1) A clearly labeled table showing the size and number of unique values for each input, and the time it took for each implementation on each input.

Sorting Method	Test_input.txt (1000)	Test_input2.txt (100k)	Test_input3.txt (10M)
StdSort	71 microseconds	7674 microseconds	126572 microseconds
QuickSelect1	47 microseconds	1393 microseconds	242091 microseconds
QuickSelect2	65 microseconds	3179 microseconds	350496 microseconds
CountingSort	121 microseconds	1034 microseconds	72559 microseconds

2) A complexity analysis of each method in O-notation.

StdSort: O(nlogn)

This algorithm uses introsort, an efficient combination of quicksort, heapsort, and insertion sort. After sorting the elements in data, access to the statistics is direct.

QuickSelect1: O(nlogn)

The quickselect helper function in quickselect1 is called recursively on subarrays with an average number of calls to it being O(logn). Though its worst case is O(n^2), it is unlikely to get to worst case due to use of median3 algorithm to find the pivot. Each call involves a linear-time partitioning operation with O(n) efficiency. Each time the helper function is called to find a different statistic, it is using a drastically smaller subarray. When the range is 20 or less, the helper function defaults to insertion sort, which has O(1) efficiency because of the small range.

QuickSelect2: O(nlogn)

The quickselect helper function in quickselect2 is similar to quickselect1 except instead of calling the function for each statistic, it uses a set of keys and checks them after each partition step. After the initial quickselect operation to find the median, the data is partitioned into two halves around the median. Then, the algorithm checks if any of the predetermined percentile indices lie within the left or right partitions. If any of the percentile indices lie within a partition, the algorithm recursively applies quickselect only to that partition to find the corresponding percentile elements. This avoids unnecessary recursive calls on partitions where the desired percentiles are not present. The helper function has time complexity of O(nlogn). It is called an average of O(logn) times and each call has a linear partition algorithm O(n), and iterates through the set of keys and calls the function again if one is found.

CountingSort: O(n)

The algorithm iterates through data vector (O(n)) and counts each occurrence using a hashmap. When constructing the count vector, it iterates through the hash map and adds pairs of keys and values to the vector, so in total it has O(n) efficiency. After, std::sort sorts the vector in ascending order of the first element in the pair. Sorting this vector has time complexity of O(klogk), where k is the number of unique values. K is usually smaller than n. Finally, the function iterates through the vector and sums the counts until the desired indices are reached. Overally, the time complexity is O(n) because it iterates through n values when inserting into the hash map and when finding the values in the vector.

3) Discussion of how having fewer unique values and more copies of each value affected the time, along with your hypotheses of why each algorithm performed as it did

Generally in sorting algorithms, when there are fewer unique elements, there will be fewer comparisons made and sorting is potentially faster.

In both quickselects, if there are repeated values it can be redundant to make comparisons especially if they are clustered together. These algorithms don't do much to consider repeated values. However, they can have lower runtime depending on the distribution of repeated values

Counting sort is most drastically benefited by unique elements. Though it had the highest runtime for 1000 values, it had the lowest for both 100k and 10M values. It is the only algorithm that is not inplace and sorts only the unique values, aka the keys in the hash map. In addition, it is the only algorithm that doesn't have O(nlogn) efficiency, so the rate it increases as more elements are imputed is the lowest.

Though quickselect2 attempts to optimize selection of percentiles by preselecting indices, it introduces additional overhead in checking and managing indices after each partition step. It must maintain and iteration through keys vector. In addition, some of the work is redundant since the recursive call may end up reprocessing the same elements of partitions multiple times. The redundant work makes it so the runtime is worse than quickselect1.

In quickselect1, the algorithm recursively partitions the data into two halves without keeping track of other keys. It is more straightforward because it has reduced overhead compared to quickselect2. Better cache locality can lead to faster memory access and lower runtime. In general, quickselect2 introduces additional redundancy and complexity that is not present in quickselect1.

4) Any additional observations you had during the project.

I had the most difficult time with quickselect2 as it was hard for me to figure out the implementation, but there was not too much coding involved. I started out doing a lot of coding but felt something was off because the project specifications said there was not much coding to do. I found that often times my indices were only slightly off by 1. It was usually because the comparisons to left and right in the helper function or insertion function were slightly off or the indices for median, p25, and p75 were slightly off. In addition, I found myself going back and forth the project specifications a lot because there was something I missed and I needed to fix my code. I wish the specifications were more organized because I felt like the information was scattered in a way that wasn't intuitive. Overall, this project was quite challenging and frustrating at times but it was actually quite fun at times.