1 Assumptions

- Assume that each line only contains one date-time value, i.e., each date-time value is separated by a "\n" character.
- Assume all valid date-time values must strictly follow the ISO 8601 format.
- Assume TZD uses either "Z" for GMT or +hh:mm or -hh:mm, i.e., +00:00 and -00:00 would not be allowed.
- Assume there is no need to perform semantic validation of the date-time value.
- Assume "a large list" has more than 50 date-time value entries and including various patterns of valid and invalid entries with duplicates.

2 Main Algorithms and Data Structures

2.1 Pseudo-code

Algorithm 1 Main algorithm

```
open and read input file
  while there are more lines do
    if this line contains a valid date-time value then
      add value to Set
    end if
  end while
  open output file
  for each entry in Set do
    write to output file + "\n"
  end for each
```

2.2 Data Structures

Strings, char array, and HashSet were used in this project.

- File I/O reads in each line in String format.
- When performing format validation, strings were converted to char arrays for easier random access.
- Since only unique date-time values are allowed in output file, use set's property for de-duplication. HashSet was chosen over TreeSet and LinkedHashSet since there is no requirement for order and to ensure best performing implementation, which is essential when the dataset is large.

3 Design Considerations

3.1 Java File I/O

To open and read a file, FileReader and BufferedReader were chosen.

- BufferedReader is a wrapper of FileReader and since it uses buffered stream, reading performance is more efficient. BufferedReader also provides the readLine() method which is useful for this program's purpose.
- BufferedReader was chosen over Scanner since BufferedReader has a larger buffer memory and Scanner class does not offer convenient method to read a line at a time.

To write to a file, FileWriter and BufferedWriter were chosen

- Similar to FileReader and BufferedReader, BufferedWriter is a wrapper of FileWriter but internally uses a buffer. Therefore, when we have a lot of read and write operations, e.g., when input and output sizes are big, this buffer will save I/O operations and thus provides better performance.
- BufferedReader was chosen over PrintWriter because PrintWriter will not thro IOException but BufferedReader will. Although PrinterWriter offers convenient methods like println(), for the purpose of this program, those methods are not useful.

3.2 Modular Programming

Instead of using one function to handle all tasks, several functions were designed and implemented for good modular programming practice. By doing so, the program design appears more logical and is easy to develop, debug, and test.

4 Testing

4.1 Data

The input file contains 55 entries and there should be only 7 unique, valid entries. Valid, invalid, and repeated entries are included in the input file.

4.2 Integrated Testing vs. JUnit Test

Integrated testing is preferable to JUnit test for the unique purpose of this program. As the main algorithm suggested, there are two fundamental functions of this program: File I/O and format check. Although File I/O is possible to be tested using JUnit test, when running the program, if there are problems related to File I/O, exceptions will be thrown. Testing for format checking can be done by including corner cases in input file.

5 Challenges

Some challenges encountered during the design and implementation of the project include:

- Define what is considered valid data and whether the criteria makes sense in real-life applications.
- Decide the best I/O implementation and consider its effect on scalability.
- Find the correct and simpliest way of implementing functions.

6 References

https://www.stackchief.com/blog/FileReader
https://www.digitalocean.com/community/tutorials/java-write-to-file
https://stackoverflow.com/questions/1747040/difference-between-java-io-printwriter-and-java-io-bufferedwriter