These numbers are the averages of three executions of each input file rounded to the nearest whole number, all raw data is below. All computation was done on the Linux machine.

	Unique Values	Size	stdSort	quickSelect1	quickSelect2	countingSort
test_input.txt	787	1K	39 µs	17 µs	22 µs	118 µs
test_input2.txt	3588	100K	4520 µs	1471 µs	1406 µs	1479 µs
test_input3.txt	5335	10M	341359 µs	124933 µs	125148 µs	79782 µs

Raw results

test_input.txt	stdSort	quickSelect1	quickSelect2	countingSort
Attempt one	40 µs	18 µs	24 µs	129 µs
Attempt two	43 µs	17 µs	21 µs	116 µs
Attempt three	34 µs	15 µs	20 µs	109 µs

test_input2.txt	stdSort	quickSelect1	quickSelect2	countingSort
Attempt one	4509 µs	1450 µs	1375 µs	1442 µs
Attempt two	4518 µs	1485 µs	1413 µs	1500 µs
Attempt three	4532 µs	1478 µs	1430 µs	1496 µs

test_input3.txt	stdSort	quickSelect1	quickSelect2	countingSort
Attempt one	341320 µs	124981 µs	125353 µs	79867 µs
Attempt two	341619 µs	125322 µs	125011 µs	79691 µs
Attempt three	341139 µs	124496 µs	125080 µs	79787 µs

The stdSort function's time complexity is dominated by std::sort which has an average and worst-case complexity of O(n log n) for sorting n elements. While there are constant time (O(1)) operations for calculations within stdSort, they are negligible compared to sorting. Overall, stdSort has a time complexity of O(n log n).

The quickSelect1 function calculates the median (P50) using quickSelectFunction, which has an average-case time complexity of O(n). However, it's crucial to note that while the average-case time complexity is linear, the worst-case time complexity can degrade to O(n^2), especially if the pivot selection strategy consistently chooses poorly. Additionally, the utilization of insertion sort for small ranges adds efficiency to the algorithm for smaller datasets, further enhancing its practical performance. It also calculates P25 and P75, each requiring another call to quickSelectFunction. However, these calls operate on smaller partitions of the dataset, with sizes proportional to n/2. Therefore, each call has an average-case time complexity of O(n/2). Finding the minimum and maximum values in the dataset involves iterating through a portion of the data, which contributes O(n) time complexity. Overall, the dominant time complexity comes from the calls to quickSelectFunction, making the average-case time complexity of quickSelect1 O(n).

The quickSelect2 function efficiently determines the positions of minimum, P25, median (P50), P75, and maximum values for keys in constant time (O(1)). These keys are then employed within the quickSelectFunction to retrieve the corresponding values after the array has undergone sorting. While the average-case time complexity of quickSelectFunction is linear at O(n), it's important to acknowledge that the worst-case scenario can degrade to O(n^2), particularly if the pivot selection strategy consistently performs poorly. Additionally, the integration of insertion sort for small ranges bolsters the algorithm's efficacy, particularly with smaller datasets. Despite the potential for quadratic time complexity in the worst-case scenario, the predominant time complexity of quickSelect2 is driven by the call to quickSelectFunction, resulting in an average-case time complexity of O(n).

The countingSort function operates in several distinct steps to calculate percentiles using counting sort. Initially, it counts the occurrences of each unique data value, utilizing an unordered map to store counts. This counting step runs in O(n) time, where n is the number of elements in the input data. Subsequently, it creates a vector of pairs from the hash map, which has a time complexity of O(u), where u is the number of unique elements. Sorting this vector using std::sort takes O(u log u) time. Following the sorting, the function calculates cumulative counts for quartiles and then determines quartile values based on these counts, each of which entails iterating through the sorted vector, resulting in O(u) time complexity for both steps. Therefore, the overall time complexity of the countingSort function is O(n + u log u), where n is the number of elements in the input data and u is the number of unique elements. Since u is bounded by n, this can be simplified to O(n log n).

Having fewer unique values and more copies of each value generally benefits sorting algorithms, as they can exploit the repetition to optimize their processes. Standard sorting algorithms like stdSort may improve due to reduced distinct comparisons and swaps needed. QuickSelect algorithms may exhibit improved performance due to reduced partitioning overhead in datasets with more duplicates. Counting sort remains efficient regardless of the number of duplicates, making it suitable for datasets with many copies of each value.

stdSort function:

Hypothesis: Standard sorting algorithms like std::sort typically benefit from having duplicates as they can exploit patterns in the data, resulting in more efficient sorting. With more copies of each value, there are fewer distinct comparisons and swaps needed, leading to faster sorting times.

quickSelect1 and quickSelect2 function:

Hypothesis: In datasets with more duplicates, the partitions created during the quickSelect algorithm tend to be smaller, leading to faster selection of the desired percentiles. Additionally, the use of insertion sort for small ranges further enhances efficiency for smaller datasets.

countingSort function:

Hypothesis: Counting sort's time complexity is unaffected by the number of duplicates, as it directly counts occurrences. Therefore, having more duplicates will not slow down the algorithm, and may even improve performance as it bypasses comparison-based operations.

Throughout the project, several notable observations emerged regarding the performance of different sorting and selection algorithms. Initially, there was an anticipation of significant runtime differences, particularly between theoretical expectations and practical results. However, as the input size increased, an unexpected trend surfaced countingSort, which initially performed poorly, emerged as the most efficient, particularly benefiting from duplicate values in larger input sets.

Another intriguing observation was the divergence in execution times between stdSort and countingSort, despite both having O(n log n) time complexity. While stdSort struggled with duplicates, countingSort excelled, demonstrating the importance of considering implementation details beyond theoretical complexities.

Furthermore, despite quickSelect1 and quickSelect2 sharing similar execution methods, their runtime remained consistently close across varying input sizes. This contrasted sharply with the considerable differences observed between stdSort and countingSort, underscoring the significance of implementation nuances.