Nick Szul

Professor Tojeira

CSCI 33500

10 May 2024

Project 3 Report

1.)

Implementation	Test Input 1(1000 integers	Test Input 2(100k integers)	Test Input 3(10 million integers)
std::sort	147 microseconds	19978 microseconds	2598745 microseconds
QuickSelect1	43 microseconds	3252 microseconds	411261 microseconds
QuickSelect2	100 microseconds	13391 microseconds	1551364 microseconds
CountingSort	559 microseconds	20120 microseconds	1768414 microseconds
Unique Values	787	3588	5335

2.)

std::sort Method:

This method has a time complexity of O(nlogn), since that is always the time complexity of std::sort

QuickSelect1:

While QuickSelect has a worst case of O(n^2), that is typically avoided and there is an average case of O(n).

QuickSelect2:

Although the key is swapped out for a vector of keys, the worst case of QuickSelect2 is still O(n^2) and the average case is O(n).

CountingSort:

The hash variation of CountingSort has a time complexity of O(n+klogk) where k is the number of unique values. This is because the comparison based sort is done on value-count pairs.

3.)For each file, as the number of integers increased the percent that were unique decreased. The first file was mostly unique values, while by the last one it was a small fraction of the total integers. When there were mostly unique values, CountingSort was the slowest. This makes sense as the time complexity of CountingSort is dependent on the number of unique values, which is O(n+klogk). Since it is n+klogk and not just klogk this was slower than the other methods for the first txt file. By the last txt file, CountingSort was no longer the slowest and was a good amount faster than std sort. At this point, the number of unique values were such a small percentage of the total that the O(n+klogk) time complexity beat out std sort's O(nlogn), which will always act accordingly to the total values, and isn't dependent on unique ones. For the other methods, as expected their time did not seem dependent on unique values. QuickSelect1 was the fastest through every txt file and QuickSelect2 was second fastest. While those two had the same time complexity, it's understandable why QuickSelect2 is slower because now we have to iterate through a vector to access keys, which will cost us extra. In addition, we must perform more recursions due to the multiple keys on both sides. Even though we called QuickSelect1 three times and QuickSelect2 once, this didn't seem to cost us too much and QuickSelect1 maintained the fastest.

4.) Prior to this project, I would have never really thought about how the number of unique values in a dataset would impact performance. The implementation and analysis of CountingSort made me realize that we can't think about these methods in a vacuum, and we have to act accordingly depending on something such as the number of unique values. If I had a set that was 80 percent unique values, I'd know to stay away from the hash variation of counting sort, since it does not deal well with many unique values. However, if it was a dataset with only 5 percent being unique values, I'd know that countingSort would perform relatively better as the percentage of unique values went down.