Tamanna Chowdhury
May 4, 2024
CSCI 335
Project 2: Report

Runtime of Various Sorting Algorithms (in milliseconds)

	StdSort	**QuickSelect1**	**QuickSelect2**	**CountingSort**
O-notation	O(n log n)	O(n)	$O(n^2)$	O(n log n)
1k	0.134387ms	0.271745ms	0.081296ms	0.356474ms
100k	14.5283ms	430.107ms	2.88846ms	6.41996ms
10M	134.531ms	24647.8ms	20.3686ms	42.9918ms

Compiled on Linux Mint 21 Vanessa, Dell Latitude 5430

Algorithmic Analysis:

StdSort:
Having fewer unique values and more copies of each value will affect as much as the other algorithms because of the use of std::sort which is in the C++ standard library. While this is generally efficient it can affect duplicates if it is not handled correctly while being sorted. This can cause issues when handling many duplicates in large datasets.

The time complexity of stdSort is used from the algorithm library std::sort. Upon looking at my code you can see that I have created a new vector called sortedData where it stores the information. First, the thing the code does is read in the data given and it sorts the data and puts it within the vector which takes O(n log n) time. Upon using std::sort it is kinda like using introsort which is a hybrid sorting algorithm. But in order to find the quartiles 25th and 75th it involves trying to access the element at specific intervals taking O(1) time while trying to find the 50th which is the middle of the index which takes about O(1) time as well. Thus due to this the complexity of this algorithm is O(n log n) on average where n is the number of elements being inputted before the vector and the time it takes to find the quartiles is O(1).

QuickSelect1:
Having fewer unique values and more copies of each value can impact the performance of sorting algorithms in various ways. If we have more duplicates it will make the sorting duplicates

will make unbalanced partitions. For instance the algorithm like QuickSelect may experience challenges when encountering numerous duplicate values.

The time complexity of QuickSelect is O(n) which is the average case where n is the number of elements in the input array. In worst case this would be O(n^2) id the pivot selected was poor causing a lot of issues to how the data will be organized. However since quickSelect is called three times in quickSelect1 the dominant time complexity contribution comes to quickselect. Thus looking at the code we are also told to find the max and the min separately meaning we are not allowed to use std::sort which would have given use a time complexity of O(n log n). Science we are mentally iterating through the entire array once it gives us a complexity of O(n). This gives us a overtall time complexity of quickSelect1 O(n) + O(n) = O(n) where n is the number of elements being put into the array. Allowing the complexity to grow linearly with the size of the input array. I also noticed that when the output got bigger the runtime took a bit longer than usual compared to all the other algorithms.

QuickSelect2:
Having fewer unique values and more copies of each value using QuickSelect may encounter having more repetitions of the same value during partition; this can lead to unbalanced partitions and causing the algorithm to approach its worst-case time complexity more often.

The time complexity of QuickSelect2 is O(n^2). The partitioning and quickSelect takes O(n) time where n is the size of the subarray being partitioned. Then the algorithm selects the quartiles using quickSelect and each time it is called it takes O(n) time. The time it takes to insert the sort is the size of the subarray being processed either less than or equal to 10 which is the threshold. Then we call insertionSort which takes the complexity of O(n^2) in the worst case. Overall the time complexity of this function since the number of quartiles to be selected in 5 and the partitions and quickSelect steps are performed within the subset of the array allowed the complexity to reach O (n^2) due to the repetition and selection. However if the number is significantly smaller than n the time complexity would be closer to O(n). Therefore the complexity of quickSelect2 can be around O(n) or O(n^2) deppengin on the size of the input array.

CountingSort:
Having fewer unique values and more copies of each value can significantly impact the performance of the counting sort algorithm. WHen there are more duplicates in the data set it will increase the time and space complexity of the algorithm causing an issue on how fast it will return the sorted data. Additionally if there are more copies of each value the size of the counting array to store the counts will also increase potentially increasing the memory usage.

The time complexity of countingSort is O(n log n). First we are saving data within a vector which means it is iterating through the entire data vector once and incrementing the count of each unique value in an unordered map. Which will take the complexity of O(n) where n is the size of inputs in the vector data. Then we are copying the key-value pairs from the unordered_map into the vector. Taking all the unique pairs n which gives us a complexity of O(n). Thus when we are sorting the vector of key-values pairs it is taking O(n log n) time. We then need to calculate the quartiles which means we have to iterate through the sorted vector once which takes about O(n) time. Finally we need to calculate the max and min using the algorithm std::minmax_element taking O(n) times. Thus the overall time complexity for this function is O(n log n).