# COMP2521 Sort Detective Lab Report


### by Gary, Sam


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 two aspects to our analysis:

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

### Correctness Analysis

To determine correctness, we tested the programs Sort A and Sort B to sort a bunch of numerical list with following properties :

1. Random numerical list
2. Reverse sorted numerical list
3. Sorted numerical list

And each list has different list size: 1000, 10000, 100000. Program Sort A and Sort B will sort with the same list each time and transfer the results to a .txt file that will compare with a sorted .txt file which is sorted by unix sort by Linux default. For a valid result, this process will be progressed 10 times.
The idea we use this design to test correctness because of comparison with sorted result by a successful program which is unix sort by Linux default and the results of SortA and SortB can determine the correctness of results of SortA and SortB. Also we tested with different properties of numerical list and ascending data size to determine whether sortA and sortB can produce a correct result or not, to improve the accuracy
Finally, we compare the result of unix sort and SortA And SortB, if there is no difference between the results, the we can conclude the correctness of the results of SortA and SortB

### Performance Analysis

In our performance analysis, we measured how each program's execution time varied as the size and initial sortedness of the input varied. We used the three type of input numerical list: random, reverse order sorted, sorted and different data size: 1000,10000,100000,100000 to get the sorting operation time of each program

We used these test cases because we tested with different data size of numerical list can help us to distinguish quicker sort type and those slower. And different type of numerical used to demonstrate the properties of sort program.

We tested a large data size to distinguish some of the sort, when deal with a large data, if the program sorted with long time in random order, the sort may be 1. 2. 3. 4. 11. Since they

all have a average operation time n^2. Otherwise if operation time is short enough then the program may be other quicker sort type.

Using the stability can also distinguish some of sorting type since some of the program is unstable.

There are 11 types of possible sort program and how to determine them:

1. Oblivious Bubble Sort
    a. - unstable and unoptimised bubble sort
    b. Operation in a long time since it needs to take care of each element
2. Bubble Sort With Early Exit
    a. - stable bubble sort that terminates when there have been no exchanges in one pass
    b. Operation in a long time, but short time in a sorted list
3. Insertion Sort
    a. Insertion sort is a stable sort as it progress sorting from element1 to element n in order, but it needs to check out each of element, so it will take a long time n^2 every time as we processing

4. Selection Sort
    a. Insertion sort is a stable sort since it's sorting process is to find the stastific element from element 1 to n in order, such as first time will insert the smallest element in the list then insert to first place. During the process it will produce a stable sorted list. It will take n^2 time because it search the element in the whole list each time. The way i want to ensure the list is insertion sort, i will compare the timing result of sorted and reverse order sorted.
    b. Produce a sorted list first, then swap the element of smallest and largest, i am excepting the operation time will be large like n^2

5. Merge Sort
    a. Stable sort that the implementation preserves the input order of equal elements in the sorted output.
    b. Operation in short time compare with others
6. Vanilla Quick Sort
    a. normal quick sort (pivot is the last element)
    b. unstable , since the pivot is starting in some certain element to cause an unstable result
    c. Operation in short time compare with others
    d. Process with a reverse order sorted list will take long time
7. Quick Sort Median of Three
    a. - pivot is the median of first last and middle elements
    b. Unstable sort and process in a short time in any types of list
8. Randomised Quick Sort
    a. - list is shuffled then vanilla quick-sorted
    b. Unstable - quick sort properties explained above
    c. Short time but unstable time since the pivot is random

9. Shell Sort Powers of Two Four ?
    a. - shell sort with intervals ..., 4096, 1024, 256, 64, 16, 4, 1
    b. Stable since the properties of merge sort, shell sort is just a improvement of insertion sort
    c. Operation in short time
10. shell Sort Sedgewick (Sedgewick-like) ?
    a. - shell sort with intervals ..., 4193, 1073, 281, 23, 8, 1
    b. Stable since the properties of merge sort, shell sort is just a improvement of insertion sort
    c. Operation in short time
11. Psuedo-Bogo Sort
    a. - choose two random array elements, if out-of-order then swap, repeat
    b. Operation in long time since it is random picking
    c. Unstable because it implement the element randomly so produce an unstable result

We were expecting to test the stability of program to distinguish what part of the program should be, like stable part or unstable part. Then we would use a bunch of input data and different size to get the operation time to distinguish is the program belong to a quicker sorting type part or slower part. Then we would edit a certain numerical list and obverse the operation time to find the properties of program. Finally, with these three steps we can almost lock the program in a small range of sorting type.

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times to improve the accuracy and validity of the timing 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.

1.stability

We also investigated the stability of the sorting programs by using the program to sort with a large data input, so we tested the program with different properties of numerical list which are random, reverse order sorted and sorted and we choose the input list size is 500000. Then the result will be compared with the result of unix sort in Linux, if there is any different between these two result, we can conclude the program is unstable. This process will be progressed multiple times to improve the accuracy and validity of conclusion.

Since we had already demonstrated the the program worked correctly, the way to demonstrated the program is unstable is that program will operate the same number and swap them, they these number have different seeds. E.g.

| **Unsorted** | **Sorted** |
|---|---|
| **3 a** | **1 d** |
| **2 b** | **2 c** |
| **2 c** | **2 b** |
| **1 d** | **3 a** |

This example will conclude the program is unstable.

# Experimental Results

## Correctness Experiments

An example of a test case and the results of that test is ...
On all of our test cases, the result file of program A and program B is no different with the results producing by unix sort.

## Performance Experiments

For Program A, we observed that is stable and sort slowly
These observations indicate that the algorithm underlying the program A *has the following characteristics* , it is stable and sort in a really slow speed.
For Program B, we observed that is table and sort quickly
These observations indicate that the algorithm underlying the program B *has the following characteristics* is quick and stable

# Conclusions

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

- ProgramA implements the *bubble with early exit* sorting algorithm
- ProgramB implements the Merge Sort sorting algorithm

# Appendix

correctness:

| | **input size** | **number of times** | **list type** | **diff with unix sort** |
|---|---|---|---|---|
| **sortA** | 1000 | 10 | random | NO |
| | 1000 | 10 | reverse order | NO |
| | 1000 | 10 | sorted | NO |
| | 10000 | 10 | random | NO |
| | 10000 | 10 | reverse order | NO |
| | 10000 | 10 | sorted | NO |
| | 100000 | 10 | random | NO |
| | 100000 | 10 | reverse order | NO |
| | 100000 | 10 | sorted | NO |
| **sortB** | 1000 | 10 | random | NO |
| | 1000 | 10 | reverse order | NO |
| | 1000 | 10 | sorted | NO |

| | | | | |
|---|---|---|---|---|
| | 10000 | 10 | random | NO |
| | 10000 | 10 | reverse order | NO |
| | 10000 | 10 | sorted | NO |
| | 100000 | 10 | random | NO |
| | 100000 | 10 | reverse order | NO |
| | 100000 | 10 | sorted | NO |

Stability:

| | **input size** | **# of times** | **list type** | **diff with unix sort?** |
|---|---|---|---|---|
| **sortA** | 500000 | 10 | random | NO |
| | 500000 | 10 | sorted | NO |
| | 500000 | 10 | reverse order | NO |
| **sortB** | 500000 | 10 | random | NO |
| | 500000 | 10 | sorted | NO |
| | 500000 | 10 | reverse order | NO |

SortA and sortB are stable

timing:

| | **input size** | **number of times** | **list type** | **operation time(averge)** |
|---|---|---|---|---|
| **sortA** | 1000 | 10 | random | 0 |
| | 1000 | 10 | reverse order | 0 |
| | 1000 | 10 | sorted | 0 |
| | 10000 | 10 | random | 0.24 |
| | 10000 | 10 | reverse order | 0.21 |
| | 10000 | 10 | sorted | 0 |
| | 100000 | 10 | random | 29.32 |
| | 100000 | 10 | reverse order | 20.98 |
| | 100000 | 10 | sorted | 0.01 |
| **sortB** | 1000 | 10 | random | 0 |
| | 1000 | 10 | reverse order | 0 |
| | 1000 | 10 | sorted | 0 |
| | 10000 | 10 | random | 0 |
| | 10000 | 10 | reverse order | 0 |
| | 10000 | 10 | sorted | 0 |
| | 100000 | 10 | random | 0.025 |

| | | | | |
|---|---|---|---|---|
| | 100000 | 10 | reverse order | 0.02 |
| | 100000 | 10 | sorted | 0.02 |
| | 1000000 | 10 | random | 0.355 |
| | 1000000 | 10 | reverse order | 0.25 |
| | 1000000 | 10 | sorted | 0.207 |