COMP2521 - Lab09 - Sort Detective

By Edward Nguyen-Do (z5309199), Christian Nguyen (z5310911)

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

Possible Sorting Algorithms: - *Bubble Sort* - *Insertion Sort* - *Selection Sort* - *Shell Sort* - *Merge Sort* - *Naive Quick Sort* - *Median-Of-Three Quick Sort* - *Randomised Quick Sort* - *Bogosort*

Experimental Design

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

- *Sorted*
- *Reversed*
- *Random*
- *Duplicates*
- *Small input size (5)*
- *Large input size (10000)*

We used these test cases because we wanted to test all possible inputs that would affect the runtime and stability of the algorithm.

Because of the way timing works on Unix/Linux, it was necessary to repeat the same test multiple times because the Unix time command works by sampling and will likely produce different results for the same program run multiple times. We will take an average over a number of timings to account for this.

We investigated the stability of the sorting programs by using input that consists of duplicate numbers with differing keys in order to differentiate between stable and adaptive sorts. For a *stable sort*, if a duplicate precedes the other in the array before it is sorted it should also precede it after it is sorted. If not, then it is an *unstable sort*.

Furthermore, we investigated whether a sorting algorithm was *adaptive* or not. Since an adaptive algorithm should have a faster runtime if the data set has some level of order before sorting, the runtime should be noticeably faster than a randomly ordered data set. A sorting algorithm that has a fast runtime if given a sorted input when compared to given random input indicates that the algorithm is *adaptive*.

We also investigated:

- *Time Complexity* In order to determine the time complexity of the sorting algorithms, we provide a large enough input so that the runtimes are easily distinguishable.

Experimental Results

Observations

Program A

For Program A, we observed that the average times for each small data set case were relatively similar despite whether or not they had duplicates. Similarly, for large data sets we had a similar observation in which duplicates also had a minimal effect on the runtime.

Also, to test stability we manually inserted duplicate numbers with a key "abc" which preceded the generated duplicate number before sorting. After sorting, the duplicate numbers maintained their order and therefore the sorting algorithm is stable.

As for the runtimes when given different degrees of sortedness, for larger input sizes it was evident that the sorting algorithm performed much faster when the input was sorted (with and without duplicates). This tells us that `sortA` is an *adaptive* sorting algorithm since the runtime is much shorter when the input is sorted.

Program B

For Program B, we observed that the runtimes remained very consistent regardless of the size of the data set and if there were any duplicates.

Similar to Program A, to test stability we manually inserted duplicate numbers with the key "abc" which preceded the number before sorting. However, the duplicate numbers did not maintain their order therefore the sorting algorithm is *not stable*.

When given different degrees of sortedness, the relative runtime for the sorting algorithm remained consistent despite whether or not it was sorted, reversed or randomised. Hence, the sorting algorithm is *not adaptive*.

When given ascending or descending data sets, the sorting algorithm runs relatively faster compared to random data sets (random approximately takes double the time).

When given a modified data set in which the median value is the second smallest value, the sorting algorithm displayed minimal variation in runtime.

Deductions

Comparing the runtimes for Program A and Program B with large input size, we can see that Program B is relatively faster and therefore Program B has a time complexity of `O(nlogn)` and Program A has a time complexity of `O(n^2)`. Furthermore, since Program A is also *stable* and *adaptive*, we can deduce that `sortA` is **bubble sort**.

From our observations, Program B is a type of **quick sort** sorting algorithm (naive, median of the three, randomised). According to *Figure 1.3* and *Figure 1.4*, the runtime for the ascending data set is similar to reversed and randomised data sets. Since **naive quick sort** takes the leftmost element as the pivot, in a reversed or sorted array the partitions would be greatly unbalanced and result in a relatively slow runtime compared to random. Hence, the sorting algorithm cannot be **naive quick sort**.

To distinguish the sorting algorithm from **randomised** or **median of three quick sort**, we can refer to *Figure 1.5*. The given data set to `sortB` consisted of a median value which created greatly imbalanced partitions. For example, in a data set which consists of numbers 1 to 9, choosing a median value of 2 results in a left partition of size 1 and a right partition of size 6. If `sortB` was a *median of three quick sort*, the runtime of this manipulated data set would greatly differ from the regular sorted, reversed and randomised data set. Thus, according to *Figure 1.5*, `sortB` generated minimal variation in runtime when given these data sets and is therefore **randomised quick sort**.

Conclusions

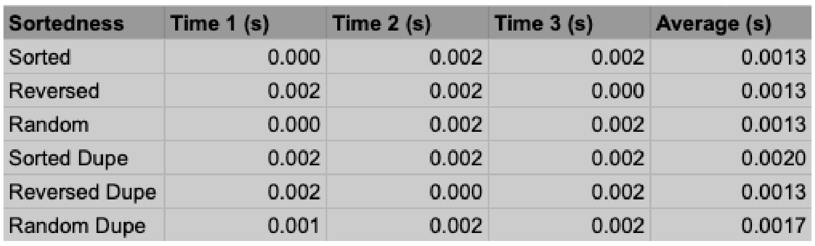
On the basis of our experiments and our analysis above, we believe that

`sortA` implements the **bubble sort** sorting algorithm `sortB` implements the **randomized quick sort** sorting algorithm

Appendix

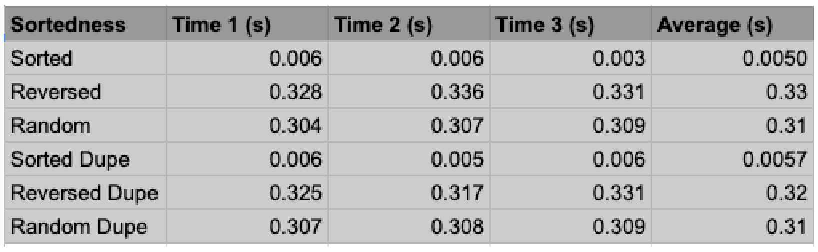
`SortA`

Figure 1.1 - `sortA` Small Input Size - 5



Sortedness	Time 1 (s)	Time 2 (s)	Time 3 (s)	Average (s)
Sorted	0.000	0.002	0.002	0.0013
Reversed	0.002	0.002	0.000	0.0013
Random	0.000	0.002	0.002	0.0013
Sorted Dupe	0.002	0.002	0.002	0.0020
Reversed Dupe	0.002	0.000	0.002	0.0013
Random Dupe	0.001	0.002	0.002	0.0017

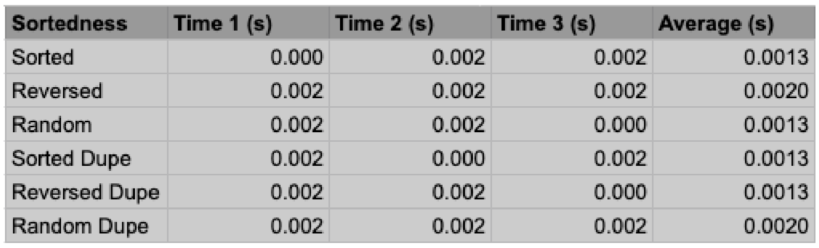
Figure 1.2 - `sortA` Large Input Size - 10000



Sortedness	Time 1 (s)	Time 2 (s)	Time 3 (s)	Average (s)
Sorted	0.006	0.006	0.003	0.0050
Reversed	0.328	0.336	0.331	0.33
Random	0.304	0.307	0.309	0.31
Sorted Dupe	0.006	0.005	0.006	0.0057
Reversed Dupe	0.325	0.317	0.331	0.32
Random Dupe	0.307	0.308	0.309	0.31

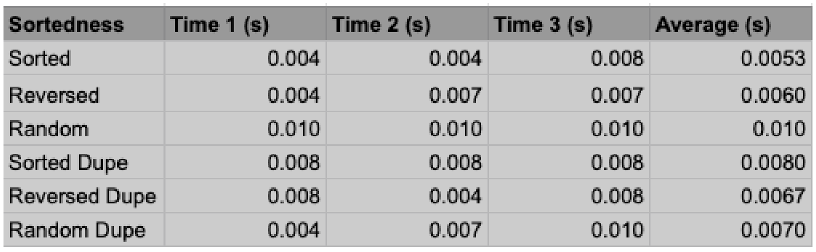
`SortB`

Figure 1.3 - `sortB` Small Input Size - 5



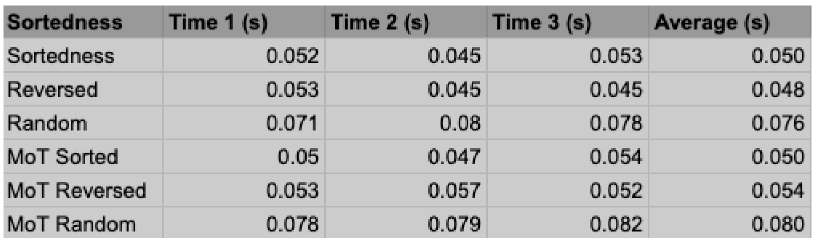
Sortedness	Time 1 (s)	Time 2 (s)	Time 3 (s)	Average (s)
Sorted	0.000	0.002	0.002	0.0013
Reversed	0.002	0.002	0.002	0.0020
Random	0.002	0.002	0.000	0.0013
Sorted Dupe	0.002	0.000	0.002	0.0013
Reversed Dupe	0.002	0.002	0.000	0.0013
Random Dupe	0.002	0.002	0.002	0.0020

Figure 1.4 - `sortB` Large Input Size - 10000



Sortedness	Time 1 (s)	Time 2 (s)	Time 3 (s)	Average (s)
Sorted	0.004	0.004	0.008	0.0053
Reversed	0.004	0.007	0.007	0.0060
Random	0.010	0.010	0.010	0.010
Sorted Dupe	0.008	0.008	0.008	0.0080
Reversed Dupe	0.008	0.004	0.008	0.0067
Random Dupe	0.004	0.007	0.010	0.0070

Figure 1.5 - `sortB` Testing for Median of Three Quick Sort Large Input Size - 250000



Sortedness	Time 1 (s)	Time 2 (s)	Time 3 (s)	Average (s)
Sortedness	0.052	0.045	0.053	0.050
Reversed	0.053	0.045	0.045	0.048
Random	0.071	0.08	0.078	0.076
MoT Sorted	0.05	0.047	0.054	0.050
MoT Reversed	0.053	0.057	0.052	0.054
MoT Random	0.078	0.079	0.082	0.080