COMP2521 Sort Detective Lab Report

by Johnson Sim

In this lab, the aim is to measure the performance of two sorting programs, without access to the code, and determine which sort algorithm each program implements.

Experimental Design

There are multiple aspects to my analysis:

1.First we tested whether each of the sorting algorithms are correct.

2. Then we tested each algorithms with large sizes to see which complexity group it is in

3. Then we did specific results to test stability and randomness

Correctness Analysis

To determine correctness, we tested each of the programs with a large input of random 100000 and compared them with the linux command diff.

We chose these inputs because it has a large enough value to increase confidence of correctness and map edge cases.

On all of our test cases it shows that no difference was found so we can be fairly confident the program was correct.

Performance Analysis

In our performance analysis, we measured and analysed how sortA ,sortB and sort(linux) execution time varied as the size and initial sortedness of the input varied. We used the following kinds of input of 100000 and 10000. We used sort as it had a know time complexity of nLogn.

We used these test cases because it gave a big enough variance to see the difference of each program (sort .sortA and sortB).

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times because the cpu is constantly working on different processes so multiple test cases gave a more accurate result

We were able to use up to quite large test cases without storage overhead because (a) we had a data generator that could generate consistent inputs to be used for multiple test runs, (b) we had already demonstrated that the program worked correctly, so there was no need to check the output.

We also investigated the stability of the sorting programs by creating a file of numbers with duplicates and keys , and input them in the sorting algorithms and seeing there output

We also investigated whether an algorithm was randomised by testing the same seed multiple times

## **Experimental Results**

### **Correctness Experiments**

An example of a test case would be using the commands

“./gen 100000 R 20 | sort -n > realsort”

“./gen 100000 R 20 | ./sortA > sorta”

“Diff sorta realsort”

This allows us to sort a large amount of numbers and see the difference to a know correct sorting algorithm.

and the results of that test is that both algorithms are correct

### **Performance Experiments**

For Program A, we observed that it was not stable , not adaptive and had complexity o n^2 which only gave us a conclusion oblivious bubble sort algorithm

For Program B, we observed that the program was stable ,had nlogn complexity ,it did not have randomised results and did not take long to do descending list. Therefore it would have to be merge sort.

## **Conclusions**

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

* ProgramA implements the Oblivious Bubble Sort sorting algorithm
* ProgramB implements the Merge Sort stable sorting algorithm

RESULTS/WORKING OUT

**ASCENDING**

1000000

./sorta = 11.472 -shows non adaptive

./sortb = 0.084 - Shows

Sort =0.088 - shows

**DESCENDING**

1000000

./sorta = 21.472 -Shows it has complexity of n^2

./sortb = 0.08

Sort = 0.096

**Random**

1000000

./sorta = 29.8

./sortb=0.176

Sort = 0.14

SO FAR GIVEN RESULTS A MUST BE N^2 AND B MUST BE NLOGN

Check stability by comparing it with sort

./sortA cant be bubble sort and insertion sort as it is not adaptive ,.

./sortA showed that it was instable with a custom list

./sortA also cant be selection sort as it is different to each type

Oblivious Bubble Sort

./sortB cant be bubble sort and insertion sort as it has complexity nLOGn and non adaptive

./sortA also cant be selection sort as it is different to each type

.

SORT A

* Oblivious Bubble Sort
  + unstable and unoptimised bubble sort

SHOWED THAT IT IS UNSTABLE AND UN OPTIMISED

* Bubble Sort With Early Exit
  + stable bubble sort that terminates when there have been no exchanges in one pass

CANT BE THIS SINCE ITS NOT STABLE

* Insertion Sort
  + standard insertion sort

CANT BE THIS SINCE IT ISNT STABLE

* Selection Sort
  + standard selection sort

CANT BE THIS SINCE ITS DIFFERENT FOR EACH INPUT

* Shell Sort Powers of Two Four
  + shell sort with intervals ..., 4096, 1024, 256, 64, 16, 4, 1

CANT BE THIS SINCE ITS NOT ADAPTIVE

* Shell Sort Sedgewick (Sedgewick-like)
  + shell sort with intervals ..., 4193, 1073, 281, 23, 8, 1

CANT BE THIS SINCE ITS NOT ADAPTIVE

SORT B

* Merge Sort stable
  + normal merge sort

(CAN ONLY BE THIS SINCE IT IS THE ONLY SORTING WHICH IS STABLE)

* Vanilla Quick Sort
  + normal quick sort (pivot is the last element) instable (n^2 for decreasing)

CANT BE THIS SINCE IT SORTS DESCENDING RELATIVELY FAST

* Quick Sort Median of Three instable
  + pivot is the median of first last and middle elements
* Randomised Quick Sort instable same seed
  + list is shuffled then vanilla quick-sorted

CONCLUSION SORTA WOULD BE OBLIVIOUS BUBBLE SORT

SORT B WOULD BE MERGE SORT