COMP2521 Sort Detective Lab Report

by Paul Hayes (z5303576)

This lab aims to analyse two sorting algorithm's performance without access to the source code, and to determine what specific implementation was used in each program. The choices are between Bubble sort, Insertion sort, Selection sort, Merge sort, Naive quicksort, Median-of-three quicksort, Randomised quicksort and Bogosort.

Experimental Design

Performance

I measured how each program's execution time varied as the size and initial sortedness of the input varied. I used the following kinds of inputs:

1. A list of random integers (4,1,3,5,2…).
2. A list of sorted integers (1,2,3,4,5…).
3. A list of reverse sorted integers (...5,4,3,2,1).
4. A list of numbers that are all equal (1,1,1,1,1…).

I used these test cases because it is known that some algorithms are adaptive, meaning that they have different best and worst case time complexities depending on the input(random, sorted, reverse). This information gives insight into what algorithm might be in use. Additionally, by testing incrementally larger data sets, the time complexity performance for each program is more noticeable. That is, it's easier to differentiate an O(n^2) and O(nlogn) behavior when the results are plotted on a graph.

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test 3 times as is good scientific practice.

Stability

I also investigated the stability of the sorting programs by using inputs that show if they have been moved in the sorting process.

5. List of Indexed numbers (4 1,1 1, 1 2, 4 2, 1 3, 2 1,...).

Stable and unstable algorithm will sort the first key the same but can differ on the second key. For example, If you sort on alphabetically first names then sort the output data on last names, the unstable algorithm *might* alter the order of the first sort. Constructing the data with indexes and carefully selected pairings I will be able to observe if the program has executed swaps.

Other techniques used were to compare the results of Linux's inbuilt sort with that of Program A and Program B.

Experimental Results

For Program A, I observed that as the input size increased for all cases except the sorted and equal input, the time complexity behaved in a quadratic manner. This would indicate an O(n^2) sort such as bubble, insertion, selection or shell sort. The result also indicates adaptivity with a best case scenario of for an already sorted input and worst case of a reverse sorted. This suggests bubble sort or insertion sort. Further methods need to be developed to distinguish between these.

These observations indicate that the algorithm underlying the program has these characteristics.

- Time complexity: O(n^2)
- Stable
- Best case = sorted
- Worst case = reversed

For Program B the larger input sizes did not significantly impact the runtime of the sort. When compared with the result of program A's volume tests this is very apparent. Furthermore, program B's closely matched linux's inbuilt sort, which is a known nlogn algorithm. Thus we can conclude that this algorithm most likely also has a time complexity of nlogn. This suggests it could be a variation of quicksort or merge sort.

The algorithm appears to be stable as can be seen in the results table. Suggesting merge sort to be the most slightly used algorithm. The worst case for merge sort is when it has to do the most amount of comparisons which could occur in the random input but is not definitive.

These observations indicate that the algorithm underlying the program has these characteristics

- Time complexity: O(nlogn)
- Stable

Conclusions

On the basis of our experiments and our analysis above, we believe that

- sortA implements the *bubble sort or insertion* sorting algorithm
- sortB implements the *merge* sorting algorithm

Appendix

Program A
Time Complexity

InputSize	random	sorted	reversed	equal
10000	0.443s	0.004s	0.494s	0.005s
20000	1.808s	0.009s	1.966s	0.009s
40000	7.688s	0.011s	7.834s	0.011s
80000	32.377s	0.020s	35.417s	0.023s
1000000	Not tested	0.276s	Not tested	0.264s

Program B

size	random	sorted	reversed	equal
10000	0.006s	0.005s	0.004s	0.005s
20000	0.015s	0.011s	0.011s	0.010s
40000	0.029s	0.015s	0.013s	0.013s
80000	0.031s	0.023s	0.021s	0.024s
1000000	0.409s	0.313s	0.305s	0.312s

Stability

Input	Output Program A	Output Program B
2 1	1 1	1 1
2 2	2 2	2 2
5 2	2 1	2 1
5 1	5 1	5 1
1 1	5 2	5 2

Observe the in row 2 and 4, if the algorithms do not have to move the value they leave it in place which suggests stability.