Lab 3 Analysis

EN.605.202
Data Structures
Summer 2022

Jose Márquez Jaramillo

Engineering and Applied Science Programs for Professionals
Johns Hopkins University
United States
August 23th, 2022

Abstract

In this lab several implementations of Merge Sorts and Quick Sorts are compared in terms of comparisons, replacements, and execution times. The benefits and tradeoffs of the different sorting mechanisms are discussed both in theory as well as in practice (through the use of cases in 33 different file sizes and orders.

Contents

1 Sorting algorithms

In order to compare the performance of the Quicksort and Mergesort, several different variations of each have been implemented, I will begin the analysis with some preliminaries of each while also providing some of their main characteristics. As we will see in the final comparison analysis, these characteristics have important implications in the number of comparisons, replacements, and the time that each algorithm takes to finish the sort [1].

1.1 Insertion Sort

Insertion sort maintains a sorted sublist in the lower position of the list. Each new item is then ”inserted” back into the previous sublist such that the sorted sublist is one item larger. This procedure generally is accomplished by using a nested loop traversing the list. This causes insertion sort to be $O(n^2)$. The maximum number of comparisons for an insertion sort is the sum of the first $n-1$ integers. This is again $O(n^2)$; however, in the best case, only one comparison needs to be done on each pass. The best case, therefore, occurs with a list that is already sorted [2].

1.2 About divide and conquer algorithms

By definition, both Merge and Quick sort can be written while making recursive calls to themselves. This can therefore be described using a recurrence equation. This equation roughly describes the overall running time on a size problem n in terms of the running time on smaller inputs. In particular, with these two types of sorting strategies (Merge and Quick sort), the algorithms do the following:

- **Divide:** The divide step just computes the middle of the subarray, which takes constant time.
- **Conquer:** The problem is recursively solved by solving two problems, each of size n/2.
- **Combine:** The algorithm finishes by combining the recursive solutions.

In general, the ”master theorem” indicates that these types of algorithms are $\Theta(n \log_2 n)$. Because the logarithm functions grow more slowly than any linear function, for large enough inputs, these algorithms perform better than algorithms like insertion sort [1].

1.3 Merge Sort

As previously described, Merge Sort is a recursive algorithm that continually splits in half. If the list is empty or has one item, it is sorted by definition (base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list [2].

While sorting numbers, merge sort has an average worst-case performance of $O(n \log_2 n)$ [3]. Because of its divide and conquer implementation, this relationship is maintained for the best, average, and worst cases. However, there are possible improvements to Merge Sort which come in the ways of implementing through linked-lists or by taking advantage of already sorted pieces of data (as in the case of Natural Merge Sort).

1.3.1 Natural Merge Sort

Natural Merge Sort is an optimization of Merge Sort: it identifies presorted areas (”runs”) in the input data and merges them. This prevents the unnecessary further dividing and merging of presorted sub-sequences. Input elements sorted entirely in ascending order are therefore sorted in $O(n)$ instead of $O(n \log_2 n)$ as in the best case. This can also generate improvements while encountering sorted data in the average case (random files or files with consecutive repeating elements) [4].

Natural Merge Sort has the principal following differences to regular Merge Sort:

- Instead of subarrays, the entire original array and the positions of the areas to be merged are passed to the method.

- Instead of returning a new array, the target array is also passed to the method for being populated.

1.3.2 Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as Quick Sort) perform poorly while making others (like Heap Sort) practically impossible [5] [6].

1.4 Quick Sort

Quick Sort uses divide and conquer to gain the same advantages as Merge Sort while not using additional storage (not under the linked list implementation). As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished. Quick Sort makes use of the partition function given a pivot value. The position where the pivot value belongs in the final sorted list is commonly called the split point. The partition process finds the split point and moves other items to the appropriate side of the list (higher elements to the right and lower elements to the left) of the pivot value [2].

For a list of length n, if the partition always occurs in the middle of the list, there will be $\log_2 n$ divisions. To find the split point, each of the n items needs to be checked against the pivot value. The result is $n \log_2 n$. This is, therefore, the best case for Quick Sort, and it occurs when the list is in a completely random manner. This leads us to the worst time complexity. If the pivot element is always the smallest or largest element of the list, the list will not be partitioned into approximately two equally sized lists. The ”left” partition will have 0 elements, while the ”right” side partition will have $n - 1$ elements or all those larger than the pivot value [7]. This complication causes Quick Sort to have a worst-case time complexity of $O(n^2)$ (the same as the previously mentioned insertion sort). In practice, the worst-case scenario occurs when attempting to sort a pre-sorted list (either in ascending or descending order). To remedy these two cases, one could select a different pivot value.

1.4.1 Hybrid Quick Sort

Because of its recursive nature, Quick Sort tends to carry extra overhead and memory usage for smaller lists than Insertion sort [8]. A possible optimization method is to use another non-recursive sorting method when the number of elements in the list is below some threshold (in our case, 50 and 100 elements). Therefore, when the partition falls in size within the threshold, utilize insertion sort to finalize the sort. At smaller list sizes, insertion sort performs fewer replacements and comparisons [9].

1.4.2 Median of three Quick Sort

Quicksort is slowest when the pivot is always the smallest or largest possible value. The best possible pivot is the median of the segment being sorted. That median can be calculated and used, but the calculation is too slow for practical implementation [10]. Instead, one generally uses the median of three values. In our implementation, those three values will be the first, last, and middle items of the list. From here:

1. Take the median element of three,
2. Swap the median element with the current pivot element (in our case, the first element),
3. and then proceed with the recursive partition, maintaining points 1 and 2.

2 Description of programs and methods

2.1 Generating files

Files are generated in the `generatefiles.py` program. This program simply uses loops to generate ascending and reverse order files, it also uses the `shuffle` method in the `random` library to generate the random file. In order to generate the repeating file, I generated 80% of the file size in ascending order, then generated the remaining 20% of the list size in ascending order. This essentially causes a repeat of 20% of the list. Finally the `shuffle` method in the `random` library is used to mix the lists.

2.2 Implementations of sorting algorithms

Most of the sorting algorithms follow the methods used in [2]. The other methods have been sourced in the comments of the functions.

2.3 On measuring comparisons, replacements, and time elapsed

As part of the Quick Sort methods, a helper `swap` is implemented. A replacement is counted whenever the `swap` function is called. Comparisons are counted whenever values are compared, usually this happened inside while loops to traverse the partitions.

Now, counting comparisons and replacements is a bit more conceptually challenging. Replacements for linked lists are counted whenever the next connection of a node is replaced with another node. On the other hand, a comparison occurs whenever the data value of a node is compared against the data value of another node.

2.4 Enhancements

- **Inclusion of extra lengths:** in an effort to add continuity to the file sizes and make the gaps more uniform, I have added the file sizes of 100, 250, and 500.
- **Use of Hybrid Quick Sort with Threshold 10:** I found in the literature that this was the most commonly used hybrid implementation for the Quick Sort. Therefore it was also implemented. In the analysis this implementation is found to be superior to both the 50 and 100 threshold hybrid implementations.

- **Use of traditional Merge Sort on a linked list:** as suggested in the lab. Comparing traditional Merge Sort to Natural Merge Sort truly highlighted the virtues of both and when one should be used over the other.
- **Measuring elapsed time:** as suggested in the lab instructions.
- **Use of Pandas and Microsoft Excel:** I decided to compile all of the information in a dictionary and then generate a Pandas DataFrame which is later exported to a CSV file. This CSV file is present in the Graphs folder of the Analysis folder of the lab.

3 Comparing the sorting algorithms

In order to visually compare the performance of the different sorting algorithms, I have created a series of similar graphs color coded by the length of the lists sorted. Blue tones correspond to smaller lists increasing in color intensity as size increases (that is 50,100,250,500) and green tones indicate larger size list increasing in color intensity as the list size increases (that is 1000, 2000, 5000, 10000). The idea is to be able to rapidly be able to identify the best and worst performing in terms of number of comparisons, number of replacements, and time of execution (in milliseconds). I will begin by comparing the Merge Sorts independently (Merge and Natural Merge Sorts) followed by the Quick Sorts (Regular Quick Sort corresponds to quicksort1, Hybrid Quick Sort with Threshold of 50 corresponds to quicksort2, Hybrid Quick Sort with Threshold of 100 corresponds to quicksort3, Median of Three Quick Sort corresponds to quicksort4, and Hybrid Quick Sort with Threshold of 10 corresponds to quicksort5).

3.1 Merge Sorts

3.1.1 Number of comparisons

For the number of comparisons for the Merge Sorts we can identify that the number of comparisons is does not change significantly for the different types of file orders (ascending, random, repeating, and reverse). In the case of order types in which there may be some already sorted elements as in the case of random and repeating files, we observe a significant improvement in the number of comparisons.

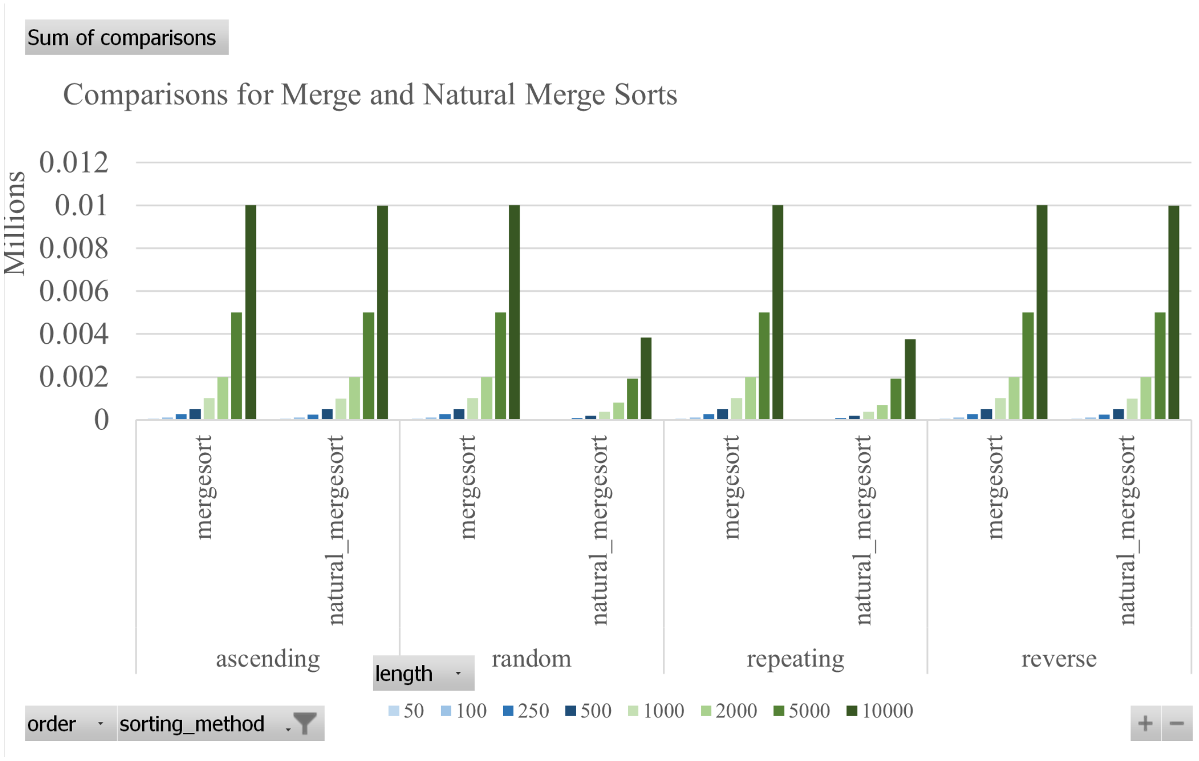

Figure 1: Comparisons for Merge Sorts

3.1.2 Number of replacements

When it comes to replacements, we see an increased number of replacements for the natural merge sorts, specially for those file types in which there could be some present order. This is to be expected since the numbers are shuffled and there are significantly more replacements to be executed. When excluding the cases of random and repeating file orders, we can definitively observe the advantages of using Natural Merge Sort on ordered data. Regular Merge Sort will execute the same number of replacements in relation to n, yet Natural Merge Sort will only generate merges and replacements when necessary. Therefore in the ascending and reverse file types we can identify a significant lower number of replacements.

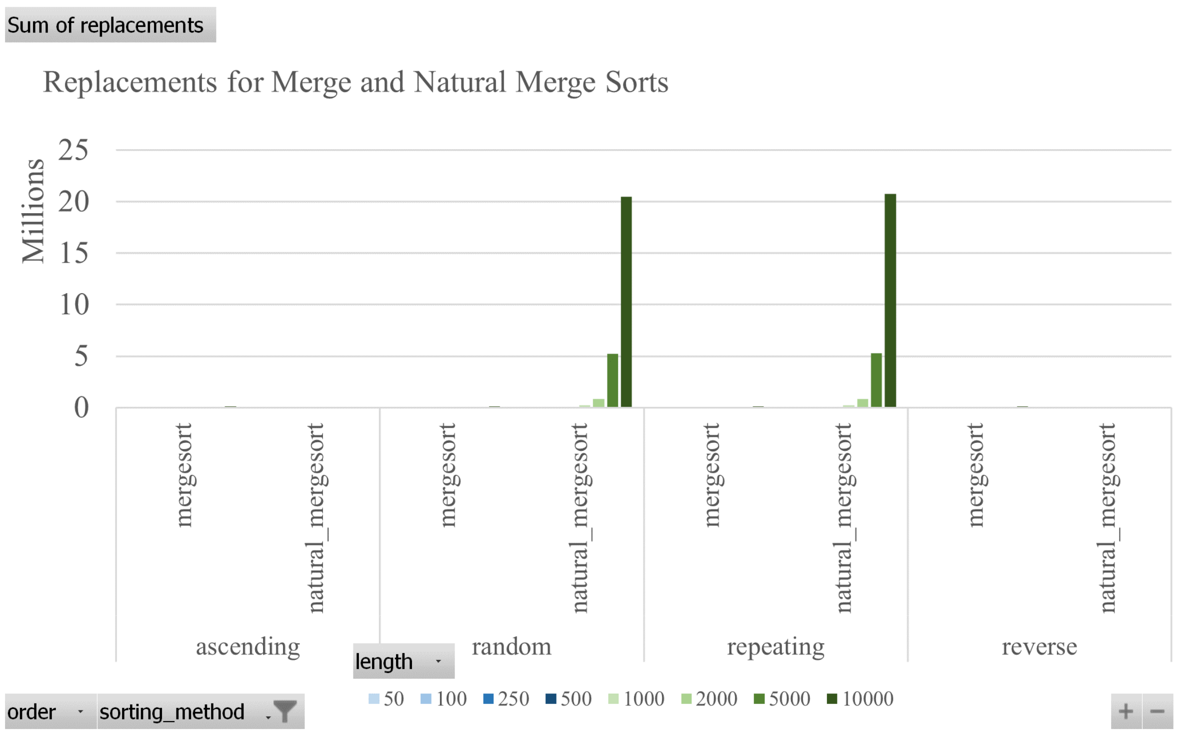

Figure 2: Replacements for Merge Sorts

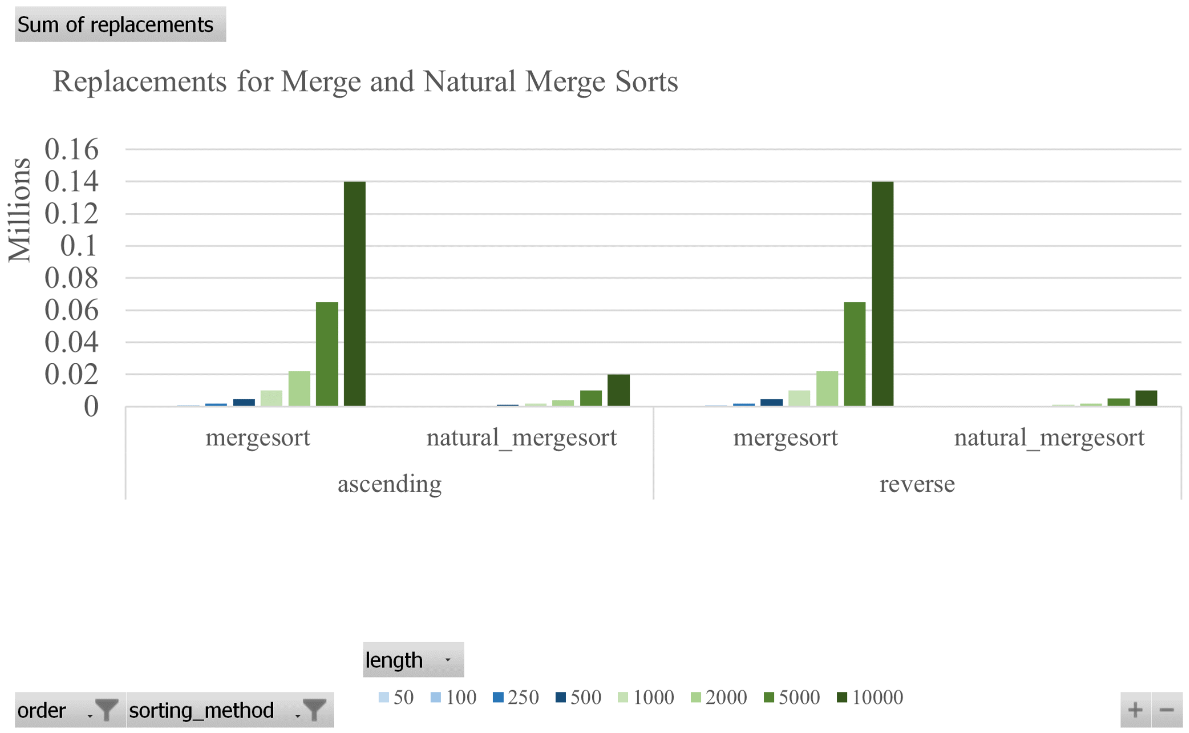

Figure 3: Replacements for Merge Sorts Excluding Random and Repeating

3.1.3 Execution times in milliseconds

Given that the number of replacements heavily dominates the level of comparisons in the cases of the random and repeating files for the Natural Merge Sort, the time these two take to execute is also significantly higher than the rest of algorithms. Excluding those two, we, once again can see the benefits of using Natural Merge Sort on data with some order.

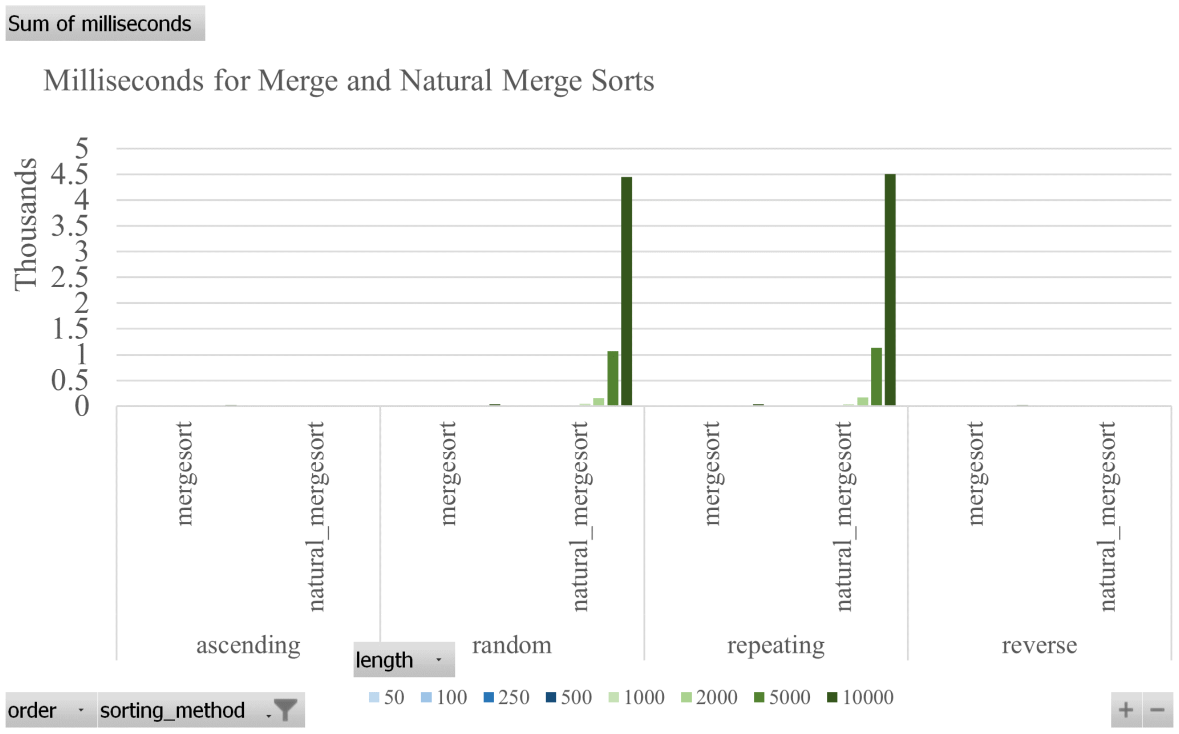

Figure 4: Milliseconds for Merge Sorts

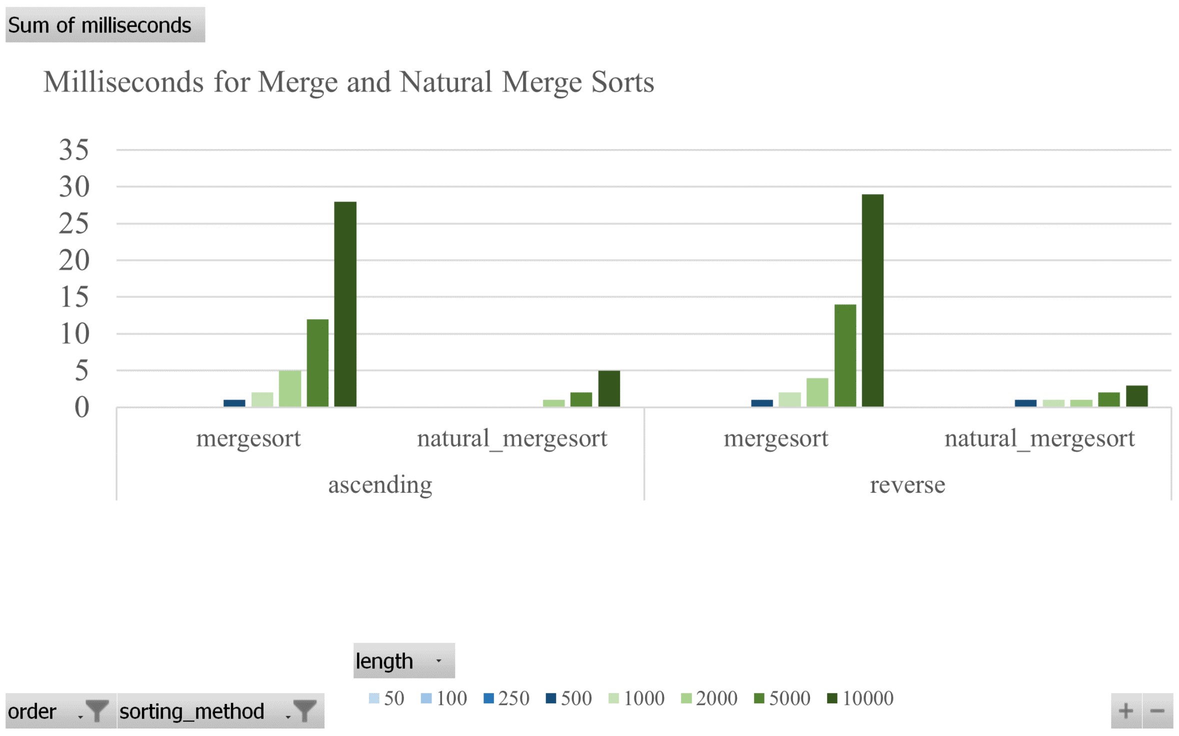

Figure 5: Milliseconds for Merge Sorts Excluding Random and Repeating

In conclusion, Natural Merge Sort behaves better than regular Merge Sort only when there might be some type of ordering present in the data. In other words, whenever there is some level of inherent ordering in the data Natural Merge Sort should be used. In the cases where there is randomness in the data, Merge Sort will outperform Natural Merge Sort.

3.2 Quick Sorts

We have previously established that the worst case for Quick Sort happens when the partitioning is heavily unbalanced (considerably more to the right or vice versa), this happens when the pivot element is either the smallest or largest element in the partition. Below we examine the numerical results for our samples. Recall that:

1. **quicksort1:** Regular Quick Sort
2. **quicksort2:** Hybrid Quick Sort with Threshold of 50
3. **quicksort3:** Hybrid Quick Sort with Threshold of 100
4. **quicksort4:** Median of Three Quick Sort
5. **quicksort5:** Hybrid Quick Sort with Threshold of 10

3.2.1 Quick Sort error of execution at worse cases

It is very important to note that in the very worse cases (ascending 5000, reverse 5000, ascending 10000, and reverse 10000), quicksort1 and quicksort4 caused the program to crash. This I believe is cause by a memory access error as the number of recursions needed to sort the lists exceeded the available resources in my laptop. The forthcoming results therefore need to take this into account and it is necessary to keep in mind that in the worst cases Hybrid Quick Sort methods were the only ones able to handle the sorting. This because as the partition sizes become reduced, the last threshold amount of elements are sorted by insertion sorting, reducing drastically the number of recursions needed.

3.2.2 Number of Comparisons

The charts below are very indicative of what was previously discussed in the preliminaries. It is also important to repeat that quicksort1 and quicksort4 were not able to handle the very edge worse cases. Given the amount of comparisons needed, the differences between the hybrid approaches are negligible at the worse cases.

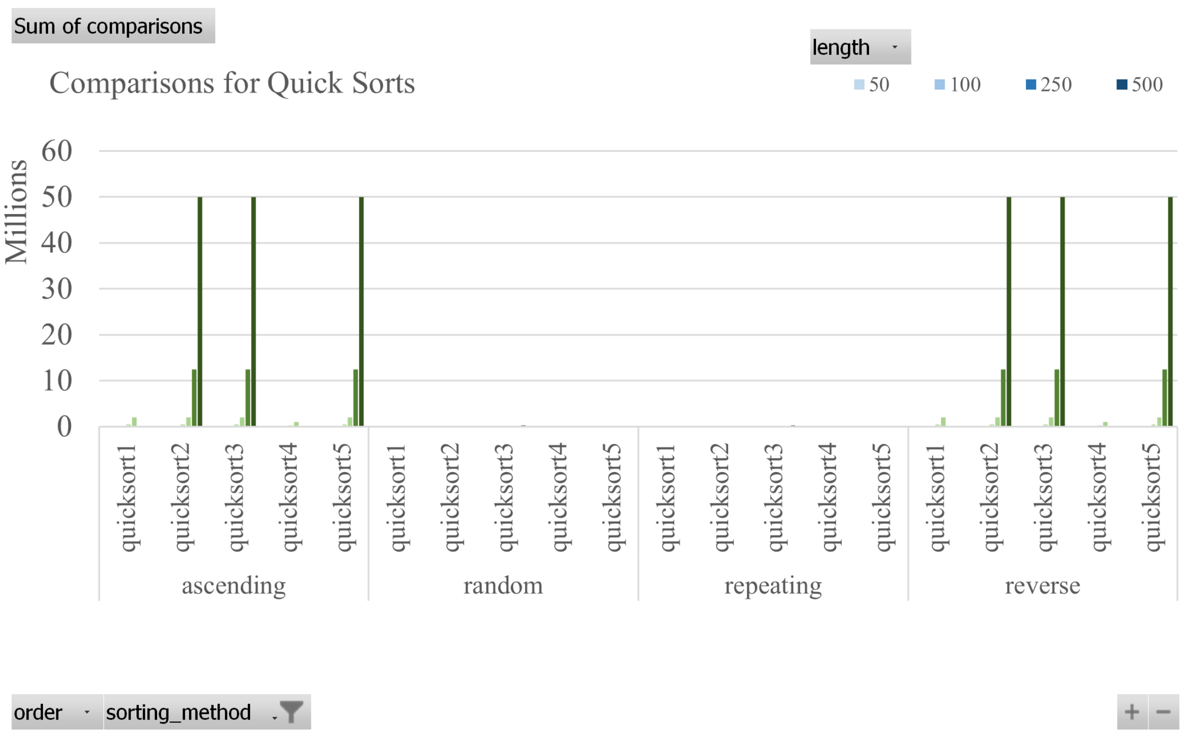

Figure 6: Comparisons for Quick Sort

At the best cases, that is where there is randomness on the data (random and repeating), the number of comparisons is considerably lower for quicksort4 (Median of Three Quick Sort) and quicksort5 (Hybrid Quick Sort with Threshold of 10).

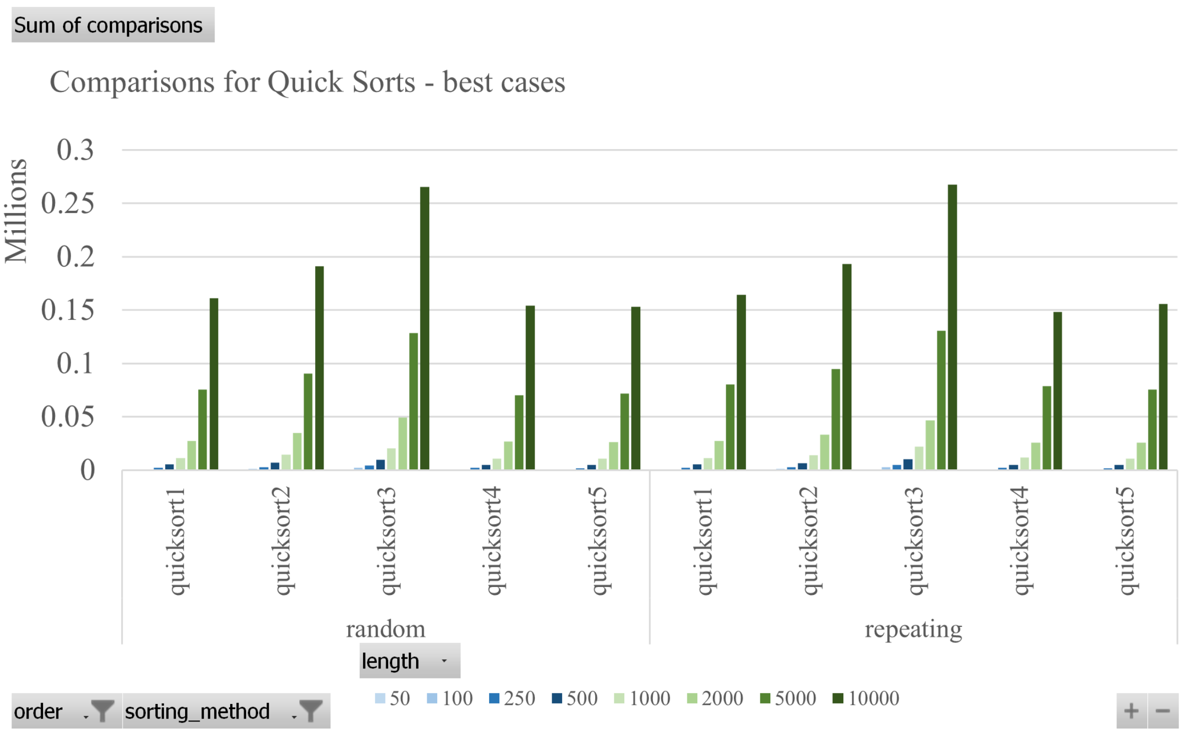

Figure 7: Comparisons for Quick Sort - Best Cases

3.2.3 Number of Replacements

Given that under the worse cases data is already in ascending or descending order, the number of replacements is much lower. Also notice that the number of comparisons strictly dominates (millions) over the number of replacements (thousands).

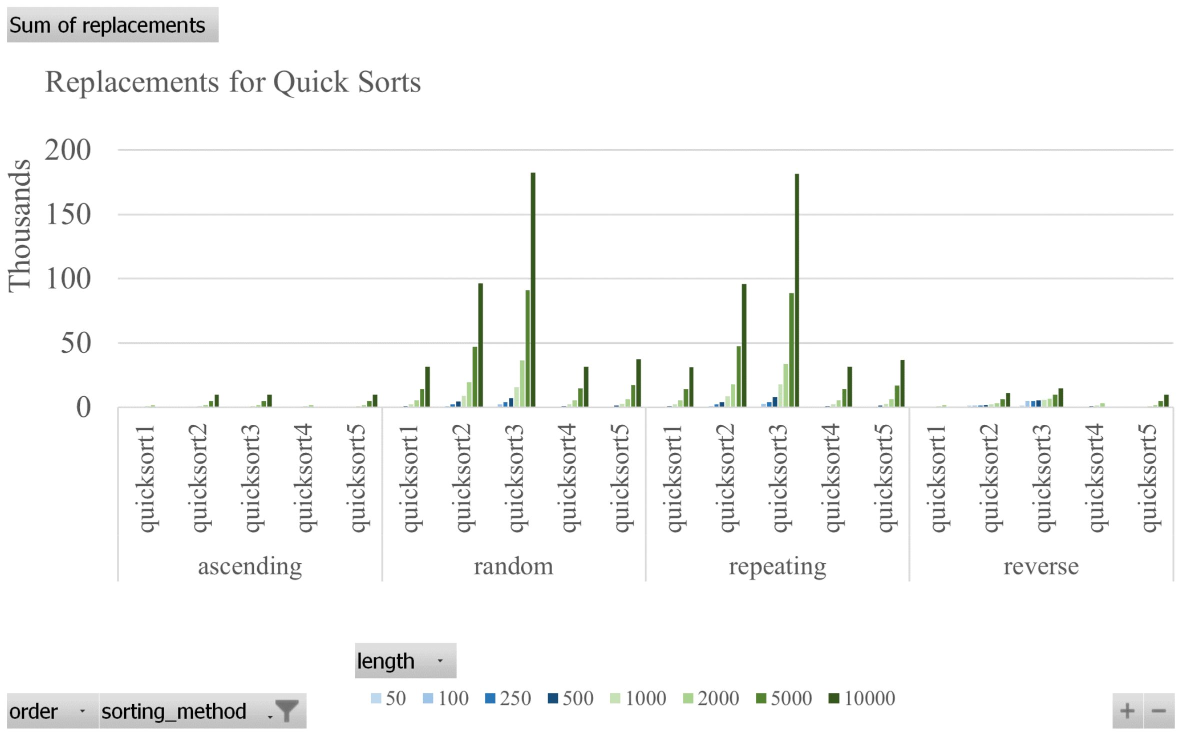

Figure 8: Replacements for Quick Sorts

Taking a look at the better cases, we can identify once again the benefits of quicksort4 (Median of Three Quicksort) and quicksort5 (Hybrid Quick Sort with Threshold of 10). We have previously indicated the benefits of the Median of Three Quick Sort but at this point we can indicate that for the characteristics at hand, quicksort5 outperforms quicksort2 and quicksort3 (or those Hybrid Quick Sorts with Thresholds of 50 and 100, respectively).

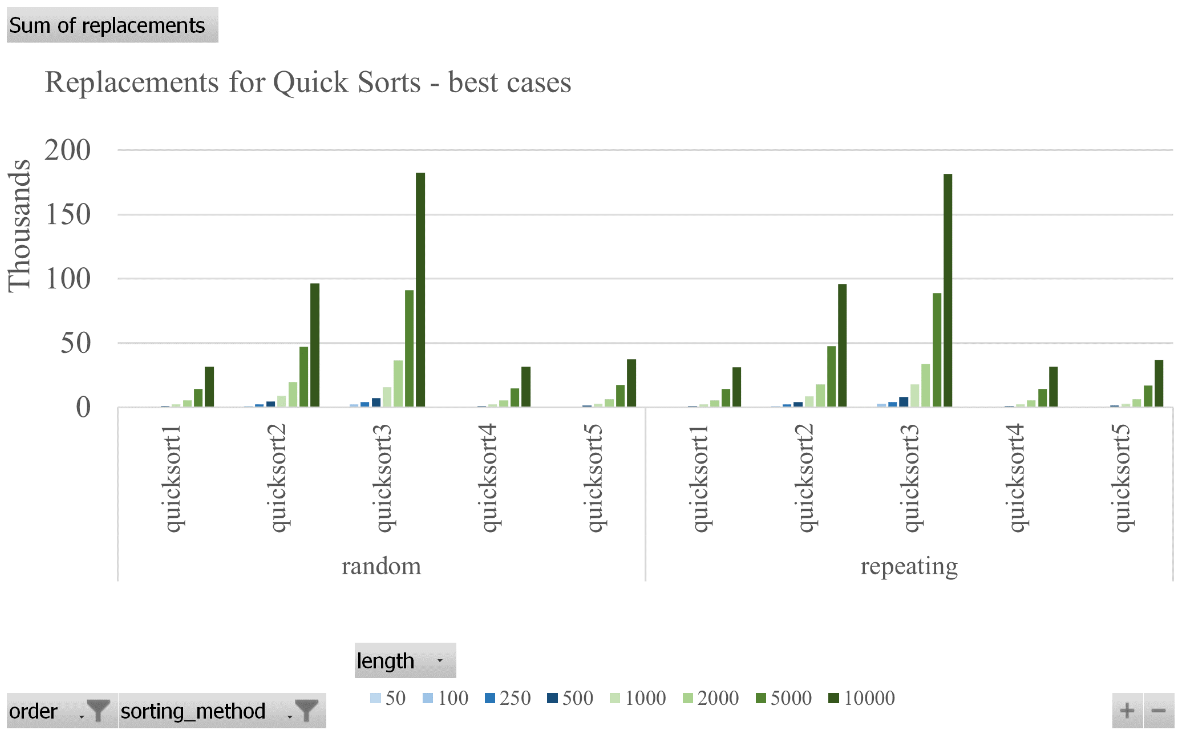

Figure 9: Replacements for Quick Sorts - best cases

3.2.4 Execution times in milliseconds

Given that quicksort4 and quicksort5 have less comparisons and less replacement levels, execution times are also better.

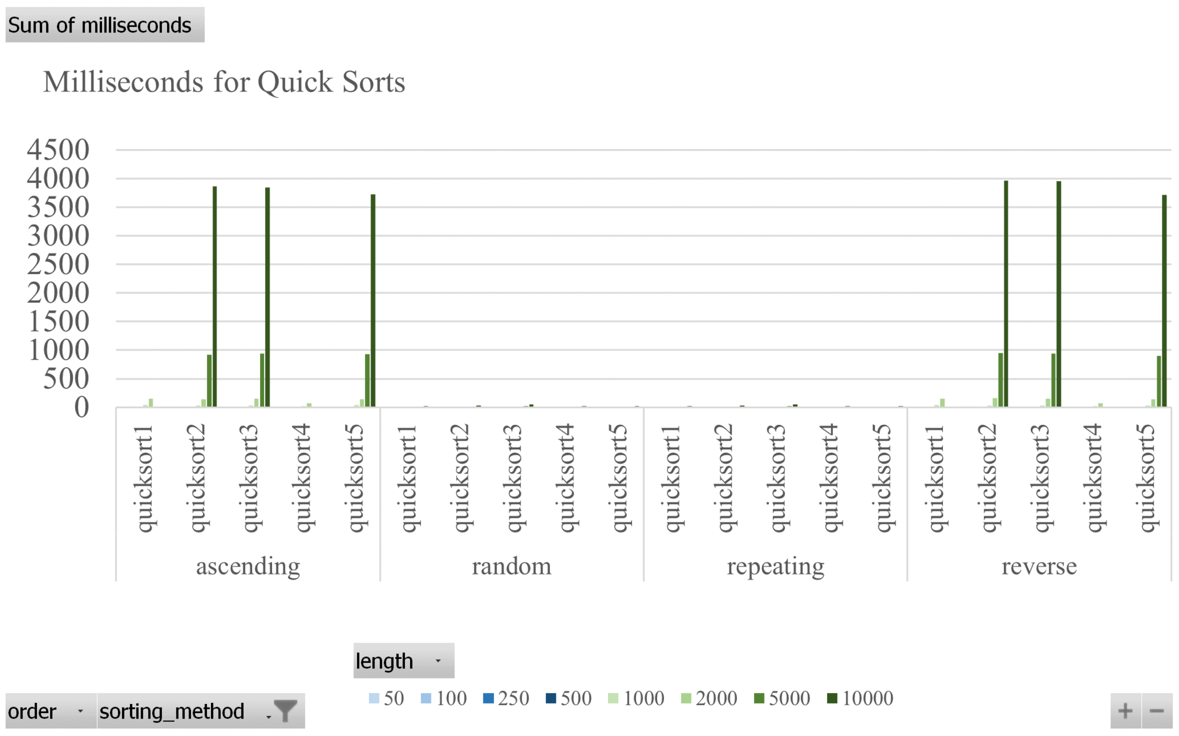

Figure 10: Milliseconds for Quick Sorts

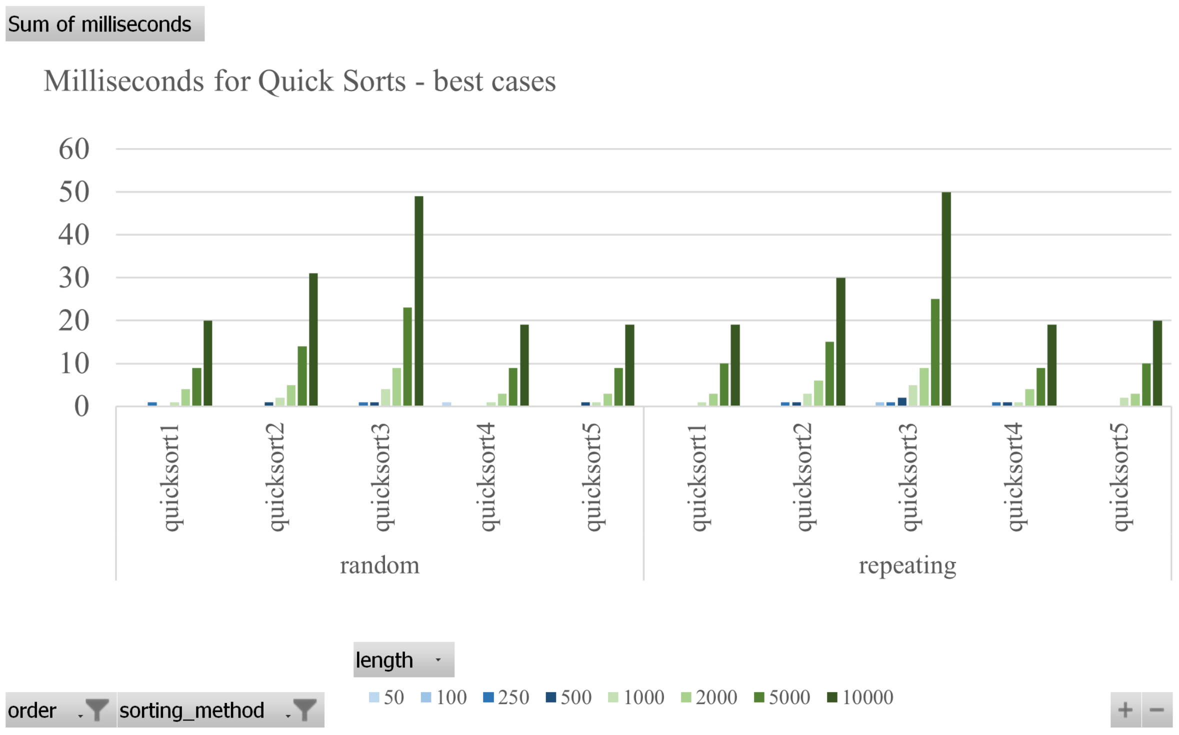

Figure 11: Milliseconds for Quick Sorts - best cases

4 Merge Sorts Vs. Quick Sorts 4 and 5

Taking the comparisons among the Merge Sorts and the best of the Quick Sorts, we can compare the best among the two groups, keeping in mind the conclusions we gathered from the Merge Sort analysis. It is important also to note the main difference in implementation between the Merge and Quick sort methods, which is the use of a linked list instead of a regular python list. This significantly improved the use of memory while sorting and reduced issues related to recursion loads on memory.

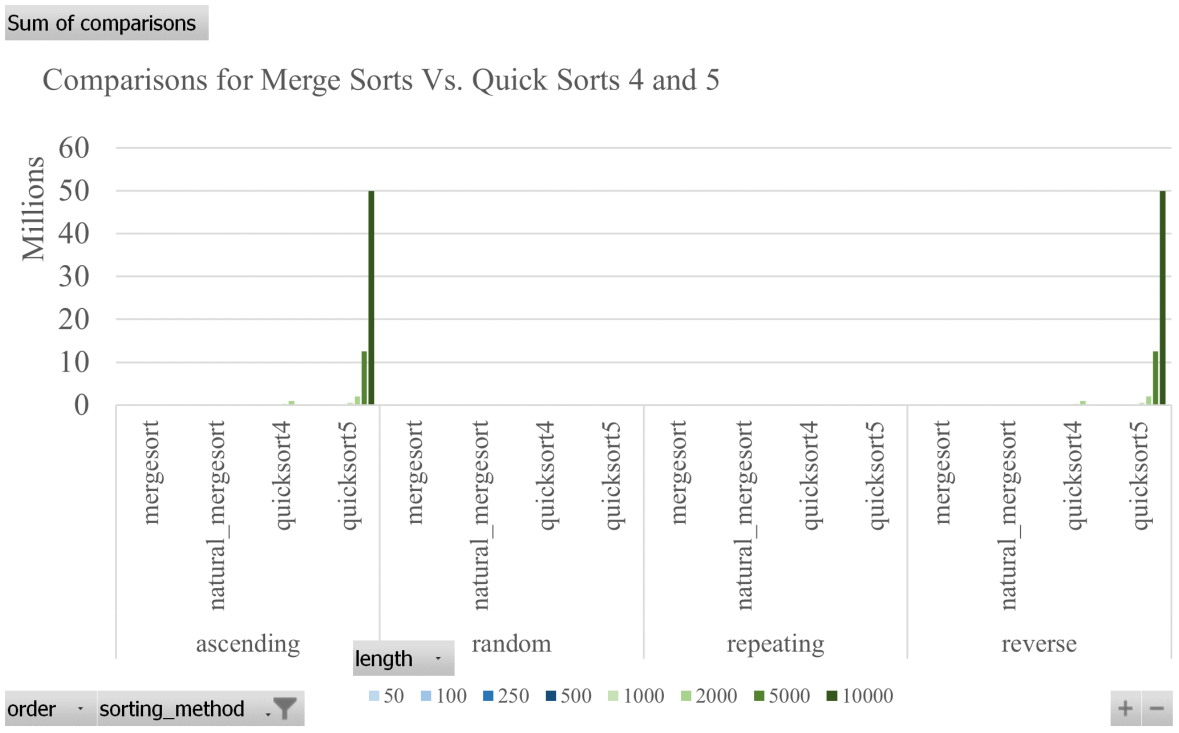

Figure 12: Comparisons for Merge Sorts Vs. Quick Sorts 4 and 5

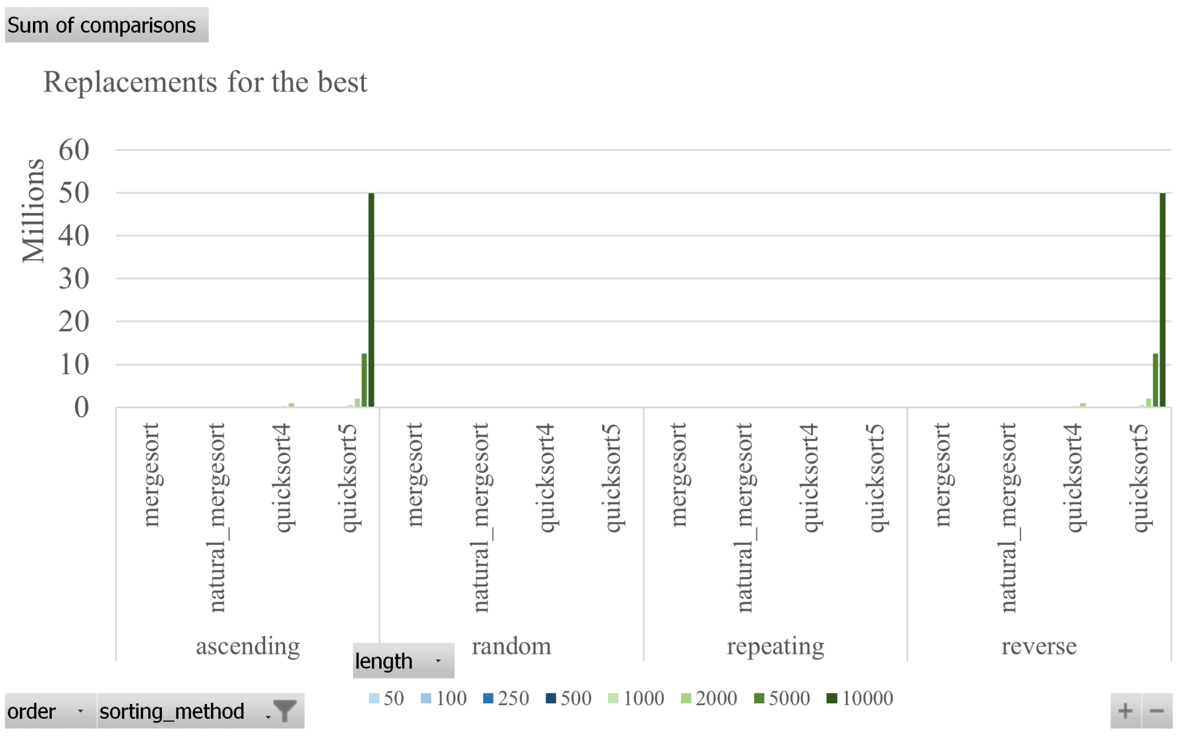

Figure 13: Replacements for Merge Sorts Vs. Quick Sorts 4 and 5

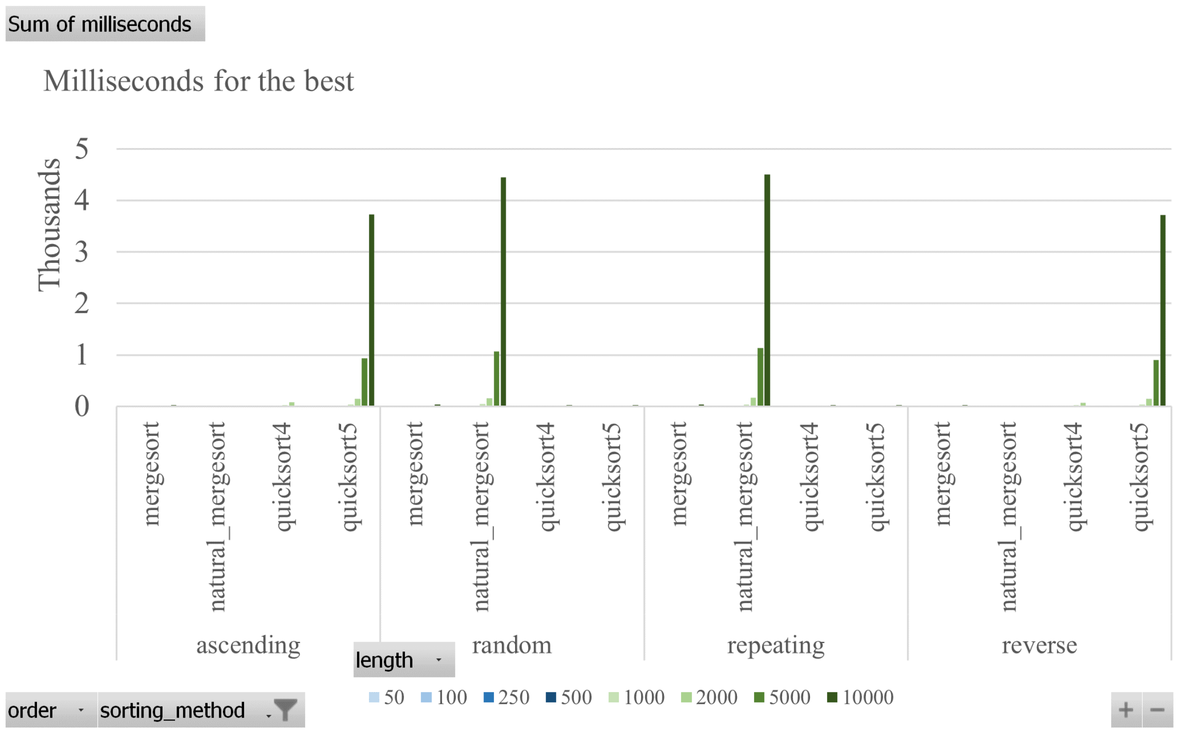

Figure 14: Milliseconds for Merge Sorts Vs. Quick Sorts 4 and 5

5 What I learned and what I would do differently

5.1 What I learned

I have summarized above several concepts which have contributed to the development of this lab, below I will explicitly list some of the concepts that I learned as a consequence of going through this exercise:

- Limitations of recursive algorithms and use of access memory
- Advantages of using a linked list over a python list implementation for sorting
- Implementation of hybrid sorting algorithms
- Between Quick Sort and Insertion Sort, which algorithm or hybrid approach is preferable depending on the circumstance.
- Benefits of the implementation of linked lists for sorting.

5.2 What I would do differently

I came into access errors on Quick Sorts on large lists and could not sort those with sizes of 5,000 and 10,000 for ascending and reverse order. These two constitute the worst cases for Quick Sort, and my program crashed every time. I believe a non-recursive implementation can help resolve this situation, and it is worth experimenting. A possible Iterative implementation of Quick Sort can be found here [11]. Another important enhancement for a more appropriate apples to apples comparison would be the use of a linked list implementation on the Quick Sort algorithms.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.

[2] B. N. Miller and D. L. Ranum, *Problem Solving with Algorithms and Data Structures Using Python*. Franklin Beedle Associates.

[3] Wikipedia, "Merge sort — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Merge%20sort&oldid=1095865966, 2022, [Online; accessed 25-August-2022].

[4] S. Woltmann, "Merge sort – algorithm, source code, time complexity," Aug 2022. [Online]. Available: https://www.happycoders.eu/algorithms/merge-sort/

[5] "Merge sort for linked lists," Jul 2022. [Online]. Available: https://www.geeksforgeeks.org/merge-sort-for-linked-list/

[6] I. Jackson, "Mergesort for linked lists," May 2022. [Online]. Available: https://www.chiark.greenend.org.uk/~sgtatham/algorithms/listsort.html

[7] S. Woltmann, "Quicksort – algorithm, source code, time complexity," Jul 2022. [Online]. Available: https://www.happycoders.eu/algorithms/quicksort/

[8] "Advanced quick sort (hybrid algorithm)," Aug 2021. [Online]. Available: https://www.geeksforgeeks.org/advanced-quick-sort-hybrid-algorithm

[9] Wikipedia, "Quicksort — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Quicksort&oldid=1105928267, 2022, [Online; accessed 25-August-2022].

[10] D. Gries, "Essential improvements to basic quicksort," 2018. [Online]. Available: https://www.cs.cornell.edu/courses/JavaAndDS/files/sort3Quicksort3.pdf

[11] "Iterative quick sort," Jul 2022. [Online]. Available: https://www.geeksforgeeks.org/iterative-quick-sort/