Rio Simpson
Project 3 Report

Table 1: Test Inputs and Sort Timings (in MS) for StdSort, QuickSelect1, QuickSelect2, and CountingSort

Sorting Algorithm	Timings by Input Size (in Milliseconds)		
	1K	100K	10M
StdSort	0.040576	5.17735	457.872
QuickSelect1	0.019423	1.43801	145.593
QuickSelect2	0.046206	1.55713	161.233
CountingSort	2.31791	1.50166	96.9821

1. StdSort utilizes *std::sort* to sort an array, and obtains it's desired 5-summary values by using random access. Since the underlying algorithm of *std::sort* is intro-sort, a hybrid of quicksort, heapsort, and insertion-sort, we know that the average time complexity of the StdSort algorithm is O(nlogn), and its worst time complexity is O(n^2). With the test cases provided, we notice that there is a gradual increase in the time to complete a larger input, with a drastically significant increase at an input of 10M values. The time to assign the 5 summary values is negligible at a time complexity of O(1) since we are able to use random access on calculated indexes for those values. Therefore, the overall time complexity of the StdSort function is dominated by the sorting operation, which is O(n log n), where n is the size of the input vector.
2. QuickSelect1 utilizes a QuickSelect algorithm by which we find the 25th percentile value, median (50th percentile value), and 75th percentile value. Each call to QuickSelect initiates additional recursive calls within a specified range. Since the recursive calls reduce the range of items QuickSelect needs to search through, its asymptotic behavior converges. As such, the findings support the theoretical time complexity of O(n). In fact, the times it took for QuickSelect1 to finish each input, which are 100x in difference, is correspondingly 100x longer from the smaller inputs to the larger inputs. It is worth noting there is some additional overhead due to two loops used to find the min and max values, though their time complexity is O(n/4) since they are restricted to a range from 0 to the P25 index and P75 index to the end of the data vector. Therefore, the overall time complexity of the quickSelect1 function is O(n).
3. QuickSelect2 utilizes a QuickSelect algorithm by which we find the 25th percentile value, median (50th percentile value), and 75th percentile value. In this implementation, a vector of keys is passed as opposed to a single key. In this way, each recursive call can look at multiple points and call QuickSelect on those points incrementally and dynamically. However, the time complexity for the QuickSelect implementation in quickSelect2 is proportional to the time complexity of QuickSelect1 times the number of keys in the collection, meaning the average time complexity of QuickSelect remains O(n) with a worse case of O(n^2) if there is an unbalanced pivot. We see this in the timing found, in how the timings for QuickSelect2 per input is proportional to QuickSelect1, if not slightly worse.
4. CountingSort utilizes a counting-sort algorithm by which we transfer the values of an input into an unordered_map or hash table where each value is the key and its occurrence is the value

associated with the key. We find the respective 5-summary data points by iterating through the map and checking if the total number of items/values that have passed fall within some tolerance of the desired data points, specifically their indexes relative to the size of the input. Overall, the time complexity of building the map takes O(n) since we iterate through the input; the time complexity to transfer key-values to a vector of pairs takes O(m), where m is the size of the map (where m < n since duplicate values are incremented to a key); the time complexity to sort the vector of pairs is O(m log m) since we use std::sort on a vector of m items; and the time complexity of finding three of the 5-summary values is O(3m/4) since we iterate through the vector up to the 75th percentile – the added time to find min and max is negligible at a time complexity of O(1) with random access. Overall, the time complexity of CountingSort reduces to O(m log m) as dictated by the sorting algorithm.

Tables 2-4: Gamma Distribution Inputs of Different Alpha and Beta Values, and Sort Timings (in MS) for StdSort, QuickSelect1, QuickSelect2, and CountingSort

- Decreasing the alpha value clusters values closer to the minimum value
- Increasing the beta value increases the range of generated values

Table 2: Alpha 1, Beta 100

Sorting Algorithm	Timings by Input Size (in Milliseconds)		
	1K	100K	10M
StdSort	0.039554	3.70527	444.825
QuickSelect1	0.019149	1.29954	541.503
QuickSelect2	0.045289	1.84068	441.725
CountingSort	0.104342	0.922381	95.4845
Unique Count:	491	1181	1706

Table 3: Alpha 2, Beta 500

Sorting Algorithm	Timings by Input Size (in Milliseconds)		
	1K	100K	10M
StdSort	0.041214	5.8297	467.765
QuickSelect1	0.014002	1.57116	173.253
QuickSelect2	0.044759	1.61159	163.755
CountingSort	0.228899	2.0526	99.5275
Unique Count:	806	3968	6515

Table 4: Alpha 3, Beta 1000

Sorting Algorithm	Timings by Input Size (in Milliseconds)		
	1K	100K	10M
StdSort	0.034844	5.75203	520.304

QuickSelect1	0.01576	1.49353	137.308
QuickSelect2	0.043466	1.47881	140.282
CountingSort	0.166181	2.28979	88.1462
Unique Count:	919	8652	14377

Discussion

- When the number of unique values decreases and the number of copies of each value increases, the CountingSort algorithm performs significantly better than the other algorithms. This is because CountingSort is not a comparison-based sorting algorithm. Instead, it counts the number of occurrences of each value and then uses this information to place each value in its correct position in the sorted array. As such, CountingSort can more effectively sort given inputs
- On the other hand, the performance of StdSort, QuickSelect1, and QuickSelect2 algorithms decreases as the number of unique values decreases. These algorithms are comparison-based, meaning they compare pairs of elements to determine their relative order. When there are fewer unique values and more copies of each value, these algorithms end up making many unnecessary comparisons, which slows them down.
- In conclusion, the stability and versatility of CountingSort shows that it not only handles large inputs well, but inputs whose values range from evenly distributed to heavily clustered, and inputs with numerous duplicate values

Observations

- The time to find the 25th percentile value, median, and 75th percentile value in QuickSelect2 would have been O(m) since the loop could also be used to find the number of unique values (for which are values that, by definition, only occur once). However, gradescope would not accept this implementation and took the unique value as the total number of individual elements regardless of their count, or in this case the size of the map.
- While the time complexities of QuickSelect1 and QuickSelect2 are similar, as is their functionality since they use the same pivot method, each has their own advantages depending on the input. Most notably, QuickSelect2 seems to handle inputs with a greater number of duplicates and fewer unique values.