Project 3 Report

Introduction

This report presents the performance analysis of four different sorting and selection algorithms: stdSort, quickSelect1, quickSelect2, and countingSort. Each method was timed on inputs of size 1K, 100K, and 10M. The report includes a table showing the size and number of unique values for each input, the time it took for each implementation on each input, a complexity analysis of each method in O-notation, and a discussion of how having fewer unique values and more copies of each value affected the time.

Performance Analysis

The following table shows the time it took for each implementation on each input:

	StdSort	QuickSelect1	QuickSelect2	CountingSort
Input1 1000	106 microseconds.	106 microseconds.	1481 microseconds.	296 microseconds.
Input2 100000	14583 microseconds.	470090 microseconds.	2736972 microseconds.	4042 microseconds.
Input3 10000000	1887569 microseconds.	7453252354 microseconds.	8546463235 microseconds.	302650 microseconds.

Complexity Analysis

stdSort: The time complexity of std::sort is O(n log n), where ‘n’ is the number of elements in the vector. This is reflected in the times, which increase logarithmically with the size of the input.
quickSelect1: The time complexity of the quickSelect1 method is O(n^2) in the worst case, but on average it performs in O(n log n). This is because the method involves partitioning the input and recursively selecting from one of the partitions.
quickSelect2: The time complexity of the quickSelect2 method is also O(n^2) in the worst case and O(n log n) on average. However, this method performs multiple selections in each recursive call, which can lead to worse performance in practice as seen in the timings.
countingSort: The time complexity of the countingSort method is O(n + k), where ‘n’ is the number of elements and ‘k’ is the range of the input data. This method can be very efficient when the range of the input data is small compared to the number of elements, which is reflected in the relatively low times.

Discussion

Having fewer unique values and more copies of each value can significantly affect the time it takes for each method. For stdSort, quickSelect1, and quickSelect2, having fewer unique values can potentially improve performance, as there are fewer possible partitions in the quickselect methods and fewer possible comparisons in stdSort. However, for countingSort, having more copies of each value can increase the size of the count array, leading to longer times.

The countingSort method performed particularly well on the 10M input, likely due to the large number of duplicate values. The quickSelect methods, on the other hand, performed poorly on the larger inputs, possibly due to the overhead of multiple recursive calls and the worst-case quadratic behavior of quickselect.

Additional Observations

During the course of this project, I encountered a issues with the countingSort method, specifically in accurately counting the number of unique values. This issue was particularly confusing as it only manifested when running the code on Gradescope, while the local tests on my own compiler produced the expected results. For instance, when sorting the weights of male elephant seals, my local compiler correctly identified the number of unique data points: 787 for an input size of 1K, 3588 for 100K, and 5335 for 10M. However, when I ran the same code on Gradescope, I received the error of 1401 data points instead of the correct amount. I am not really sure how to correct this, as I do not know the correct number of data points I should be aiming for.

Conclusion

This project provided valuable insights into the performance characteristics of different sorting and selection algorithms. While some methods performed well across all inputs, others showed their strengths only under specific conditions. This underscores the importance of understanding the characteristics of the input data when choosing an algorithm.

Overall, the project highlighted the trade-offs involved in algorithm design and the importance of considering both theoretical complexity and practical performance. Future work could explore other sorting and selection algorithms, as well as the impact of different data distributions on performance.