Fardeen Bhuiyan

Class: CSCI 335 Software Analysis and Design III

Professor: Justin Tojeira

Project 3 Algorithms Analysis Report

StdSort Method (in microseconds)

	Trial 1	**Trial 2**	**Trial 3**
testinput.cpp (1k inputs, 787 unique values)	74	80	64
testinput2.cpp (100k inputs, 3588 unique values)	4631	4685	4621
testinput3.cpp (10M inputs, 4776 unique values)	71,557	71,470	72,338

QuickSelect1 Method (in microseconds)

	Trial 1	**Trial 2**	**Trial 3**
testinput.cpp (1k inputs, 787 unique values)	122	124	122
testinput2.cpp (100k inputs, 3588 unique values)	7326	7434	7321
testinput3.cpp (10M inputs, 4776 unique values)	125,319	126,086	125,374

QuickSelect2 Method (in microseconds)

	Trial 1	**Trial 2**	**Trial 3**
testinput.cpp (1k inputs, 787 unique values)	100	117	115
testinput2.cpp (100k inputs, 3588 unique values)	4783	4792	4613
testinput3.cpp (10M inputs, 4776 unique values)	67,350	67,094	67,068

CountingSort Method (in microseconds)

	Trial 1	**Trial 2**	**Trial 3**
testinput.cpp (1k inputs, 787 unique values)	779	694	832
testinput2.cpp (100k inputs, 3588 unique values)	15,612	15,543	15,185
testinput3.cpp (10M inputs, 4776 unique values)	227,291	228,616	228,657

Analysis

std::sort() or standard sort is a built-in sorting method in C++ that utilizes introspective sort. Introspective aka introsort is a hybrid algorithm that starts off with quicksort, turns to heapsort when the number of elements being sorted exceeds the recursion level threshold, and then ends off using insertion sort once the number of elements being sorted reaches below a certain threshold. This hybrid sorting is useful because quicksort is on average the fastest pure sorting method with an average of O(n log n). Heapsort is very useful to switch over to because although quicksort has an excellent average time, its worst case time is a very poor O(n^2). Heapsort's worst case on the other hand is just O(n log n). So, switching from quicksort to heapsort after O(log n) operations optimizes the performance). Insertion sort is a great last step because it has no overhead memory, and is a very fast algorithm for small amounts of data. Here, insertion sort would be O(n) since we already sort through most of the data before then. Thus, the total complexity of std::sort() would be O(n log n).

In our primary function that was timed (void stdSort(...)), we start off std::sort(data.begin(), data.end()) to sort the vector from smallest to largest values at an average complexity of O(n log n). Accessing the minimum and maximum each were O(1) constant time, using the front() and back() vector operations. We call the stdSortMiddle function 3 times to calculate the 25th percentile, 50th percentile, and 75 percentile of the vector. Each call was also O(1) constant time as the function primarily accesses indices in constant time. So outside of printing out outputs, the total calculated complexity of the **void stdSort(...) method is O(n log n) average time.**

In quickSelect1.cpp, the quick_select(std::vector<int> & data, int left, int right, int k) function is called 3 times in quickSelect1 (const std::string & header, std::vector<int> data). Within quick_select(std::vector<int> & data, int left, int right, int k) if the data is greater than 20 inputs it utilizes quick selection, while 20 or less inputs utilizes the insertion sort method. In quick selection, the pivot is found through calling the median3 function to find the most optimal pivot to start with. This has a complexity of O(1). We have an infinite for loop that breaks only when reaching the break operation. Within that for loop, we have two while loops that check while the data on the left of the pivot is greater than the pivot and while the data on the right side is less than the pivot, and if the right current element is less than the left current element, then you swap them. Recursion of quick_select(std::vector<int> & data, int left, int right, int k) is then called so that we keep partitioning on the left and right side (depending on whether the target k is in the left or right side, respectively) until our target is finally in the correct position.

Unlike quicksort, quickselect implementation utilizes one recursion call instead of two. The average time for quickselect is O(n). This is demonstrated in quick_select(std::vector<int> & data, int left, int right, int k) where the recursion calls take place. If we have a good pivot (from the medianOf3 method), this should only by O(n). A bad pivot can result in a very large amount of recursions.

Finally, when there are 20 inputs or less, the insertion method is used which has a worst case time complexity of O(n^2), but the size is so small in proportion to the entire data that it can be seen as negligent.

Assuming the input size is over 20, **that makes the quickselect algorithm implementation have an average case of O(n). The worst case of O(n^2)** can occur if the worst pivot is constantly chosen (worst as in smallest or the largest value of each recursion), and this can lead to large amounts of searching and swapping.

The QuickSelect2.cpp is fairly similar to QuickSelect1.cpp, however we are using multiple keys in one function call for the quick_select function. Our new quick_select function has a vector of keys which contains the minimum, 25th, 50th, 75th percentiles, and the maximum. Through this, void quickSelect2 (const std::string & header, std::vector<int> data) calls void quick_select(std::vector<int> & data, int left, int right, std::vector<int>& keys) only once when the input size is greater than 20, and the insertion sort function is called at an input less than or equal to 20.
Once again, in quick_select(std::vector<int> & data, int left, int right, std::vector<int>& keys), we find the pivot through the median3 function. We utilize the same for loop and inner while loops, and std::swaps. Now the pivot should be correctly placed.
However, we have two new vectors. Each one is supposed to contain the indices to the left and to the right of the pivot, respectively. But not all indices. Only the indices located at each key's percentile of the key vector. For example, the 25th, 50th, 75th percentiles are indices placed in the left vector if they are less than the pivot index. Finally, instead of checking if each individual key is less than or greater than the pivot, we check if the left and right vectors containing the keys are not empty. If they are not empty, we call recursion on the respective side. Once again, in this function if the input size is less than or equal to 20, we call insertion sort instead of going through quick select. Insertion sort could have a worse case of O(n^2), but the input size is too small to be too impactful on the entire complexity.
Assuming we choose a good pivot (which is likely because we are using the median of 3 method), the recursion calls would lead to this **algorithm being O(n) average case.** If the pivot is too large or too small, **the worst case can occur in which the algorithm would become O(n^2).**

The CountingSort.cpp we implemented utilized hashing. A hashmap is created. The function loops through the entire vector of data and tries to find if the hashmap already contains the element. If it does not contain the element, the hashmap inserts the data as a key, and sets the value as 1. To understand this, each slot of the hashmap contains a pair: a key, and a value. In the context of this function, the key will contain the seal's weight (5637 for example), while the value of the pair will contain the amount of times this number has shown up (ex: there are 3 seals with this same weight). If the hashmap already contains this seal weight, we increment the existing pair's value. The concept behind this is that one slot can "contain multiple instances of the same weight". Now of course, the hashmap is gonna have way less slots than the original data vector, so we place all the hashmap's key-value pairs into a keyValuePairs vector. We use std::sort (which has an average complexity of O(n log n) to sort this vector. We shall change the n to v to signify that these are now unique values. We use a for loop to iterate through this vector and find which element contains the minimum, 25th, 50th, 75th percentiles, and the maximum. A unique counter increments once per iteration. The for- loop would be an O(v) complexity. However, the addition of the std::sort() function (which is O(v log v)) would allow the **countingSort method to work on an average case of O(v + v log v) complexity, which is practically O(v log v) complexity.**

Observations and Understanding:

At face value, countingSort didn't appear like it would match up to the other sorting algorithms. First, unlike quickselect, countingSort is a sorting algorithm which means it goes through every element in the data. QuickSelect1 and QuickSelect2 are selection algorithms and looked very optimal in partitioning the data and were even more effective when a good pivot was chosen (which typically results from the median of 3 method). QuickSelect2 also looked like it would be more effective than QuickSelect1 because it checks multiple positions (keys) in one call. It was, however, difficult to infer which method would be faster between stdSort and QuickSelect2, since stdSort uses introspection hybrid sorting which is very fast in its own way.
In the end, QuickSelect 2 was a bit faster than stdSort when it came to the largest test input size, but was similar with the other input sizes. Perhaps an even larger input would yield the same results to a higher degree. QuickSelect1 came in third, with countingSort coming in last.

More duplicate values in a set of inputs can be detrimental to the efficiency of quickselect. Quickselect tries to partition data based on whether the surrounding elements are less than or equal to the pivot. Unique values help secure more balanced placement. Let's say the pivot has a duplicate and now we are comparing the duplicate with the pivot and are trying to place it in the correct position. We don't make much progress in this step because we need to compare the duplicate afterwards once again. Too many duplicates can push the O complexity closer to O(n^2).
For stdSort, less unique values would be detrimental to the quicksort component. But, the heapsort component is not so detrimentally affected by duplicate values as the heap restructuring process should still flow seamlessly. The insertion sort component actually favors duplicates because less comparisons need to be sorted one at a time. Finally, for the countingSort with hashmap implementation, both unique elements and duplicates work well because unique keys are simply stored in their own slot with a value of 1, and duplicate keys are placed in the same slot with a value greater than 1.

<u>Side Note Below:</u>

Even though it doesn’t currently appear like that, I had measured the timing for each measured cpp file without including the character outputs (“std::cout”). If there were any lines in the way of that, I had commented out the character output before measuring.