Anton Ha

CSCI 335

May 5th, 2024

Project 3 Report

For project 3, we are assigned to find the minimum, maximum, 25th percentile, 50th percentile, and 75th percentile of a data set. The four algorithms that will be implemented to complete the assignment are two quick select methods, counting sort method, and lastly using standard library sorting algorithm. For the std sort implementation, we just call std sort on the data set and print the five number summary. The first quick select method is by calling quick select three times finding first the half of the data set size largest element, then the quarter of the data set size largest element size and then the third quarter of the data set size largest element size. Lastly, from the 0th index to the 25th percentile - 1 index, we find the minimum for the minimum of the data set, then from the 75th percentile index to the end of the array, we find the maximum for the maximum of the data set. The second quick select method will be recursively called with a vector of keys rather than one key, and update the keys on the upper and lower half of the pivot until there are no more keys left. This method will be called with 0, 25th percentile, 50th percentile, 75th percentile and the last index of the data set. For counting sort, we will have a frequency hashmap of the data set, keep a vector of the unique values, call std sort on the unique values, and then iterate through the unique vector keeping count of the frequency to print the five number summary.

Std sort typically has a O(n log n) run time, and since we only call std sort once in the std sort method and just print the five number summary in O(1)*5 = O(1). In total making the whole algorithm O(n log n).

For the quick select one method, we are calling quick select three times, and since quick select has average runtime of O(n), O(n) * 3 = O(n). Lastly, we find the minimum and maximum by iterating a distance of the 25th percentile index, which will not affect overall runtime as it is not greater than O(n). Summing up, in the average case this algorithm will be O(n), but in the worst case the algorithm can be O(n^2) as the worst case of quick select is O(n^2).

For the quick select two method, since the quick select algorithm's average case is O(n), we are recursively calling quick select with the keys of the updated half the corresponding pivot, reducing our problem size in half. The method runtime will be on average O(n) similarly to the quick select one method, but we have to account for recursion overhead and memory usage in the recursive calls. Also similarly to the quick select one method, accounting for the worst case of the method, overall runtime will be O(n^2).

Lastly, the counting sort method has many steps in the method that has to be taken to account. First, creating the frequency hashmap takes O(n) time as it iterates through the vector keeping track of the count inside a hashmap. Second, we create a vector of the unique values by iterating through the hashmap and adding the keys to the vector, taking O(size of hashmap (h)) time. Next, we std sort the vector which has O(h log h) time. Next, we iterate through the unique values and print the five number summary, which has O(h) time. Overall this algorithm appears to be a great method in finding the five number summary, but we have to account for the duplicates in the data set. The more duplicates we have in the data set, the faster the method will be, however if the data set has no unique values, we have a worst case of 3O(n) + O(n log n) = O(n log n). Additionally, it's important to note how much memory we are going to be using with the hashmap and the additional unique vector.

As you can see, in our analysis of the four methods, it perfectly lines up with the result of the timing of each method. As we can see below, quick select one is the fastest algorithm, then quick select two, and then std sort or counting sort. Since quick select one does not have to deal with the overhead of recursive calls, it makes sense for it to be faster than quick select two. Additionally, it makes sense these two methods are faster than counting sort and std sort because O(n) is less than O(n log n). For the timing situation of std sort and counting sort all depends on the data set, as mentioned above, if the data set contains a decent amount of duplicates, counting will typically be faster than std sort. However, if there are not as many duplicates in the data set, counting sort will perform unnecessary computations making it slower than the straight forward n log n solution.

Timing Table

	Input One	Input Two	Input 3
Std Sort	131µs	19008µs	186810µs
Quick Select One	32µs	3296µs	26070µs
Quick Select Two	66µs	4464µs	37491µs
Counting Sort	415µs	12765µs	95193µs

```
test_input1.txt is being ran!
Std Sort: 131 microseconds.
QuickSelect One: 32 microseconds.
QuickSelect Two: 66 microseconds.
Counting Sort: 415 microseconds.

test_input2.txt is being ran!
Std Sort: 19008 microseconds.
QuickSelect One: 3296 microseconds.
QuickSelect Two: 4464 microseconds.
Counting Sort: 12765 microseconds.

test_input3.txt is being ran!
Std Sort: 186810 microseconds.
QuickSelect One: 26070 microseconds.
QuickSelect Two: 37491 microseconds.
Counting Sort: 95193 microseconds.
```