Time in microseconds (average)

Input	Std Sort	Quickselect1	Quickselect2	Counting Sort
Testinput 1	207	71	282	593
Testinput 2	34045	6492	33798	11088
Testinput 3	3826657	505251	3940163	799898

Unique Values: 787 3588 5335

Complexity analysis of each method in O-notation:

Std sort: O (n log n)

Std::sort uses introsort which is n log n for average and worst case, n being the number of elements in the data vector. Quicksort + heapsort + insertion sort
The sorting function was the main and dominant function, finding numbers for percentiles were all O(1) because of the random access property of vectors.
Overall O (n log n)

Quickselect 1: O (n)

Insertion sort : O (n) since best case and for small number of inputs (20)
Quickselect implementation : O (n)
- Median: O(1) used to find a good pivot point for each quickselect call
- Partition: O (n) while loops iterates i and j and swaps vaiues when out of place
- Recursive Calls: O (n/2) recurses quickselect on half of the array

Find min and max: O(n) looks at at most n/2
Quickselect 3 Calls: O (n) from O (3n^2) or O (3 n)
Worst O (n^2) with bad pivots but median function solves that problem, mostly.

Quickselect 2: O (n)

Insertion sort : O (n) since best case and for small number of inputs (20)
Quickselect implementation : O (n)
- Median: O(1) used to find a good pivot point for each quickselect call
- Partition: O (n) while loops iterates i and j and swaps values when out of place
- Recursive Calls: O (n/2) recurses quickselect on half of the array

Find min and max: O(n) it looks at at most n/2
Quickselect 3 Calls: O (n) from O (3n^2) or O (3 n)
Worst O (n^2) with bad pivots but median function solves that problem, mostly.

Since this one takes a vector of keys, the partitioning sizes for each call will be smaller than the sizes for bquickselect1, and it shows in the results that this will perform better in larger inputs. This will make recursive calls fewer when it performs better, while quickselect1 was faster in smaller inputs due to the difference in keys.

CountingSort : O(n) (O(n + m + m log m))
O(n log n) worst case and gets closer to n log n the more unique inputs there are
Hash map creation and insert : O(n), adds all values into map and increments at the same time, each insert is O(1)
Vector of pairs: O (m), m is equal to the number of unique values in the map, in this case was : 787, or 3588 or 5335 from the inputs. Time would heavily depend on the number of unique values. Worst case O (n)
Sorting vector of pairs using std sort : O(m log m) : Sorts unique values in vector
Finding percentiles: O(m)
Iterates the vector to find the correct pair for each percentile. Since there are m pairs in the vector, it totals to O (m).
Find min/max: O(n)
Iterates the data vector to find the minimum and maximum
Find unique values: O(1): Return size of pair vector

O(n) + O(m) + O(m log m) + O(n) = O (n + m)
Consistently the slowest algorithm because it involves multiple different operations
Counting sort heavily depends on the number of unique values, and sorting the pairs in the vector becomes the more dominant function when there are more unique values. Counting sort theoretically should perform similar to the other O(n) or n log n algorithms, but since there are more iterative functions in counting sort, they add up to more time cost.

Misc:

Having more unique values barely improved std sort for 1K integers, and other algorithms improved just slightly. There were improvements with larger sizes, like 10K with less unique values.
Using another test file with 400 unique numbers vs test_input2, most algorithms saw a significant improvement when there were less unique values (more duplicates).
STD sort for 10K integers had around a 25% speed improvement comparing a file with 400 unique values vs the test input with 3588 unique values.

Unique Values	Std Sort	Quickselect 1	Quickselect 2	Counting Sort
400	27361	4524	31880	8501
3588	34045	6492	33798	12025

Std sort 24% faster with more duplicates
Quickselect1 43%
Quickselect2 6%
Counting sort 42%
STD Sort's speed was consistently close to quickselect2, and the two were the fastest algorithms for the 10 million integer input. Quickselect 1 was the fastest for !K and 100K inputs, but got slower for the 10M input. Counting sort was consistently the slowest algorithm.