CSCI 33500
Project 3 Report
Mingjun Lu

Data input-> Methods	test_input.txt 1K	test_input2.txt 100K	test_input3.txt 10M
StdSort&Time(microseconds)	Execution time: **47.748** microseconds	Execution time: **4511.61** microseconds	Execution time: **498595** microseconds
QuickSelect1&Time(microseconds)	Execution time: **118.107** microseconds	Execution time: **14184.7** microseconds	Execution time: **2.67391e+06** microseconds
QuickSelect2&Time(microseconds)	Execution time: **103.401** microseconds	Execution time: **13507.1** microseconds	Execution time: **2.30441e+06** microseconds
CountingSort & Time(microseconds) & Unique	Execution time: **809.975** microseconds Unique: 787	Execution time: **27154.8** microseconds Unique: 3588	Execution time: **1.89267e+06** microseconds Unique: 5335

StdSort:
The `stdSort` function leverages `std::sort` from the C++ Standard Library, which is a hybrid algorithm combining quicksort, heapsort, and insertion sort. The average complexity of `std::sort` is O(n log n), where n represents the number of elements in the vector. This complexity stems from the adaptive nature of `std::sort`, optimizing for different data distributions and sizes.

QuickSelect1:
`QuickSelect1` is designed to efficiently find the k-th smallest element in an unsorted array. The algorithm operates by recursively partitioning the array around a pivot, similar to the partition step in quicksort. The average time complexity is O(n), which results from progressively halving the problem size with each recursive call. However, in the worst case, when pivot selection leads to unbalanced partitions, the complexity can escalate to O(n^2).

QuickSelect2:
`QuickSelect2` focuses specifically on finding five-number summaries. Like the standard Quickselect, it uses a partition operation and recursive targeting of specific indices, maintaining an average time complexity of O(n). The method selects the median of the target index set as the pivot during each recursion, which helps minimize the number of recursive calls and maintain linear complexity on average, though it can still reach O(n^2) in the worst-case scenarios.

CountingSort :
The `CountingSort` method for quartiles operates with a complexity of O(n + k) where n is the number of elements and k denotes the range of input values. This method is notably effective for data with limited unique elements, as it primarily

counts occurrences without comparing elements. The sorting of counts is managed within the same complexity frame, ensuring efficiency when k is much smaller than n . Although insertion sort is employed for very small subsets, its impact on the overall complexity is marginal.

The performance of std::sort can be positively impacted when there are fewer unique values with more copies of each. This can lead to quicker sorting because elements that are the same do not need to be repeatedly compared during the sorting process. Assuming when many elements are identical, the algorithm potentially performs fewer swaps since identical elements are inherently in the correct order relative to each other once placed. This can reduce the number of operations required to partition data during the quicksort phase of introsort.

QuickSelect1 benefits from fewer unique values. The partitions around the pivot can effectively segregate larger portions of the array when the pivot is a common value, reducing the number of recursive calls needed to find the target percentile or k-th smallest number.That means less recursion. Assuming the presence of duplicates can simplify the partitioning process, as once a common pivot is chosen, many elements can be instantly classified as less than, equal to, or greater than the pivot, which often results in a dramatic reduction of the search space after each partition.

QuickSelect2's efficiency is enhanced with fewer unique values. The recursive approach that focuses on specific indices can leverage common values to quickly achieve the necessary partition without further redundant sorting of the sections of the array that are already effectively sorted by the presence of duplicates.Assuming Fewer unique values mean that the algorithm may often encounter the pivot or near-pivot conditions quickly, reducing the problem size faster and minimizing the total number of recursive steps needed.

Both QuickSelect1 and QuickSelect2 benefit from having fewer unique values in the dataset, leading to reduced recursion, simplified partitioning, and quicker convergence to target statistical values or specific elements

CountingSort is exceptionally efficient in scenarios with fewer unique values. This method counts occurrences rather than performing direct element comparisons, which makes it incredibly fast when many elements are identical since the counting operation is simple and the range of counts (k) is small relative to the size of the data (n).Assuming with a large number of duplicates, CountingSort swiftly accumulates the counts of each distinct element, and since there are fewer distinct elements to manage, the subsequent steps of accumulating counts to find quantiles become quicker. Additionally, the space required to store the frequency of each unique value remains minimal, enhancing overall efficiency. The condition of having a large number of duplicates reduces the range of counts (k)
in CountingSort.

I observed during the project that CountingSort had the slowest execution time when the amount of data was small (test_input.txt). But as the amount of data increases, when the amount of data is large, CountingSort will run faster than two QuickSelect. In my observations, `stdSort` performed the best for small datasets, while `QuickSelect2` was the fastest for medium-sized datasets. However, as the data volume increased, `stdSort` remained the fastest overall, but `CountingSort` outperformed both `QuickSelect` algorithms. When implementing QuickSelect1, optimizing recursion and selecting the pivot index proved to be challenging. Unlike QuickSelect2, which sorts within the existing data, QuickSelect1 necessitates a new sort operation each time, leading to additional computation and requiring optimization for recursive calls.