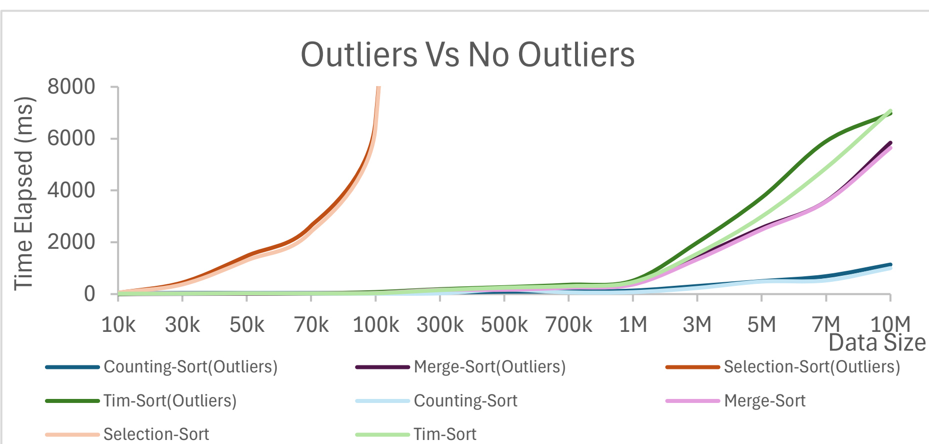
The graph above shows that when it comes to different data types, four algorithms will also have different performance. In this design, Counting-Sort performs best in both String and Number types while Selection-Sort performs badly in these data types. In **String** data types**, Counting-Sort and Merge-Sort** perform generally better than Number data types, while **Selection-Sort and Tim-Sort** perform better in **Number** data types instead of String data types. Numerical data often benefits the Tim-Sort adaptive nature make it can handle quickly with the number types as it can make use of numeric comparisons. In the Selection-Sort, numerical data is consistent and predictable making the comparison and swaps straightforward. The time complexity of the four algorithms when applied to String and Number data types: **Counting-Sort < Merge-Sort < Tim-Sort < Selection-Sort.**

Design 5: Duplicates



From the graph above, we can notice that there would be a positive effect on Counting-Sort algorithm when duplication is being introduced. This is because the Counting-Sort no need to perform repeated comparisons and it just counts the occurrences of each value and builds the sorted array based on those counts. For the Tim-Sort and Merge-Sort, it does not have any significant effect as both of them can handle duplication effectively as it does not increase the workload of these two sorting algorithms. While it comes to Selection-Sort, it has a negative impact when the data size increase. This might be due to the redundant comparisons and swaps that offer no real advantage in progressing the sorting. In contrast with non-duplicate data, every comparison brings this sorting closer to the final sorted state and lead it to slightly better performance. The time complexity of the four algorithms when applied to duplicate-data: **Counting-Sort < Merge-Sort < Tim-Sort < Selection-Sort.**

Design 6: Presence of Outlier

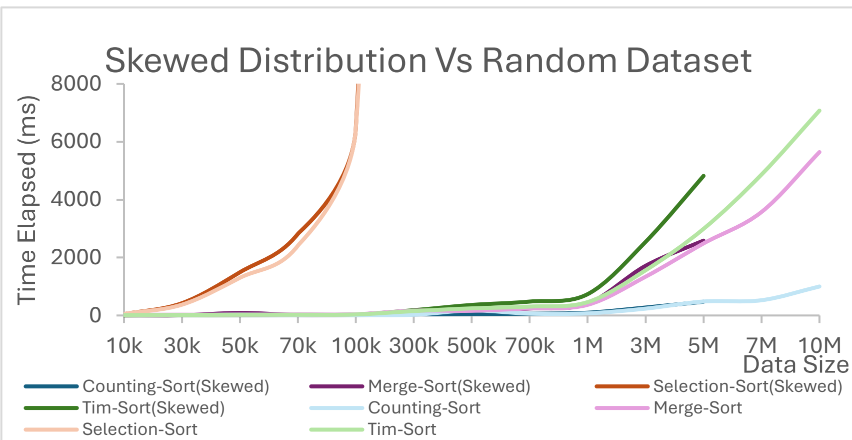


In this experiment design, the presence of outliers had a significant impact on the performance of the sorting algorithms. The **Merge Sort** and **Tim Sort** algorithms showed a noticeable increase in sorting time when outliers were introduced. The graph shows that **Merge Sort (with outliers)** consistently took longer as the data size increased, with the curve showing a steeper rise compared to the scenario without outliers.

This can be attributed to the recursive nature of Merge Sort, which has to split the array and merge it back together, causing outliers to have a disproportionate effect on the merging process.

Interestingly, **Counting Sort** remained unaffected by outliers. Since Counting Sort does not perform element comparisons, the presence of outliers does not significantly alter the sorting process. As the graph illustrates, the performance curve of **Counting Sort (with outliers)** closely mirrors that of **Counting Sort (no outliers)**. On the other hand, **Selection Sort** was the most negatively impacted by outliers. Its already high time complexity was exacerbated by the presence of outliers, particularly as the data size increased. This is likely because Selection Sort always searches for the minimum value in each pass, and outliers can make this process slower and less efficient. Overall, the time complexity of the four algorithms for datasets with outliers, **Counting Sort < Tim Sort < Merge Sort < Selection Sort.**

Design 7: Data Distribution

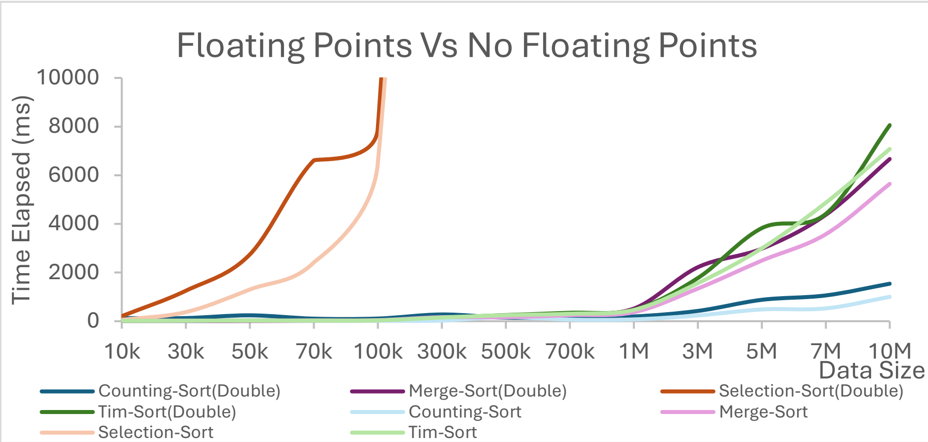


In this experiment design, the performance of each algorithm was compared between a skewed data distribution and a random dataset. The graph demonstrates that **Counting Sort** and **Merge Sort** performed relatively well in skewed distribution, with **Merge Sort** showing a slight performance degradation in skewed. The reason for this is that **Merge Sort** divides the list into smaller sub lists, and even when the data is skewed, the recursive division helps maintain its time efficiency.

Counting Sort also handled the skewed distribution effectively. Counting Sort's performance is largely unaffected by this skew because it only needs to count how many elements fall into each value, regardless of where the bulk of the values lie. However, **Selection Sort** and **Tim Sort** showed significant degradation in performance with skewed distributions. This is because **Selection Sort** compares elements in a brute-force manner, and skewed distributions lead to longer comparisons, particularly when the majority of elements are clustered around one value. **Tim Sort's** performance typically benefits from data that is more evenly distributed or partially sorted. In a skewed distribution, with most values clustered in one part of the range, the sorting process becomes less efficient because unbalanced data clusters require more comparisons and movements.

In summary, the time complexity of the four algorithms for datasets with skewed distribution, **Counting Sort < Merge Sort < Tim Sort < Selection Sort.**

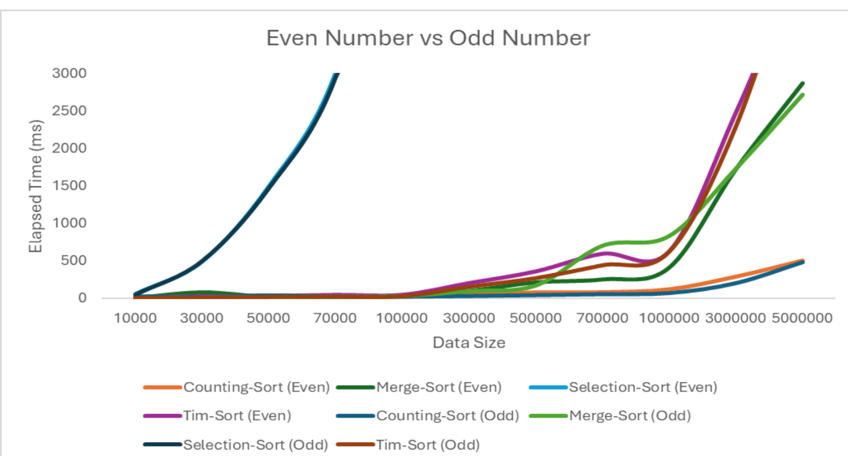
Design 8: Floating-point Data



In this experiment design, the algorithms were tested on datasets with floating-point numbers, and the performance was compared to datasets containing only integer values. The results from the graph show that **Merge Sort** and **Tim Sort** took longer when sorting floating-point data compared to integer data. This increase in sorting time can be attributed to the fact that comparisons between floating-point numbers are more computationally expensive than comparisons between integers, due to the precision required in floating-point arithmetic.

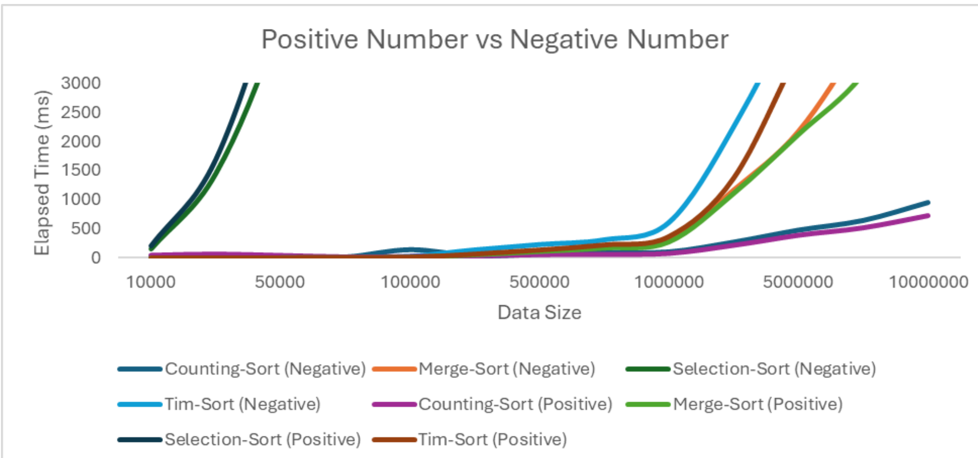
Selection Sort was particularly slow when handling floating-point data. The brute-force nature of **Selection Sort** amplifies the effect of increased comparison time with floating-point numbers, resulting in longer execution times across all data sizes. Interestingly, **Counting Sort** performed well with floating-point data. This was expected because **Counting Sort** is primarily designed for number sorting and relies on creating a count array based on the range of the numbers. Since it only counts the frequency of each floating-point numbers, **Counting Sort** is having only slight increase in sorting time for floating-point datasets. Overall, the time complexity of the four algorithms for datasets with floating-point numbers, **Counting Sort < Merge Sort < Tim Sort < Selection Sort.**

Design 9: Even and Odd Number



From the graph above, it is evident that the **Selection-Sort, Counting-Sort and Tim-Sort algorithms perform slightly better on odd numbers**, whereas the **Merge-Sort algorithm is more efficient on even numbers**. Merge-Sort algorithm maintains a relatively stable performance across different data sizes, with a slight edge in efficiency on even datasets, making it a strong choice for larger odd or even number datasets. Tim-Sort algorithm shows consistent performance similar to Merge-Sort algorithm, with a slight improvement on odd number dataset. On the other hand, Selection-Sort algorithm is overall the least efficient algorithm with a time complexity of $O(n^2)$. Although the efficiency of Counting-Sort algorithm decreases with larger datasets, it remains the most efficient algorithm overall, with a time complexity of O(n+k). The time complexity of the four algorithms when applied to even or odd data: **Counting-Sort < Merge-Sort < Tim-Sort < Selection-Sort.**

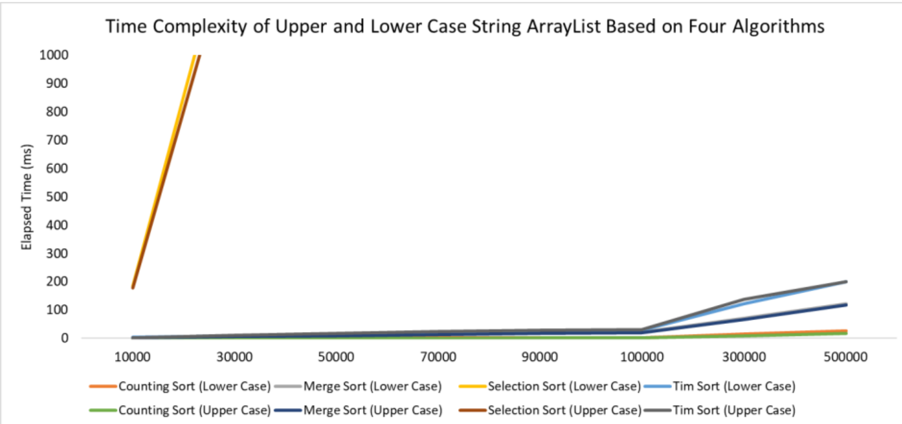
Design 10: Positive and Negative Numbers



The graph indicates that **all the sorting algorithms perform more efficiently on dataset of positive number** compared to those of negative numbers. As stated by Idrizi, F. et al. (2017), Counting-Sort algorithm is only appropriate for positive numbers because it becomes extremely complex, particularly when counting negative numbers. In general, the input of Counting-Sort algorithm consists of a collection of n items, where each has a non-negative integer key whose maximum value is at most k. However, among the algorithms, Counting-Sort algorithm proves to be the most efficient, exhibiting a time complexity of O(n+k).

Merge-Sort and Tim-Sort are comparison-based algorithms with a time complexity of O(n log n), and they maintain stable performance regardless of whether the dataset contains positive or negative numbers. However, Merge-Sort still perform slightly better than Tim-Sort due to its straightforward divide-and-conquer approach, which is less dependent on data characteristics. Selection-Sort, the least efficient algorithm with a time complexity of $O(n^2)$, shows a consistent pattern of being the slowest, regardless of the sign of the numbers. Its inefficiency stems from its repetitive comparisons and swaps, which are equally costly for both positive and negative numbers. The time complexity of the four algorithms when applied to positive or negative numbers: **Counting-Sort < Merge-Sort < Tim-Sort < Selection-Sort.**

Design 11: Upper- and Lower-Case Words



According to the graph, it denotes the time complexity of upper- and lower-case String ArrayList based on the four different algorithms. It was noticed that the result for each algorithm is almost the same, showing only slightly different. The time complexity of the four algorithms based on different input size of **ArrayList** is ranked as follow**: Merge-Sort< Counting-Sort< Tim-Sort< Selection-Sort.** Overall, the lower-case String has a shorter elapsed time to sort the Strings compared to upper-case String for the algorithms except for Tim-Sort. This is because, based on the **Unicode sequence**, the lower-case characters come before upper-case characters, allowing more efficient sorting arrangements.

Conclusion

From the 11 experimental designs, it was realized that Counting Sort outperformed among the four sorting algorithms as it is non-comparison-based sorting method. Generally, Counting Sort is efficient in sorting non-negative integers as well as the Strings. This is closely related to its ability of sorting data by mapping the input values directly to their corresponding locations in the output array since element-by-element comparisons are not included. In contrast, Selection Sort was the most inefficient algorithm as it is time consuming during the sorting process. Selecting the smallest element from the huge dataset during each iteration and exchanging with the position inside the list takes plenty of time. Meanwhile, Merge Sort and Tim Sort maintained a stable performance. Tim Sort is a hybrid sorting algorithm that combines the best features of Merge Sort and Insertion Sort. It is highly efficient for real-world data, robust and adaptive sorting algorithm, making it the default choice in many programming environments. In conclusion, the performance the four algorithms can be ranked as **Counting-Sort > Merge-Sort > Tim-Sort > Selection-Sort.** Part of the result obtained is showcased in the Appendix.

Appendix

Experimental Design: Random String Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	3	10	187	23
30K	7	13	1605	28
50K	4	12	3898	34
70K	3	18	7592	37
90K	6	24	11850	40
100K	12	28	13157	46
300K	11	78	160045	165
500K	16	129	383704	245

Experimental Design: Sorted String Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	0	2	205	1
30K	1	6	1732	4
50K	3	10	4805	19
70K	3	16	9711	21
90K	5	20	15355	26
100K	6	21	16836	25
300K	16	69	151199	93
500K	22	124	427824	138

Experimental Design: Reversed String Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	0	2	314	3
30K	1	6	2416	7
50K	1	10	4602	14
70K	2	15	7577	21
90K	5	19	11713	27
100K	4	21	13117	31
300K	12	70	274131	135
500K	43	122	538650	247

Experimental Design: Nearly Sorted String Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	0	3	235	2
30K	1	8	2192	11
50K	3	14	5881	20
70K	5	21	10945	30
90K	5	28	17027	35
100K	5	30	19356	45
300K	18	99	310057	145
500K	33	174	881005	217

Experimental Design: String Array List Contains Duplicates

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	0	2	165	3
30K	0	6	1437	7
50K	1	10	3627	11
70K	2	14	7097	19
90K	1	19	11580	20
100K	2	21	12633	23
300K	10	69	140698	113
500K	14	120	343813	178

Experimental Design: Upper Case String Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	0	2	178	2
30K	1	6	1428	11
50K	1	10	5213	18
70K	1	14	8169	25
90K	2	18	12419	30
100K	2	21	15597	32
300K	10	67	153965	137
500K	17	117	526632	201

Experimental Design: Lower Case String Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	0	2	184	4
30K	2	6	1523	8
50K	2	11	5872	15
70K	2	15	8985	20
90K	3	19	13587	27
100K	2	22	17758	30
300K	15	71	172192	123
500K	26	122	579569	201

Experimental Design: Random Number Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	22	24	56	17
30K	15	28	374	20
50K	30	25	1302	24
70K	18	36	2412	26
100K	15	43	6465	35
300K	27	166	59107	153
500K	175	164	173036	247
700K	66	248	376851	308
1000K	75	369	690117	462
3000K	237	1341	-	1562
5000K	484	2490	-	2993
7000K	527	3575	-	4868
10000K	1002	5645	-	7079

Experimental Design: Nearly Sorted Number Linked List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	18	1	231	3
30K	18	3	2625	4
50K	18	7	6421	8
70K	28	10	13209	15
100K	23	15	37925	26
300K	239	58	749184	73
500K	85	103	2388899	130
700K	135	154	-	193
1000K	135	231	-	759
3000K	371	842	-	1559
5000K	757	2006	-	2719
7000K	1060	2449	-	4102
10000K	1367	3910	-	5001

Experimental Design: Number Array List Contains Duplicates

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	13	7	53	15
30K	18	16	359	39
50K	27	25	3431	48
70K	18	22	2036	16
100K	26	34	17498	47
300K	43	100	111457	142
500K	199	196	307826	266
700K	70	245	519008	313
1000K	263	304	991331	445
3000K	334	1159	-	1801
5000K	496	2357	-	3498
7000K	607	3159	-	4662
10000K	813	4629	-	7542

Experimental Design: Number Linked List Contains Outliers

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	30	1	142	2
30K	37	4	1236	8
50K	71	7	3643	10
70K	33	10	7192	14
100K	31	15	14773	22
300K	87	57	144895	79
500K	152	110	474420	139
700K	80	168	-	239
1000K	134	274	-	503
3000K	339	1119	-	1947
5000K	525	2058	-	3312
7000K	678	3518	-	6503
10000K	937	6007	-	11724

Experimental Design: Skewed Number Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	12	2	47	3
30K	15	7	425	9
50K	14	89	1494	17
70K	16	19	2819	24
100K	18	27	6470	35
300K	36	103	70688	184
500K	52	194	182034	365

Experimental Design: Skewed Number Linked List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	29	1	192	2
30K	15	4	1681	5
50K	13	7	4894	11
70K	21	10	9223	17
100K	19	14	19387	36
300K	44	59	184806	105
500K	65	113	570455	194
700K	190	177	-	299
1000K	114	292	-	497
3000K	412	1136	-	1686
5000K	541	2115	-	4394

Experimental Design: Negative Number Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	30	2	63	17
30K	19	7	436	38
50K	34	12	1238	50
70K	11	18	2468	51
100K	14	27	5222	70
300K	31	563	55095	124
500K	48	182	167734	247
700K	75	271	439923	279
1000K	85	557	1394980	396
3000K	243	1573	-	1559
5000K	392	2774	-	3895
7000K	779	3945	-	5141
10000K	734	5790	-	7589

Experimental Design: Negative Number Linked List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	33	1	152	1
30K	14	4	1449	2
50K	24	7	4127	15
70K	14	10	8176	17
100K	141	16	16952	31
300K	36	64	273207	141
500K	66	121	1160627	233
700K	92	185	2780859	314
1000K	103	301	-	613
3000K	272	1169	-	2257
5000K	480	2175	-	4518
7000K	648	3836	-	7817
10000K	958	5838	-	12791

Experimental Design: Even Number Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	15	2	58	16
30K	17	78	490	31
50K	19	15	1491	32
70K	23	19	3000	47
100K	24	28	6234	48
300K	52	97	72104	204
500K	79	211	372547	360
700K	81	250	1000134	596
1000K	121	408	2501494	627
3000K	290	1723	-	2479
5000K	505	2870	-	4551

Experimental Design: Double Number Array List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	129	17	209	15
30K	130	8	1252	24
50K	243	14	2741	29
70K	106	21	6606	31
100K	110	30	7904	39
300K	277	101	77717	144
500K	147	205	365564	253
700K	157	318	1363518	348
1000K	195	518	4212621	470
3000K	423	2218	-	1773
5000K	880	2981	-	3826
7000K	1060	4373	-	4409
10000K	1542	6666	-	8058

Experimental Design: Double Number Linked List

Elapsed Time(ms) of Sorting Algorithms

Data size	Counting-Sort	Merge-Sort	Selection-Sort	Tim-Sort
10K	225	4	157	0
30K	109	8	1495	7
50K	110	13	4333	15
70K	102	11	8838	16
100K	111	18	17599	32
300K	329	70	323119	94
500K	197	133	1615395	237
700K	174	216	3417386	477
1000K	205	326	5225032	630
3000K	956	1270	-	2597
5000K	861	2878	-	3687
7000K	865	4139	-	7145
10000K	4925	6541	-	11235

References

Abdon, A. L. M., Ang, J. C. M., Domingo, K. N., & Gutierrez, J. E. (2024). Automated Windows-Based Timelining Tool for Memory and Disk Image Analysis: Leveraging Timsort Algorithm with WinPmem, FTK Imager, Volatility 3 and The Sleuth Kit.

Baeldung. (n.d.). *Straight selection sort*. Baeldung. Retrieved from https://www.baeldung.com/cs/straight-selection-sort

Counting Sort (With Code). (n.d.). Www.programiz.com. https://www.programiz.com/dsa/counting-sort

GeeksforGeeks. (2024, September 4). *Selection sort algorithm*. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/selection-sort-algorithm-2/

Hanafi, M. R., Faadhilah, M. A., Putra, M. T. D., & Pradeka, D. (2022). Comparison Analysis of Bubble Sort Algorithm with Tim Sort Algorithm Sorting Against the Amount of Data. *Journal of Computer Engineering, Electronics and Information Technology*, *1*(1), 29-38.

Idrizi, F., Rustemi, A., & Dalipi, F. (2017, June). A new modified sorting algorithm: a comparison with state of the art. In *2017 6th Mediterranean Conference on Embedded Computing (MECO)* (pp. 1-6). IEEE.

Lobo, J., & Kuwelkar, S. (2020, July 1). *Performance Analysis of Merge Sort Algorithms*. IEEE Xplore. https://doi.org/10.1109/ICESC48915.2020.9155623

P. Asha Rani, Chinnaiah, M. C., Kumari, A., G. Preethika, & Reddy, Y. P. (2023). *HLS Based Design and Optimization of Merge Sort Algorithm for High Performance Computing*. https://doi.org/10.1109/incet57972.2023.10170313

Tyagi, A., & Ahlawat, A. K. (2023, April 1). *A New Optimized Version of Merge Sort*. IEEE Xplore. https://doi.org/10.1109/ICETET-SIP58143.2023.10151579

Zhang, Y., Zhao, Y., & Sanan, D. (2018). A verified timsort C implementation in isabelle/hol. *arXiv preprint arXiv:1812.03318*.