Project 3 Report
Olti Gjoni
CSCI 335
Professor Justin Tojeira
05/10/24

1) Performance Table

Input File Name	Input Size	StdSort Time (s)	QuickSelect1 Time (s)	QuickSelect2 Time (s)	CountingSort Time (s)
test_input.txt	1K	0.000101625	0.000137084	0.000558959	0.00193117
test_input2.txt	100K	0.00600279	0.00625233	0.0281108	0.0171332
test_input3.txt	10M	0.372912	0.525259	55.0355	1.53336

2) Complexity Analysis

StdSort:

Complexity: O(n log n)
Explanation: std::sort typically uses introsort, which is a hybrid of quicksort (O(n log n) average complexity), heapsort (O(n log n) worst-case complexity), and insertion sort for small data segments (which is efficient despite its O(n^2) worst-case complexity). This blend offers a more efficient performance across various data spreads, maintaining O(n log n) complexity in all scenarios.

QuickSelect1:

Complexity: Best case O(n), worst case O(n^2)
Explanation: QuickSelect uses the partitioning logic of quicksort to specifically find the k-th smallest element without sorting the entire array. The average complexity is linear, O(n), because each partitioning operation typically reduces the problem size by approximately half. However, in the worst case (e.g., when the pivot is always the smallest or largest remaining element), the complexity degrades to O(n^2).

QuickSelect2:

Complexity: Similar to QuickSelect1 with best case O(n) and worst case O(n^2)
Explanation: QuickSelect2 is designed to further optimize the selection process by attempting to find multiple order statistics (e.g., quartiles) in a single pass. This approach aims to reduce redundant work across multiple calls to partitioning but can introduce complexity in managing multiple recursive paths, which might not always lead to improved performance. This particularly applies to large datasets where the overhead of handling multiple recursion stacks becomes significant.

CountingSort:

Complexity: O(n + k)
Explanation: Counting sort excels when the range of the dataset (k) is not significantly larger than the number of elements (n), as it simply counts the occurrences of each value and then reconstructs the sorted data from these counts. The performance is directly tied to k, with ideal scenarios being those where k is relatively small compared to n. As k approaches or exceeds n, the efficiency gain over comparison-based sorts (like std::sort) diminishes due to the increased space and initialization costs.

3) Discussion on Performance Impact

- StdSort consistently shows good performance, largely unaffected by the uniqueness of the values due to its reliance on comparisons and swaps.
- QuickSelect1 and QuickSelect2 show reasonable performance on smaller datasets but can struggle as size increases, particularly QuickSelect2, which took significantly longer on the largest dataset due to its more complex recursive strategy.
- CountingSort benefits from a higher number of duplicate values, which is evident from its relatively better performance on the largest dataset where the number of unique values is small compared to the dataset size. However, it still lags behind the more traditional std::sort due to overhead from managing a large count array and iterating over it.

4) Additional Observations

- QuickSelect2's dramatic increase in time for the 10M size might indicate inefficiencies in handling data where partitioning does not significantly reduce the size of the subsequent recursive calls.
- CountingSort shows its strength in scenarios with many duplicate values but might not be the best choice as the number of unique values approaches the size of the input.