Technical Report

January 25, 2022

1 Requirements Summary

The application should enable users to input a list of integers. These numbers will be of an unknown size, and the list will be of an undetermined length. The inputted list shall be ordered, and the execution time of the sorting algorithm must be recorded. The sorted list, execution time, and the order in which the data was sorted, has to be stored within a database. Sorted lists should be retrievable, and presented to the users. Users must be given the option to export all sorted lists to a JSON file. All important information and progress, must be visible to users via a console screen.

2 Summary of Decisions

I made a single assumption, that the inputted values would be given as a comma separated list. This design choice alleviated the need, to iteratively request new input, then append this new input into a list. While this choice simplified the input process, it required extra care to ensure, that all the comma separated values were Integers. Fortunately, the simplistic nature of the task, necessitated no extra technologies beyond native Python modules. This was because, the database specification could be fulfilled with SqLite, and there were built-in algorithms that could effectively sort lists. Furthermore, as no complex UI was needed, a simple and intuitive console user interface was used. I employed a test-driven development methodology, consequently, test cases were developed alongside the source code. Finally, the code documentation and style adhered to PEP standards.

3 Requirement Fulfilment

There were six requirements that needed to be fulfilled for the completion of the task. Whilst they were not grouped into functional and non-functional, nor arranged based upon their interdependency or natural progression, each one, came with design decisions that meaningfully altered the underline technical specifications of the application.

R1. *Allow the user to enter a variable amount of numbers, of any integer value and in random order.*

The user can enter variable length CSV (comma separated value) records. This format expects the numbers to be proceeded by a comma, until the last value is reached, for example 1, 2, 3, 4. The utility of this string format is two fold; Its easy to understand and its a standardized data format. With simple string manipulations *(Class: CustomCSVEntry.raw_csv_input)*, its possible to generate a list of elements that proceed a comma. The result of this progress is list of string elements, however these can be mapped to a function, which converts the strings to integers types. If any of these elements can not be phased into an integer type, an error message will be provided and displayed to the user.

R2. *Sort these numbers in either ascending or descending order – the user should choose the order.*

If contained within a Python list, numerical elements can be sorted in an ascending or descending order, with the *Sort* function. The denotation of order, only calls for a single named parameter *(reverse=<bool>)*. Then Python will use Tim sort to arrange the list. By default, this process will sort in an ascending order *(Class: CustomCSVEnty.initiate_sort)*. Thus as the list has to be sorted, the user only needs to be given the option to sort the list descendingly. However it must be made clear, that if its not sorted this descendingly, an ascending order will take precedence. This is accomplished with the use of prompts *(Script: main.manual_csv_input)*.

R3. *The ordered sequence should be inserted into a database along with the direction that the sequence was sorted in and the time taken to perform the sort.*

This requirement came with a hidden presupposition, that the execution time of a sort function, would have been recorded, thus ready to be sorted within a database. Thankfully, this was easy to implement. It only called for the current time to be captured, before and after the execution of Pythons built-in sort function. After which, the start time of the execution, can be subtracted from the end time. This is known as elapsed or delta time. The time was recorded in nanoseconds since the last epoch, using Pythons native time modules *(Class: CustomCSVEntry.initiate_sort)*. It was then converted into milliseconds, and rounded to five decimal places, which provided us with a decent level of precision. It should be noted however, that if the list is very small, the delta time will be minuscule thus rounded to zero, or the Python library will fail to capture the time difference. To prove time was being captured, a test case was constructed that input a large list *(Script: test_custom_CSV_entry.test_large_sort)*. The equation for this calculation, delta time, is provided below:

$$\Delta T = round((T_{End} - T_{Start}) * 1000, 5) \tag{1}$$

The second phrase of the requirements implementation used SqLite, a relational database, and one lighter then the standard MySQL. Whilst theirs a philosophical argument to be had, as to whether or not the use of a relation database was appropriate, considering a vast swathe of its features will go unused, it was cost effective in terms of time. The reason for this, its a native Python module, and I have prior experience with SQL. To start with an SQL database, a table had to be created *(Class: DatabaseHandler.create_connection)*. The format of this table, autoincrmenet its primary key, expected the list of Intergers to be given as a String, the sort-order as an Interger, and the time as a Real. All of these fields needed data as per the task, accordingly, none of them could be null. The table only needed to be created once, then data can be inserted via an insertion query *(Class: DatabaseHandler.insert_entry)*. This query was provided as a parameterized string. Database commits were done through method chaining, to allow for technical possibly of multiple inserts at once *(Class: DatabaseHandler.commit)*.

R4. *Feedback to the user the result of the operation (i.e. whether the operation was successful, any validation issues with the submission or any errors that occurred).*

This requirement was the easiest to fulfil. Messages of either errors, or the successful commitment of record, to the database, is shown on screen via simple messages *(Script: main)*. If an error does occur, as the application is simple, recovery can merely be provided as a redo of the task, that the user was currently performing.

R5. *Display the results of all sorts including the sort direction and time taken.*

This was achieved with a SQL select query, that retrieves the table of lists from the SqLite database *(Class: DatabaseHandler.select_all_entries.* The execution of this query results in a list of tuples, each tuple corresponds to a row within the table, thus the tuple contains the data found within a row. These tuples can be fed into an CSV class *(Class: CustomCSVEntry.create_entry_from_tuple)*, then conveniently printed to the console, due to its use of method overriding and string formatting *(Class: CustomCSVEntry.__str__)*. The overriding of this method is a standard staple within Python, and in many other languages such as Java, when an objects attributes need to be printed. The results of this are displayed on a separate console menu.

R6. *Allow the user to export all of the sorts as JSON.*

This was achieved similarly to requirement five (R5). Only this time, the selection of lists obtained via the use of a select query, was inserted into a dictionary *(Script: main.view_all_records)*. This dictionary, a set of key value pairs, took its key from the primary key of the table, and the values as Integer lists. This dictionary was then converted into a JSON, using another native module to Python, namely the JSON library. This newly created JSON can be written to disk, and the user can decide its filename *(Script: main.view_all_records)*.