COMP2521 Sort Detective Lab Report

by Bonita Chan Sze Hang, Haoran Wang

In this lab, the aim is to measure the performance of two sorting programs(ie. analyse their time complexity, stability and adaptivity), without access to the code, and determine which sort algorithm each program implements.

Experimental Design

There are two aspects to our analysis:

- determine that the sort programs are actually correct
- measure their performance over a range of inputs

Hypothesis

we expect the

- Oblivious Bubble Sort to be unstable and not adaptive, with the time complexity O(n^2) .
- Bubble Sort With Early Exit to be stable and have average time complexity O(n^2) , and best-case time complexity O(n) because of its adaptivity.
- Insertion Sort to be stable, adaptive and have quatratic time complexity in worst and average cases, but O(n) time complexity in best case.
- Selection Sort to be not stable, not adaptive and have quadratic time complexity in best, average and worst cases.
- Merge Sort to have time complexity of n log n regardless of the input, and be stable, not adaptive.
- Vanilla Quick Sort to be n log n in best and average case and n^2 in worst case , and not stable and not adaptive.
- Quick Sort Median of Three to be nlogn in best and average case and n^2 in worst case , and not stable and not adaptive. since the pivot chosen is the median of of the first, middle and last items, the worst case is very unlikely to happen.
- Randomised Quick Sort to be nlogn in best and average case and n^2 in worst case , and not stable and not adaptive.we choose a random pivot. This makes the quadratic worst case almost impossible.
- Shell Sort Powers of Two Four to have a quadratic worst case and linear best case and have non- stable and adaptive property.
- Shell Sort Sedgewick (Sedgewick-like) to have better performance than Shell Sort Powers of Two Four.
- Psuedo-Bogo Sort to be executed in linear time if the input is sorted, or else the expected time complexity is n.n!, but could be infinite in the worst case. It is unstable.

Correctness Analysis

To determine correctness, we tested each program with using three different types of data, which are random data, reservely sorted data and ascending sorted data.

Then, we observe whether the outputs of each sort programs are perfectly sorted as we expected by comparing the results with the unix sort in the machine.

Performance Analysis

We tested our experiments with 4 types of numerical inputs:

1. Ascending sorted inputs -- which can be used to determine if the sort is adaptive. If there is a significant decrease in the runtime of program relative to the time spended on the random sort testing , it can be conclude that sort has the adaptive property.
2. Random inputs -- which can be used to test the performance of the given sort in average case(ie. to record the runtime). Trying to test with many various inputs the plotting the result by line graph can helps in identifying relationship between the size of input and its time complexity.
3. Descending ordered inputs -- which are used in order to attempt to identify the worst-case time complexity of the given sort, since some of sort algorithms will perform worst in the revserly order input such as bubble sort and insertion sort.
4. inputs with duplicates -- which aims to determine whether the given sort has the stable property. If the algorithm always preserved the order of the duplicates no matter what kind of inputs are used, we can draw a conclusion that the sort is stable.

By using these four kinds of inputs, we can generally distinguish sort by its time complexity and its characteristics(stability and adaptivity) at the first step. However, if the some sorts have same performance or characteristics, more specific tests are required later.

Experimental Results

Correctness Experiments

As we expected, two given sorts are able to sort the various inputs perfectly and on all of our test cases, a total of 3 times were carried out in order to make sure that the results are accurate enough.

Performance Experiments

For **Program A**, we observed
--- An O(n^2) time complexity when plotting the execution time of the algorithm against the size of the random input set. (graph 4)
-- An extremely short execution time, once the inputs are ascending sorted and rather long execution time when the inputs are reversely sorted. (graph 4)

These observations indicate that the algorithm underlying the program has the characteristics that it is adaptive and has an O(n^2) time complexity in average case. Thus, it can be concluded that this sort program is either insertion sort or bubble sort. Then, we test the program according to the distinguishing characteristics between the insertion sort and bubble sort, which is that the insertion sort will have time complexity O(n^2) if the inputs data are like 10, 1, 2, 3, 4, 5, 6, 7, 8, 9 this pattern with the first element being largest and the rest of elements being ascending order, but bubble sort will be O(n) since the bubble sort will carry the first element to the last one at the first iteration, then it will finished execution as it is sorted. That is to say, bubble sort will be way faster than insertion sort with this kind of specific inputs.
After plotting the graph(graph 5), we observe that the runtime has the quadratic property against the increasing inputs. thus, the sort A program is insertion sort.

For **Program B**, we observed that for the three different orders of data, the time kept to be very small for small datas, until it went to the data size of 50000. The Program is also able to handle with large data size during a relatively short runtime.
This observation indicate that the algorithm underlying the program has an O(nlogn) time complexity in all the cases.(ascending sorted, reversely sorted and random). Therefore, the sort program can be either quick sort or merge sort.
Then, we generate some data with duplicates to determine the stability of given sort and the result is the give sort are not able to preserve the order of original inputs. (graph 8 and graph 9) ,then it is not stable and maybe one of two quicksort (randomized quick sort or quicksort with median of three)
However, we also observed that the running times of program in ascending sorted inputs and descending sorted inputs are always shorter than running times of program in random inputs, which can be explained by the characteristic of quick sort median three because in the ascending or descending cases, quicksort median three can partition the array to equal size, which is very efficient.
Thus, it can be concluded that it is quicksort median three.

Conclusions

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

- ProgramA implements the *insertion* sorting algorithm

- ProgramB implements the quicksort median three sorting algorithm

Appendix

Data size	Random	Ascending	Descending

1000	*0.11*	*0*	*0.33*
2000	*0.46*	*0*	*2.34*
3000	*1.30*	*0*	*7.34*
4000	*2.49*	*0*	*16.89*
5000	*6.62*	*0*	*32.80*
6000	*9.61*	*0*	*Can't be determined*
10000	*37.3*	*0*	*Can't be determined*
1000000	*Can't be determined*	*0.16*	*Can't be determined*

Graph 1: The time of our tests for Program A

Data size	*"Nearly sorted"* Ascending data
100	*0*
1000	*1.52*
2000	*11.63*
3000	*40.26*
4000	*94.12*
5000	*181.96*

Graph 2: The time for sorting an almost sorted list for Program A

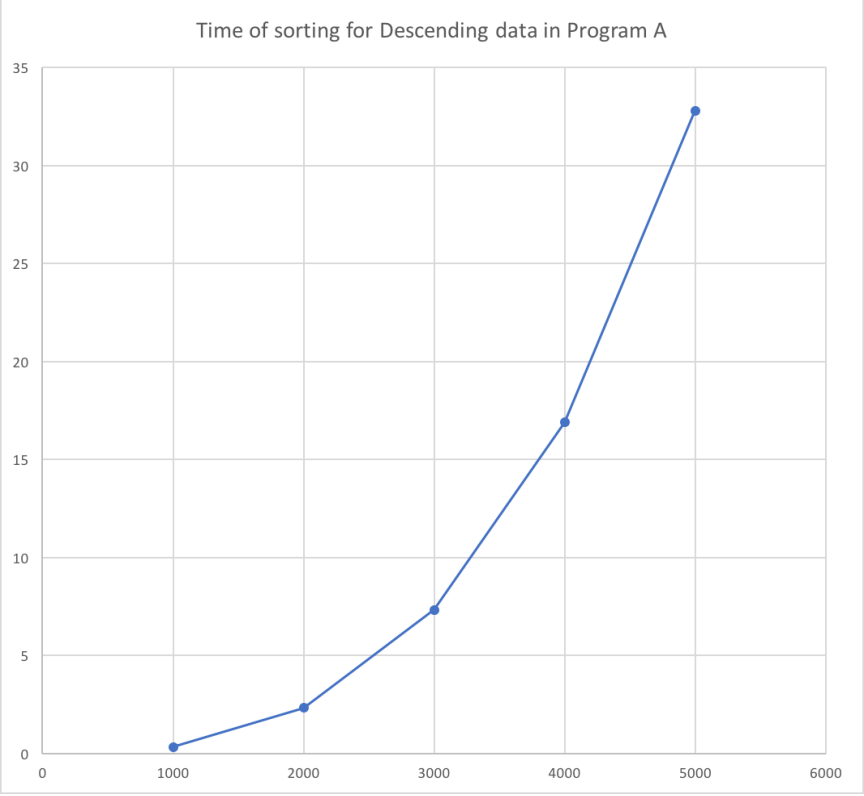
Data size	Random	Ascending	Descending
50000	*0.02*	*0.02*	*0.01*
60000	*0.03*	*0.03*	*0.01*

70000	*0.04*	*0.03*	*0.02*
100000	*0.05*	*0.04*	*0.03*
200000	*0.10*	*0.10*	*0.07*
300000	*0.15*	*0.13*	*0.12*
500000	*0.24*	*0.18*	*0.16*
800000	*0.32*	*0.27*	*0.23*
1000000	*0.40*	*0.36*	*0.28*

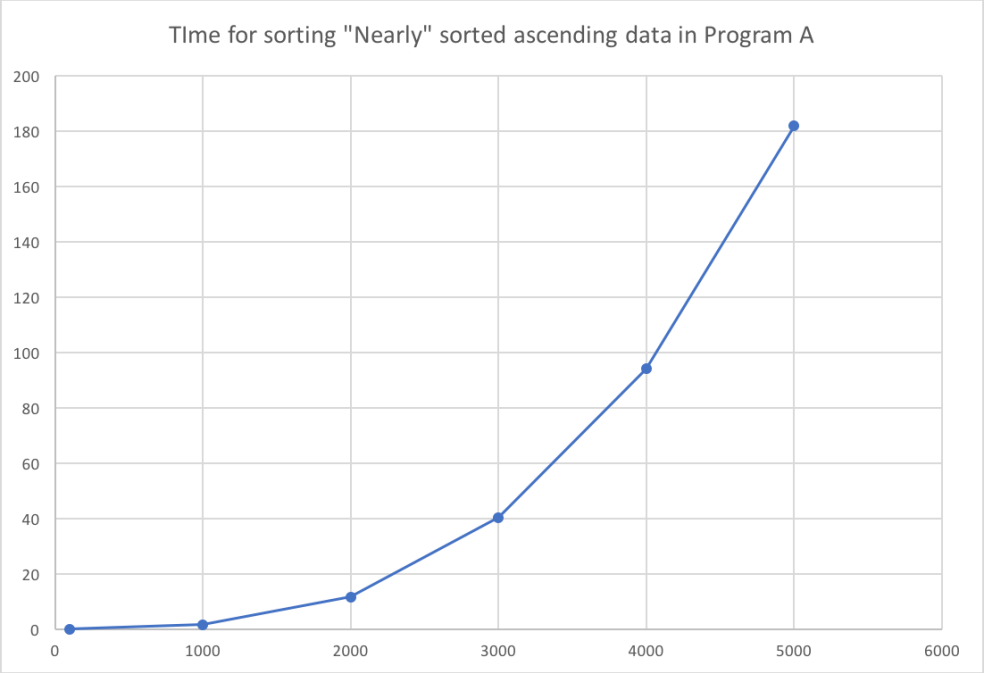
Graph 3: The time of our tests for Program B

Graphs for Program A

Graph 4: The time of sorting for both Random and Ascending data

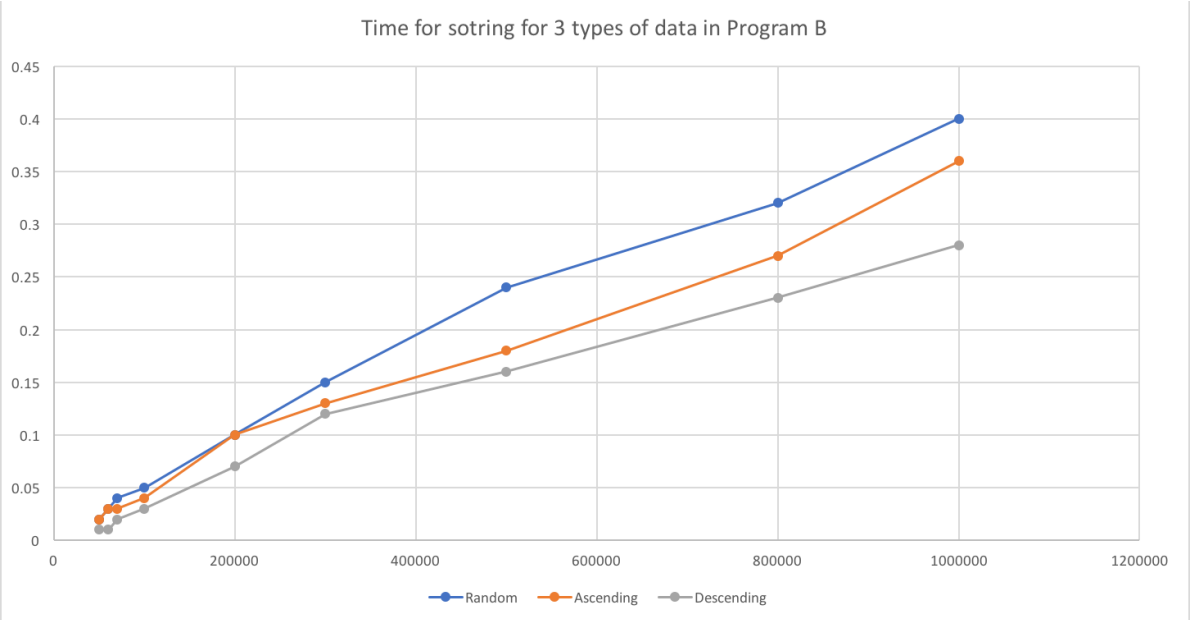


Graph 5: The time of sorting for Descending data

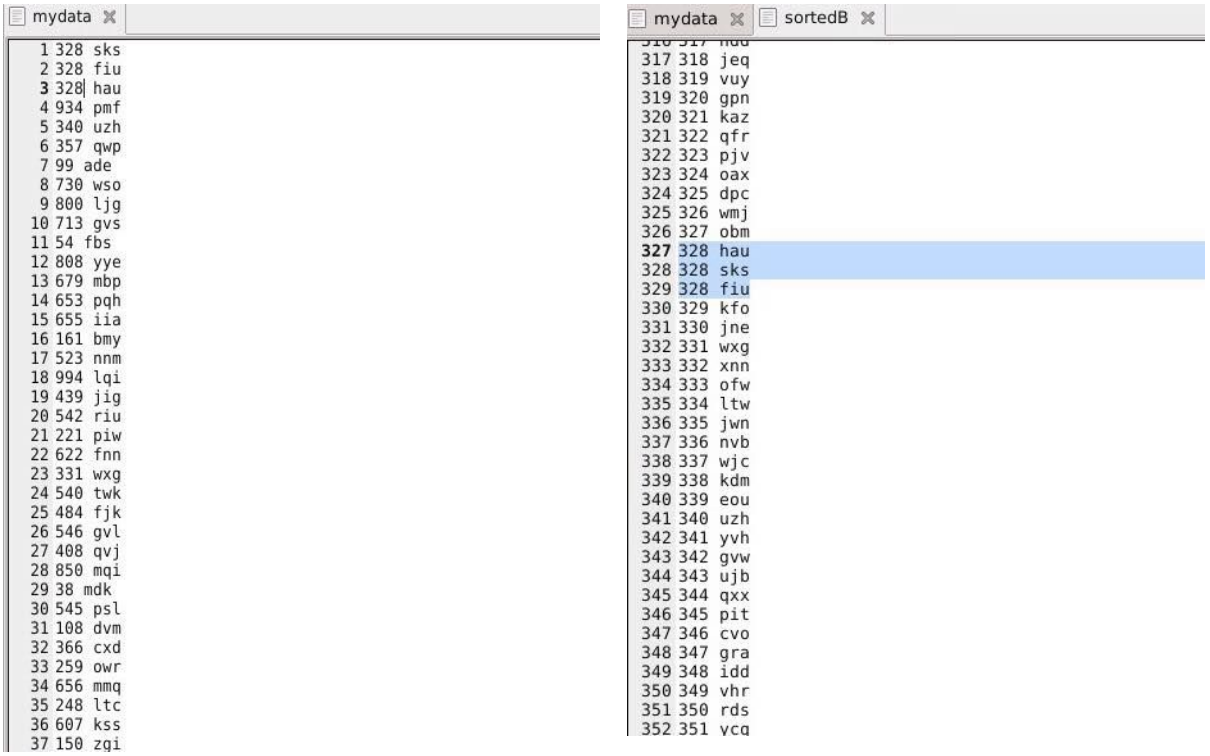


Graph 6: The time for sorting nearly sorted data

Graphs for Program B



Graph 7: The graph of comparison between times for sorting Random, Ascending and Descending data



Graph 8 and 9: Checking whether the sort will sort it according to their order (to check if the sort is stable)